Towards lighter and faster deep neural networks with parameter pruning

Nathan Hubens

▶ **To cite this version:**

Nathan Hubens. Towards lighter and faster deep neural networks with parameter pruning. Information Retrieval [cs.IR]. Institut Polytechnique de Paris; Université de Mons, 2022. English. NNT : 2022IPPAS025 . tel-03953635

HAL Id: tel-03953635

https://theses.hal.science/tel-03953635

Submitted on 24 Jan 2023

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.

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAS025

Thèse de doctorat

UMONS
University of Mons

Towards Lighter and Faster Deep Neural Networks with Parameter Pruning

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis en cotutelle avec l'Université de Mons

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Mons (Belgique), le 07/12/2022, par

NATHAN HUBENS

Composition du Jury :

Véronique Moeyaert Professeure, Université de Mons	Président
John Lee Professeur, Université Catholique de Louvain	Rapporteur
Ioan Tabus Professeur, Université de Tampere	Rapporteur
Bruno Grilhères Senior Expert Machine Learning, Airbus Defence and Space	Examinateur
Bernard Gosselin Professeur, Université de Mons	Directeur de thèse
Matei Mancas Docteur, Université de Mons	Co-directeur de thèse
Titus Zaharia Professeur, Télécom SudParis	Directeur de thèse
Marius Preda Maître de Conférence, Télécom SudParis	Co-directeur de thèse
Thierry Dutoit Professeur, Université de Mons	Invité

"If I had more time, I would have written a shorter thesis."

— Nathan Hubens (inspired by Blaise Pascal)

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisors, Professors Bernard Gosselin and Titus Zaharia, for their warm reception in their respective departments. They knew how to be present while leaving me great freedom of work, and I had the privilege to benefit from their valuable teachings during the last four years.

I would also like to thank my co-supervisors, professors Marius Preda and Matei Mancas, for their availability and their attentive listening. I will remember their open-mindedness and curiosity, always proposing new paths to explore.

Moreover, I would like to thank the other members of my thesis committee: Professor Thierry Dutoit, Professor Sidi Mahmoudi, Professor Gilles Louppe and Doctor Khaled Mammou, who ensured the good evolution of the thesis by giving me constructive remarks, suggestions and advice.

A huge thank you to the members of the jury: Professor Véronique Moeyaert, Professor Ioan Tabus, Professor John Lee and Doctor Bruno Grilhères. Their comments and advice helped to make this document better.

I am deeply thankful to my fellows at ISIA Lab, more particularly to the Paysager 2 team: Maio, Victor, Luca, Bastien and Omar, with whom I shared a lot of good moments and that have been of inestimable help. The coffee-conversations and the *boulets-frites-orval* shared together will be deeply missed. I wish them many twerks.

Sincere thanks to my colleagues at the Artemis department: Traian, Abhaya-Dhathri, Nicolas, Minderis, Chao, Zied, Lisa, Tan-Khoa, for their contagious good mood, the philosophical debates, and our daily games of table soccer.

Finally, this work would not have been possible without the unconditional support of all my friends and family. I can never thank them enough for their presence and encouragement in good, as well as in bad times.

I would like to address my last thanks to Adélaïde, who has been sharing my life for the past 10 years. She continues to be my main source of inspiration and my daily motivation.

Contents

1 Fundamentals **9**

1.1 Introduction 10

1.1.1 A bit of History 10

1.2 Multi-Layer Perceptron 12

1.2.1 Definition 12

1.2.2 Perceptron 13

1.2.3 Training 14

1.3 Convolutional Neural Networks 17

1.3.1 Definition 17

1.3.2 Convolutional Layer 18

1.3.3 Pooling Layer 20

1.3.4 Training 20

1.4 In Brief 21

2 Neural Network Compression **23**

2.1 Introduction 24

2.2 Sparse Neural Networks 25

2.2.1 Designed Sparsity 25

2.2.2 Learned Sparsity 28

2.2.3 Ephemeral Sparsity 31
2.3 Knowledge Distillation 33
2.4 Quantization 35
2.4.1 Post-Training Quantization 38
2.4.2 Quantization Aware Training 38
2.4.3 Automatic Mixed Precision 39
2.5 Compact Neural Network Architectures 40
2.5.1 Matrix Factorization 40
2.5.2 Batch Normalization Folding 45
2.5.3 Kernel Size Reduction 46
2.5.4 Channel Amount Reduction 48
2.6 In Brief 50

3 Neural Network Pruning **51**
3.1 Introduction 52
3.2 Motivation 53
3.2.1 Improves Generalization 53
3.2.2 Lowers Complexity 54
3.2.3 Reduces Processing Time and Storage 55
3.3 Neural Network Pruning 56
3.3.1 How to prune ? 57
3.3.2 Where to prune ? 62
3.3.3 What to prune ? 63
3.3.4 When to prune ? 69
3.4 In Brief 73

4 Developed Tools: FasterAI **75**
4.1 Introduction 76
4.2 Sparsify 77
4.2.1 Granularity 78
4.2.2 Context 80
4.2.3 Criteria 82
4.2.4 Schedule 84
4.2.5 Lottery Ticket Hypothesis 87
4.2.6 Prune 88

4.3 Distill 89
4.4 Regularize 90
4.5 Misc 91
4.5.1 Batch Normalization Folding 92
4.5.2 Fully-Connected Layers Decomposition 92
4.6 In Brief 93

5 Advances in Neural Network Pruning **95**
5.1 Introduction 96
5.2 How to prune? 96
5.2.1 Methodology 97
5.2.2 Experiments 100
5.2.3 Discussion & Conclusion 105
5.3 Where to prune? 106
5.3.1 Methodology 108
5.3.2 Experiments 110
5.3.3 Discussion & Conclusion 116
5.4 What to prune? 116
5.4.1 Methodology 117
5.4.2 Experiments 120
5.4.3 Discussion & Conclusion 124
5.5 When to prune? 124
5.5.1 Methodology 125
5.5.2 Experiments 126
5.5.3 Discussion & Conclusion 131
5.6 In Brief 132

6 Use-Case: DeepFake Detection **133**
6.1 Introduction 134
6.1.1 DeepFakes Detection Challenge 134
6.1.2 Related Work 135
6.2 Methodology 136
6.2.1 The Dataset 136
6.2.2 The Network 137
6.2.3 Data Augmentation 141

6.2.4 Compression of the solution 142
6.3 Results 143
6.3.1 Ablation Study 143
6.3.2 Comparison to other methods 144
6.3.3 Interpretation 145
6.4 Proof-of-Concept: Deepfake Buster 146
6.5 Discussion and Conclusions 148
6.6 In Brief 149

7 Conclusions **151**

Bibliography **157**

List of Figures **169**

List of Pseudo-Code **177**

List of Tables **179**

List of acronyms

AI	Artificial Intelligence	4
AGI	Artificial General Intelligence	11
AM	Activation Maximization	117
AGP	Automated Gradual Pruning	124
AMP	Automatic Mixed-Precision	38
ANN	Artificial Neural Network	10
BM	Bitmap	55
BN	Batch Normalization	45
CNN	Convolutional Neural Network	10
COO	Coordinate Offset	55
CSR	Compressed Sparse Row	55
CSC	Compressed Sparse Column	55
DFDC	DeepFake Detection Challenge	134
DL	Deep Learning	4
DNN	Deep Neural Network	4
DSD	Dense-Sparse-Dense	86
FCL	Fully-Connected Layer	13
FCN	Fully-Connected Network	25

FFN Feed-Forward Network 12
FLOPs Floating Point Operations 104
LTH Lottery Ticket Hypothesis 70
LTHR Lottery Ticket Hypothesis with Rewinding 70
MDL Minimum Description Length 54
MLP Multi-Layer Perceptron 12
NAS Neural Architecture Search 154
NLP Natural Language Processing 4
NMS Non-Maximum Suppression 139
OCP One-Cycle Pruning 124
PAI Pruning At Initialization 71
PAT Pruning After Training 71
PDT Pruning During Training 71
PTQ Post-Training Quantization 38
QAT Quantization-Aware Training 38
RNN Recurrent Neural Network 106
ReLU Rectified Linear Unit 140
SGD Stochastic Gradient Descent 14
SVD Singular Value Decomposition 43
XAI Explainable Artificial Intelligence 153

Introduction

"The beginning is the most important part of the work."
— Plato

Motivations

Since their resurgence in 2012, Deep Neural Networks (DNNs) have become ubiquitous in most disciplines attributed to the field of Artificial Intelligence (AI), such as image recognition, speech processing, Natural Language Processing (NLP), and recommender systems. This Deep Learning (DL) era has emerged in the last ten years thanks to the considerable improvements in high-performance hardware and access to a large amount of data, enabling the creation of those so-called Deep Neural Networks. However, over the last few years, neural networks have grown exponentially deeper, involving more and more parameters, and with an amount of computing doubling every 3.4 months [1]. Nowadays, it is not unusual to encounter architectures involving several billions of parameters, while they mostly contained hundreds of thousand less than ten years ago [2]. Indeed, it has recently been discovered that the performance of a neural network evolves as a power-law with three factors: the model size, the dataset size, and the amount of computing for training [3]. Such a discovery corroborates the trend of increasingly larger networks and has further motivated the main research actors to dedicate even more resources to neural network training. As represented in Figure 0.1, reporting the evolution of the parameter count in state-of-the-art models, such an increase has been observed in most fields of application of DL.

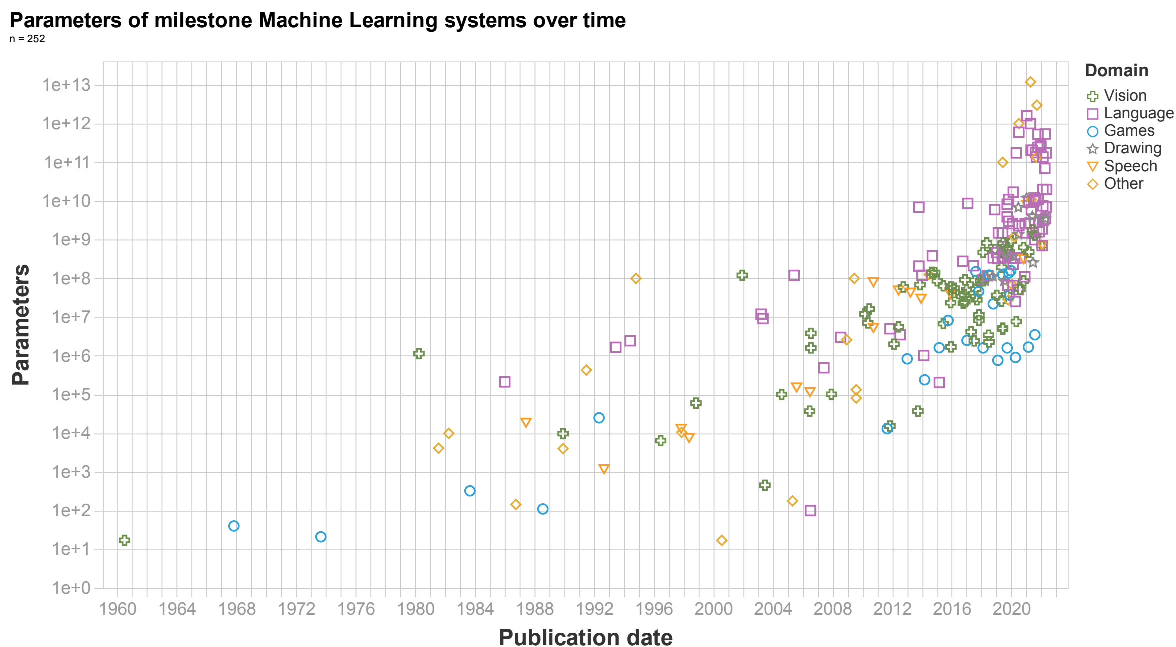


Figure 0.1. *Evolution of the amount of parameters present in Deep Learning architectures. Each research field has known an exponential increase in parameters involved in their architectures. Image created with [2].*

This generalized increase in the number of parameters makes such large models compute-intensive and essentially energy inefficient. Indeed, it has been reported that some large models require more than 400 years of compute time, thus costing more than $3 million and releasing an amount of CO_2 equivalent to more than 50 years of the one emitted by a single person [4,5]. In addition to the training costs, the inference pass of such models is also impacted. It is currently estimated that approximately 90% of the workload of a machine learning model is dedicated to inference processing [6, 7], making deployed models costly to maintain but also their use in applications where latency is a critical requirement, e.g. autonomous driving, very challenging. On top of that, a growth in parameter count implies an increase in the storage footprint of such networks. This severely hinders the deployment of such solutions on resource-constrained devices, as the costs of transfer and storage of such models become an issue. Even in large-scale environments, where the amount of stored neural networks become important, storage and transfer costs of networks have to be minimized for better cost-efficiency.

For these reasons, much research has recently been conducted to provide techniques able to reduce the amount of storage and computing required by neural networks. Such techniques can operate at different levels of the deep learning development stack. Indeed, some techniques are involved in hardware optimization to improve the throughput and energy efficiency of neural networks, leading to either new hardware designs, providing acceleration capabilities for matrix multiplications, or leveraging new memory access methods to support specialized processing dataflows [8]. Other techniques operate directly on the models, and seek to either design more efficient architectures with fewer parameters or computations, or develop techniques that reduce the computations requirements from existing networks [9].

Among those techniques, neural network pruning has been recently at the forefront of research. This technique creates sparsely connected neural networks, usually by starting from trained models, and gradually removing their parameters. It has been shown that such a technique allows to produce non-trivially sparse models without degrading their performance, illustrating the over-parameterization of current neural networks. However, although pruning is a prevalent compression technique, there is currently no standard way of implementing or evaluating novel pruning techniques, making the comparison with previous research challenging [10]. Moreover, pruning methods are still in an early stage of development, and are thus primarily designed for the research community. Indeed, most pruning works are usually implemented in a self-contained and sophisticated way, making it troublesome for non-researchers to apply such techniques to obtain smaller and faster neural networks without having to learn all the intricacies of the field.

Objectives

This thesis aims to tackle the problem of neural network compression and, more specifically, neural network pruning.

More precisely, the objectives of the thesis are:

- Provide a universal description of the pruning problem, facilitating the evaluation and benchmarking of novel methods;
- Create a well-rounded toolbox for a seamless implementation of different compression techniques, and based on the previous pruning description;
- Propose new pruning techniques, improving on current methods in the field;
- Validate the efficiency of the developed techniques through a real use-case and through challenges;
- Make the research reproducible and openly available to the AI community.

Contributions

According to the motivations and objectives of the thesis, contributions can be summarized as:

- The description of the pruning problem according to 4 independent parameters, sufficient to fully describe most pruning techniques: granularity, context, criteria, and schedule;
- The creation of `FasterAI` [1], an open-source library for neural networks compression, suited for professionals as well as enthusiasts;
- A study of the granularity of pruning, highlighting which granularity to use depending on the type of training.
- A study of the context of pruning, highlighting which context to use depending on the type of training.
- The proposal of a novel method to improve current pruning criteria, allowing to reduce redundancy while retaining rare filters.
- The proposal of a novel pruning schedule, better incorporated in the training dynamics.
- The validation of proposed techniques for a real use-case scenario, DeepFake Detection.

[1] https://github.com/nathanhubens/fasterai

- The creation of a proof-of-concept application, allowing users to perform DeepFake Detection with a lightweight solution.

Each contribution comes with an open-source implementation, available on GitHub [2]. If applicable, contributions have directly been incorporated into the developed `FasterAI` library for more convenient usage. The `FasterAI` library was further improved during a 6-month internship at AMD [3], and is now an integral part of the development of one of their projects.

Moreover, those contributions are supported by publications, published as peer-reviewed papers, listed in order of apparition:

- **Nathan Hubens**, Matei Mancas, Marc Decombas, Marius Preda, Titus Zaharia, Bernard Gosselin, and Thierry Dutoit. "An Experimental Study of the Impact of Pre-Training on the Pruning of a Convolutional Neural Network". In Proceedings of the 3rd International Conference on Applications of Intelligent Systems (APPIS), 2020.
- **Nathan Hubens**, Matei Mancas, Bernard Gosselin, Marius Preda, and Titus Zaharia. "One-Cycle Pruning: Pruning ConvNets under a Tight Training Budget". In Sparsity in Neural Networks: Advancing Understanding and Practice (SNN), 2021.
- **Nathan Hubens**, Matei Mancas, Bernard Gosselin, Marius Preda, Titus Zaharia. "Fake-buster: a lightweight solution for deepfake detection". SPIE Optical Engineering + Applications: Applications of Digital Image Processing, 2021
- Maiorca, A.; **Hubens, N.**; Laraba, S. and Dutoit, T.. "Towards Lightweight Neural Animation: Exploration of Neural Network Pruning in Mixture of Experts-based Animation Models". In Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP), 2022.
- **Nathan Hubens**, Matei Mancas, Bernard Gosselin, Marius Preda, and Titus Zaharia. "Improve Convolutional Neural Network Pruning by Maximizing Filter Variety". In Image Analysis and Processing (ICIAP), 2022.
- **Nathan Hubens**, Matei Mancas, Bernard Gosselin, Marius Preda, and Titus Zaharia. "Fasterai: A Lightweight Library for Sparse Neural Networks". In Sparsity in Neural Networks: Advancing Understanding and Practice (SNN), 2022.
- **Nathan Hubens**, Matei Mancas, Bernard Gosselin, Marius Preda, and Titus Zaharia. "One-Cycle Pruning: Pruning ConvNets under a Tight Training Budget". In International Conference of Image Processing (ICIP), 2022.

[2] https://github.com/nathanhubens/contributions
[3] https://www.amd.com/en/corporate/research

- **Nathan Hubens**, Matei Mancas, Bernard Gosselin, Marius Preda, and Titus Zaharia. “Fasterai: A Lightweight Library for Neural Networks Compression”. Under Review in Important Features Selection in Deep Neural Networks, MDPI Electronics, 2022.
- **Nathan Hubens**, Matei Mancas, Bernard Gosselin, Marius Preda, and Titus Zaharia.“Induced Feature Selection by Structured Pruning”. Preprint at ArXiv, 2022.

Thesis Structure

This thesis is organized according to a top-down approach, by first identifying the big picture and presenting general concepts, then gradually diving deeper into all of the components. In particular, the thesis is structured as follows:

- Chapter 1 presents the theoretical notions required for the reader to understand the field of deep learning and its principal components, the neural networks.
- The Chapter 2 provides an introduction to the main compression techniques that currently exist for neural networks.
- In Chapter 3, we dive into a particular compression technique: parameter pruning. It consists in removing parameters in a network while aiming to keep its performance intact. In particular, we identify 4 research axes according to which pruning can be considered, and base our description according to those.
- The Chapter 4 describes `FasterAI`, a PyTorch-based library that aims to bring compression capabilities to newcomers, but also to provide to researchers an exhaustive tool to implement new compression methods and to perform various experiments easily.
- Chapter 5 presents advances in each of the previously identified 4 research axes of pruning.
- The Chapter 6 describes a use-case scenario chosen to test the efficiency of compression techniques developed throughout the thesis. For this purpose, we have chosen the case of DeepFake Detection, aiming to detect manipulations that have occurred in an image or a video.

CHAPTER 1

Fundamentals

Contents

1.1 Introduction **10**
1.1.1 A bit of History 10
1.2 Multi-Layer Perceptron **12**
1.2.1 Definition 12
1.2.2 Perceptron 13
1.2.3 Training 14
1.3 Convolutional Neural Networks **17**
1.3.1 Definition 17
1.3.2 Convolutional Layer 18
1.3.3 Pooling Layer 20
1.3.4 Training 20
1.4 In Brief **21**

“Nothing has yet been said that’s not been said before.”
— Publius Terentius

1.1 Introduction

This chapter aims to introduce the topic of artificial neural networks and convolutional neural networks. Firstly, a brief introduction to Artificial Neural Networks (ANNs) and their training will be given in section 1.2. Subsequently, Convolutional Neural Networks (CNNs) will be discussed more thoroughly in section 1.3.

1.1.1 A bit of History

For a very long time, the idea of an autonomous and decision-making entity has fed the fantasies of many humans from all over the world. Examples of such a system can be traced back to Greek mythology, where Talos, a giant automaton made of bronze offered by the god Hephaestus, was responsible for protecting Europa in Crete from pirates and invaders by throwing rocks at any ship approaching the island. It was also present in Chinese mythology, when chancellor Zhuge Liang visited the house of Yueying Huang, and was greeted by two large watchdogs. After panicking at the sight of those dogs, he realized that they were actually made of wood. Yueying Huang was said to be talented in creating artificial intelligent animals. Another example is present in Jewish folklore, where the golem is an animated and anthropomorphic being entirely created from inanimate matter. The creation of such an entity could be achieved by the insertion of a piece of paper with any of God's names on it, into the mouth of the clay figure.

The creation of articulated living beings continued to interest most eminent engineers and inventors throughout the years, such as Leonardo Da Vinci, who created complex mechanical soldiers and lions, and Jacques de Vaucanson, well renowned for the creation of many automata. However, while creating more and more realistic version of physical bodies, those were devoid of decision capabilities. The creation of automated entities then led to a field, named robotics, which is primarily disjoint to creating AI, as it does not seek to create a logical machine, mimicking the decision process of a human brain.

AI as we know it nowadays finds its roots in the assumption that the process of human thought can be mechanized. Early attempts to develop such a method were proposed by philosophers such as Aristotle, who gave a formal analysis of the syllogism, and Euclid, whose *Elements* mathematical treatise was a model of formal reasoning, and al-Khwārizmī, who developed algebra. However, in the 17th century, Leibniz envisioned a universal language of reasoning, the *Characteristica universalis*, that would reduce argumentation to calculation, and conducted to foundations of the evolution of AI. His work was later developed by Hobbes and Descartes that postulated that rational thought could be made as systematic as algebra or geometry.

The next breakthrough happened in the 20th century in the field of mathematics with Boole's work *The Laws of Thought* [11] and more particularly with the first programmable digital computer being innovated in the 1943. This was shortly followed by McCulloch and Pitts, who formalized the computational model of an artificial neuron [12]. In 1950, Alan Turing published a landmark paper [13] in which he speculated about the possibility of creating machines that can *think* by themselves. He proposed a way to verify if a machine was able to think, called the *Turing test*. But it was in 1956 with the Darmouth Conference organized by Minsky, Shannon and Rochester that the term AI was first used and accepted as the name for this field of research. In these early days, the advances in AI were accompanied with very high expectations, so high that Simon and Minsky claimed that the problem of creating Artificial General Intelligence (AGI), i.e. a machine able to perform any intellectual task that a human being can, was about to be solved, particularly with Rosenblatt invention providing to an articificial neuron the ability to learn, giving the Perceptron [14]. A few years later, scientists realized that the problem was more complex than what they thought and the promises that they made failed to materialize. Researchers and government funds turned away from the field, marking the start of the first *AI winter*.

The 1980s brought a renewal in the field with the apparition of *expert systems*, simulating human experts' knowledge and analytical skills. It did not take long before seeing funding and researchers coming back in the field, driving new expectations about future AI capabilities. However, those networks took too much time to train, were still expensive to maintain, and difficult to scale. As a result, the interest progressively died down in the 1990s, announcing the second AI winter.

In the past decade, considerable improvements in computer hardware and access to large amounts of data allowed to enter a new AI era, *Deep Learning*. Those improvements lead to deeper neural networks, which are currently driving progress in many research fields such as image and video processing, speech recognition and text analysis. Those architectures are now able to compete with human-level accuracy in most basic perception tasks. Such successes revived AI investments, with most prominent tech companies leading the research. Nowadays, in the 2020s, we are currently in similar situations as the early 1960s and 1980s, with the AI field defying all expectations again. There are still ongoing debates in the scientific community about the timeline before AGI will be achieved. On the other hand, some scientists try to mitigate the short-term expectations, to avoid disappointing investors and potentially to lead to a third AI winter. Their task begins by ensuring that the general public has a clear understanding of what AI is capable of, and most importantly, not capable of nowadays.

In the next chapter, we will present more formally the mathematical models of Artificial Neural Networks and how they are able to learn, which shapes what is known nowadays as the Artificial Intelligence field.

1.2 Multi-Layer Perceptron

1.2.1 Definition

An Multi-Layer Perceptron (MLP) is the most basic type of ANN. As the name suggests, it consists of several layers of perceptrons, the fundamental computational unit. Each layer performs a change in the representation of its input data, such that the whole system maps its inputs x to outputs a. A MLP composed of 4 layers is represented in Figure 1.1.

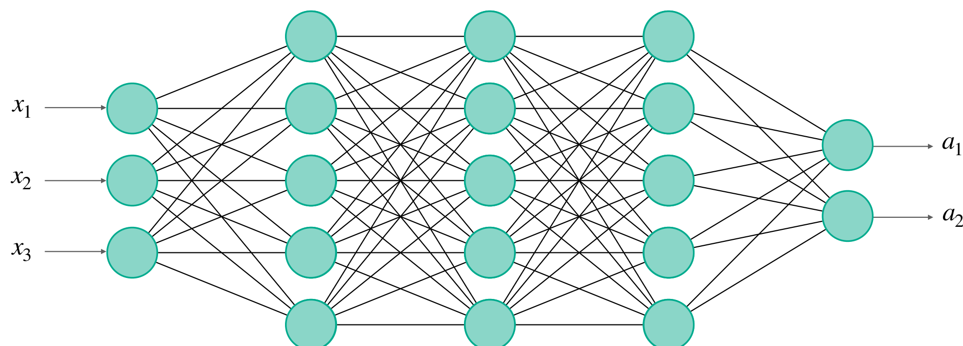


Figure 1.1. Representation of a 4-layer MLP. It takes a 3-dimensional input and has a 2-dimensional output. Each neuron is connected to all previous and following neurons.

This type of neural network falls into the Feed-Forward Network (FFN) category, i.e. networks whose connections do not form any cycle or recursive structure. Despite the inherent simplicity of its components, MLPs are able to represent highly complex functions, and are thus capable of high performance on complex perceptual tasks. In particular, it has been proven that even a two-layer MLP is able to approximate any continuous and multivariate function to any desired precision [15, 16], which is rightfully called the *Universal Approximation Theorem*. However, for such a theorem to be verified, the number of neurons needs to grow very large. To counteract this effect, several layers are usually stacked to increase the network's abstraction and expressiveness, allowing it to model more complex functions with fewer parameters.

1.2.2 Perceptron

The perceptron is the most elementary computation unit in the domain of ANNs. It is inspired by a biological neuron in the sense that it activates for particular incoming data. When connected to other neurons, it allows the detection of complex features from the input data. In particular, a perceptron takes in a set of input values, associated with a weight. It then computes the weighted sum of each input, adds in the bias, and sends results to a non-linear activation function. More formally, the output of a perceptron is computed as:

$$a = f(\mathbf{w}^T\mathbf{x}) = f(\sum_{i=1}^{n} w_i x_i + b) \quad (1.1)$$

where $\mathbf{x}$ is the vector containing the inputs, $\mathbf{w}$ is the vector containing the weights, b is the bias and $f(.)$, a non-linear function. A perceptron is represented in Figure 1.2.

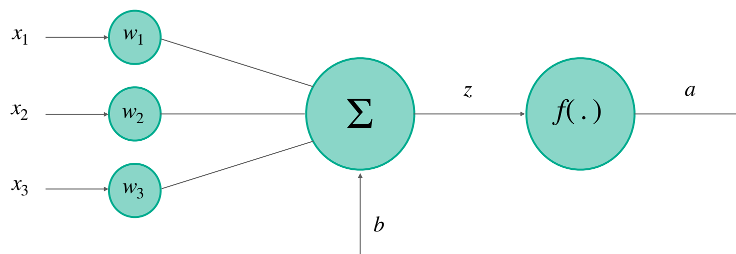


Figure 1.2. The representation of a single artificial neuron, also called *perceptron*. Each of the 3 inputs is associated with a weight. The perceptron computes the weighted sum of the inputs, adds in the bias, and sends the results through an activation function.

When several perceptrons are aggregated into a single layer, i.e. they all take the same input values, they are called a Fully-Connected Layer (FCL), motivated by the fact that all of the inputs are connected to all of the neurons. The weights of such a layer are thus a concatenation of each perceptron weight vector, which gives a weight matrix, describing how each input is connected to each output.

1.2.3 Training

A MLP with L layers takes in two set of parameters :

$$(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, ..., W^{(l)}, b^{(l)}, ...W^{(L)}, b^{(L)}) \tag{1.2}$$

where $W^{(l)}$ contains the weight matrices of layer l and $b^{(l)}$ the biases.

The training phase consists in finding values for the parameters (W, b) of all the layers in the network so that the network's output given the input training data is as close as possible to its targets. The main challenge is to avoid the network to *overfit*, i.e. adapt too much to the training data, but to learn general features, that can be applied to unseen data. The training is performed iteratively by computing the output of the network for a given input and updating the weights so that it minimizes the so-called *loss function*, defined as :

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}_i, y_i) \tag{1.3}$$

where $\mathcal{L}$ denotes the loss function, $\hat{y}$ the network's predictions, and y the ground truth labels, computed for n input data.

The purpose of this loss function is to measure how far the current output of the network is from the target output and to use this distance score as a feedback signal to adjust the values of the weights. The algorithm in charge of adjusting the weights is called Stochastic Gradient Descent (SGD), which updates trainable parameters according to their gradient value for each training iteration as:

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} \tag{1.4}$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial b_{i}^{(l)}} \tag{1.5}$$

where α is the learning rate, $w_{ij}^{(l)}$ denotes the weights between neuron j in layer l and neuron i in layer $l+1$ and $b_i^{(l)}$ denotes the bias added to neuron i in layer $l+1$.

Numerically evaluating the analytical expression for the gradient of each parameter can be computationally expensive. However, this can be resolved by using the *backpropagation* algorithm, computing the gradient for the parameter of the last layer, then progressively going back in the network, computing gradients with the so-called *chain rule*. The value of the gradient for a weight $w_{i,j}^{(l)}$ at layer l, can be computed as :

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} \tag{1.6}$$

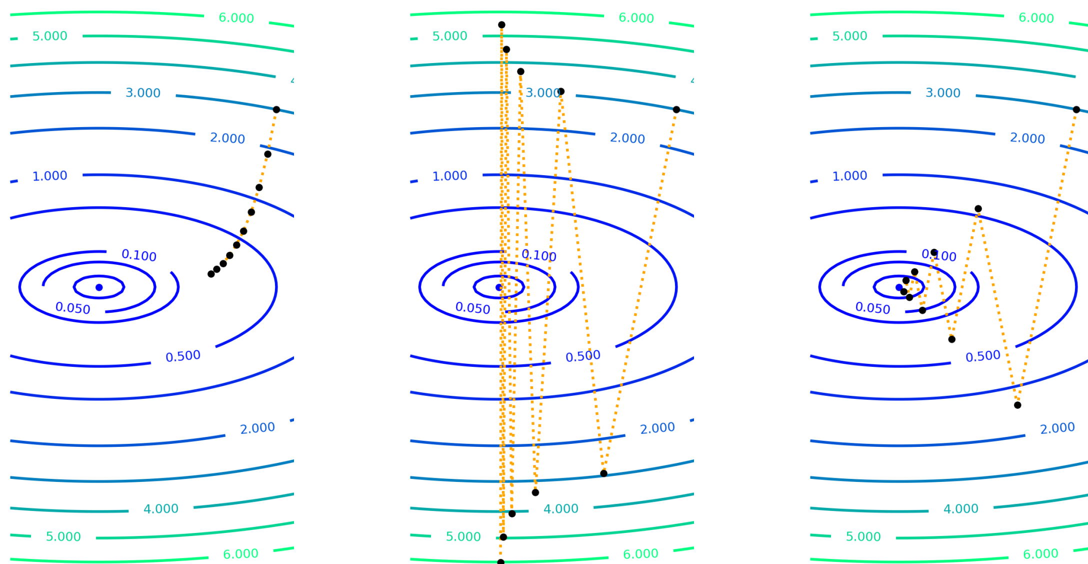
with $z_j^{(l)}$ the output of the neuron j, as defined in equation 1.1. After having computed the gradient, we can update each weight according to equations 1.4 and 1.5.

Training a neural network thus consists of two phases: (1) A **forward pass**, where a subset of training data, namely a *batch*, is fed to the network, producing the output values from which the loss value can be computed; (2) a **backward pass**, computing the gradient of each parameter and updating them accordingly. The combination of forward pass and backward pass on the entire training dataset is called an *epoch*. Several epochs are usually required in order to ensure the proper convergence of the network's weights and generalization on the test set.

Learning Rate

The learning rate α in Equation 1.5 is an essential hyperparameter to set when training a neural network as it determines the size of the steps of updates of each parameter in the network. As can be observed in Figure 1.3a, when the learning rate is set too small, tiny updates are performed to the parameters, leading to slow convergence of the training. On the other hand, if the learning rate is set too large, the updates will be substantial, leading to divergent behavior in training, as represented in Figure 1.3b. The goal is thus to find the learning rate value so that loss updates allow to rapidly converge towards an optimal value, such as represented in Figure 1.3c.

For a long time, the learning rate value was determined by trial and error. However, recent research developed good practice to identify an optimal value. In particular, the *learning rate range test* is nowadays a ubiquitous method to find such learning rates [17]. This method consists in training a network with a very small learning rate, then gradually increasing it during the training. By observing the evolution of the loss value when such a technique is applied, we can usually observe the behavior represented in Figure 1.4.



(a) Example of a learning rate chosen too small.

(b) Example of a learning rate chosen too large.

(c) Example of a well-chosen learning rate.

Figure 1.3. Examples of impact of learning rate values on convergence of SGD. The optimal value greatly depends on the architecture and dataset, making it a difficult hyperparameter to set.

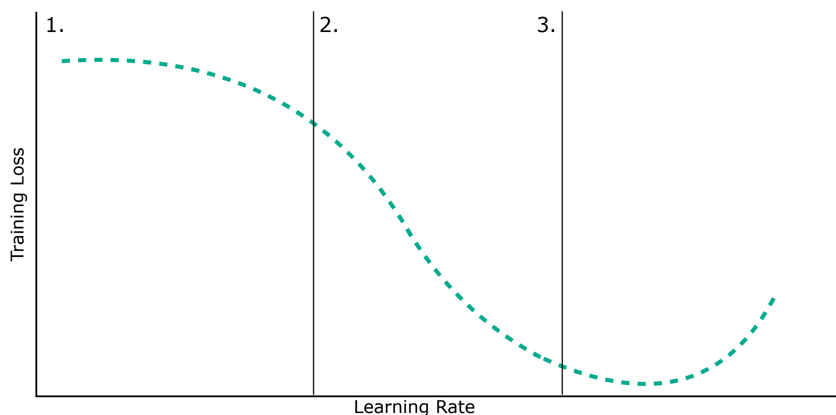


Figure 1.4. The Learning Rate Range Test. It consists in training a model starting from a very small learning rate and gradually increasing its value. The area of greatest descent in the loss is where the optimal learning rate value is located.

It consists of three parts:

1. The first part is when the learning rate is too low, leading to tiny updates and thus, not much improvement in the loss value;
2. The optimal range, where updates lead to steep decrease in the loss value;
3. The last part, where the learning rate begins to be too high for the training, leading to an increasing loss value.

The optimal learning rate value is thus located in the optimal range, ideally the highest value that can be afforded.

1.3 Convolutional Neural Networks

A Convolutional Neural Network is a particular kind of the Multi-Layer Perceptron model. This kind of network takes inspiration from how the visual cortex processes neurobiological signals [18]. Indeed, the visual cortex has small regions of cells that are sensitive to specific regions of the visual field. In particular, CNNs have been created for natural signals that present three major properties:

- **Locality**, i.e. there is a strong local correlation between close values of the signal;
- **Stationarity**, i.e. similar features appear several times in the signal;
- **Compositionality**, i.e. the features hierarchically compose the signal.

1.3.1 Definition

CNNs adds two types of layers to the standard MLP layers: a *convolution* layer and a *pooling* layer. The idea behind these two layers is to extract the local structures of the signal. Early layers of a CNN are able to extract elementary structures, e.g. edges, corners, gradients of color, while deeper layers are able to extract more complex features, e.g. faces or eyes, because they combine previously extracted features.

From a high-level point of view, a CNN is architecturally separated into two parts, each designed to fulfill a different purpose. The first part of the network, containing only convolutional layers and pooling layers, is aimed at performing feature extraction. The second part uses fully-connected layers and performs classification based on the extracted features. A complete network is illustrated in Figure 1.5.

A CNN is not always used to perform classification and it can be used to do feature extraction only. In this case, the second part of the network is not required.

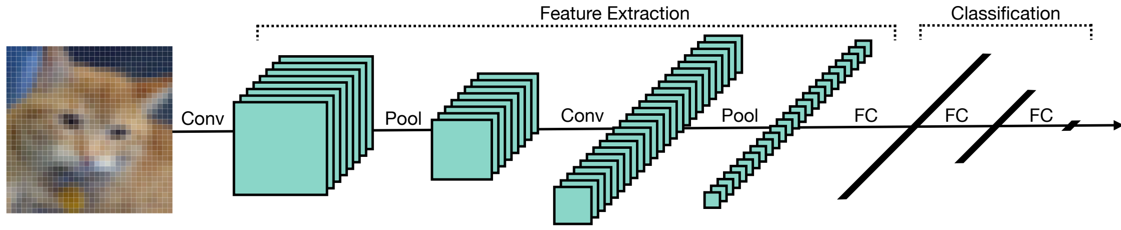


Figure 1.5. Representation of the LeNet-5 CNN [19]. It is composed of several convolution layers (Conv) and pooling layers (Pool), connected to an MLP, with Fully-Connected layers (FC) .

1.3.2 Convolutional Layer

The CNNs derive their names from the convolution operator. Convolution layers act as filters and their purpose is to extract a set of features from input images.

The extraction of features is done by performing a convolution operation on the image, with a *convolution filter*. The filter contains a set of kernels, which are matrices composed of trainable weights. Depending on the values of those weights, the kernel extracts different features from the input images, producing the so-called *feature map*. This feature map is then usually sent to a non-linear activation function, producing the *Activation Map*.

For an input image I of dimension $w \times h \times d$, with w the width, h the height and d the depth of the image (1 for a grey image and 3 for a color image, corresponding to the RGB channels) and a filter composed of d kernels K of dimension $K_h \times K_w$, with d being equal to the depth of the image, the filter slides over and convolves with the image, producing a feature map. The convolution result is the sum of the element-wise multiplication of the filter weights W and the original image, computed as:

$$z_{ij} = \sum_{a=\frac{-K_h}{2}}^{\frac{K_h}{2}} \sum_{b=\frac{-K_w}{2}}^{\frac{K_w}{2}} W_{ab} x_{(i+a)(j+b)} \tag{1.7}$$

The convolution operation is more usually written as:

$$z = W * x \tag{1.8}$$

With $*$ denoting the convolution operation. An example of the application of the convolution operation with a convolution filter computing an average value is represented in Figure 1.6.

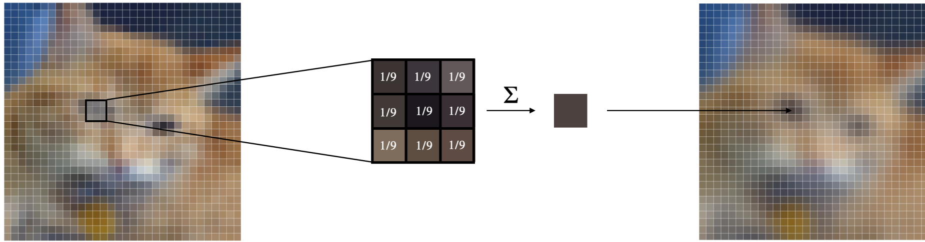


Figure 1.6. Application of the convolution filter to an input image. The convolution filter slides over the whole image and computes the sum of element-wise multiplication between the filter and the corresponding pixels. In this case, the filter is computing an average, leading to an image that is blurred.

The step size of successive convolution operation is called the *stride* and is a parameter of convolutional layers. Using a larger stride thus computes fewer activation pixels and results in a smaller sized output. Even with a unitary stride, the dimension of feature maps decreases as we progress in the network, as pixels on borders are lost for a kernel dimension large than 1×1. In order to keep our image dimension intact, we can add *padding*, i.e. add rows and columns on the input image so that the output dimensions are as expected. There exist several types of padding, such as repeating the nearest values, or doing a reflection of the image but the most commonly used is the *zero-padding*, which adds rows and columns of zero value pixels on each side of the image. The output dimensions $O_h \times O_w$ of the feature map after convolution are determined by:

$$O_h = \frac{h - K_h + 2p}{s} + 1, \tag{1.9}$$

$$O_w = \frac{w - K_w + 2p}{s} + 1 \tag{1.10}$$

with p the padding value, i.e. the number of rows and columns that are added to the image.

1.3.3 Pooling Layer

After a convolution layer, a pooling layer can sometimes be added. This additional layer does not have trainable weights but is only concerned with reducing the spatial dimensions of the image. By downsizing the images, the pooling layer allows the following convolutional layer to operate on a downsampled image and thus, to take effect on a virtually larger input size.

To do so, a pooling layer first subsamples the feature map into several submatrices, then performs a selection operation on each submatrix. Several selection methods such as average or maximum have been providing good performance. and are nowadays the most commonly used. The example of a max-pooling operation with a filter of dimension 2×2 and a stride of 2 is represented in Figure 1.7.

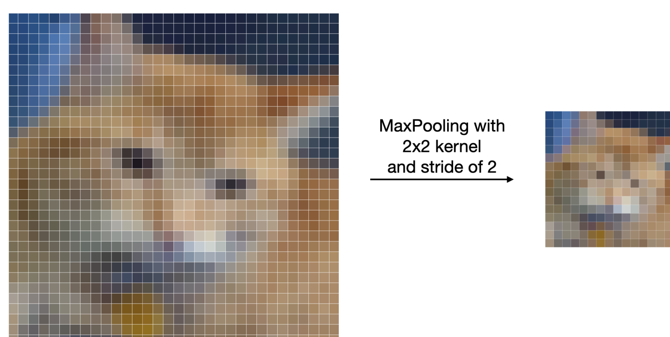


Figure 1.7. Result of a max-pooling operation with a 2×2 kernel and stride of 2. The resulting image consists in the maximum value contained for each position of the max-pooling kernel.

1.3.4 Training

As for MLPs, the general objective when training a CNN is to minimize the network cost function. By training our CNN, each filter learns the weights that allows it to extract relevant features. Each parameter in the convolutional layers is updated by following the same methods as described in Section 1.2.3. Because of the assumptions made about input data, i.e. locality, stationarity and compositionality, CNNs usually possess fewer parameters than MLPs. As a result, their training is often easier as fewer parameters make up for an easier convergence. Such assumptions about input data, allowing to adapt a model are called *inductive biases*.

1.4 In Brief

Summary 1

- The field of Artificial Intelligence has always **fascinated** human kind. We may be closer than ever to achieving Artificial General Intelligence.
- Multi-Layer Perceptrons are the most basic kind of neural network. Although being simple, they are already **sufficient to approximate any function**, provided that they have enough parameters.
- Multi-Layer Perceptrons learn their weights with the use of Stochastic Gradient Descent, using a loss as a feedback signal to **adjust their weights** to fit the training data better.
- Convolutional Neural Networks is a particular architecture of Multi-Layer Perceptrons involving Convolution Filters as weights.
- Convolutional Neural Networks have been designed under 3 assumptions, making them more efficient on input data that present: (1) **locality**, i.e. a strong local correlation between close values; (2) **stationarity**, i.e. similar features appearing several times; (3) **compositionality**, i.e. the features hierarchically compose the signal.

CHAPTER 2

Neural Network Compression

Contents

2.1 Introduction . . . 24
2.2 Sparse Neural Networks . . . 25
2.2.1 Designed Sparsity . . . 25
2.2.2 Learned Sparsity . . . 28
2.2.3 Ephemeral Sparsity . . . 31
2.3 Knowledge Distillation . . . 33
2.4 Quantization . . . 35
2.4.1 Post-Training Quantization . . . 38
2.4.2 Quantization Aware Training . . . 38
2.4.3 Automatic Mixed Precision . . . 39
2.5 Compact Neural Network Architectures . . . 40
2.5.1 Matrix Factorization . . . 40
2.5.2 Batch Normalization Folding . . . 45
2.5.3 Kernel Size Reduction . . . 46
2.5.4 Channel Amount Reduction . . . 48
2.6 In Brief . . . 50

“Pluralitas non est ponenda sine necessitate.”
— William of Occam

2.1 Introduction

An efficient neural network usually corresponds to: (1) a model having **few parameters**; (2) a model having a **fast inference**. Although generally related, obtaining one of those does not necessarily lead to obtaining the other.

In this chapter, we primarily rely on the first aforementioned definition of efficiency and present the compression techniques that are mainly concerned with reducing the number of parameters in a neural network. In particular, those compression techniques aim to find an optimum between parameter amount and generalization capacity. This corresponds to finding Pareto-optimal models, such that no model gets a better accuracy for a given parameter amount or no model can have fewer parameters for a given accuracy, as represented in Figure 2.1.

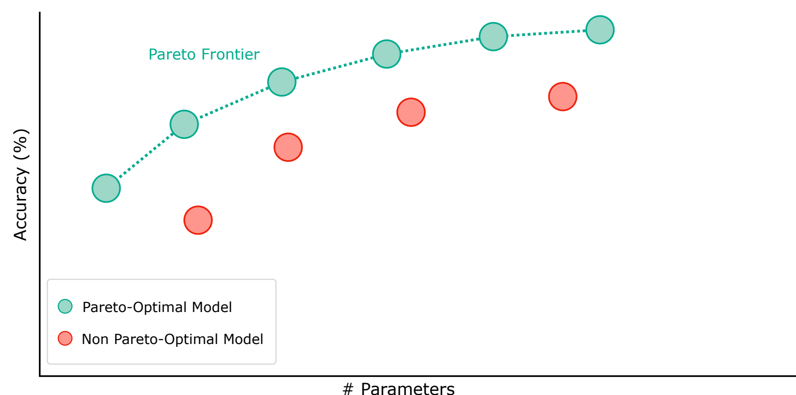


Figure 2.1. Pareto Optimality. The Pareto Frontier represents the limit at which no model can either be more accurate or with fewer parameters.

