THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité - **Informatique**

Informatique, Télécommunication et Électronique (Paris) - ED130

Deep Neural Network Compression for Visual Recognition

Compression de Réseaux de Neurones Profonds pour la Reconnaissance Visuelle

Présentée par

Robin Dupont

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Soutenue publiquement le 8 décembre 2023

Devant un jury composé de :

Mme	**Jenny Benois-Pineau** Professeure, Université de Bordeaux	*Rapportrice*
M.	**Titus Bogdan Zaharia** Professeur, Télécom SudParis	*Rapporteur*
M.	**Pierre Beauseroy** Professeur, Université de Technologie de Troyes	*Examinateur*
M.	**Nicolas Gac** Professeur, Université Paris-Saclay	*Examinateur*
M.	**Vincent Gripon** Professeur, IMT Atlantique	*Examinateur*
Mme	**Alice Lebois** Ingénieure, Netatmo	*Co-encadrante de thèse*
M.	**Hichem Sahbi** Chercheur CNRS (HDR), Sorbonne Université	*Directeur de thèse*

Abstract

Thanks to the miniaturisation of electronics, embedded devices have become more and more ubiquitous, since the 2010s, realising various tasks all around us. As their usage is developing, there is a growing demand for these devices to process data and make complex decisions efficiently. Deep neural networks are powerful tools to achieve this goal, however, these networks are often too heavy and complex to fit on embedded devices. Thus, there is a compelling need to devise methods to compress these large networks without significantly compromising their efficacy. This PhD thesis introduces two innovative methods, centred around the concept of pruning, aiming to compress neural networks while ensuring minimal impact on their accuracy.

This PhD thesis first introduces a budget-aware method for compressing large neural networks with weight reparametrisation and budget loss that does not require fine-tuning. Traditional pruning methods often rely on post-training saliency indicators to remove weights, disregarding the targeted pruning rate. Our approach integrates a budget loss, driving the pruning process towards a specific value during training, thereby achieving a joint optimisation of topology and weights. By soft-pruning the smallest weights using weight reparametrisation, our method significantly mitigates accuracy degradation in comparison to traditional pruning techniques. We show the effectiveness of our approach across various datasets and architectures.

This PhD thesis later focuses on the extraction of effective subnetworks without weight training. Our goal is to identify the best subnetwork topology in a large network without optimising its weights while still delivering compelling performance. This is achieved using our novel Arbitrarily Shifted Log Parametrisation, which serves as a differentiable relaxation of discrete topology sampling, enabling the training of masks that represent the probability of selection of the weights. Alongside, a weight rescaling mechanism (referred to as Smart Rescale) is also introduced, which allows

enhancing the performance of the extracted subnetworks as well as speeding up their training. Our proposed approach also finds the optimal pruning rate after one training pass, thereby circumventing computationally expensive gird-search and training across various pruning rates. As shown through comprehensive experiments, our method consistently outperforms closely related state-of-the-art techniques and allows designing lightweight networks which can reach high sparsity levels without significant loss in accuracy.

Résumé

Grâce à la miniaturisation de l'électronique, les dispositifs embarqués sont devenus de plus en plus omniprésents depuis les années 2010, réalisant diverses tâches tout autour de nous. À mesure que leur utilisation se développe, la demande pour des dispositifs traitant les données et prennant des décisions complexes de manière efficace augmente. Les réseaux de neurones profonds sont des outils puissants pour atteindre cet objectif, cependant, ces réseaux sont souvent trop lourds et complexes pour être intégrés dans des appareils embarqués. C'est pourquoi il est impératif de concevoir des méthodes pour compresser ces grands réseaux de neurones sans compromettre significativement leur performance. Cette thèse de doctorat introduit deux méthodes innovantes, centrées autour du concept d'élagage, visant à compresser les réseaux de neurones tout en assurant un impact minimal sur leur précision.

Cette thèse de doctorat introduit d'abord une méthode prenant en compte le budget pour compresser de grands réseaux de neurones à l'aide de reparamétrisation des poids et d'une fonction de coût budgétaire, le tout ne nécessitant pas de fine-tuning. Les méthodes d'élagage traditionnelles s'appuient souvent sur des indicateurs de saillance post-entraînement pour supprimer les poids, négligeant le taux d'élagage ciblé. Notre approche intègre une fonction de coût budgétaire, guidant le processus d'élagage vers une valeur spécifique de parcimonie pendant l'entraînement, réalisant ainsi une optimisation conjointe de la topologie et des poids. En simulant l'élaguage des poids les plus petits en cours d'entraînement grâce à la reparamétrisation des poids, notre méthode atténue significativement la perte de la précision par rapport aux techniques d'élagage traditionnelles. Nous démontrons l'efficacité de notre approche à travers divers ensembles de données et architectures.

Cette thèse de doctorat se concentre ensuite sur l'extraction de sous-réseaux efficaces, sans entraînement des poids. Notre objectif est d'identifier la meilleure topologie d'un sous-réseau dans un grand réseau sans en opti-

miser les poids tout en offrant des performances convaincantes. Ceci est réalisé grâce à notre méthode appelée Arbitrarily Shifted Log-Parametrisation, qui sert à échantillonner des topologies discrètes de manière différentiable, permettant l'entraînement de masques représentant la probabilité de sélection des poids. Parallèlement, un mécanisme de recalibrage des poids (appelé Smart Rescale) est également introduit, permettant d'améliorer les performances des sous-réseaux extraits ainsi que d'accélérer leur entraînement. Notre approche proposée trouve également le taux d'élagage optimal après un unique entraînement, évitant ainsi la recherche exhaustive d'hyperparamètres et un entraînement pour chaque taux d'élagage. Nous montrons à travers un ensemble expériences que notre méthode surpasse constamment les techniques de l'état de l'art étroitement liées et permet de concevoir des réseaux légers pouvant atteindre des niveaux élevés de parcimonie sans perte significative de précision.

Contents

List of Figures

List of Tables

Acronyms

`NaN` Not a Number

AI Artificial Intelligence

ANN Artificial Neural Network

ASLP Arbitrarily Shifted Log Parametrisation

BN Batch Normalisation

CNN Convolutional Neural Network

Conv Convolutional

DNN Deep Neural Network

DWR Dynamic Weight Rescaling

FC Fully Connected

FFT Fast Fourier Transform

FLOP Floating Point Operation

FP32 single-precision floating-point format

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

GS Gumbel-Softmax

IoT Internet of Things

KD Knowledge Distillation

LT Lottery Ticket

LTH Lottery Ticket Hypothesis

MAC Multiply-Accumulate

MLP Multilayer Perceptron

NAS Neural Architecture Search

OBS Optimal Brain Surgeon

ReLU Rectified Linear Unit

SC Signed Constant

SGD Stochastic Gradient Descent

SKD Scaled Kaiming distribution

SR Smart Rescale

STE Straight Through Estimator

STGS Straight Through Gumbel-Softmax

TA Teacher Assistant

WR Weight Rescaling

Remerciements

La thèse est un défi aussi bien scientifique qu'humain et qui ne peut être relevé sans l'aide de nombreuses personnes qui m'ont apporté leur temps, leurs idées, leurs conseils et leur soutien. Je souhaite ici les remercier.

Je tiens tout d'abord à remercier mon directeur de thèse **Hichem Sahbi** pour son encadrement durant ces quelques années.

Je souhaite remercier **Jenny Benois-Pineau** et **Titus Bogdan Zaharia** pour avoir accepté d'être rapporteurs de cette thèse et pour le temps consacré à ce manuscrit. Mes remerciements s'étendent également à **Pierre Beauseroy**, **Nicolas Gac** et **Vincent Gripon** pour leur participation en tant que membres du jury.

Cette thèse est une thèse CIFRE, menée en partenariat avec Netatmo où j'ai pu rencontrer et travailler avec de brillants collègues. Je remercie tout particulièrement **Alice Lebois**, **Mohammed-Amine Alaoui** mais aussi **Guillaume Michel** et **Mehdi Felhi** qui ont été de grands soutiens et m'ont aidé à progresser tout au long de cette thèse. Je veux également remercier **Chadi Gabriel**, **Steeve Vu**, **Fabien Freling** et **Yacine Mezaguer** et toutes les autres personnes que j'ai pu côtoyer chez Netatmo pour les discussions enrichissantes, leur soutien, leurs coups de main et les bons moments que nous avons pu passer ensemble.

Je remercie l'école doctorale et en particulier son directeur **Habib Mehrez** pour sa bienveillance, son soutien et son aide précieuse.

Je remercie mes proches dont l'intensité et l'exigence de ces quelques années m'ont un peu éloigné, à mon grand regret. Je remercie mes amis qui m'ont accompagné et soutenu. Je pense à **Hugo**, **Pierre**, **Antoine**, **Nicolas**, **Arnaud**, **Paul-Octave**, le groupe des **Lamas** et ses nombreux

docteurs, le groupe des **Cryptogourmets**, **Arthur**, **Brian** et en particulier **Léo** pour m'avoir inspiré à entreprendre, moi aussi, l'aventure qu'a été cette thèse.

Je remercie très chaleureusement **Odile** pour son aide inestimable et ses précieux conseils qui m'ont aidé à avancer et ne pas baisser les bras.

Je remercie mes parents **Denis** et **Véronique** ainsi que ma sœur **Pauline**. Ils m'ont toujours soutenu dans tous les défis que je m'étais lancé et j'espère que je les aurai rendus fiers d'être arrivé au bout de celui-ci.

Enfin, je remercie **Alice**, qui, en plus d'être ma moitié, a été ma béquille pendant ces quelques années. Elle m'a écouté, soutenu, accompagné et encouragé, parfois au mépris de ses propres ambitions et projets. Elle a été mon moteur, tant et si bien que tout ceci n'aurait pas été possible sans elle.

À toutes celles et à tous ceux que j'ai cités, mais aussi à celles et ceux que j'ai oubliés, merci.

Chapter 1

Introduction

Contents

1.1 Context

From the spinning jenny, blast furnace and steam engine that sparked the first industrial revolution to the Internet of Things (IoT) devices that drives the fourth, the objective of mechanising labour and optimising productivity has been a persistent theme throughout the past centuries. The first industrial revolution, which dates back to 1760, introduced mechanisation through the use of water wheels and steam engines. The second industrial revolution, starting towards the end of the XIXth century, is linked to the development of automobiles, crude oil extraction and assembly lines powered by electric energy. The third industrial revolution, also called the digital revolution, took place in the second half of the XXth century and brought electronics, information and communication technology, and automated production. The Fourth Industrial Revolution, often known as Industry 4.0, inaugurates the digital integration of production chains as well as smart and connected devices that lead to more efficient manufacturing systems. The fourth industrial revolution focuses on the interconnectivity of devices and the development of their computational capabilities. This track leads to the emergence of ever-connected IoT devices with embedded computing facilities, such as smartphones, autonomous vehicles or satellites, that leverage Artificial Intelligence (AI) algorithms.

In parallel with these industrial revolutions, the field of AI has seen substantial growth and development. The term *Artificial Intelligence* was first used at the Dartmouth workshop in 1956 which is considered to be the founding event of AI as a research field [133]. It launched decades of research into machine learning and natural language processing among others [142]. In the subsequent decades, AI saw significant strides, including the development of rule-based systems, called expert systems [49], in the 1970s and the early exploration of machine learning in the 1980s [169]. These advancements occurred alongside the third industrial revolution, setting the stage for further progress in AI. In the late XXth and early XXIst centuries, coinciding with the premises of the fourth industrial revolution and helped with substantial progress in computational power of Graphics Processing Units (GPUs), AI started to draw tremendous attention from both researchers and industrials with the advent of Deep Learning. The latter is a subfield of machine learning which uses multi-layer Artificial Neural Network (ANN) to learn and model complex patterns in datasets in an end-to-end fashion, bringing significant improvement over manually engineered data representation. The fast development of Deep learning has been driving advancements in various domains such as natural language processing [12, 27, 193], image and speech recognition [102, 175, 67, 61, 13, 4], text and image generation [53, 98, 12], video game playing [173, 174] and molecule folding [95] to name a few.

The conquest of new fields and the quest for performance improvement of Deep Learning models have led to a significant increase in their computational complexity and size (see figure 3.12), particularly regarding their number of parameters. The sheer size of modern ANNs, called Deep Neural Networks (DNNs), presents a significant barrier to their deployment on embedded devices or IoT devices whose memory and computational resources are inherently limited. To circumnavigate this hurdle, the prevalent approach is to offload computations onto remote servers, leveraging the ever-interconnected nature of modern IoT devices and appliances.

Nonetheless, several compelling reasons exist for conducting embedded computations instead of moving them to the cloud. First, processing the data locally on premises ensures better data privacy, since the latter does not need to leave the device to be processed on the cloud. Indeed cloud instances can be located on various continents or countries where the legislation about data privacy might be different from the one of the countries

where the data is collected. Second, local computations can distribute the processing and limit communications. This is particularly relevant in more ways than one: first, it can reduce the cost of communication and bandwidth, which are typically billed to companies by cloud providers. Second, in some scenarios, the device might not have access to a large bandwidth or cannot afford to transmit a lot of data, which can be the case for remote areas or some devices with a low power budget. Third, local computations can lead to greater responsiveness by reducing latency, which might be critical in some applications such as autonomous vehicles. Fourth, local computations can enable autonomy, which is particularly relevant for devices that cannot rely on internet access, such as Mars rovers, submarine drones or any other devices that need to process data in radio silence.

The fourth industrial revolution and the rapid evolution in the field of AI have opened up a myriad of applications, with AI algorithms and in particular DNNs, offering significant potential to enhance the capabilities of IoT devices. However, the deployment of these advanced DNNs on IoT devices presents a significant challenge due to the inherent computational and memory constraints of such devices. The sheer size and complexity of modern DNNs, which have been instrumental in their success, become a barrier when considering on-device deployment. This presents a compelling case for the development of lightweight neural networks, tailored for IoT devices, that maintain the power of their larger counterparts while being significantly reduced in size and computational requirements. Such lightweight neural networks can also benefit all areas where saving computational resources is of interest. Consequently, there is a need for dedicated research efforts to design methods that yield lightweight neural networks. This thesis aims to contribute to this effort by introducing pruning methods that can reduce the size of neural networks while preserving their performances, with a focus on topology selection. We introduce two new pruning methods: The first performs joint topology and weight optimisation allowing for a minimal loss in performance after pruning compared to standard methods. The second approach does not require any weight training and instead focuses on stochastic yet differentiable topology selection, achieving compelling results overall and outperforming other related state-of-the-art methods that, again, do not train the weights.

1.2 Industrial Context

This research work is a CIFRE thesis with Netatmo, a French company specialising in smart devices that is now part of the Legrand Group. Notably, Netatmo commercialises security cameras for individual use that perform tasks such as face recognition and object detection using DNNs. The objective is to run the DNNs directly on these cameras, sidestepping the need to send data to distant servers. This approach aligns well with the reasons outlined in the previous section, particularly in ensuring data privacy. Moreover, it allows for a subscription-free business benefiting the end user, since there is no need to pay for cloud infrastructures dedicated to running DNNs. Therefore, Netatmo needs to develop lightweight neural networks that can be run on embedded devices while maintaining the performance of their larger and more complex counterparts. The models should be lightweight in order to, on the one hand, run on limited hardware, and on the other hand, be fast enough to perform, for instance, real-time intruder detection and alerting.

1.3 Why Deep learning ?

Deep learning is a subfield of machine learning that is the subject of intense research efforts and numerous publications. It employs Artificial Neural Networks, called Deep Neural Networks (DNNs), that aim to learn and model complex patterns in unstructured data in an end-to-end fashion. Deep learning models have proven their effectiveness in numerous domains and have been particularly performant in the field of computer vision [67, 160, 121]. Computer vision, which lies at the heart of Netatmo smart camera functionalities, encompasses algorithms that enable computers to interpret and understand the visual world and in particular detect and classify objects.

DNNs are the backbone of most advanced computer vision applications, including Netatmo facial recognition and object detection features. More specifically, Convolutional Neural Networks (CNNs), a specific type of DNNs can process images directly, reducing the need for manual feature extraction, and their capacity for hierarchical feature learning makes them particularly effective for tasks such as object recognition and classification. Their architecture is such that they perform well at recognising patterns

in unstructured data and are able to learn gradually more complex and abstract concept representations from raw data, enabling them to outperform other machine learning models and humans in computer vision tasks (see figure 1.1).

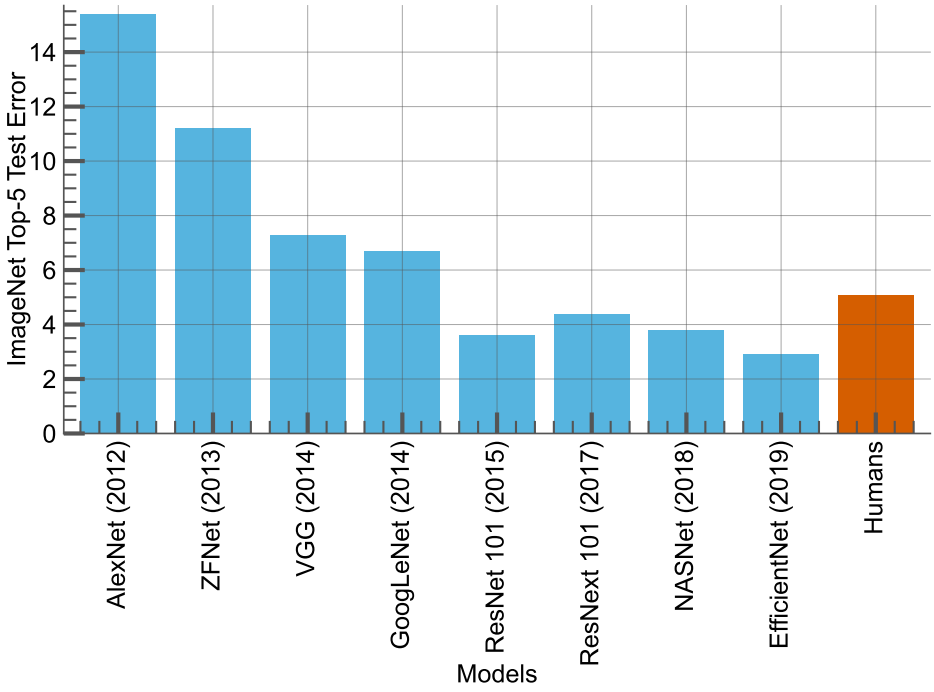


Figure 1.1: Models top-5 accuracy on ImageNet [25] compared to human performance.

Given the nature of tasks the Netatmo cameras are designed to perform, deep learning and Deep Neural Networks are not just a choice but a necessity. They represent the state of the art in computer vision tasks that outperforms other algorithms and allows for accurate and reliable object detection and recognition.

1.4 Challenges

While deep learning, particularly through the use of CNNs, is the technology of choice for computer vision applications, it comes with its challenges that need to be addressed, especially in the context of deploying these deep and large models on embedded devices. These challenges include model complexity and computational requirements. The necessity of compressing neural networks has been highlighted previously and also comes with its

challenges that include: preserving the performance and controlling the size of the compressed model as well as training time.

One of the most significant challenges in deploying deep learning models and especially CNNs on embedded devices is the large model size. These models often have millions of parameters and this makes them computationally heavy and challenging to fit into the limited memory of embedded devices. Secondly, these complex models require substantial computational resources to operate. This translates into slow computations which is a critical issue for devices which aim to perform real-time tasks.

Compressing large neural networks is a necessity to deploy them on embedded devices. However, it comes with its challenges. First, the compressed model should maintain the performance of the original model. However, the original large model is trained with all its parameters and thus depends on all of them. Consequently, removing more than a few can lead to degraded performance.

Second, the compressed model should be small enough to fit into the limited memory of embedded devices. It means that the process should be controlled to ensure that the size of the resulting model does not exceed the memory budget. However, compressing the model too much can lead to an irrecoverable loss in performance. The compression procedure and hyperparameters should be carefully chosen to ensure that the produced model has enough capacity to perform the task at hand. This is often achieved by grid-searching the optimal set of hyperparameters, which can be time-consuming.

Third, the compression process should be fast enough to be practical. Indeed, this process is often performed after the training of the original model and often requires fine-tuning the compressed one to compensate for the loss of performance. This fine-tuning step can be computationally expensive and time-consuming, effectively doubling the training time of the model in some scenarios.

To conclude, while deep learning and CNNs represent an exciting advancement in computer vision applications, several challenges need to be addressed for efficient and effective deployment on embedded devices. Addressing these challenges forms the crux of this research, with a particular

focus on model compression techniques to reduce the size and complexity of neural networks without significant loss in performance.

1.5 Contributions

This thesis tackles the challenge of compressing DNNs through pruning, a technique that aims to reduce the size of a neural network by removing redundant or unnecessary parameters, subsequently detailed in chapter 3. The contributions detailed in this manuscript focus on methods to identify the parameters to prune as well as minimise the impact of their removal on the final performance. These contributions are as follows:

Budget-aware pruning with weight reparametrisation. The two main challenges when pruning a neural network are first, determining which weights should be removed and then, mitigating the loss of performance introduced by weight removal. The first challenge is often referred to as determining the saliency of the weights, which is a score that reflects the importance of the weights in the network. The second challenge is often sidestepped and the pruned network is simply fine-tuned to recover the lost performance. To address both of these challenges, we propose the following main contributions:

- A numerically stable reparametrisation function, used in both our weight reparametrisation and our budget regularisation loss (subsequently detailed), that acts as a surrogate differentiable ℓ_0 norm.

- A weight reparametrisation that embeds the saliency score of the weight in its expression and therefore value. This reparametrisation allows to soft-prune the weights during training thereby significantly mitigating the performance drop that occurs after pruning. Moreover, this reparametrisation does not require the introduction of auxiliary variables to determine the saliency of the weights, leading to a minimal impact on memory and computational requirements.

- A budget regularisation loss that allows to drive the optimisation procedure to respect a given budget. This budget regularisation loss benefits directly from the aforementioned reparametrisation function to compute the current weight budget. It is optimised jointly with the

original loss, leading to an optimal solution in terms of performance and budget.

- A comprehensive set of experiments that demonstrate the effectiveness of our method and validate each one of its components on various datasets and architectures.

These contributions have been published in the following article:

- Robin Dupont, Hichem Sahbi, and Guillaume Michel. Weight reparametrization for budget-aware network pruning. In *2021 IEEE International Conference on Image Processing, ICIP 2021, Anchorage, AK, USA, September 19-22, 2021*, pages 789–793. IEEE, 2021.

Pruning without weight training with stochastic sampling. As mentioned above, a major hurdle in pruning is determining which weights to remove. This is especially challenging since weights, and consequently their saliency, can fluctuate throughout training. This implies that pruning should either be reversible or performed at the end of training. We propose a different approach that does not require training the network to determine the saliency of the weights, the latter being fixed throughout the process. Instead, we sample a subset of weights (effectively pruning the other weights) forming a subnetwork of the original network and evaluate its performance. This allows us to search for a topology that is both lightweight and performant inside the original network without training its weights. The main contributions of this method are as follows:

- A stochastic weight sampling method that is computationally efficient, numerically stable, differentiable and allows sampling weights while training their probability of being selected, represented by latent masks. The optimisation of the latter allows to learn the saliency of the weights without training the network, and therefore identifying and extracting an effective subnetwork.
- A pruning strategy for the masks that freeze the topology and performs better than averaging methods previously used in the state-of-the-art. Moreover, this pruning strategy allows to discover the optimal pruning rate for the network, eliminating the need for costly grid search to determine it.

- An efficient learnt-based weight rescaling mechanism to compensate for the disruption in weight distribution caused by stochastic sampling. This rescaling is less computationally intensive, more flexible and allows for smoother variations of the scaling factor than other rescaling methods.
- A comprehensive set of experiments that demonstrate the effectiveness of our method and validates each one of its components on various datasets and architectures, as well as comparison with other closely related state-of-the-art methods in various configurations.
- A public repository containing the implementation of our method and the methods we compare against, as well as detailed code and instructions to reproduce our results.

These contributions have been published in the following article:

- Robin Dupont, Mohammed Amine Alaoui, Hichem Sahbi, and Alice Lebois. Extracting effective subnetworks with Gumbel-Softmax. In *2022 IEEE International Conference on Image Processing, ICIP 2022, Bordeaux, France, 16-19 October 2022,* pages 931–935. IEEE, 2022.

1.6 Outline

The rest of this thesis is organised as follows:

Chapter 2 offers an introduction to deep learning, providing a detailed overview of its foundational and core concepts. It first explores early architectures, beginning with the *Perceptron* and the Multilayer Perceptron (MLP). The focus of the chapter then shifts towards neural network training, giving formal definitions of the loss function, regularisation, and optimisation process. A dedicated section delves into Convolutional Neural Networks, exposing and detailing their building blocks, and the evolution of their architectures. Then, the architectures used in the experiments of chapters 4 and 5 are detailed. Additionally, this chapter lists and describes prominent datasets, namely CIFAR-10, CIFAR-100, and TinyImageNet, and discusses their respective train, validation, and test sets.

Chapter 3 introduces deep neural network compression and presents state-of-the-art methods divided into different families. The chapter begins with acceleration techniques and presents a range of methods whose goal is to speed up matrix operations or convolutions. Then, it explores the teaching paradigm, highlighting methods that rely on a large pre-trained network to improve the training of lightweight ones. Furthermore, the chapter addresses the design aspects of lightweight architectures introducing building blocks for efficient architecture design and Neural Architecture Search. Afterwards, the chapter discusses methods to compress and optimise existing architectures and in particular pruning. Finally, the chapter presents the positioning of our methods and the rationale behind them.

Chapter 4 presents our pruning method based on weight reparametrisation and budget regularisation. It starts by outlining closely related work. Then, the core method components are examined, starting with our weight reparametrisation and then our budget loss. Afterwards, a general overview of the algorithm is provided. Furthermore, the chapter details experiments assessing our method performance in various configurations as well as experiments validating the components of our method and the choices of hyperparameters. A conclusion summarises the key findings and highlights of our method for neural network pruning.

Chapter 5 delves into our stochastic pruning method without weight training. It starts with an introduction and examination of closely related work. Then, it details the first core component of our method, namely Arbitrarily Shifted Log Parametrisation, a method for extracting effective subnetworks using the Gumbel-Softmax technique that solves various issues that arose from previous methods. Afterwards, it introduces our weight-rescaling technique and presents its main benefits, as well as our pruning strategy to freeze the stochastic topology. Subsequently, a method and algorithm overview outlines the key points of our methods. Furthermore, the chapter exposes a comprehensive set of experiments that compares our method against other state-of-the-art methods in various scenarios and validates the components of our method. The chapter concludes by summarising our findings and results.

Chapter 2

Deep Learning Overview

Contents

2.1 Introduction

Deep Learning is a subfield of machine learning that focuses on the study of Deep Neural Networks (DNNs) which have their roots in Artificial Neural Networks (ANNs). DNNs aim to learn a representation from unstructured data such as raw images [102], text [12] or audio [61], in an end-to-end fashion. DNNs have been used to solve a wide range of tasks, including image and speech recognition [102, 175, 67, 61, 13, 4], natural language processing [12, 27, 193], object detection [159, 160], semantic segmentation [125, 118], text and image generation [53, 98, 12] as well as exotic domains like video games [173, 174] or molecules folding [95]. ANNs were initially conceptualised based on the understanding of biological neural networks

present in the brain [134, 73]. Rosenblatt proposed in [166] a theoretical model of a neuron, denoted the *perceptron*, which was capable of learning a linear decision boundary. The perceptron model was later extended to multiple layers of neurons, giving rise to the Multilayer Perceptron (MLP) [167, 169]. A Multilayer Perceptron is a type of artificial neural network that extends the concept of a single-layer perceptron by including one or more hidden layers of neurons connected downstream from an input layer and upstream to an output layer. Each layer is fully connected to the next, allowing the model to learn and represent more complex, non-linear relationships in the input data. Although more capable than the perceptron, the MLP is still limited by its depth. The next advance came from the stacking of multiple layers, leading to Deep Neural Networks.

In the context of DNNs, the term *deep* denotes the stacking of many layers within a neural network. The concept of DNNs is based on the idea that the depth and the numerous layers can help in learning features at various levels of abstraction, enabling the network to learn complex hierarchical pattern representations. For instance, in the context of image recognition, lower layers learn local features like edges and textures, while deeper layers learn to identify more abstract concepts like shapes or objects.

The rise of DNNs was made possible by several factors. On the one hand the increase in computational power, and in particular the use of GPUs, made the training of large and deep networks feasible. Indeed, AlexNet, the first CNN to win the ImageNet Large Scale Visual Recognition Challenge [102], was trained on two GPUs in parallel to accelerate computations. Nowadays, the use of GPUs or dedicated hardware such as Tensor Processing Unit [94] is ubiquitous and supported by all the major deep learning frameworks [1, 147]. On the other hand, the availability of large-scale datasets such as ImageNet [25] allowed to train or pre-train deep networks with millions of parameters without overfitting.

This chapter aims to give an overview of the different neural network architectures, building blocks, training techniques and datasets that are widely used in Deep Learning for computer vision and in our experiments. Section 2.2 introduces the early neural network architectures, namely the perceptron and the MLP. Section 2.3 focuses on the functional definition of a neural network and its training. Section 2.4 presents the building blocks and architectures of various CNNs for computer vision, and in particular

the ones we benchmark our methods with (see sections 4.4 and 5.5). Finally, Section 2.5 gives an overview of the most prevalent datasets that we used in our experiments.

2.2 Early Architectures

In this section, we present the perceptron [166] and then the Multilayer Perceptron [167, 169]. Both are the two founding neural network architectures that led to the development of Deep Neural Networks.

2.2.1 Perceptron

The *perceptron* is a model of artificial neuron, capable of learning a linear decision boundary. It was proposed by Rosenblatt in 1958 [166] and conceptualised based on the understanding of biological neural networks present in the brain [134, 73]. The perceptron is composed of inputs that are weighted and summed before being passed through a nonlinear function referred to as an activation function. The conceptual representation of the perceptron is displayed in figure 2.1 and its mathematical formulation is defined in equation (2.1):

$$\hat{y} = g(\sum_{i=1}^{n} w_i \cdot x_i + b) \tag{2.1}$$

where x_i is the ith input, w_i its associated weight, n is the number of inputs, b is the bias, g is the activation function, and $\hat{y}$ is the output of the perceptron. This formulation can also be written in vector form as in equation (2.2):

$$\hat{y} = g(\mathbf{w}^T\mathbf{x} + b) \tag{2.2}$$

where $\mathbf{x}$ is the vector of inputs and $\mathbf{w}$ is the vector of weights. The activation function g is typically a nonlinear function, such as the sigmoid or the hyperbolic tangent (see figure 2.5). Due to its shallow architecture, the perceptron cannot learn complex decision boundaries. Nevertheless, it

is possible to stack several perceptrons to learn nonlinear decision boundaries, leading to a Multilayer Perceptron.

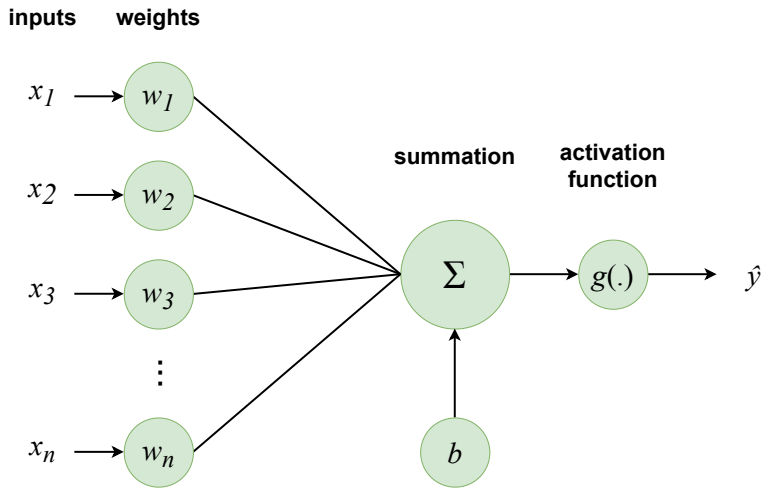


Figure 2.1: Conceptual scheme of the *perceptron*. Each input x_i is multiplied by its associated weight w_i and summed to the other weighted inputs. The bias b is added to the sum and the result is passed through an activation function g to produce the output $\hat{y}$.

2.2.2 Multilayer Perceptron

The Multilayer Perceptron (MLP) is an extension of the perceptron model, comprising multiple layers of perceptrons, also referred to as neurons [169]. A MLP with one hidden layer is represented in figure 2.2. In the latter, the circles represent the neurons and the connections between them, representing weights, are materialised by lines. The MLP is the simplest type of feedforward ANN. Feedforward refers to the fact that the connections between neurons in the MLP form a directed acyclic graph, where the outputs of the neurons from one layer are passed to the next, with no backward connections or feedback. Using the same notations as in equation (2.2), the vector form of the MLP displayed in figure 2.2 can be written as in equation (2.3), where the subscript of activation functions g_i, weight matrices $\mathbf{w}_i$ and bias vectors $\mathbf{b}_i$ denotes their belonging to the ith layer.

$$\hat{\mathbf{y}} = g_2(\mathbf{w}_2^T \cdot g_1(\mathbf{w}_1^T \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \tag{2.3}$$

Each layer of the MLP, being fully connected to the next one, enables the MLP to handle problems that the perceptron cannot solve, such as problems requiring nonlinear decision boundaries. Furthermore, Cybenko proved in [24] that an MLP can approximate continuous functions on compact subsets of $\mathbb{R}^n$. This result is known as the *Universal Approximation Theorem*. Before the emergence of Deep Learning, MLPs have been applied to various domains, including voice recognition, image recognition, and machine translation [199].

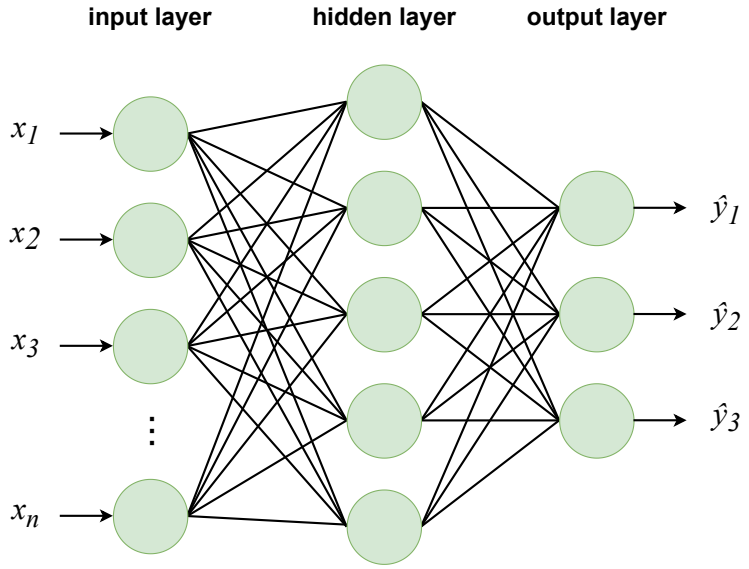


Figure 2.2: Conceptual scheme of a MLP with one hidden layer. Each circle represents a neuron and each line a connection associated with a weight.

2.3 Neural Network Training

Neural Network Training revolves around the optimisation of a mapping function that learns to predict an output given input data by adjusting its internal parameters, also referred to as weights. This optimisation, also called *training*, involves iteratively tuning these weights so that the discrepancy between the output predicted by the model and the reference output is minimised. Weights tuning relies on gradient-based methods that hinge around two core components: the *backpropagation* algorithm to compute the gradients and the Stochastic Gradient Descent (SGD) algorithm to update the weights.

2.3.1 Functional Definition

Neural networks can be defined as a mapping function from an input space $\mathcal{X}$ to an ouput space $\mathcal{Y}$. This mapping function f is characterised by a set of parameters θ, often called *weights*. The training of a neural network consists in tuning the parameters θ so that, given an input X, the mapping function f output, denoted $\hat{y}$, is as close as possible to the associated true output y. This training is done iteratively by using example pairs $(X, y) \in \mathcal{X} \times \mathcal{Y}$, where $X \in \mathcal{X}$ is the input and $y \in \mathcal{Y}$ is the output. In the context of image classification, X is an image and y is a label that indicates the class of the associated image. A functional representation of a neural network is given in equation (2.4), where f is the neural network, θ is the set of parameters of the network, $X \in \mathcal{X}$ is the input given to the neural network and $\hat{y}$ is the output.

$$\begin{aligned} f &: \mathcal{X} \to \mathcal{Y} \\ X &\mapsto f(X, \theta) = \hat{y} \end{aligned} \tag{2.4}$$

Considering image classification, the output $\hat{y}$ is a probability vector where the largest coefficient is the one whose index corresponds to the predicted class of the input image. This vector is generally converted into a one-hot vector, where the only non-zero coefficient is at the index of the predicted class. The true label y, referred to as the ground truth, is the class index so that $y \in [\![0; C-1]\!]$, where C is the number of classes considered. The ground truth can also be converted into a one-hot vector.

2.3.2 Loss Function and Regularisation

Training a neural network aims at finding the optimal parameters θ that maximise a performance, quantified by a metric, often based on the discrepancy between the predicted output $\hat{y}$ and the true output y. However, optimising directly the metric might be intractable. To solve this issue, one may define a differentiable cost function and minimise the latter as a proxy for optimising the metric. Considering k example pairs (X_k, y_k), the cost function $\mathcal{J}(\theta)$, also referred to as the *empirical risk*, is defined in the following equation:

$$\mathcal{J}(\theta) = \frac{1}{k}\sum_{i=1}^{k}\mathcal{L}(f(X_k,\theta),y_k) \tag{2.5}$$

where $\mathcal{L}$ is the loss function. Note that the true data distribution, and therefore the risk, is unknown. This is why the empirical risk, computed with a set of example pairs, is used instead. The minimisation of the empirical risk alone is not sufficient to ensure good overall performance. Indeed, the neural network could learn to perfectly predict the output of the training set but may fail to generalise to unseen data. This phenomenon is called *overfitting*. To prevent overfitting, we add a regularisation term to the empirical risk. The regularisation term, denoted $\mathcal{R}$, is a function of the parameters θ of the neural network which penalises the complexity of the model, and thus prevents overfitting. To account for regularisation, the cost function in equation (2.5) is updated to:

$$\mathcal{J}_r(\theta) = \frac{1}{k}\sum_{i=1}^{k}\mathcal{L}(f(X,\theta),y) + \mathcal{R}(\theta) \tag{2.6}$$

Loss function. In equations (2.5) and (2.6), the loss function $\mathcal{L}$ is a measure of the discrepancy between the ground truth y and the predicted output. Contrary to the metric P which might be non-differentiable, the loss function is differentiable so that its minimisation can be achieved using gradient-based methods, subsequently detailed in section 2.3.3. The choice of the loss function depends on the task at hand. For classification tasks (not only images), the loss function is often the *cross-entropy* loss. For a binary classification problem, the ground truth is a binary variable $y \in \{0,1\}$ and the predicted output is a scalar $f(X,\theta) = \hat{y} \in [0,1]$. The binary cross-entropy loss is defined as follows:

$$\mathcal{L}(\hat{y},y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}) \tag{2.7}$$

The binary cross-entropy loss defined in equation (2.7) can be extended to problems with more than two classes. For a classification problem with C classes, the ground truth is a one-hot vector $\mathbf{y} \in \{0,1\}^C$ and the output is

a C-dimensional vector $f(X, \theta) = \hat{\mathbf{y}} \in \mathbb{R}^C$. The multi-class cross-entropy loss is defined as follows:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{C} y_i \log\left(\phi(\hat{\mathbf{y}})_i\right) \tag{2.8}$$

In the above equation, $\hat{\mathbf{y}}$ is the unnormalised raw output vector of the neural network and ϕ is the softmax function, whose expression is given in equation (2.9). The softmax function is used to convert the raw output vector of real numbers into a probability distribution. Note that some models which use the softmax as the activation function of their last layer output directly a probability distribution, in which case the softmax is not needed.

Considering a vector $\mathbf{z} = [z_1, \ldots, z_n]$, the j-th component of vector $\mathbf{z}$ normalised by the softmax function is given by:

$$\phi(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{n} \exp(z_k)} \tag{2.9}$$

Regularisation. The regularisation term $\mathcal{R}$ is a differentiable function of the weights θ. It acts as a control mechanism to avoid overfitting by preventing the weights of the neural network from becoming too large, which can lead to overly complex models that overfit the training data. This is typically achieved by adding a penalty proportional to the magnitude of the weights, thereby keeping them small.

Common types of regularisation include ℓ_1 and ℓ_2 regularisation, whose expressions are shown in equations (2.10) and (2.11) respectively. ℓ_1 regularisation [188], adds a penalty equal to the absolute value of the magnitude of the weights. On the other hand, ℓ_2 regularisation [78], adds a penalty equivalent to the square of the magnitude of the weights. Both methods aim to reduce the magnitude of the weights, but ℓ_1 regularisation is more targeted towards feature selection, effectively pushing some weights to 0, whereas ℓ_2 restrains globally their magnitude.

The regularisation term $\mathcal{R}$ is added to the cost function with a regularisation coefficient, usually denoted as λ, which is a hyperparameter that

balances the trade-off between fitting the training data (minimising the loss $\mathcal{L}$) and limiting the complexity of the model (minimising $\mathcal{R}$).

For a network with L layers and parameters $\theta = \{\mathbf{w}_1, \ldots, \mathbf{w}_L\}$, the ℓ_1 and ℓ_2 regularisation term is defined as follows:

$$\mathcal{R}_{\ell_1}(\theta) = \lambda \sum_{i=1}^{L} \|\mathbf{w}_i\|_1 \tag{2.10}$$

$$\mathcal{R}_{\ell_2}(\theta) = \frac{\lambda}{2} \sum_{i=1}^{L} \|\mathbf{w}_i\|_2^2 \tag{2.11}$$

where $\|.\|_1$ and $\|.\|_2^2$ respectively denote the sum of the absolute value and the sum of the squaring of each element of the vector.

2.3.3 Loss Optimisation

As mentioned before, the training of a neural network involves finding the optimal set of parameters θ that minimises a cost function $\mathcal{J}(\theta)$. This process of optimisation is typically carried out using gradient-based methods which rely on the iterative adjustment of the parameters in the opposite direction of the gradient of the cost function. The gradient of a function provides the direction of the steepest ascent at a given point [11]. Thus, by moving the parameters in the opposite direction of the gradient, we seek to descend to a local minimum of the function.

Backpropagation. One critical step in the optimisation process is the computation of the gradient of the cost function with respect to the parameters, $\nabla\mathcal{J}(\theta)$. These gradients are computed with the *backpropagation* algorithm [169] which is an application of the *chain rule* (see equation (2.12)) to efficiently compute these gradients. It involves a forward pass through the network to compute the outputs and thus the loss, and a backward pass to calculate the gradients. During the backward pass, the partial derivative of the cost with respect to each parameter is computed, starting from the output layer and going back to the input layer. The previously computed

derivatives from the subsequent layers are used to compute the ones of the earlier layers, making the backpropagation algorithm computationally efficient.

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{2.12}$$

Stochastic Gradient Descent. Once the gradients are calculated, they are used to update the parameters. The most prevalent method for parameter updates is Stochastic Gradient Descent (SGD), a derivative of the Robbins–Monro algorithm [164]. In SGD, the gradient of the loss function is computed for a random subset of the data (a *batch* or *mini-batch*), and the weights are shifted in the direction that decreases the loss function. This is achieved by subtracting the gradient of the cost function with respect to that parameter multiplied by a learning rate η:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta \frac{\partial \mathcal{J}(\theta)}{\partial \theta_i} \tag{2.13}$$

where $\theta_i^{(t)}$ is the ith parameter at iteration t. The SGD algorithm is detailed in algorithm 1. The use of mini-batches in SGD leads to a trade-off between computational efficiency and estimation accuracy. Indeed, the gradient is estimated using a subset of the entire training set, which is, on the one hand, less accurate than using the whole dataset, but on the other hand, less computationally intensive. The size of the mini-batch, which is a hyperparameter of the training algorithm, determines this trade-off and should also be chosen depending on the computational and memory resources available. Note that the size of modern datasets, subsequently detailed in section 2.5, makes it intractable to evaluate the gradients on the whole dataset in one step, hence the use of mini-batches.

Learning Rate. The learning rate is a hyperparameter that determines the step size of the update at each iteration while moving toward a minimum of the loss function (see equation (2.13)). Setting the learning rate too high can cause the learning process to converge too quickly or overshoot while setting it too low can make the learning process slow to converge, as

Algorithm 1 Stochastic Gradient Descent Algorithm

Require: Learning rate η, mini-batch size m, Initial parameters $\theta^{(0)}$, $m' \geq m$ training pairs $(X, y) \in \mathcal{X} \times \mathcal{Y}$, Loss function $\mathcal{J}$
  while Stopping criterion not met **do**
    Sample mini-batch of size m from training set
    Compute gradient estimate on mini-batch: $\hat{g} \leftarrow \nabla \mathcal{J}(\theta)$
    Update parameters: $\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \hat{g}$
  end while
  return Optimal parameters θ

shown in figure 2.3.

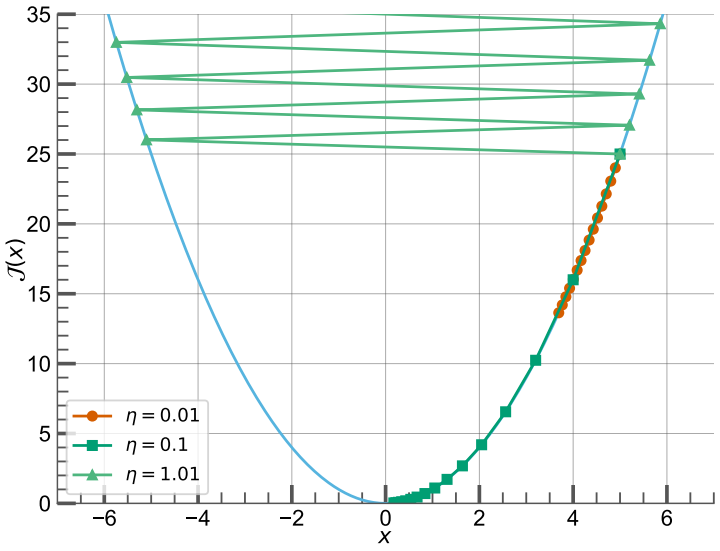


Figure 2.3: Illustration of the effect of the learning rate on the convergence of the gradient descent. The gradient descent has been applied iteratively for 20 epochs. On the one hand, a too-high learning rate ($\eta = 1.01$) causes the gradient descent to overshoot the minimum of the loss function. On the other hand, a too-low learning rate ($\eta = 0.01$) causes the gradient descent to converge slowly.

