Compression for deep learning

Diana Resmerita

▶ **To cite this version:**

Diana Resmerita. Compression for deep learning. Signal and Image Processing. Université Côte d'Azur, 2022. English. NNT : 2022COAZ4043 . tel-03783658

HAL Id: tel-03783658

https://theses.hal.science/tel-03783658v1

Submitted on 22 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

Compression pour l'apprentissage en profondeur

Diana RESMERITA

Laboratoire d'Informatique, de Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 Université Côte d'Azur CNRS

Présentée en vue de l'obtention du grade de docteur en Automatique, Traitement du Signal et des Images **d'**Université Côte d'Azur

Dirigée par : Lionel FILLATRE, Professeur, Université Côte d'Azur
Co-encadrée par : Rodrigo CABRAL FARIAS, Maître de Conférence, Université Côte d'Azur
Soutenue le : 10 mars 2022

Devant le jury, composé de :
Thierry ARTIÈRES, Professeur, Aix-Marseille Université
Vincent GRIPON, Professeur, IMT Atlantique
Jean-Marc BROSSIER, Professeur, Grenoble INP
Stefan DUFFNER, Maître de Conférences, Université de Lyon, INSA de Lyon
Benoît DUPONT DE DINECHIN, CTO, Kalray

COMPRESSION POUR L'APPRENTISSAGE EN PROFONDEUR

Compression for Deep Learning

Diana RESMERITA

⋈

Jury :

Rapporteurs

Thierry ARTIÈRES, Professeur, Aix-Marseille Université
Vincent GRIPON, Professeur, IMT Atlantique

Examinateurs

Jean-Marc BROSSIER, Professeur, Grenoble INP

Directeur de thèse

Lionel FILLATRE, Professeur, Université Côte d'Azur

Co-encadrant de thèse

Rodrigo CABRAL FARIAS, Maître de Conférence, Université Côte d'Azur

Membres invités

Stefan DUFFNER, Maître de Conférences, Université de Lyon, INSA de Lyon
Benoît DUPONT DE DINECHIN, CTO, Kalray

Université Côte d'Azur

Diana RESMERITA

Compression pour l'apprentissage en profondeur

xiii+147 p.

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : output.tex – 28/6/2022 – 7:50

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

To my beloved mother and stepfather.
To my loved Mircea.

-

In the memory of my father and my grandmother.
May you rest in peace.

Compression for Deep Learning

Abstract

Autonomous cars represent complex applications that need powerful hardware machines to be able to function properly. Tasks such as lane-keeping, reading and understanding traffic signs or avoiding obstacles are solved by employing convolutional neural networks (CNNs) for object detection and classification. It is highly important that all the networks work in parallel in order to transmit all the necessary information and take a common decision. Nowadays, as the networks improve, they also have become bigger and more computational expensive. Deploying even one network becomes challenging. Compressing the networks can solve or at least alleviate this issue. Therefore, the first objective of this thesis is to find deep compression methods in order to cope with the memory and computational power limitations present on embedded systems. The compression methods need to be adapted to a specific processor, Kalray's MPPA, for short term implementations. Our contributions of this thesis mainly focus on compressing the network post-training for storage purposes, which means compressing the parameters of the network without retraining or changing the original architecture and the type of the computations. In the context of our work, we decided to focus on quantization. Our first contribution consists in comparing the performances of uniform quantization and non-uniform quantization, in order to identify which of the two has a better rate-distortion trade-off and could be quickly supported in the company. The company's interest is also directed towards finding new innovative methods for future MPPA generations. Therefore, our second contribution focuses on comparing standard floating-point representations (FP32, FP16) to non-standard arithmetical representations such as BFloat16, MSFP8, Posit8. The results of this analysis are in favor for Posit8. This motivated the company Kalray to conceive a decompressor from FP16 to Posit8. Since many compression methods already exist, we decided to move to an adjacent topic which aims to quantify the effects of quantization error on the network's accuracy. This is the second objective of the thesis. Finally, we focus on defining a new distortion measure adapted to our requirements, which represents a mainly theoretical contribution. Under reasonable assumptions, such as Normal input distribution, the distortion measure takes into account only the last layer of the network, the Softmax layer, and is adapted to a binary classification model. A set of experiments were done, using simulated data and small networks, which showcase the potential of the method.

Keywords: Data Compression, Deep Compression, Deep Learning, Deep Neural Networks, Floating-point, Quantization, Statistical Analysis, Error Approximation.

Compression pour l'apprentissage en profondeur

Résumé

Les voitures autonomes sont des applications complexes qui nécessitent des machines puissantes pour pouvoir fonctionner correctement. Des tâches telles que rester entre les lignes blanches, lire les panneaux ou éviter les obstacles sont résolues en utilisant plusieurs réseaux neuronaux convolutifs (CNN) pour classer ou détecter les objets. Il est très important que tous les réseaux fonctionnent en parallèle afin de transmettre toutes les informations nécessaires et de prendre une décision commune. Aujourd'hui, à force de s'améliorer, les réseaux sont devenus plus gros et plus coûteux en termes de calcul. Le déploiement d'un seul réseau devient un défi. La compression des réseaux peut résoudre ce problème. Par conséquent, le premier objectif de cette thèse est de trouver des méthodes de compression profonde afin de faire face aux limitations de mémoire et de puissance de calcul présentes sur les systèmes embarqués. Les méthodes de compression doivent être adaptées à un processeur spécifique, le MPPA de Kalray, pour des implémentations à court terme. Nos contributions se concentrent principalement sur la compression du réseau après l'entraînement pour le stockage, ce qui signifie compresser des paramètres du réseau sans réentraîner ou changer l'architecture originale et le type de calculs. Dans le contexte de notre travail, nous avons décidé de nous concentrer sur la quantification. Notre première contribution consiste à comparer les performances de la quantification uniforme et de la quantification non-uniforme, afin d'identifier laquelle des deux présente un meilleur compromis taux-distorsion et pourrait être rapidement prise en charge par l'entreprise. L'intérêt de l'entreprise est également orienté vers la recherche de nouvelles méthodes innovantes pour les futures générations de MPPA. Par conséquent, notre deuxième contribution se concentre sur la comparaison des représentations en virgule flottante (FP32, FP16) aux représentations arithmétiques alternatives telles que BFloat16, MSFP8, Posit8. Les résultats de cette analyse sont en faveur de Posit8. Ceci a motivé la société Kalray à concevoir un décompresseur de FP16 vers Posit8. Puisque de nombreuses méthodes de compression existent déjà, nous avons décidé de passer à un sujet adjacent qui vise à quantifier théoriquement les effets de l'erreur de quantification sur la précision du réseau. Il s'agit du deuxième objectif de la thèse. Nous remarquons que les mesures de distorsion bien connues ne sont pas adaptées pour prédire la dégradation de la précision dans le cas de l'inférence pour les réseaux de neurones compressés. Nous nous concentrons sur la définition d'une nouvelle mesure de distorsion avec une expression analytique qui a une forme de rapport signal/bruit. Un ensemble d'expériences a été réalisé en utilisant des données simulées et de petits réseaux, qui montrent le potentiel de la mesure.

Mots-clés : Compression, réseaux de neurones profond, quantification, virgule flottante, analyse statistique, approximation des erreurs.

Acknowledgements

First and foremost, I would like to thank Thierry Artières and Vincent Gripon for accepting to report on the thesis. I would also like to thank Jean-Marc Brossier and Stefan Duffner for being part of the jury of my thesis. Their interesting comments have been much appreciated.

I would also like to thank my thesis director, Lionel Fillatre, and my co-supervisor, Rodrigo Cabral Farias, who have always helped and challenged me when needed. Your advice has been so precious.

Next, I would like to thank all the people from Kalray from Sophia Antipolis and Montbonnot which have followed and supported me and my work throughout this journey. Thanks to the entire CTO Office and KaNN/Compute team: Benoît Dupont de Dinechin, Nicolas Brunie (ex. Kalray), Julien Le Maire, Mathieu Kapfer, Lucas Mahieu. A special thanks to Y-Meuil Cotteverte, Pierre Guironnet de Massas, Romaric Blanc.

As for all my PhD friends from the lab, I have so many beautiful memories with you. I will always remember the coffee breaks, the gossip and all the fun we had together. Melpo, Somia, Vasilina, thank you for being there for me in the lab and outside. Ninad, I am grateful to have you as a friend. Cyprien and Marie, it was (sometimes) nice working in the same office as you. Laeti x2, Giulia, Sara and all the other people from the second floor, thank you for making me part of your group (and for the coffee). Thank you Lyes for your excitement but also your drama (it made me feel better about my life). Special mention for Eva, Xavier, Arne, Anca, Sarah, the Guy (who is not actually in the lab). The list of names is long. For those who I do not mention, just know that I did not forget you. If I have ever passed by your office, I have enjoyed the talks we had and hope to see you again outside the lab. Rémy, thank you for all the help and support during the years that we spend together as flatmates, the gossip and of course for the food.

I would also like to thank my old friends: Mathias, Mehdi, Kevin, Thaís, Dragoș and those from Romania whom I rarely see, but who have always been there for me.

Finally, I would like to thank my family and my boyfriend, Mircea. I would not have done this without your first push, your support, advice and understanding. I hope I have made you proud. *Vă iubesc!*

Table of contents

1 Introduction **1**
1.1 About Deep Learning compression . . . 1
1.2 Scientific and industrial objectives . . . 2
1.2.1 Challenges for real time applications . . . 2
1.2.2 The company . . . 3
1.2.3 Objectives and limitations . . . 5
1.3 Outline and contributions . . . 6

2 Deep Learning background **9**
2.1 Basic knowledge on Deep Neural Networks . . . 11
2.1.1 Formalization of a neuron . . . 11
2.1.2 Formal architecture of a feedforward network . . . 13
2.1.3 Training vs. inference . . . 14
2.2 Image Classification architectures . . . 16
2.2.1 Datasets . . . 19
2.2.2 Well-known Networks . . . 22
2.3 Object Detection architectures . . . 23
2.3.1 Datasets . . . 25
2.3.2 Well-known Networks . . . 26
2.4 Conclusion . . . 27

3 Deep Learning Compression **29**
3.1 General presentation of techniques . . . 31
3.2 Network Pruning . . . 33
3.3 Network Quantization . . . 35
3.3.1 Scalar and vector quantization . . . 36
3.3.2 Low precision quantization . . . 37
3.4 Conclusion . . . 40

4 Preliminary study on compression methods **41**
4.1 Methods . . . 43
4.1.1 Pruning . . . 43
4.1.2 Quantization . . . 43
4.1.3 Binarization . . . 44
4.1.4 Deep Compression . . . 44
4.2 Experiments . . . 44
4.2.1 Experimental settings . . . 44
4.2.2 Compression performance comparison . . . 46
4.3 Conclusion and Perspectives . . . 51

5 Uniform vs non-uniform quantization for storage purposes 53
5.1 Data Compression . . . 55
5.1.1 General compression workflow . . . 55
5.1.2 Scalar quantization . . . 56
5.1.3 Uniform quantization . . . 56
5.1.4 Non-uniform Quantization . . . 57
5.2 Rate distortion theory . . . 59
5.2.1 Background on rate-distortion theory . . . 59
5.2.2 Rate distortion trade-off . . . 60
5.3 Experiments . . . 61
5.3.1 CNN on MNIST . . . 61
5.3.2 VGG on CIFAR-100 . . . 66
5.4 Conclusions and Perspectives . . . 68

6 IEEE 754 and alternative formats for storage purposes 69
6.1 Industrial stakes . . . 71
6.2 Floating-point formats in Deep Learning . . . 73
6.2.1 IEEE 754 floating-point formats . . . 73
6.2.2 Brain floating-point format . . . 73
6.2.3 Microsoft floating-point 8 . . . 74
6.2.4 Posit . . . 74
6.2.5 Comparison between data formats . . . 75
6.3 Parameter Compression . . . 77
6.4 Experimental Results . . . 78
6.4.1 Experiment 1 . . . 78
6.4.2 Experiment 2 . . . 79
6.4.3 Experiments 3 and 4 . . . 79
6.5 Conclusion and Perspectives . . . 80

7 Effect of quantization error on Softmax Layer 83
7.1 Sensitivity analysis in neural networks . . . 85
7.2 Problem statement . . . 86
7.2.1 Deep neural networks . . . 86
7.2.2 Minimum Bayes risk . . . 87
7.3 Distortion measure for classifiers . . . 88
7.4 Distortion measure applied to uniform quantization . . . 89
7.4.1 Approximation of the distortion function . . . 89
7.5 Experiments . . . 91
7.5.1 Numerical simulations . . . 91
7.5.2 One-hidden-layer neural network on Sonar . . . 93
7.6 Conclusion and Perspectives . . . 96

8 Conclusions and Future works 97
8.1 Conclusion . . . 97
8.2 Perspectives . . . 98
8.2.1 Rate distortion measure . . . 98

8.2.2 End-to-end deep learning compression tool 98
8.2.3 Quantization and matrix acceleration 99

Bibliography **101**

List of Figures **115**

Liste of Tables **119**

Appendix

A Chapter 3: Nvidia TensorRT study 123
A.1 Introduction 123
A.2 MLPerf 123
A.3 TensorRT optimizations 124
A.4 TensorRT and Tensorflow 127
A.5 Conclusion 129
B Chapter 6: Alternative floating point formats 130
B.1 Additional data on network parameters 130
B.2 Compression on all parameters 131
B.3 Compression of trainable parameters (without BN parameters) 133
B.4 Compression without biases 134
B.5 Compression without biases and BN parameters 136
C Chapter 7: Upper bound approach 137
C.1 Upper bound for distortion measure 137
C.2 Experiments for the upper bound 140
C.3 Softmax classifier on synthetic data 141
C.4 One-hidden-layer ReLU neural network on Sonar 142
D Chapter 7: Proofs of theorems 142
D.1 Proof theorem 7.4.1: approximation of $a_j(\hat{\mathbf{w}})$ 142
D.2 Proof theorem 7.4.2: approximation of the distortion $d(\mathbf{w})$ 145
D.3 Proof corollary 7.4.3: approximation of the distortion $d(\mathbf{w})$ 146

CHAPTER 1

Introduction

1.1 About Deep Learning compression

Nowadays, numerous applications such as visual recognition, natural language understanding and robotics are of great interest. Deep learning techniques have become more and more successful due to their effectiveness in targeting these kinds of applications. The main task of Deep Learning is one-image inference for image classification, object detection or semantic segmentation.

Convolutional Neural Networks (CNNs) are extremely effective in image classification. They allow fast and precise image recognition. Essentially, the architectures rely on stacked convolutional and fully connected layers, which account for most of the resources involved when inferencing with the model. Nowadays, CNNs are highly requested in the embedded system domain for many real time applications such as object detection for autonomous cars and video surveillance.

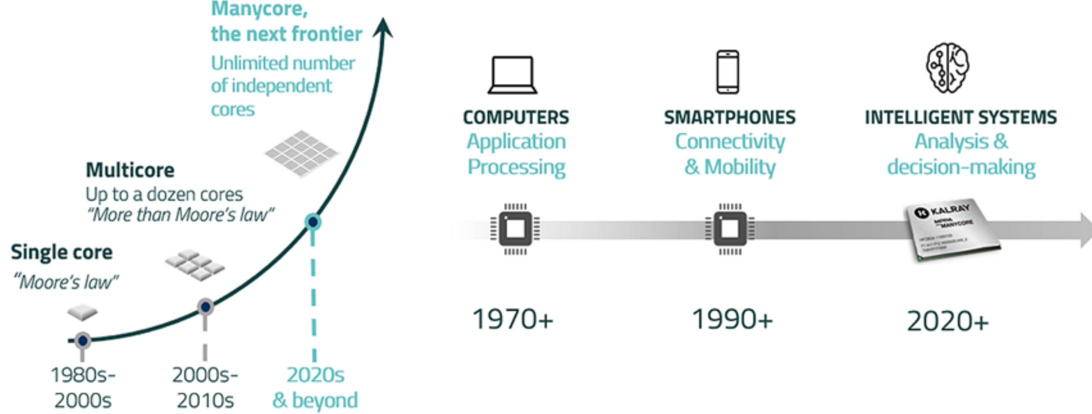


Figure 1.1: Processor History [Kalray, 2021]. Intelligent processors are the next technology wave.

Processors (see Figure 1.1) specially designed for deep neural networks are a must. Portability and real-time inference are critical in the case of many applications. When using these real time applications, we want the neural networks to process the inputs and give the correct result with stringent latency constraints. As the networks improved, the architectures became more complex

and the models now require significantly increased computing and memory resources.

1.2 Scientific and industrial objectives

1.2.1 Challenges for real time applications

Inference algorithms designed for CNNs can easily overwhelm the resources of embedded systems. When running a complex network, the platform must overcome several challenges, including limited memory, limited computational power and long inference time. These challenges are detailed below.

Low Latency

Neural networks are mainly used for applications that require real-time complex decisions. During the inference phase, CNNs process 2D images or frames from videos in order to extract features and classify the surrounding environment. Thus, it is essential for latency to remain low in order for the application to function properly.

Each frame needs to be processed in a very short time, which is called timeframe. Unforeseen buffering time may delay critical tasks, leading to serious consequences. The only way to ensure low latency is to process input data directly on the embedded platform, without sending it to the cloud.

Processing capabilities

As previously said, the most compute-intensive operations in CNNs are convolutions. For example, processing a 224x224 pixel RGB image requires billions of FLOPs (FLoating-point OPerations) operations, which usually represent about 60% of the overall computational load. Thus, to run at 60 images-per-second — a satisfactory frame rate for a reactive system — the model would require a good amount of GFLOPs, as one multiply-add operation corresponds to 2 FLOPs. Even for high-end embedded CPUs or GPUs, the required amount of computation is difficult to sustain.

One way to ease the CPU load of CNN inference is the exploitation of Single Instruction Multiple Data (SIMD) instructions. Another is the transition from 32-bit to 16-bit floating-point data representations, as the effects on inference accuracy are generally non-significant. Further computational savings can be achieved by moving to 8-bit or 16-bit fixed-point or fractional arithmetic. However, this may have a significant impact on accuracy and require the network to be retrained. For this thesis, we assume the CNNs have been trained using floating-point arithmetic and will not be retrained, so inference uses floating-point arithmetic.

High memory bandwidth

In CNNs, the number of parameters required to perform convolutions rises with each successive layer (as each layer is processed with different parameters weights). This means that by the last CNN layer, the total amount of data related to parameters may be higher than the amount

of neurons and, thus, the underlying computing unit is overloaded by a huge quantity of data. For instance, in the case of GoogLeNet [Szegedy et al., 2015] where there are about 7 million weight parameters, if input from camera sensors at 60 frames per second is being processed in an autonomous vehicle, the CNN's weight parameter processing will result in 1.6 GB per second of sustained bandwidth.

Memory bandwidth becomes an issue with the parallel execution over numerous cores. For instance, if the application is spread in parallel over 16 cores, the need for DDR bandwidth rises to over 25.6 GB/s. In general, to avoid limitations associated with DDR bandwidth, processors leverage their memory hierarchy, which is composed of caches and prefetch engines. However, in the case of CNN processing, this type of memory hierarchy is not effective. Thus, changing the way parameters are stored is the key for handling the Deep Learning memory challenges.

Low Power Consumption

Though many applications use CNNs, its use is most pervasive in embedded technology: post-processing in cameras, automatic detection in drones, live detections and decisions in autonomous vehicles, etc. This means that the processor or hardware solution must be in line with the needs of embedded solutions: low power consumption with a Small Form Factor (SFF). SFF is a term used to describe a device smaller than standard devices.

As powerful these large models are, they also consume a significant amount of energy. Because of their memory requirements, they have to be stored in the off-chip memory (DRAM) and parts of the models are retrieved each time they are used. The energy cost is dominated mainly by memory accesses. Table 1.1 shows the energy costs of arithmetic and memory operations in a 45nm CMOS processor [Han et al., 2016a].

If the memory size is reduced, then more parts of the model can be stored in on-chip memory (SRAM), in order to avoid too expensive memory accesses.

Operation	int ADD	float ADD	int MULT	float MULT	32KB SRAM	DRAM
Energy [pJ]	0,1	0,9	3,1	3,7	5	640

Table 1.1: Energy consumption for 32 bits operations [Han et al., 2016a].

1.2.2 The company

Kalray is a semiconductor company specialized in Massively Parallel Processors Array (MPPA) for intelligent systems. Manycore technology offers a great approach to support efficient AI computing. Kalray focuses on high-performance applications such as data centers and autonomous vehicles. Figure 1.2 shows a schema of Kalray's 3rd generation processor, aka *Coolidge*, which is formed of five independent computing clusters connected to each other which have an external memory DDR. Each cluster has 16 high performance processing cores which share 4MB of Shared Memory (SMEM). The power of Kalray's processor comes from its partitioning. Compu-

tations can be done independently and in parallel by distributing them across the clusters, while the presence of multiple cores makes the processing faster.

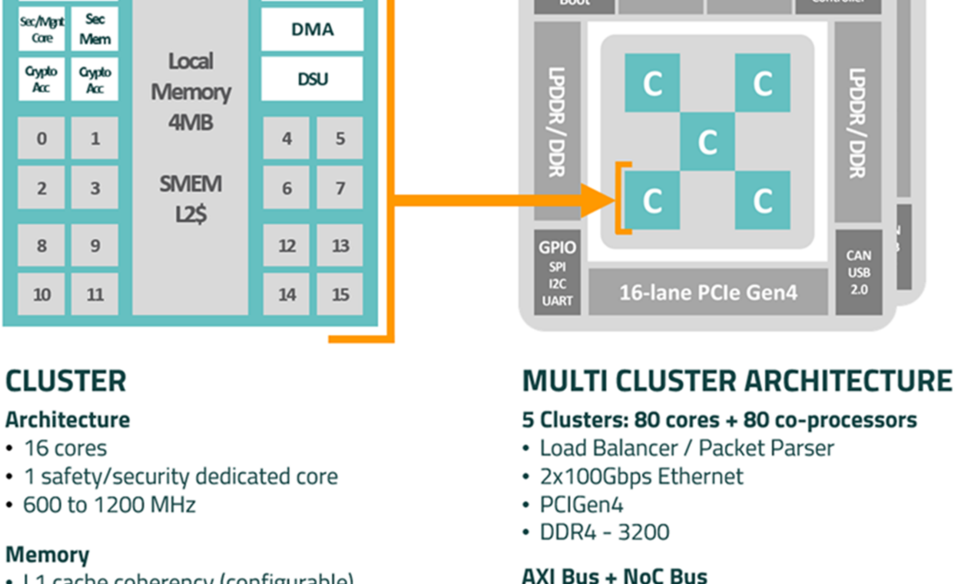


Figure 1.2: Top level diagram of Kalray's cluster partition [Kalray, 2021]. The cluster (on the left) and processor (on the right). A list of the main features is given for each one of the units.

In order to avoid deep neural network developers to focus on speeding up the networks using optimizations during the training and inference phases, acceleration frameworks for inference execution have been used to maximize throughput, minimizing latency, optimizing memory usage and reducing energy requirements. Kalray also provides a solution called **Kalray Neural Network** (KaNN), a tool that acts as a code generator and execution engine on the MPPA processor. KaNN (see Figure 1.3) allows running trained neural networks in different frameworks such as TensorFlow [TensorFlow, 2021], Caffe [Jia et al., 2014], ONNX [ONNX, 2019] and it guarantees an optimal execution of the networks.

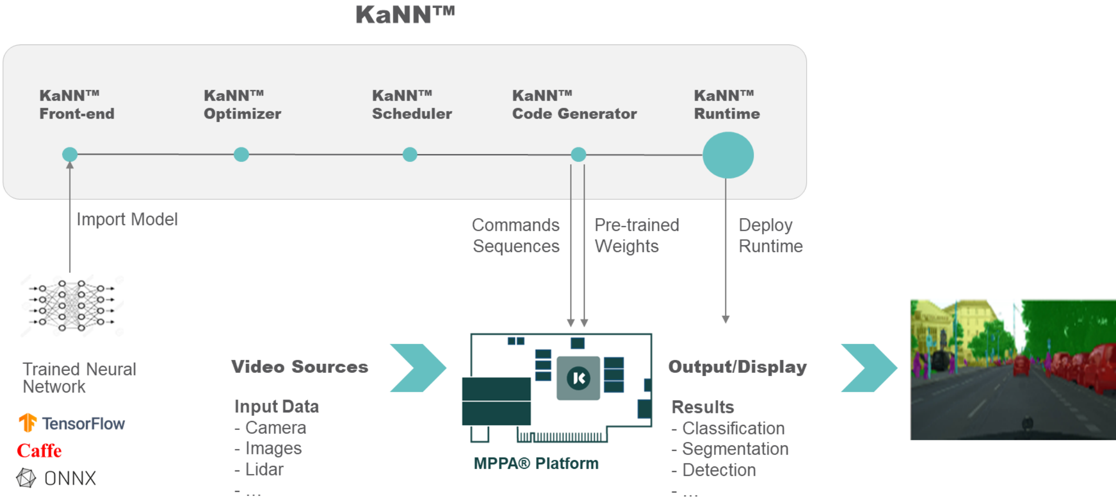


Figure 1.3: Diagram of KaNN [Kalray, 2021]. A trained neural network is first given to the KaNN's front-end. In order to generate an optimized computation graph, further steps are performed such as optimization, memory allocation and scheduling. These steps are followed by code generation and deployment. During the deployment step, the generated code is sent to the MPPA Platform. Execution is done by giving an input to the generated KaNN model which will return either the class in the case of a classification network, or display the segmentation or detected objects.

As autonomous cars are critical low latency applications, it is highly necessary to reduce the complexity of the models, the amount of communication and the occupied memory to save on power consumption and reduce network connectivity. To leverage the capabilities of the processor, the model and the weights of the networks are stored in the DDR, the network is split into layers and the weights are transferred layer by layer into the clusters where the processing cores share the same memory.

1.2.3 Objectives and limitations

The main goal of our work is to help KaNN speed up the calculations during the inference phase and to reduce the memory used by the networks. It is necessary to reproduce the same results as the supported frameworks and to use methods adapted to the characteristics of the MPPA processor that will give better performance. To tackle this subject, we studied and compared compression and acceleration methods that have been proposed by academic researchers and R&D companies.

Our second objective is more theoretical and is focused on understanding how the network decisions are impacted by the compression error.

Since this thesis is done in collaboration with a company, we need to take into account several technical limitations. Given that the company's processors are used only for inference, in our work, we mostly focused on compression methods for storage purposes, which does not involve training. We assume the CNNs have been already trained and will not be retrained. The training

phase is assumed to be done using floating-point arithmetic, and the inference also uses floating-point arithmetic.

1.3 Outline and contributions

In this thesis, we present a study of weight compression methods which can be applied without changing the architecture of the networks, or retraining the model. We explore several methods applied on several CNN architectures. Finally, we propose a new distortion measure using the Bayes risk which gives us more insights on how the networks are impacted by quantization. We applied the distortion on the last layer of binary classification models.

The thesis is organized as follows:

Chapter 2 presents the background on Deep Learning. We mostly focus on image classification, but a short description for object detection is also provided. The state of the art of compression for Deep Learning is given in **Chapter 3**.

Chapter 4 presents a preliminary study on methods for compressing Deep Learning. We compare several methods of compression with and without training, such as pruning, quantization and binarization.

Chapter 5 focuses on storage purposes and provides a comparison between two quantization methods (uniform and non-uniform). In this chapter, we also introduce data compression and rate-distortion theory. The work presented here has also been published in [Resmerita et al., 2019a, Resmerita et al., 2019b].

In **Chapter 6**, we focus on benchmarking some innovative alternatives for storage purposes. New alternatives to the standard floating-point have been designed which require less precision. We are interested in investigating how these alternatives perform as storage formats for deep neural networks. The work done in this chapter was also published in [Resmerita et al., 2020, Resmerita et al., 2021c].

Chapter 7 is our main theoretic contribution. We introduce a new distortion measure which computes the gap between the original and the compressed model classification risks when applying a given quantization algorithm. Our theoretical analysis is done only on the last layer of a neural network, which represents the classification phase. This work has been published in [Resmerita et al., 2021b, Resmerita et al., 2021a].

Finally, a general conclusion and perspectives are given in **Chapter 8**.

The list of accepted articles is given below.

- *Compression des réseaux de neurones profonds à base de quantification uniforme et non-uniforme*, Diana Resmerita and Rodrigo Cabral Farias and Benoît Dupont de Dinechin and

Lionel Fillatre. At GRETSI 2019, XXVIIème Colloque francophone de traitement du signal et des images.

- *Compression des réseaux de neurones profonds à base de quantification uniforme et non-uniforme*, Diana Resmerita and Rodrigo Cabral Farias and Benoît Dupont de Dinechin and Lionel Fillatre. At ORASIS 2019.

- *Benchmarking Alternative Floating-Point Formats for Deep Learning Inference*, Diana Resmerita and Rodrigo Cabral Farias and Benoît Dupont de Dinechin and Lionel Fillatre. At COMPAS 2020.

- *Représentations arithmétiques flottantes de taille réduite pour le Deep Learning*, Diana Resmerita and Rodrigo Cabral Farias and Benoît Dupont de Dinechin and Lionel Fillatre. At CORESA 2021.

- *Classification error approximation of a compressed linear softmax layer*, Diana Resmerita and Rodrigo Cabral Farias and Benoît Dupont de Dinechin and Lionel Fillatre. At EUSIPCO 2021, European Signal Processing Conference.

- *Distortion approximation of a compressed softmax layer*, Diana Resmerita and Rodrigo Cabral Farias and Benoît Dupont de Dinechin and Lionel Fillatre. At SSP 2021, IEEE Statistical Signal Processing Workshop.

- *A Posit8 Decompression Operator for Neural Networks Inference*, Orégane Desrentes, Diana Resmerita and Benoît Dupont de Dinechin. At CONGA 2022.

CHAPTER 2

Deep Learning background

In this chapter, we present the deep learning background. We start by presenting the basic knowledge on deep learning: neurons, feedforward networks, the training and inference stages. We then present image classification architectures which are the main focus of our work. Finally, we introduce object detection networks which represent the real-time target applications.

2.1 Basic knowledge on Deep Neural Networks 11
2.1.1 Formalization of a neuron 11
2.1.2 Formal architecture of a feedforward network 13
2.1.3 Training vs. inference 14
2.2 Image Classification architectures 16
2.2.1 Datasets 19
2.2.2 Well-known Networks 22
2.3 Object Detection architectures 23
2.3.1 Datasets 25
2.3.2 Well-known Networks 26
2.4 Conclusion 27

Artificial Intelligence (AI) has become an essential part of the technology industry. AI is often used to refer to Machine Learning (ML) or Deep Learning (DL). As shown in Figure 2.1, Machine Learning is a subset of AI, and Deep Learning is a specific kind of ML. **Machine Learning** is essentially associated with prediction and classification algorithms.

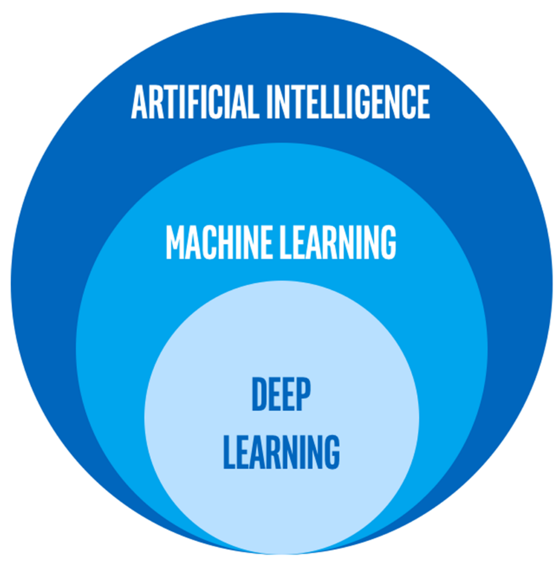


Figure 2.1: Artificial Intelligence, Machine Learning and Deep Learning [Intel, 2021].

Deep Learning uses multi-layer models comprising a very large number of parameters. These models are mainly based on neural networks. By adding more layers, the deep neural networks can represent functions of higher complexity. Most tasks of DL consist in mapping an input vector/matrix to an output vector. Given a sufficiently large training dataset of labeled examples, deep neural networks can accomplish classification, localization and recognition tasks.

This chapter first presents the basic notions of Deep Learning, followed by the description of well-known models and datasets used in two tasks: image classification and object detection. **Image classification** is a fundamental task that attempts to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label. Typically, image classification refers to images in which only one object appears and is analyzed. In contrast, **object detection** involves both classification and localization tasks, and is used to analyze more realistic cases in which multiple objects may exist in an image.

2.1 Basic knowledge on Deep Neural Networks

2.1.1 Formalization of a neuron

Neural networks are inspired from how the human brain works (see Figure 2.2). An artificial neuron, also called a perceptron, is an multivariate function characterized by a linear combination of the inputs $x_1, \ldots, x_n$, weighted by a vector of parameters $(w_1, \ldots, w_n)$ and a bias b. This value

is then passed to a non-linear function, known as an activation function, to become the neuron's output.

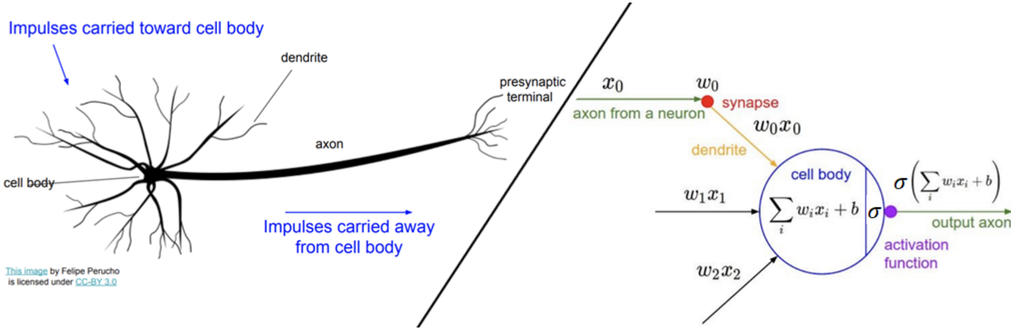
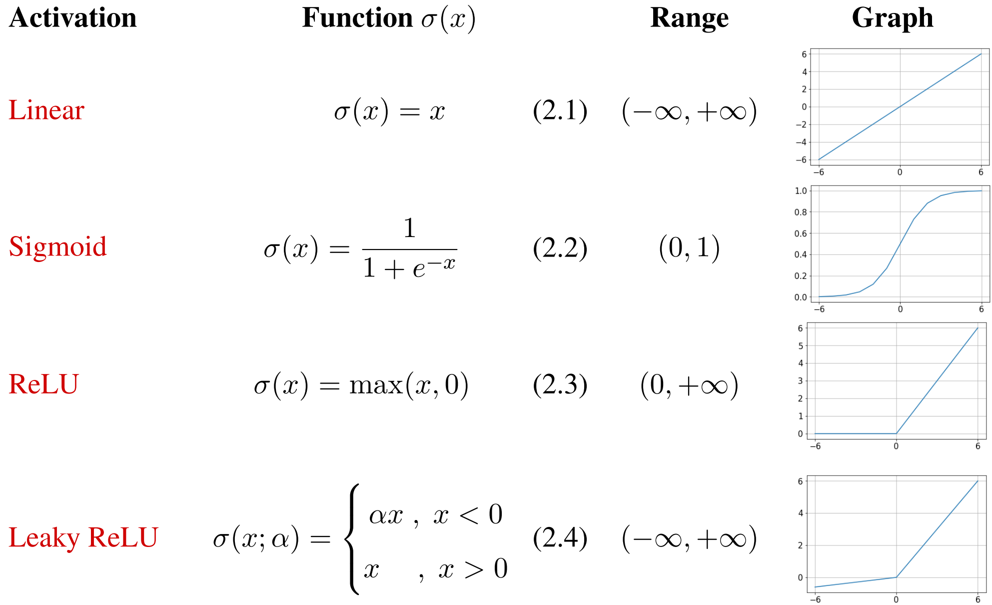


Figure 2.2: Formalization of a neuron. Side-by-side illustrations of biological and artificial neurons [Fei-Fei et al., 2021]. Each neuron receives dendrites or inputs. The cell body of the neuron corresponds to the weighted sum with an added bias. The result of this operation is given to the activation function and represents the impulses carried out from the cell body.

Activation function. The activation function σ represents the threshold of stimulation of the neuron. This non-linear function is applied to the affine transformation. A list of commonly used activation functions is given in Table 2.1. The simplest activation function is the linear function (2.1). It is also called the identity function. A popular activation function is the sigmoid (2.2). The sigmoid's value is between 0 and 1 and its curve looks like an S-shape. The most used activation function is the Rectified linear unit (ReLU) (2.3). This function allows faster and more effective training. The Leaky ReLU (2.4) is another activation function which is used in more recent networks.



Activation	**Function $\sigma(x)$**		**Range**	**Graph**
Linear	$\sigma(x) = x$	(2.1)	$(-\infty, +\infty)$	
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	(2.2)	$(0, 1)$	
ReLU	$\sigma(x) = \max(x, 0)$	(2.3)	$(0, +\infty)$	
Leaky ReLU	$\sigma(x;\alpha) = \begin{cases} \alpha x \ , & x < 0 \\ x \ , & x > 0 \end{cases}$	(2.4)	$(-\infty, +\infty)$	

Table 2.1: Commonly used activation functions.

The output y_i of the i-th neuron of a network of inputs $x_1, ..., x_n$ weighted by $w_{i,1}, ..., w_{i,n}$ is given in (2.5), where we associate the i-th neuron with an activation function σ_i:

$$y_i = \sigma_i \left(\sum_{j=1}^{n} x_j w_{i,j} + b \right). \tag{2.5}$$

2.1.2 Formal architecture of a feedforward network

Consider a classification problem of C classes. The goal of a feedforward network is to assign a class to a given input. To do this, the neural network maps an input $\mathbf{x}$ to an output $\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x})$, where $\boldsymbol{\theta}$ is a vector of estimated parameters. These networks are typically composed of artificial neurons, organized in layers [Goodfellow et al., 2016].

Given a vector $\mathbf{x}_0 = (x_1, ..., x_{n_0})$ of dimension n_0, a feedforward neural network generates an output $\hat{\mathbf{y}}(\mathbf{x}) = (\hat{y}_1(\mathbf{x}), \dots, \hat{y}_C(\mathbf{x}))$ which falls into the C-simplex given by:

$$\mathrm{S}_C = \left\{ \hat{\mathbf{y}} \in \mathbb{R}^C \;\middle|\; \hat{y}_i \geq 0, \sum_{i=1}^{C} \hat{y}_i = 1 \right\}. \tag{2.6}$$

To simplify the notation, we use $\hat{y}_i = \hat{y}_i(\mathbf{x})$. The value $\hat{y}_i$ works like the probability that the network decides that $\mathbf{x}$ belongs to the class $i \in \{1, \dots, C\}$.

Multilayer Perceptron. A multilayer perceptron (MLP) is a type of neural network composed of multiple layers of perceptrons in a directed acyclic graph. The structure of a network (see Figure 2.3) is generally divided into 3 parts: input layer, hidden layers and output layer. For example, the function $f_{\boldsymbol{\theta}}(\mathbf{x})$ can be written as the following composition:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{f}_{K+1}(\mathbf{f}_K(\dots \mathbf{f}_1(\mathbf{f}_0(\mathbf{x})))). \tag{2.7}$$

The functions $\mathbf{f}_k$ for $0 \leq k \leq K+1$ are called layers. A layer is a set of neurons that have no connection between them. The number of layers in this composition gives the depth of the network. The vector of parameters $\boldsymbol{\theta}$ consists of weights and biases:

$$\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_{K+1}), \text{ where } \boldsymbol{\theta}_k = (\mathbf{W}_k, \mathbf{b}_k). \tag{2.8}$$

For a network with K hidden layers, $\mathbf{f}_0$ is the input layer $\mathbf{f}_0(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^{n_0}$, $\mathbf{f}_{K+1}$ is the output layer and the hidden layers are from $\mathbf{f}_1$ to $\mathbf{f}_K$. The inputs of each layer (except for the input layer) are weighted by a weight matrix $\mathbf{W}_k$ and added to a bias vector $\mathbf{b}_k$. A layer where all the input elements are used in the weighted sum is also called a **Fully Connected layer** (FC).

As mentioned above, in each layer an activation function σ_k is applied to the weighted combination. The different layers are formalized as follows:

$$\mathbf{z}_0 = \mathbf{f}_0(\mathbf{x}) = \mathbf{x}, \tag{2.9}$$

$$\mathbf{z}_k = \mathbf{f}_k(\mathbf{x}) = \sigma_k(\mathbf{W}_k \mathbf{f}_{k-1}(\mathbf{x}) + \mathbf{b}_k), \tag{2.10}$$

$$\mathbf{z}_{K+1} = \mathbf{f}_{K+1}(\mathbf{x}) = \sigma_{K+1}(\mathbf{W}_{K+1} \mathbf{f}_K(\mathbf{x}) + \mathbf{b}_{K+1}), \tag{2.11}$$

$$\mathbf{W}_k \in \mathbb{R}^{n_k \times n_{k-1}}, \ \mathbf{b}_k \in \mathbb{R}^{n_k},$$

$$\mathbf{W}_{K+1} \in \mathbb{R}^{C \times n_K}, \ \mathbf{b}_{K+1} \in \mathbb{R}^{C}. \tag{2.12}$$

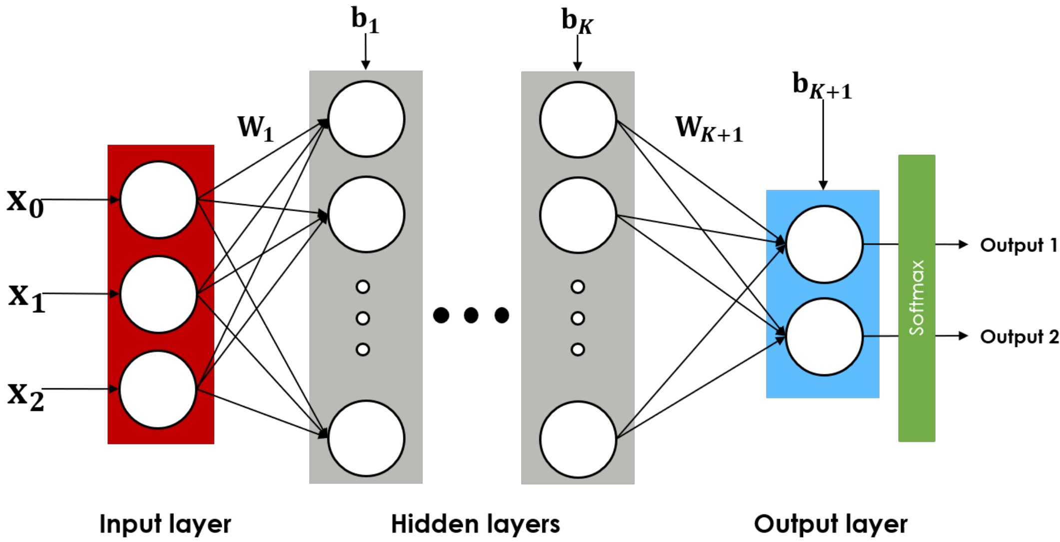


Figure 2.3: Architecture of a feedforward network with K hidden FC layers.

Usually the output layer uses a different activation function than the hidden layers. In the case of a classification with C classes, the activation function used is softmax [Bishop, 2006].

Softmax. Softmax is an output activation function used to normalize each component of the input vector to a value between 0 and 1. The result is interpreted as a probability that indicates the confidence that an entry belongs to a certain class. The function applied to the entire vector is written as

$$\sigma_{\text{softmax}}(\mathbf{x}) = \frac{1}{\sum_{i=1}^{C} e^{x_i}} \left(e^{x_1}, \dots, e^{x_C} \right), \ \forall \mathbf{x} \in \mathbb{R}^C. \tag{2.13}$$

The output vector $\hat{\mathbf{y}}$ is of size C and it is called a soft one-hot encoding vector. In the case of a two-class model, it is equivalent to a sigmoid function (2.2). To decode the output vector, the decision rule is given in (2.14) and it chooses the class with the highest probability given in $\hat{\mathbf{y}}$.

$$\delta(\hat{\mathbf{y}}) = \underset{i \in \{1,\dots,C\}}{\arg\max} \ \hat{y}_i. \tag{2.14}$$

Note that if there are 2 or more classes with the highest probability, $\arg\max$ will return the index corresponding to the first occurrence.

2.1.3 Training vs. inference

Before being deployed in "real world" applications, Deep Learning networks need to be trained. The training phase is used to adjust the parameters of the created network, while the inference phase is used to classify data. The two phases are related (see Figure 2.4). Both phases use the same forward propagation step. However, the training phase is formed of the forward propagation step and adds another step, called backward propagation.

Suppose a training dataset of m samples $\mathcal{S} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})|1 \leq i \leq m\}$, where $\mathbf{x}^{(i)}$ is the input and $\mathbf{y}^{(i)}$ is the ground truth, also called label. Training a network means estimating and improving the parameters $\mathbf{W}_k$ and $\mathbf{b}_k$ of each layer to maximize the accuracy on the training data given in (2.15):

$$ACC(f_{\boldsymbol{\theta}}, \mathcal{S}) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}\left\{\delta\Big(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})\Big) = \delta\Big(\mathbf{y}^{(i)}\Big)\right\}. \tag{2.15}$$

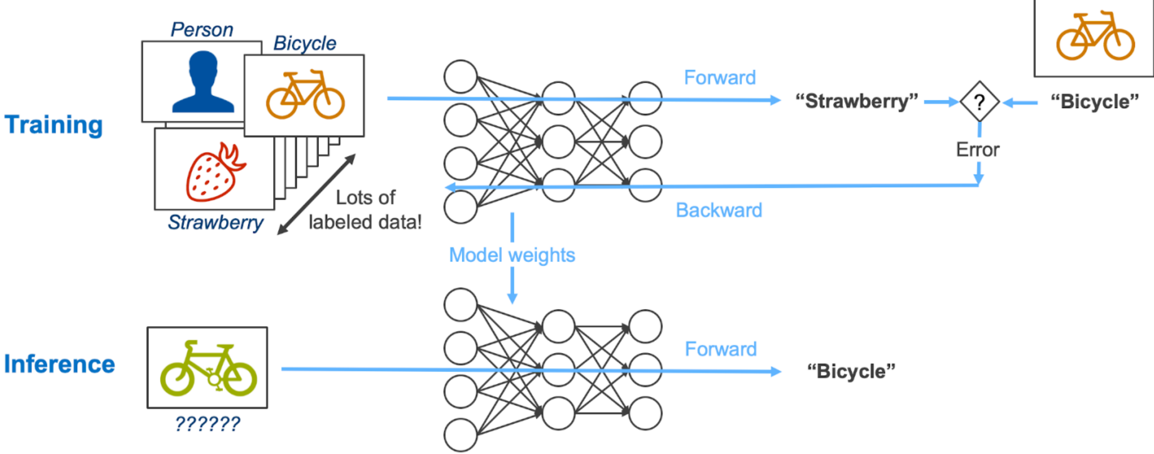


Figure 2.4: Training vs. inference [Intel, 2021].