There are thus two ways to create more efficient models: (1) Reduce the number of parameters while keeping the performance unaltered; (2) Improve the performance while maintaining the same number of parameters.

The different techniques presented in this chapter are concerned with the first manner of generating models closer to the Pareto frontier, i.e. reducing the number of parameters. In particular, we expose the different manners to create sparse neural networks in Section 2.2. Then, the technique of Knowledge Distillation is presented in Section 2.3. Subsequently, the Quantization technique is detailed in Section 2.4. Finally, recent techniques for creating efficient neural network architectures are presented in 2.5.

2.2 Sparse Neural Networks

A first set of techniques to compress neural networks concerns the introduction of sparsity in the weight matrices, i.e. replacing elements with zeros. In particular, we identify three main ways to create sparse models: (1) **Designed Sparsity**, i.e. sparsity in the weights that is willingly imposed when designing the architecture; (2) **Learned Sparsity**, i.e. sparsity that first undergoes a phase of learning in order to be applied; (3) **Ephemeral Sparsity**; i.e. sparsity being imposed in a reversible way.

2.2.1 Designed Sparsity

Sparsity in the weights can be introduced at the design step of a neural network architecture. This usually results from assumptions about the data, i.e. inductive biases, which translates to simpler architectures. In particular, we illustrate this phenomenon for the convolution operation.

Convolution as a particular case of fully-connected layer

In Section 1.3, we described the three properties of the input signal that allowed to create the convolution operation: (1) **Locality**, i.e. a strong correlation between values that are close in the signal; (2) **Stationarity**, i.e. similar features can appear several times in the signal; (3) **Compositionality**., i.e. features compose the signal in a hierarchical manner.

Introducing such inductive biases in the operation allows convolutions to perform better than regular fully-connected layers when the aforementioned assumptions are met on the training data. The introduction of convolutions and the apparition of CNNs thus lead to more simply connected neural networks, having sparse connectivity when compared to Fully-Connected Networks (FCNs).

The convolution operation can thus be seen as a particular case of a fully-connected layer, where sparsity has been imposed by design. As an illustration, we describe how a Fully-Connected layer can be re-expressed when considering the three hypotheses about input data. For readability, we will show here how simplifications can be made in the case of 1D-convolutions, i.e. when the training data is one-dimensional, e.g. a monophonic audio signal, and will omit the bias term. In the case of a fully-connected layer, the operation performed on the input data is a dot product between the layer's weight and the data itself. It is usually written as:

$$W \cdot \mathbf{x} = \mathbf{y}$$

with W, the weights of the layer, $\mathbf{x}$ the input data and $\mathbf{y}$, the resulting output. As seen in Section 1.2, a perceptron containing n weights takes a n-dimensional input, and performs a dot product between its weights and the input. If the output of that layer is m-dimensional, the weight matrix is thus of shape $m \times n$. The previous operation can be extended to:

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \cdots & w_{2,n} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \cdots & w_{3,n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & w_{m,3} & w_{m,4} & \cdots & w_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

The first property, **locality**, suggests that data points close to each other are strongly correlated, while the correlation to points that are far away is negligible. As a result, we can remove the interaction between data points that are far away from each other by restricting the weight matrix to compute only around a neighborhood. This translates to a sparse weight matrix, where data points that are far from each other are not connected. By introducing this property on the previously obtained weight matrix, we have:

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \cdots & w_{1,k} & 0 & 0 & \cdots & 0 \\ 0 & w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,k} & 0 & \cdots & 0 \\ 0 & 0 & w_{3,1} & w_{3,2} & w_{3,3} & \cdots & w_{3,k} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & w_{m,n-k} & \cdots & w_{m,n-2} & w_{m,n-1} & w_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

The resulting operation is called a **locally-connected layer**, where weights are only affected by its k closest data points, while having no impact on other data points that are further away.

The second property, **stationarity**, states that similar patterns can appear in several places in the signal. This implies that the same feature extractor can be used several times in the weight matrix. As a result, a weight-sharing technique can be applied, so that the same set of weight is repeated throughout the matrix. Applying such a property results in the following operation:

$$\begin{bmatrix} w_1 & w_2 & w_3 & \cdots & w_k & 0 & 0 & \cdots & 0 \\ 0 & w_1 & w_2 & w_3 & \cdots & w_k & 0 & \cdots & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & \cdots & w_k & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & w_1 & \cdots & w_{k-2} & w_{k-1} & w_k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

This results in a Toeplitz weight matrix, possessing k, with $k \leq n$, trainable parameters instead of the initial $m \times n$.

When combining the two first data properties to modify the fully-connected weight matrix accordingly, the latter now behaves as a sliding window, significantly reducing the number of trainable parameters in the layer. The obtained convolution operation is usually expressed by:

$$W * \mathbf{x} = \mathbf{y}$$

with $*$ being the convolution operator.

The Figure 2.2 graphically summarizes the consequence of the locality and stationarity hypothesis on a fully-connected layer.

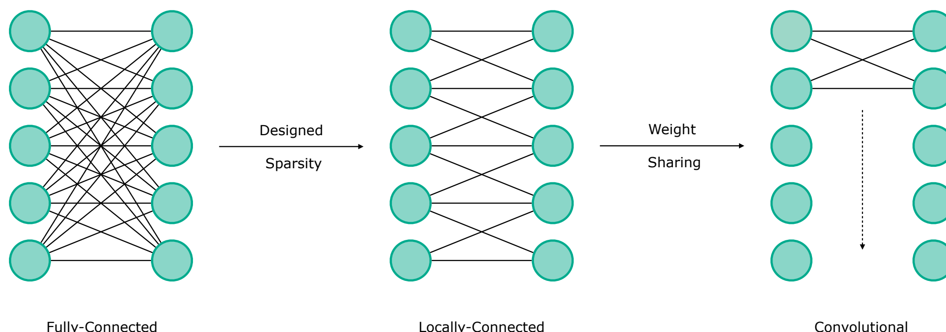


Figure 2.2. Convolution as designed sparsity. Each assumption about input data allows to simplify and sparsify the original fully-connected network. Image inspired by [20].

The last property, **compositionality**, has no impact on the individual operation but supports the rationale behind the stacking of several convolutional layers in order to create a convolutional neural network. Indeed, feature extraction in a CNN is performed hierarchically, with shallow layers extracting low-level information about the input data, while deeper layers are able to build upon features extracted previously to discover high-level information. An illustration of such a phenomenon is provided in Figure 2.3.

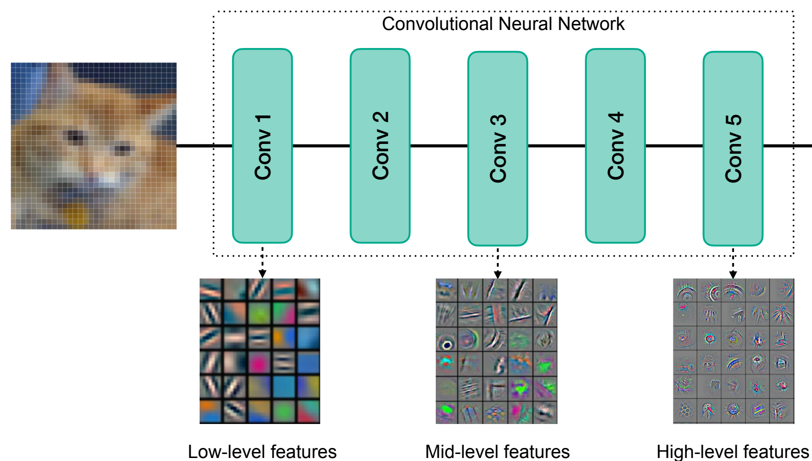


Figure 2.3. Convolutional Neural Network extracting data features in a hierarchical manner, illustrating the compositionality property of input data. While shallow layers allows to extract low-level features, deeper layers extract higher-level features.

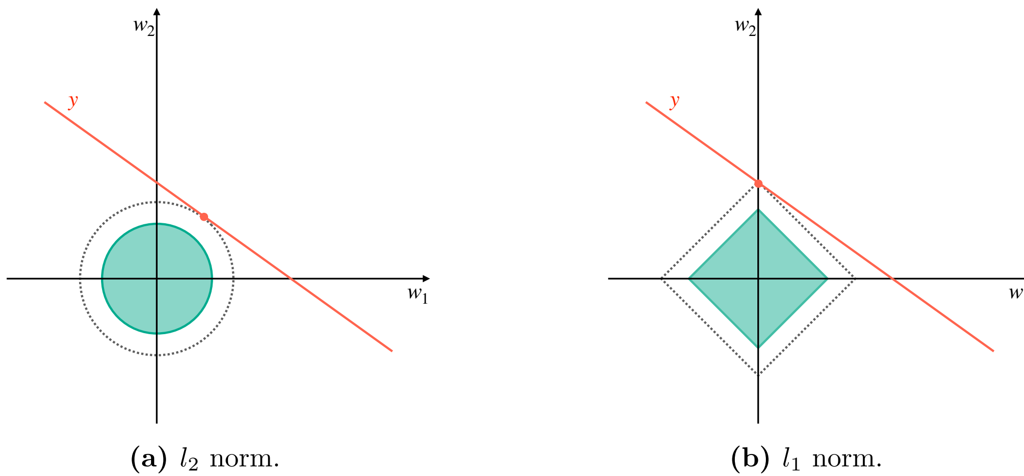
2.2.2 Learned Sparsity

Instead of being explicitly designed in the architecture, sparsity can be introduced in the training process. This results in more flexible methods as fewer assumptions about the data are made. In particular, we present three methods to learn sparse models: weight decay, pruning and growing.

Weight Decay

As the name suggests, **weight decay** is a technique that concerns the reduction of the weight magnitudes in a neural network. It is generally associated to imposing a l_2 penalty on the weights, computed as $\|W\|_2 = \sqrt{\sum_i |w_i|^2}$, and was historically directly specified in the weight update rule. However, since the introduction of more complex optimization processes, the penalty is preferably imposed in the loss function [21].

Instead of penalizing the weights based on their l_2 norm, we can use the l_1 norm. While also reducing the magnitude of weights, l_1 norm has the convenient property of acting as a feature selection method, by introducing sparsity in the weights. This can be graphically interpreted when solving a problem in a 2D space, as represented in Figure 2.4. In that scenario, as the constraint region imposed by the l_1 penalty possesses corners, the solution is likely to happen at one of those corners, producing a sparse solution, i.e. one of the parameters w_1 or w_2 is equal to 0. This is not the case for l_2 penalty, where sparse solutions are less likely.



(a) l_2 norm. **(b)** l_1 norm.

Figure 2.4. Representation of the solution imposed by different types of norms. While l_2-norm rarely leads to sparse solutions, the sharp edged shape imposed by l_1-norm often leads to solution that are sparse.

In practice, the regularization term is added to the loss value $\mathcal{L}$, with a hyperparameter λ mitigating its effect, as:

$$\tilde{\mathcal{L}} = \mathcal{L} + \lambda R(W) \tag{2.1}$$

with, in the case of l_1 regularization:

$$R(W) = \|W\|_1 = \sum_i |w_i| \tag{2.2}$$

Pruning

Instead of encouraging weight values to decay towards zero, it can be done in a more strictly, directly replacing certain weights with zeros. This process is called parameter **pruning** and thus removes connections in a neural network. Pruning methods have been a popular way to introduce sparsity in neural networks in recent years. Such a technique allows for obtaining a final sparse neural network which usually reaches non-trivial levels of sparsity without witnessing performance degradation. The simplified process is represented in Figure 2.5.

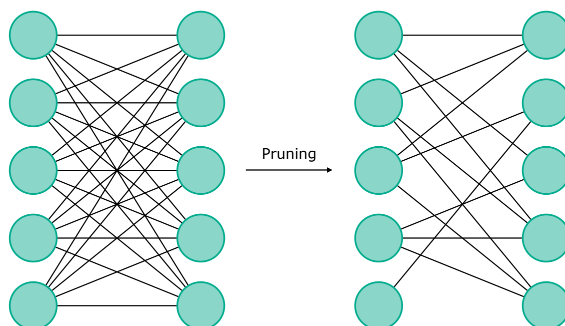


Figure 2.5. Pruning connections in a neural network. By starting from an overparameterized network and removing connections, a sparse network can be obtained.

Growing

The method followed by most compression techniques is to start with a network that is supposed to be overparameterized with a too large capacity and then to reduce the number of parameters to obtain a capacity that can reach the optimal trade-off between performance and model size. There exist techniques that allow to reach such a trade-off by turning the problem upside down. Instead of starting from a network having too many parameters, they start from a network having too few. The goal is then to progressively add parameters until the desired performance is obtained. Such a technique is called neural network **growing** as we grow weights during the training. The general concept of network growing is represented in Figure 2.6.

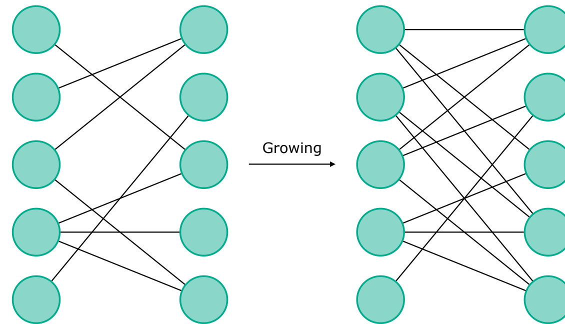


Figure 2.6. Growing connections in a neural network. By starting from an extremely sparsely-connected network and gradually adding connections, a sparse network can be obtained.

2.2.3 Ephemeral Sparsity

Sparsity can also be introduced in an ephemeral manner which, as opposed to other sparsity methods, restores values that have been sparsified after training. Such an operation is usually not performed with the purpose of compressing a neural network, but rather to improve the generalization performance of a model, leading to models closer to the Pareto-optimality.

Dropout

The most common ephemeral sparsity technique is the so-called **Dropout**. It introduces sparsity at training time to decrease neuron *co-adaptation*, i.e. neuron units having too highly correlated behavior. In practice, it is performed by deactivating some neuron in the training phase, along with all its incoming and outgoing connections, with a defined probability p. By this means, neurons cannot rely on their neighbors as they might be deactivated at any moment, forcing neurons to learn good feature extractions on their own, and increasing the model robustness.

For a neural network composed of n neurons, there are 2^n possible subnetworks, all sharing a large portion of weights. At each training iteration, Dropout thus samples a subnetwork, consisting of the units that survived. As a result, it provides a way of efficiently combining exponentially many different neural network architectures, thus acting as a sort of *model ensembling* [22]. A representation of Dropout is provided in 2.7.

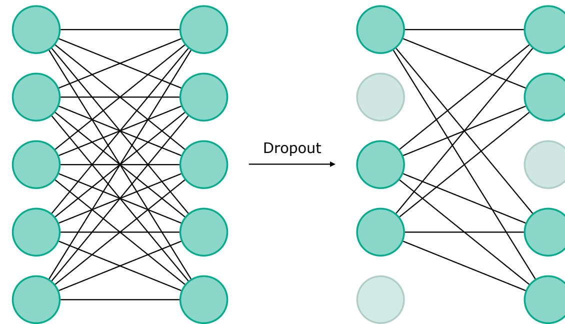


Figure 2.7. Illustration of the Dropout technique, introducing sparsity in the neurons. Greyed Neurons are temporarily dropped and do not participate in the current forward pass, forcing neurons to not rely on other ones.

DropConnect

A generalization of the Dropout technique is called **DropConnect**. Instead of setting complete neuron units, DropConnect randomly drops weight connections, allowing for greater flexibility. The principle is represented in Figure 2.8.

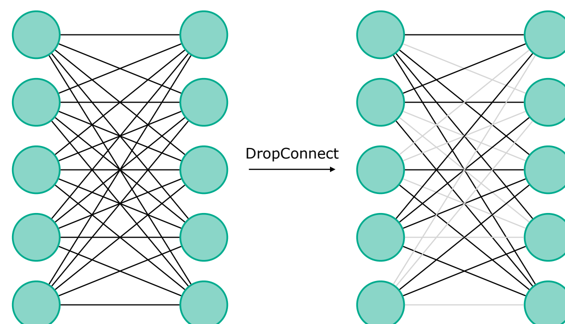


Figure 2.8. Illustration of the DropConnect technique, introducing sparsity in the connections. Greyed connections are temporarily dropped and do not participate in the current forward pass.

DropBlock

Instead of operating on the weights, several methods have been proposed to zero-out parts of the feature maps. The **Cutout** method first introduced such an operation, randomly dropping input pixels [23]. It was later generalized by **DropBlock**, removing blocks of feature maps from the whole network. The idea consists of dropping sections of the input image, such that those dropped sections are propagated across all layers, leading to a final representation of the image, containing no information about the missing section other than what can be recovered from the context [24]. An example of such an operation is represented in Figure 2.9.

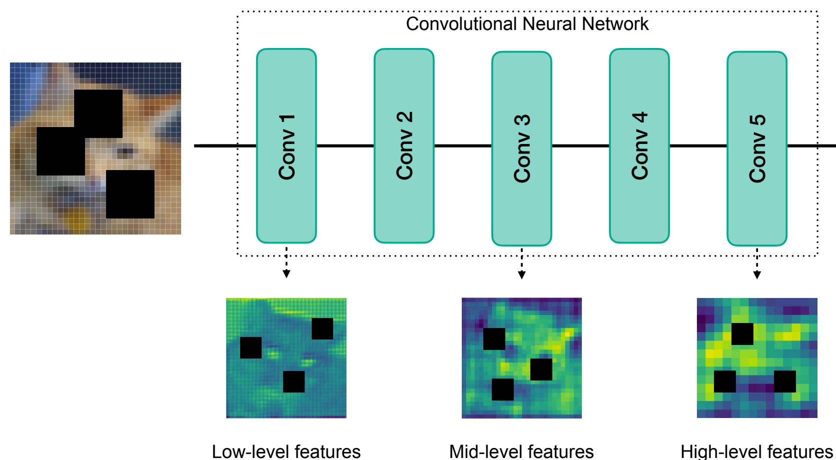


Figure 2.9. DropBlock technique introducing sparsity in the activations. Such techniques remove pixels from input image or activation maps, forcing the neuron to recognize generic features about the images.

2.3 Knowledge Distillation

The **Knowledge Distillation** method is a compression technique involving a student-teacher based training. In such a training, a large and performant model, the *teacher*, guides a small and less performant model, the *student*, in its learning process [25]. This is achieved by encouraging the student network to replicate the teacher model's predictions, i.e. the *logits*, and the correct predictions coming from the dataset, i.e. the ground-truth *labels*. The rationale behind this teacher-student training is to provide inter-class information, called the *Dark Knowledge* [26], that is not present in hard-labeled data, and is usually discarded when using a softmax activation function.

Initially, Knowledge Distillation was applied by comparing the predictions of the teacher and the student after passing through an altered softmax function. This new function is similar to the basic softmax function but possesses a temperature parameter T, parameterizing how much inter-class information is retained. The equation of such an operation is given by:

$$p_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j^n \exp\left(\frac{z_j}{T}\right)} \tag{2.3}$$

with p_i the prediction value for a given class, z_i the logit value corresponding to that class and j the amount of different classes. When the temperature T is set to 1, then the previous equation is equivalent to a regular softmax.

Knowledge Distillation was later extended to the different set of techniques involving a teacher-student training, which can use computations coming from different places in the network. The general Knowledge Distillation concept is illustrated in Figure 2.10.

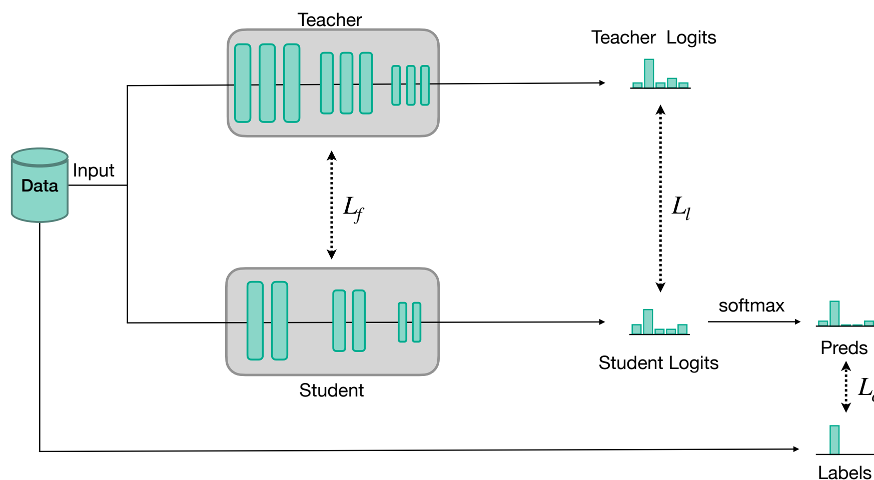


Figure 2.10. Illustration of the Knowledge Distillation Concept. The student model is guided in its learning phase by taking hints from features and/or predictions of the teacher, but still uses labels from the data.

In that case, we are able to compare the logits from the teacher and the students according to the desired loss function L_l, which we call the *logits loss.* Moreover, intermediate states from the networks can also be compared, e.g. the activation maps and attention maps, between which is applied the L_f loss function, which we call the *feature loss*. The classification loss L_c computed between the student predictions and the hard labels coming from data is still applied to ensure the good convergence of the

student network.The global loss, used to train the student, can thus be expressed in its general form as:

$$\mathcal{L} = \beta(\alpha L_l + (1-\alpha)L_f) + (1-\beta)L_c \tag{2.4}$$

It consists of a linear interpolation between the classification loss L_c and the distillation loss, itself being an interpolation between the logits loss L_l and the feature loss L_f.

2.4 Quantization

Quantization of a neural network is the process of reducing the numerical precision of its weights or activations. Most of the time, the storage of the weights, as well as the computation of activations is performed using the single-precision floating-point format, i.e. each value is encoded with 32 bits. Reducing this precision therefore saves processing time and storage. However, this usually results in a lossy process. Indeed, as represented in Figure 2.11, quantization reduces the representation possibilities, by squeezing a small range of floating-point values into a fixed number of information channels [27].

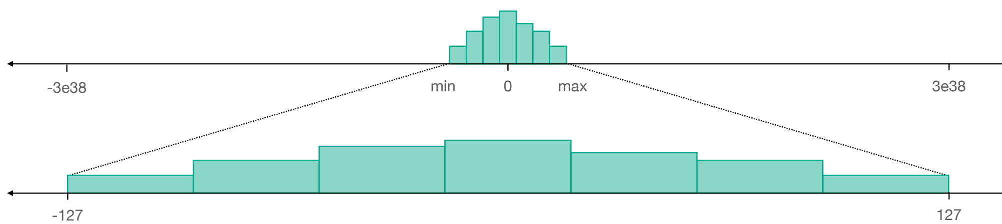


Figure 2.11. Mapping of a small range of floating-point format `float32` to a fixed-point format `int8`. Quantized representation only has 255 information channels, leading to loss of precision. Image inspired by [27].

A floating-point vector $\mathbf{x}$ can approximately be expressed as a transformation of its quantized values $\mathbf{x_q}$ as:

$$\widehat{\mathbf{x}} = s \cdot (\mathbf{x}_\mathrm{q} - z) \approx \mathbf{x} \tag{2.5}$$

where s is a floating-point scaling factor and z the zero-point, which are used to map the floating point value to the quantized grid.

There exist several schemes of quantization:

- **Asymmetric quantization**, which is the most commonly used scheme as it allows an efficient implementation of fixed-point arithmetic. After setting the scale parameter s, the zero-point z and the bit-width b, we can first map the real-valued vector x to the unsigned quantized grid $0, ..., 2b - 1$ as:

$$\mathbf{x}_\mathrm{q} = \mathrm{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right) \tag{2.6}$$

  where $\lfloor \cdot \rceil$ is the round-to-nearest operator and clamping is defined as:

$$\mathrm{clamp}(x; a, c) = \begin{cases} a, & x < a \\ x, & a \leq x \leq c \\ c, & x > c \end{cases} \tag{2.7}$$

  which leads to a final representation of the vector $\mathbf{x}$ as:

$$\widehat{\mathbf{x}} = s\left[\mathrm{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil + z; 0, 2^b - 1\right) - z\right] \tag{2.8}$$

  The grid limits are thus set to $[-sz, s(2^b - 1 - z)]$. Values outside of this range will be clipped, incurring a clipping error. This error can be reduced by expanding the quantization range. However, this requires to increase the scale factor, thus increasing the rounding error.

- **Symmetric Quantization** is a simplified version of the general asymmetric case, where the zero-point z is set to 0, which slightly reduces the computational overhead. Furthermore, the symmetric quantization can either be signed, in which case the quantization operation is defined by:

$$\mathbf{x}_\mathrm{q} = \mathrm{clamp}\left(\left\lfloor \frac{\mathbf{x}}{s} \right\rceil; -2^{b-1}, 2^{b-1} - 1\right) \tag{2.9}$$

  which leads to a final representation of the vector $\mathbf{x}$ as:

$$\widehat{\mathbf{x}} = s\left[\text{clamp}\left(\left\lfloor\frac{\mathbf{x}}{s}\right\rceil; -2^{b-1}, 2^{b-1}-1\right)\right] \tag{2.10}$$

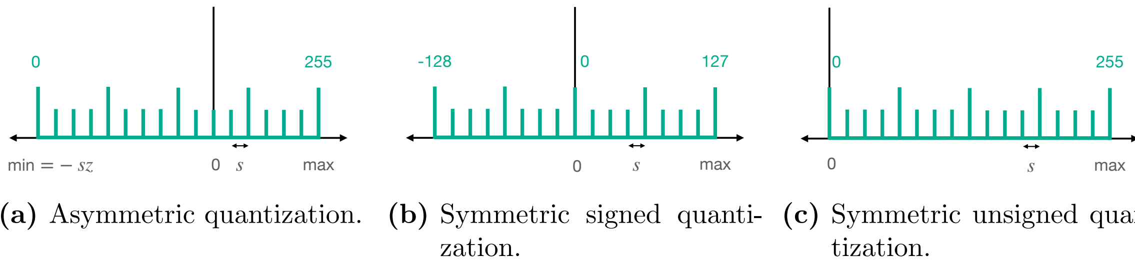
Or unsigned, in which case the quantization operation is defined by

$$\mathbf{x}_{\text{q}} = \text{clamp}\left(\left\lfloor\frac{\mathbf{x}}{s}\right\rceil; 0, 2^{b}-1\right) \tag{2.11}$$

which leads to a final representation of the vector $\mathbf{x}$ as:

$$\widehat{\mathbf{x}} = s\left[\text{clamp}\left(\left\lfloor\frac{\mathbf{x}}{s}\right\rceil; 0, 2^{b}-1\right)\right] \tag{2.12}$$

A representation of quantization grids for all quantization operations is available in Figure 2.12.



(a) Asymmetric quantization. **(b)** Symmetric signed quantization. **(c)** Symmetric unsigned quantization.

Figure 2.12. A visual illustration of the different uniform quantization grids for int8 representation. The floating-point grid is in black, the integer quantized grid in green. Image inspired from [28].

The quantization process is not only challenging to represent weight and activation values, but it also induces several problems related to the training of a neural network. In particular, the following three problems might occur when training using low-precision values:

- **Imprecise Weight Update.** When we update the weights by performing the operation $w = w - \alpha\frac{\partial L}{\partial w}$, the updating term $\alpha\frac{\partial L}{\partial w}$ is usually ranging around a value of $1e-3$ [29]. The lack of power of representation of lower precision can thus lead to imprecise updates.

- **Gradient Underflow**. As we compute the gradients with a lower precision, they can sometimes be so small that they are not representable and thus are replaced by 0, leading to no update at all.
- **Loss Overflow**. The problem of representation does not only concerns small numbers. Indeed, with lower precision, big numbers are more difficult to represent and can make the training diverge.

There are however several techniques able to cope with those problems. In the following sections, we will describe the three most important techniques:(1) Post-Training Quantization (PTQ); (2) Quantization-Aware Training (QAT) and (3) Automatic Mixed-Precision (AMP).

2.4.1 Post-Training Quantization

The most straightforward manner to avoid the problems related to training a quantized network is to train it in high precision and perform quantization afterwards. This is the principle behind the PTQ method, which quantizes the weights and activations of a pre-trained model in a single step, without the need of additional training. This method can be data-free or require a small calibration set to fine-tune the quantized network and minimize the loss of information between the high-precision network and the new low-precision one. The general concept of PTQ is represented in Figure 2.13.

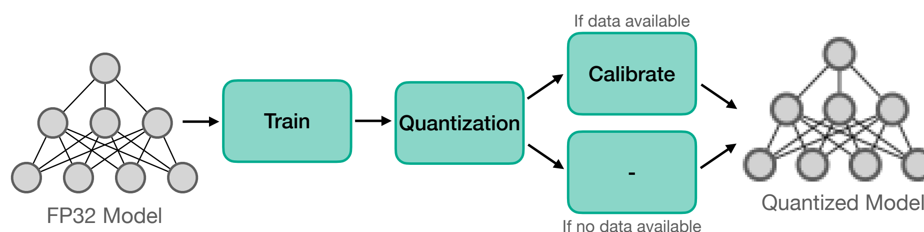


Figure 2.13. Illustration of the Post-Training Quantization process. Quantization is applied on the trained model, further requiring calibration is data is available.

2.4.2 Quantization Aware Training

When aiming for low precision quantization, PTQ can have limitations as they do not mitigate the significant quantization error induced. For that reason, QAT has been introduced to help the model to take the quantization noise source into account.

In practice, QAT is performed by introducing extra modules that emulate the quantization of each computation module. The main principle is that QAT simulates low-precision forward pass computation in the training process, introducing quantization error as noise during the training. By doing so, the weights are optimized to be more robust to quantization. The general concept of QAT is represented in Figure 2.14.

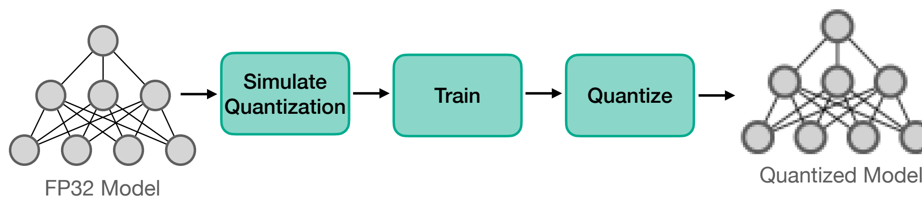


Figure 2.14. Illustration of the Quantization-Aware Training process. It emulates the quantization noise during the training, before quantizing the model.

2.4.3 Automatic Mixed Precision

AMP is an affordable way to perform quantization as it leverages the power of quantization with minor changes to the model or training loop. This method consists in training a model where compute-intensive operations are performed in low precision while critical operations such as gradient computation or backpropagation are performed in higher precision. The principle behind AMP is represented in Figure 2.15.

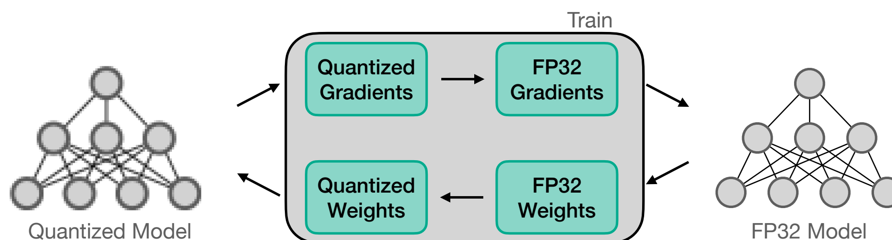


Figure 2.15. Illustration of the Automatic Mixed-Precision process. Computation of the forward pass is performed on the quantized model while the computation of the backward pass is performed on the full precision master model, helping reducing issues related to low-precision gradient descent updates.

Such a method thus allows to significantly speed up the training as considerable operations are performed using lower precision but without witnessing too much performance degradation as important computations are done with high precision.

2.5 Compact Neural Network Architectures

Several research studies have been concerned with creating more efficient architectures, i.e. architectures able to reach similar performance while minimizing the number of parameters or compute operations. In this Section, we present several design methodologies that have been used to create compact architectures.

2.5.1 Matrix Factorization

This technique consists in expressing a weight matrix by a product of smaller matrices. By replacing the initial matrix with the new product, a gain in performance can generally be achieved.

Depthwise Separable Convolution

As presented in Section 2.2.1, convolutions are the usual building block when the task involves natural data such as images. Although convolutions already are more efficient in terms of number of parameters and computations than fully-connected layers, it has been shown that they are not the optimal trade-off between performance and computational cost [30]. A stronger hypothesis about CNNs is that the activation maps of the hidden layers are mostly uncorrelated. This means that the computations in the spatial space can be separated from the ones in the channel space. As a result, an alternative to the convolution operation, called the **depthwise separable convolution**, has been proposed. It consists of two operations: (1) a *depthwise* convolution which only operates on spatial features; (2) a *pointwise* convolution, which only operates on channel features. Using this type of convolution in the MobileNet architecture allowed it to reach better accuracy of ImageNet classification than the ImageNet Large Scale Visual Recognition Challenge winners, VGG16 [31] and AlexNet [32], which use regular convolutions, for respectively 32 and 45 times fewer parameters.

In the regular convolutional layer, the convolution is performed over multiple input channels, with a filter as deep as the input image, and the channels are combined to generate each channel of the output. So, if an input image has three channels, applying a convolution operation results in an output image with only one channel per pixel, as illustrated in Figure 2.16.

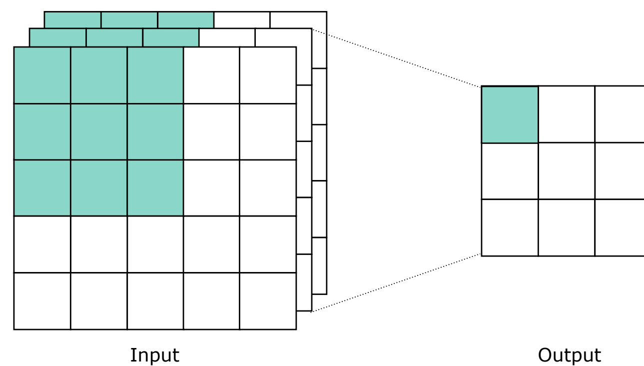


Figure 2.16. Representation of the effect of applying a regular convolution on an input image of three channels. The convolution operates on both spatial dimension and channel dimension.

Depthwise convolutions differ from regular convolutions as they keep the channels separated. As a result, for an input image with three channels, applying a depthwise convolution operation on the image results in an output image with also three channels. The principle is shown in Figure 2.17.

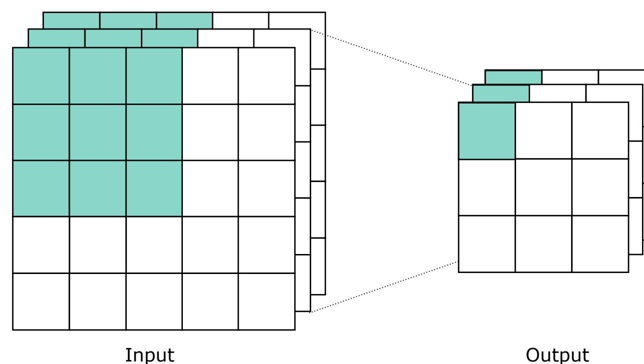


Figure 2.17. Representation of the effect of applying a depthwise convolution on an input image of three channels. The depthwise convolution only operates on the spatial dimension.

This depthwise convolution is followed by a pointwise convolution which actually is a regular convolution with a kernel of dimension 1×1. The goal is thus to combine the channels previously kept separated by the depthwise convolution to obtain a single resulting channel. The principle of pointwise convolution is shown in Figure 2.18.

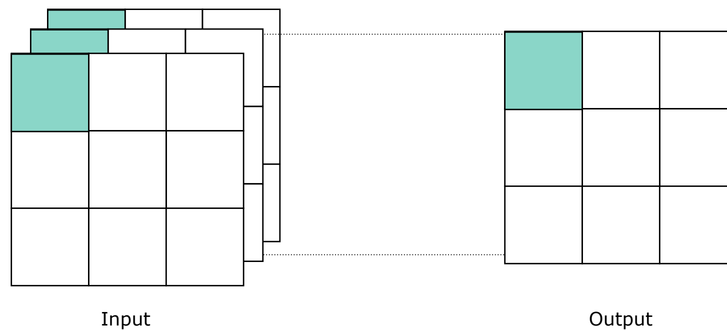


Figure 2.18. Representation of the effect of applying a pointwise convolution on an input image of three channels. The pointwise convolution only operates on the channel dimension.

We can compare the number of parameters needed to do a regular convolution to those needed for a depthwise separable convolution. The number of parameter in a standard convolution can be computed as:

$$K_h \cdot K_w \cdot M \cdot N \tag{2.13}$$

where M is the number of input channels, N is the number of output channels, $K_h \times K_w$ is the dimension of the kernel.

The total amount of parameter in a depthwise separable convolution is given by the sum of its two constituent operations. The depthwise convolution only operates on individual channels, and does not create new combinations of them, while the pointwise convolution is a regular convolution with a kernel size of 1×1. The sum of both terms gives us the following expression.

$$K_h \cdot K_w \cdot M + M \cdot N \tag{2.14}$$

The reduction of the parameter amount is given by computing the ratio between both types of convolutions, computed from Equations 2.13 and 2.14:

$$\text{Reduction} = \frac{K_h \cdot K_w \cdot M + M \cdot N}{K_h \cdot K_w \cdot M \cdot N} = \frac{1}{N} + \frac{1}{K_h \cdot K_w} \tag{2.15}$$

In most architectures, the kernel dimensions are set to 3×3. The amount of output channels is usually a factor of 8, ranging from 64 to 512. This means that we can expect a reduction of parameters of a factor close to $9\times$ by replacing regular convolution with depthwise separable convolutions. Applying the same reasoning to the amount of compute operations provides the same ratio. This means that not only depthwise separable convolution allow a become to be significantly lighter, but also faster. By implementing the MobileNet architecture with both types of convolutions, it has been observed that the depthwise-convolutional MobileNet has $7\times$ fewer parameters and $8.5\times$ fewer operations, for a drop in accuracy of only 1% on ImageNet classification compared to the regular-convolutional MobileNet [30], illustrating that depthwise separable convolutions help to move networks closer to the Pareto optimality.

Fully-Connected Layers Decomposition

Generally, CNNs are composed of two parts:

1. A **feature extraction** part using only convolutional operations.
2. A **classification** part using only fully-connected operations.

For a long time, the second part of the network has been responsible for most of the parameter count in the network. Indeed, for competition-winning networks such as AlexNet, ZFNet or VGG-16, the classification part represented around 90% of the parameters in the network, as presented in Table 2.1, thus largely dominating their global storage footprint.

	Total (M)	Conv (M)	FC (M)	% FC
AlexNet	61.10	2.46	58.63	**95.96**
ZFNet	62.36	37.26	58.63	**94.04**
VGG16	138.36	14.71	123.64	**89.36**

Table 2.1. The repartition of parameter in each part of a CNN, designed for ImageNet-1K classification.

The Singular Value Decomposition (SVD) allows expressing a large weight matrix X into the product of three smaller ones as:

$$X = U\Sigma V^T \tag{2.16}$$

with Σ the diagonal matrix of singular values, ordered by importance.

We can approximate the matrix X by selecting the leading $k \times k$ sub-blocks of Σ, and the corresponding k leading columns of U and V. The larger the value of k, the better the approximation. The concept of this truncated-SVD decomposition is represented in Figure 2.19.

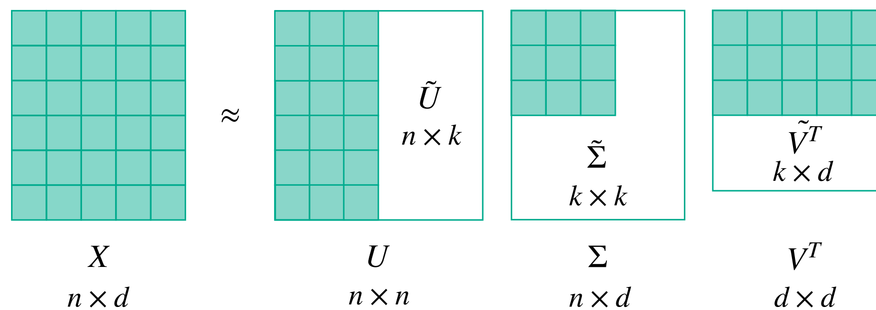


Figure 2.19. Truncated-SVD decomposition of a fully-connected layer. Keeping only a subset of singular values allows to greatly reduce the total amount of parameters.

Nowadays, architectures usually reduce the amount of fully-connected layers to its bare minimum, i.e. a single layer, reducing the benefit of using a decomposition. However, in the case of efficient architectures able to significantly reduce the number of parameters in the convolutional part, the fully-connected layer starts to dominate again the parameter count, especially when the number of classes to discriminate increases. The example chosen of MobileNetV2, SqueezeNet and ShuffleNetV2 x0.5, all designed to be parameter efficient, is provided in Table 2.2.

	Total (M)	Conv (M)	FC (M)	% FC
MobilenetV2	3.50	2.22	1.28	**36.55**
SqueezeNet	1.25	0.74	0.51	**41.09**
ShuffleNetV2 x0.5	1.37	0.34	1.03	**74.99**

Table 2.2. The repartition of parameter in each part of an already parameter-efficient CNN, designed for ImageNet-1K classification. Because the feature extraction part is parameter-efficient, the fully-connected layer represent a large part of the parameter count.

2.5.2 Batch Normalization Folding

The Batch Normalization (BN) layer is a normalization layer, usually placed between the computation layer and the activation function, and whose role is to normalize the input data [33]. More formally, the BN operation is performed as:

$$\begin{aligned} \widehat{z} &= \frac{z - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\ y &= \gamma \widehat{z} + \beta \end{aligned} \tag{2.17}$$

This normalization is performed in two steps:

1. Subtract to incoming data the mean $\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} z_i$ and divide it by the standard deviation $\sigma_{\mathcal{B}} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(z_i - \mu_{\mathcal{B}})^2}$ of input data batches. Normalizing each incoming batch to a mean of 0 and standard deviation of 1 at each computation layer has been shown to significantly improve the training performance and help obtain a better-behaved optimization process. The BN is believed to reduce the so-called *internal covariate shift*, i.e. the change in the distribution of network activations due to the change in network parameters during training [33];
2. Multiply the result by a weight γ and add a bias β, two learnable parameters of the Batch Normalization layer. Such parameters allow the data distribution to be shifted and scaled adequately, potentially undoing the previous step if needed.

In order to apply BN at inference, moving average of those statistics are computed and kept fixed once the training has ended. Those values can thus be incorporated into the computation layer preceding each of them. This can be achieved by re-expressing, the weights and the bias of the computation layer, taking the normalization effect into account. The output of a Batch Normalization layer y, given an input z is given by:

$$y = \gamma \frac{z - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta \tag{2.18}$$

With z the output of the previous computation layer, expressed as:

$$z = Wx + b \tag{2.19}$$

with W and b respectively being the weights and bias of the layer.

From the equations 2.18 and 2.19, we can re-arrange the weights W and bias b and express those terms, accounting for the parameters of the batch normalization layer as:

$$\begin{aligned} W_{\text{fold}} &= \gamma \cdot \frac{\mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\ b_{\text{fold}} &= \gamma \cdot \frac{b - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta \end{aligned} \tag{2.20}$$

By replacing the parameters W and b of the computation layer with their folded counterparts W_{fold} and b_{fold}, this computation layer now normalizes incoming data according to the computed statistics. As a result, the entire Batch Normalization layer can be considered as useless and removed from the network without impacting its performance, which provides a reduction of the number of parameters and computation.

2.5.3 Kernel Size Reduction

While the most commonly encountered kernel dimension in CNNs is 3×3, it is not unusual in recent architectures to observe kernels of larger dimensions, especially in shallow layers. The rationale behind using large kernels is to help the network to extract higher dimension features earlier. However, it has recently been found that using higher dimensions filters actually negatively impacts the parameter count while showing no improvements over smaller filters [34].

To better understand this phenomenon, let us first introduce the concept of *receptive field*, i.e the region of the input space that a particular filter is affected by. In a CNN using kernel dimensions of $K_h \times K_w$, the first layer operates on a region of the input image that only depends on its kernel size, i.e $K_h \times K_w$. The second layer then operates on regions $K_h \times K_w$ of the result of the first layer. However, when reported to the input image, this virtually corresponds to a region that is larger than $K_h \times K_w$, as each value in the second layer was already computed from a neighborhood from input image. Deeper layer thus computes on increasingly larger regions, i.e. they have a larger receptive field. This phenomenon is represented in Figure 2.20 for kernels of dimensions 3×3 and a unitary stride.

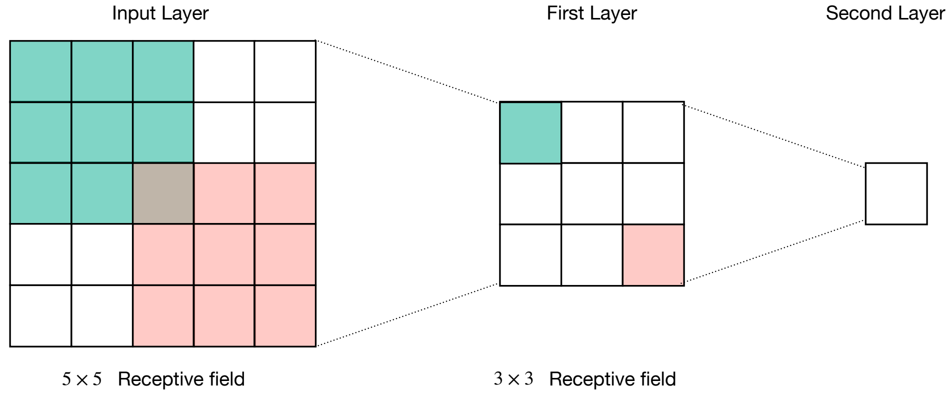


Figure 2.20. Receptive field of a pixel from the second layer, when using 3×3 filters. Each pixel of each feature from the second layer is computed from an equivalent region of 5×5 of the input image.