Alternative methods. To enhance the performance of SGD, various modifications and extensions have been proposed, such as SGD with momentum [181, 150], RMSProp [75], or Adam [100]. These methods aim to adjust the learning rate dynamically or dampen the oscillations in the gradient descent to achieve faster and more stable convergence.

For instance, SGD with momentum [181, 150] uses a momentum coefficient γ and smoothes the variations of the descent direction, thus preventing the optimisation from getting stuck in small local minima. The momentum term is a moving average of the gradient, here denoted v, and it is used to update the parameters as shown in equation (2.14). In this equation, the momentum coefficient $\gamma \in [0, 1]$ is a hyperparameter that is typically set close to 1, 0.9 being a common value.

$$\begin{aligned} v_{t+1} &= \gamma v_t + \eta \nabla \mathcal{J}(\theta) \\ \theta_{t+1} &= \theta_t - v_{t+1} \end{aligned} \tag{2.14}$$

2.4 Convolutional Neural Networks for Computer Vision

In the field of computer vision, CNNs have emerged as effective architectures that enable high performance on image classification tasks. The effectiveness of CNNs lies in their architecture that leverages the Convolutional (Conv) layers to automatically learn abstract features from visual data in a hierarchical fashion. In this section, we explore the building blocks of CNNs and various architectures that have been widely used and became *de facto* standards in the literature.

2.4.1 Building Blocks

This section covers the most common building blocks of CNNs for computer vision. These building blocks are organised in layers that are stacked to form neural network architectures subsequently detailed in sections 2.4.2 and 2.4.3.

Convolutional layer. Conv layers are one of the core building blocks of CNNs. Each convolution layer performs a series of spatial convolutions on the input data using a set of learnable filters or kernels. These filters are designed to extract low-level features such as edges, corners, and textures in the early layers, while they learn high-level features like object parts or

even whole objects in the deeper layers. Contrary to manual feature engineering, the features learned by Conv layers are learned in a *end-to-end* fashion. The 2D convolution operation is defined in equation (2.15) :

$$Y_{ij} = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} X_{i-a,j-b} \cdot K_{ab} \tag{2.15}$$

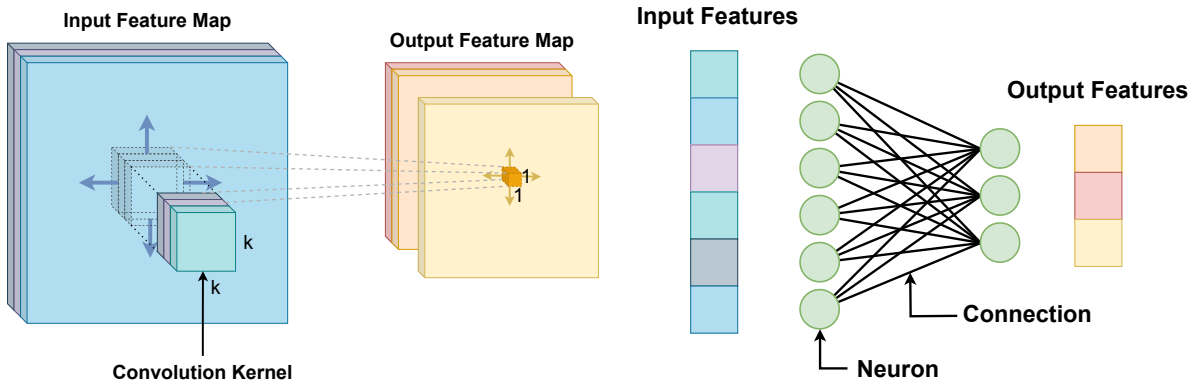
where $\mathbf{X}$ is the input, $\mathbf{K}$ is the kernel of size $k_h \times k_w$ and (i, j) are the spatial coordinates in the output feature map. Note that some Deep Learning frameworks implement *cross-correlation* instead of convolution. In the former, the kernel is not spatially flipped leading to the cross-correlation not being commutative [52]. The Conv layer kernels are typically smaller than the input along width and height dimensions (they are generally 3×3 [67]) but comprise as much channels as the input. During the forward pass, each kernel is spatially convolved channel-wise with the input and the convolution outputs are summed along the channel dimension to yield a single scalar for each kernel position on the input (see also figure 2.4a).

Conv layers are more computationally efficient than Fully Connected (FC) layers, as they have a form of weight sharing baked in. Indeed, the same kernel is applied to every location of the input, which brings two main benefits: *(i)* the number of parameters is independent of the input size and *(ii)* a single learned kernel, acting as a feature detector, can be used in multiple locations. This is especially useful for early feature detector that detects basic shapes or textures. In addition, because of the kernels being convolved across the whole input, Conv layers are also less sensitive to spatial translations that might occur in different instances of the same class.

Fully connected layer. FC layers, also known as *Dense* layers are often the last layers of a CNN, effectively serving as a classifier, whereas the Conv layers act as a feature extractor. FC layers perform high-level reasoning by conducting non-linear transformations of the extracted features and combining them to make decisions. In an FC layer, each neuron is connected to every neuron in the previous layer. A FC layer can be described as a matrix-vector product as in equation (2.16) (see figure 2.4b).

$$\mathbf{y} = \mathbf{w}^T \cdot \mathbf{x} + \mathbf{b} \tag{2.16}$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight matrix and b is the bias. In the context of CNNs, before passing the output of the last Conv layer to the first FC layer, it needs to be flattened or reshaped into a single column vector. The final layer in a CNN is a FC layer that has a number of neurons equal to the number of output classes, and it typically uses a softmax activation to output a probability distribution over those classes.



(a) Convolutional Layer **(b)** Fully Connected Layer

Figure 2.4: Conceptual representation of a Convolutional and a Fully Connected layer. The Convolutional layer (figure 2.4a) takes a multi-channel input and produces a multi-channel output. Each coefficient of the output is computed by applying a convolution operation at a corresponding location in the input. The Fully Connected layer (figure 2.4b) takes a vector input and produces a vector output. Each connection is represented by a weight in the weight matrix.

Activation functions. They are often applied to the output feature map of a convolutional or fully connected layer, resulting in the *activation map* or *activations*. These functions introduce non-linearity into the model, allowing it to learn more complex patterns [125]. A common activation function used in CNNs is the Rectified Linear Unit (ReLU), represented as $f(x) = \max(0, x)$. Other functions like the sigmoid $f(x) = 1/(1 + e^{-x})$ or tanh $f(x) = (e^x - e^{-x})/(e^x + e^{-x})$ functions have been used (see figure 2.5), however, the ReLU is preferred over the latter for its computational efficiency and its ability to mitigate the vanishing or exploding gradient problem [77, 50].

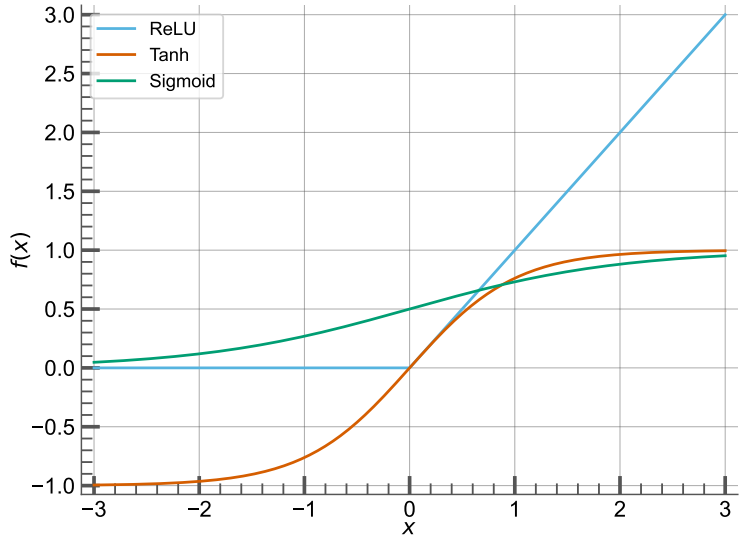


Figure 2.5: ReLU, tanh and sigmoid activation functions. Best viewed in colours.

Pooling. This operation is often employed after Conv layers in a CNN and aims at progressively reducing the spatial extent of the input representation, thus reducing the number of parameters and computations in the network. This also helps control overfitting and increases the receptive field of the subsequent layers. The pooling operation is performed independently on each input channel, so the number of channels remains unchanged. The two most common types of pooling are *max* and *average pooling*. The former selects the maximum value in each window (often of size 2×2), while the latter computes the average value of the window. Given an input matrix $\mathbf{X}$, the output matrix $\mathbf{Y}$ for a certain spatial location (i, j) is defined in equation (2.17) for *max pooling* and equation (2.18) for *average pooling*:

$$Y_{ij}^{\max} = \max_{(a,b)\in[0,k_h-1]\times[0,k_w-1]} X_{i+a,j+b} \tag{2.17}$$

$$Y_{ij}^{\text{avg}} = \frac{1}{k_h \times k_w} \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} X_{i+a,j+b} \tag{2.18}$$

where k_h and k_w represent the height and width of the pooling windows respectively. Note that pooling has no learnable parameters. It only downsamples the input based on a fixed function.

Batch Normalisation. Batch Normalisation (BN) is a technique introduced in [90] to combat the issue of internal covariate shift in deep neural networks, thereby accelerating training and improving generalization. Covariate shift refers to the changes in the distribution of features in the training and test dataset, which can lead to slow convergence, make the network harder to train or hinder its generalisation capabilities. BN normalises the input of the layer by adjusting and scaling the activations of the previous one. For each mini-batch of inputs (for instance, the activation map of the previous layer), it computes the mean and variance of the activations and performs normalization. The transformation is defined as follows:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{2.19}$$

where x_i is the input, μ_B is the mini-batch mean, σ_B^2 is the mini-batch variance, and ε is a small constant for numerical stability. After normalization, the method allows the network to learn an affine transformation for each activation, permitting the network to control the mean and standard deviation of the input distribution, formalised in equation (2.19):

$$y_i = \gamma \hat{x}_i + \beta \tag{2.20}$$

where, γ and β are the learnable parameters of the affine transformation. BN has the advantage of making the network less sensitive to the initial weights, allowing higher learning rates, and reducing the need for Dropout, among other regularisers. However, its effectiveness decreases in the case of small batch sizes, as the estimate of the batch mean and variance becomes less accurate.

Dropout. Dropout is a regularization technique used to prevent overfitting in neural networks. Dropout was introduced in [178] and works by randomly deactivating a proportion of neurons in a layer during each training iteration. More specifically, during the forward pass, each neuron has a probability p of being temporarily removed from the network, effectively breaking up co-adaptations between neurons and forcing them to learn more robust and independent features. The output of Dropout is given in equation (2.21):

$$r_i \sim \text{Bernoulli}(1-p), \quad \mathbf{y} = \frac{\mathbf{x} \odot \mathbf{r}}{1-p} \tag{2.21}$$

In the above equation, $\mathbf{x}$ denote the output of a layer processed with dropout, $\mathbf{r}$ is a binary mask vector of the same shape as $\mathbf{x}$, where each element of $\mathbf{r}$ is independently drawn from a Bernoulli distribution with probability $1-p$, leading to $r_i = 1$ if the associated weight is kept and a $r_i = 0$ if not. The product $\mathbf{x} \odot \mathbf{r}$ is scaled by $1-p$ to ensure that the expected value of $\mathbf{x}$ remains unchanged. During the evaluation, the dropout is changed to an identity function.

2.4.2 Architectures Evolution

The evolution of Convolutional Neural Networks is characterised by a consistent increase in their size and performance, alongside the introduction of new architectural modifications to address the limitations of their predecessors (see figure 2.9). In this section, we present a historical overview of the CNNs evolution and we subsequently detail the architectures that we used in our experiments.

One of the earliest CNN was introduced in 1998: LeNet-5 was developed for digit recognition [106], constituting a relatively simple network with 5 layers with learnable parameters: 2 Conv layers and 3 fully connected layers. Its size is significantly smaller compared to the contemporary models (see figure 2.9). With the introduction of AlexNet [102] in 2012, the network size considerably grew, comprising more layers and neurons to handle more complex tasks, like large-scale image recognition. AlexNet tackled the overfitting issue in LeNet-5 using data augmentation and dropout techniques, while also introducing and popularising the ReLU

activation function.

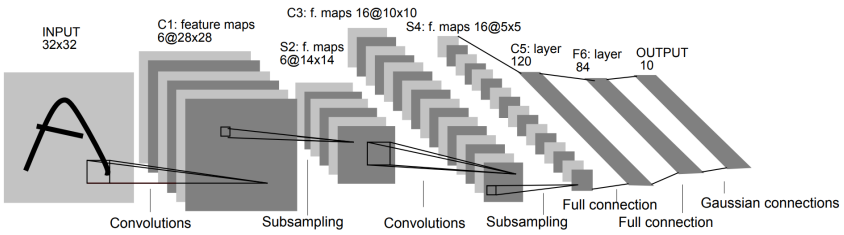


Figure 2.6: Architecture of LeNet-5, a Convolutional Neural Network used for handwritten digit recognition. Image taken from [106]

The next advancement was the VGG networks family [175] which proposed much deeper architectures with up to 19 layers, which is a significant increase over the 8 layers of AlexNet. However, the increased depth led to the *vanishing gradient* problems, which refers to the situation in training a deep neural network where gradients are backpropagated through layers and become increasingly small, effectively preventing the weights of earlier layers from learning and updating effectively. The VGG networks also introduced the practice of stacking multiple convolutional layers with small 3×3 filters instead of using larger ones. The same year, Google's Inception (or GoogLeNet) [183] was introduced, addressing the vanishing gradient issue with its novel inception modules, which allowed the network to learn at varying scales and increased computational efficiency, without overly increasing the network size. GoogleNet was also the first CNN that was not a simple stack of layers and processed a single input with different blocks in parallel before merging them.

Later, the ResNet models family was proposed in [67], which effectively tackled the vanishing gradient problem by introducing skip (or shortcut) connections, allowing gradients to backpropagate directly through several layers. These shortcut connections also allowed the network to grow in depth up to 152 layers without a significant increase in computational cost. However, a challenge remained with the constant need for careful design to manage feature-map sizes. Indeed, stacking numerous layers, with their channel count increasing with depth, can lead to an explosion in the number of parameters as well as increased memory consumption.

In response, DenseNet [85] was proposed. It connects each layer to every other following layer of the same block in a feed-forward fashion. By

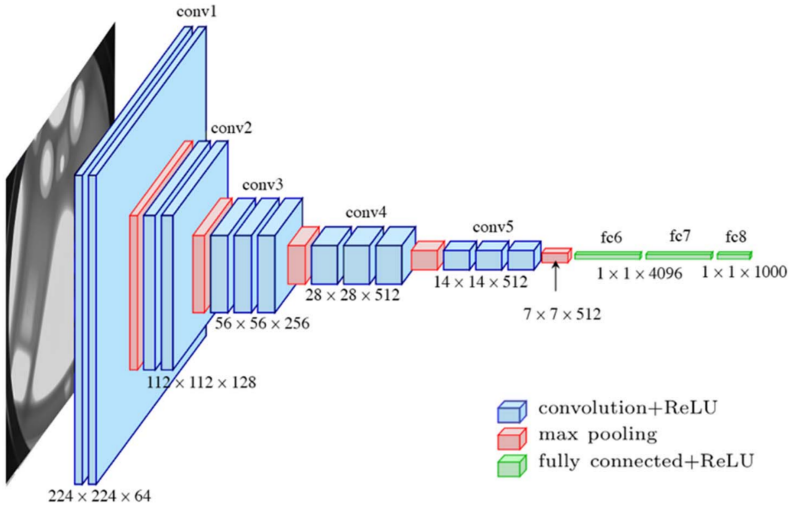


Figure 2.7: Architecture of the VGG16 network introduced in [175]. Image taken from [39]

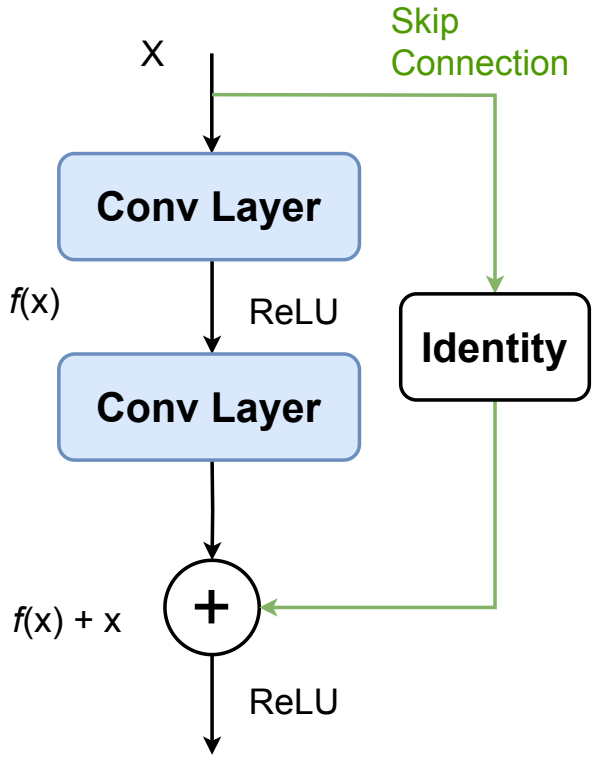


Figure 2.8: A residual block and its skip connection used in ResNets[67]. The identity skip connection allows for the gradient to be backpropagated directly through several layers, thus mitigating the *vanishing gradient* problem.

reinforcing the propagation of features and gradients through the network, the DenseNet architecture alleviates the vanishing-gradient problem and further improves the information flow from earlier layers to later ones by reusing earlier features in the deeper layers. Thus, through these chronological advancements, neural networks not only grew in size but also improved in performance, thereby becoming more efficient and capable of handling more complex tasks.

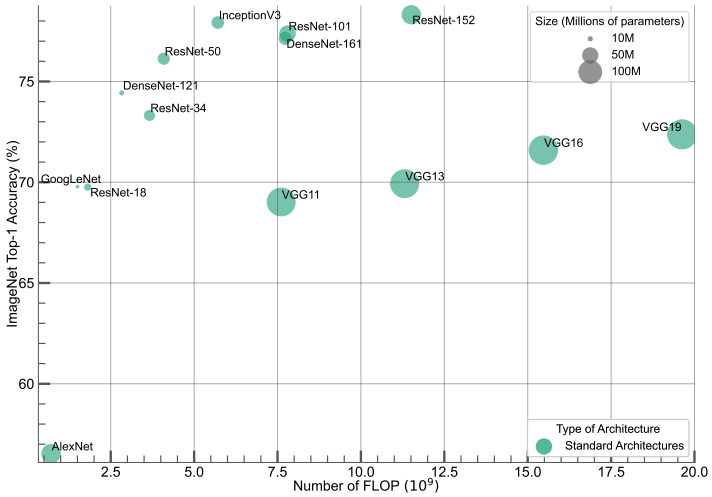


Figure 2.9: Networks size comparison. The *x-axis* represents the number of Floating Point Operations (FLOPs) required to process a single image. The *y-axis* represents the Top-1 accuracy on the ImageNet [25] dataset and the size of the circles represents the number of parameters in the network. Numbers are taken from [154]

2.4.3 Architectures Used in Experiments

In the subsequent paragraphs, we detail the architectures that we used in our experiments. We chose these architectures because they are representative of the state-of-the-art in image classification and they are widely used in the pruning literature. Table 2.1 gives an overview of the different network architectures.

VGG16. The VGG16 network [175] is a 16-layer CNN composed of 13 Conv layers and 3 fully connected layers. VGG16 was originally designed for ImageNet [25] and in our experiments with CIFAR-10 and CIFAR-100 (described in section 2.5) we use a slightly modified version of VGG16

	Conv2	Conv4	Conv6	VGG16	ResNet20	ResNet18
Number of Parameters	4,301,642	2,425,930	2,262,602	14,728,266	269,034	11,685,608
Number of layers	5	7	9	14	20	18
Number of Conv layers	2	4	6	13	19	17
Number of FC layers	3	3	3	1	1	1

Table 2.1: Number of parameters for the used neural network architectures. The number of parameters is given for the CIFAR-10 dataset, except for the ResNet18 architecture, whose number of parameters is given for the TinyImageNet dataset.

where we replace the 3 FC layers with an average pooling layer and a single FC layer [120]. The Conv layers filters are of size 3×3 with a stride of 1. The max-pooling layers are of size 2×2 with a stride of 2. Each Conv layer is followed by a ReLU activation function. The VGG16 network is illustrated in figure 2.10.

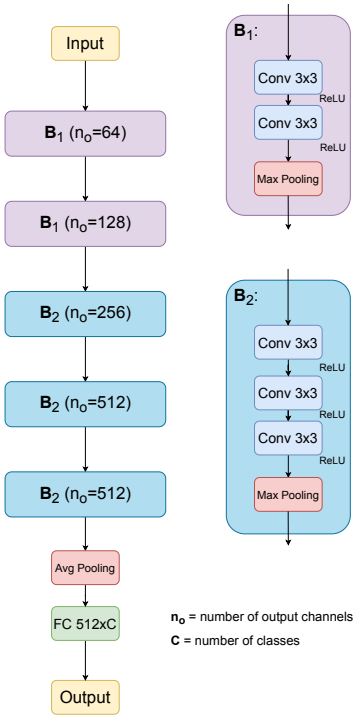


Figure 2.10: VGG16 adapted for CIFAR-10 and CIFAR-100.

ResNet{18,20}. The ResNet18 and ResNet20 networks [67] are 18 and 20-layer CNNs respectively. These layers are organized into stages, with ResNet18, represented in figure 2.11b, consisting of 4 stages with 2 *Basic Blocks* (detailed subsequently), while ResNet20, represented in figure 2.11a, is structured into 3 stages, each containing 3 *Basic Blocks*. The *Basic*

Blocks, also referred to as *Residual Blocks*, are composed of Conv layers (see figures 2.11a and 2.11b) and follow the principle of learning the residual function:

$$f(x) = h(x) - x \tag{2.22}$$

where $h(x)$ is the mapping usually learned by previous architectures such as VGG16. The representation of a residual block is given by equation (2.23) (see also figure 2.8):

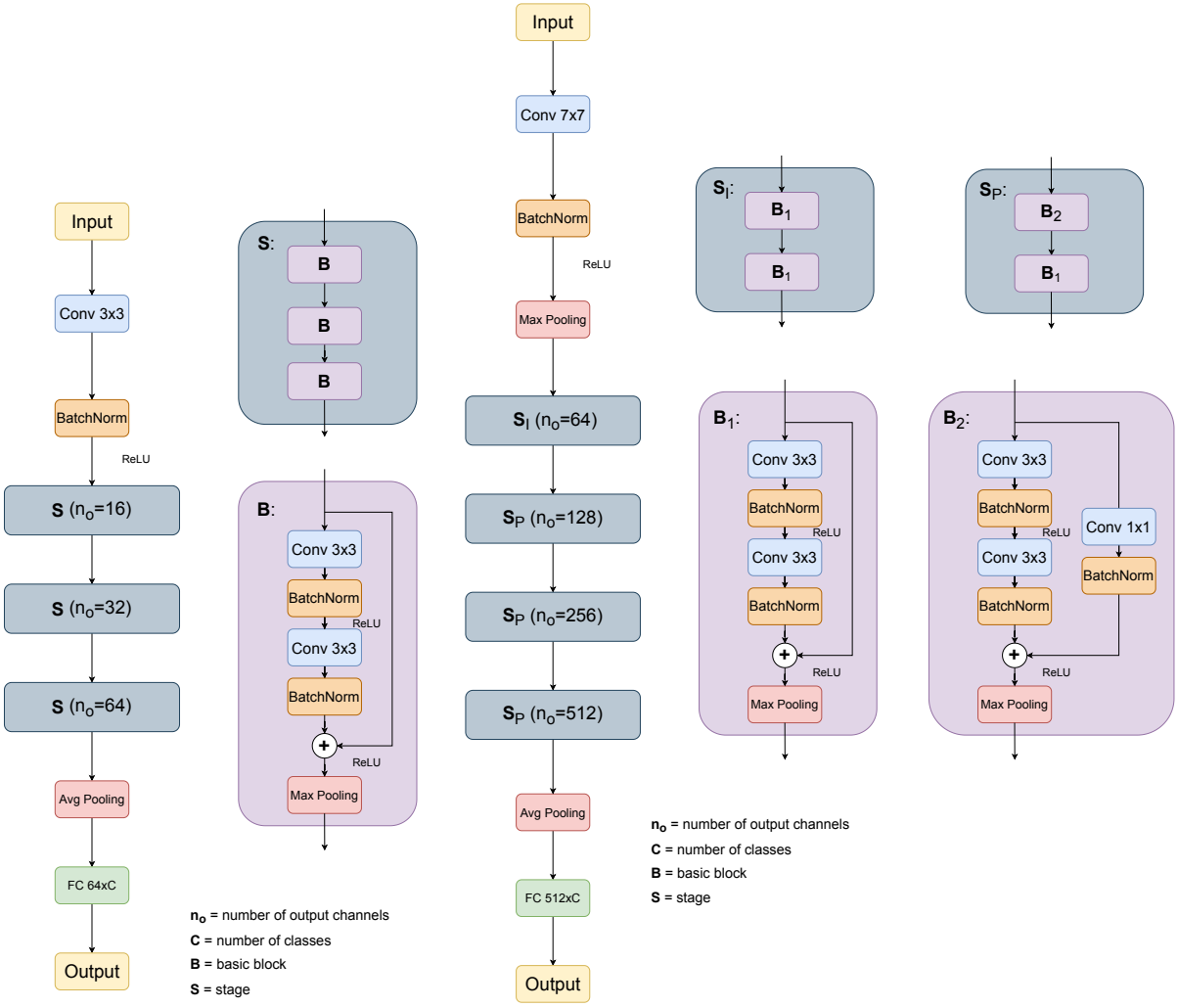
$$y = f(x, \theta) + x \tag{2.23}$$

where x is the input, f represents the residual function, θ are the weights of the block, and y is the output. In equation (2.23) $+x$ denotes the skip connection, which enables direct backpropagation of the gradient to earlier layers.

Conv{2,4,6}. Conv2, Conv4 and Conv6 are shrunk down versions of the VGG16 network architecture, composed of 2, 4 and 6 Conv layers respectively and 3 FC layers. Although Conv2, Conv4 and Conv6, introduced by Frankle and Carbin in [43], are not widely featured in existing literature, we chose to employ them due to their use in the methods we benchmark against. The Conv layers are stacked in increasing depth, and their convolutional filters are of size 3×3 with a stride of 1. The max-pooling layers are of size 2×2 with a stride of 2. They are represented in figure 2.12.

2.5 Datasets

In this thesis, we focus on image classification and supervised learning, a machine learning paradigm in which the model is trained using labelled data. In the context of image classification, the labelled data are pairs of images and labels which represent the class of their associated image. We denote an input image X and its corresponding label y. Each image X



(a) ResNet20 **(b)** ResNet18

Figure 2.11: ResNet20 and ResNet18 architectures. ResNet20 (figure 2.11a) is tailored for CIFAR-10 and comprises 3 stages encompassing 3 *Basic Blocks* of 2 Conv layers each, with an identity skip connection in each block. ResNet18 (figure 2.11b) is tailored for ImageNet and is composed of 4 stages encompassing 4 *Basic Blocks* of 2 convolutional layers each. There are two types of blocks: $\mathbf{B}_\mathrm{I}$ with an identity skip connection and $\mathbf{B}_\mathrm{P}$ with a projection skip connection. The projection skip connection is used to match the dimensions between the input and the output of the block.

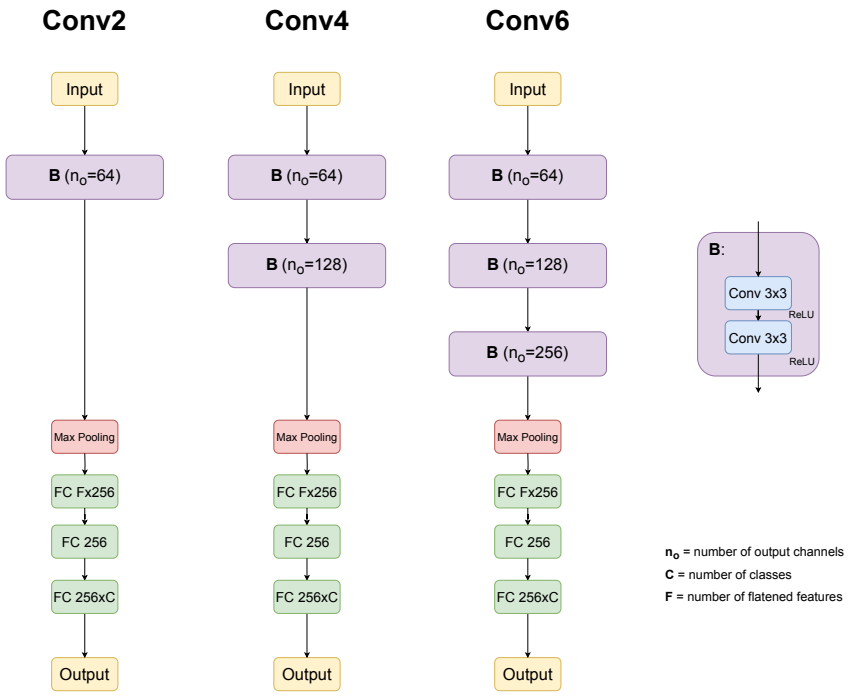


Figure 2.12: Conv2, Conv4 and Conv6 architectures. The number of flat features **F** corresponds to the size of the feature map of the last block **B**, once vectorised. **F** = 16384, 8192 and 4096 for Conv2, Conv4 and Conv6, respectively for input images of size 32×32.

belongs to the set of all images of the dataset $\mathcal{X}$, and each label y belongs to the set of all labels of the dataset $\mathcal{Y}$. The ensemble of the image-label pairs are gathered in a dataset, denoted $\mathcal{D}$, which is formally a set of pairs (X, y), where $X \in \mathcal{X}$ and $y \in \mathcal{Y}$, so that $\mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$.

Following these formal notations, the subsequent sections give details about the datasets used in our experiments. We evaluated our methods on three different datasets tailored for image classification: CIFAR-10 [162], CIFAR-100 [162] and TinyImageNet [104]. The following paragraphs give details about these datasets and table 2.2 sums up their main characteristics.

Dataset	Number of images	Number of classes	Image size	Size of test set
CIFAR-10	60,000	10	32x32	10,000
CIFAR-100	60,000	100	32x32	10,000
TinyImageNet	100,000	200	64x64	10,000

Table 2.2: The number of images, of classes, image size and size of the test set for the three datasets used: CIFAR-10, CIFAR-100 and TinyImageNet.

2.5.1 CIFAR-10

CIFAR-10 [162] is a widely used dataset in machine learning and computer vision. This is a labelled subset of the *80 Million Tiny Images* dataset [190]. CIFAR-10 is a simple yet challenging dataset that allows for quicker iteration or hyperparameter tuning than larger datasets such as ImageNet [170], but it is significantly more complex than the MNIST dataset [26], which contains grayscale handwritten digits images. The CIFAR-10 dataset contains 60,000 colour images of size 32x32 pixels, split into 10 classes, namely: plane, car, bird, cat, deer, dog, horse, ship, and truck. Each class contains 6,000 images. The dataset is divided into two subsets: a training set, composed of 50,000 images and a test set containing 10,000 of them.

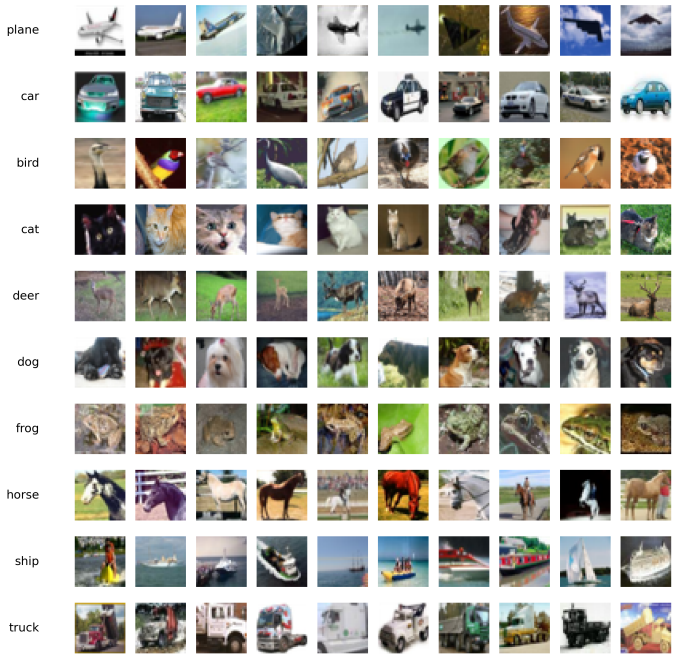


Figure 2.13: A sample of images from CIFAR-10. Each row contains images from one of the 10 classes: plane, car, bird, cat, deer, dog, frog, horse, ship, and truck

2.5.2 CIFAR-100

CIFAR-100 [162] is a more challenging version of CIFAR-10. Like the latter, it is a labelled subset of the *80 Millions Tiny Images* and is composed

of 60,000 colour images of size 32x32 pixels. However, instead of 10 classes, CIFAR-100 contains 100 classes of 600 images each. As a result, each class has far fewer images than in CIFAR-10. CIFAR-100 is also divided into two sets: a training and a test set, composed of 50,000 and 10,000 images respectively.

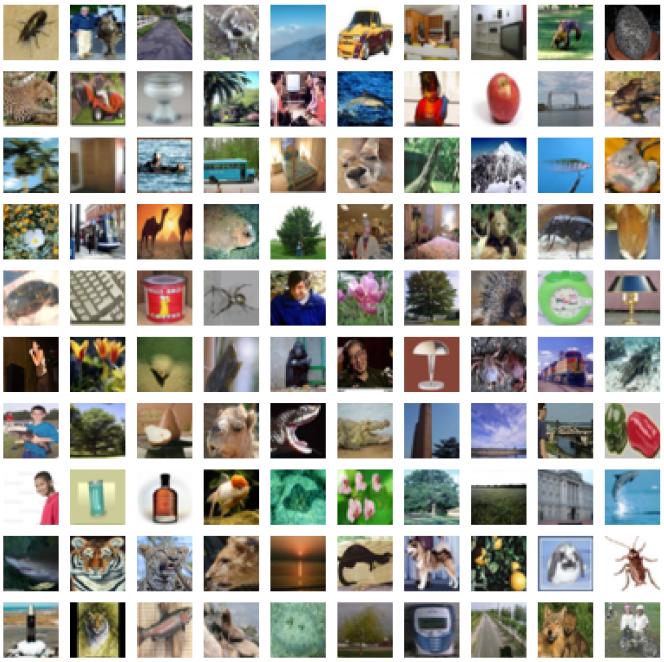


Figure 2.14: A sample of images from CIFAR-100. Each image represents an instance of one of the 100 distinct classes.

2.5.3 TinyImageNet

TinyImageNet is another popular dataset in machine learning and computer vision, conceived as a subset of the larger ImageNet dataset [170]. It comprises 100,000 colour images of size 64x64 pixels, split into 200 classes, whereas ImageNet contains 1.2 million images of size 256x256 pixels, split into 1,000 classes. The dataset is divided in 3 sets: the train set, which contains 500 images per class, the validation and test sets, which both contain 50 images. The scaled-down image size and the reduced number of images make TinyImageNet more computationally manageable than Ima-

geNet while still being challenging by offering diversity in the image classes.

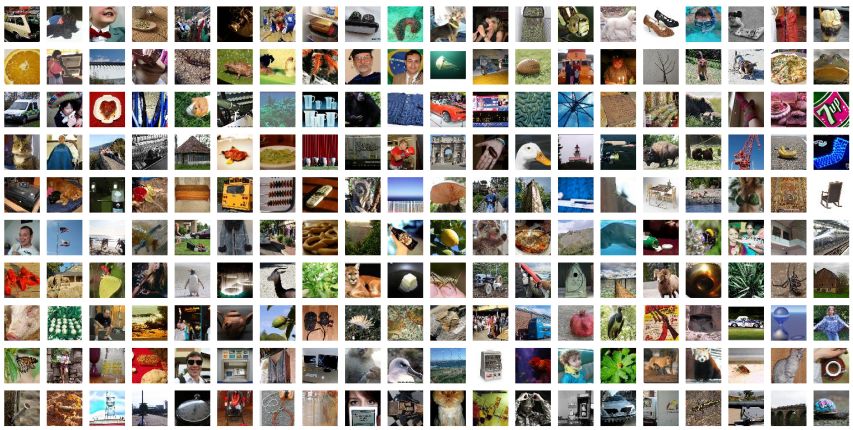


Figure 2.15: A sample of images from the Tiny ImageNet dataset. Each image represents an instance of one of the 200 distinct classes.

2.5.4 Train, Validation and Test Sets

In our experiments, for each dataset, we use 3 distinct sets for training, validation and testing. The training set serves to tune the weights of the model, while the validation set is used to monitor the evolution of the performance metric on unseen data throughout the training. The validation metric provides the necessary triggers for the early stopping policy (*i.e.* interrupting the training prematurely if the validation metrics do not change over a given number of iterations). The test set, on the other hand, is used to evaluate the model's performance on entirely new data and to report the final test accuracy. When utilizing datasets like CIFAR-10 and CIFAR-100, only training and testing sets are available. For these datasets, we split the given train set using the following proportions: 90% is used for training the network and the remaining 10% is for validation. On the other hand, the TinyImageNet dataset does provide training, validation, and testing sets, but the test set lacks annotations. Hence, we use 90% of the original training set for model training and the remaining 10% for validation. Instead of the original unannotated test set, we repurpose the original validation set to serve as the test set. This is a common strategy employed by other implementations [203, 195, 194].

Chapter 3

Deep Neural Network Compression

Contents

3.1 Introduction

The fast development of neural networks has led, on the one hand, to the enhancement of their performance, but also, on the other hand, to a significant growth in size and parameter count. The rapid evolution and adoption of these networks has given rise to various applications [102, 12, 174, 95], particularly embedded ones[99, 103], whose resources are highly

constrained in terms of computing power, energy consumption and memory footprint. Alongside the increase in the size of these networks, compression techniques [105, 60, 59] have been devised, in order to enable the use of these algorithms in embedded applications or resource-constrained environments.

This chapter focuses on state-of-the-art neural network compression methods, predominantly based on various operations applied to the weights of an already existing large neural network. This chapter is organised as follows: Section 3.2 examines fast convolution techniques, which aim to accelerate the computation of convolutions in neural networks, thereby reducing both the runtime and computational resources required. Thereafter, section 3.3 delves into Knowledge Distillation (KD), a process by which the knowledge of a larger, more complex network (referred to as the *teacher*) is transferred to a smaller and more efficient network (called the *student*), enabling the latter to achieve comparable performance with a reduced footprint. Subsequently, section 3.4 explores architecture design methods that aim at producing more efficient and effective networks. Section 3.4.1 details ad-hoc architectures, referred to as *Efficient Architectures*. These architectures are lightweight networks that revolve around a core technique to reduce their size while preserving performance as much as possible. Hence, section 3.4.2 discusses NAS, a method that automates the discovery of optimal network architectures tailored to specific tasks or constraints, potentially leading to more compact and efficient designs. Afterwards, section 3.5 presents two categories of techniques that harness an existing neural network and refine its architecture to produce a more compact and efficient model. First, section 3.5.1 focuses on *quantisation* and *binarisation* techniques, which aim to lower the numerical precision of weights and activations of networks in order to speed up their computation and reduce their memory footprint. Lastly, section 3.5.2 considers neural network pruning, which seeks to remove redundant or insignificant connections and weights from networks, resulting in sparser and more computationally efficient models.

3.2 Accelerating Computation in Neural Networks

Among various operations and functions used in neural networks, two fundamental mathematical operations, convolution and matrix multiplication are used extensively and are the backbone of most computations in neural networks. However, performing these operations can be computationally demanding, particularly with large and complex networks. This may lead to long and heavy computations, posing a challenge for real-time or resource-limited applications. To mitigate this issue, some research efforts have focused on developing techniques to speed up these operations. These strategies encompass optimizing the underlying algorithms to leveraging hardware acceleration, with the objective of enhancing the speed and efficiency of neural network computations.

3.2.1 Fast Fourier Transform

The most popular algorithms for accelerating convolution operations rely on the FFT [18, 116, 151], and leverage the Convolution Theorem. The Convolution Theorem states that the convolution of two signals in the source domain is the product of the two signals in the Fourier domain, as shown in equation (3.1):

$$F(x * y) = F(x) \cdot F(y) \tag{3.1}$$

where x and y are the two signals in the source domain, $x * y$ is the convolution of x and y and finally $F(x)$ and $F(y)$ are the Fourier transforms of x and y, respectively. Then, to obtain the result of the convolution in the source domain, the inverse Fourier transform, denoted F^{-1}, is applied as follows:

$$x * y = F^{-1}(F(x) \cdot F(y)) \tag{3.2}$$

The convolution theorem allows for faster computation of the 2D convolution by using the FFT to compute the convolution in the frequency domain, and the inverse FFT to convert the result back to the source domain [144].

3.2.2 Optimised Matrix Multiplication Algorithms

It is possible to accelerate matrix multiplication by directly optimising the underlying algorithm. The Strassen algorithm [180], used in [19], is a fast method for matrix multiplication that reduces the computational complexity from the standard $\mathcal{O}(n^3)$ to approximately $\mathcal{O}(n^{2.807})$ by recursively dividing the matrices of size n into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$, reorganising and combining these multiplications to perform only 7 instead of 8 matrix multiplications (see equations (3.5) and (3.6)).

Considering a matrix multiplication of two square matrices A and B of size $2n$ with $n \in \mathbb{N}$, defined in equation (3.3), the output of the standard bloc matrix multiplication, referred to as C, is defined in equation (3.4). Note that A_{ij} and B_{ij} are either a scalar if $n = 1$, or a matrix of size $\frac{n}{2} \times \frac{n}{2}$.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{3.3}$$

The computation of C requires 8 matrix multiplications, as shown in equation (3.4).

$$C = A \cdot B = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix} \tag{3.4}$$

The Strassen algorithm reduces the number of multiplications to 7 by defining the following 7 products, referred to as P_i, with $i \in [\![1; 7]\!]$:

$$
\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22})B_{11} \\
P_3 &= A_{11}(B_{12} - B_{22}) \\
P_4 &= A_{22}(B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12})B_{22} \\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
\tag{3.5}
$$

The result of the matrix multiplication is then obtained by combining these products, as shown in equation (3.6).

$$
C = \begin{bmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 - P_2 + P_3 + P_6 \end{bmatrix} \tag{3.6}
$$

The Strassen algorithm has later been refined by Coppersmith and Winograd, who introduced the Coppersmith-Winograd algorithm [20]. The latter brings down the complexity to $\mathcal{O}(n^{2.376})$. This algorithm is used in various works, mostly targeted towards a specific Field Programmable Gate Array (FPGA) processor [122, 129, 197].

3.2.3 Leveraging Matrix Structures

Using a particular matrix structure also speeds up the standard operations used in a neural network. Fully connected layers can be effectively accelerated by forcing the use of specific matrix structures. For instance, Cheng et al. devised a method where dense layers standard operation is replaced by a circulant projection [15]. The circulant matrix can be stored in a memory-efficient way and can be further sped up with FFT. C is an example of a circulant matrix (see equation (3.7)).

$$C = \begin{pmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{pmatrix} \quad T = \begin{pmatrix} a & b & c & d \\ e & a & b & c \\ f & e & a & b \\ g & f & e & a \end{pmatrix} \tag{3.7}$$

Likewise, convolutional operations can be accelerated thanks to Toeplitz matrices. A Toeplitz matrix, or diagonal-constant matrix, has the unique characteristic of each descending diagonal from left to right being constant. T is an example of a Toeplitz matrix (see equation (3.7)). This property is particularly useful for convolutions, as they can be expressed as a multiplication by a Toeplitz matrix [54], as shown in equation (3.8). This algorithm has been used in [114], focusing on FPGA architectures. Note that the representation of convolution as a product with a Toeplitz matrix can further be accelerated by using the aforementioned optimisations to the matrix multiplication algorithm, such as the Strassen or Coppersmith-Winograd algorithm.

Let x be a signal of length N and h be a kernel of length M, expressed as a Toeplitz matrix H. The convolution of x and h can be expressed as:

$$h * x = Hx = \begin{bmatrix} h_0 & 0 & \cdots & 0 \\ h_1 & h_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_{M-1} & h_{M-2} & \cdots & h_0 \\ 0 & h_{M-1} & \cdots & h_1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{M-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \tag{3.8}$$

3.2.4 Practical Applications and Limitations

The algorithms presented in sections 3.2.1 to 3.2.3 offer significant acceleration in the computation of convolution operations and are widely implemented and used in state-of-the-art software [154, 1]. In particular, the Coppersmith-Winograd algorithm is used in various Deep Learning frameworks [1, 147] or neural network GPU libraries [21] where the fastest algorithm is automatically selected based on the tensor sizes and the hardware. However, depending on the operand size, the total processing speed can be bound to the hardware and more specifically, to the memory throughput and data access speed, more than the computation time [200, 31].

3.3 Teaching Paradigm

The teaching paradigm embraces a class method that aims to transfer the knowledge of a large, complex and accurate network, referred to as the *teacher*, to a smaller and more efficient one called the *student*. The student is trained with a combination of the main task loss as well as a supplementary supervision signal which is derived from the feature maps of the teacher network at various depths.