The goal of **training** is to maximize the accuracy function, but this is not an easy task. In practice, we cannot do this and we minimize a loss function by using the gradient descent method. In classification, the loss function generally used is the **cross-entropy** function H :

$$\mathcal{L}(f_{\boldsymbol{\theta}}, \mathcal{S}) = \sum_{i=1}^{m} H\left(\mathbf{y}^{(i)}, f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})\right) = -\sum_{i=1}^{m}\sum_{j=1}^{C} y_j^{(i)} \log_2 f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) \tag{2.16}$$

The **gradient descent** is an optimization algorithm that iteratively finds the minimum of a function. Figure 2.5 illustrates how the algorithm works. Starting from a random point, the algorithm reduces the value of the function by moving the point using small steps in the opposite sign of the gradient of the loss, denoted with $\nabla_{\theta}\mathcal{L}$. To minimize $\mathcal{L}(\boldsymbol{\theta})$ for a multi-dimensional input $\boldsymbol{\theta}$, the gradient descent proposes a new point:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \gamma \nabla_{\theta}\mathcal{L}. \tag{2.17}$$

The notion of partial derivative is used. The partial derivative $\frac{\delta}{\delta\boldsymbol{\theta}_i}\mathcal{L}(\boldsymbol{\theta})$ measures how the function changes if only the element $\boldsymbol{\theta}_i$ changes. A learning rate $\gamma \in \mathbb{R}^+$ determines the step size. The learning rate should be adapted to the size of the model.

Next, we present **back-propagation**. The back-propagation algorithm [Rumelhart et al., 1986] is a learning procedure which allows the information from the loss function to be transmitted backward through the network in order to adjust the weights. More precisely, back-propagation is a method for computing the gradients. It is often mistaken for **stochastic gradient descent**

[Bottou, 1998] which is actually a training algorithm which uses the gradient to efficiently update parameters and minimize the error.

When training a network, the back-propagation computes the gradients of the loss function with respect to the parameters. To adjust the parameters from $\mathbf{W}_k$ and $\mathbf{b}_k$ to $\mathbf{W}'_k$ and $\mathbf{b}'_k$, the partial derivatives of the loss function with respect to each parameter need to be computed. These partial derivatives are used in the following formulas:

$$\mathbf{W}'_k = \mathbf{W}_k - \gamma \, \nabla_{\mathbf{W}_k}\mathcal{L} \, , \tag{2.18}$$

$$\mathbf{b}'_k = \mathbf{b}_k - \gamma \, \nabla_{\mathbf{b}_k}\mathcal{L} \, , \tag{2.19}$$

where $\nabla_{\mathbf{W}_k}\mathcal{L}$ and $\nabla_{\mathbf{b}_k}\mathcal{L}$ denote the vector of partial derivatives.

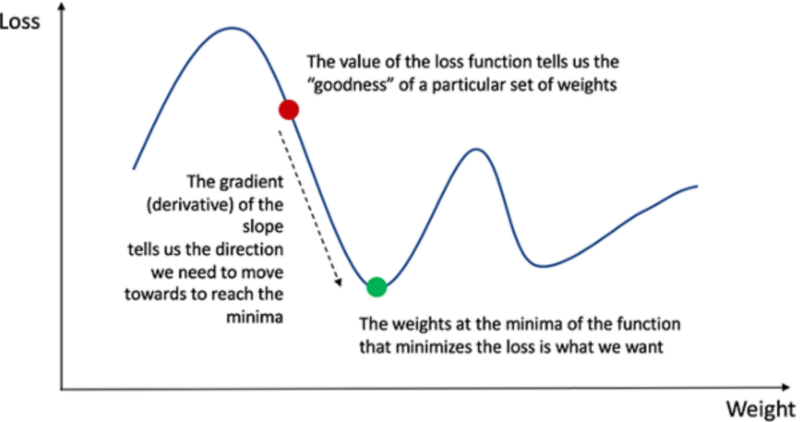


Figure 2.5: Gradient descent algorithm [Loy, 2018].

The **inference** process uses the trained network to make predictions, usually, on unseen data. The testing dataset is formed of unseen data. Deploying a trained network for inference can be trivial. However, for reasons explained in the next section, a trained model is often modified and simplified before being deployed for inference.

2.2 Image Classification architectures

Convolutional Neural Networks (CNNs) are considered the reference in image classification. Convolutional networks are neural networks that use Convolution layers additionally to Fully Connected layers. A typical convolution network (see Figure 2.6) uses multiple types of layers. Convolution layers are always followed by an activation operation, and then, by a pooling layer. These layers are used to construct a **feature map** that is used for classification. For implementation reasons, the convolution and the activation layers are fused into a single layer.

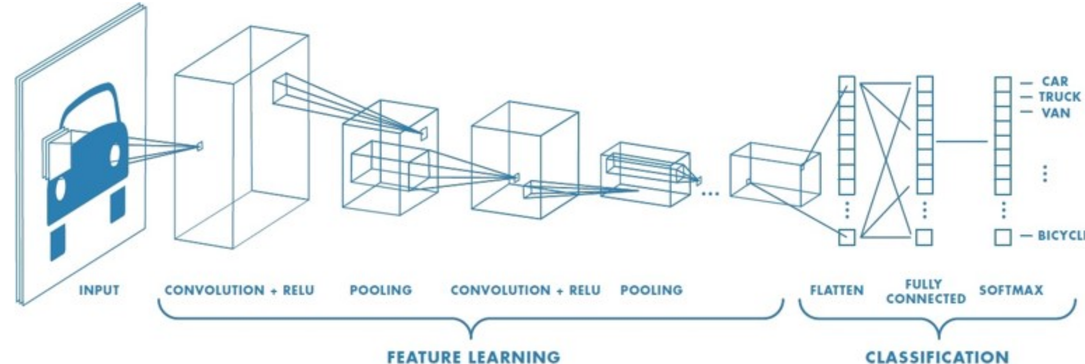


Figure 2.6: An example of a typical CNN. Convolutions are applied to images, and the output of each convolved image is used as the input to the next layer. Pooling layers are used to subsample the images. After obtaining the feature map, all the features are linked using an FC layer and softmax is applied to get the final results [Mathworks, 2021].

Convolutional layers. Convolutions detect features such as edges, texture and patterns. In Deep Learning, the principle of convolution is to slide a mask over an image. In convolutional network terminology, the **image** is the input of a layer $\mathbf{Z}_k$ and the mask is referred to as **kernel** and consists of estimated weights $\mathbf{W}_k$.

Convolutions are usually applied to more than one dimension. For example, the result of convolving a two-dimensional image $\mathbf{Z}_k$ of size $h \times w$ with a two-dimensional kernel $\mathbf{W}_k$ of size $h' \times w'$ is given by

$$\mathbf{S}(i,j) = (\mathbf{Z}_k * \mathbf{W}_k)(i,j) = \sum_{m=1}^{h'} \sum_{n=1}^{w'} \mathbf{Z}_k(m,n)\mathbf{W}_k(i-m, j-n). \tag{2.20}$$

The indexes $i,\ j,\ m,\ n$ in the convolution are supposed to verify the boundary conditions. An example of a two-dimensional convolution is given in Figure 2.7.

It is well known that discrete convolutions can be performed as matrix multiplication $\mathbf{S}$. This is done by transforming one of the inputs into a **Toeplitz matrix** [Gray, 1972] at the cost of introducing redundant data. A common used approach in neural networks is doing an image-to-column transformation ***im2col*** [Chetlur et al., 2014, Li et al., 2019] which transforms the local regions of the input image into columns and the weights of the CONV layer into rows. The results are equivalent to performing a large matrix multiplication which needs to be reshaped to the proper output dimension.

Three-dimensional convolutions are more often used. The third dimension represents the number of filters or channels used for an image. A filter has multiple channels. A channel can correspond to a color channel (an image has three channels: red, green, blue) or to the output of a previous filter.

Pooling layers. When using convolutions, small changes to the input can easily impact the feature map. A common approach to solve this problem is to reduce the size of the feature map and

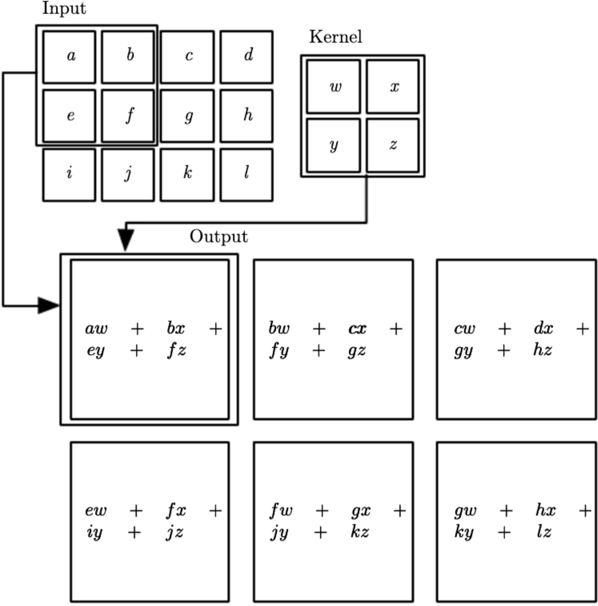


Figure 2.7: A two-dimensional convolution [Goodfellow et al., 2016].

keep the most important elements. A method for down-sampling is to use a pooling layer. The most common pooling operation is Max Pooling. It calculates the maximum value within a region of pixels from the feature map, which corresponds to the more important features. For a two-dimensional image $\mathbf{Z}_k$ of size $h \times w$, the output dimension is given by p the pooling window size and s the stride. For the (i, j) pixel in the output image, Max Pooling is applied as follows:

$$\text{POOL}^{\text{MAX}}(\mathbf{Z}_k)(i,j) = \max_{m \in [0,p;s], n \in [0,p;s]} \mathbf{Z}_k(i+m, j+n), \tag{2.21}$$

where $m \in [0, p; s]$ means that the index m goes from 0 to p with a step s. A visual representation is also given in Figure 2.8.

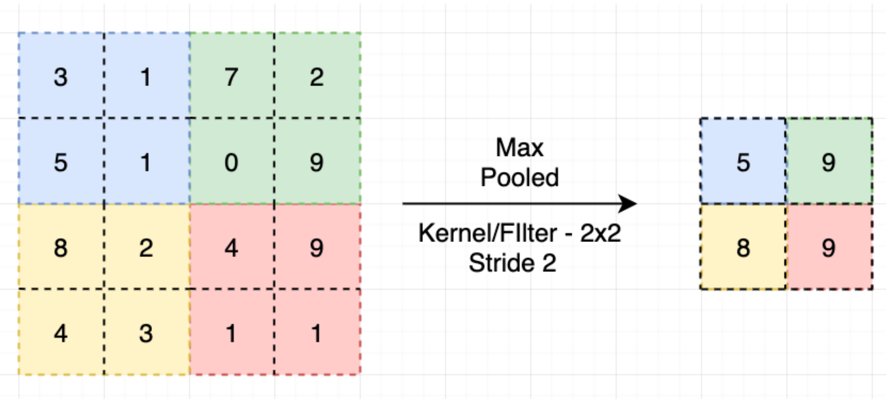


Figure 2.8: An example of Max Pooling with a 2×2 filter and a stride of size 2 [Rana, 2020].

The two types of layers mentioned above are generally used to construct the feature map, as shown in Figure 2.6. However, another type of layer called **Batch Normalization** [Ioffe and Szegedy, 2015] has been used in more recent network architectures [He et al., 2016]. The Batch Normalization layer (BN) is used to standardize the inputs of the layers. It takes a batch of inputs and normalizes them by centering and reducing the scale. This makes the training faster and the networks more stable. An example of network is given in Figure 2.9.

In the training stage, the BN layer depends on mini-batches to learn the statistical description of the inputs (mean and variance). Optionally, it applies a scale γ and an offset β. In the inference phase, BN is applied as a single linear transformation. For a given input $\mathbf{x}$, BN is written as follows:

$$\mathrm{BN}_{\gamma,\beta}^{\text{inference}}(\mathbf{x}) = \frac{\gamma}{\sqrt{\mathrm{Var}[\mathbf{x}] + \epsilon}}\mathbf{x} + \left(\beta - \frac{\gamma\mathbb{E}[\mathbf{x}]}{\sqrt{\mathrm{Var}[\mathbf{x}] + \epsilon}}\right), \tag{2.22}$$

where ϵ is an arbitrary small constant added for numerical stability. Note that $\mathbb{E}[\mathbf{x}]$ and $\mathrm{Var}[\mathbf{x}]$ depend on the distribution of the random variable $\mathbf{x}$.

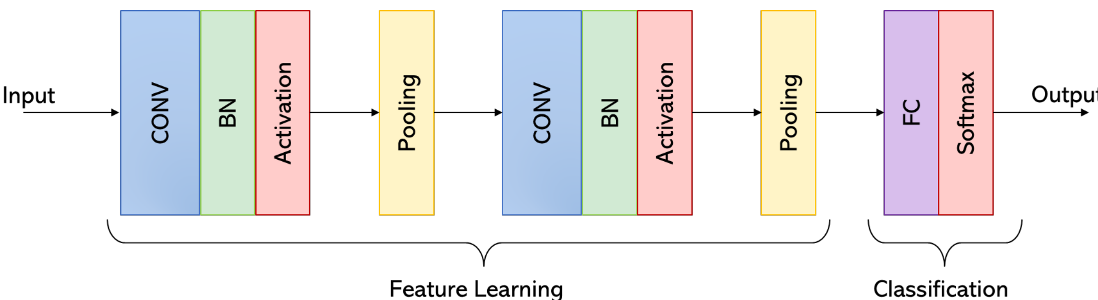


Figure 2.9: An example of network which uses the BN layer. Usually, BN is inserted after CONV layers and before the activation function.

After constructing the feature map, fully connected layers are used to connect the features all together and to perform the classification task as shown in Figure 2.6.

2.2.1 Datasets

In order to train and test the models, a dataset is required. This dataset is usually split into three sets: training, validation and testing. The validation set is used in the training stage to check how the training goes. Sometimes, the validation set is not provided. In this case, data scientists split the training set. In the case of image classification, the images need to be labelled into at least 2 categories, or classes.

The most popular datasets used for classification are presented below.

Figure 2.10: Samples of the MNIST Dataset.

MNIST [LeCun et al., 1998]. The MNIST dataset (see Figure 2.10) contains images of handwritten digits (0 through 9). MNIST is a starting point dataset in model training which consists of 10 classes. It is used in image classification to train classifiers, simply to test new architectures and to ensure they work. MNIST is divided into two sets: the training set has 60K examples and the test set has 10K. All of its images are in black and white and of the same size (28×28 pixels).

CIFAR-10/100 [Krizhevsky et al., 2009]. The CIFAR-10 dataset (see Figure 2.11) is also a well-known image classification benchmark dataset. The dataset has 60K color images with 10 different classes comprising 6K images per class. The images are colored RGB and of size 32×32 pixels each. There are 50K training images and 10K test images. CIFAR-100 is just like the CIFAR-10, except it has 100 classes. There are 500 training images and 100 testing images per class.

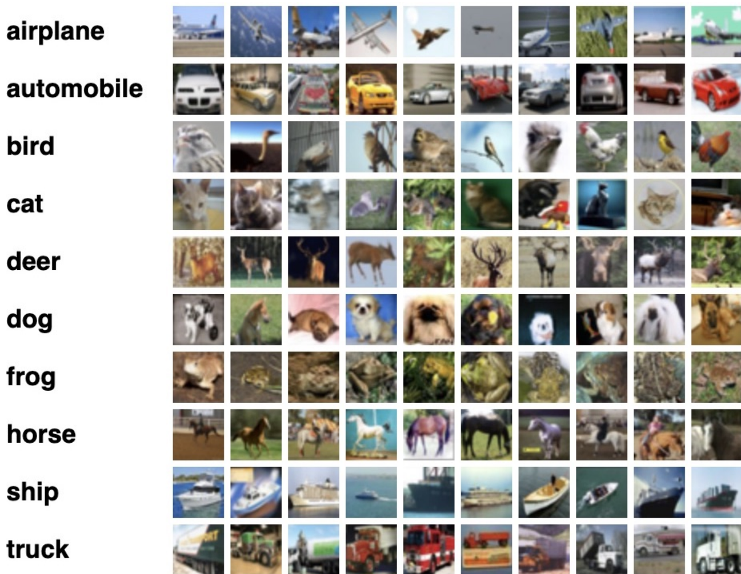


Figure 2.11: Samples of the CIFAR Dataset.

ImageNet [Deng et al., 2009]. ImageNet (see Figure 2.12) is a dataset of over 15 million labeled high-resolution images belonging to roughly 22K categories. The images were collected from the web and labeled by human labelers. This dataset is well-known since 2010, because it is used in a competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories. In all, there are roughly 1.2 million training images, 50K validation images, and 150K testing images. The dataset provided in ILSVRC are generally used to show the performances of models at a bigger scale.

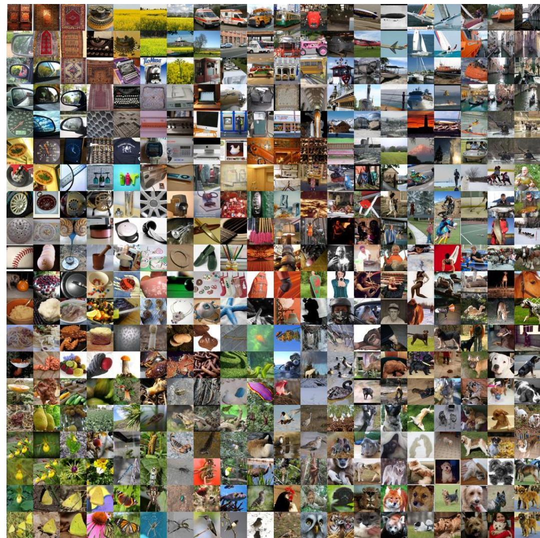


Figure 2.12: Samples of the ImageNet Dataset.

2.2.2 Well-known Networks

Many CNN architectures have shown great results. The first known CNN architecture was done in the 90s and is called **LeNet-5** [LeCun et al., 1998] network. This basic architecture consists of two blocks of a convolutional and an average pooling layer, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier. This architecture was used on the MNIST dataset.

In 2012, with the appearance of **AlexNet** [Krizhevsky et al., 2012], neural networks have become more precise for image classification and more popular in real applications. AlexNet consists of 5 convolutional layers and 3 fully connected layers and uses ReLU activation function, multiple GPUs and overlapping Max Pooling. After AlexNet, the trend was to improve the network's accuracy by adding more convolutional layers. As a result, the VGG network was introduced, the most used version being **VGG16** [Simonyan and Zisserman, 2015] which has 13 convolutional and 3 fully connected layers. This network has 138M parameters and takes up about 500MB of storage space.

Other architectures have been introduced in order to reduce the number of parameters and to achieve lower error rates. Among the first network to use Batch Normalization layers and residual connections were Residual networks. The most popular used architecture is **ResNet50** [He et al., 2016]. **Inception-v1** [Szegedy et al., 2015] is the first architecture using blocks/modules instead of stacking convolutions. This architecture offers a radical reduction in the number of parameters. **Xception** [Chollet, 2017] is an adaptation from Inception, where the Inception modules have been replaced with depthwise separable convolutions. **SqueezeNet** [Iandola et al., 2016] is an updated and improved version of AlexNet with 50x fewer parameters than the prior model. Mobilenets are a class of efficient models that are conceived for mobiles and embedded systems. These networks are based on a streamlined architecture and use depth-wise separable convolutions. **MobileNet** [Howard et al., 2017] has 4.25M parameters and it requires 16MB memory size.

To improve accuracy and processing time, some architectures are scaled up when more resources are available. This is the case of VGG that scale up from 11 to 19 layers and ResNet from 18 to 200 layers, but other networks use larger input image resolutions or increase depth and width. **EfficientNet** [Tan and Le, 2019] has been the most recent state-of-the-art network family for high quality and quick image classification. EfficientNet uses a compound coefficient to uniformly scale all dimensions of depth/width/resolution. The baseline architecture is called EfficientNet-B0, and it is based on a technique used in **MobileNetV2** [Howard et al., 2018]. Baseline model achieves 1% more accuracy than ResNet50, while having $5\times$ fewer parameters and $11\times$ fewer operations. Models from B1 through B7 are scaled up from baseline B0. EfficientNet-B7 achieves state-of-the-art 84.4% on ImageNet while being $8.4\times$ smaller and $6.1\times$ faster than Gpipe [Huang et al., 2019]. In 2021, **EfficientNetV2** [Tan and Le, 2021], an improved version of EfficientNet has been released.

Computational Workload. The computational workload of a CNN inference is the result of an intensive use of Multiply Accumulate (MAC) operation. Floating point operations (FLOPs) are often used to describe how many operations are required to run a single instance of a given model. Most of these operations occur in the convolutional and fully connected layers. Convolu-

Model	Year	Acc (Top 1)	Memory size	Parameters	FLOPs
LeNet-5	1998	99.1%	3.5 MB	866K	28M
AlexNet	2012	57.3%	217 MB	62.3M	724M
VGG16	2014	71.3%	528 MB	138M	15.5G
VGG19	2014	72.7%	548 MB	144M	19.6G
GoogleNet	2015	68.7%	40 MB	7M	1.58G
InceptionV3	2015	78.8%	92 MB	23M	6G
ResNet50	2015	75.3%	98MB	25.6M	3.86G
ResNet101	2015	76.4%	171 MB	44.4M	7.57G
ResNet152	2015	77.0%	232 MB	60M	11.3G
SqueezeNet	2016	57.6%	4.7 MB	1.25M	861.34M
Xception	2016	79.0%	88 MB	22.9M	11G
MobileNetV1	2017	70.4%	16 MB	4.25M	569M
MobileNetV2	2018	71.3%	14 MB	3.5M	480M
Gpipe (Amoeba-Net)	2018	84.4%	2.1GB	557M	225.7G
EfficientNet-B0	2019	76.3%	46 MB	5.3M	4.07G
EfficientNet-B7	2019	84.4%	256 MB	66M	37G
EfficientNetV2-S	2021	78.7%	101 MB	21.1M	8.42G

Table 2.2: Overview of well-known CNNs. Accuracy evaluated on ImageNet test set (except for LeNet-5 which is evaluated on MNIST).

tional layers are generally responsible for more than 90% of execution time during the inference [Abdelouahab et al., 2018]. On the other hand, most of the parameters are coming from the fully connected layers. Due to this unbalanced computation to memory ratio, CNNs accelerators follow different strategies when implementing the convolutional and fully connected parts of inference.

An overview of the presented architectures is given in Table 2.2. We display the year when the network appeared, its performance (accuracy), its size (memory size and number of parameters) and computational workload (FLOPs). Also, more details can be found in [Khan et al., 2020].

2.3 Object Detection architectures

Object detection is another computer vision task that involves identifying, classifying, but also giving the location of one or more objects in a given image. After image recognition and classification, this task became the focal point for research and industry since it is closer to real life applications such as autonomous cars and surveillance systems. The difference between classification and object detection is shown in Figure 2.13. It is a challenging problem that uses a method of object recognition and classification (i.e. CNNs) and adds an object localization task that gives the position and the size of the object.

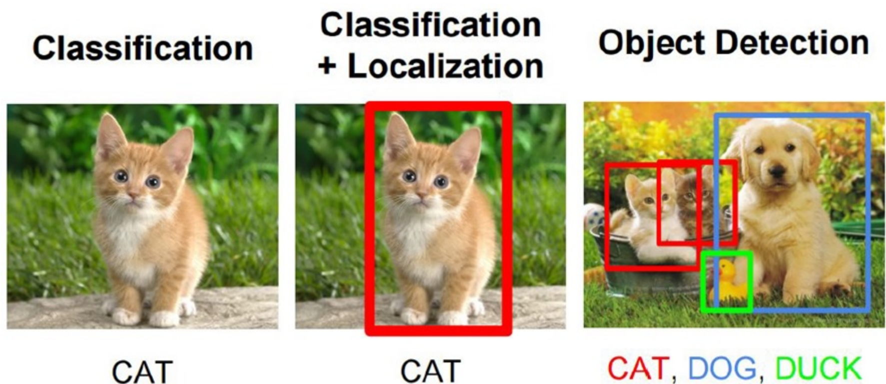


Figure 2.13: Difference between classification, object detection [Cogneethi, 2021].

Generally, there are three steps in an object detection model: **region of interest selection**, **feature extraction** and **classification**.

Region of interest selection. This step is used to locate objects in an image and indicate their location with a bounding box/region. The model first proposes a set of regions by using selective search [Uijlings et al., 2013] or using a regional proposal network [Ren et al., 2015]. A second approach used for region selection is by sliding a window on the image.

Feature extraction and classification. The two other steps are done on each region candidate by using a CNN architecture such as VGG16, ResNet, MobileNet. These steps are straightforward, but it can be quite slow. Some detection models are described as having 2,000 regions per image at test time [Girshick et al., 2014]. A different family of networks skips the region selection stage and runs the detection directly over the dense sampling of possible locations. These models are faster and simpler, but are potentially less precise. These detectors are preferred for real time applications, therefore, research has focused mostly on improving these type of networks.

Mean average precision (mAP). The evaluation metric used for object detectors is called the mean average precision [Everingham et al., 2010]. It is different from the accuracy used in classification as it has to take into consideration the intersection of the bounding boxes. For each bounding box, an overlap between the predicted bounding box and the ground truth bounding box is measured. This is given by the intersection over union (IoU):

$$\mathrm{IoU} = \frac{\text{area of intersection}}{\text{area of union}}. \tag{2.23}$$

The IoU is used to find the Precision and Recall. Precision measures how accurate the prediction is:

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \tag{2.24}$$

where TP stands for true positive and FP for false positive. While Recall measures how well the predictions were made:

$$\mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{TN}}, \tag{2.25}$$

where TN stands for true negative. Then, the Average Precision (AP) represents the area under the precision-recall curve (PR curve). It is given by averaging the precision over a set of evenly spaced recall levels. Finally, mAP is the average of AP over all classes C:

$$\text{mAP} = \frac{1}{C}\sum_{i=1}^{C} \text{AP}_i. \tag{2.26}$$

2.3.1 Datasets

Pascal VOC [Everingham et al., 2010]. The PASCAL Visual Object Classes Challenge (Pascal VOC) dataset contains 20 object categories. Each image in this dataset has pixel-level segmentation annotations, bounding box annotations, and object class annotations. This dataset has been widely used as a benchmark for object detection, semantic segmentation, and classification tasks. The most used VOC datasets are the ones from 2007 and the one from 2012. The VOC07 dataset has a total of 9963 images, making a total of 24640 objects. The images are divided into a training (2.5K), validation (2.5K) and test (5K). The VOC12 dataset has a total of 11540 images and 27450 objects that are divided in a training set (5.7K), a validation set (6K) and a test set (11K). In order to have more samples for training, some works train networks on the union of the two datasets.

MS COCO [Lin et al., 2014]. The Microsoft Common Objects in Context (MS COCO) dataset is a large-scale object detection, segmentation and captioning dataset (see Figure 2.14). The dataset consists of 328K images. For object detection, the dataset has bounding boxes and per-instance segmentation masks with 80 object categories. The first version of MS COCO dataset was released in 2014. It contains 164K images split into training (83K), validation (41K) and test (41K) sets. In 2015, an additional test set of 81K images was released, which included all the previous test images and 40K new images. Based on community feedback, in 2017 the training/validation split was changed from 83K/41K to 118K/5K. The new split uses the same images and annotations. The 2017 test set is a subset of 41K images of the 2015 test set.

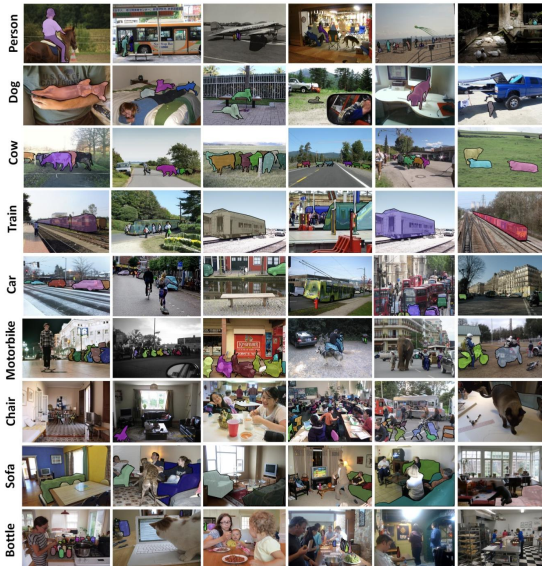


Figure 2.14: Samples from MS COCO dataset.

KITTI [Geiger et al., 2013]. Karlsruhe Institute of Technology and Toyota Technological Institute (KITTI) is one of the most popular datasets for autonomous driving (see Figure 2.15). It consists of hours of traffic scenarios recorded with a variety of sensor modalities, including high-resolution RGB, grayscale stereo cameras, and a 3D laser scanner. The dataset contains 7481 training images and 7518 test images, making it a total of 80256 labeled objects.

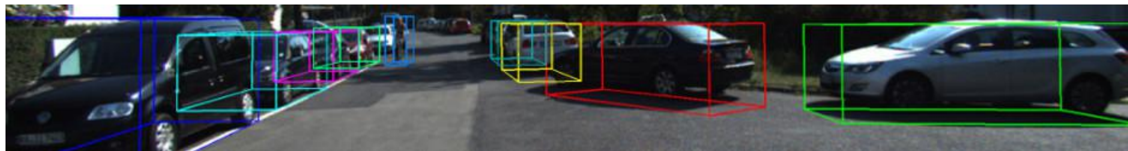


Figure 2.15: Samples from KITTI dataset.

2.3.2 Well-known Networks

Numerous networks for object detection have appeared in the past decade. Table 2.3 gives an overview over the most popular networks. Object detection models are generally based on CNN architectures and easily configurable (input size and number of regions). In contrast to classification networks, the level of complexity may also vary for the same architecture because of the input size, the numbers of selected regions or some parameters that can be configurable. Therefore, we do not add the memory size or number of parameters. We provide the frame per second (FPS) which is critical because these type of models target real-time applications.

Model	Year	Dataset	mAP	FPS	Input resolution
R-CNN	2014	VOC07	66	0.28	227 x 227
Faster R-CNN (VGG16)	2015	VOC07	73.2	7	1000 x 600
YOLO (VGG16)	2016	VOC07	66.4	21	448 x 448
SSD512	2016	VOC07	76.8	19	512 x 512
Retina-Net-101	2017	COCO	53.1	11	500 x 500
YOLOv2	2017	COCO	48.1	40	608 x 608
YOLOv3	2019	COCO	57.9	20	608 x 608
YOLOv4	2020	VOC07+12	82.39	44	512 x 512

Table 2.3: Table of well-known object detection networks. The FPS was obtained on different GPUs. It serves as an indicator of the evolution made in object detection.

Object detection networks can be divided in two staged Detectors and one stage Detectors.

Two staged Detectors. **R-CNN** [Girshick et al., 2014] uses an algorithm called Selective Search which selects around 2000 candidate boxes and feeds them to a CNN followed by a Support Vector Machine (SVM) to classify. Selective search uses local cues like texture, intensity, color to generate all the possible locations of the object. The detector has a second output for the bounding boxes which uses regression. Improvements have been brought to this model, and it was transformed in **Faster R-CNN** [Ren et al., 2015]. This algorithm uses a network called Region Proposal Network (RPN) that outputs proposed regions in a more efficient manner.

One stage Detectors. **You Only Look Once** (YOLO) [Redmon et al., 2016] is one of the popular algorithms in object detection used by researchers and developers all over the world. The reason why YOLO is so popular is that compared to the other networks it processes images in real time still achieving a good mAP. The original version divides each image into a grid of $s \times s$ and each grid predicts n bounding boxes and a confidence score which indicates whether the box contains an object or not. A lot of versions and improvements have been made for YOLO, making it the reference algorithm used in object detection. However, some other models have appeared, for example the **Single Shot Detector** (SSD) [Liu et al., 2016] approach discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios. The Single Shot Detector network combines predictions from multiple feature maps with different resolutions to naturally handle objects of various sizes. **Retina-Net** [Ross and Dollár, 2017] is another network that introduces new techniques to extract features.

2.4 Conclusion

The role of this chapter was to present Deep Learning notions, in order to give the reader a better understanding of the general scientific context of this work. We presented the most commonly used layers in classification networks and we gave an overview of state-of-the-art CNNs. A brief presentation of object detection and an overview of state-of-the-art networks was also given. Throughout this thesis, we focus mostly on feedforward and CNN type architectures. Several networks which are presented in this chapter will be used in evaluations. Most importantly, we formally defined neurons, layers and the architecture of a feedforward network. These key elements will be used in the following chapters. The next chapter focuses on state-of-the-art compression strategies used on CNNs.

CHAPTER 3

Deep Learning Compression

In this chapter, we present the state-of-the-art for deep learning compression. We start by giving a general overview of the compression techniques for neural networks. Then, we give more details on pruning and quantization algorithms.

3.1 General presentation of techniques **31**
3.2 Network Pruning **33**
3.3 Network Quantization **35**
3.3.1 Scalar and vector quantization 36
3.3.2 Low precision quantization 37
3.3.2.1 Reduced precision 37
3.3.2.2 Binary and ternary networks 39
3.4 Conclusion **40**

Many real-time applications such as autonomous cars, smart cameras and smartphones use Deep Learning models that are based on CNNs. However, CNNs have a massive number of parameters and operations and these real-time applications usually use limited hardware and require low latency. Thus, deploying CNNs can be challenging.

A solution for this issue is compressing the parameters of the networks which helps us meet the memory and inference time requirements. Many compression techniques are used to develop efficient networks. Several surveys can be found on the subject [Sze et al., 2017, Cheng et al., 2018, Ge, 2018, Neill, 2020, Mishra et al., 2020, Berthelier et al., 2021, Gholami et al., 2021]. A presentation of these techniques are given in the following sections.

3.1 General presentation of techniques

During this thesis, a study has been done over a great number of works and techniques. We split these techniques into multiple categories: quantization, pruning, computational acceleration, low rank factorization and knowledge distillation (see Figure 3.1).

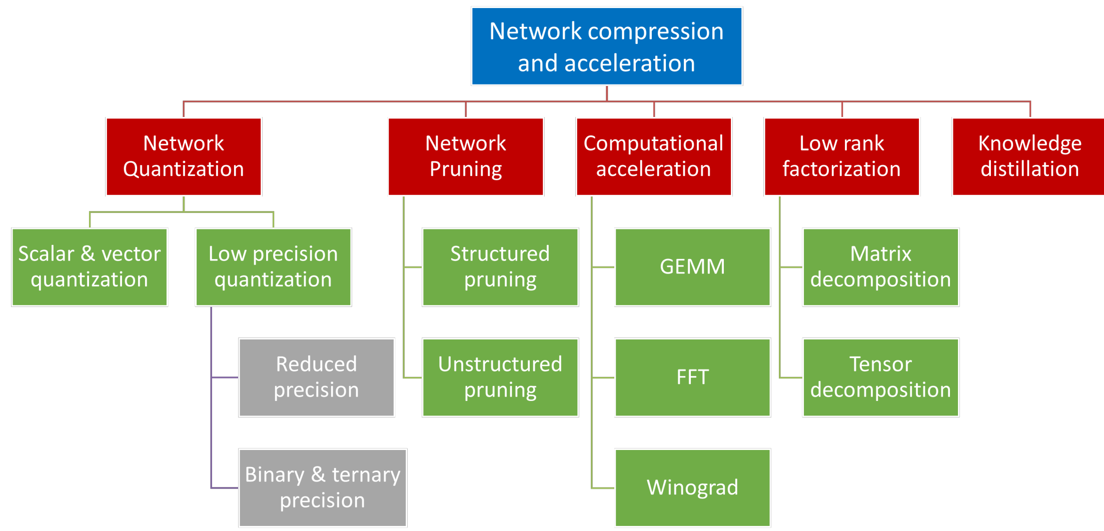


Figure 3.1: The main categories of network compression approaches (inspired from [Ge, 2018]).

To give a complete panorama of the compression methods, we will start by giving a short overview for each technique. We further present the two most important and popular methods in Deep Learning compression: pruning and quantization. These two methods work well together. This is shown in [Han et al., 2016b] where a three-step pipeline is used to compress a network (see Figure 3.2). Two of the steps are pruning and quantization, and the third one is Huffman encoding [Huffman, 1952]. This method compresses an AlexNet [Krizhevsky et al., 2012] network up to 35x without sacrificing performance. Another combination between pruning technique and a quantization scheme is proposed in [Hacene et al., 2020] which effectively reduces the complexity of convolutions.

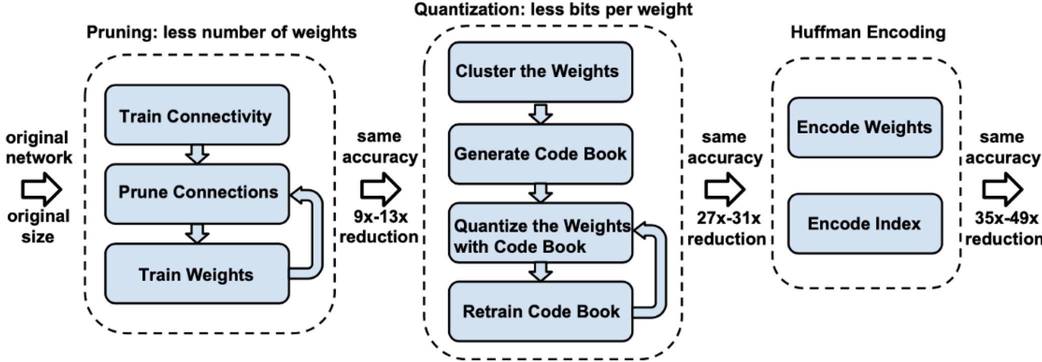


Figure 3.2: Deep Compression three-step pipeline [Han et al., 2016b].

In the end, we will focus only on quantization. The reasons for choosing quantization to the detriment of pruning are mentioned in the following sections. An experimental study of the two techniques is given in the next chapter.

Knowledge distillation. Knowledge distillation (KD) effectively learns a small student model from a large teacher model. It has received rapid, increasing attention from the community [Bucilua et al., 2006, Ba and Caruana, 2014, Urban et al., 2016, Hinton et al., 2015]. The main idea is that the student model mimics the teacher model in order to obtain a competitive or even a superior performance. Basically, a KD system is composed of three key components: knowledge, distillation algorithm, and teacher-student architecture. The knowledge is what we extract from the teacher model. The key problem is how to transfer the knowledge from a large teacher model to a small student model. A comprehensive survey on all the elements and the challenges on this subject is given in [Gou et al., 2021]. Although KD shows great success, it is out of the scope of this thesis due to the need to train new networks.

Low rank factorization. Networks are usually over-parameterized [Denil et al., 2013]. Low rank factorization methods identify redundant parameters by using matrix and tensor decomposition. Nyström method [Kumar et al., 2012, Giffon et al., 2019] is used to efficiently generate low-rank approximations. The popular low-rank approximation approach based on singular value decomposition (SVD) [Eckart and Young, 1936, Klema and Laub, 1980] is generally applied to the weights of fully connected layers where compact storage is achieved by keeping only the most prominent components of the decomposed matrices. Matrix and tensor decomposition methods such as Tucker [Tucker, 1966], Canonical Polyadic (CP) [Harshman et al., 1970] or product of sparse matrices [Giffon et al., 2021] are also handy tools for speeding up inference with networks with many parameters. The decomposed layers are represented by layers with reduced parameter dimensions. While these approaches reduce storage size and time complexity, they also introduce a high amount of error. Moreover, some works [Yu et al., 2017b, Phan et al., 2020] show that low rank factorization works hand in hand with other sparsity strategies such as pruning.

Computational acceleration. In order to speed up the execution of CNNs, efficient matrix multiplication algorithms have been proposed. A common method for dealing with layers' complexity

is to use GEMM [Cong and Xiao, 2014], a matrix multiplication procedure that is part of the BLAS library [Lawson et al., 1979]. Some works [Sze et al., 2017] show that GEMM is efficient for FC layers, while on CONV layers it can lead to redundant data in the input. Fast Fourier Transform (FFT) [Mathieu et al., 2014, Vasilache et al., 2015] is a well known algorithm for convolutions. Most convolutions are computed using FFT. Another computational transformation is Winograd's algorithm [Winograd, 1980]. In [Lavin and Gray, 2016], it is shown that Winograd obtains x7.28 speed up compared to GEMM and that FFT work well for convolutions with large filters (size larger than 5) and Winograd for small filters (size smaller than 3).

In this thesis, we mainly focus on the memory size issue and, differently from the previously presented approaches, we rely on a compression-based approach. Compression-based methods aim to reduce DNN complexity either by pruning [Mao et al., 2017, Anwar et al., 2017] its weights, or by quantizing them [Gong et al., 2014, Gysel, 2016, Hubara et al., 2016, Rastegari et al., 2016, Krishnamoorthi, 2018, Jacob et al., 2018]. In some of these works, the compression step is considered during training [Rastegari et al., 2016, Campbell and Broderick, 2018], while in others [Han et al., 2016b, Krishnamoorthi, 2018] compression is carried out in a post-training step. In both cases, the objective is to choose compressed DNN weights such that the incurred loss of inference performance with respect to the uncompressed performance is as small as possible.

3.2 Network Pruning

Pruning is a popular technique used to remove unimportant parameters. This technique increases significantly the sparsity of the parameters, and, if applied correctly, may reduce the memory footprint and computational costs of neural networks. If the hardware is adapted, multiplications can also be omitted, which provides an even more efficient inference. Pruning was introduced in early development of neural networks [Reed, 1993].

Recently, many approaches have been proposed. They are mainly divided in two categories: structured pruning approaches and unstructured pruning approaches. As presented in [Cheng et al., 2018], unstructured pruning refers to *fine-grained* pruning and the structured pruning approaches are *vector-level*, *kernel-level*, *group-level* and *filter-level* pruning. Figure 3.3 shows the different types of pruning.

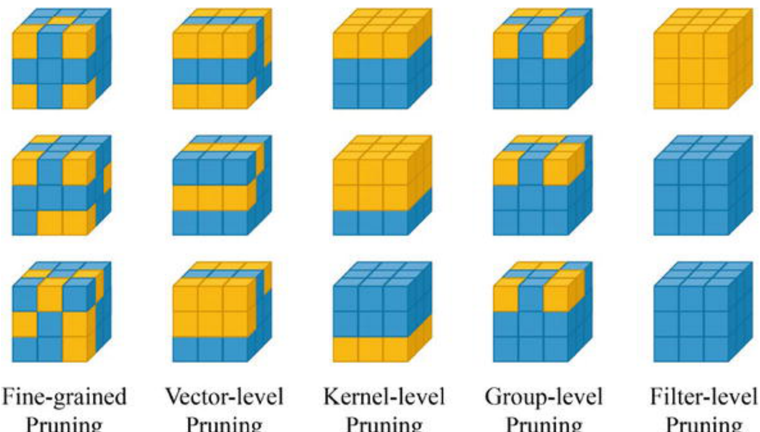


Figure 3.3: Pruning methods applied to a 4-dimensional weight tensor of a convolution layer [Cheng et al., 2018]. This example contains three 3-D filters of three kernels each. A filter is a cube and a kernel is a slice in the cube. The elements in yellow are pruned weights. The fine-grained approach removes the parameters in an unstructured way. The vector-level approach prunes a vector from a kernel, while the kernel-level method prunes a kernel from the filter. Group-level methods use a sparse pattern to prune the parameters on the filters. Finally, filter-level pruning remove filters.

Structured pruning. Structured pruning [Anwar et al., 2017, Guo et al., 2016, Mao et al., 2017, Molchanov et al., 2017, Frankle and Carbin, 2018, Dai et al., 2019, Tessier et al., 2020] removes channels or filters. When applying structured pruning, neurons are also removed. In practice, this method is easier to implement, as it translates to simply reducing the dimensions of the matrices without making important changes to the architectures. However, structured pruning suffers from considerable accuracy loss and limits the sparsity rate. Anwar *et al.* [Anwar et al., 2017] applies a structured pruning on three levels (filter, kernel and intra-kernel) and greatly reduces the complexity of convolution calculations. Mao *et al.* [Mao et al., 2017] finds that vector pruning has better performance because it takes up less space than *fine-grained* pruning.

Unstructured pruning. Unstructured pruning [Han et al., 2016b, Wen et al., 2016, He et al., 2017, He et al., 2019, Tessier et al., 2020] focuses on removing unnecessary individual weights, which ensures a higher flexibility and usually achieves high compression rate with minor accuracy loss. However, this approach is known to be hard to accelerate. It gives an irregular sparsity and requires additional information to locate the non-zero weights during inference. This approach needs to be used on flexible hardware with more cache, such as CPUs. Unstructured pruning is not recommended for GPUs because with this method, GPUs are underutilized and can decrease inference speed [Wen et al., 2016].

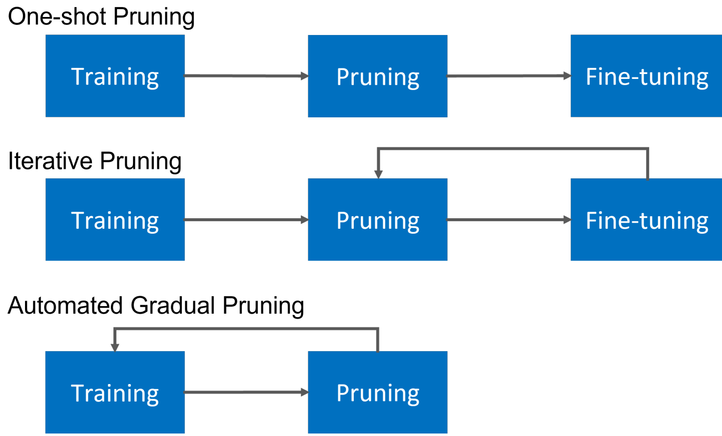


Figure 3.4: Pruning scheduling types.

Another important part of pruning is how and when we apply the method. Figure 3.4 shows the different scheduling methods used. This highly impacts the performance of the network. In one-shot pruning, we only remove the weights of a trained network and uses a fine-tuning (retraining) step to adapt the remaining parameters to the removal. Iterative pruning [Han et al., 2016b] applies pruning followed by fine-tuning several times and leads to better results. The last method, called Automated gradual pruning [Zhu and Gupta, 2018], includes pruning in the training step.

