Convolutional neural networks pruning and its application to embedded vision systems

Hugo Tessier

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE PAYS DE LA LOIRE –
IMT-ATLANTIQUE

ECOLE DOCTORALE N° 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Signal, Image, Vision*

Par
Hugo TESSIER

Convolutional Neural Networks Pruning and its Application to Embedded Vision Systems

Thèse présentée et soutenue à Brest, à l'IMT Atlantique, le 09/02/2023
Unité de recherche : Lab-STICC, CNRS UMR 6285
Thèse N° : 2023IMTA0350

Rapporteurs avant soutenance :

Damien Querlioz	Chargé de recherche, Centre de Nanosciences et de Nanotechnologies
Stefan Duffner	Maître de conférences, Université de Lyon

Composition du Jury :

Président :	Eva Dokladalova	Professeur, ESIEE Paris
Examinateurs :	Thomas Hannagan	AI Research Scientist, Stellantis
	Damien Querlioz	Chargé de recherche, Centre de Nanosciences et de Nanotechnologies
	Stefan Duffner	Maître de conférences, Université de Lyon
Dir. de thèse :	Vincent Gripon	Directeur de recherche, IMT Atlantique
	Matthieu Arzel	Professeur, IMT Atlantique

Invité(s)

Mathieu Léonardon	Maître de conférences, IMT Atlantique
David Bertrand	Embedded Systems Expert – AI, Stellantis

Contents

Acknowledgments

First, I would like to thank all the people who made this thesis possible, especially Thomas Hannagan and David Bertrand, from Stellantis, for the opportunity they gave me to do this thesis, which will be a major milestone in my future career. Unfortunately, circumstances prevented me from spending more time on site at the company, but I nonetheless greatly appreciated the guidance they provided me, as well as the exchanges we had the opportunity to have. Although I plan to pursue an academic career, I hope to have the opportunity to collaborate together again in the future. I also thank my directors and supervisors from IMT Atlantique: Vincent Gripon, Matthieu Arzel and Mathieu Léonardon, for the great quality of their supervision, all the knowledge they have transmitted to me and for having me introduced to the field of academic research, and it is to be expected that this thesis is only the beginning of our collaboration. Then, I must also thank the members of my thesis commitee, for having contributed to the improvement of this manuscript and for having allowed the completion of this thesis: Eva Dokladalova, Damien Querlioz and Stefan Duffner; I must also mention the members of my CSI comitee, for having ensured the proper conduct of this thesis : Nicolas Dangy-Caye and Benoît Miramond.

Second, I would like to thank all the people I had the opportunity to collaborate or discuss with during my thesis, especially the team of Xavier Touati, from Renesas, and Ghouti Boukli Hacene, from Sony. This allowed me, in particular, to have interactions and discussion of an industrial nature, which helped considerably in pinpointing the issues of this thesis topic.

Third, I would like to thank all the people with whom I had the opportunity to cohabit, espcially my colleagues from the BRAIn team. Finally, I would like to thank my family, especially my mother, as well as my friends.

Résumé long

Introduction

Cette thèse CIFRE fut l'occasion d'une collaboration entre le constructeur automobile Stellantis et le laboratoire Lab-STICC, et plus particulièrement l'équipe BRAIn, située à l'IMT Atlantique. L'intérêt de Stellantis pour la conduite autonome, ou pour bien d'autres tâches d'automatisation (comme l'aide à la conduite), a amené l'entreprise à s'intéresser à la vision par ordinateur, et donc fatalement aux solutions à l'état de l'art depuis une décennie : les réseaux de neurones profonds, et plus particulièrement les réseaux à convolutions.

Le succès des réseaux de neurones s'explique en bonne partie par l'augmentation des ressources en calcul disponible, que ce soit en terme de puissance des processeurs ou des cartes graphiques, ou en terme de super-calculateurs disponibles publiquement. En effet, les réseaux de neurones tendent à être très coûteux en terme de calcul, et s'ils s'implémentent sous forme de multiplications matricielles denses, qui s'accélèrent très bien sur carte graphique, ce coût devient prohibitif sur des types de matériels à la puissance limitée, tels que ce que l'on trouve dans le domaine de l'embarqué.

Les applications industrielles, telles que celles qui intéressent Stellantis, sont typiquement confrontées à ce genre de contraintes. Prenons l'exemple d'un véhicule autonome : ce type de véhicule doit pouvoir prendre des décisions à la fois précises et robustes – puisqu'elles sont d'un enjeu vital – avec une réactivité en temps réel – à l'échelle de la milliseconde. Or, il n'est pas possible d'équiper chaque véhicule avec un super-calculateur : les processeurs, conçus pour ce type d'application, sont au contraire de puissance limitée, afin de respecter les contraintes, en terme de consommation énergétique, de taille, de prix et de dégagement thermique, qui sont inhérentes à ce type de contexte applicatif. Cela signifie que l'on est forcément limité, dans les types de réseaux de neurone que l'on peut utiliser, d'autant plus que la complexité de ceux-ci tend à se corréler avec leur performance : on se retrouve à devoir choisir entre précision (gros réseaux coûteux) et latence (petits réseaux peu performants), alors que l'application requiert d'avoir les deux en même temps.

Heureusement, la performance des réseaux de neurones n'est pas juste une question d'augmenter leur complexité indéfiniment, et d'ensuite espérer que leur précision suive : les réseaux se construisent par assemblage d'opérations plus élémentaires, appelées « couches », si bien que l'on parle d'architectures de réseaux de neurones ; et différentes architectures peuvent, pour la même complexité, donner des performances différentes. Il y a donc une question d'efficacité des réseaux de neurones, avec l'enjeu de trouver les architectures permettant de satisfaire les deux contraintes de latence et de performance à la fois.

C'est la raison pour laquelle toute une partie de la littérature s'est dédiée à ce qu'on

appelle la compression de réseaux de neurones, qui consiste à réduire le coût d'architectures existantes sans en dégrader les performances. Une méthode notable de cette littérature est l'élagage, qui consiste à retirer des parties jugées inutiles de réseaux. C'est sur cette méthode que s'est focalisée cette thèse, qui fut l'occasion d'en étudier en profondeur la littérature dédiée et de proposer quelques contributions originales.

Lors de cette thèse, nous avons notamment pu explorer différentes problématiques, plus ou moins théoriques ou pratiques, qui s'étendent des fondements mathématiques de l'élagage – et leurs éventuels soucis – jusqu'à l'implémentation de réseaux élagués sur matériel embarqué. C'est cette approche globale qui a permis, non seulement de proposer des contributions, mais aussi de construire un discours original, exposant la nécessité de répondre à certaines questions théoriques pour résoudre des problèmes matériels, comme l'efficacité énergétique des architectures élaguées.

Ce manuscrit est alors structuré, après un chapitre introduisant des notions générales d'apprentissage profond, autour des aspects identifiés comme clefs dans l'élagage : les critères, les méthodes et les structures, avant de s'intéresser aux questions matérielles et de conclure.

Vision par ordinateur, apprentissage profond et compression

La vision par ordinateur regroupe les tâches d'analyse et de traitement des images par ordinateur ; on peut y distinguer plusieurs types de problèmes, comme la classification d'image ou la segmentation sémantique. La classification d'image revient à attribuer à une image, centrée sur un objet, la probabilité d'appartenir à chacune des classes d'une liste prédéfinie – on peut voir ce problème comme une fonction, prenant une image en entrée et renvoyant un vecteur en sortie. Les ensembles de données, que nous avons utilisés pour la classification, sont ImageNet ILSVRC2012 et CIFAR-10 ; comme métriques de performance, nous avons utilisé les précisions Top-1 et Top-5. La segmentation sémantique revient à faire une classification pixel par pixel, de sorte à détourer les objets de classes différentes, dans une image comportant plusieurs objets – on peut voir ça comme une fonction prenant une image en entrée et renvoyant une image (ou tenseur) en sortie. Pour la segmentation sémantique, nous avons utilisé l'ensemble de données Cityscapes, et comme métrique la mIoU (*mean Intersection over Union*).

Les réseaux de neurones, utilisés pour résoudre les tâches de vision par ordinateur, sont en majorité des réseaux à convolutions. Ce type de réseaux, à l'instar des ResNet pour la classification ou des HRNet pour la segmentation, est composé des types d'opérations suivants :

- Convolution : couches appliquant, à chacun des canaux d'une image d'entrée, différents noyaux de convolution ; ces noyaux sont organisés en filtres, si bien qu'une couche comporte autant de filtre qu'elle produit de canaux en sortie, et chaque filtre contient autant de noyaux qu'il y a de canaux en entrée.
- Linéaire : produit matriciel entre un vecteur d'entrée et une matrice de paramètres, pour obtenir un vecteur de sortie ; le plus souvent utilisé à la fin d'un réseau de classification, justement pour opérer la classification après avoir projeté les données d'entrée dans un espace, si possible, linéairement séparable.
- Batch-normalisation : normalise chaque canal, indépendamment, afin d'obtenir – pour l'ensemble de données entier – une distribution centrée réduite ; ensuite une opération affine est appliquée afin d'excentrer de nouveau la distribution, de façon spécifiquement apprise, cette fois-ci.

- ReLU : un type de fonction d'activation très simple ; ces fonctions servent à introduire des non-linéarités dans le réseau, qui sont nécessaires pour son expressivité – sans quoi il serait simplement une fonction totalement linéaire.
- Additions : dans certains types de structures, à l'intérieur des réseaux, les sorties de plusieurs couches différentes sont sommées ensembles ; typiquement, à la sortie de chaque bloc résiduel – qui sont des éléments caractéristiques des ResNet –, la sortie d'un bloc est sommée avec son entrée, ce que l'on appelle une connexion résiduelle.
- Pooling : étape de réduction de la résolution d'une image/tenseur, que ce soit en sélectionnant les valeurs les plus grandes ou en les moyennant ; on en trouve typiquement en début de réseau, pour réduire la taille des images, et à la fin, avant la couche linéaire de classification, pour réduire les tenseurs en vecteurs.

Ces réseaux sont entraînés à l'aide de la méthode de descente stochastique de gradient. La descente de gradient consiste à chercher à minimiser une fonction de perte $\mathcal{L}$ en mettant à jour les paramètres $\mathbf{w}$ du réseau, de façon itérative, selon la valeur de leur dérivée $\frac{\delta \mathcal{L}}{\delta \mathbf{w}}$. La version stochastique, de la descente de gradient, consiste à faire chaque itération sur un sous-ensemble de l'ensemble de données sur lequel entraîner. Classiquement, lors de l'apprentissage, on rajoute à $\mathcal{L}$ un terme de régularisation, appelé *weight-decay*, qui applique une pénalité $\mu \|\mathbf{w}\|_2$ à la magnitude des paramètres.

Il y a plusieurs types de méthodes pour compresser ce type de réseaux ; par exemple :

- La quantification, qui réduit la précision à laquelle sont représentés les paramètres du réseau, ce qui réduit leur empreinte mémoire et la complexité des opérations dans lesquels ils sont impliqués.
- La distillation, qui améliore la performance d'un réseau plus petit en l'entraînant à l'aide d'un réseau plus grand et performant.
- Le *clustering*, qui vise à trouver une représentation qui permette de reconstruire les valeurs des paramètres d'un réseau, sans avoir à les stocker tels quels.

Celle sur laquelle cette thèse se focalise est l'élagage, qui consiste à retirer des parties jugées inutiles d'un réseau. Trois aspects principaux sont importants dans l'élagage :

- Le critère d'élagage, qui sert à identifier les parties inutiles d'un réseau.
- La méthode d'élagage, qui définit la façon dont effectivement retirer ces parties.
- La structure d'élagage, qui définit quel type de partie, quelle granularité élaguer dans le réseau.

Critères d'élagage

Tout d'abord, considérons comment caractériser l'importance de paramètres isolés, pour déduire s'il est pertinent de les élaguer. La littérature a principalement proposé deux façon de caractériser l'importance d'un paramètre. La première consiste à approximer la différence, dans la fonction $\mathcal{L}$, induite par la suppression d'un paramètre w_i par la dérivée de $\mathcal{L}$ en ce même paramètre, soit $|\mathcal{L}_{w_i=0} - \mathcal{L}| \approx \frac{\delta \mathcal{L}}{\delta w_i}$. Le problème, avec cette métrique, est qu'elle sous-entend une condition de proximité, de sorte que la différence Δ dans la valeur des paramètres, introduite par la mise de w_i à 0, soit assimilable à une différence infinitésimale δ ; le fait que cette approximation soit obtenue par développement de Taylor, dans la plupart des démonstrations motivant ce critère, confirme

d'autant plus cette condition passée sous silence.

La seconde méthode consiste à simplement retirer les paramètres les plus petits en valeur absolue. Ce critère est beaucoup plus trivial à justifier théoriquement : les paramètres les plus petits sont ceux qui, en le mettant à zéro, introduisent globalement la moindre différence dans les valeurs des paramètres en $\mathbf{w}$. La simplicité de cette mesure – beaucoup plus simple que de devoir calculer le gradient – et la robustesse de la justification théorique – qui, bien que simpliste, ne souffre pas de conditions cachées, rendant le critère faux dès que l'on dépasse une certaine valeur des paramètres – explique en grande partie la popularité de ce critère.

Utiliser ce type de critère suffit dans le cas de l'élagage de paramètres isolés ; toutefois, dans le cas de l'élagage « structuré » d'éléments plus larges, comportant plusieurs paramètres, il faut trouver des solutions pour généraliser ces critères. Une façon simple est de considérer la norme de la métrique sur l'ensemble de paramètres à élaguer : par exemple, dans le cas de l'élagage d'un filtre de convolution sur la base de la magnitude des paramètres, on peut élaguer les filtres dont la norme de l'ensemble des paramètres est la plus petite. Le problème avec cette méthode est qu'elle ne fonctionne qu'à condition que les filtres à comparer comportent le même nombre de paramètres – ce qui n'est plus vrai entre des couches différentes – et que la distribution des valeurs suive une même loi – sans quoi les valeurs de leurs normes ne sont plus comparables.

C'est pourquoi un critère populaire, en élagage structuré de filtres, qui permet de résoudre certains de ces problèmes, est la magnitude du coefficient multiplicatif de la partie affine des couches de batch-normalisation. Cette technique permet de caractériser l'importance d'un canal entier (filtre de convolution de la couche précédente et noyaux de convolution dans chaque filtre de la couche suivante) sur la base de la valeur d'un seul paramètre ; de plus, la normalisation préalable, de chaque canal en entrée de cette partie affine, permet de rendre les valeurs de ces paramètres plus facilement comparables entre elles ou d'une couche à l'autre.