The size of the receptive field $\mathcal{R}^{(l)}$ of a layer l can be computed by the following expression:

$$\mathcal{R}^{(l)} = \mathcal{R}^{(l-1)} s^{(l-1)} + (K^{(l-1)} - s^{(l-1)}) \tag{2.21}$$

with s being the stride value and K the kernel dimension. We can thus replace the layer having a larger kernel by the adequate amount of layer with kernels of dimensions 3×3. Using smaller filters has two main advantages:

1. Using more layers and thus more non-linearities allows the network to extract richer and more expressive features. Indeed, the expressivity in neural networks is believed to grow exponentially with its depth [35].
2. Using smaller kernels is more parameter efficient. Indeed, if we take the example of a single layer with a 7×7 kernel and suppose that all feature maps have a number of channel C, then the total amount of parameter in that layer is $C \times 7 \times 7 \times C = 49C^2$. However this layer can be replaced by three layers of 3×3 kernels, equating to a number of parameter of $3 \times (C \times 3 \times 3 \times C) = 27C^2$.

For those reasons, kernels of dimension 3×3 should be privileged over higher resolution when model efficiency is to be taken into account.

2.5.4 Channel Amount Reduction

In a Convolutional Neural Network layer, the number of channels is another hyperparameter to set. A widely adopted heuristic is to set channels to a power of two, with the number of channels increasing as going deeper in the network, as extracted features are incrementally more specific. The challenge thus resides in finding the correct amount of channels, avoiding unnecessary computations and storage. Several recent research propose to create bottlenecks in the network to be more parameter efficient. In particular, two types of bottleneck modules have emerged: (1) Inverted Residuals; (2) Fire Module.

Inverted Residuals

The Inverted Residual module has been proposed for the MobileNetV2 architecture [36]. While the MobileNetV1 proposed to use Depthwise Separable Convolutions to replace regular convolutions as detailed in Section 2.5.1, the second version proposes to go one step further.

The Inverted Residual module takes in a low-dimensional tensor with n channels and performs three separate convolutions, as represented in Figure 2.21. The first is a pointwise convolution, expanding the low-dimensional input tensor to a higher-dimensional space. The expansion factor is here referred to as ϵ, leading to ϵn channels. The second operation is a depthwise convolution, only achieving spatial filtering of the higher-dimensional tensor. The spatially-filtered feature map is then projected back to a low-dimensional subspace, by using a second pointwise convolution. As the initial and final feature maps are of the same dimension n, they can be added back together, facilitating gradient flow during backpropagation.

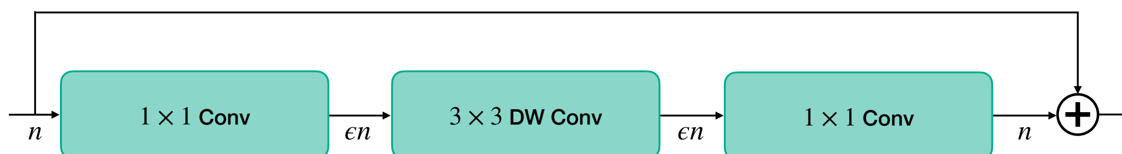


Figure 2.21. The Inverted Residual module, proposed by MobileNetV2. It consists of a first pointwise convolution expanding the channels to allow feature extraction by a depthwise convolution and finally another pointwise convolution projecting the result back to its lower dimension.

This module ensures that the amount of data flowing between blocks stays reasonably small as it is encoded as low-dimensional features. The Inverted Residual thus acts as a decompressor-filter-compressor block, allowing low-dimensional information to be first expanded for feature extraction on spatial information, before being projected back to its low dimension.

Fire Module

The Fire module has been first introduced for the SqueezeNet architecture [37]. It consists of two modules: (1) a Squeeze part; (2) an Expand part. The general Fire block is represented in Figure 2.22.

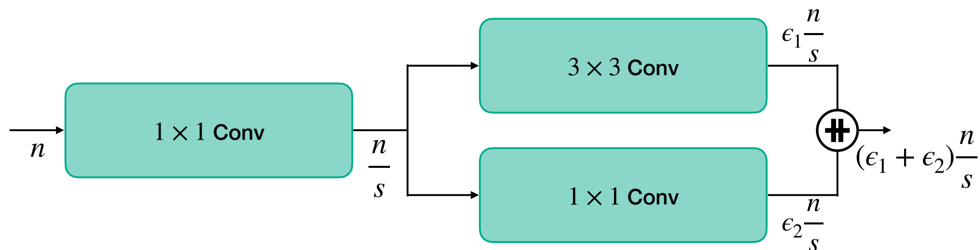


Figure 2.22. The Fire module proposed by SqueezeNet. It consists of a first pointwise convolution squeezing the number of channels, then a mix between pointwise and regular 3×3 convolutions performing the feature extraction on lower-dimensional data.

The goal behind the Fire module is two-fold: (1) reduce the kernel size of convolutions in the network, as explained in Section 2.5.3 by replacing 3×3 convolutions by pointwise convolution where possible; (2) reduce the number of incoming channels to the remaining 3×3 to minimize their amount of parameters.

The Fire module takes in a high-dimension tensor, with n channels and is first reduced by a factor s through the squeeze module, a pointwise convolution. The low-dimensional input tensor then flows through two branches. It is first expanded by a factor ϵ_1, for the 3×3 convolutional branch, and by a factor ϵ_2, for the other pointwise convolution. The output tensors are then concatenated to form a higher-dimensional output tensor. This allows expensive filtering operations to be performed on lower-dimensional data, allowing to greatly reduce the number of parameters involved in the network.

2.6 In Brief

Summary 2

- The parameter count in Neural Networks can be reduced by introducing sparsity in the weights. This sparsity can be introduced in different manners: (1) It can be **designed**, as it is the case for CNNs; (2) it can be learned during training, which is called **pruning**; (3) it can be **ephemeral**, and mainly used for its regularization capabilities.

- The **Knowledge Distillation** technique allows taking advantage of a large and competent model, i.e. the teacher, by using it to guide the learning process of a smaller model; i.e. the student. The student uses cues from the teacher that can come from: (1) the **logits** ; (2) **intermediate** computation states.

- Quantization does not reduce the number of parameters contained in the network but rather reduces the precision that those parameters are stored in. It can be introduced in several ways: (1) **Post-Training Quantization**; (2) **Quantization Aware Training**; (3) **Automatic Mixed Precision**

- Matrix decomposition consists of approximating and replacing heavy computation layers by smaller ones. In the case of CNNs, both the convolutions and fully-connected layers can be decomposed.

- Batch Normalization Folding is a technique allowing the batch normalization layer to be removed at inference time and thus accelerates the network and reduces parameters.

- The size of kernels in a CNN is a crucial hyperparameter. In practice, it is best to stack layers of 3×3 convolution instead of using larger ones, as it benefits from: (1) an increased **expressivity** due to more non-linearities; (2) a **gain** in parameter amount.

- The number of channels contained in each layer can be reduced for further compression by introducing bottlenecks in the network. Two well-known methods are: (1) **Inverted residual**; (2) **Fire module**.

CHAPTER 3

Neural Network Pruning

Contents

3.1 Introduction **52**

3.2 Motivation **53**

3.2.1 Improves Generalization 53

3.2.2 Lowers Complexity 54

3.2.3 Reduces Processing Time and Storage 55

3.3 Neural Network Pruning **56**

3.3.1 How to prune ? 57

3.3.2 Where to prune ? 62

3.3.3 What to prune ? 63

3.3.4 When to prune ? 69

3.4 In Brief **73**

"Every block of stone has a statue inside it and it is the task of the sculptor to discover it."
— Michelangelo Buonarroti

3.1 Introduction

This chapter presents the topic of neural network pruning, i.e. the removal from the network of parameters considered less valuable. The inspiration behind neural network pruning is taken from how the human brain evolves during early life. Indeed, between birth and adulthood, the number of synapses, i.e. the structures responsible for transmitting signals to other neurons, dramatically varies. More precisely, the brain experiences a large amount of growth in synapses during infancy, then gradually proceeds to a pruning operation, removing any synapse that is not needed anymore, allowing it to become more efficient over time and to reach an optimum between performance and energy consumption [38]. This phenomenon is called *synaptic pruning*, and is illustrated in Figure 3.1.

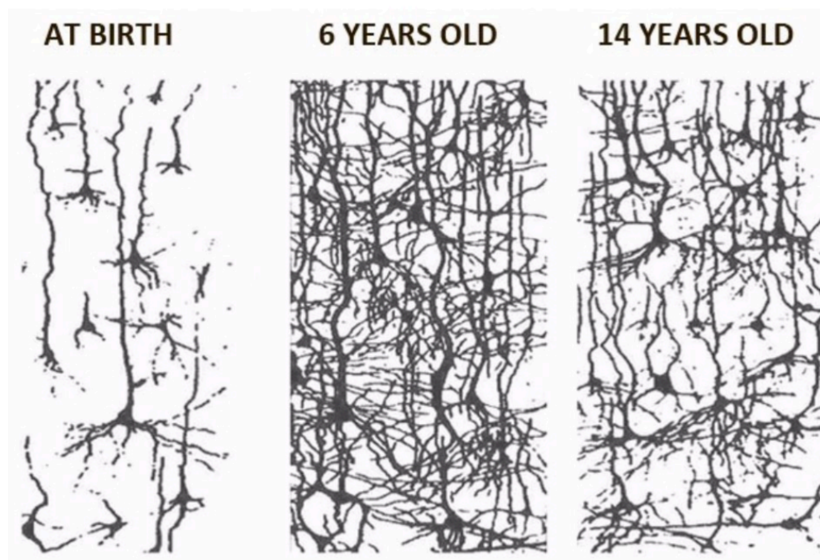


Figure 3.1. Evolution of the synaptic density in the human brain during early life. A growth in the amount of synapses happens during childhood, followed by a synaptic pruning process. Image inspired by [38].

This chapter details the neural network pruning technique. In particular, it first introduces the motivations behind introducing sparsity in the weights of a neural network in Section 3.2. In the Section 3.3, we decompose the neural network pruning problem into 4 questions that need to be answered to fully describe each pruning technique.

3.2 Motivation

There are several motivating factors for performing such a pruning process in a neural network:

- It improves **generalization** by regularizing over-parametrized networks.
- It **reduces the complexity** of the network by identifying well-performing smaller networks.
- It **reduces energy costs, computations, storage and latency** which are all beneficial for deployment on both mobile devices and on remote servers.

These features will be detailed in the following sections, before expanding on the details of neural network pruning.

3.2.1 Improves Generalization

The generalization of a neural network measures how well the latter performs on unseen data, i.e how good the network is at extracting generic features from training data. Several pieces of work have reported an improvement in generalization performance when sparsity increases in the network [39, 40]. When plotted against the sparsity, the generalization performance typically follows an Occam's Hill [41], which can be divided into three parts, as represented in Figure 3.2:

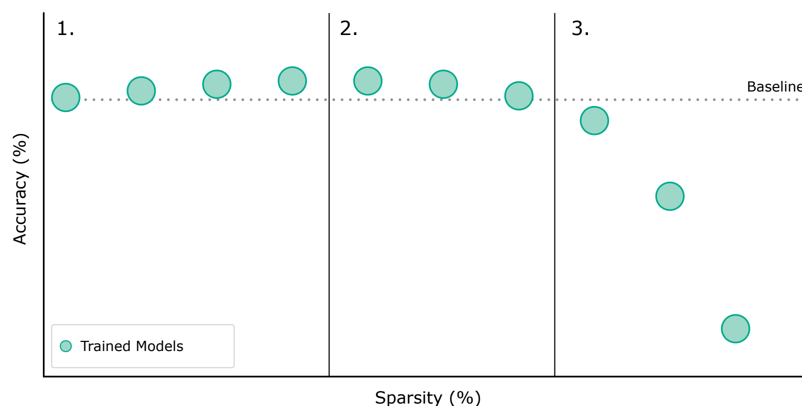


Figure 3.2. Schematic representation of evolution of accuracy of trained models according to pruning level. The performance of models follows a Occam's Hill shape, where accuracy slightly increases with sparsity, then suddenly drops at extreme sparsity levels.

1. A first part where the performance increases with sparsity. In this part, pruning acts as a regularizer, helping the network to improve generalization.

2. A second part where performance reaches a plateau and where adding sparsity does not really improve nor decrease the performance.
3. A third part where the performance quickly decreases.

Despite several works reporting that adding parameters in a network helps it to generalize better [42, 43], it has been shown that removing parameters in a network can also improve generalization [44]. The reason behind this apparent contradiction is that the pruning process can be seen as a noise injection mechanism that, despite degrading the model's performance by introducing instability, helps the learning process to reach a flatter region of the loss landscape [44] and thus reduce overfitting. The sharper the minimum in the training loss the model has converged to, the worse the generalization capability on the test data, as a subtle shift in the distribution of the data and thus in the loss function results in a big jump in performance [45]. On the other hand, if the model has converged to a wider loss solution, then the performance on the test data does not suffer much from a subtle shift in distribution. This phenomenon is represented in Figure 3.3. In practice, we thus aim to make our network converge to an area as flat and wide as possible.

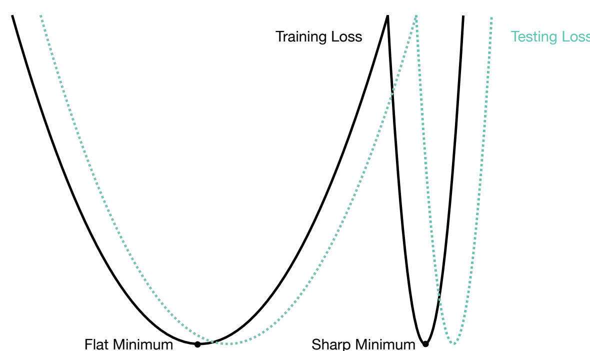


Figure 3.3. Conceptual representation of flat and sharp minimum of the loss function. When a model finds its solution in a flat part of the loss, a subtle change of distribution does not affect the performance too much. The opposite happens if the solution belongs to a sharp minimum. Image inspired by [45].

3.2.2 Lowers Complexity

It has been shown that models with flat minima can be described with low precision, whereas those having a sharp minimum require high precision [46]. This comes from the Minimum Description Length (MDL) principle, which states that fewer bits of information are required to describe a flat minimum [47]. The MDL principle introduces the notion of *parametric complexity*, a measure of the richness of the model, i.e. its

ability to fit random data. It has been shown that larger models are able to fit random labels, but also more easily than smaller ones [42], indicating that models having more parameters are more complex from a parametric complexity sense. The MDL principle is a more formal expression of the Occam's Razor [41], which can be defined as:

> Let $\mathcal{H}_1, \mathcal{H}_2, ...$ be a list of candidate models, able to explain the dataset $\mathcal{D}$. The best model is the one that minimizes:
>
> $$\mathcal{H}_{best} = \arg\min_{\mathcal{H}}[\mathcal{L}(\mathcal{H}) + \mathcal{L}(\mathcal{D}|\mathcal{H})]$$
>
> with $\mathcal{L}(\mathcal{H})$ the length of the description of $\mathcal{H}$ in bits, and $\mathcal{L}(\mathcal{D}|\mathcal{H})$, the length in bits of the description of the data $\mathcal{D}$ when encoded by model $\mathcal{H}$. Then, the best model to explain $\mathcal{D}$ is the smallest one.

As for Occam's Razor, the MDL principle thus favors simple models, i.e. with small parametric complexity. Such a definition is also in line with the Kolmogorov complexity, defined as the length of the shortest binary computer program that describes the object [48]. As smaller models lead to flatter minima, thus requiring fewer bit to describe, they also happen to have a lower Kolmogorov complexity.

3.2.3 Reduces Processing Time and Storage

By removing parameters and introducing sparsity in neural networks, pruning is able not only to reduce the operations required to evaluate a model, but also the memory needed to store such a model. However, sparse neural networks require overhead to index their non-zero elements. This overhead depends on the encoding scheme used to store sparse weights. The simplest scheme is Bitmap (BM), which stores a map with n bit, each bit indicating whether an element is present or not. It is mostly efficient for low sparsity levels, typically between 10% and 70% [20]. Another encoding scheme that is efficient for moderate levels of sparsity, i.e. between 70% and 90% is the Delta coding where only the difference between two elements is stored. In the high sparsity, i.e. sparsities $> 90\%$ regime, schemes known from scientific and high-performance computing such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) can be considered. Other schemes such as Coordinate Offset (COO), storing each non-zero element together with its absolute offset, would be very effective for extreme sparsity levels i.e. $> 99.9\%$, but those levels are yet to be achieved for neural networks.

In some cases, it is not necessary to store indices of non-zero elements. This can happen when complete neurons or filters are zeroed-out. In that case, the model can be re-

arranged into a smaller and dense structure, eliminating the need for dedicated sparse storage and benefiting directly from the pruning process in terms of speed and storage.

3.3 Neural Network Pruning

Neural network pruning thus consists in introducing sparsity in the weights of a network. More formally, pruning is the application of a binary criteria deciding on the weights to assign a value of zero. In practice, this is done by performing an element-wise multiplication between the weights W and the binary pruning mask m as:

$$W = m \odot W \tag{3.1}$$

By zeroing-out some weight, we ensure they do not take part in the forward propagation process anymore. However, to ensure that those weights are not updated, the same mask is applied during the backpropagation phase of training, multiplying the respective gradient of sparse weights by the mask as:

$$dW = m \odot \frac{\partial \mathcal{L}}{\partial W} \tag{3.2}$$

The primary objective of pruning techniques is thus to define the pruning mask m to be applied. To find those masks and apply pruning to any architectures, we identify four fundamental questions to be addressed, and find that answering each of those questions allows to define current pruning techniques unequivocally. Those questions are:

- **How to prune?** The pruning of parameters can be performed at different structure levels, which we refer to as the *granularities*. On the one hand we have the so-called *structured* granularities, i.e when there is an intent to keep some structure in the pruned network, typically when the pruning is performed at the level of vector, kernel or filter. On the other hand, we have *unstructured* pruning, i.e when it is performed at the granularity of individual weight, leading to sparse weight matrices devoid of any structure in terms of how the zero elements are arranged.
- **Where to prune?** Two schemes exist to define from which layer should parameters be removed. This can be done either in a *locally*, imposing a chosen amount of sparsity in each layer separately, or in a *globally*, comparing parameters from the whole model and leading to layers with potentially different sparsity levels.
- **What to prune?** A pruning *criteria* assessing the importance of parameters in the network needs to be defined. Such pruning criteria are generally either *data-agnostic*, meaning that they do not require any training data information and are

only based on the values of the weights themselves, or they can be *data-aware*, thus requiring training information such as gradient values.

- **When to prune?** Pruning can occur at different times in the training process. In particular, this can happen before, during or after the training. The pruning schedules, i.e. how the level of sparsity evolves during the pruning process, usually fall into one of the following categories: *one-shot*, i.e. when sparsity is induced in a single step; *iterative*, i.e. when sparsity is induced in several steps; and *gradual*, i.e. when sparsity evolves gradually during the training process.

3.3.1 How to prune ?

When pruning a neural network, the first question addresses the granularity at which the sparsity is introduced, i.e. the structure of block of weights that are removed. Literature usually differentiates between two categories: (1) unstructured, i.e. when the sparsity is induced at the level of individual weights; (2) structured, i.e. when sparsity is induced at a more coarse level, such as kernels or filters. As we will see, structured pruning is usually used as a catch-all term, but a lot more nuance can be provided.

In the case of Convolutional Neural Networks, we can represent the weights constituting a layer by a 4-dimensional tensor: $C_o \times C_i \times K_h \times K_w$, with C_o and C_i being respectively the number of output and input channels and K_h and K_w being the height and width of the kernels. Weights constituting a convolutional layer are represented in the Figure 3.4.

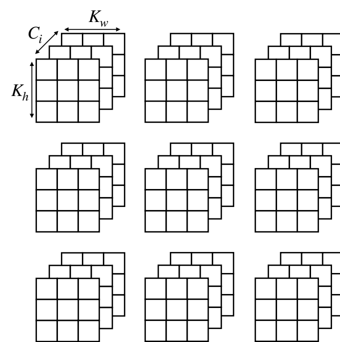


Figure 3.4. Representation of C_o filters, composed of C_i kernels of $K_h \times K_w$ weights each, and constituting the layer of a CNN.

Parameters can thus be removed from the weight tensor according to any of those four dimensions. In particular, we can remove weights (0-dimensional tensors), vectors (1-dimensional tensors), slices (2-dimensional tensors) or even filters (3-dimensional tensors). In this section, we will also propose and describe a novel family of granularities, that we name *shared* granularities.

- **Weight-Level Pruning.** This granularity removes parameters in an *unstructured* way, by comparing each 0-dimensional element contained in the layer and removing the ones considered as less important. Since there are no extra constraints on the pruning pattern, the parameters can be pruned with a high sparsity. Figure 3.5 shows how weight pruning affects the structure of the weights.

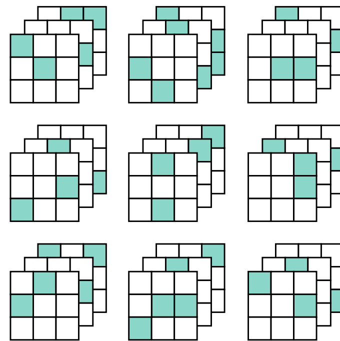
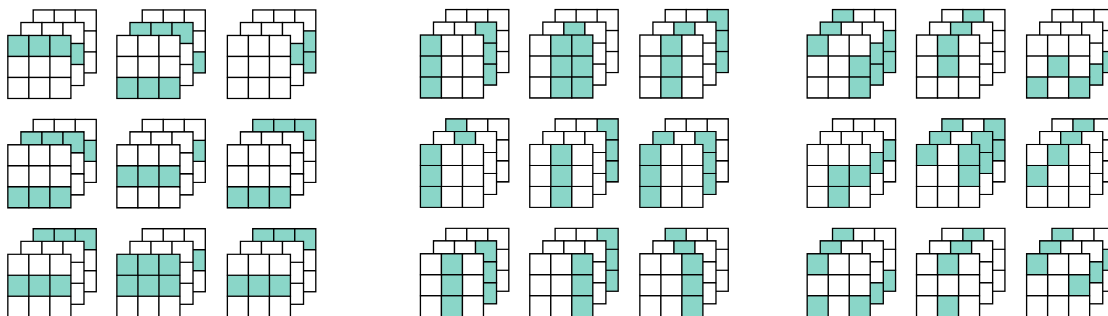


Figure 3.5. Representation of weight pruning. As weights are removed individually from each filter, no sparsity pattern is emerging.Removed weight are in color.

- **Vector-Level Pruning.** It removes 1-dimensional vectors from the initial weight tensor. The most common structures, represented in Figure 3.6 are: (a) **row** vectors; (b) **column** vectors; (c) **channel** vectors. In addition to those three granularities, there is another one that is possible. We name it *shared-weight* granularity, removing weights along the C_i dimension, in an apparently unstructured pattern from a single filter, but sharing it across all filters, as represented in Figure 3.7.



(a) Row Pruning. **(b)** Column Pruning **(c)** Channel Pruning.

Figure 3.6. Variations of vector pruning in a convolutional layer. Vectors that are horizontal, vertical and transversal to the filters can be removed. Removed weight are in color.

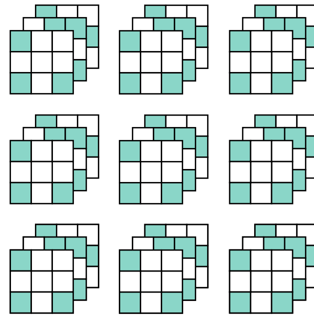
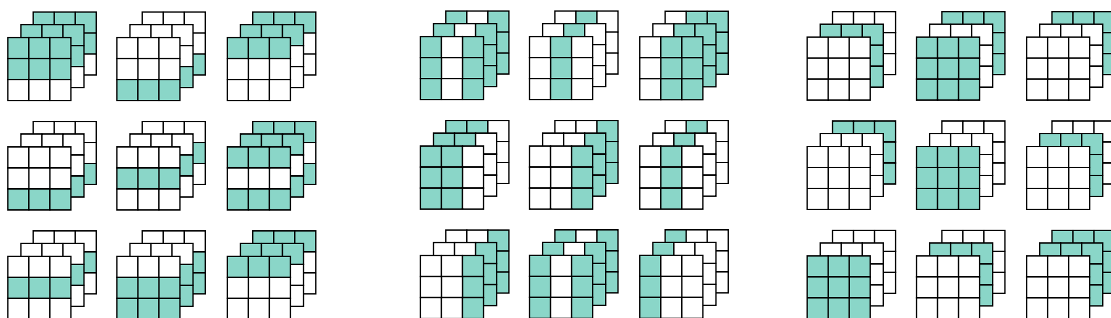


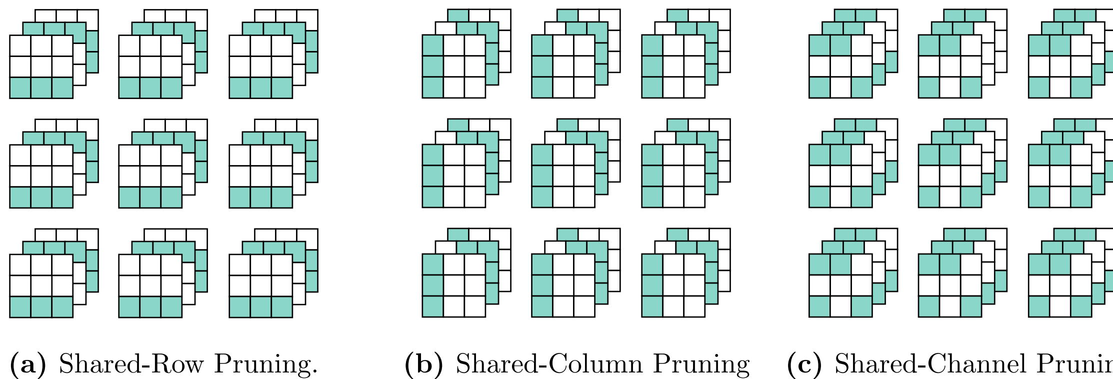
Figure 3.7. Representation of Shared-Weight Pruning. Although it appears to be unstructured at the scale of an individual filter, the same sparsity pattern is shared across all filters of the layer. Removed weights are in color.

- **Slice-level pruning.** This type of granularity removes 2-dimensional tensors from the weight tensor. This type of granularity is most of the time associated with kernel pruning but, as represented in Figure 3.8, there are more options that can be considered. In particular, we can remove: (a) **vertical slices**; (b) **horizontal slices**; (c) **kernels**. In addition, as it was the case for the shared-weight granularity, we can remove weights along the C_i dimension, which leads to granularities that are similar to the vector-level ones presented in 3.6, but this time shared in all filters, as represented in Figure 3.9: (a) **shared-row**; (b) **shared-columns**; (c) **shared-channels**.



(a) Horizontal Slice Pruning. **(b)** Vertical Slice Pruning **(c)** Kernel Pruning.

Figure 3.8. Variations of slice-level pruning in a convolutional layer. Slices that are horizontal, vertical and transversal to the filters can be removed. Removed weight are in color.



(a) Shared-Row Pruning. **(b)** Shared-Column Pruning **(c)** Shared-Channel Pruning.

Figure 3.9. Variations of shared-vector pruning in a convolutional layer. The same vector patterns emerge in all filters of the layer. Removed weight are in color.

- **Filter-Level Pruning.** This type of pruning removes 3-dimensional tensors from the weight tensor and particularly concerns removing entire convolutional filters, as represented in Figure 3.10. Again, there are other 3-dimensional granularities that are possible, i.e. the shared counterparts of slice granularities, as represented in Figure 3.11: (a) **shared horizontal slice**; (b) **shared vertical slice**; (c) **shared kernel**.

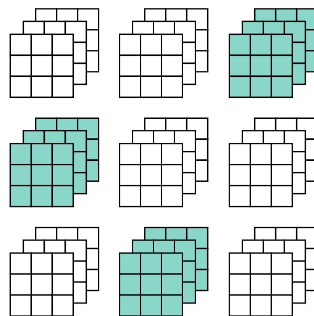
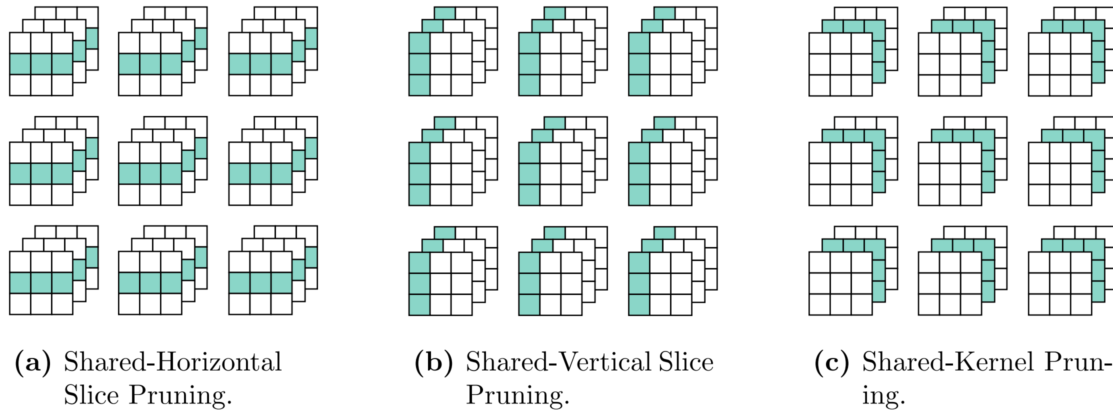


Figure 3.10. Filter Pruning. Entire filters are removed from the layer.Removed weight are in color.

After enumerating those granularities, it is now obvious that the *structured* and *unstructured* categories are insufficient to categorize all the possible sparsity structures. Instead of falling in either of those groups, granularity rather evolves along a *structured* scale, ranging from weight pruning being the less structured to filter pruning being the most structured granularity. Also, the less structured the pruning, the more precise will be the decision on the weights to remove. As a result, a higher sparsity level can be reached before witnessing a performance degradation. On the other hand, the



(a) Shared-Horizontal Slice Pruning.

(b) Shared-Vertical Slice Pruning.

(c) Shared-Kernel Pruning.

Figure 3.11. Variations of shared-slice pruning in a convolutional layer. The same slice is removed from all filters in the layer. Removed weight are in color.

more structured the granularity, the easier it is for hardware speed-up, as there is less overhead to store the zero-weight indices. There is thus a trade-off to find between performance and speed-up when choosing at which granularity the pruning should be performed.

Several granularities allow going one step further. Indeed, when pruning according to those granularities, it is possible to completely remove parts from the network, allowing to return to a dense model, effectively taking advantage of the compression to witness speed-up without any dedicated resource. Two granularities allow performing such feature: (1) **filter** and (2) **shared-kernel**.

Let us consider two consecutive convolutional layers $i + 1$ and $i + 2$, in which we want to remove structures. Each of these layers take feature maps, i.e. the results of the previous layer as input. We differentiate the effects of those two granularities, as depicted in Figure 3.12.

Filter Removal. If we want to remove a single filter from layer $i + 1$, the direct implication on the network is that the removed filter will not produce a resulting feature map anymore. The corresponding feature map $i + 1$ should thus be removed as well. But as those feature maps serve as input of layer $i+2$, removing a feature map directly affects that particular layer. Indeed, the corresponding kernels in each filter of layer $i + 2$ are now found to be useless and can also be removed.

Share-Kernel Removal. If we now want to apply pruning with the granularity of shared-kernel, the impact on the network will be very similar but with the dependencies

inverted. Indeed, if we want to remove a single shared-kernel of all filters in layer $i+2$, then the input feature map corresponding to that kernel will not be useful anymore and can be removed. Moreover, the filter in layer $i+1$ also has no utility anymore as its corresponding output feature maps have been removed, and can be dropped as well.

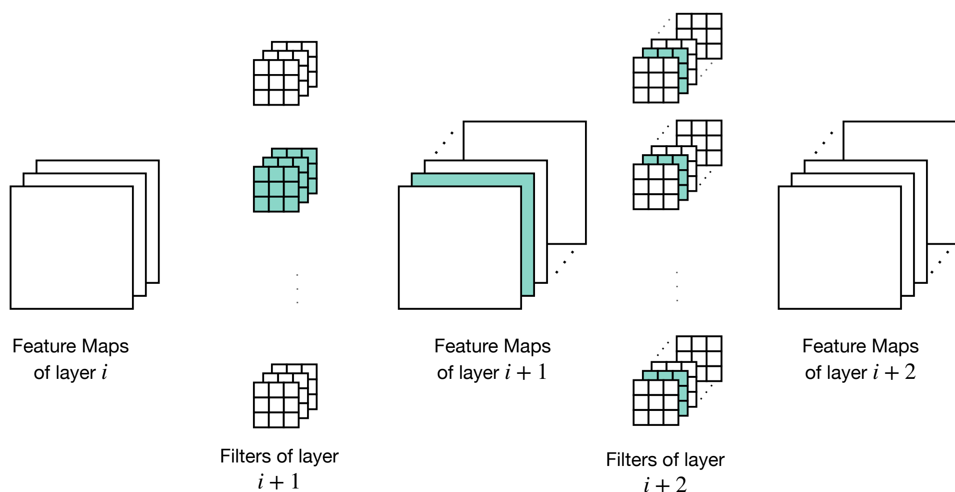


Figure 3.12. Filter and shared-kernel granularities allow to remove parts from the model. This enables the architecture of the model to be changed to a smaller and dense one. Removed weight are in color.

Removing filters or shared-kernel has exactly the same impact on the architecture for all intermediate layers. The difference between both granularities resides in the very first and last layers, as filter pruning impacts the last layer by removing some of its output, while shared-kernel impacts the very first layer, by removing some of its input.

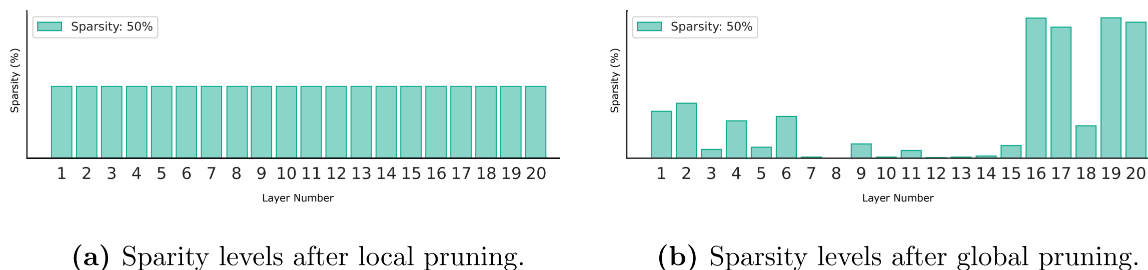
3.3.2 Where to prune ?

When removing parameters of a neural network, their utility may vary a lot depending on the layer from where they come. Indeed, as indicated in Section 2.2.1, neural networks are built in a hierarchical manner, with shallow layer extracting low-level information and deeper layers extracting high-level information about the input data. Generally, a network possesses fewer neurons in shallow layers than in deep layers. This comes from the fact that low-level features are more generalizable to a wide range of applications, which is not the case for more specific, high-level features [49]. As a result, deeper layers usually contain redundant features, that can be removed without hurting the network. For that reason, there exist two main ways to distinguish where in the network to apply the pruning: (1) **Local** pruning, where the selection of the parameters to remove is performed on a per-layer comparison. This implies that each layer will have the same final sparsity level; (2) **Global** pruning, where the selection

of the parameters to remove is performed on the whole model. This implies that each layer will potentially have different sparsities.

Pruning a network according to the local context usually leads to decent results. However, local pruning assumes that all network layers are equally important. This is not necessarily true, which can make local pruning more challenging when the network involves bottleneck layers, where reducing the number of parameters too much can profoundly affect the performance of the model. On the other hand, using a global context is usually more computationally expensive, as the pruning process potentially sorts and compares several millions of weights. However, because it is more flexible and does not introduce any prior on the importance of the weights according to their depth in the network, global pruning usually provides better-performing neural networks.

A representation of the per-layer sparsity level for both pruning contexts in the case of a 20-layer network being pruning to 50% of sparsity can be found in Figure 3.13. It can be observed that local pruning leads to all layers having the same sparsity level, as represented in Subfigure 3.13a and that global pruning leads to layers having different sparsity levels as represented in Subfigure 3.13b.



(a) Sparity levels after local pruning.

(b) Sparsity levels after global pruning.

Figure 3.13. Effect of local and global pruning on the individual sparsity level of a 20-layer neural network. Local pruning leads to equally sparse layers while global pruning leads to layers that have different sparsity levels.

3.3.3 What to prune ?

After choosing the granularity and the context according to which the network will be pruned, we need to select a criteria of selection for its weights. This selection criteria is used as a proxy for parameter importance. In particular, different heuristic criteria were developed to identify the promising structures to be pruned without harming the prediction performance. We classify these criteria into two categories: (1) **Data-**

Agnostic, concerning criteria using only weight information to decide which weight to remove; (2) **Data-Aware**, concerning criteria using training information such as gradients or the loss in order to select the weights to remove.

Data-Agnostic Methods

After grouping weights in the model according to a chosen granularity, the pruning criteria assigns a score to each weight group. A threshold λ is then computed according to the desired sparsity, and the values of the binary pruning mask m are assigned to 1 where the weight values are above λ and 0 where they are below. Several ways exist to score the weights, and the more common ones will be detailed below. In particular, we provide the expression of the mask m, and illustrate the resulting decision boundaries in relation to weights values at initialization W_i and their value at the current training step W_f.

- **Random:** A random criteria is often used as a control case. As the name suggests, the values of the mask m are then assigned randomly. In this case, the mask m is a random variable, following a Bernoulli distribution with probability $p = 1 - s$, s being the desired sparsity. In this case, the mask m is computed by the following expression:

$$m = \begin{cases} 1: & \text{with } \mathbb{P} = 1 - s \\ 0: & \text{with } \mathbb{P} = s \end{cases} \tag{3.3}$$

  A representation of selected weights to remove is provided in Figure 3.14.

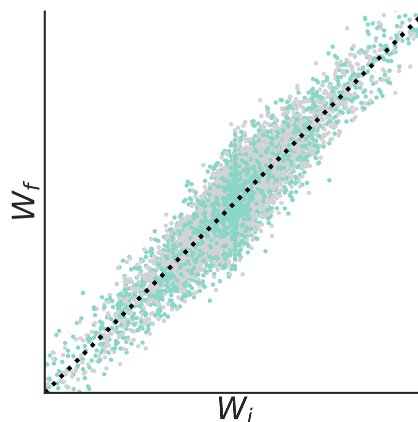


Figure 3.14. Result of the application of the Random criteria on the weights. Parameters that are greyed out are the parameters removed.

- **Large Final:** The mask is determined according to the magnitude of the parameters after training. In this case, the highest magnitude ones will be retained while others will be removed. This criteria is often named "l_1-pruning" as the weights are ranked by their l_1-norm [50]. In this case, the mask m is computed by the following expression:

$$m = \begin{cases} 1, & \text{if } \left|w_f\right| > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

We represent in Figure 3.15 the decision boundaries imposed by such a criteria.

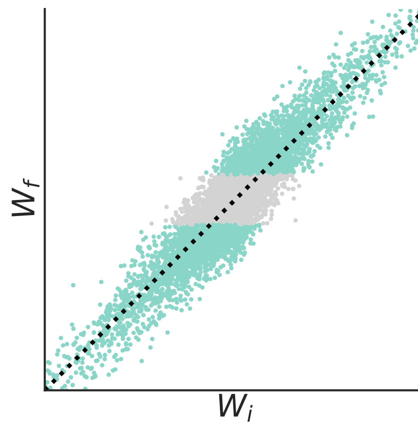


Figure 3.15. Result of the application of the Large Final criteria on the weights. Parameters that are greyed out are the parameters removed.

- **Large Intitialization:** The mask is determined according to the initial value of the parameters and is totally independent of their current value. The computation of the mask is very similar to the large final criteria but in this case, the initial value of the weights W_i is used, according to the following expression:

$$m = \begin{cases} 1, & \text{if } |w_i| > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

The decision boundaries are provided in Figure 3.16.

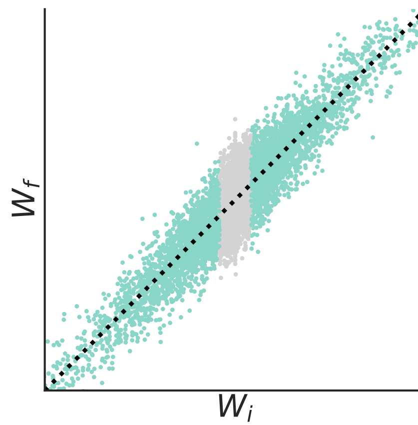


Figure 3.16. Result of the application of the Large Initialization criteria on the weights. Parameters that are greyed out are the parameters removed.

- **Large Initialization Large Final:** This criteria combines both previous ones, by selecting the weights that have the highest initial and final values. The mask is computed by the following expression:

$$m = \begin{cases} 1, & \text{if } \min(\left|w_f\right|, |w_i|) > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

The decision boundaries are provided in Figure 3.17.

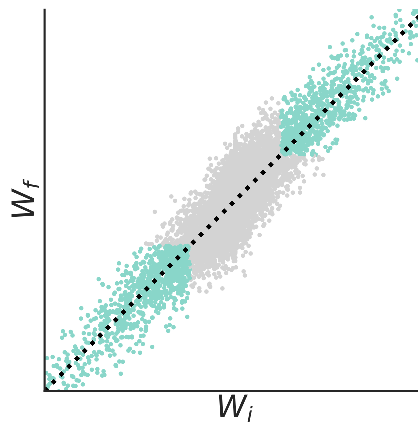


Figure 3.17. Result of the application of the Large Initialization Large Final criteria on the weights. Parameters that are greyed out are the parameters removed.

- **Magnitude Increase Pruning:** Magnitude Increase Pruning proposes to retain weights whose magnitude has increased during the fine-tuning process, indicating that the optimization phase has given them more importance for the task at hand [51]. To compute the mask m, it is thus required to compare the current magnitude of the weights to the magnitude they had at initialization, as:

$$m = \begin{cases} 1, & \text{if } |w_f| - |w_i| > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

The decision boundaries are provided in Figure 3.18.

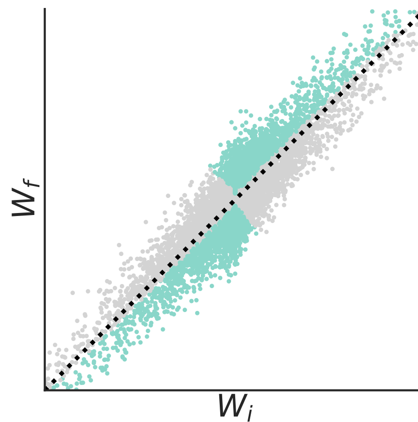


Figure 3.18. Result of the application of the Magnitude Increase criteria on the weights. Parameters that are greyed out are the parameters removed.

- **Movement Pruning:** the movement criteria is a variation of magnitude increase one. The rationale is slightly different, as movement puts emphasis on weights that undergo significant change in their value, as it indicates that those weights are highly updated by the optimization process. The mask m is computed following:

$$m = \begin{cases} 1, & \text{if } |w_f - w_i| > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

The decision boundaries are provided in Figure 3.19.

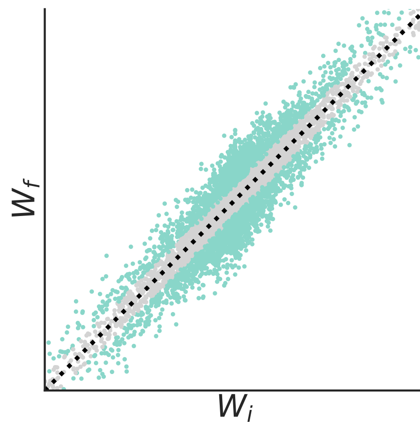


Figure 3.19. Result of the application of the Movement criteria on the weights. Parameters that are greyed out are the parameters removed.

Data-Aware Methods

Data-aware methods are techniques that use information from the data in order to score the weights. There also exists plenty of methods, but only the most commonly used will be described:

- **Hessian.** One of the first proposed criteria was based on the second-order derivative of the loss function [52, 53]. However, these Hessian-based methods were rapidly set aside, the Hessian matrix being neither diagonal nor positive definite in general, they were intractable for large neural networks.

  The mask is computed by:

$$m = \begin{cases} 1, & \text{if } \frac{w_f^2 H_f}{2} > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.9}$$

  Where H_f is the value of the Hessian matrix corresponding to w_f , where the Hessian $H = \delta^2 L/\delta W^2$.

- **First-Order Taylor Approximation.** To reduce the computation needs of the Hessian matrix, some work approximated the squared difference of the loss with a first-order approximation [54]. This allowed to compute the criteria of weights, using only their value and gradient, readily available during backpropagation.

In practice, the mask m is computed by:

$$m = \begin{cases} 1, & \text{if } g_f w_f^2 > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.10}$$

where g_f is the gradient corresponding to w_f.

- **Gradient.** Other works have hypothesized that the parameter importance is solely defined by its gradient magnitude [55]. This means that, if the gradient magnitude is high, a weight significantly impacts the loss, and thus must be preserved.