3.3.1 Knowledge Distillation

Methods in the teaching paradigm are mostly based on the seminal work of Hinton et al. [74], better known as Knowledge Distillation (KD). The latter seeks to train simple networks with KD yielding better performances compared to those trained from scratch. KD relies on teacher and student networks, where the logits of the former are used as an additional supervision signal for the latter. When trained separately, the student network can only rely on classification labels in order to learn its own data representation while KD relies on the logits of the trained teacher network which provide more insight about the latent data representation.

For a classification problem, the loss used to train the student network with KD can be expressed as:

$$\mathcal{L}_{\text{total}} = \underbrace{\mathcal{L}_{\text{CE}}(\hat{y}_s, y)}_{\text{Task loss}} + \lambda \frac{T^2}{2} \underbrace{\mathcal{L}_{\text{CE}}\left(\frac{\hat{y}_s}{T}, \frac{\hat{y}_t}{T}\right)}_{\text{Distillation loss}} \tag{3.9}$$

where $\mathcal{L}_{\text{CE}}$ is the cross-entropy loss, $\hat{y}_s$ and $\hat{y}_t$ are the logits of the student and teacher networks respectively, y is the ground truth label, T is the temperature parameter and λ is a mixing coefficient balancing the two losses. Note that the distillation loss is scaled by $\frac{T^2}{2}$ to ensure that the relative contribution of the task loss and distillation loss stays balanced if the temperature changes.

3.3.2 Feature-Map Matching

Inspired by KD, [165] introduced FitNet, a two-stage training algorithm, where an intermediate layer of the teacher is chosen as a *hint*[1] for an intermediate layer of the student. Initially, the first layers of the student are trained to mimic the hint feature map. Then, the whole student network is trained with standard KD against the whole teacher. In the first step, a regressor is needed in order to adapt the dimensions of the feature map, which may differ from the teacher to the student networks, as illustrated in figure 3.1. Yim et al. argue that the direct feature map matching utilised by FitNets is overly restrictive. Drawing inspiration from the techniques used in [48] for style transfer, they propose an alternative method. In the context of style transfer, the Gram matrix of the feature maps is employed to encapsulate the texture information of an image. Adapting this approach, the method presented in [206] calculates the Gram matrix across the feature maps of multiple layers. This computed Gram matrix, dubbed as the Flow of Solution Procedure matrix, then serves as a *hint* for the student network, guiding its training process. In practice, handling full-dimensional feature maps is cumbersome. That is why, in order to avoid this issue, [209] use an attention map generated by squashing the feature maps to a 2D map allowing for a smaller 2D regressor to match attention map dimensions.

[1] *Hint* is the terminology used by Romero et al. [165] to denote a feature map used as a target for the student network.

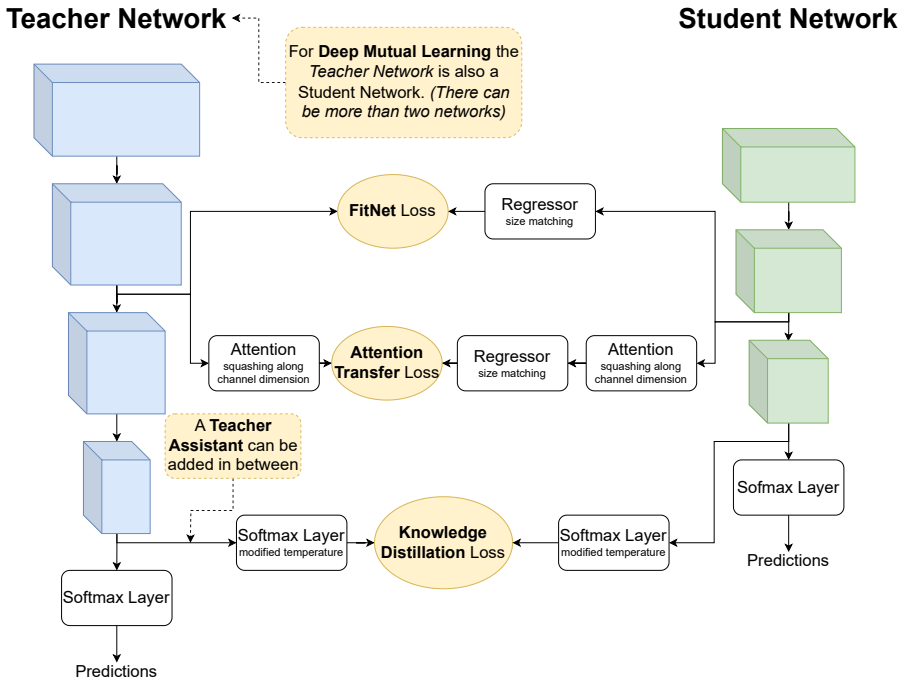


Figure 3.1: Overview of various knowledge distillation frameworks. From top to bottom, left to right: Deep Mutual Learning [212], FitNet [165], Attention Transfer [209], Teacher Assistant [137] and Knowledge Distillation [74].

3.3.3 Deep Mutual Learning

Note that the aforementioned knowledge transfer methods require teacher-student pairs and assume that teachers are large trained models. [212] relax this assumption by proposing *Deep Mutual Learning*, which enables a pool of networks of different architectures to learn together, provided that they have the same logit dimensions, and none of the models in the pool requires a pretraining step. The uncertainty of each model is distilled into each other, which creates additional knowledge.

3.3.4 Teacher Assistant

In all the aforementioned methods, the efficacy of knowledge distillation, and consequently, the final performance of the student network, is significantly influenced by the disparity in size between the student and teacher networks. This size discrepancy, when excessive, may cause the student network to encounter difficulties in aligning with the teacher logits, thus preventing optimal knowledge distillation. To tackle this issue, Mirzadeh et al. introduced the concept of *Teacher Assistant*: networks of intermediary dimensions aiming at bridging the size gap between student and teacher

[137]. The Teacher Assistant (TA) approach proposes to ensure effective knowledge transfer through a stepwise transfer of knowledge, starting from the teacher to the TA, and finally from the TA to the student. This technique allows each model to learn from a slightly simpler model than itself. Empirical evidence shows that the TA approach tends to outperform traditional one-step distillation in various experiments and across different network architectures, resulting in improved performances. However, it is important to note that it does introduce additional computational overhead due to the necessity of additional training steps for the TA, and careful selection of the size and number of TAs. These considerations underscore that while the TA strategy is effective in managing the size disparity problem, it also adds complexity to the distillation process.

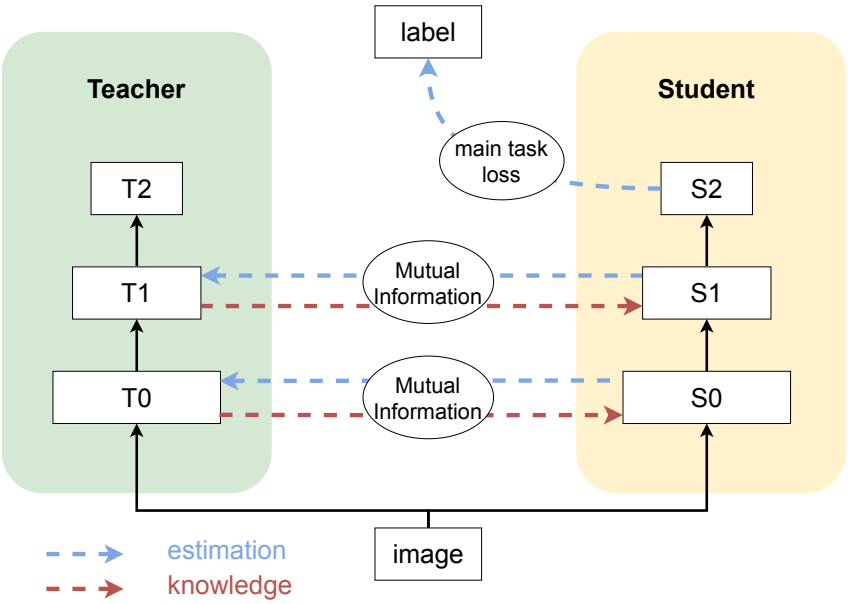


Figure 3.2: Conceptual scheme of [2]. The student network efficiently learns the main task while retaining high mutual information with the teacher network. The mutual information is maximised by learning to estimate the distribution of the activations in the teacher network, provoking the transfer of knowledge. Adapted from the original scheme found in [2].

3.3.5 Alternative Distillation Losses

Other approaches that do not rely on direct feature map or logit matching have been proposed. [2] introduced *Variational Information Distillation*, which indirectly maximises the mutual information between the student and the teacher. This is done by using *variational information maximisation* [8] to maximise a variational lower bound of the mutual informa-

tion, since directly maximising the latter is intractable in practice (see figure 3.2). Likewise, [146] proposed a *Probabilistic Knowledge Transfer* method that does not match logits or feature maps, but rather represents the latter as a probability distribution and minimises divergence between the two (see figure 3.3).

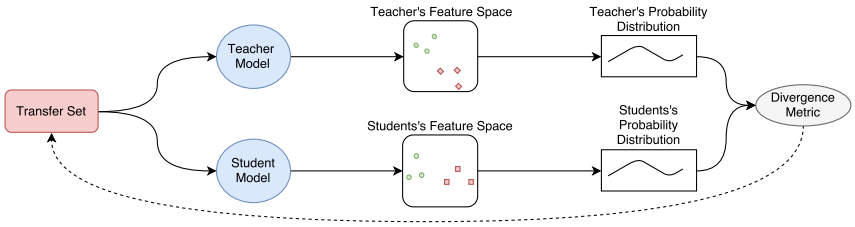


Figure 3.3: Conceptual scheme of the Probabilistic Knowledge Transfer method. Both the student and the teacher feature maps are modelled using probability distributions. The divergence of the latter is minimised in order to transfer knowledge from the teacher to the student. Illustration taken from [146].

3.4 Architecture Design

The architectural design of neural networks, while contributing significantly to their performance, often inflates their computational and memory requirements. This increased complexity, although beneficial for the final performance, could limit the deployment of these networks in resource-constrained environments. Thus, formulating effective and efficient neural network architectures is of significant importance. The design of neural networks is a problem that not only involves designing suitable building blocks but also determining their organization and interconnections. This section scrutinizes these aspects by focusing on handmade and automatic efficient architecture design.

Section 3.4.1 introduces building blocks to design efficient architectures. These building blocks have been meticulously engineered in the state-of-the-art to strike a balance between computational efficiency and performance. Properly incorporating these blocks can result in architectures better suited to their operating environments, enhancing efficiency while maintaining the desired level of performance.

Thereafter, section 3.4.2 delves into the field of Neural Architecture Search. The primary aim of NAS is to design, in an automatic fashion,

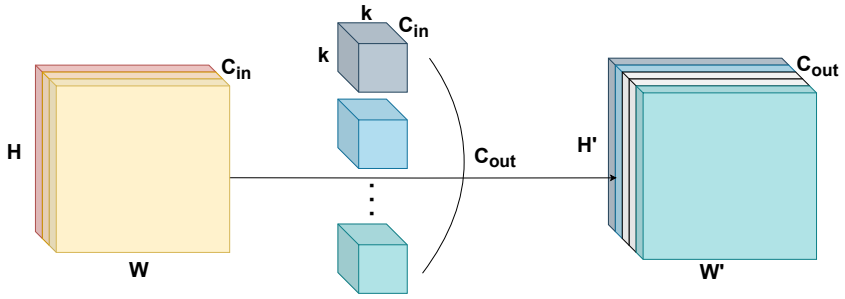
network architectures that demonstrate a high level of efficiency and performance for a given task. By doing so, it eliminates the need for manual design and the associated iterative trial-and-error approaches that would otherwise be necessary to assess and evaluate the impact and effectiveness of each design decision [79, 171, 80]. Although NAS was not initially targeted at generating lightweight architectures, the principles and methods described in this section can be adapted to optimise the architecture search for efficiency and compactness.

This section explores techniques aimed at the creation of efficient and effective neural networks through the careful selection and assembly of optimised building blocks. The organization of these components plays an important role in network compression and optimization, highlighting that high performance can also be reached with designs that are less resource-demanding.

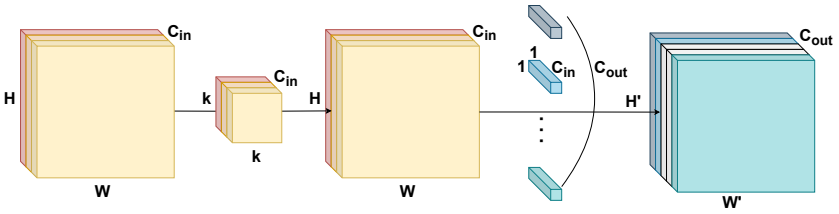
3.4.1 Building Blocks for Efficient Architecture Design

Depthwise Separable Convolutions. One of the initial strategies towards achieving efficiency in neural network architectures is the use of depthwise separable convolutions. This technique, used in MobileNet [81] and EfficientNet [184], separates the standard convolution operation into two distinct steps: a depthwise convolution and a pointwise convolution (see figure 3.4). By decomposing the operations in this manner, the computational complexity is markedly reduced while still retaining the ability to capture spatial and channel-wise information. Consider an input feature map with C_{in} channels of arbitrary width and height and C_{out} convolution kernels of size $k \times k \times C_{\text{in}}$. A standard convolution algorithm will need $C_{\text{in}} \times C_{\text{out}} \times k \times k$ Multiply-Accumulate (MAC) operations to produce a $1 \times 1 \times C_{\text{out}}$ element of the output feature map. In contrast, a depthwise separable convolution algorithm will first apply a $k \times k \times 1$ convolution kernel to the C_{in} channels and then perform C_{out} pointwise convolutions with $1 \times 1 \times C_{\text{in}}$ kernels to produce the same $1 \times 1 \times C_{\text{out}}$ element. This effectively reduces the number of parameters to $C_{\text{in}} \times (C_{\text{out}} + k \times k)$, essentially reducing the number of computations required to produce a $1 \times 1 \times C_{\text{out}}$ element by a factor of

$$\frac{C_{\text{out}} \times k \times k}{C_{\text{out}} + k \times k}.$$



(a) Standard Convolution



(b) Depthwise Separable Convolution

Figure 3.4: Illustration schemes of the standard and depthwise separable convolution. The standard convolution uses C_{out} kernels of size $k \times k \times C_{\text{in}}$. The depthwise separable convolution is split into two steps: *(i)* a convolution with C_{in} kernels of size $k \times k$ and *(ii)* a convolution with C_{out} kernels of size $1 \times 1 \times C_{\text{in}}$. Best viewed in colours.

Fire Module. An alternative approach for designing efficient architectures involves the integration of *fire modules*, as proposed in [89]. These modules, represented in figure 3.5, aim to minimise computational requirements by employing two distinct strategies: *(i)* diminishing the number of input channels supplied to the following conventional $k \times k$ convolutions and *(ii)* substituting a portion of the resource-intensive $k \times k$ convolutions with pointwise convolutions, which possess k^2 times fewer parameters. The initial strategy is applied within the *Squeeze Layer* of the *fire module*, which decreases the number of input channels delivered to the *Expand Layer*, subsequently reducing the number of parameters in the *Expand Layer* kernels. The second strategy is implemented in the *Expand Layer*, where some 3×3 convolutions are replaced with 1×1 variants. Although the 1×1 convolutions capture less spatial information, they are significantly

less computationally demanding than the 3×3 ones.

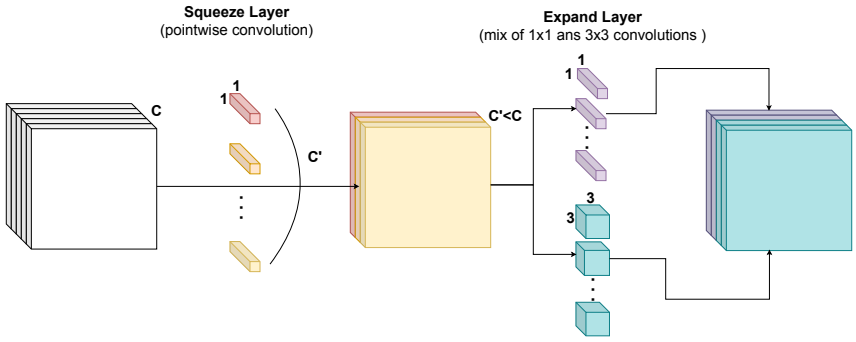


Figure 3.5: Illustration scheme of the fire module. The fire module is composed of a *squeeze layer* (pointwise convolution designed to reduce the number of channels fed to the following layer) and an *expand layer* (convolution with mixed 1×1 and 3×3 kernels. The 1×1 kernels replace some of the 3×3 kernels, being less computationally intensive.). Best viewed in colours.

ShuffleNet. Pushing the concept of depthwise separable convolutions further, [211] introduces pointwise group convolutions and channel shuffle operations to enhance efficiency while maintaining accuracy. Pointwise group convolutions were initially introduced in [102], though their original purpose was not for compression. Instead, group convolutions in [102] were used to enable distributed training across multiple GPUs with limited memory. However, ShuffleNet [211] leverages this concept for network efficiency by dividing the input channels into groups and performing convolutions on each group independently. This approach reduces the number of operations and the computational cost compared to traditional convolutions. To counteract the potential loss of expressive power caused by the separation of channels into groups, ShuffleNet incorporates *channel shuffle operations* as shown in figure 3.6. This technique allows for information exchange between groups, effectively maintaining accuracy by ensuring that different groups can capture diverse features in the input.

Learned group convolutions. Following ShuffleNet, CondenseNet was introduced in [86], incorporating learned group convolutions to further enhance efficiency. Unlike the predefined group convolutions in ShuffleNet, CondenseNet learns which channels should be grouped together, enabling the network to adapt its structure for a specific task. This results in better utilisation of network capacity and reduces redundancy. CondenseNet leverages the DenseNet architecture [85] to further improve performance. Thanks to the densely connected architecture, features discarded in any

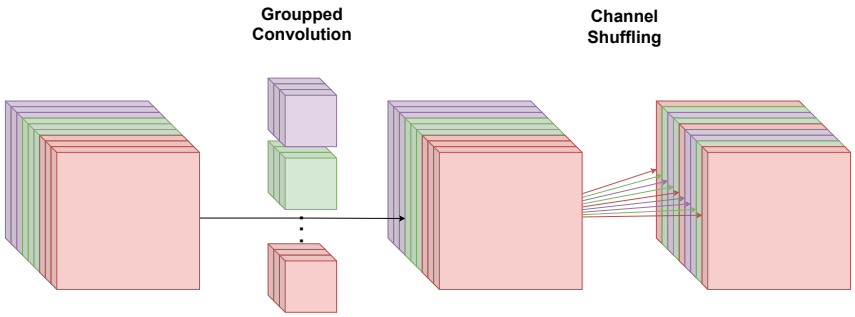


Figure 3.6: Illustration scheme of grouped convolution with channel shuffling. Each filter only acts on a subset of the input tensor (here represented by a matching colour). The channels of the yielded tensor are shuffled to ensure the subsequent groups can access information from all the previous groups. Best viewed in colours.

layer can still be recovered in subsequent ones.

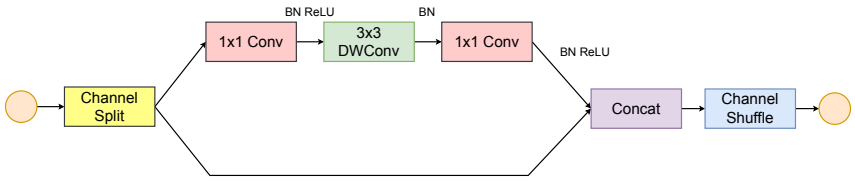
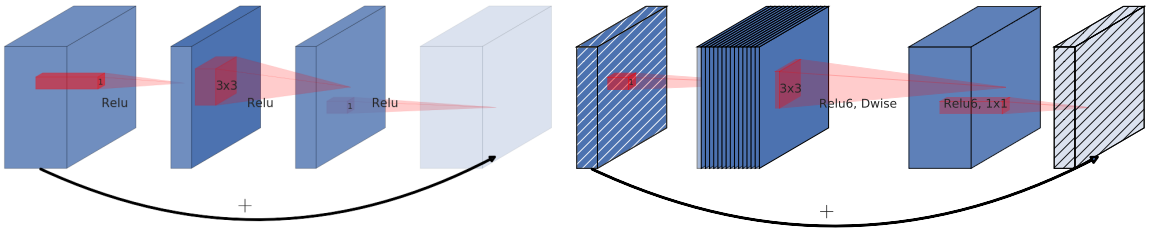


Figure 3.7: Illustration scheme of the path taken by the feature maps after the channel split block. Adapted from the original scheme found in [131].

ShuffleNetV2. Building on the success of ShuffleNet, ShuffleNetV2 was introduced in [131], focusing on enhancing network efficiency through the combination of strided convolution and channel split. Strided convolution helps to reduce the spatial extent of feature maps, thereby reducing the computation cost. The Channel Split technique efficiently processes the input feature maps while maintaining the expressive power of the architecture. Channel Split works by dividing the input feature maps into two equal parts. One part is passed through the main branch of the ShuffleNet unit, while the other part is sent through the identity branch, which leaves its input unchanged. In the main branch, a sequence of pointwise and 3×3 convolutions are performed. After both the main branch and the identity branch complete their respective operations, the two parts are concatenated along the channel dimension and the channels are shuffled. Finally, the output feature maps are passed to the next ShuffleNet unit in the network. This process is represented in figure 3.7. This approach balances computational efficiency with the expressive capacity of the model.

Inverted residual and Linear bottlenecks. Depthwise Separable Convolutions were employed in MobileNet [81]. Sandler et al. introduced skip connections and residual blocks into the MobileNetV2 architecture [171], initially proposed in [67]. They also introduced the concept of inverted residuals and linear bottlenecks. In conventional residual blocks, the input is first compressed, then expanded, and finally compressed again after being added to the original input. With inverted residual bottlenecks, on the other hand, this process is reversed: the input is first expanded, then a depthwise separable convolution is applied, and finally, it is compressed again. In this architecture, the skip connections link the feature maps of smaller size, instead of the larger ones. This allows for a more memory-efficient architecture. The standard residual blocks and the inverted residual blocks are shown in figure 3.8. The linear bottlenecks, on the other hand, are convolutions with a linear activation function. This takes advantage of the property that high-dimensional feature maps can be embedded in a lower-dimensional manifold. To do this, it is necessary to use linear transformations since non-linear ones could potentially destroy information as reported in [171, 58].



(a) Standard Residual Block **(b)** Inverted Residual Block

Figure 3.8: Illustration scheme of the residual block and the inverted residual block. Note that on the inverted residual block, the feature maps with the lower number of channels are the ones connected via the skip connection, whereas it is the opposite on the standard residual block. Diagonally hatched layers do not use non-linearities. The grey colour indicates the beginning of the next block. Both illustrations are taken from [29]. Best viewed in colours.

Squeeze-and-Excitation modules. Advancing from MobileNet and MobileNetV2, its third version [80] incorporated Squeeze-and-Excitation modules initially introduced in [83]. These modules adaptively recalibrate channel-wise feature responses, amplifying important features and suppressing less relevant ones. The Squeeze-and-Excitation module (represented in figure 3.9) performs *squeeze* and *excitation* operations. The squeeze operation uses global average pooling to create a channel descrip-

tor that summarises the spatial information for each channel. The excitation operation uses this descriptor to learn non-linear interactions between channels through two fully connected layers. The outputs of this mini-network are per-channel modulation weights that recalibrate the original feature maps, scaling or "exciting" them by these weights.

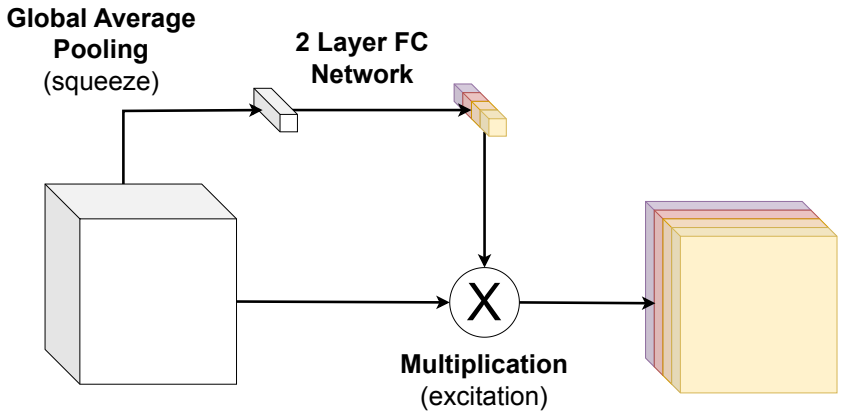


Figure 3.9: Illustration scheme of the Squeeze-and-Excitation module. The original feature map is *squeezed* into a channel descriptor through global average pooling. This descriptor is then used to learn the interdependencies between the channels through two fully connected layers. The output is then multiplied layerwise with the original feature map (*excitation*). Best viewed in colours.

The architectures we reviewed in this section revolve around specific key techniques such as depthwise separable convolutions, fire modules, channel shuffling, and Squeeze-and-Excitation modules, among others. These architectures, while highly efficient, are manually crafted and require a significant degree of human expertise, intuition, and time to develop, optimise, and fine-tune. The manual design of these architectures often relies on a deep understanding of the tasks at hand, the data they will process, and the constraints of the environment in which they will operate. However, the process of designing these efficient architectures can be automated, which is the subject of the next section. Sizes and performance of network architectures detailed in this section can be compared to standard architecture sizes in figure 3.10.

3.4.2 Automatic Architecture Design Through Neural Architecture Search

Neural Architecture Search (NAS) is a method that automates the discovery of neural network architectures, potentially leading to more compact,

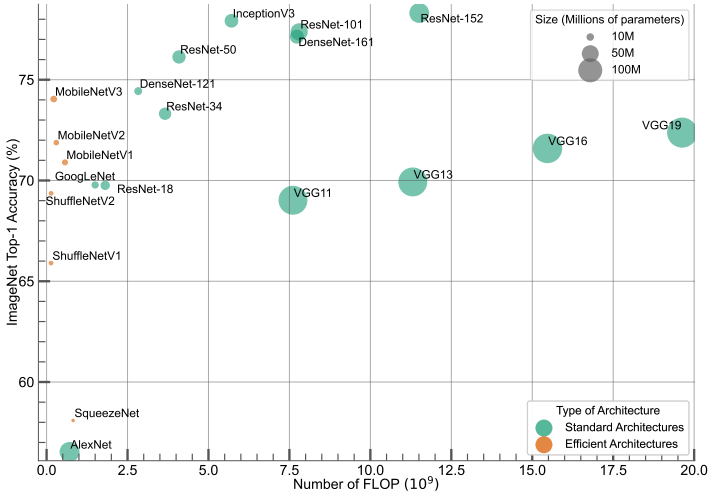


Figure 3.10: Figure 2.9 updated with the size and performance of the efficient architectures detailed in section 3.4.1. Best viewed in colours.

efficient designs and reducing the need for manual intervention. Although NAS might not explicitly aim at producing lightweight architectures, it can still yield designs that strike a good balance between performance and computational cost [185, 184]. By using automated methods to search for optimal architectures, it is possible to further enhance the efficiency of neural networks, opening up new possibilities for their deployment in resource-constrained environments. NAS has emerged as an essential paradigm, aiming to automate the traditionally manual and labour-intensive process of designing efficient neural networks [136]. Early network architectures were indeed entirely handcrafted, requiring significant human effort and expertise. However, these manual methods are being replaced by NAS techniques, which seek to automatically determine the optimal network structure given a training set [201, 38].

The performance and efficiency of NAS are fundamentally determined by two key aspects: the *search space* and the *search strategy*. The search space, as the name implies, defines the set of all possible architectures that can be discovered by the NAS algorithm. It could be as broad as all possible configurations of a certain type of network, such as CNNs, or as narrow as different arrangements of a specific set of layers [118]. The search strategy, on the other hand, determines how the NAS algorithm navigates through this search space in order to optimise its given objective. This could involve gradient-based strategies [119, 204], or stochastic methods, such as

evolutionary algorithms and reinforcement learning [219, 158]. The choice of search space and search strategy significantly influences the ability of NAS to discover effective and efficient architectures and is thus a critical aspect of NAS research. In the following paragraphs, we will delve deeper into some of the major strategies and their impact on the field of NAS.

Search space. The search space is a critical aspect of NAS as it bounds the possibilities of architectures and significantly influences the outcome of the search. The search space could be as broad as all possible configurations of a certain network type or as specific as various arrangements of a predefined set of layers or blocks. For instance, [219] define their search space as a set of repeatable sub-structures composed of basic layers (convolution layers, fully connected layers, BN layers, etc...) often called *cells* that are stacked to form the final architecture, while [202] design their search space based on the connectivity patterns between network blocks. DARTS [119] propose a continuous search space where the architecture is parameterized as a differentiable function, allowing for efficient search using gradient-based methods. Hierarchical search spaces, on the other hand, offer a strategic approach which allows to manage the complexity of the architecture search in NAS [118, 185]. In such a setup, the architecture is divided into several levels of hierarchy, with each one searched independently. This structure enables a more systematic and organized exploration of the search space, allowing the algorithm to uncover useful patterns and configurations at different levels of the network. The EfficientNet models are exemplary of innovative architecture search strategies [184]. This series utilizes both NAS and *compound scaling*. A baseline, EfficientNet-B0, was developed through multi-objective NAS, optimizing both accuracy and Floating Point Operations (FLOPs). Subsequently, a compound scaling method was applied to this baseline, uniformly scaling depth, width, and resolution via a *compound coefficient*. This approach yielded a series of progressively larger EfficientNet models, whose performances are shown in 3.11.

Search strategy. The search strategy is another major component of NAS, dictating how the algorithm explores the search space to find the optimal architecture. A wide range of search strategies have been proposed. Evolutionary algorithms [158] use principles of natural evolution such as mutation, crossover, and selection to explore the search space. Despite their potential to find high-quality solutions, these methods often require

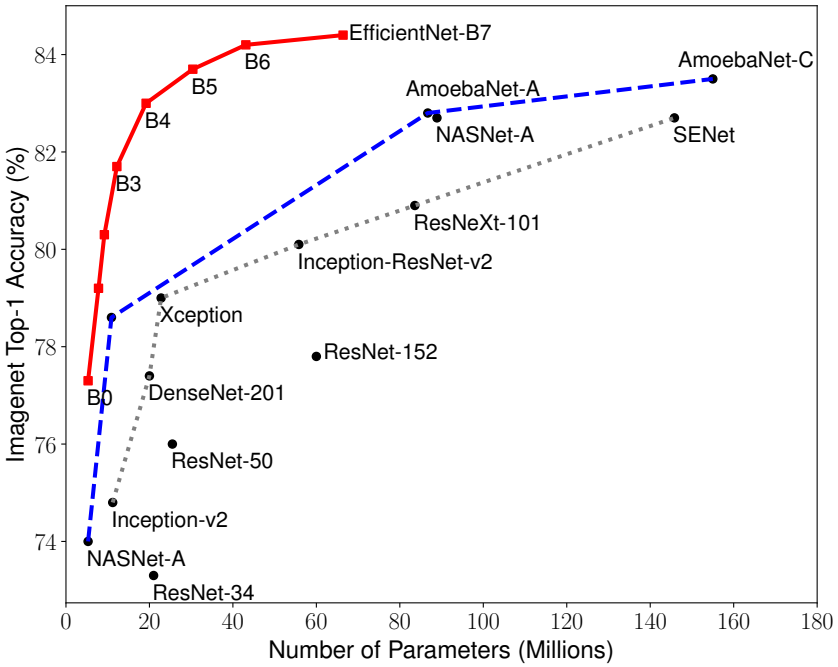


Figure 3.11: ImageNet top-1 accuracy vs model size (in millions of parameters). The EfficientNet family of models significantly outperforms other models of similar size, obtained either by NAS or manual design. This graph is taken from [184].

substantial computational resources due to the large number of evaluations needed. Reinforcement Learning-based methods [219] employ a policy network to generate architectures and a reward signal, typically validation accuracy, to guide the search. While reinforcement learning methods can effectively navigate large search spaces, their success heavily depends on the quality of the reward signal. Gradient-based methods [119, 204] make the search space continuous and use gradient descent for optimization, which enables efficient exploration of the search space but requires careful regularization to prevent overfitting. [10] uses Bayesian optimization to build a probabilistic model of the objective function and uses it to select promising architectures, balancing exploitation and exploration. This method can be sample-efficient but might struggle with high-dimensional spaces. These diverse strategies offer multiple paths to navigate the complex landscape of architecture search, each with its unique compromises between efficiency, effectiveness, and computational demands.

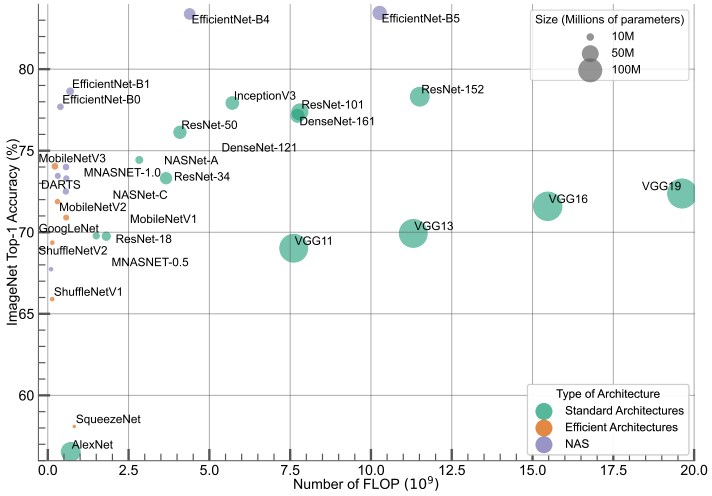


Figure 3.12: Figure 3.10 updated with the size and performance of architectures detailed in section 3.4.2. Best viewed in colours.

3.5 Compressing and Optimising an Existing Architecture

While the prior sections have primarily focused on constructing new, efficient network architectures and mechanisms for automatic architecture discovery, this part of the chapter transitions towards compressing and optimising existing neural networks. The methods discussed in this section, namely *quantisation*, *binarisation* and *pruning*, are specifically geared towards leveraging and enhancing already existing architectures or trained models. Instead of developing a new architecture, these techniques seek to make an existing architecture more efficient by modifying its weights and connections.

Section 3.5.1 delves into the methodologies of quantisation and binarisation. These methods endeavour to reduce the numerical precision of weights and activations in a network, without a significant drop in overall performance. This process can significantly speed up computations and decrease memory usage, contributing to the increased efficiency of a pre-existing network, especially in environments with limited hardware or memory resources.

Subsequently, section 3.5.2 examines the application of pruning techniques. Pruning refers to the elimination of redundant or insignificant

weights and connections in a network, leading to a sparser and more effective architecture. Pruning an existing network can further enhance efficiency by reducing the computational resources needed with a minimal or controlled impact on the performance.

Through these methods, this section aims to demonstrate how the effectiveness of existing neural networks can be optimised, thereby offering another viable path towards generating more efficient models without creating new architectures from scratch.

3.5.1 Lower Precision Weights and Activations Representation

Quantization is the process of converting continuous, high-resolution input values into a lower-resolution and typically discrete representation. Historically, the training of neural networks has largely relied on the use of single-precision floating-point format (FP32). FP32 has been the default choice due to its wide support across various hardware platforms and software libraries, which has made it a practical and convenient choice for the majority of machine learning tasks [182]. However, using single-precision floating-point format is not always necessary, and it is possible to constrain neural networks to use lower precision values, effectively quantising its parameters or feature maps, while maintaining compelling performances. Quantising a neural network can result in a reduced memory footprint as well as faster computation if the operations are implemented to leverage the specificity of the quantisation or paired with appropriate hardware.

Quantising a neural network has been proposed as early as the 1990s [7, 40]. This later regains traction as Vanhoucke et al. leveraged Single Input Multiple Data instructions (SIMD) of x86 processor to speed up the fixed-point 8-bit operations [192]. Gupta et al. [57] used uniform quantisation with fixed-point 16-bit representation and stochastic rounding to train neural networks. Quantisation has also been applied together with K-means clustering [179]. [60] uses K-means clustering to iteratively compute a lookup table or *code book* for the weights. This codebook is later further compressed using Huffman coding [88]. Note that this method is mostly useful for storage, but for training or inference, the weights need to be decompressed and their original value fetched in the code book before

being used.

Logarithmic quantisation. Logarithmic quantisation provides compelling alternatives to uniform quantisation. On the one hand, logarithmic quantisation enables quantising weights with a larger dynamic range compared to uniform or linear quantisation. On the other hand, multiplication can be conveniently represented as an inexpensive bit shift operation if operands are properly represented in the logarithmic base. This is particularly beneficial for FPGA implementations [3]. To leverage this potential speedup, [117] forced the weight representation to be a power of two and [214] improved this technique by applying it iteratively.

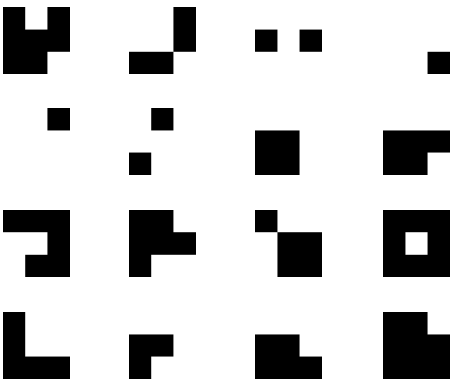


Figure 3.13: Example of binarised kernels and activations in a convolutional layer. The kernels are taken from the first layer of a CNN trained on CIFAR-10. Image taken from [87].

Binarisation. A more extreme version of the quantisation has been proposed in [23], where the weight values are either -1 or $+1$. The concept of minimising the bit-width of weights to a bare minimum is called *binarisation*. This allows for a dramatic simplification of the computation in the neural network at the expense of a drop in performance. Binarisation has been further developed in [87], where the authors proposed a method to binarise both weights and activations (see examples of binarised kernels in figure 3.13). DoReFa-Net [216] built upon the success of binarised neural networks and introduced the stochastic 8-bit quantisation of the gradients during the backward pass to accelerate both training and inference.

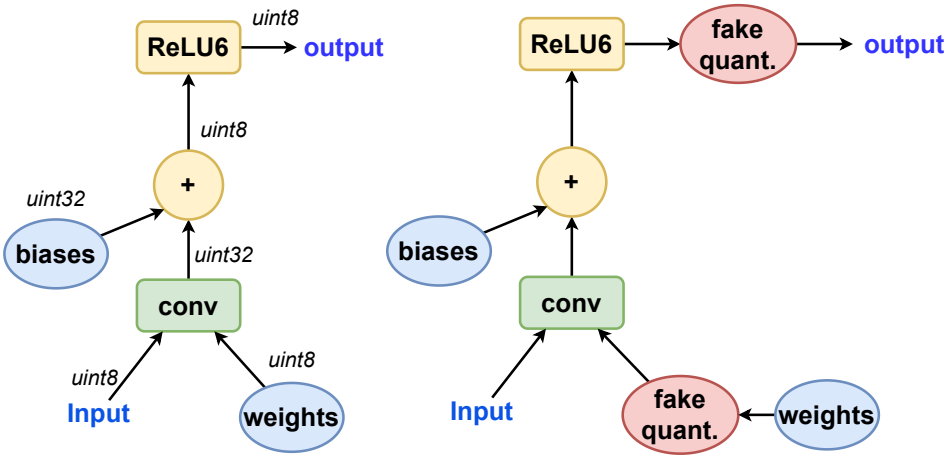
When to quantise. Quantisation methods that quantise weights or activations after the training are called Post-Training Quantisation methods. Quantising an already existing network is a widely used technique in the

most famous deep learning framework [187, 33, 155, 143]. Because they quantise the weights after the training, these methods are fast and easy to apply. However, they often introduce an irreversible information loss and a performance drop that needs to be compensated for [112]. In order to solve this issue, several works proposed to take into account the effect of quantisation on the weights and feature maps during the training. These methods are called Quantisation-Aware Training methods. Such methods include BinaryConnect [23], which use a variant of Bayesian inference called Expectation Back Propagation [17, 176]. Another binarisation method uses Straight Through Estimator (STE) [9] to bypass the binarisation function in the backward pass [87]. STE is also employed for quantisation in [91] which uses it together with *fake quantisation* nodes for 8-bit quantisation (see figure 3.14). The fake quantisation nodes are injected inside the computation graph and simulate the effect of quantisation in the forward pass.

Quantisation and binarisation are solutions to compress and accelerate neural networks. The potential of these techniques is vast, as they offer significant reductions in memory usage and enhanced computational speed when implemented correctly and paired with suitable hardware. However, these benefits are not without their drawbacks. On the downside, such techniques introduce a certain degree of error which can result in a performance drop, especially if not properly managed during the training process. This information loss is particularly notable in the case of Post-Training Quantisation methods, which necessitate additional efforts to mitigate these performance impacts. To address this, Quantisation-Aware Training methods have been developed, which incorporate the effects of quantisation during the training phase itself. The more extreme approach, binarisation, further accentuates the advantages and disadvantages observed in quantisation. While it offers extreme compression of neural networks, this often comes at the cost of significant accuracy loss.

3.5.2 Removing Weights and Connections

Lightweight neural networks can be obtained from a larger network through pruning. Pruning is the process of removing weights or connections, identified as redundant or unnecessary, while limiting to a minimum the impact on the performance of the network. The identification of the latter, often



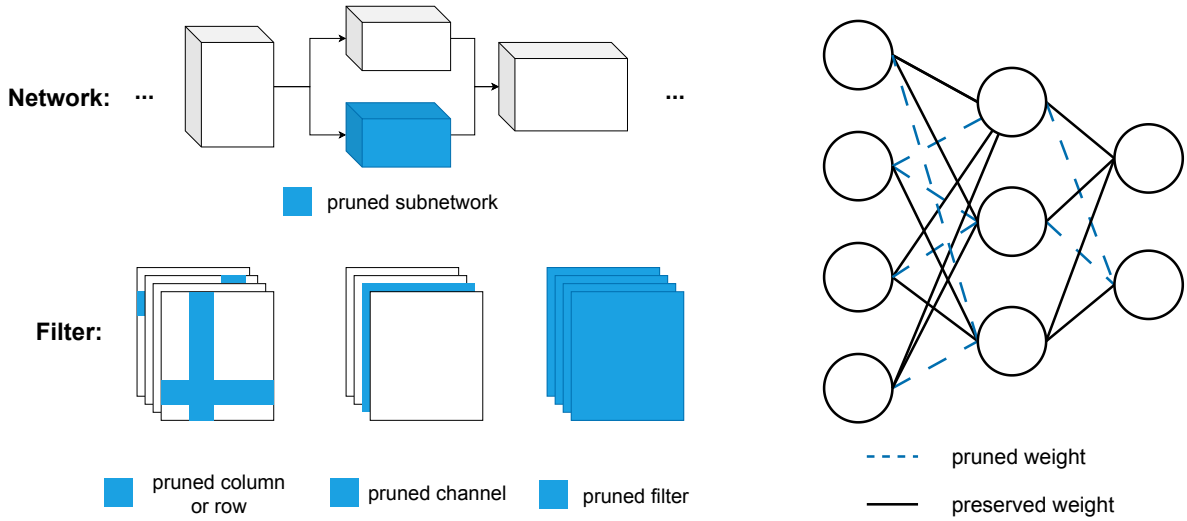
(a) Fake quantisation inference

(b) fake quantisation training

Figure 3.14: Fake quantisation nodes (*fake quant.*) are included in the computation graph of figure 3.14b, whereas figure 3.14a represent the computaion graph used during inference. During the inference, weights are stored in `uint8` format, whereas the bias are not, because their computational overhead is negligible.[91]. Both illustrations are adapted from [91].

referred to as determining the *saliency* of weights, has been a hot spot in the pruning literature [111, 16, 113]. Pruning a neural network removes weights and consequently reduces the theoretical computational complexity of the network as well as its memory footprint. The fraction of weights removed during pruning is often denoted as the *pruning rate*, which is commonly defined as the fraction of the number of weights removed from the original network over the number of initial weights in that network. Arguably, the first pruning method, introduced in 1989, was based on biased weight decay [62]. In the following years, LeCun et al. proposed a pruning method entitled Optimal Brain Damage [105] that used the Taylor expansion of the loss hessian matrix to identify the weights whose removal would have the least impact on the loss. This method, and in particular the computation of the hessian matrix was refined in Optimal Brain Surgeon [63, 65, 64]. As neural networks have become larger and more computationally intensive (see section 2.4.2 and figure 2.9), pruning has been receiving increased attention as a method to compress the latter. Pruning methods can be classified into two categories: *structured* and *unstructured*.

Structured pruning. Structured pruning involves the removal of entire structural components of the network, such as rows, columns, channels, filters, layers or even whole subnetworks (note that pruning a channel in layer $\ell + 1$ implies pruning a filter in layer ℓ, and vice-versa). This type of



(a) Structured pruning

(b) Unstructured pruning

Figure 3.15: Conceptual illustrations of structured and unstructured pruning.

pruning results in regular[2] network structures that are easier to exploit on typical hardware and do not necessitate a specific sparse computing library or hardware, making it an attractive approach for practical deployment. To some extent, structured pruning can be seen as a subcategory of Neural Architecture Search (section 3.4.2), where the search space would be the structure of the network to be pruned. Structured pruning leads to reduced computation complexity as well as significant memory footprint reduction, however, it also presents unique challenges. The impact of removing structural components can be much greater than eliminating individual weights, hence, structured pruning often requires more careful consideration of the trade-off between the model performance and complexity reduction. Since structured pruning operates on a coarser scale, it typically results in lower pruning rates compared to unstructured pruning.

Weight-dependant structured pruning. One of the main categories of structured pruning for CNN is weight-dependent pruning. This strategy assesses the importance of filters based on their respective weights. The Pruning Filter for Efficient ConvNets method [110] focuses on the filter norms as their saliency indicator. Filters with smaller ℓ_1 norms, which result in weak activations, are assumed to contribute less to the final classification decision, hence they become the prime candidates for pruning.

[2]regular in this context means that all weight tensors are dense and that the acceleration of computations does not rely on sparse computing enhancement.