Enfin, le dernier aspect important des critères concerne le moment, relativement à l'entraînement, de les appliquer. Cela n'a, en effet, pas beaucoup de sens d'élaguer un paramètre, sur la base de sa valeur, alors que le réseau n'est pas encore entraîné ou n'a pas fini de converger. Et pourtant, bien des méthodes implique d'élaguer au cours de l'entraînement, ou au moins de faire un ré-entraînement post-élagage.

Ce type de problématique nous a amené à réfléchir à l'élagage sous un autre angle. Considérons l'évolution de $\mathcal{L}$ selon les différentes valeurs de $\mathbf{w}$ possibles. Après entraînement, supposons que l'on se situe à l'optimum global, atteint pour une valeur particulière de $\mathbf{w}$. Mettre un paramètre à zéro revient à se déplacer en un nouveau point, en lequel les paramètres ne sont pas garantis de demeurer, relativement, des minima locaux : un même paramètre peut se situer à un minimum avant élagage, et ensuite devenir, pour la même valeur, un maximum local. On se rend alors compte que, dans l'absolu, la question de l'élagage ne devrait pas être de considérer juste les propriétés des valeurs des paramètres à l'optimum, pour savoir lesquels élaguer ; on devrait plutôt rechercher quelle est la projection de $\mathcal{L}$ permettant le meilleur optimum possible (après ré-entraînement) selon la quantité de paramètres à maintenir à zéro, et ce peu importe si, sur le moment, les paramètres à retirer sont contre-intuitifs. Bien qu'un tel paradigme semble difficile à mettre en œuvre, cette façon de voir permet de mettre en évidence quels sont les problèmes inhérents à l'élagage, et surtout au déplacement et à la projection qu'il implique.

Méthodes d'élagage

La méthode la plus classique d'élagage suit un principe très simple : le réseau est entraîné jusqu'à convergence, puis une portion des paramètres est élaguée, et enfin le réseau reçoit quelques cycles supplémentaires d'entraînement ; ces deux dernières étapes peuvent être itérées jusqu'à atteindre le taux d'élagage souhaité.

Bien que simple et suffisamment efficace pour demeurer compétitive encore aujourd'hui, cette méthode souffre de plusieurs problèmes théoriques, que notre représentation de l'élagage, sous forme de déplacement et de projection, met bien en évidence. En effet, ce type d'élagage revient à éliminer entièrement une proportion de paramètres, ce qui revient à se déplacer abruptement dans $\mathcal{L}$, et ce, de manière définitive, de sorte que l'on se limite sans retour possible à la projection obtenue par l'élagage.

Non seulement ce type d'altération de la trajectoire de $\mathbf{w}$ dans $\mathcal{L}$ brise les garanties mathématiques du bon fonctionnement de la descente de gradient, mais, de plus, nous avons déjà évoqué que baser l'élagage simplement sur les valeurs contenues dans $\mathbf{w}$ amenait potentiellement à ne pas cibler les paramètres les plus pertinents. Le caractère itératif de la méthode classique vise à atténuer ce problème, en élaguant moins de paramètres à la fois – mais toujours de façon complète, ce qui ne l'élimine donc pas.

La littérature a cherché bien des moyens de palier ces problèmes : élagage graduel tout au long de l'entraînement, système de repousse des paramètres, apprentissage automatique de masques d'élagage, etc. De très nombreuses méthodes proposent des alternatives pour résoudre chacun des problèmes évoqués ci-dessus. Toutefois, chacune d'entre elles persiste à élaguer les paramètres de façon abrupte, en faisant passer leur valeur directement à zéro. C'est pourquoi une dernière famille de méthodes a attiré notre attention, qui consiste à relaxer l'élagage sous la forme d'une pénalité, inclue dans la fonction de perte, sur la magnitude des paramètres, de sorte à faire décroître progressivement leur valeur, au lieu de les élaguer manuellement.

Cela nous a conduit à proposer une méthode originale, *Selective Weight Decay* (alias SWD), qui, pour n'importe quel critère et structure permettant de désigner un sous-ensemble $\mathbf{w}^*$ à élaguer, peut être définit comme l'ajout d'un troisième terme à $\mathcal{L}$, de sorte à obtenir :

$$\mathcal{L} = \underbrace{\mathcal{F}(\mathcal{N}(\mathbf{X}), \mathbf{Y})}_{\text{Terme d'erreur}} + \underbrace{\mu \left\|\mathbf{w}\right\|_2}_{\text{Weight Decay}} + \underbrace{a\mu \left\|\mathbf{w}^*\right\|_2}_{\text{SWD}},$$

avec :

$$a(s) = a_{début} \left(\frac{a_{fin}}{a_{début}} \right)^{\frac{s}{s_{final}}},$$

de sorte que ce coefficient d'importance croisse de façon exponentielle, passant d'une valeur $a_{début}$ en début d'entraînement à une valeur a_{fin} quand celui s'achève.

Nous avons pu tester cette méthode et obtenir des résultats prometteurs, en classification sur ImageNet ILSVRC2012 et CIFAR-10. Notamment, si toutes les méthodes d'élagage semble étrangement converger pour les taux d'élagage les plus bas, SWD se démarque fortement pour les taux les plus intenses, pour lesquels les méthodes de référence semblent produire un phénomène de *layer collapse*, qui provoque l'effondrement de la performance des réseaux. Nous avons également proposé une recherche par grille des paramètres $a_{début}$ et a_{fin} à choisir ; cette recherche nous a permis de confirmer que le régime, permettant les meilleures performances finales (ainsi qu'un réel élagage par la seule force de la pénalisation), était bien celui conforme à notre intuition : commencer avec une valeur de a faible, pour laisser le réseau apprendre, et finir sur une valeur très forte, afin d'écraser les paramètres ciblés.

Structures d'élagage

La structure d'élagage a une importance déterminante sur le rapport entre nombre de paramètres et performance – les granularités plus fines permettant, généralement, un meilleur rapport –, ainsi que sur la facilité d'implémentation sur matériel – les granularités plus grossières étant plus simples à exploiter – de même que sur les types de coûts réduits par l'élagage. Si la littérature tend souvent à résumer la question de la compression au rapport entre nombre de paramètres et performance, il ne s'agit pas d'une approximation très satisfaisante, puisqu'elle ne tient compte : 1) ni d'à quel point le taux de compression, en terme de paramètres, se répercute sur le nombre d'opérations exécutées par le processeur, ni sur l'empreinte mémoire réelle, 2) ni de l'empreinte mémoire des représentations intermédiaires, c'est à dire des produits intermédiaires en sortie de chacune des couches.

Passons en revue quelques exemples de structures :

- L'élagage « non-structuré » de paramètres est la granularité la plus fine et libre possible, mais elle est difficile à exploiter sur matériel (en particulier sur carte graphique, plus adaptées aux multiplications matricielles denses) et ne réduit pas les dimensions des couches, ce qui ne change donc pas la taille des représentations intermédiaires. Cela signifie qu'en absence d'implémentation adaptée, ce type d'élagage ne fait qu'introduire des zéros dans les paramètres, sans réellement réduire ni l'empreinte mémoire, ni le nombre d'opérations.
- Élaguer le même paramètre dans le même noyau de chaque filtre d'une couche de convolution permet de tirer partie de l'algorithme *im2col*, fréquemment utilisé pour executer les convolutions sous forme de multiplication matricielle : cette méthode implique de réarranger les paramètres de la couche sous forme d'une matrice ; or, dans cette matrice, faire ce type d'élagage peut revenir à élaguer des colonnes de cette matrice, ce qui allège à la fois la matrice des poids et celles des données sur lesquelles appliquer la convolution. Ce type d'élagage permet donc de réduire l'empreinte mémoire et le nombre d'opérations.
- L'élagage « structuré » de filtres de convolution a le grand avantage de se répercuter directement sur l'architecture du réseau et les dimensions de la couche : il y a donc, sans équivoque, réduction du nombre d'opérations et de l'empreinte mémoire des paramètres et des représentations intermédiaires, dont on réduit le nombre de canaux. De plus, ce type d'élagage est a priori trivial à exploiter, puisque c'est l'architecture elle-même du réseau qui est élaguée.

C'est pour cette raison que, dans nos travaux concernant le matériel embarqué, nous avons choisi de nous consacrer à l'élagage de filtres (alias « élagage structuré »). Toutefois, cela nous a confronté à un problème, rarement évoqué, de ce type d'élagage : les interdépendances dimensionnelles entre couches. En effet, si deux couches de convolution se suivent, alors élaguer un filtre dans la première produit un canal de moins en sortie, si bien que la couche suivante doit recevoir un canal de moins en entrée, ce qui requiert d'élaguer le noyau correspondant dans chacun de ses filtres. Ne pas faire attention à faire correspondre les dimensions des couches ne peut que l'empêcher de fonctionner. Cet exemple est très simple à résoudre, et ne pose pas spécialement soucis. Cependant, les blocs résiduels, que l'on trouve aussi bien dans les ResNet que dans les HRNet, contiennent une opération d'addition qui implique que les deux tenseurs à sommer soient élagués exactement de la même manière, sans quoi, quand bien même ils seraient de la même dimension, l'addition sommerait ensemble des canaux qui ne l'étaient pas auparavant. Combiné avec le principe de connexions résiduelles, ce pro-

blème fait que les réseaux de type ResNet peuvent contenir des dépendances longue-portée entre des couches au travers de tout le réseau. Se préoccuper à élaguer de la même manière, de part et d'autre des additions, amène donc à sévèrement contraindre l'élagage et à laisser très peu de degrés de liberté – c'est ce que fait la majorité de la littérature.

Exploiter l'élagage sur matériel embarqué

Afin de se libérer des contraintes, imposées par les problèmes d'interdépendances entre les couches que nous avons déjà évoqués, et de mesurer l'efficacité énergétique de réseaux élagués bien plus librement, nous avons élaboré une solution permettant d'exploiter n'importe quelle distribution de l'élagage structuré dans les ResNet ou HRNet. Cette solution s'appelle *Dimensional Clear Out* (alias DCO) et peut être divisée en deux parties.

La première partie de DCO consiste en une méthode d'identification des indépendances entre couches suite à l'élagage. En effet, du fait des interdépendances longue portée et des additions, si les couches concernées sont élaguées différemment, alors retrouver le schéma précis des canaux devant être additionnés, concaténés ou retirés, demande de tenir compte de toute la combinatoire en amont ; or, ce schéma est nécessaire, non seulement pour préserver la fonction du réseau après réduction de l'architecture, mais également pour ajuster correctement les dimensions d'entrées des couches suivantes. Afin d'effectuer cette identification de façon automatique, nous préparons d'abord une copie du réseau à élaguer, que nous transformons en réseau entièrement linéarisé, donc sans fonction d'activation, ni biais, ni normalisation, et dont les paramètres sont tous mis à une valeur de un. Ensuite, les paramètres à élaguer sont mis à zéro, et l'on peut alors faire une inférence et calculer la dérivée du résultat obtenu. Du fait des transformations du réseau, si un paramètre a un gradient nul, c'est qu'il est nécessairement élagué ou déconnecté en amont ou en aval, ce qui permet alors de réajuster, de façon fiable, les dimensions de toutes les couches du réseau.

La seconde partie de DCO consiste à remplacer les additions par un opérateur sur mesure, appelé « indexation-addition », qui consiste en une addition chez laquelle on peut préciser les canaux à sommer ensemble ou à concaténer, de sorte à pouvoir être appliqué à des tenseurs de dimensions différentes ou ayant été élagués différemment. Pour créer automatiquement ces opérateurs, nous insérons des couches provisoires dans le réseau, que nous élaguons ensuite en utilisant la première partie de DCO ; cela nous permet d'obtenir automatiquement les schéma de connexions entre canaux, et ainsi de produire immédiatement les indexations-additions adaptées. Ainsi, en assemblant les deux parties de DCO, nous sommes à même de prendre un ResNet ou un HRNet, dont des filtres auraient été élagués sans aucune précaution concernant les dépendances, et tout de même de l'exploiter matériellement – au lieu de simplement laisser les paramètres élagués à zéro sans les retirer.

Ce sont donc des réseaux HRNet, pour la segmentation sémantique sur Cityscapes, que nous avons élagués, en utilisant DCO, et dont on a pu mesurer la consommation énergétique et la latence sur NVIDIA Jetson AGX Xavier, en utilisant TensorRT pour l'optimisation et l'inférence. Nous avons comparé des HRNet-48 élagués avec des HRNet-32 et 18 non-élagués ; le coefficient donne la largeur de base, en canaux, pour toutes les couches du réseau, si bien que pour un HRNet-18 de même architecture et profondeur, toutes les couches sont moins larges que celles de HRNet-48 par une même proportion. Pour l'élagage, nous avons utilisés deux méthodes différentes (dont SWD) avec

le même critère (coefficient de batch-normalisation) et la même structure (filtres avec DCO).

Nos résultats se sont révélés étonnants. Lorsque l'on ne considère que le rapport entre paramètres et performance, l'élagage semble à première vue donner de meilleures performances que les références non-élaguées, pour le même nombre de paramètres ; cependant, lorsque l'on considère les opérations ou la consommation énergétique, le rapport s'inverse : les réseaux élagués consomment plus que les références, alors qu'ils contiennent moins de paramètres. Après investigation, nous nous sommes rendu compte que cela venait de la distribution de l'élagage, qui tendait à cibler des couches dans lesquelles un même nombre de paramètres était impliqué dans moins d'opérations qu'en moyenne – ce qui produit donc des réseaux pauvres en paramètres, mais denses en opérations. L'étrangeté de la chose provient non seulement du fait qu'un tel biais n'était a priori pas prévisible dans la définition du critère d'élagage, mais également du fait que SWD, tout en donnant de meilleures performances que l'autre méthode, produisait également des réseaux plus pauvres en opérations – ce qui tend à invalider l'hypothèse selon laquelle l'élagage ciblerait les paramètres pauvres en opération parce que ce sont ceux qui dégraderaient le moins les performances.

Conclusion

Notre présentation de l'élagage, comme l'assemblage de trois aspects relativement distincts, et nos contributions – SWD, DCO et nos mesures de consommation énergétique – nous on permis de montrer que, non seulement l'élagage est loin d'être aussi simple que son principe intuitif laisserait supposer, mais qu'il s'y trouve également des questions théoriques, auxquelles il est nécessaire de répondre afin d'obtenir un élagage qui puisse être réellement efficace.