Keeping state-of-the-art performance while imposing high levels of sparsity during training and inference is still an open problem. However, we recall that the company's processor is used only for the inference phase. Since pruning usually requires the models to be retrained afterwards as shown in Figure 3.4 and our processor does not have this capability, we consider pruning out of the scope o f this thesis, but the results and advances on the subject are still of interest for future generations of the processor.

3.3 Network Quantization

Quantization is an approach that has shown great success for both inference and training. Quantization exploits the sparsity of DNNs to reduce both storage and processing requirements. Several approaches have been published on the quantization of CNNs networks, which deal with the problem of memory and of speed of calculations.

Quantization refers to the process of approximating continuous amplitude data with a finite, preferably small, set of amplitude values. The input to a quantizer corresponds to the original data, and the output always corresponds to one value among a finite number of levels. An in-detail description of quantization for data compression will be given in Chapter 5. Here, we present the works that have been done in network quantization.

Generally, neural networks use 32-bit floating-point (FP32) precision for both training and inference which leads to large computational and storage costs. To save memory, the proposed ap-

proaches for quantizaton focus on reducing the precision used to represent the parameters. We divide quantization techniques into two categories. The first category groups methods that are used to quantize the parameters for storage purposes. This category we call **scalar or vector quantization**. The second category is called **low precision quantization** and it refers to reducing the precision requirements of the weights and activation using low precision formats which not only reduces the size of the parameters but also changes the precision of the operations.

3.3.1 Scalar and vector quantization

Scalar and vector quantization approaches originate from data compression. The main idea is to create a codebook which contains a set of values that are used to represent the original data. Then the values of the original data are mapped to the ones in the codebook. The codebooks are dictionaries or look-up tables (LUTs). The size of the codebook is much smaller than the original data, these methods obtain a significant compression ratio without sacrificing the accuracy. Several methods use this type of approach to represent the parameters or weights of the networks.

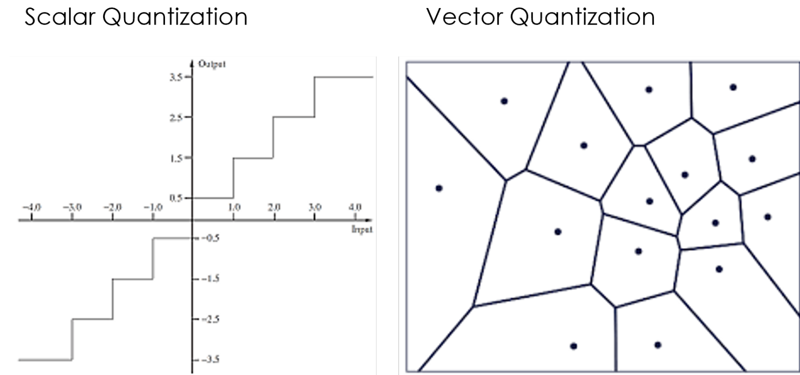


Figure 3.5: Scalar quantization vs 2D Vector quantization.

In **scalar quantization** (SQ) [Jacob et al., 2018, Han et al., 2016b], each input value is processed separately and outputs a single value. In **vector quantization** (VQ) [Gong et al., 2014, Le Tan et al., 2018] the input data is a vector and is processed and mapped to a vector output. VQ can be seen as a generalization of SQ to the space $\mathbb{R}^N$. Figure 3.5 shows a visual representation of the two quantization methods and how the space is partitioned.

In information theory [Cover and Thomas, 2006, Gersho and Gray, 1991], vector coding is considered to always obtain a better performance than scalars, even if the samples are independent random variables. However, in practice, SQ seems to give satisfactory results and few VQ algorithms have been used for network compression.

The classical methods can achieve a very large compression ratio without loss in accuracy.
Gong *et al.* [Gong et al., 2014] applies VQ on FC layers in networks for image classification and object detection and obtains a compression ratio of 16-24 with only 1% loss in accuracy for classification on ImageNet.

Han *et al.* [Han et al., 2016b] quantizes the weights by using a non-uniform quantization algorithm called k-means [Steinhaus, 1957, MacQueen, 1967] and applying it on each weight of the network. Then, a LUT is used to share the values from the codebook. The whole compression process used by Han *et al.* is shown in Figure 3.2. Quantization has shown a reduction of 27x-31x while maintaining the same accuracy.

Product quantization (PQ) [Gong et al., 2014, Jégou et al., 2011] is a form of vector quantization, but it is applied on blocks (subvectors). This method splits each column vector into blocks and learns the same codebook for all the subvectors. Each quantized vector is obtained by assigning its subvectors to the nearest codeword in the codebook. PQ generalizes both scalar and vector quantization.

In Deep Learning, low precision quantization is a form of scalar quantization. Due to the popularity of this subject, details on low precision methods are given in the following Section.

3.3.2 Low precision quantization

3.3.2.1 Reduced precision

Another way to quantize models is by reducing the precision requirements of the weights and activations of the model. While Deep Learning computations normally rely on standard binary 32 IEEE 754 floating-point (FP32) arithmetic, it has been observed that significant savings in memory footprint and increases in performance/efficiency can be achieved by using 16-bit representations for training [Micikevicius et al., 2018] and 8-bit representations for inference with acceptable precision loss [Jacob et al., 2018].

Low precision quantization is the most studied approach. In many cases, neural networks are very tolerant to low numerical accuracies. A floating-point model can be quantized to a fixed point model with almost no loss of precision. Gupta *et al.* [Gupta et al., 2015] uses a 16-bit fixed-point representation with stochastic rounding to train a CNN. Courbariaux *et al.* [Courbariaux et al., 2015a] has good results after training with 10-bit for activations and 12-bit for parameters. Such an approach brings several advantages: the memory footprint is smaller (we reduce the data size), the transfer is faster, and we need less RAM and cache for activations. Consequently, the energy consumption is reduced. On the other hand, after each calculation step, the new data must also be compressed to a low precision format. This step adds to the computational complexity.

Currently, the mainstream approach for Deep Learning inference with alternate arithmetic relies on integer quantization of FP32 using 8-bit formats. Using, for example, 8 bits precision numbers reduces the model size by a factor of 4. It also reduces the working and cache memory for activations, which makes the computations much faster and consumes less power, since moving 8-bit data is 4 times more efficient than moving 32 floating-point data [Krishnamoorthi, 2018].

Recently, new data formats have been introduced due to their hardware efficiency. The format **Bfloat16** (BF16) [Dean et al., 2012] is a 32-bit floating-point representation, truncated into 16 bits. It retains the characteristics of a float32, but only supports a 7-bit mantissa. BF16 is faster than float16 and its precision seems to be sufficient for CNNs. MSFP8 [Chung et al., 2018] is similar

to BF16. It uses a float16 truncated into 8 bits. Another data type called **Posit** [Gustafson and Yonemoto, 2017] is designed to replace floats. Posits offer higher precision while being simpler in hardware, thus more economical in energy consumption. A thorough presentation on these alternative data formats is given in Chapter 6.

Next, we present different industrial solutions used for faster inference by having efficient kernel computations in reduced precision.

R&D Solutions for Industries. The need for compression in Deep Learning is becoming increasingly important. Companies are contributing significantly to advances in the field. Quantization schemes used by industrial solutions allow neural network inference to be carried out using integer-only arithmetic, a more efficient method than using only floating-points for inference. Faster inference through efficient computation is achieved when using Gemmlowp, a small self-contained low-precision GEMM library [GemmlowP, 2019], Nvidia TensorRT [Nvidia, 2017] and Rosetta [Borisyuk et al., 2018].

Gemmlowp is a low precision matrix multiplication library that is used by Tensorflow [TensorFlow, 2021] and Tensorflow Lite [TensorFlow Lite, 2021]. The compression of the weights and inputs is done in unsigned 8-bit fixed point (uint8). The method used is uniform quantization, ensuring that the zero value is quantized without error. Krishnamoorthi [Krishnamoorthi, 2018] shows that, for an 8-bit quantized array, the storage size is reduced by a factor of 4, inference is 2-3x faster on CPUs and 10x faster on processors with fixed-point SIMD capabilities.

Rosetta takes inspiration from Jacob *et al.* [Jacob et al., 2018] and it quantizes the weights and activations from float32 to uint8. The approach is identical to that of Gemmlowp. However, the quantization does not apply to all layers in order to improve the accuracy of the prediction. A calibration dataset is used to define an input saturation threshold by minimizing the L2 norm of the prediction error, unlike TensorRT which minimizes the relative entropy. TensorRT also uses 8-bit inference performed using uniform quantization to move from a 32-bit floating-point representation (FP32) to a signed 8-bit fixed point representation (INT8). The bias is ignored and the activations are saturated at a threshold defined by a calibration data set. A detailed study on TensorRT is also given in the Appendix A.

Caffe [Jia et al., 2014], another well-known framework, implements an automatic approximation tool, Ristretto [Gysel, 2016], to compress float32 data to any data format. Ristretto uses 3 quantization strategies: dynamic fixed point [Courbariaux et al., 2015a], minifloat and power of two [Tang and Kwan, 1993]. Gysel [Gysel, 2016] shows that the dynamic fixed point is most suitable. This approximation gives the best result for low precision, but it requires more chip area than the classical fixed point.

MicroAI [Novac et al., 2021] is a framework specially designed for end-to-end deep neural networks training, quantization and deployment for microcontrollers. Execution is done either in FP32 or fixed-point using 16 bits or 8 bits. The quantization scheme can be easily adjusted for different use cases. The quantization method used, first, computes the number of bits required to represent the unsigned integer, then, it uses this information to determine the scaling factor needed to truncate the real value.

3.3.2.2 Binary and ternary networks

Binarization is another form of quantization. Binarization reduces the data representation to 1 bit, which means two values (i.e. $\{-1,+1\}$). The advantages of binarization are computation reduction because the networks are quantized in a simple way, memory saving (32x) and significant acceleration (58x speedup [Rastegari et al., 2016]) because binarization replaces the multiplication operation with operations that are more hardware friendly.

Another upside of binarization is that binary neural networks (BNNs) are more robust than full precision networks because of the small magnitudes and also help understand the behavior and structure of the model and the importance of the layers.

A type of BNNs is called the **Naive Binary Neural Networks** that directly quantize the weights and activations to 1-bit. In 2015, [Courbariaux et al., 2015b] proposes **BinaryConnect**, a method that quantizes FP32 weights in $+1$ and -1 using stochastic binarization. This network achieved state-of-the-art results, but only on small datasets. Following the paradigm of BinaryConnect, Courbariaux proposed another method called **Binarized Neural Network** [Hubara et al., 2016] which binarizes weights and activations during training and replaces traditional convolutional operations with **XNOR** and **popcount** operations to have a full binary network.

However, direct binarization leads to large quantization errors. Other solutions have been brought to minimize this error. Some methods keep some layers in full precision. One of the pioneer methods in binarization is presented in [Rastegari et al., 2016]. It has two versions, one called **Binary Weight Networks** in which weights are binarized and the second one, **XNORNET**, where both weights and activations are binarized. This method uses Binarized Neural Network with scaling factors, except for the first and last layer which remain in FP32. Rastegari *et al.* also changes the order of the layers in the CNN as shown in Figure 3.6.

Figure 3.6: A CNN block (left) vs a XNORNET block (right) [Rastegari et al., 2016]. In a typical CNN block, the order of operations is the following: CONV, Batch Norm, activation function. In a XNORNET block, the input is first normalized and, then, we apply a binary activation and a binary CONV. The Pooling operation is in the last position in both cases.

Ternary networks use 2-bit data. The second bit is needed to represent the 0 value. The QNN network is also proposed by Hubara *et al.* [Hubara et al., 2017], it uses the same architecture as Binarized Neural Network, but with 2-bit activations. Zhu *et al.* [Zhu et al., 2017] proposes a ternary model that adapts the scaling factors for each layer. Despite their high compression ratio and reduced computational complexity, there is a significant drop in accuracy. Therefore, the use

of such a model does not seem reliable enough for industrial applications.

More details on how binary networks work, the techniques that are used and what is their performance can be found in [Qin et al., 2020].

3.4 Conclusion

In this chapter, we presented several strategies used to compress deep learning networks. We displayed structured and unstructured pruning techniques. Pruning usually needs to alter the architecture of the network or retrain the weights, which is not possible in the company's context. We end the chapter with an overview of the state-of-the-art in quantization which will be our main focus throughout this thesis. Scalar quantization seems to be one of the most popular methods with a significant number of developed algorithms. We identified several interesting candidates, including uniform and non-uniform quantization. Various possible ways to reduce the precision of the traditionally used FP32 by using INT8, alternative floating-point formats, or even 1 bit representation.

In the next chapter, a practical study is presented. State-of-the-art methods such as pruning, scalar quantization and binarization are put to test on classification networks LeNet-5 and ResNet50.

CHAPTER 4

Preliminary study on compression methods

This chapter presents a preliminary study which focuses on testing several compression methods such as pruning, quantization, binarization. We compare these methods to each other and to Deep Compression [Han et al., 2016b], a state-of-the-art method which jointly uses pruning, quantization and Huffman encoding. The experiments were performed using Lenet5 and Resnet50 on small datasets such as MNIST, CIFAR-10.

4.1 Methods **43**
4.1.1 Pruning 43
4.1.2 Quantization 43
4.1.3 Binarization 44
4.1.4 Deep Compression 44
4.2 Experiments **44**
4.2.1 Experimental settings 44
4.2.2 Compression performance comparison 46
4.3 Conclusion and Perspectives **51**

4.1 Methods

We take a look at well-known compression methods such as pruning, quantization and binarization. We performed experiments in order to compare different compression techniques which can be used for storage purposes. For each of these methods, we compute the compression rate τ which is a measurement of the bit-rate reduction in size of data representation after applying the compression algorithm. A description of the three methods is given below.

4.1.1 Pruning

One way of applying pruning is to force to 0 the value of the weights which have weak connections [Han et al., 2015]. We focus on iterative pruning, which means that we prune weights, and then retrain the pruned network.

Pruning is applied to each layer of the network. We first define the desired compression rate, which indicates the number of weights we set to 0 in all the layers. Given p the total number of pruned weights and n the total number of weights, the compression rate is defined as follows:

$$\tau_p = 100\left(1 - \frac{p}{n}\right). \tag{4.1}$$

4.1.2 Quantization

We use a scalar quantization scheme, and we apply it post-training, only to the weights of the layers. No compression is applied to the inputs of the layers. This compression method aims to store the weights as an R-bit integer in order to save memory space. We do not apply the quantization to the biases.

For an R-bit quantization, the number of quantization levels is given by $N = 2^R - 1$. The clamping range is defined between a and b, a being the smallest value and b the largest. For a given weight matrix $\mathbf{W}$, generally, the range is set between $\min(\mathbf{W})$ and $\max(\mathbf{W})$. We set a and b so that we can exactly represent the value 0 as an integer. The quantization algorithm [Krishnamoorthi, 2018] is defined as follows:

$$\delta = \delta(a, b, N) = \frac{b-a}{N} \tag{4.2}$$

$$w_q = q(w, a, b, N) = \left\lfloor \frac{w-a}{\delta} \right\rceil \delta + a \tag{4.3}$$

$$\hat{w} = w_q|_{[a,b]} = \min(\max(w_q, a), b), \tag{4.4}$$

where $\lfloor \cdot \rceil$ represents the rounding operation and $w|_{[a,b]}$ the clamping of w between a and b.

We use the following compression rate of the quantized model:

$$\tau_q = 100\left(1 - \frac{R}{32}\right). \tag{4.5}$$

4.1.3 Binarization

For binarization, we use Binary Weight Networks [Rastegari et al., 2016] which use FP32 activations and binary weights. In order to binarize a tensor of weights, we estimate the original FP32 weights $\mathbf{W}$ using a binary filter $\mathbf{W}_B$ and a scaling factor δ such as

$$\mathbf{W} \approx \delta \mathbf{W}_B. \tag{4.6}$$

The optimal binary tensor $\mathbf{W}_B$ is given by the sign of the weights and the scaling factor δ is given by

$$\delta^* = \frac{||\mathbf{W}||_1}{n}, \tag{4.7}$$

where n is the number of elements in $\mathbf{W}$ and $||\mathbf{W}||_1$ is the 1-norm.
The compression rate we use for binarization is the same one as for quantization when $R = 1$:

$$\tau_b = 100\left(1 - \frac{1}{32}\right). \tag{4.8}$$

4.1.4 Deep Compression

For the last compression method considered, we combined pruning, quantization and added a coding algorithm called Huffman coding [Huffman, 1952]. This method is called Deep Compression [Han et al., 2016b], and it is presented as a way to reduce almost 98% of the size of the network with almost no loss of accuracy.

Pruning is used to remove useless connections. Quantization creates clusters with the remaining non-zero weights. Each cluster is represented by a value, and all the weights from the same cluster will be replaced by the representative value given to the cluster. Finally, Huffman Coding is a lossless variable-rate source coding method, used in order to compress further by reducing the number of bits needed to encode the results of quantization.

4.2 Experiments

4.2.1 Experimental settings

We first perform experiments using a simple model, LeNet5 [LeCun et al., 1998], on two datasets (MNIST, CIFAR-10). The datasets are presented in Section 2.2.1.

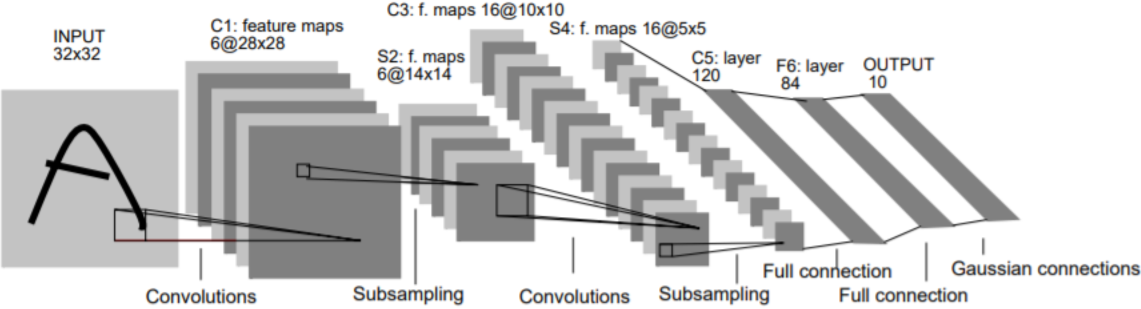


Figure 4.1: Architecture of LeNet5 [LeCun et al., 1998].

As shown in Figure 4.1, the model is composed of three convolutional layers and two fully connected layers. Down-sampling using Max Pooling is applied after the two first convolutions. The network returns a vector of size C, which corresponds to the number of classes of the dataset.

A similar experiment is performed using ResNet50 [He et al., 2016], a larger scale network with a different architecture. This model is composed of a total of 50 convolutional and fully connected layers grouped into blocks as shown in Figure 4.2. The network also uses layers of Max Pooling and Batch Normalization.

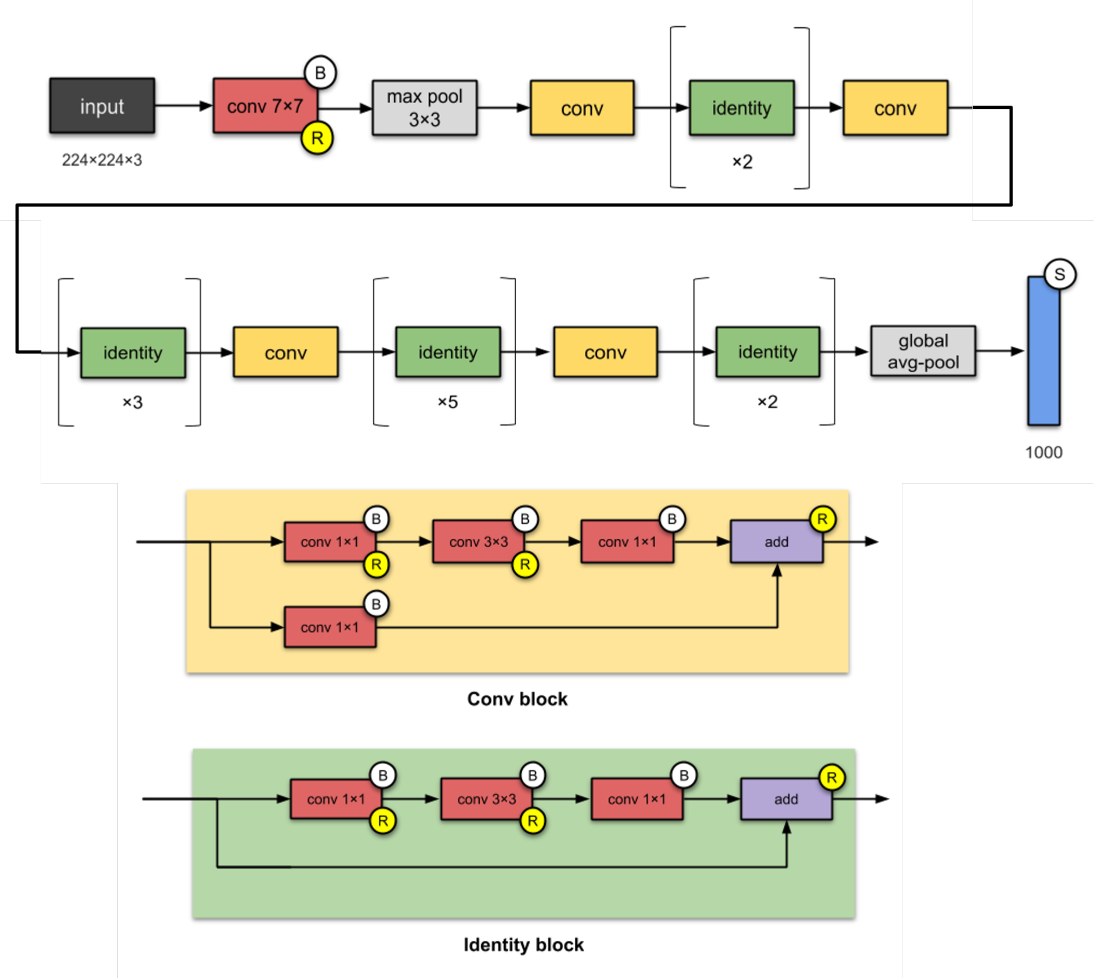


Figure 4.2: Illustration of ResNet50's architecture from [Karim, 2019]. Symbol **B** stands for Batchnorm layer, **R** stands for ReLU activation function and **S** represents Softmax.

We evaluate the accuracy of the network without compression. We obtain an accuracy of 99.1% for LeNet5 on MNIST, 64.5% for LeNet5 on CIFAR-10 and 82.27% for ResNet50 on CIFAR-10. Then, we compress the network using each one of the presented methods, and we compare the results. We are aware that the accuracy of ResNet50 could be improved by using training strategies, however this is not our goal. We recall, our goal is not to test training strategies and obtain the best

accuracy of a network, but to study the accuracy drop with respect to the compression method used and the percentage of compression rate.

4.2.2 Compression performance comparison

Pruning. We first present the benchmark of pruning with and without retraining. We applied pruning on LeNet5 at different compression rates, and we compared the accuracy on MNIST and CIFAR-10 datasets (see Figures 4.3 and 4.4).

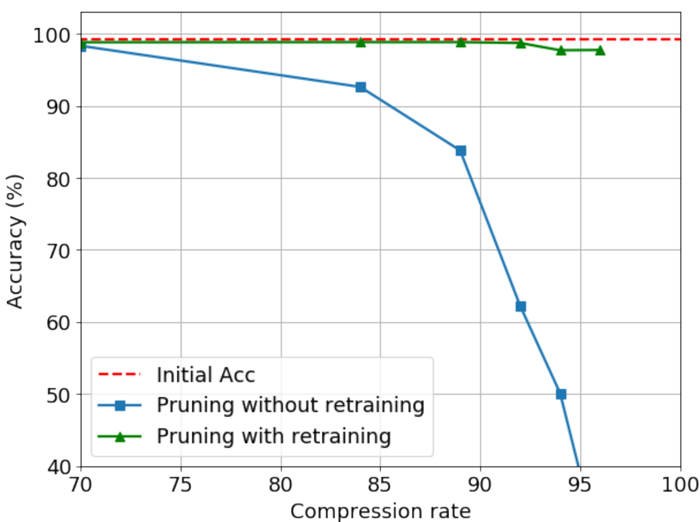


Figure 4.3: Accuracy comparison in the case of pruning with and without retraining using Lenet5 on MNIST.

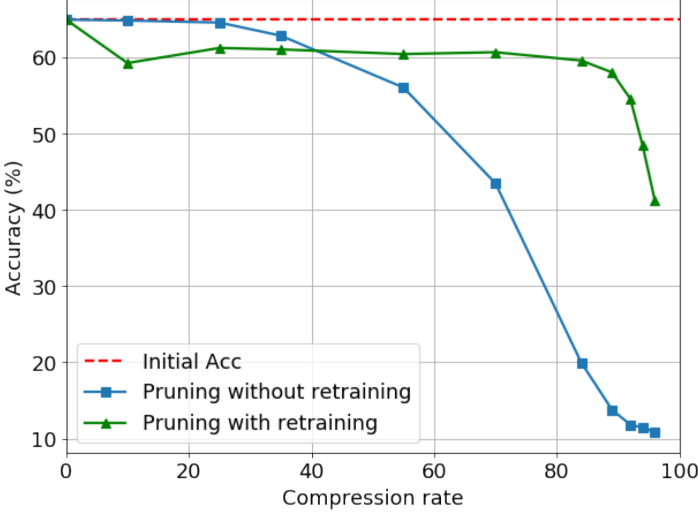


Figure 4.4: Accuracy comparison in the case of pruning with and without retraining using Lenet5 on CIFAR-10.

In red, we represent the accuracy of the original LeNet5 network. The blue curve shows the accuracy we obtain when pruning without retraining, and in green we have the accuracy of pruning with retraining. We can see that applying pruning with retraining generally provides much better accuracy than pruning alone. An interesting observation is that for a small compression rate (up to 40% on CIFAR-10), pruning without retraining performs better than with retraining. However, if we want to achieve higher compression rate, retraining the models improves connections. Iterative pruning is known to perform a 90% compression rate without loss of accuracy on simple datasets such as MNIST, but some accuracy drop is noticed in the case of larger datasets (i.e. CIFAR-10). In our experiments, we notice 5-7% accuracy drop for a compression rate around 90%.

Quantization. For convolutional layers, we notice that the range of the weights is not the same for each channel tensor. We applied R-bit quantization on the whole weight tensor, then on each output channel tensor. We refer to the first method as Layer Quantization and the second one as Channel Quantization.

When testing quantization with different bits applied to LeNet5 on the MNIST dataset, it results that Layer and Channel Quantization have the same accuracy for quantization from 30 to 6 bits. When we use less than 6 bits (from 5 to 2), Channel Quantization performs on the average a gain of 1% accuracy in comparison to Layer Quantization. For future benchmark comparison, we will use Channel Quantization.

When testing CIFAR-10 and MNIST datasets (see Figures 4.8 and 4.7), Channel Quantization performs the same accuracy as the uncompressed LeNet5 for bits in range of 30 to 3 bits. When using 2-bit or 1-bit quantization, accuracy drops and for 1-bit quantization the model cannot classify correctly images (accuracy near 10%). To conclude, Channel Quantization is the best method to compress the weights of a neural network, without loss of accuracy, when the compression rate does not exceed 93%.

Binarization. By applying the binarization rules seen in the previous section, we can easily compute BWN. Since we no longer use 32 bits weight values, the compression rate is about 96% (4.8). This has an impact on the accuracy of the original neural network, but it depends on the model used and the tested dataset.

When testing LeNet5 on MNIST (see Figure 4.7), the binarized LeNet5 with the BWN algorithm performs 88.1% accuracy compared to 99.1% accuracy for the original model. A large performance drop can be noticed in the case of image classification using a bigger and more complex dataset. In the case of CIFAR-10 dataset (see Figure 4.8), binarized LeNet5 accuracy drops to 17.6% for a 64.5% accuracy of the original LeNet5. To solve this issue, other binarization schemes are proposed. A solution would be to retrain the binarized neural network and binarize it again after each epoch, but the gradient also needs to be binarized and projected, which is out of scope of our work.

Deep Compression. We will present the accuracy of Deep Compression by studying benchmarks of Pruning alone, Pruning and Quantization, and finally Deep Compression (Pruning + Quantization + Huffman Coding). We applied compression on LeNet5 network with these meth-

ods, and we tested their accuracy on CIFAR-10 dataset. You can find the benchmark in Figures 4.5 and 4.6.

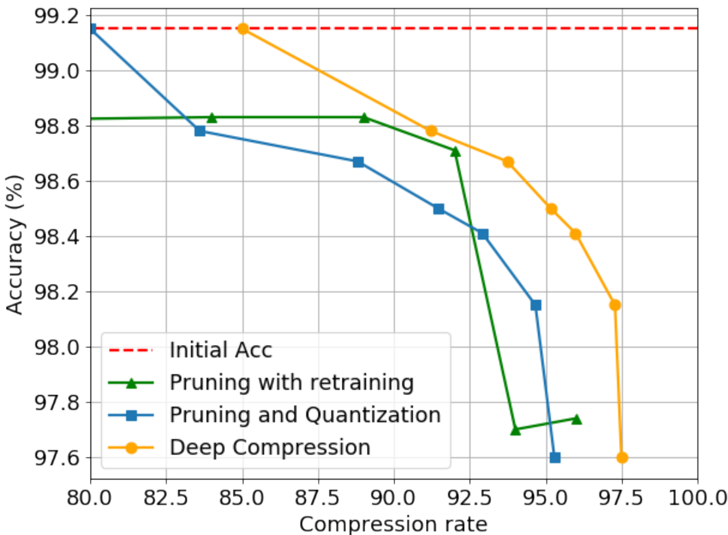


Figure 4.5: Accuracy comparison between the steps used in Deep Compression: pruning with retraining, pruning with quantization and, finally, Deep Compression (with Huffman Coding). The model used is Lenet5 on MNIST.

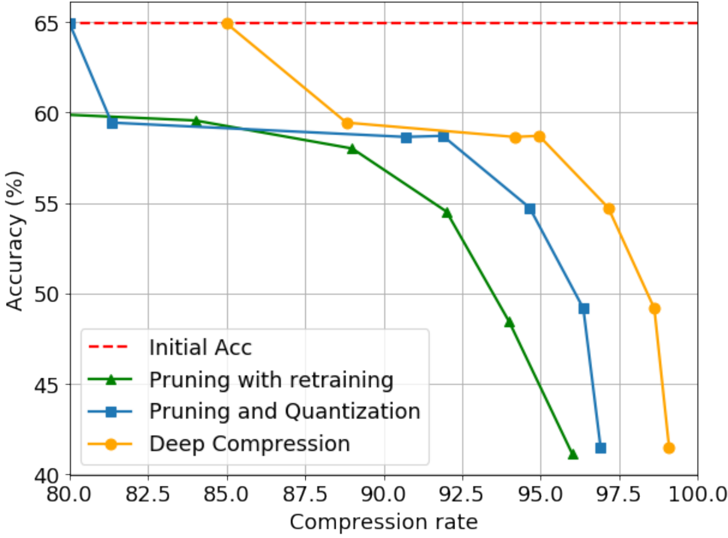


Figure 4.6: Accuracy comparison between the steps used in Deep Compression: pruning with retraining, pruning with quantization and, finally, Deep Compression (with Huffman Coding). The model used is Lenet5 on CIFAR-10.

In this benchmark, we notice a minor loss of accuracy when Deep Compression is applied in the case of LeNet5 on MNIST. When using CIFAR-10, the loss in accuracy is greater, but the model

still performs well given the compression rate achieved. When applying pruning and quantization together, the accuracy is better than when retrained pruning alone, we compress further without loss of accuracy. We can explain it by the fact that when we prune too many weights (compression rate over 90%), accuracy will drop fast because too many weights are forced to zero. While when we apply quantization, we further compress the network without forcing more weights to zero, but assigning them to a close representable value. When running Huffman Coding, accuracy remains the same as before, but we gain on the average 5% compression than the quantization compression rate.

Deep Compression is said to compress up to 98% without loss of accuracy. When we implemented this method, we could not get a lossless model when tested on CIFAR-10, this dataset being far more difficult to classify than MNIST. The paper we read showed lossless compression on MNIST, we could compress LeNet5 at the same compression rate tested on MNIST with a low loss of accuracy (between 1% and 3%, with an 99% original accuracy). In the case of CIFAR-10, for a compression rate of 97%, a 7% accuracy drop is noticed.

LeNet5 on MNIST. Figure 4.7 displays the results for each method using LeNet5 on MNIST. As explained before, in red we represent the accuracy of the original LeNet5 (uncompressed model), the red dot is the accuracy of binarization, the green curve is the accuracy of Channel Quantization, the blue curve is the accuracy of Pruning with retraining, and the orange curve is the accuracy of the Deep Compression method (Pruning + Quantization + Huffman Coding). X-coordinate is the compression rate of LeNet5, and the Y-coordinate is the accuracy of the compressed model. To better show the results of compression, we "zoom" on the picture: X-coordinate is between 80 and 100, Y-coordinate is between 85 and 100. We can see that Channel Quantization, Pruning and Deep Compression performs very well until a compression rate of 94% (an average loss of less than 1% compared to uncompressed LeNet5).

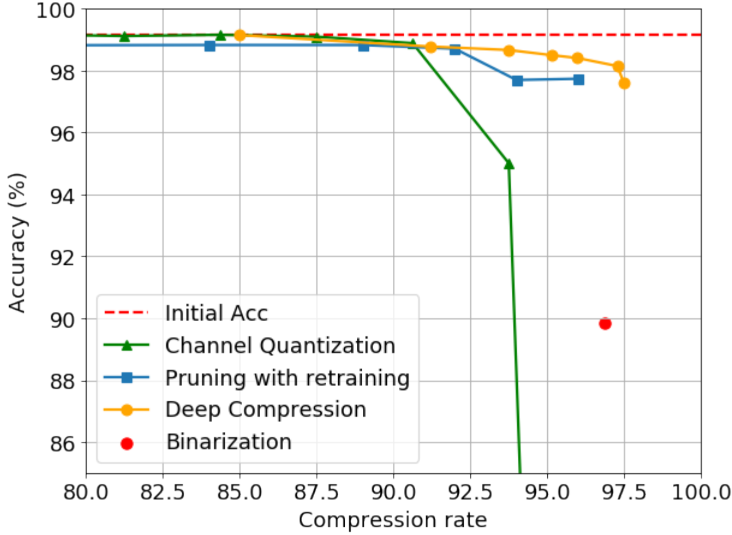


Figure 4.7: Accuracy comparison between pruning, quantization, deep compression and binarization using Lenet5 on MNIST.

For a compression rate greater than 94%, the accuracy of Channel Quantization drops significantly and Pruning accuracy begins to drop. Pruning still performs a very good accuracy, but then fails to maintain a good accuracy when reaching a 97% compression rate. Binarization achieves a better accuracy than the others model do, except for Deep Compression, when reaching this extreme compression rate (88% accuracy for 97% compression rate). But as previously explained binarization works well in the case of small networks and simple datasets like MNIST. On the other hand, Deep Compression performs well achieving a compression rate of 98% with a loss in accuracy of 1%, which confirms that this method is the best one.

LeNet5 on CIFAR-10. Figure 4.8 shows the benchmark results using Lenet5 on CIFAR-10. There is an obvious loss of accuracy on CIFAR-10 greater than with MNIST. This is certainly due to the complexity of the dataset. Until a compression rate of 90%, all the methods maintain a good accuracy (accuracy drop under 5%). Afterwards, the accuracy of pruning and quantization gradually drops. Pruning obtains around 40% accuracy, while Channel Quantization achieves only 12% accuracy for 97% compression rate, similar to the benchmark done on MNIST. Binarization performs similar to Channel Quantization because without retraining, the weights are not properly adapted to larger datasets. Finally, we can see that, as for MNIST, the Deep Compression methods still perform well at the very high compression rate of 97%, even with an accuracy drop of 11% compared to LeNet5 (53.5% accuracy Deep Compressed model, 64.5% LeNet5).

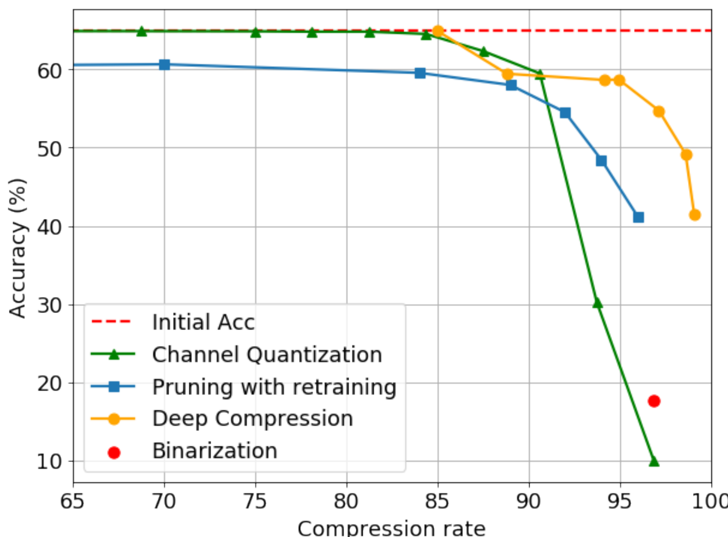


Figure 4.8: Accuracy comparison between pruning, quantization, deep compression and binarization using Lenet5 on CIFAR-10.

Resnet50 on CIFAR-10. Results on a large scale network like Resnet50 on CIFAR-10 are shown in Figure 4.9. They are similar to the results previously presented. The drop in accuracy is less significant for Pruning and Deep Compression. Pruning obtains 75% accuracy while achieving 96% compression rate. The best results are again obtained using Deep Compression with 99% compression rate and only 5% drop in accuracy. Channel Quantization gives good results up to

88% of compression rate and past this point, the accuracy drops fast. Binarization gives 10% accuracy, the same result as Channel Quantization.

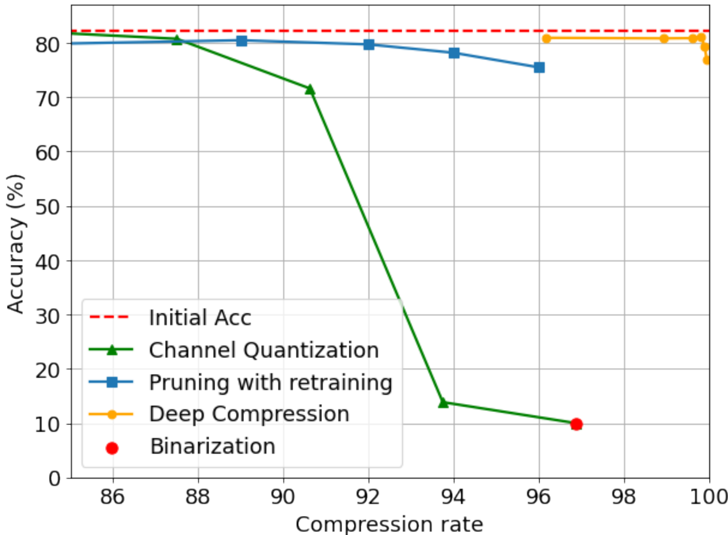


Figure 4.9: Accuracy comparison between pruning, quantization, deep compression and binarization using ResNet50 on CIFAR-10.

4.3 Conclusion and Perspectives

Binarization is limited in its performance due to the absence of retraining, which is out of the scope of this thesis. Channel Quantization, Pruning and Deep Compression methods perform well and reduce accuracy loss even when we compress the model at a higher rate. Regarding the benchmarks presented, the Deep Compression method is the one to favor, but both pruning and Deep Compression require retraining the models. Without retraining, the performance of these methods would not be acceptable. We recall that we are looking for methods which can be used on the MPPA. The MPPA is a proceesor which can be used for deep learning inference tasks. Training is not supported.

Given the context of our work, for the rest of the thesis, we will focus on quantization. Given that layer quantization is only 1% less performant than channel quantization, we will focus on the former method.

CHAPTER 5

Uniform vs non-uniform quantization for storage purposes

This chapter proposes a numerical study on uniform and non-uniform quantization applied to deep neural networks. We are looking to study the rate-distortion trade-off achieved when quantizing the weights of the networks in order to reduce the memory storage used for the parameters. The second goal of this chapter is to see how the layers react to compression (i.e. which layers we should compress more and which should not). In this chapter, we also introduce data compression and rate-distortion theory. To evaluate the quantization methods, we use the accuracy and two distortion measures: Kullback-Leibler divergence and the Mean Squared Error. This helps us to identify which one of the quantization methods performs better, but also to verify if these measures are useful in deep network compression.

5.1 Data Compression **55**
5.1.1 General compression workflow 55
5.1.2 Scalar quantization 56
5.1.3 Uniform quantization 56
5.1.4 Non-uniform Quantization 57
5.2 Rate distortion theory **59**
5.2.1 Background on rate-distortion theory 59
5.2.2 Rate distortion trade-off 60
5.3 Experiments **61**
5.3.1 CNN on MNIST 61
5.3.2 VGG on CIFAR-100 66
5.4 Conclusions and Perspectives **68**

This thesis focuses on quantization because it allows us to reduce the size of previously trained networks without relying on special libraries. We identify two trends in quantization: low precision quantization, often used in the industrial initiative, and classical quantization. This chapter focuses on the second approach, while the next chapter will focus on a form of low precision quantization.

Here, we evaluate inference performance under two types of scalar quantization uniform and non-uniform. Non-uniform quantization is more efficient, but it requires managing a look-up table. On the other hand, uniform quantization is easier to use, but it is expected to be less efficient. This is why we compare the two quantization methods, and we apply them to the parameters of each layer of pre-trained neural networks.

The performance of the quantizers is evaluated by looking at the accuracy of the classification and two other types of distortions. The first distortion concerns the weights of the network, while the second concerns the statistical distribution at the output of the CNN. We wish to highlight the behavior of the two and which of the two is linked to the accuracy. Last but not least, we study the impact of the error for each layer of the network. The objective of this work is to answer several questions. How can we choose the best method to apply to reduce the complexity of a neural network without losing performance? How can we know if a network has been well compressed? In particular, what are the appropriate evaluation criteria?

This chapter is organized as follows. Section 5.1 introduces data compression and the scalar quantization methods used to compress neural networks. Section 5.2 introduces rate distortion theory. Section 5.3 describes the numerical experiments performed. Finally, Section 5.4 concludes the chapter.

5.1 Data Compression

5.1.1 General compression workflow

Compression is an application of information theory that originated with Shannon [Shannon, 1948]. The goal of data compression is to represent information in a more compact form. Compressed data uses a reduced number of bits, so it occupies less space than the original data.

Generally, compression workflow (see Figure 5.1) is formed by 3 main stages: transformation, quantization and coding. There can also be a stage of pre-processing to make the compression more effective.

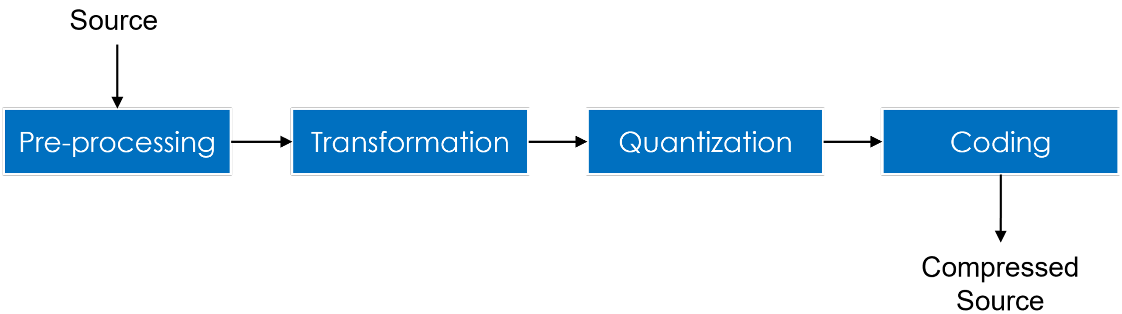


Figure 5.1: Compression workflow.

The interest of applying a compression algorithm to a neural network is to be able to reproduce its classification qualities as well as possible with a network that is less expensive in terms of time, memory space and energy consumption. More precisely, it is a question of representing the function $f_{\boldsymbol{\theta}}$ by a function $f_{\hat{\boldsymbol{\theta}}}$ where $\hat{\boldsymbol{\theta}} = Q(\boldsymbol{\theta})$ is a compressed form of $\boldsymbol{\theta}$. We recall the vector of parameters $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_{K+1}), \text{ where } \boldsymbol{\theta}_k = (\mathbf{W}_k, \mathbf{b}_k). \tag{5.1}$$

The compression function $Q : \mathbb{R}^{|\boldsymbol{\theta}|} \to \mathcal{A}$, where $\mathcal{A}$ is a finite discrete set, reduces the representation size of the vector $\boldsymbol{\theta}$.

5.1.2 Scalar quantization

Let w be a component of $\boldsymbol{\theta}$ a parameter of a trained neural network. We define Q_R^t a scalar quantizer of size $L = 2^R$ where R is the number of quantization bits and $t \in \{U, NU\}$ indicates the type of quantization (U for Uniform and NU for Non-Uniform). The quantizer Q_R^t transforms a continuous value of the interval $\mathcal{D} = [\boldsymbol{\theta}_{\min}, \boldsymbol{\theta}_{\max}] \subset \mathbb{R}$ into a discrete value $\mathcal{A} = \{a_1, a_2, ..., a_R\} \subset \mathbb{R}$. The quantizer partitions the interval $\mathcal{D}$ into several subintervals $\mathcal{D}_i = [d_i, d_{i+1}[$, for $1 \leq i \leq L$, such that

$$\hat{w} = Q_R^t(w) = a_i \text{ if } w \in [d_i, d_{i+1}[, \tag{5.2}$$

where d_i, for $1 \leq i \leq L+1$, are the quantization thresholds.

The quantization step $q_i = d_{i+1} - d_i$ corresponds to the width of the interval $\mathcal{D}_i$. The quantization introduces an error, called distortion, defined by : $\Delta = Q_b^t(w) - w$.

5.1.3 Uniform quantization

The uniform quantizer is defined by a constant step $q = q_1 = \dots = q_L$ and quantization levels a_i, which represent the center points of the quantized intervals. To quantize the parameters $\boldsymbol{\theta}_k$ of the kth layer with R bits, we compute the quantization step q as follows:

$$q = \frac{\max(\boldsymbol{\theta}_k) - \min(\boldsymbol{\theta}_k)}{2^R}. \tag{5.3}$$

Then, for each value $w \in \boldsymbol{\theta}_k$, we compute the index i_w:

$$i_w = \left\lfloor \frac{w - \min(\boldsymbol{\theta}_k)}{q} \right\rfloor, \quad (5.4)$$

where $\lfloor \cdot \rfloor$ denotes the floor function.
If we know i_w, we can easily deduce the quantized value $\hat{w}$:

$$\hat{w} = Q_R^U(w) = q_i\, i_w + \frac{q}{2} + \min(\boldsymbol{\theta}_k),\ \forall w \in \boldsymbol{\theta}_k. \quad (5.5)$$

A visual representation of how uniform quantization works is given in 5.2.