The Filter Pruning via Geometric Median method [70] calculates the geometric median of a set of filters and prunes those filters that are nearest to this geometric median, rather than the ones considered less important by [110]. The filters close to the geometric median are considered by He et al. to be redundant with other filters in the same layers. Wang et al. used another approach to determine redundancy in [198]: The filters are organized into a graph based on their proximity in the space in which they are defined. A redundancy metric is computed for each graph and the least important filters are pruned in the graph with the highest redundancy (as per the authors, any other method for filter importance evaluation can be used [110, 149, 139]). This process is iteratively applied until the targeted pruning rate is reached. These weight-dependent strategies tend to be straightforward and usually demand lower computational costs compared to other methods [68]. They provide an intuitive understanding of how different filters contribute to the overall network performance based on their weight characteristics.

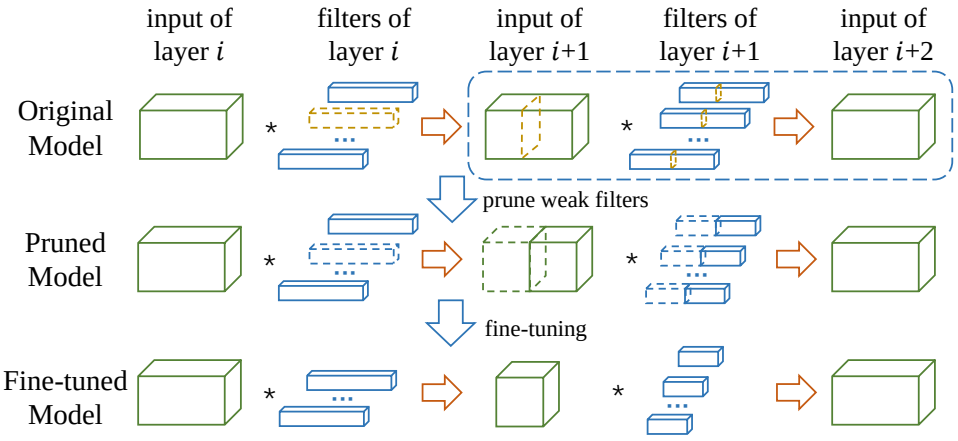


Figure 3.16: Illustration Scheme of ThiNet. The dotter filters and corresponding channels are the ones to be pruned. Once they are removed, the pruned network is fine-tuned. Image taken from [130]

Activation-based structured pruning. Another prominent category of structured pruning is activation-based pruning, where *activation* denotes the result yielded by a layer for given input data. This method takes advantage of activation maps (also called feature maps) for filter pruning. Removing a channel in a feature map is equivalent to removing the filter that computed this channel. He et al. proposed a method to prune filters based on a LASSO regression selection while minimising the least square reconstruction error or the feature map [71]. Hu et al. capitalised

on the abundance of zeros in feature maps that follow the ReLU activation function. They introduced the method called Average Percentage of Zeros (APoZ) that identifies channels in the feature map with a high count of null outputs. These channels, which contribute minimally to the final outcome, can hence be pruned. While the aforementioned methods consider only the feature map of the layer to be pruned, techniques like ThiNet [130] and Approximated Oracle Filter Pruning [28] exploit the relationships between layers to guide pruning. They take into consideration the effect a filter removal in one layer has on the next, allowing for more contextual pruning decisions. More globally, approaches such as Neuron Importance Score Propagation [208] and Discrimination-aware Channel Pruning [218] consider the holistic effect of removing a filter. They aim to understand and quantify the total impact on network performance when a specific filter is removed, accounting for cascading effects across all layers.

Regularisation-based structured pruning. Other methods learn structured sparse networks by introducing various sparsity regularizers. Some methods focus on Batch Normalisation parameters, employing methods like Gated Batch Normalisation [207] and Network Slimming [124]. These methods aim to push certain BN parameters towards zero, effectively disabling corresponding channels and inducing sparsity. Network Slimming applies a ℓ_1 regularisation on the scaling factors γ of the BN [124], whereas [207] adds the ℓ_1 regularisation on scalar factors associated with feature map channels. Kang and Han use BN parameters to craft a mask that prunes channels whose output is likely to be null once evaluated by the ReLU [96].

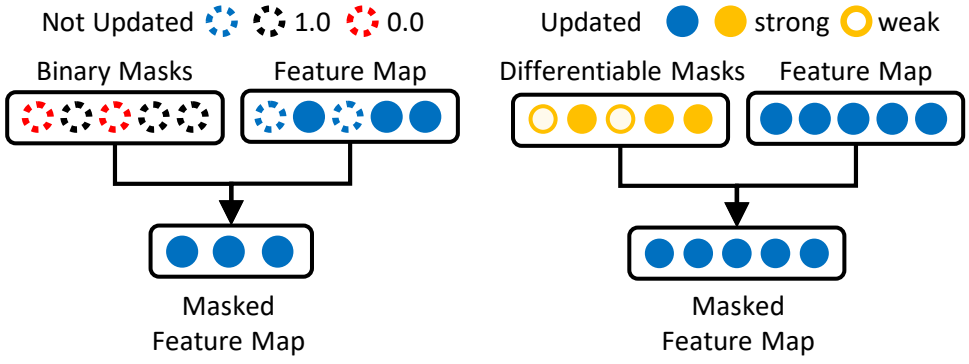


Figure 3.17: Comparison of the method described in [96] (right) and standard channel pruning (left). The differentiable mask allows for a soft pruning that can be reverted during the training. Image taken from [96]

Taylor Expansion-based structured pruning. The Taylor Expansion is a tool that can be used to approximate the change in the loss function due to pruning. Early unstructured pruning methods, Optimal Brain Damage [105] and Optimal Brain Surgeon [64] used Taylor expansion of the hessian matrix to remove weights with the least impact on the loss function. More recently, in [139], Molchanov et al. used first-order Taylor expansion of the loss function to compute the importance score and prune the feature maps. This was later refined in [140] where the authors computed the importance score on the weight, rather than the feature maps, to lower memory consumption.

Variational structured pruning. Variational Bayesian methods provide a way to tackle the computationally intensive process of inferring posterior probability distributions in large data sets, by approximating the posterior distribution with a variational distribution [42]. Specific methods like Variational Pruning [213] and Recursive Bayesian Pruning [217] use this approach to create more effective and stable pruning mechanisms for neural networks. Variational Pruning, models channel importance as random variables, utilizing the centrality property of the Gaussian distribution to induce sparsity [213], while [217] targets the posterior of redundancy, assuming inter-layer dependency among channels.

Dynamic structured pruning. Dynamic pruning represents a different approach in which neural networks are pruned during both training and inference, which facilitates maintaining the model representation capability and offers superior resource consumption-accuracy trade-offs. In the training phase, Dynamic Network Surgery [56] introduces the concept of dynamic pruning through an unstructured approach. This method applies a binary mask indicating the importance of weights and updates it alternately with all the weights, allowing incorrectly pruned parameters an opportunity to revive. Soft Filter Pruning [69] implements a structured version of dynamic pruning. Instead of employing a fixed mask throughout the training, which could limit the optimization space, Soft Filter Pruning dynamically generates masks based on the ℓ_2 norm of filters at every epoch. The dynamic nature of Soft Filter Pruning allows soft-pruned filters to be updated in the next epoch, with new masks being formed based on the updated weights. Focusing on the inference stage, Lin et al. introduced Runtime Neural Pruning [115] which employs a unique framework consisting of a CNN backbone and a recurrent neural network as a decision

network. This approach enables the model to adapt to the properties of different input images by dynamically adjusting its complexity. For easier tasks or simpler images, the network can become sparser, saving computational resources. Deep Reinforcement Learning pruning [14] learns both the static and dynamic importance of channels. The static importance refers to the channel's relevance for the entire dataset, while the dynamic importance is tied to a specific input. Deep Reinforcement Learning pruning [14] applies reinforcement learning to generate a unified pruning decision based on these two aspects of channel importance. More recently, Elkerdawy et al. recently introduced Fire Together Wire Together [37], another dynamic pruning method that treats pruning as a self-supervised binary classification problem. It employs a prediction head to train learnable binary masks and uses a crafted ground truth mask to guide the learning after each convolutional layer. This head takes advantage of the ReLU activation function, which zeros out negative values, to identify the filters that will yield zero activations based on the input and that will be subsequently pruned.

As previously discussed, structured pruning removes indivisible groups of weights and therefore yields regular network architectures that can be implemented in standard deep learning frameworks in a straightforward way. Despite its practicality, structured pruning enforces a strong topological prior by pruning entire groups of weights from the original network, which consequently leads to a lower sparsity rate compared to its counterpart, unstructured pruning. Unstructured pruning provides a more flexible approach by removing individual weights from the original network structure. This process not only offers greater adaptability compared to structured pruning, but also results in higher pruning rates.

Unstructured pruning. In the early stages, unstructured pruning methodologies identified weights that could be eliminated based on their influence on the Hessian of the loss function [105, 64, 65]. A simpler and more tractable strategy for larger networks was later introduced by Han et al. in [59], suggesting a straightforward heuristic: the pruning of weights with the smallest magnitude (*i.e.* absolute value), also referred to as *magnitude pruning*. This technique presented in [59] devises an iterative method wherein a portion of the smallest magnitude weights are removed, followed by the retraining of the network to compensate for the drop of the accuracy. This cycle is reiterated until the preferred level of sparsity is attained.

Furthermore, magnitude pruning has been used together with quantisation and compression techniques to minimise the storage footprint of a network [60]. Magnitude pruning has also been used in energy-efficient CNN design, as detailed in [205]. In this research, the energy consumption of each layer is evaluated, and the layers with the highest energy expenditure are pruned using unstructured magnitude pruning. This layer is then fine-tuned to retain the network accuracy. This process is repeated iteratively until a noticeable drop in accuracy is observed. Dynamic Network Surgery [56] puts forward a derivative of magnitude pruning. A mask, whose value is updated during training, is computed for each weight. This mask is used to stochastically prune a weight or splice[3] it. The saliency of the weights is ascertained based on the magnitude of the associated mask.

Effective subnetworks. More recently, unstructured pruning researches focus on the discovery of small subnetworks inside the original network. In other words, to identify a subset of weight that can perform, under certain conditions or assumptions, as well as the original network. Most notably, the Lottery Ticket Hypothesis [43] states that within a large, randomly initialized neural network, there exist subnetworks or *Lottery Tickets* that are capable of training effectively when isolated from the rest of the weights. These smaller networks, found by pruning the smallest magnitude weights from the trained original network, are observed to train faster and achieve comparable or even superior performance with respect to the original network. The *lottery ticket* is found by training the original network up to convergence, then pruned with magnitude pruning and finally, the remaining weights are reinitialized to their original values, it is to say the value they had before the training even started. The resulting subnetwork is the *lottery ticket*. This research sparked significant interest and various works: [215] proposed an analysis of the results presented in [43]. These results [43] have been extended to larger networks in [44], the necessity of training the original network to convergence to find the Lottery Tickets has been challenged in [123]. The existence of the *lottery ticket* (or in other words the subnetwork) has been theoretically proven in [132] and the requirements on the theoretical size of the original network have been later refined in [148, 145].

[3]to splice is the verb used by the authors (Guo et al.) to denote the reactivation of a weight that was previously pruned

3.6 Positioning

Within the various deep neural network compression methods presented in this chapter, we choose to focus specifically on pruning in the context of supervised image classification. Among various types of pruning, our interest goes towards unstructured pruning due to the flexibility it offers, in particular, its potential for achieving high pruning rates compared to structured pruning.

Our decision to work on pruning is rooted in the following considerations:

- First, pruning allows the creation of lightweight networks while preserving or sometimes improving the performance of the original network.
- Then, pruning integrates well with other compression techniques and can be applied in conjunction with them, on any kind of architecture.
- Finally, pruning does not necessitate the creation and development of an architecture from scratch. It can be applied to an already existing architecture to compress it, which makes it possible to develop small, lightweight networks without the need for extensive research into the creation of the base architecture. This approach allows freeing exploration and research and development efforts that can be allocated to other topics.

Despite its numerous advantages, pruning is not without challenges. One such challenge is the identification of the weights to be preserved. This is a topic that is the subject of many works detailed in this chapter. Additionally, the preserved weights typically require fine-tuning, which can impose a substantial computational cost. This process of fine-tuning can be both time-consuming and resource-intensive.

Chapter 4 delves into the fine-tuning issue and presents a new pruning method that circumvents the need for the expensive fine-tuning step. Its budget loss together with the weight reparametrisation allows for joint optimisation of the topology and the weights of the network without the need for auxiliary variables. As a result, the obtained lightweight networks preserve accuracy after effective pruning and do not require fine-tuning. Furthermore, chapter 5 tackles the challenge of identifying relevant weights without even having to train them. It proposes a method of topology

selection given a set of untrained weights that achieves compelling performances, thereby also sidestepping the fine-tuning. In contrast to other methods, the optimal pruning rate is discovered in one shot by our pruning strategy that circumvents the costly gird search for its value. This innovative approach opens up new possibilities for further reducing the computational costs associated with pruning and provides a new direction for future research in this area, as well as neural network training without weight tuning in general.

3.7 Conclusion

The evolution of neural networks, along with the growing demand for their deployment in resource-constrained environments, has underlined the need for neural network compression techniques. This chapter has examined the state-of-the-art methodologies for reducing the computational demands and memory footprints of deep neural networks, thereby facilitating their usage in a variety of application domains.

First, we explored chapter 2 the historical progression and the major architectures of deep neural networks, illustrating the connection between their complexity and performance. Then, in this chapter, we first investigated the techniques for accelerating computations within neural networks, emphasising the role of fast convolution techniques in reducing both runtime and computational resources. Our focus then shifted to Knowledge Distillation, a paradigm that allows the transfer of knowledge from a large, complex network to a smaller, more efficient one. The core idea is to teach a lightweight student network to mimic the behaviour of a teacher network, thus achieving comparable performance with a reduced footprint. Next, we delved into efficient architecture design methods, including bespoke architectures designed to minimise size while maintaining performance, and Neural Architecture Search strategies for automating the discovery of optimal architectures. Lastly, we addressed the strategies for compressing and optimising existing neural networks, considering both quantisation and binarisation techniques that lower the numerical precision of weights and activations, as well as pruning techniques that remove redundant or insignificant weights and connections, resulting in sparser and more computationally efficient models. In conclusion, these techniques provide a multi-faceted approach to neural network compression and acceleration,

with each offering unique advantages and trade-offs.

The next chapters will present our contributions to neural network compression based on pruning. The chapter 4 details our first contribution that consists in a method to simultaneously train and prune neural networks while matching a budget. This method allows bypassing the need for the finetuning step present in most methods based on magnitude pruning by jointly optimising the topology and the weights without the need for additional auxiliary parameters.

Chapter 4

Weight Reparametrization for Budget-Aware Network Pruning

Contents

Chapter Abstract

This chapter addresses the challenge of compressing large neural networks, whose time and memory footprints are increasingly high. Although large neural networks have shown impressive performances across various domains, they are not deployable on embedded devices due to their size. Among the existing techniques that yield lightweight neural networks, pruning is a popular approach that seeks to reduce the size of neural networks by removing redundant or unnecessary weights. However, most of the pruning methods rely on saliency indicators

that identify removable weights after training without considering the targeted pruning rate.

In this chapter, we propose an alternative pruning approach based on weight reparametrization. Our method incorporates a budget loss that drives sparsity toward the targeted pruning rate during training. The weight reparametrisation acts as a mask that soft-prunes the smallest weights, while the budget loss serves as a surrogate ℓ_0 norm that regulates the network budget. As a result, our approach significantly mitigates the performance drop, induced by pruning, with respect to the unpruned network compared to other methods. We demonstrate experimentally the effectiveness of our method across various pruning rates, datasets, and architectures, including Conv4, VGG16, ResNet20 on CIFAR-10 and CIFAR-100, and ResNet18 on the more challenging TinyImageNet dataset.

We also evaluate the effectiveness and relevance of our method in a comparative experimental analysis using different settings. Furthermore, we evaluate the proposed approach with an already tuned and pruned initialisation and show that it outperforms common fine-tuning methods. Our results show that our proposed approach is a promising alternative to existing pruning techniques, providing an efficient and effective way to reduce the size of neural networks while eliminating the need for costly fine-tuning steps.

This chapter presents work that has resulted in the publication of the following conference article:

- Robin Dupont, Hichem Sahbi, and Guillaume Michel. Weight reparametrization for budget-aware network pruning. In *2021 IEEE International Conference on Image Processing, ICIP 2021, Anchorage, AK, USA, September 19-22, 2021*, pages 789–793. IEEE, 2021.

4.1 Introduction and Related Work

This chapter addresses the challenge of pruning large neural networks without degrading their performance. Most of the existing pruning methods degrade the performance of neural networks due to the pruning step that

removes weights from the networks. Therefore, a costly fine-tuning step is usually required in order to compensate for the loss in performance. The method we introduce in this chapter circumvents this issue and yields lightweight pruned networks with minimal performance degradation. This is achieved through a combination of weight reparametrisation that encompasses a surrogate ℓ_0 norm and a budget loss that drives the sparsity toward the predefined pruning rate.

Pruning is an excellent way to obtain lightweight neural networks because it reduces the number of parameters in a pre-trained network without the need to design a new architecture. Instead of starting from scratch, pruning techniques can be applied to existing architectures, which have been trained and tested on large-scale datasets. Pruning aims to reduce the number of network parameters by removing redundant or unnecessary weights from a given network referred to as the *original network*. It then yields a sparsified and lightweight architecture, hereafter referred to as *pruned network*. Pruning methods can be split into two major categories: *(i)* unstructured weight pruning, where individual weights of a given network are removed based on their importance, and *(ii)* structured pruning, where entire columns, rows, channels, filters or even entire parts, such as skip connections of a residual network [67], are removed. Our method belongs to the first category. The subsequent sections provide an overview of related works on unstructured pruning, pruning with weight reparametrisation, budget loss and pruning without fine-tuning, followed by the contributions of this chapter.

4.1.1 Unstructured Magnitude Pruning.

Unstructured magnitude pruning has emerged as an effective heuristic for determining the saliency of the weights. This method focuses on removing independent weights from the global structure of the network, hence offering greater flexibility compared to structured pruning which imposes a strong topological prior by eliminating whole sections of the original network. Magnitude pruning revolves around the hypothesis that the smallest weights contribute less to the final output of the network and can thereby be removed with minimal impact on the performance. Considering a weight tensor $\mathbf{w}$, if p represents a pruning function, magnitude pruning can be for-

malised and implemented as follows:

$$\mathbf{w}_{\text{pruned}} = p(\mathbf{w}, \alpha) = \mathbf{w} \odot \mathbf{m}_\alpha \tag{4.1}$$

where α is a threshold, $\odot$ denotes the Hadamard product and $\mathbf{m}_\alpha$ is a binary mask that is defined as follows:

$$(\mathbf{m}_\alpha)_{ij} = \begin{cases} 0 & \text{if } |w_{ij}| \leq \alpha \\ 1 & \text{otherwise} \end{cases} \tag{4.2}$$

In equation (4.2), the threshold α is typically chosen to be the k-th percentile of the weights so that the pruning rate, defined as the fraction of non-zero weights, is equal to k.

Unstructured magnitude pruning has been used in various works and in particular in [59] where a three-step process is proposed:

1. The process begins with standard training to identify the most important connections within the network.

2. This is followed by a magnitude pruning step where weights with the smallest magnitude (or absolute value) are removed until a given pruning rate is reached.

3. The final step involves fine-tuning the remaining weights to compensate for any loss of accuracy caused by the pruning.

The authors also propose an iterative variant where steps 2 and 3 are repeated while gradually increasing the pruning rate until the final pruning rate is reached. This method was used in [60], where it was combined with quantisation and Huffman coding. In both methods [59, 60], obtaining the final network is computationally intensive due to the fine-tuning step, which can be all the more computationally intensive if the magnitude

pruning and fine-tuning are performed iteratively.

Unstructured magnitude pruning has gained significant renewed interest with the advent of the Lottery Ticket Hypothesis (LTH) [43]. An empirical study in [43] demonstrated the existence of subnetworks, termed Lottery Ticket (LT), within large pre-trained networks. These subnetworks, when trained with initial weights from the larger networks, yielded comparably accurate classifiers. To isolate these lottery tickets, the authors rely on magnitude pruning or its iterative variant to identify the subnetworks. The large network is trained to convergence, then its weights are pruned with magnitude pruning, and restored to their original values. Despite the potency of this approach, its practical application is often hindered by the computationally intensive training and fine-tuning steps required to obtain a trained lightweight subnetwork.

4.1.2 Weight Reparametrisation

Weight reparametrisation is a technique where the weights are expressed as a function of other variables. Typically the weights used in the network, here denoted $\hat{\mathbf{w}}$ and referred to as *apparent weights*, are expressed as a function of the *latent weights* $\mathbf{w}$ and other variables. In [172], Schwarz et al. presented a weight reparametrisation based on raising a weight to the power of n while preserving its sign. This reparametrisation is formalised as:

$$\hat{\mathbf{w}} = \mathbf{w} \odot |\mathbf{w}|^{n-1} \tag{4.3}$$

where n is a hyperparameter of the method, typically set between 1 and 3 [172]. This reparametrisation creates a *rich get richer (sic)* dynamic according to the authors, referring to the fact that the weights with the largest magnitude are all the more increased. The pruning is enforced by pruning reparametrised weights with magnitude pruning.

4.1.3 Pruning with Budget

Most pruning techniques, including the ones presented in this section, enforce a pruning rate after the initial training. Consequently, the optimisation process does not take into account the final weight budget that will be allocated to the network. Some work tackled this issue by adding a loss that drives the network to respect a budget. Note that the works described subsequently all fall into the *structured* pruning category.

Lemaire et al. introduced in [108] a Budget-Aware Regularisation loss or BAR loss that performs structured pruning on the channels of the activations (and consequently of the kernels of the layer yielding it). This loss is combined with the task loss to drive the sparsity toward the targeted pruning rate as described in equation (4.4). The relative importance of both losses is set with a strictly positive mixing coefficient λ.

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda \mathcal{L}_{\text{BAR}} \tag{4.4}$$

The BAR loss is responsible for introducing sparsity in the network and for driving the sparsity toward the targeted pruning rate. It is defined as follows:

$$\mathcal{L}_{\text{BAR}}(\Phi, V, a, b) = \mathcal{L}_S(\Phi) f(V, a, b) \tag{4.5}$$

where Φ is a mask sampled from the *hard concrete* distribution [127], a continuous relaxation of the Bernoulli distribution. V is the current budget used by the network, computed as the fraction of active channels in the activations and referred to as the *activation volume* by the authors, b and a are hyperparameters detailed subsequently. This BAR loss is composed of two parts. The first part, $\mathcal{L}_S(\Phi)$ introduces sparsity in the network by penalising the masks Φ with the hard concrete loss [127], and the second part $f(V, a, b)$ controls the budget. The function f implements a variant of the log barrier function [11], where a controls the steepness of the function and b is the target budget.

ChipNet [189] is another channel-based structured pruning method that includes a sparsity inducing loss $\mathcal{L}_c$ dubbed as *crispness loss* and a budget loss $\mathcal{L}_b$. Both losses are combined with the task loss with two mixing coefficients α_1 and α_2 respectively, as shown in equation (4.6):

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \alpha_1 \mathcal{L}_c + \alpha_2 \mathcal{L}_b \tag{4.6}$$

Each channel of the network is associated with a mask, and the *crispness loss* ensures crisp mask values, that is to say, close to 0 or 1. For the budget loss, authors of [189] propose four variants depending on the type of budget that is enforced, namely: a *channel budget* that computes the fraction of channels that are not pruned, a *volume budget* that is equivalent to the *activation volume* of [108], a *parameter budget* and finally a *FLOP budget*. In their experiments, the authors reported that they fine-tuned the networks after the pruning with the same hyperparameters as the initial training phase and in particular with the same number of epochs, effectively doubling the training time.

4.1.4 Pruning without fine-tuning

Most pruning methods cause a performance drop following the pruning step. This is due to the fact that the weights that are removed play a non-negligible role in the network. In this context, fine-tuning is a common technique that aims to recover the lost performance but fine-tuning is a computationally intensive task. Some works propose pruning methods that are designed to circumvent the need for fine-tuning [63, 64, 96].

Optimal Brain Surgeon (OBS) introduced in [63] by Hassibi and Stork is a Hessian-based pruning method that builds upon [105] but contrary to the former, it relaxes the diagonal assumption of the loss Hessian matrix. OBS is a second-order pruning method that uses the Hessian matrix of the loss function to identify the most removable weights and determine the optimal update for the remaining weights in order to mitigate the performance drop. Among their contributions in [63], [63] also proposed an iterative method to compute the inverse of the Hessian matrix but this is outside of the scope of this section. OBS considers the following expression

of the loss function Taylor expansion at a local minimum:

$$\delta\mathcal{L} = \left(\frac{\partial\mathcal{L}}{\partial\mathbf{w}}\right)^T \cdot \delta\mathbf{w} + \frac{1}{2}\delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w} + O(\|\delta\mathbf{w}\|^3) \tag{4.7}$$

where $\delta\mathcal{L}$ is the variation of the loss function, $\delta\mathbf{w}$ is the variation of the weights and $\mathbf{H}$ is the loss Hessian matrix w.r.t. the weights. In equation (4.7), at a local minimum, the first term vanishes and the author neglects higher-order terms. The equation (4.7) is updated to:

$$\delta\mathcal{L} \approx \frac{1}{2}\delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w} \tag{4.8}$$

Each weight is associated with a saliency indicator that represents its impact on the loss, whose variation is quantified by equation (4.8), as well as an update vector that is applied to the other weights to mitigate the performance drop if the considered weight were to be pruned. Both quantities (saliency indicator and update vector) are computed by minimising equation (4.8) with respect to the weight to be pruned. The authors then propose to remove the weight with the smallest saliency indicator and update the remaining weights.

This process is repeated until a defined stopping criterion. Hassibi and Stork suggest stopping the pruning process when the impact on the loss function is no longer negligible compared to the value of the loss itself. Once the pruning is stopped, the remaining weights can be used as is and do not necessitate further fine-tuning thanks to the corrections applied with the update vector. The authors applied this method successfully on small networks (a few tens of thousands of weights) compared to modern architectures (see table 2.1). This method is intractable in practice with larger neural networks, in particular the computation of the Hessian matrix whose size is quadratic in the number of weights of the network.

Whereas OBS corrects the remaining weights after pruning, it is possible to introduce sparsity while training the network. Kang and Han introduced a stochastic structured pruning method in [96] that prunes the

channels after a BN layer that have the highest probability of being inhibited by the ReLU activation function. Each channel is associated with a latent mask q, parametrised by a function of the shift and scale parameters of the BN layer: $\Phi(\beta, \gamma)$. The mask applied to the channel during training is then obtained by binarising q, using the Gumble Softmax trick [92] to preserve differentiability. Sparsity is then introduced in the masks by adding a regularisation loss defined as follows:

$$\mathcal{L}_{\text{sparse}}(B, C) = \sum_{\beta_j \in B, \gamma_j \in C} \beta_j + s|\gamma_j| \tag{4.9}$$

where B and C are the set of shift and scale parameters of the BN, and s is a hyperparameter of the method. This setup allows for a joint optimisation of the masks through the BN parameters and the network weights. Once the training is completed, the masks are binarised following equation (4.10), where c is a given threshold, and then multiplied with their associated channels to prune the network. Since the optimisation of the masks is done jointly with the network weights, the network does not necessitate further fine-tuning.

$$q(\delta; \beta, \gamma) = \begin{cases} 0 & \text{if } \Phi(\beta, \gamma) \geq c \\ 1 & \text{otherwise} \end{cases} \tag{4.10}$$

These structured or unstructured methods propose different saliency indicators and pruning criteria that aim at identifying and removing redundant or unnecessary weights or groups of weights. Removing weights is typically done after the training phase. This approach does not take into account the final desired model size, or weight budget, during the optimization process. In other words, the pruning strategy is an afterthought and is not integrated into the training process. This results in an inefficient process where the network is first trained with a large number of weights, many of which are later pruned. This introduces a loss of functional performance - depending on the task considered - that needs to be compensated for (with the exception of [96, 63]). This is achieved through fine-tuning the sparse or lightened networks obtained after applying the pruning criterion. Fine-tuning is a computationally intensive task and requires additional training

time [59, 60]. Moreover, the amount of weights pruned is enforced after the initial training, meaning that the final targeted size or weight budget is never considered in the optimisation procedure. Indeed, separating the optimisation of topology from weight tuning is sub-optimal and therefore introduces a performance drop when the pruning is enforced. Thus, the remaining weights need to undergo a fine-tuning step that adapts them to the enforced sparse topology. Furthermore, the pruned connections are permanently removed with no possibility of reactivating them during fine-tuning.

4.1.5 Contributions

In order to address the aforementioned issues, namely the need for a costly fine-tuning step and the lack of consideration for the final budget and topology, we introduce a novel weight reparametrisation that learns not only the weights of a surrogate lightweight network but also its topology. This weight reparametrisation acts as a regulariser that models the tensor of the parameters of a network, again referred to as the *surrogate network*, as the Hadamard product of a weight tensor and an implicit mask. The latter makes it possible to implement unstructured pruning constrained with a budget loss that precisely controls the number of non-zero weights in the resulting network. Experiments conducted on the CIFAR-10, CIFAR-100 and the TinyImageNet classification tasks, using standard primary architectures (namely Conv4, VGG16, ResNet20 and ResNet18), show the ability of our method to train effective surrogate pruned networks without any fine-tuning.

In what follows, for the sake of conceptual simplicity, we will adopt a conventional approach where multidimensional tensor entries are indexed by i, in addition to the layer index ℓ, effectively vectorising these entities for ease of manipulation.

The rest of this chapter is organised as follows: section 4.2 details the proposed method and in particular the weight reparametrisation in section 4.2.1 and the budget loss in section 4.2.2. An overview of the method and a general algorithm are given in section 4.3. Section 4.4 presents the results of our comprehensive experiments, including performance comparisons, the experimental validation of our two main components, the impact of the mixing coefficient λ that balances the task loss and the budget loss,

and the initialisation. Finally, section 4.5 concludes the chapter by summarising our contributions and our key findings.

4.2 Pruning with Weight Reparametrisation and Budget Loss

Consider the general case of a multi-layer neural network. Following the notations introduced in section 2.3, a neural network is represented as a function f of two variables: θ and X. Function f embodies the network topology, which is essentially a graph, whose edge values are determined by θ. More specifically, θ represents the collection of weights of the network, with $\theta = \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_L\}$ and L denoting the number of layers. Each element $\mathbf{w}_\ell$ of θ is a multi-dimensional weight tensor encompassing a total of ν_ℓ elements, associated with layer ℓ. Following the notation introduced in section 2.5, the parameter X of f represents the input given to the network. Each input X is associated with a label y, also called ground truth. This functional conception of a neural network can be formally written as:

$$\begin{aligned} f\colon \mathbb{R}^{\dim(X)} &\to \mathbb{R}^{\dim(y)} \\ X \quad &\mapsto f(X, \theta) \end{aligned} \tag{4.11}$$

The discrepancy between the output of the neural network and the ground truth $y \in \mathcal{Y}$ is computed with a loss function $\mathcal{L}$, whose expression depends on the considered task. This loss is then minimised by updating the parameters θ of the network, thanks to the backpropagation algorithm [168, 169] and gradient descent methods.

When used in the loss function, the ℓ_0 norm is perfectly suited for pruning a network by, on the one hand, acting as a sparsity-inducing regulariser for the weights (θ), and on the other hand, by indicating the number of non-zero weights in the network, which is useful for computing the weight budget.

We aim to propose an end-to-end method that fits into the backpropagation framework. Therefore, adding a ℓ_0 regulariser and a ℓ_0 based weight

budget is not possible since the ℓ_0 norm is not differentiable. Thus, we propose our differentiable reparametrisation, which seeks to define a novel weight expression related to magnitude pruning [105, 59]. This expression corresponds to the Hadamard product involving a weight tensor and a function applied entry-wise to the same tensor (as shown in figure 4.1). This function acts as a mask that *(i)* multiplies weights by soft-pruning factors which capture their importance and *(ii)* pushes less important weights to zero through a particular budget added to the loss function $\mathcal{L}$.

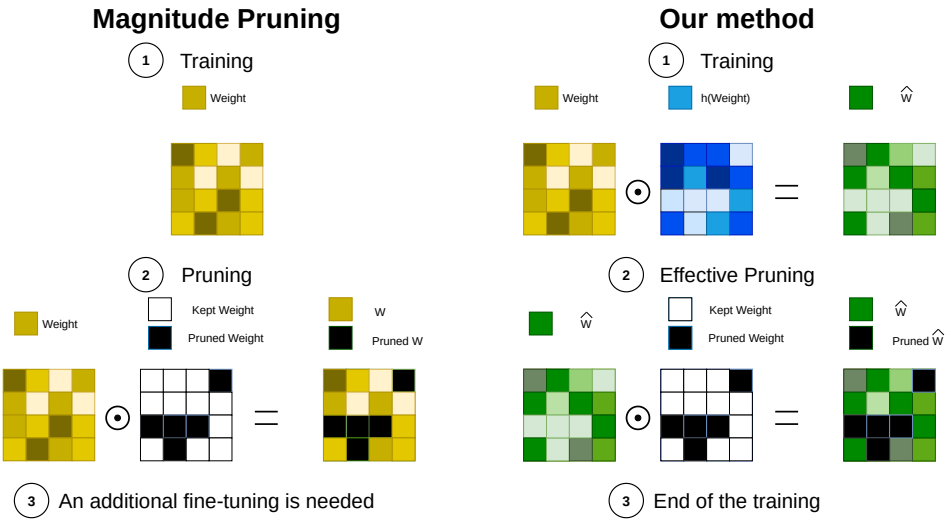


Figure 4.1: Comparison of our method and magnitude pruning. Magnitude pruning does not include any prior on weights during the initial training phase and needs an additional fine-tuning procedure. Our method embeds a saliency measure based on the weight magnitude in the reparametrisation and does not require fine-tuning. Best viewed in colour.

Our proposed framework allows for joint optimisation of the network weights and topology. On the one hand, it prevents situations where, because of excessively pruning a layer, the output from layer ℓ of the network is no longer transmitted to layer $\ell + 1$, a scenario we refer to as *disconnections*. These disconnections can lead to degenerate networks with an irrecoverable performance drop. Due to these disconnections, the network output becomes entirely uncorrelated with the input, and it becomes impossible to update the weights located upstream of this disconnection. On the other hand, our framework allows reaching a targeted pruning budget in a more convenient way than ℓ_1 regularisation (see section 4.4.4). Our reparametrisation also helps to minimise the performance drop between the original and the pruned surrogate networks by maintaining competitive performances without fine-tuning. Learning the weight values and the

network topology only requires one step that achieves pruning as a part of network design. This step zeroes out the targeted number of connections by constraining their reparametrised weights to vanish.

4.2.1 Weight Reparametrisation

We consider the original network f as a stack of L layers. The global expression of f can be recursively defined by the application of the layer ℓ to the output of the layer $\ell - 1$. Without a loss of generality, we omit the bias for clarity. This expression is shown in equation (4.12).

$$f(\mathbf{x}) = g_L\big(\mathbf{w}_L \cdot g_{L-1}(\mathbf{w}_{L-1} \cdot g_{L-2} \dots \mathbf{w}_2 \cdot g_1(\mathbf{w}_1 \cdot \mathbf{x}))\big), \tag{4.12}$$

with g_ℓ being a nonlinear activation associated to $\ell \in \{1, \dots, L\}$ and $\{\mathbf{w}_\ell\}_\ell$ denoting a set of weight tensors where each tensor is associated with a specific layer ℓ in the network. Keeping the same topology but changing the values of the weight, we now consider the surrogate network $\hat{f}$ with weights $\{\hat{\mathbf{w}}_\ell\}_\ell$. Equation (4.12) can be rewriten as equation (4.13). The activation function and the topology of f and $\hat{f}$ are the same. Only the weights are updated.

$$\hat{f}(\mathbf{x}) = g_L\big(\hat{\mathbf{w}}_L \cdot g_{L-1}(\hat{\mathbf{w}}_{L-1} \cdot g_{L-2} \dots \hat{\mathbf{w}}_2 \cdot g_1(\hat{\mathbf{w}}_1 \cdot \mathbf{x}))\big). \tag{4.13}$$

In the above equation, $\hat{\mathbf{w}}_\ell$ is referred to as *apparent weight* tensor, which is a reparametrisation of $\mathbf{w}_\ell$ that includes a prior on its saliency. An apparent weight $\hat{\mathbf{w}}_\ell$ of $\hat{f}$ is derived from a *latent weight* $\mathbf{w}_\ell$ by applying the following reparametrisation:

$$\hat{\mathbf{w}}_\ell = \mathbf{w}_\ell \odot h_t(\mathbf{w}_\ell), \tag{4.14}$$

with h_t being the reparametrisation function and t its temperature parameter (see equation (4.19)). Here, $\odot$ represents the Hadamard product. It means that it is element-wise, and every single weight has its own reparametrisation. This reparametrisation function enforces the prior that the smallest weights should be removed from the network and acts as a surrogate ℓ_0 norm for the budget loss (see section 4.2.2). In order to achieve this objective, h_t should exhibit four properties:

1. $\forall x \in \mathbb{R}, \quad 0 \leq h_t(x) \leq 1$
2. $h_t(x) \in C^1$ on $\mathbb{R}$

3. $h_t(x) = h_t(-x)$

4. $\forall a, \varepsilon \in \mathbb{R}^{+*},\ \exists\ t \in \mathbb{R}^{+*} \mid h_t(x) \leq \varepsilon, x \in [-a, a]$

First Property - Constrained Image

$$\forall x \in \mathbb{R},\quad 0 \leq h_t(x) \leq 1 \tag{4.15}$$

There should not be any co-adaptation between the weights and their reparametrisation. In other words, the reparametrisation function should only act as a means to select or not the weight. It should not act as a scaling factor for the latent weight and scale it so that the apparent weight becomes larger than the latent weight. Constraining the image prevents the value of the weights from increasing rapidly to compensate for the removal of the smallest weights. Finally, the apparent weights should have the same sign as the latent weights. That is why the image of $\mathbb{R}$ by h_t should be the segment $[0, 1]$.

Second Property - Differentiability

$$h_t(x) \in C^1 \text{ on } \mathbb{R} \tag{4.16}$$

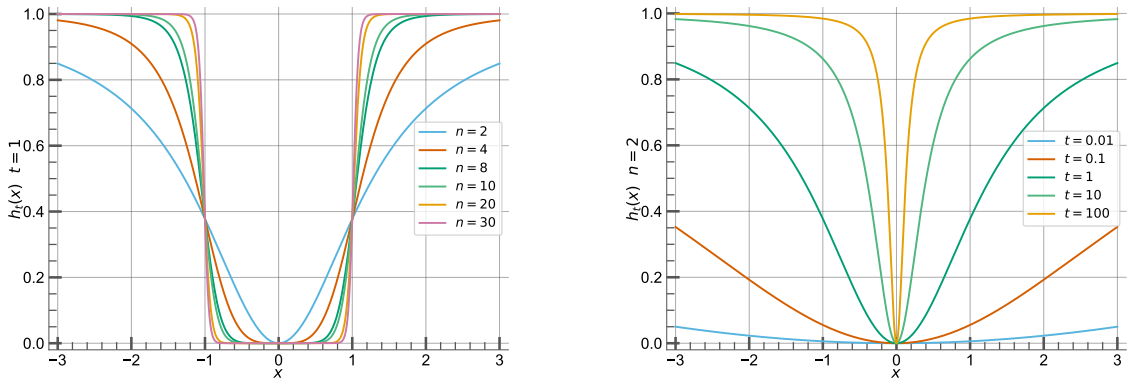
Our method should fit in the backpropagation framework [169]. Since the optimisation will be achieved by gradient descent, the reparametrisation function should be differentiable to ensure a computable gradient.

Third Property - Symmetry

$$h_t(x) = h_t(-x) \tag{4.17}$$

The reparametrisation function should not induce any bias toward the positive or negative weights so that only their magnitudes matter. It implies that the reparametrisation function should be symmetric.

Fourth Property - Upper Bounded Segment



(a) h_t with $t = 1$ and varying n **(b)** h_t with $n = 2$ and varying t

Figure 4.2: Reparametrisation function h_t with varying temperature parameter t and power n. t controls the width of the pit, and n controls the steepness of the slope.

$$\forall a, \varepsilon \in \mathbb{R}^{+*},\ \exists\ t \in \mathbb{R}^{+*} \mid h_t(x) \leq \varepsilon, x \in [-a, a] \tag{4.18}$$

The last property ensures the existence of a temperature parameter t, which allows upper-bounding the response of h_t on any interval for any arbitrary ε. More formally, for any arbitrarily large a and arbitrarily small ε, it exists a temperature t which guarantees that $h_t(x)$ is smaller than ε, provided that x is in the segment $[-a, a]$. Hence, h_t acts as a band-stop filter, eliminating the smallest weights where the parameter t controls the width of that filter. Figure 4.2b shows the impact of t on the shape of the function, more precisely on the width of its pit, when the expression of h_t is set using equation (4.20).

Weight distribution varies tremendously from one layer to another. In order to match a specific budget (see section 4.2.2), the width of the stop-band, controlled by t, is tuned according to the weight distribution of each layer. The manual setting of this parameter is non-trivial and cumbersome, so in practice, t is learned as a part of gradient descent on a layer-by-layer basis.

Considering the aforementioned four properties of h_t, a simple choice of that function is:

$$\tilde{h}_t(x) = \exp\left\{-\frac{1}{(tx)^n}\right\},\ n \in 2\mathbb{N}, \tag{4.19}$$

where n controls the crispness of $\tilde{h}_t$. Here and in what follows, $\tilde{h}_t$ denotes the expression of equation (4.19) for the reparametrisation function. The exponent n is not considered as a parameter of h_t (or $\tilde{h}_t$) since we use a fixed value for our experiments (section 4.4), whereas t is a learnt parameter and varies from one layer to another. Figure 4.2a shows the impact of n on the general sharpness of the function. Although the function whose expression is given in equation (4.19) satisfies the four above properties, in practice, $\tilde{h}_t$ suffers from numerical instability as it generates Not a Number (`NaN`) outputs in most of the widely used deep learning frameworks. Due to the way backpropagation works, a single `NaN` in a weight tensor makes the whole optimisation process for the entire network no longer possible. We consider instead a stabilised variant with similar behaviour, as equation (4.19), that still satisfies the four above properties. This numerically stable variant is defined as:

$$h_t(x) = C_1\left(\exp\left\{-\frac{1}{(tx)^n + 1}\right\} - C_2\right), \tag{4.20}$$

with $C_1 = \frac{1}{1-e^{-1}}$ and $C_2 = e^{-1}$. In what follows, we use the expression of equation (4.20) for the reparametrisation function h_t, whereas $\tilde{h}_t$ refers to the expression of equation (4.19), which is its numerically unstable version.

The addition of the scalar value 1 at the denominator in equation (4.20) is a way to achieve numerical stability. In equation (4.19), the denominator $(tx)^n$ has the potential to approach very small values that result in numerical instabilities, leading to `NaN` outputs. The addition of 1 to the denominator makes the function numerically stable and avoids producing `NaN` outputs. This solution is favoured over adding a small value, such as an arbitrarily small ε, as the latter requires careful consideration of its magnitude and may result in either dramatic alterations to the shape of the function or continued numerical instability if not carefully chosen. The addition of the value 1 to the denominator provides a straightforward and sufficient mean to stabilise the function. Constants C_1 and C_2 are introduced to compensate for the slight alterations to the shape of the function caused by the addition of 1 to the denominator, and thus, to ensure that the first property (equation (4.15)) is satisfied. Although both $\tilde{h}_t$ and h_t satisfy the four properties, they do not have the exact same shapes, as illustrated in figure (4.3).

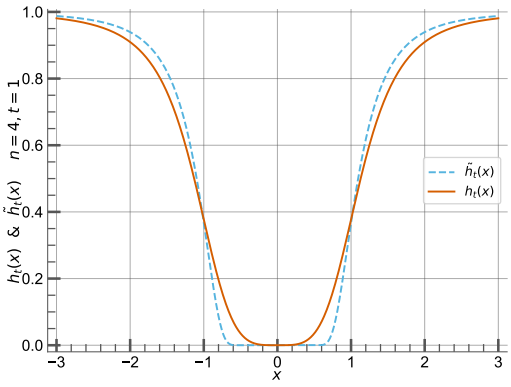


Figure 4.3: The unstable reparametrisation function $\tilde{h}_t$ and its stable alternative h_t, with $t = 1$ and $n = 4$ for both functions.

In the next sections, the application of function h_t to a multi-dimensional tensor is element-wise. Consequently $h_t(\mathbf{z})$ denotes the tensor whose entries are the result of applying h_t to each one of the corresponding entries of $\mathbf{z}$.

4.2.2 Budget Loss

Most of the traditional pruning methods in deep learning do not explicitly incorporate the targeted weight budget during optimisation. The amount of weights pruned is typically enforced post-training, which may lead to suboptimal results compared to methods that consider the weight budget during optimisation. Our method introduces a budget loss, in addition to the main task loss, that drives the network to match and satisfy a given weight budget during the training process. Consequently, the trained network can be pruned to the desired pruning rate with a marginal loss in performance without fine-tuning.