En effet, si l'on se base sur le résultat de nos expériences, alors l'élagage est saboté par un biais, dans le critère d'élagage, qui conduit à produire des architectures inefficaces ; il est alors nécessaire de comprendre les raisons théoriques de ce biais, afin éventuellement de le résoudre. Cet exemple est d'autant plus frappant que le problème ne s'expose qu'une fois que l'on se confronte aux problématiques matérielles, puisqu'il demeure insoupçonné tant qu'on ne réfléchit qu'en terme de rapport entre nombre de paramètres non-nuls et précision d'un réseau. De même, sur un plan plus fondamental, concevoir l'élagage comme un déplacement et une projection dans $\mathcal{L}$ met en évidence les limites de l'élagage basé sur la valeurs des paramètres – donc simplement sur $\mathbf{w}$ et sa stricte proximité, sans considérer $\mathcal{L}$ de façon globale.

Si ces observations tendent à montrer que l'élagage n'est pas garanti d'être une méthode de compression viable en l'état, elles montrent également que, non seulement il demeure énormément de pistes d'ouvertures et de choses à approfondir à l'avenir dans la littérature, mais aussi que l'élagage touche à des aspects très fondamentaux des réseaux de neurones et leur entraînement, ce qui en fait un sujet d'étude intéressant à l'échelle de l'apprentissage automatique profond en général.

Résumé

Depuis près d'une décennie, l'état de l'art en vision par ordinateur est détenu par les réseaux de neurones profonds et, plus particulièrement, par les réseaux à convolution. Que ce soit en classification, en segmentation sémantique ou en détection d'objet, les réseaux de neurones sont désormais une référence absolue. C'est pourquoi ces réseaux sont devenu indispensables dans de nombreuses applications industrielles, comme la conduite autonome. De fait, de nombreuses entreprises, à l'instar de Stellantis, se mettent à montrer l'ambition de faire de l'intelligence artificielle une part significative de leur activité.

Malheureusement, la performance impressionnante de ces réseaux de neurones a un coût : celui d'une complexité algorithmique, d'une empreinte mémoire et d'une consommation énergétique qui ne sont pas viables sur matériel embarqué, tel que celui que l'on peut trouver sur des véhicules et dont la puissance de calcul est limitée. Il n'est pas aisé de réduire ces coûts, puisque cela signifie généralement réduire la performance des réseaux ; c'est pourquoi une part conséquente de la littérature s'est consacrée à concevoir des réseaux, ou des méthodes de conception, plus efficaces et permettant un bien meilleurs compromis entre performance, robustesse, consommation énergétique, occupation mémoire, latence et dissipation thermique. C'est ce domaine du *deep learning* (apprentissage profond) que l'on nomme « compression de réseaux de neurones ».

Ce champ de recherche compte plusieurs familles de méthodes, dont l'élagage, qui est particulièrement populaire au sein de la littérature depuis 2015. Son principe de base est de simplifier les réseaux en supprimant des parties jugées inutiles, c'est à dire que l'on peut retirer sans affecter la performance du réseau. C'est la méthode sur laquelle cette thèse s'est concentrée spécifiquement.

Rapidement durant cette thèse, l'élagage s'est révélé être d'une complexité surprenante, que ce soit d'un point de vue théorique ou pratique. Comment prédire l'importance d'un groupe de paramètres, alors qu'il y en a des millions dans un réseau ? Comment les retirer peut-il réduire les coûts du réseau, alors que les processeurs utilisés ne gèrent vraisemblablement pas les multiplication de matrices creuses ? Comment s'y prendre pour retirer les paramètres sans perturber l'entraînement et dégrader les performances ? Toutes ces questions, qui dans un premier temps peuvent ne pas venir immédiatement à l'esprit, structurent en réalité toute la littérature sur l'élagage ; chacune d'entre elles a un impact déterminant sur la capacité de l'élagage à produire des réseaux de neurones effectivement plus efficaces.

Pour répondre à ces questions, nous avons décidé de structurer les chapitres de ce manuscrit autour de ce que nous avons identifié comme étant les dimensions les plus caractéristiques de l'élagage. Après un chapitre introductif et un autre présentant tout le bagage nécessaire à la compréhension du domaines, les quatre chapitres principaux vont respectivement traiter des critères d'élagage, des méthodes d'élimination de paramètres, des structures d'élagage et de l'implémentation des réseaux élagués sur ma-

tériel embarqué ; ces quatre aspects répondent respectivement à ces quatre questions : comment déterminer qu'une partie peut être retirée ; comment la retirer ; quel genres de parties retirer et comment s'assurer que ces élagages aboutissent bien à des gains matériels.

Critères : Nous pouvons trouver de nombreux critères différents, dans la littérature, pour déterminer l'importance d'un paramètre dans un réseau de neurones. Deux d'entre eux sortent du lot : la "saillance", qui cherche à prédire la dégradation de la performance suite à l'élagage d'un paramètre à l'aide de sa dérivée, et la magnitude de la valeur elle-même du paramètre. Malgré les démonstrations théoriques qui, dans la littérature, ont amené à l'élaboration du critère de saillance, les synthétiser dans ce manuscrit a révélé que certaines lacunes cachaient le fait que ce critère ne pouvait fonctionner qu'à condition que les paramètres concernés soient proches de zéro, ce qui le rend redondant avec le critère de magnitude.

En dehors de donner l'importance d'un paramètre seul, plusieurs autres aspects des critères détiennent une importance cruciale ; notamment la généralisation de ces critères à des groupes de paramètres (ce qui est indispensable pour l'élagage structuré), la distribution du taux de l'élagage entre les couches (qui a un impact sur l'efficacité énergétique des réseaux élagués, comme nous le verrons au chapitre dédié à l'implémentation sur matériel embarqué) et la pertinence des critères relativement au moment de leur application durant l'entraînement.

Les discussions de ce chapitre nous ont finalement conduites à proposer une nouvelle façon de présenter l'élagage, que nous n'avons jamais aperçue dans la littérature malgré son évidence : l'élagage peut se résumer à un déplacement et une projection dans le paysage de la fonction de perte $\mathcal{L}$. Une telle façon de voir rend immédiatement évidents certains comportements pourtant initialement contre-intuitifs de l'élagage, comme sa propension à bloquer les réseaux élagués dans des minima locaux ou alors les interactions entre certains paramètres dont l'inter-dépendance n'est pas évidente de prime abord.

Méthodes : Cette façon de présenter l'élagage comme un déplacement a également le mérite de mettre en évidence la raison pour laquelle la méthode « entraînement, élagage et *fine-tuning* » (poursuite de l'entraînement au plus faible taux d'apprentissage) pose problème : supprimer abruptement des paramètres équivaut à un déplacement brusque du point $\mathbf{w}$ dans le paysage de $\mathcal{L}$ au beau milieu de l'entraînement, ce qui ne peut que le perturber. Rendre cette transformation moins abrupte est ce qui sous-tend tacitement toute la littérature consacrée à trouver la meilleure méthode pour supprimer les paramètres identifiés comme dispensables par un critère d'élagage.

Dans ce chapitre, nous présentons une méthode de notre cru pour résoudre ce problème : le Selective Weight Decay [1] (SWD), qui élague des paramètres à l'aide d'une pénalisation sélective et progressive de leur valeur tout au long de l'entraînement. Nos expériences montrent qu'avec le bon choix d'hyperparamètres, SWD est capable d'éliminer les paramètres voulus d'une façon douce, ce qui permet d'atteindre des taux d'élagage très importants sans détruire les réseaux, comme le feraient d'autres méthodes plus classiques.

Structures : L'élagage non-structuré de paramètres indépendants produit des matrices creuses dont beaucoup de matériels peinent à tirer parti. De plus, ce type d'élagage n'est pas à même de réduire, par exemple, l'occupation mémoire des représenta-

tions intermédiaires des données entre chaque couche, alors qu'elle peut excéder celle des paramètres eux-même. C'est pour cette raison que la littérature a exploré d'autres types d'éléments, de structures à élaguer ; la plus notable est le filtre de convolution (l'équivalent d'un neurone entier pour les couches de convolution) du fait de sa simplicité d'implémentation et de son efficacité pour réduire à la fois le nombre de paramètres et d'opération et la taille des produits intermédiaires.

Toutefois, un problème qui est souvent éludé dans la littérature est celui des dépendances dimensionnelles entre couches : élaguer un filtre ne signifie pas seulement produire un canal en moins dans la sortie, mais cela signifie aussi que la couche suivante doit accueillir une entrée comportant également un canal en moins. Ce type de dépendances est beaucoup plus complexe à identifier qu'il n'y paraît à première vue, ce qui a amené toute une partie de la littérature à le contourner en contraignant la distributions des filtres à élaguer. Malheureusement, une telle contrainte réduit les degrés de liberté laissés à l'élagage qui, donc, est moins à même de produire des architectures réellement efficaces.

Matériel : Après une récapitulation de ce qui, au juste, a un coût dans un réseau de neurones du point de vue du matériel, ce chapitre présente notre étude de l'efficacité énergétique des réseaux élagués sur GPU embarqué [2]. Au préalable, nous introduisons le Dimensional Clear-Out [3] (DCO), une solution que nous avons élaborée pour résoudre les problèmes d'interdépendance entre couches et ainsi permettre d'étudier l'efficacité de réseaux élagués librement sans contrainte.

À notre grande surprise, notre étude a révélé un comportement inattendu, de l'élagage global, qui dégrade cruellement l'efficacité énergétique des réseaux élagués : tandis qu'ils contiennent effectivement peu de paramètres tout en gardant une bonne performance, il se trouve en réalité qu'ils comportent beaucoup plus d'opérations en comparaison de réseaux simplement plus petits dès le départ. Cela est dû au critère d'élagage, qui semble viser en priorité les paramètres impliqués dans moins d'opérations que la moyenne ; la chose surprenante, c'est que ce parti-pris n'est pas intentionnel. Ce comportement spontané et majoritairement inexpliqué est toujours à étudier et montre que les questions théoriques portant sur, par exemple, les critères d'élagage, ont bel et bien un impact décisif sur l'aptitude de l'élagage à produire des architectures énergétiquement pertinentes.

Conclusion : Au final, ce manuscrit, par son étude thématique de l'élagage, fût l'occasion de présenter un certain recul et de discuter ce qui, à l'avenir, semble avoir le plus besoin d'être étudié en profondeur dans la littérature de l'élagage – qu'il s'agisse d'une comparaisons méticuleuses et exhaustives de différentes méthodes ou d'une formalisation théorique plus rigoureuse de certains problèmes ayant tendance à être négligés. Il est absolument nécessaire de résoudre ces problèmes, non seulement pour améliorer l'élagage, mais surtout pour déterminer si oui ou non cette méthode est capable de produire des réseaux plus efficaces que des réseaux basiques simplement plus petits et qui peuvent être conçus et entraînés sans recourir à l'élagage.

Malgré le caractère décevant de cette conclusion, notre travail montre surtout que, malgré la popularité et l'effervescence de cette littérature ces dernières années, il y a toujours une importante marge d'amélioration pour proposer des contributions dans ce domaine à l'avenir. De plus, nous avons pu montrer une grande intersection entre certaines aspects subtils de l'élagage et des sujets très fondamentaux du deep learning.

Abstract

For the past decade, the state of the art in computer vision has been held by deep neural networks and, more particularly, convolutional neural networks. Whether it is in classification, semantic segmentation or object detection, neural networks are now an undisputed go-to. Therefore, they are now considered essential for many industrial applications, such as autonomous driving. This is the reason why many companies, such as Stellantis, now show novel ambitions of making artificial intelligence a key part of their future activities.

Unfortunately, the impressive performance of neural networks comes with a cost: that of an algorithmic complexity, a memory footprint and an energy consumption that are not viable on embedded hardware, such as those to be found on vehicles, whose processing power is limited. Reducing such costs is not a trivial issue, as it usually means reducing the performance of networks; this is why a consequent part of the literature is dedicated to designing networks, or ways to produce them, that are more efficient and allow for a better trade-off between performance, robustness, power consumption, memory occupation, latency and thermal dissipation. This domain of deep learning is called neural networks compression.

This field counts multiple families of methods, including pruning, that is especially popular in the literature since 2015. Its basic principle is to simplify networks by removing parts deemed unnecessary, *i.e.* that can be removed without harming the performance of the network. This is the specific family of method that this thesis focused onto.

Quickly during this thesis, pruning proved to hide a surprising complexity, both theoretical and practical: how to predict the actual importance of a group of parameters while there are millions in a network? How can removing such parameters reduce the cost of the network, while processors can possibly not be able to leverage sparse matrix multiplication? How should we remove parameters in order not to disrupt training and harm the performance? All these questions may not immediately come to mind at first, but actually structure the whole literature of pruning and each of them has a crucial impact on the ability of pruning to successfully improve the efficiency of neural networks.

To tackle these questions, we decided to structure the chapters of this manuscript around what we identified to be the main dimensions that characterize pruning: after an introductory chapter and another dedicated to the necessary background to understand the context of the field, the four main chapters tackle respectively the pruning criteria, the removal methods, the pruning structure and the implementation of pruned networks on hardware; these four aspects respectively answers to the four questions: how to tell that a part is unnecessary, so we can remove it; how to effectively remove it; what type of parts to remove and how to make sure the introduced sparsity can effectively be leveraged on hardware.

Criteria: The literature has developed multiple criteria to tell the importance of a parameter in a neural network. Two of them stand out: the saliency criterion, that aims at predicting the degradation of accuracy following pruning by using the value of its derivative, and the magnitude of the value of the parameters themselves. Despite the theoretical attention that the saliency criterion received, it appears that some oversights eluded the fact that this criterion only works for parameters that are close to zero, which makes it redundant with the magnitude criterion.

Besides telling the importance of a single parameter, multiple other aspects hold a crucial importance, notably the generalization of criteria to groups of weights (essential for structured pruning), the distribution of the pruning rate across layers (which has an impact on the energetic efficiency of pruned networks, as seen in the chapter concerning hardware implementation) and the relation between the relevance of criteria and the moment, during training, when they are applied.

The discussions of this chapters finally led us to expose a way to view pruning, that we never saw in the literature despite its simplicity: that of seeing pruning as a displacement in the landscape of the loss function $\mathcal{L}$ and a projection. Such a presentation immediately makes self-evident some otherwise counter-intuitive behaviors of pruning, such as the tendency of pruning to get pruned networks stuck in a local minimum or interactions between weights whose interdependence may not be obvious at first.