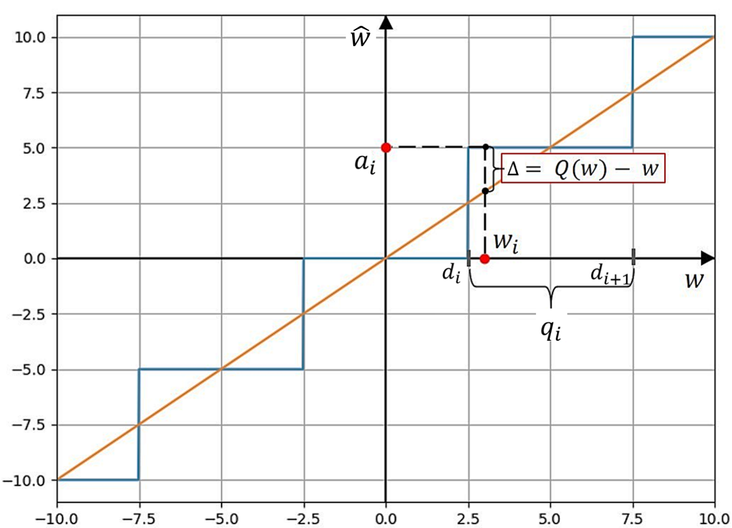


Figure 5.2: Representation of uniform quantization.

The advantage of this method is that it requires only one multiplication, therefore, it is more suitable for computer hardware that cannot access parallel mapping tables. Uniform quantization is optimal when R is large and if variable-rate encoding is used.

5.1.4 Non-uniform Quantization

Even though, uniform quantization is simpler, it can be non-optimal. A non-uniform quantization algorithm can be tailored to the specific distribution of the input data.

The non-uniform algorithm is an iterative algorithm that aims at building a quantization dictionary, or codebook, in order to minimize the mean squared error between the original and compressed data. Non-uniform quantization, denoted with Q_R^{NU}, is defined by intervals $\mathcal{D}_i$ of varying size and the centroids of these intervals which are the quantization levels a_i.

It is used on a training set $\mathcal{E} = \{w_1, \ldots, w_M\}$ composed of M values to quantize, where M is a rather large integer. The computation is performed in an iterative way, verifying successively two optimality conditions. If we know the quantization levels a_i, we apply the nearest neighbor condition to compute the best intervals $\mathcal{D}_i$ for the training set, as shown in (5.6). After choosing the intervals, we can compute the best quantization level a_i by computing the centroid of $\mathcal{D}_i$ (5.7).

Let us denote (m) and $(m+1)$ the iterations m and $m+1$, and $|\mathcal{D}_i^{(m+1)}|$ the cardinal of $D_i^{(m+1)}$ at iteration $m+1$. The number of partitions L is fixed. The algorithm is given by

$$d_i^{(m+1)} = \frac{a_{i+1}^{(m)} + a_i^{(m)}}{2}, \ \ \forall 1 \leq i \leq L, \tag{5.6}$$

$$a_i^{(m+1)} = \frac{1}{\left|\mathcal{D}_i^{(m+1)}\right|} \sum_{w_i \in \mathcal{D}_i^{(m+1)}} w_i, \ \ \forall 1 \leq i \leq L. \tag{5.7}$$

The standard algorithm used for non-uniform quantization was first described by Lloyd, and it is called the Lloyd algorithm [Lloyd, 1982]. Figure 5.3 schematizes the way the algorithm works.

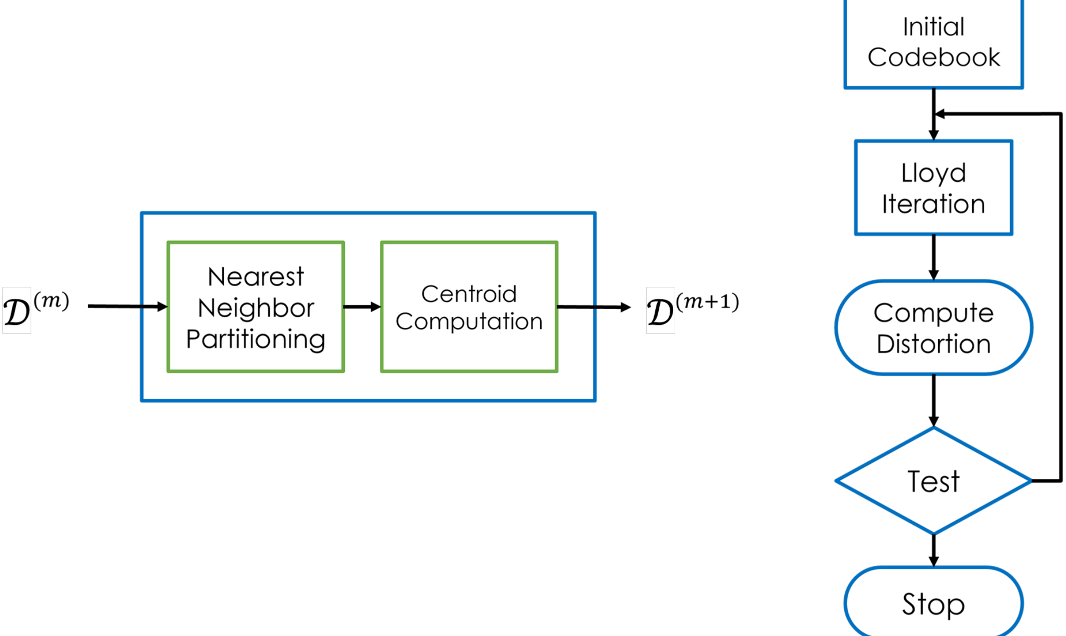


Figure 5.3: Lloyd's algorithm (left) and Iteration (right) [Gersho and Gray, 1991].

As mentioned before, the interest of such a quantization is that it adapts to the distribution of the data. However, the algorithm spends a lot of time calculating the distances between the centroids and the other points.

A visual representation of non-uniform quantization side by side to uniform quantization is given in Figure 5.4. The two methods are applied to the weights of an FC layer. We display the histogram of the weights which has a Gaussian shape, how the weights are partitioned and where the quantization levels are placed. Non-uniform quantization outputs quantization levels closer to the peak of the Gaussian curve because the density of weights is higher in that region.

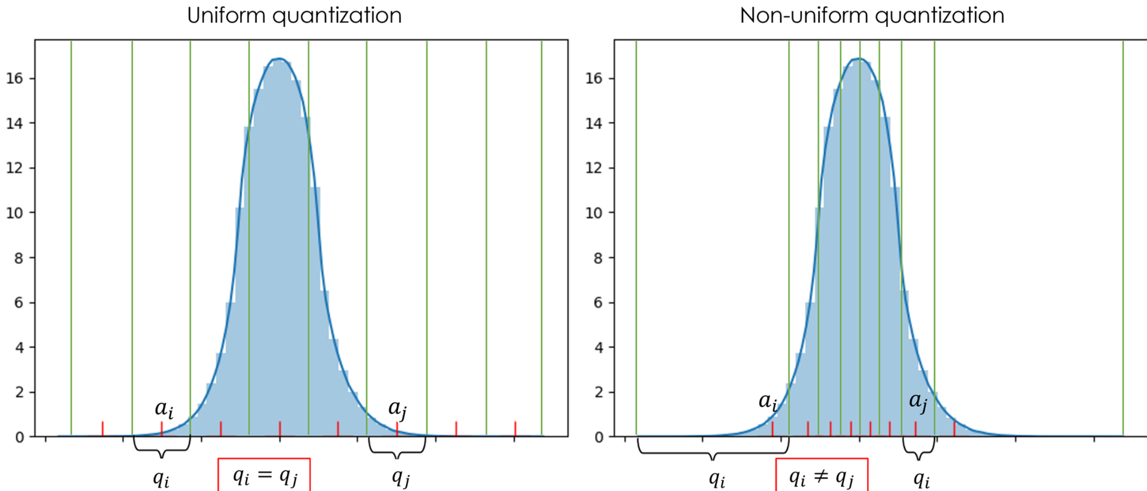


Figure 5.4: Uniform quantization vs non-uniform quantization applied to the weights of an FC layer.

5.2 Rate distortion theory

5.2.1 Background on rate-distortion theory

Considering the weights as the source of the compression algorithm. Let $W_1, W_2, \dots, W_n$ i.i.d. $\sim\ p(w),\ w \in \mathcal{W}$. We recall the compressed value of w is represented by $\hat{w} \in \hat{\mathcal{W}}$.

Definition 5.2.1. A distortion measure is a mapping

$$d : \mathcal{W} \times \hat{\mathcal{W}} \to \mathcal{R}^{+} \tag{5.8}$$

that measures the non-negative "cost" $d(w, \hat{w})$ of representing the weight w by $\hat{w}$.

The most popular distortion measure on a symbol-to-symbol basis is the squared error distortion:

$$d(w, \hat{w}) = (w - \hat{w})^2. \tag{5.9}$$

The extension to sequences is called the **Mean Squared Error** (MSE). Given the weights for the k-th layer of a network denoted with $\mathbf{W}_k$ and their compressed version $\hat{\mathbf{W}}_k$, the MSE is defined as:

$$d_{\text{MSE}}(\mathbf{W}_k, \hat{\mathbf{W}}_k) = \frac{1}{|\mathbf{W}_k|} \sum_{w \in \mathbf{W}_k} (w - \hat{w})^2, \tag{5.10}$$

where $|\mathbf{W}_k|$ is the total number of weights.

In data compression, we generally speak of encoding and decoding functions, but this is not the case for DNN compression. In Deep Learning compression, we are not interested in decoding, or

optimally reconstructing the weights' values. The goal is to use the compressed weights without changing the accuracy of the model. In this case, the distortion measure is defined on the output of the network. The distance between the output of the initial network $\mathbf{y}$ and the output of the compressed network $\hat{\mathbf{y}}$ can be measured by using **Kullback-Leibler divergence** (KL) [Kullback and Leibler, 1951]:

$$d_{\mathrm{KL}}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{c=1}^{C} y_c \log\left(\frac{y_c}{\hat{y}_c}\right). \tag{5.11}$$

The **rate-distortion function** describes the minimum transmission bit-rate R for a given distortion D and a source $\mathbf{W}$. We recall the theorem given by [Cover and Thomas, 2006, p. 301]:

Theorem 5.2.1. *The rate distortion function for an i.i.d random variable W with distortion measure $d(w, \hat{w})$ is defined as*

$$R(D) = R^{(I)}(D) = \min_{p(\hat{w}|w):\mathbb{E}[d(W,\hat{W})]\leq D} I(W, \hat{W}), \tag{5.12}$$

where $p(\hat{w}|w)$ is the conditional distribution for which the joint distribution $p(w, \hat{w}) = p(w)p(\hat{w}|w)$ satisfies the expected distortion constraint and $I(W, \hat{W})$ is the mutual information between the original source and the compressed one.

Calculating this optimum bit-rate is not relevant for us because we are not currently interested in an optimal quantizer that might be too complex to implement with a neural network. In [Nokleby et al., 2016], the rate-distortion theory is used to analyze the approximation of the posterior function involved in the Bayes classifier, not the accuracy loss. This theory is also used in [Gao et al., 2019] to analyze the Kullback-Leibler (KL) divergence between the classifier outputs before and after the compression.

5.2.2 Rate distortion trade-off

In this chapter, we propose a simple compression strategy for the entire neural network. More precisely, we aim at approximating the function $\mathbf{f}_{\boldsymbol{\theta}}$ by a function $\mathbf{f}_{\hat{\boldsymbol{\theta}}}$, where $\hat{\boldsymbol{\theta}} = Q_R^t(\boldsymbol{\theta})$ is a compressed form of $\boldsymbol{\theta}$. We propose to use a different quantizer per layer. We denote $Q_{R_k}^t$ the quantizer applied to the parameters of the $\mathbf{f}_k$ layer where $1 \leq k \leq K+1$. The quantized layer $\hat{\mathbf{f}}_k$ is written as:

$$\hat{\mathbf{f}}_k(\mathbf{x}) = \sigma_k(Q_{R_k}^t(\mathbf{W}_k)\mathbf{f}_{k-1}(\mathbf{x}) + Q_{R_k}^t(\mathbf{b}_k)). \tag{5.13}$$

We are searching for a sequence $(Q_{R_1}^t, ..., Q_{R_{L+1}}^t)$ to quantize the entire network, in order to find a trade-off between the memory size occupied by the network weights and the quality of the classification. The accuracy of the network (2.15) should not change (less than 1% loss). We are interested in two complementary distortion measures: the Mean Squared Error (5.10) and the Kullback-Leibler divergence (KL) (5.11).

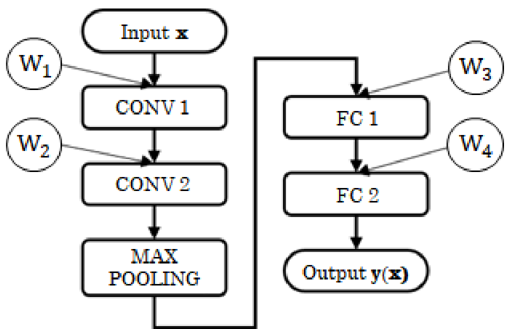


Figure 5.5: Architecture of the CNN.

5.3 Experiments

5.3.1 CNN on MNIST

For our first experiments, we first use a CNN network trained on the dataset MNIST. A description of the MNIST dataset is given in the subsection 2.2.1. Figure 5.5 shows the network's architecture and Table 5.1 summarizes the information about the layers. This network has a similar architecture to LeNet-5 [LeCun et al., 1998], but with fewer layers, only two convolutional layers and one pooling layer. The size of the inputs and outputs, the number of parameters of each layer and the memory size used are displayed. The pooling layer does not appear in the table because it cannot be quantized. The variables $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$, $\boldsymbol{\theta}_3$, $\boldsymbol{\theta}_4$ represent the parameters and group the weights $\mathbf{W}_k$ and the biases $\mathbf{b}_k$ of each layer as defined in (5.1). In practice, the values of $\boldsymbol{\theta}$ are represented in FP32 format.

Layers	Input	Output	Parameters	Size
CONV 1	28x28x1	26x26x32	320	1.28 KB
CONV 2	26x26x32	24x24x64	18,496	73.9 KB
FC 1	9,216	128	1,179,776	4.71 MB
FC 2	128	10	1,290	5.16 KB
Total	-	-	1,199,882	4.79 MB

Table 5.1: Parameters of the CNN trained on MNIST.

For each experiment, the compression ratio τ is calculated

$$\tau = \frac{\text{Original size}}{\text{Size after compression}}. \tag{5.14}$$

The accuracy (2.15) is computed on the test dataset, the MSE (5.10) is computed on the parameters and the KL divergence (5.11) is computed on the output of the network. The accuracy of the non-compressed model for the test dataset is 99.16%.

Layer by layer quantization. Quantization is applied to a single isolated layer, and the other layers remain configured with the initial uncompressed parameters. We are interested in the results obtained for quantization with 1 to 6 bits, because from 7 to 32 bits the loss of accuracy is

negligible. Figure 5.6 shows the obtained accuracy, Figure 5.7 the MSE and Figure 5.8 the KL divergence.

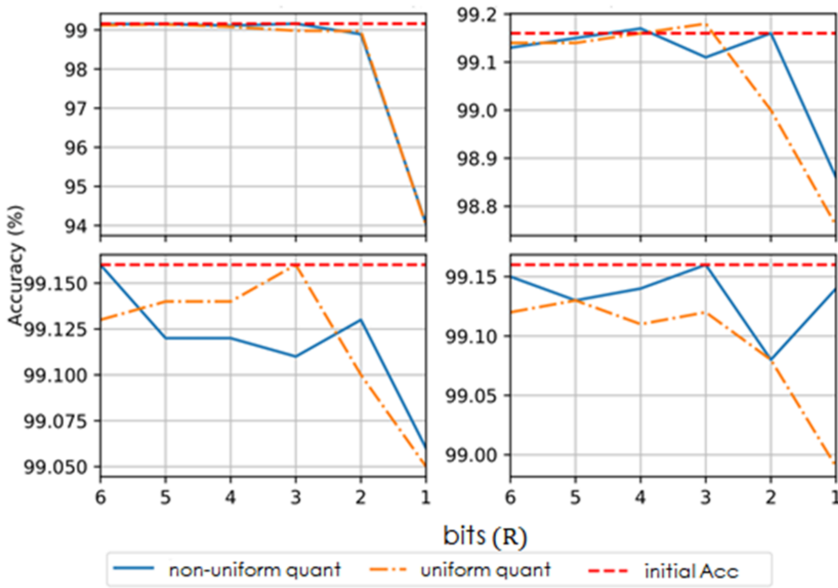


Figure 5.6: **Accuracy**: CONV 1 (top left), CONV 2 (top right), FC 1 (bottom left), FC 2 (bottom right).

At the top left of each figure, the quantization results for CONV 1 are displayed. For quantization between 4 and 6 bits, the loss in performance is very small. Starting from 3 bits, the accuracy of the network decreases slightly by 0.18% for Q^U. With 2 bits, the accuracy drops for both methods by about 0.70%. For a 1 bit quantization, the network loses 5% of accuracy. The KL divergence shows a small loss of information for 2 to 6 bits. With 1 bit, the information loss of Q^U is significant compared to Q^{NU}, but it does not impact the accuracy.

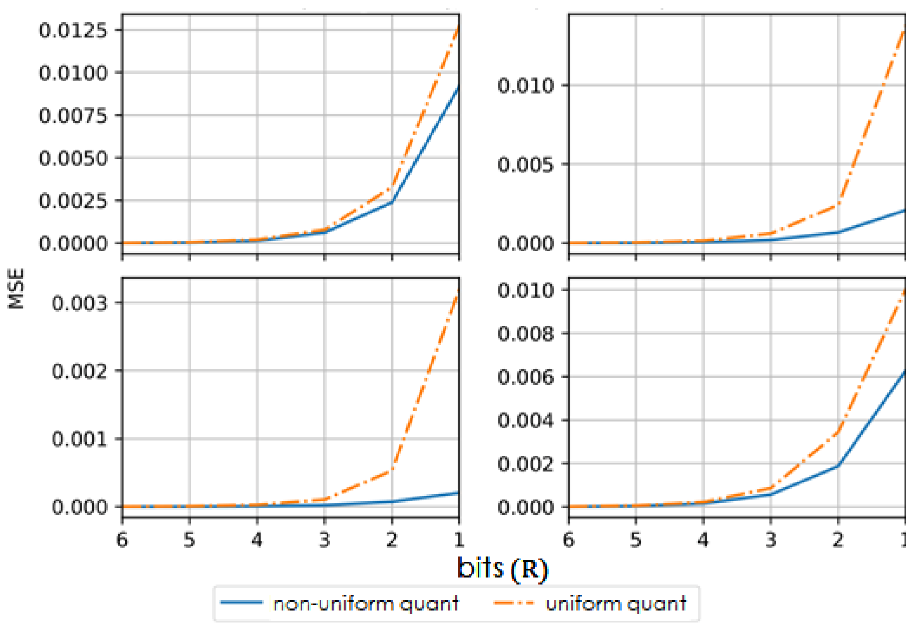


Figure 5.7: **MSE**: CONV 1 (top left), CONV 2 (top right), FC 1 (bottom left), FC 2 (bottom right).

The second convolution layer is shown at the top right of each figure. We notice that it is less sensitive to quantization than the first layer. The drop in accuracy is insignificant. We lose less than 1% in both cases of compression. The divergence shows a greater loss of information for Q^{NU}, while the distortion indicates a greater error for Q^{U}.

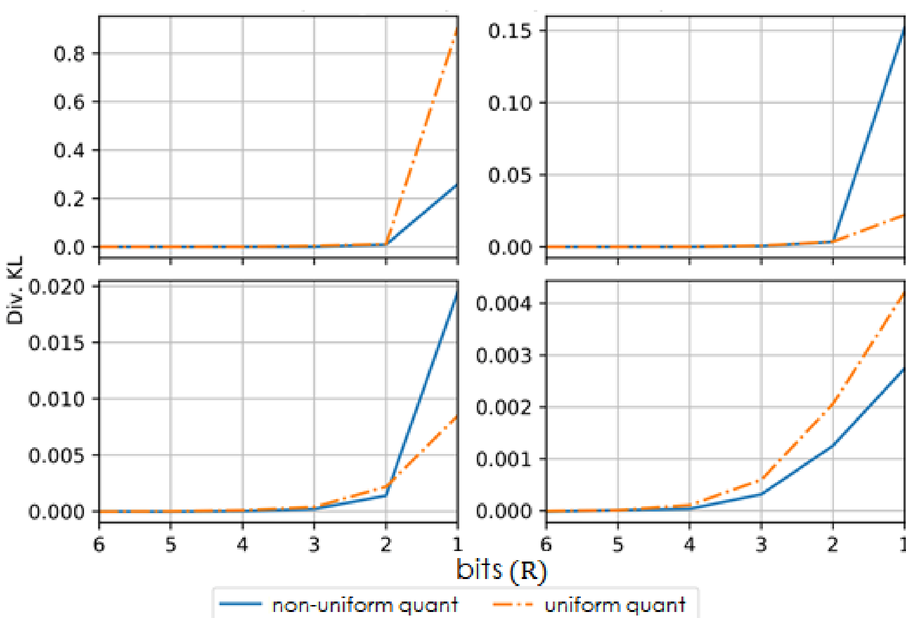


Figure 5.8: **Divergence KL**: CONV 1 (top left), CONV 2 (top right), FC 1 (bottom left), FC 2 (bottom right).

In the case of FC layers, the loss is even smaller. FC 1, the largest layer in the network with 1×10^6 parameters, is shown at the bottom left in the figures. With 1 bit, the accuracy drops only by 0.10%-0.11%. The divergence is up to 40x smaller and the distortion reduced by 4x. The last layer, FC 2, placed at the bottom right, has fewer parameters. Again, we see that the loss is negligible. In the case of Q^{NU}, we lose 0.02% accuracy and in the case of Q^{U} we get almost the same accuracy as for FC 1. The distortion and divergence also indicate a very small error and information loss.

There are cases where quantization improves network accuracy. This may indicate a lack of training. During training, when we minimize the loss function, we may fall into a local minimum. It is assumed that quantization plays a role here and that by adding errors, one effectively arrives at a smaller value of the cost function.

We find that non-uniform quantization performs better because it fits the data distribution better overall. However, there are some cases where uniform quantization performs better. Overall, the KL divergence and MSE are greater for the first layer than for the others. The role of the first layer is to build a good base of descriptors for the whole network. It is therefore more sensitive to quantization.

Table 5.2 contains compression statistics for each layer: the method used, the accuracy after compression, the number of quantization bits, the size of the layer after compression, the compression ratio τ_i of the layer and the compression ratio τ of the whole network. For each layer and for each compression method, we indicate only one result, the one that seems to have the best compression ratio for an almost unchanged level of accuracy.

Layer	Method	ACC	Size	τ_i	τ
CONV 1	Q_3^U	98.89 %	132B	9.69	1.0001
	Q_3^{NU}	99.11 %	150B	8.53	1.0001
CONV 2	Q_3^U	99.00 %	6.94KB	10.64	1.014
	Q_2^{NU}	99.16 %	4.64KB	15.92	1.014
FC 1	Q_1^U	99.05 %	147.48KB	31.93	21.025
	Q_1^{NU}	99.06 %	147.47KB	31.93	21.026
FC 2	Q_1^U	98.99 %	173B	29.82	1.001
	Q_1^{NU}	99.14 %	169B	30.53	1.001

Table 5.2: Statistics of compression for each layer.

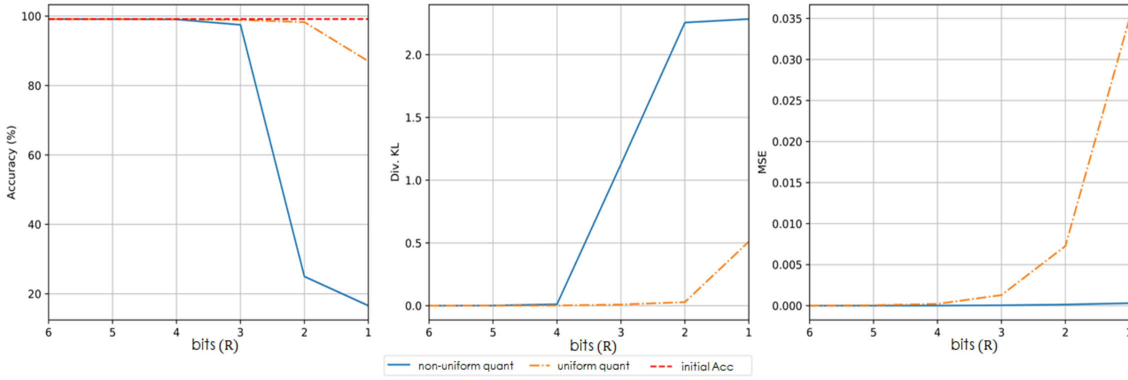


Figure 5.9: Accuracy, Div. KL and MSE for one quantizer for the entire network.

Quantizing the whole network. The second evaluation was done using the same quantizer Q_R^t for the whole network. In Figure 5.9, we see that the network reacts differently for the two quantization methods. Interestingly, in the case of Q^{NU}, the accuracy of the network goes down to 17%, while Q^U loses at most 13% of accuracy. Even if the MSE distortion is obviously much greater for Q^U, the loss of information is significant for Q^{NU} at the output of the network. KL Divergence has a similar shape to the accuracy. However, we notice a significant increase in the KL value for Q_3^{NU} which is not equivalent to the real decrease in accuracy noticed in the accuracy plot.

Adaptive quantization of the entire network. The last evaluation is done on the whole network by applying a different quantizer (same type but different number of bits) for each layer. In the previous experiments, we observed that for 5 or 6 bits, the error is negligible. For this evaluation summarized in Table 3, we have therefore applied a quantization between 1 and 4 bits for each layer. We tested all possible combinations and extracted 2 cases for each method: the first case gives the best accuracy and the second case presents a good trade-off between accuracy and compression rate. The results obtained are displayed in Table 5.3. Uniform quantization is still quite efficient compared to non-uniform quantization.

Method	R_1 CONV 1	R_2 CONV 2	R_3 FC 1	R_4 FC 2	ACC	τ
$Q^U_{R_k}$	4	3	3	4	99.18%	10.66
	3	2	2	2	98.80%	16
$Q^{NU}_{R_k}$	4	4	4	3	99.16%	8
	4	2	2	2	98.33%	15.98

Table 5.3: Compression statistics for the network, with quantizers adapted to each layer.

5.3.2 VGG on CIFAR-100

We validate our results on a larger scale network, VGG15 on CIFAR-100 [Krizhevsky et al., 2009]. The architecture of VGG is similar to the architecture previously presented, but with 13 CONV layers and 2 FC layers. The CONV layers are grouped into blocks of 2 or 3 layers followed by a Max Pooling layer. The schema is given in Figure 5.10.

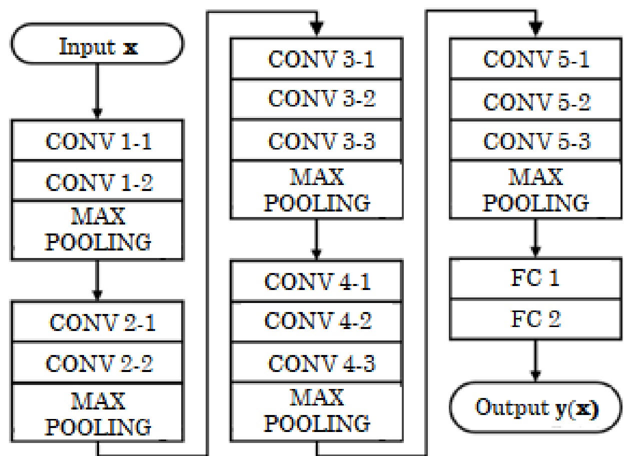


Figure 5.10: Architecture of VGG.

We compressed each of the layers of the network using a quantizer Q^t_R, with R between 1 and 8. Figures 5.11 and 5.12 show the results obtained for the first and last layer. We put side by side the accuracy, the KL divergence and the MSE. For the first layer, we notice that the quantization methods have different behaviors. While both quantization methods at 1 or 2 bits give the same poor accuracy, the uniform quantization is more stable than the non-uniform quantization for the other rates. When looking at the divergence or the MSE, the unstable phenomenon cannot be seen. Going on to the last layer, the accuracy drop is bigger for the non-uniform quantization at almost every bit rate. Again, this behavior cannot be noticed when looking at the divergence or the MSE. In the case of uniform quantization, the KL divergence has a steep growth from 4 bits to 1.

Compressing the first layer has a big impact on the way the network predicts, moving through the network, we observe the impact is reduced, which makes us believe that the feature representation given by the first layers are more important, and as the networks get deeper, they become more robust.

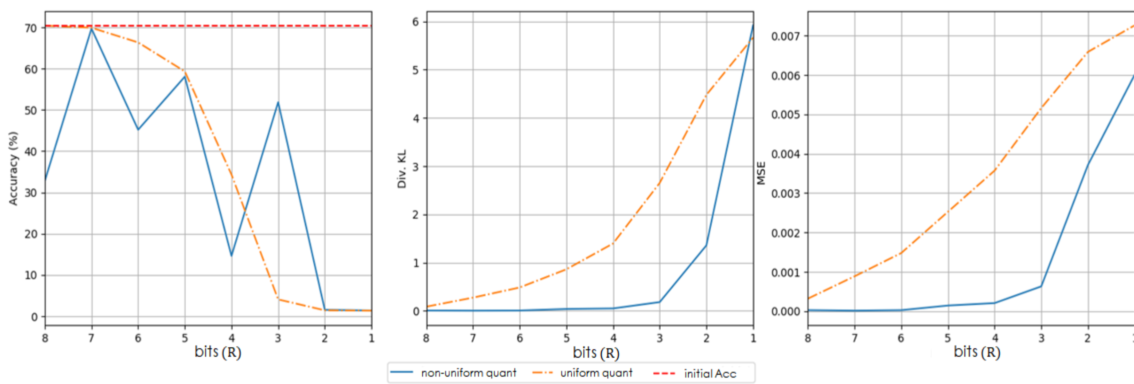


Figure 5.11: Accuracy, Div. KL and MSE for the first layer of VGG15

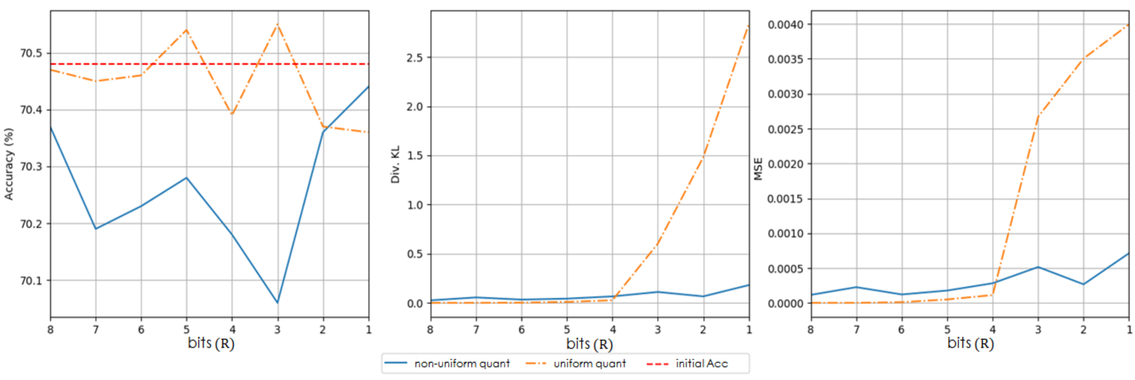


Figure 5.12: Accuracy, Div. KL and MSE for the last layer of VGG15

Given that for a small network, compressing with the same quantizer cannot give good results, we move on to an adaptive approach. We decided not to do all the combinations of quantizers since this would be too time-consuming. We chose a rate R between 1 and 8 and defined two strategies. First, we took the **Best Acc/layer** which means that we chose the smallest bit rate that gives us the best accuracy. The second strategy we call **Rate-Acc trade-off** and, in this case, we chose the smallest bit rate which allows an accuracy drop of less than 1% per layer which leads to 4-5% drop in accuracy. The results are shown in Table 5.4. Overall, non-uniform quantization gives a better compression rate in both strategies. However, in the case of the first strategy, uniform quantization has the smallest accuracy drop (less than 1%).

Method	Strategy	Acc	Div. KL	MSE	τ
Original	-	70.40%	-	-	-
$Q^{U}_{R_k}$	Best Acc/layer	69.88%	2.86	0.005	5.54
	Rate-Acc trade-off	66.43%	6.59	0.007	7.83
$Q^{NU}_{R_k}$	Best Acc/layer	67.18%	0.53	0.001	7
	Rate-Acc trade-off	65.18%	0.82	0.002	13.45

Table 5.4: Compression statistics for VGG15.

5.4 Conclusions and Perspectives

The role of this chapter is fundamental for both theoretical research and development aspects. We have compared two scalar quantization methods, uniform and non-uniform, with a purpose to reduce memory storage of parameters. This comparison provides us with some interesting general observations. First, networks are robust to quantization even for really low bit rates. Our experiments show that quantization works better if it is layer adapted. Uniform quantization is as efficient as non-uniform quantization and easy to implement. The first layers are very sensitive to quantization error, while the last layers of the network result in insignificant losses.

An important observation would be that the KL divergence and MSE are not good enough distortion measures to predict performance degradation in the case of inference with compressed neural networks. A different distortion measure should then be used, which focuses explicitly only on the decision made by the network. We will exploit this idea in Chapter 7 where we introduce a new distortion measure based on the Bayes risk and will also focus on the theoretical analysis of the observed phenomena.

From an industrial perspective, we can conclude that uniform quantization is a good fit for deep neural networks. For future developments, we plan on implementing uniform quantization techniques on parameters in order to map FP32 numbers to INT8. Moreover, Kalray's industrial initiative focuses on finding innovative techniques which can leverage the capabilities of the manycore processor for Deep Learning applications. To respond to the industrial stakes, alternative techniques of low precision quantization exist and will be studied in the next chapter.

CHAPTER 6

IEEE 754 and alternative formats for storage purposes

In this chapter, we experiment with reduced floating-point representations by considering alternatives to the standard IEEE binary32 floating-point format: Bfloat16, Posit8 and MSFP8. Our experiments show that the Bfloat16 format gives better accuracy than the classic FP16, and that two Posit8 formats can be used without significantly modifying the accuracy of conventional convolutional neural networks. Finally, our results indicate that MSFP8 is not a suitable format for the neural network types we have considered.

6.1 Industrial stakes **71**
6.2 Floating-point formats in Deep Learning **73**
6.2.1 IEEE 754 floating-point formats 73
6.2.2 Brain floating-point format 73
6.2.3 Microsoft floating-point 8 74
6.2.4 Posit 74
6.2.5 Comparison between data formats 75
6.3 Parameter Compression **77**
6.4 Experimental Results **78**
6.4.1 Experiment 1 78
6.4.2 Experiment 2 79
6.4.3 Experiments 3 and 4 79
6.5 Conclusion and Perspectives **80**

Deep learning networks normally rely on standard binary32 IEEE 754 floating-point (FP32) arithmetic for both training and inference. Reducing the footprint of neural networks by using low precision quantization on parameters has been highly studied and deployed [Berthelier et al., 2021]. We recall from Chapter 3 that many well-known frameworks [Krishnamoorthi, 2018, Borisyuk et al., 2018, Nvidia, 2018] support the rounding of FP32 parameters to lower precision formats, such as half precision standard binary16 IEEE 754 floating-point (FP16) [Micikevicius et al., 2018]. Some works even indicate that FP16 as a replacement for FP32 is not a good fit for Deep Learning applications [Ho and Wong, 2017]. Further precision reduction is supported, by using uniform quantization to map the FP32 numbers to integers on 8 bits (INT8) [Jacob et al., 2018, Nvidia, 2017].

6.1 Industrial stakes

We recall from the introduction that Kalray provides a tool called KaNN which is used to execute neural networks on Kalray's processor. Figure 6.1 showcases the processing steps of KaNN. Compression is applied to the pre-trained weights in the pre-processing step. During the runtime, the compressed weights are transferred to the MPPA and on to the clusters. Finally, the weights would be decompressed just before computations. The motivation for this work comes from the need to find new and innovative ways to compress the pre-trained weights in the pre-processing step and reduce the number of bits. By reducing the number of bits, not only the storage requirements will be reduced, but also the transfer time will be lowered.

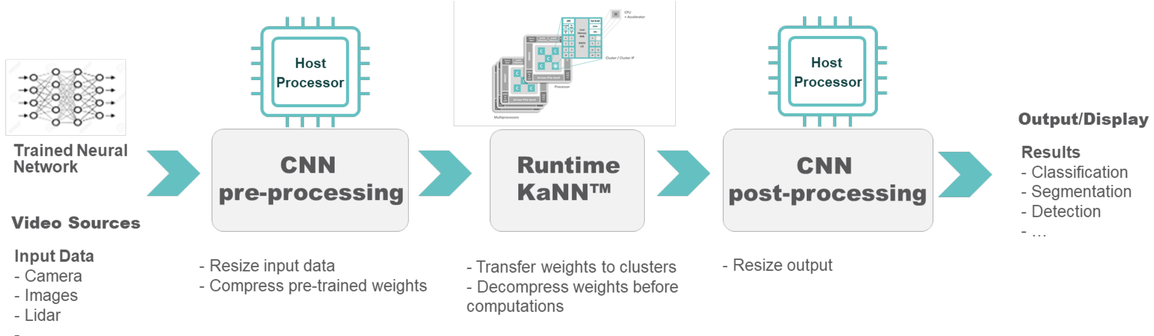


Figure 6.1: KaNN processing steps. A trained neural network and the input source are being pre-processed. The pre-processing steps include compressing the pre-trained weights and resizing the inputs. During KaNN's runtime, the weights are transferred to the clusters on the MPPA and then they are decompressed just before the computation step. When the runtime is finalized, the output goes to the post-processing step, where it is resized if needed.

In this chapter, we follow alternative approaches to FP32 parameter compression by rounding them to other formats. From Chapter 5, it is clear that in Deep Learning networks, each layer has its own precision and dynamic range requirements. Therefore, we are wondering if there are other formats which can fit all the layers' needs. Custom formats from mainstream AI practitioners, such as Brain Floating Point Format [Dean et al., 2012] and Microsoft Floating Point [Chung

et al., 2018] and others [Köster et al., 2017, Paresh Kharya, 2020] aim to replace FP32 for both training and inference.

Another alternative floating-point format is called Posit [Gustafson and Yonemoto, 2017] which belongs to the family of Universal numbers (Unum) [Tichy, 2016]. Posits are considered a hardware-friendly version of Unum. We have identified Posits as a good candidate for Deep Learning applications because of their flexibility in choosing the dynamic range and precision of the numbers. Despite using a reduced number of bits, Posits are configurable and can fit different precision or dynamic range requirements. Posit arithmetic is already used for inference in [Murillo et al., 2020b, Lu et al., 2020]. However, as previously mentioned, we are not interested in computing directly using the Posit format. Furthermore, some Posit operators have already been proposed in [Jaiswal and So, 2019, Xiao et al., 2020, Murillo et al., 2020a]. These operators include a decompressing component which transforms a number from a Posit format into a representation similar to a floating-point number of non-standard size. We take inspiration from these works.

Figure 6.2: Runtime steps. The weights are compressed during pre-processing. At runtime, the compressed weights are transferred from the external memory of the platform to internal memory of each cluster. After the transfer, they are decompressed and the computation is performed. Finally, the output given by the computation is transferred to the next cluster

We intend to implement a decompressor from Posit using 8 bits (Posit8) to FP16 in order to leverage the capabilities of existing operators on the manycore processor that efficiently performs FP16 matrix multiply-accumulate operations for Deep Learning inference. As shown in Figure 6.2, this decompression step would be added during KaNN's runtime. Extending the Kalray MPPA processing element with instructions that decompress Posit8 numbers to FP16 numbers enables to reduce further the footprint of the neural network parameters with an acceptable loss of accuracy or precision.

The lack of results on the performance of each format motivates us to investigate some of these alternative floating-point formats for storing the network parameters, while still doing the operations using FP32 arithmetic. We put to the test three alternative formats: BF16, MSFP8 and Posit8 representations, comparing their performances to FP32 and FP16 on classification and detection neural networks. The goals are to highlight which arithmetic representations are the most suited to these types of networks, and to find if one can reasonably store neural network weights in only 8 bits without retraining the network.

The chapter is organized as follows. In Section 6.2, we present all the data formats including the standard floating-point. For each format we give the state-of-the-art and a formal definition. Section 6.3 gives details on the method of compression. We benchmark the correctness of results

for different types of deep neural networks in which such formats are used. Section 6.4 describes the experiments and the results, while conclusions are presented in Section 6.5.

6.2 Floating-point formats in Deep Learning

6.2.1 IEEE 754 floating-point formats

For a long time, Deep Learning networks computations have used the standard FP32 arithmetic. Both weights and activations are represented in FP32 by default. A recent trend is to use the binary16 floating-point standard format (FP16) instead of FP32, since FP16 can be used for faster inference and training [TensorFlow Lite, 2021], leading to significant savings in memory footprint and to an increase in performance/efficiency even during training [Micikevicius et al., 2018].

A floating-point number can be expressed by using a triplet (s, m, e) so that:

$$x = (-1)^s \cdot \beta^{e-bias} \cdot m, \tag{6.1}$$

where β is the radix of the floating-point system, $s \in \{0, 1\}$ is the sign used to differentiate negative from positive numbers, $m = m_0 m_1 ... m_{p-1}$ is the mantissa with a leading hidden bit set to 1, p is the precision and $e \in [e_{min}, e_{max}]$ is the exponent. The use of a $bias$ allows to obtain a negative exponent in order to represent bigger and smaller numbers.

The IEEE 754 standard describes binary formats ($\beta = 2$) and decimal formats ($\beta = 10$). Binary formats such as FP32 and FP16 are often used in neural networks. Let us denote the representation of x in a format F with x_F, so we can write x_{FP32} and x_{FP16} as follows:

$$x_{FP32} = (-1)^s \cdot 2^{e-127} \cdot m, \text{ with } p = 23, \tag{6.2}$$

$$x_{FP16} = (-1)^s \cdot 2^{e-15} \cdot m, \text{ with } p = 10. \tag{6.3}$$

A complete formal definition of standard floating-point is given in [Muller et al., 2018].

6.2.2 Brain floating-point format

An alternative 16-bit format to the standard FP32 and FP16 was developed by Google and is called Brain floating-point, or **Bfloat16** (BF16). It was first introduced in 2012 as part of a distributed training framework *DistBelief* [Dean et al., 2012] as a low-precision storage format used to reduce communication costs between nodes.

BF16 is similar to the standard IEEE 754. The BF16 format is a 16-bit truncated version of FP32 with the mantissa reduced to 7 bits [Intel, 2018]. A number represented with this data type can be written as:

$$x_{BF16} = (-1)^s \cdot 2^{e-127} \cdot m, \text{ with } p = 7. \tag{6.4}$$

This format is very attractive for Deep Learning training because it provides the same dynamic range as FP32 for half the bit-width, while the conversion from/to FP32 is straightforward. Mainstream processors now compute using this format [Lutz, 2019, Abadi et al., 2016] and a number of works related to it have been recently published [Kalamkar et al., 2019, Burgess et al., 2019]. One should note that the BF16 format is actually only used for multiplication operands, whose results are still accumulated in FP32.

6.2.3 Microsoft floating-point 8

Microsoft introduced a data format called **MSFP8** [Chung et al., 2018]. This alternative format, is also similar to the floating-point system. It is equivalent to FP16 truncated to 8 bits.
This data type has 1 bit sign, a 5-bit exponent and a 2-bit mantissa. It is represented as:

$$x_{MSFP8} = (-1)^s \cdot 2^{e-15} \cdot m, \text{ with } p = 2. \tag{6.5}$$

MSFP8 is the equivalent of BF16 for FP16 and has the same advantages as BF16 but with a smaller size and more limited precision.

Microsoft also proposes some variations of this format. In [Chung et al., 2018], a variation of this format, MSFP9, with 3 bits of mantissa has also been presented. Recurrent neural networks are specially targeted by these formats. Later on, they have made the format even more configurable, using up to 11 bits. Even though these highly configurable formats are said to achieve state-of-the-art performance, they are developed for FPGAs. In our case, configurable formats are not easy to develop and use on our processors. Therefore, we will only look at MSFP8.

6.2.4 Posit

Another family of reduced bit-width floating-point formats is obtained by choosing suitable parameters of the **Posit** representation introduced in [Gustafson and Yonemoto, 2017].

A Posit<n,es> representation is parametrized by n, the total number of bits, and es, the number of exponent bits. The main difference with an IEEE 754 binary floating-point representation is the *regime* field, which has a dynamic width and encodes a power of $2^{2^{es}}$ in unary numeral. De Dinechin *et al.* [de Dinechin et al., 2019] discuss the advantages and disadvantages of Posit representations. They advise using Posit as a storage-only format in order to benefit from the compact encoding, while still relying on standard IEEE binary floating-point arithmetic for numerical stability guarantees.

Recent works have used Posit representations to efficiently implement neural network inference in hardware [Cococcioni et al., 2020a]. *Deep Positron* [Carmichael et al., 2019] is a DNN architecture adapted for FPGA which uses an exact-multiply-and-accumulate (EMAC) algorithm for acceleration of ultra low-precision arithmetic ($\leq$ 8 bits). Their results show that Posit<8,1> outperforms classic fixed-point and floating-point formats. However, only results for small datasets have been presented. Another approach is to use Posit<8,1> on a log domain for the multiplicands, while converting to a linear domain for the accumulations [Johnson, 2018]. Again, the large dynamic range that motivates using the Posit representations in machine learning inference requires high-precision or exact accumulations.

Unlike IEEE numbers and the previous alternative formats, Posits have 4 elements: sign, regime, exponent and mantissa. A Posit<n, es> is defined only by n, the total number of bits and es, the maximum number of bits dedicated to the exponent. The components of a Posit have dynamic lengths and are determined according to their priorities. Firstly, bits are assigned to the sign and the regime. If some bits remain, they are assigned to the exponent and, lastly, to the mantissa. The regime is a run-length encode signed value and can be seen as a different type of exponent. Table

6.1 has been extracted from [Carmichael et al., 2019] and shows how the regime is interpreted.

Binary	0001	001	01	10	110	1110
Regime value	-3	-2	-1	0	1	2

Table 6.1: Regime interpretation.