  In this case, the mask m is computed by:

$$m = \begin{cases} 1, & \text{if } \frac{|g_f|}{\sum_{k=1}^{n} |g_{f,k}|} > \lambda \\ 0, & \text{otherwise} \end{cases} \tag{3.11}$$

  where k is the amount of gradients compared when applying the pruning.

3.3.4 When to prune ?

A neural network is parameterized by a set of weights W. Those weights evolves during training, such that we can define the set of weight values possessed by the network along training as:

$$\{W_0, W_1, \cdots, W_t \cdots, W_T\}$$

where t denotes the t-th training iteration and T is the total number of training iterations. Let us denote m_T the pruning mask computed on the final set of values W_T. Traditional pruning methods require to compute m_T and to apply it on W_T, thus creating the pruned network. However, recent techniques have proposed to decouple the selection of mask from its application. The mask is computed based on the weights W_T but is applied on the weights W_0, suggesting that pruning could be applied at initialization. This idea was recently popularized by LTH [56], that goes as follows:

> **The Lottery Ticket Hypothesis:** A randomly-initialized, dense neural network contains a subnetwork that is initialized such that — when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations.

Such a subnetwork is said to have won at the initialization lottery and is thus named a *winning ticket*. However, a limitation of the LTH is its lack of generalization to larger datasets and architectures [57]. Indeed, it was shown that resetting the weights to their original values was hindering the good convergence in early training iterations and the inability to find winning tickets. The authors resolve the problem by weakening the original hypothesis. Instead of resetting the weights to initialization, they reset them to some phase early in training [57]. This generalization is called Lottery Ticket Hypothesis with Rewinding (LTHR) as the weights are now rewound to a previous training iteration. Found subnetworks are no longer called winning tickets but rather *matching tickets* as they are found from an early iteration instead of initialization. The process to find such a subnetwork is represented in Figure 3.20 and consists of six steps.

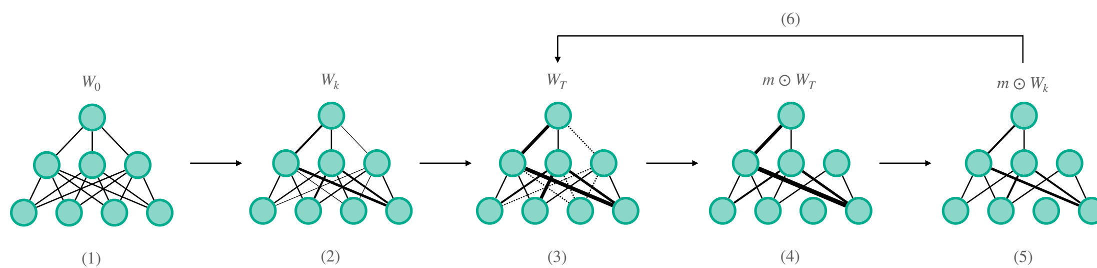


Figure 3.20. Performing Lottery Ticket Hypothesis with Rewinding to discover the optimal subnetwork. If the iteration k is set to 0, then this process corresponds to the classic Lottery Ticket Hypothesis.

1. Initialize the weights of a neural network to a value W_0;
2. Train the network for k iterations , creating the set of weights W_k and save those weights;
3. Train the network to convergence, creating the set of weights W_T;
4. Prune the smallest magnitude weights by applying a binary mask m_T;
5. Reinitialize the weights to the saved value, i.e. at iteration k, and apply the mask found at the preceding step;
6. Return to step 3 until reaching the desired level of sparsity.

When the iteration k is chosen to be 0, then LTHR degenerates into the classic Lottery Ticket Hypothesis (LTH). Those results present important implications, as they suggest that it should be possible to extract the winning ticket without undergoing the time-consuming pruning process. More surprisingly, when applying the mask m_T on the network W_0, i.e. on the untrained network, the latter is already able to reach non-random accuracies even without any training.

When elaborating on the schedules of pruning, there are two main aspects to take into account: (1) When is the pruning process starting? That is, on which set of weights is the mask applied? (2) When is the mask computed ? That is, how does the sparsity evolve during training?

There exist three possibilities of when the pruning process should occur:

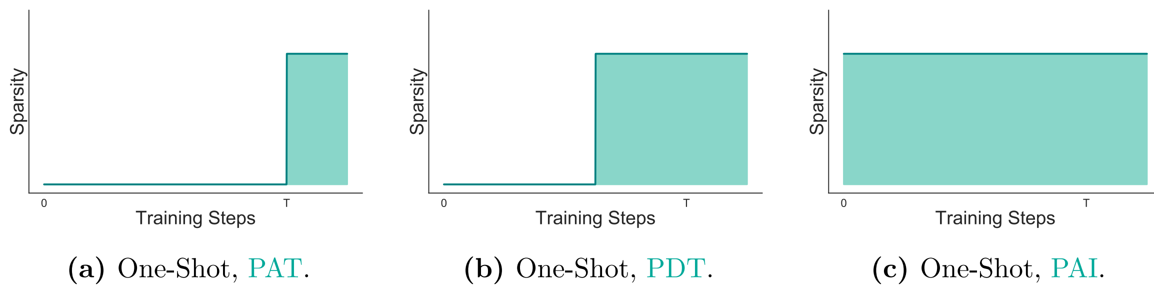
- **Pruning After Training (PAT).** This type of pruning, i.e. after the model has been trained to convergence, is the most basic way to prune a model. In this pruning scheme, we prune from the model W_T, creating the model $W_T \odot m_T$. We may eventually fine-tune the model to recover from potential lost performance endured during the pruning process. We then obtain a pruned model $W_M \odot m_T$, with $M > T$. Sometimes, the pruning process can also be continued, updating the mask until desired sparsity is obtained.

- **Pruning During Training (PDT).** This type of pruning computes the mask m on a set of weights that has been trained, but not to convergence, W_t, creating the sparse network $W_t \odot m_t$, that can further be trained to create the network $W_T \odot m_t$. Doing so ensures that sufficient information about weight's importance is available, but usually results in a cheaper training process.

- **Pruning At Initialization (PAI).** The last method intends to compute the mask m and apply it to the initial set of weights, W_0. This has for objective of decreasing the training costs in order to find the sparse model. The mask can however be found in several manners. Indeed, the mask can be said to be post-selected, i.e. the mask is computed on weights W_T and applied to create the sparse network $W_0 \odot m_T$. This is the case with methods such as Lottery Ticket Hypothesis [56]. The mask can also be said pre-selected, i.e. computed on weights W_0, creating the sparse network $W_0 \odot m_0$, such as in SNIP [55] or GraSP [58] methods. In both cases, the pruning phase is followed by a challenging training of a sparse model to convergence.

The pruning mask m can be computed in different ways. This defines how sparsity evolves during the pruning phase. In particular, three categories can be identified:

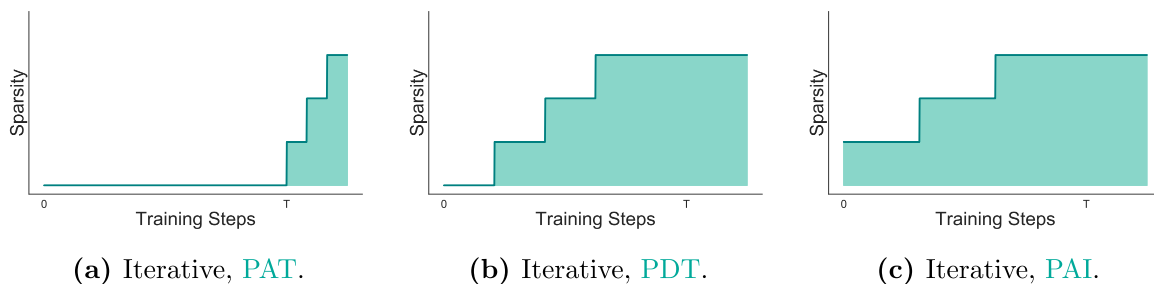
- **One-Shot Pruning.** As the name suggests, One-Shot Pruning computes the mask m and prunes the network, i.e. applies the mask m, in a single step. In this case, when the model has been pruned, the mask is no longer updated. Although this method is effective to some extent, it generally provides suboptimal results. An illustration of One-Cycle Pruning applied at different training iterations can be found in Figure 3.21.



(a) One-Shot, PAT. **(b)** One-Shot, PDT. **(c)** One-Shot, PAI.

Figure 3.21. Evolution of sparsity for the One-Shot pruning schedule, removing parameters in a single pruning phase, and applied at different training times.

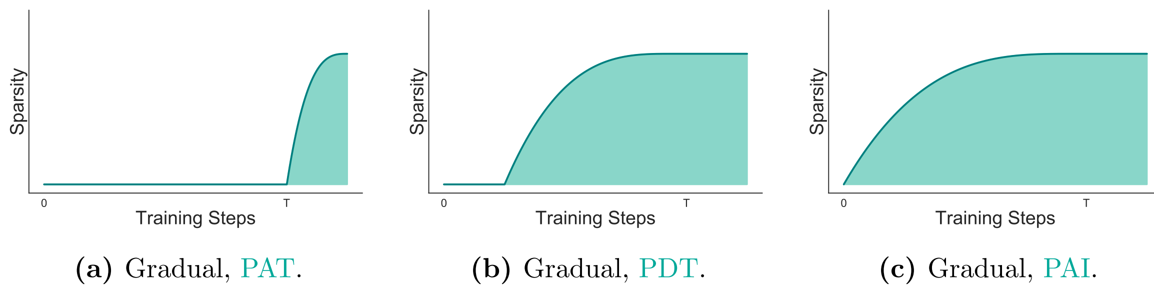
- **Iterative Pruning.** This method proposes to perform the pruning interleaved with retraining phases. This proved to reach higher levels of accuracy before witnessing a performance drop, but also to require a more extensive training budget to obtain the final sparse model. An illustration of Iterative Pruning applied at different training iterations can be found in Figure 3.22.



(a) Iterative, PAT. **(b)** Iterative, PDT. **(c)** Iterative, PAI.

Figure 3.22. Evolution of sparsity for the Iterative pruning schedule, removing parameters in several pruning steps, and applied at different training times.

- **Gradual Pruning.** Gradual Pruning provides a smoother schedule, better integrated into the training dynamics. It usually follows a slightly more complex schedule function, allowing to smoothly update the mask during the training. The most famous Gradual Pruning method has been proposed to remove a large part of weights early in training, then gradually reduce the rate of removal until training has been completed [59]. In particular, it is represented in Figure 3.23.



(a) Gradual, PAT. **(b)** Gradual, PDT. **(c)** Gradual, PAI.

Figure 3.23. Evolution of sparsity for the Iterative pruning schedule, removing parameters gradually along the training, and applied at different training times.

3.4 In Brief

Summary 3

- Neural Network pruning introduces sparsity in the weights, which allows the network to: (1) improve the **generalization** of models, (2) lower their **complexity**, (3) reduce their **processing** time and **memory** storage.
- We identify four open questions that must be addressed to define a pruning technique: (1) **How** to prune? (2) **Where** to prune it? (3) **What** to prune a model? (4) **When** to prune?
- The answer to "How to prune?" concerns the pruning **granularities**, ranging from weights to entire filters.
- The answer to "Where to prune?" concerns the pruning **context**, which affects the sparsity level of each layer in the network.
- The answer to "What to prune?" concerns the pruning **criteria**, predicting the importance of the weights.
- The answer to "When to prune?" concerns the pruning **schedule**, defining how the sparsity evolves during the training phase.

CHAPTER 4

Developed Tools: FasterAI

Contents

4.1 Introduction . . . 76
4.2 Sparsify . . . 77
4.2.1 Granularity . . . 78
4.2.2 Context . . . 80
4.2.3 Criteria . . . 82
4.2.4 Schedule . . . 84
4.2.5 Lottery Ticket Hypothesis . . . 87
4.2.6 Prune . . . 88
4.3 Distill . . . 89
4.4 Regularize . . . 90
4.5 Misc . . . 91
4.5.1 Batch Normalization Folding . . . 92
4.5.2 Fully-Connected Layers Decomposition . . . 92
4.6 In Brief . . . 93

"We shape our tools and thereafter our tools shape us."
— Marshall McLuhan

4.1 Introduction

Although many theoretical research studies have been conducted on compression, the field still lacks convenient tools for: (1) practical case applications; (2) testing and performing research. Also, because of this lack of tools, there is no standard way of implementing new compression techniques, making the comparison with previous techniques more difficult [10]. To solve this issue, we propose `FasterAI` [60], an open-source library, released under an Apache-2.0 license. It also includes extensive documentation and several tutorials to help users to get acquainted with the library [1].

Compression libraries such as PyTorch Pruning [61] and Sparse ML [62] have been released in the last few years but those are mainly concerned with sparsification, neglecting other compression techniques such as Knowledge Distillation and Regularization. Another library, Nervana Distiller [63], provides a complete compression toolset, but is intended primarily for research usage. Also, most of those libraries require implementing new compression techniques in a self-contained way, limiting the opportunities for extensive experiments. In `FasterAI`, we aim to reduce the need for implementation to its bare minimum. Indeed, implementing a new method in `FasterAI` boils down to writing a single line of code. Moreover, to the best of our knowledge, `FasterAI` is the first compression library available for both fastai [64] and PyTorch Lightning [65].

The objective of `FasterAI` is twofold: (1) allow inexperienced users to apply compression techniques; (2) allow researchers to easily implement new compression methods and perform various experiments. `FasterAI` is organized around four modules:

Sparsify. The first module is responsible for making sparse neural networks, either in a *static* way, when retraining cannot be considered, or in a *dynamic* way, using callback systems, thus occurring during the training of the neural network.

Distill. This module is in charge of Knowledge Distillation techniques, i.e. training with a teacher-student paradigm, where a large model guides a smaller one to reach better performance, compressing the knowledge of a large model into a smaller one.

Regularize. The regularize module handles group regularization methods, i.e. techniques adding a penalty term on the magnitude of the weights, where some weights will be pushed towards 0, leading to a sparse learned model.

Misc. The last module includes singular compression methods such as batch normalization folding, removing batch normalization layers, which are useless for inference. It also includes factorization methods for fully-connected layers, that replace large weight matrices with smaller ones, thus reducing the total amount of weight.

[1] https://nathanhubens.github.io/fasterai/

This chapter is divided into four sections, each one describing a compression module of FasterAI. In particular, we want to highlight how convenient it is to perform different kinds of experiments, either using well-known techniques, or creating novel ones. Indeed, by leveraging the callback system of recent deep learning libraries such as fastai and Pytorch lightning, FasterAI provides a user-friendly and high-level API, allowing to easily combine and customize compression techniques. To summarize, with FasterAI, we provide:

- An extensive, documented and open-source PyTorch-based neural network compression library.
- A new granular design approach for compression techniques, allowing to seamlessly perform thousands of different compression methods, by simply choosing between available options.
- A framework suited for practical cases as well as for research, by providing common compression techniques available out-of-the-box and allowing the conception of new compression methods in a single line of code.

To this day, the FasterAI library has been downloaded more than $12,000$ times and is currently being used in the industry sector at AMD [2], in one of their research project.

4.2 Sparsify

The core of FasterAI resides in its sparsify module, containing capabilities allowing the creation of sparse networks. FasterAI possesses two main ways to create a sparse network: (1) the *static* way, by using the Sparsifier class, able to sparsify either a specified layer, or the whole model, (2) the *dynamic* way, by using the SparsifyCallback, that must be used in conjunction with training, and that removes weights while the network is learning. Examples of usage for both methods are expressed in Pseudo-Code 4.1.

While the static way is faster to apply as it does not require any additional step, the lack of retraining after the removal of some parameters deeply impairs the model performance. For that reason, the dynamic way is most of the time preferred when trying to achieve compression while keeping the performance as high as possible. Within FasterAI, we make the distinction between the process of sparsification, i.e. making the neural network's weight sparse, and pruning, i.e. physically removing those sparse weights. Indeed, the SparsifyCallback does not allow to remove any network's weight but rather to create a binary mask, of the same structure as the weights, and applies it to either sparsify a weight (when the mask value is 0) or keep it unchanged (when the mask value is 1). Weights sparsified in the process are still present in the computation

[2] https://www.amd.com/en/corporate/research

```
# (1) Static
sp=Sparsifier(model, granularity, context, criteria)
sp.prune_model(sparsity)

# (2) Dynamic
sp_cb=SparsifyCallback(sparsity, granularity, context, criteria, schedule)
learner.fit(n_epochs, cbs=sp_cb)
```

Pseudo-Code 4.1. The two ways of sparsifying a model. The `static` is done offline, disconnected from training, while the `dynamic` is performed during training, allowing the model to recover from lost performance.

graph but do not participate in the final decision anymore.

The whole power of the sparsification capabilities of `FasterAI` lies in its `SparsifyCallback`, designed around four independent building blocks: `granularity`, `context`, `criteria`, and `schedule`, fully describing the most common sparsification techniques. Those building blocks correspond to the four main axes of research in the field, described in Section 3.3.

- `granularity`: how to sparsify?
- `context`: where to sparsify?
- `criteria`: what to sparsify?
- `schedule`: when to sparsify?

The purpose is to decompose the sparsifying problem into four subproblems. By doing so, each argument can be modified independently from the others, which allows for: (1) creating a vast number of opportunities and combinations for experiments; (2) providing a unique and versatile callback, reducing the problem of implementing a novel sparsification technique to the modification of a single argument.

4.2.1 Granularity

In `FasterAI`, the granularity designates the structure of the blocks of weights that are removed during the sparsification process. `FasterAI` handles most common sparsifying granularities, e.g. weight, kernel and filter, but also allows the use of more seldom ones, e.g. horizontal slices, shared-kernels. In the literature, the terms unstructured and structured sparsity are often used to designate when sparsity is applied on weights (unstructured) or on larger blocks (structured). In `FasterAI`, we adopt a more nuanced approach by defining as many granularities as there are slicing combinations of the weight tensor. In the case of 2D convolutions, 16 granularities, described in Section

3.3.1, are thus available by default. On top of the presented granularities, suited for ConvNets, `FasterAI` also proposes granularities for Fully-Connected Layers, as well as for Self-Attention layers, required in Transformers architectures [66].

By following `PyTorch` conventions [61], the weights of a Convolutional layer are given by a 4D tensor of dimension `[O, I, Kh, Kw]`, with `O,I` being respectively the output and input dimensions, and `Kh, Kw`, dimensions of the convolutional kernel. The selection of granularities is given by the Pseudo-Code 4.2.

```
Weight                  (0D) = Weights[o, i, kh, kw]
Column                  (1D) = Weights[o, i, : , kw]
Row                     (1D) = Weights[o, i, kh, : ]
Shared-Weight           (1D) = Weights[:, i, kh, kw]
Channel                 (1D) = Weights[o, :, kh, kw]
Kernel                  (2D) = Weights[o, i, : , : ]
Shared-Channel          (2D) = Weights[:, :, kh, kw]
Shared-Column           (2D) = Weights[:, i, : , kw]
Shared-Row              (2D) = Weights[:, i, kh, : ]
Vertical-Slice          (2D) = Weights[o, :, kh, : ]
Horizontal-Slice        (2D) = Weights[o, :, : , kw]
Shared-Vertical-Slice   (3D) = Weights[:, :, kh, : ]
Shared-Horizontal-Slice (3D) = Weights[:, :, : , kw]
Shared-Kernel           (3D) = Weights[:, i, : , : ]
Filter                  (3D) = Weights[o, :, : , : ]
Layer                   (4D) = Weights[:, :, : , : ]
```

Pseudo-Code 4.2. Slicing of 4D weight tensor of dimension `[O, I, Kh, Kw]` to extract the desired granularity.

As a proof-of-concept, we conduct an experiment to highlight granularity's impact on the performance of a neural network. We choose the ResNet-18 architecture [67], a CNN using residual connections, as it is a model commonly used for benchmarking and apply it to the CALTECH-101 dataset [68], various in images and classes. We thus compare the final validation accuracy when training for 30 epochs, obtained after sparsifying with each of the available granularities. There are four sparsity levels that are studied: 30%, 50%, 70% and 90%. Two initialization methods are considered: (1) the model is either trained from scratch, i.e. randomly initialization, or (2) fine-tuned from a pre-trained weights. To serve as a baseline, the context, criteria and schedule are respectively set to `local`, `large_final` and `one_cycle`. The results are presented in Table 4.1. For readability constraints, the names of criteria in Table 4.1 have been abbreviated, e.g. `s-v-slice` corresponds to the `shared_vertical_slice` criteria.

	Scratch				Fine-tune			
	30%	50%	70%	90%	30%	50%	70%	90%
Baseline	80.61 ± 0.42				90.03 ± 0.54			
weight	80.40±0.22	80.20±0.71	79.74±0.85	78.46±0.40	91.39±0.14	91.67±0.56	91.43±0.10	88.75±0.90
column	81.04±0.46	80.58±0.15	80.34±0.22	75.60±0.50	91.85±0.59	91.23±0.34	90.43±0.47	83.55±1.53
row	81.13±1.01	80.18±0.56	79.80±0.19	75.97±0.65	91.56±0.79	90.57±0.54	90.88±1.06	85.12±0.97
s-weight	80.49±0.69	80.14±0.19	78.57±1.03	66.23±0.97	90.83±1.20	90.06±0.25	88.40±0.67	46.97±0.87
channel	81.29±0.43	79.50±0.62	79.81±0.63	65.85±0.32	91.16±0.41	90.34±1.03	86.60±0.43	44.97±0.62
kernel	80.27±0.49	79.50±0.51	79.52±1.52	67.34±1.57	91.79±0.43	91.25±0.43	89.88±0.73	77.64±0.80
s-channel	79.54±0.87	80.85±0.98	78.21±0.56	38.60±3.12	90.75±0.55	90.55±0.52	86.20±1.34	23.03±0.92
s-column	79.69±0.64	79.87±0.23	80.23±0.40	59.90±1.71	90.70±0.69	89.93±0.31	86.00±0.32	41.28±3.53
s-row	80.51±0.34	79.78±0.29	77.92±0.56	57.99±0.89	90.30±0.51	89.88±0.53	85.38±0.57	40.43±0.92
v slice	80.01±0.85	80.43±0.91	78.28±0.35	48.09±1.55	90.59±0.45	89.46±0.71	82.90±0.30	29.81±1.78
h slice	80.22±0.79	79.19±0.82	77.13±1.60	43.95±1.02	91.01±0.87	88.75±0.36	82.48±0.96	29.60±2.35
s-v-slice	79.91±0.84	80.40±0.50	76.97±0.47	74.98±0.92	89.28±0.35	89.17±0.62	75.47±0.75	73.49±1.09
s-h-slice	79.81±0.76	80.07±0.64	76.22±0.11	74.58±0.35	89.77±0.22	88.60±0.58	73.41±1.21	69.15±0.77
s-kernel	79.96±0.11	81.00±0.21	75.84±1.60	48.56±0.94	89.77±0.43	89.30±0.48	81.25±0.27	31.86±1.23
filter	80.76±1.65	78.41±1.01	72.92±2.40	37.47±1.90	89.86±0.85	87.78±0.18	77.52±1.84	29.05±0.09

Table 4.1. Results of sparsifying ResNet-18 for all available granularities. Context, criteria and schedule are respectively set to `local`, `large_final` and `one_cycle`. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

4.2.2 Context

In `FasterAI`, the context refers to the locality of selection of the weights. In the literature, the two most common options are: (1) *local* pruning, selecting the weights in each layer separately, producing equally sparse layers, and (2) *global* pruning, selecting the weights by comparing those of the whole network, producing different sparsity levels for each layer. Both techniques are expressed in a simplified way in Pseudo-Code 4.3.

```
# (1) Local
for layer in layers:
  mask = compute_mask(layer.weight, sparsity)
  pruned_model = prune_layer(layer, mask)

# (2) Global
global_weights = concat[(layer.weight) for layer in layers]
mask = compute_mask(global_weights, sparsity)
pruned_model = prune_model(mask)
```

Pseudo-Code 4.3. Representation of `local` sparsification, performed in each layer independently and `global` sparsification, performed on all the layers.

FasterAI can handle both methods by default, only by selecting the local or global method accordingly. in the SparsifyCallback. In the case the user wants to specify the sparsity level of layers, FasterAI accepts a list of sparsities, that will be applied to the corresponding layers.

In Table 4.2, we provide the results for the same experiments as the one conducted in Subsection 4.2.1, but using a global context instead. For readability constraints, the name of the granularities in Table 4.2 have also been abbreviated.

	Scratch				**Fine-tuned**			
	30%	50%	70%	90%	30%	50%	70%	90%
Baseline	80.61 ± 0.42				90.03 ± 0.54			
weight	80.58±1.21	80.42±0.73	80.31±1.01	79.63±0.59	91.12±0.59	91.17±0.19	91.79±0.35	89.90±0.14
column	80.51±0.58	80.78±0.48	80.03±1.08	78.87±0.85	91.54±0.50	90.90±0.46	91.43±0.61	87.48±0.44
row	81.13±1.30	80.65±0.71	79.50±1.52	78.57±0.45	91.19±0.88	91.72±0.26	91.32±0.90	87.78±0.48
s-weight	79.96±0.57	80.45±0.30	76.84±0.72	58.92±3.78	90.99±0.23	90.96±0.68	84.65±1.28	35.83±1.12
channel	80.78±0.52	80.53±0.12	77.79±1.31	63.91±0.74	91.27±0.22	90.55±0.38	80.91±0.70	18.29±1.88
kernel	81.89±0.39	80.49±0.29	79.74±0.41	72.08±1.91	91.08±0.37	91.05±0.52	90.43±0.36	85.25±0.66
s-channel	79.01±1.33	78.70±0.72	73.54±0.92	72.87±1.10	89.00±0.35	85.65±1.72	74.03±1.52	61.82±1.69
s-column	80.87±0.93	79.74±0.39	75.78±0.54	68.85±3.59	90.28±0.22	89.26±0.51	72.14±0.52	18.47±5.31
s-row	80.36±0.87	80.80±0.67	74.58±1.68	68.31±1.23	91.21±0.45	89.57±0.11	65.55±4.13	23.01±11.27
v slice	80.34±0.99	78.72±0.54	75.09±0.39	45.40±4.10	90.81±0.62	87.98±0.64	76.68±0.66	25.36±1.99
h slice	80.36±0.67	79.83±0.34	75.47±0.22	47.52±1.77	90.85±0.41	87.96±0.34	78.46±0.62	27.66±2.97
s-v-slice	79.54±0.36	78.67±0.27	78.14±1.03	78.39±0.90	90.04±0.43	87.56±0.86	85.45±0.40	81.89±0.85
s-h-slice	79.72±0.23	79.19±0.45	78.79±0.49	78.36±0.79	89.30±0.42	86.49±1.05	85.39±0.43	83.39±0.47
s-kernel	79.18±0.56	76.79±1.32	75.64±1.47	75.05±1.64	86.34±0.47	84.76±1.43	75.33±6.07	25.60±2.26
filter	79.36±1.22	78.52±0.65	72.83±1.44	68.09±4.39	90.39±0.89	84.43±0.11	62.64±1.58	44.15±1.64

Table 4.2. Results of sparsifying ResNet-18 for all available granularities. Context, criteria and schedule are respectively set to global, large_final and one_cycle. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

As can be observed by comparing Table 4.1 and Table 4.2, global sparsifying seems to achieve better results, especially for higher sparsity levels. This can be explained by the fact that global methods are more flexible, as they allow the sparsifying process to select weights from anywhere in the network, reducing the risk of removing too many weights in the more important layers, which can hurt performance.

4.2.3 Criteria

The criteria is a fundamental component of any sparsifying technique as it acts as a proxy for weight importance. In practice, applying the desired criteria to each group of weights returns a score, according to which the selection of remaining weights will be based. The groups of weights having the lowest score will be zeroed out first in the sparsifying process, while those having the largest will be retained. There currently exist many sparsifying criteria, with 14 currently available by default in `FasterAI`. Most of them have been described in Section 3.3.3 and are now expressed in a simplified way, following PyTorch notation, in Pseudo-Code 4.4. To that end, we define `wi` and `wf` respectively as the initial and final values of the weights, i.e. their values at initialization and at the current training step. Additionally, `FasterAI` keeps track of the values of the weights during training. This paves the way to creating criteria using first-order information, thus considering the training dynamics.

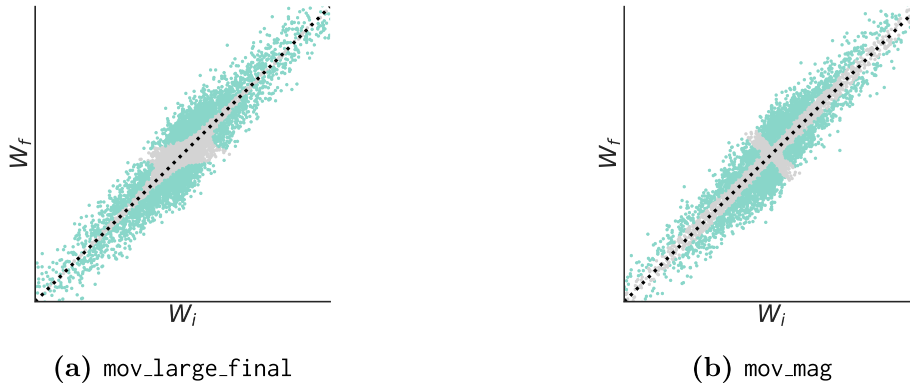
```
random                  = torch.randn_like(wf)
large_final             = torch.abs(wf)
squared_final           = torch.square(wf)
small_final             = torch.neg(torch.abs(wf))
large_init              = torch.abs(wi)
small_init              = torch.abs(torch.neg(wi))
large_init_large_final = torch.abs(torch.min(wf, wi))
small_init_small_final = torch.abs(torch.neg(torch.max(wf, wi)))
magnitude_increase      = torch.sub(torch.abs(wf),torch.abs(wi))
movement                = torch.abs(torch.sub(wf,wi))
```

Pseudo-Code 4.4. The list of all criteria available in `FasterAI` and their corresponding PyTorch implementation.

Because of how the criteria are implemented in `FasterAI`, it is very convenient to create custom criteria. Indeed, implementing new selection criteria boils down to writing a single function, to be applied to each weight before the computing of the sparsification mask. For example, we introduce a novel criteria named `mov_large_final`, similar to the `movement` one, but emphasizing the final value of weights more. Similarly, we introduce, `mov_mag`, which considers weights whose absolute value has moved the most. Those criteria are expressed in Pseudo-Code 4.5 and represented in Figure 4.1.

```
mov_large_final = torch.abs(torch.mul(x, torch.sub(x,y))))
mov_mag         = torch.abs(torch.sub(torch.abs(x),torch.abs(y))))
```

Pseudo-Code 4.5. Custom criteria and their corresponding implementation in PyTorch.



(a) `mov_large_final` (b) `mov_mag`

Figure 4.1. Result of the application of our proposed criteria on the weights. Parameters that are greyed out are the parameters removed.

In Table 4.3, we report the comparison between all the available criteria. Experiments are conducted in the same conditions as previous ones. The granularity is fixed to `weight`, the context to `local`, and the schedule to `one_cycle`. For readability constraints, the name of criteria in Table 4.3 have been abbreviated, e.g. `large i,f` corresponds to the `large_i_large_f` criteria.

	Scratch				Fine-Tune			
	30%	50%	70%	90%	30%	50%	70%	90%
Baseline	80.61 ± 0.42				90.03 ± 0.54			
random	80.53±0.70	80.38±0.23	80.01±0.94	66.34±0.92	90.24±1.47	90.32±0.55	86.38±0.80	30.23±4.25
large f	80.03±1.07	80.93±0.97	80.82±0.48	77.99±0.72	91.41±0.43	91.17±0.32	91.41±1.08	88.09±0.45
small f	79.05±0.66	75.42±0.44	48.49±1.42	19.38±3.42	84.37±0.61	74.20±0.50	18.76±0.27	0.91±0.52
sq f	80.76±0.71	79.34±1.10	79.58±0.58	78.61±0.23	91.39±0.35	90.99±0.30	90.97±0.12	88.97±0.92
large i	80.42±0.85	79.67±0.44	79.61±0.73	75.62±0.25	91.56±0.52	92.58±0.49	92.34±1.13	90.30±0.49
small i	80.67±0.90	80.78±1.19	79.60±0.74	73.19±0.50	86.52±0.09	85.61±1.03	83.42±1.20	0.88±0.31
large i,f	80.03±0.43	80.07±0.54	79.18±0.57	72.37±0.78	92.21±0.43	92.43±0.89	91.74±0.12	88.00±0.67
small i,f	80.25±0.97	76.39±0.83	65.50±0.76	23.27±1.66	85.23±0.83	76.75±1.07	31.27±1.31	3.48±3.88
mag inc	80.40±0.44	80.27±0.34	79.72±0.16	79.19±1.99	90.83±0.92	90.06±0.52	88.88±0.38	83.13±0.61
mov	80.45±0.63	79.98±0.43	80.20±0.39	78.26±0.37	89.73±0.19	89.99±0.43	88.58±0.63	80.56±1.23
movmag	80.16±1.16	82.24±0.67	81.47±0.38	79.12±0.14	90.90±0.14	90.96±0.99	89.93±0.22	86.18±0.48
movmag	80.05±0.37	80.99±0.97	80.32±0.61	79.69±0.57	90.12±1.52	90.55±0.83	90.06±0.34	88.22±0.25

Table 4.3. Results of sparsifying ResNet-18 for all available criteria. Granularity, context and schedule are respectively set to `weight`, `local` and `one_cycle`. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

From those results, we can observe that the criteria has a minor effect on the performance for low sparsity levels. Indeed, the network still possesses enough capacity to compensate for removed weights and achieve decent performance. When the sparsity level increases, criteria based on lower weight values, e.g. `small f` and `small i` perform poorly. This happens because weights having low values do not participate much in final results, and thus are not holding much discriminative information about the data.

4.2.4 Schedule

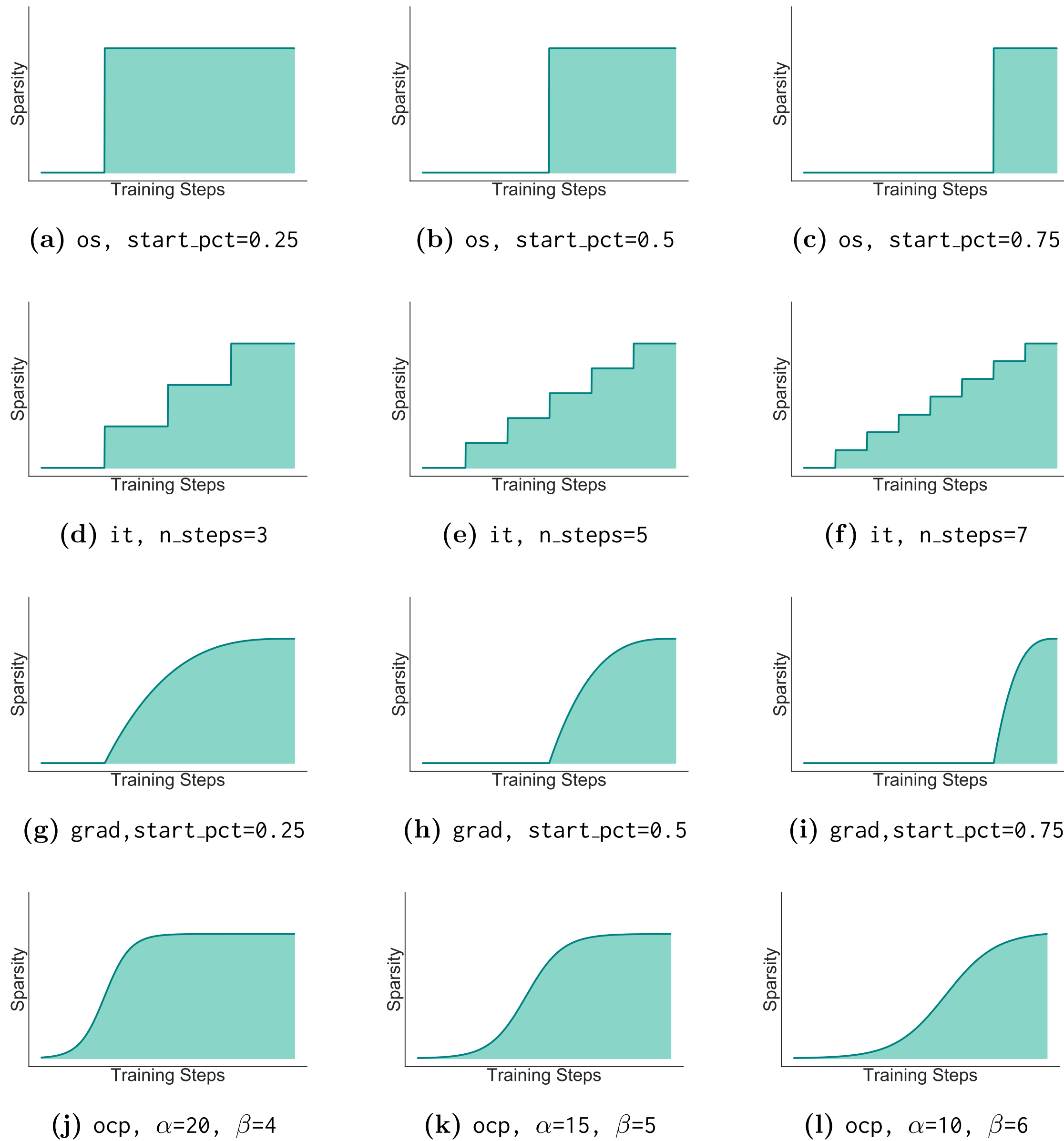
The last argument required in the `SparsifyCallback` is the sparsification schedule. It defines when the sparsification process will occur during the training phase. In `FasterAI`, all schedules are implemented within a single class. To do so, each schedule has to be defined according to three parameters:

- `start_pct`: the percentage of training at which the sparsification process starts, i.e. for how long the model will be pre-trained.
- `end_pct`: the percentage of training at which the sparsification process stops, i.e. for how long the model will be fine-tuned after being sparsified.
- `schedule_function`: the function describing the evolution of the sparsity during the training. There are four currently available by default: `one_shot`, `iterative`, `gradual` [59], and `one_cycle` [69], expressed in the Pseudo-Code 4.6.

```
def one_shot(sparsity, train_step): return sparsity

def iterative(sparsity, train_step, n_steps=5):
    return  (sparsity/n_steps)*(np.ceil((train_step)*n_steps))

def gradual(sparsity, train_step): return sparsity * (1 - train_step)**3

def one_cycle(sparsity, train_step, α=14, β=6):
    return (1+np.exp(-α+β)) / (1 + (np.exp(-α*train_step+β)))*sparsity
```

Pseudo-Code 4.6. Available schedules in `FasterAI` and their PyTorch implementation.

In Figure 4.2, we represent variations of the four available sparsifying schedules. As can be observed, adjustments to `start_epoch` and `end_epoch` can be made to further help the user customize the pruning schedule as desired. For example, in Figure 4.2a, the One-Shot pruning schedule could also be used with a value of `start_pct=0`, becoming what is more well-known as Pruning at Initialization [70], achieving the target amount of sparsity right from the start of training. For readability constraints, we abbreviate the names of our schedules, e.g. `one_shot` becomes `os`.



(a) os, start_pct=0.25 (b) os, start_pct=0.5 (c) os, start_pct=0.75

(d) it, n_steps=3 (e) it, n_steps=5 (f) it, n_steps=7

(g) grad,start_pct=0.25 (h) grad, start_pct=0.5 (i) grad,start_pct=0.75

(j) ocp, α=20, β=4 (k) ocp, α=15, β=5 (l) ocp, α=10, β=6

Figure 4.2. Evolution of sparsity along training for the available pruning schedules. While the sched_func parameters defines the general evolution, the schedule can further be customized by modifying the start_pct and end_pct values.

We report in Table 4.4 the results of applying the schedules represented in Figure 4.2. Experiments have been conducted in the same training conditions as previous ones. The granularity has been set to weight, context to local and criteria to large_final. For readability constraints, in Table 4.4, the name of the schedule directly refers to the subfigure index in Figure 4.2.

	Scratch				Fine-tune			
	30%	50%	70%	90%	30%	50%	70%	90%
Baseline	80.61 ± 0.42				90.03 ± 0.54			
(a)	79.60±1.02	79.69±0.69	79.29±0.38	77.33±0.25	91.47±0.54	91.48±0.27	91.63±0.13	91.08±0.34
(b)	80.32±0.95	81.02±0.39	80.31±0.31	78.59±0.94	91.99±0.56	91.28±0.18	91.65±0.36	90.55±0.73
(c)	80.58±0.73	80.12±0.34	80.49±0.61	79.45±1.00	91.90±0.74	90.94±0.48	91.34±0.57	89.28±0.34
(d)	80.73±0.14	80.31±0.71	80.31±0.43	77.94±1.05	91.59±0.94	91.28±0.25	91.32±0.20	88.42±0.18
(e)	80.65±1.12	80.49±1.07	81.33±0.80	72.39±1.34	90.86±0.18	90.65±0.50	90.34±0.63	85.47±0.67
(f)	81.05±0.67	81.05±0.52	79.80±0.72	69.91±3.01	90.90±0.34	90.63±0.34	91.08±0.31	80.94±0.57
(g)	80.78±0.55	80.01±0.41	80.65±0.69	79.54±0.83	90.96±0.43	90.43±0.65	91.52±0.57	90.94±0.49
(h)	80.31±0.86	81.24±0.58	80.47±0.54	79.96±0.42	91.34±0.28	91.05±0.97	90.96±0.41	89.72±0.47
(i)	81.25±1.18	81.35±0.88	80.65±1.21	78.26±0.94	90.86±0.69	91.36±0.43	91.25±0.66	87.24±0.65
(j)	80.56±0.76	80.74±0.26	81.27±0.69	79.60±0.12	91.45±0.53	91.89±0.05	92.01±0.65	91.87±0.22
(k)	80.74±0.27	80.11±0.86	79.08±0.35	78.83±0.38	91.58±0.50	91.94±0.56	92.03±0.85	90.61±0.45
(l)	80.32±0.36	81.25±0.33	80.80±0.43	79.46±0.62	91.58±0.23	90.83±0.19	91.87±0.51	88.13±0.20

Table 4.4. Results of sparsifying ResNet-18 for all available criteria. Granularity, context and criteria are respectively set to `weight`, `local` and `large_final`. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

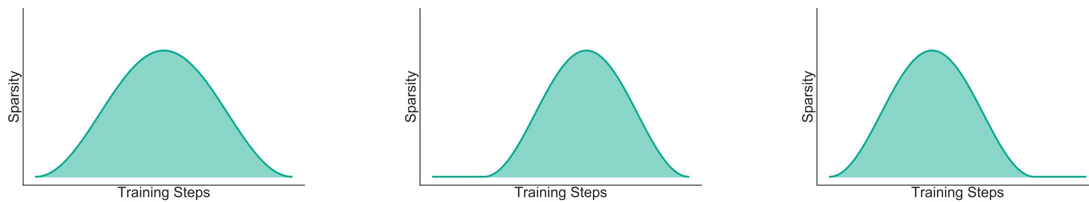
As can be observed, schedules implying a weight removal later in training seem to produce suboptimal results, especially in the fine-tuning regime. Indeed, removing parameters close to the end of training does not provide enough time for the network to adjust its remaining weights to accommodate its weight loss. Also, schedules producing a gradual increase in sparsity, such as the Gradual and One-Cycle, seem to provide better and more stable results.

By modifying the three schedule parameters, users can also create their own pruning schedule or easily implement other existing ones, such as the Dense-Sparse-Dense (DSD) schedule [71] for example, which increases the sparsity for the first half of training, then gradually decay it until the network is 0% sparse again. The corresponding `schedule_function`, defining how the sparsity evolves with such a schedule, is provided in Pseudo-Code 4.7.

```python
def dsd(sparsity, t_step):
    if t_step<0.5: return (1 + math.cos(math.pi*(1-t_step*2))) * sparsity / 2
    else: return (1 - math.cos(math.pi*(1-t_step*2))) * sparsity / 2
```

Pseudo-Code 4.7. Implementation of the Dense-Sparse-Dense technique in `FasterAI`.

By then modifying the values of `start_pct` and `end_pct` in the `SparifyCallback`, we can further customize our pruning schedule, as displayed in Figure 4.3. Such `schedule_function` also shows that it is possible not only to use a schedule to perform sparsification, but also weight growing, i.e. start from a sparse network, and gradually retrain previously zeroed-out connections.



(a) `dsd, start_pct=0, end_pct=1` **(b)** `dsd, start_pct>0, end_pct=1` **(c)** `dsd, start_pct=0,end_pct<1`

Figure 4.3. Variation of the `dsd` schedule. The use of `start_epoch` and `end_epoch` help to further customize a given pruning schedule by affecting the start and end of the pruning process.

4.2.5 Lottery Ticket Hypothesis

In Section 3.3.4, we described the Lottery Ticket Hypothesis experiments, that are used to discover the optimal subnetworks from initialization. Those experiments were conducted by removing individual weights, globally, according to their l_1-norm, and by following an iterative schedule. `FasterAI` handles such LTH experiments by default but allows to expand them to any granularities, contexts, criteria and schedules, opening the way to many novel experiments about finding winning tickets. To accomplish such procedure in `FasterAI`, some additional arguments can be provided to the `SparsifyCallback`:

- `lth`: whether to perform LTH, i.e. reinitialize weights to their saved value after each pruning round.
- `rewind_epoch`: the epoch of training where weights values are saved for further reinitialization.
- `reset_end`: whether to reset the weights to their saved values after training.

Performing the classic Lottery Ticket Experiments [56, 72] following the original pruning settings in `FasterAI` can be achieved with Pseudo-Code 4.8.

```
# Classic LTH
SparsifyCallback(sp,'weights','global', large_final, iterative, lth=True)

# LTH with Rewinding
SparsifyCallback(sp, 'weights', 'global', large_final, iterative, lth=True,\
rewind_epoch=1)
```

Pseudo-Code 4.8. Changes to `SparsifyCallback` in order to perform Lottery Tickets Experiments.