The considered budget is weight-based and should quantify the targeted fraction of active connections in the network. To build the budget loss, we first introduce a cost function that quantifies the number of active connections in the network. Let $C(\{\mathbf{w}_1, \ldots, \mathbf{w}_L\})$ be the *observed* cost associated to a neural network and C_{target} the *targeted* one. C_{target} is the number of connections that should be active at the end of the training procedure. The budget loss is defined as

$$\mathcal{L}_{\text{budget}} = \big(C(\{\mathbf{w}_1, \ldots, \mathbf{w}_L\}) - C_{\text{target}}\big)^2. \tag{4.21}$$

This budget loss is combined with the main task loss (a classification loss in our experiments - see section 4.4). The budget loss $\mathcal{L}_{\text{budget}}$ is quadratic in order to ensure the minimisation of the difference between the *observed* and the *targeted* costs. For better conditioning of this combination, we normalise the budget loss by C_{initial}. The latter corresponds to the cost of the original unpruned network, which is set in practice to the number of its parameters (see also section 4.4). Hence, equation (4.21) is updated as:

$$\mathcal{L}_{\text{budget}} = \left(\frac{C(\{\mathbf{w}_1, \ldots, \mathbf{w}_L\}) - C_{\text{target}}}{C_{\text{initial}}}\right)^2. \tag{4.22}$$

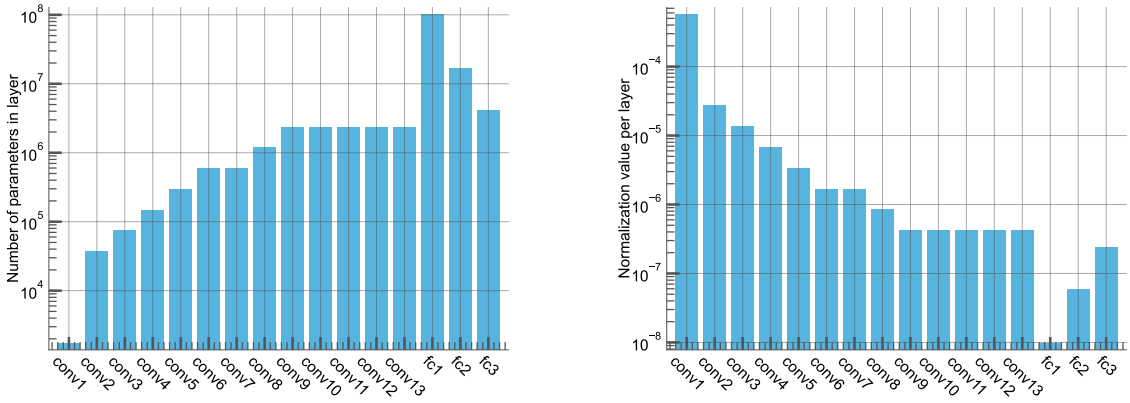
Finally, the two losses are combined together via a strictly positive mixing hyperparameter λ that controls the relative importance of the budget loss $\mathcal{L}_{\text{budget}}$ compared to the main task loss $\mathcal{L}_{\text{task}}$, leading to

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda \cdot \mathcal{L}_{\text{budget}}. \tag{4.23}$$

Ideally, the budget of a neural network could be evaluated as the number of multiply-add operations, often referred to as FLOPs or MACs[1], needed for a forward pass or through the ℓ_0 norm of its weights. However, in their basic form, neither are known to be differentiable and, therefore, cannot be used in a gradient-based optimisation. In order to circumvent these limitations, we use our weight reparametrisation as a surrogate measure of ℓ_0, and we define the cost function as in equation (4.24).

$$C(\{\mathbf{w}_1, \ldots, \mathbf{w}_L\}) = \sum_{\ell=1}^{L} \sum_{i=1}^{\nu_\ell} h_t(w_{\ell i}). \tag{4.24}$$

[1]The number of MAC operations or FLOPs for a layer cannot be fully determined by its number of parameters since it is heavily dependent on the input size. More details are given in appendix A.1



(a) Number of parameters **(b)** Normalisation factor

Figure 4.4: Log-scale plot of number of parameters and normalisation factor per layer for a VGG16 network. The significant differences in terms of the number of parameters yields dramatically different normalisation factors. Some of them are 4 orders of magnitude apart, and all of them are vanishingly small compared to a common main task loss value.

One may argue that the cost should be normalised layer-wise and, therefore, that the right-hand term of equation (4.24) should be written as

$$\sum_{\ell=1}^{L} \frac{\sum_{i=1}^{\nu_\ell} h_t(w_{\ell i})}{\nu_\ell}$$

However, the number of elements in a layer greatly varies from one to another (as displayed in figure 4.4). As a result, the budget loss relative importance would vary from one layer to another. More importantly, the optimisation process would have less incentive to introduce sparsity in larger layers since their normalisation factor would make the budget loss negligible compared to other layers or the main task loss. This is critical since the large layers are generally the ones where the highest pruning rates can be achieved [177]. Regarding the aforementioned reasons, a better alternative is to normalise by the initial cost C_{initial}, as done in equation (4.22).

4.3 Method and Algorithm Overview

Our method is a combination of a weight reparametrisation and a budget loss, both described in the previous sections. The two are combined in a

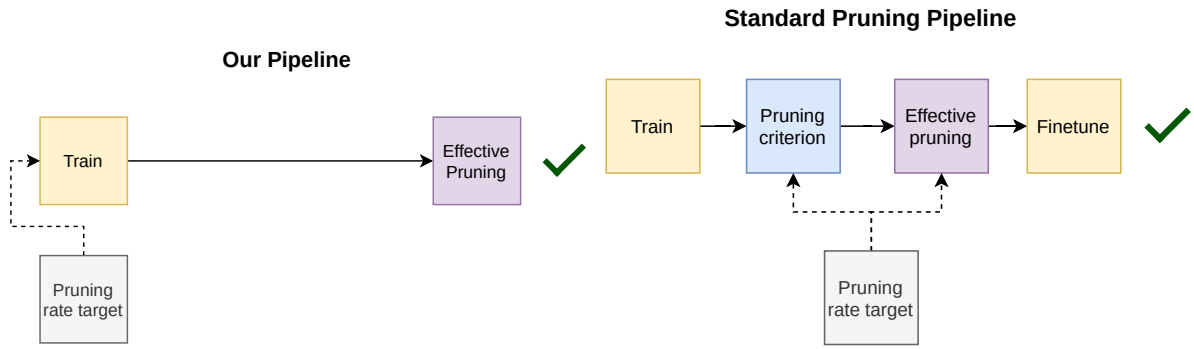
global method that can be used in a standard training procedure using gradient descent. Once the neural network trained using the method detailed in section 4.2, we proceed to prune the smallest weights, w.r.t. their magnitude, to match and enforce the predetermined targeted budget. During this stage, we set the smallest weights to zero until the budget requirement is met. This process is referred to as *effective pruning*.

In sections 4.4.2 to 4.4.5 our results are obtained after following this procedure, which is described in algorithm 2. In other words, the network is first trained with our reparametrisation and our budget loss, then the *effective pruning* step is applied, and finally, the performance is evaluated. In section 4.4.6, we assess the performance of our method with an already trained and pruned initialisation. In this precise setup, since the initialisation is already pruned to match the targeted budget, the *effective pruning* step is not needed and thus not applied.

Algorithm 2 Our training procedure

Require: Dataset $\mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$, network f, weights θ, number of epochs E, mixing coefficient λ, learning rate η, pruning rate p
  for $t = 1$ to E **do**
    for each $(X, y) \in \mathcal{D}$ **do**
      $\mathcal{L} = \mathcal{L}_{task}(y, f(X, \theta_t)) + \lambda \cdot \mathcal{L}^{p}_{budget}(\theta_t)$ {Compute the loss: task loss and budget loss}
      $\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}$ {Backpropagate the loss and update the weights}
    end for
  end for
  return Trained network f
  Perform *effective pruning* on the weights θ: set to 0 the smallest p% of the weights $w \in \theta$
  return Trained and pruned network f

The reader can grasp a better understanding of the key differences of our method compared to the standard pruning pipeline that applies to most pruning methods, not only magnitude pruning method, by looking at figure 4.5. It highlights the fact that the targeted pruning rate is taken into account from the beginning thanks to the budget loss, and therefore, the network does not need a fine-tuning step after the *effective pruning* step. On the contrary, the standard pruning pipeline applies the *pruning*



(a) Our Pipeline **(b)** Standard Pruning Pipeline

Figure 4.5: Principle scheme of our pruning pipeline and the standard pruning pipeline. With our pruning pipeline, the targeted pruning rate that will be enforced during the *effective pruning* step, is taken into account from the beginning. Thus, our method does not need a fine-tuning step. In contrary, the standard pruning pipeline applies the *pruning criterion* and the *effective pruning* after the initial training. This results in a drop in performance that needs to be compensated for with fine-tuning.

criterion and the *effective pruning* after the initial training. This results in a drop in performance that needs to be compensated for with fine-tuning.

4.4 Experiments

In this section, we will study the effectiveness of our method for compressing Convolutional Neural Networks image classification models, as well as its impact on accuracy. To that extent, we will review the impact of both our reparametrisation and our budget loss. For this purpose, we use three reference databases in the field of computer vision: CIFAR-10 [162], CIFAR-100 [162], and TinyImageNet [104], presented in section 2.5. We will evaluate the impact of our method on several neural network architectures: VGG16 [175], Conv4 [43], ResNet18, and ResNet20 [67], introduced in section 2.4.3.

4.4.1 Experimental Setup

Performances of our method are evaluated on CIFAR-10 and CIFAR-100 with Conv4, VGG16 and ResNet20. On TinyImageNet, we evaluate our method on ResNet18. We compare our method against magnitude pruning [59]. The key differences between our method and magnitude pruning

are the following: (*i*) our method uses a budget loss to encourage sparsity, which takes into account the final pruning rate from the beginning of the training process and (*ii*) our method does not require fine-tuning after pruning. Because of the latter, we compare our method against magnitude pruning with and without fine-tuning. Both methods share the following setup: networks are trained during 300 epochs with an initial learning rate of 0.1. A *Reduce On Plateau* policy is applied to the learning rate: if the validation accuracy is not improving for 10 epochs in a row, then the learning rate is decreased by a factor of 0.3. A weight decay is applied on the weights with a penalisation factor of 5×10^{-5}. This value is lower than the more conventional value of 1×10^{-4}, because we want some weights to be able to drift away from the origin, and therefore, escape from the pit of h_t. An Early Stopping policy was used to stop the training prematurely if no improvement in the test accuracy is observed in 60 epochs. To keep the comparison fair, for magnitude pruning, the 300 epochs are split into two phases: the first 150 epochs are dedicated to the training of the network, and the last 150 epochs are used for fine-tuning the pruned network. In the fine-tuning phase, the learning rate is divided by 100 for better convergence.

4.4.2 Performances

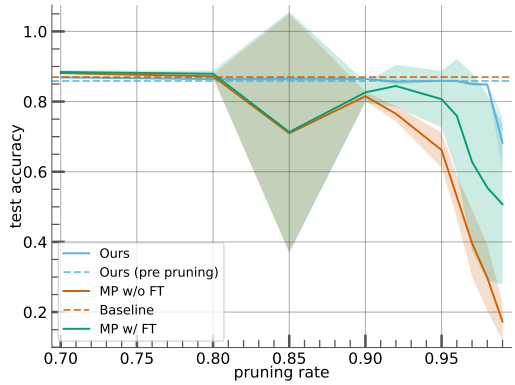
Results are reported on figures 4.6 to 4.8. In these figures, our method (denoted *Ours*) is compared to magnitude pruning with and without fine-tuning (denoted *MP w/ FT* and *MP w/o FT*, respectively). All three methods are evaluated on the test set of the dataset once the network has been pruned up to the pruning rate indicated on the *x-axis*. The test accuracy is reported on the *y-axis* as a float between 0 and 1 (0 being all images wrongly classified and 1 being all images correctly classified). Each solid line representing a method is the mean of 5 independent runs. The coloured area surrounding the solid line represents the standard deviation. In addition to the three methods, the dashed lines represent the performances of an unpruned network trained without weight reparametrisation and budget loss (denoted *baseline*) and the accuracy of our method before the effective pruning (denoted *Ours (pre pruning)*), i.e. before we set the smallest weight, w.r.t. their magnitude, to zero. Sub-figures (c) and (d) of figures 4.6 to 4.8. represents the number of epochs (*y-axis*) needed to obtain the best model for each method, depending on the pruning rate

(*x-axis*).

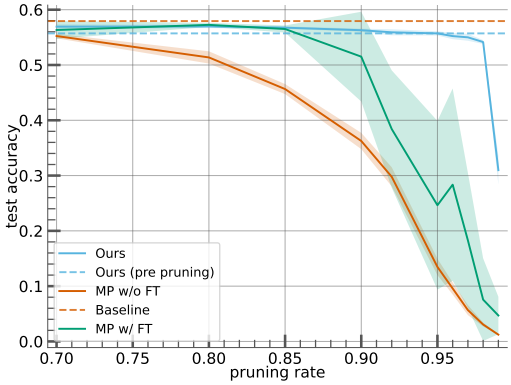
Overall, our method performs consistently better than magnitude pruning without fine-tuning (*MP w/o FT*) and, for almost all pruning rates, better than magnitude pruning with fine-tuning (*MP w/ FT*) on the CIFAR-10 and CIFAR-100 datasets. In particular, our method significantly outperforms magnitude pruning in both setups (with and without fine-tuning) for Conv4 networks (cf. figure 4.6a). For the VGG16 network, we observe the same trend, although the difference is less significant. For ResNet20, magnitude pruning slightly overtakes our method on CIFAR-100 for high pruning rates (more than 90%). Figures 4.6c, 4.6d, 4.7c, 4.7d, 4.8c and 4.8d show that our method requires an equivalent number of epochs compared to magnitude pruning for a higher level of performance (*i.e.* a higher test accuracy). Magnitude pruning requires fewer epochs than our method, only at high pruning rates (more than 90%). On TinyImageNet (figure 4.9), our method outperforms magnitude pruning with and without fine-tuning including at very high pruning rates (95%). In all scenarios (figures 4.6 to 4.9), our method produces much more stable results, and variations from one run to another are significantly smaller than the ones in magnitude pruning. Indeed, the combination of the reparametrisation and the budget loss acts, on the one hand, as a regulariser and, on the other hand, helps to prepare the network for the effective pruning step.

4.4.3 Optimal Value of λ

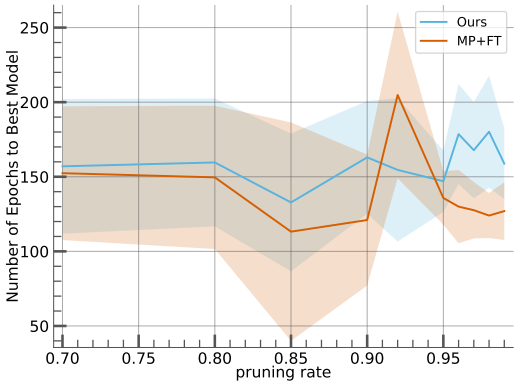
Our method relies on a budget loss whose relative importance compared to the main task loss is controlled by a parameter λ (cf. equation (4.23)). The choice of this parameter is crucial to ensure a good tradeoff between *(i)* adhering to the budget constraint, which ensures that the weights set to zero during the effective pruning step are already vanishingly small if the constraint is satisfied. This implies that zeroing these weights will have a minimal impact on performance; and *(ii)* the optimisation of the main task loss, which directly impacts the final performance. The achieved budget as a function of the parameter λ is shown for different pruning rates in figures 4.10a to 4.10c. In these figures, the achieved budget is computed as the sum of the weight reparametrisations divided by the number of weights in the original network.



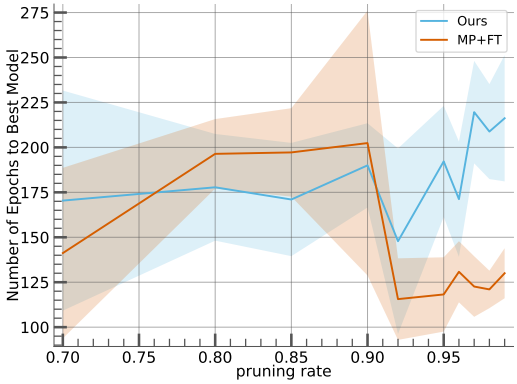
(a) Conv4 - CIFAR-10



(b) Conv4 - CIFAR-100

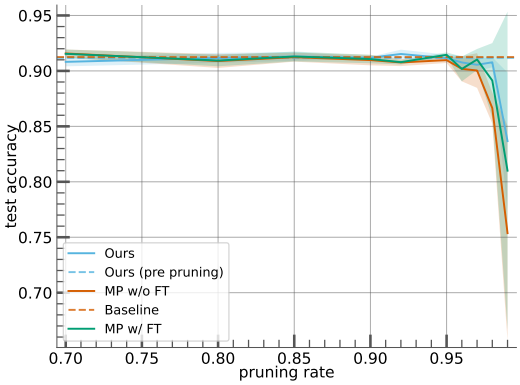


(c) Conv4 - CIFAR-10 (Number of Epochs)

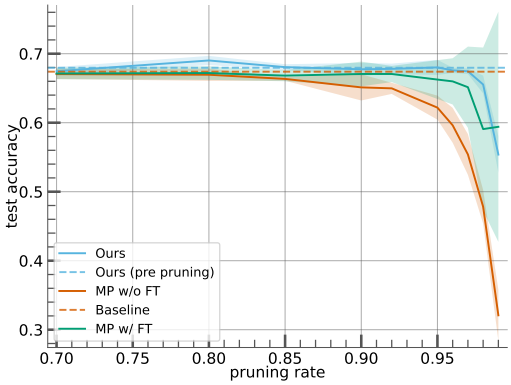


(d) Conv4 - CIFAR-100 (Number of Epochs)

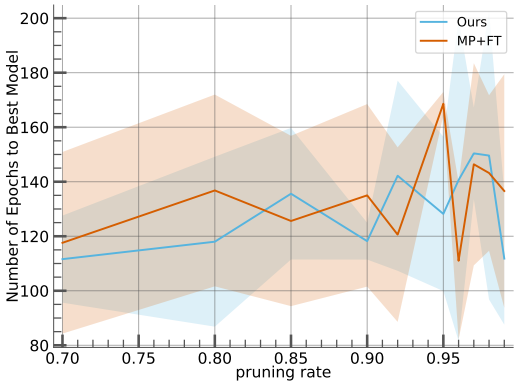
Figure 4.6: Performances comparison of our method *(Ours)* against magnitude pruning without *(MP w/o FT)* and with fine-tuning *(MP w/ FT)* with a Conv4 network on CIFAR-10 and CIFAR-100 datasets, for different pruning rates. Figure 4.6a and figure 4.6b show the testing accuracy of the model and figure 4.6c and figure 4.6d the number of epochs needed to obtain the best model. Best viewed in colours.



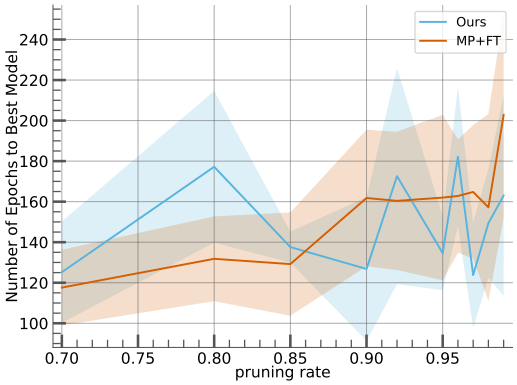
(a) VGG16 - CIFAR-10



(b) VGG16 - CIFAR-100

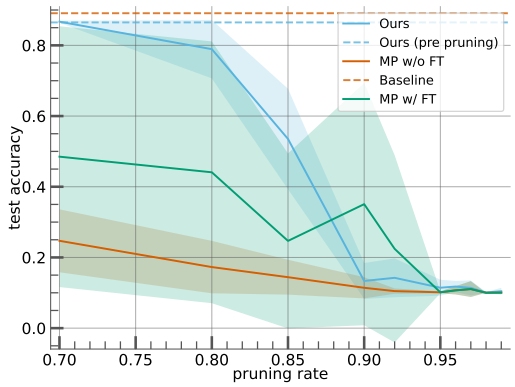


(c) VGG16 - CIFAR-10 (Number of Epochs)

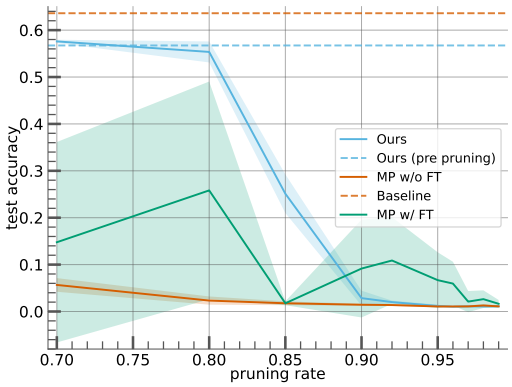


(d) VGG16 - CIFAR-100 (Number of Epochs)

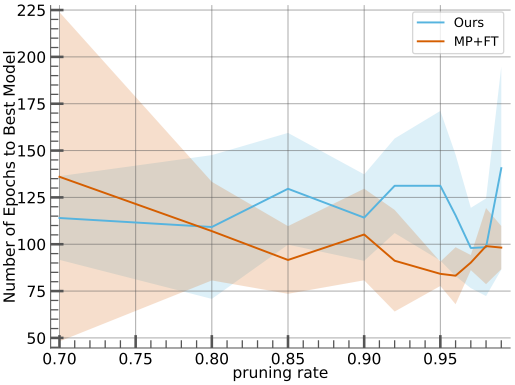
Figure 4.7: Performances comparison of our method *(Ours) against magnitude pruning with fine-tuning* (MP+FT) with a VGG16 network on CIFAR-10 and CIFAR-100 datasets, for different pruning rates. Figure 4.7a and figure 4.7b show the testing accuracy of the model and figure 4.7c and figure 4.7d the number of epochs needed to obtain the best model. Best viewed in colours.



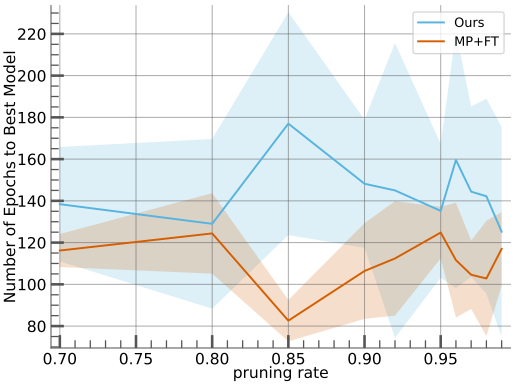
(a) ResNet20 - CIFAR-10



(b) ResNet20 - CIFAR-100



(c) ResNet20 - CIFAR-10 (Number of Epochs)



(d) ResNet20 - CIFAR-100 (Number of Epochs)

Figure 4.8: Performances comparison of our method *(Ours) against magnitude pruning with fine-tuning* (MP+FT) with a ResNet20 network on CIFAR-10 and CIFAR-100 datasets, for different pruning rates. Figure 4.8a and figure 4.8b show the testing accuracy of the model and figure 4.8c and figure 4.8d the number of epochs needed to obtain the best model. Best viewed in colours.

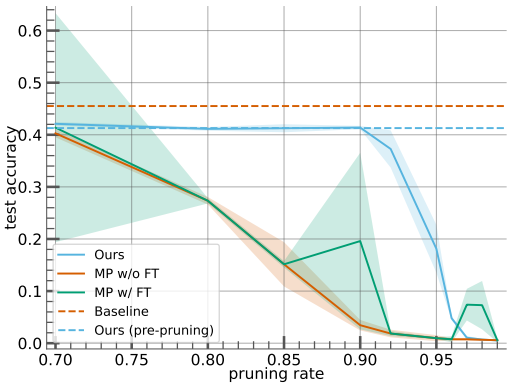
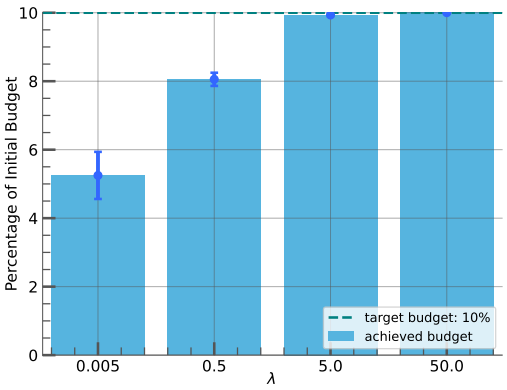


Figure 4.9: Performances comparison of our method *(Ours) against magnitude pruning with fine-tuning* (MP+FT) with a ResNet18 network on TinyImageNet dataset, for different pruning rates.

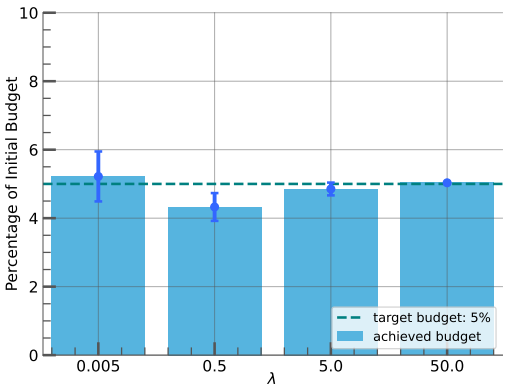
Pruning Rate (%)	λ	**Achieved Budget** (%)	**Test Accuracy (post pruning)** (%)
90	0.005	5.25 ± 0.69	85.83 ± 0.83
	0.5	8.06 ± 0.19	86.34 ± 0.64
	5	9.93 ± 0.03	85.82 ± 0.74
	50	**10.00 ± 0.01**	**86.52 ± 0.46**
	500	10.03 ± 0.00	85.55 ± 0.49
95	0.005	5.22 ± 0.73	**86.27 ± 0.32**
	0.5	4.33 ± 0.41	85.66 ± 0.74
	5	4.85 ± 0.19	86.11 ± 0.48
	50	5.03 ± 0.02	85.37 ± 0.37
	500	**5.00 ± 0.00**	10.00 ± 0.00
99	0.005	4.60 ± 0.29	40.52 ± 5.27
	0.5	3.69 ± 0.38	42.45 ± 9.02
	5	1.89 ± 0.45	**76.85 ± 6.34**
	50	**2.09 ± 0.15**	10.00 ± 0.00
	500	2.28 ± 0.01	10.00 ± 0.00

Table 4.1: Impact of the parameter λ on the achieved budget and the post-pruning test accuracy of the model for a Conv4 network on the CIFAR-10 dataset for various pruning rates. Although a high value of λ ensures the targeted budget is reached, it also leads to a lower test accuracy when the pruning rate increases.

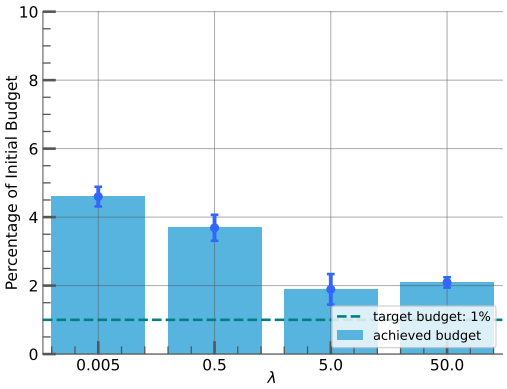
For low pruning rates (figure 4.10a), a relatively low value of λ does not guarantee adherence to the budget constraint and the final network has a smaller achieved budget than the targeted one. Similarly, for higher pruning rates (figure 4.10c), a low value of λ results in a budget in excess compared to the targeted one. In both cases, the performances of networks trained with a low value of λ are subpar compared to higher values, as reported in table 4.1. In contrast to low values of λ, high values might lay too much emphasis on the budget loss, and the network performances are negatively impacted, even though the budget is satisfied or almost satisfied. This is especially the case for a pruning rate of 95% associated with $\lambda = 500$ and also for a pruning rate of 99% for values of $\lambda \geq 50$. Following the abovementioned observations, we set the value of λ to 5 for all the experiments. This value strikes the best balance between the two objectives: budget loss and main task loss. Note that various scheduling for λ have been considered, but they do not significantly improve performance (see appendix A.2).



(a) Pruning 90% of the weights



(b) Pruning 95% of the weights



(c) Pruning 99% of the weights

Figure 4.10: Impact of the parameter λ on the achieved final budget for a Conv4 network on CIFAR-10 dataset, for various pruning rates. A too-small value of λ does not make the actual budget match the desired budget. The actual budget is either too small (figure 4.10a) or too high (figure 4.10c) compared to the target, depending on the applied pruning rate.

4.4.4 Validation of the Budget Loss

In order to establish the importance and the effectiveness of the budget loss in our method, we present in this section the results of a comparative experimental analysis with alternative variants. Specifically, we investigated the impact of the budget loss by comparing it with two other variants: *(i)* a variant where the budget loss is removed, and *(ii)* a variant where the budget loss is replaced with a regularisation loss based on the ℓ_1 norm of the network weights. In order to remove the budget loss, the value of λ is set to zero 0 in equation (4.23). In the second variant, we varied the mixing coefficient λ between 0.1 and 100. Considering the same issue of loss conditioning as in section 4.2.2, the ℓ_1 norm is divided by the total number of parameters, denoted N, before being added to the global loss. This specific global loss is expressed as:

$$\mathcal{L} = \mathcal{L}_{task} + \lambda \cdot \frac{1}{N} \sum_{\ell=1}^{L} ||\mathbf{w}_\ell||_1 \tag{4.25}$$

Where $||\ .\ ||_1$ represents the ℓ_1. Values of N for most of the used architectures are reported in table 2.1.

In both variants, the reparametrisation is kept in order to isolate the impact of the budget loss. We evaluated the performance of our approach and the two variants on the CIFAR-10 and CIFAR-100 datasets using Conv4, VGG16, and ResNet20 networks. The results are presented in figures 4.11 to 4.13. In the figures mentioned above, the variant without the budget loss is denoted *w/o budget*, and the variant with a ℓ_1 regularisation loss is referred to as ℓ_1 *reg.*. The results denoted *w/ budget* represents our method in the same setup as in section 4.4.2.

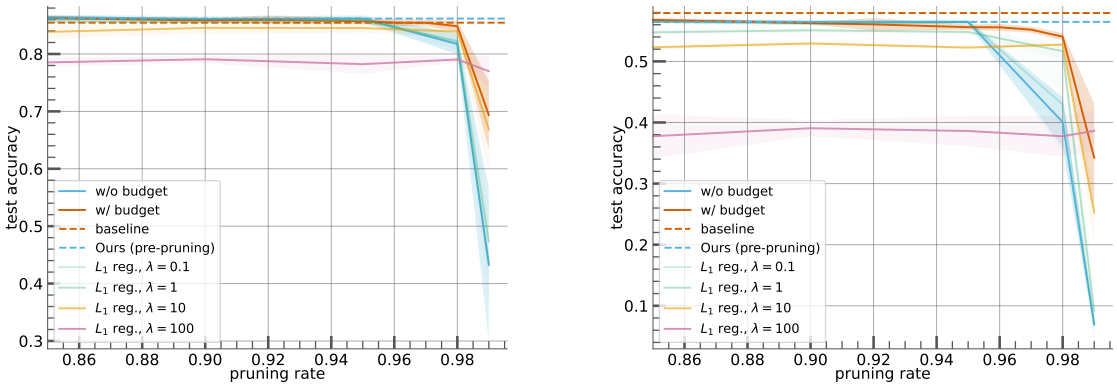
Removing the budget loss (variant *(i) - w/o budget*) negatively impacts the network performances. The test accuracy is systematically lower than the one obtained with the budget loss. This is particularly visible in figures 4.11b and 4.13b. Removing the budget loss does not push the optimisation to introduce sparsity, let alone to respect the targeted budget. Since sparsity was not introduced beforehand, the effective pruning step has a negative impact on the network performance. Indeed, the latter was not trained with a prior on either targeted sparsity or targeted budget, embedded in the loss.

Replacing the budget loss with a ℓ_1 regularisation loss (variant *(ii)* - ℓ_1 *reg.*) also impacts negatively the performance, with the exception of the ResNet20 network (figure 4.12). Although performances are generally worse than our method (*w/ budget*), results indicate that the mixing coefficient λ has major importance. Indeed, the ℓ_1 regularisation does not target a precise budget, however, it still helps optimisation to introduce sparsity in the network. Thus, for certain pruning rates, the variant *(ii)* can exhibit better results than our method (especially visible on figures 4.12a and 4.12b). Nevertheless, the choice of λ is critical and not trivial. Because of the absence of a budget loss, the optimisation does not take any targeted level of sparsity into account. The tradeoff between optimising the main task loss and the ℓ_1 regularisation loss is controlled only by the parameter λ, which makes it difficult to find a single value suited for a large range of pruning rates, networks and datasets. Since variant *(ii)* does not target any specific sparsity, in figures 4.11 to 4.13, we vary the pruning rate to examine the performance of trained networks at various sparsity levels.

In contrast, our method is able to achieve good performance across multiple and diverse conditions, without the need to try different values of λ. Experiments presented in this section reveal that the budget loss is a critical component of the method to train networks and prune them while introducing a minimal impact on the performance if no fine-tuning is applied. While the ℓ_1 regularisation loss may achieve superior performance in certain cases, our method is simpler to implement since it does need to search for a value of λ per architecture, and more robust in its applicability across various scenarios.

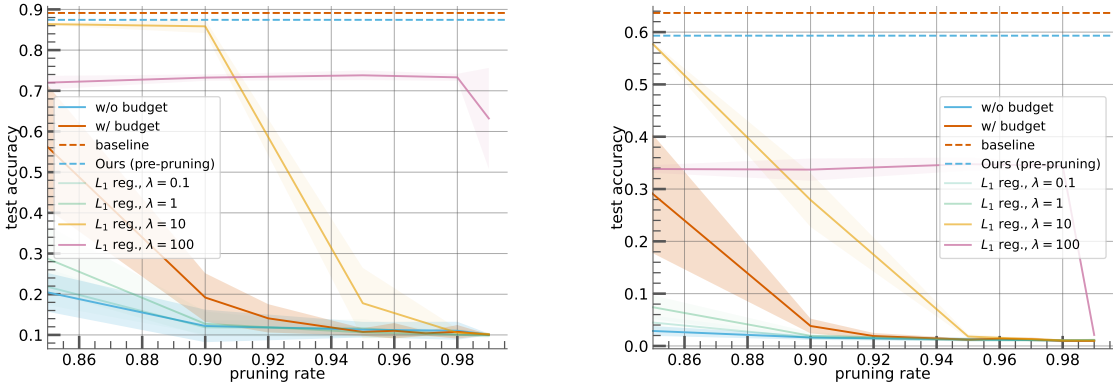
4.4.5 Validation of the Reparametrisation

The proposed method comprises two primary components: budget loss and weight reparametrization. The previous section establishes the significance of the budget loss in achieving optimal performance. In this section, we study the impact of incorporating weight reparametrisation. To establish the necessity of weight reparametrisation, we compare our approach with a variant where the budget loss is applied but the weight reparametrisation is not. This variant is denoted *budget only* in the following. The objective of this variant and the comparison is to isolate the impact of weight reparametrization. The budget loss is evaluated in the same way as described in



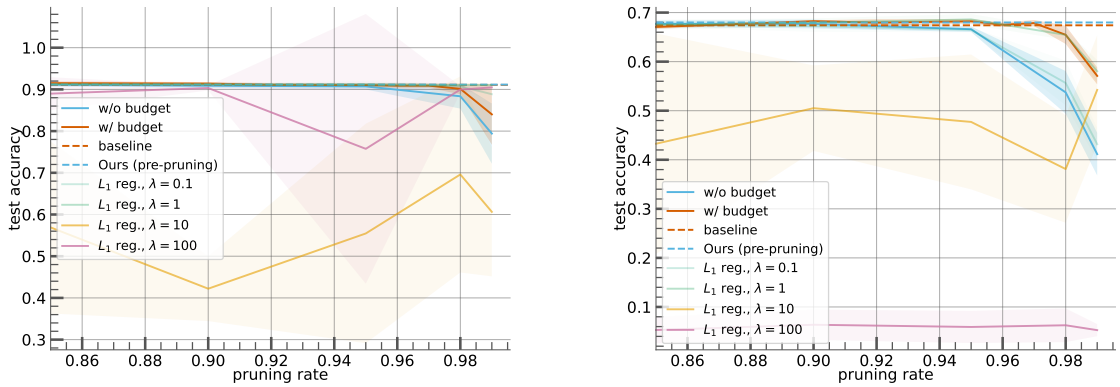
(a) Conv4 CIFAR-10 **(b)** Conv4 CIFAR-100

Figure 4.11: Comparison of our method and its variant without the budget loss. The experimental results are referred to as ℓ_1 *reg.*, wherein the budget loss is replaced by a ℓ_1 regularisation loss on the network weights. The mixing coefficient λ is varied from 0.1 to 100, depending on the experiment. *w/o budget* corresponds to the absence of the budget loss (this is equivalent to $\lambda = 0$). On the other hand, *w/ budget* corresponds to our method, with the same setup as described in section 4.4.2. Results are presented for a Conv4 network, trained on CIFAR-10 (figure 4.11a) and CIFAR-100 (figure 4.11b). Best viewed in colours.



(a) ResNet20 CIFAR-10 **(b)** ResNet20 CIFAR-100

Figure 4.12: Comparison of our method and its variant without the budget loss. The experimental results are referred to as ℓ_1 *reg.*, wherein the budget loss is replaced by a ℓ_1 regularisation loss on the network weights. The mixing coefficient λ is varied from 0.1 to 100, depending on the experiment. *w/o budget* corresponds to the absence of the budget loss (this is equivalent to $\lambda = 0$). On the other hand, *w/ budget* corresponds to our method, with the same setup as described in section 4.4.2. Results are presented for a ResNet20 network, trained on CIFAR-10 (figure 4.12a) and CIFAR-100 (figure 4.12b). Best viewed in colours.



(a) VGG16 CIFAR-10 **(b)** VGG16 CIFAR-100

Figure 4.13: Comparison of our method and its variant without the budget loss. The experimental results are referred to as ℓ_1 *reg.*, wherein the budget loss is replaced by a ℓ_1 regularisation loss on the network weights. The mixing coefficient λ is varied from 0.1 to 100, depending on the experiment. *w/o budget* corresponds to the absence of the budget loss (this is equivalent to $\lambda = 0$). On the other hand, *w/ budget* corresponds to our method, with the same setup as described in section 4.4.2. Results are presented for a VGG16 network, trained on CIFAR-10 (figure 4.13a) and CIFAR-100 (figure 4.13b). Best viewed in colours.

section 4.2.2. Note that in the *budget only* variant, the budget evaluation uses our reparametrisation function as a surrogate ℓ_0 norm but the weights used to produce the network output are the standard weights $\mathbf{w}$, not the reparametrised weights $\hat{\mathbf{w}}$.

We evaluate the performance of our method and the *budget only* variant on Conv4, ResNet20, and VGG16 networks using CIFAR-10 and CIFAR-100 datasets (see figures 4.14 to 4.16). Both the proposed approach and *budget only* variant are trained with the same hyperparameters, namely, the learning rate, weight decay, number of epochs and the *Reduce on Plateau* policy for the learning rate. For the *budget only* variant, we perform experiments by varying the mixing coefficient λ from 0.5 to 500. The performances did not vary significantly w.r.t. λ, suggesting that the mixing coefficient does not have a significant impact on the performance. Therefore, in order to ensure the clarity of figures 4.14 to 4.16, we only display the results for $\lambda = 5$.

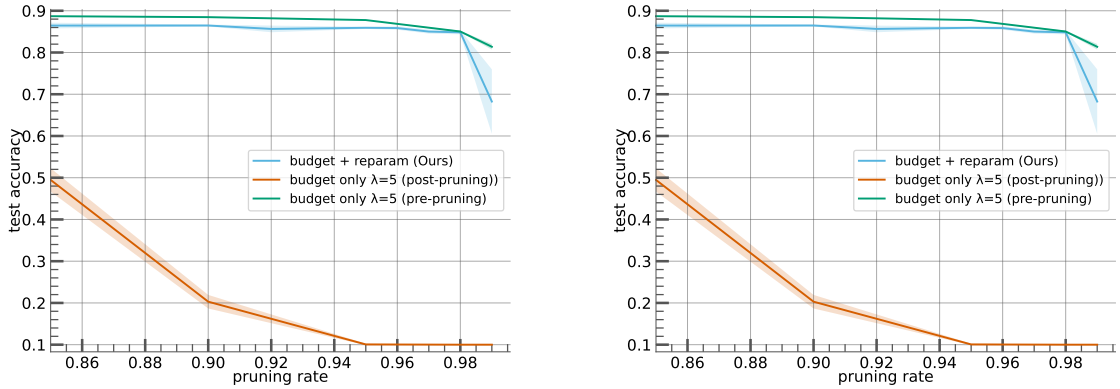
Figures 4.14 to 4.16 present the results of the performance comparison. Our method is referred to as *budget + reparam* and is evaluated after pruning, whereas the *budget only* variant results are presented both before and after pruning. On Conv4 and VGG16 (figures 4.14 and 4.16,

Dataset	Network	Pruning Rate (%)	Achieved Budget (%)
CIFAR-10	Conv4	90	9.99 ± 0.00
		95	4.97 ± 0.01
		99	0.98 ± 0.00
	ResNet20	90	9.83 ± 0.02
		95	4.88 ± 0.01
		99	0.98 ± 0.00
	VGG16	90	10.00 ± 0.00
		95	5.00 ± 0.00
		99	1.00 ± 0.00
CIFAR-100	Conv4	90	9.94 ± 0.02
		95	4.91 ± 0.02
		99	0.98 ± 0.00
	ResNet20	90	9.84 ± 0.02
		95	4.91 ± 0.01
		99	1.00 ± 0.00
	VGG16	90	10.00 ± 0.00
		95	5.00 ± 0.00
		99	1.00 ± 0.00

Table 4.2: Achieved budget for the *budget only* variant. Results are presented for $\lambda = 5$. Across all experiments, the achieved budget matches closely the targeted budget, which is computed as (1−pruning rate)×100 and is expressed in percent.

respectively), our method performs on par with the *budget only* variant before pruning while being already pruned, up to very high pruning rates (more than 98%). On the contrary, and even for ResNet20, the *budget only* post-pruning variant performs poorly. The *budget only* variant performance is massively impaired by the *effective pruning* step, even though the budget is thoroughly respected (table 4.2). In comparison, our method performs much better than the latter when *effective pruning* is applied to both methods. The budget loss alone enforces a stricter adherence to the targeted budget (table 4.2), however, the lack of reparametrisation fails to prepare the network for the *effective pruning* step. Indeed, weights are not soft-pruned and the network is not prepared for sparsity.

The results presented in this comparison and the ones of section 4.2.2 show that both components of our method are of crucial importance. In



(a) Conv4 CIFAR-10 **(b)** Conv4 CIFAR-100

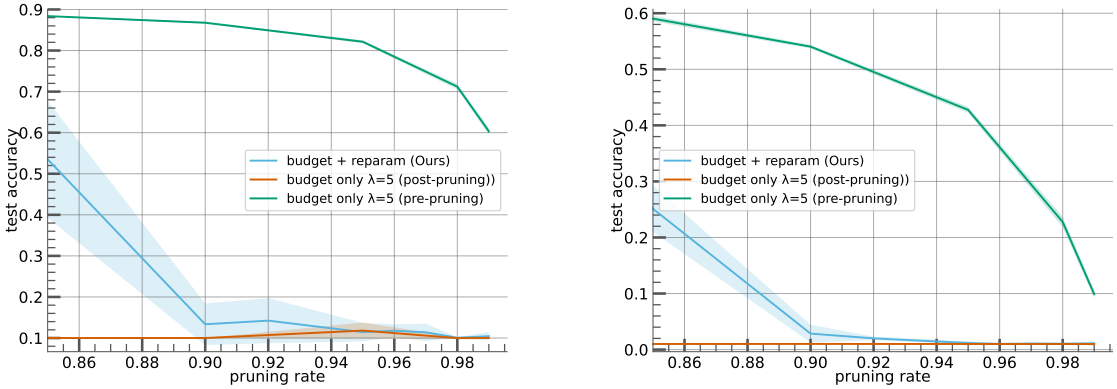
Figure 4.14: Comparison of our method and its variant without the reparametrization on Conv4, evaluated on CIFAR-10 and CIFAR-100. Our method (*budget + reparam*) has similar performance to the *budget only* variant before pruning, whereas our method, is already pruned. Once pruned, the *budget only* variant is significantly impaired.

particular, the reparametrization allows for a considerably better generalization of the network after pruning, thus enabling a much higher level of performance. Sections 4.4.4 and 4.4.5 provide empirical evidence that no component of our method can be removed without significant impairment of the performance, and therefore, they function in synergy.

4.4.6 Tuned Initialisation

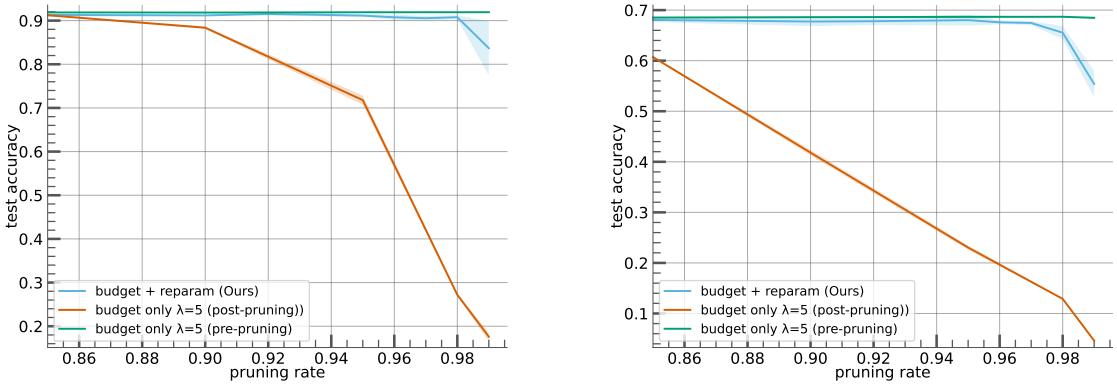
In the previous sections, weights were initialised with the standard Kaiming initialisation scheme [66] (see appendix A.3). In this section we study an alternative initialisation scheme: we initialise the weights of the network with trained and pruned weights. These weights are obtained by training the network in its standard configuration (*i.e.* without reparametrisation and budget loss) up to convergence and then the weights are pruned with magnitude pruning at a specified pruning rate. In other words, our method is used to fine-tune the weights of a trained and pruned network. This fine-tuning setup is of particular interest since the major deep learning frameworks [147, 1] provides pretrained weights for various architectures [154] but it is up to the end user to prune and fine-tune them according to their needs.