Methods: This way of seeing pruning as a displacement also highlights why the regular train/prune/fine-tune method is problematic: it sums up as suddenly changing the position of the point $\mathbf{w}$ in the landscape of $\mathcal{L}$ during the training process, which disrupts it. Making this transformation less abrupt is tacitly the motivation behind the literature dedicated to finding the best way to remove parameters, once they have been identified as removable by a pruning criterion.

In this chapter, we expose our own proposal to solve this issue: Selective Weight Decay [1] (SWD), that prunes parameters through a progressively increasing penalization of the value of the specific weights to remove all throughout training. Our experiments show that, with the right choice of hyper-parameters, SWD can effectively remove parameters in a smooth way that allows reaching very high pruning rates at which more classical methods lead to the destruction of neural networks.

Structures: Pruning independently parameters in an unstructured manner produces sparse tensors that tend to be poorly handled by many hardware. On top of that, such a type of sparsity can hardly reduce, for example, the size of intermediate representations of data between each layer, while the memory footprint of such representations can exceed that of parameters. This is why the literature developed multiple types of pruning structures (*i.e.* the types, or the granularity, of the elements to remove from a network), the most notable of them being convolution filters (the equivalent of whole neurons for convolution layers) because of how simple such a pruning is to leverage and how it successfully reduces the number of parameters and operations and the size of intermediate representations.

However, one problem that is often eluded in the literature is the dimensional interdependencies between layers: pruning a filter produces one fewer channel in the output, but thus the following layer must take one fewer channel as an input. Such dependencies are much tougher to identify than it may seem at first, which leaded the literature to circumvent the whole problem by constraining the distribution of pruned filters. Unfortunately, such constraints reduce the degrees of freedom, and therefore the potential

efficiency, of pruned network architectures.

Hardware: After a recapitulation of what, exactly, has a cost on hardware in neural networks, this chapter shows our study of the energetic efficiency of pruned networks on embedded GPU [2]. Beforehand, we introduce Dimensional Clear-Out [3] (DCO), a solution we developed to tackle the problem of dimensional dependencies between layers, which allowed us to study the efficiency of networks that could be pruned without constraints.

Unfortunately, our study highlighted a counter-intuitive behavior of global pruning, that critically harms the energetic efficiency of pruned networks: these pruned networks contain very few parameters while keeping a good performance, which makes them seem better than simply smaller networks trained from scratch, but they actually turn out to contain many more operations than them, because the used pruning criterion tended to aim at parameters involved in fewer operations on average—while such a bias was completely unintentional. This spontaneous and mostly unexplained behavior is still to investigate and show that theoretical questions surrounding, for example, pruning criteria, have indeed a decisive impact on the ability of pruning to produce architectures that are more efficient than simply smaller baseline networks.

Conclusion: Finally, this manuscript, through its thematic review of pruning, provided hindsight and discussions that point out multiple critical aspects to investigate in the future, whether they are the need for a thorough comparison between a range of different methods or that of a rigorous theoretical formalization of sometimes overlooked problems. Solving these problems looks to be an absolute necessity, not just to improve pruning, but to actually figure out whether or not it is able at all to produce networks that are more efficient than trivial smaller networks, that can be designed and trained from scratch without the need for a specific method such as pruning.

Despite this somewhat anti-climatic conclusion, our works shows above all that, despite its popularity, the field of pruning still has room for progress and many contributions in the future. Moreover, we showed that these subtle aspects of pruning actually encompass some fundamental issues that overlap with many other domains of deep learning.

Chapter 1

Introduction

Contents

Preamble

This manuscript is the result of a CIFRE (Convention Industrielle de Formation par la REcherche)[1] thesis realized in collaboration between the Stellantis[2] multinational automotive manufactoring corporation and both the IMT Atlantique[3] technological university (Grande École) and the Lab-STICC laboratory[4], more particularly the BRAIn team[5], and has been funded by the ANRT (Association Nationale Recherche Technologie)[6]. It birthed out of the interest of Stellantis for vehicle automation and the expertise of IMT Atlantique in the domain of deep learning and compression of deep neural networks [4].

Originally intended to focus on the compression, in general, of convolutional neural networks (CNN)—that are ubiquitous in the domain of computer vision, which is of particular interest for vehicle automation—, this thesis quickly focused specifically on the domain of neural networks pruning, in order to allow for a much deeper investigation and more relevant hindsight encompassing from the most theoretical aspects of pruning to the stakes of its actual implementation on embedded hardware. This chapter will expose in details the motivations behind this thesis, which stakes underline its research scope and how our work responds to its underlying questions.

1.1 Why Work on Convolutional Neural Networks?

1.1.1 Performance of CNNs

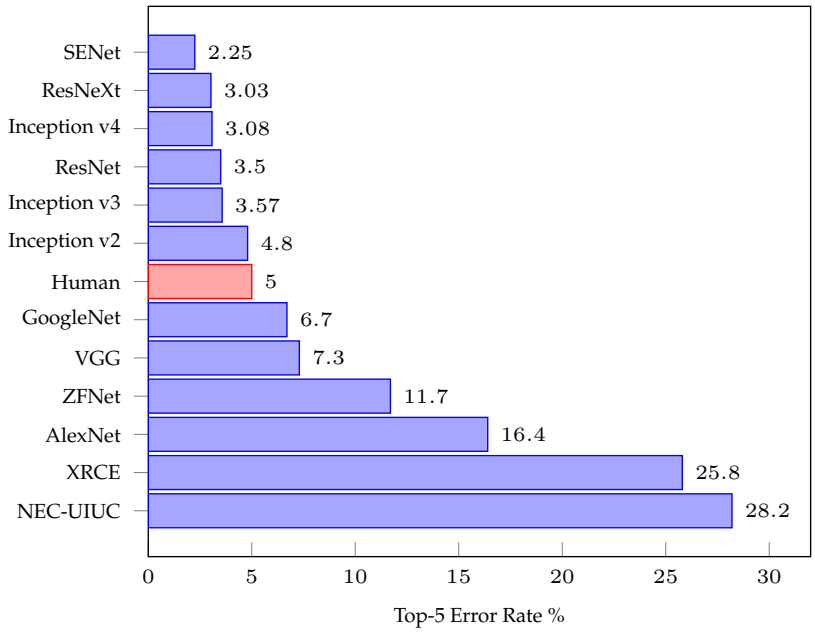


Figure 1: Classification performance of CNNs and humans on ImageNet ILSVRC2012. Figure directly reproduced from Alzubaidi *et al.* [5].

Deep learning 's history is not one of a direct and immediate success: originally theo-

[1] www.enseignementsup-recherche.gouv.fr/fr/les-cifre-46510
[2] www.stellantis.com/fr
[3] www.imt-atlantique.fr/fr
[4] labsticc.fr/fr
[5] labsticc.fr/en/teams/brain
[6] www.anrt.asso.fr/fr

rized in the 50's [6], it had a first rise in the 80's-90's, with notably the birth of the first convolutional neural networks [7]. However, the lack of computation power (as well as perceived theoretical limitations [8]) put a stop to the development of this research domain and lead to the drought of the 2000's. The breakthrough of AlexNet [9] on the ImageNet contest in 2012 put back deep learning under the spotlight and showed that, thanks to the computing power of newly-popularized Graphics Processing Unit (GPU), neural networks, mostly using matrix multiplications, could become big enough to yield competitive performance while being trained under acceptable delays—largely because stochastic gradient descent, back-propagation, batch-computing and the parallelization power of GPUs [10] go together well. Since then, convolutional neural networks managed to outclass humans at image classification [5], as illustrated in Figure 1.

Countless papers have now been published in the field and deep learning is now ubiquitous in computer vision [11], audio processing [12], language processing [13], image generation [14] and many other domains. Therefore, neural networks are considered a staple in such fields and, since image classification is basically the prototypical use-case of deep learning, convolutional neural networks are especially notorious and widespread in the literature. The fact is: many domains of deep learning, especially compression [15], use such networks as their default test subject.

This ability of deep neural networks to solve tasks that are seemingly impossible for humans [16] and to always push further their performance in many contests such as ImageNet tend to bias a large part of the literature toward considering only the raw performance of networks at solving the task (for example, Top-1 accuracy on image classification). However, such a philosophy quickly reaches its limits in some domains.

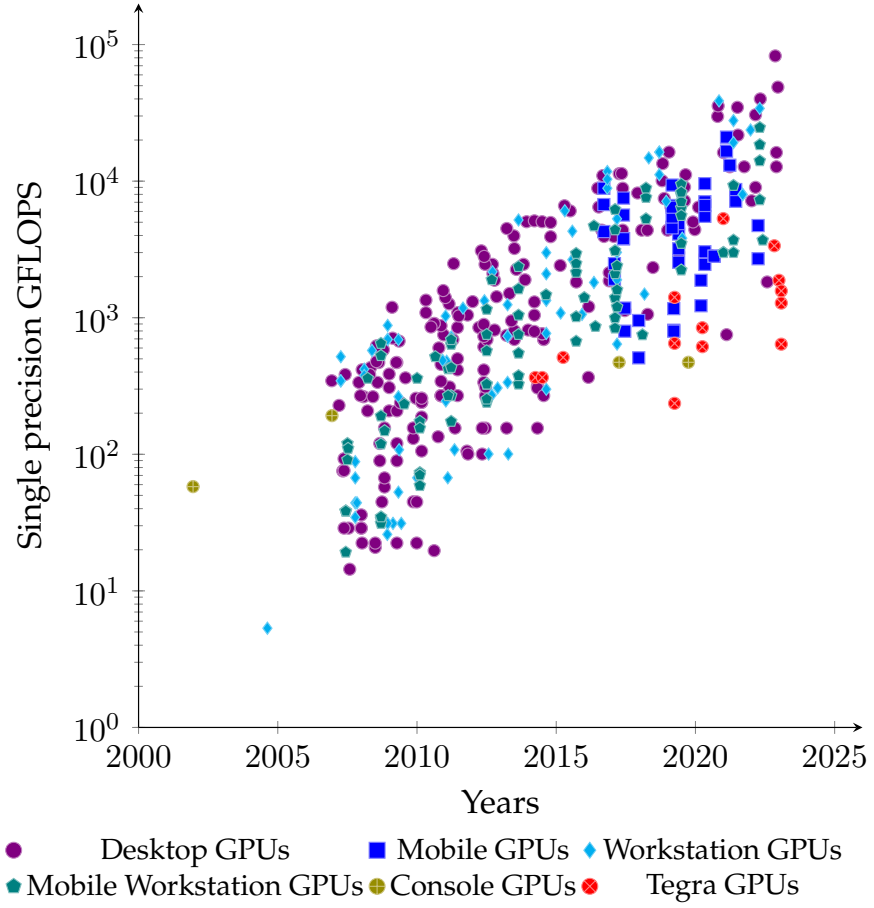
1.1.2 In the Wake of the Race for Performance

As previously pointed out, what initially hindered and then lead to the explosive expansion of deep learning is the availability of sufficient computation power. Figure 2a shows the increase in computation power of NVIDIA GPUs over the years—these GPUs are omnipresent in the field, because of libraries such as CUDA[7] and cuDNN [17]. Figure 2b shows the increase in the number of parameters of classification networks on the ImageNet ILSVRC2012 [18] dataset.

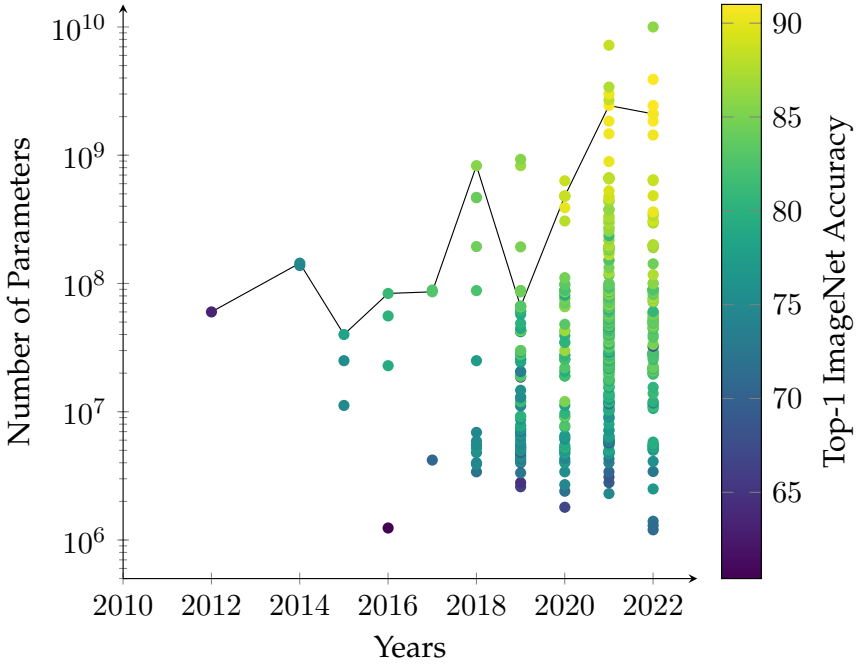
In 2012, the best GPU was the GeForce GTX 680 desktop GPU with a computation power of 3090.43 GFLOPS; in 2022, the GeForce RTX 4090 is estimated at a theoretical power of 82.6 TFLOPS, which is roughly $26.7\times$ the power of the best GPU in 2012. In 2012, the best network on ImageNet was AlexNet [9], with an accuracy of 63.0% and counting 60M parameters; in 2022 it is CoCa [19], with an accuracy of 91% and 2.1G parameters, which is $350\times$ the size of AlexNet.

Of course, the raw power of GPUs is not the only determining factor to the size of CNNs, because: 1) the biggest networks are often designed by companies such as Google [19], that have access to huge clusters of dedicated Tensor Processing Units (TPU), which most laboratories do not have unlimited or free access to and 2) between 2012 and 2022, many laboratories made investments and have equipped themselves with GPU clusters, so that the size of the networks trained in such laboratories mostly scale with the number of GPUs they accumulated over the years (as long as these GPUs are not too outdated to train modern networks on them). Figure 3 shows the evolution of the computation power of supercomputers over the years: in 2012, the Titan

[7] `developer.nvidia.com/cuda-downloads`



(a) Evolution of the computation power of NVIDIA GPUs over the years.



(b) Evolution of the size and accuracy of CNNs over the years. The curve links the network of best accuracy each year.

Figure 2: Increase of the computation power of GPUs and of the cost of CNNs over the years. Datas were extracted from `en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`, `en.wikipedia.org/wiki/Tegra` and `paperswithcode.com/sota/image-classification-on-imagenet`.