The numerical value of a Posit number is given by (6.6) where k is the regime value, e is the exponent and m is the mantissa:

$$x_{Posit<n,es>} = (-1)^s \cdot \left(2^{2^{es}}\right)^k \cdot 2^e \cdot m. \tag{6.6}$$

The Posit standard defines 4 formats: Posit<8,0>, Posit<16,1>, Posit<32,2> and Posit<64,4>. In our work, we are only interested in the 8-bit Posit. Since Posit is highly configurable and new, we would like to test different exponent sizes. In addition to $es = 0$, exponent sizes (es) of 1, 2, 3 are reported useful to compress image classification and object detection network parameters [Carmichael et al., 2019, Cococcioni et al., 2020b].

6.2.5 Comparison between data formats

Figure 6.3 gives a visual representation of the formats previously presented.
In order to choose the best arithmetic representation for a number x, one needs to consider two aspects: dynamic range and precision. The dynamic range is the range of numbers that can be represented by a particular data type. As written in (6.7), it is given by the decimal logarithm of the ratio between the largest representable number to the smallest one:

$$DR = \log_{10}\left(\frac{2^{2^{es-1}}}{2^{2-2^{es-1}-p}}\right), \tag{6.7}$$

where p is the size of the mantissa.

Knowing the total size and exponent size means that we can determine the dynamic range of a given format. Table 6.2 summarizes the components of the formats while also presenting their dynamic range.

Format	FP32	FP16	BF16	MSFP8	Posit<8,0>	Posit<8,1>	Posit<8,2>	Posit<8,3>
Exponent	8	5	8	5	0	1	2	3
Mantissa	23	10	7	2	5	4	3	2
Regime	-	-	-	-	2-7	2-7	2-7	2-7
Dynamic range	83.38	12.04	78.57	9.63	3.61	7.22	14.45	28.89

Table 6.2: Comparison of components and dynamic ranges for data formats. Note that the components of Posits have dynamic length. The indicated values of exponent and mantissa for Posit represent the maximum number of bits the components can have. The regime has priority over the bits.

Figure 6.3: Visual comparison of formats. Standard formats FP32 and FP16 are shown on top. BF16 has the same size as FP16, but the same exponent size as FP32. MSFP8 has a 5-bit exponent size, the same as FP16, but only a 2-bit mantissa. Posits have variable sized fields. In this figure, we illustrate the composition of the Posit and the priority of each component: regime, exponent and the fraction (or mantissa).

The 32bit IEEE 754 standard representation has a very wide dynamic range, much larger than needed for DNNs. The alternative BF16 preserves almost the same dynamic range as FP32, while FP16 has a much smaller one. We note that the latter still has a dynamic range which generally is enough for most DNNs. Even if MSFP8 has almost the same dynamic range as FP16, it comes with a very small precision. Concerning the Posit, one of its advantages is the possibility of choosing es to adjust the trade-off between the dynamic range and precision as to best meet the needs of any given application. This stems from the fact that an increase of the es also implies, among others, a decrease in the number of bits available for the fractional part, which in turn decreases the precision. A visual representation of Posit<8,es>, with es from 0 to 3 is provided in Figure 6.4. The range gets bigger as es increases. It is noticeable that the format is highly dense around 0.

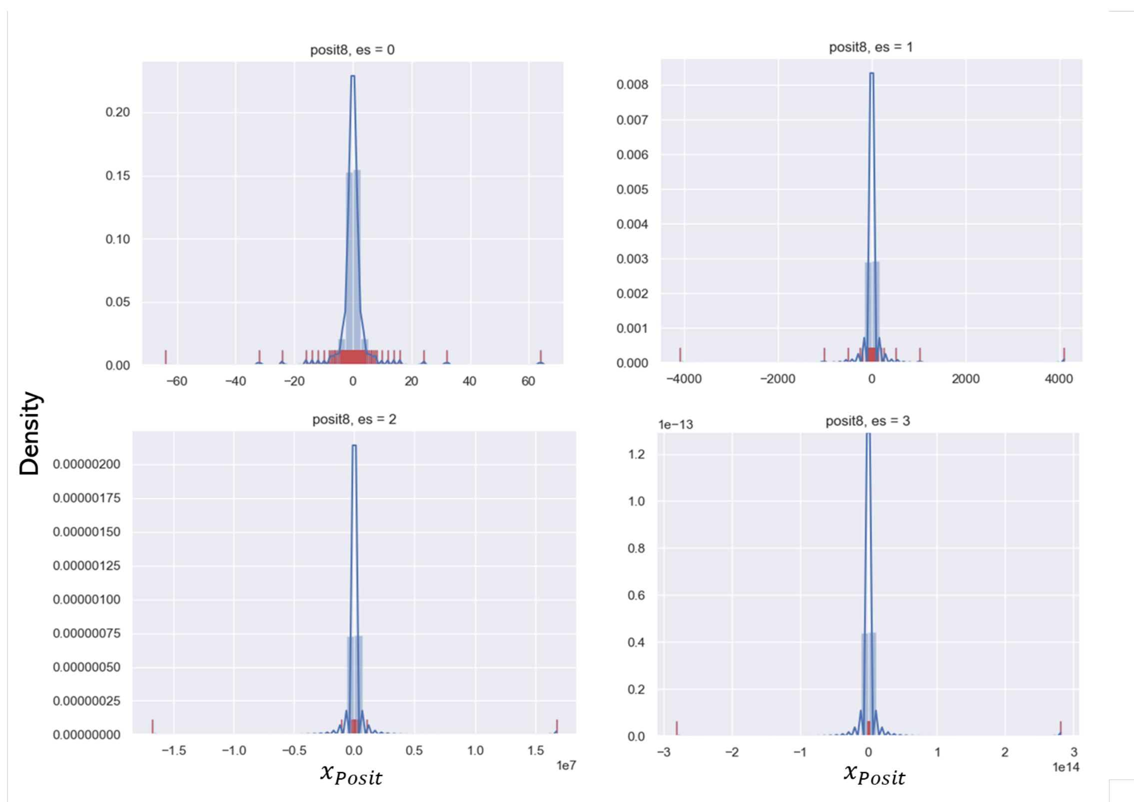


Figure 6.4: Histogram of Posit<8,es> values. Parameter es is between 0 and 3. The red pikes indicate the true representative values. The blue curve displays the density.

6.3 Parameter Compression

From the work presented in Chapter 3, we already know that deep neural networks have high storage requirements and these come primarily from convolutional and fully connected layers (i.e. VGG16 with 138M parameters which requires 552MB to store them in FP32).

However, for more recent networks, a small percentage of the parameters come from Batch Normalization (BN) layers. An example of network with BN layers is ResNet50 [He et al., 2016]. It has about 25M parameters from which only 106K come from BN layers. Yet, these weights can have a high impact on a model's performance because of their role in adjusting and scaling inputs. In Appendix B.1, we provide a table containing the number of weights, biases and the ranges of parameters for 7 well-known networks.

All the following experiments were carried out on 13 classification networks and 1 object detection network. Different evaluation criteria were studied: Accuracy Top 1 (ACC-1), Accuracy Top 5 (ACC-5) for classification and Mean Average Precision (mAP) for detection. Note that Acc-1 is the conventional accuracy, where the class with the highest probability is the model's answer and has to match the correct class. ACC-5 means that any the correct answer must be in the top

5 highest probability classes in order to be considered correct. We use pre-trained state-of-the-art neural networks. We compress their parameters in order to reduce the storage size of the networks. As mentioned in the previous section, alternative data types such as BF16, MSFP8 and Posit can have properties better suited for the storage of DNNs. We convert parameters from FP32 format to FP16 and to each of the alternative types, after which we analyze the impact on the results of different classification and detection networks.

We decide to perform 4 test cases on the validation dataset of ImageNet, to see if the type of compressed parameters have a big impact on the performance. In the first one, we compress all the parameters without taking into consideration their type. For our second experiment, we do not compress the parameters that come from the BN layers. In our third experiment, we do not compress the biases and for the last experiment, we exclude both the biases and the BN parameters.

In all our experiments, operations are done in FP32. We simulate low precision storage by replacing the parameters with the values given by the alternative formats. Conversion from FP32 to BF16 is done by using a FP32 with the last 16 bits frozen at 0. Similarly, for the MSFP8 we use a FP16 where the last 8 bits are fixed at 0. Regarding Posit, we notice that small length Posits can represent numbers with high precision and a wide dynamic range. Thus, in our comparisons, we chose to evaluate 8-bit Posits with an exponent es which varies between 0 and 3. Note that values in Posit <8,0> and Posit <8,1> can be represented exactly in FP16. A Posit <8,2> has 8 values of greater magnitude which are not representable in FP16, but can be represented well by a BF16. For Posit <8,3>, 46 values are not representable in FP16 and 12 values are not representable in BF16. A dictionary containing the 255 values given by each Posit type is obtained by relying on different implementations [Gustafson and Yonemoto, 2017] [Posithub Survey, 2019]. In this case, the conversion is done by replacing the parameters with the closest values from the dictionary.

The next section presents our results and observations. The benchmark tables are added in the Appendix B. To simplify the presentation, we include smaller tables, containing only 5 classification models: VGG16 [Simonyan and Zisserman, 2015], ResNet50 [He et al., 2016], InceptionV3 [Szegedy et al., 2016], Xception [Chollet, 2017], MobileNetV2 [Howard et al., 2018].

6.4 Experimental Results

6.4.1 Experiment 1

Table 6.3 contains the results obtained for 5 classification networks which have different architectures. We also display the results obtained with FP32 and FP16 to be able to compare with the standard floating-point representations. The mAP results for the object detection network (YOLOV3 [Redmon and Farhadi, 2018]) are shown in Table 6.4.

DNN	Criterion	FP32	FP16	BF16	MSFP8	Posit			
						<8,0>	<8,1>	<8,2>	<8,3>
VGG16	ACC-1	70.6	70.6	70.8	69.7	10.2	70.8	70.5	70
	ACC-5	91.3	91.3	91.2	90.3	25.2	91.0	91.0	90
ResNet50	ACC-1	75.7	71.3	75.5	62.8	0.0	27.7	73.2	66
	ACC-5	93.3	90.2	93.5	83.8	0.0	91.4	91.4	88.7
InceptionV3	ACC-1	71.1	71.1	71.3	44.8	65.1	69.4	69.7	63.1
	ACC-5	89.9	89.9	90.0	67.9	86.1	91.0	89.5	85.3
Xception	ACC-1	73.5	73.4	73.6	37.5	70.6	72.4	72.1	63.8
	ACC-5	92.1	92.2	91.7	60.6	90.9	91.4	90.9	86.0
MobileNetV2	ACC-1	71.2	71.2	71	0.2	12.7	12.3	11.0	3.2
	ACC-5	90.0	90.0	89.6	0.6	24.7	25.7	24.4	9.9

Table 6.3: Results for classification networks. Conversion is applied to all parameters.

After this first experiment, we can conclude that the network's accuracy with BF16 compression always remains close to the accuracy of the original network. Furthermore, compression with BF16 is overall better than compression with FP16. On the other hand, MSFP8 is not suitable for the types of networks studied. Two bits of precision are evidently not enough. Regarding Posits, results are promising, but can still be improved. We also need to identify which is the best configuration for the Posit format. The next experiments will focus only on Posits. The complete Table can be found in Appendix B.2.

DNN	Criterion	FP32	FP16	BF16	MSFP8	Posit			
						<8,0>	<8,1>	<8,2>	<8,3>
YOLO	mAP	0.41595	0.41595	0.41585	0.3022	0.4025	0.4155	0.411	0.394

Table 6.4: Results for detection network. Conversion is applied to all parameters.

6.4.2 Experiment 2

The second experiment is carried out for the networks that have batch normalization layers. Here, we do not compress the parameters of batch normalization layers. The corresponding results are presented in Table 6.5 and the full table is given in the Appendix B.3. Compression without the BN parameters considerably reduces the loss in accuracy compared to full parameter compression. We observe that, to improve the performance of some networks, it is better to avoid compressing all the parameters that come from BatchNorm (cf ResNet, Inception, Xception).

6.4.3 Experiments 3 and 4

Many techniques [Szymon Migacz, 2017] avoid changing the bias when compressing. We wanted to see if this also had an impact on the performance of the networks. Our results show that the biases do not have a big impact on the classification of the tested networks, so not compressing the biases can improve the accuracy compared to the first and second experiments, but the loss is still too big ($> 1\%$). The tables are added in Appendix B.4 and B.5.

DNN	**Criterion**	**FP32**	**Posit**			
			<8,0>	**<8,1>**	**<8,2>**	**<8,3>**
ResNet50	ACC-1	75.7	71.3	75.0	75	73.6
	ACC-5	93.3	9.8	92.7	92.8	92.6
InceptionV3	ACC-1	71.1	66.0	70.9	70.1	69.9
	ACC-5	89.9	86.8	90.7	89.1	88.5
Xception	ACC-1	73.5	72.1	72.6	72.8	68.8
	ACC-5	92.1	91.3	91.7	91.3	89.4
MobileNetV2	ACC-1	70.8	25.3	53.5	52.7	39.4
	ACC-5	89.8	47.0	76.9	77.3	63.1

Table 6.5: Results for classification networks. Conversion is applied to parameters from convolutions and fully connected layers. Parameters from BN layers are kept in FP32.

We distinguish several representative architectures for classification networks. Traditional classification networks such as VGG have biases and do not contain a BatchNorm layer. The compression of this type of network is lossless (or with negligible loss) for Accuracy Top 1. ResNet50 has biases and BatchNorm layers. Following the tables added in the appendices, we can see that compression without bias does not reduce the accuracy loss. On the other hand, the compression without BatchNorm parameters has a loss of only 0.7% on the accuracy top 1. InceptionV3 and Xception contain very few biases (1000 in total). Therefore, the biases have little impact on the classification result. Compression without BatchNorm has reduced loss to 0.2% and 0.9% which is a big improvement. MobileNet networks also have few biases, as for InceptionV3 and Xception. The use of Posit8 is not suitable for MobileNet type networks. Without compressing the BatchNorm layer parameters, accuracy is improved, but the loss remains too high for real applications.

Observe that for networks containing normalization layers (ResNet50, InceptionV3, Xception and MobileNetV2), the loss of performance seems more pronounced when using 8-bit formats. Our experiments show that not compressing the parameters of the Batch Normalization layers improves the performance of all the networks. However, despite the improvement, MobileNetV2 remains with a significant accuracy loss.

Overall, compression with BF16 gives better results than with FP16. Despite its lower precision, BF16 seems to be sufficient for neural networks. On the other hand, the reduced precision of MSFP8 leads to a significant loss of performance for all tested networks. Posit <8,0> and Posit <8,3> formats do not give good results. For these formats, a non-negligible loss of performance is observed in both conventional classification (VGG16) and detection networks.

6.5 Conclusion and Perspectives

This chapter experiments with three floating-point representations of parameters for Deep Learning inference in classification and detection networks. These alternatives to the classic IEEE 754 binary floating-point standard save memory capacity and bandwidth as they fit into 16 bits and 8 bits. However, computations are still carried using standard FP32 arithmetic. BF16 generally has

a reduced impact on performance, if any, regardless of the considered network. MSFP8 lacks in precision and does not give acceptable results for the networks we have considered. The Posit8 representation with 1 or 2 exponent bits also tends to perform well with most neural networks, with slight exceptions. To obtain a satisfying compression of the parameters without accuracy loss, a good trade-off between the dynamic range and the precision is needed.

For future work, we are interested in simulating an end-to-end low-precision model and compare our results with state-of-the-art implementations for 8-bit quantization with fixed-point. Furthermore, it would be interesting to perform an adaptive method using Posit8 which enables the possibility of choosing the format with the right precision and dynamic range for a specific layer.

As mentioned in the beginning of the chapter, the results we have obtained using Posits also motivate considering the inclusion of a Posit8 to FP16 hardware decompressor in future Kalray MPPA processor, as the tensor coprocessors of the MPPA processor already include exact FP16 to FP32 dot product operators.

Figure 6.5: Runtime steps. The weights are compressed during pre-processing. At runtime, the compressed weights are transferred from the external memory of the platform to internal memory of each cluster. After the transfer, they are decompressed and the computation is performed. The output of the computation is compressed and ready to be transferred to the next cluster.

Secondly, we also consider implementing a FP16 to Posit8 compressor that would allow us to do benchmarks by compressing also the activations between the layers of the networks (see Figure 6.5). This would be useful to reduce the transfer costs and speed up the inference task.

CHAPTER 7

Effect of quantization error on Softmax Layer

This chapter is the main contribution of this thesis. It proposes a theoretical analysis of the quantization method applied post-training on the parameters of the softmax layer of a neural network. We start by presenting the simplified problem statement. We recall some preliminaries on neural networks, introduce the Bayes risk and our working assumptions. Next, we define a new distortion measure that yields insight into the connection between the accuracy loss (distortion) and the number of bits (rate) assuming uniform quantization. The outline of this chapter is the following. Section 7.1 introduces the state-of-the-art for sensitivity analysis. Section 7.2 presents some preliminaries on neural networks, the Bayes risk and our working assumptions. In Section 7.3, we introduce the distortion measure. We analyze and we propose some theoretical tractable approximations of this distortion in Section 7.4. Furthermore, in Section 7.5 we discuss the quality of our approximations with numerical simulations and test our distortion on several neural networks and datasets. Finally, Section 7.6 concludes the chapter.

7.1 Sensitivity analysis in neural networks **85**
7.2 Problem statement **86**
7.2.1 Deep neural networks 86
7.2.2 Minimum Bayes risk 87
7.3 Distortion measure for classifiers **88**
7.4 Distortion measure applied to uniform quantization **89**
7.4.1 Approximation of the distortion function 89
7.5 Experiments **91**
7.5.1 Numerical simulations 91
7.5.2 One-hidden-layer neural network on Sonar 93
7.5.2.1 Linear hidden layer 93
7.5.2.2 ReLU hidden layer 94
7.6 Conclusion and Perspectives **96**

Due to significant increase in deep neural networks complexity, numerous approaches with little to no loss in accuracy have been proposed. Many compression methods have shown promising results, but how can we choose the best method to reduce the complexity of a neural network without loss in performance? How do we know if a network has been compressed well? Classical distortion measures such as MSE and KL Divergence fail to measure the gap between the original accuracy and the accuracy of the compressed model. Sensitivity analysis can be the answer to these questions.

7.1 Sensitivity analysis in neural networks

Sensitivity analysis in neural networks refers to understanding how error can affect the network's decisions. This error comes from input or weight perturbations which can be caused by noisy inputs, hardware problems or compression. Sensitivity analysis can be used for different purposes: optimization, robustness, generalization, decision boundary visualization or compression. As seen above, with compression techniques, these errors can cause a drop in accuracy. By analyzing the error propagation, we will better understand how the errors impact the network. The general structure used in sensitivity analysis is given in Figure 7.1.

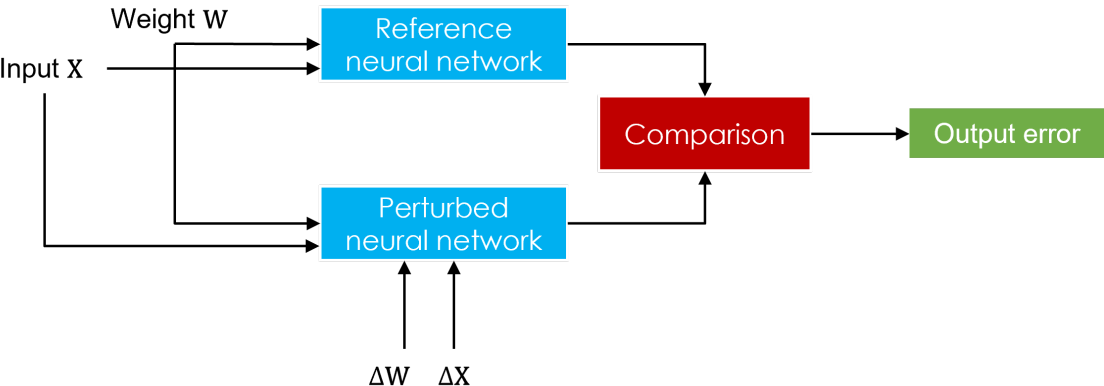


Figure 7.1: General structure of sensitivity analysis methods [Yeung et al., 2010].

In the 1990s, a number of studies emerged on sensitivity and perturbation analysis. The approaches presented in these studies can be classified according to the techniques used. There are papers that have chosen to start from a geometric technique to study the sensitivity. Other papers have used statistical methods, and another category identified is that of papers that use analytical approaches.

Piché [Piché, 1995] studies the effects of the error of the weights in an ensemble of Madaline networks. His method helps to identify important trends caused by weight perturbations rather than trends associated with a specific network. In his approach, Piché uses a statistical method to analyze the errors. He assumed that the inputs and weights are independent and centered and the errors are small. Piché defined the expression for sensitivity as the ratio of the variance of the output error to the variance of the output.

Xie *et al.* [Xie and Jabri, 1992] propose a statistical model to analyze the effects of weight quantization on MLP networks. The approach presented in this paper assumes that the inputs and weights have been quantized by N bits and follow a uniform distribution on $[-\Delta 2^{N-1}, \Delta 2^{N-1}]$, where Δ is the quantization step. The output y is uniformly distributed over $[-max(|y|), max(|y|)]$ and the nonlinear activation is approximated by a linear function. The assumptions are not realistic, since the distribution of the data changes depending on the application used. Moreover, neural networks are known for their non-linearity. Even if the assumptions are not realistic, they help simplify the formulas for the analysis of the quantization effect.

Choi and Choi [Choi and Choi, 1992] introduce a statistical method for measuring the sensitivity of MLPs with differentiable nonlinear functions. The sensitivity here is defined as a ratio of the standard deviation of the output errors and the standard deviation of the errors of the weights or inputs provided that the error tends to zero.

Dundar *et al.* [Dundar and Rose, 1995] extend the approach of [Xie and Jabri, 1992] but use the sigmoid function and not an approximation of the nonlinear function. The assumptions are the following: the inputs are continuous and uniform between 0 and 1; the weights are uniformly distributed in the interval $[-\Delta 2^{N-1}, \Delta 2^{N-1}]$, N being the number of bits used for quantization and Δ the quantization level; the quantization is done with a sufficiently large number of bits ($N \geq 8$) and the error is uniform and centered in the range $[-\Delta/2, \Delta/2]$.

Zeng et al. [Zeng and Yeung, 2001] uses a hypercube technique for the computation of the sensitivity. The inputs and weights are between 0 and 1. They are uniform and independent.

We take inspiration from sensitivity analysis and the rate distortion theory presented in Chapter 5 to propose a new distortion function which measures the gap between the Bayes risk of a classifier before and after the compression. Since this distortion is not tractable, we derive a theoretical closed-form approximation when the last fully connected layer of a deep neural network is compressed with a uniform quantizer. This approximation provides insight into the relationship between the accuracy loss and some key characteristics of the neural network. Numerical simulations show that the approximation is reasonably accurate.

7.2 Problem statement

7.2.1 Deep neural networks

We consider a classification problem with two classes $C = 2$. Let $f(\mathbf{x}_0)$ be a deep neural network of $K+1$ layers with $\mathbf{x}_0 \in \mathbb{R}^{n_0}$ being the input. The hidden layers are from $\mathbf{x}_1 \in \mathbb{R}^{n_1}$ to $\mathbf{x}_{K-1} \in \mathbb{R}^{n_{K-1}}$ and the output layer is denoted with $\mathbf{x}_K \in \mathbb{R}^{n_K}$. We recall the definition of a DNN given in (2.10):

$$\mathbf{x}_k = \sigma(\mathbf{W}_k \mathbf{x}_{k-1} + \mathbf{b}_k), \quad \forall 1 \leq k \leq K-1, \tag{7.1}$$

$$\hat{\mathbf{y}} = f(\mathbf{x}_0) = \mathbf{x}_K = \sigma_{\text{softmax}}(\mathbf{W}_K \mathbf{x}_{K-1} + \mathbf{b}_K), \tag{7.2}$$

The last layer, called the softmax layer [Goodfellow et al., 2016], depends on $\mathbf{W}_K \in \mathbb{R}^{2 \times n_{K-1}}$ and $\mathbf{b}_K \in \mathbb{R}^2$.

The output of the neural network $\hat{\mathbf{y}} = \hat{\mathbf{y}}(\mathbf{x}_0) = (\hat{y}_1(\mathbf{x}_0), \hat{y}_2(\mathbf{x}_0))$ is interpreted as a soft one-hot encoding vector. To decode $\hat{\mathbf{y}}$, we use the decision rule, denoted $\delta_f(\mathbf{x}_0)$, given by:

$$\delta_f(\mathbf{x}_0) = \arg\max_{i\in\{0,1\}} \hat{y}_i(\mathbf{x}_0). \tag{7.3}$$

It chooses the class with the highest probability given in $\hat{\mathbf{y}}$. To simplify the notations, the vector $\mathbf{x}_{K-1}$ will be denoted $\mathbf{x}$ in the rest of the chapter and n_{K-1} will be denoted n.
Let us note that δ_f can be rewritten as a linear classifier without the operators argmax and softmax. The decision rule (7.3) is equivalent to the linear decision rule $\delta_{f_{\mathbf{w}}}$:

$$\delta_{f_{\mathbf{w}}}(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{w}^T\mathbf{x} > \lambda, \\ 1 & \text{otherwise}, \end{cases} \tag{7.4}$$

where $\mathbf{w} = \mathbf{w}_0 - \mathbf{w}_1$, $\lambda = b_1 - b_0$ and $\mathbf{w}^T$ denotes the transpose of $\mathbf{w}$. Note that $\mathbf{w}_0$ and $\mathbf{w}_1$ represent the first and the second row of $\mathbf{W}_K$ and b_0, b_1 are the two components of the bias vector $\mathbf{b}_K$. The same transformation can also be applied to the compressed version. Since $\mathbf{x}$ follows a Gaussian distribution, $\mathbf{w}^T\mathbf{x}$ also follows a Gaussian distribution:

$$\mathbf{w}^T\mathbf{x} \sim \mathcal{N}(\mathbf{w}^T\mu_j, \mathbf{w}^T\Sigma_j\mathbf{w}). \tag{7.5}$$

We want to compare this classifier with the compressed version $\delta_{f_{\hat{\mathbf{w}}}}(\mathbf{x})$ defined as $\delta_{f_{\mathbf{w}}}(\mathbf{x})$ in (7.4) with $\mathbf{w}$ replaced by $\hat{\mathbf{w}}$, a compressed version of $\mathbf{w}$. We do not quantize $\lambda \in \mathbb{R}$ but the extension is trivial.

7.2.2 Minimum Bayes risk

The classification performance of a neural network (before or after a compression) is measured by the accuracy of the softmax layer, i.e., the accuracy $\mathrm{acc}(\delta_{f_{\mathbf{w}}}) = 1 - r(\delta_{f_{\mathbf{w}}})$ of the linear classifier $f_{\mathbf{w}}$ where $r(\delta_{f_{\mathbf{w}}})$ is the Bayes risk [Poor, 1994]:

$$r(\delta_{f_{\mathbf{w}}}) = \pi_0\mathbb{P}_0(\delta_{f_{\mathbf{w}}}(\mathbf{x}) \neq 0) + \pi_1\mathbb{P}_1(\delta_{f_{\mathbf{w}}}(\mathbf{x}) \neq 1), \tag{7.6}$$

where $\mathbb{P}_j(\cdot)$ stands for the conditional probability distribution of $\mathbf{x}$ given the class C_j and π_j is the prior probability of C_j.

In a layer of a ReLU neural network, a significant part of the neurons is generally quiet, i.e., their values are zero or very close to zero. Hence, for the classification task, only a part of the coefficients in $\mathbf{x}$, the non-zero coefficients, contribute to the decision. To simplify the notation, we will consider, without any loss of generality, that all the neurons of the last layer are non-zero. Furthermore, we assume that the input vector $\mathbf{x}$ of the last layer (i.e., the non-zero coefficients) follows a multivariate normal distribution:

$$\mathbf{x} \sim \mathcal{N}(\mu_j, I_n) \ \text{ under } \ C_j, \tag{7.7}$$

where $\mu_j \in \mathbb{R}^n$ is a known mean vector and $I_n \in \mathbb{R}^{n\times n}$ is the identity covariance matrix of size n.

We have experimented and analyzed various architectures, including Fully Connected and Convolutional Neural Networks [Simonyan and Zisserman, 2015, He et al., 2016], that support this

assumption (after the normalization of the coefficients to get the identity covariance matrix). Other related works [Piché, 1995, Gao et al., 2019] also assume that the inputs and weights follow a Gaussian distribution.

Let $\Phi(\cdot)$ be the cumulative distribution function of the standard normal distribution. The risks $\mathbb{P}_j(\delta_{f_{\mathbf{w}}}(\mathbf{x}) \neq j)$ in (7.6) are then

$$\mathbb{P}_j(\delta_{f_{\mathbf{w}}}(\mathbf{x}) \neq j) = \Phi\left((-1)^k a_j(\mathbf{w})\right), \tag{7.8}$$

where

$$a_j(\mathbf{w}) = \frac{\lambda - \mathbf{w}^T \mu_k}{\|\mathbf{w}\|_2}, \; j = 0, 1. \tag{7.9}$$

The calculation of (7.8) comes from the error analysis of a linear classifier as detailed in [Poor, 1994]. Similar results are obtained for the compressed classifier, denoted $\delta_{f_{\hat{\mathbf{w}}}}$, with the compressed weights $\hat{\mathbf{w}}$, where the $a_j(\mathbf{w})$'s are replaced with the $a_j(\hat{\mathbf{w}})$'s.

Let us assume that the training of the neural network leads to an almost optimal linear classifier in the last layer. It means that $\delta_{f_{\mathbf{w}}}(\mathbf{x})$ is the optimal Bayes classifier that minimizes the Bayes risk (7.6). It follows from [Poor, 1994] that the optimal parameters are given by:

$$\mathbf{w} = \mu_1 - \mu_0, \quad \lambda = \ln\frac{\pi_0}{\pi_1} + \frac{1}{2}(\|\mu_1\|_2 - \|\mu_0\|_2) \tag{7.10}$$

$$a_j(\mathbf{w}) = \frac{\ln\frac{\pi_0}{\pi_1}}{\|\mu_1 - \mu_0\|_2} + (-1)^k \frac{\|\mu_1 - \mu_0\|_2}{2}, \; j = 0, 1. \tag{7.11}$$

7.3 Distortion measure for classifiers

Commonly used distortion measures do not always reflect the accuracy loss when compressing the network. We define a particular distortion measure based on the classification risk. To distinguish the network before and after the compression, we use the notation $f_{\mathbf{w}}$ and, respectively $f_{\hat{\mathbf{w}}}$, for the uncompressed, resp. compressed, neural network. Hence, we consider the distortion function to be the absolute difference between the risks of the two classifiers:

$$d(\mathbf{w}, \hat{\mathbf{w}}) = |r(\delta_{f_{\mathbf{w}}}) - r(\delta_{f_{\hat{\mathbf{w}}}})|. \tag{7.12}$$

We want to understand the evolution of d as a function of the number of bits.
Hence, this chapter focuses on the study of the distortion measure (7.12). This will bring us closer to understanding the impact of compression methods on the last layer of a neural network and determine the minimal number of bits needed to ensure a given quality of the classification.
Using the risk from (7.6), we can rewrite (7.12) as follows

$$d(\mathbf{w}, \hat{\mathbf{w}}) = \left|\sum_{i=0}^{1} \pi_i [\mathbb{P}_i(f_{\mathbf{w}}(\mathbf{x}) > \lambda) - \mathbb{P}_i(f_{\hat{\mathbf{w}}}(\mathbf{x}) > \lambda)]\right|. \tag{7.13}$$

The distortion $d(\mathbf{w}, \hat{\mathbf{w}})$ can be easily computed by using (7.8). However, a numerical computation does not offer any information on the joint role of $\mathbf{w}$ and $\hat{\mathbf{w}}$. Without any additional assumptions

on the error, we cannot gain insight into the quality of the compression process applied to $\mathbf{w}$. In the case we are not interested in how $\hat{\mathbf{w}}$ was produced from $\mathbf{w}$, yet we want to measure the gap between the risks $r(\delta_{f_{\mathbf{w}}})$ and $r(\delta_{f_{\hat{\mathbf{w}}}})$, the approach would be to propose a bound which joins together $\mathbf{w}$ and $\hat{\mathbf{w}}$. This approach has been studied and published in [Resmerita et al., 2021a]. The computational details and some experiments can be found in the Appendix C.1.

This next section proposes a theoretical closed-form approximation when the last fully connected layer of a deep neural network is compressed with a uniform quantizer. This is a more direct approach that does not use bounds.

7.4 Distortion measure applied to uniform quantization

In this section, we want to predict the distortion induced by uniform quantization knowing properties of the dataset (the means of the classes), the architecture of the model (number of neurons of a layer) and also the rate of the quantization.

Uniform quantization. We recall that to quantize the weights $\mathbf{w}$ with R bits, we use a uniform quantizer as described in [Gersho and Gray, 1991] with a constant quantization step q that depends on the range of the vector $\mathbf{w} = (w_1, \ldots, w_n)$:

$$q = \frac{\max_{1\leq i\leq n} w_i - \min_{1\leq i\leq n} w_i}{2^R}. \tag{7.14}$$

In the rest of the chapter, we will assume that the quantization noise of a uniform quantizer $Q(\cdot)$ is uniform, i.e.,

$$\hat{w}_i = Q(w_i) = w_i + \Delta_i, \quad \forall 1 \leq i \leq n, \tag{7.15}$$

where $\hat{w}_i$ is the compressed version of w_i, Δ_i follows a uniform distribution $\mathcal{U}([-q/2, q/2])$ with mean $\mathbb{E}[\Delta_i] = 0$ and variance $\text{var}[\Delta_i] = \frac{q^2}{12}$. This assumption is common with a uniform quantizer [Widrow and Kollár, 2008] when q is not large.

We obtain $\hat{\mathbf{w}} = \mathbf{w} + \Delta$ where $\Delta = (\Delta_1, \ldots, \Delta_n)$ and the Δ_i's are independent.
Since $\hat{\mathbf{w}}$ is random, we study the expectation of the distortion, as a function of $\mathbf{w}$, with respect to the quantization noise, i.e.,

$$d(\mathbf{w}) = \mathbb{E}_\Delta[d(\mathbf{w}, \hat{\mathbf{w}})] = \mathbb{E}_\Delta[d(\mathbf{w}, \mathbf{w} + \Delta)], \tag{7.16}$$

where $\mathbb{E}_\Delta[\cdot]$ denotes the expectation with respect to Δ.

7.4.1 Approximation of the distortion function

The approximation of the distortion function (7.12) is done in two steps: i) we approximate the coefficients $a_j(\hat{\mathbf{w}})$ of the compressed classifier as a function of the uncompressed $a_j(\mathbf{w})$ and then ii) we approximate the distortion itself. Our first result is summarized in the following theorem. The proof for the theorem are detailed in the Appendix D.1.

Theorem 7.4.1. *Assume that* $\hat{\mathbf{w}} = \mathbf{w} + \Delta$ *as in (7.15). Then,*

$$a_j(\hat{\mathbf{w}}) = \frac{a_j(\mathbf{w})}{\sqrt{1+\gamma}} + o_p(\gamma/n), \tag{7.17}$$

where γ *is defined by*

$$\gamma = \frac{n\,q^2}{12\|\mathbf{w}\|_2^2}. \tag{7.18}$$

The term $o_p(\gamma/n)$ *denotes a random variable with mean and variance that are no larger than* γ/n.

Theorem 7.4.1 introduced the very important quantity γ that explains the impact of the quantization on $a_j(\hat{\mathbf{w}})$ with respect to $a_j(\mathbf{w})$. The quantity $1/\gamma$ can be interpreted as a signal-to-quantization-noise ratio where the signal is $\mathbf{w}$ and the quantization noise is Δ. Indeed, the numerator $\|\mathbf{w}\|_2^2/n$ in $1/\gamma$ can be interpreted as the average power of a coefficient w_i in $\mathbf{w}$ whereas the denominator $q^2/12$ is the variance of the corresponding noise Δ_i as defined in (7.15). Furthermore, since $\gamma > 0$, (7.17) shows that the coefficient $a_j(\hat{\mathbf{w}})$ shrinks toward zero. Hence, depending on the sign of $a_j(\mathbf{w})$, the conditional risks can increase or decrease. The next theorem is the main result of this chapter. It gives an approximation of the expectation of the distortion (7.16).

Theorem 7.4.2. *Assume that* $\hat{\mathbf{w}} = \mathbf{w} + \Delta$ *as in (7.15). Then,*

$$d(\mathbf{w}) = \left|\eta(\gamma)(\pi_0 a_0 \varphi(a_0) - \pi_1 a_1 \varphi(a_1)) + \epsilon(\gamma/n)\right|, \tag{7.19}$$

where $a_j = a_j(\mathbf{w})$, $\eta(\gamma)$ *is defined by*

$$\eta(\gamma) = 1 - \frac{1}{\sqrt{1+\gamma}}, \tag{7.20}$$

$\varphi(\cdot)$ *is the probability density function of the standard normal distribution and* $\epsilon(\gamma/n)$ *is non-random error term that is of the same order as* γ/n.

Theorem 7.4.2 shows that $d(\mathbf{w})$ is controlled by γ through the coefficient $\eta(\gamma)$. The distortion is also controlled by the value of the Bayes risk before compression. As shown in (7.8), the value of the Bayes risk depends on $a_0(\mathbf{w})$ and $a_1(\mathbf{w})$ that are involved in (7.19). If γ is not close enough to zero, the approximation (7.19) may not be accurate enough. Corollary 7.4.3 gives a more complex but more accurate approximation.

Corollary 7.4.3. *Assume that* $\hat{\mathbf{w}} = \mathbf{w} + \Delta$ *as in (7.15). Then,*

$$d(\mathbf{w}) = \left|\frac{\eta(\gamma)}{6}(\pi_0 a_0 \varrho(a_0) - \pi_1 a_1 \varrho(a_1)) + \varepsilon(\gamma/n)\right|, \tag{7.21}$$

where $t \mapsto \varrho(t)$ *is given by*

$$\varrho(t) = \varphi(t) + 4\varphi\left(\frac{\zeta(\gamma)}{2}t\right) + \varphi\left(\frac{t}{\sqrt{1+\gamma}}\right), \tag{7.22}$$

$\eta(\gamma)$ *is defined in (7.20),* $\zeta(\gamma)$ *is*

$$\zeta(\gamma) = \frac{1}{2}\left(1 + \frac{1}{\sqrt{1+\gamma}}\right) = 1 - \frac{\eta(\gamma)}{2}, \tag{7.23}$$

and $\varepsilon(\gamma/n)$ *is an error term that is of the same order as* γ/n.

The definition of $\varrho(t)$ looks like Simpson's rule [Gautschi, 2011] because our approximation is based on a numerical integration. For this reason, the error term $\varepsilon(\gamma/n)$ that depends on the $a_j(\mathbf{w})$'s is smaller than $\epsilon(\gamma/n)$.

7.5 Experiments

7.5.1 Numerical simulations

In order to analyze the accuracy of the two proposed approximations of the distortion, several experiments have been conducted. The presented comparison scenarios were performed in a theoretical context using the optimal parameters and bias under the Gaussian assumption. We chose the means μ_0 and μ_1 of the classes by using a fixed radius α and an angle θ that varied between 0 and 180 degrees to simulate different difficulty levels of the classification problem. The means are generated in the plane given by two arbitrary chosen orthonormal vectors (b_0, b_1) as follows:

$$\mu_0 = \alpha\, b_0 \text{ and} \tag{7.24}$$

$$\mu_1 = \alpha\, (b_0 \cos\theta + b_1 \sin\theta). \tag{7.25}$$

The weights $\mathbf{w}$ and bias λ were computed using (7.10) with equiprobable classes $\pi_0 = \pi_1 = 1/2$. To simulate the quantization error Δ, given a number of bits R, we generated a uniform error between $[-q/2, q/2]$ and added it to the weights. The average of the risk (7.6) and the expected distortion (7.16) were computed using 1000 Monte Carlo samples of Δ.

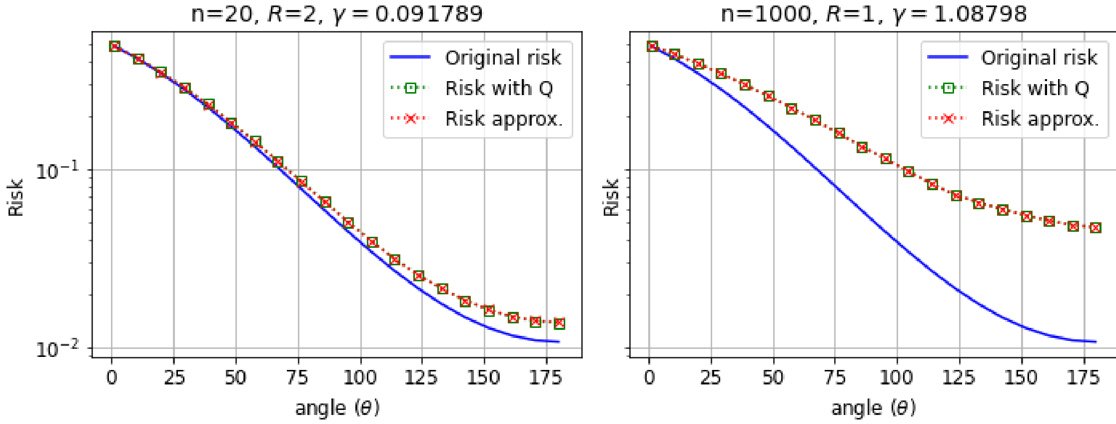


Figure 7.2: Comparison of classification risks for two scenarios.

We first focus on the quality of our approximations. To better illustrate the impact of γ on the approximations, we present two scenarios. Figure 7.2 shows the risk r before and after quantization and the risk computed with the approximation of $a_j(\hat{\mathbf{w}})$ given in Theorem 7.4.1. We generated the angle θ between the class means from 1 to 180 degrees. On the left side, we set the number of neurons $n = 20$ and the rate $R = 2$. In this case, we have γ small, close to 0. On the right side, we show the results when $R = 1$ bit and $n = 1000$ neurons: γ is not close to 0. In both cases, the approximation overlaps with the theoretical risk of the compressed model. Our evaluations validate the correctness of the approximation. It is worth mentioning that $\sqrt{1+\gamma}$ controls well the risk of the compressed model as established in Theorem 7.4.1. We also notice that γ, and therefore the distortion, strongly depends on the number of neurons n.

Figure 7.3 presents the distortions for the same scenarios. For the same given number of neurons and rate, we computed the true distortion (7.12) and also the two approximations (7.19) and (7.21).

On the left side, we observe that both approximations overlap with the theoretical distortion that we get with Monte Carlo. On the right side, we show the results when γ is not close to 0. In this scenario, we used $R = 1$ bit and $n = 1000$ neurons. We notice that the first approximation performs well until $\theta = 50$. After this point, we notice a small loss in precision. It is visible that the second approximation performs better in both cases. This result shows also that the number of neurons has an impact on the accuracy of our approximation. The error term $\varepsilon(\gamma/n)$ in Corollary 7.4.3 is smaller when n is large.

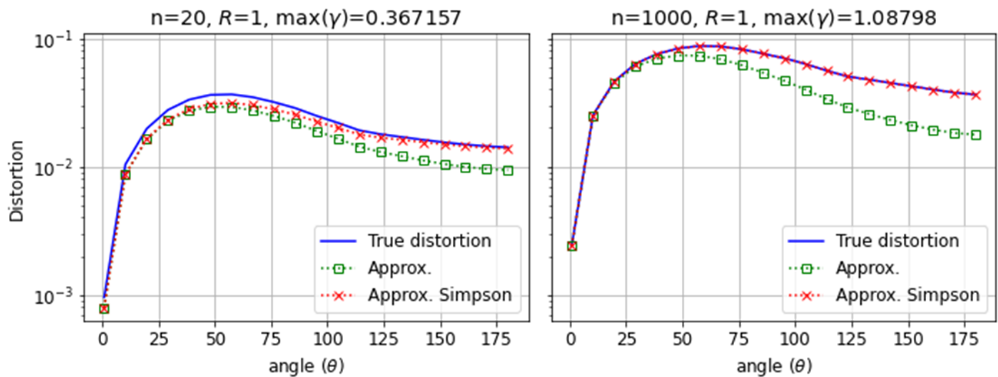


Figure 7.3: Comparison of the distortion approximations for two scenarios.

Figure 7.4 shows the rate distortion trade-off which is our main interest point. We present two scenarios with different numbers of neurons n and angles α. We quantized the weights at a rate R varying between 1 and 6 and we evaluated the true distortion measure and the two approximations. It is worth noting that as the rate increases, the distortion decreases. We find the behavior of the distortion consistent with the impact quantization has over data. On the left, both approximations perform well and overlap with the true distortion value. On the right, we observe that the approximation given in (7.19) follows closely the true distortion value, while the approximation given in (7.21) still performs better.

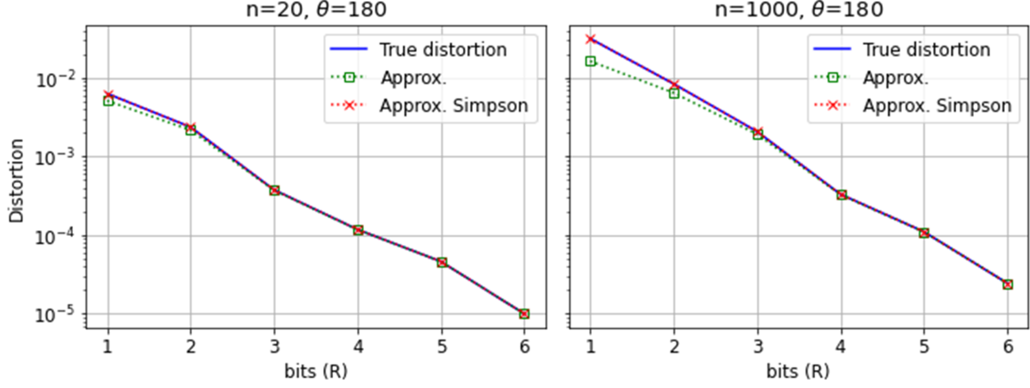


Figure 7.4: Rate distortion trade-off for two scenarios.

7.5.2 One-hidden-layer neural network on Sonar

We performed experiments using a one-hidden-layer network. Given that our distortion measure is adapted to two class models, we could not use multi-class datasets such as MNIST, CIFAR or ImageNet. Therefore, we trained the network on the **Sonar** dataset [Gorman and Sejnowski, 1988]. The dataset is composed of $N = 208$ instances, $n = 60$ attributes and two classes.

The network we trained had one FC layer with 60 neurons and a final softmax layer. We extracted the output of the hidden layer, which is the input of the softmax layer. For a fixed number of bits, we generated a uniform error Δ, which we add to the weights of the softmax layer.