In Table 4.5, we report the results obtained when performing the classic LTH and LTHR techniques using the same architecture and datasets as previous experiments. Each LTH round is performed for 30 epochs and, when using rewinding, the weights are rewound to their epoch 1 value. We can observe that, in our case, both techniques provide similar results. Also, results show that it is possible to find high-performing pruned networks, even for high sparsity levels.

LTH				**LTH with Rewind**			
30%	50%	70%	90%	30%	50%	70%	90%
80.05±0.72	81.86±0.40	84.59±0.09	84.79±0.37	80.65± 1.02	82.97± 1.15	83.92± 0.49	84.77±0.46

Table 4.5. Results of pruning ResNet-18 and VGG-16 with 4 different schedules. Mean and standard deviation of accuracy over 3 rounds are reported.

4.2.6 Prune

As described previously, sparsification is usually introduced by applying a binary mask, multiplying the value to keep by 1, and those to remove by 0. This leads to a sparse network, difficult to accelerate in practice. However, as described in Section 3.3.1, some particular granularities allow the sparse weights to be physically removed from the network, effectively taking advantage of the compression to witness speed-up without any dedicated resource. Two granularities allow performing such a feature: (1) `filter` and (2) `shared-kernel`.

As it removes parameters that have no impact on the computation of the result, the pruning is considered to be lossless, as it reduces the number of parameters and operation of the network, without altering its performance. To perform such an operation in `FasterAI`, the code required is expressed in Pseudo-Code 4.9, with `model` being the model, sparsified according to the filter granularity beforehand.

```
pruner = Pruner()
pruned_model = pruner.prune_model(model)
```

Pseudo-Code 4.9. Code required to prune a filter-sparse model with `FasterAI`.

4.3 Distill

`FasterAI` also brings Knowledge Distillation capabilities to users with the help of its `Distill` module. This is managed by `KnowledgeDistillationCallback`, which offers Knowledge Distillation capabilities in a single line of code. As Knowledge Distillation is managed by an independent callback, it can be used in conjunction with `SparsifyCallback`, for even more flexibility to extreme compression or perform original experiments. The `FasterAI` usage for the `KnowledgeDistillationCallback` is given below in Pseudo-Code 4.10, where `layers_std` and `layers_tch` are optional lists of layers, which will be used to compute the feature loss L_f if desired, and α, β the interpolation parameters as defined in Equation 2.4.

```
kd_cb = KnowledgeDistillationCallback(tch, L_l, L_f, layers_std, layers_tch, α, β)
```

Pseudo-Code 4.10. Code required to perform Knowledge Distillation in `FasterAI`.

Knowledge Distillation losses can be modified or created according to the user's needs. There are currently three logit losses and four feature losses available by default in `FasterAI`. We compare two of those losses in the same training conditions as previous experiments. In this scenario, the teacher model is a ResNet-34 model trained for 30 epochs from pre-trained weights, and the student is a ResNet-18 model starting from random initialization. In particular, we compare a logit loss to a feature loss, used individually for different interpolation levels: (1) `SoftTarget`, the loss computed between the logits of the teacher and the student and (2) `Attention`, a loss computed from features extracted after each residual block of the teacher and the student. We report the results in Table 4.6. It can be observed that basing the Knowledge Distillation process on logits provides better results than Attention. While `SoftTarget` compares the respective predictions of the teacher and the student, `Attention` holds a stronger hypothesis, that the layers used to compare are extracting the same information, which can make it harder to set up correctly.

SoftTarget				Attention			
0.3	0.5	0.7	0.9	0.3	0.5	0.7	0.9
83.32±0.14	84.65±0.31	84.74±0.49	84.34±0.17	81.56±0.25	81.55±0.41	81.64±0.27	81.73±0.67

Table 4.6. Results of applying Knowledge Distillaion from a ResNet34 to a ResNet18 architecture for different interpolation values of β. Mean and standard deviation of accuracy over 3 rounds are reported.

4.4 Regularize

The `regularize` module of `FasterAI` concerns regularization techniques reducing the magnitude of weights in the network, as described in Section 2.2.2. Moreover, it allows to do so according to a chosen granularity. Such regularization can be applied to groups of weights, according to a chosen granularity. However, as it is dependent on the optimization process, the sparsity level cannot be defined beforehand. It is nonetheless possible to control the importance of the penalty, to impose more or less sparsity in the final network, thanks to a penalty factor α. The final loss thus receives an extra term, adding the absolute value of weights for each layer l, according to the chosen granularity, as:

$$\mathcal{L} = L_c + \alpha \sum_l R(W_l) \tag{4.1}$$

With L_c the classification loss, generally a cross-entropy computed between predictions and labels, and $R(W_l) = \frac{1}{G} \sum_g \sum_i |w_{g,i}|$ the regularization term, G being the number of elements in each group.

Such regularization can be applied in `FasterAI` by using the `RegularizationCallback`, is presented in Pseudo-Code 4.11.

```
reg_cb = RegularizationCallback(granularity, α)
```

Pseudo-Code 4.11. Code required to perform Group Regularization in `FasterAI`.

We provide results of experiments conducted for different values of penalty α in Table 4.7, and for all of available granularities. Those experiments are using same the dataset and architecture as previous ones.

	1e-7	1e-5	1e-3	1e-1
Baseline		80.61 ± 0.42		
weight	80.91±0.44	79.45±0.88	54.50±0.94	23.91±0.54
column	80.45±0.29	81.02±1.32	65.68±1.32	25.91±0.41
row	80.62±1.31	79.39±0.58	65.35±0.49	25.67±0.92
s-weight	80.11±0.30	80.49±0.65	80.67±0.42	66.05±0.59
channel	80.40±0.49	81.56±0.31	80.67±0.11	63.02±1.14
kernel	80.34±0.14	81.29±0.16	73.16±0.14	27.81±0.84
s-channel	80.35±0.63	79.94±0.59	80.31±0.90	81.00±1.22
s-column	80.09±1.14	80.80±0.31	81.46±0.23	72.96±0.49
s-row	80.71±0.91	81.47±0.30	80.29±0.83	74.25±0.21
v slice	80.46±0.92	80.45±0.52	80.43±0.35	72.96±0.84
h slice	80.73±0.40	80.85±0.85	80.98±0.76	71.85±0.63
s-v-slice	81.02±0.74	81.31±0.39	80.58±0.74	80.56±1.52
s-h-slice	80.40±0.14	80.98±0.89	80.12±1.07	81.33±1.92
s-kernel	81.42±0.59	80.93±0.57	81.66±0.41	77.70±0.84
filter	81.13±0.43	81.15±1.31	80.56±0.47	76.48±1.01

Table 4.7. Results of regularizing ResNet-18 with four different penalty strengths. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

As can be observed, a higher value of α leads to a degradation in accuracy, as too much penalty is added to the loss value, making the optimization process put more emphasis on having small magnitude weights instead of an accurate network. Moreover, we can see that, as opposed to sparsifying, regularization performs better for more coarse granularities. This can be explained by the fact that the penalty value is dependent on the granularity structure, as the l_1-norm is averaged over the size of each block. This means that smaller structures will be penalized more, with the regularization term driving the loss value, thus giving more importance to the l_1-norm of weights than to the correct classification of data.

4.5 Misc

The last module of `FasterAI` is composed of compression techniques that do not fall into previous categories, but might potentially deserve their own category in the future. Such techniques for example include:

- Batch Normalization Folding
- Fully-Connected layer decomposition

4.5.1 Batch Normalization Folding

It is possible to perform Batch Normalization Folding in `FasterAI`, i.e. remove the batch normalization layers from the trained network. This consists in replacing the values of convolutions preceding the batch normalization layer using Equation 2.20. Once the weights and bias of the computation layer have been modified, the batch normalization layer can be considered useless and removed, slightly reducing the network's total amount of parameters and computation. This operation can be achieved in `FasterAI` by following the Pseudo-Code 4.12.

```
bn = BN_Folder()
bn.fold(model)
```

Pseudo-Code 4.12. Code required to perform Batch Normalization Folding in `FasterAI`.

Batch Norm Folding is thus another lossless compression technique as it does not affect the performance of the neural network but helps to reduce the number of parameters and computation of the model. The results of applying batch normalization folding to a trained ResNet-18 are reported in Table 4.8. As can be observed, the accuracy remains the same while the parameter count slightly decreases.

	Trained Model	BN Folded Model
Accuracy (%)	80.61±0.42	80.61±0.42
# Parameters	11228838	11224038

Table 4.8. Results of performing batch normalization folding on ResNet-18. Mean and standard deviation of accuracy over three rounds are reported.

4.5.2 Fully-Connected Layers Decomposition

Fully-Connected Layers Decomposition, as described in Section 2.5.1, can also be performed in `FasterAI` to further reduce the number of parameters contained in Fully-Connected Layers. In `FasterAI`, this corresponds to performing a truncated-SVD, which can be achieved in `FasterAI` by applying the Pseudo-Code 4.13.

The `pct_removed` term corresponds to the percentage of singular values kept from the matrix of singular values Σ. The results of applying Fully-Connected Layers Decom-

```
fc = FC_Decomposer()
fc.decompose(model, pct_removed)
```

Pseudo-Code 4.13. Code required to perform Knowledge Distillation in `FasterAI`.

position on a trained ResNet-18 are reported in Table 4.9. As can be observed, a high compression rate can be achieved before affecting the performance negatively.

	Trained Model	pct_removed 25%	pct_removed 50%	pct_removed 75%
Accuracy (%)	80.61±0.42	80.84±0.17	80.67±0.61	77.24±0.74
# FC Parameters	52326	46766	31416	15452

Table 4.9. Results of decomposing the FCLs of ResNet-18 with 3 different compression levels. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

4.6 In Brief

Summary 4

- `FasterAI` is a PyTorch-based library, that leverages the callback system of fastai and PyTorch Lightning to provide **extensive** and **readily-available** compression techniques.
- `FasterAI` is built on four modules, each dedicated to a research field of neural network compression: (1) `sparsify`, concerned with techniques introducing sparsity in the weights; (2) `distill`, concerned with techniques related to Knowledge Distillation; (3) `regularize`, concerned with techniques reducing the value of weights during training; (4) `misc`, related to other compression techniques.
- The core of `FasterAI` lies in the `SparsifyCallback` class, allowing to compose any pruning technique by customizing its parameters: `granularity`, `context`, `criteria` and `schedule`.
- Experiments conducted for each module demonstrate the **flexibility** of `FasterAI` and the ability to **combine** and **customize** different compression techniques.

CHAPTER 5

Advances in Neural Network Pruning

Contents

5.1 Introduction **96**
5.2 How to prune? **96**
5.2.1 Methodology 97
5.2.2 Experiments 100
5.2.3 Discussion & Conclusion 105
5.3 Where to prune? **106**
5.3.1 Methodology 108
5.3.2 Experiments 110
5.3.3 Discussion & Conclusion 116
5.4 What to prune? **116**
5.4.1 Methodology 117
5.4.2 Experiments 120
5.4.3 Discussion & Conclusion 124
5.5 When to prune? **124**
5.5.1 Methodology 125
5.5.2 Experiments 126
5.5.3 Discussion & Conclusion 131
5.6 In Brief **132**

"Less is more."
— Mies Van Der Rohe

5.1 Introduction

This chapter presents contributions in the field of neural network pruning that have been proposed throughout the realization of the thesis. As presented in Section 3, we identify four open research axis regarding neural network pruning, which we formalize into four questions:

- How to prune?
- Where to prune?
- What to prune?
- When to prune?

The present chapter is therefore subdivided into four sections, each one aiming to provide elements of answer to each one of those questions.

5.2 How to prune?

When it comes to pruning granularities, the most commonly used ones are those lying at both ends of the spectrum presented in Section 3.3.1, i.e. weight pruning and filter pruning. Weight pruning is very popular because it allows to operate on the smallest structures available in a network, thus being very precise in the removal of the weights. As a result, weight pruning allows to reach high sparsity level without degrading a model's performance. However, because of how scattered the weight removal is, it is challenging to take advantage of such sparsity for actual speed-up and compression. On the other hand, filter pruning allows to operate at one of the most coarse structures in the network. Consequently, keeping the performance intact requires imposing a limited level of sparsity. Although it does not allow to reach comparable levels of sparsity to weight pruning, removing entire filters allows for a more significant and straightforward speed-up because it intrinsically changes the neural network architecture to a smaller one.

As presented in Section 3.3.1, we propose another granularity able to provide practical speed-up. We call it *shared-kernel* as it removes kernels in a position that is shared with every filter in the same layer. As a result, the architecture can also be changed to a smaller one, hence providing speed-up.

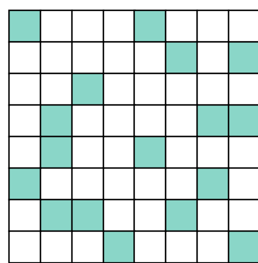
In this contribution, we thus would like to compare the performances of filter pruning to those of shared-kernel. In particular, we highlight a distinct difference in how both granularities impact input images and describe our results in the following section.

5.2.1 Methodology

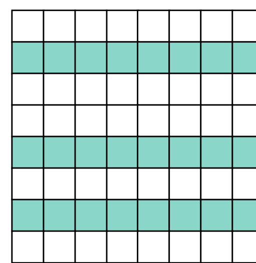
To better grasp the implications of filter and shared-kernel granularities, we first introduce the general concept with Multi-Layer Perceptrons.

Pruning of Multi-Layer Perceptrons

In Multi-Layer Perceptrons, parameter pruning can be performed in two ways: (1) in an unstructured manner, removing individual connections in the network; (2) structured, removing complete neurons and all the related connections. If we represent the weights of each layer as 2D matrices of shape $M \times N$, with M and N being respectively the dimension of output and input of a layer, then unstructured pruning removes individual values in the matrix, while structured pruning removes a complete weight row. Those two methods are represented in Figure 5.1.



(a) Unstructured Pruning



(b) Structured Pruning

Figure 5.1. Different types of sparsity in a fully-connected weight matrices. Zeroed weights are in color.

We express in Pseudo-Code 5.1 the different methods of weight selection, following `NumPy` standards.

```
Weight (0D) = Weights[m,n]
Row (1D) = Weights[m,:]
```

Pseudo-Code 5.1. Granularity selection for a Fully-Connected Layer.

As depicted in Figure 5.2, when zeroing a complete row in the weight matrix of layer i, this has for effect to also zero out the corresponding output result in the hidden layer. As the next layer now has an input of smaller dimension, the following weight matrix of layer $i + 1$ must be adapted accordingly, i.e. corresponding columns are zeroed out.

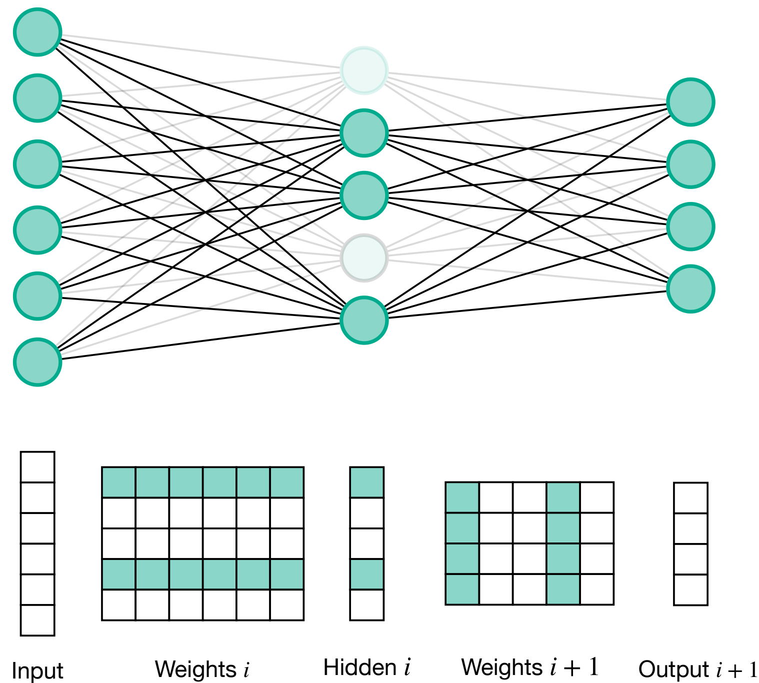


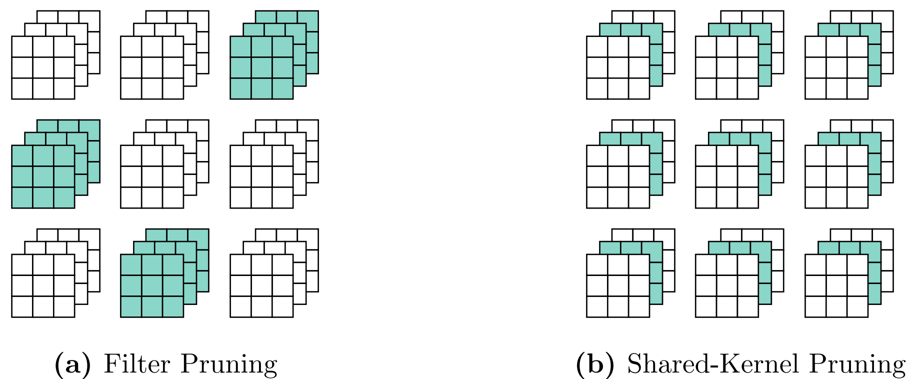
Figure 5.2. Pruning rows in layer i impacts its output and consequently the columns of layer $i+1$. Zeroed weights are in color.

The selection of which neuron to remove is usually performed by applying a pruning criteria to its input weights. However, instead of selecting neurons to remove based on their input weights, we can base the selection on their output weights. To do so, we remove a column in the weight matrix, which has the effect of removing the neuron from the previous layer instead of the current one. By referring to Figure 5.2, this would be equivalent to basing the removal selection on columns in the weight matrix of layer $i+1$, leading to the removal of rows in the weight matrix of layer i. Column pruning thus acts as the inverse operation of row pruning and leads to different neurons being dropped. Moreover, because pruning is usually applied to each layer of the network, both methods have a significant difference in their impact on the first and last layers of the network. Indeed, row pruning leads to the removal of neurons from the last layer, while column pruning leads to the removal of neurons from the input layer.

The column pruning is particularly interesting because it acts as a feature selection method that discards input data. Indeed, each neuron of the input layer corresponds to an input data feature, e.g. a pixel from an image. For this reason, removing a single column from the first layer's weight matrix removes the corresponding data feature from all input data. We can thus decide on the amount of input information to keep by selecting the sparsity level in the first weight matrix.

Pruning of Convolutional Neural Networks

As described in Section 3.3.1, removing filters in a Convolutional Neural Network impacts the feature map and further requires removing every kernel that correspond to it. This is thus the equivalent of row pruning for MLP. The opposite operation, which would be the equivalent column pruning of MLP, is to base the pruning process on removing the same kernel in each filter of a layer, which in turn allows the removal of the previous filter corresponding to that kernel. As it requires removing a kernel from a position shared from all filters in the layer, we name this granularity *shared-kernel*. Both techniques are represented in Figure 5.3.



(a) Filter Pruning **(b)** Shared-Kernel Pruning

Figure 5.3. Different types of structured granularities for CNNs. Those granularities allow to remove parts of the network, leading to a smaller architecture.

Those two granularities have an equivalent impact on the architecture for all intermediate layers. However, those two approaches have a key difference at the very first and very last convolutional layers. Indeed, for the very first layer, the previous inputs are the model inputs. Removing a shared-kernel from the first convolution layer is thus equivalent to removing a channel to all input images. This last side-effect is the one we want to take advantage of. Shared-kernel pruning not only allows a change in architecture, effectively inducing compression and speed-up, but also acts as a feature selection, reducing the input dimension by removing input channels. The selection of filters and shared-kernels following the `NumPy` standard is provided in Pseudo-Code 5.2.

```
Weights = Array(O, I, Kh, Kw)

Shared-Kernel (3-Dim) = Weights[:,i,:,:]
Filter(3-Dim) = Weights[o, :, :, :]
```

Pseudo-Code 5.2. Filter and Shared-Kernel Selection.

5.2.2 Experiments

Experiments are conducted on two types of architectures: Multi-Layer Perceptrons and Convolutional Neural Networks. In particular, we want to compare the performance of the different structured granularities, but also investigate how it affects input data.

Multi-Layer Perceptrons

In our first experiment, we compare the effects of row or column pruning on the performance of an MLP and how pruning induces a feature selection of input data.

Pruning Method. The different parameters used for performing pruning are summarized in Table 5.1, expressed using `FasterAI` terminology. In our experiments, we compare the row and column granularities. The weight importance is evaluated locally, using the `large_final` criteria, selecting weights having the largest l_1-norm. Moreover, the schedule followed for pruning is the One-Cycle Pruning, gradually adjusting the sparsity level during the training [69]. Because the network only contains fully-connected layers, we apply pruning only on the two first layers, avoiding issues occurring when row pruning in the last layer, significantly degrading performance.

Granularity	Criteria	Context	Schedule
`row` vs `column`	`large_final`	`local`	`one_cycle`

Table 5.1. Pruning parameters used in `FasterAI`.

Datasets and Architecture. For our experiments, the datasets are chosen to be various in terms of image resolution and number of classes. In particular, we have selected three datasets: MNIST [73], CIFAR-10 [74], CIFAR-100 [74]. Those datasets are used to train a 3-Layer MLP, with ReLU non-linearities between each computation layer.

Training Procedure. Our network is trained from randomly initialized weights. For datasets involving color images, we first convert them to grayscale, following a weighted average of color channels [75], to keep the same architecture for all MLP experiments. Models are trained for 20 epochs, using a learning rate warm-up method [76] until a nominal value of $1e-3$, then gradually decay until the end of the training.

In Table 5.2, we report the results for the two studied granularities, pruned to 25%, 50% and 75% of sparsity.

		Row	Columns
MNIST			
Sparsity	25%	**97.79 ± 0.12**	97.71 ± 0.16
	50%	97.75 ± 0.10	**97.90 ± 0.13**
	75%	96.81 ± 0.20	**97.30 ± 0.16**
CIFAR-10			
Sparsity	25%	40.93 ± 0.39	**41.73 ± 0.21**
	50%	39.72 ± 0.37	**42.05 ± 0.80**
	75%	38.11 ± 0.27	**42.63 ± 0.33**
CIFAR-100			
Sparsity	25%	13.71 ± 0.33	**14.11 ± 0.19**
	50%	12.90 ± 0.22	**14.09 ± 0.15**
	75%	11.62 ± 0.14	**15.01 ± 0.34**

Table 5.2. Accuracies of MLP model for three different datasets and for three levels of sparsity. Reported values are mean and standard deviation over 5 iterations

Although not reaching decent performance on all datasets due to the lack of capacity of the network, columns pruning overperforms row pruning in most cases. Also, we can analyze which input features were removed when column pruning, by matching the removed weights to the corresponding input features, as depicted in Figure 5.4.

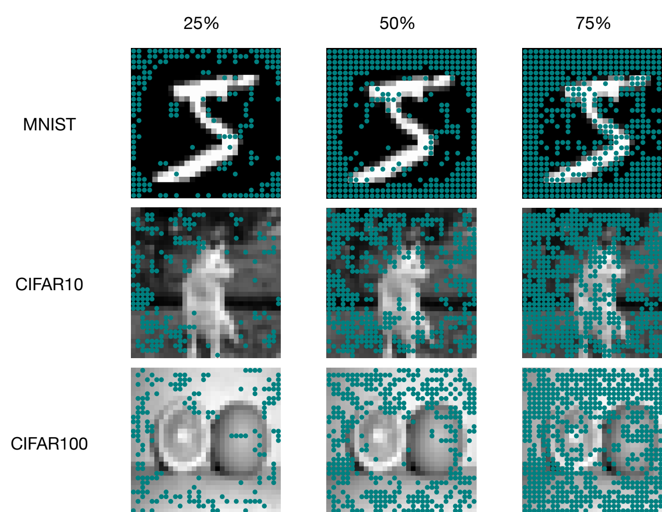


Figure 5.4. Pixels removed after column pruning. Colored pixels can be removed from all input images, effectively reducing the storage need of the dataset.

We can observe a general trend of preserving central pixels, corresponding to the region where most of the important information is contained [77].

Being able to retrieve the information about which pixels are the most useful in a dataset for training has important implications. Indeed, the results presented in Table 5.2 show that training a model with only 25% of input pixels does not negatively affect the network's performance, and sometimes even seems to help it reach better accuracy. This implies that, instead of storing the images in full resolution, only a part of the image can be stored and used for training. This allows for a greater reduction of the storage cost of the dataset, as well as a reduction of the computation cost of the model.

Convolutional Neural Networks

In the second experiment, we evaluate the effects of filter or shared-kernel pruning on the network's performance and how it affects incoming data.

Pruning Method. The different parameters used for performing pruning are summarized in Table 5.3, where they are expressed using `FasterAI` terminology. In our experiments, we compare the filter granularity to the shared-kernel granularity. The comparison is performed according to the global context, to better highlight how those granularities differ in the selection of parameters to remove along the network. The criteria according to which weight importance is evaluated is `large_final`, meaning that the group of weights having the lowest l_1-norms are considered less useful and removed. Moreover, the schedule followed for pruning is the One-Cycle Pruning.

Granularity	Criteria	Context	Schedule
`filter` vs `shared_kernel`	`large_final`	`global`	`one_cycle`

Table 5.3. Pruning parameters used in `FasterAI`.

Datasets and Architectures. We evaluate our method on the ResNet-18 [67] architecture, trained on three datasets having a good variability of number of classes, image resolution and content. As targeted datasets, we have selected CIFAR-10 [74], CIFAR-100 [74],and the CALTECH-101 corpus [68].

Training Procedure. When experimenting with CNNs, we consider different initialization schemes. Indeed, networks are either trained from scratch, i.e. random initialization, or fine-tuned from pre-trained weights. The images are first resized to

224 × 224 and are augmented by using horizontal flips, rotations, warping and random cropping. CNNs are trained for 50 epochs, with a learning rate warm-up method [76] until a nominal value of $1e-3$, then gradually decay until the end of the training.

The results of our experiment, performed for four levels of sparsity: 30%, 50%, 70% and 90% are reported in Table 5.4. As can be observed, the shared-kernel pruning overperforms filter pruning for most datasets and sparsity levels. More particularly, the difference is more noticeable for high sparsity levels, where pruning based on filter granularity often drops dramatically.

		Scratch		**Fine-tune**	
		Filter	Shared-Kernel	Filter	Shared-Kernel
CIFAR-10					
Sparsity	30%	**94.69 ± 0.08**	94.56 ± 0.04	95.52 ± 0.07	**95.76 ± 0.37**
	50%	92.60 ± 0.17	**92.70 ± 0.74**	94.22 ± 0.19	**94.64 ± 0.13**
	70%	91.11 ± 0.17	**91.13 ± 0.25**	91.41 ± 0.24	**91.61 ± 0.35**
	90%	50.47 ± 4.24	**79.70 ± 0.86**	80.55 ± 1.71	**82.86 ± 0.18**
CIFAR-100					
Sparsity	30%	76.34 ± 0.19	**76.00 ± 0.25**	**74.54 ± 0.19**	74.06 ± 0.33
	50%	74.51 ± 0.34	**74.57 ± 0.11**	68.87 ± 0.36	**69.89 ± 0.25**
	70%	70.06 ± 0.83	**70.06 ± 0.80**	61.96 ± 1.34	**66.55 ± 0.18**
	90%	21.55 ± 3.54	**47.36 ± 2.66**	34.18 ± 2.63	**49.66 ± 3.80**
CALTECH-101					
Sparsity	30%	82.24 ± 0.72	**82.36 ± 0.46**	90.86 ± 0.61	**90.99 ± 0.27**
	50%	80.94 ± 0.64	**81.31 ± 1.23**	87.33 ± 0.27	**89.68 ± 0.74**
	70%	70.50 ± 0.63	**75.93 ± 0.49**	85.69 ± 0.54	**87.20 ± 0.55**
	90%	34.99 ± 1.86	**61.58 ± 1.72**	60.27 ± 1.81	**70.97 ± 1.54**

Table 5.4. Comparison of accuracies of ResNet-18 trained on three different datasets and for four levels of sparsity. Reported values are mean and standard deviation for three iterations.

When performing the shared-kernel pruning, the input layer is affected by the removal of parameters. As a result, some channels from the input RGB images can be dropped. Because only channels are affected, there are fewer options that might occur, i.e. there can be 0, 1 or 2 channels that are removed. This number depends on the number of shared-kernels that are removed from the first convolutional layer when performing the

pruning operation. In Figure 5.5, we report the different scenarios and the remaining channels corresponding. We observe that the first channel to be removed is consistently the blue one, the next being the red one. This seems to suggest that the green channel holds the most information in an image, which correlates with how the human visual system works, i.e. more weight is given to green colors, then red and finally blue when mixing colors for grayscale conversion [75].

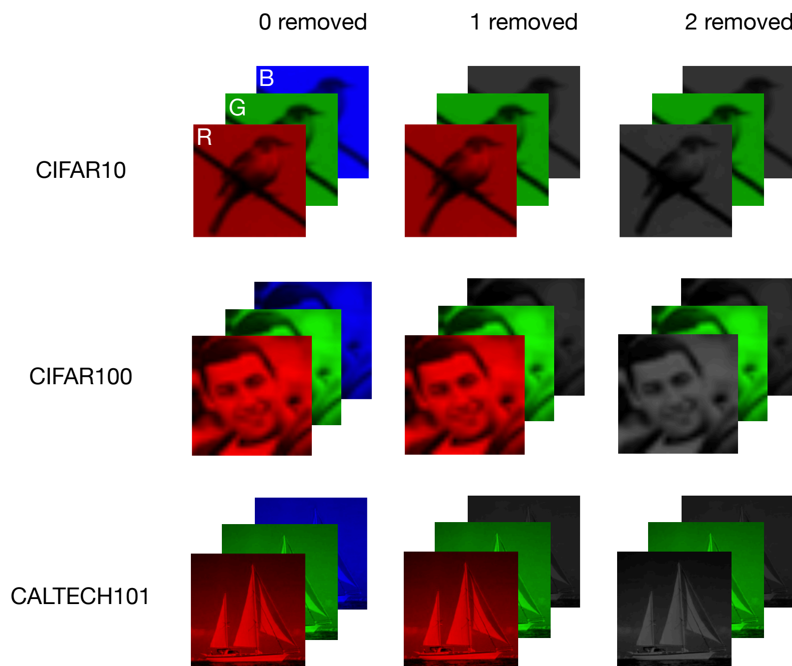


Figure 5.5. Remaining channels after performing shared-kernel pruning. The blue channel is the first to be removed, and the red one is the second. Removed channels are greyed out.

As observed for MLPs, results presented in Figure 5.5 suggest that the entire image is not necessary when performing the training of a neural network. Indeed, additional data storage gains can be obtained by only storing the relevant channels for our sparsity level, thus reducing storage costs by 1/3 or 2/3, depending if we keep 1 or 2 channels. This also suggests that using other color spaces such as YUV that hold a lot of information in the luminance channel might be considered.

We also report the impact of both granularities on the remaining amount of parameters and Floating Point Operations (FLOPs) in Table 5.5. As can be observed, the shared-kernel granularity usually leads to models that contain a smaller amount of parameters. On the other hand, the impact on the number of FLOPs seems to be more dataset and

sparsity-level dependent, with shared-kernel performing better on CALTECH-101 and for high levels of sparsity. These differences mainly come from the global pruning setting, which removes weights at different parts of the network, thus impacting more the parameter count if a filter is removed in late layers as they contain more channels, and impacting more the number of operations when a filter is removed in early layers, as they operate on larger resolution images.

		Scratch		**Fine-tune**	
		Filter	Shared-Kernel	Filter	Shared-Kernel
CIFAR-10					
Sparsity	30%	4.57/1.47	**3.54/1.44**	8.42/**0.98**	**3.76**/1.05
	50%	**2.53/0.98**	2.64/1.16	4.27/**0.53**	**2.54**/0.83
	70%	0.97/**0.44**	**0.59/0.44**	1.88/**0.25**	**0.72**/0.43
	90%	**0.02/0.02**	**0.02/0.02**	**0.10/0.10**	0.14/0.11
CIFAR-100					
Sparsity	30%	5.00/1.36	**3.43/1.27**	9.46/**0.73**	**7.90**/0.81
	50%	2.50/**0.83**	**2.33**/0.97	6.47/**0.42**	**2.24**/0.49
	70%	0.98/**0.30**	**0.42**/0.35	4.47/0.32	**0.63/0.30**
	90%	**0.01/0.02**	**0.01/0.02**	0.54/0.12	**0.01/0.02**
CALTECH-101					
Sparsity	30%	4.14/1.47	**3.53/1.41**	5.56/1.39	**3.10/1.22**
	50%	2.23/1.30	**1.77/1.10**	2.49/1.14	**1.36/0.94**
	70%	**0.41**/0.76	0.43/**0.56**	0.74/0.72	**0.57/0.64**
	90%	0.02/0.12	**0.01/0.03**	0.13/0.35	**0.05/0.17**

Table 5.5. Comparison of remaining #Params(1e6)/FLOPs(1e9) for ResNet-18 trained on different datasets for four sparsity levels.

5.2.3 Discussion & Conclusion

For this first research axis, we provide an element of answer to the question of "How to prune?", we propose another granularity according to which Convolutional Neural Networks can be pruned. This granularity, which we call *shared-kernel*, removes common kernels in all of the filters in a layer. This granularity benefits from the same property as removing filters as it allows to physically remove sparse structures from the network, and to re-arrange the architecture to have a smaller and dense one, allowing

for practical speed-up without the need for dedicated systems.

The experiments conducted on three datasets: CIFAR-10, CIFAR-100 and CALTECH-101, demonstrate that shared-kernel pruning is an alternative to the popular filter pruning as it allows to provide better accuracy, especially for high levels of sparsity, but also a more significant reduction in parameters and, in some cases, operations.

Moreover, because it removes incoming data, shared-kernel pruning allows acquiring information about the data itself. Indeed, as it was also demonstrated for its MLP-equivalent, such granularity allows to remove part of the data that are considered less valuable. As a result, this pruning technique not only allows to reduce the storage and compute costs for the neural network, but also for the dataset in itself.

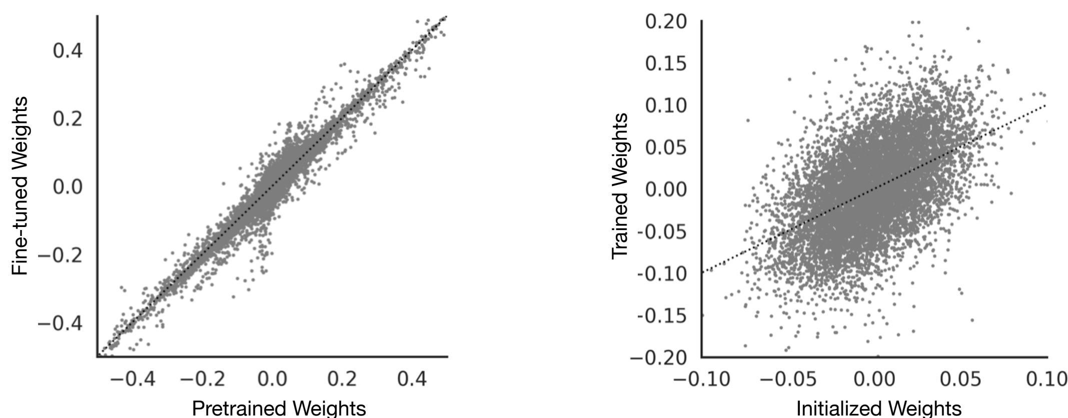
Such a pruning approach could further be extended to other and types of architectures such as Transformers or Recurrent Neural Network (RNN) architectures, providing other types of insights about the input data.

5.3 Where to prune?

As described in Section 3.3.2, there exist two primary contexts of selection for parameter pruning: local and global. While local pruning leads to layers of equal sparsity, performing global pruning yields a network of layers with different sparsity levels. However, when performing global pruning, the assumption that we can directly compare weights from early layers to those of the last layers is missing a key factor: networks are built in a hierarchical manner, and layers are not extracting the same kind of information depending on their depth in the network. Indeed, weights of early layers are only able to extract low-level features, likely to be common to most image classification tasks. On the other hand, weights belonging to deeper layers are focusing on higher-level features, which are more dependent on the classification task that is performed, and thus less likely to be transferable to another classification task.

Also, when initializing the weights in a neural network, two choices are usually available: (1) random initialization, which ensures that the network has no prior knowledge of a task. In this case, when the network is trained on a new task, it is said to be trained **from-scratch**; (2) pre-trained initialization, which is obtained after training the network on a generic database that usually contains a large number of data points and classes, such as ImageNet [78]. The goal is to have a network that already performs on a broad set of task before training it on a target one. In this case, when the network is trained, it is said to be **fine-tuned**.

When comparing the evolution of the weight values in the model when using both of those training methods, we are confronted with a striking difference. Indeed, in the fine-tuning regime, the weight values after training are almost entirely determined by their initialization values. While also present in the from-scratch training regime, this phenomenon is more moderate. The evolution of weight values for both training regimes is represented in Figure 5.6.



(a) Plot of the weight distribution after training, i.e. fine-tuning, against their distribution at initialization, i.e. pre-trained values.

(b) Plot of the weight distribution after training against their distribution at initialization, i.e. randomly initialized.

Figure 5.6. Difference of weight distributions when the network is fine-tuned compared to when it is trained from-scratch for the first convolutional layer of ResNet-18 trained on CIFAR-10.

Usually, the pre-training task involves many classes, e.g. 1000 in the case of ImageNet, and the fine-tuning tasks that we are interested in often involve fewer. Furthermore, because weight values are kept close during training, there are potentially many parameters that will no longer be relevant to the fine-tuning task. This means that, when performing pruning in such a network, more parameters should be removed in deeper layers of the network rather than in early ones.

In this contribution, our goal is to investigate how this initialization difference impacts the pruning of a neural network and, more particularly, how it affects the locality of where the pruning happens in the network. To do so, we adopt an empirical approach, studying the sensitivity to pruning of each layer, and performing pruning accordingly. It allows to understand more about which layer contains more important parameters, and to provide an alternative to the local and global pruning methods.

5.3.1 Methodology

To better apprehend how important the parameters of a network depend on the layer they belong to, we propose to perform a *sensitivity analysis* of the network. This provides a heuristic to discover which parameters are the least useful in the network. It consists of a 4-steps process:

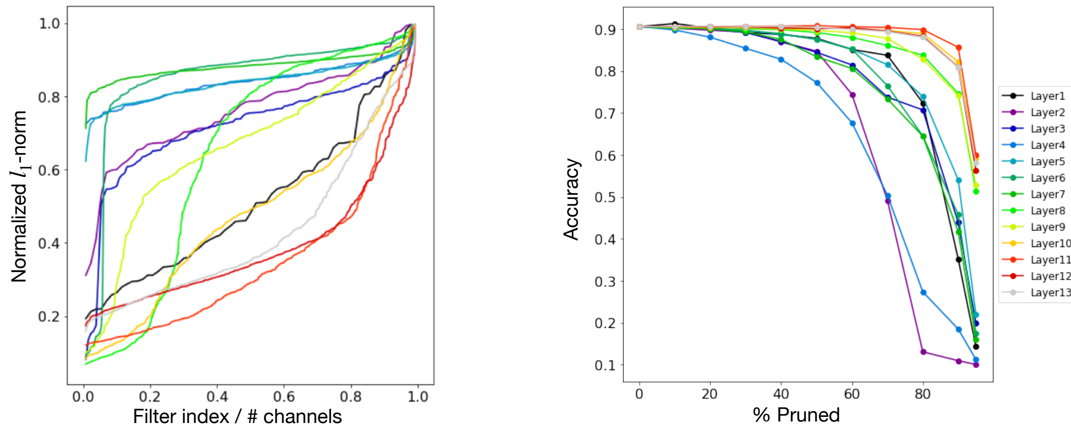
1. Select a layer l from a model.
2. Sort the parameters in layer l according to their l_1-norm.
3. Remove portions of parameters having the lowest l_1-norm, reporting the accuracy of the remaining model each time.
4. Restore the initial model and perform the previous operations for each layer.

This can be done with `FasterAI` by iterating over each layer, removing a fraction of filters and reporting the accuracy, as represented in Pseudo-Code 5.3.

```
sparsifier = Sparsifier(model, granularity, context, criteria)

for l in model.layers():
    for sp in sparsities:
            pruned_model = sparsifier.prune_layer(l, sp)
            _, acc = pruned_model.validate()
```

Pseudo-Code 5.3. Performing sensitivity analysis with `FasterAI`.

After performing the sensitivity analysis, we can plot the evolution of accuracy when parameters are removed from each layer. Layers whose accuracy remains stable for higher sparsity levels are said to be more robust, or least sensitive, to pruning. Those are the layers that will be first targeted when performing pruning. As an example, Figure 5.7 provides an example of a sensitivity analysis performed on VGG-16 trained on MNIST. In Figure 5.7a are represented the sorted l_1-norms of filters in each layer of the network. The Figure 5.7b demonstrates the evolution of accuracy when performing the sensitivity analysis. In that case, the layer that is the least sensitive to pruning is `Layer11`. As can be observed, the l_1-norms of filters are closely related to their sensitivity. Indeed, layers that are the least sensitive to pruning are the ones whose majority of filters have a low l_1-norm, while layers with more high-normed filters are more sensitive to pruning. The sensitivity analysis can thus be performed in an equivalent manner by studying the l_1-norm of filters contained in a layer [50].



(a) Filters ranked by ascending l_1-norm for a VGG16 network trained on MNIST.

(b) Sensitivity of each convolutional layer pruned individually.

Figure 5.7. Visualization of the sensitivity to pruning of a VGG-16 trained on MNIST. Layers with most low-norm filters are those that are less sensitive to pruning.

Performing filter pruning following an iterative schedule usually consists of a three-step method: (1) train the network to convergence, (2) prune a portion of the convolution filters, according to a chosen criterion and, (3) fine-tune the model to recover from the lost performance. Steps (2) and (3) are then repeated, alternating pruning and fine-tuning until the desired sparsity is reached. We propose to add the sensitivity analysis prior to the pruning step, to provide information about the layers from which parameters should be removed. The whole proposed process is represented in Figure 5.8.

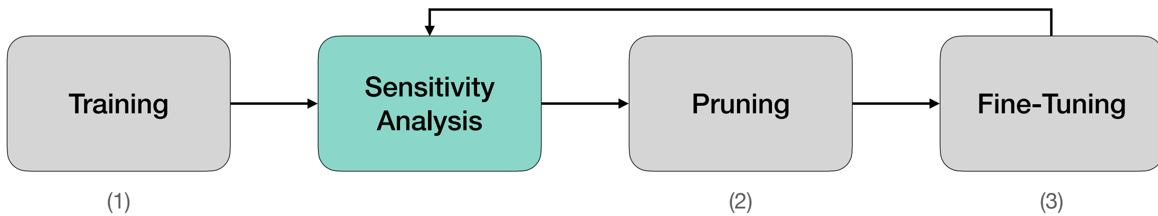


Figure 5.8. The proposed pruning pipeline. We introduce a fourth step in the common iterative pruning process, aiming to evaluate the sensivity of each layer to pruning. By then performing pruning in the least sensitive layer, we ensure to reduce the number of parameter contained in the network without affecting the performance too much.

5.3.2 Experiments

In this section, we evaluate the effects of selecting the layers to prune based on their sensitivity as presented in section 5.3.1 .

Pruning Method. The different parameters used for performing pruning are summarized in Table 5.6, where they are expressed using `FasterAI` terminology. Basically, we operate at the granularity of entire convolution filters, whose importance is evaluated using their l_1-norm. The context of pruning is neither local or global, but rather identified empirically by first performing the sensitivity analysis and removing filters from the least sensitive layer. The pruning is performed iteratively, layer by layer and by alternating phases of pruning and fine-tuning. During each pruning phase, we remove a subset of filters from the current layer. As a result, the sparsity level is not imposed beforehand, but discovered in the process.

Granularity	Criteria	Context	Schedule
`filter`	`large_final`	-	`iterative`

Table 5.6. Pruning parameters used in `FasterAI`.

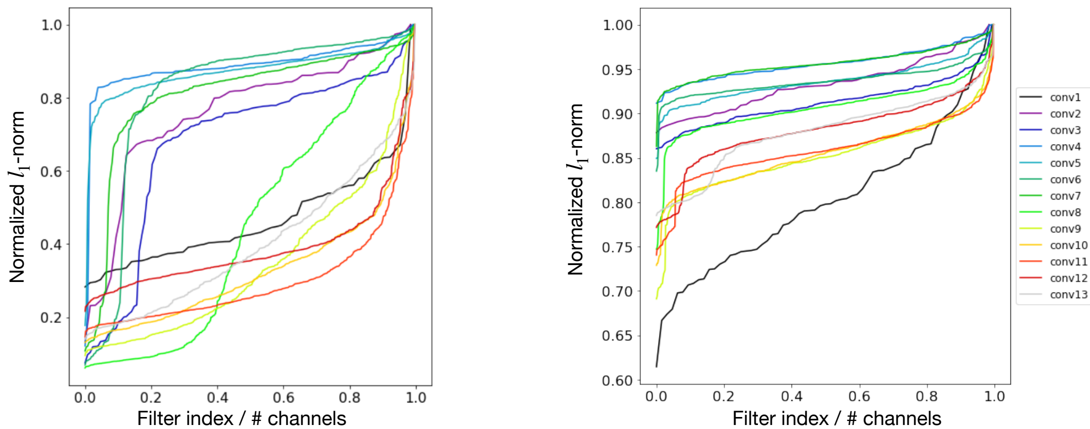
Datasets. We have carried out our experiments on four different datasets, chosen to be various in resolution and content. In particular, we use CIFAR-10 [74], CIFAR-100 [74], SVHN [79], and CALTECH-101 corpus [68].