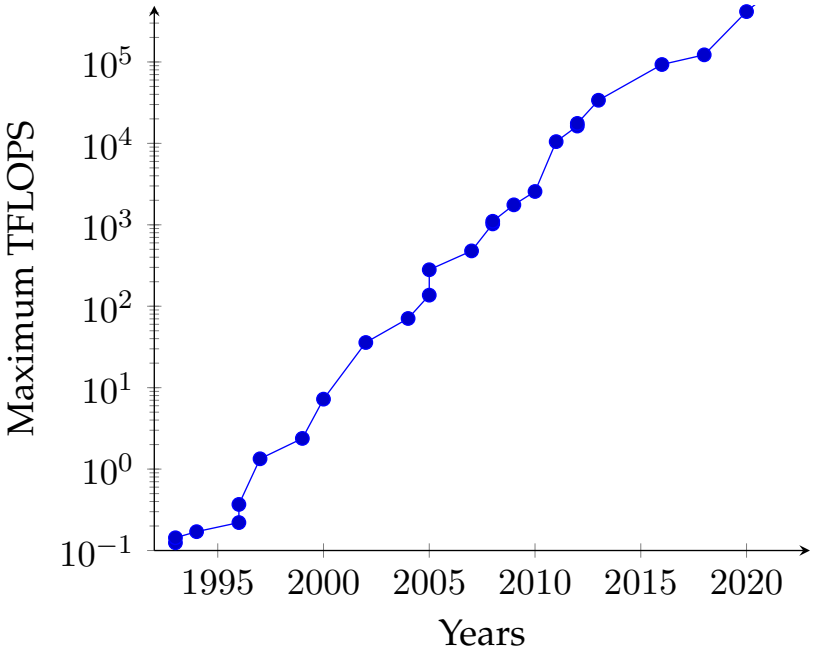


Figure 3: Most powerful supercomputers over years; data extracted from `en.wikipedia.org/wiki/History_of_supercomputing`.

super computer has a power of 17.590 PFLOPS; in 2022, Frontier has a power of 1.102 exaFLOPS, which means $62.6\times$ that of Titan.

This race in computation power does not involve solely the domain of deep learning—after all, the supercomputers we mentioned are not specifically dedicated to deep neural networks—, but it is especially problematic in this field, as the ability to train high performance networks on challenging datasets is often the key to publish in the most influential conferences. This fracture between big companies, that have access to a huge computation power, and more modest laboratories explains why a large portion of the literature settled on using, for example, ResNet-50 (25M parameters) on ImageNet instead of more recent and state-of-the-art networks—and yet, despite this compromise, they often still need to resort to using public supercomputers available for researchers, such as the Jean Zay calculator[8].

Therefore, the computational cost of deep neural networks, far from being a negligible aspect, actually underlies many crucial—and even polemical—aspects of the field: computational and academical supremacy of big companies or laboratories, ecological impact of GPU clusters, lack of consideration for efficiency or even mere understanding of how the performance of neural networks work—instead of indefinitely increasing the number of parameters.

The other big concern of such a growth in the cost of CNNs, and actually the one that motivated the launch of this thesis, is the disparity between the desktop/workstation GPUs and those actually available for embedded devices.

[8] `www.idris.fr/jean-zay/`

1.1.3 Efficient CNNs for Limited Embedded Hardware and Industrial Applications

While the cost of training neural networks is a major concern, that of running successfully trained networks on less powerful devices is another in its own right. Indeed, Figure 2a shows in different colors the power of GPUs of different types, including desktop GPUs, on which neural networks can be trained simply, and Tegra GPUs, that are those that fit System on Module (SoM) that include a GPU, CPU, memory, power management, high-speed interfaces, such as the NVIDIA jetson AGX Xavier[9], that are more susceptible to be used in actual embedded industrial applications, such as autonomous vehicles.

The problem is that these embedded GPUs are far less powerful than regular ones: the most powerful Tegra GPU available—the AGX Orin 64GB GPU from 2021, with its 5.32 FP32 TFLOPS—is roughly as powerful as a GeForce GTX TITAN Black from 2014, with its 5.12 FP32 TFLOPS: virtually, running a modern CNN on an embedded device equates to running it on an at least 8 years old hardware. Of course, comparing the power of devices on the sole basis of their theoretical maximum FLOPS is a rough estimate, but it illustrates well the gap between what is possible in laboratory and what is actually efficient in practice, in real-world application.

Fortunately, Figure 2b also shows that, at equal accuracy, the number of parameters seems to decrease: not only do we train CNNs better and have designed more efficient architectures (we can think of the gap between VGG [20] and ResNet [21], with the latter having a lot fewer parameters for a better accuracy than the former), but also because of the field of neural networks compression, that we will describe first in Section 1.3.1.

However, in industrial applications, raw performance or computation cost are not the only variable to consider: robustness, explainability, latency or energy consumption are metrics that can be very important too, for different reasons. These industrial constraints will be explained in the following section.

1.2 Industrial Context

This CIFRE thesis, initiated by Stellantis, has to be put in a particular industrial context. To explain it, we will first review what is Stellantis, what is its interest in CNNs and what are the problems that this thesis tackles.

1.2.1 Stellantis

Stellantis is a multinational automotive manufactoring corporation, born from the fusion between Groupe PSA and Fiat Chrysler Automobiles NV (FCA) in 2021. It counts 16 brands and more than 400k employees. With the creation of an internal software department and the intent to make from 20 to 50% of artificial-intelligence-based services and revenues, deep learning is expected to occupy a crucial place in Stellantis in the years to come. Between January and October 2022, the AI staff grew from 10 to 83 employees, with a goal of 500 by 2030.

However, this thesis begun before this new policy and the fusion between PSA and FCA, so the context of the creation of this thesis was slightly different: that of a company in which the involvement of AI was expected but still to be confirmed. More precisely,

[9] www.nvidia.com/fr-fr/autonomous-machines/jetson-agx-xavier/

the specific application that was envisaged was that of autonomous vehicles or, more generally, automation in vehicles.

1.2.2 Automated Vehicles

Autonomous driving has received intense media coverage in the last years. However, one often overlooked point is that there is a gradient between traditional vehicles and fully-autonomous ones. Indeed, there are many functionalities to automate before reaching full autonomy, and this is why, in this chapter, we will not discuss solely autonomous vehicles but rather automation in general.

Usually, the automation rate of vehicles is divided into 5 levels (*cf.* SAE International J3016)[10]:

0. No assistance: the human driver is fully in charge of controlling the vehicle.
1. Driver assistance ("hands on"): the driver still controls the vehicle, but some functions help facilitating some tasks (cruise control, collision warning, automatic emergency braking, etc.).
2. Partial automation ("hands off"): the driver still supervises (driving assist in traffic jams, parking assist, etc.).
3. Conditional automation ("eyes off"): the vehicle performs most tasks, but the driver can override it if necessary (highway autopilot, platooning, etc.).
4. High automation ("mind off"): fully automatized vehicle, but in specific circumstances.
5. Full automation ("steering wheel optional"): no need for any human intervention.

Of course, such vehicles, depending on their level of automation, require many sensors (optic cameras, LiDARs, accelerometers, microphones, hydrometers, etc.) to detect any important information (presence of obstacles or road markings, velocity of the vehicle, whether the road is wet or not, etc.) and, more importantly in the scope of this manuscript, the devices and algorithms to process them and infer a relevant decision. It is, precisely, the ability of CNNs (or deep neural networks in general) to process data and produce a decision that makes them especially interesting in this industry.

There are multiple types of tasks, tackled by CNNs in the literature, that are of main interest for autonomous driving: image semantic segmentation [22] or detection [23], sound processing [12], Simultaneous Localization And Mapping (SLAM) [24] and many others. All of them are explored in an extensive literature, and some works have even studied their application to autonomous driving [25].

1.2.3 Industrial Problem

In the previous sections, we mentioned that CNNs could need an important budget in computation power to yield the best performance possible. Obviously, embedded hardware in vehicles cannot afford such a power: not only car batteries cannot power up big processors durably, but also it would be absurd to fit a vehicle with as many high-end processors as there are functions to automatize through neural networks. Therefore, neural networks have to be run on low-power devices with limited computation power and possibly in a way that allows running multiple different CNNs on a same device.

[10] `www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic`

However, the raw computational power or available memory are not the only constraints to come with this type of application. Indeed, autonomous vehicles need to take vital decisions in, sometimes, the span of a few milliseconds, depending on the vehicle's velocity. On top of that, these decisions have to be accurate and robust. The problem is that we have vital constraints on performance, robustness and latency, while most vehicles cannot afford embedding the complex hardware that would be powerful enough to run big networks quickly enough. Indeed, not only would such hardware be expensive, but also, both their high processing power and the large energetic cost of memory accesses and management would be unmanageable for such vehicles and their electrical/electronic (E/E) architecture—the thermal dissipation can be a problem too. This means that every aspect of the problem is constrained and none can be sacrificed to help the others: we need to fit the performance of cumbersome state-of-the-art networks onto what would be considered as outdated hardware, in another context, and make it work in real-time; there is no way to circumvent this issue, and it is a blocking point.

To solve this problem, we have to make either the hardware or the networks more efficient. Some manufacturers propose dedicated accelerators, such as the R-Car Automotive System-on-Chips (SoCs) by Renesas[11]; however, this thesis focuses on the other side of the problem: finding how to optimize networks for a given hardware, that automobile constructors rarely have the occasion to design themselves. For that purpose, we decided to do our studies on an embedded GPU by NVIDIA, the Jetson AGX Xavier[12].

1.3 Problem Statement

We now have all the context that is necessary to understand the initial scope of this thesis: CNNs, or at least those found in the literature and winners of contests, are too expensive for real-world applications, such as autonomous driving, that can only afford embedded hardware whose computing resources, memory amount and power consumption are drastically limited when compared to cloud servers. We need to make these networks fit such devices, while keeping their performance and staying under a strict budget in latency or energy. As we already showed in Figure 2b, the improvement of the performance of neural networks came along an increase in both the required computation power and the number of parameters: to get better networks, we would need bigger networks on more powerful hardware, which seems contradictory with the needs and practical constraints of industrial applications.

In order to prove this unfortunate rule wrong, the domain of neural networks compression aims at reducing the cost of neural networks while keeping their performance intact.

1.3.1 Neural Networks Compression: a Wide Domain

In Figure 2b, we saw that at equal accuracy, networks seemed to become more lightweight over the years. This is because it is possible to find families of architectures that are so fundamentally more efficient that the apparent trade-off curve between accuracy and the number of parameters gets shifted. We illustrate this fact with Figure 4, that compares different families of architectures. The most dramatic shift is between

[11] www.renesas.com/us/en/products/automotive-products/automotive-system-chips-socs

[12] www.nvidia.com/fr-fr/autonomous-machines/embedded-systems/jetson-agx-xavier/

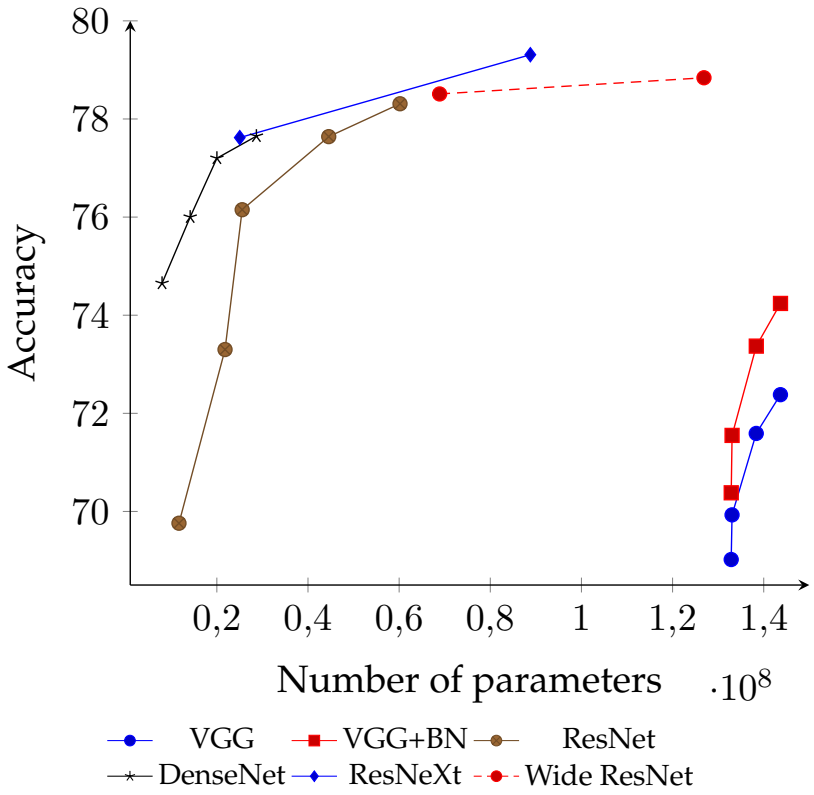


Figure 4: Top-1 accuracy on ImageNet and number of parameters of CNNs. Each curve is a separate family of CNN. Reported accuracy is extracted from `pytorch.org/vision/0.8/models.html` and the numbers of parameters is calculated using the Pytorch implementation of the same networks.

VGGs and ResNets: the whole tradeoff curve is shifted both towards smaller numbers of parameters and better accuracy at the same time.

Therefore, saying that the performance of CNNs scales with their number of parameters does not reflect the actual extent of the question: yes, for a same type of architecture, of the same efficiency, the performance usually scales with the size of the network; however, between architectures, this efficiency can vary drastically.

It is explicitly this efficiency that the domain of compression aims at maximizing, so that, for a fixed budget in parameters, memory or operations, the accuracy can be improved. To do so, multiple solutions exist: producing more efficient architectures, which is the domain of factorization or neural architecture search; reducing the memory occupation of parameters, through quantization or clustering; or sparsifying a network through pruning.

Pruning, that basically involves making a network more lightweight by removing its most unnecessary parts, has been very popular since 2015 [26]—partly because of how intuitive its basic principle is. However, its extensive literature and the fact that some papers raise the alarm concerning the methodological rigor of the field [27] point out that this domain may not be as simple as expected.

1.3.2 Dwelling Deep into Pruning

Indeed, as we will see in the rest of this manuscript, pruning can be separated in multiple aspects, and that each of them raises its own questions and has, implicitly, a ded-

icated literature. While some papers [27] warned the domain about the lack of comparability between papers, because of the absence of a common standardized benchmark, our own discussions, that we will expose all over this manuscript, point out the need for a more meticulous comparison of pruning methods on a more conceptual level: the important issue is not only to compare different methods on the same benchmark, but also to tell what precisely makes them different and what is the individual benefit of each of these differences.