Two experiments were carried out. In each of these experiments, we used a different activation function: a linear activation function, and, then, ReLU. We are aware that the distribution of weights and inputs depend on many factors: the architecture of the network, initialization, training. The change in the activation function also impacts the distribution of the input. For each experiment, a visualization of both $\mathbf{x}$ and $\mathbf{w}$ is done in order to showcase the differences.

7.5.2.1 Linear hidden layer

The first experiment is done using a linear activation function at the end of the hidden layer. Keeping the layer linear ensures the assumption of normality of $\mathbf{x}$.

Input and weight visualization. Figure 7.5 displays the distribution of $\mathbf{x}$ for both classes and the weights $\mathbf{w}$ of the network. A normal test has been performed which showed that the inputs follow a normal distribution.

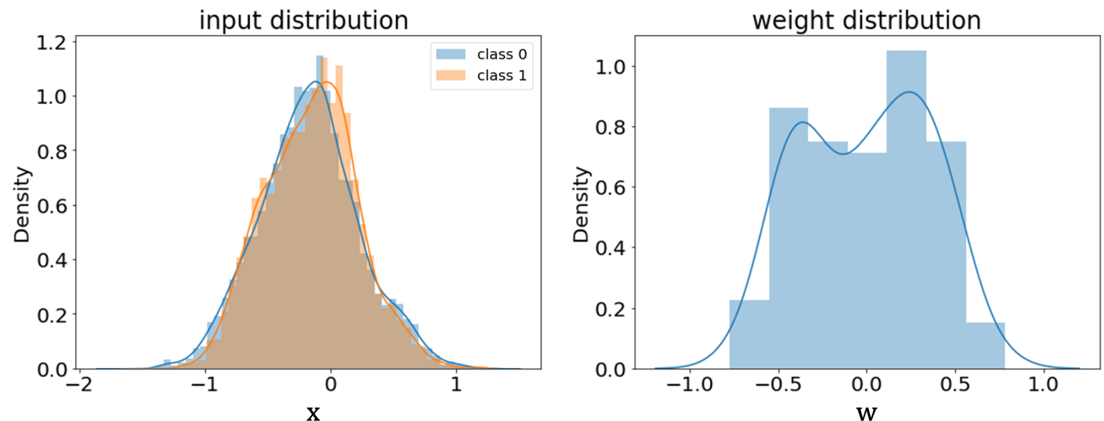


Figure 7.5: On the left, the distribution of the input for each class in the case of one hidden linear layer. On the right, the distribution of the trained weights.

In Figure 7.6, we showcase the distribution of $\mathbf{w}^T\mathbf{x}$ which also follows a normal distribution. In this practical case, our assumptions are valid.

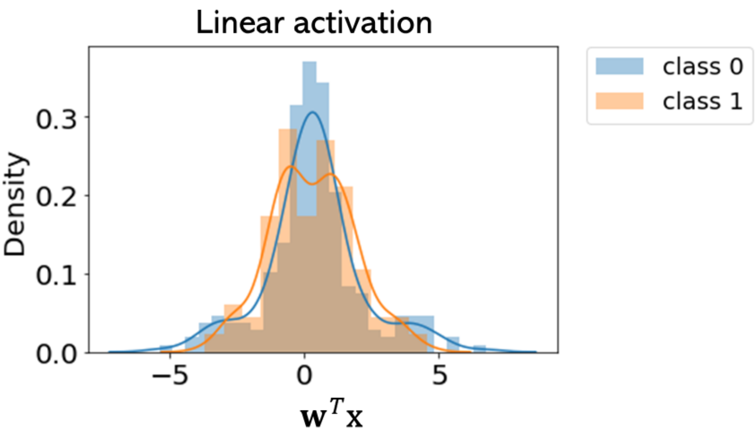


Figure 7.6: The distribution of $\mathbf{w}^T\mathbf{x}$ for each class for a network with a hidden linear layer. Note that the class 0 values have a higher p-value than class 1.

Results. After training the network, we obtained an empirical risk $\hat{r}(\delta_{f_{\mathbf{w}}}) = 0.0837$. In Figure 7.7, we observe our approximations have a similar behavior to the empirical distortion value $\hat{d}$. However, the approximation using Simpson is not performing better than the one given in Theorem 7.4.2.

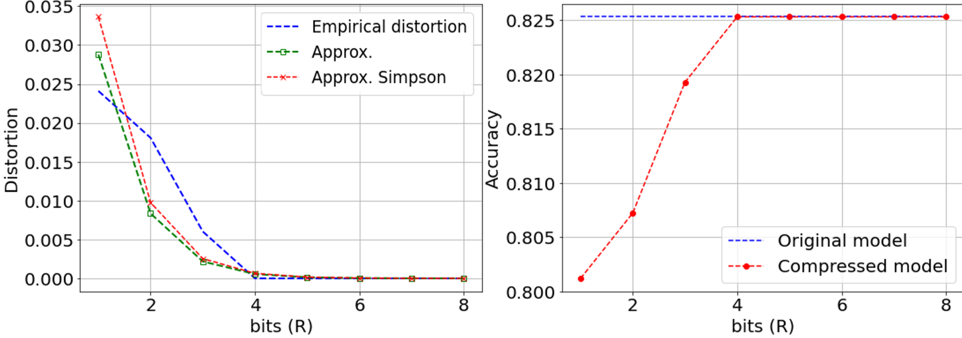


Figure 7.7: On the left, the rate distortion trade-off for one hidden linear layer. On the right, the accuracy for the original and compressed model.

7.5.2.2 ReLU hidden layer

The second experiment is done using the ReLU activation function. ReLU is a nonlinear function. By adding it to the hidden layer, there is a high chance of invalidating the assumption of normality made on the input.

Input and weight visualization. In Figure 7.8, we show the distribution of $\mathbf{x}$ and $\mathbf{w}$ in the case of ReLU. As predicted, the input fails the normality test, while the weights fail to reject the null hypothesis. It is true that in our case, only the input is considered a random variable, yet having the weights follow a normal distribution ensures that the linear combination between the input and the weights has a normal distribution. The shape of $\mathbf{w}^T\mathbf{x}$ is shown in Figure 7.9.

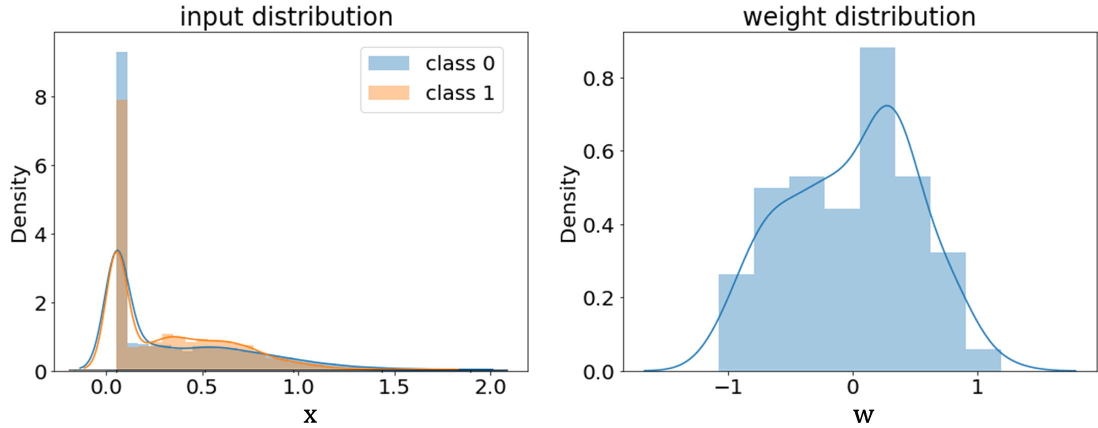


Figure 7.8: On the left, the distribution of the input for each class in the case of one hidden ReLU layer. On the right, the distribution of the trained weights.

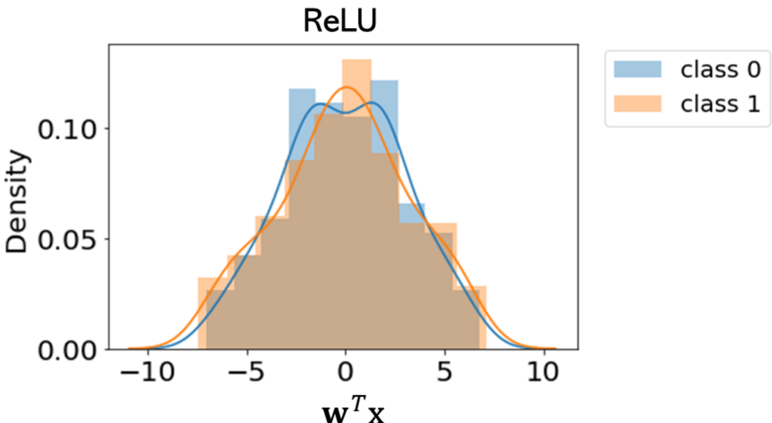


Figure 7.9: The distribution of $\mathbf{w}^T\mathbf{x}$ for each class in the case of a hidden ReLU layer.

Results. Training the network leads to an empirical risk $\hat{r}(\delta_{f_\mathbf{w}}) = 0.095$. In the case of the ReLU, we observe a higher difference between the empirical distortion $\hat{d}$ and the approximations, which is noticeable especially for 1 bit compression. We notice a slight improvement when using the Simpson approximation. Even though, the behavior of our approximations is less similar to the empirical distortion than in the previous case, they still perform reasonably well.

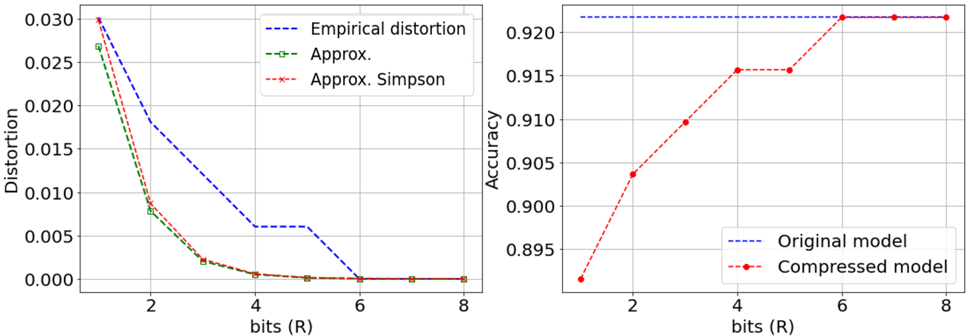


Figure 7.10: On the left, the rate distortion trade-off for one hidden ReLU layer. On the right, the accuracy for the original and compressed model.

7.6 Conclusion and Perspectives

In this chapter, we propose a new distortion measure in order to obtain a rate distortion curve. We have proposed two approximations for the average absolute perturbation on the accuracy of a classification neural network when its last layer is uniformly quantized. The approximations are accurate, whatever the number of bits is. They are easy to evaluate since they are given in closed-form.

Not only the approximations are accurate, but they also provide us with interesting insight on the impact quantization error has on the network. We show that γ is a key element in this distortion. As the number of bits increases, γ decreases. Finally, we have shown that the distortion explicitly depends on a signal-to-quantization-noise ratio we have introduced in this chapter.

As for future work, a direction would be to analyze if the initial Gaussian assumption on the inputs of the last layer can be dropped. We intuitively believe that simply using the central limit theorem can be enough to support the assumption. From our experiment using ReLU, we noticed that the approximations perform reasonably well even though the input does not follow a normal distribution. An extension of the analysis to a multiclass model and to the quantization of the other layers is necessary.

CHAPTER 8

Conclusions and Future works

8.1 Conclusion

Neural networks are used in critical real-time applications. The networks can easily overwhelm the resources of embedded systems. As stated in the introduction, this thesis is done in collaboration with Kalray a company which produces microprocessors. We have taken into account the requirements and limitations presented by the company. We mostly focused on compression methods for storage purposes, which does not involve training.

Overall, the objectives of the thesis were: (1) explore and identify deep compression methods for storage purposes which can be supported by Kalray's processors and (2) quantify the impact of quantization error on the networks' accuracy.

Regarding the first objective, we performed a study of weight compression methods. Several algorithms such as pruning, quantization and binarization were explored and we identified quantization as our main focus. Quantization can be applied without changing the architecture of the networks, or retraining the model. Algorithms such as uniform, non-uniform and low precision quantization were applied on several CNN architectures. Our work motivated the integration of quantization algorithms and incorporating look-up tables in future Kalray MPPA processors. For low precision quantization, we performed experiments using three floating-point representations of parameters for deep learning inference in classification and detection networks. We found that the Posit8 representation with 1 bit or 2 exponent bits also tends to perform well with most neural networks, with slight exceptions. Our results also motivated considering the inclusion of a Posit8 to FP16 hardware decompressor in future Kalray MPPA processor, as the tensor coprocessors of the MPPA3 processor already include exact FP16 to FP32 dot product operators.

Regarding the second objective, which is of theoretical nature. It aims to provide more insight on how neural networks are impacted by quantization. We focused on the statistical analysis of the impact of quantization error on the accuracy of classification models. In order to quantify the impact, we introduced a new distortion measure that calculates the difference between the Bayes risk of the model before and after compression with a given quantization algorithm. Our theoretical analysis is performed only on the last layer of a neural network in the case of a binary classification. We applied the approximations on real data using a one hidden layer neural network. We noticed the distortion measure has a signal-to-noise ratio form. Moreover, we observed that the quality of the approximation could therefore essentially depend on the dimension of the hidden

layer for a given number of bits. It seems that, for any number of bits, the approximation can be very efficient if the number of neurons is large enough. A more accurate approximation, but more complex, is given in as corollary. They are easy to evaluate since they are given in closed-form. Although the method is still in its early stages, our results look promising.

8.2 Perspectives

Finally, we discuss the general perspectives of the work presented in this thesis. Several directions can be taken.

8.2.1 Rate distortion measure

Regarding the distortion measure presented in Chapter 7, extending the analysis to deeper architectures and multiclass models would be one of the most straightforward focuses. Focusing on these two aspects will get us closer to having a distortion measure which can be used on complex networks used in real-time applications. In order to take into account quantization from previous layers such as convolutions, the idea would be to start by injecting error into the input. First, a detailed analysis should be performed on the effect of the ReLU function on the inputs with and without error. Assuming the inputs follow a Gaussian distribution, we can consider that the input after ReLU follows a rectified Gaussian distribution. The propagated error which is added to the input can be assumed to follow a Normal distribution.

In order to improve the presented method, one lead would be to study the possibility of removing the Gaussian assumptions on the inputs of the last layer. We can still assume that the output of the layer before the activation function follows a Gaussian distribution. This assumption is reasonable, since a non-zero input of the last layer is a sum of a significant number of values coming from the previous layer. Hence, we can invoke the central limit theorem to support this assumption.

As for quantization algorithm, in this work, we focused on uniform quantization error due to ease of interpretation. For future work, another interesting direction would be to focus on other types of compression algorithms and quantization errors. Taking as example Posits, which were presented in Chapter 6, we would need to estimate the distributions of each Posit<8,es> and of the new injected error. The new error depends on both the number of bits fixed at 8 and the es parameter which changes the dynamic range (variance) of the format.

8.2.2 End-to-end deep learning compression tool

One idea would be to study other compression methods complementary to quantization in order to create an end-to-end tool adapted for future generations of the processor. Some interesting strategies would be pruning, or low rank approximation.

As presented in Chapter 3, pruning is one of the targeted methods. In Chapter 4, we show that pruning is a powerful method which is able to obtain a compression rate of 90% without loss. For now, this method has not been supported due to hardware limitations. To take advantage of this method, the processors need to adapt to sparsity. Hardware friendly and efficient pruning techniques have been proposed [Yu et al., 2017a, Li et al., 2020, Hubara et al., 2021]. Another

limitation is the need to retrain the network. The MPPA is a proceesor which can be used only for deep learning inference tasks. Training is not supported. If needed, training should be done on a different platform (CPU/GPU).

Another method would be low rank approximation which was also presented in Chapter 3. It is used to decompose a matrix in multiple sparse matrices. This technique may reduce both the storage requirements and the computational complexity of the networks. However, since it is a lossy compression method, it might accumulate errors. Some works [Yu et al., 2017b, Swaminathan et al., 2020, Phan et al., 2020] mention that low rank approximation and pruning work well together.

These methods remain open topics which deserve an in-depth evaluation.

8.2.3 Quantization and matrix acceleration

We recall this thesis focuses only on compression methods for storage purposes. Even though acceleration computations have not been tackled throughout this manuscript, it has been discussed. We are interested in algorithms that can speed up the execution of a Deep Learning network and we need to determine which of these techniques are the most promising with respect to the architecture. In order to speed up the execution of CNNs, efficient matrix multiplication algorithms have been proposed.

A common method for dealing with layers is to use GEMM [Cong and Xiao, 2014], a matrix multiplication procedure that is part of the BLAS library [Lawson et al., 1979]. Convolutions are computed with the FFT algorithm [Mathieu et al., 2014, Vasilache et al., 2015]. However, the Winograd algorithm [Winograd, 1980, Lavin and Gray, 2016] is efficient for small convolutions, which are the most usual. This algorithm can reduce the number of multiplications by a factor of 2.25x. The use of the Winograd algorithm with low precision quantization is of great interest to Kalray. A state-of-the-art analysis has been done on scientific papers and industrial solutions, but not many works tackle this subject. [Gong et al., 2018] discussed Winograd convolution and 8 bits low precision inference. They explore the INT8 Winograd convolution. They apply the Winograd transformation F(2,3) on quantized values and then they use a scaling factor after transformation. Different scaling factors for weights and activations. They have tested their algorithm on VGG16 and the accuracy loss Top-1 and Top-5 is within 0.25% and 0.30%. Intel MKL-DNN [Intel, 2019] has supported int8 Winograd in convolution. For them, Winograd and quantization work independently. Recently, more scientific works have emerged on the subject [Yao et al., 2020, Li et al., 2021] which show high potential speeding up deep neural networks. An interesting next step would be to implement a proof of concept to evaluate Winograd using INT8 fixed point quantization.

My publications

[Resmerita et al., 2019a] Resmerita, D., Farias, R. C., de Dinechin, B. D., and Fillatre, L. (2019a). Compression des réseaux de neurones profonds à base de quantification uniforme et non-uniforme. In *Colloque GRETSI*.

[Resmerita et al., 2019b] Resmerita, D., Farias, R. C., de Dinechin, B. D., and Fillatre, L. (2019b). Compression des réseaux de neurones profonds à base de quantification uniforme et non-uniforme. In *ORASIS*.

[Resmerita et al., 2020] Resmerita, D., Farias, R. C., de Dinechin, B. D., and Fillatre, L. (2020). Benchmarking alternative floating-point formats for deep learning inference. In *COMPAS*.

[Resmerita et al., 2021a] Resmerita, D., Farias, R. C., de Dinechin, B. D., and Fillatre, L. (2021a). Classification error approximation of a compressed linear softmax layer. In *European Signal Processing Conference (EUSIPCO)*.

[Resmerita et al., 2021b] Resmerita, D., Farias, R. C., de Dinechin, B. D., and Fillatre, L. (2021b). Distortion approximation of a compressed softmax layer. In *2021 IEEE Statistical Signal Processing Workshop (SSP)*, pages 491–495.

[Resmerita et al., 2021c] Resmerita, D., Farias, R. C., de Dinechin, B. D., and Fillatre, L. (2021c). Représentations arithmétiques flottantes de taille réduite pour le deep learning. In *CORESA*.

Bibliography

[Abadi et al., 2016] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

[Abdelouahab et al., 2018] Abdelouahab, K., Pelcat, M., Berry, F., and Sérot, J. (2018). Accelerating CNN inference on FPGAs: A Survey. working paper or preprint.

[Anwar et al., 2017] Anwar, S., Hwang, K., and Sung, W. (2017). Structured pruning of deep convolutional neural networks. *JETC*, 13(3):32:1–32:18.

[Ba and Caruana, 2014] Ba, J. and Caruana, R. (2014). Do deep nets really need to be deep? In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

[Berthelier et al., 2021] Berthelier, A., Chateau, T., Duffner, S., Garcia, C., and Blanc, C. (2021). Deep model compression and architecture optimization for embedded systems: A survey. *J. Signal Process. Syst.*, 93:863–878.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[Borisyuk et al., 2018] Borisyuk, F., Gordo, A., and Sivakumar, V. (2018). Rosetta: Large scale system for text detection and recognition in images. In *Proceedings of the 24th ACM SIGKDD, KDD 2018*, pages 71–79.

[Bottou, 1998] Bottou, L. (1998). On-line learning and stochastic approximations. In *In On-line Learning in Neural Networks*, pages 9–42. Cambridge University Press.

[Bucilua et al., 2006] Bucilua, C., Caruana, R., and Niculescu-Mizil, A. (2006). Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 535–541, New York, NY, USA. Association for Computing Machinery.

[Burgess et al., 2019] Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., and Mansell, D. (2019). Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91. IEEE.

[Campbell and Broderick, 2018] Campbell, T. and Broderick, T. (2018). Bayesian coreset construction via greedy iterative geodesic ascent. In *International Conference on Machine Learning*, pages 698–706. PMLR.

[Carmichael et al., 2019] Carmichael, Z., Langroudi, H. F., Khazanov, C., Lillie, J., Gustafson, J. L., and Kudithipudi, D. (2019). Deep positron: A deep neural network using the posit number system. In *DATE*, pages 1421–1426. IEEE.

[Cheng et al., 2018] Cheng, J., Wang, P.-s., Li, G., Hu, Q.-h., and Lu, H.-q. (2018). Recent advances in efficient computation of deep convolutional neural networks. *Frontiers of Information Technology & Electronic Engineering*, 19(1):64–77.

[Chetlur et al., 2014] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759.

[Choi and Choi, 1992] Choi, J. Y. and Choi, C.-H. (1992). Sensitivity analysis of multilayer perceptron with differentiable activation functions. *IEEE Transactions on Neural Networks*, 3(1):101–107.

[Chollet, 2017] Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258.

[Chung et al., 2018] Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Ghandi, M., Lo, D., Reinhardt, S., Alkalay, S., Angepat, H., Chiou, D., Forin, A., Burger, D., Woods, L., Weisz, G., Haselman, M., and Zhang, D. (2018). Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38:8–20.

[Cococcioni et al., 2020a] Cococcioni, M., Rossi, F., Ruffaldi, E., and Saponara, S. (2020a). Fast approximations of activation functions in deep neural networks when using posit arithmetic. *Sensors*, 20(5):1515.

[Cococcioni et al., 2020b] Cococcioni, M., Rossi, F., Ruffaldi, E., Saponara, S., and de Dinechin, B. D. (2020b). Novel Arithmetics in Deep Neural Networks Signal Processing for Autonomous Driving: Challenges and Opportunities. *IEEE Signal Processing Magazine*, 38(1):97–110.

[Cogneethi, 2021] Cogneethi (2021). Object detection intro. https://cogneethi.com/evodn/object_detection_intro/.

[Cong and Xiao, 2014] Cong, J. and Xiao, B. (2014). Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning - ICANN 2014 - 24th*, pages 281–290.

[Courbariaux et al., 2015a] Courbariaux, M., Bengio, Y., and David, J. (2015a). Low precision arithmetic for deep learning. In Bengio, Y. and LeCun, Y., editors, *3rd ICLR*.

[Courbariaux et al., 2015b] Courbariaux, M., Bengio, Y., and David, J.-P. (2015b). Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131.

[Cover and Thomas, 2006] Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory*. Wiley-Interscience, New York.

[Cox and Wermuth, 1991] Cox, D. R. and Wermuth, N. (1991). A simple approximation for bivariate and trivariate normal integrals. *International Statistical Review*, 59(2):263–269.

[Dai et al., 2019] Dai, X., Yin, H., and Jha, N. K. (2019). Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497.

[de Dinechin et al., 2019] de Dinechin, F., Forget, L., Muller, J.-M., and Uguen, Y. (2019). Posits: the good, the bad and the ugly. In *CoNGA'19: Proceedings of the Conference for Next Generation Arithmetic*, pages 1–10. ACM.

[Dean et al., 2012] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

[Denil et al., 2013] Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. (2013). Predicting parameters in deep learning. In Burges, C. J. C., Bottou, L., Ghahramani, Z., and Weinberger, K. Q., editors, *NIPS 26*, pages 2148–2156.

[Dundar and Rose, 1995] Dundar, G. and Rose, K. (1995). The effects of quantization on multilayer neural networks. *IEEE Transactions on Neural Networks*, 6:1446 – 1451.

[Eckart and Young, 1936] Eckart, C. and Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218.

[Everingham et al., 2010] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. (2010). The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338.

[Fei-Fei et al., 2021] Fei-Fei, L., Karpathy, A., and Johnson, J. (2021). Cs231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/index.html.

[Frankle and Carbin, 2018] Frankle, J. and Carbin, M. (2018). The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*.

[Gao et al., 2019] Gao, W., Liu, Y.-H., Wang, C., and Oh, S. (2019). Rate distortion for model Compression:From theory to practice. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2102–2111. PMLR.

[Gautschi, 2011] Gautschi, W. (2011). *Numerical Analysis*. Birkhäuser Basel.

[Ge, 2018] Ge, S. (2018). *Efficient Deep Learning in Network Compression and Acceleration*, chapter 6, pages 95–114. IntechOpen.

[Geiger et al., 2013] Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237.

[GemmlowP, 2019] GemmlowP (2019). Gemmlowp: a small self-contained low-precision gemm library. https://github.com/google/gemmlowp.

[Gersho and Gray, 1991] Gersho, A. and Gray, R. M. (1991). *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, USA.

[Gholami et al., 2021] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2021). A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*.

[Giffon et al., 2019] Giffon, L., Ayache, S., Artières, T., and Kadri, H. (2019). Deep networks with adaptive nyström approximation. In *IJCNN 2019-International Joint Conference on Neural Networks*.

[Giffon et al., 2021] Giffon, L., Ayache, S., Kadri, H., Artières, T., and Sicre, R. (2021). PSM-nets: Compressing Neural Networks with Product of Sparse Matrices. In *IJCNN*, Virtual Event, United States.

[Girshick et al., 2014] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587.

[Gong et al., 2018] Gong, J., Shen, H., Zhang, G., Liu, X., Li, S., Jin, G., Maheshwari, N., Fomenko, E., and Segal, E. (2018). Highly efficient 8-bit low precision inference of convolutional neural networks with intelcaffe. In *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*, ReQuEST '18, New York, NY, USA. Association for Computing Machinery.

[Gong et al., 2014] Gong, Y., Liu, L., Yang, M., and Bourdev, L. D. (2014). Compressing deep convolutional networks using vector quantization. *CoRR*.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[Google, 2021] Google (2021). Google using bfloat16 with tensorflow models. https://cloud.google.com/tpu/docs/bfloat16.

[Gorman and Sejnowski, 1988] Gorman, R. P. and Sejnowski, T. J. (1988). Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1(1):75–89.

[Gou et al., 2021] Gou, J., Yu, B., Maybank, S. J., and Tao, D. (2021). Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819.

[Gray, 1972] Gray, R. (1972). On the asymptotic eigenvalue distribution of toeplitz matrices. *IEEE Transactions on Information Theory*, 18(6):725–730.

[Guo et al., 2016] Guo, Y., Yao, A., and Chen, Y. (2016). Dynamic network surgery for efficient dnns. *arXiv preprint arXiv:1608.04493*.

[Gupta et al., 2015] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. In *Proceedings of the 32nd ICML*, volume 37, pages 1737–1746.

[Gustafson and Yonemoto, 2017] Gustafson, J. L. and Yonemoto, I. T. (2017). Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86.

[Gysel, 2016] Gysel, P. M. (2016). *Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks*. PhD thesis, University of California Davis.

[Hacene et al., 2020] Hacene, G. B., Gripon, V., Arzel, M., Farrugia, N., and Bengio, Y. (2020). Quantized guided pruning for efficient hardware implementations of deep neural networks. In *2020 18th IEEE International New Circuits and Systems Conference (NEWCAS)*, pages 206–209. IEEE.

[Han et al., 2016a] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. (2016a). Eie: efficient inference engine on compressed deep neural network. In *ACM/IEEE 43rd Annual ISCA*, pages 243–254. IEEE.

[Han et al., 2016b] Han, S., Mao, H., and Dally, W. J. (2016b). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Bengio, Y. and LeCun, Y., editors, *4th ICLR*.

[Han et al., 2015] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1135–1143, Cambridge, MA, USA. MIT Press.

[Harshman et al., 1970] Harshman, R. A. et al. (1970). Foundations of the parafac procedure: Models and conditions for an" explanatory" multimodal factor analysis. *UCLA Working Papers in Phonetics*.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of IEEE conference on CVPR*, pages 770–778.

[He et al., 2019] He, Y., Liu, P., Wang, Z., Hu, Z., and Yang, Y. (2019). Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4340–4349.

[He et al., 2017] He, Y., Zhang, X., and Sun, J. (2017). Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397.

[Hinton et al., 2015] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[Ho and Wong, 2017] Ho, N.-M. and Wong, W.-F. (2017). Exploiting half precision arithmetic in nvidia gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7.

[Howard et al., 2018] Howard, A., Zhmoginov, A., Chen, L.-C., Sandler, M., and Zhu, M. (2018). Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. In *CVPR*.

[Howard et al., 2017] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*.

[Huang et al., 2019] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. (2019). Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112.

[Hubara et al., 2021] Hubara, I., Chmiel, B., Island, M., Banner, R., Naor, S., and Soudry, D. (2021). Accelerated sparse neural training: A provable and efficient method to find N: M transposable masks. *CoRR*, abs/2102.08124.

[Hubara et al., 2016] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks. In *NIPS 29*, pages 4107–4115.

[Hubara et al., 2017] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2017). Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18:187:1–187:30.

[Huffman, 1952] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

[Iandola et al., 2016] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 MB model size.

[Intel, 2018] Intel (2018). BFLOAT16 – Hardware Numerics Definition Revision 1.0.

[Intel, 2019] Intel (2019). Intel(R) MKL-DNN. https://oneapi-src.github.io/oneDNN/v0/index.html.

[Intel, 2021] Intel (2021). The difference between deep learning training and inference. https://www.intel.com/content/www/us/en/artificial-intelligence/posts/deep-learning-training-and-inference.html.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.

[Jacob et al., 2018] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., and Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE CVPR 2018*, pages 2704–2713.

[Jaiswal and So, 2019] Jaiswal, M. K. and So, H. K.-H. (2019). Pacogen: A hardware posit arithmetic core generator. *Ieee access*, 7:74586–74601.

[Jia et al., 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*.

[Johnson, 2018] Johnson, J. (2018). Rethinking floating point for deep learning. *ArXiv*, abs/1811.01721.

[Jégou et al., 2011] Jégou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128.

[Kalamkar et al., 2019] Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., et al. (2019). A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*.

[Karim, 2019] Karim, R. (2019). Illustrated: 10 cnn architectures. https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d.

[Khan et al., 2020] Khan, A., Sohail, A., Zahoora, U., and Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516.

[Klema and Laub, 1980] Klema, V. and Laub, A. (1980). The singular value decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control*, 25(2):164–176.

[Köster et al., 2017] Köster, U., Webb, T., Wang, X., Nassar, M., Bansal, A. K., Constable, W., Elibol, O., Gray, S., Hall, S., Hornof, L., Khosrowshahi, A., Kloss, C., Pai, R. J., and Rao, N. (2017). Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Krishnamoorthi, 2018] Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*.

[Krizhevsky et al., 2009] Krizhevsky, A., Nair, V., and Hinton, G. (2009). Cifar10/100.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *NIPS 25*, pages 1097–1105.

[Kullback and Leibler, 1951] Kullback, S. and Leibler, R. A. (1951). On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86.

[Kumar et al., 2012] Kumar, S., Mohri, M., and Talwalkar, A. (2012). Sampling methods for the Nyström method. *The Journal of Machine Learning Research*, 13(1):981–1006.

[Lavin and Gray, 2016] Lavin, A. and Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021.

[Lawson et al., 1979] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323.

[Le Tan et al., 2018] Le Tan, D.-K., Le, H., Hoang, T., Do, T.-T., and Cheung, N.-M. (2018). Deepvq: A deep network architecture for vector quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Li et al., 2020] Li, B., Kong, Z., Zhang, T., Li, J., Li, Z., Liu, H., and Ding, C. (2020). Efficient transformer-based large scale language representations using hardware-friendly block structured pruning. *CoRR*, abs/2009.08065.

[Li et al., 2021] Li, G., Jia, Z., Feng, X., and Wang, Y. (2021). Lowino: Towards efficient low-precision winograd convolutions on modern cpus. In Sun, X., Shende, S., Kalé, L. V., and Chen, Y., editors, *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, pages 81:1–81:11. ACM.

[Li et al., 2019] Li, X., Liang, Y., Yan, S., Jia, L., and Li, Y. (2019). A coordinated tiling and batching framework for efficient gemm on gpus. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 229–241, New York, NY, USA. Association for Computing Machinery.

[Lin et al., 2014] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[Lin and Bai, 2011] Lin, Z. and Bai, Z. (2011). *Probability Inequalities*. Springer-Verlag Berlin Heidelberg.

[Liu et al., 2016] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer.

[Lloyd, 1982] Lloyd, S. P. (1982). Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137.

[Loy, 2018] Loy, J. (2018). How to build your own neural network from scratch in python. https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6.

[Lu et al., 2020] Lu, J., Fang, C., Xu, M., Lin, J., and Wang, Z. (2020). Evaluations on deep neural networks training using posit number system. *IEEE Transactions on Computers*, 70(2):174–187.

[Lutz, 2019] Lutz, D. (2019). Arm floating point 2019: Latency, area, power. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 97–98. IEEE.

[MacQueen, 1967] MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967).

[Mao et al., 2017] Mao, H., Han, S., Pool, J., Li, W., Liu, X., Wang, Y., and Dally, W. J. (2017). Exploring the regularity of sparse structure in convolutional neural networks. *CoRR*.

[Mathieu et al., 2014] Mathieu, M., Henaff, M., and LeCun, Y. (2014). Fast training of convolutional networks through ffts. In *ICLR*.

[Mathworks, 2021] Mathworks (2021). Réseau neuronal convolutif. https://fr.mathworks.com/discovery/convolutional-neural-network-matlab.html.

[Micikevicius et al., 2018] Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. (2018). Mixed precision training. In *ICLR*. OpenReview.net.

[Mishra et al., 2020] Mishra, R., Gupta, H. P., and Dutta, T. (2020). A survey on deep neural network compression: Challenges, overview, and solutions. *arXiv preprint arXiv:2010.03954*.

[Molchanov et al., 2017] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2017). Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[Muller et al., 2018] Muller, J.-M., Brunie, N., de Dinechin, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., and Torres, S. (2018). *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9.

[Muller, 1959] Muller, M. E. (1959). A note on a method for generating points uniformly on n-dimensional spheres. *Comm. Assoc. Comput. Mach.*, 2:19–20.

[Murillo et al., 2020a] Murillo, R., Del Barrio, A. A., and Botella, G. (2020a). Customized posit adders and multipliers using the flopoco core generator. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

[Murillo et al., 2020b] Murillo, R., Del Barrio, A. A., and Botella, G. (2020b). Deep pensieve: A deep learning framework based on the posit number system. *Digital Signal Processing*, 102:102762.

[Neill, 2020] Neill, J. O. (2020). An overview of neural network compression. *arXiv preprint arXiv:2006.03669*.

[Nokleby et al., 2016] Nokleby, M., Beirami, A., and Calderbank, R. (2016). Rate-distortion bounds on bayes risk in supervised learning. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 2099–2103. IEEE.

[Novac et al., 2021] Novac, P.-E., Boukli Hacene, G., Pegatoquet, A., Miramond, B., and Gripon, V. (2021). Quantization and deployment of deep neural networks on microcontrollers. *Sensors*, 21(9).

[Nvidia, 2017] Nvidia (2017). 8 bit inference with TensorRT. http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inferencewith-tensorrt.pdf.

[Nvidia, 2018] Nvidia (2018). The nvidia deep learning accelerator. http://nvdla.org/.

[Nvidia, 2021] Nvidia (2021). Mlperf. https://www.nvidia.com/en-us/data-center/mlperf/.

[Oberbroeckling, 2021] Oberbroeckling, L. A. (2021). Chapter 11 - numerical integration. In Oberbroeckling, L. A., editor, *Programming Mathematics Using MATLAB®*, pages 183–191. Academic Press.

[Orabona and Pal, 2015] Orabona, F. and Pal, D. (2015). Optimal non-asymptotic lower bound on the minimax regret of learning with expert advice. *stat*, 1050:6.

[Paresh Kharya, 2020] Paresh Kharya, N. (2020). TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

[Phan et al., 2020] Phan, A. H., Sobolev, K., Sozykin, K., Ermilov, D., Gusak, J., Tichavský, P., Glukhov, V., Oseledets, I. V., and Cichocki, A. (2020). Stable low-rank tensor decomposition for compression of convolutional neural network. *CoRR*, abs/2008.05441.

[Piché, 1995] Piché, S. W. (1995). The selection of weight accuracies for madalines. *IEEE Transactions on Neural Networks*, 6:432 – 445.

[Poor, 1994] Poor, H. V. (1994). *An Introduction to Signal Detection and Estimation*. Springer-Verlag New York, 2nd edition.

[Posithub Survey, 2019] Posithub Survey, P. (2019). Survey of posit hardware and software development efforts. https://posithub.org/docs/PDS/PositEffortsSurvey.html.

[Qin et al., 2020] Qin, H., Gong, R., Liu, X., Bai, X., Song, J., and Sebe, N. (2020). Binary neural networks: A survey. *Pattern Recognition*, 105:107281.

[Rana, 2020] Rana, K. (2020). Pooling layer — short and simple. https://ai.plainenglish.io/pooling-layer-beginner-to-intermediate-fa0dbdce80eb.

[Rastegari et al., 2016] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: ImageNet classification using binary convolutional neural networks. In *Comput. Vis. ECCV*, pages 525–542. Springer.

[Redmon et al., 2016] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788.

[Redmon and Farhadi, 2018] Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *CoRR*, abs/1804.02767.

[Reed, 1993] Reed, R. (1993). Pruning algorithms-a survey. *IEEE Transactions on Neural Networks*, 4(5):740–747.

[Ren et al., 2015] Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99.

[Ross and Dollár, 2017] Ross, T.-Y. and Dollár, G. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2980–2988.

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

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(3):379–423.

[Simonyan and Zisserman, 2015] Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *3rd ICLR*.

[Steinhaus, 1957] Steinhaus, H. (1957). Sur la division des corps matériels en parties. *Bull. Acad. Pol. Sci., Cl. III*, 4:801–804.

[Swaminathan et al., 2020] Swaminathan, S., Garg, D., Kannan, R., and Andres, F. (2020). Sparse low rank factorization for deep neural network compression. *Neurocomputing*, 398:185–196.

[Sze et al., 2017] Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[Szegedy et al., 2016] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826.

[Szymon Migacz, 2017] Szymon Migacz, N. (2017). TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf.

[Tan and Le, 2019] Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR.

[Tan and Le, 2021] Tan, M. and Le, Q. (2021). Efficientnetv2: Smaller models and faster training. In Meila, M. and Zhang, T., editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10096–10106. PMLR.

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

[Tensorflow, 2019] Tensorflow (2019). High performance inference with tensorrt integration. https://blog.tensorflow.org/2019/06/high-performance-inference-with-TensorRT.html.

[TensorFlow, 2021] TensorFlow, G. (2021). Tensorflow quantization library. https://www.tensorflow.org/api_docs/python/tf/quantization.

[TensorFlow Lite, 2021] TensorFlow Lite, G. (2021). Tensorflow lite quantization solution. https://www.tensorflow.org/lite/performance/post_training_quantization.

[Tessier et al., 2020] Tessier, H., Gripon, V., Léonardon, M., Arzel, M., Hannagan, T., and Bertrand, D. (2020). Rethinking weight decay for efficient neural network pruning. *arXiv preprint arXiv:2011.10520*.

[Tichy, 2016] Tichy, W. (2016). The end of (numeric) error: An interview with john l. gustafson. *Ubiquity*, 2016:1–14.

[Tucker, 1966] Tucker, L. (1966). Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31.

[Uijlings et al., 2013] Uijlings, J., Sande, K., Gevers, T., and Smeulders, A. (2013). Selective search for object recognition. *International Journal of Computer Vision*, 104:154–171.

[Urban et al., 2016] Urban, G., Geras, K., Kahou, S. E., Aslan, O., Wang, S., Caruana, R., Mohamed, A., Philipose, M., and Richardson, M. (2016). Do deep convolutional nets really need to be deep (or even convolutional)? *Arxiv*, 521.

[Vasilache et al., 2015] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. (2015). Fast convolutional nets with fbfft: A GPU performance evaluation. In Bengio, Y. and LeCun, Y., editors, *ICLR*.

[Wen et al., 2016] Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. *arXiv preprint arXiv:1608.03665*.

[Widrow and Kollár, 2008] Widrow, B. and Kollár, I. (2008). *Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications*. Cambridge University Press, Cambridge, UK.

[Winograd, 1980] Winograd, S. (1980). *Arithmetic complexity of computations*, volume 33. Siam.

[Xiao et al., 2020] Xiao, F., Liang, F., Wu, B., Liang, J., Cheng, S., and Zhang, G. (2020). Posit arithmetic hardware implementations with the minimum cost divider and squareroot. *Electronics*, 9(10):1622.

[Xie and Jabri, 1992] Xie, Y. and Jabri, M. A. (1992). Analysis of the effects of quantization in multilayer neural networks using a statistical model. *IEEE Transactions on Neural Networks*, 3:334 – 338.

[Yao et al., 2020] Yao, Y., Li, Y., Wang, C., Yu, T., Chen, H., Jiang, X., Yang, J., Huang, J., Lin, W., Shu, H., and Lv, C. (2020). INT8 winograd acceleration for conv1d equipped ASR models deployed on mobile devices. *CoRR*, abs/2010.14841.

[Yeung et al., 2010] Yeung, D. S., Cloete, I., Shi, D., and Ng, W. W. (2010). *Principles of Sensitivity Analysis*, pages 17–24. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Yu et al., 2017a] Yu, J., Lukefahr, A., Palframan, D., Dasika, G., Das, R., and Mahlke, S. (2017a). Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 548–560, New York, NY, USA. Association for Computing Machinery.

[Yu et al., 2017b] Yu, X., Liu, T., Wang, X., and Tao, D. (2017b). On compressing deep models by low rank and sparse decomposition. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 67–76.

[Zeng and Yeung, 2001] Zeng, X. and Yeung, D. S. (2001). Sensitivity analysis of multilayer perceptron to input and weight perturbations. *IEEE Transactions on Neural Networks*, 12:1358 – 1366.

[Zhu et al., 2017] Zhu, C., Han, S., Mao, H., and Dally, W. J. (2017). Trained ternary quantization. In *ICLR*.

[Zhu and Gupta, 2018] Zhu, M. and Gupta, S. (2018). To prune, or not to prune: Exploring the efficacy of pruning for model compression. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net.

Pages web

[Kalray, 2021] Kalray (2021). Kalray MPPA Manycore. https://www.kalrayinc.com/products/mppa-technology.

[ONNX, 2019] ONNX, O. N. N. E. (2019). https://onnx.ai/.