Network architectures. In order to validate our results, we have adopted two well-known network architectures. The first one is VGG-16 [31]. Here, we have replaced the original fully-connected layers with a Global Average Pooling layer and two narrow fully-connected layers. In this way, most parameters are contained in the convolutional layers. The network thus consists of 13 convolutional layers and two fully-connected layers. The second network retained is MobileNetV1 [30], specifically designed to achieve efficiency both in parameter number and in computation complexity. MobileNet uses a factorized form of convolutions called Depthwise Separable Convolutions. The MobileNet architecture used in our experiments thus consists of one standard convolution layer acting on the input image, 13 depthwise separable convolutions, and finally a global average pooling and two fully connected layers.

Training Procedure. Two types of initialization are evaluated: (1) Networks initialized from pre-trained weights and fine-tuned on the selected datasets, which we refer to as Fine-Tuned or FT networks in the experiments; (2) Networks randomly

initialized and directly trained on the datasets, that we refer to as From-Scratch or FS networks in the experiments. Images of our dataset are first resized to 224×224 and augmented using horizontal flips, rotations, image warping and cropping. Our models are first trained until convergence, with an initial learning rate of $1e-3$ and a step decay scheduling, gradually reducing the learning rate every 40 epochs. They are then tested on the validation set to get the baseline accuracy. After each pruning phase, a retraining is performed for 5 epochs, with the lowest learning rate reached during baseline training. We also monitor the accuracy on the validation set, and proceed to another pruning phase if it has not dropped by more than 1% from the baseline accuracy.

VGG16. The experiment is first conducted on the VGG-16 network. As shown in Figure 5.9a, the result of the sensitivity analysis before starting the pruning process indicates that most of the least important filters are contained in the later layers, suggesting that those layers will be less sensitive to pruning than others. On the other hand, Figure 5.9b illustrates that the network trained from-scratch, FS-Network, has a more even distribution of filter importance, suggesting that the network is more sensitive to pruning than FT-Network, specifically in the later layers.

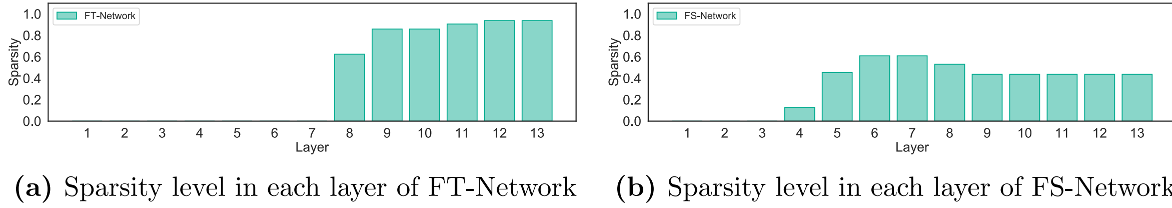


(a) Filters ranked by ascending l_1-norm for FT-Network.

(b) Filters ranked by ascending l_1-norm for FS-Network.

Figure 5.9. Visualization of the importance of filters of VGG-16 trained on CIFAR-10. The deep layers of FT-Network possesses an important fraction of low-norms filters, whereas FS-Network has filters of balanced norms.

The results of performing our pruning method on both networks are summarized in Figure 5.10. As can be observed, FT-Network undergoes extensive pruning in deeper layers, while pruning is more spread across all layers for FS-Network.



(a) Sparsity level in each layer of FT-Network **(b)** Sparsity level in each layer of FS-Network

Figure 5.10. Repartition of sparsities in the layers of VGG-16 trained on CIFAR-10.

Table 5.7 summarizes the results on all the tested datasets, and shows the resulting number of parameters, their corresponding storage size, as well as the number of FLOPs needed for an input image to traverse the whole network at the testing phase.

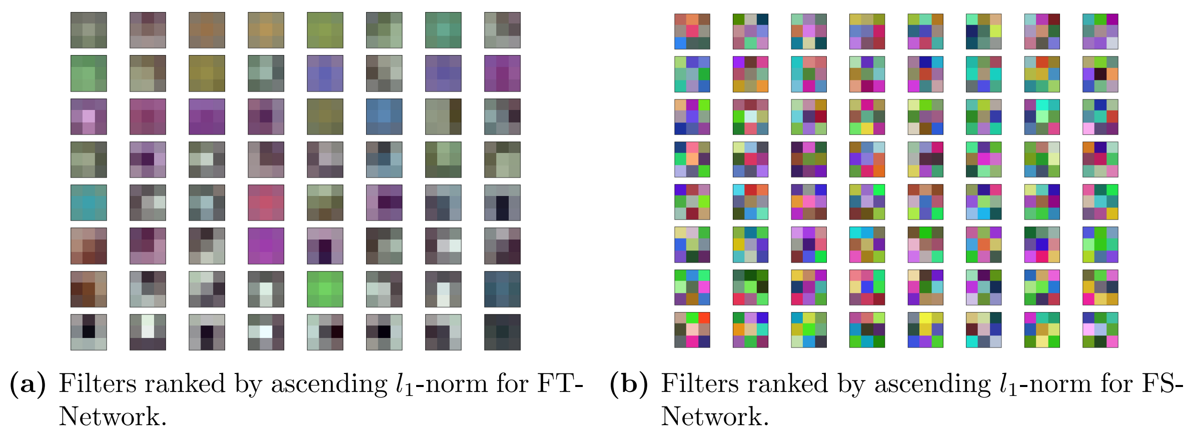
Dataset	Network	Params (M)	FLOPs (M)
CIFAR10	Baseline	14.98	627.48
	FT-pruned	**3.43**	421.20
	FS-pruned	8.26	**397.85**
CIFAR100	Baseline	15.03	627.57
	FT-pruned	**8.26**	503.47
	FS-pruned	8.69	**444.56**
SVHN	Baseline	14.98	627.48
	FT-pruned	**3.15**	414.12
	FS-pruned	4.08	**311.04**
CALTECH-101	Baseline	15.03	30,720.99
	FT-pruned	**8.44**	**25,402.11**
	FS-pruned	13.59	30,171.17

Table 5.7. Results of the pruning on VGG-16 for the four studied datasets. FT-Network leads to smaller networks, while FS-Network leads to faster ones, due to the location of removed filters in the network.

We can observe that for all the tested datasets, pruning is more effective in terms of parameters removed for FT-Network than for FS-Network. However, fewer parameters do not necessarily lead to a greater reduction in FLOPs. This phenomenon can be explained by the fact that, while most of the parameters are contained in the later layers, most operations are performed in the first ones, where the resolutions of the activation maps are higher. For FT-Network, most of the pruning is performed in the later layers. In contrast, for FS-Network the pruning is more distributed throughout

the network. Thus, FT-Network has the fewest parameters but FS-Network often has the fewest FLOPs.

A closer look at the filters of the first convolutional layer, represented in Figure 5.11, exhibits the difference between learned filters. The reason of this difference is because the pre-training of FT-Network helped to find valuable filters on the ImageNet database. As FS-Network only had access to fewer and smaller images, it could not learn the same kind of filters by itself and had to distribute the feature extraction across the network, reason why the later layers are more sensitive to pruning.

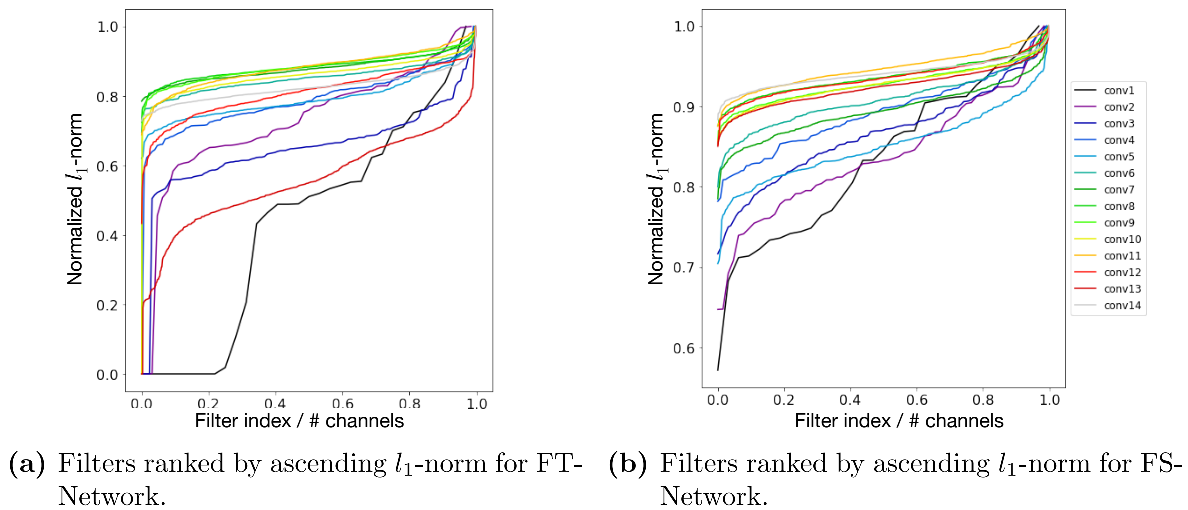


(a) Filters ranked by ascending l_1-norm for FT-Network. **(b)** Filters ranked by ascending l_1-norm for FS-Network.

Figure 5.11. Visualization of the 64 filters in the first convolutional layer of VGG-16 trained on CIFAR-10. FT-Network filters have more structure than those of FS-Network.

MobileNet. The Depthwise Separable Convolutions operations that are used in MobileNet are composed of two operations.: (1) a Depthwise Convolution, filtering each input map independently; (2) a Pointwise Convolution, i.e. a regular convolution with a kernel dimension of 1×1. As most parameters are contained in the second operation, we propose to operate the pruning only on Pointwise Convolutions.

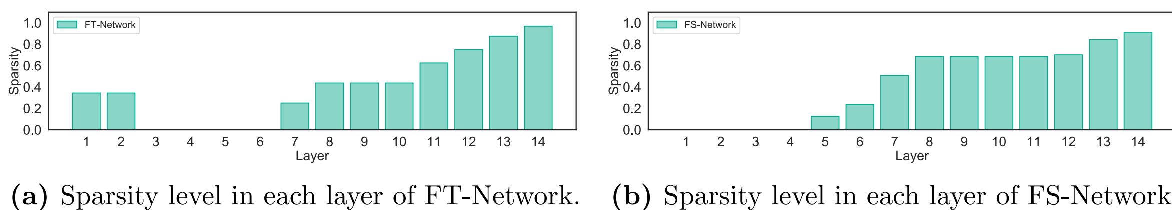
The first sensitivity analysis, performed before starting the pruning process, reveals a smaller difference between the sensitivity of FT-Network, represented in Figure 5.12a and FS-Network, represented in Figure 5.12b, than in the case of VGG16. However, we can still observe that the highest norm filters of FS-Network are still in the later layers, which is not necessarily the case for FT-Network. This again suggests that FT-Network can be pruned further in the later layers. Also, the FT-Network surprisingly possesses filters with a l_1-norm of 0 in its first convolutional layer.



(a) Filters ranked by ascending l_1-norm for FT-Network. **(b)** Filters ranked by ascending l_1-norm for FS-Network.

Figure 5.12. Visualization of the importance of filters of MobileNet trained on CIFAR-10. The deep layers of FT-Network possesses an important fraction of low-norms filters, whereas FS-Network has filters of balanced norms.

The results of the pruning process for both types of initializations is reported in Figure 5.13. At the exception of the first layer, for which we observed a fraction of low-normed filters in 5.12b and that are consequently removed,most pruning of the FT-Network is performed in late layers, while it is slightly more spread across the layers for FS-Network.



(a) Sparsity level in each layer of FT-Network. **(b)** Sparsity level in each layer of FS-Network.

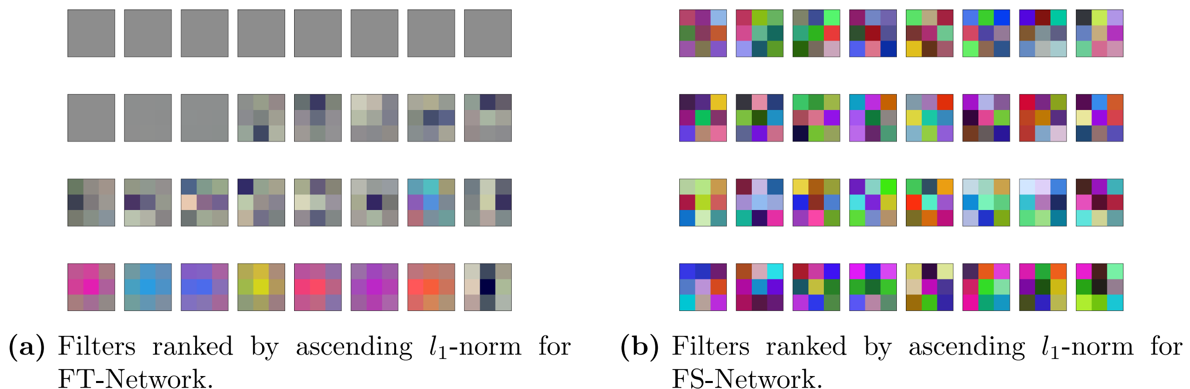
Figure 5.13. Repartition of sparsities in the layers of MobileNet trained on CIFAR-10

The results of the pruning of MobileNet on the different datasets are summarized in Table 5.8. As it was the case for VGG16, the pruning of FT-Network also leads to a smaller number of parameters than for the FS-Network, even if the difference is smaller in this case. However, due to the location of the removal, FS-Network is reaching a comparable or lower FLOPs count.

Dataset	Network	Params (M)	FLOPs (M)
CIFAR10	Baseline	3.76	24.23
	FT-pruned	1.05	13.83
	FS-pruned	**0.98**	**12.26**
CIFAR100	Baseline	3.80	24.32
	FT-pruned	**2.54**	**21.42**
	FS-pruned	3.12	21.89
SVHN	Baseline	3.76	24.23
	FT-pruned	**1.50**	17.66
	FS-pruned	1.75	**16.57**
CALTECH-101	Baseline	3.81	1136.59
	FT-pruned	**3.08**	**1078.30**
	FS-pruned	3.43	1105.83

Table 5.8. Results of the pruning on MobileNet for the four studied datasets. FT-Network leads to smaller network while not necessarily the fastest ones, due to the location of removed filters in the network.

Again, looking at the filters of the first convolutional layer exhibits the difference of learned filters between FT-Network, represented in Figure 5.14a, and the FS-Network, represented in Figure 5.14b. The filters of the FT-Network show some structure where those from FS-Network do not appear to. Moreover, as it was suggested in Figure 5.12a, we can observe the low-norm filters in the first convolutional layer.



(a) Filters ranked by ascending l_1-norm for FT-Network.

(b) Filters ranked by ascending l_1-norm for FS-Network.

Figure 5.14. Visualization of the 32 filters in the first convolutional layer of MobileNet trained on CIFAR-10. Filters of FT-Network exhibit more structure than the filters of FS-Network.

5.3.3 Discussion & Conclusion

For this second research axis, concerned about the question "Where to prune?", we provide elements of answers for the two most common methods of neural network initialization: random and pre-trained. In particular, we have compared the sensitivity to pruning of both initialization schemes and discovered that it significantly impacts the location of parameters to remove.

The experiments conducted on three dataset: CIFAR-10, CIFAR-100, SVHN and CALTECH-101, demonstrate that fine-tuned networks tend to possess fewer useful filters in later layers, which leads to pruned networks that contain fewer parameters. However, the number of parameters does not necessarily correlate to a lower amount of computations, as later layers process images of low resolution. For that reason, networks trained from-scratch, where the pruning can be performed more evenly across the network, usually require fewer computations.

According to the targeted usage of the compressed network, either to require smaller storage space or fewer computations, a specific initialization scheme and pruning technique can thus be used. This can help to improve existing CNNs architectures, but also to find new training strategies, better suited to target lower storage or compute budgets.

5.4 What to prune?

Our third pruning contribution concerns the pruning criteria and aims to provide an answer to the question: "What to prune?". In this regard, we propose a novel pruning method, that can be applied prior to any criteria, and allowing to reduce the redundancy of weights remaining in a pruned network.

The most commonly used pruning criteria is magnitude pruning, removing weights having the lowest l_1-norm in the network. Indeed, weights having the smallest magnitude participate less in the activations of the model and can be thus removed without impacting too much the performance. Despite its apparent simplicity, magnitude pruning has been shown to be highly generalizable to different datasets and architectures [80]. More recently, the movement criteria, particularly efficient in the fine-tuning regime, has been proposed [51]. This criteria was designed to tackle the problem that we observed in Section 5.2, i.e. that the weight values after fine-tuning are close to their initialization values. While efficient, the magnitude and movement criteria only base their decision on the value of the weight and do not explicitly seek to reduce the redundancy. Indeed, as long as they are above the pruning threshold, redundant weights

can be kept by those criteria. Moreover, weights that might be unique and have an important impact on the final decision might be removed if they happen to be below the decision threshold.

Therefore, we propose injecting a filter clustering method before applying common pruning criteria to each group independently. By doing so, we ensure that, not only do we reduce the redundancy in the filters, but also that we do not remove the rare ones. This clustering method is inspired by a neural network interpretation method called Activation Maximization (AM) [81]. The idea behind AM is to generate an input image that maximizes a filter output activation. This image is found by performing gradient ascent in the input space, updating each pixel value until the response of the filter is maximal, as represented in Figure 5.15. The images created will thus correspond to images that excite the most a given convolutional filter, i.e the feature that that particular filter is looking for.

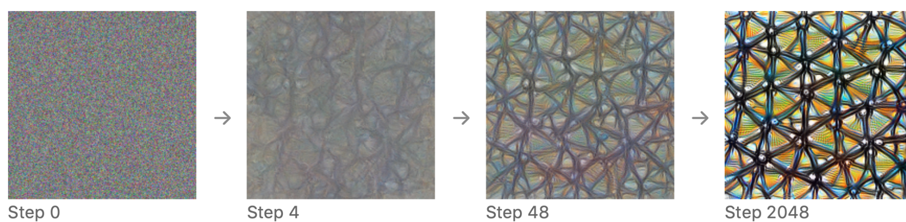


Figure 5.15. Illustration of the Activation Maximization technique. It starts from random noise and gradually optimizes the values of the pixels to activate a particular neuron. Image taken from [82].

5.4.1 Methodology

Starting from the three-step iterative pruning schedule, we propose an additional step, prior to pruning, as represented in Figure 5.16. Indeed, before selecting the weights to remove according to a chosen criteria, we first cluster filters exhibiting similar behaviour and to perform pruning in each cluster separately, and consequently only on redundant filters. By doing so, pruning will only retain independent filters while also retaining filters that have uncommon behaviors, thus maximizing the variety of remaining filters.

For each convolutional filter in a layer of the network, we can synthesize its corresponding *signature* image based on Activation Maximization. The goal is then to perform K-Means clustering of those images, effectively grouping similar images together while also keeping unique ones in their dedicated group. To effectively reduce dimensionality and facilitate the task of K-Means clustering, we first feed our images to the convolutional part of an AlexNet model [32] pre-trained on ImageNet [78], encoding those

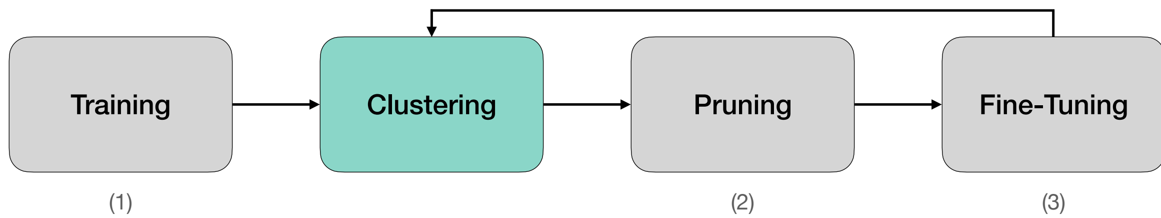


Figure 5.16. The proposed pruning pipeline. We introduce a fourth step in the common iterative pruning process, aiming to cluster convolution filters by similar functionality. By then performing pruning in each cluster, we ensure that we remove redundant filters, while preserving the rare ones.

into a feature vector, which will serve as input data to the clustering algorithm. The proposed clustering technique is represented in Figure 5.17.

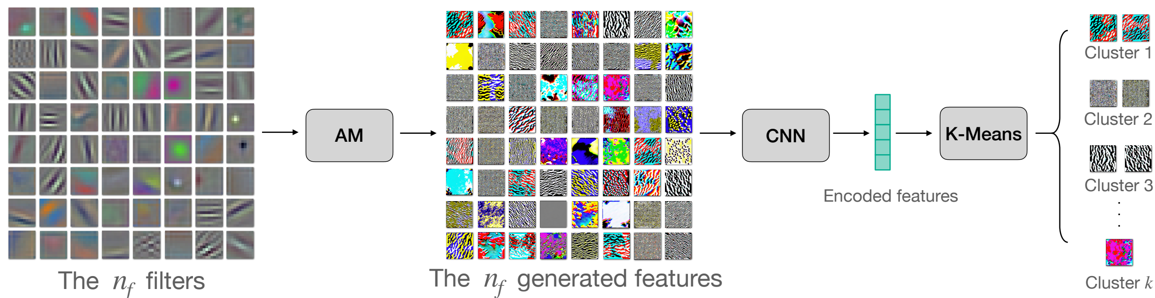


Figure 5.17. Representation of the clustering process. We first generate the feature image corresponding to each filter with the Activation Maximization technique. Those synthesized images are then encoded to a lower dimension by a pre-trained CNN, and clustered with K-Means, allowing to group filters sensitive to similar features together.

Once each feature image has been clustered, we can then apply the pruning process, selecting remaining filters according to a chosen criteria but, this time not by comparing all the filters in the layer, but by comparing filters whose feature images are located in the same group. By doing so in each group, we will only preserve each feature's best representative filter. The number of clusters k is thus chosen as a compression parameter, depending on the desired sparsity. Setting a high number of clusters k creates more groups and thus removes fewer filters.

When comparing common criteria before and after the addition of our clustering method, we observe that a greater variety of filters are retained. As an example, Figure 5.18 represents features extracted from the filters of the first layer of a simple CNN, AlexNet [32], using the Activation Maximization technique. Three clusters of similar features have been highlighted in color, and the corresponding remaining features are shown for each pruning technique, with the removed one being greyed out. We can observe that, by clustering similar features, we ensure that: (1) redundant features are removed and (2) rare features are being kept, which is not the case when using magnitude and movement pruning alone, where some features belonging to the same cluster are still present.

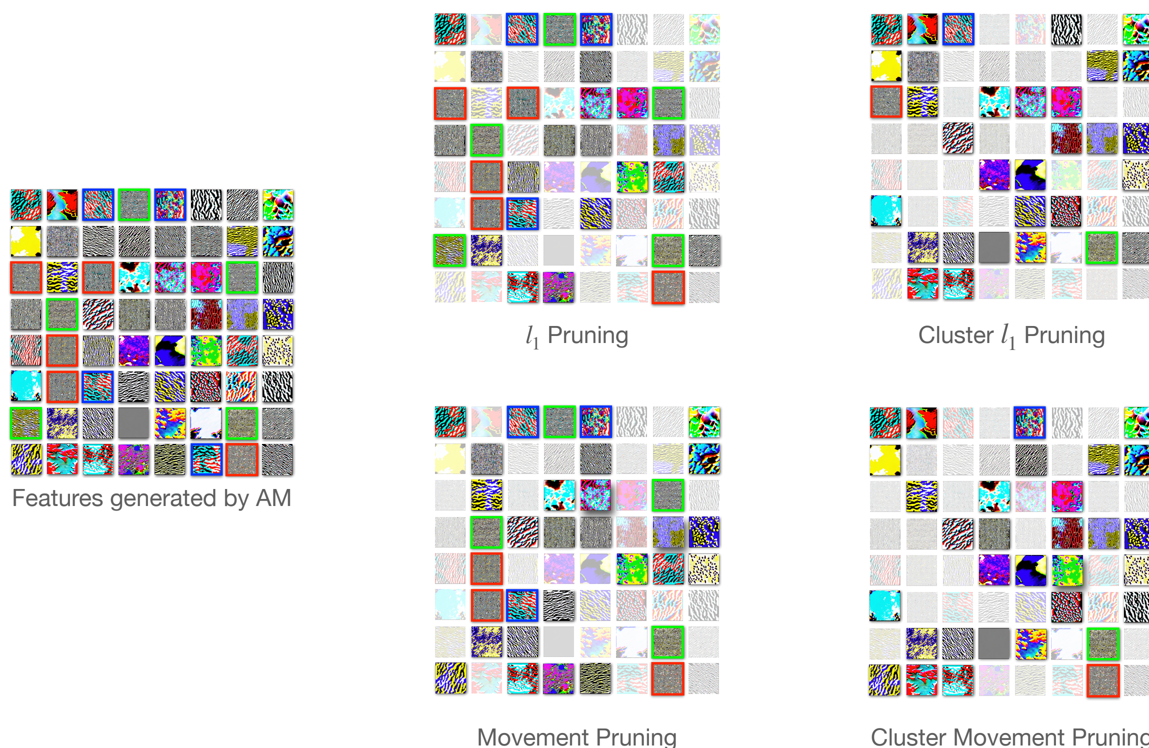


Figure 5.18. Comparison of the remaining features after applying different pruning techniques until a sparsity of 50% in the first layer of AlexNet. Three dominant clusters are highlighted in color. Features removed are greyed out.

5.4.2 Experiments

Comparison to Common Criteria

In this section, we evaluate the effects of adding the proposed extra clustering step in the iterative pruning pipeline and apply the pruning criteria in each group separately.

Pruning Method. The different parameters used for performing pruning are summarized in Table 5.9, where they are expressed using `FasterAI` terminology. We use the proposed 4-step schedule presented in Figure 5.16. To do so, we iterate over each layer and remove a specific number of filters in order to reach the desired sparsity. Before proceeding to the pruning step, our clustering method takes care of dividing the filters of a target layer into k groups, with $k = s \times n_f$, s being the desired sparsity in percent and n_f the number of filters in that layer. The desired pruning criteria is then separately applied to each filter group, retaining a single filter from each cluster. We evaluate the benefits of our extra clustering step according to two pruning criteria: (1) l_1-norm of the filters, i.e. remove the filters that possess the lowest norm, and (2) movement pruning, i.e. only keep the filters whose magnitude has increased the most during training.

Granularity	Criteria	Context	Schedule
`weight`	`large_final` **vs** `movement`	`local`	`iterative`

Table 5.9. Pruning parameters used in `FasterAI`.

Datasets and Architectures. For our experiments, the datasets have been chosen to be varied in terms of image resolution and number of classes. In particular, we evaluate our methods on the three following datasets: (1) CIFAR-10 [74], (2) CIFAR-100 [74], and (3) Caltech-101 [68]. Those datasets are then tested on two types of popular convolutional network architectures: VGG-16 [31] and ResNet-18 [67]. In particular, we use a modified version of VGG-16 which consists of 13 convolutional layers and 2 fully-connected layers, with each convolutional layer being followed by a batch normalization layer [83].

Training Procedure. The networks used for our experiments are initialized from pre-trained weights, i.e. networks were previously trained on ImageNet and we reuse their weights. Images of our dataset are first resized to 224×224 and are augmented by using horizontal flips, rotations, image warping and random cropping. We train each model for 15 epochs, at a learning rate of $1e-3$, and using the 1cycle learning rate

method [76], where the training starts with a learning rate warm-up until a nominal value, then gradually decays until the end of the training. After each pruning phase, we fine-tune our model for 3 epochs, with a learning rate of $3e-4$, to allow the network to recover from the loss of its parameters.

Results. We report the results of our experiments for VGG-16 in Table 5.10, and for ResNet-18 in Table 5.11. As can be observed, for almost all sparsity levels, datasets and criteria tested, adding the proposed clustering step is beneficial to the pruning process. Indeed, for the same sparsity level, accuracy increases up to 5% can be observed, which also means that the same networks could be pruned to a higher sparsity level without witnessing performance degradation.

		l_1	Cluster l_1	Movement	Cluster Movement
CIFAR-10					
Sparsity	60%	90.89 ± 0.17	**92.39 ± 0.07**	91.55 ± 0.18	**92.45 ± 0.26**
	70%	89.47 ± 0.09	**90.91 ± 0.13**	90.35 ± 0.12	**90.89 ± 0.10**
	80%	84.95 ± 0.08	**87.04 ± 0.23**	86.50 ± 0.19	**87.48 ± 0.10**
CIFAR-100					
Sparsity	60%	54.94 ± 0.21	**58.72 ± 0.30**	54.81 ± 0.34	**57.29 ± 0.71**
	70%	47.56 ± 0.94	**51.67 ± 0.61**	47.59 ± 0.29	**51.25 ± 0.41**
	80%	35.67 ± 0.90	**39.30 ± 0.65**	36.82 ± 1.42	**42.30 ± 0.94**
Caltech-101					
Sparsity	60%	86.63 ± 0.39	**87.44 ± 0.28**	86.94 ± 0.28	**87.40 ± 0.51**
	70%	84.87 ± 0.41	**86.01 ± 0.79**	82.89 ± 0.19	**84.43 ± 0.25**
	80%	**79.18 ± 0.63**	79.03 ± 0.72	76.00 ± 0.59	**78.75 ± 0.63**

Table 5.10. Results of applying different pruning criteria on VGG-16. The benefit of applying our clustering method before selecting the filters to remove translates to a higher accuracy for most sparsity levels and datasets. Values in bold are the best when comparing a criteria with and without the clustering process. Accuracies and standard deviation over 3 runs are reported.

		l_1	Cluster l_1	Movement	Cluster Movement
CIFAR-10					
Sparsity	60%	93.32 ± 0.11	**93.76 ± 0.18**	92.73 ± 0.16	**93.57 ± 0.10**
	70%	92.17 ± 0.11	**92.20 ± 0.11**	90.79 ± 0.14	**92.35 ± 0.03**
	80%	89.58 ± 0.17	**90.13 ± 0.09**	87.04 ± 0.24	**89.53 ± 0.23**
CIFAR-100					
Sparsity	60%	71.65 ± 0.22	**72.61 ± 0.41**	70.95 ± 0.13	**72.27 ± 0.37**
	70%	67.18 ± 0.17	**68.46 ± 0.25**	66.19 ± 0.15	**68.44 ± 0.21**
	80%	59.14 ± 0.13	**60.32 ± 0.30**	58.50 ± 0.41	**59.86 ± 0.23**
Caltech-101					
Sparsity	60%	92.63 ± 0.05	**93.00 ± 0.16**	90.77 ± 0.18	**91.81 ± 0.26**
	70%	88.75 ± 0.42	**89.48 ± 0.30**	85.32 ± 0.27	**87.75 ± 0.24**
	80%	79.89 ± 0.10	**80.31 ± 0.35**	75.05 ± 0.56	**76.29 ± 0.39**

Table 5.11. Results of applying different pruning criteria on ResNet-18. The benefit of applying our clustering method before selecting the filters to remove translates to a higher accuracy for most sparsity levels and datasets. Values in bold are the best when comparing a criteria with and without the clustering process. Accuracies and standard deviation over 3 runs are reported.

Application to Lottery Ticket Hypothesis

In addition to comparing the impact of our proposed clustering technique, we also would like to compare the quality of the remaining subnetworks, obtained after pruning. Such an analysis may be performed using the Lottery Ticket Hypothesis.

Finding Lottery Tickets. As described in Section 3.3.4, Lottery Tickets are found using an iterative process, repeatedly pruning a portion of remaining weights according to their l_1-norm, then resetting them to their initial value, i.e. the value before any training has occurred [56]. It was later generalized to higher complexity use-cases by resetting weights to a value from early training instead [57].

Comparison of Tickets. We compare the performance of discovered tickets using the same pruning criteria, datasets and training procedure as described in Section 5.4.2, and for the ResNet-18 architecture. As our experiment concerns large datasets and complex architecture, we propose to study the effect of our pruning technique on the Lottery Ticket Hypothesis with Rewinding. To uncover the tickets, we adopt the same methodology as presented in Section 5.4.1, but reinitializing the weights after each

pruning step to the value they had after the initial training step, *i.e.* to their value after Step (1) in Figure 5.16. The operation is performed for each criteria evaluated in the Section 5.4.2 and for sparsity levels ranging from 10% to 80%. After extracting the subnetwork, we train it for 15 epochs and then compare the versions obtained with and without the addition of our variety-enforcing clustering method.

Results. From this experiment, whose results are reported in Figures 5.21, we can observe that in most cases, the addition of the clustering method prior to the criteria selection helps to find a better performing ticket, thus validating the quality of the pruned network. While adding a clustering technique before applying the pruning criteria seems profitable in most cases, it benefits movement pruning the most. Indeed, increases up to 2% in accuracy may be observed in the case of l_1 pruning, and up to 5% in the case of movement pruning.

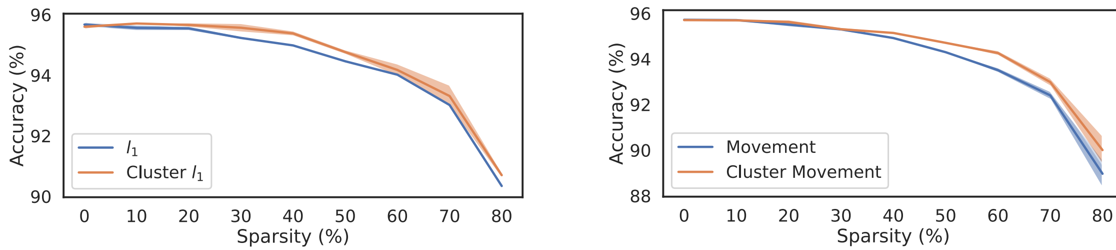


Figure 5.19. Results of the Lottery Ticket Hypothesis with Rewind test for different sparsities, performed with ResNet-18 on CIFAR-10.

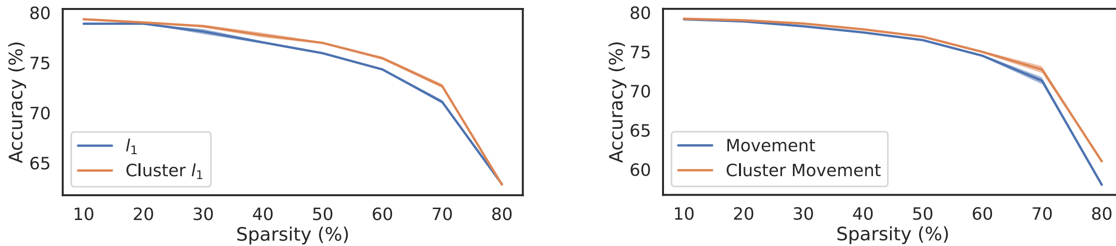


Figure 5.20. Results of the Lottery Ticket Hypothesis with Rewind test for different sparsities, performed with ResNet-18 on CIFAR-100.

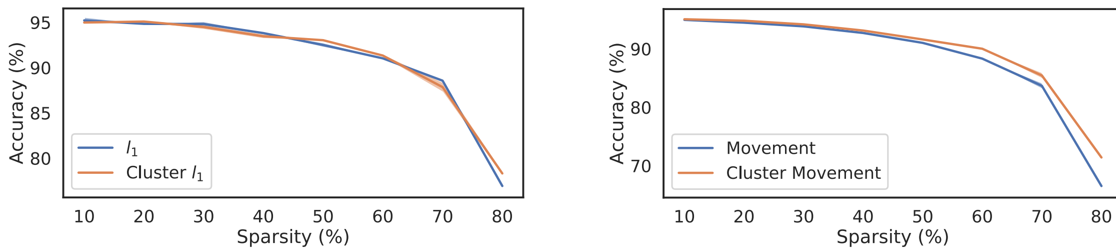


Figure 5.21. Results of the Lottery Ticket Hypothesis with Rewind test for different sparsities, performed with ResNet-18 on CALTECH-101.

5.4.3 Discussion & Conclusion

In this contribution, we propose a novel pruning method, introducing a clustering process before applying the pruning criteria. This clustering process, based on an interpretation technique called Activation Maximization groups filters sensitive to similar features in the input image. By then applying the pruning criteria to each feature group, we ensure that pruning is applied on redundant filters, and that rare filters, which may be alone in their group, are retained. Experiments have shown that our method leads to better results than classical methods on both VGG-16 and ResNet-18 architectures and for CIFAR-10, CIFAR-100 and CALTECH-101 datasets. Those results demonstrate that one should avoid pruning rare or unique filters and that keeping a wide filter variability is crucial to achieving both a higher pruning rate and a lower accuracy loss. Moreover, by performing Lottery Ticket Hypothesis with Rewinding tests, we have demonstrated that the subnetworks discovered after pruning were of better quality, as they were able to reach higher performance in the same training time.

5.5 When to prune?

Our fourth pruning contribution concerns the pruning schedule and aims to provide an answer to the question: "When to prune?". In this regard, we propose a novel pruning schedule, allowing to reach higher sparsity levels in a shorter period of time.

In particular, we propose a new pruning schedule, called One-Cycle Pruning (OCP), expanding on the state-of-the-art of gradual pruning techniques, i.e. pruning while the network is training. As of now, only one gradual pruning schedule has been proposed, the Automated Gradual Pruning (AGP) [59]. The latter performs most of the pruning early in the training, the motivation being that early training is the phase where the network has the most redundancy. However, recent studies have shown that the training of a neural network is conditioned very early by its regularization and parameters [72]. As a result, applying regularization too early in the network can irremediably impair the network performance, no matter how much training is performed afterwards [84]. However, to leverage the regularization effect of the sparsity imposed in the weights, it is still necessary to perform gentle pruning in early phases [85]. Finally, because pruning methods affect the optimization process, it is necessary to provide enough training iterations for the sparse network to fine-tune its remaining weights, allowing it to accommodate from the loss of capacity.

The purpose of One-Cycle Pruning is thus to propose a schedule that: (1) prunes gently in early training iterations; (2) provides a more extended fine-tuning phase to allow the network to reach an optimal performance; (3) performs the pruning and training jointly,

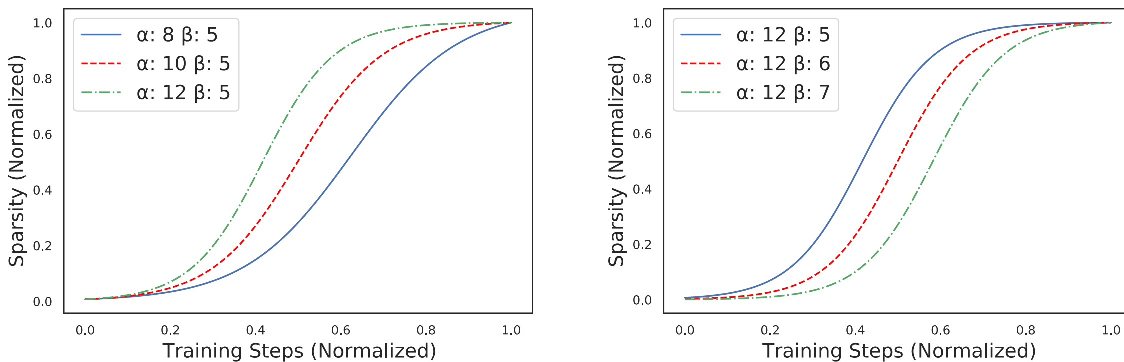
removing any sparsity level discontinuities. Adopting such a schedule thus provides the regularization benefits of sparse weights, but also performs the pruning process in a single training stage, drastically reducing the training budget.

5.5.1 Methodology

The proposed method for pruning consists of starting from a dense network and inducing sparsity during the whole training phase. The idea is thus to make the network jointly optimize for a given task, taking the pruning constraints into account. More precisely, we propose to induce sparsity in the network according to the following schedule:

$$s_t = s_i + (s_f - s_i) \cdot \frac{1 + e^{-\alpha + \beta}}{1 + e^{-\alpha t + \beta}} \tag{5.1}$$

with s_t, the level of sparsity at training step t, s_i and s_f respectively the initial and final level of sparsity, and α, β being two tuning parameters, modifying either the steepness of the scheduling, as represented in Figure 5.22a, or its horizontal offset, as represented in Figure 5.22b, to better suit the problem or architecture that is used.



(a) Evolution of the schedule for different α variations.

(b) Evolution of the schedule for different β variations.

Figure 5.22. Visualization of the variation of the scheduling for different α and β values. The α parameter affects the steepness of the curve, while β affects the horizontal offset.

The implementation of OCP in `FasterAI` is provided in Pseudo-Code 5.4.

```python
def sched_onecycle(start, end, pos, α,β):
    return start + (end-start)*(1+np.exp(-α+β)) / (1 + (np.exp((-α*pos)+β)))
```

Pseudo-Code 5.4. One-Cycle Pruning

Those design parameters α and β may differ according to the architecture, dataset and training procedure that are employed. However, by performing a grid-search to find the best pair for a Resnet-18 trained on CIFAR-10 to 90% sparsity, we find that the final accuracy is pretty stable to different α and β, as reported in Table 5.12. The best pair seems to be $\alpha = 14$ and $\beta = 5$, which will be used as default in further experiments.

		β				
		3	4	5	6	7
α	13	93.10 ± 0.18	93.23 ± 0.12	93.30 ± 0.07	93.31 ± 0.05	92.97 ± 0.07
	14	93.13 ± 0.06	93.16 ± 0.07	**93.46 ± 0.13**	93.09 ± 0.18	93.25 ± 0.12
	15	93.10 ± 0.03	93.21 ± 0.14	93.19 ± 0.19	93.17 ± 0.08	93.28 ± 0.03

Table 5.12. Grid search of α and β for Resnet-18 trained on CIFAR-10 to for 90% sparsity. Mean and standard deviation over 3 rounds are reported.

5.5.2 Experiments

Comparison to traditional schedules

In this section, we compare our proposed schedule, the One-Cycle Pruning, to other commonly used pruning schedules.

Pruning Methods. The different parameters used for performing pruning are summarized in Table 5.13, where they are expressed using `FasterAI` terminology.

Granularity	Criteria	Context	Schedule
`weight`	`large_final`	`local`	`one_shot` **vs** `iterative` **vs** `gradual` **vs** `one_cycle`

Table 5.13. Pruning parameters used in `FasterAI`.

We compare our pruning technique to several state-of-the-art pruning schedules: One-Shot Pruning, Iterative Pruning and Automated Gradual Pruning. As explained in Section 4.2.4, pruning schedules may be further customized by setting starting and ending points. We thus perform a grid search to identify those points for each schedule. The ending point for all schedules is found to be at the very end of training, i.e. the pruning process is carried on until training ends. On the other hand, the best starting iterations are found at 40%, 20% and 20% of the training for One-Shot Pruning, Iterative Pruning and AGP, respectively. In the case of OCP, default values identified in Section 5.5.1 are used. A visual comparison of all the schedules that we will study can be found in Figure 5.23.

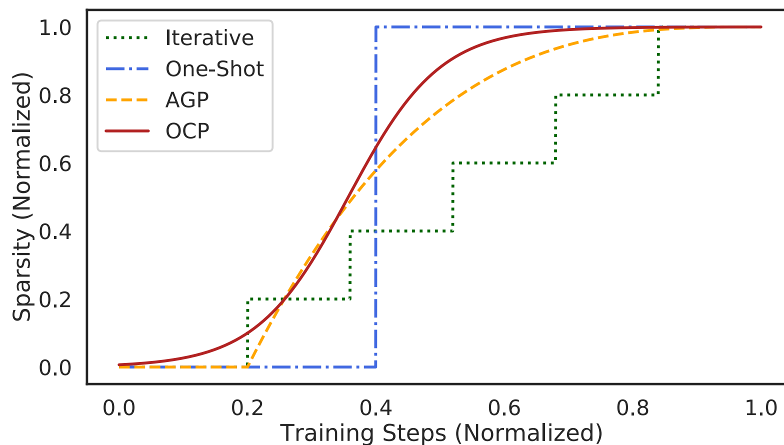


Figure 5.23. Comparison of the evolution of sparsity during training of the 4 studied pruning schedules.

Datasets and Architectures. For our experiments, the datasets have been chosen to be various in terms of image resolution and number of classes. In particular, we evaluate our methods on the three following datasets: CIFAR-10 [74], CIFAR-100 [74] and Caltech-101 [68]. Moreover, those datasets are tested on two types of popular convolutional network architectures: VGG-16 [31] and ResNet-18 [67].

Training Procedure. The networks used for our experiments are trained from a random initialization. The images are first resized to 224×224 and are augmented by using horizontal flips, rotations, image warping and random cropping. We train each model using the 1cycle learning rate method [76], where the training starts with a learning rate warm-up until a nominal value of $1e-3$, then gradually decays until the end of the training. For a fairer comparison, the experiments are conducted under a fixed training budget of 50 epochs. In particular, two experiments are conducted: (1) A comparison

of the final validation accuracy of models trained with the different schedules; (2) A comparison of the training time required to reach a target validation accuracy.

Experiment 1. The first experiment is conducted with a fixed training budget of 50 epochs. The objective is to identify the pruning schedule that is the most efficient for a tight training budget. The experiment is conducted for three levels of sparsity, namely 80%, 90% and 95% and repeated three times. The results are reported in Table 5.14.