While this thesis was originally intended to tackle the whole field of compression and could have turned into a high-level comparison of the relative efficiency of each method and of their combination, the surprising complexity of pruning and all the interesting epistemic questionings it raised quickly led us to explore exclusively pruning, as it appeared to us that it was not possible to provide a truly relevant hindsight about this field without taking the time to actually explore it in depth.

1.3.3 Definitive Problem Statement

We can now formulate what is the actual scope of this manuscript. Improving the accuracy of CNNs, to both fit the limited hardware and reach the required performance for industrial applications such as autonomous driving, can involve many different methods. Each method has its dedicated literature, with its questions, its problems and its evolution over the years and the papers. It would be rather unlikely that summing up each of them to a single archetypal method, to be compared with the others, would give off the full extent of what each field has to offer in term of efficiency or even raw scientific knowledge. Staying at a surface level would only sum up as doing dubious comparisons between strawmen, without any guarantee that it makes any sense.

We think that such a superficial work would not have been of great interest for Stellantis, and the implied lack of depth would have harmed the academic relevance of this thesis. This is why we instead decided to restrain the scope of this thesis to focus on one specific domain: pruning, that has a particularly extensive, competitive and somewhat confusing literature. Indeed, the arguable success of a blog article[13], that we wrote during this thesis, shows the appeal of a more thorough and pedagogic presentation of the field for those who want to understand pruning—since this is not an peer-reviewed publication, we will not mention this article again.

In this manuscript, we will describe what are the main questions and aspects that structure the whole domain of neural networks pruning. We will present the dedicated literature and our hindsight about each of them, as well as our own academic contributions and how they fit in the field. Also, our work mostly considers the comparison between various types of costs of the network to its performance at solving the task; since the question of robustness or of how performance should really be measured seems to be a very non-trivial question [28] in its own right, we decided to consider it out of our scope.

Similarly, we restricted the scope of our study to computer vision CNNs—and more specifically a limited array of CNNs—not only because observations made on such networks can easily be transposed to other types of architectures, but also because it is the same type of networks that are used in the literature. Moreover, diversifying the types of networks on which to make the experiments would have costed too much, for a limited methodological benefit, considering how large the domain to explore was.

[13] `towardsdatascience.com/neural-network-pruning-101-af816aaea61`

Exploring successively each dimension of pruning, in this manuscript, will allow reviewing from its most theoretical aspects, that are necessary to improve the performance of networks, to those that have a direct impact on their cost when running on embedded devices, thus covering the two implied goals of compression: maximizing the accuracy and reducing the cost. As we already pointed out, both aspects are equally necessary for industrial applications.

1.4 Contributions and Outlines

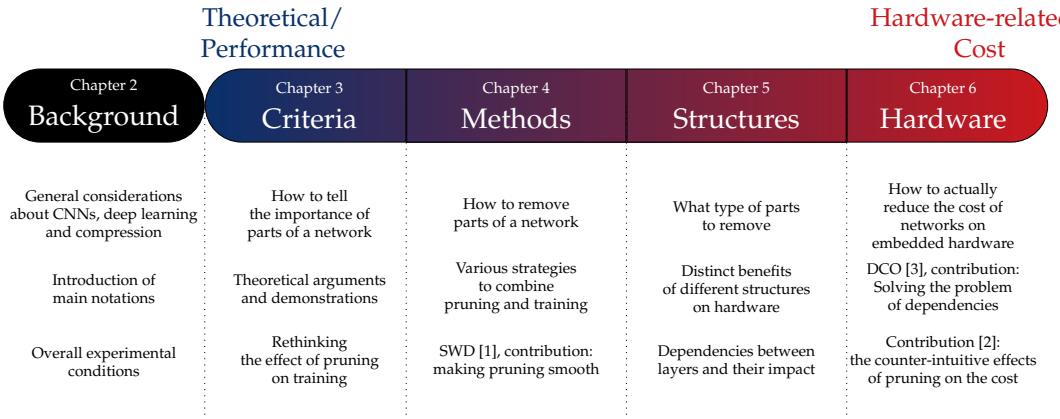


Figure 5: Graphic summary of the following chapters of this manuscript.

For this manuscript, we chose a thematic approach to structure our chapters, that will each tackle one distinct aspect of pruning. Therefore, each chapter is more or less independent, even though there will be references between them. As illustrated in Figure 5, there is a logical progression in the order of chapters, dealing first with the most fundamental aspects of pruning to conclude on its impact on the cost of networks on hardware, which allows in the end to provide a thorough and fully in-depth hindsight of the field, encompassing all the elements, in the scope, that can be of interest for the industrial application that motivated the launch of this thesis. Chapter 2 is different from the others, as it does not deal exclusively with pruning and instead provides all the necessary notions, notations and explanations to correctly apprehend the rest of the manuscript.

In Chapters 4 and 6, we will present 3 different contributions: Selective Weight Decay (SWD) [1], a pruning method that aims at solving some fundamental problems with removing parts of a network during training; Dimensional Clear-Out (DCO) [3], a method to identify and solve problems of dependencies when performing structured pruning and finally, one final contribution [2] that presents some measurements of the reduction of the energy consumption of CNNs through pruning, and how these measurements highlight some counter-intuitive behaviors of pruning. These three contributions respond to the two aspects we mentioned when stating the overall problem tackled by this thesis in Section 1.3.3: on one hand, improving the performance of pruned networks [2]; on the other hand, helping their implementation on embedded hardware [3] and studying how it reduces their cost [2]. All chapters will not present a contribution, since we review them in the chapters whose theme is the most related to that of the contributions. However, the chapters that do not feature an academic contribution still show our personal hindsight on their subject and the eventual—and sometimes ignored—problems of their respective literature.

We will now provide a short summary of the content of each chapter, highlighting our academic or informal contributions in each one:

- Chapter 2, "Deep Learning for Computer Vision and Compression Methods": presentation of the tasks, datasets, layers/operations and network architectures used in the rest of the manuscript. Also details what is the principle of training a neural network, what is compression, what are the main compression methods in the literature and why pruning can be separated in criteria, methods, structures and hardware implementation.
- Chapter 3, "Pruning Criteria": details how to tell the importance of a weight, then how to distribute the pruning rate across layers, how to generalize criteria to groups of weights or larger structures and when to measure these criteria relatively to training. In this chapter, we also provide our hindsight concerning both the demonstrations backing up certain criteria—and why these demonstrations tend to overlook one major problem—and a new way of presenting pruning, which explains intuitively many behaviors of pruning and provides a new way of considering what should be pruning criteria in theory. This chapter is related to the most theoretical side of pruning, and is necessary to maximize the performance of pruned networks.
- Chapter 4, "Removal Methods": presents the logic that links together the various families of methods to remove parts of networks (*i.e.* the parameters to remove after having identified them through a pruning criterion). This chapter shows that, implicitly, all these methods aim at solving the same fundamental problem of pruning, which is that the harsh perturbation introduced by pruning disrupts training. This chapter also presents SWD [1], a pruning method of our own. These methods both impact the final performance of the pruned network and the training time, which constitutes a first step towards more practical considerations.
- Chapter 5, "Pruning Structures": explains what are the different types of structures to prune and what are their benefits, notably in term of cost at inference or of reduction in performance. It also presents the problem of the interdependence of layers in a network, which is a thorny issue when pruning large structures; this problem is barely mentioned in the literature, while it clearly had a decisive impact on some practices in the field. This chapter is closely related to Chapter 6, and directly tackles the practical side of the problem stated in Section 1.3.3.
- Chapter 6, "Leveraging Pruning on Hardware": finally, this chapter presents which aspects, in neural networks, have a cost in energy, memory or latency on embedded hardware. It also presents a solution, called DCO [3], to solve the interdependence problem presented in the chapter before. Finally, using DCO, one last contribution [2] shows the impact of pruning on the energy consumption of networks; these experiments also show that some behaviors of pruning criteria have a direct impact on the energetic efficiency of pruned network, which closes the loop: the theoretical aspects of pruning finally have a direct impact on their most practical aspects.

Chapter 2

Deep Learning for Computer Vision and Compression Methods

Contents

This chapter will first set the context of the experiments featured in this manuscript. We will review the considered tasks, datasets and architectures as well as the training conditions. This will allow reviewing briefly the history and the usual practices of the respective domains. Once these bases are properly defined, we will review the overall domain of neural networks compression and, at last, especially that of pruning. In this last part, we will break down what are the different key notions of pruning that will be mobilized and developed in this manuscript.

2.1 Deep Learning for Computer Vision

Computer vision tasks are both a staple in the deep learning literature and one of the core domains to be addressed in various industrial applications such as the one that serves as the context of this thesis. Therefore, all experiments in this manuscript will be performed on standard tasks, datasets and networks of the literature. Whether or not the conclusions to be drawn from them can easily be transposed in an industrial context will be discussed for each individual task and dataset in Section 2.1.2.

2.1.1 Tasks

The two tasks that we tackled are image classification and semantic segmentation: one ubiquitous in the literature and the other closer to industrial constraints.

2.1.1.1 Image Classification

Being both one of the oldest [29] and most widespread [18] tasks in deep learning, image classification, its datasets and its networks are almost the default canvas in which to study neural networks. The vast majority of compression or, more specifically, pruning publications focus mainly, or even solely, on this task.

The principle of image classification is straightforward: it involves attributing one label, among n possible ones, to an image $\mathbf{X}$, that contains c channels and have a height h and a width w. This can be seen as a mathematical function $\mathcal{N}$ that takes $\mathbf{X}$ as an input and outputs a vector $\mathcal{N}(\mathbf{X})$ of length n, with each element of $\mathcal{N}(\mathbf{X})$ being the probability of $\mathbf{X}$ to belong to the corresponding class—the class to choose as the overall answer is the one whose probability is the highest. Even though images are usually processed in batches, it is simpler to consider only one image at a time in our notations—it suffices to know that the same operations are applied to multiple images in parallel, with limited interactions between them (apart from the computation of the batch-normalization or of the gradient). An image classification network can be defined as:

$$\begin{aligned} \mathcal{N} : \mathbb{R}^{c\times h\times w} &\to \mathbb{R}^n \\ \mathbf{X} &\mapsto \mathcal{N}(\mathbf{X}) \end{aligned}$$

Even though this task is extremely standard and ubiquitous, it brings some biases that motivated the extension of our work to the task of semantic segmentation. The two main biases are:

- Most of the time, if not always, the output space has a much reduced dimensionality compared to the input space, *i.e.* $c \times h \times w \gg n$. Therefore, most classification networks have the same overall design pattern of progressively reducing the definition space of its intermediate representations, loosing strategically (mostly spatial) information to result in a classification vector that contains no spatial information at all. This approach is not universal and many tasks do not use it,

which has a direct impact on the cost of the network. Therefore, when studying neural networks compression, it is important to diversify the tasks to tackle.

- Image classification assumes that the input images each involve only one object of interest that matches only one class. This is remote from most actual use cases in the industry since, for example, automation involves dealing with different types of situations, including unpredictable ones. For example, automated vehicles have to be able to detect different types of objects, possibly unknown ones, and react accordingly and reliably.

2.1.1.2 Semantic Segmentation

While staying, by certain aspects, analogous to image classification, semantic segmentation stays closer to actual application contexts such as autonomous driving [25]. It involves performing a pixel-wise classification of an image. Therefore, a semantic segmentation network can be defined as:

$$\begin{aligned} \mathcal{N} : \mathbb{R}^{c \times h \times w} &\rightarrow \mathbb{R}^{n \times h \times w} \\ \mathbf{X} &\mapsto \mathcal{N}(\mathbf{X}) \end{aligned}$$

Here, $\mathcal{N}(\mathbf{X})$ is not a mere vector but a tensor that contains the probability of each pixel to belong to each of the n classes. The actual segmentation of the input images can be deduced by taking the $argmax$ of $\mathcal{N}(\mathbf{X})$ to produce single-channel segmented images.

In contrast to image classification, the input and output spaces in semantic segmentation are of roughly similar dimensionality—so much that most networks actually reduce the resolution of the output segmented images, to cut on the number of operations to perform, and upsample them to the original shape afterward. As we will see in the rest of this manuscript, this difference has a significant impact on how to handle the compression of semantic segmentation networks.

We chose to stick to this simple type of segmentation without worrying about more subtle cases such as instance-level semantic labeling tasks, that combines detection with segmentation. We did not want to bring additional complexity to the tasks to tackle while it would not bring a more significant hindsight on the impact of the task on the architecture and of the architecture on the cost of neural networks.

Now that we have introduced the two tasks involved in this manuscript and justified this choice, we will present the precise datasets that we used.

2.1.2 Datasets

Since all our experiments were conducted on supervised learning datasets, we can already notate that a dataset $\mathcal{D}$ is a set of pairs $(\mathbf{X}, \mathbf{y})$ with $\mathbf{X}$ an image and $\mathbf{y}$ its corresponding groundtruth, whether it is a classification label or a segmented image. A given dataset contains a fixed amount N of image/label pairs.

The three datasets we used are ImageNet ILSVRC2012, CIFAR-10 and Cityscapes—two classification datasets and one semantic segmentation one. ImageNet and CIFAR-10 are very widespread and standard in the literature; Cityscapes is a reoccurring reference when dealing with semantic segmentation, which is itself less common. CIFAR-10 is ubiquitous to tune the hyperparameters of a method but is a bit too much of a toy problem to be relevant for industry; ImageNet is the main reference on which most deep learning methods are tested; Cityscapes is specially dedicated to autonomous driving and is, therefore, very relevant in the context of this thesis.

2.1.2.1 ImageNet ILSVRC2012

Figure 6: Examples of images from ImageNet ILSVRC2012

Since the historical performance of AlexNet [9], at the ImageNet Large Scale Visual Recognition Challenge of 2012 (ILSVRC2012) [18], that introduced deep learning as the state of the art in computer vision, the ImageNet classification dataset became the main reference for evaluating the performance of a method in the deep learning literature. It contains 1,281,167 training images, 50,000 validation images and 100,000 test images, classified in 1000 classes. Images are usually cropped into images of size 224×224 and all contain 3 RGB channels. Figure 6 provides some examples originating from the dataset.

Because of the number of images and their size, this dataset tends to be the final acid test of methods developed using smaller datasets that are faster to train on, such as CIFAR-10. Indeed, training on ImageNet can take up to a week, depending on the available hardware, which makes it a poor candidate for hyperparameter search.