We compare the performances of a network trained in a standard way, then pruned with magnitude pruning and finally fine-tuned with two meth-



(a) ResNet20 CIFAR-10 **(b)** ResNet20 CIFAR-100

Figure 4.15: Comparison of our method and its variant without the reparametrization on ResNet20, evaluated on CIFAR-10 and CIFAR-100. Due to the small size of the network (see table 2.1), the pruned version of our method (*budget + reparam*) and the *budget only* variant cannot keep up with the unpruned version. Nevertheless, if considering the pruned versions, our method scores better, thanks to the addition of the reparametrization.

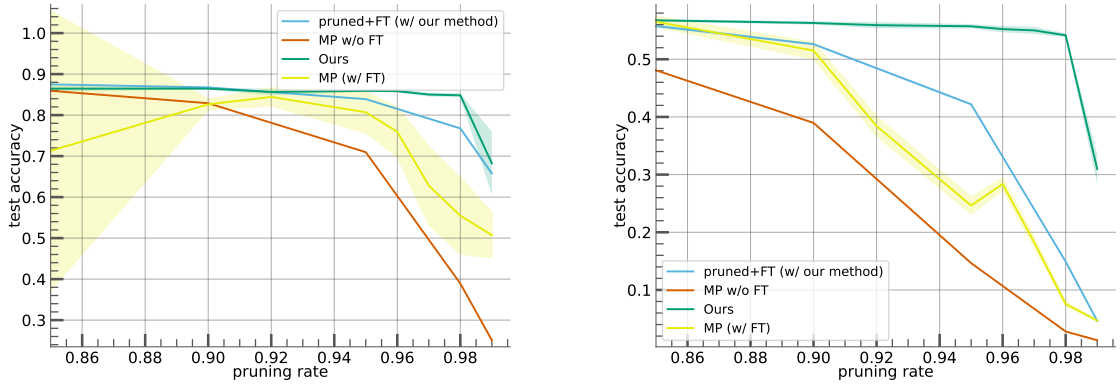


(a) VGG16 CIFAR-10 **(b)** VGG16 CIFAR-100

Figure 4.16: Comparison of our method and its variant without the reparametrizationn VGG16, evaluated on CIFAR-10 and CIFAR-100. Our method (*budget + reparam*) has similar performance to the *budget only* variant before pruning, whereas our method, is already pruned. Once pruned, the *budget only* variant is significantly impaired.

ods: Our method and standard fine-tuning [59]. Put simply, this section describes a process where the initial weights of the network are not randomly initialised, but are trained and pruned weights. Note that the pruning definitely zeroes out the smallest magnitude weights and they are not reactivated during the fine-tuning. The latter only tunes the remaining unpruned weights. This setup is evaluated on Conv4, ResNet20 and VGG16 for both CIFAR-10 and CIFAR-100. The results are shown in figures 4.17 to 4.19. First, a network is trained for 150 epochs on the main classification task. Then it is pruned up to a specified pruning rate. The pruning criterion used is the magnitude of the weights where weights with the smallest absolute values are removed in an unstructured way. The pruned network is then fine-tuned for 300 epochs with an early stopping criterion based on the validation accuracy. The training is stopped prematurely if the validation accuracy does not improve for 30 epochs. When fine-tuned with our methods, the pruned network is treated as the original network. The initial latent weights of our method ($\mathbf{w}$) are the ones of the trained and pruned network. They are reparametrised as in 4.2.1.

Except for results with the Conv4 networks, fine-tuning a network with our method overperforms the conventional fine-tuning method by a comfortable margin on ResNet20 across all pruning rates (figure 4.18) and VGG16 (figure 4.19) for pruning rates higher than 96%. At initialisation, the weights of the network are already pruned to the targeted pruning rate. With the enforcement of the budget through budget loss, the remaining unpruned weights cannot dwindle to zero, hence preserving them during the fine-tuning process. In contrast, in the absence of budget loss, the remaining weight values are not restricted, allowing them to possibly vanish, resulting in further a drop in network capacity with weights of vanishingly small magnitude. This behaviour is highlighted by the following additional experimental results displayed in figure 4.20. This experiment involves a comparison between two initial setups: one where the initial weights are only fine-tuned and another where they are fine-tuned and then pruned. The outcomes demonstrate that initialising a network with weights that are fine-tuned and then pruned beforehand yields significantly superior results. Moreover, the networks produced by fine-tuning with our method are much more consistent from one run to another. This is illustrated by the standard deviation being so small that the coloured area around the solid line is barely visible on the graphs. This is not the case for conventional fine-tuning where performances vary greatly from one run to another. This



(a) Conv4 CIFAR-10 **(b)** Conv4 CIFAR-100

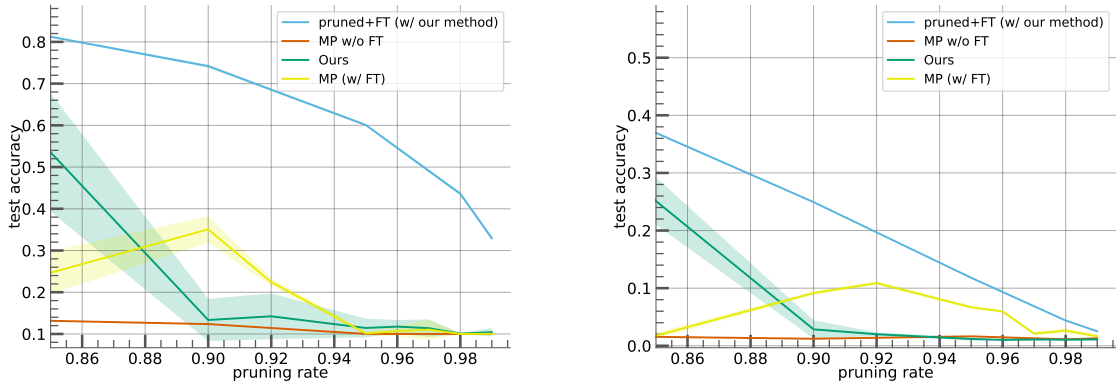
Figure 4.17: Fine-tuning of a Conv4 network pruned by magnitude pruning (*MP w/o FT*) on the CIFAR-10 and CIFAR-100 datasets for various pruning rates. Conventional (*MP w/ FT*) fine-tuning is compared to fine-tuning with our method (*pruned+FT (w/ our method)*). Our method, described in section 4.3, is shown for comparison purposes (*Ours*). On this network, our method performs better than other approaches. Fine-tuning the network with our method provides better results than fine-tuning it with a conventional method.

is especially visible in figure 4.17a.

Although the method proposed in this chapter was not initially intended for fine-tuning networks, it demonstrates superior performance compared to widely-used standard fine-tuning techniques. Notably, it exhibits enhanced recovery from the performance decline that occurs following pruning. Furthermore, the performance consistency of networks fine-tuned with our method is significantly higher across multiple runs in comparison to networks generated by traditional fine-tuning methods. The slight variation observed in the results indicates increased robustness against the inherent randomness of the optimization process. Consequently, the final performance is less dependent on specific initializations, batch orders, or data augmentation seeds than with standard fine-tuning approaches.

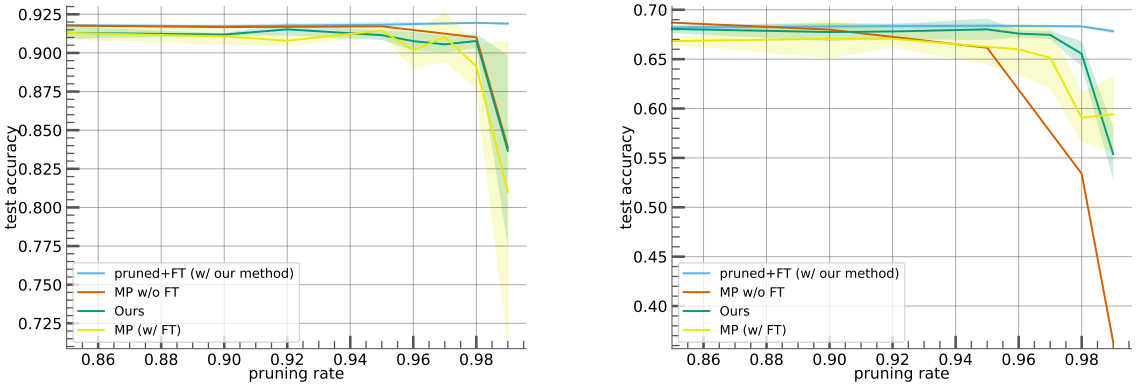
4.5 Conclusion

This chapter introduces a novel approach for pruning neural networks, which addresses the limitations of conventional pruning pipelines. The latter usually apply a pruning criterion and prune networks after an initial training phase without taking into account the target pruning rate. This



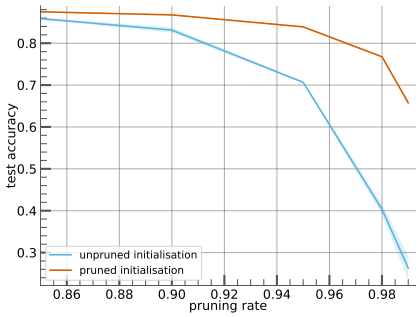
(a) ResNet20 CIFAR-10 **(b)** ResNet20 CIFAR-100

Figure 4.18: Fine-tuning of a ResNet20 network pruned by magnitude pruning (*MP w/o FT*) on the CIFAR-10 and CIFAR-100 datasets with various pruning rates. Conventional (*MP w/ FT*) fine-tuning is compared to fine-tuning with our method (*pruned+FT (w/ our method)*). Our method, described in section 4.3, is shown for comparison purposes (*Ours*). On this network, fine-tuning with our method considerably outperforms other approaches.

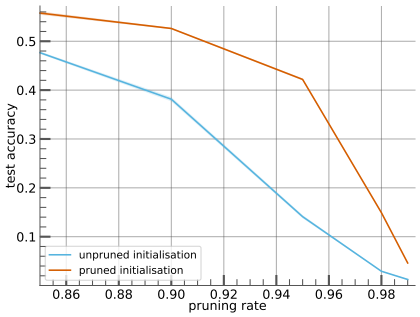


(a) VGG16 CIFAR-10 **(b)** VGG16 CIFAR-100

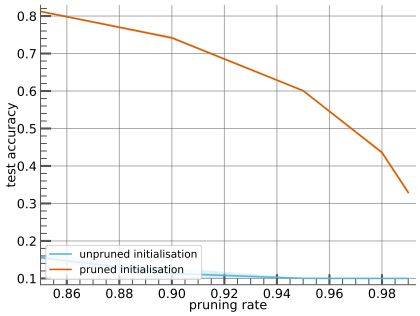
Figure 4.19: Fine-tuning of a ResNet20 network pruned by magnitude pruning (*MP w/o FT*) on the CIFAR-10 and CIFAR-100 datasets with various pruning rates. Conventional (*MP w/ FT*) fine-tuning is compared to fine-tuning with our method (*pruned+FT (w/ our method)*). Our method, described in section 4.3, is shown for comparison purposes (*Ours*). On this network, fine-tuning with our method performs on par with other methods up to 95% of pruning. For higher pruning rates, it outperforms other approaches.



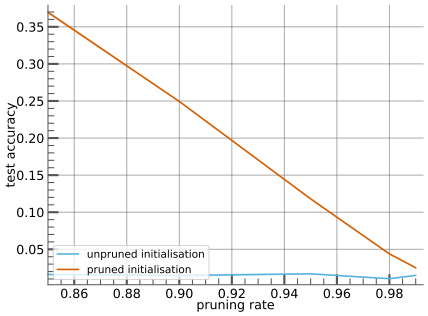
(a) Conv4 - CIFAR-10



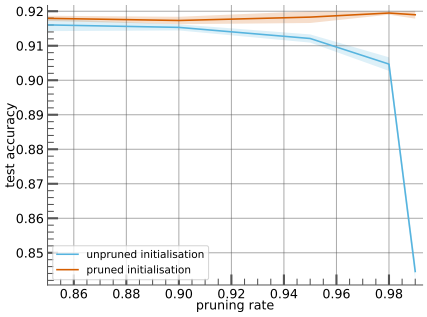
(b) Conv4 - CIFAR-100



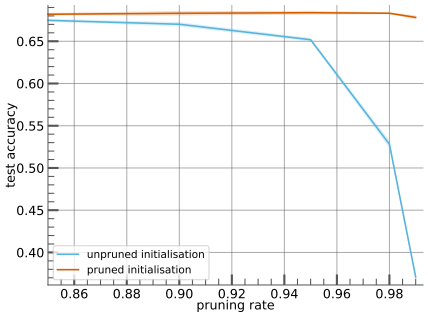
(c) ResNet20 - CIFAR-10



(d) ResNet20 - CIFAR-100



(e) VGG16 - CIFAR-10



(f) VGG16 - CIFAR-100

Figure 4.20: Comparison of fine-tuning a network whose initialisation has been trained from scratch (denoted *unpruned initialisation*) or trained from scratch and pruned with magnitude pruning (denoted *pruned initialisation*). Fine-tuning a pruned initialisation always outperforms fine-tuning an unpruned initialisation in the tested configurations.

often requires a fine-tuning phase to restore the accuracy loss that follows pruning and the subsequent topology alteration. In contrast, the proposed method does not require fine-tuning to achieve superior performance compared to state-of-the-art magnitude pruning methods. Experimental evaluations conducted on various datasets and networks commonly used for image classification benchmarking validate this claim.

Our approach described in this chapter consists of two key components: a budget loss and a weight reparametrisation function. Comparative analyses demonstrate the importance of both components, as variants without either the budget loss or the reparametrisation, result in inferior performance compared to the full-fledged method. While the proposed method is designed to avoid computationally intensive fine-tuning, it can still be used for fine-tuning and performs comparatively better than standard fine-tuning.

The proposed method of this chapter focuses on weight reparametrisation with budget loss to enhance the network robustness to pruning: compared to a network trained in the absence of such strategies, our use of reparametrisation and budget loss substantially mitigates the performance degradation typically induced by pruning. This forces less useful weights to take small values, binding the weight value to the network topology. As such, the optimisation process learns both the weights and topology under the hypothesis that weights with smaller magnitudes will be removed. This method highlights the importance of determining the optimal topology in addition to the optimal weights, achieved in this chapter with the prior that magnitude is a saliency factor for weight relevance in the topology.

Notwithstanding the improved performance of the proposed method compared to magnitude pruning, it still suffers from some limitations. Namely, the value of the weight saliency is bound to the reparametrisation which is in turn bound to the weight value. As a consequence, the weight saliency is only determined by the latter. The resulting limitation is that the current method cannot treat weights with the same value (or similar values) differently. Put simply, the topology is bound to the magnitude of the weights, but not their position in the network.

In the next chapter, the introduced approach seeks to determine the optimal topology without training the weights and without binding their

saliency (or relevance) to their magnitude. The saliency of the initial weights is determined by a trained mask. In other words, the next method aims at determining the best topology, given a set of fixed weights.

Chapter 5

Effective Subnetworks Extraction without Weight Training

Contents

Chapter Abstract

This chapter focuses on the development of lightweight and efficient neural networks for image classification tasks, particularly in visual category recognition. These lightweight networks are increasingly important for intelligent embedded systems with limited computational and energy resources. Pruning techniques are popular in designing lightweight networks, but they require weight training, pruning and fine-tuning. These

weights are pruned based on criteria or saliency indicators that are learned alongside the weights.

This chapter introduces approaches that extract effective subnetworks by pruning large untrained networks, without weight training. A new method, named Arbitrarily Shifted Log Parametrisation (ASLP), is proposed to extract effective subnetworks from a large, untrained deep neural network using the Straight Through Gumbel-Softmax (STGS) technique, which enables the training of stochastic discrete variables while still preserving differentiability. Additionally, a weight rescaling mechanism, referred to as Smart Rescale (SR), is introduced. It rescales the weight distributions of the selected subnetworks and as a result, improves the performance and reduces the number of epochs required for training as shown later in experiments. Finally, we introduce a novel pruning strategy that automatically finds the pruning rate yielding the best performances once the training is completed, eliminating the need to iteratively search and strictly enforce a specific pruning rate throughout the training.

The ASLP method, which integrates the STGS sampling technique and the SR mechanism, is evaluated through experiments on CIFAR-10, CIFAR-100, and TinyImageNet datasets. In most cases, ASLP outperforms other state-of-the-art methods and consistently surpasses them for various network architectures. Further experiments show that the sparsity of the networks extracted with ASLP can be increased with minimal impact on the performance. These experiments also show that our method can accept a broad range of learning rates and is robust to extremely large learning rate values. Additionally, the experiments show the effectiveness of the SR mechanism regarding performance improvement and the reduction in the number of epochs needed to reach convergence.

This chapter presents work that has resulted in the publication of the following conference article:

- Robin Dupont, Mohammed Amine Alaoui, Hichem Sahbi, and Alice Lebois. Extracting effective subnetworks with Gumbel-Softmax. In *2022 IEEE International Conference*

on Image Processing, ICIP 2022, Bordeaux, France, 16-19 October 2022, pages 931–935. IEEE, 2022.

Our code for the ASLP method, as well as the reimplementation of the comparative methods used in this chapter, is publicly available at:

- `https://github.com/N0ciple/ASLP`

5.1 Introduction and Related Work

In this chapter, we tackle the challenge of extracting lightweight and effective subnetworks from large, untrained neural networks. The goal is to obtain subnetworks that have both a smaller number of parameters compared to the original network and compelling performances. Contrary to chapter 4 where the focus was on soft-pruning weights based on their value during the training, the method presented in this chapter relies on training latent masks. The latter represents a parametrisation of the probabilities for the weights to be selected or not in a sampled topology. The method described in chapter 4 binds the saliency of the weights to their magnitude in a fully deterministic way. In contrast, the method introduced in this chapter is, on the one hand, stochastic, and on the other hand, decorrelating the pruning criterion from the weight value by relying on a separate latent mask to represent the weight saliency.

This method exhibits several advantages over the method presented in the previous chapter: First, as we just mentioned, the pruning criterion is decorrelated from the value of the weight, offering a different approach where the weight value is not the only parameter determining its saliency. Indeed, two weights with the same value can have different probabilities of being selected since their associated latent mask can evolve differently. Second, this method does not train the weights but selects the best subset of network connections to minimise the loss used to train the network. Consequently, this method is perfectly suited for applications where training the weight might not be possible and activating or deactivating a weight is the only option. Finally, from a global standpoint, this method provides a new path to neural network training that does not rely on weight training, but rather on topology selection through the same widely spread gradient-based optimisation framework used by standard training. The following sections provide an overview of the related works on pruning at

initialisation, Lottery Tickets, effective subnetwork existence and extraction, followed by the contributions of this chapter.

5.1.1 Pruning at initialisation

In general, extracting a lightweight subnetwork is still a challenging problem [46] and is computationally demanding as this amounts to full training of large networks (until convergence) prior to their pruning. Instead of pruning the network after an initial training phase, existing alternatives approach this problem by pruning the network weights just after their initialisation and before training [107, 196, 186]. The resulting sparse network is then trained after this initial pruning step, as shown in figure 5.1.

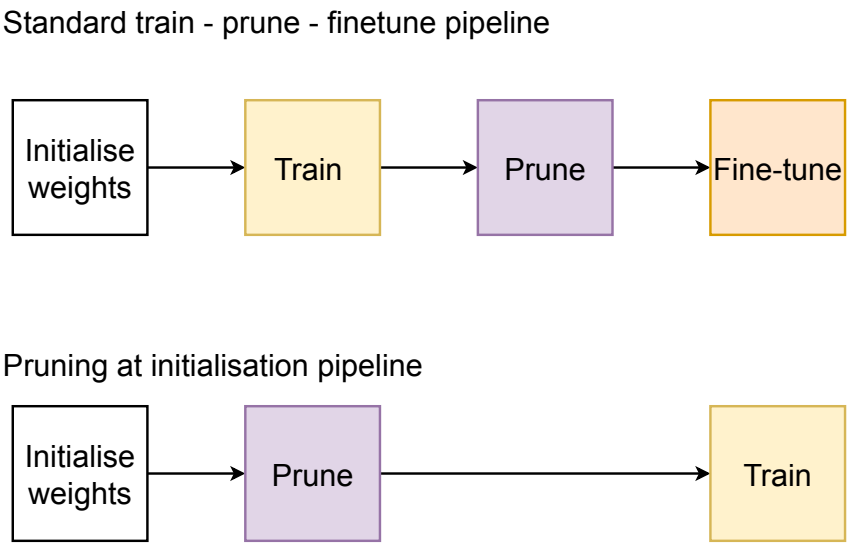


Figure 5.1: Comparison of a standard *train-prune-finetune* pipeline and the *prune at initialisation* pipeline. In the latter, the network is pruned before training.

Single-Shot Network Pruning (SNIP) was introduced by Lee et al. in [107]. The authors devise a new criterion to determine the importance of a connection even before the start of training. The criterion, called *connection sensitivity*, is based on the influence of a connection on the loss function. The more a connection can change the loss function output, the more important it is considered to be. Considering a weight w_j, the authors define its sensitivity as:

$$s_j = \frac{\left|\frac{\partial \mathcal{L}}{\partial c_j}\right|}{\sum_{k=1}^{N} \left|\frac{\partial \mathcal{L}}{\partial c_k}\right|} \tag{5.1}$$

where N is the number of connections in the network and c_j in an auxiliary variable introduced by the authors that represents the presence ($c_j = 1$) or absence ($c_j = 0$) of the weight w_j in the network. The connections are then sorted by their connection sensitivity score and the top-k connections are kept to match a given pruning rate.

GraSP (Gradient Signal Preservation) [196] is a refinement of SNIP that takes into account the *gradient flow*. The authors seek to preserve the latter in order to allow large gradients in the subsequent network training. The scores of the weights are defined in a vectorised way as :

$$\mathbf{S}(-\mathbf{w}) = -\mathbf{w} \odot \mathbf{Hg} \tag{5.2}$$

where $\odot$ is the Hadamard product, $\mathbf{w}$ is the vector of weights, $\mathbf{H}$ is the Hessian matrix of the loss function with respect to the weights and $\mathbf{g}$ is the gradient of the loss function with respect to the weights. Considering how the score is defined in equation (5.2), pruning is achieved by removing the top-k weights that reduce the gradient flow to match a given pruning rate.

Both SNIP and GraSP require a single mini-batch to compute their respective scores. Another pruning method known as SynFlow [186] is data-free and seeks to preserve *synaptic flow*, defined subsequently, in a given network in order to prevent *layer collapse*. The latter is defined by the authors as the complete pruning of a layer, which effectively renders the network untrainable. For a given layer ℓ, the *synaptic flow* score of the weights θ_ℓ of a layer ℓ is defined as :

$$\mathcal{S}_{\mathrm{SF}}(\mathbf{w}_\ell) = \frac{\partial \left(\mathbb{1}^T \left(\prod_{\ell=1}^{L} |\mathbf{w}_\ell| \right) \mathbb{1} \right)}{\partial \, \mathbf{w}_\ell} \odot \mathbf{w}_\ell \tag{5.3}$$

where L is the total number of layers in the network and $\mathbb{1}$ is the all ones vector. Contrary to previous methods, namely SNIP [107] and GraSP [196], SynFlow does not necessitate any data to compute the scores. These scores are computed and updated iteratively for 100 steps, regardless of the targeted dataset or batch size. Since it prevents layer collapse, network pruning with SynFlow can reach higher pruning rates than with SNIP or GraSP (see figure 5.2).

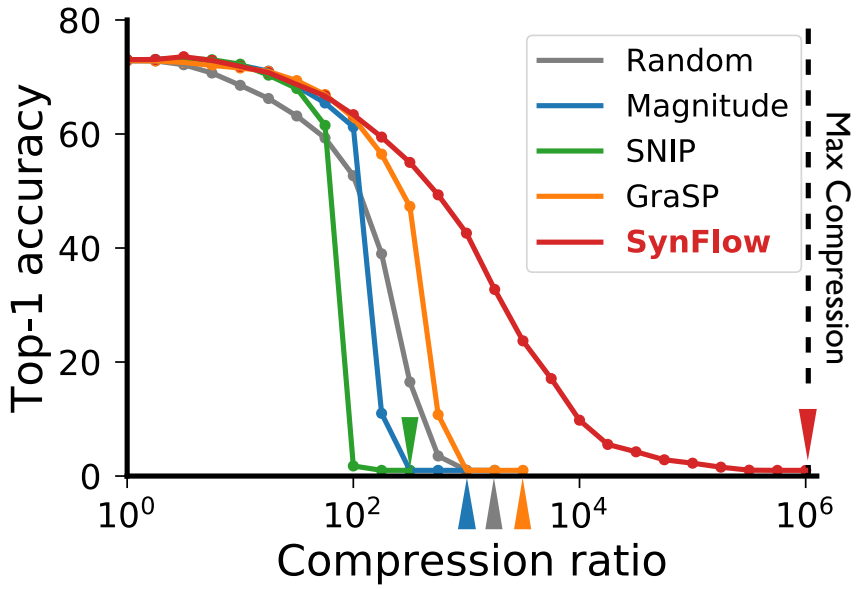


Figure 5.2: Synflow accuracy compared to SNIP and GraSP for different pruning rates. Methods are benchmarked on VGG16 trained on CIFAR-100. Illustration taken from [186]

These aforementioned methods allow pruning a network at initialisation but still require training the weights. Moreover, while these methods outperform the basic benchmark of random pruning, their accuracy is still below the one of post-training magnitude pruning [46]. In contrast to these works, our proposed solution in this chapter identifies effective subnetworks by training only their topology and without any weight tuning. Our solution yields sparse and lightweight subnetworks that achieve compelling performances and does not need further weight fine-tuning.

5.1.2 Lottery Tickets

As discussed in chapters 3 and 4, pruning methods, either structured or unstructured, are particularly successful at simplifying large neural networks, and seek to remove connections with the least perceptible impact on classification accuracy. Structured pruning consists in *jointly* removing groups of weights, entire channels or subnetworks [110, 124], whereas unstructured pruning aims at removing weights *individually* [59, 60].

Unstructured pruning has witnessed a recent surge in interest in the wake of the LTH [43]; an empirical study in [43] demonstrates that large pre-trained networks encompass lightweight subnetworks, referred to as *LTs*, which can achieve comparable performance to the original large networks in a similar number of epochs when trained in isolation with initial weights taken from the large network. To identify these LTs, the large network is trained until convergence, followed by pruning the smallest weights based on their magnitude. The remaining weights are then rewound to their original value, that is, the value they had before the training of the large network began. This resulting subnetwork is known as a *Lottery Ticket*. Frankle and Carbin also leveraged *iterative magnitude pruning* to identify LTs, where the pruning rate is gradually increased during training until it reaches the desired pruning rate [43].

Rewinding the weights to their original values does not allow to find LT for larger architectures, as noted by [123, 47]. Frankle and Carbin proposed a weaker version of the LTH where the weight values are not reset to their original values, but instead to an early stage of the training corresponding to the network reaching a stable state, described in [45]. Figure 5.3 provides conceptual illustrations of the different existing methods devised by Frankle and Carbin to find a LT.

Another study [123] pushes that finding further and concludes that only the topology of these subnetworks is actually important in order to reach compelling performances. Liu et al. [123] point out that the weights of the LTs are not important and can be randomly initialised, provided that the optimisation procedure is carefully designed: Liu et al. used a common SGD optimiser with momentum instead of using an Adam optimiser [100] with a low learning rate as Frankle and Carbin did in [43], suggesting that using an Adam optimiser might hinder the training of randomly initialised

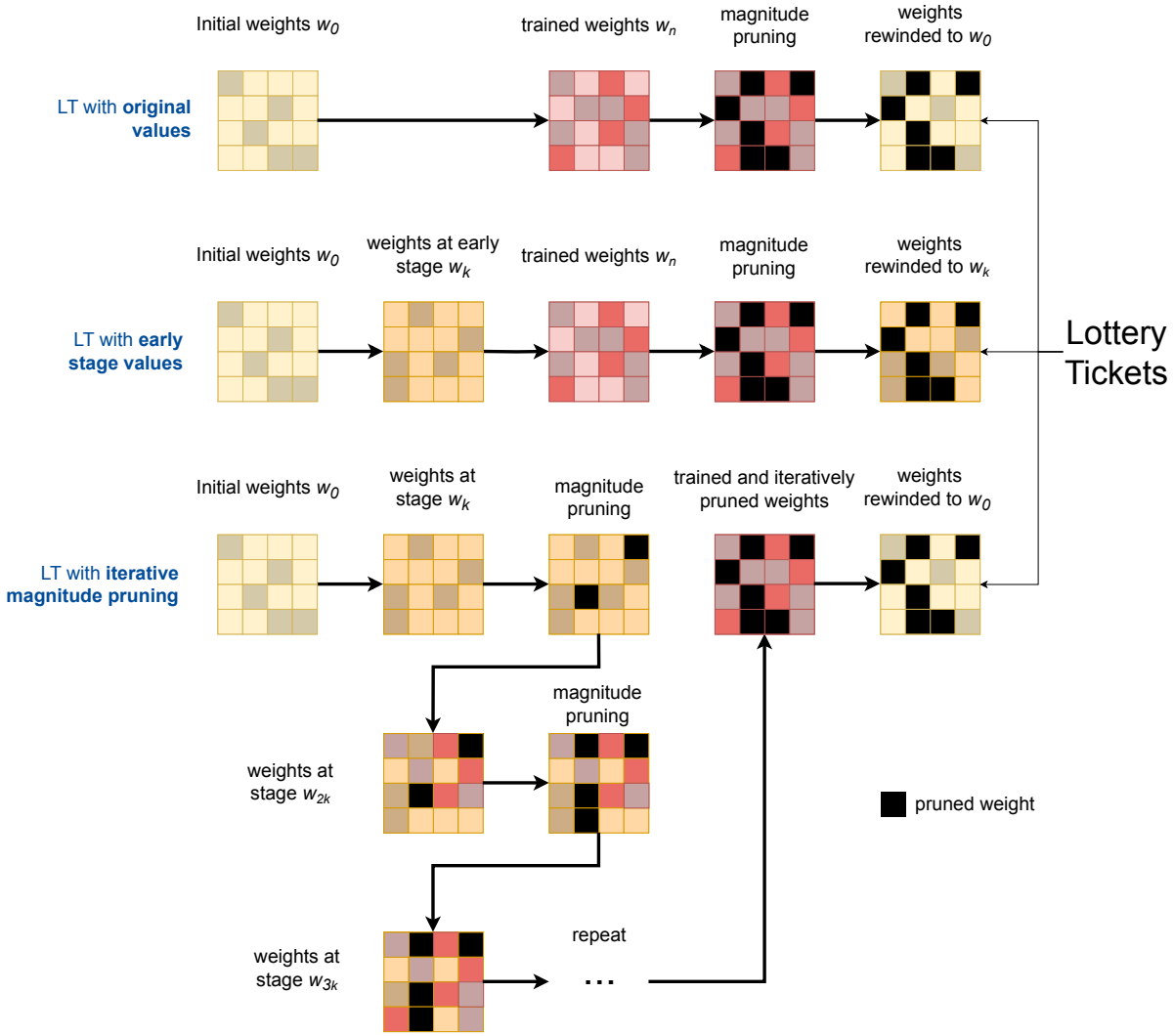


Figure 5.3: Conceptual illustration of the different processes to obtain a Lottery Ticket: reinitialising the weights to their original values with one-shot magnitude pruning (*LT with original values*), reinitialising the weights to their early stage values with one-shot magnitude pruning (*LT with early stage values*) and iterative magnitude pruning (*LT with iterative magnitude pruning*). Best viewed in colour.

LT.

The aforementioned works [43, 45, 123] focus on finding a Lottery Ticket that still needs to be trained in order to reach a satisfying level of performance. In contrast, our proposed method extracts a subnetwork that already achieves compelling performances without any weight training.

5.1.3 Existence of effective subnetworks

At first, it seems counterintuitive that there exists a subnetwork in a large network, that can achieve compelling performances without any weight training. This has been first conjectured in [157] as the Strong Lottery Ticket Hypothesis. A few theoretical analyses provide evidence that such subnetworks do exist. [132] demonstrate that a neural network of width d and depth l can be approximated by pruning a randomly initialised one that is a factor $O(d^4l^2)$ wider and twice as deep. The upper bound on the network width has later been improved by [145] to $O(d^2 \log(dl))$ under the assumption of a hyperbolic weight distribution. This upper bound has eventually been refined to $O(d \log(dl))$ by [148] for a broad class of weight distributions, including the uniform one which is widely used for weight initialisation [66].

5.1.4 Subnetwork topology extraction

Although the existence of effective subnetworks with untrained weights has been established, no constructive proof has been provided in order to identify them. In this context, several methods proposed heuristics to extract the lightweight and efficient subnetwork from a large untrained network [215, 157].

Supermark is a method introduced by Zhou et al. in [215] which is the first attempt to extract efficient subnetworks from a large untrained network using stochastic mask training. Each weight of the network is stochastically sampled following a Bernoulli distribution parametrised by a latent variable m. To that extent, weights are reparametrised as follows:

$$\hat{w} = w \times \text{Bern}(\sigma(m)) \tag{5.4}$$

where $\hat{w}$ is the *effective weight* (also referred to as the *apparent weight* in chapter 4) used in the network, w_i is the original frozen weight and σ the sigmoid function. At each iteration, a random variable is sampled from the Bernoulli distribution parametrised by m, which either selects or prunes the corresponding weight. The sampling being nondifferentiable, it is not possible to train directly m with SGD. Instead, the authors proposed to use the STE [9], a technique that approximates the gradient in the backward pass with a continuous surrogate function of the forward pass non-differentiable function. Zhou et al. also introduced a weight rescaling mechanism, called Dynamic Weight Rescaling (DWR), to mitigate the disruption of weight statistics due to pruning [66]. More details are given in sections 5.3.1 and 5.3.2.

During training, weights are frozen and only the masks are allowed to vary. However, the major drawback of this method resides in the vanishing gradient issue of the sigmoid which makes mask training numerically challenging. Ramanujan et al. [157] proposed another alternative, entitled Edge-popup, based on binarised saliency indicators learned with STE, which selects the most prominent weights in the resulting subnetworks. Each weight w_{ij}, corresponding to the connection between neurons i and j, is associated with a latent saliency indicator s_{ij}. During the forward pass, the weights associated with the top-k saliency indicators are selected and the others are pruned. Similarly to Supermask, binarised saliency indicators are not differentiable, therefore, the latter is made with STE. The authors consider the following expression for the weights in the backward pass:

$$\hat{w}_{ij} = s_{ij} w_{ij} \tag{5.5}$$

Edge-popup enforces the pruning rate *a priori*, thereby determining the value of k. This value is the same for all the layers, imposing a constant pruning rate throughout all the layers of the network, which is suboptimal. Indeed, the optimal pruning rate is layer-dependent and varies from one layer to another. Moreover, finding the pruning rate giving the highest performances has to be made through a cumbersome and time-consuming grid search. Like Supermask, Edge-popup also includes a weight rescaling

mechanism, based on a learnt rescaling factor that rescales the weight distribution in a layer-wise fashion, subsequently detailed in section 5.3.2.

5.2 Contributions

Considering the limitation of the aforementioned related work, namely, the challenges in mask training due to the sigmoid mask parametrisation, and the time-consuming nature of finding the optimal pruning rate, we introduce in this chapter a new stochastic subnetwork selection method based on Gumbel-Softmax (GS). The latter allows sampling subnetworks whose weights are the most relevant for classification. The proposed contribution also relies on a new mask parametrisation, entitled Arbitrarily Shifted Log Parametrisation (ASLP), that allows better conditioning of the gradient and thereby mitigates numerical instability during mask optimisation. Besides, when combining ASLP with a learned weight rescaling mechanism, training is accelerated and the accuracy of the resulting subnetworks improves as shown later in experiments. Our proposed pruning strategy is designed such that it does not necessitate any prior information regarding the optimal pruning rate that would yield the best performance. Instead, it automatically sets the optimal rate, eliminating the need for an exhaustive grid search.

The rest of this chapter is organized as follows: section 5.3 delves into our proposed method, named Arbitrarily Shifted Log Parametrisation (ASLP), for extracting efficient subnetworks using Gumbel-Softmax, including stochastic weight sampling and our weight rescaling approach. The overall workflow of our proposed method is detailed in section 5.4. In section 5.5, we share the results of our comprehensive experiments, including performance benchmarks, the impact of our weight rescaling approach, the effect of increasing the pruning rate after training and the impact of the learning rate on the training convergence speed and final performance. We conclude the chapter in section 5.6, summarizing our contributions and our key findings. As we will demonstrate, our new approach overcomes several of the challenges associated with previous techniques, offering a more efficient and effective way to extract high-performance subnetworks without weight training.

5.3 Extracting Effective Subnetworks with Gumbel-Softmax

Considering the same formalism as in chapter 4, let f_θ be a deep neural network whose weights are defined as $\theta = \{\boldsymbol{w}_1, \dots, \boldsymbol{w}_L\}$, with L being its depth, $\boldsymbol{w}_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ its ℓ^{th} layer weights, and d_ℓ the dimension of ℓ. The output of a given layer ℓ is defined as

$$\mathbf{z}_\ell = g_\ell(\boldsymbol{w}_\ell \otimes \mathbf{z}_{\ell-1}), \tag{5.6}$$

being g_ℓ an activation function and $\otimes$ the usual matrix product. Without loss of generality, we omit the bias in the definition of (5.6).

5.3.1 Stochastic Weight Sampling

Given a network f_θ, weight pruning consists in removing connections in the graph of f_θ. A node in this graph refers to a neural unit while an edge corresponds to a cross-layer connection. Pruning is usually obtained by freezing and zeroing out a subset of weights in θ, and this is achieved in practice by multiplying $\boldsymbol{w}_\ell$ by a binary mask $\boldsymbol{m}_\ell \in \{0,1\}^{\dim(\boldsymbol{w}_\ell)}$. The binary entries of $\boldsymbol{m}_\ell$ are set depending on whether the underlying layer connections are kept or removed, so equation (5.6) becomes

$$\mathbf{z}_\ell = g_\ell((\boldsymbol{m}_\ell \odot \boldsymbol{w}_\ell) \otimes \mathbf{z}_{\ell-1}). \tag{5.7}$$

Here $\odot$ stands for the element-wise matrix product. In chapter 4, the *effective pruning* step was achieved by setting the values of $\boldsymbol{m}_\ell$ to zero or one depending on the magnitude of the weight reparametrisation (see equation (4.14) and section 4.3). Consequently, the value of the masks $\boldsymbol{m}_\ell$ are only determined by the value of the weights and not the topology of the network. In this chapter, we propose another approach to obtain the masks $\boldsymbol{m}_\ell$ that is not bound to the value of the weights. In equation (5.7), the masks $\boldsymbol{m}_\ell$ are stochastic and sampled from a Bernoulli distribution. However, sampling is not a differentiable operation, therefore, optimising

directly $\boldsymbol{m}_\ell$ is not possible. To overcome this issue, while still relying on SGD, the Straight Through Estimator (STE) technique is applied together with a reparametrisation of the mask.

Straight Through Estimator. Zhou et al. [215] consider a Bernoulli parametrisation of $\boldsymbol{m}_\ell$ in order to sample masks in equation (5.7). Since sampling is not a differentiable operation, they rely on the STE. It is a technique developed in [9] that enables the training of neural networks with discrete activations, such as binary or quantised activations. The technique involves using a differentiable relaxation to the non-differentiable activation function during the backward pass, and using the non-differentiable function in the forward pass. This allows for the use of SGD to optimise the network, which was previously not possible with discrete activations. It is worth noting that STE is a heuristic that does not provide the correct gradient, but it is effective in practice [9].

In order to apply STE to the problem of Bernoulli stochastic mask sampling, the definition of $\boldsymbol{m}_\ell$ is based on another *latent* parametrisation $\hat{\boldsymbol{m}}_\ell$, detailed subsequently, and obtained by applying a sigmoid function $\sigma(.)$ to $\hat{\boldsymbol{m}}_\ell$. This allows optimizing $\hat{\boldsymbol{m}}_\ell$ using gradient descent by considering the following surrogate of equation (5.7) in the backward pass of the backpropagation algorithm:

$$\mathbf{z}_\ell = g_\ell((\sigma(\hat{\boldsymbol{m}}_\ell) \odot \boldsymbol{w}_\ell) \otimes \mathbf{z}_{\ell-1}). \tag{5.8}$$

As a result, although masks $\boldsymbol{m}_\ell$ are sampled and thus disconnected from the computation graph (sampling being not differentiable), their reparametrisation $\hat{\boldsymbol{m}}_\ell$ can be updated as if they were used in the computation graph as shown in equation (5.7).

Gumbel-Softmax. In what follows, we consider an alternative to STE based on Gumbel-Softmax (GS) [92] that demonstrates better performances for differentiable categorical sampling, which is the process of randomly selecting a category from a given set of categories, where each category has a specified probability of being chosen. Gumbel-Softmax is a technique that can be used to approximate a discrete categorical distribution with a continuous relaxation. Gumbel-Softmax works by using the Gumbel distri-

bution [55] to add noise to a categorical distribution and then applying the Softmax function to obtain a continuous relaxation of the discrete distribution. The proposed method, dubbed as Straight Through Gumbel-Softmax (STGS), is based on a variant of Gumbel-Softmax combined with Straight Through Estimator. In the forward pass, the softmax of GS is replaced by an argmax operator. Since this operator is not differentiable, the standard softmax is considered in the backward pass. The argmax operator allows sampling from a categorical distribution, as the limit of GS (*i.e.*, when its softmax temperature approaches zero).

Gumbel-Softmax applied to weight sampling. Let z be a categorical random variable, associated with n-class probability distribution $\mathcal{P} = [\pi_1, \ldots, \pi_n]$. In order to sample in a differentiable manner, the Gumbel-Softmax estimator takes as an input a vector of log-probabilities

$$\log(\mathcal{P}) = [\log(\pi_1), \ldots, \log(\pi_n)] \tag{5.9}$$

then it disrupts the latter with a random additive noise sampled from the Gumbel distribution, and finally takes its argmax, yielding a categorical variable. More formally, following [92], the value q of our categorical variable z is obtained as

$$q = \underset{k}{\operatorname{argmax}} \, [\log(\pi_k) + g_k], \tag{5.10}$$

with g_k being independent and identically distributed samples from the Gumbel distribution with zero mean and unit variance, denoted $\mathcal{G}(0, 1)$.

For mask sampling, only two possible outcomes are considered. Either the corresponding weight is selected and its mask is set to 1, or it is pruned from the sampled topology and its mask is set to 0. In what follows, and unless stated otherwise, we omit ℓ from $\boldsymbol{w}_\ell$ and we write it for short as $\boldsymbol{w}$. Let w_{ij} be the weight associated with the i-th and j-th neurons respectively belonging to two consecutive layers. Since there are two possible outcomes

for the masks, we define a two-class categorical distribution $\mathcal{P}_{ij}$ on $\{0, 1\}$ as

$$\begin{cases} \mathcal{P}_{ij}(z = 1) = \pi_1^{ij} \\ \mathcal{P}_{ij}(z = 0) = \pi_2^{ij} \end{cases} \tag{5.11}$$

with, again, $\pi_1^{ij} = p_{ij}$ and p_{ij} being the probability to keep the underlying connection. Since there are only two mutually exclusive outcomes, $\pi_2^{ij} = 1 - p_{ij}$. In other words, keeping the weight w_{ij} (or not) in the sampled topology is a Bernoulli trial with a probability p_{ij}. Considering equation (5.10), a binary mask m_{ij} is defined as

$$m_{ij} = 1_{\{q_{ij}=1\}} \tag{5.12}$$

$1_{\{\}}$ being the indicator function and following equation (5.10), q_{ij} is

$$q_{ij} = \text{argmax}_{k\in\{1,2\}}\big[\log(\pi_k^{ij}) + g_k^{ij}\big] \tag{5.13}$$

with $\pi_1^{ij} = p_{ij}$ and $\pi_2^{ij} = 1 - p_{ij}$, the probability for a weight to be selected or not in the sampled topology, respectively.

The proposed STGS algorithm enables the learning of probabilities p_{ij} for each weight w_{ij} through SGD. However, optimizing p_{ij} (with SGD) raises a major issue. Since the optimisation is not constrained, p_{ij} can take values larger than 1 or smaller than 0. As a consequence, it could no longer be interpreted as a probability, moreover, $\log(p_{ij})$ and $\log(1 - p_{ij})$ would also be undefined.

On another hand, solving constrained SGD, besides being computationally expensive and challenging, may result in a worse local minimum. In order to overcome all these issues, one may consider an alternative reparametrisation $p_{ij} = \sigma(\hat{m}_{ij})$, similar to the reparametrisation in [215], with $\hat{m}_{ij}$ being a latent mask variable and σ the sigmoid function which bounds

p_{ij} in $[0, 1]$. However, this workaround suffers in practice from numerical instability in gradient estimation and is also computationally demanding. Indeed, the combination of the logarithmic and the sigmoid functions leads to severe numerical instabilities, that necessitate a cumbersome stabilisation by adding ε to prevent p_{ij} and $(1 - p_{ij})$ from being to close to 0. The logarithmic function, which is applied to these quantities, is not defined on 0 and tends to $-\infty$ at its vicinity. Furthermore, it is important to note that the above formulation is computationally intensive since it requires the evaluation of log and exponential for every mask in the network.

Arbitrarily Shifted Log Parametrisation. In order to solve the issues related to STGS in the context of this chapter, in particular, the need for numerical stabilisation and computational complexity, another alternative is to consider the following expressions for $\log(\mathcal{P}_{ij})$:

$$\log(\mathcal{P}_{ij}) = \begin{bmatrix} \log(p_{ij}) = \hat{m}_{ij} \\ \log(1 - p_{ij}) = \log(1 - \exp(\hat{m}_{ij})) \end{bmatrix} \tag{5.14}$$

and learn the underlying mask $\hat{m}_{ij}$. However, this reparametrisation is also flawed in the same way as the aforementioned sigmoid reparametrisation, namely: numerically unstable and high computational cost, again due to the combination of logarithmic and exponential functions.