		ResNet-18				VGG-16			
		One-Shot	Iterative	AGP	One-Cycle	One-Shot	Iterative	AGP	One-Cycle
CIFAR-10									
Sparsity	80%	93.10±0.03	93.13±0.03	93.22±0.22	**93.49±0.14**	90.25±0.14	90.64±0.19	**90.87±0.15**	90.84±0.09
	90%	92.42±0.21	91.72±0.08	92.85±0.09	**93.31±0.20**	89.82±0.19	89.76±0.18	90.67±0.25	**90.72±0.40**
	95%	91.58±0.04	87.54±0.39	92.04±0.07	**92.76±0.16**	89.73±0.37	81.46±2.87	90.56±0.31	**90.67±0.11**
CIFAR-100									
Sparsity	80%	74.21±0.09	74.18±0.29	74.78±0.09	**74.81±0.16**	67.83±0.19	67.80±0.15	67.93±0.06	**68.34±0.38**
	90%	73.34±0.23	71.80±0.05	73.83±0.41	**74.50±0.24**	67.33±0.16	62.66±1.31	67.88±0.39	**68.24±0.45**
	95%	71.68±0.16	62.88±0.27	71.92±0.30	**73.34±0.21**	66.16±0.49	61.95±0.70	**67.51±0.19**	**67.51±0.16**
Caltech-101									
Sparsity	80%	80.31±0.89	79.78±0.56	81.93±0.85	**82.31±0.88**	77.81±0.96	78.23±0.35	78.45±0.85	**78.90±0.88**
	90%	79.87±0.54	77.84±0.31	80.89±0.90	**81.84±0.16**	78.77±1.06	74.42±2.79	**78.57±0.21**	78.56±0.31
	95%	78.57±1.02	73.83±1.28	78.76±1.27	**79.81±0.92**	76.99±0.78	42.61±2.60	78.68±0.53	**78.99±0.50**

Table 5.14. Results of pruning ResNet-18 and VGG-16 with 4 different schedules. Mean and standard deviation of accuracy over 3 rounds are reported. Best results are in bold.

As can be observed in Table 5.14, One-Cycle Pruning usually outperforms other pruning schedules. To better understand how the performance of the network evolves during the training, we present the evolution of accuracy of the studied pruning schedules when pruning ResNet-18 to a sparsity of 95% on CIFAR-10 in Figure 5.24.We can observe that, although One-Cycle Pruning is the only method starting to prune at first iterations, it allows to reach a higher accuracy right from the start. We can also observe that Iterative Pruning witnesses a significant performance drop when starting its last pruning iteration. This can be explained by the fact that, the learning rate being decayed towards the end of the training, a too-large perturbation happening at that moment does not allow the network to recover quickly enough.

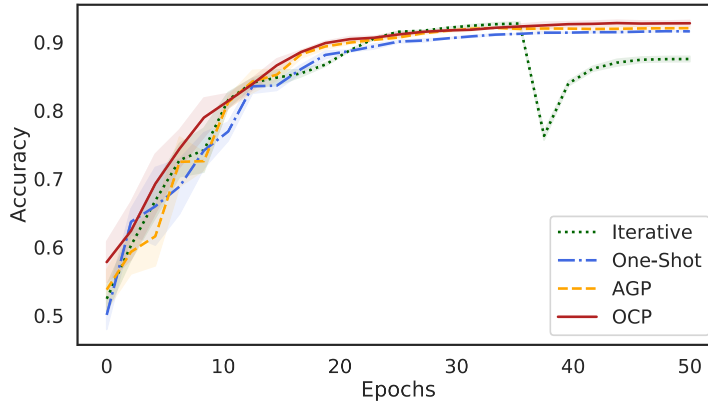


Figure 5.24. Evolution of accuracy of ResNet18 trained on CIFAR10, when applying different pruning schedules to a sparsity of 95%

Experiment 2. To better emphasize the impact of the pruning schedule on the training dynamics, we propose to perform another experiment. This time, the target validation is fixed and we let the training budget evolve according to the needs of each pruning method in order to reach the desired performance. For One-Shot and Iterative pruning, the pre-training budget is kept identical, only the fine-tuning budget is extended, i.e. the training after the pruning has occurred. We provide in Table 5.15 the results of the training budget required to reach 90%, 70% and 80% of accuracy on CIFAR-10, CIFAR-100, Caltech-101 dataset respectively, using ResNet-18 pruned to a sparsity of 95%. Training budget is expressed relatively to method One-Cycle Pruning.

	One-Shot	Iterative	AGP	One-Cycle
CIFAR-10	2.5×	4×	**1×**	**1×**
CIFAR-100	3.33×	7.5×	1.25×	**1×**
Caltech-101	2×	3.2×	1.4×	**1×**

Table 5.15. Training budget required to prune ResNet-18 to 95% to achieve a fixed validation accuracy.

Overall, the technique requiring the most-important training budget while providing the worst validation accuracy when that budget is fixed is the Iterative Pruning. Several papers have also reported a similar phenomenon, indicating that Iterative Pruning required a long fine-tuning time in order to compensate [50, 86] for the removal of weights. On the other hand, One-Cycle Pruning seems to overperform other pruning

schedules both when the training budget is fixed and when the target accuracy is fixed, indicating that such a schedule is able to reach higher performance faster.

Experiments with Lottery Tickets

We also conduct experiments concerning the Lottery Ticket Hypothesis, described in Section 2.2.1. In particular, we want to compare the quality of winning tickets found with One-Cycle Pruning to the ones found with other schedules.

Finding Lottery Tickets The experiment to discover winning tickets consists in repeating several rounds of one-shot pruning, and reinitializing the remaining weights to their original value after each pruning round [56]. This experiment thus requires several rounds of one-shot pruning to unveil winning tickets. We propose to compare the quality of found tickets when they are uncovered with different pruning schedules.

Comparison of Tickets. We conduct the LTH experiments using ResNet-18 and CIFAR-10 with the same hyperparameters as described in Section 5.5.2. In the original LTH experiments, each pruning round was performed with One-Shot Pruning. We thus perform the LTH, with different pruning schedules, until 95% of sparsity, creating the network $W_0 \odot m$, whose mask m consists of 95% of zeroes. We report in Table 5.16 the accuracy of those tickets retrained for a single epoch.As can be observed, the ticket obtained with One-Cycle Pruning significantly outperforms the subnetwork discovered with other schedules.

	One-Shot	**Iterative**
Accuracy (%)	72.213 ± 0.009	85.778 ± 0.003
	AGP	**OCP**
	86.114 ± 0.001	**89.239 ± 0.004**

Table 5.16. Validation accuracy of the reinitialized ResNet-18 sub-networks found by several pruning schedules at their last pruning round, *i.e.* at a sparsity level of 95%. The training is performed on the CIFAR-10 dataset.

Stability of OCP. To further study the difference in the results, we conduct a stability analysis of the found Lottery Tickets [57]. This analysis consists of retraining two copies, subject to a different SGD noise, of the same ticket $W_0 \odot m$, thus leading to two different trained versions $W_1 \odot m$ and $W_2 \odot m$. By then linearly interpolating their set of weights,

we can create a new network with weights $W_3 \odot m = \alpha(W_1 \odot m) + (1 - \alpha)(W_2 \odot m)$ with $\alpha \in [0, 1]$. The two copies will then be said linearly connected, i.e. they converged to the same linearly connected minimum, if the validation error of $W_3 \odot m$ remains stable for all the α values. We report in Figure 5.25 the instability evolution against the sparsity level. We denote instability error the validation error, i.e. 1−accuracy, with network $W_3 \odot m$, whose weights were interpolated at half-way between the two trained copies, i.e. for $\alpha = 0.5$. We can observe that OCP, as well as AGP, become stable at really low level of sparsity, i.e. after performing only a few rounds of LTH, while One-Shot Pruning and Iterative Pruning take more rounds before getting stable. However, AGP seems to be the only one to show signs of increasing instability for high sparsity values, i.e. for late LTH pruning rounds.

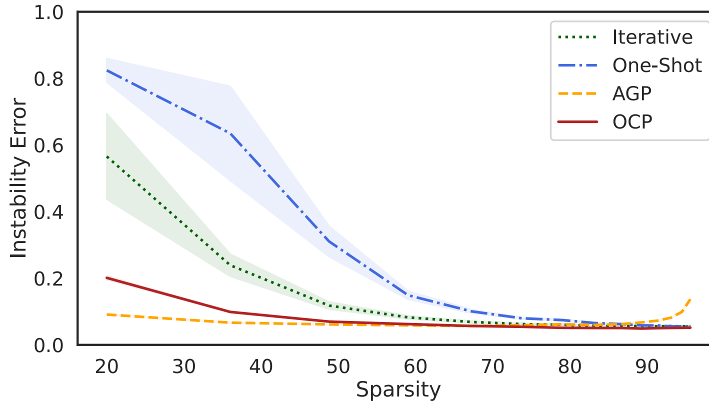


Figure 5.25. Evolution of instability error after each round of the Lottery Ticket Hypothesis when using a different pruning schedule. A low instability error indicates that the found ticket $W_0 \odot m$ is stable to retraining for that particular sparsity level.

5.5.3 Discussion & Conclusion

In this contribution, we proposed One-Cycle Pruning, a novel pruning schedule that allows a network to be pruned during the training phase, removing the need for an initial pre-training phase but also a complex and time-consuming fine-tuning phase. This pruning schedule has been designed to prune gently in early training iterations while providing a long fine-tuning phase. When compared to common pruning schedules, One-Cycle Pruning provides comparable or better results when the training budget is fixed, but also requires significantly less computation to attain a target accuracy.

We also investigated the quality of found tickets, by performing the LTH with different pruning schedules and showed that OCP is able to discover better-performing tickets, that also happen to be more stable when submitted to SGD noise.

5.6 In Brief

Summary 5

- Our contribution to the **granularity** is the proposal of a novel way to select and remove filters by turning the selection problem upside-down. Instead of selecting filters, we select shared-kernels, i.e. kernels shared among all filters of a layer. Despite not improving much the performance of a model for a fixed pruning ratio, removing shared-kernels allows to further reduce the number of operation and remaining parameters. Furthermore, because of the specificity of the granularity, it also allows to remove input features from the input images, reducing the size of the dataset that is used for training.

- The contribution about the pruning **context** is the introduction of another way to select from which layer to prune filters. In particular, this was studied by comparing two types of initialization: random and pre-trained weights. This dramatically impacts the layers from which filters can be removed. While random initialization leads to an even removal between all layers in the network, pre-trained weights remove filter towards the end of the network.

- The contribution about the pruning **criteria** concerns the introduction of a clustering technique prior to the selection of weight by the pruning criteria. In particular, this allows to explicitly reduce the redundancy while preserving rare features in the network, helping to improve the final performance of the pruned network.

- Our contribution to the **schedule** research axis is a novel schedule named One-Cycle Pruning, that allows to perform pruning in a single training stage. It thus performs pruning and training in a common process, allowing to both save computations but also to reach higher performing and more stable models.

CHAPTER 6

Use-Case: DeepFake Detection

Contents

6.1 Introduction . . . 134
- 6.1.1 DeepFakes Detection Challenge . . . 134
- 6.1.2 Related Work . . . 135

6.2 Methodology . . . 136
- 6.2.1 The Dataset . . . 136
- 6.2.2 The Network . . . 137
- 6.2.3 Data Augmentation . . . 141
- 6.2.4 Compression of the solution . . . 142

6.3 Results . . . 143
- 6.3.1 Ablation Study . . . 143
- 6.3.2 Comparison to other methods . . . 144
- 6.3.3 Interpretation . . . 145

6.4 Proof-of-Concept: Deepfake Buster . . . 146

6.5 Discussion and Conclusions . . . 148

6.6 In Brief . . . 149

"People almost invariably arrive at their beliefs not on the basis of proof but on the basis of what they find attractive."
— Blaise Pascal

6.1 Introduction

Over the last few years, the rise of synthetic media has forced us to realize that we cannot trust images, videos and audio records anymore. With the growth of generative models, we are now able to create realistic content and even worse, those models are now widely available to the public at large. The commodification of such technologies will importantly impact the way we consume information every day. In particular, DeepFakes, i.e. manipulation technique swapping the identity of someone in an image or video, have produced increasing worries among specialists. Indeed, despite not being perfect, such techniques already have enabled a new attack vector for social engineering, blackmailing, and political destabilization.

Fake news are particularly spreading on social media. They are estimated to be shared six times as fast as real news [87]. With a perpetually growing number of social media users and more accessible tools for DeepFake creation, the sharing of fake news could pose a real problem in the near future. Indeed, if deep fakes reach a point where humans can no longer distinguish true information from false one, this could greatly impair mainstream media information.

For this reason, we propose to investigate the problem of DeepFake detection with the use of deep learning techniques. Moreover, because DeepFake creation is now available at scale, the detection method should be able to process large amounts of data in restricted time frames to keep up with the increasing number of shared media. To do so, we propose to apply the knowledge acquired by previous contributions to compress a neural network, reducing its parameter count and processing time, while retaining its performance.

The motivation is twofold: (1) show that compression techniques can be applied to real-case applications, involving datasets of a greater complexity than most benchmarking datasets: (2) provide a light detection network, suited for spotting DeepFakes images on a large flow of images, such as present on social media.

6.1.1 DeepFakes Detection Challenge

The DeepFake Detection Challenge (DFDC) is a challenge that has been created following a collaboration between several big technology actors such as Facebook, Microsoft, Amazon, the Partnership on AI's Media Integrity Steering Committee and academics partners such as Cornell Tech, MIT, University of Oxford, UC Berkeley, University of Maryland, College Park, and University at Albany-SUNY.

This challenge has two objectives: (1) Produce accessible technologies for DeepFake Detection, helping users to discover when a video or image has deceptively been altered; (2) Release a high-quality and freely available dataset, to speed-up research in the field and provide a common benchmark, facilitating comparison of newly proposed methods.

This challenge was thus the perfect opportunity for us to join the research of DeepFake detection and to be at the forefront of new developments in the field. During the challenge, we proposed a DeepFake detection method that reached respectively top 4% and top 6% on the public and private leaderboards [1].

6.1.2 Related Work

DeepFakes creation often generates artifacts that are usually exploited by recent methods. Those artifacts can generally be separated into two groups: *spatial* and *temporal*. Detection techniques can thus focus on either of those types of artifacts, or even both.

Spatial Artifacts Detection. This type of artifact generally appears when the generated content is embedded into the original frame. Treating the problem as image classification allows CNN models to detect the artifacts present in the frames [88], thus classifying each frame individually. Those artifacts can come in many forms and dimensions, requiring different techniques to be detected. For example, mesoscopic information can be analyzed to detect the forgery [89]. Because the generated and original contents usually do not come from the same source, some warping artifacts may appear during the blending phase, which can also be detected [90]. Model fingerprints related to the forgery method that was used also can be exploited [91, 92].

Temporal Artifacts Detection. Because a majority of DeepFake generation techniques are used in frame-by-frame mode on videos, this can leave temporal inconsistencies or artifacts such as flickering or jitter that can be detected [93–95]. For example, biometric features such as eye blinking, lip-sync and facial movements can be exploited to detected manipulated faces [96, 97]. More recent methods even study the visual heartbeat rhythm to assess the veracity of a video [98].

Hybrid Techniques. Because both spatial and temporal artifacts usually appear in DeepFakes videos, they can both be exploited in order to detect forged videos. Combining those techniques usually requires more data-intensive models such as 3D CNNs [99], or a combination of CNN and RNN [100].

[1] https://www.kaggle.com/competitions/deepfake-detection-challenge/leaderboard

6.2 Methodology

6.2.1 The Dataset

The dataset used for our research is the DFDC dataset [101], released for the challenge. This dataset belongs to the so-called third generation of Deepfakes datasets [102], as it contains a large number of high-quality videos, with agreements from individuals appearing in the dataset. It is the largest currently available dataset consisting of 124k videos, divided into three subsets: (1) Training Set; (2) Validation Set; (3) Test Set.

Training Set. The training set provided in DFDC is comprised of 119,154 video clips, of 486 unique subjects and from which 100,000 videos are altered. Different techniques can be used to create DeepFakes, e.g. DFAE, MM/NN face swap [103], NTH [104] and FS-GAN [105], that are of various quality to better reflect the sporadic behavior of social networks. Each video is 10 seconds long, and composed of 300 frames. Examples of manipulated faces can be found in Figure 6.1.

Figure 6.1. Examples of manipulated faces present in the DFDC dataset. Different deception methods are used to generate the faces.

Validation Set. The validation set consists of 4,000 video clips, which are also 10 seconds long, in which 2,000 clips contain Deepfakes. This set was created using 214 unique subjects, none of which were present in the training set. The same Deepfake generation techniques were used, with the addition of a new one, StyleGAN [106]. Augmentations, e.g. noise, blur, JPEG artifacts, and distractions, e.g. logo or face overlay, are also added to 80% of the videos.

Test Set. The test set is comprised of 5,000 video clips of 260 unique subjects that have not been seen before. Again, augmentations and distractions were added to 80% of the set, including new distractions such as dog masks or flower crown filters.

6.2.2 The Network

In this paper, we propose a lightweight solution for DeepFakes detection. The general pipeline is represented in Figure 6.2 and is composed of two networks: (1) a face detection network, extracting detected faces contained in an image or a video; (2) a DeepFake recognition network, classifying the extracted faces as real or fake.

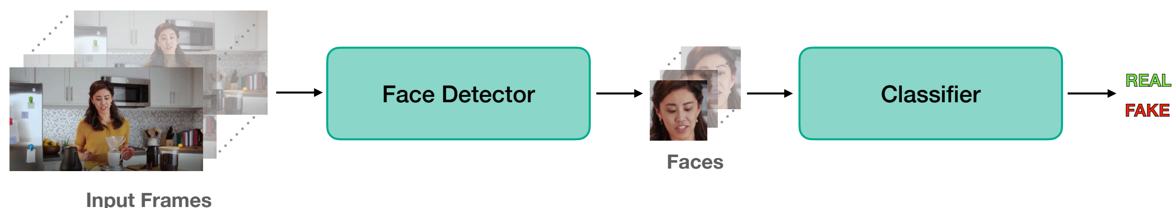


Figure 6.2. The general framework of our proposed solution. It takes a single frame (an image) or several frames (a video) as an input, that are fed to a face detection network, to extract the different faces that are present. Those faces are then used as the input of our DeepFake detection network, yielding the final prediction.

Face Detection. To detect faces, several state-of-the-art methods have been considered. The objective is to find a good trade-off between detection performance and speed of processing. In particular, several publicly-available methods have been tested on 50 videos chosen randomly from the DFDC dataset, each of 300 frames. The detection accuracy and the speed of execution are reported in Table 6.1.

Algorithm	Accuracy (%)	Speed (im/s)
MTCNN [107]	87.5	14.2
BlazeFace [108]	61.0	133.6
DSFD [109]	95.8	3.9
RetinaFace [110]	97.7	7.5

Table 6.1. Characteristics of Face Detection models. Speed is measured on a Nvidia GTX 1080 GPU.

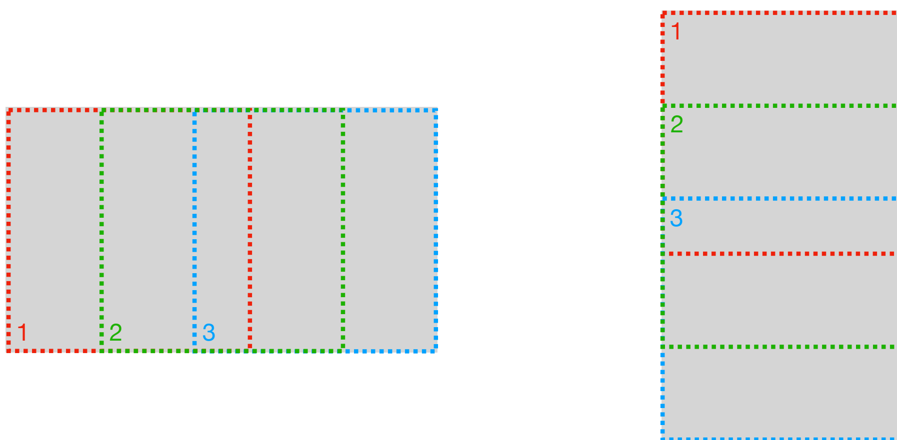
As can be observed, the method providing the best detection accuracy is RetinaFace, while the fastest one is BlazeFace.

We investigate the reason that could make BlazeFace less performant than other methods and observe that the proposed model first resizes the input frame to a 128×128 square image before performing the prediction. Because videos in the dataset are of dimension 1920×1080, or 1080×1920, BlazeFace thus greatly downsizes frames, losing crucial information about the data. Moreover, as images are not initially of a square shape, it introduces some distortion in the image, making the detection of faces more difficult. Instead, we propose to improve the preprocessing step of BlazeFace by extracting three square crops from the image and performing the detection on those three extracted crops independently, as presented in Pseudo-Code 6.1.

```
Image = Array(H, W, C)
if W<H:
        left_crop= Image[:, :H, :]
        middle_crop= Image[:, W//2-H//2:W//2+H//2, :]
        right_crop=  Image[:, -H:, :]
if H>W:
        top_crop= Image[:W, :, :]
        middle_crop= Image[H//2-W//2:H//2+W//2, :, :]
        bottom_crop=  Image[-W:, :, :]
else:
        pass
```

Pseudo-Code 6.1. Implementation of the proposed cropping method

This allows to extract the crops for horizontal images as represented in Figure 6.3a or vertical images as represented in Figure 6.3b.



(a) Different crops for an horizontal Image. **(b)** Different crops for a vertical Image.

Figure 6.3. The proposed crop extraction method for different image formats.

After having extracted the three crops, we finally feed them to BlazeFace to make its prediction. Detected faces from the crops are then filtered by an Non-Maximum Suppression (NMS), keeping only the best face in case there are overlapping predictions, i.e. if a face happens to be inside multiple crops. The accuracy and processing speed of the new method is reported in Table 6.2.

Algorithm	Accuracy (%)	Speed (im/s)
Improved BlazeFace	89.6	50.4

Table 6.2. Characteristics of Face Detection models. Speed is measured on a Nvidia GTX 1080 GPU

As can be observed in Table 6.2, extracting three crops to perform face detection with BlazeFace significantly improves the results of the face detection mechanism. It also allows us to obtain a better trade-off between accuracy and speed than other methods.

Moreover, there are two failure modes for detection models: (1) not detecting a face when there is one, also known as a *false negative*; (2) detecting a face when there is not, also known as a *false positive*. From our experiments, we find that the proposed BlazeFace detection is more sensitive to false negatives than false positives, with 89.6% of frames correctly predicted, 7.4% of missing faces and 3% of extra faces. To further improve the quality of our face extraction step, we decide to provide a fallback model for the cases of BlazeFace not detecting a face in an image. In this regard, the most accurate face extractor, RetinaNet, is used when no face is detected in a frame. The principle of our final proposed face detection method can be found in Figure 6.4.

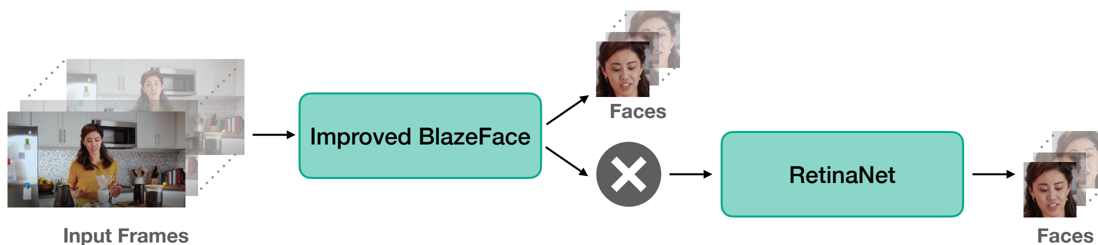


Figure 6.4. The general framework of our proposed face extraction method. In case no face is found with our main face extractor, a more accurate second one is used as a fallback.

The final performance, provided in Table 6.3, indicates that it provides a substantial increase in the detection rate, with an accuracy of 96.8%, with only 0.1% of false negatives and 3% of false positives, at the cost of a slight decrease in processing speed.

Algorithm	Accuracy (%)	Speed (im/s)
Improved BlazeFace with Fallback	96.8	48.3

Table 6.3. Characteristics of Face Detection models.

DeepFake Classification. After having extracted the faces in the frames of our input videos, we now have to classify them as real faces or DeepFakes. We empirically find that extracting the faces from 32 evenly-spaced frames out of the initial 300 provides a good performance, while avoiding the classification model to overfit. The classification network used for this purpose uses a custom architecture inspired by the design of ResNets [67], but replacing the common convolution blocks by our custom block, that we name *DFBlock*. This block, represented in Figure 6.5, consists of 2 modules: (1) an inverted residual module [36], composed of a first pointwise convolution, allowing to expand the amount of channels before performing the spatial feature extraction by a depthwise convolution and finally squeeze the channels by another pointwise convolution, each of those convolutions being followed by a batch normalization layer and a Rectified Linear Unit (ReLU); (2) a Squeeze-Excite module [37], a content-aware mechanism that weights each feature map adaptively, which, at the cost of a slight increase in the total parameter count, allows to substantially increase the classification performance.

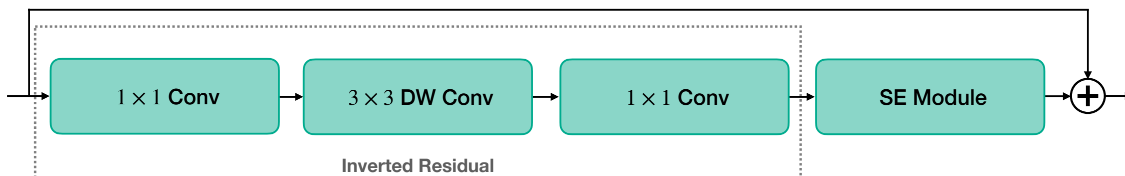


Figure 6.5. DFBlock, the building block of DFBuster. It consists of an inverted residual module, a squeeze-excite (SE) module.

Our architecture, which we name *DFBuster*, and represented in Figure 6.6, is composed of a stem of 3 convolutional layers, then 4 double DFBlocks, each followed by a MaxPooling layer. After the first block, we insert a Self-Attention module, enforcing attention to long-range dependencies, helping the model to focus on object shapes rather than local regions [111], significantly improving the performances while slightly increasing the number of parameters. This Self-Attention module also allows to retrieve information about the network's decision, helping us to further interpret the results.

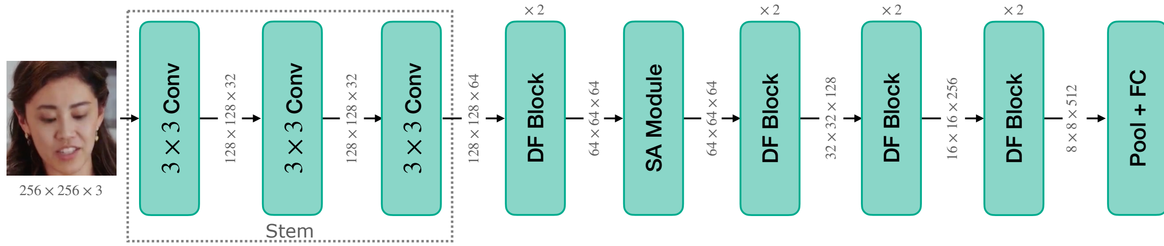


Figure 6.6. Our proposed DeepFake classification network: DFBuster. It consists of a 3-convolutional layers stem, and 4 double DFBlocks. A Self-Attention (SA) module is also introduced to further improve classification performance.

6.2.3 Data Augmentation

As we discussed in Section 6.2.1, organic content found on the internet is subject to come from various environments and to undergo quality degradation. For those reasons, the data augmentation that we apply during training has been thoughtfully chosen, in order to make our model as robust as possible. The augmentations we apply to our training data are represented in Figure 6.7. They consist of image transforms of two types: (1) transforms that increase the scenarios of recording scenes such as warping, zooming, and contrast; (2) transforms replicating picture acquisition imperfections that can be observed on organic videos such as JPEG artifacts, motion blur, and Gaussian noise.

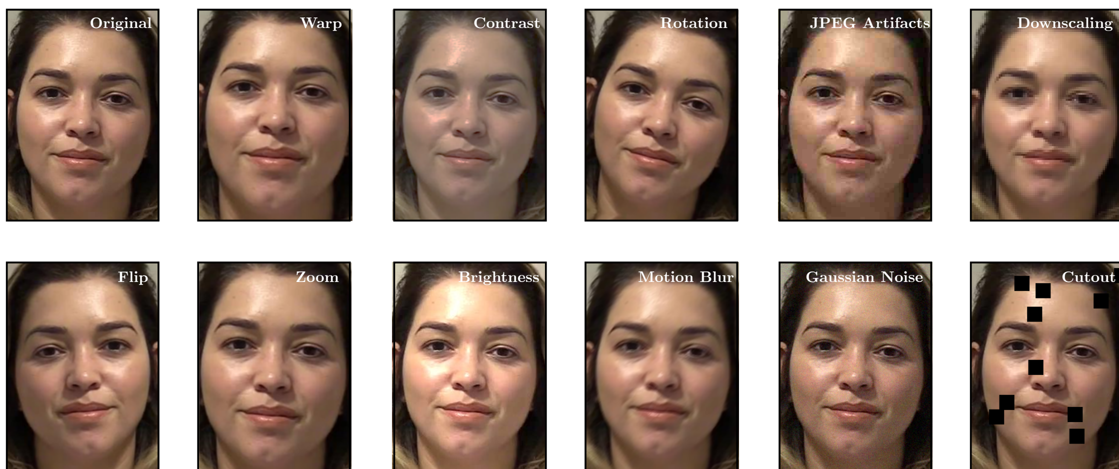


Figure 6.7. Illustration of the augmentations applied to training data. The transforms have been chosen to create a model robust to organic videos that it might encounter in real-world settings.

To make our model even more robust and less prone to overfitting, the different data augmentation techniques are injected following a Progressive Learning technique [112]. This method is inspired by Curriculum Learning methods, i.e. starting the training with easy examples then gradually increase their difficulty [113]. The idea is thus to impose subtle transformations as the training starts, then gradually increase their strength as the training progresses. By doing so, we ensure that the model has the ability to extract relevant features from the images, before learning to differentiate images in more challenging contexts. We also use a technique called Mixup [114], which blends two input images as well as their labels, which we found to be very efficient to improve our classification performance on this dataset.

The training is performed for 20 epochs, with a learning rate of $1e-2$, with the AdamW optimizer [21] and following the 1cycle learning rate technique, performing a learning rate warm-up, then gradually decaying the learning rate value towards zero [76].

6.2.4 Compression of the solution

By design, the architecture is meant to be as parameter efficient as possible. However, there is still much that can be made to further reduce the number of parameters and processing speed. For this purpose, we consequently use pruning, quantization, knowledge distillation and batch normalization folding techniques.

Pruning. In order to reduce the number of parameters, we apply pruning to our architecture. The pruning parameters, expressed using the `FasterAI` terminology, are summarized in Table 6.4. In particular, we perform filter pruning, allowing us to easily take advantage of the removal of parameters for model speedup. The selection of filters to remove is performed according to their l_1-norm, with local and global removal both evaluated. Moreover, pruning is applied right from the start of training, by using the One-Cycle Pruning schedule, presented in Section 5.5, which not only allows to reach a higher accuracy than most pruning schedules, but allows to do it in a more constrained training budget. We empirically find the upper bound for sparsity to impose in the network to be at 50%, above which the performance of the model decreases.

Granularity	Criteria	Context	Schedule
`filter`	`large_final`	`global` **vs** `local`	`one_cycle`

Table 6.4. Pruning parameters used in `FasterAI`.

Quantization. The quantization that we apply in our work uses the Automatic Mixed-Precision [115] technique, presented in Section 2.4.3. It allows us to obtain a model using FP16 precision, but performing all the training updates using FP32. As the forward pass is performed using FP16, Mixed-Precision thus allows us not only to have a model that has a faster inference, but also a faster training.

Knowledge Distillation. Pruning and Quantization are usually not lossless, i.e. a trade-off must be found between the reduction of the size of the network and the performance drop. In order to mitigate that drop, we propose to perform Knowledge Distillation [26] where the student model is the pruned and quantized one and the teacher model is the model before compression. Doing so encourages the compressed model to better recover its lost performance.

Batch Normalization Folding. After training, the parameters of the Batch Normalization layers in a model are fixed. This means that we can blend the effect of normalization into the weights of the convolution operation preceding it. We can thus replace the weights and biases of convolutional layers preceding batch normalizations as presented in Section 2.5.2 and remove the batch normalization layers.

6.3 Results

6.3.1 Ablation Study

To better understand the impact of each technique used to create our final lightweight model, we perform an ablation study, whose results are presented in Table 6.5.

	Accuracy	Loss	Model Size (Mb)	ROC-AUC
Baseline	74.34	0.5563	46.22	0.893
+ Self-Attention	75.36	0.5511	46.35	0.903
+ Squeeze-Excite	75.54	0.5445	49.17	0.909
+ Mixup	78.24	0.5312	49.17	0.917
+ Progressive Learning	81.64	0.5098	49.17	0.935
+ Pruning (Local)	82.08	0.5057	19.13	0.937
+ Mixed-Precision	81.79	0.5062	9.59	0.929
+ KD	81.93	0.5041	9.59	0.934
+ BN Folding	81.93	0.5041	9.58	0.934

Table 6.5. Ablation Study of the proposed model.

The baseline model uses simplified DFBlocks, which only contain the inverted residual module. It is also trained using the data augmentation presented in Figure 6.7. The first phase of our ablation study uses techniques aiming to improve the classification performance, while maintaining the parameter count. It consists in gradually adding components to the architecture, i.e. the Self-Attention and Squeeze-Excite modules, which help the model to reach a higher performance, at the price of a slight increase in the parameter count. We then adjust the data augmentation process, to incorporate the Mixup and Progressive Learning methods, which dramatically improve the network's performance. The second phase of our ablation study concerns the methods aiming to reduce the parameter count while maintaining the performance. In particular, we show that the pruning, Knowledge Distillation, Quantization and Batch Normalization lead to a 5× reduction of the model size, while keeping the performance intact.

6.3.2 Comparison to other methods

We compare our DeepFake classifier to other state-of-the-art classifiers such as Mesonet [89], ResNet-18 [67], EfficientNet-B0 [116], Xception [117], that were used for DeepFake classification, but also to the two best solutions proposed at the DeepFake Detection Challenge.

	ROC-AUC	Model Size (Mb)	FLOPS (1e9)	Inference Time (ms)
MesoNet	0.786	0.91	0.06	3.84 ± 0.22
Resnet18	0.886	357.53	1.81	4.35 ± 0.24
EfficientNet-B0	0.913	127.65	0.39	16.17 ± 0.13
Xception	0.929	665.08	38.92	13.52 ± 0.32
Selim Seferbekov [118]	0.984	6109.12	111.00	197.02 ± 21.76
WM [119]	0.985	1722.85	95.84	56.21 ± 8.63
DFBuster - Global	0.941	7.23	0.88 (FP16)	6.17 ± 0.19
DFBuster - Local	0.934	9.58	0.42 (FP16)	6.09 ± 0.21

Table 6.6. Comparison to the state-of-the-art DeepFake classification methods. The Inference Time is computed on single image, using a Nvidia GTX 1080 and averaged over 10 iterations.

As can be observed in Table 6.6, our proposed model is able to outperform most of state-of-the-art classification models, with the exception of Selim Seferbekov and WM,

the two top solutions of the DeepFake Detection Challenge. However, those two solutions involve a technique called *Model Ensembling*, consisting in making several vote for the final prediction, which results in a very costly model storage and inference pass. The solution providing the lightest and fastest solution is by far MesoNet. However, even though it was able to accurately classify on the Face2Face dataset [120], the DFDC dataset is more challenging and requires more power than MesoNet is able to provide. We thus find that our solution is able to provide a better trade-off between performance and computation costs.

6.3.3 Interpretation

As it is increasingly difficult to notice DeepFakes, especially for the human eye, it is essential to provide tools to better understand our model's decisions. This can be achieved with attention techniques, highlighting parts of the input image that are crucial for the decision of our model. The self-attention module that we introduced in DFBuster provides such a functionality. Indeed, it enforces the model to consider long-range dependencies, thus better capturing details from the input image. The attention maps generated by our model can then be visualized, as presented in Figure 6.8. As can be observed, the highlighted parts generally concern central face parts, such as the eyes, nose, and mouth. This shows that DeepFakes generation methods still create visual artifacts that classification models can exploit.

Figure 6.8. Examples of extracted faces and their corresponding attention map. Face parts that are the most useful for the final prediction are highlighted.

6.4 Proof-of-Concept: Deepfake Buster

As a proof-of-concept, we developed a DeepFake Detection application. This application was created using a Jupyter Notebook [121], allowing to execute pieces of code sequentially. This notebook has been turned into a standalone web application, with interactive widgets using the Voilà extension [122]. Finally, the application is hosted and turned into an executable environment, making it reproducible by anyone with the Binder service [123].

More precisely, the application runs a simplified version of the solution proposed in Section 6.2. The face extraction network is exclusively based on the improved BlazeFace network and the classification network based on a compressed version of the model proposed for the DFDC. The application, represented in Figure 6.9, is composed of 3 buttons:

- An `Upload` button that the user can click to upload an image or video.
- A `Get Prediction` button that the user can click to obtain the prediction of the model about the uploaded file.
- A `Show Attention` button that the user can click to show the Attention heatmap once the application has made its prediction.

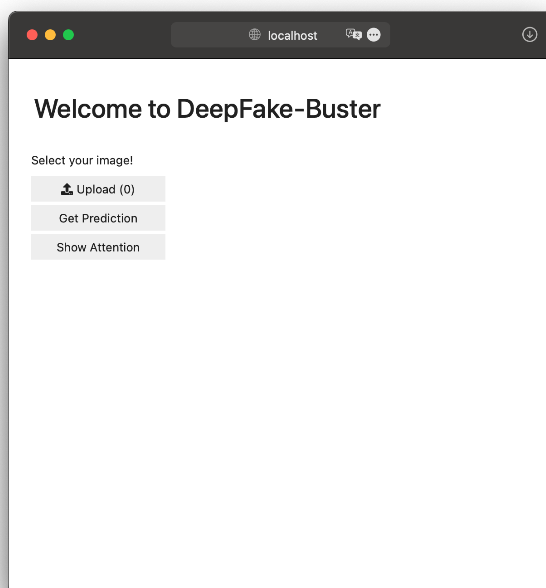
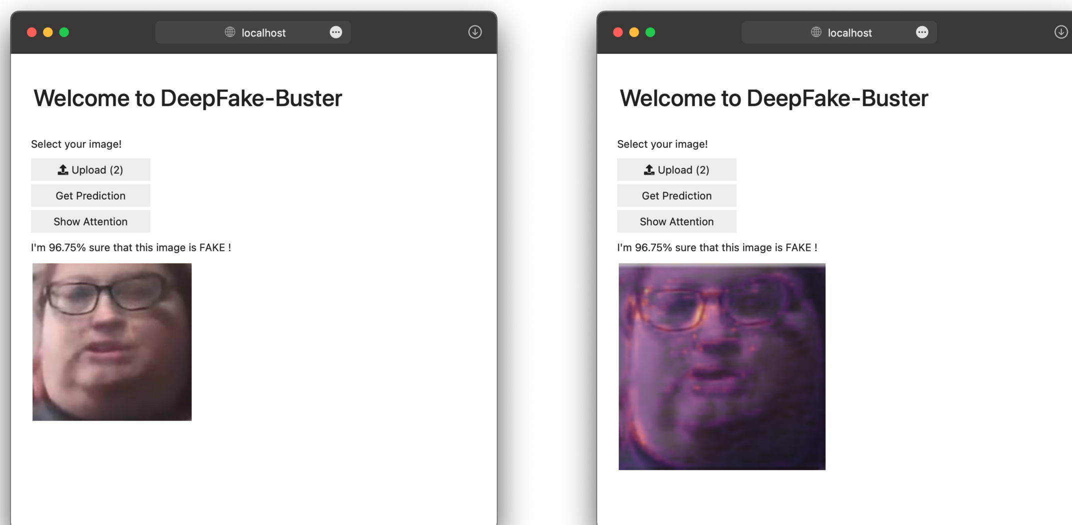


Figure 6.9. The DeepFake Buster application, running in a web browser.

An example of the usage of the application is presented in Figure 6.10, where it is tested for a manipulated face extracted from a video of the DFDC dataset. The prediction result, depicted in Figure 6.10a, illustrates that the model is confident about the falsity of the provided image. The result of the attention analysis, provided in Figure 6.10b, highlights parts of the image that the model found valuable for its prediction. The most important parts are concentrated towards the person's glasses, indicating that the model found clues of forgery in the glasses, likely because of the presence of rendering artifacts.

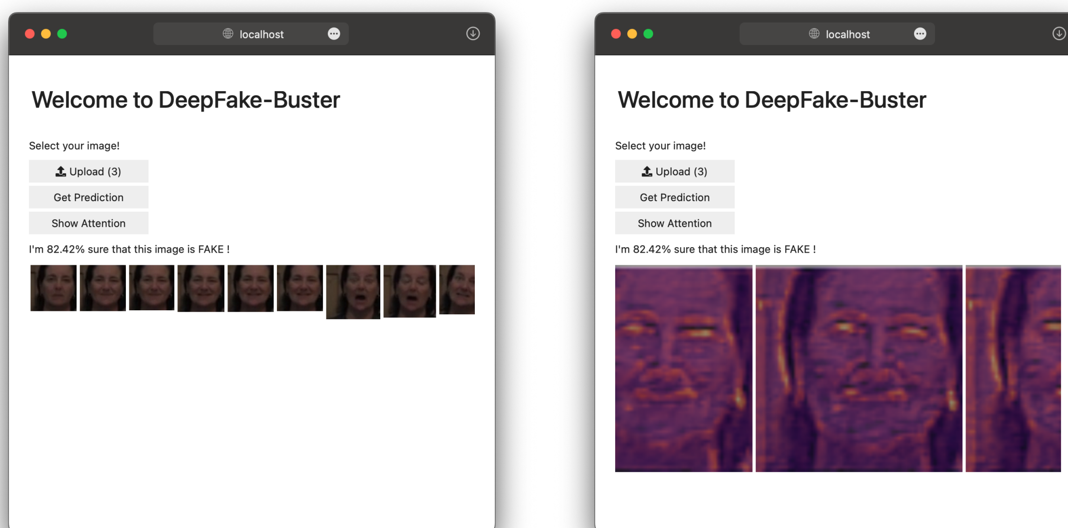


(a) Result of the prediction on the provided face.

(b) Result of the prediction of the attention heatmap on the provided face.

Figure 6.10. Example of usage of the web-based application. An image extracted from a forged video is uploaded and fed to the model, which predicts it to be false and provides an attention map, allowing for a better interpretation of results.

The DeepFake Buster application also allows the user to upload videos. Those videos will first be decomposed into frames, then the model predicts its results on each frame separately, as shown in Figure 6.11a. The Attention option is also available, indicating important parts in each video frame, as depicted in Figure 6.11b.



(a) Result of the prediction on the provided video and the extracted faces.

(b) Result of the prediction of the attention heatmap on the provided face.

Figure 6.11. Example of usage of the web-based application. A forged video is uploaded and fed to the model, which first extracts faces from each frames, and then makes its prediction. It also provides the corresponding attention maps, allowing for a better interpretation of results.

6.5 Discussion and Conclusions

In this contribution, we propose a lightweight solution for DeepFake detection. This solution consists of two networks: (1) a face extraction network, using an improved version of BlazeFace; (2) a classification network, that has been built with efficient computation layers, but also several compression techniques such pruning, knowledge distillation, quantization and batch normalization folding.

We show that our solution is able to compete with state-of-the-art DeepFake detection techniques in terms of performance, but also that it provides a better trade-off between performance and compute costs. Moreover, because it contains a Self-Attention layer, it is possible to directly visualize the important features for the model's prediction.

In addition, we propose a proof-of-concept application, running the detection model in a web-based application, that can be used by anybody that desires to verify if the video or picture being looked at is real or fake.

6.6 In Brief

Summary 6

- DeepFakes creation is becoming a **commodity**, with many publicly-available tools for image and video creation.
- To counteract the menaces of DeepFakes, several influential actors joined forces to organize the DeepFake Detection Challenge **challenge** and release an extensive **dataset**.
- We propose Fake-Buster, a **lightweight** neural network, designed for DeepFakes Detection.
- Our solution offers competitive performance, while being **smaller** and **faster**.
- We validate our results on real-case scenarios, by participating to the DFDC, and rank at **4%** and **6%** on the public and private test set respectively.
- We also release a **proof-of-concept application** to perform DeepFake detection on the web.

CHAPTER 7

Conclusions

"Every new beginning comes from some other beginning's end."
— Seneca

The present thesis proposes a number of novel research ideas in the field of neural network compression, and neural network pruning in particular. During our research, we identified a lack of description conventions for pruning methods. For that reason, the first presented contribution concerns a novel description of pruning techniques, developed according to four axes, that we hope will help to federate further research. Consequently, we propose a compression library, built according to the provided pruning description, thus allowing for a seamless mapping between research ideas and their implementation. In addition to the pruning study and description and the related pruning toolbox, we proposed four theoretical contributions, each advancing in one of the four description axes. We also presented a proof-of-concept application, combining different contributions and methods learned throughout this dissertation, and applied to a real-case scenario concerning DeepFake detection on visual data. In this Conclusion, we proceed to synthesize our different contributions and provide ideas for improvements and further development. Finally, we conclude with more general considerations about the field of neural network compression and the future of deep learning.