2.1.2.2 CIFAR-10/100

There exist 2 different CIFAR [30] datasets, that are both composed of 32×32 RGB images and divided into a train and a test set:

- CIFAR-10 is divided into 10 classes and counts 50k training images and 10k test images. Classes are equally balanced and are mutually exclusive.
- CIFAR-100 is divided into 100 classes and counts also 50k training images and 10k test images. It is also equally balanced, which means that each class has fewer images. Images are also divided into 20 superclasses that can be used as

Figure 7: Examples of images from CIFAR-10

a coarser label, even though training on CIFAR-100 typically use exclusively the fine labelling of the 100 mutually exclusive classes.

CIFAR-10 is very widespread as a training dataset for quick yet reliable tests and hyperparameters search. It is both much faster to train than ImageNet and more complex and interesting to use than old datasets such as MNIST—even though it is still too simple to be considered consistent with industrial applications. CIFAR-100 is a bit more rarely used because it presents a much more complicated challenge, with more classes and a lot fewer examples per class. Figure 7 provides some examples originating from the CIFAR-10 dataset.

2.1.2.3 Cityscapes

The Cityscapes dataset [25] contains high-resolution images of urban street scenes that were captured from a vehicle in 50 different cities in Germany. It features various seasons, times of the day, weather conditions, types of featured objects, scene layouts and background.

The subset we used contains 2975 training images and 500 validation ones. The test set is not annotated, so we do not count it. Images are of size 1024×2048, even though they are usually cropped to 512×1024 during training. We use the finely annotated groundtruth segmented images, that are labeled for 30 different classes, even though the literature only uses 19 of them, which we do too. Figure 8 provides some examples originating from the Cityscapes finely annotated subset.

Figure 8: Four examples from the Cityscapes dataset. The original images and their fine annotations are superposed.

2.1.3 Operators

Neural networks for image classification and semantic segmentation share mostly the same types of operators. In order to expose more easily the specific architectures to be found in the literature and, more specifically, those we used in our experiments, we will review all the involved operators.

2.1.3.1 Linear layers

Linear layers are the original main component of neural networks and are at the heart of perceptrons [6] and multi-layered perceptrons. Since we never see anymore networks that contain only one layer, we will notate every layer operation as indexed by ℓ, with $\ell \in \{0, \dots, L-1\}$ with L the number of layers in the network $\mathcal{N}$. The input $\mathbf{X}_\ell$ and output $\mathbf{X}_{\ell+1}$ of a linear layer $\mathcal{N}_\ell$ are vectors of size c_ℓ and $c_{\ell+1}$, and its parameters are composed of a $c_\ell \times c_{\ell+1}$ matrix $\mathbf{W}_\ell$ of weights and a vector $\mathbf{b}_\ell$ of biases of size $c_{\ell+1}$. The layer operates a matrix multiplication between $\mathbf{X}_\ell$ and $\mathbf{W}_\ell$ and sums $\mathbf{b}_\ell$ to the result; we notate this operation as:

$$\begin{aligned} \mathcal{N}_\ell : \mathbb{R}^{c_\ell} &\to \mathbb{R}^{c_{\ell+1}} \\ \mathbf{X}_\ell &\mapsto \mathbf{X}_\ell \cdot \mathbf{W}_\ell + \mathbf{b}_\ell \end{aligned}$$

Historically, a "neuron" corresponds to the c_ℓ weights and the single bias that are involved in the production of one value in $\mathbf{X}_{\ell+1}$, for a total of $c_{\ell+1}$ neurons. In the architectures involved in our experiments, linear layers are only found at the end of image classification networks, where they serve as a linear classifier that takes as an input the embedding produced by a convolutional network and returns the classification vector $\mathcal{N}(\mathbf{X})$ described in Section 2.1.1.1

2.1.3.2 Convolution layers

In deep learning applied to computer vision, convolution layers, or more precisely 2D convolution layers, are ubiquitous. They are designed to operate on tensors, that can be viewed as an image containing c_ℓ channels called "feature maps"; these tensors are of size $c_\ell \times h_\ell \times w_\ell$ with h_ℓ and w_ℓ being the height and width of the feature maps that are taken as the input of the ℓ-th layer of the network. Convolution layers contain weights $\mathbf{W}_\ell$ of size $c_{\ell+1} \times c_\ell \times k_h \times k_w$, with k_h and k_w being the height and width of the

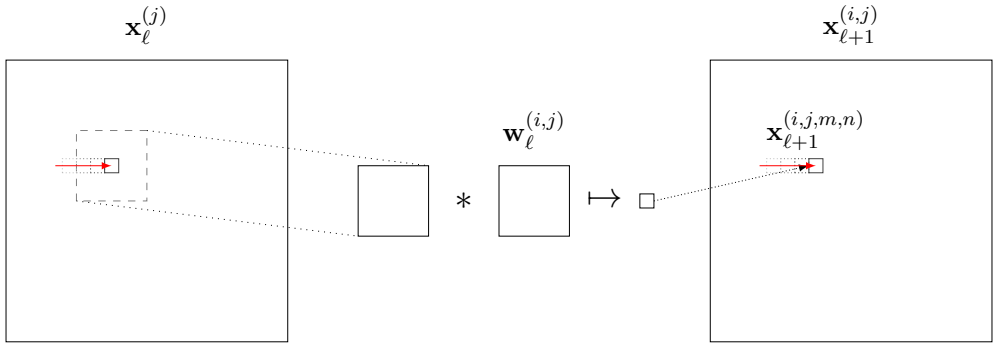


Figure 9: The convolution of $\mathbf{X}_\ell^{(j)}$ by $\mathbf{W}_\ell^{(i,j)}$ acts like an element-wise product between $\mathbf{W}_\ell^{(i,j)}$ and a sliding window in $\mathbf{X}_\ell^{(j)}$ to produce the pixel $\mathbf{X}_{\ell+1}^{(i,j,m,n)}$ in $\mathbf{X}_{\ell+1}^{(i,j)}$.

convolution kernels—the "features" to extract from the input to produce the "feature maps". Since kernels are almost always squares, we will instead notate $k_w = k_h = k_\ell$ with $k\ell$ the size of the kernel in layer ℓ. Convolution layers also optionally contain biases $\mathbf{b}_\ell$ of size $c_{\ell+1}$. The operation performed by the convolution layer can be written as:

$$\mathcal{N}_\ell : \mathbb{R}^{c_\ell \times h_\ell \times w_\ell} \to \mathbb{R}^{c_{\ell+1} \times h_{\ell+1} \times w_{\ell+1}}$$

$$\mathbf{X}_\ell \mapsto \mathbf{X}_{\ell+1} : \forall i \in \{0, \dots, c_{\ell+1} - 1\}, \mathbf{X}_{\ell+1}^{(i)} = \mathbf{b}^{(i)} + \sum_{j=0}^{c_\ell - 1} \mathbf{W}_\ell^{(i,j)} \star \mathbf{X}_\ell^{(j)}$$

With $\star$ being the 2D cross-correlation operator. Since convolution layers, in some cases, have a "stride" hyperparameter s_ℓ (that we supposed to be the same for height and width), the cross-correlation is defined as:

$$\mathbf{W}_\ell^{(i,j)} \star \mathbf{X}_\ell^{(j)} = \mathbf{X}_{\ell+1}^{(i,j)} \in \mathbb{R}^{h_{\ell+1} \times w_{\ell+1}} : \forall (m,n) \in \{0, \dots, h_{\ell+1}\} \times \{0, \dots, w_{\ell+1}\},$$

$$\mathbf{X}_{\ell+1}^{(i,j,m,n)} = \sum_{a=0}^{k_\ell - 1} \sum_{b=0}^{k_\ell - 1} \mathbf{W}_\ell^{(i,j,a,b)} \mathbf{X}_\ell^{(j, a - m s_\ell, b - n s_\ell)}$$

Another important hyperparameter is the "padding" p_ℓ (that we supposed to be the same for height and width), that impacts the resolution of the output feature maps by adding a margin, filled with zeros, around the input feature maps. Therefore, we have:

$$h_{\ell+1} = \left\lfloor \frac{h_\ell + 2p_\ell - (k_\ell - 1) - 1}{s_\ell} \right\rfloor \qquad w_{\ell+1} = \left\lfloor \frac{w_\ell + 2p_\ell - (k_\ell - 1) - 1}{s_\ell} \right\rfloor$$

To sum it up in a more informal way: the convolution (or rather cross-correlation, but not only are the two operations almost the same, but the confusion in terminology is ubiquitous in the literature) behaves like that: as illustrated in Figure 9, each pixel $\mathbf{X}_{\ell+1}^{(i,j,m,n)}$ in the partial feature map $\mathbf{X}_{\ell+1}^{(i,j)}$ is the element-wise product between the kernel $\mathbf{W}_\ell^{(i,j)}$ and a cropped window of size $k_\ell \times k_\ell$ of the input feature map $\mathbf{X}_\ell^{(j)}$ that slides over s_ℓ pixels for each new pixel in $\mathbf{X}_{\ell+1}^{(i,j)}$. Each output feature map $\mathbf{X}_{\ell+1}^{(i)}$ is the sum of the convolution of a different kernel for each feature map in the input $\mathbf{X}_\ell$; all kernels

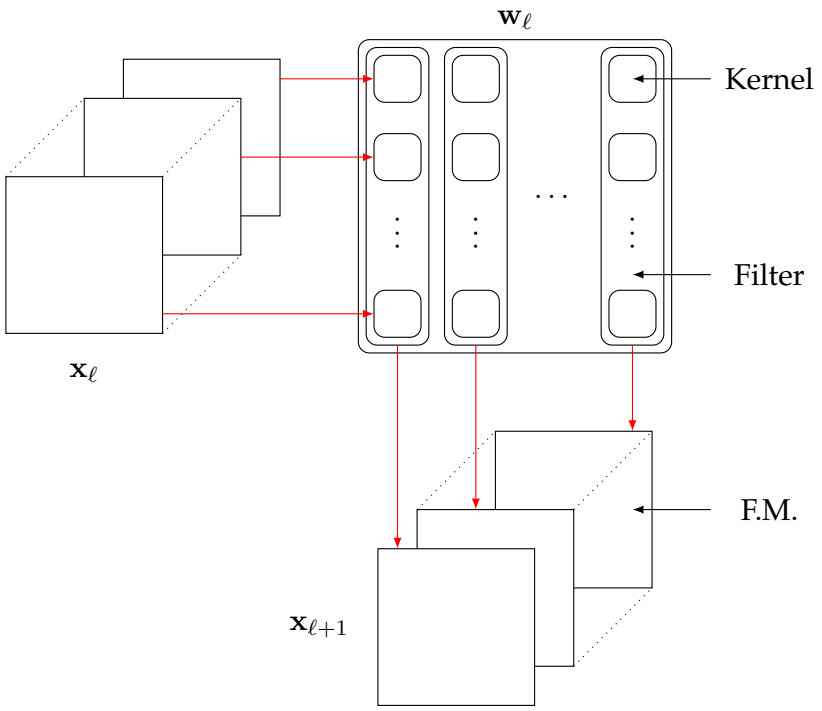


Figure 10: Structure of a convolution layer: each filter contains as many kernels as there are input feature maps (F.M.) and produces one output feature map.

involved in the production of this feature map (as well as the eventual corresponding bias) are called a *filter*. A convolution layer counts $c_{\ell+1}$ filters. The overall structure of a convolution layer is illustrated in Figure 10.

Two other hyperparameters, that are not to be found in the networks we used but that exist nonetheless, are:

- "groups": Splits the channels in the input feature maps as well as the filters, such that the filters of each split only operate on the feature maps of the corresponding split in the input, which allows reducing the number of operations and parameters. If there are as many groups as input feature maps, which means that every output feature map is the product of only one cross-correlation between the corresponding input channel and one kernel, then, once combined with convolutions with kernels of size 1×1, we get what are called "depthwise-separable convolutions".
- "dilation": Introduces a gap between pixels in the kernels in order to operate on a larger area in the feature maps while keeping the same number of parameters. Dilated convolutions are also called "atrous" convolutions.

2.1.3.3 Batch-Normalization layers

The use of batch-normalization (or "BatchNorm") layers [31] is almost systematic in modern networks such as ResNets [21]. It comes after every layer in a network, so that it can be considered as a part of them, especially since convolution layers rarely contain biases anymore, that are instead included in the following batch-normalization layer. Batch-normalization layers operate independently on each feature map of their input and apply two operations: 1) a normalization of each feature map accordingly to "running statistics" that are progressively updated during training (and frozen during

inference) and 2) multiplication and addition with a learned weight and bias. Therefore, this layer can be described as:

$$\mathcal{N}_\ell : \mathbb{R}^{c_\ell \times h_\ell \times w_\ell} \rightarrow \mathbb{R}^{c_\ell \times h_\ell \times w_\ell}$$

$$\mathbf{X}_\ell \mapsto \mathbf{X}_{\ell+1} : \forall i \in \{0, \ldots, c_\ell - 1\}, \mathbf{X}_{\ell+1}^{(i)} = \frac{\mathbf{X}_\ell^{(i)} - E\left[\mathbf{X}_\ell^{(i)}\right]}{\sqrt{Var\left[\mathbf{X}_\ell^{(i)}\right] + \epsilon}} * \mathbf{W}^{(i)} + \mathbf{b}^{(i)}$$

The interest of batch-normalization is twofold: 1) it prevents "covariate shift", i.e. the tendency of the distribution of layer's inputs to change during training, which harms training, and 2) it prevents the case where some feature maps tend to be systematically in a range of value that gets them nullified by the ensuing activation function, which maximizes the network's usage of its parameters: for example, ReLU activation functions nullify all negative values; if a feature map's distribution is normal, then at least half of its values are expected not to be nullified.

The specificity of batch-normalization is that $E\left[\mathbf{X}_\ell^{(i)}\right]$ and $Var\left[\mathbf{X}_\ell^{(i)}\right]$ are actually computed on the same channel in every image of a training batch at once. This means that training a batch-normalization layer requires a sufficient batch size to compute properly the mean and variance of each channel.

2.1.3.4 ReLU activation functions

Non-linear activation functions are necessary for functioning neural networks. Since AlexNet [9], one of the most popular ones are Rectified Linear Units (ReLU) functions, that are simply defined as:

$$ReLU : \mathbb{R} \rightarrow \mathbb{R}^+$$
$$x \mapsto max(x, 0)$$

This activation function has two advantages: 1) it is extremely simple and fast to compute and differentiate and 2) since its derivative equals 1 in the positive domain, it does not introduce the "vanishing gradient" problems that was encountered with previous activation functions, whose derivative tended to be smaller than one.