In what follows, we propose an equivalent formulation which turns out to be highly effective and numerically more stable. Instead of using the logarithmic probabilities outlined in equation (5.9) as the input for the STGS that would eventually lead to the formulation of equation (5.14), we adopt the ensuing expression at the weight level:

$$\begin{bmatrix} \hat{m}_{ij} \\ 0 \end{bmatrix} \tag{5.15}$$

Here, the second coefficient of the vector, normally representing $\log(1 - p_{ij})$, is set and fixed to 0. It is important to note that this formulation is

not the same as the one of equation (5.9). We interpret the formulation of equation (5.15) as:

$$\begin{bmatrix} \hat{m}_{ij} \\ 0 \end{bmatrix} = \log\big(\mathcal{P}_{ij}(.)\big) + c = \begin{bmatrix} \log(p_{ij}) + c \\ \log(1 - p_{ij}) + c \end{bmatrix}, \tag{5.16}$$

In the above expression, instead of using $\log(\mathcal{P}_{ij}(.))$ as an input for STGS, we interpret equation (5.9) as $\log(\mathcal{P}_{ij}(.)) + c$, which is the input of the argmax in equation (5.10).

The constant $c \in \mathbb{R}$ does not need to be known. Adding this constant ensures that even if $\hat{m}_{ij} > 0$, we can still interpret p_{ij} as a probability with $\log(p_{ij}) \in]-\infty, 0] \Leftrightarrow p_{ij} \in [0, 1]$. This is enforced by setting the second coefficient of equations (5.15) and (5.16) to zero, rather than computing it explicitly. Although different, the formulation of equation (5.16) is theoretically equivalent to the aforementioned sigmoid reparametrisation (see equations (5.4) and (5.8)). Indeed, solving the system of equation (5.16) with respect to $\hat{m}_{ij}$ yields $p_{ij} = \sigma(\hat{m}_{ij})$ (see proposition 5.3.1).

Proposition 5.3.1 (Formulation equivalence)**.** *The formulation in equation* (5.16) *is equivalent to defining* $p_{ij} = \sigma(\hat{m}_{ij})$ *with* σ *the sigmoid function, provided that* $p_{ij} \in]0, 1[$.

Proof. Consider the following system of equations:

$$\begin{cases} \hat{m}_{ij} = \log(p_{ij}) + c & (1) \\ 0 = \log(1 - p_{ij}) + c & (2) \end{cases}$$

Substracting (2) from (1) yields:

$$(1) - (2) \Leftrightarrow \hat{m}_{ij} = \log\left(\frac{p_{ij}}{1 - p_{ij}}\right)$$

$$\Leftrightarrow \frac{1}{p_{ij}} - 1 = \exp(-\hat{m}_{ij})$$

$$\Leftrightarrow p_{ij} = \frac{1}{\exp(-\hat{m}_{ij}) + 1}$$

$$\Leftrightarrow p_{ij} = \sigma(\hat{m}_{ij})$$

□

Differently put, the formulation in equation (5.16) considers a reparametrisation $\hat{m}_{ij} = \log(p_{ij}) + c$ and $\log(1 - p_{ij}) + c = 0$ which is strictly equivalent to the sigmoid one while being computationally more efficient and also numerically stable.

Proposition 5.3.1 assumes that $p_{ij} \in]0, 1[$, which, in practice, is verified. For the probabilities p_{ij} to reach 0 or 1, the masks $\hat{m}_{ij}$ would need to reach $\pm\infty$. This scenario cannot happen during training since $\hat{m}_{ij}$ are initialised to 0 (see section 5.5) and the sigmoid function applied to them makes the gradients of $\hat{m}_{ij}$ vanishingly small when $\hat{m}_{ij}$ deviate from 0. Because $\hat{m}_{ij}$ are updated following the SGD algorithm, vanishingly small gradients result in vanishingly small updates of $\hat{m}_{ij}$. In practice, it prevents the $\hat{m}_{ij}$ reaching $\pm\infty$ and thus $\sigma(\hat{m}_{ij}) = p_{ij}$ from reaching 0 or 1, which validates the assumption of proposition 5.3.1 that $p_{ij} \in]0, 1[$.

A crucial point to consider is that adding any arbitrary constant c to each coefficient of the log-probability vector does not change the outcome of Gumbel-Softmax sampling. This is because it does not alter the outcome of the argmax function, which remains unchanged regardless of the value of c, provided that the same value c is added to both coefficients, which is the case in our formulation (*c.f.* equation (5.16)). The presence of this constant c, whose value is arbitrary, that shifts the log-probabilities of our probability parametrisation gives the name of the method: Arbitrarily Shifted Log Parametrisation.

5.3.2 Smart Weight Rescaling

Subnetwork selection may disrupt the dynamic of the forward pass [66, 157], and thereby requires adapting weights accordingly. [66] establish that the variance of the initial weight distribution has a critical impact on the network performances. Since in the context of this chapter, the weights are not trained, it is all the more important to address this issue. The sampling of the weights alters the original weight distribution and therefore its statistics.

Dynamic Weight Rescaling (DWR) [215], along with the Scaled Kaiming distribution (SKD) [157], are two recognised strategies for adjusting the weights of chosen subnetworks to mitigate the aforementioned issue. Each approach has its limitations, which we address with our proposed weight rescaling method.

Dynamic Weight Rescaling. The DWR [215] method calculates the effective pruning rate at each training step on a layer-by-layer basis, referred to as the *observed* pruning rate. This is achieved by dividing the number of active weights in the layer (corresponding to a mask value of 1) by the total number of weights in the layer. Subsequently, all weights are rescaled by multiplying them with the inverse of the observed pruning rate. A drawback of DWR is that it demands the storage of sampled masks and the calculation of the observed pruning rate at each training step for every layer in the network. This makes the procedure computationally demanding and increases the memory requirements. Furthermore, the flexibility and adaptability of the method are limited due to the rescaling being tied to the pruning rate; there is no guarantee that the inverse of the observed pruning rate is the optimal factor to prevent changes to the weight distribution statistics. The performance boost attributable to DWR, as noted by Zhou et al., might be due to the increase of the standard deviation of the Xavier (or Glorot) initialisation [51] that the author used. Experimentally, a larger standard deviation improves the performance within the context of this chapter. Indeed, the Kaiming initialisation[1] [67], has a larger standard deviation than the Xavier initialisation and achieves superior results [157, 177].

[1]Kaiming and Glorot initialisation are detailed in appendix A.3

Scaled Kaiming distribution. The Kaiming initialisation [67] was presented by Ramanujan et al. in [157]. Similar to DWR, the weights are rescaled to safeguard the weight statistics from alteration. The rescaling factor here is the inverse of the square root of the pruning rate. Unlike DWR, the pruning rate in this method is enforced, not observed, making it less computationally intensive than DWR. However, it shares the same limitation as DWR in that the rescaling factor is directly tied to the pruning rate.

Smart Rescale. In what follows, we consider our proposed weight adaptation mechanism, referred to as Smart Rescale (SR). Instead of handcrafting this rescaling factor proportionally to the pruning rate (as achieved for instance in [215]), SR is learned layerwise and provides an effective (and also efficient) way to adapt the dynamic of the forward pass without retraining the entire weights of the selected subnetwork. These localised adjustments of distributions provide an advantage over the aforementioned methods that depend on a scaling factor that is reliant on the pruning rate and which is the same across all layers in the case of the Scaled Kaiming distribution. This flexibility ends up reducing the number of epochs needed to reach convergence and also improving accuracy (to some extent) as shown later in section 5.5.

Furthermore, SR improves accuracy as it adjusts the weights to maintain the statistics of the distribution of the original network weights, preserving the representative power of the network. Hence, rather than forcing the weights to follow an arbitrary distribution or scale, they are guided by the data-driven SR method which results in better performance and reduced training time. With SR, the ℓ-th layer network output becomes

$$\mathbf{z}_\ell = g_\ell(s_\ell \times (\boldsymbol{m}_\ell \odot \boldsymbol{w}_\ell) \otimes \mathbf{z}_{\ell-1}), \tag{5.17}$$

where s_ℓ refers to the rescaling factor of the ℓ-th layer (see also algorithm 3). Smart Rescale increases the flexibility of subnetwork selection and adaptation compared to DWR (which is again bound to the pruning rate). Moreover, scaling factors obtained with SR vary smoothly, consequently making the training more stable with SGD compared to the ones obtained with DWR which are again inversely proportional to the observed pruning rates,

and changes of the latter are more abrupt due to stochastic mask sampling.

5.3.3 Freezing the Topology via Thresholding

A network trained with ASLP has a stochastic topology that is sampled at each forward pass. To evaluate such a network, we chose to freeze its topology so that its outputs and thus performance are deterministic. This section presents a pruning strategy to evaluate a network trained with our method on a fixed topology. Once the *latent* masks $\hat{m}_{ij}$ are trained, a pruning step is applied to extract a subnetwork from the original heavy and unpruned network. This pruning fixes the values of the masks m_{ij} to either 0 or 1 (*c.f.* equation (5.7)), effectively freezing the network topology which was previously stochastic. This pruning step does not enforce a specific pruning rate, it is rather based on thresholding the weights probability of being selected p_{ij}. The resulting *observed* pruning rate is computed as the fraction of weights whose p_{ij} is below the said threshold, denoted τ. This pruning step can be defined by applying the function ξ_τ to each mask $\hat{m}_{ij}$, and assigning its result to m_{ij}, as shown in equation (5.18).

$$m_{ij} \leftarrow \xi_\tau(\hat{m}_{ij}) = \begin{cases} 1 & \text{if } \; p_{ij} \geq \tau \Leftrightarrow \hat{m}_{ij} \geq \sigma^{-1}(\tau) \\ 0 & \text{otherwise.} \end{cases} \tag{5.18}$$

where σ^{-1} is the inverse of the sigmoid function, also known as the logit function, whose expression is given in equation (5.19).

$$\sigma^{-1}(x) = \log\left(\frac{x}{1-x}\right) \tag{5.19}$$

Our pruning strategy seeks to retain the weights that have the highest probability of being present in the sampled topologies. Specifically, we set τ so that retained weights must be selected, on average, in at least half the sampled topologies. This implies that the weight selection probabilities p_{ij}, which are defined as $\sigma(\hat{m_{ij}})$, must be greater than or equal to $\tau = 0.5$, otherwise, the related weight w_{ij} is pruned. In other words, the weight

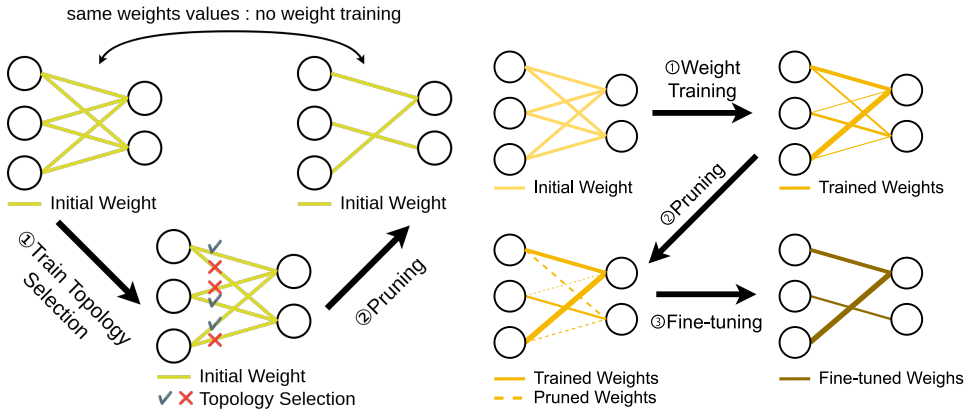
is kept if the binary event of keeping a connection is more likely than its removal. Since p_{ij} is defined as $\sigma(\hat{m}_{ij})$, in terms of latent masks, it means that a weight is kept if its associated latent mask $\hat{m}_{ij} \geq \sigma^{-1}(0.5) = 0$. We refer to this pruning method as *thresholding*.

5.4 Method Overview and Algorithm

Our method introduced a new perspective on neural network training. Rather than relying on the common training of the weights, our focus is on identifying the optimal topology by selecting a subset of the weights. This approach delivers an effective sparse subnetwork that demonstrates compelling performance compared to standard weight training, all without the need for weight training. Such a strategy is particularly beneficial in scenarios where a lightweight neural network is required due to limited computing resources or where weight training may not be feasible.

Proceeding with this innovative approach, our method integrates the Straight Through Gumbel-Softmax sampling technique and the Smart Rescale mechanism, resulting in a comprehensive method named Arbitrarily Shifted Log Parametrisation (ASLP). This strategy constructs lightweight neural networks by sampling topologies from a large, untrained network and learns the probability of selecting each weight. Probabilities are determined using Stochastic Gradient Descent in conjunction with a standard loss function, specifically cross-entropy loss for image classification tasks. The training procedure for our method is detailed in algorithm 3. Unless stated otherwise, this procedure is implemented in section 5.5.

The core differences between our approach and standard pruning pipelines are illustrated in figures 5.4a and 5.4b. As highlighted in figure 5.4a, our method focuses exclusively on topology selection without weight training, whereas conventional pruning pipelines rely on weight training and subsequent fine-tuning.



(a) Our pruning pipeline **(b)** Standard pruning pipelines

Figure 5.4: Overview of our pruning pipeline and standard pruning pipelines. Our pipeline performs topology selection only: weights are not trained. On the contrary, standard pruning pipelines rely on weight training and fine-tuning.

Algorithm 3 Our training procedure

Require: Dataset $\mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$, network f, weights θ, latent masks $\hat{\boldsymbol{m}}$, number of epochs n, learning rate η

for $t = 1$ to n **do**

for each $(X, y) \in \mathcal{D}$ **do**

Forward Pass:

$q_{ij} \leftarrow \text{argmax} \begin{bmatrix} \hat{m}_{ij} + g_{ij} \\ 0 + g'_{ij} \end{bmatrix}$ {Sample of a topology}

$m_{ij} \leftarrow 1_{\{q_{ij}=1\}}$ {Give the masks m_{ij} their values}

$\mathcal{L}\big(f_\theta(X, s_\ell(\boldsymbol{m}_\ell \odot \boldsymbol{w}_\ell)), y\big)$ {Compute the loss with masked weights and SR}

Backward pass: {In the backward pass, $\nabla_{\hat{\boldsymbol{m}}}\mathcal{L}$ is computed as q_{ij} is obtained through a softmax instead of an argmax}

$\hat{\boldsymbol{m}}_{t+1} = \hat{\boldsymbol{m}}_t - \eta \nabla_{\hat{\boldsymbol{m}}}\mathcal{L}$ {Backpropagate the loss and update the masks}

end for

end for

return Network f with unchanged weights θ and trained latent masks $\hat{\boldsymbol{m}}$.

5.5 Experiments

In this section, we evaluate the efficacy of our proposed method and we investigate the influence of various parameters and configurations. Section 5.5.1 details the experimental setups, section 5.5.2 presents the performances of our method against other state-of-the-art methods, namely Edge-popup [157] and Supermask [215], both detailed in section 5.1.4. Section 5.5.3 validates our weight-rescaling strategy, section 5.5.4 studies the impact of the learning rate on the performance of our method, and validates our choice of learning rate. Finally, section 5.5.5 investigates the impact of imposing a fixed pruning rate after the training and presents experimental results that support the effectiveness of our thresholding pruning strategy.

5.5.1 Experimental Setup

Our experiments were conducted on the CIFAR-10, CIFAR-100 and Tiny-ImageNet datasets which are described in section 2.5. Unless stated otherwise, on each table we report the test accuracy evaluated on the test set of the datasets. This accuracy is given in percentages with the standard deviation. The latter is given numerically in tables or represented by the shaded area around curves for figures. Each data point is obtained by averaging 5 independent runs. The architectures considered are Conv2, Conv4, Conv6, VGG16, ResNet-20 and ResNet-18, which are presented in section 2.4.3

In order to demonstrate the efficacy of our method in a standard image classification scenario, we compare our approach with state-of-the-art methods, specifically Edge-popup [157] and Supermask [215]. We re-implemented both methods in PyTorch [147] and employed a uniform training procedure for all methods: networks are trained for 1000 epochs with a fixed learning rate of 50 (except for Edge-popup, which utilises a learning rate of 0.1). The learning rate of SR is set to 10^{-3}, and neither weight decay nor ℓ_2 regularisation is applied. This section examines several configurations initially presented in [215], which encompass combinations of techniques or enhancements employed for method evaluation. The various techniques include the application of Weight Rescaling (WR), the use of Signed Constant (SC) weight distribution [215, 157] and data augmentation.

Weight Rescaling. Each method discussed in this section incorporates its own weight rescaling technique: Dynamic Weight Rescaling (DWR) for [215], Scaled Kaiming distribution (SKD) for [157] and Smart Rescale (SR) for ASLP (Ours). All of these techniques are denoted as WR in this section. The three of them have been detailed in section 5.3.2

Signed Constant Distribution. The signed constant distribution was introduced by [215]. Weights sampled from this distribution can take only two values: $-\sigma$ and σ, where σ represents the standard deviation of the weight tensor upon initialization using the widely adopted Kaiming initialization [66], which is tailored to initialise weights in such a way that the variance remains the same across every layer during both forward and backward passes, especially for neural networks with ReLU activation functions (More details are given in appendix A.3). Zhou et al. report that it improves performances over the standard weight initialisation scheme.

Data augmentation. Although Zhou et al. [215] did not used any data augmentation, it is a widely accepted practice in image classification and is generally applied even if not explicitly mentioned by the authors (for example Ramanujan et al. used data augmentation in their code [41] altough it is not mentioned in the original article [157]). Consequently, we consider two configurations: with and without data augmentation. The data augmentation we apply has been observed in various state-of-the-art implementations [41, 163, 109] and is the following: first, images are padded with zeroes, next, a random crop of the original size is extracted from the padded image. Lastly, a random horizontal flip is performed. An example of this data augmentation pipeline is displayed in figure 5.5.

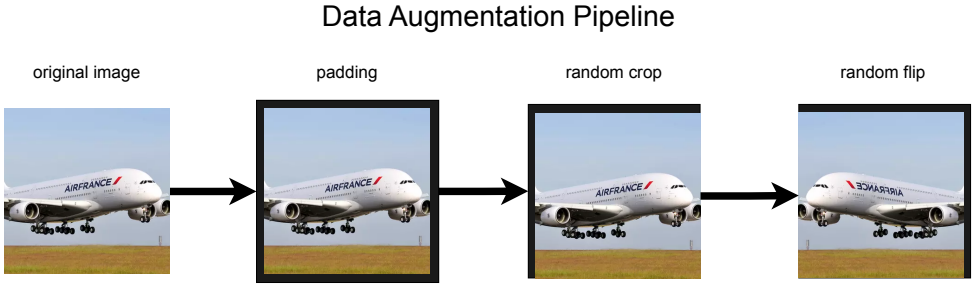


Figure 5.5: Data Augmentation pipeline example used for CIFAR-10 and CIFAR-100.

Pruning Strategy. To prune the networks trained with our method, we chose to freeze the topology with our *threhsolding* pruning strategy, described in section 5.3.3, which thresholds the probabilities of selection p_{ij} and consequently the latent masks $\hat{m}_{ij}$. As a matter of comparison, we also consider the setting in [215] which is an averaging strategy to evaluate Supermask performance. It consists in sampling ten different topologies, yielding effectively 10 different subnetworks. The performances of each of these subnetworks are evaluated independently giving 10 test accuracies that are averaged to obtain the final test accuracy. We refer to this pruning strategy as *averaging*. In the Edge-popup method [157], described in section 5.1.4, the pruning rate (denoted k by the authors) is inherently incorporated with two primary characteristics: *(i)* the set of pruned weights is deterministic (the pruned weights are the ones associated with the bottom-k saliency indicators), and *(ii)* the fraction of pruned weights is strictly equal to the predetermined pruning rate. As a result, a distinct pruning step enforcing the predetermined pruning rate is redundant since it would not bring any changes to the network structure or the values of the weights.

5.5.2 Performances

Overall, the analysis of our ASLP method results, presented in tables 5.2 to 5.5, shows that ASLP outperforms both Edge Popup and Supermask methods on CIFAR-10 and CIFAR-100 datasets, a trend consistent across all tested networks, including Conv2, Conv4, Conv6, VGG16, and ResNet20. Notably, our *thresholding* pruning strategy demonstrated superior performance compared to Supermask *averaging* approach (see section 5.3.3 for details of both strategies). The *thresholding* strategy ensures that the most likely-to-be-selected weights are incorporated into the final frozen network. It is accomplished by setting a threshold on p_{ij}, the probability for a weight of being selected. Only those weights exceeding this threshold are preserved, leading to the retention of the most valuable connections in the network. This approach focuses on the utilisation of high-impact weights, which contribute to the enhanced performance of the pruned network. Contrastingly, the *averaging* strategy employed by the Supermask method [215] takes a different approach. It draws 10 random subnetworks, each possessing a distinct set of weights. However, this strategy introduces a risk, as these randomly selected weight sets may contain weights that

contribute minimally to the overall performance of the network. The inclusion of such potentially useless weights could reduce the performance of the pruned network.

Remarkably, for larger networks such as VGG16 and ResNet-20, our ASLP method employing the *thresholding* strategy also consistently surpasses all other methods (see table 5.4). Nevertheless, a ResNet-18 trained on TinyImagenet with ASLP is outperformed by its counterpart trained with Edge-popup (see table 5.5).

We put forth several hypotheses to account for the reduced performance. First, the ResNet-18 architecture employed is the standard PyTorch implementation [153], designed for the ImageNet dataset [25]. As a result, it is tailored for 224×224 pixel images, while TinyImageNet images are only 64×64 pixels. With a smaller input image size, each pixel encompasses a larger area of the input space, and an overly large receptive field for a smaller image like TinyImageNet may lead to the loss of crucial spatial information. We opted against upscaling TinyImageNet images to 224×224 as a preprocessing step in order to prevent an exponential increase in computational cost.

Secondly, the ResNet-18 network is among the largest networks we examined, having roughly 3 times the number of parameters of a Conv4 network (refer to table 2.1). As a result, there are numerous possible weight combinations and subnetworks. Viewing ASLP as a Neural Architecture Search method, the search space for ResNet-18 is considerably larger than for the Conv{2,4,6} networks. Nevertheless, the number of sampled topologies is only equal to the product of the number of batches and the number of epochs during which the network is trained. Table 5.1 presents the details of 2 setups: ResNet-18 trained on TinyImageNet and Conv4 trained on CIFAR-10. It shows that the fraction of explored topologies by our ASLP method is considerably higher for the Conv4 network than for the ResNet-18 one in the given configurations. The number of explored topologies might be sufficient to sweep the search space for Conv{2,4,6} networks and find a compelling and effective subnetwork, but it might not be enough for a ResNet-18 network.

However, this hypothesis warrants further clarification. Indeed, the VGG16 network has more parameters than the ResNet-18 (see table 2.1),

	Conv4 & CIFAR-10	**ResNet-18 & TinyImageNet**
Number of parameters (N)	2,425,930	11,685,608
Train Dataset Size (number or images)	50,000	90,000
Batch Size used for training	256	256
Nb. of batches for 1 epoch	196	352
Nb. of explored topologies in 10^3 epochs (E)	196,000	352,000
Nb. of possible topologies (P)	$P = 2^N \approx 5 \times 10^{10^{5.86}}$	$P = 2^N \approx 5 \times 10^{10^{6.54}}$
Fraction of explored topologies (ν_{exp})	$\nu_{\text{exp}}^{\text{Conv4}} = \frac{E}{P} \approx 10^{-10^{5.86}}$	$\nu_{\text{exp}}^{\text{ResNet-18}} = \frac{E}{P} \approx 10^{-10^{6.54}}$
Ratio of fractions of explored topologies	$\frac{\nu_{\text{exp}}^{\text{Conv4}}}{\nu_{\text{exp}}^{\text{ResNet-18}}} \approx 10^{10^{6.44}}$	

Table 5.1: Comparison of the number of explored topologies for the Conv4 and ResNet-18 networks with CIFAR-10 and TinyImageNet, respectively. Since a new topology is sampled for every batch, the number of explored topologies (E) is computed as the product of the number of batches and the number of epochs during which the network is trained (here 10^3). The number of possible topologies (P) is computed as the number of possible weight combinations in the network (2^N). The fraction of explored topologies is computed as the ratio of the fraction of explored topologies for the Conv4 network and the fraction of explored topologies for the ResNet-18 network. In these experimental setups, the fraction of explored topologies for the Conv4 network is significantly higher than the fraction of explored topologies for the ResNet-18 network.

nevertheless, in our experiments, ASLP achieves superior results compared to other methods on the VGG16 architecture. We suggest the following explanation: First, the CIFAR-10 and CIFAR-100 datasets are simpler datasets with fewer classes, compared to TinyImageNet. Secondly, although the VGG16 has more parameters, it has fewer weights in its fully connected layers since the datasets it is benchmarked on have fewer classes. Indeed, a VGG16 network tailored for CIFAR-100 has 51,200 parameters in its last fully connected layer, whereas a ResNet-18 network designed for TinyImageNet has 102,400. Thus, the search space for the fully connected part of the VGG network is significantly smaller than on the ResNet-18 network. Therefore, because of the smaller search space, ASLP finds a more optimal subset of weights for the last layer of the VGG16 network than the ResNet-18 one. This final fully connected layer of a neural network plays a crucial role in determining its overall performance and accuracy in predicting outcomes.

		∅	SC	WR	WR+SC
Conv2	ASLP (thresholding)	**75.70 ± 0.30**	**75.81 ± 0.69**	**76.48 ± 0.68**	**76.92 ± 0.24**
	ASLP (averaging)	75.42 ± 0.25	75.50 ± 0.56	76.05 ± 0.44	76.44 ± 0.19
	[215] (averaging)	-	-	-	-
	[157] ($k = 50\%$)	74.18 ± 0.76	75.19 ± 0.56	74.51 ± 0.31	75.45 ± 0.44
Conv4	ASLP (thresholding)	**83.03 ± 0.31**	**83.73 ± 0.46**	**83.59 ± 0.29**	**84.06 ± 0.31**
	ASLP (averaging)	82.29 ± 0.25	83.22 ± 0.56	82.79 ± 0.30	83.46 ± 0.49
	[215] (averaging)	-	-	-	-
	[157] ($k = 50\%$)	82.38 ± 0.29	83.61 ± 0.38	81.71 ± 0.59	83.55 ± 0.32
Conv6	ASLP (thresholding)	**84.98 ± 0.33**	**86.49 ± 0.36**	**85.32 ± 0.27**	**86.21 ± 0.34**
	ASLP (averaging)	84.24 ± 0.28	85.67 ± 0.34	84.51 ± 0.35	85.49 ± 0.38
	[215] (averaging)	-	-	-	-
	[157] ($k = 50\%$)	84.67 ± 0.35	85.87 ± 0.13	84.37 ± 0.58	85.84 ± 0.51

(a) With data augmentation.

		∅	SC	WR	WR+SC
Conv2	ASLP (thresholding)	**68.24 ± 0.14**	**68.11 ± 0.64**	**66.84 ± 0.46**	**66.05 ± 0.93**
	ASLP (averaging)	68.09 ± 0.35	67.69 ± 0.52	65.79 ± 0.65	65.35 ± 0.83
	[215] (averaging)	67.12 ± 0.25	66.34 ± 0.41	56.71 ± 2.99	56.26 ± 1.64
	[157] ($k = 50\%$)	-	-	-	-
Conv4	ASLP (thresholding)	**71.64 ± 0.36**	**69.74 ± 1.37**	**72.85 ± 0.48**	**72.08 ± 0.62**
	ASLP (averaging)	70.88 ± 0.47	68.77 ± 1.42	71.82 ± 0.53	71.09 ± 0.69
	[215] (averaging)	68.09 ± 0.84	67.48 ± 0.52	58.13 ± 2.39	53.84 ± 5.00
	[157] ($k = 50\%$)	-	-	-	-
Conv6	ASLP (thresholding)	**73.32 ± 0.42**	**69.83 ± 1.46**	**76.20 ± 0.91**	**75.30 ± 0.89**
	ASLP (averaging)	72.62 ± 0.57	69.53 ± 1.68	75.24 ± 0.69	74.50 ± 0.96
	[215] (averaging)	70.71 ± 0.98	69.16 ± 1.92	44.77 ± 17.02	36.59 ± 15.32
	[157] ($k = 50\%$)	-	-	-	-

(b) Without data augmentation.

Table 5.2: Comparison of ASLP **test accuracy** against Edge-Popup and Supermask [157, 215] on **CIFAR-10** using various configurations. We reimplemented the configurations tested by the authors in their articles. Performances are presented with (table 5.2a) and without (table 5.2b) data augmentation, Weight Rescaling (WR), and Signed Constant (SC) weight distribution. A dash denotes a configuration that was not tested by the authors. Our method performances are reported for both the *thresholding* and *averaging* setups detailed in section 5.3.3. For Edge-popup, we use the value of k which yeilds the best test accuracy for Conv{2,4,6}, as reported in [157]. Across all setups, our method ASLP outperforms Edge-Popup and Supermask.

		∅	**SC**	**WR**	**WR+SC**
Conv2	ASLP (thresholding)	**38.64 ± 0.92**	38.31 ± 0.75	**41.81 ± 0.84**	**42.06 ± 0.76**
	ASLP (averaging)	38.49 ± 0.61	38.18 ± 0.81	41.12 ± 0.66	41.17 ± 0.54
	[215] (averaging)	-	-	-	-
	[157] ($k = 50\%$)	38.47 ± 0.46	**39.83 ± 0.46**	38.57 ± 0.59	39.87 ± 0.78
Conv4	ASLP (thresholding)	**47.78 ± 1.18**	49.33 ± 0.77	**50.33 ± 0.39**	**51.49 ± 0.43**
	ASLP (averaging)	47.18 ± 1.17	48.78 ± 0.79	49.39 ± 0.30	50.17 ± 0.50
	[215] (averaging)	-	-	-	-
	[157] ($k = 50\%$)	47.75 ± 0.63	**50.16 ± 0.47**	48.20 ± 0.72	50.02 ± 0.65
Conv6	ASLP (thresholding)	51.09 ± 0.92	53.00 ± 0.52	**51.70 ± 0.48**	52.85 ± 0.50
	ASLP (averaging)	50.22 ± 1.09	51.72 ± 0.73	50.56 ± 0.33	51.59 ± 0.24
	[215] (averaging)	-	-	-	-
	[157] ($k = 50\%$)	**51.13 ± 0.39**	**53.48 ± 0.51**	51.06 ± 1.11	**54.01 ± 0.35**

(a) With data augmentation

		∅	**SC**	**WR**	**WR+SC**
Conv2	ASLP (thresholding)	**38.72 ± 0.59**	38.64 ± 1.23	**42.42 ± 0.30**	**41.95 ± 0.68**
	ASLP (averaging)	38.40 ± 0.81	**38.71 ± 1.05**	41.66 ± 0.39	41.42 ± 0.55
	[215] (averaging)	38.09 ± 1.03	37.28 ± 0.47	26.03 ± 2.23	23.49 ± 1.36
	[157] ($k = 50\%$)	-	-	-	-
Conv4	ASLP (thresholding)	**47.56 ± 0.36**	**49.30 ± 0.54**	**50.39 ± 0.58**	**51.16 ± 0.94**
	ASLP (averaging)	46.89 ± 0.52	48.74 ± 0.47	49.55 ± 0.57	50.23 ± 0.87
	[215] (averaging)	45.84 ± 1.01	47.72 ± 0.75	27.70 ± 2.41	27.53 ± 5.20
	[157] ($k = 50\%$)	-	-	-	-
Conv6	ASLP (thresholding)	**51.43 ± 0.41**	**53.10 ± 0.27**	**51.52 ± 0.35**	**53.22 ± 0.54**
	ASLP (averaging)	50.47 ± 0.42	52.00 ± 0.27	50.38 ± 0.33	51.82 ± 0.34
	[215] (averaging)	49.19 ± 0.75	50.66 ± 0.47	2.54 ± 1.63	9.21 ± 5.50
	[157] ($k = 50\%$)	-	-	-	-

(b) Without data augmentation

Table 5.3: Comparison of ASLP **test accuracy** against Edge-Popup and Supermask [157, 215] on **CIFAR-100** using various configurations. We use the configurations tested by the authors in their articles. Performances are presented with (table 5.2a) and without (table 5.2b) data augmentation, Weight Rescaling (WR), and Signed Constant (SC) weight distribution. A dash denotes a configuration that was not tested by the authors. Our method performances are reported for both the *thresholding* and *averaging* setups detailed in section 5.3.3. For Edge-popup, we use the value of k which yeilds the best test accuracy for Conv{2,4,6}, as reported in [157]. For smaller networks, ASLP outperforms the other methods, with the exception of the SC setup for Conv2 and Conv4. However, for Conv6, ASLP performance is superior when data augmentation is disabled, while Edge-popup achieves better results with data augmentation enabled (except for the WR setup).

		Dataset	
		CIFAR-10	**CIFAR-100**
ResNet-20	ASLP (thresholding)	**81.08 ± 0.50**	**44.63 ± 0.91**
	ASLP (averaging)	78.85 ± 0.41	42.91 ± 1.14
	[215] (averaging)	69.83 ± 1.20	30.60 ± 0.91
	[157] ($k = 50\%$)	75.09 ± 1.41	22.47 ± 1.37
VGG16	ASLP (thresholding)	**24.93 ± 0.69**	**8.66 ± 0.33**
	ASLP (averaging)	24.93 ± 0.77	8.58 ± 0.32
	[215] (averaging)	25.07 ± 0.34	7.97 ± 0.35
	[157] ($k = 50\%$)	23.05 ± 0.84	6.65 ± 0.38

Table 5.4: Comparison of ASLP **test accuracy** against Edge-Popup and Supermask [157, 215] on both **CIFAR-10** and **CIFAR-100** datasets using VGG16 and ResNet-20 architectures. The results showcase the scenario with data augmentation, Weight Rescaling (WR) and Signed Constant (SC) weight distribution. Across all datasets and network architectures, ASLP surpasses the comparative methods in its *thresholding* configuration, detailed in section 5.3.3.

		TinyImageNet
ResNet-18	ASLP (thresholding)	33.56 ± 1.18
	ASLP (averaging)	34.16 ± 0.26
	[215] (averaging)	34.83 ± 0.46
	[157] ($k = 50\%$)	**38.00 ± 0.26**

Table 5.5: Comparison of ASLP **test accuracy** against Edge-Popup and Supermask [157, 215] on **TinyImageNet** datasets using ResNet-18 architecture. The results showcase the scenario with data augmentation, Weight Rescaling (WR) and Signed Constant (SC) weight distribution. The *thresholding* and *averaging* configurations are detailed in section 5.3.3. Edge-popup [157] performs the best in this scenario.

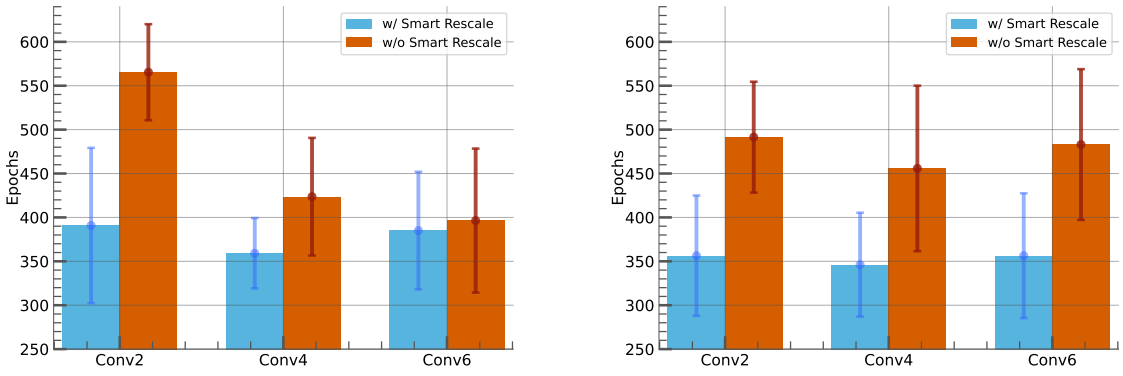
	Pruning Rate	
	CIFAR-10	**CIFAR-100**
Conv2	51.80 ± 0.14	51.90 ± 0.16
Conv4	51.78 ± 0.46	52.83 ± 0.40
Conv6	51.28 ± 0.40	52.15 ± 1.35
ResNet-20	51.63 ± 0.09	52.73 ± 0.28
VGG16	60.81 ± 1.56	60.89 ± 1.04
	TinyImageNet	
ResNet-18	52.73 ± 0.28	

Table 5.6: Comparison of observed **pruning rates** of the ASLP method across various neural network architectures and datasets (CIFAR-10 and CIFAR-100) after applying the *thresholding* procedure, detailed in section 5.3.3. The results are presented as mean percentages of pruned weights with their respective standard deviations, for the setup with data augmentation, Weight Rescaling (WR) and Signed Constant (SC) weight distribution.

5.5.3 Validation of the Weight Rescaling Mechanism

In tables 5.2 and 5.3, we observe that our weight rescaling technique, entitled Smart Rescale (SR), positively impacts performance, as corroborated by the increase in test accuracy compared to the baseline (referred to as ∅). This improvement is consistent across CIFAR-10 and CIFAR-100 datasets, with and without data augmentation. Besides enhancing accuracy, SR also contributes to a reduction in the number of epochs necessary for convergence. This observation is supported by figure 5.6, which shows a significant decrease in the number of epochs prior to convergence for all tested architectures and datasets. Networks used in figure 5.6 have been trained with and without SR using data augmentation in both cases. The total number of epochs is set to 1,000 and an early stopping policy, described in section 5.5, is applied. The point of convergence is defined as the moment when the early stopping policy halts the training process. Again, the training process is stopped if there is no improvement in the validation accuracy over the last 60 epochs. The reason for the enhanced performance and reduced training time when using SR can be attributed to its flexibility. Unlike DWR and SC, which impose a pruning-bound scaling factor, SR provides a more flexible approach, permitting an adaptation of weight distributions for each layer individually. The layer-wise adaptation allows for limiting the exhaustive search of topology (and thereby reducing the training time) by enabling a slight adjustment of the weight distribution.

Besides improving performances and reducing the number of epochs prior to convergence, SR is also an efficient alternative to Dynamic Weight Rescaling (DWR) [215]. DWR requires rectifying weights layerwise using the inverse of the observed pruning rates. In order to find the observed pruning rate for each layer, it is necessary to store the sampled masks and compute their active fraction. These layerwise evaluations introduce a substantial overhead at each training epoch. On the other hand, SR involves straightforward scalar multiplications for each layer, resulting in reduced complexity. In our experiments, enabling DWR increases the epoch runtime by 0.2 seconds for Conv4 network, while our SR method increases it by 0.13 seconds only. This corresponds to a 35% reduction in training overhead when using SR compared to DWR on a Conv4 network.



(a) CIFAR-10 **(b)** CIFAR-100

Figure 5.6: Impact of Smart Rescale (SR) on the number of epochs required to reach convergence for Conv{2,4,6} on CIFAR-10 and CIFAR-100.

5.5.4 Effect of the Learning Rate on Training Performances

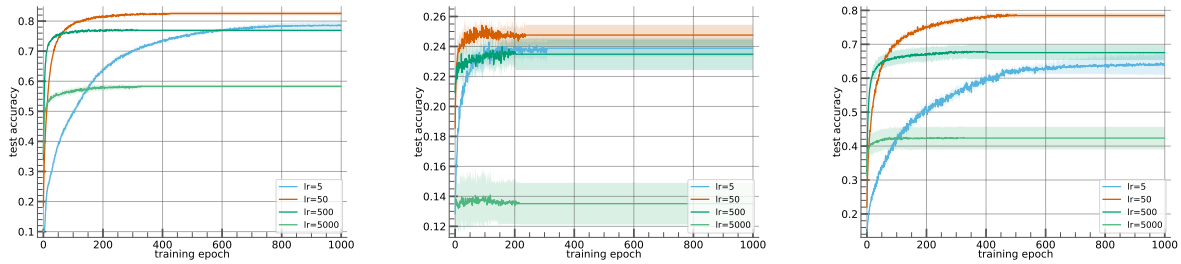
The learning rate is an essential hyperparameter for training neural networks as it controls the magnitude of the network parameter updates. When training networks with our ASLP method introduced in this chapter, we set the learning rate to 50. This value is significantly higher than the learning rates typically used for training neural networks. For instance, the learning rates to train various baseline models reported in [22], are several orders of magnitude lower than 50. An excessively high learning rate typically results in a diverging loss function resulting in a network failing

to learn a data representation. However, in the case of ASLP, increasing the learning rate up to arbitrarily high values does not cause the loss function to diverge. This section investigates the impact of the learning rate on the final performances of the network and its convergence speed. It also provides elements to justify the choice of a learning rate of 50.

Figure 5.7 shows the evolution of the test accuracy for Conv4, VGG16 and ResNet-20 on CIFAR-10, when trained with different learning rates. The solid line represents the average of five independent runs and the shaded area of the corresponding colour indicates the standard deviation. For ease of visualisation and comparison, the curves of test accuracies have been padded with their last value in order to make them all 1000 epochs long. Networks have been trained with data augmentation, WR and SC. Results from figure 5.7 indicate that a high learning rate makes the loss function and the accuracy converge to their final values more quickly. Networks trained with ASLP and a high learning rate (500 or 5000) exhibit better performance than the ones trained with a lower learning rate (5 or 50) when considering only the first 50 epochs. Nevertheless, the former are eventually outperformed by the latter if the training is run for more epochs. Conversely, an excessively low learning rate may not allow networks to reach satisfying performance levels in a reasonable amount of time. Our experimental findings indicate that using a scheduling policy on the learning rate does not improve performance. In other words, opting for a high learning rate and subsequently decreasing it yields worse results than maintaining a lower, constant learning rate. We found that a learning rate of 50 strikes the optimal balance between performance and training speed. Interestingly, this learning rate remains consistent across all architectures and datasets.

In standard training, high learning rate values cause the optimisation to fail because of parameters becoming too large and resulting in `NaN` values. However, this is not the case for ASLP which exhibits robustness to high learning rates. This robustness can be attributed to the fact that, in ASLP, the trained variables are the latent masks, denoted by $\hat{\boldsymbol{m}}$, which can be interpreted as probabilities of selection by applying a sigmoid function (refer to section 5.3.1 and proposition 5.3.1). The sigmoid function ensures that: *(i)* the latent masks cannot take extreme values leading to `NaN` because of increasingly small gradients as the latent masks move away from the origin (see section 5.3.1), and *(ii)* the sigmoid output is bounded

between 0 and 1 resulting in probabilities of selection close to 0 or 1, but not altering in a dramatical way the output of the network that otherwise might also lead to `NaN` values. Besides, due to the vanishingly small gradients as the masks move further away from the origin, it is practically impossible to reactivate those weights, let alone reactivate them with a smaller learning rate. This accounts for the observation that reducing the learning rate after using a high learning rate does not enhance performance.



(a) Conv4 **(b)** VGG16 **(c)** ResNet-20

Figure 5.7: Evolution of the test accuracy for Conv4, VGG16 and ResNet-20 trained with ASLP (with data augmentation, WR and SC) on CIFAR-10 for various learning rates. A learning rate of 50 yields the optimal balance between performance and training speed.

5.5.5 Post Training Pruning Rate Adjustment

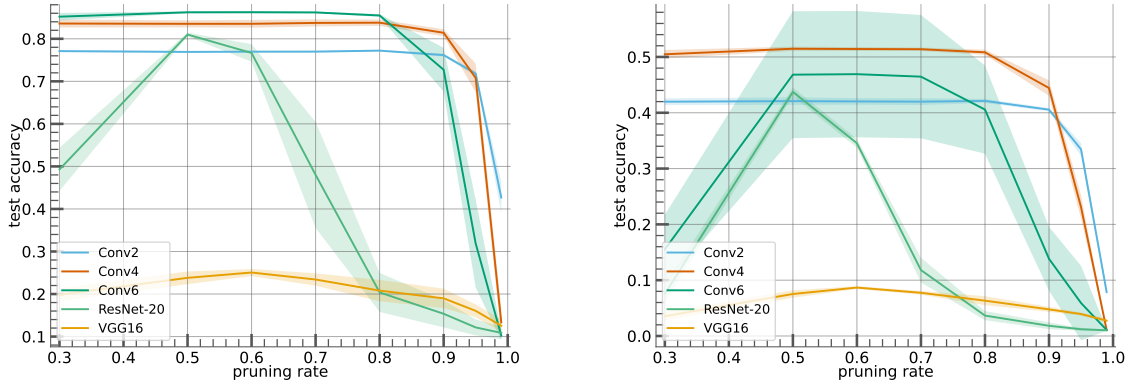
This section investigates the impact of pruning a network trained with ASLP to a given pruning rate instead of using the *thresholding* strategy, described in section 5.3.3. Unlike Edge-popup [157] where the pruning rate is a hyperparameter of the method, the proposed ASLP approach does not enforce a predefined pruning rate during training. Instead, it determines an optimal subset of weights that enables the network to minimises the loss function by updating weights probabilities of selection p_{ij} through back-propagation [169]. Rather than being enforced, the pruning rate is observed and is determined by thresholding the p_{ij} following equation (5.18), as explained in section 5.3.3. The observed pruning rate, which is the fraction of weights whose probabilities of selection p_{ij} are smaller than the threshold τ in equation (5.18), lies just above 50% for the tested architectures (60% for VGG16), as reported in table 5.6. However, instead of thresholding the p_{ij} and observing the pruning rate, it can be adjusted by pruning the weights on the magnitude of their associated latent masks $\hat{\boldsymbol{m}}$, which enforces the pruning rate *a posteriori* by considering the probabilities of selection p_{ij}

as saliency scores. This is equivalent to changing the value of the threshold τ in equation (5.18) in order to match a given pruning rate. Notably, Conv{2,4,6} networks trained with ASLP, and pruned a posteriori with a given pruning rate, achieve compelling performances on CIFAR-10 and CIFAR-100 datasets for pruning rates up to 85%, whereas their observed pruning rate is approximately 50% (see figure 5.8 and table 5.6). In the figure 5.8, the solid line represents the average test accuracy of five independent runs and the shaded area represents the standard deviation.