List of Figures

1.1 Processor History [Kalray, 2021]. Intelligent processors are the next technology wave. . . . 1
1.2 Top level diagram of Kalray's cluster partition [Kalray, 2021]. The cluster (on the left) and processor (on the right). A list of the main features is given for each one of the units. . . . 4
1.3 Diagram of KaNN [Kalray, 2021]. A trained neural network is first given to the KaNN's front-end. In order to generate an optimized computation graph, further steps are performed such as optimization, memory allocation and scheduling. These steps are followed by code generation and deployment. During the deployment step, the generated code is sent to the MPPA Platform. Execution is done by giving an input to the generated KaNN model which will return either the class in the case of a classification network, or display the segmentation or detected objects. 5

2.1 Artificial Intelligence, Machine Learning and Deep Learning [Intel, 2021]. 11
2.2 Formalization of a neuron. Side-by-side illustrations of biological and artificial neurons [Fei-Fei et al., 2021]. Each neuron receives dendrites or inputs. The cell body of the neuron corresponds to the weighted sum with an added bias. The result of this operation is given to the activation function and represents the impulses carried out from the cell body. . . . 12
2.3 Architecture of a feedforward network with K hidden FC layers. . . . 14
2.4 Training vs. inference [Intel, 2021]. . . . 15
2.5 Gradient descent algorithm [Loy, 2018]. . . . 16
2.6 An example of a typical CNN. Convolutions are applied to images, and the output of each convolved image is used as the input to the next layer. Pooling layers are used to subsample the images. After obtaining the feature map, all the features are linked using an FC layer and softmax is applied to get the final results [Mathworks, 2021]. . . . 17
2.7 A two-dimensional convolution [Goodfellow et al., 2016]. . . . 18
2.8 An example of Max Pooling with a $2{\times}2$ filter and a stride of size 2 [Rana, 2020]. 18
2.9 An example of network which uses the BN layer. Usually, BN is inserted after CONV layers and before the activation function. . . . 19
2.10 Samples of the MNIST Dataset. . . . 20
2.11 Samples of the CIFAR Dataset. . . . 21
2.12 Samples of the ImageNet Dataset. . . . 21
2.13 Difference between classification, object detection [Cogneethi, 2021]. . . . 24
2.14 Samples from MS COCO dataset. . . . 25
2.15 Samples from KITTI dataset. . . . 26

3.1 The main categories of network compression approaches (inspired from [Ge, 2018]). 31
3.2 Deep Compression three-step pipeline [Han et al., 2016b]. . . . 32

3.3 Pruning methods applied to a 4-dimensional weight tensor of a convolution layer [Cheng et al., 2018]. This example contains three 3-D filters of three kernels each. A filter is a cube and a kernel is a slice in the cube. The elements in yellow are pruned weights. The fine-grained approach removes the parameters in an unstructured way. The vector-level approach prunes a vector from a kernel, while the kernel-level method prunes a kernel from the filter. Group-level methods use a sparse pattern to prune the parameters on the filters. Finally, filter-level pruning remove filters. 34
3.4 Pruning scheduling types. 35
3.5 Scalar quantization vs 2D Vector quantization. 36
3.6 A CNN block (left) vs a XNORNET block (right) [Rastegari et al., 2016]. In a typical CNN block, the order of operations is the following: CONV, Batch Norm, activation function. In a XNORNET block, the input is first normalized and, then, we apply a binary activation and a binary CONV. The Pooling operation is in the last position in both cases. 39

4.1 Architecture of LeNet5 [LeCun et al., 1998]. 44
4.2 Illustration of ResNet50's architecture from [Karim, 2019]. Symbol **B** stands for Batchnorm layer, **R** stands for ReLU activation function and **S** represents Softmax. 45
4.3 Accuracy comparison in the case of pruning with and without retraining using Lenet5 on MNIST. 46
4.4 Accuracy comparison in the case of pruning with and without retraining using Lenet5 on CIFAR-10. 46
4.5 Accuracy comparison between the steps used in Deep Compression: pruning with retraining, pruning with quantization and, finally, Deep Compression (with Huffman Coding). The model used is Lenet5 on MNIST. 48
4.6 Accuracy comparison between the steps used in Deep Compression: pruning with retraining, pruning with quantization and, finally, Deep Compression (with Huffman Coding). The model used is Lenet5 on CIFAR-10. 48
4.7 Accuracy comparison between pruning, quantization, deep compression and binarization using Lenet5 on MNIST. 49
4.8 Accuracy comparison between pruning, quantization, deep compression and binarization using Lenet5 on CIFAR-10. 50
4.9 Accuracy comparison between pruning, quantization, deep compression and binarization using ResNet50 on CIFAR-10. 51

5.1 Compression workflow. 56
5.2 Representation of uniform quantization. 57
5.3 Lloyd's algorithm (left) and Iteration (right) [Gersho and Gray, 1991]. 58
5.4 Uniform quantization vs non-uniform quantization applied to the weights of an FC layer. 59
5.5 Architecture of the CNN. 61
5.6 **Accuracy**: CONV 1 (top left), CONV 2 (top right), FC 1 (bottom left), FC 2 (bottom right). 62
5.7 **MSE**: CONV 1 (top left), CONV 2 (top right), FC 1 (bottom left), FC 2 (bottom right). 63

5.8 **Divergence KL**: CONV 1 (top left), CONV 2 (top right), FC 1 (bottom left), FC 2 (bottom right). 64
5.9 Accuracy, Div. KL and MSE for one quantizer for the entire network. 65
5.10 Architecture of VGG. 66
5.11 Accuracy, Div. KL and MSE for the first layer of VGG15 67
5.12 Accuracy, Div. KL and MSE for the last layer of VGG15 67

6.1 KaNN processing steps. A trained neural network and the input source are being pre-processed. The pre-processing steps include compressing the pre-trained weights and resizing the inputs. During KaNN's runtime, the weights are transferred to the clusters on the MPPA and then they are decompressed just before the computation step. When the runtime is finalized, the output goes to the post-processing step, where it is resized if needed. 71
6.2 Runtime steps. The weights are compressed during pre-processing. At runtime, the compressed weights are transferred from the external memory of the platform to internal memory of each cluster. After the transfer, they are decompressed and the computation is performed. Finally, the output given by the computation is transferred to the next cluster 72
6.3 Visual comparison of formats. Standard formats FP32 and FP16 are shown on top. BF16 has the same size as FP16, but the same exponent size as FP32. MSFP8 has a 5-bit exponent size, the same as FP16, but only a 2-bit mantissa. Posits have variable sized fields. In this figure, we illustrate the composition of the Posit and the priority of each component: regime, exponent and the fraction (or mantissa). 76
6.4 Histogram of Posit<8,es> values. Parameter es is between 0 and 3. The red pikes indicate the true representative values. The blue curve displays the density. 77
6.5 Runtime steps. The weights are compressed during pre-processing. At runtime, the compressed weights are transferred from the external memory of the platform to internal memory of each cluster. After the transfer, they are decompressed and the computation is performed. The output of the computation is compressed and ready to be transferred to the next cluster. 81

7.1 General structure of sensitivity analysis methods [Yeung et al., 2010]. 85
7.2 Comparison of classification risks for two scenarios. 91
7.3 Comparison of the distortion approximations for two scenarios. 92
7.4 Rate distortion trade-off for two scenarios. 92
7.5 On the left, the distribution of the input for each class in the case of one hidden linear layer. On the right, the distribution of the trained weights. 93
7.6 The distribution of $\mathbf{w}^T\mathbf{x}$ for each class for a network with a hidden linear layer. Note that the class 0 values have a higher p-value than class 1. 94
7.7 On the left, the rate distortion trade-off for one hidden linear layer. On the right, the accuracy for the original and compressed model. 94
7.8 On the left, the distribution of the input for each class in the case of one hidden ReLU layer. On the right, the distribution of the trained weights. 95
7.9 The distribution of $\mathbf{w}^T\mathbf{x}$ for each class in the case of a hidden ReLU layer. 95
7.10 On the left, the rate distortion trade-off for one hidden ReLU layer. On the right, the accuracy for the original and compressed model. 96

A.1 Optimizations applied in TensorRT. 125
A.2 Example of graph partition [Tensorflow, 2019]. TensorRT supported nodes in green. Starting from the bottom, one node is added at a time (see orange box). The only constraint is that the subgraph should be a direct cyclic graph and have no loops. If a loop is formed (as shown in the 4th image), then it goes back, the subgraph is complete, and a new subgraph is created for the last node. 128
A.3 Layer conversion and Engine built in TensorRT. The figure on the top left shows TensorFlows subgraph before conversion to TensorRT layers. Phase 2 is done first. Second graph on the top right, TensorFlow operations are converted to TensorRT layer (indicated in green). The third graph (bottom left) shows the result after the conversion phase, where all TensorFlow operations are converted to TensorRT layers. Phase 3 creates a TensorRT engine from the graphs. 129
C.4 Distortions $\hat{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$, $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $D(\mathbf{w}, \hat{\mathbf{w}}_s)$ for synthetic data experiment with XNOR-NET (left) and TFLite (right) as a function of the scaling factor. The orange circle and the green square represent respectively the minimum of $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $D(\mathbf{w}, \hat{\mathbf{w}}_s)$ and, in red, we show the scaling factors of XNOR-NET and TFLite. 141
C.5 Distortions $\hat{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$, $\tilde{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $\tilde{D}(\mathbf{w}, \hat{\mathbf{w}}_s)$ for Sonar dataset with XNOR-NET (left) and TFLite (right) as a function of the scaling factor. The orange circle and the green square represent the minimum of $\tilde{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $\tilde{D}(\mathbf{w}, \hat{\mathbf{w}}_s)$ and the red triangles show the scaling factors of XNOR-NET and TFLite. 143

Liste of Tables

1.1 Energy consumption for 32 bits operations [Han et al., 2016a]. 3

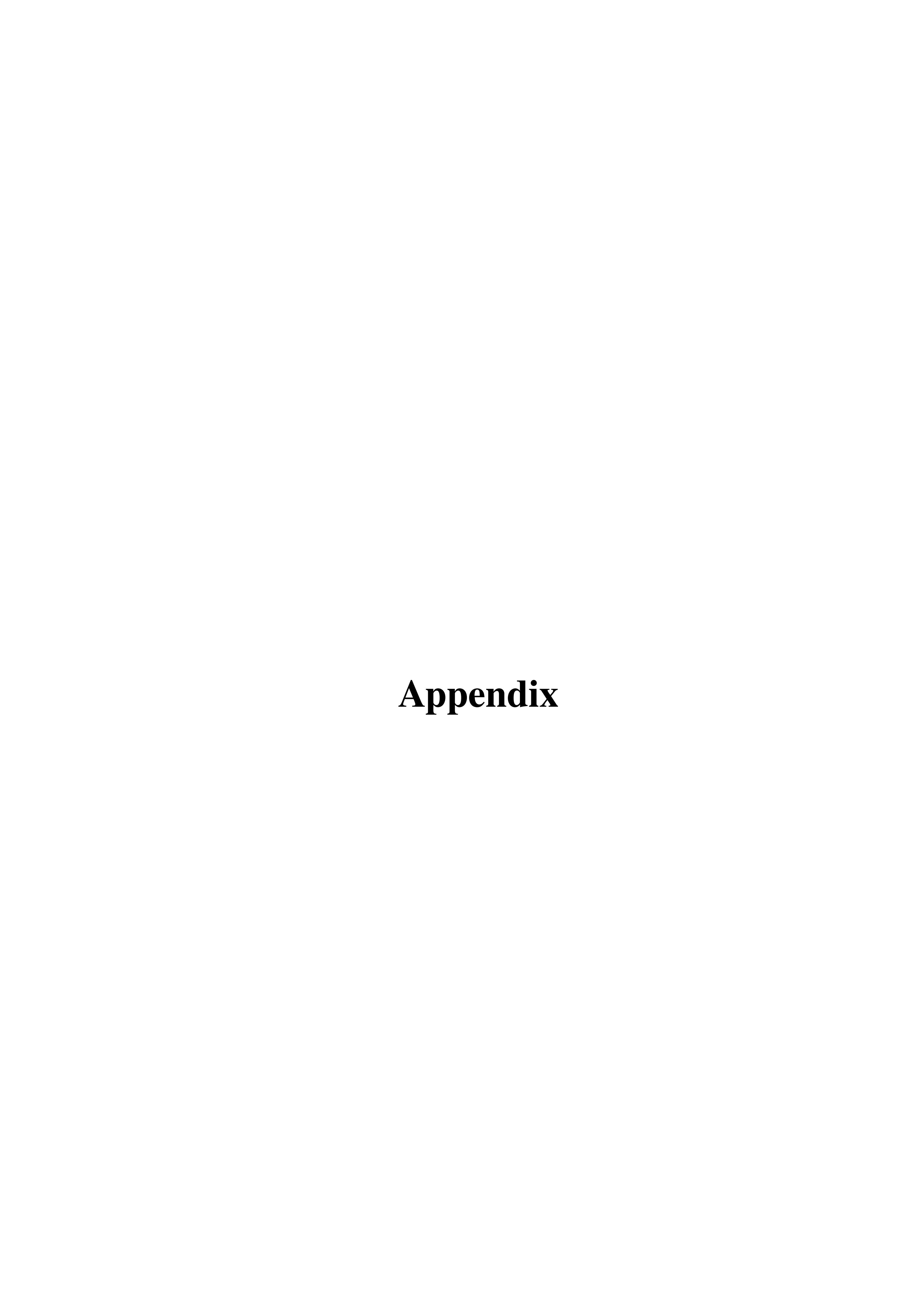
2.1 Commonly used activation functions. 12
2.2 Overview of well-known CNNs. Accuracy evaluated on ImageNet test set (except for LeNet-5 which is evaluated on MNIST). 23
2.3 Table of well-known object detection networks. The FPS was obtained on different GPUs. It serves as an indicator of the evolution made in object detection. 26

5.1 Parameters of the CNN trained on MNIST. 61
5.2 Statistics of compression for each layer. 65
5.3 Compression statistics for the network, with quantizers adapted to each layer. . . 66
5.4 Compression statistics for VGG15. 67

6.1 Regime interpretation. 75
6.2 Comparison of components and dynamic ranges for data formats. Note that the components of Posits have dynamic length. The indicated values of exponent and mantissa for Posit represent the maximum number of bits the components can have. The regime has priority over the bits. 75
6.3 Results for classification networks. Conversion is applied to all parameters. . . . 79
6.4 Results for detection network. Conversion is applied to all parameters. 79
6.5 Results for classification networks. Conversion is applied to parameters from convolutions and fully connected layers. Parameters from BN layers are kept in FP32. 80

1 Performances of three well-known Nvidia GPUs [Nvidia, 2018]. The table shows only the classification and object detection networks. The metric used is the number of inferences/second, or frames/second (fps) for a 1-batch inference. The results were published by Nvidia. 125
2 Number of parameters for each network grouped in several categories: bias, trainable weights, non-trainable weights and the total. 130
3 Intervals (min, max) for the entire network and for the special parameters: bias and non-trainable weights. 130
4 Comparisons between FP32, FP16, BF16 and MSFP8. For each network, we highlight the rows that indicate the formats which work the best. 131
5 Results for Posits. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$). 133
6 Results for Posits when we exclude the BN parameters from the conversion. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$). 134

7 Results for Posits when we exclude the biases from the conversion. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$). 136
8 Results for Posits when we exclude the BN parameters and the biases from the conversion. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$). 137



Appendix

A Chapter 3: Nvidia TensorRT study

A.1 Introduction

Compressing neural networks is a highly discussed topic. Kalray also needs to understand how its competitors compress and accelerate the networks for the inference task. In order to improve KaNN, Kalray's framework solution for accelerating networks, several questions need to be answered. What are the techniques that they support? How do they choose the proper algorithm for accelerating a particular network? How are the performance results computed? The answers to these questions can bring us closer to understanding what methods perform better in practice and are worth the time to implement in the close future.

One of the well-known R&D solutions for industries is TensorRT (TRT). TRT is a library developed by Nvidia for faster inference on Nvidia's GPUs. TensorRT is built on CUDA. It can give around 4-5x faster inference on real-time services and embedded applications and 40x faster inference compared to CPU only performance.

The main motivation for looking into TensorRT is because of their impressive performance published in the benchmarking tool, called MLPerf. This state-of-the-art study was done by using reverse engineering, since not all necessary information were provided by Nvidia. The study requires some knowledge on well-known networks and Deep Learning frameworks. We start by presenting MLPerf and the rules imposed, then the results and finally, we dive into the different optimization methods.

A.2 MLPerf

MLPerf [Nvidia, 2021] is a benchmarking tool built for measuring training and inference performance of ML hardware, software, and services. It is a widely accepted benchmark by the entire community, including researchers, developers, hardware manufacturers, builders of machine learning frameworks, cloud service providers, application providers, and end users. MLPerf v0.7 inference results were released in October 2020. Each MLPerf Inference benchmark is defined by a model, a dataset, a quality target, and a latency constraint.

A.2.1 Divisions

MLPerf Inference has two divisions for submitting results: closed and open. Participants can send results to either or both, but they must use the same data set. The closed division enables comparison of different systems. Submitters employ the same models, data sets, and quality targets to ensure comparability across wildly different architectures. This division requires preprocessing, postprocessing, and a model that is equivalent to the reference implementation. It also permits calibration for quantization and prohibits retraining.

MLPerf provides trained weights and biases in FP32 format for both the reference and alternative implementations. MLPerf also allows and enables quantization to many numerical formats to ensure architecture neutrality. The approved list includes INT4, INT8, INT16, UINT8, UINT16, FP11 (1-bit sign, 5-bit mantissa, and 5-bit exponent), FP16, BF16 [Google, 2021], and FP32. Quantization to lower-precision formats typically requires calibration to ensure sufficient inference quality. Therefore, to ensure sufficient inference quality, MLPerf also provides a small fixed calibration dataset for all models.

Additionally, for image classification using MobileNets-v1 224 and object detection using SSD-MobileNets-v1, MLPerf will provide a retrained INT8 (asymmetric for TFLite and symmetric for PyTorch/ONNX) model. All implementations are allowed as long as the latency and accuracy bounds are met. Weights can be modified according to the quantization rules. It is allowed to use variations of matrix-multiplication or convolution algorithms, mathematically equivalent transformations, fusing or unfusing operations, replacing dense operations with mathematically equivalent sparse operations. Some techniques that are not allowed: complete weight replacement, discarding non-zero weight elements (including pruning), weight quantization algorithms that are similar in size to the non-zero weights they produce, hard coding the total number of queries, online learning or related techniques.

The open division fosters innovation in ML systems, algorithms, optimization, and hardware/-software co-design. Participants must still perform the same ML task, but they may change the model architecture and the quality targets. This division allows arbitrary pre- and post-processing and arbitrary models, including techniques such as retraining. Some restricted retraining rules are given, but they are not mandatory. In general, submissions are directly comparable neither with each other nor with closed submissions. Each open submission must include documentation about how it deviates from the closed division.

A.2.2 Nvidia's results

Table 1 shows the results that Nvidia published on the performances of three of their well-known GPUs. For the image classification task, the only tested model in v0.7 is ResNet50 v1.5 which has 25.6M parameters and 8.2GOPS/input and achieves 76.456% TOP-1 accuracy in FP32. For object detection, two models are of interest. On the heavier side, SSD-ResNet34 which has 36.3M parameters and 433GOPS/input and achieves 0.20mAP, and on the lighter side SSD-MobileNetV1 with 6.91M parameters and 2.47GOPS/input and with 0.22 mAP performance. Nvidia with TensorRT has shown remarkable performance both in the datacenters (server, offline) and edge systems (single stream, multi-stream and offline). The results are shown in the table below for all the MLPerf tasks, including image classification and object detection.

A.3 TensorRT optimizations

Given the impressive results TensorRT has shown in MLPerf, we are interested in understanding how TensorRT optimizes the networks to increase performance and reduce memory requirements. TensorRT performs 5 types of optimization: layer fusion, precision calibration for INT8 quantization, kernel auto-tuning, dynamic tensor memory and multiple stream execution. The optimiza-

	Nvidia T4 (inferences/second)	Nvidia A100 (inferences/second)	Nvidia Jetson Xavier (inferences/second)
ResNet v1.5 (Image Classification)	6,112	37,331	2,075
MobileNet-v1 (Small Single Shot Detector)	995	6,401	2,533
ResNet-34 (Large Single Shot Detector)	139	974	51
3D U-Net (Medical Imaging)	7	42	2.3

Table 1: Performances of three well-known Nvidia GPUs [Nvidia, 2018]. The table shows only the classification and object detection networks. The metric used is the number of inferences/second, or frames/second (fps) for a 1-batch inference. The results were published by Nvidia.

tions are shown in Figure A.1. We detail the first two types as they are the most interesting in the context of this work.

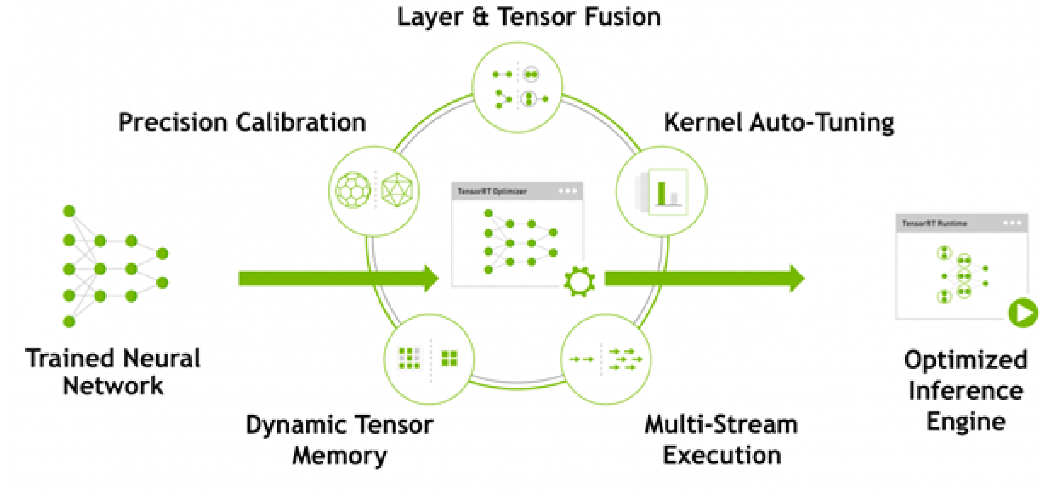


Figure A.1: Optimizations applied in TensorRT.

A.3.1 Horizontal and Vertical fusion of layers and tensors

To optimize the GPU memory and bandwidth, TensorRT fuses nodes into a single kernel, which reduces the cost of reading, writing and transferring the data for each layer. The supported types of fusions can be divided in the following categories:

1. Fusion of elementwise operations (i.e. Scale + Activation)
2. Convolution fusions (i.e. Conv + elementwise operation): convolutions followed by a simple sum, max, min or other elementwise
3. Shuffle fusion (i.e. Shuffle + Shuffle, Shuffle + Reduce)

4. Reduction operator fusions (sum of squares, L1Norm, L2Norm): these operators are mostly used during the training task

Another category would be QDQ Fusion. QDQ nodes help convert from FP32 to INT8 and vice versa. A quantized INT8 graph consists of quantization and dequantization operators with scales and zero points. More information on INT8 quantization is provided in the next item.

A.3.2 INT8 Quantization and precision calibration

Using lower precision reduces memory usage, allowing the deployment of larger networks. Data transfers take less time. Computational performance increases especially on GPUs with Tensor Core support for that precision. By default, TensorRT uses FP32 inference, but it also supports FP16 and INT8. While running FP16 inference, it automatically converts FP32 weights to FP16 weights.

TensorRT is quantizing both weights and activations in **INT8 precision**. The quantization is performed per layer. If the precision is not specified, then TensorRT will choose INT8 implementation only if it results in a higher performance network. If the implementation is faster in a higher precision, TensorRT will use it.

The paradigm used for quantization is for the most part the one used by Tensorflow [Krishnamoorthi, 2018] which we presented in Chapters 4 and 7. It is simply represented by a linear quantization. To quantize an input x, the following expression is used:

$$y = \left\lfloor \frac{x}{s} + z \right\rceil, \text{ where } y \in [-128, 127], \tag{A.1}$$

where s denote the scaling factor for the output y and $\lfloor \cdot \rceil$ the rounding operation.The quantization paradigm requires defining a zero-point constant z which represents the quantized value of the real value 0. Note that TensorRT only supports INT8 activations [-128, 127] and INT8 weights [-127, 127]. This means the value range is symmetric, therefore, the zero-point is equal to 0. To enable INT8 inference, one needs to provide TensorRT a dynamic range for each tensor including weights, input, and output tensors. One way to choose the dynamic range is by using the INT8 calibrator tool. In the case of quantization aware training (or if the calibration has not generated a satisfactory dynamic range for certain tensors), one can also skip this step and set custom per tensor dynamic ranges. However, the range needs to be symmetric. If this condition is not respected, then TensorRT chooses the larger absolute value of the provided bounds.

INT8 Calibration provides a way to generate the dynamic range per tensor. The calibrators compute the scaling factor for each tensor. This tool is useful as a post-training technique to generate the appropriate quantization scale. This process requires the network to pass a dataset of around 500 representative samples to estimate the scaling factors. TRT provides two main calibrators Entropy (recommended for CNN, required by DLA), MinMax (preferred for NLP tasks and recommended for BERT like networks). By default, calibration happens before layer fusion. During calibration, the builder runs a FP32 engine on the calibration dataset, records histograms of the distribution of each tensor, then builds a table from the histograms and, finally, builds the INT8 engine using the calibration table and the network definition. Quantized ONNX models can be

created using Quantization Aware Training (QAT) where FakeQuantization nodes are inserted to capture dynamic range (TensorFlow) or scale/zero-point (PyTorch).

A.3.3 Other optimization strategies

Kernel auto-tuning. While optimizing models, there is some kernel specific optimization which can be performed during the process. This selects the best layers, algorithms, and optimal batch size based on the target GPU platform. For example, there are multiple ways of performing convolution operation. TensorRT chooses the most optimal way on the selected platform. From our understanding, this is done simply by testing all the possible algorithms such as Winograd [Lavin and Gray, 2016], GEMM [Cong and Xiao, 2014], FFT [Mathieu et al., 2014] and combinations.

Dynamic tensor memory. TensorRT improves the memory reuse by allocating memory to tensor only for the duration of its usage. It helps in reducing the memory footprints and avoiding allocation overhead for fast and efficient execution.

Multiple stream execution. TensorRT is designed to process multiple input streams in parallel. This is basically Nvidia's CUDA stream.

A.4 TensorRT and Tensorflow

Tensorflow and TensorRT have worked closely together to speed up Deep Learning inference using GPUs. Three operations are performed in the optimization phase of the process:

Phase 1: Graph partition. TensorRT scans the Tensorflow graph backwards in order to find sub-graphs that it can optimize based on supported operations. It adds one node at a time to the subgraph, then wraps each TRT-compatible subgraph in a single node (TRTEngineOp) and uses the new node to replace the subgraph.

An example of graph partition is given in Figure A.2.

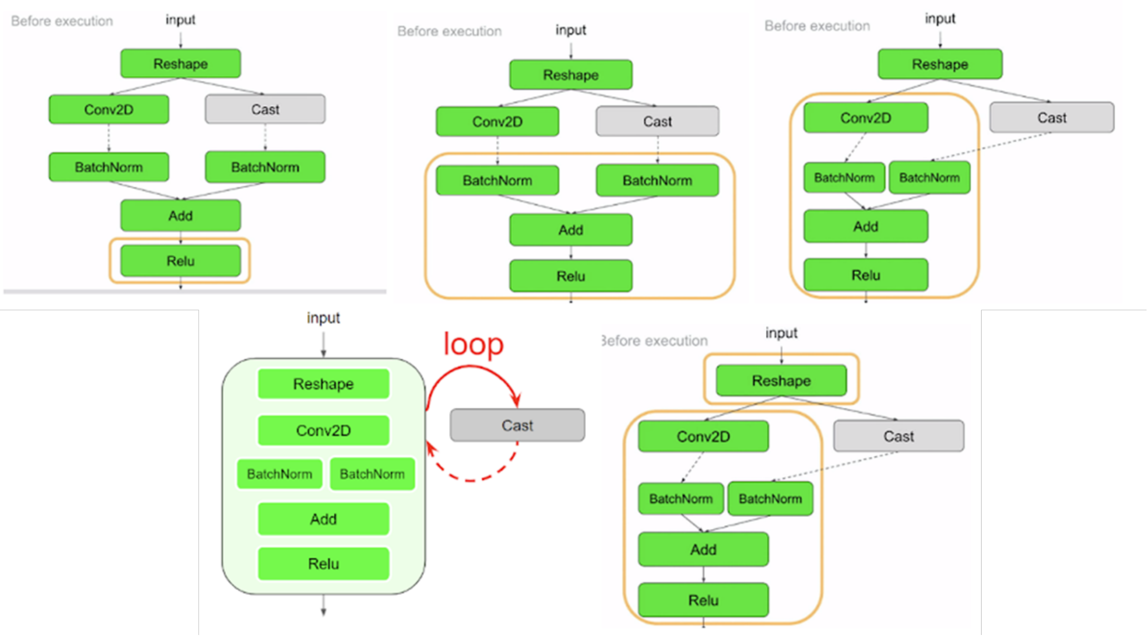


Figure A.2: Example of graph partition [Tensorflow, 2019]. TensorRT supported nodes in green. Starting from the bottom, one node is added at a time (see orange box). The only constraint is that the subgraph should be a direct cyclic graph and have no loops. If a loop is formed (as shown in the 4th image), then it goes back, the subgraph is complete, and a new subgraph is created for the last node.

Phase 2: Layer conversion The layer conversion is the second step of the optimization process. It is done when calling the *convert* method on the builder. The graph is processed in topological order and each Tensorflow operation in the subgraph is replaced by one or more TensorRT layer

Phase 3: Engine optimization The last step is applying TensorRT optimizations such as layer or tensor fusion, calibration for low precision) and kernel auto-tuning. These optimizations are transparent to the user and are applied to the current GPU. Calling the *build* method will apply this phase and will transform a subgraph TRTEngineOp into a TRT engine. The execution of phase 2 and 3 is illustrated in Figure A.3.

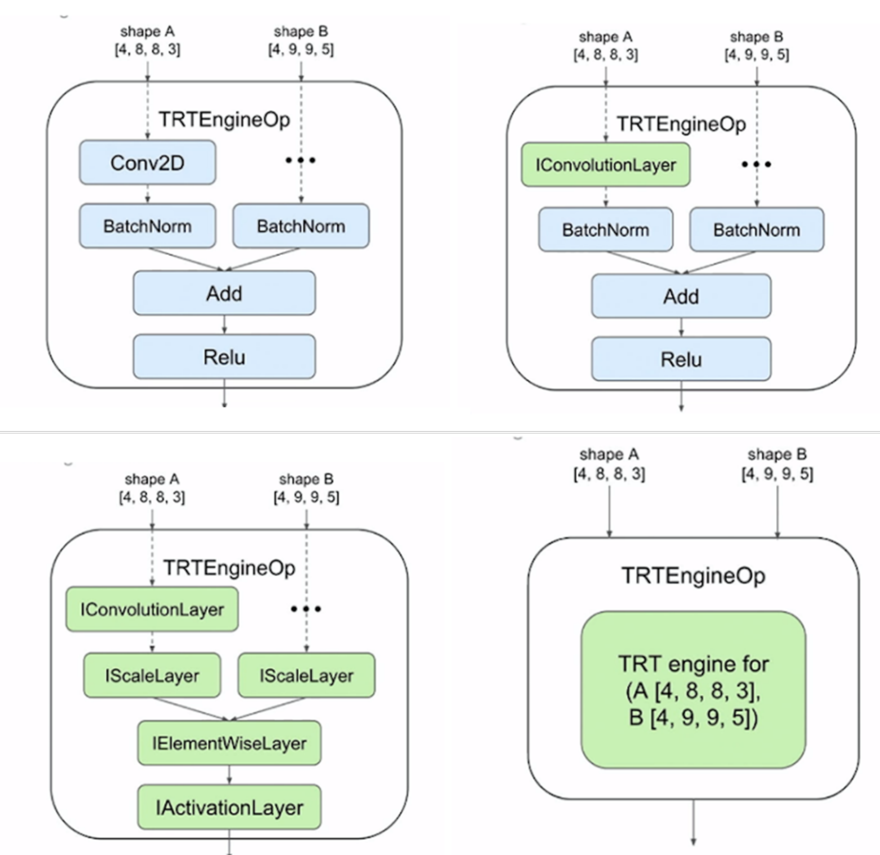


Figure A.3: Layer conversion and Engine built in TensorRT. The figure on the top left shows TensorFlows subgraph before conversion to TensorRT layers. Phase 2 is done first. Second graph on the top right, TensorFlow operations are converted to TensorRT layer (indicated in green). The third graph (bottom left) shows the result after the conversion phase, where all TensorFlow operations are converted to TensorRT layers. Phase 3 creates a TensorRT engine from the graphs.

Operation patterns such as Conv > Bias > ReLU or Conv > Bias > Batchnorm > ReLU are usually fused together. Therefore, quantization nodes should not be inserted between these layers.

A.5 Conclusion

In this section, we presented a preliminary study of TensorRT. We analyzed the main optimization strategies. We focused on understanding layer fusion and INT8 quantization. Both methods provide a great reduction of inference time. Layer fusion reduces the reading, writing and transfer time, while INT8 quantization reduces memory footprint.However, we are missing information on the algorithms used for matrix computation acceleration. Finding conclusive information in state-of-the-art works is challenging. In order to contribute directly to KALRAY's efforts to improve its own acceleration framework, it is essential to become more familiar with other existing acceleration solution and to use approaches such as benchmarking or profiling tools to understand the performance obtained. A practical study is required and it will be done in the future.

B Chapter 6: Alternative floating point formats

This section presents the complete results of our experiments using alternative floating point formats to compress the parameters. This work was discussed in Chapter 6. This section provides more information on the type and number of parameters of each tested network.

We recall that, in Chapter 5, we discussed that the MSE and the KL Divergence are not reliable distortion measures. This section contributes with extra information on the topic.

B.1 Additional data on network parameters

Table 2 contains a thorough description of the parameters of several networks. We give the total number of parameters, the number of biases, trainable weights that come from CONV and FC layers and lastly, non-trainable weights that come from BN layers. Table 3 displays the range of the parameters of the entire network and compares it to the one of the biases and the non-trainable parameters.

Network	#Bias	#Trainable	#Non-trainable	Non-trainable (%)	Total
ResNet50	27560	25 583 592	53 120	0.21	25 636 712
InceptionV3	1000	23 817 352	34 432	0.14	23 851 784
Xception	1000	22 855 952	54 528	0.24	22 910 480
MobileNet	1000	4 231 976	21 888	0.51	4 253 864
MobileNetV2	1000	3 504 872	34 112	0.96	3 538 984
VGG16	13416	138 357 544	0	0.00	138 357 544
VGG19	14696	143 667 240	0	0.00	143 667 240

Table 2: Number of parameters for each network grouped in several categories: bias, trainable weights, non-trainable weights and the total.

Network	(MIN, MAX)	Bias	Non-trainable
ResNet50	(-6.6483035; 83614.734)	(-0.024051076; 0.029003482)	(-6.6483035, 83614.734)
InceptionV3	(-11.790578; 17.908556)	(-1.0875583; 1.230254)	(1.8646851e-14, 12.138739)
Xception	(-25.056831; 2704.8052)	(-0.93594396; 0.95873684)	(-25.056831, 2704.8052)
MobileNet	(-48.049644; 418.2356)	(-2.3272734; 2.242322)	(-48.049644, 418.2356)
MobileNetV2	(-28.696346; 415.6262)	(-0.86876506; 1.2093042)	(-28.696346, 415.6262)
VGG16	(-1.0271513; 9.431553)	(-1.0271513; 9.431553)	-
VGG19	(-1.4180313; 9.611258)	(-1.4180313; 9.611258)	-

Table 3: Intervals (min, max) for the entire network and for the special parameters: bias and non-trainable weights.

The following tables show the complete benchmark result presented in Chapter 6. In addition to the accuracy Top 1 (ACC-1) and Top 5 (ACC-5), we also display the MSE between the weights in FP32 and the compressed weights and the KL divergence. In addition to these measures, we also display what we call the true distortion measure d which is the difference between the original

Acc-1 and the Acc-1 after compression. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$), respectively.

B.2 Compression on all parameters

First, Table 4 showcases the results for BF16 and MSFP8. The results are compared to the FP32 and FP16. For the majority of the networks, the original accuracy is improved when using BF16. Although MSFP8 uses fewer bits and performs acceptably in the case of VGG16, we highlight BF16 as the clear winner between the two formats.

Network	TYPE	ACC-1	ACC-5	MSE	DIV KL	True distortion d
VGG16	FP32	70.6	91.3	-	-	0.000
	FP16	70.6	91.3	1.33E-12	0	0.000
	BF16	70.8	91.2	3.57904E-10	0	-0.2
	MSFP8	69.7	90.3	3.08E-07	0.657328	0.9
VGG19	FP32	70.1	90.4	-	-	0.000
	FP16	70.1	90.4	1.30E-12	0	0.000
	BF16	70.3	90.5	3.47205E-10	0	-0.2
	MSFP8	67.9	89.4	3.06E-07	0.788984	2.2
ResNet50	FP32	75.7	93.3	-	-	0.000
	FP16	71.3	90.2	1.29E-10	0	4.400
	BF16	75.5	93.5	3.46208E-09	0	0.2
	MSFP8	62.8	83.8	3.04E-06	0.734521	12.9
InceptionV3	FP32	71.1	89.9	-	-	0.000
	FP16	71.1	89.9	6.71E-11	0	0.000
	BF16	71.3	90.0	6.39523E-09	0	-0.2
	MSFP8	44.8	67.9	5.64E-06	2.416483	26.3
Xception	FP32	73.5	92.1	-	-	0.000
	FP16	73.4	92.2	7.02E-08	0	0.100
	BF16	73.6	91.7	2.83023E-08	0	-0.1
	MSFP8	37.5	60.6	2.49E-05	2.621412	36
MobileNet	FP32	70.8	89.8	-	-	0.000
	FP16	71.1	89.8	2.15E-08	0	-0.300
	BF16	70.9	89.5	1.63E-07	0.009764	-0.100
	MSFP8	0.1	0.5	0.000144644	7.469399	70.700
MobileNetV2	FP32	71.2	90.0	-	-	0.000
	FP16	71.2	90.0	1.78E-08	0	0.000
	BF16	71.0	89.6	1.19E-07	0.003675	0.200
	MSFP8	0.2	0.6	0.000114913	5.367676	71.000

Table 4: Comparisons between FP32, FP16, BF16 and MSFP8. For each network, we highlight the rows that indicate the formats which work the best.

Table 5 displays our results on 11 image classification networks using the Posit format. We compare again our results to the FP32 and FP16. It is interesting to see that overall Posits show good

results ($d < 1$) only using 8 bits. Furthermore, this benchmarks also validates the conclusion from Chapter 5 concerning MSE and KL divergence which are not good distortion measures in our context. If we look at VGG19 and compare the results for Posit<8,1> and Posit<8,2>, we see that the MSE and KL div are smaller in the case of Posit<8,2> even though the true distortion is higher $d_{Posit<8,1>} < d_{Posit<8,2>}$. Other similar examples can be observed in all the tables presented in this section.

Network	**TYPE**	**ACC-1**	**ACC-5**	**MSE**	**DIV KL**	**True distortion d**
SqueezeNet v1.0	FP32	56.9	80.8	-	-	0.000
	FP16	56.9	80.8	1.47E-10	0	0.000
	Posit<8,0>	55.5	79.8	2.01E-05	0.071458	1.400
	Posit<8,1>	56.3	80.8	3.65E-06	0.01815	0.600
	Posit<8,2>	56.1	80.2	3.79E-06	0.018668	0.800
	Posit<8,3>	55.8	79.2	9.38E-06	0.080963	1.100
SqueezeNet v1.1	FP32	59		-	-	0.000
	FP16	59	81	1.27E-10	0	0.000
	Posit<8,0>	56.6	79.8	2.03521E-05	0.122127	2.400
	Posit<8,1>	58.4	80.7	3.40925E-06	0.034299	0.600
	Posit<8,2>	56.8	80.2	3.48166E-06	0.044015	2.200
	Posit<8,3>	54.4	78.1	8.31E-06	0.219605	4.600
AlexNet	FP32	55.6	78.9	-	-	0.000
	FP16	55.6	78.9	2.47E-12	0	0.000
	Posit<8,0>	50.6	75	1.75E-05	0.38653	5.000
	Posit<8,1>	54.7	79	4.28E-07	0.009454	0.900
	Posit<8,2>	55	78.5	1.50E-07	0.006283	0.600
	Posit<8,3>	55.2	78.2	1.77E-07	0.012769	0.400
VGG16	FP32	70.600	91.300	-	-	0.000
	FP16	70.600	91.300	1.33E-12	0	0.000
	Posit<8,0>	10.200	25.200	8.51E-06	4.293608	60.400
	Posit<8,1>	70.800	91.000	2.38E-07	0.009494	-0.200
	Posit<8,2>	70.500	91.000	8.02E-08	0.004375	0.100
	Posit<8,3>	70.000	90.700	1.09E-07	0.015521	0.600
VGG19	FP32	70.1	90.4	-	-	0.000
	FP16	70.1	90.4	1.30E-12	0	0.000
	Posit<8,0>	4.8	16.3	8.79E-06	4.87205	65.300
	Posit<8,1>	70.1	90	2.41E-07	0.008593	0.000
	Posit<8,2>	69.9	90	7.96E-08	0.00518	0.200
	Posit<8,3>	70.6	90.4	1.06E-07	0.013078	-0.500
ResNet50	FP32	75.7	93.3	-	-	0.000
	FP16	71.3	90.2	1.29E-10	0	4.400
	Posit<8,0>	0	0	644.0061646	10.78829	75.700
	Posit<8,1>	27.7	46.9	502.5577087	4.274016	48
	Posit<8,2>	73.2	91.4	29.99231911	0.159246	2.5
	Posit<8,3>	66	88.7	17.84280396	0.356127	9.700
	FP32	68	88.6	-	-	0.000
	FP16	68	88.6		0	0.000

GoogleNet	Posit<8,0>	66.9	87.7	2.08E-05	0.048339	1.100
	Posit<8,1>	68.1	88.5	2.33E-06	0.006361	-0.100
	Posit<8,2>	67.8	88.4	2.51E-06	0.00962	0.200
	Posit<8,3>	68	88.2	5.92E-06	0.029825	0.000
InceptionV3	FP32	71.1	89.9	-	-	0.000
	FP16	71.1	89.9	6.71E-11	0	0.000
	Posit<8,0>	65.1	86.1	2.07E-05	0.802937	6
	Posit<8,1>	69.4	91	1.87E-06	0.176595	1.700
	Posit<8,2>	69.7	89.5	2.01E-06	0.259748	1.400
	Posit<8,3>	63.1	85.3	4.49E-06	0.85415	8.000
Xception	FP32	73.5	92.1	-	-	0.000
	FP16	73.4	92.2	7.02E-08	0	0.100
	Posit<8,0>	70.6	90.9	0.793459237	0.209779	2.9
	Posit<8,1>	72.4	91.4	0.161132693	0.067384	1.100
	Posit<8,2>	72.1	90.9	0.011052876	0.128806	1.400
	Posit<8,3>	63.8	86	0.011075924	0.772246	9.700
MobileNet	FP32	70.8	89.8	-	-	0.000
	FP16	71.1	89.8	2.15E-08	0	-0.300
	Posit<8,0>	25.3	47	2.67E-05	3.330651	45.500
	Posit<8,1>	53.5	76.9	8.76702E-06	1.149143	17.3
	Posit<8,2>	52.7	77.3	1.31647E-05	1.124676	18.1
	Posit<8,3>	39.4	63.1	4.19E-05	2.114368	31.400
MobileNetV2	FP32	71.2	90	-	-	0.000
	FP16	71.2	90	1.78E-08	0	0.000
	Posit<8,0>	12.7	24.7	3.03E-05	3.958292	58.500
	Posit<8,1>	12.3	25.7	7.84E-06	1.149143	58.900
	Posit<8,2>	11	24.4	1.17E-05	1.17E-05	60.200
	Posit<8,3>	3.2	9.9	3.31E-05	5.286118	68.000

Table 5: Results for Posits. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$).

B.3 Compression of trainable parameters (without BN parameters)

Table 6 shows the results when compression is applied to parameters from CONV and FC layers. Parameters from BN layers are kept in their original form. As shown in Tables 2, they do not represent a high percentage of the number of parameters, but adding compression error to these parameters can change the whole performance of the networks.

Network	TYPE	ACC-1	ACC-5	MSE	DIV KL	True distortion d
ResNet50	FP32	75.7	93.3	-	-	0.000
	FP16	75.7	93.3	1.36E-11	0	0.000
	Posit<8,0>	71.3	9.8	2.02E-05	0.209466	4.400
	Posit<8,1>	75	92.7	1.05E-06	0.018011	0.700
	Posit<8,2>	75	92.8	6.99E-07	0.025143	0.700
	Posit<8,3>	73.6	92.6	8.87E-07	0.07765	2.100
InceptionV3	FP32	71.1	89.9	-	-	0.000
	FP16	71.1	89.9	6.71E-11	0	0.000
	Posit<8,0>	66	86.8	2.03E-05	0.698037	5.100
	Posit<8,1>	70.9	90.7	1.47E-06	0.123788	0.200
	Posit<8,2>	70.1	89.1	1.17E-06	0.218421	1.000
	Posit<8,3>	69.9	88.5	1.63E-06	0.366651	1.200
Xception	FP32	73.5	92.1	-	-	0.000
	FP16	73.4	92.2	7.02E-08	0	0.100
	Posit<8,0>	72.1	91.3	2.02E-05	0.06861	1.400
	Posit<8,1>	72.6	91.7	3.21E-06	0.022671	0.900
	Posit<8,2>	72.8	91.3	3.20E-06	0.050335	0.700
	Posit<8,3>	68.8	89.4	7.21E-06	0.380569	4.700
MobileNet	FP32	70.8	89.8	-	-	0.000
	FP16	71.1	89.8	6.40E-10	0	-0.300
	Posit<8,0>	25.3	47	2.67E-05	3.330651	45.500
	Posit<8,1>	53.5	76.9	8.76702E-06	1.149143	17.300
	Posit<8,2>	52.7	77.3	1.31647E-05	1.124676	18.100
	Posit<8,3>	39.4	63.1	4.19106E-05	2.114368	31.400
MobileNetV2	FP32	71.2	90	-	-	0.000
	FP16	71.2	90	4.74E-10	0	0.000
	Posit<8,0>	12.7	24.7	3.03E-05	3.958292	58.500
	Posit<8,1>	12.3	25.7	7.84E-06	4.033997	58.900
	Posit<8,2>	11	24.4	1.17E-05	4.246772	60.200
	Posit<8,3>	3.2	9.9	3.31E-05	5.286118	68.000

Table 6: Results for Posits when we exclude the BN parameters from the conversion. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$).

B.4 Compression without biases

Table 7 displays the results in the case where we do not compress the biases. These results show that even though a bias value may be critical for successful classification because it allows you to shift the activation function to the left or right, compressing them or not does not have much impact on the accuracy of the networks.

Network	TYPE	ACC-1	ACC-5	MSE	Div KL	True distortion d
VGG16	FP32	70.600	91.300	-	-	0.000
	FP16	70.600	91.300	1.33E-12	0	0.000
	Posit<8,0>	10.3	25.3	8.51E-06	4.293608406	60.300
	Posit<8,1>	70.8	91	2.34E-07	0.009494399	-0.200
	Posit<8,2>	70.5	91	7.07E-08	0.004375368	0.100
	Posit<8,3>	70	90.6	7.47E-08	0.015520616	0.600
VGG19	FP32	70.1	90.4	-	-	0.000
	FP16	70.1	90.4	1.30E-12	0	0.000
	Posit<8,0>	5	16.5	8.79E-06	4.87205	65.100
	Posit<8,1>	69.9	90.1	2.38E-07	0.008593	0.200
	Posit<8,2>	69.9	90	7.10E-08	0.00518	0.200
	Posit<8,3>	70.6	90.5	7.42E-08	0.013078	-0.500
ResNet50	FP32	75.7	93.3	-	-	0.000
	FP16	71.3	90.2	1.29E-10	0	4.400
	Posit<8,0>	0	0.5	644.0061646	10.788291	75.700
	Posit<8,1>	27.7	46.8	502.5577087	4.274016	48
	Posit<8,2>	73.2	91.4	29.99231911	0.159246	2.5
	Posit<8,3>	66	88.7	17.84280396	0.3561275	9.700
InceptionV3	FP32	71.1	89.9	-	-	0.000
	FP16	71.1	89.9	6.71E-11	0.000931	0.000
	Posit<8,0>	65.1	86.1	2.07E-05	0.802937	6
	Posit<8,1>	69.5	90.9	1.87E-06	0.176594	1.600
	Posit<8,2>	69.7	89.5	2.00E-06	0.259747	1.400
	Posit<8,3>	63.1	85.3	4.48E-06	0.854149	8.000
Xception	FP32	73.5	92.1	-	-	0.000
	FP16	73.4	92.2	7.02E-08	0	0.100
	Posit<8,0>	70.9	90.4	0.793459237	0.209778	2.6
	Posit<8,1>	72.4	91.4	0.161132693	0.067383	1.100
	Posit<8,2>	72.1	90.8	0.011052873	0.128806	1.400
	Posit<8,3>	63.8	86	0.011075915	0.772246	9.700
MobileNet	FP32	70.8	89.8	-	-	0.000
	FP16	71.1	89.8	2.15E-08	0	-0.300
	Posit<8,0>	14.5	29.9	0.16864796	3.330651	56.3
	Posit<8,1>	48	72.9	0.0097967	1.149143	22.8
	Posit<8,2>	39.5	63.2	0.001612978	1.124676	31.3
	Posit<8,3>	3.5	10.5	0.001786359	2.114367	67.3
MobileNetV2	FP32	71.2	90	-	-	0.000
	FP16	71.2	90	1.78E-08	0	0.000
	Posit<8,0>	4	10.4	0.075218514	3.958291	67.2
	Posit<8,1>	16.3	31.6	0.004354046	1.149143	54.900
	Posit<8,2>	11.1	25.5	0.001311715	4.000217	60.1
	Posit<8,3>	1.1	3.9	0.001561229	5.286117	70.1

Table 7: Results for Posits when we exclude the biases from the conversion. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$).