Contributions

The following synthesizes the different contributions presented in the thesis:

- **A universal description of pruning techniques.** We identify four components allowing us to unequivocally and completely define currently existing pruning techniques. Those components are: the granularity, the context, the criteria, and the schedule. Defining the pruning problem according to those components allows us to subdivide the pruning problem into four mostly independent subproblems and also to better determine potential research lines.
- **A lightweight compression library.** As of today, few tools are available to make neural network compression accessible. Indeed, either those tools are dedicated to research, making them often inconvenient to apply to other usecases than those they were designed for; or they are one-sided, only allowing to apply a single compression method. To fill this gap, we proposed `FasterAI`, a PyTorch-based library, that is intended to be helpful to researchers, eager to create and experiment with different compression techniques, but also to newcomers, that desire to compress their neural network for a concrete application. In particular, the sparsification capabilities of `FasterAI` have been built according to the previously defined pruning components. This allows to perform various experiments seamlessly, but also to more easily integrate new methods.
- **Advances in pruning granularity.** In this study, we propose an alternative to the filter granularity when performing pruning. The filter granularity is often

used because of its ability to reduce the number of parameters in a network while allowing it to remain dense. Indeed, once filters have been zeroed-out, they can be removed from the architecture, and re-expressed accordingly. As a result, such type of pruning does not require dedicated resources to take advantage of parameter compression and speed-up. However, instead of selecting filters to remove, the same benefit can be achieved by removing kernels in the same position from all filters in a layer. We call this granularity *shared-kernel* and show that, not only does it allow to reach higher compression rates in some cases, but also that it gives indications of important input features. This also allows to reduce the size of the dataset to be stored and the size of input data to be used.

- **Advances in pruning context.** When comparing parameters to remove during the pruning process, two methods are usually used: (1) local pruning, which compares the parameters in each layer independently; (2) global pruning , which compares parameters from the whole network. In this contribution, we propose to base our selection on the sensitivity of different layers. More precisely, we compare the sensitivity of models that are trained in two ways: from random initialization and from pretrained weights. We observe that networks trained from pretrained weights usually possess deep layers that are less sensitive to pruning than networks trained from random initialization. As a result, the pruned fine-tuned models usually possess fewer parameters but more computation than networks trained from-scratch. This suggests that pruning methods can be tailored for different initialization methods according to the budget constraints.
- **Advances in pruning criteria.** There exist a lot of different pruning criteria that are used as a proxy for parameter importance. Simple ones such as evaluating parameters according to their magnitude turned out to be the most efficient and generalizable. However, such criteria are lacking of a critical concept: they do not seek to explicitly reduce the redundancy in parameters that they use. To overcome this situation, we propose to use a technique inspired from Explainable Artificial Intelligence (XAI), called Activation Maximization, to cluster parameters having similar functionalities in the decision process. By then applying common pruning criteria in each group, we ensure to retain a candidate for each functionality, while removing redundant ones.
- **Advances in pruning schedule.** Most pruning techniques are suggesting to start from a trained model. While generating decent results, such techniques turn out to be time-consuming. Moreover, it has been shown that pruning acts as a regularization, and that such regularization should be used early in training to take advantage of its effects. For this reason, we propose the *One-Cycle* schedule, allowing to start the pruning directly from the start of the training phase. As a result, networks are trained in a significantly shorter training time and usually reach better performance.

- **A use-case.** To demonstrate the advantages of compression techniques and use the knowledge acquired in previous experiments, we decide to apply our compression technique to an actual problem: the DeepFake detection. To do so, we propose a lightweight solution, able to extract faces from a video and to perform classification on those faces. The architecture uses many techniques to be as efficient as possible.
- **A proof of concept application.** To further demonstrate the capabilities that can be achieved by neural network compression, we provide a publicly available and web-based application that executes the previously designed DeepFake detection technique. This application thus allows anyone to perform detection on an image or video of their choice.

Perpectives

Several research directions can be inferred from the proposed work. In particular, we propose four directions for future works:

- **Sparse training.** The best pruning techniques currently require some data information before removing weights in the network. Studying the possibility to directly start from a sparse network could not only save training time, but also would allow training models whose dense counterparts would not fit in memory.
- **Hardware co-design.** Pruning techniques are often incompatible with most current hardware and compilation kernels. Although specialized kernels to support sparse computing are being developed, one could investigate at a lower level the interaction between hardware and pruning techniques, to provide the best trade-off between cost, accuracy and sparsity.
- **Architecture Design.** Pruning can be seen as a Neural Architecture Search (NAS) technique. As demonstrated in Section 2.2.2, it is possible to remove connections but also to grow them. One could thus study novel techniques, at the intersection of pruning and growing, allowing to discover new and efficient architectures.
- **Interaction of compression techniques.** Pruning is only one of available compression techniques and, as shown in Chapter 2, one way among others to discover models that are closer to the Pareto optimality. To some extent, there thus are interactions and potential interferences that can occur when mixing different compression techniques. One could thus study the different ways to combine compression techniques to develop novel methods and more efficient models.

Final Considerations

Although it held for decades, Moore's Law is coming to an end as we approach the fundamental limits of CMOS technologies [124]. This means that we are not guaranteed more computing power in the future. Instead, we might enter a new era of collaboration between software, hardware, and machine learning communities. Indeed, to continue to provide more computing capabilities, there is currently a shift from task-agnostic to domain-specialized hardware, optimizing for operations common to neural networks. To be economically viable, machine learning accelerator hardware must have a lifetime of at least three years, making the development of new hardware architectures challenging, especially in a fast-paced field such as machine learning [125]. On the other hand, the quest for more efficiency will thus undeniably go through neural network compression techniques, and particularly sparse computations. Indeed, popular hardware vendors have recently started to release accelerators for sparse computations [126], which will continue to facilitate the development and deployment of new pruning methods.

For those reasons, co-design approaches should be preferred in order to facilitate a simultaneous breakthrough in hardware, software and algorithmic fields [127]. However, the main danger of increasingly specialized hardware is that it will hinder novel research directions. Indeed, it is believed that the design choices of hardware have shaped the machine learning field as we know it today [128]. For example, several techniques such as Capsule Networks include new operations to fix some Convolutional Neural Networks shortcomings. However, because those operations are not optimally-implemented on current hardware, performances of Capsule Networks are highly sub-optimal and interest in those architectures has dried out [129].

Debates are currently ongoing about future trends and research directions, that will allow us to reach Artificial General Intelligence [130]. However, it is unclear whether current versions of neural networks will be sufficient to create a general thinking machine, nor will current hardware enable to develop and execute such methods. It is therefore essential to keep the software-hardware search space as wide as possible to continue exploring new research directions and avoid machine learning researchers becoming what they fear the most, models that have overfit to their environment.

Bibliography

[1] D. Amodei, D. Hernandez, G. Sastry, J. Clark, G. Brockman, and I. Sutskever, “Ai and compute”, 2018, accessed on 17 September 2022. [Online]. Available: https://openai.com/blog/ai-and-compute/

[2] J. Sevilla, P. Villalobos, J. F. Cerón, M. Burtell, L. Heim, A. B. Nanjajjar, A. Ho, T. Besiroglu, and M. Hobbhahn, “Parameter, compute and data trends in machine learning”, 2021, accessed on 2 August 2022. [Online]. Available: https://www.lesswrong.com/s/T9pBzinPXYB3mxSGi/p/GzoWcYibWYwJva8aL

[3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models”, *CoRR*, vol. abs/2001.08361, 2020.

[4] E. Strubell, A. Ganesh, and A. McCallum, “Energy and policy considerations for deep learning in NLP”, in *Proceedings of the Association for Computational Linguistics, ACL*, 2019.

[5] D. A. Patterson, J. Gonzalez, Q. V. Le, C. Liang, L.-M. Munguía, D. Rothchild, D. R. So, M. Texier, and J. Dean, “Carbon emissions and large neural network training”, *ArXiv*, vol. abs/2104.10350, 2021.

[6] J. Barr, “Amazon ec2 update – inf1 instances with aws inferentia chips for high

performance cost-effective inferencing", 2019, accessed on 8 September 2022. [Online]. Available: https://aws.amazon.com/fr/blogs/aws/amazon-ec2-update-inf1-instances-with-aws-inferentia-chips-for-high-performance-cost-effective-inferencing/

[7] G. Leopold, "Aws to offer nvidia's t4 gpus for ai inferencing", 2019, accessed on 4 October 2022. [Online]. Available: https://www.hpcwire.com/2019/03/19/aws-upgrades-its-gpu-backed-ai-inference-platform/

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey", *Proceedings of the IEEE*, 2017.

[9] R. Mishra, H. P. Gupta, and T. Dutta, "A survey on deep neural network compression: Challenges, overview, and solutions", *arXiv*, vol. abs/2010.03954, 2020.

[10] D. W. Blalock, J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *ArXiv*, vol. abs/2003.03033, 2020.

[11] G. Boole, *Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities*, 1853.

[12] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *The Bulletin of Mathematical Biophysics*, 1943.

[13] A. M. Turing, "Computing machinery and intelligence", *Mind*, 1950.

[14] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, 1958.

[15] G. Cybenko, "Approximation by superpositions of a sigmoidal function", *Mathematics of Control, Signals, and Systems, MCSS*, 1989.

[16] K. Hornik, "Approximation capabilities of multilayer feedforward networks", *Neural Networks*, 1991.

[17] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay", *ArXiv*, vol. abs/1803.09820, 2018.

[18] D. H. Hubel and T. N. Wiesel, "Receptive fields of single neurons in the cat's striate cortex", *Journal of Physiology*, 1959.

[19] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied

to document recognition", in *Proceedings of the IEEE*, 1998.

[20] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks", *The Journal of Machine Learning Research, JMLR*, 2021.

[21] L. Ilya and H. Frank, "Decoupled weight decay regularization", in *International Conference on Learning Representations, ICLR*, 2019.

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *Journal of Machine Learning Research, JMLR*, 2014.

[23] T. Devries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout", *arXiv*, vol. abs/1708.04552, 2017.

[24] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "Dropblock: A regularization method for convolutional networks", in *Advances in Neural Information Processing Systems, NeurIPS*, 2018.

[25] C. Bucilua, R. Caruana, and A. Niculescu-Mizil, "Model compression", in *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining, SIGKDD*, 2006.

[26] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network", in *Advances in Neural Information Processing Systems Workshops, NeurIPS*, 2015.

[27] TensorFlow Model Optimization team, "Quantization aware training with tensorflow model optimization toolkit - performance with accuracy", 2020, accessed on 5 October 2022. [Online]. Available: https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html

[28] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization", *ArXiv*, vol. abs/2106.08295, 2021.

[29] Fei-Fei Li and Jiajun Wu and Ruohan Gao, "CS231n Convolutional Neural Networks for Visual Recognition", 2022, accessed on 23 July 2022. [Online]. Available: https://cs231n.github.io/neural-networks-3/

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

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

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

[33] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in *Proceedings of the 32nd International Conference on Machine Learning, ICML*, 2015.

[34] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of tricks for image classification with convolutional neural networks", 2019.

[35] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. S. Dickstein, "On the expressive power of deep neural networks", in *Proceedings of the International Conference on Machine Learning, ICML*, 2017.

[36] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation", *arXiv*, vol. abs/1801.04381, 2018.

[37] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, "Squeezenext: Hardware-aware neural network design", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshop, CVPR*, 2018.

[38] R. Shore, "Rethinking the brain: New insights into early development", *Families and Work Institute*, 1997.

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

[40] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. Dally, "Exploring the regularity of sparse structure in convolutional neural networks", 2017.

[41] C. Rasmussen and Z. Ghahramani, "Occam's razor", in *Advances in Neural Information Processing Systems, NeurIPS*, 2000.

[42] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning (still) requires rethinking generalization", *in Communications of the Association for Computing Machinery, ACM*, 2021.

[43] B. Neyshabur, R. Tomioka, and N. Srebro, "In search of the real inductive bias: On the role of implicit regularization in deep learning." in *International Conference on Learning Representations Workshop, ICLR*, 2015.

[44] B. Bartoldson, A. Morcos, A. Barbu, and G. Erlebacher, "The generalization-stability tradeoff in neural network pruning", in *Advances in Neural Information Processing Systems, NeurIPS*, 2020.

[45] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima", in *International Conference on Learning Representations, ICLR*, 2017.

[46] S. Hochreiter and J. Schmidhuber, "Flat Minima", *Neural Computation*, 1997.

[47] J. Rissanen, "A Universal Prior for Integers and Estimation by Minimum Description Length", *The Annals of Statistics*, 1983.

[48] P. D. Grünwald and P. M. B. Vitányi, "Kolmogorov complexity and information theory. with an interpretation in terms of questions and answers", *Journal of Logic, Language and Information*, 2003.

[49] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks", in *in European Conference of Computer Vision, ECCV*, 2014.

[50] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. Graf, "Pruning filters for efficient convnets", in *International Conference on Learning Representations, ICLR*, 2017.

[51] V. Sanh, T. Wolf, and A. Rush, "Movement pruning: Adaptive sparsity by fine-tuning", in *Advances in Neural Information Processing Systems, NeurIPS*, 2020.

[52] B. Hassibi, D. G.Stork, and G. Wolff, "Optimal brain surgeon and general network pruning", in *in IEEE International Conference on Neural Networks, ICNN*, 1993.

[53] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage", in *Advances*

in Neural Information Processing Systems, NeurIPS, 1990.

[54] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning", 2019.

[55] N. Lee, T. Ajanthan, and P. H. S. Torr, "Snip: single-shot network pruning based on connection sensitivity", in *International Conference on Learning Representations, ICLR*, 2019.

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

[57] J. Frankle, G. K. Dziugaite, D. Roy, and M. Carbin, "Linear mode connectivity and the lottery ticket hypothesis", in *International Conference on Machine Learning, ICML*, 2020.

[58] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow", in *International Conference on Learning Representations, ICLR*, 2020.

[59] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression", in *in International Conference on Learning Representations, ICLR*, 2018.

[60] N. Hubens, "FasterAI: Prune and Distill your models with FastAI and PyTorch", 2020. [Online]. Available: https://github.com/nathanhubens/fasterai

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

[62] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, B. Nell, N. Shavit, and D. Alistarh, "Inducing and exploiting activation sparsity for fast inference on deep neural networks", in *International Conference on Machine Learning, ICML*, 2020.

[63] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A python package for DNN compression research", vol. abs/1910.12232,

2019.

[64] J. Howard and S. Gugger, "Fastai: A layered api for deep learning", *Information*, 2020.

[65] W. Falcon and The PyTorch Lightning team, "PyTorch Lightning", https://github.com/PyTorchLightning/pytorch-lightning, 2019.

[66] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need", in *Advances in Neural Information Processing Systems, NeurIPS*, 2017.

[67] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", 2016.

[68] R. F. L. Fei-Fei and P. Perona, "Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories", 2003.

[69] N. Hubens, M. Mancas, B. Gosselin, M. Preda, and T. Zaharia, "One-cycle pruning: Pruning convnets under a tight training budget", *in International Conference on Image Processing, ICIP*, 2022.

[70] J. Frankle, G. K. Dziugaite, D. Roy, and M. Carbin, "Pruning neural networks at initialization: Why are we missing the mark?" in *International Conference on Learning Representations, ICLR*, 2021.

[71] S. H. et al., "Dsd: Dense-sparse-dense training for deep neural networks", in *International Conference on Learning Representations, ICLR*, 2017.

[72] J. Frankle, D. J. Schwab, and A. S. Morcos, "The early phase of neural network training", in *International Conference on Learning Representations, ICLR*, 2020.

[73] Y. LeCun and C. Cortes, "MNIST handwritten digit database", 2005.

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

[75] I. T. Union, *Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios*, 2007.

[76] L. N. Smith and N. Topin, "Super-convergence: very fast training of neural networks using large learning rates", in *International Society for Optics and Photonics, SPIE*, 2019.

[77] H. Touvron, A. Vedaldi, M. Douze, and H. Jegou, "Fixing the train-test resolution discrepancy", in *Advances in Neural Information Processing Systems, NeurIPS*, 2019.

[78] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, 2009.

[79] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning", 2011.

[80] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks", in *International Conference on Machine Learning, ICML*, 2019.

[81] D. Erhan, Y. Bengio, A. Courville, and P. Vincent, "Visualizing higher-layer features of a deep network", *Technical Report, Univeristé de Montréal*, 01 2009.

[82] C. Olah, A. Mordvintsev, and L. Schubert, "Feature visualization", *Distill*, 2017.

[83] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning", in *International Conference on Learning Representations, ICLR*, 2019.

[84] A. Achille, M. Rovere, and S. Soatto, "Critical learning periods in deep networks", in *International Conference on Learning Representations, ICLR*, 2019.

[85] A. Golatkar, A. Achille, and S. Soatto, "Time matters in regularizing deep networks: Weight decay and data augmentation affect early learning dynamics, matter little near convergence", in *Advances in Neural Information Processing Systems, NeurIPS*, 2019.

[86] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks", in *Advances in Neural Information Processing Systems, NeurIPS*, 2015.

[87] S. Vosoughi, D. Roy, and S. Aral, "The spread of true and false news online", *Science*, 2018.

[88] S. Tariq, S. Lee, H. Kim, Y. Shin, and S. S. Woo, "Detecting both machine and human created fake face images in the wild", in *Proceedings of the International Workshop on Multimedia Privacy and Security, MPS*, 2018.

[89] D. Afchar, V. Nozick, J. Yamagishi, and I. Echizen, “Mesonet: a compact facial video forgery detection network”, in *International Workshop on Information Forensics and Security (WIFS)*, 2018.

[90] Y. Li and S. Lyu, “Exposing deepfake videos by detecting face warping artifacts”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Workshop CVPR*, 2019.

[91] N. Yu, L. Davis, and M. Fritz, “Attributing fake images to gans: Learning and analyzing gan fingerprints”, *International Conference on Computer Vision, ICCV*, 2019.

[92] J. Pu, N. Mangaokar, B. Wang, C. K. Reddy, and B. Viswanath, “Noisescope: Detecting deepfake images in a blind setting”, in *Annual Computer Security Applications Conference, ACSAC*, 2020.

[93] I. Amerini, L. Galteri, R. Caldelli, and A. Bimbo, “Deepfake video detection through optical flow based cnn”, *International Conference on Computer Vision Workshop, ICCVW*, 2019.

[94] D. Güera and E. J. Delp, “Deepfake video detection using recurrent neural networks”, in *IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS*, 2018.

[95] E. Sabir, J. Cheng, A. Jaiswal, W. AbdAlmageed, I. Masi, and P. Natarajan, “Recurrent convolutional strategies for face manipulation detection in videos”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshop, CVPR*, 2019.

[96] S.-Y. Wang, O. Wang, A. Owens, R. Zhang, and A. A. Efros, “Detecting photoshopped faces by scripting photoshop”, *IEEE International Conference on Computer Vision, ICCV*, 2019.

[97] L. Yuezun, C. Ming-Ching, and L. Siwei, “In ictu oculi: Exposing ai created fake videos by detecting eye blinking”, in *in IEEE International Workshop on Information Forensics and Security, WIFS*, 2018.

[98] H. Qi, Q. Guo, F. Juefei-Xu, X. Xie, L. Ma, W. Feng, Y. Liu, and J. Zhao, “Deeprhythm: Exposing deepfakes with attentional visual heartbeat rhythms”, *ACM International Conference on Multimedia, MM*, 2020.

[99] Y. Wang and A. Dantcheva, "A video is worth more than 1000 lies. comparing 3dcnn approaches for detecting deepfakes", *2020 15th IEEE International Conference on Automatic Face and Gesture Recognition, FG*, 2020.

[100] D. M. Montserrat, H. Hao, S. K. Yarlagadda, S. Baireddy, R. Shao, J. Horvath, E. Bartusiak, J. Yang, D. Guera, F. Zhu, and E. J. Delp, "Deepfakes detection with automatic face weighting", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, 2020.

[101] B. Dolhansky, J. Bitton, B. Pflaum, J. Lu, R. Howes, M. Wang, and C. Canton-Ferrer, "The deepfake detection challenge dataset", *arXiv*, vol. abs/2006.07397, 2020.

[102] Y. Li, P. Sun, H. Qi, and S. Lyu, "Celeb-DF: A Large-scale Challenging Dataset for DeepFake Forensics", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, 2020.

[103] D. Huang and F. De La Torre, "Facial action transfer with personalized bilinear regression", in *European Conference for Computer Vision, ECCV*, 2012.

[104] E. Zakharov, A. Shysheya, E. Burkov, and V. Lempitsky, "Few-shot adversarial learning of realistic neural talking head models", *International Conference on Computer Vision, ICCV*, 2019.

[105] Y. Nirkin, Y. Keller, and T. Hassner, "Fsgan: Subject agnostic face swapping and reenactment", *International Conference on Computer Vision, ICCV*, 2019.

[106] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks", 2019.

[107] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks", *IEEE Signal Processing Letters*, 2016.

[108] V. Bazarevsky, Y. Kartynnik, A. Vakunov, K. Raveendran, and M. Grundmann, "BlazeFace: Sub-Millisecond Neural Face Detection on Mobile GPUs", *arXiv*, vol. abs/1907.05047, 2019.

[109] J. Li, Y. Wang, C. Wang, Y. Tai, J. Qian, J. Yang, C. Wang, J. Li, and F. Huang, "DSFD: Dual Shot Face Detector", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, 2019.

[110] J. Deng, J. Guo, E. Ververas, I. Kotsia, and S. Zafeiriou, “RetinaFace: Single-Shot Multi-Level Face Localisation in the Wild”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, 2020.

[111] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, “Self-attention generative adversarial networks”, in *Proceedings of the International Conference on Machine Learning, ICML*, 2019.

[112] M. Tan and Q. V. Le, “EfficientNetV2: Smaller Models and Faster Training”, in *Proceedings of the International Conference on Machine Learning, ICML*, 2021.

[113] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning”, in *Proceedings of International Conference on Machine Learning, ICML*, 2009.

[114] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, “Mixup: Beyond Empirical Risk Minimization”, in *International Conference on Learning Representations, ICLR*, 2018.

[115] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, “Mixed precision training”, in *International Conference on Learning Representations (ICLR)*, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ

[116] M. Tan and Q. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks”, in *International Conference on Machine Learning, ICML*, 2019.

[117] F. Chollet, “Xception: Deep learning with depthwise separable convolutions”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, 2017.

[118] S. Seferbekov, “Dfdc solution”, in *Kaggle*, 2020. [Online]. Available: https://github.com/selimsef/dfdc_deepfake_challenge

[119] H. Zhao, H. Cui, , and W. Zhou, “Dfdc solution”, in *Kaggle*, 2020. [Online]. Available: https://github.com/cuihaoleo/kaggle-dfdc.

[120] J. Thies, M. Zollhöfer, M. Stamminger, C. Theobalt, and M. Nießner, “Face2face: real-time face capture and reenactment of RGB videos”, *in Communications of the Association for Computing Machinery, ACM*, 2019.

[121] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and

C. Willing, “Jupyter notebooks - a publishing format for reproducible computational workflows”, in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016.

[122] The Voilà Development Team, “Voilà”, 2020, accessed on 15 May 2021. [Online]. Available: https://github.com/voila-dashboards/voila

[123] The Binder Team, “Binder”, 2017, accessed on 15 May 2021. [Online]. Available: https://github.com/jupyterhub/binderhub

[124] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture”, *in Communications of the Association for Computing Machinery, ACM*, 2019.

[125] J. Dean, “The deep learning revolution and its implications for computer architecture and chip design”, *in IEEE International Solid- State Circuits Conference, ISSCC*, 2020.

[126] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy, “Nvidia ampere architecture in-depth”, 2020, accessed on 4 October 2022. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[127] F. Sun, M. Qin, T. Zhang, L. Liu, Y.-K. Chen, and Y. Xie, “Computation on sparse neural networks and its implications for future hardware”, in *Proceedings of the ACM/EDAC/IEEE Design Automation Conference, DAC*, 2020.

[128] S. Hooker, “The hardware lottery”, *in Communications of the Association for Computing Machinery, ACM*, 2021.

[129] P. Barham and M. Isard, “Machine Learning Systems Are Stuck in a Rut”, in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019.

[130] G. Marcus, “The new science of alt intelligence”, 2022, accessed on 4 October 2022. [Online]. Available: https://garymarcus.substack.com/p/the-new-science-of-alt-intelligence

List of Figures

0.1 Evolution of the amount of parameters present in Deep Learning architectures. 4

1.1 Representation of a 4-layer MLP. 12

1.2 The representation of a single artificial neuron, also called *perceptron*. Each of the 3 inputs is associated with a weight. The perceptron computes the weighted sum of the inputs, adds in the bias, and sends the results through an activation function. 13

1.3 Examples of impact of learning rate values on convergence of SGD. . . 16

1.4 Learning Rate Range Test. 16

1.5 Representation of the LeNet-5 CNN [19]. 18

1.6 Application of the convolution filter to an input image. 19

1.7 Result of a max-pooling operation with a 2×2 kernel and stride of 2 . 20

2.1 Pareto Optimality of compressed model. 24

2.2 Convolution as designed sparsity. 27

2.3 Convolutional Neural Network extracting data features in a hierarchical manner, illustrating the compositionality property of input data. 28
2.4 Representation of the solution imposed by different types of norms. . . 29
2.5 Pruning connections in a neural network. 30
2.6 Growing connections in a neural network. 31
2.7 Illustration of the Dropout technique, introducing sparsity in the neurons. 32
2.8 Illustration of the DropConnect technique, introducing sparsity in the connections. 32
2.9 Illustration of the DropConnect technique, introducing sparsity in the activations. 33
2.10 Illustration of the Knowledge Distillation Concept. 34
2.11 Mapping of a small range of floating-point format `float32` to a fixed-point format `int8`. 35
2.12 A visual illustration of the different uniform quantization grids for int8 representation. 37
2.13 Illustration of the Post-Training Quantization process. 38
2.14 Illustration of the Quantization-Aware Training process. 39
2.15 Illustration of the Automatic Mixed-Precision process. 39
2.16 Representation of the effect of applying a regular convolution on an input image of three channels. 41
2.17 Representation of the effect of applying a depthwise convolution on an input image of three channels. 41
2.18 Representation of the effect of applying a pointwise convolution on an input image of three channels. 42
2.19 Truncated-SVD decomposition of a fully-connected layer 44
2.20 Receptive field of a pixel from the second layer, when using 3×3 filters. 47
2.21 The Inverted Residual module, proposed by MobileNetV2. 48
2.22 The Fire module proposed by SqueezeNet 49

3.1 Evolution of the synaptic density in the human brain during early life. 52

3.2 Schematic representation of evolution of accuracy of trained models according to pruning level 53

3.3 Conceptual representation of flat and sharp minima of the loss function 54

3.4 Representation of C_o filters, composed of C_i kernels of $K_h \times K_w$ weights each, and constituting the layer of a CNN. 57

3.5 Representation of weight pruning. 58

3.6 Variations of vector pruning in a convolutional layer. 58

3.7 Representation of Shared-Weight Pruning. Although it appears to be unstructured at the scale of an individual filter, the same sparsity pattern is shared across all filters of the layer. Removed weights are in color. . . 59

3.8 Variations of slice-level pruning in a convolutional layer. 59

3.9 Variations of shared-vector pruning in a convolutional layer. 60

3.10 Filter Pruning. 60

3.11 Variations of shared-slice pruning in a convolutional layer. 61

3.12 Filter and shared-kernel granularities allow to remove parts from the model. 62

3.13 Effect of local and global pruning on the individual sparsity level of a 20-layer neural network. 63

3.14 Result of the application of the Random criteria on the weights. Parameters that are greyed out are the parameters removed. 64

3.15 Result of the application of the Large Final criteria on the weights. Parameters that are greyed out are the parameters removed. 65

3.16 Result of the application of the Large Initialization criteria on the weights. Parameters that are greyed out are the parameters removed. 66

3.17 Result of the application of the Large Initialization Large Final criteria on the weights. Parameters that are greyed out are the parameters removed. 66

3.18 Result of the application of the Magnitude Increase criteria on the weights. Parameters that are greyed out are the parameters removed. 67

3.19 Result of the application of the Movement criteria on the weights. Parameters that are greyed out are the parameters removed. 68

3.20 Lottery Ticket Hypothesis with Rewinding 70

3.21 Evolution of sparsity for the One-Shot pruning schedule., applied at different training times. 72

3.22 Evolution of sparsity for the Iterative pruning schedule, applied at different training times. 72

3.23 Evolution of sparsity for the Iterative pruning schedule, applied at different training times. 73

4.1 Result of the application of our proposed criteria on the weights. Parameters that are greyed out are the parameters removed. 83

4.2 Evolution of sparsity along training for the available pruning schedules. While the `sched_func` parameters defines the general evolution, the schedule can further be customized by modifying the `start_pct` and `end_pct` values. 85

4.3 Variation of the `dsd` schedule. The use of `start_epoch` and `end_epoch` help to further customize a given pruning schedule by affecting the start and end of the pruning process. 87

5.1 Different types of sparsity in a fully-connected weight matrices. Zeroed weights are in color. 97

5.2 Pruning rows in layer i impacts its output and consequently the columns of layer $i + 1$. Zeroed weights are in color. 98

5.3 Different types of structured granularities for CNNs. Those granularities allow to remove parts of the network, leading to a smaller architecture. 99

5.4 Pixels removed after column pruning. Colored pixels can be removed from all input images, effectively reducing the storage need of the dataset.101

5.5 Remaining channels after performing shared-kernel pruning. The blue channel is the first to be removed, and the red one is the second. Removed channels are greyed out. 104

5.6 Difference of weight distributions when the network is fine-tuned compared to when it is trained from-scratch for the first convolutional layer of ResNet-18 trained on CIFAR-10. 107

5.7 Visualization of the sensitivity to pruning of a VGG-16 trained on MNIST. Layers with most low-norm filters are those that are less sensitive to pruning. 109

5.8 The proposed pruning pipeline. We introduce a fourth step in the common iterative pruning process, aiming to evaluate the sensivity of each layer to pruning. By then performing pruning in the least sensitive layer, we ensure to reduce the number of parameter contained in the network without affecting the performance too much. 109

5.9 Visualization of the importance of filters of VGG-16 trained on CIFAR-10. The deep layers of FT-Network possesses an important fraction of low-norms filters, whereas FS-Network has filters of balanced norms. . 111

5.10 Repartition of sparsities in the layers of VGG-16 trained on CIFAR-10. 112

5.11 Visualization of the 64 filters in the first convolutional layer of VGG-16 trained on CIFAR-10. FT-Network filters have more structure than those of FS-Network. 113

5.12 Visualization of the importance of filters of MobileNet trained on CIFAR-10. The deep layers of FT-Network possesses an important fraction of low-norms filters, whereas FS-Network has filters of balanced norms. . . 114

5.13 Repartition of sparsities in the layers of MobileNet trained on CIFAR-10 114

5.14 Visualization of the 32 filters in the first convolutional layer of MobileNet trained on CIFAR-10. Filters of FT-Network exhibit more structure than the filters of FS-Network. 115

5.15 Illustration of the Activation Maximization technique. 117

5.16 The proposed pruning pipeline. We introduce a fourth step in the common iterative pruning process, aiming to cluster convolution filters by similar functionality. By then performing pruning in each cluster, we ensure that we remove redundant filters, while preserving the rare ones. 118

5.17 Representation of the clustering process. We first generate the feature image corresponding to each filter with the Activation Maximization technique. Those synthesized images are then encoded to a lower dimension by a pre-trained CNN, and clustered with K-Means, allowing to group filters sensitive to similar features together. 118

5.18 Comparison of the remaining features after applying different pruning techniques until a sparsity of 50% in the first layer of AlexNet. Three dominant clusters are highlighted in color. Features removed are greyed out. 119

5.19 Results of the Lottery Ticket Hypothesis with Rewind test for different sparsities, performed with ResNet-18 on CIFAR-10. 123

5.20 Results of the Lottery Ticket Hypothesis with Rewind test for different sparsities, performed with ResNet-18 on CIFAR-100. 123

5.21 Results of the Lottery Ticket Hypothesis with Rewind test for different sparsities, performed with ResNet-18 on CALTECH-101. 123

5.22 Visualization of the variation of the scheduling for different α and β values. 125

5.23 Comparison of the evolution of sparsity during training of the 4 studied pruning schedules. 127

5.24 Evolution of accuracy of ResNet18 trained on CIFAR10, when applying different pruning schedules to a sparsity of 95% 129

5.25 Evolution of instability error after each round of the Lottery Ticket Hypothesis when using a different pruning schedule 131

6.1 Examples of manipulated faces present in the DFDC dataset. Different deception methods are used to generate the faces. 136

6.2 The general framework of our proposed solution. It takes a single frame (an image) or several frames (a video) as an input, that are fed to a face detection network, to extract the different faces that are present. Those faces are then used as the input of our DeepFake detection network, yielding the final prediction. 137

6.3 The proposed crop extraction method for different image formats. . . . 138

6.4 The general framework of our proposed face extraction method. In case no face is found with our main face extractor, a more accurate second one is used as a fallback. 139

6.5 DFBlock, the building block of DFBuster. It consists of an inverted residual module, a squeeze-excite (SE) module. 140

6.6 Our proposed DeepFake classification network: DFBuster. It consists of a 3-convolutional layers stem, and 4 double DFBlocks. A Self-Attention (SA) module is also introduced to further improve classification performance. 141

6.7 Illustration of the augmentations applied to training data. The transforms have been chosen to create a model robust to organic videos that it might encounter in real-world settings. 141

6.8 Examples of extracted faces and their corresponding attention map. Face parts that are the most useful for the final prediction are highlighted. . 145

6.9 The DeepFake Buster application, running in a web browser. 146

6.10 Example of usage of the web-based application. An image extracted from a forged video is uploaded and fed to the model, which predicts it to be false and provides an attention map, allowing for a better interpretation of results. 147

6.11 Example of usage of the web-based application. A forged video is uploaded and fed to the model, which first extracts faces from each frames, and then makes its prediction. It also provides the corresponding attention maps, allowing for a better interpretation of results. 148

List of Pseudo-Code

4.1 The two ways of sparsifying a model. The `static` is done offline, disconnected from training, while the `dynamic` is performed during training, allowing the model to recover from lost performance. 78

4.2 Slicing of 4D weight tensor of dimension `[O, I, Kh, Kw]` to extract the desired granularity. 79

4.3 Representation of `local` sparsification, performed in each layer independently and `global` sparsification, performed on all the layers. 80

4.4 The list of all criteria available in `FasterAI` and their corresponding PyTorch implementation. 82

4.5 Custom criteria and their corresponding implementation in PyTorch. . . 82

4.6 Available schedules in `FasterAI` and their PyTorch implementation. . . . 84

4.7 Implementation of the Dense-Sparse-Dense technique in `FasterAI`. . . . 86

4.8 Changes to `SparsifyCallback` in order to perform Lottery Tickets Experiments. 88

4.9 Code required to prune a filter-sparse model with `FasterAI`. 89

4.10 Code required to perform Knowledge Distillation in FasterAI. 89
4.11 Code required to perform Group Regularization in FasterAI. 90
4.12 Code required to perform Batch Normalization Folding in FasterAI. . . . 92
4.13 Code required to perform Knowledge Distillation in FasterAI. 93

5.1 Granularity selection for a Fully-Connected Layer. 97
5.2 Filter and Shared-Kernel Selection. 99
5.3 Performing sensitivity analysis with FasterAI. 108
5.4 One-Cycle Pruning 126

6.1 Implementation of the proposed cropping method. 138

List of Tables

2.1 The repartition of parameter in each part of a CNN, designed for ImageNet-1K classification. 43

2.2 The repartition of parameter in each part of an already parameter-efficient CNN, designed for ImageNet-1K classification. Because the feature extraction part is parameter-efficient, the fully-connected layer represent a large part of the parameter count. 44

4.1 Results of sparsifying ResNet-18 for all available granularities. Context, criteria and schedule are respectively set to `local`, `large_final` and `one_cycle`. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline. 80

4.2 Results of sparsifying ResNet-18 for all available granularities. Context, criteria and schedule are respectively set to `global`, `large_final` and `one_cycle`. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline. 81

4.3 Results of sparsifying ResNet-18 for all available criteria. Granularity, context and schedule are respectively set to `weight`, `local` and `one_cycle`. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline. 83

4.4 Results of sparsifying ResNet-18 for all available criteria. Granularity, context and criteria are respectively set to `weight`, `local` and `large_final`. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline. 86

4.5 Results of pruning ResNet-18 and VGG-16 with 4 different schedules. Mean and standard deviation of accuracy over 3 rounds are reported. . . 88

4.6 Results of applying Knowledge Distillaion from a ResNet34 to a ResNet18 architecture for different interpolation values of β. Mean and standard deviation of accuracy over 3 rounds are reported. 90

4.7 Results of regularizing ResNet-18 with four different penalty strengths. Mean and standard deviation of accuracy over three rounds are reported. The darker the shade of red, the further the accuracy is from the baseline. 91

4.8 Results of performing batch normalization folding on ResNet-18. Mean and standard deviation of accuracy over three rounds are reported. . . 92

4.9 Results of decomposing the FCLs of ResNet-18 with 3 different compression levels. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline. 93

5.1 Pruning parameters used in `FasterAI`. 100

5.2 Accuracies of MLP model for three different datasets and for three levels of sparsity. Reported values are mean and standard deviation over 5 iterations 101

5.3 Pruning parameters used in `FasterAI`. 102

5.4 Comparison of accuracies of ResNet-18 trained on three different datasets and for four levels of sparsity. Reported values are mean and standard deviation for three iterations. 103

5.5 Comparison of remaining #Params(1e6)/FLOPs(1e9) for ResNet-18 trained on different datasets for four sparsity levels. 105

5.6 Pruning parameters used in `FasterAI`. 110

5.7 Results of the pruning on VGG-16 for the four studied datasets. FT-Network leads to smaller networks, while FS-Network leads to faster ones, due to the location of removed filters in the network. 112

5.8 Results of the pruning on MobileNet for the four studied datasets. FT-Network leads to smaller network while not necessarily the fastest ones, due to the location of removed filters in the network. 115

5.9 Pruning parameters used in `FasterAI`. 120

5.10 Results of applying different pruning criteria on VGG-16. The benefit of applying our clustering method before selecting the filters to remove translates to a higher accuracy for most sparsity levels and datasets. Values in bold are the best when comparing a criteria with and without the clustering process. Accuracies and standard deviation over 3 runs are reported. 121

5.11 Results of applying different pruning criteria on ResNet-18. The benefit of applying our clustering method before selecting the filters to remove translates to a higher accuracy for most sparsity levels and datasets. Values in bold are the best when comparing a criteria with and without the clustering process. Accuracies and standard deviation over 3 runs are reported. 122

5.12 Grid search of α and β for Resnet-18 trained on CIFAR-10 to for 90% sparsity. Mean and standard deviation over 3 rounds are reported. . . . 126

5.13 Pruning parameters used in `FasterAI`. 126

5.14 Results of pruning ResNet-18 and VGG-16 with 4 different schedules. Mean and standard deviation of accuracy over 3 rounds are reported. Best results are in bold. 128

5.15 Training budget required to prune ResNet-18 to 95% to achieve a fixed validation accuracy. 129

5.16 Validation accuracy of the reinitialized ResNet-18 sub-networks found by several pruning schedules at their last pruning round, *i.e.* at a sparsity level of 95% 130

6.1 Characteristics of Face Detection models. Speed is measured on a Nvidia GTX 1080 GPU. 137

6.2 Characteristics of Face Detection models. Speed is measured on a Nvidia GTX 1080 GPU 139

6.3 Characteristics of Face Detection models. 140

6.4 Pruning parameters used in `FasterAI`. 142

6.5 Ablation Study of the proposed model. 143

6.6 Comparison to the state-of-the-art DeepFake classification methods. The Inference Time is computed on single image, using a Nvidia GTX 1080 and averaged over 10 iterations. 144

ECOLE
DOCTORALE

Titre : Compression et accélération de réseaux de neurones profonds par élagage synaptique

Mots clés : Élagage de réseaux de neurones, Compression de réseaux de neurones, Apprentissage profond.

Résumé : Depuis leur résurgence en 2012, les réseaux de neurones profonds sont devenus omniprésents dans la plupart des disciplines de l'intelligence artificielle, comme la reconnaissance d'images, le traitement de la parole et le traitement du langage naturel. Cependant, au cours des dernières années, les réseaux de neurones sont devenus exponentiellement profonds, faisant intervenir de plus en plus de paramètres. Aujourd'hui, il n'est pas rare de rencontrer des architectures impliquant plusieurs milliards de paramètres, alors qu'elles en contenaient le plus souvent des milliers il y a moins de dix ans.

Cette augmentation généralisée du nombre de paramètres rend ces grands modèles gourmands en ressources informatiques et essentiellement inefficaces sur le plan énergétique. Cela rend les modèles déployés coûteux à maintenir, mais aussi leur utilisation dans des environnements limités en ressources très difficile.

Pour ces raisons, de nombreuses recherches ont été menées pour proposer des techniques permettant de réduire la quantité de stockage et de calcul requise par les réseaux neuronaux. Parmi ces techniques, l'élagage synaptique, consistant à créer des modèles réduits, a récemment été mis en évidence. Cependant, bien que l'élagage soit une technique de compression courante, il n'existe actuellement aucune méthode standard pour mettre en œuvre ou évaluer les nouvelles méthodes, rendant la comparaison avec les recherches précédentes difficile.

Notre première contribution concerne donc une description inédite des techniques d'élagage, développée selon quatre axes, et permettant de définir de manière univoque et complète les méthodes existantes. Ces composantes sont : la granularité, le contexte, les critères et le programme. Cette nouvelle définition du problème de l'élagage nous permet de le subdiviser en quatre sous-problèmes indépendants et de mieux déterminer les axes de recherche potentiels.

De plus, les méthodes d'élagage en sont encore à un stade de développement précoce et principalement destinées aux chercheurs, rendant difficile pour les novices d'appliquer ces techniques. Pour combler cette lacune, nous avons proposé l'outil FasterAI, destiné aux chercheurs, désireux de créer et d'expérimenter différentes techniques de compression, mais aussi aux nouveaux venus, souhaitant compresser leurs modèles pour des applications concrètes. Cet outil a de plus été construit selon les quatre composantes précédemment définis, permettant une correspondance aisée entre les idées de recherche et leur mise en œuvre.

Nous proposons ensuite quatre contributions théoriques, chacune visant à fournir de nouvelles perspectives et à améliorer les méthodes actuelles dans chacun des quatre axes de description identifiés. De plus, ces contributions ont été réalisées en utilisant l'outil précédemment développé, validant ainsi son utilité scientifique.

Enfin, afin de démontrer que l'outil développé, ainsi que les différentes contributions scientifiques proposées, peuvent être applicables à un problème complexe et réel, nous avons sélectionné un cas d'utilisation : la détection de la manipulation faciale, également appelée détection de DeepFakes. Cette dernière contribution est accompagnée d'une application de preuve de concept, permettant à quiconque de réaliser la détection sur une image ou une vidéo de son choix.

L'ère actuelle du Deep Learning a émergé grâce aux améliorations considérables des puissances de calcul et à l'accès à une grande quantité de données. Cependant, depuis le déclin de la loi de Moore, les experts suggèrent que nous pourrions observer un changement dans la façon dont nous concevons les ressources de calcul, conduisant ainsi à une nouvelle ère de collaboration entre les communautés du logiciel, du matériel et de l'apprentissage automatique. Cette nouvelle quête de plus d'efficacité passera donc indéniablement par les différentes techniques de compression des réseaux neuronaux, et notamment les techniques d'élagage.

Title : Towards Lighter and Faster Deep Neural Networks with Parameter Pruning

Keywords : Neural Network Pruning, Neural Network Compression, Deep Learning.

Abstract : Since their resurgence in 2012, Deep Neural Networks have become ubiquitous in most disciplines of Artificial Intelligence, such as image recognition, speech processing, and Natural Language Processing. However, over the last few years, neural networks have grown exponentially deeper, involving more and more parameters. Nowadays, it is not unusual to encounter architectures involving several billions of parameters, while they mostly contained thousands less than ten years ago.

This generalized increase in the number of parameters makes such large models compute-intensive and essentially energy inefficient. This makes deployed models costly to maintain but also their use in resource-constrained environments very challenging.

For these reasons, much research has been conducted to provide techniques reducing the amount of storage and computing required by neural networks. Among those techniques, neural network pruning, consisting in creating sparsely connected models, has been recently at the forefront of research. However, although pruning is a prevalent compression technique, there is currently no standard way of implementing or evaluating novel pruning techniques, making the comparison with previous research challenging.

Our first contribution thus concerns a novel description of pruning techniques, developed according to four axes, and allowing us to unequivocally and completely define currently existing pruning techniques. Those components are: the granularity, the context, the criteria, and the schedule. Defining the pruning problem according to those components allows us to subdivide the problem into four mostly independent subproblems and also to better determine potential research lines.

Moreover, pruning methods are still in an early development stage, and primarily designed for the research community. Indeed, most pruning works are usually implemented in a self-contained and sophisticated way, making it troublesome for non-researchers to apply such techniques without having to learn all the intricacies of the field. To fill this gap, we proposed FasterAI toolbox, intended to be helpful to researchers, eager to create and experiment with different compression techniques, but also to newcomers, that desire to compress their neural network for concrete applications. In particular, the sparsification capabilities of FasterAI have been built according to the previously defined pruning components, allowing for a seamless mapping between research ideas and their implementation.

We then propose four theoretical contributions, each one aiming at providing new insights and improving on state-of-the-art methods in each of the four identified description axes. Also, those contributions have been realized by using the previously developed toolbox, thus validating its scientific utility.

Finally, to validate the applicative character of the pruning technique, we have selected a use case: the detection of facial manipulation, also called DeepFakes Detection. The goal is to demonstrate that the developed tool, as well as the different proposed scientific contributions, can be applicable to a complex and actual problem. This last contribution is accompanied by a proof-of-concept application, providing DeepFake detection capabilities in a web-based environment, thus allowing anyone to perform detection on an image or video of their choice.

This Deep Learning era has emerged thanks to the considerable improvements in high-performance hardware and access to a large amount of data. However, since the decline of Moore's Law, experts are suggesting that we might observe a shift in how we conceptualize the hardware, by going from task-agnostic to domain-specialized computations, thus leading to a new era of collaboration between software, hardware, and machine learning communities. This new quest for more efficiency will thus undeniably go through neural network compression techniques, and particularly sparse computations.

Institut Polytechnique de Paris
91120 Palaiseau, France