Moreover, the results presented in figure 5.8 provide further support for the *thresholding* strategy. This figure displays the test accuracy of networks trained with ASLP and pruned a posteriori, as described in the above paragraph, for various pruning rates. Sweeping through the pruning rate enables us to determine the optimal pruning rate for each network and dataset combination. The optimal pruning rate is defined as the pruning rate that yields the highest test accuracy. The results presented in the figure suggest that the optimal pruning rate for Conv{2,4,6} and ResNet-20 networks lies at 50%, and at 60% for VGG16. These rates are precisely the observed pruning rates obtained using the ASLP method with the *thresholding* pruning strategy (see table 5.6). In other words, the ASLP method together with *thresholding* pruning automatically determines the optimal pruning rate for an architecture in one shot without the need for a costly grid search. This is a significant advantage over the Edge-popup method [157] which requires a full training for each tested pruning rate.

5.6 Conclusion

In this chapter, we introduced the Arbitrarily Shifted Log Parametrisation method, which focuses on selecting efficient subnetworks from large, untrained neural networks through stochastic pruning, without training the weights. Arbitrarily Shifted Log Parametrisation is a stochastic subnetwork selection method that uses Gumbel-Softmax sampling and a new mask parametrisation to optimize the subnetwork topology while mitigating numerical instabilities. Additionally, we presented the Smart Rescale technique to accelerate training and improve the accuracy of the resulting subnetworks. Our experimental results show that ASLP outperforms closely related state-of-the-art methods, such as Edge-popup [157] and Supermask [215], on the CIFAR-10 and CIFAR-100 datasets across various



(a) cifar10 **(b)** cifar100

Figure 5.8: Comparative analysis of ASLP performance for CIFAR-10 and CIFAR-100 datasets using various network architectures (Conv{2,4,6}, ResNet-20, and VGG16) at different pruning rates. ASLP performances are evaluated with WR, SC and data augmentation. Results demonstrate that Conv{2,4,6} networks maintain strong performance even at higher pruning rates and indicate that the pruning rate achieved by thresholding is equivalent to the pruning rate yielding the best test accuracy when sweeping through the possible pruning rates.

network architectures. Our proposed *thresholding* pruning strategy consistently yields better performance than Supermask *averaging* approach, while finding the optimal pruning rate without the need for costly grid-search, contrary to Edge-popup. Furthermore, our Smart Rescale method leads to faster convergence and improved accuracy with a lower overhead compared to other weight-rescaling techniques, such as Dynamic Weight Rescaling. Arbitrarily Shifted Log Parametrisation also exhibits robustness to substantially high learning rates, ensuring stable performance across different network architectures and datasets.

All in all, the Arbitrarily Shifted Log Parametrisation method provides a promising solution for selecting efficient subnetworks from large untrained neural networks through stochastic pruning, offering improved performance and faster convergence and a new perspective on neural network training which focuses on topology selection rather than weight training.

Chapter 6

Conclusion and Perspectives

Contents

6.1 Summary of contributions

In this thesis, we addressed the issue of Deep Neural Networks compression, specifically from the perspective of pruning, and in particular, we focused on the problem of performance drop after pruning. We proposed several solutions to address this issue and ultimately questioned the very necessity of training the weights. We summarised our contributions in the following paragraphs.

Budget-aware pruning with weight reparametrisation. Pruning a network post-training introduces a performance drop that needs to be compensated for with fine-tuning. In chapter 4 we propose a budget-aware pruning method based on a weight reparametrisation. Respecting a budget throughout training allows for joint optimisation of the weights and the topology. Moreover, by controlling the number of parameters that will remain, it encourages the network not to use more capacity and therefore weights than what will be allowed once pruning is enforced. To reach this goal, we introduce in chapter 4 two main components that work together. On the one hand, a budget regularisation loss that computes the current weight budget at each training step, guiding the optimisation process to adhere to it. On the other hand, a weight reparametrisation that embeds the saliency of the weights in their expression and thereby soft-prune them during training. Both components are based on our reparametrisation function that acts as a surrogate ℓ_0 norm and have been carefully designed to be differentiable and numerically stable.

We validated our approach by comparing our method against magnitude pruning with and without fine-tuning on various datasets and network architectures. Our method performs consistently better than magnitude pruning without fine-tuning and, for almost all tested pruning rates, better than magnitude pruning with fine-tuning. We also validated the relevance of each component of our method individually in a set of comparative experiments. Finally, we provided experimental results to discuss and support the choice of the mixing coefficient and tested our method on trained and pruned initialisation to show the importance of budget enforcement and weight reparametrisation, even on already pruned networks when they undergo fine-tuning.

Pruning without weight training with stochastic sampling. When it comes to estimating the saliency of weights, the general approach is to derive an indicator based on their value, such as magnitude pruning which considers the absolute value of the weight as its saliency. However, these approaches, by design, cannot treat differently two connections with the same weight value. In chapter 5, we proposed a new stochastic approach to extract lightweight subnetworks from a large untrained network. This approach estimates the importance of a weight based on trained masks which are auxiliary variables that represent their associated weight saliency and are consequently not bound to the value of the weights. Furthermore, to also tackle the aforementioned issue with the necessity to fine-tune pruned networks, the method detailed in chapter 5 does not require any weight training and relies purely on topology selection through the optimisation of the auxiliary masks. This method works by stochastically sampling topologies from a large untrained network, based on the value of the masks, interpreted as probabilities of selection of the corresponding weight. These sampled topologies are evaluated to eventually identify a subnetwork with compelling performances. The subnetwork is extracted by pruning the weights of the large network identified as redundant from the larger network. Notably, the performance of this subnetwork does not experience any drop when compared to the larger network before pruning. To achieve this, we introduced two components called Arbitrarily Shifted Log Parametrisation (ASLP) and Smart Rescale (SR). The former is a computationally efficient and numerically stable technique that relies on Gumbel-Softmax to train the masks in a stochastic context. The latter is an efficient learnt-based weight rescaling mechanism that allows the network to rescale the weight distributions in order to mitigate the

disruption of the weight distribution statistics caused by the pruning. We also introduce a thresholding strategy responsible for pruning the weights, that allows to effectively *freeze* the topology.

We validated our approach by comparing our method against other state-of-the-art methods on various datasets and network architectures. Our method performs better than those other methods in most tested scenarios, offering higher accuracy. We also provided experimental results to validate the relevance of our SR mechanism and thresholding strategy, support our choice of learning rate and finally, show that our method is robust to modification of the pruning rate post-training. Finally, our code has been made publicly available [1] and contains the instructions to reproduce our results as well as a reimplementation of the state-of-the-art method we benchmark against in PyTorch.

6.2 Perspectives

In this section, we discuss the perspectives and future works that could be undertaken to improve the methods we proposed in this thesis as well as push forward the findings we made.

Experimental validation on larger datasets and architectures. In our experiments, we chose to focus on results reliability and therefore we chose to run every configuration for every experiment at least 5 times to average the results and provide their standard deviation. This choice was made to avoid drawing conclusions based on a single run that could be an outlier. However, this choice comes at the cost of computational time and resources, thereby limiting the scale of datasets and architectures we could evaluate.

Futur works and development efforts could target the evaluation of our method on larger networks and datasets, namely the ResNet-50 architecture [67], Vision Transformers [30], both in combination with the ImageNet dataset [170]. A larger dataset like ImageNet would allow to sample more topologies and therefore explore the topology space more thoroughly.

[1] Code available at: `https://github.com/NOciple/ASLP`

Structured Pruning. The methods introduced in chapters 4 and 5 are unstructured pruning methods, meaning that they prune weights individually which is a flexible approach that allows to reach high pruning rates. However, the speedup obtained by unstructured pruning is not straightforward and could necessitate additional optimisations. On the other hand, structured pruning methods, which prune weights in groups, yield networks with lower pruning rates but with a regular structure. This regularity can be exploited to obtain a more straightforward speedup in the most popular Deep Learning frameworks [147, 1].

Our method, Arbitrarily Shifted Log Parametrisation (ASLP), could benefit from a structured pruning approach. In addition to the aforementioned network regularity, using a structured approach could allow to reduce the number of masks to train. Instead of training a mask per weight, it is possible to train a mask per group of weights. This could lead to significant memory savings and speedups during training since the sampling operation takes a heavy toll on the GPU. Our preliminary works on a semi-structured approach, where we start by pruning the network with a structured approach and then perform an unstructured pruning step afterwards, limits the sampling: we only sample the weights that are not pruned by the structured part. This approach is promising since it can reduce on average the number of masks to sample. However, the theoretical sampling speedup is not observed in practice due to memory latency caused by partial access to the masks. A careful reimplementation of the mask partial selection and sampling logic could resolve this issue and allow for faster sampling.

Controlling mask magnitude. In chapter 5, we used a learning rate value of 50 that is several orders of magnitude higher than standard learning rates used in baselines training [22]. This choice is motivated and explained in section 5.5.4. However, this high learning rate together with vanishingly small gradients as masks move away from the origin (as explained in section 5.3.1) can lead to masks being stuck at their high or low value and therefore being effectively frozen.

Adding a regularisation term to the loss function that penalises masks with extreme values, or any other mechanism that can limit the magnitude of the masks could help to mitigate this issue and prevent a mask from being frozen. Our preliminary experiments with naive regularisation

loss show improved results in the aforementioned semi-supervised setup.

Better initialisation scheme. The ASLP method introduced in chapter 5 extracts a lightweight and effective neural network from a large untrained one. The weights of the large network are initialised with state-of-the-art methods such as Kaiming initialisation [66] and are not modified. However, these initialisations are designed with weight training in mind and might not be optimal for the ASLP method which does not train the weights.

A better initialisation scheme could be designed to improve the performance of the ASLP method. This initialisation scheme could be inspired by trained weight distributions and could be designed to be more robust to the pruning and sampling operation.

Training through pruning. ASLP and experiments conducted in chapter 5 showed that it is possible to achieve compelling performances without training the weights. This raises the question of the very necessity of training the weights and opens the way for new research directions that investigate the possibility of training a network through pruning. Furthermore, in this context, the word *training* is to be understood *lato sensu* and could include any strategy that selects a topology, not necessarily strategies that rely on gradient-based mask training as we proposed.

Appendix A

Appendix

A.1 Relationship between Multiply-Accumulate Operations and the Number of Parameters

For a convolution operation, the number of parameters of a layer is not representative of its computational complexity. Each kernel has to be spatially convolved with the entire input. The resulting convolutional complexity is, for one part, highly dependent on the input size, and for the other part, higher than the number of parameters.

Without loss of generality, consider a 2D square matrix M of size $m \times m$, and a 2D convolution kernel K of size $k \times k$, with $k \ll m$. The output of the spatial convolution of M by K is denoted O. The matrix O is of size $(m-k+1) \times (m-k+1)$. Each one of the $(m-k+1)^2$ elements of O necessitates k^2 multiplications and k^2-1 additions. For the sake of simplicity, we will consider k^2 Multiply-Accumulates (MACs) operations per element of O. The total number of MACs needed to compute O, denoted μ, is therefore:

$$\mu = (m-k+1)^2 \times k^2$$

Considering that there are k^2 elements in K, the ratio between the number of MACs and the number of parameters is:

$$\frac{\mu}{k^2} = (m-k+1)^2$$

Since $k \ll m$, the ratio $\frac{\mu}{k^2}$ is always greater than 1, and grows quadratically with m. Therefore, for a 2D convolution, the computational complexity can roughly be estimated as $(m-k+1)^2$ times the number of parameters in the convolution kernel.

A.2 Scheduling of the Mixing Coefficient λ

This section presents the test accuracy of a Conv4 network trained on CIFAR-10 with the method introduced in section 4.2.1 when scheduling is applied on the mixing coefficient λ. Two trends are tested for λ: increasing and decreasing. Given a $\lambda_{\max}$, a maximum number of epochs $e_{\max}$, the current epoch e, and a shape parameter p, the decreasing scheduling is defined as follows:

$$\lambda_e = \lambda_{\max} \sqrt[p]{1 - \left(\frac{e}{e_{\max}}\right)^p}$$

and the increasing scheduling is defined as follows:

$$\lambda_e = \lambda_{\max} \left(1 - \sqrt[p]{1 - \left(\frac{e}{e_{\max}}\right)^p}\right)$$

Examples of the evolution of λ for different values of p are shown in figure A.1 and the related performances in table A.1.

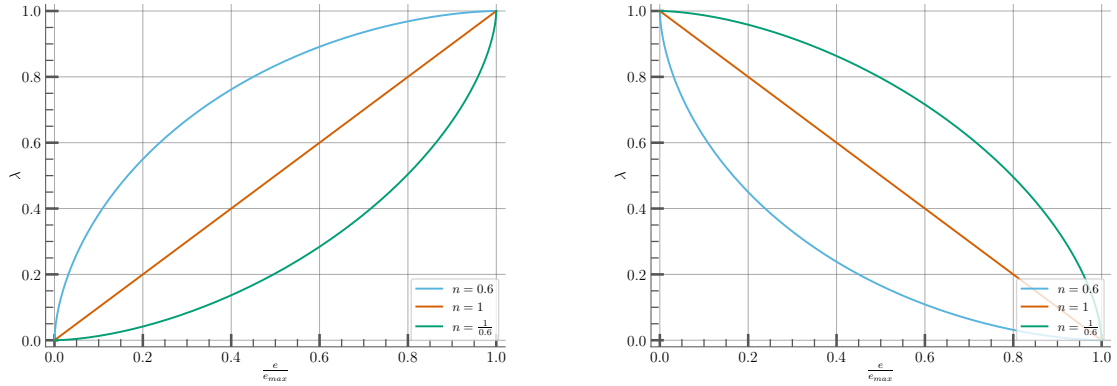
A.3 Xavier and Kaiming Initialisations

Glorot and Kaiming initializations are strategies for initializing the weights of neural networks. They are designed to help mitigate the issues of vanishing and exploding gradients, which can occur during the training of deep neural networks.

Glorot Initialization, also known as Xavier Initialization, suggests that the initial weights of the network should be drawn from a distribution with zero mean and a specific variance. The variance is dependent on the

pruning rate (%)	λ **Trend**	p	**Test Accuracy** (%)
90	incr.	0.6	85.46 ± 0.18
		1	85.43 ± 0.46
		$\frac{1}{0.6}$	84.96 ± 0.53
	decr.	0.6	85.36 ± 0.59
		1	85.55 ± 0.47
		$\frac{1}{0.6}$	85.50 ± 0.29
95	incr.	0.6	84.00 ± 0.85
		1	79.28 ± 0.96
		$\frac{1}{0.6}$	66.43 ± 05.13
	decr.	0.6	83.53 ± 0.65
		1	83.42 ± 0.50
		$\frac{1}{0.6}$	84.07 ± 0.92
99	incr.	0.6	14.27 ± 3.21
		1	11.05 ± 01.26
		$\frac{1}{0.6}$	10.33 ± 0.43
	decr.	0.6	34.36 ± 33.40
		1	52.51 ± 29.65
		$\frac{1}{0.6}$	45.37 ± 32.32

Table A.1: Conv4 test accuracy on CIFAR-10, with $\lambda = 50$, for increasing (*incr.*) and decreasing (*decr.*) scheduling for various pruning rates and values of the parameter p. The networks have been trained for 300 epochs.



(a) Increasing scheduling **(b)** Decreasing scheduling

Figure A.1: Evolution of the mixing coefficient λ for different values of p and for increasing and decreasing scheduling. Best viewed in color.

number of input and output neurons in the weight tensor. Kaiming initialisation is a modification of Glorot initialisation that is tailored for neural networks with ReLU activations. It is designed to take into account the fact that ReLU activations nullify half of the input values. These two types of initialisation can be used with either a normal or uniform distribution. They impact the standard deviation (and consequently the variance) of the underlying distribution. The standard deviation for Glorot normal initialisation is computed as follows:

$$\sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}} \tag{A.1}$$

where n_{in} and n_{out} are the number of input and output neurons in the weight tensor. The standard deviation for Kaiming normal initialisation is computed as follows:

$$\sigma = \frac{1}{\sqrt{n_{\text{in}}}} \tag{A.2}$$

where n_{in} is the number of input neurons in the weight tensor.

It is important to note that, when using the Pytorch framework, this standard deviation is adapted depending on the type of non-linearities used

in the network. For instance, using ReLU activation functions require multiplying the standard deviation by $\sqrt{2}$ [152].

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL `http://arxiv.org/abs/1603.04467`.

[2] S. Ahn, S. X. Hu, A. C. Damianou, N. D. Lawrence, and Z. Dai. Variational information distillation for knowledge transfer. In *CVPR*, 2019. doi: 10.1109/CVPR.2019.00938. URL `http://openaccess.thecvf.com/content_CVPR_2019/html/Ahn_Variational_Information_Distillation_for_Knowledge_Transfer_CVPR_2019_paper.html`.

[3] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary neural networks for resource-efficient ai applications. In *2017 international joint conference on neural networks (IJCNN)*, pages 2547–2554. IEEE, 2017.

[4] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang,

Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 173–182. JMLR.org, 2016. URL `http://proceedings.mlr.press/v48/amodei16.html`.

[5] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):1–18, 2017.

[6] Sercan Ö Arık, Mike Chrzanowski, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Andrew Ng, Jonathan Raiman, et al. Deep voice: Real-time neural text-to-speech. In *International conference on machine learning*, pages 195–204. PMLR, 2017.

[7] Wolfgang Balzer, Masanobu Takahashi, Jun Ohta, and Kazuo Kyuma. Weight quantization in boltzmann machines. *Neural Networks*, 4(3):405–409, 1991.

[8] David Barber and Felix Agakov. The im algorithm: a variational approach to information maximization. *Advances in neural information processing systems*, 16(320):201, 2004.

[9] Y. Bengio, N. Léonard, and A. C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432, 2013.

[10] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, pages 2546–2554, 2011. URL `https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cfd12577bc2619bc635690-Abstract.html`.

[11] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[12] Pawel Budzianowski and Ivan Vulic. Hello, it's GPT-2 - how can I help you? towards the use of pretrained language models for task-oriented dialogue systems. In Alexandra Birch, Andrew M. Finch, Hiroaki Hayashi, Ioannis Konstas, Thang Luong, Graham Neubig, Yusuke Oda, and Katsuhito Sudoh, editors, *Proceedings of the 3rd Workshop on Neural Generation and Translation@EMNLP-IJCNLP 2019, Hong Kong, November 4, 2019*, pages 15–22. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-5602. URL https://doi.org/10.18653/v1/D19-5602.

[13] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, pages 4960–4964. IEEE, 2016. doi: 10.1109/ICASSP.2016.7472621. URL https://doi.org/10.1109/ICASSP.2016.7472621.

[14] Jianda Chen, Shangyu Chen, and Sinno Jialin Pan. Storage efficient and dynamic flexible runtime channel pruning via deep reinforcement learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/a914ecef9c12ffdb9bede64bb703d877-Abstract.html.

[15] Yu Cheng, Felix X. Yu, Rogério Schmidt Feris, Sanjiv Kumar, Alok N. Choudhary, and Shih-Fu Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 2857–2865. IEEE Computer Society, 2015. doi: 10.1109/ICCV.2015.327. URL https://doi.org/10.1109/ICCV.2015.327.

[16] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[17] Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhen-zhong Lan. Training binary multilayer neural networks for image classification

using expectation backpropagation. *CoRR*, abs/1503.03562, 2015. URL http://arxiv.org/abs/1503.03562.

[18] Lu Chi, Borui Jiang, and Yadong Mu. Fast fourier convolution. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/2fd5d41ec6cfab47e32164d5624269b1-Abstract.html.

[19] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In Stefan Wermter, Cornelius Weber, Wlodzislaw Duch, Timo Honkela, Petia D. Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa, editors, *Artificial Neural Networks and Machine Learning - ICANN 2014 - 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings*, volume 8681 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 2014. doi: 10.1007/978-3-319-11179-7_36. URL https://doi.org/10.1007/978-3-319-11179-7_36.

[20] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

[21] NVIDIA Corporation. cuDNN: NVIDIA CUDA Deep Neural Network library. https://developer.nvidia.com/cudnn, 2014. Accessed: May 29, 2023.

[22] Nvidia Corporation. Nvidia deep learning examples for tensor cores, 2023. URL https://github.com/NVIDIA/DeepLearningExamples. [Online; accessed 23-June-2023].

[23] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

[24] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[26] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. doi: 10.1109/MSP.2012.2211477.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1423. URL `https://doi.org/10.18653/v1/n19-1423`.

[28] Xiaohan Ding, Guiguang Ding, Yuchen Guo, Jungong Han, and Chenggang Yan. Approximated oracle filter pruning for destructive CNN width optimization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 1607–1616. PMLR, 2019. URL `http://proceedings.mlr.press/v97/ding19a.html`.

[29] Ke Dong, Chengjie Zhou, Yihan Ruan, and Yuzhi Li. Mobilenetv2 model for image classification. In *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*, pages 476–480, 2020. doi: 10.1109/ITCA52113.2020.00106.

[30] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=YicbFdNTTy`.

[31] Charles-Éric Drevet, Md. Nazrul Islam, and Éric Schost. Optimization techniques for small matrix multiplication. *ACM Commun. Comput. Algebra*, 44(3/4):107–108, 2010. doi: 10.1145/1940475.1940488. URL https://doi.org/10.1145/1940475.1940488.

[32] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[33] Marat Dukhan, Yiming Wu, and Hao Lu. Qnnpack: Open source library for optimized mobile deep learning. https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/, 2018. Accessed: 26/05/2023.

[34] R. Dupont. ASLP - Our implementation. https://github.com/N0ciple/ASLP, 2022.

[35] Robin Dupont, Hichem Sahbi, and Guillaume Michel. Weight reparametrization for budget-aware network pruning. In *2021 IEEE International Conference on Image Processing, ICIP 2021, Anchorage, AK, USA, September 19-22, 2021*, pages 789–793. IEEE, 2021. doi: 10.1109/ICIP42928.2021.9506265. URL https://doi.org/10.1109/ICIP42928.2021.9506265.

[36] Robin Dupont, Mohammed Amine Alaoui, Hichem Sahbi, and Alice Lebois. Extracting effective subnetworks with gumbel-softmax. In *2022 IEEE International Conference on Image Processing, ICIP 2022, Bordeaux, France, 16-19 October 2022*, pages 931–935. IEEE, 2022. doi: 10.1109/ICIP46576.2022.9897718. URL https://doi.org/10.1109/ICIP46576.2022.9897718.

[37] Sara Elkerdawy, Mostafa Elhoushi, Hong Zhang, and Nilanjan Ray. Fire together wire together: A dynamic pruning approach with self-supervised mask prediction. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 12444–12453. IEEE, 2022. doi: 10.1109/CVPR52688.2022.01213. URL https://doi.org/10.1109/CVPR52688.2022.01213.

[38] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[39] Max Ferguson, Ronay Ak, Yung-Tsun Tina Lee, and Kincho H Law. Automatic localization of casting defects with convolutional neural networks. In *2017 IEEE international conference on big data (big data)*, pages 1726–1735. IEEE, 2017.

[40] Emile Fiesler, Amar Choudry, and H John Caulfield. Weight discretization paradigm for optical neural networks. In *Optical interconnections and networks*, volume 1281, pages 164–173. SPIE, 1990.

[41] Allen Institute for AI. hidden-networks: A repository for research on neural networks, 2023. URL `https://github.com/allenai/hidden-networks/blob/dddf2d093de568fc76d460a77fa2650e56e79c1a/data/cifar.py#L29C18-L29C18`. [Online; accessed 23-June-2023].

[42] Charles W Fox and Stephen J Roberts. A tutorial on variational bayesian inference. *Artificial intelligence review*, 38:85–95, 2012.

[43] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL `https://openreview.net/forum?id=rJl-b3RcF7`.

[44] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. The lottery ticket hypothesis at scale. *CoRR*, abs/1903.01611, 2019. URL `http://arxiv.org/abs/1903.01611`.

[45] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 3259–3269. PMLR, 2020. URL `http://proceedings.mlr.press/v119/frankle20a.html`.

[46] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Pruning neural networks at initialization: Why are we missing the mark? *arXiv preprint arXiv:2009.08576*, 2020.

[47] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *CoRR*, abs/1902.09574, 2019. URL `http://arxiv.org/abs/1902.09574`.

[48] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015. URL `http://arxiv.org/abs/1508.06576`.

[49] Joseph C Giarratano and Gary Riley. *Expert systems: principles and programming*. PWS Publishing Co., 1994.

[50] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[51] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[52] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[53] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63 (11):139–144, 2020.

[54] Robert M Gray et al. Toeplitz and circulant matrices: A review. *Foundations and Trends® in Communications and Information Theory*, 2(3):155–239, 2006.

[55] E.J. Gumbel. Les valeurs extrêmes des distributions statistiques. *Annales de l'Institut Henri Poincaré*, 5(2):115–158, 1935.

[56] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1379–1387, 2016. URL `https://proceedings.neurips.cc/paper/2016/hash/2823f4797102ce1a1aec05359cc16dd9-Abstract.html`.

[57] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[58] Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep pyramidal residual networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6307–6315. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.668. URL https://doi.org/10.1109/CVPR.2017.668.

[59] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural network. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 1135–1143, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html.

[60] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1510.00149.

[61] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014. URL http://arxiv.org/abs/1412.5567.

[62] Stephen Hanson and Lorien Pratt. Comparing biases for minimal network construction with back-propagation. *Advances in neural information processing systems*, 1, 1988.

[63] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In Stephen Jose Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992]*, pages 164–171. Morgan Kaufmann, 1992.

[64] Babak Hassibi, David G. Stork, and Gregory J. Wolff. Optimal brain surgeon and general network pruning. In *Proceedings of International Conference on Neural Networks (ICNN'88), San Francisco,*

CA, USA, March 28 - April 1, 1993, pages 293–299. IEEE, 1993. doi: 10.1109/ICNN.1993.298572. URL https://doi.org/10.1109/ICNN.1993.298572.

[65] Babak Hassibi, David G. Stork, and Gregory J. Wolff. Optimal brain surgeon: Extensions and performance comparison. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, pages 263–270. Morgan Kaufmann, 1993.

[66] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

[68] Yang He and Lingao Xiao. Structured pruning for deep convolutional neural networks: A survey. *CoRR*, abs/2303.00566, 2023. doi: 10.48550/arXiv.2303.00566. URL https://doi.org/10.48550/arXiv.2303.00566.

[69] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2234–2240. ijcai.org, 2018. doi: 10.24963/ijcai.2018/309. URL https://doi.org/10.24963/ijcai.2018/309.

[70] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 4340–4349. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.00447. URL http://openaccess.thecvf.com/content_CVPR_2019/html/He_Filter_Pruning_via_Geometric_Median_for_Deep_Convolutional_Neural_Networks_CVPR_2019_paper.html.

[71] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 1398–1406. IEEE Computer Society, 2017. doi: 10.1109/ICCV.2017.155. URL https://doi.org/10.1109/ICCV.2017.155.

[72] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: automl for model compression and acceleration on mobile devices. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VII*, volume 11211 of *Lecture Notes in Computer Science*, pages 815–832. Springer, 2018. doi: 10.1007/978-3-030-01234-2_48. URL https://doi.org/10.1007/978-3-030-01234-2_48.

[73] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[74] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015. URL http://arxiv.org/abs/1503.02531.

[75] Geoffrey Hinton. Neural networks for machine learning, lecture 6.6 - rmsprop: Divide the gradient by a running average of its recent magnitude. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, 2012.

[76] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL http://arxiv.org/abs/1207.0580.

[77] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[78] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

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

[80] Andrew Howard, Ruoming Pang, Hartwig Adam, Quoc V. Le, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, and Yukun Zhu. Searching for mobilenetv3. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 1314–1324. IEEE, 2019. doi: 10.1109/ICCV.2019.00140. URL https://doi.org/10.1109/ICCV.2019.00140.

[81] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[82] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *CoRR*, abs/1607.03250, 2016. URL http://arxiv.org/abs/1607.03250.

[83] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 7132–7141. Computer Vision Foundation / IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00745. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Hu_Squeeze-and-Excitation_Networks_CVPR_2018_paper.html.

[84] G. Huang, S. Liu, L. van der Maaten, and K. Q. Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *CVPR*, 2018. doi: 10.1109/CVPR.2018.00291. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Huang_CondenseNet_An_Efficient_CVPR_2018_paper.html.

[85] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[86] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2752–2761, 2018.

[87] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In Daniel D. Lee,

Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4107–4115, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/d8330f857a17c53d217014ee776bfd50-Abstract.html.

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

[89] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. URL http://arxiv.org/abs/1602.07360.

[90] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/ioffe15.html.

[91] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 2704–2713. Computer Vision Foundation / IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00286. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Jacob_Quantization_and_Training_CVPR_2018_paper.html.

[92] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=rkE3y85ee.

[93] Steven A Janowsky. Pruning versus clipping in neural networks. *Physical Review A*, 39(12):6600, 1989.

[94] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Bo-

den, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[95] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[96] Minsoo Kang and Bohyung Han. Operation-aware soft channel pruning using differentiable masks. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5122–5131. PMLR, 2020. URL `http://proceedings.mlr.press/v119/kang20a.html`.

[97] Ehud D. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Trans. Neural Networks*, 1(2):239–242, 1990. doi: 10.1109/72.80236. URL `https://doi.org/10.1109/72.80236`.

[98] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4401–4410, 2019.

[99] Chanwoo Kim, Dhananjaya Gowda, Dongsoo Lee, Jiyeon Kim, Ankur Kumar, Sungsoo Kim, Abhinav Garg, and Changwoo Han. A review of on-device fully neural end-to-end automatic speech recognition algorithms. In *2020 54th Asilomar Conference on Signals, Systems, and Computers*, pages 277–283. IEEE, 2020.

[100] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[101] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *Advances in neural information processing systems*, 30, 2017.

[102] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural*

Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States, pages 1106–1114, 2012. URL https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[103] Sampo Kuutti, Richard Bowden, Yaochu Jin, Phil Barber, and Saber Fallah. A survey of deep learning applications to autonomous vehicle control. *IEEE Transactions on Intelligent Transportation Systems*, 22(2):712–733, 2020.

[104] Ya Le and Xuan S. Yang. Tiny imagenet visual recognition challenge. ., 2015.

[105] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pages 598–605. Morgan Kaufmann, 1989. URL http://papers.nips.cc/paper/250-optimal-brain-damage.

[106] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791. URL https://doi.org/10.1109/5.726791.

[107] N. Lee, T. Ajanthan, and P. H. S. Torr. Snip: single-shot network pruning based on connection sensitivity. In *ICLR*, 2019.

[108] Carl Lemaire, Andrew Achkar, and Pierre-Marc Jodoin. Structured pruning of neural networks with budget-aware regularization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9108–9116, 2019.

[109] Eric Mingjie Li. Rethinking the value of network pruning - implementation, 2019. URL https://github.com/Eric-mingjie/rethinking-network-pruning/blob/2ac473d70a09810df888e932bb394f225f9ed2d1/cifar/network-slimming/main.py#L67. [Online; accessed 23-June-2023].

[110] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=rJqFGTslg.

[111] Zhuo Li, Hengyi Li, and Lin Meng. Model compression for deep neural networks: A survey. *Computers*, 12(3):60, 2023.

[112] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021. doi: 10.1016/j.neucom.2021.07.045. URL https://doi.org/10.1016/j.neucom.2021.07.045.

[113] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[114] Siyu Liao, Ashkan Samiee, Chunhua Deng, Yu Bai, and Bo Yuan. Compressing deep neural networks using toeplitz matrix: Algorithm design and fpga implementation. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1443–1447. IEEE, 2019.

[115] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 2181–2191, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/a51fb975227d6640e4fe47854476d133-Abstract.html.

[116] Jinhua Lin and Yu Yao. A fast algorithm for convolutional neural networks using tile-based fast fourier transforms. *Neural Process. Lett.*, 50(2):1951–1967, 2019. doi: 10.1007/s11063-019-09981-z. URL https://doi.org/10.1007/s11063-019-09981-z.

[117] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1510.03009.

[118] Chenxi Liu, Liang-Chieh Chen, Florian Schroff, Hartwig Adam, Wei Hua, Alan L. Yuille, and Li Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019,*

Long Beach, CA, USA, June 16-20, 2019, pages 82–92. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.00017. URL http://openaccess.thecvf.com/content_CVPR_2019/html/Liu_Auto-DeepLab_Hierarchical_Neural_Architecture_Search_for_Semantic_Image_Segmentation_CVPR_2019_paper.html.

[119] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=S1eYHoC5FX.

[120] Kang Liu. Pytorch models for cifar-10, 2020. URL https://github.com/kuangliu/pytorch-cifar.

[121] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I*, volume 9905 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016. doi: 10.1007/978-3-319-46448-0_2. URL https://doi.org/10.1007/978-3-319-46448-0_2.

[122] Xingyu Liu, Jeff Pool, Song Han, and William J Dally. Efficient sparse-winograd convolutional neural networks. *arXiv preprint arXiv:1802.06367*, 2018.

[123] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. In *ICLR*, 2019. URL https://openreview.net/forum?id=rJlnB3C5Ym.

[124] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2755–2763. IEEE Computer Society, 2017. doi: 10.1109/ICCV.2017.298. URL https://doi.org/10.1109/ICCV.2017.298.

[125] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[126] I. Loshchilov and F. Hutter. SGDR: stochastic gradient descent with warm restarts. In *ICLR (Poster)*, 2017.

[127] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[128] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=H1Y8hhg0b.

[129] Liqiang Lu and Yun Liang. Spwa: An efficient sparse winograd convolutional neural networks accelerator on fpgas. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[130] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 5068–5076. IEEE Computer Society, 2017. doi: 10.1109/ICCV.2017.541. URL https://doi.org/10.1109/ICCV.2017.541.

[131] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 122–138, Cham, 2018. Springer International Publishing. ISBN 978-3-030-01264-9.

[132] E. Malach, G. Yehudai, S. Shalev-Shwartz, and O. Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *ICML*, 2020. URL http://proceedings.mlr.press/v119/malach20a.html.

[133] John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon. A proposal for the dartmouth summer research project on artificial intelligence. Available at AI Magazine Vol 27 No 4, https://www.aaai.org/ojs/index.php/aimagazine/article/view/1904, 1956.

[134] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[135] Ian McGraw, Rohit Prabhavalkar, Raziel Alvarez, Montse Gonzalez Arenas, Kanishka Rao, David Rybach, Ouais Alsharif, Haşim Sak, Alexander Gruenstein, Françoise Beaufays, et al. Personalized speech recognition on mobile devices. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5955–5959. IEEE, 2016.

[136] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017. URL `http://arxiv.org/abs/1703.00548`.

[137] S.-I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *AAAI*, 2020. URL `https://aaai.org/ojs/index.php/AAAI/article/view/5963`.

[138] Dmitry Molchanov, Arsenii Ashukha, and Dmitry P. Vetrov. Variational dropout sparsifies deep neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2498–2507. PMLR, 2017. URL `http://proceedings.mlr.press/v70/molchanov17a.html`.

[139] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL `https://openreview.net/forum?id=SJGCiw5gl`.

[140] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 11264–11272. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.01152. URL `http://openaccess.thecvf.com/`

content_CVPR_2019/html/Molchanov_Importance_Estimation_for_Neural_Network_Pruning_CVPR_2019_paper.html.

[141] Michael Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 1, [NIPS Conference, Denver, Colorado, USA, 1988]*, pages 107–115. Morgan Kaufmann, 1988. URL http://papers.nips.cc/paper/119-skeletonization-a-technique-for-trimming-the-fat-from-a-networ

[142] Nils J Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann, 1998.

[143] Nvidia. 8-bit inference with tensorrt. https://developer.nvidia.com/tensorrt, 2021. Accessed: 26/05/2023.

[144] Alan V Oppenheim, Alan S Willsky, Syed Hamid Nawab, and Jian-Jiun Ding. *Signals and systems*, volume 2. Prentice hall Upper Saddle River, NJ, 1997.

[145] L. Orseau, M. Hutter, and O. Rivasplata. Logarithmic pruning is all you need. In *NeurIPS 2020*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1e9491470749d5b0e361ce4f0b24d037-Abstract.html.

[146] Nikolaos Passalis and Anastasios Tefas. Learning deep representations with probabilistic knowledge transfer. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XI*, volume 11215 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2018. doi: 10.1007/978-3-030-01252-6_17. URL https://doi.org/10.1007/978-3-030-01252-6_17.

[147] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32:*

Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[148] A. Pensia, S. Rajput, A. Nagle, H. Vishwakarma, and D. S. Papailiopoulos. Optimal lottery tickets via subset sum: Logarithmic over-parameterization is sufficient. In *NeurIPS 2020*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1b742ae215adf18b75449c6e272fd92d-Abstract.html.

[149] Adam Polyak and Lior Wolf. Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175, 2015. doi: 10.1109/ACCESS.2015.2494536. URL https://doi.org/10.1109/ACCESS.2015.2494536.

[150] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[151] Harry Pratt, Bryan M. Williams, Frans Coenen, and Yalin Zheng. FCNN: fourier convolutional neural networks. In Michelangelo Ceci, Jaakko Hollmén, Ljupco Todorovski, Celine Vens, and Saso Dzeroski, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2017, Skopje, Macedonia, September 18-22, 2017, Proceedings, Part I*, volume 10534 of *Lecture Notes in Computer Science*, pages 786–798. Springer, 2017. doi: 10.1007/978-3-319-71249-9_47. URL https://doi.org/10.1007/978-3-319-71249-9_47.

[152] Pytorch. Pytorch weights initialisation. Software, 2023. URL https://pytorch.org/docs/stable/nn.init.html.

[153] Pytorch. Resnet-18 implementation. Software, 2023. URL https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py.

[154] PyTorch. Pytorch vision models. https://pytorch.org/vision/master/models.html, Accessed 2023. Accessed on May 24, 2023.

[155] Qualcomm. Snapdragon neural processing engine sdk. https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk, 2021. Accessed: 26/05/2023.

[156] Ramchalam Kinattinkara Ramakrishnan, Eyyüb Sari, and Vahid Partovi Nia. Differentiable mask for pruning convolutional and recurrent networks. In *17th Conference on Computer and Robot Vision, CRV 2020, Ottawa, ON, Canada, May 13-15, 2020*, pages 222–229. IEEE, 2020. doi: 10.1109/CRV50864.2020.00037. URL https://doi.org/10.1109/CRV50864.2020.00037.

[157] V. Ramanujan, M. Wortsman, A. Kembhavi, A. Farhadi, and M. Rastegari. What's hidden in a randomly weighted neural network? In *CVPR*, 2020.

[158] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka I. Leon-Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2902–2911. PMLR, 2017. URL http://proceedings.mlr.press/v70/real17a.html.

[159] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.91. URL https://doi.org/10.1109/CVPR.2016.91.

[160] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 91–99, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/14bfa6bb14875e45bba028a21ed38046-Abstract.html.

[161] Yi Ren, Yangjun Ruan, Xu Tan, Tao Qin, Sheng Zhao, Zhou Zhao, and Tie-Yan Liu. Fastspeech: Fast, robust and controllable text to speech. *Advances in neural information processing systems*, 32, 2019.

[162] Canadian Institute For Advanced Research. Cifar-10 and cifar-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html, 2009.

[163] Facebook AI Research. Openlth: A framework for lottery tickets and beyond, 2020. URL https://github.com/facebookresearch/open_lth/blob/2ce732fe48abd5a80c10a153c45d397b048e980c/datasets/cifar10.py#L46C50-L46C50. [Online; accessed 23-June-2023].

[164] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[165] A. Romero, N. Ballas, S. Ebrahimi Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. In *ICLR*, 2015. URL http://arxiv.org/abs/1412.6550.

[166] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[167] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[168] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[169] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323 (6088):533–536, 1986.

[170] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. URL https://doi.org/10.1007/s11263-015-0816-y.

[171] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018. doi: 10.1109/CVPR.2018.00474. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Sandler_MobileNetV2_Inverted_Residuals_CVPR_2018_paper.html.

[172] Jonathan Schwarz, Siddhant Jayakumar, Razvan Pascanu, Peter E Latham, and Yee Teh. Powerpropagation: A sparsity inducing weight

reparameterisation. *Advances in neural information processing systems*, 34:28889–28903, 2021.

[173] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[174] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[175] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.1556.

[176] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 963–971, 2014. URL https://proceedings.neurips.cc/paper/2014/hash/076a0c97d09cf1a0ec3e19c7f2529f2b-Abstract.html.

[177] Kartik Sreenivasan, Jy-yong Sohn, Liu Yang, Matthew Grinde, Alliot Nagle, Hongyi Wang, Kangwook Lee, and Dimitris S. Papailiopoulos. Rare gems: Finding lottery tickets at initialization. *CoRR*, abs/2202.12002, 2022. URL https://arxiv.org/abs/2202.12002.

[178] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15 (1):1929–1958, 2014. doi: 10.5555/2627435.2670313. URL https://dl.acm.org/doi/10.5555/2627435.2670313.

[179] Hugo Steinhaus et al. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci*, 1(804):801, 1956.

[180] Volker Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[181] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[182] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[183] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 1–9. IEEE Computer Society, 2015. doi: 10.1109/CVPR.2015.7298594. URL `https://doi.org/10.1109/CVPR.2015.7298594`.

[184] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*. PMLR, 2019. URL `http://proceedings.mlr.press/v97/tan19a.html`.

[185] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 2820–2828. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.00293. URL `http://openaccess.thecvf.com/content_CVPR_2019/html/Tan_MnasNet_Platform-Aware_Neural_Architecture_Search_for_Mobile_CVPR_2019_paper.html`.

[186] H. Tanaka, D. Kunin, D. L. Yamins, and S. Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. In *NeurIPS*, 2020.

[187] Tencent. Ncnn: A high-performance neural network inference framework optimized for the mobile platform. `https://github.com/Tencent/ncnn`, 2021. Accessed: 26/05/2023.

[188] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288, 1996.

[189] Rishabh Tiwari, Udbhav Bamba, Arnav Chavan, and Deepak K Gupta. Chipnet: Budget-aware pruning with heaviside continuous approximations. *arXiv preprint arXiv:2102.07156*, 2021.

[190] Antonio Torralba, Rob Fergus, and William T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008. doi: 10.1109/TPAMI.2008.128.

[191] Inc. Uber Technologies. masked_layers.py in deconstructing-lottery-tickets repository, 2019. URL `https://github.com/uber-research/deconstructing-lottery-tickets/blob/master/masked_layers.py`. Licensed under the Uber Non-Commercial License.

[192] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[193] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

[194] Alvin Wan. Neural-backed decision trees source code, 2020. URL `https://github.com/alvinwan/neural-backed-decision-trees`.

[195] Alvin Wan, Lisa Dunlap, Daniel Ho, Jihan Yin, Scott Lee, Henry Jin, Suzanne Petryk, Sarah Adel Bargal, and Joseph E. Gonzalez. Nbdt: Neural-backed decision trees, 2020.

[196] C. Wang, G. Zhang, and R. B. Grosse. Picking winning tickets before training by preserving gradient flow. In *ICLR*, 2020.

[197] Xuan Wang, Chao Wang, Jing Cao, Lei Gong, and Xuehai Zhou. Winonn: Optimizing fpga-based convolutional neural network accelerators using sparse winograd algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11): 4290–4302, 2020.

[198] Zi Wang, Chengcheng Li, and Xiangyang Wang. Convolutional neural network pruning with structural redundancy reduction. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 14913–14922. Computer Vision Foundation / IEEE, 2021. doi: 10.1109/CVPR46437.2021.01467. URL https://openaccess.thecvf.com/content/CVPR2021/html/Wang_Convolutional_Neural_Network_Pruning_With_Structural_Redundancy_Reduction_CVPR_2021_paper.html.

[199] Philip D Wasserman and Tom Schwartz. Neural networks. ii. what are they and why is everybody so interested in them now? *IEEE expert*, 3(1):10–15, 1988.

[200] R. Clinton Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.*, 27(1-2):3–35, 2001. doi: 10.1016/S0167-8191(00)00087-9. URL https://doi.org/10.1016/S0167-8191(00)00087-9.

[201] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *CoRR*, abs/2301.08727, 2023. doi: 10.48550/arXiv.2301.08727. URL https://doi.org/10.48550/arXiv.2301.08727.

[202] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=rylqooRqK7.

[203] Hanyuan Xu. Image classification on tiny imagenet, 2018. URL https://github.com/DennisHanyuanXu/Tiny-ImageNet.

[204] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. PC-DARTS: partial channel connections for memory-efficient architecture search. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=BJlS634tPr.

[205] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6071–6079. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.643. URL https://doi.org/10.1109/CVPR.2017.643.

[206] Junho Yim, Donggyu Joo, Ji-Hoon Bae, and Junmo Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 7130–7138. IEEE Computer Society, 2017. doi: 10.1109/CVPR.2017.754. URL https://doi.org/10.1109/CVPR.2017.754.

[207] Zhonghui You, Kun Yan, Jinmian Ye, Meng Ma, and Ping Wang. Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 2130–2141, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/b51a15f382ac914391a58850ab343b00-Abstract.html.

[208] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. NISP: pruning networks using neuron importance score propagation. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 9194–9203. Computer Vision Foundation / IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00958. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Yu_NISP_Pruning_Networks_CVPR_2018_paper.html.

[209] S. Zagoruyko and N. Komodakis. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer. In *ICLR*, 2017. URL https://openreview.net/forum?id=Sks9_ajex.

[210] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[211] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018. doi: 10.1109/CVPR.2018.00716.

[212] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu. Deep mutual learning. In *CVPR*, 2018. doi: 10.1109/CVPR.2018.00454. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Zhang_Deep_Mutual_Learning_CVPR_2018_paper.html.

[213] Chenglong Zhao, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. Variational convolutional neural network pruning. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 2780–2789. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.00289. URL http://openaccess.thecvf.com/content_CVPR_2019/html/Zhao_Variational_Convolutional_Neural_Network_Pruning_CVPR_2019_paper.html.

[214] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=HyQJ-mclg.

[215] H. Zhou, J. Lan, R. Liu, and J. Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *NeurIPS*, 2019.

[216] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[217] Yuefu Zhou, Ya Zhang, Yan-Feng Wang, and Qi Tian. Accelerate CNN via recursive bayesian pruning. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 3305–3314. IEEE, 2019. doi: 10.1109/ICCV.2019.00340. URL https://doi.org/10.1109/ICCV.2019.00340.

[218] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jin-Hui Zhu. Discrimination-aware channel pruning for deep neural networks. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 883–894, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/55a7cf9c71f1c9c495413f934dd1a158-Abstract.html.

[219] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=r1Ue8Hcxg.