B.5 Compression without biases and BN parameters

Finally, Table 8 presents the results when we compress the parameters without compressing the BN parameters and the biases. The results show that this doesn't improve the accuracy.

Network	TYPE	ACC-1	ACC-5	MSE	DIV KL	True distortion d
VGG16	FP32	70.600	91.300	-	-	0.000
	FP16	70.600	91.300	1.33E-12	0	0.000
	Posit<8,0>	10.3	25.3	8.51E-06	4.289988	60.300
	Posit<8,1>	70.8	91	2.34E-07	0.009038	-0.200
	Posit<8,2>	70.5	91	7.07E-08	0.004146	0.100
	Posit<8,3>	70	90.6	7.47E-08	0.015904	0.600
VGG19	FP32	70.1	90.4	-	-	0.000
	FP16	70.1	90.4	1.30E-12	0	0.000
	Posit<8,0>	5	16.5	8.79E-06	4.860992	65.100
	Posit<8,1>	69.9	90.1	2.38E-07	0.009127	0.200
	Posit<8,2>	69.9	90	7.10E-08	0.005488	0.200
	Posit<8,3>	70.6	90.5	7.42E-08	0.013541	-0.500
ResNet50	FP32	75.7	93.3	-	-	0.000
	FP16	71.3	90.2	1.29E-10	0	4.400
	Posit<8,0>	71.5	90.8	2.05E-05	0.218295	4.200
	Posit<8,1>	74.8	93.3	1.53E-06	0.026634	0.9
	Posit<8,2>	75	92.5	2.58E-06	0.053205	0.7
	Posit<8,3>	70.8	90.3	8.24E-06	0.220395	4.900
InceptionV3	FP32	71.1	89.9	-	-	0.000
	FP16	71.1	89.9	6.71E-11	0	0.000
	Posit<8,0>	65.8	86.7	2.06E-05	0.704354	5.3
	Posit<8,1>	69.5	90.7	1.79E-06	0.174118	1.600
	Posit<8,2>	68.7	89.6	1.85E-06	0.241575	2.400
	Posit<8,3>	64.6	85.8	3.77E-06	0.711136	6.500
Xception	FP32	73.5	92.1	-	-	0.000
	FP16	73.4	92.2	7.02E-08	0	0.100
	Posit<8,0>	72.2	91.2	2.03E-05	2.03E-05	1.3
	Posit<8,1>	72.9	91.2	3.31E-06	3.31E-06	0.600
	Posit<8,2>	72.5	91.2	3.60E-06	3.60E-06	1.000
	Posit<8,3>	69.5	89.2	8.77E-06	0.354522	4.000
	FP32	70.8	89.8	-	-	0.000
	FP16	71.1	89.8	2.15E-08	0	-0.300

MobileNet	Posit<8,0>	23.8	44.2	3.79E-05	3.567855	47.000
	Posit<8,1>	52.6	77.2	2.04E-05	1.183558	18.2
	Posit<8,2>	47.9	71.6	2.85E-05	1.596312	22.9
	Posit<8,3>	10.5	24.2	9.89E-05	5.332177	60.300
MobileNetV2	FP32	71.2	90	-	-	0.000
	FP16	71.2	90	1.78E-08	0	0.000
	Posit<8,0>	9.6	20.7	5.14E-05	4.394608	61.600
	Posit<8,1>	15.3	32.4	2.96E-05	3.607061	55.900
	Posit<8,2>	11.6	26.1	3.93E-05	4.000217	59.600
	Posit<8,3>	1.4	4.3	0.000136137	5.687109	69.800

Table 8: Results for Posits when we exclude the BN parameters and the biases from the conversion. For each network, we highlight the rows that indicate the formats which work the best. We use green, yellow or red to indicate if the compression performs well ($d < 1$), average ($1 \leq d < 5$) or bad ($d > 5$).

This section provides additional information on the tested networks and the full benchmark results. For each network, we clearly highlight which small floating-point format performs better. Furthermore, we add two other metrics (MSE and KL Divergence) to point out that they do not work properly as distortion measures.

C Chapter 7: Upper bound approach

Numerous methods to quantize the weights with a single bit or more have been proposed. However, the loss of accuracy involved in the compression is scarcely studied from a theoretical point of view. In this section, we propose a new distortion measure which assesses the gap between the Bayes risk of a classifier before and after the compression. Since this distortion is not easily tractable, we derive a theoretical approximation when the last fully connected layer of a deep neural network is compressed under the assumption that the layer inputs follow a multivariate normal distribution. Numerical results show that the approximation performs well on both synthetic and real data. We also show that heuristic quantizers proposed in the literature may not be optimal.

The method presented in this section is different from the one presented in Chapter 7. Here, we do not make assumptions on the way the weights were compressed which leads to using an upper bound to derive the approximation.

C.1 Upper bound for distortion measure

We briefly recall useful definitions and notations.

Let us consider a deep neural network $f(\mathbf{x})$ composed of $K+1$ layers with $\mathbf{f}_0$ being the input layer $\mathbf{f}_0(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^{n_0}$ where $n_0 = n$. The hidden layers are from $\mathbf{f}_1$ to $\mathbf{f}_{K-1}$ and the output layer

is denoted with $\mathbf{f}_K$. We define the model as follows

$$\mathbf{f}_k(\mathbf{x}) = \sigma(\mathbf{W}_k \mathbf{f}_{k-1}(\mathbf{x}) + \mathbf{b}_k),\ 1 \leq k < K, \tag{C.2}$$
$$\hat{\mathbf{y}} = f(\mathbf{x}) = \mathbf{f}_K(\mathbf{x}) = \text{softmax}(\mathbf{W}\mathbf{f}_{K-1}(\mathbf{x}){+}\mathbf{b}), \tag{C.3}$$

where $\mathbf{W}_k \in \mathbb{R}^{n_k \times n_{k-1}}$, $\mathbf{b}_k \in \mathbb{R}^{n_k}$ and $\sigma(\cdot)$ is a nonlinear activation function (typically, the ReLU function). The last layer, called the linear softmax layer, depends on $\mathbf{W} \in \mathbb{R}^{2 \times n_{K-1}}$ and $\mathbf{b} \in \mathbb{R}^2$. The output of the neural network $\hat{\mathbf{y}} = (\hat{y}_0, \hat{y}_1)$ is interpreted as a soft one-hot encoding vector. The decision rule, denoted $\delta_f(\mathbf{x})$, is

$$\delta_f(\mathbf{x}) = \delta(f(\mathbf{x})) = \delta(\hat{\mathbf{y}}), \tag{C.4}$$

and chooses the most probable component of $\hat{\mathbf{y}}$. We can rewrite $\delta_{f_\mathbf{W}}$ as a linear classifier without the operators argmax and softmax. The decision rule C.4 is equivalent to the linear decision rule:

$$\delta_{f_\mathbf{W}}(\mathbf{f}_{K-1}) = \begin{cases} 0 & \text{if } \tilde{\mathbf{w}}^T \mathbf{f}_{K-1} > \lambda, \\ 1 & \text{otherwise,} \end{cases} \tag{C.5}$$

where $\tilde{\mathbf{w}} = \mathbf{w}_0 - \mathbf{w}_1$, $\lambda = b_1 - b_0$ and $\tilde{\mathbf{w}}^T$ denotes the transpose of $\tilde{\mathbf{w}}$. Note that $\mathbf{w}_0$ and $\mathbf{w}_1$ represent the first and the second row of $\mathbf{W}$ and b_0, b_1 are the two components of the bias vector $\mathbf{b}$. To simplify the notations, the vector $\mathbf{f}_{K-1}$ will be denoted $\mathbf{f}$ in the rest of the section. We consider that the vector $\mathbf{f}$ follows a multivariate normal distribution. In the case of two classes, we assume that

$$\mathbf{f} \sim \mathcal{N}(\mu_i, \Sigma_i) \ \text{ under } \ C_i, \tag{C.6}$$

where μ_j is a known mean vector and Σ_j is a known strictly positive definitive covariance matrix. Let us define the conditional gap

$$d_i(\mathbf{w}, \hat{\mathbf{w}}) = \mid \mathbb{P}_i(f_\mathbf{w}(\mathbf{x}){>}\lambda){-}\mathbb{P}_i(f_{\hat{\mathbf{w}}}(\mathbf{x}){>}\lambda) \mid \tag{C.7}$$

as the gap between the probability errors conditioned by the class C_i. It follows that

$$d_i(\mathbf{w}, \hat{\mathbf{w}}) \leq \pi_0 d_0(\mathbf{w}, \hat{\mathbf{w}}) + \pi_1 d_1(\mathbf{w}, \hat{\mathbf{w}}). \tag{C.8}$$

Using [Lin and Bai, 2011, Chap. 1, inequality 1.3.c], we get that

$$d_i(\mathbf{w}, \hat{\mathbf{w}}) \leq \mathbb{P}_i(f_\mathbf{w}(\mathbf{x}){>}\lambda, f_{\hat{\mathbf{w}}}(\mathbf{x}){\leq}\lambda) + \mathbb{P}_i(f_\mathbf{w}(\mathbf{x}){\leq}\lambda, f_{\hat{\mathbf{w}}}(\mathbf{x}){>}\lambda). \tag{C.9}$$

The conditional gap is bounded by the sum of the probabilities when the classifiers disagree. It is equivalent to

$$d_i(\mathbf{w}, \hat{\mathbf{w}}) \leq \mathbb{P}_i(\mathbf{w}^T\mathbf{f}{>}\lambda, \hat{\mathbf{w}}^T\mathbf{f}{\leq}\lambda) + \mathbb{P}_i(\mathbf{w}^T\mathbf{f}{\leq}\lambda, \hat{\mathbf{w}}^T\mathbf{f}{>}\lambda). \tag{C.10}$$

This form is simpler because it involves the couple of variables $(\mathbf{w}^T\mathbf{f}, \hat{\mathbf{w}}^T\mathbf{f})$ that follows a bivariate normal distribution with a non-zero correlation coefficient. The distribution of this random couple is studied in the following lemma.

Lemme C.1. *Let $i \in \{0,1\}$. Then, we have the equalities*

$$\mathbb{P}_i(\mathbf{w}^T\mathbf{f} > \lambda, \hat{\mathbf{w}}^T\mathbf{f} \leq \lambda) = \mathbb{P}_i(X > a_i(\mathbf{w}), Y \leq a_i(\hat{\mathbf{w}})), \tag{C.11}$$
$$\mathbb{P}_i(\mathbf{w}^T\mathbf{f} \leq \lambda, \hat{\mathbf{w}}^T\mathbf{f} > \lambda) = \mathbb{P}_i(X \leq a_i(\mathbf{w}), Y > a_i(\hat{\mathbf{w}})), \tag{C.12}$$

where X and Y denote two standard normal variables with correlation coefficient ϱ_i such as

$$a_i(\mathbf{w}) = \frac{\lambda - \mathbf{w}^T\mu_i}{\sqrt{\mathbf{w}^T\Sigma_i\mathbf{w}}}, \quad a_i(\hat{\mathbf{w}}) = \frac{\lambda - \hat{\mathbf{w}}^T\mu_i}{\sqrt{\hat{\mathbf{w}}^T\Sigma_i\hat{\mathbf{w}}}}, \tag{C.13}$$

$$\varrho_i = \varrho(i, \mathbf{w}, \hat{\mathbf{w}}) = \frac{\mathbf{w}^T\Sigma_i\hat{\mathbf{w}}}{\sqrt{\mathbf{w}^T\Sigma_i\mathbf{w}}\sqrt{\hat{\mathbf{w}}^T\Sigma_i\hat{\mathbf{w}}}}. \tag{C.14}$$

Proof.

We normalize the component $\mathbf{w}^T\mathbf{f}$ by removing its mean and dividing it by its standard deviation given in (7.5). We do the same for the component $\hat{\mathbf{w}}^T\mathbf{f}$. A short calculation yields the correlation coefficient.

□

It is well known that the bivariate normal distribution is not easy to compute except for some very specific cases. Fortunately, some accurate approximations exist. In this chapter, we use the simple approximation given in [Cox and Wermuth, 1991] which is easy to interpret. By applying this approximation to (C.11), we get

$$\mathbb{P}_i(X > a_i(\mathbf{w}), Y \leq a_i(\hat{\mathbf{w}})) \approx \Phi(-a_i(\mathbf{w}))\Phi\left(-\xi_{i,\mathbf{w},\hat{\mathbf{w}}}\right), \tag{C.15}$$

$$\xi_{i,\mathbf{w},\hat{\mathbf{w}}} = \frac{\varrho_i\, \mu(a_i(\mathbf{w})) - a_i(\hat{\mathbf{w}})}{\sqrt{1-\varrho_i^2}}, \quad \mu(a_i(\mathbf{w})) = \frac{\phi(a_i(\mathbf{w}))}{\Phi(-a_i(\mathbf{w}))}. \tag{C.16}$$

This approximation expresses the probability as a product of two simple terms. The first term depends only on $\mathbf{w}$ and is thus independent from the compression. The second term quantifies the dependency between $\mathbf{w}$ and its compressed form $\hat{\mathbf{w}}$ through $\xi_{i,\mathbf{w},\hat{\mathbf{w}}}$. We can do the same for the second inequality (C.12) in Lemma C.1. Finally, we get an approximation $D_i(\mathbf{w}, \hat{\mathbf{w}})$ of the upper bound $d_i(\mathbf{w}, \hat{\mathbf{w}})$ in (C.10):

$$\begin{aligned} d_i(\mathbf{w}, \hat{\mathbf{w}}) &\leq \mathbb{P}_i(\mathbf{w}^T\mathbf{f} > \lambda, \hat{\mathbf{w}}^T\mathbf{f} \leq \lambda) + \mathbb{P}_i(\mathbf{w}^T\mathbf{f} \leq \lambda, \hat{\mathbf{w}}^T\mathbf{f} > \lambda) \\ &\approx \Phi(-a_i(\mathbf{w}))\Phi\left(-\xi_{i,\mathbf{w},\hat{\mathbf{w}}}\right) + \Phi(-a_i(\hat{\mathbf{w}}))\Phi\left(-\xi_{i,\hat{\mathbf{w}},\mathbf{w}}\right) \\ &= D_i(\mathbf{w}, \hat{\mathbf{w}}), \end{aligned} \tag{C.17}$$

where $\xi_{i,\hat{\mathbf{w}},\mathbf{w}}$ is similar to $\xi_{i,\mathbf{w},\hat{\mathbf{w}}}$ provided that we swap the role of $\mathbf{w}$ and $\hat{\mathbf{w}}$. Therefore, we get the following approximation $D(\mathbf{w}, \hat{\mathbf{w}})$ of $d(\mathbf{w}, \hat{\mathbf{w}})$:

$$d(\mathbf{w}, \hat{\mathbf{w}}) \approx \pi_0 D_0(\mathbf{w}, \hat{\mathbf{w}}) + \pi_1 D_1(\mathbf{w}, \hat{\mathbf{w}}) = D(\mathbf{w}, \hat{\mathbf{w}}). \tag{C.18}$$

This approximation is a closed form expression. Even if we cannot ensure that $D(\mathbf{w}, \hat{\mathbf{w}})$ is truly an upper bound, the advantage of using the approximation $D(\mathbf{w}, \hat{\mathbf{w}})$ over the true value $d(\mathbf{w}, \hat{\mathbf{w}})$ is to ease the interpretation of the effects of compression over the accuracy.
By analyzing the expression of the approximation, one can note that its value mainly relies on

three quantities: $a_i(\mathbf{w})$ in (C.13), $a_i(\hat{\mathbf{w}})$ in (C.13) and the correlation ϱ_i between the compressed and uncompressed weights in (C.14). The constant $a_i(\mathbf{w})$ depends on properties of the dataset (the means of the classes) and not on the compressed network architecture. The value $a_i(\hat{\mathbf{w}})$ depends on the compressed network. Under some appropriate assumptions on the number of neurons and the compression bit-rate, $a_i(\hat{\mathbf{w}})$ can be approximated by $a_i(\mathbf{w})$ weighted by a corrective term depending on ϱ_i. Under the same assumptions, the correlation ϱ_i can be approximated analytically as a function of the number of neurons of a layer and the compression bit-rate. Further details and a more in-depth analysis of these approximations in the case of uniform quantization are presented in Chapter 7.

C.2 Experiments for the upper bound

Several experiments were carried out in order to analyze the proposed approximation $D(\mathbf{w}, \hat{\mathbf{w}})$ in (C.18). The first experiment was done using a softmax classifier on synthetic data, while the second one was performed on a one-hidden-layer network trained on the Sonar dataset.

We used the standard binary and uniform quantization to compress the weights $\mathbf{w}$ into $\hat{\mathbf{w}}_s$ where $\hat{\mathbf{w}}_s$ underlines that the compressed weights $\hat{\mathbf{w}}$ depend on a scaling factor $s > 0$ to tune the quantizer. The binarization process produces

$$\hat{\mathbf{w}}_s = s \cdot \mathrm{sign}(\mathbf{w}), \tag{C.19}$$

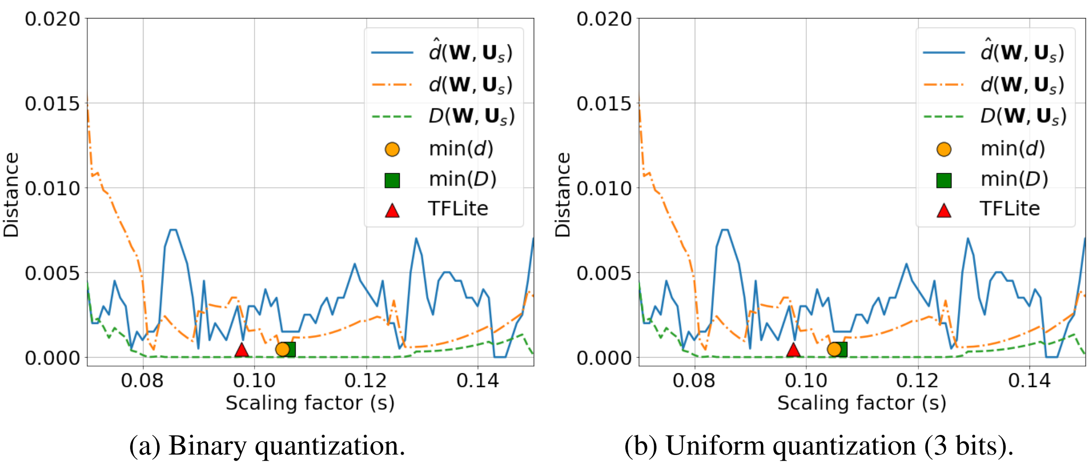
where $\mathrm{sign}(\mathbf{w})$ means that the element-wise sign function is applied to each element of $\mathbf{w}$. The uniform quantization produces

$$\hat{\mathbf{w}}_s = \left\lfloor \frac{\mathbf{w}}{s} \right\rceil, \tag{C.20}$$

where the element-wise $\lfloor \cdot \rceil$ operation approximates its input with the closest integer.

We are looking for the best scaling factor s that minimizes $d(\mathbf{w}, \hat{\mathbf{w}}_s)$. For both experimental settings, we iterated over s from 0 to 2 by a step of 10^{-3}. At each iteration, we compressed the weights with the methods mentioned above. When we use the synthetic data, we computed the theoretical distortion $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ and its approximation $D(\mathbf{w}, \hat{\mathbf{w}}_s)$. For the real dataset, we cannot compute the theoretical distortion because we do not know the exact parameters of the assumed normal distribution. We estimated the values of μ_0, μ_1, Σ_0 and Σ_1 in (C.6) from the samples. Then, we computed $\tilde{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $\tilde{D}(\mathbf{w}, \hat{\mathbf{w}}_s)$ by replacing the true values μ_0, μ_1, Σ_0 and Σ_1 by their estimates in the definition of $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $D(\mathbf{w}, \hat{\mathbf{w}}_s)$. In both datasets, we evaluated the empirical distortion $\hat{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$ by computing the empirical Bayes risks. The minimum of $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ with respect to s is denoted min(d). We use the same notation min($\cdot$) for the other distortions. We also take a look at the true Bayes risk $r(\delta_{f_{\hat{\mathbf{w}}_s}})$ for the synthetic data and at the empirical risk $\hat{r}(\delta_{f_{\hat{\mathbf{w}}_s}})$ for the real data.

Additionally, we compared our results to two state-of-the-art methods that have shown promising results, namely XNOR-NET [Rastegari et al., 2016] for binary quantization and Tensorflow Lite [Jacob et al., 2018, Krishnamoorthi, 2018] for uniform quantization:



(a) Binary quantization.

(b) Uniform quantization (3 bits).

Figure C.4: Distortions $\hat{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$, $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $D(\mathbf{w}, \hat{\mathbf{w}}_s)$ for synthetic data experiment with XNOR-NET (left) and TFLite (right) as a function of the scaling factor. The orange circle and the green square represent respectively the minimum of $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $D(\mathbf{w}, \hat{\mathbf{w}}_s)$ and, in red, we show the scaling factors of XNOR-NET and TFLite.

XNOR-NET. XNOR-NET quantizes $\mathbf{w}$ by solving the optimization problem $\min ||\mathbf{w} - s\mathbf{w}_b||_2^2$ where $\mathbf{w}_b$ is a binary matrix and $s > 0$. The optimal solution $\hat{\mathbf{w}}_{s_{\text{XNOR-NET}}}$ is the product of the optimal binary matrix $\mathbf{w}_b^* = \text{sign}(\mathbf{w})$ with the optimal scaling factor, $s_{\text{XNOR-NET}} = ||\mathbf{w}||_1/n$ where $||\mathbf{w}||_p$ is the p-norm.

TFLite. The TFLite approach is based on the standard uniform scalar quantizer. The quantization of an entry w of $\mathbf{w}$ into a quantized value w of $\hat{\mathbf{w}}_{s_{\text{TFLITE}}}$ proceeds as follows: $w = \left\lfloor \frac{w}{s_{\text{TFLITE}}} \right\rceil + z$, where the scaling factor is defined as $s_{\text{TFLITE}} = (w_{\text{max}} - w_{\text{min}})/N$ with $N = 2^R - 1$ and the parameter z represents the quantized value of the real value 0. This method does not involve any optimization.

C.3 Softmax classifier on synthetic data

The experiments in this subsection were performed on synthetic data by employing a binary softmax classifier without any hidden layers. In order to train our model, we generated a two-class dataset with $N = 2 \times 10^3$ samples, 10^3 samples per class. Each sample of the generated data has $n = 10$ features and was drawn from a multivariate normal distribution with a given mean and covariance for each class. Following [Muller, 1959], means were drawn from a 10 dimensional sphere with radius of 1 for C_0 and 5 for C_1. The variance is considered spherical with the intensity 4 for C_0 and 2.25 for C_1.

The training leads to a Bayes risk $r(\delta_{f_{\mathbf{w}}}) = 0.1152$. We quantized the weights of the trained model using XNOR-NET (binary weights) and TFLite with only 3 bits due to the small number of weights the model contains (20 values).

Fig. C.4, on the left, shows the results with the binarization (XNOR-NET) and, on the right, the results for the uniform quantization (TFLite). It must be noted that $d(\mathbf{w}, \hat{\mathbf{w}}_s)$ overlaps with $\hat{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$. We observe that $D(\mathbf{w}, \hat{\mathbf{w}}_s)$ follows the same shape as the actual error. Our approximation is able to outperform XNOR-NET in terms of the obtained optimal scaling factor: $s_D = 0.267$ is reasonably close to the theoretical minimum $s_d = 0.285$ and better than $s_{\text{XNOR-NET}} = 0.139$. The Bayes risks for each scaling factor are the following: $r(\delta_{f_{\hat{\mathbf{w}}_{s_d}}}) = 0.1751$, $r(\delta_{f_{\hat{\mathbf{w}}_{s_D}}}) = 0.1752$ and $r(\delta_{f_{\hat{\mathbf{w}}_{s_{\text{XNOR-NET}}}}}) = 0.2255$. On the other hand, TFLite proposes a scaling factor that is close to the optimal one, but our approximation is closer: $s_{\text{TFLITE}} = 0.0976$, $s_D = 0.106$ and $s_d = 0.105$. By looking at the risk, we see that our approximation is extremely close to the theoretical minimum giving a risk of 0.1153 and better than TFLite $r(\delta_{f_{\hat{\mathbf{w}}_{s_{\text{TFLITE}}}}}) = 0.1175$.

C.4 One-hidden-layer ReLU neural network on Sonar

We also performed experiments on a one-hidden-layer neural network trained on the Sonar dataset [Gorman and Sejnowski, 1988]. It is composed of $N = 208$ instances, $n = 60$ attributes and two classes. The network we trained had one fully connected ReLU layer with 60 neurons and a final softmax layer. After training, we obtained an empirical risk $\hat{r}(\delta_{f_{\mathbf{w}}}) = 0.0727$. We quantized the weights of the last layer using the same methods as in the previous subsection, except, for the uniform quantization where we used 8 bits.

In Fig. C.5, on the left, we present the results obtained using binarization and, on the right, the results with uniform quantization. Although the 1-bit quantization performs slightly worse than the 8-bit quantization, both quantizations perform well. We observe that the XNOR-NET scaling factor $s_{\text{XNOR-NET}} = 0.4432$ is far from the estimated theoretical minimum $s_{\hat{d}} = 0.3062$, while our approximation is closer $s_{\hat{D}} = 0.2522$. XNOR-NET has a higher Bayes risk $\hat{r}(\delta_{f_{\hat{\mathbf{w}}_{s_{\text{XNOR-NET}}}}}) = 0.0913$. Our approximation gives the same empirical risk as the one obtained with the estimated theoretical $\hat{r}(\delta_{f_{\hat{\mathbf{w}}_{s_{\hat{D}}}}}) = \hat{r}(\delta_{f_{\hat{\mathbf{w}}_{s_{\hat{d}}}}}) = 0.0865$, which are closer to the original risk. Using the uniform quantization, we observe that our approximation with $s_{\hat{D}} = 0.0105$ is almost the same as TFLite $s_{\text{TFLITE}} = 0.0107$, both close to $s_{\hat{d}} = 0.0111$. The empirical risk values are all three at 0.0721. Although the normal assumption is not perfectly satisfied in the last layer (because of the ReLU), it is worth noting that our approximation still performs well.

D Chapter 7: Proofs of theorems

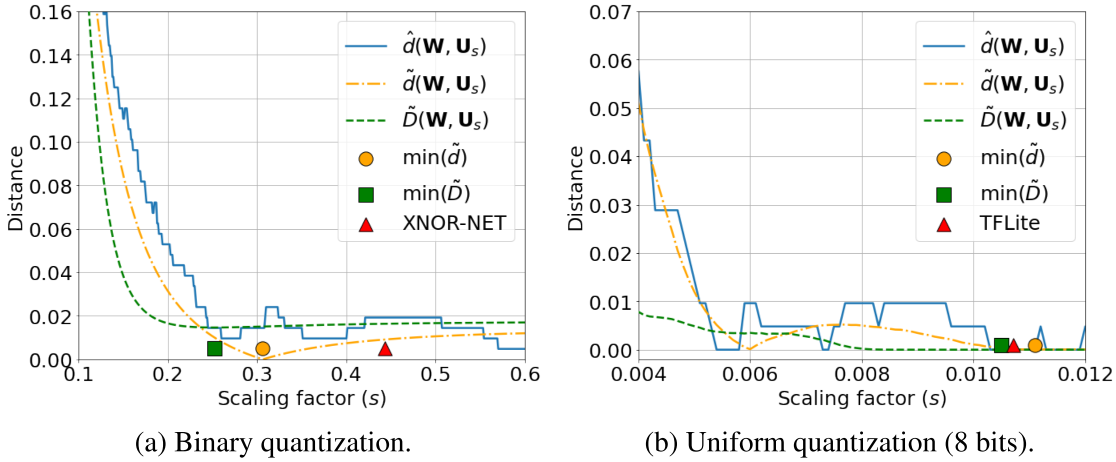
D.1 Proof theorem 7.4.1: approximation of $a_j(\hat{\mathbf{w}})$

The coefficient $a_j(\hat{\mathbf{w}})$ is

$$a_j(\hat{\mathbf{w}}) = \frac{\lambda - (\mathbf{w} + \Delta)^T \mu_j}{\sqrt{(\mathbf{w} + \Delta)^T (w + \Delta)}}, \tag{D.21}$$

where λ is a constant and μ_j is the mean of the class j.
We recall that to quantize the weights, we use uniforme quantization with a quantization step computed as follows:

$$q = \frac{\max_{1 \le i \le n} w_i - \min_{1 \le i \le n} w_i}{2^R}. \tag{D.22}$$



(a) Binary quantization. (b) Uniform quantization (8 bits).

Figure C.5: Distortions $\hat{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$, $\tilde{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $\tilde{D}(\mathbf{w}, \hat{\mathbf{w}}_s)$ for Sonar dataset with XNOR-NET (left) and TFLite (right) as a function of the scaling factor. The orange circle and the green square represent the minimum of $\tilde{d}(\mathbf{w}, \hat{\mathbf{w}}_s)$ and $\tilde{D}(\mathbf{w}, \hat{\mathbf{w}}_s)$ and the red triangles show the scaling factors of XNOR-NET and TFLite.

Approximation of the denominator

Let us denote the denominator with

$$A = (w + \Delta)^T(w + \Delta) = w^T w + 2w^T\Delta + \Delta^T\Delta. \tag{D.23}$$

Using the uniform distribution to model the components of Δ, we find

$$\mathbb{E}[A] = \|\mathbf{w}\|_2^2 + \frac{nq^2}{12}, \tag{D.24}$$

and

$$\begin{aligned}
\mathrm{var}[A] = \mathrm{var}[2w^T\Delta + \Delta^T\Delta] &= \mathbb{E}[(2w^T\Delta + \Delta^T\Delta)^2] - \mathbb{E}[2w^T\Delta + \Delta^T\Delta]^2 && \text{(D.25)}\\
&= 4\mathbb{E}[(w^T\Delta)^2] + 4\mathbb{E}[(w^T\Delta)(\Delta^T\Delta)] + \mathbb{E}[(\Delta^T\Delta)^2] - \mathbb{E}[\Delta^T\Delta]^2 && \text{(D.26)}\\
&= 4\|\mathbf{w}\|_2^2\frac{q^2}{12} + \frac{4}{5}\frac{q^4}{144}n + \frac{q^4}{144}n^2 - \frac{q^4}{144}n^2 && \text{(D.27)}\\
&= 4\|\mathbf{w}\|_2^2\frac{q^2}{12} + \frac{q^4}{144}\left(\frac{4}{5}n + n^2\right) - \frac{q^4}{144}n^2 && \text{(D.28)}\\
&= 4\|\mathbf{w}\|_2^2\frac{q^2}{12} + \frac{4}{5}\frac{q^4}{144}n. && \text{(D.29)}
\end{aligned}$$

Finally, we have

$$Y = 1/\sqrt{\left(\|\mathbf{w}\|_2^2 + \frac{nq^2}{12}\right)\left(1 + \frac{2w^T\Delta + \Delta^T\Delta - \frac{nq^2}{12}}{\|\mathbf{w}\|_2^2 + \frac{nq^2}{12}}\right)} \tag{D.30}$$

$$= 1/\sqrt{\left(\|\mathbf{w}\|_2^2 + \frac{nq^2}{12}\right)\left(1 + \frac{2w^T\Delta}{\|\mathbf{w}\|_2^2 + \frac{nq^2}{12}} + \frac{\Delta^T\Delta - \frac{nq^2}{12}}{\|\mathbf{w}\|_2^2 + \frac{nq^2}{12}}\right)} \tag{D.31}$$

$$= 1/\sqrt{\|\mathbf{w}\|_2^2\left(1 + \frac{nq^2}{12\|\mathbf{w}\|_2^2}\right)(1 + Y_1 + Y_2)}. \tag{D.32}$$

It is clear that the expectations of Y_1 and Y_2 are

$$\mathbb{E}[Y_1] = \mathbb{E}[Y_2] = 0, \tag{D.33}$$

and the variances are:

$$\mathrm{var}[Y_1] = \frac{4\|\mathbf{w}\|_2^2\frac{q^2}{12}}{\left(\|\mathbf{w}\|_2^2 + \frac{nq^2}{12}\right)^2} \tag{D.34}$$

$$= \frac{4\|\mathbf{w}\|_2^2\frac{q^2}{12}}{\|\mathbf{w}\|_2^4 + 2\|\mathbf{w}\|_2^2\frac{nq^2}{12} + \left(\frac{nq^2}{12}\right)^2} \tag{D.35}$$

$$= \frac{4\frac{q^2}{3\|\mathbf{w}\|_2^2}}{1 + \frac{2nq^2}{12\|\mathbf{w}\|_2^2} + \left(\frac{nq^2}{12\|\mathbf{w}\|_2^2}\right)^2} \tag{D.36}$$

$$= \frac{4\gamma}{n\left(1 + 2\gamma + \gamma^2\right)} = \frac{4}{n}\gamma + o_p\left(\gamma\right), \tag{D.37}$$

where

$$\gamma = \frac{nq^2}{12\|\mathbf{w}\|_2^2}, \tag{D.38}$$

and

$$\mathrm{var}[Y_2] = \frac{\frac{4nq^4}{720}}{\left(\|\mathbf{w}\|_2^2 + \frac{nq^2}{12}\right)^2} \tag{D.39}$$

$$= \frac{4}{5}\frac{\gamma^2}{n\left(1 + 2\gamma + \gamma^2\right)} = \frac{4}{5n}\gamma^2 + o_p(\gamma^2). \tag{D.40}$$

The terms $o_p(\gamma)$ and $o_p(\gamma^2)$ denote a random variable with mean and variance that are no larger than γ and γ^2, respectively. Therefore, we can write an approximation to order 0,

$$Y = \frac{1}{\sqrt{\|\mathbf{w}\|_2^2\left(1 + \gamma\right)}}\left(1 + o_p\left(\gamma^{\beta}\right)\right), \tag{D.41}$$

with $0 < \beta < 1/2$.

The step q depends on the distribution of $\mathbf{w}_i$. We can use the approximation given in [Orabona and Pal, 2015]:

$$\mathbb{E}[\max_i w_i - \min_i w_i] \approx 2\sigma\sqrt{2\ln d}. \tag{D.42}$$

This approximation is interesting because it depends on two quantities directly related to the layer of neurons: the number of neurons d and the standard deviation σ of the weights.
It is clear that

$$\mathbb{E}[\|w\|_2^2] = d\sigma^2. \tag{D.43}$$

Finally, by using this approximation in (D.38), we can approximate the value of γ. It is interesting to note that a very rough calculation shows that

$$\gamma \approx \frac{n8\sigma^2 \ln n}{12n\sigma^2 2^{2R+2}} = \frac{\ln n}{12}\frac{1}{2^{2R-1}}. \tag{D.44}$$

The quality of the approximation could thus essentially depend on the dimension n of the hidden layer for a given number R of bits. It seems that, whatever the number of bits, the approximation can be very efficient if the number n of neurons is sufficiently large.

Using the previous approximation of the denominator (D.41), we expand and replace the denominator as follows:

$$a_j(\hat{\mathbf{w}}) = \frac{\lambda - \mathbf{w}^T\mu_j - \Delta^T\mu_j}{\sqrt{\mathbf{w}^T\mathbf{w}}\sqrt{1+\gamma}}(1 + o_p(\gamma/n)), \tag{D.45}$$

where $o_p(\gamma/n))$ denotes a random variable with mean and variance not larger than γ/n.
We rewrite the coefficient $a_j(\hat{w})$ using $a_j(w)$

$$a_j(\hat{\mathbf{w}}) = \frac{1}{\sqrt{1+\gamma}}\left(a_j(\mathbf{w}) - \frac{\Delta^T\mu_j}{\|\mathbf{w}\|_2}\right)(1 + o_p(\gamma/n)). \tag{D.46}$$

Since $\mathbb{E}\left[\frac{\Delta^T\mu_j}{\|\mathbf{w}\|_2}\right] = 0$ and $\text{var}\left[\frac{\Delta^T\mu_j}{\|\mathbf{w}\|_2}\right] = \frac{\gamma\|\mu_j\|_2^2}{n}$, (D.46) can be rewritten as follows:

$$a_j(\hat{\mathbf{w}}) = \frac{a_j(\mathbf{w})}{\sqrt{1+\gamma}} + o_p\left(\gamma/n\right). \tag{D.47}$$

D.2 Proof theorem 7.4.2: approximation of the distortion $d(\mathbf{w})$

The distortion measure $d(\mathbf{w}, \hat{\mathbf{w}})$ is the absolute value of difference between the original risk and the risk with quantization. The conditional distortion can be written as follows:

$$d_j = |R_j(\hat{\mathbf{w}}) - R_j(\mathbf{w})|. \tag{D.48}$$

Consider the original risk

$$R_j(\mathbf{w}) = \Phi\left((-1)^j a_j(\mathbf{w})\right), \text{ for } j \in \{0, 1\}, \tag{D.49}$$

and the risk with quantization

$$R_j(\hat{\mathbf{w}}) = \Phi\left((-1)^j a_j(\hat{\mathbf{w}})\right), \text{ for } j \in \{0, 1\}. \tag{D.50}$$

The problem can be reduced to a simple measuring of the area between $a_j(\mathbf{w})$ and $a_j(\hat{\mathbf{w}})$. This can be done by using an approximation for definite integrals. The rectangle method [Oberbroeckling, 2021] is the simplest method used to compute an approximation of a definite integral. We apply this method to the distortion conditioned by the class 1:

$$\begin{aligned} d_1(\mathbf{w}) &= \left|\Phi\left(-a_1(\hat{\mathbf{w}})\right) - \Phi\left(-a_1(\mathbf{w})\right)\right| && \text{(D.51)}\\ &= \left|\Phi\left(a_1(\mathbf{w})\right) - \Phi\left(a_1(\hat{\mathbf{w}})\right)\right| && \text{(D.52)}\\ &\approx \left|\left(a_1(\mathbf{w}) - a_1(\hat{\mathbf{w}})\right)\varphi\left(a_1(\mathbf{w})\right)\right|. && \text{(D.53)} \end{aligned}$$

We replace $a_1(\hat{\mathbf{w}})$ with the approximation given in (D.47) which leaves us with

$$\begin{aligned} d_1(\mathbf{w}) &= \left|\left(a_1(\mathbf{w}) - \frac{a_1(\mathbf{w})}{\sqrt{1+\gamma}} + o_p(\gamma/n)\right)\varphi\left(a_1(\mathbf{w})\right)\right| && \text{(D.54)}\\ &= \left|\left(1 - \frac{1}{\sqrt{1+\gamma}}\right)a_1(\mathbf{w})\varphi\left(a_1(\mathbf{w})\right) + \epsilon(\gamma/n)\right|. && \text{(D.55)} \end{aligned}$$

$\epsilon(\gamma/n)$ is a non-random error term is of the same order as γ/n. We apply the same method for the class 0 and the average distortion measure is given by

$$d(\mathbf{w}) = \left|\eta(\gamma)\left(\pi_0 a_0 \varphi(a_0) - \pi_1 a_1 \varphi(a_1)\right) + \epsilon(\gamma/n)\right|, \tag{D.56}$$

where $a_j = a_j(\mathbf{w})$, $\eta(\gamma)$ is defined by

$$\eta(\gamma) = 1 - \frac{1}{\sqrt{1+\gamma}}. \tag{D.57}$$

D.3 Proof corollary 7.4.3: approximation of the distortion $d(\mathrm{w})$

A more accurate approximation than theorem 7.4.2 can be acquired by using Simpson's rule [Gautschi, 2011]. Simpson's 1/3 rule is defined as follows:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right], \tag{D.58}$$

where a and b are the end points.

We consider the definite integral which can be written from the conditional risks before and after compression. The integral for class 1 is written as follows:

$$\begin{aligned} R_1(\hat{\mathbf{w}}) - R_1(\mathbf{w}) &= \Phi\left(a_1(\hat{\mathbf{w}})\right) - \Phi\left(a_1(\mathbf{w})\right) && \text{(D.59)}\\ &= \int_{a_1(\mathbf{w})}^{a_1(\hat{\mathbf{w}})} \varphi(x)dx, \ \text{for } j \in \{0,1\}. && \text{(D.60)} \end{aligned}$$

By applying (D.58) to (D.60) we obtain:

$$d_1(\mathbf{w}) = \int_{a_1(\mathbf{w})}^{a_1(\hat{\mathbf{w}})} \varphi(x)dx \approx \frac{|a_1(\hat{\mathbf{w}}) - a_1(\mathbf{w})|}{6}\left[f(a) + 4f\left(\frac{a_1(\mathbf{w}) + a_1(\hat{\mathbf{w}})}{2}\right) + f(a_1(\hat{\mathbf{w}}))\right]. \tag{D.61}$$

Then, we replace $a_1(\hat{\mathbf{w}})$ with the approximation given in (D.47), which leads us to:

$$d_1(\mathbf{w}) = \left|\frac{\eta(\gamma)}{6}\left(a_1(\mathbf{w})\varrho(a_1(\mathbf{w}))\right) + \varepsilon(\gamma/n)\right|, \tag{D.62}$$

where $t \mapsto \varrho(t)$ is given by

$$\varrho(t) = \varphi(t) + 4\varphi\left(\frac{\zeta(\gamma)}{2}t\right) + \varphi\left(\frac{t}{\sqrt{1+\gamma}}\right), \tag{D.63}$$

$\eta(\gamma)$ is defined in (D.57), $\zeta(\gamma)$ is

$$\zeta(\gamma) = \frac{1}{2}\left(1 + \frac{1}{\sqrt{1+\gamma}}\right) = 1 - \frac{\eta(\gamma)}{2}, \tag{D.64}$$

and $\varepsilon(\gamma/n)$ is an error term that is of the same order as γ/n.
Computing $d_0(\mathbf{w})$ is similar and the average distortion measure is written as follows:

$$d(\mathbf{w}) = \left|\frac{\eta(\gamma)}{6}\left(\pi_0 a_0 \varrho(a_0) - \pi_1 a_1 \varrho(a_1)\right) + \varepsilon(\gamma/n)\right|. \tag{D.65}$$

Compression pour l’apprentissage en profondeur

Diana RESMERITA

Résumé

Les voitures autonomes sont des applications complexes qui nécessitent des machines puissantes pour pouvoir fonctionner correctement. Des tâches telles que rester entre les lignes blanches, lire les panneaux ou éviter les obstacles sont résolues en utilisant plusieurs réseaux neuronaux convolutifs (CNN) pour classer ou détecter les objets. Il est très important que tous les réseaux fonctionnent en parallèle afin de transmettre toutes les informations nécessaires et de prendre une décision commune. Aujourd’hui, à force de s’améliorer, les réseaux sont devenus plus gros et plus coûteux en termes de calcul. Le déploiement d’un seul réseau devient un défi. La compression des réseaux peut résoudre ce problème. Par conséquent, le premier objectif de cette thèse est de trouver des méthodes de compression profonde afin de faire face aux limitations de mémoire et de puissance de calcul présentes sur les systèmes embarqués. Les méthodes de compression doivent être adaptées à un processeur spécifique, le MPPA de Kalray, pour des implémentations à court terme. Nos contributions se concentrent principalement sur la compression du réseau après l’entraînement pour le stockage, ce qui signifie compresser des paramètres du réseau sans réentraîner ou changer l’architecture originale et le type de calculs. Dans le contexte de notre travail, nous avons décidé de nous concentrer sur la quantification. Notre première contribution consiste à comparer les performances de la quantification uniforme et de la quantification non-uniforme, afin d’identifier laquelle des deux présente un meilleur compromis taux-distorsion et pourrait être rapidement prise en charge par l’entreprise. L’intérêt de l’entreprise est également orienté vers la recherche de nouvelles méthodes innovantes pour les futures générations de MPPA. Par conséquent, notre deuxième contribution se concentre sur la comparaison des représentations en virgule flottante (FP32, FP16) aux représentations arithmétiques alternatives telles que BFloat16, MSFP8, Posit8. Les résultats de cette analyse sont en faveur de Posit8. Ceci a motivé la société Kalray à concevoir un décompresseur de FP16 vers Posit8. Puisque de nombreuses méthodes de compression existent déjà, nous avons décidé de passer à un sujet adjacent qui vise à quantifier théoriquement les effets de l’erreur de quantification sur la précision du réseau. Il s’agit du deuxième objectif de la thèse. Nous remarquons que les mesures de distorsion bien connues ne sont pas adaptées pour prédire la dégradation de la précision dans le cas de l’inférence pour les réseaux de neurones compressés. Nous nous concentrons sur la définition d’une nouvelle mesure de distorsion avec une expression analytique qui a une forme de rapport signal/bruit. Un ensemble d’expériences a été réalisé en utilisant des données simulées et de petits réseaux, qui montrent le potentiel de la mesure.

Mots-clés : Compression, réseaux de neurones profond, quantification, virgule flottante, analyse statistique, approximation des erreurs.

Abstract

Autonomous cars represent complex applications that need powerful hardware machines to be able to function properly. Tasks such as lane-keeping, reading and understanding traffic signs or avoiding obstacles are solved by employing convolutional neural networks (CNNs) for object detection and classification. It is highly important that all the networks work in parallel in order to transmit all the necessary information and take a common decision. Nowadays, as the networks improve, they also have become bigger and more computational expensive. Deploying even one network becomes challenging. Compressing the networks can solve or at least alleviate this issue. Therefore, the first objective of this thesis is to find deep compression methods in order to cope with the memory and computational power limitations present on embedded systems. The compression methods need to be adapted to a specific processor, Kalray’s MPPA, for short term implementations. Our contributions of this thesis mainly focus on compressing the network post-training for storage purposes, which means compressing the parameters of the network without retraining or changing the original architecture and the type of the computations. In the context of our work, we decided to focus on quantization. Our first contribution consists in comparing the performances of uniform quantization and non-uniform quantization, in order to identify which of the two has a better rate-distortion trade-off and could be quickly supported in the company. The company’s interest is also directed towards finding new innovative methods for future MPPA generations. Therefore, our second contribution focuses on comparing standard floating-point representations (FP32, FP16) to non-standard arithmetical representations such as BFloat16, MSFP8, Posit8. The results of this analysis are in favor for Posit8. This motivated the company Kalray to conceive a decompressor from FP16 to Posit8. Since many compression methods already exist, we decided to move to an adjacent topic which aims to quantify the effects of quantization error on the network’s accuracy. This is the second objective of the thesis. Finally, we focus on defining a new distortion measure adapted to our requirements, which represents a mainly theoretical contribution. Under reasonable assumptions, such as Normal input distribution, the distortion measure takes into account only the last layer of the network, the Softmax layer, and is adapted to a binary classification model. A set of experiments were done, using simulated data and small networks, which showcase the potential of the method.

Keywords: Data Compression, Deep Compression, Deep Learning, Deep Neural Networks, Floating-point, Quantization, Statistical Analysis, Error Approximation.