2.1.3.5 Tensor Additions

Modern networks such as ResNets [21] are rarely simple straightforward networks as were AlexNets [9] or VGGs [20]: they can contain various types of long-range connections between layers, which is often performed by summing together the output of different layers. In the networks we used in our experiments, tensor additions are the only operators to involve multiple input feature map tensors at once and require them to have the same dimensions. This property will prove very important later in the manuscript.

2.1.3.6 Interpolations

Even though interpolation operations are rare in classification networks, they are commonplace in semantic segmentation ones, such as HRNets [22]. They serve to adapt, and usually upsample, the resolution of feature maps. There are multiple ways to

achieve this upsampling, while we did not measure any significant difference in performance in our experiments. That is why we used only simple 2D nearest neighbor upsampling operations, as they are the least operations-consuming.

2.1.3.7 Pooling operations

Formerly widespread in deep convolutional networks, pooling operations mostly got replaced by strided convolutions to perform downsampling, like in ResNets [21]. Although rarer, "MaxPooling" operations can still occasionally be found, for example at the beginning of ResNets for ImageNet, that start by reducing the resolution of input images using maxpooling. Maxpooling operations also consists in a sliding, usually strided, window which, instead of performing a cross-correlation, selects the maximum value in the image inside that window.

However, one type of pooling operation that is widespread in classification networks is Global Average Pooling [32], that averages all values in a feature map to produce one single value, whatever the resolution of the said feature maps. The interest of this operation is threefold: 1) it is not dependent on the network's input's resolution, which allows the network to be applied on any image size, 2) it fully abstracts the last embedding intermediate representation into a features vector with no spatial information, which makes classification easier and allows using only one linear layer as the final classifier of the network, and 3) it greatly reduces the size of the said linear classifier. These last two aspects allowed to evolve from the cumbersome multi-layered classifier of VGGs [20], that took up most of the network's parameters, to the much simpler one-layer classifier of ResNets, whose cost is negligible compared to the rest of the network.

2.1.4 Architectures

Now that we have reviewed the tasks and datasets to experiment with and the operators to build networks from, we will review the classification and semantic segmentation networks that we used in our experiments. We will also briefly review the history of classification and semantic segmentation networks to understand how the literature came up with the solutions that we chose for our experiments.

2.1.4.1 History of Image Classification Networks

Before the introduction and expansion of deep convolutional neural networks (CNN), classification or detection was performed by manually extracting features out of images and then applying various machine learning methods [33]. Convolution layers [7] then changed everything by allowing learning what features to extract. Once hardware became powerful enough to allow training large and deep neural networks, CNNs yielded groundbreaking results, notably at the ImageNet [18] challenge in 2012 with AlexNet [9]. Since then, different networks became popular and each brought improvements in accuracy and in design practices; in chronological order:

- LeNet [7]: brought the use of convolution layers and pooling operations for image classification and object detection.
- AlexNet [9]: brought the use of ReLU activation functions to prevent the vanishing gradient problem.
- Network in Network [32]: brought the use of global average pooling to considerably simplify the classifier part in now almost entirely convolutional networks.

- VGG [20]: brought the practice of stacking convolution layers with kernels of size 3×3 instead of using single layers with larger kernels, as it is more efficient in terms of parameters.
- GoogLeNet [11], [34]: brought the principle of splitting the network's blocks into different parallel sub-blocks that apply different operations. This was the first instance of convolution networks that were not straightforward stacks of layers.
- ResNet [21]: brought the principle of residual connections, that are shortcuts that sum the output of a block with its input to prevent any loss of information.
- SqueezeNet [35]: one of the first attempts at designing an efficient architecture for image classification; actively uses 1×1 convolutions to change the dimensionality of the input of the "fire modules" that are the building-block of SqueezeNet.
- DenseNet [36]: improves the performance by connecting the output of a layer to all following layers in a same block.
- ResNeXt [37]: uses grouped convolutions, under the term of "cardinality", to improve the performance of ResNets.
- MobileNet [38], [39]: uses depthwise-separable convolutions to produce an efficient image-classification network.
- EfficientNet [40]: uses neural architecture search (NAS) to choose the right dimensions of depth/width/resolution to produce highly efficient image classification networks.

Because of both their simplicity and efficiency, ResNets are still the standard baseline on image classification, notably in the pruning literature. ResNet-50 on ImageNet and ResNet-20 or ResNet-56 on CIFAR are the most common network-dataset couples to be found in recent pruning papers. This is why the next subsection describes more accurately the architecture of ResNets.

2.1.4.2 ResNet

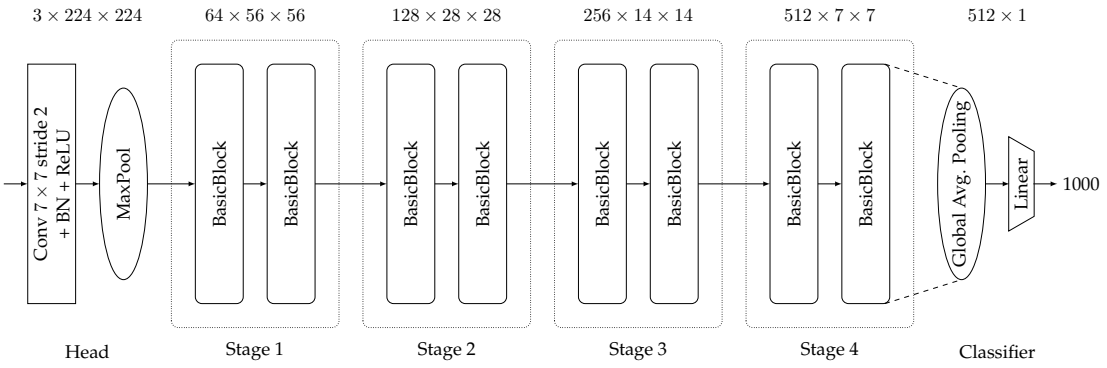
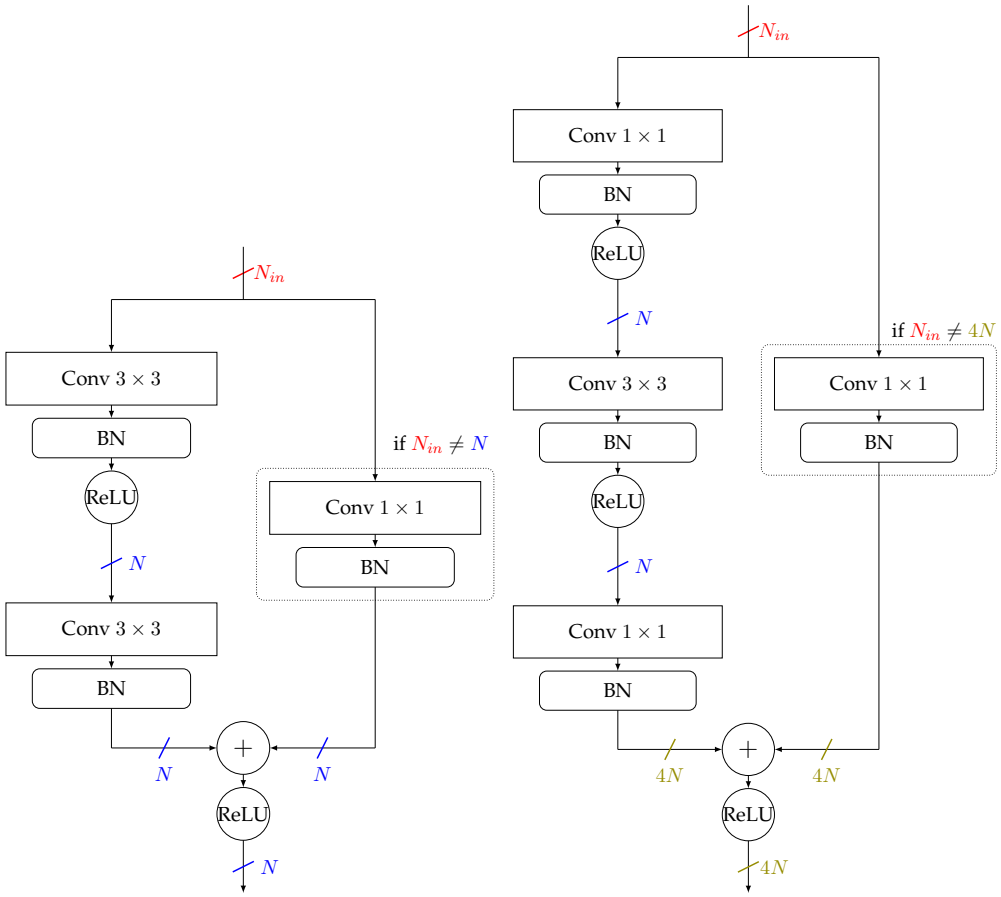


Figure 11: Architecture of ResNet-18 for ImageNet: after a head that reduces the input's resolution and increases its number of channels, the resolution is halved and the channels are doubled for each new stage, each made of two *BasicBlocks*. The final embedding is pooled, then classified into a vector of size 1000.



(a) *BasicBlock* **(b)** *Bottleneck*

Figure 12: The two variants of residual blocks: *BasicBlock* and *Bottleneck*. N is the intended width (in number of channels) of the block. The shortcut on the right is optionally replaced by a few layers if there is the need to adapt the number of channels of the input.

ResNets are composed of three different parts: 1) a small head made of a few layers, 2) a succession of residual blocks and 3) the linear classifier.

The head depends on the dataset. For CIFAR, the head is composed of one 3×3 convolution with no stride nor pooling. For ImageNet, the head is made of one 7×7 convolution layer with a stride of 2 and one maxpooling layer with a kernel of size 3×3 and a stride of 2. In either cases, the convolution is followed by a batch-normalization layer and a ReLU activation function.

Whatever the dataset, the linear classifier is built the same: one global average pooling operation and a single linear layer whose output dimension matches the number of classes in the dataset. In Pytorch [41], the $softmax$ function is included in the *CrossEntropy* loss function.

The core of ResNets rely in the succession of residual blocks. These blocks are divided into stages. For each new stage, the intermediate representations have twice the number of channels. There are two families of ResNets: those with 3 stages, such as ResNet-20 or ResNet-56 that are more adapted to CIFAR, and those with 4 stages, such as ResNet-18 or ResNet-50 that are more adapted to ImageNet. The 2^{nd}, 3^{rd} (and 4^{th} if existing) stages operate on intermediate representations of reduced resolution; this is

performed by introducing a stride of 2 in the first layer of each of these stages.

Each stage is made of a certain number of blocks, number that depends on the depth of the network. There are two different sorts of blocks: *BasicBlocks* and *Bottlenecks*. The distribution of the number of blocks across stages and the type of these blocks defines the sort of ResNet that is used: ResNet-18 is $[2, 2, 2, 2]$ *BasicBlocks* and ResNet-50 is $[3, 4, 6, 3]$ *Bottlenecks*; ResNet-20 is $[3, 3, 3]$ *BasicBlocks* and ResNet-56 is $[9, 9, 9]$ *BasicBlocks*. The overall structure of a ResNet-18 is illustrated in Figure 11.

A *BasicBlock* is made of two consecutive successions of a 3×3 convolution layer, a batch-normalization layer and a ReLU activation function; in the case of the second succession, the input of the block is summed to the output of the batch-normalization layer before being applied the ReLU activation function.

A *Bottleneck* block is composed of three parts: 1) a first 1×1 convolution (as well as a batch-normalization layer and a ReLU activation function) that reduces by four the number of channels of the input (to reach the intended width of the overall block), 2) a normal 3×3 convolution + batch-normalization + ReLU and 3) another 1×1 convolution + batch-normalization + ReLU that expands again the number of channels four times. Once again, the input of the block is summed to the output of the block before the final ReLU activation function. That reduction and expansion of four is a hyperparameter of *Bottlenecks* called "expansion".

The first block of each stage, to adapt the number of channels of the input, that is still propagated across layers through the residual connections, has an additional 1×1 convolution + batch-normalization that is applied to the input before being summed with the output of the block. Since the width of layers is constant for each layer of each block of a stage, and since this width is doubled for each stage, the initial width of the network is another defining property of a ResNet's architecture. ResNet-20 and ResNet-56 have usually an initial embedding of 16 feature maps, even though 64 are sometimes used and provide better performance despite being less efficient in terms of parameters; ResNet-18 and ResNet-50 usually have a base width of 64.

2.1.4.3 History of Semantic Segmentation Networks

Semantic segmentation started off as a modification of image classification networks. Most of those we will review here can be summed up as a decoder design, attached to a more or less modified version of a classification network that serves as an encoder. Similarly to classification networks, reviewing chronologically the most influential networks in the literature helps understanding which logic leads to the current state of the art. Here are some influential semantic segmentation networks:

- FCN [42]: first attempt at adapting a classification network, a VGG at the time, for semantic segmentation. The output of each stage of the encoder, that are of different resolutions and dimensionality, are fed to a decoder that, therefore, aggregates information at various scales at once. Here, each output is individually upsampled and projected into a segmented image (with as many channels as classes) using a convolution layer (or, more precisely, a *ConvTranspose* operation, that combines upsampling and convolution); then all these transformed outputs are summed together. Since, in the encoder, the final linear layers contain most of the information in the network, they are modified into a convolution layer whose output is fed to the decoder.
- DeconvNet [43]: takes a VGG as an encoder and remembers the index selected by each of its pooling layer. It then feeds the output of the encoder to a decoder that

applies deconvolutions and "unpooling" operations that re-expand the intermediate representations depending on the indexes selected in the encoder.

- U-Net [44]: improves FCNs by designing a pseudo-symmetrical decoder that progressively upsamples, applies convolution and sums together the output of each stage from the encoder, instead of directly transforming them into segmentation images. This network, as well as the following ones, tend to discard the linear layers at the end of the encoder.

- SegNet [45]: Very similar to DeconvNet, but discards the linear layer of the VGG encoder.

- Dilated frontend [46]: introduces the use of dilated or "atrous" convolutions for semantic segmentation. Since the kernels of a dilated convolution cover a larger area for the same number of parameters, dilating a pre-existing convolution while removing a previous pooling or stride operation allows increasing the resolution of an intermediate product while keeping the intended dimension of the features extracted by convolutions. To sum up: a regular convolution on a regular image extracts roughly the same features than the same convolution, once dilated, on the same image, but non-downsampled. Therefore, replacing the last convolutions of an encoder by dilated ones while removing strides or pooling operations allows preserving the relevance of the pretrained weights while producing outputs of higher resolutions that contain more information.

- DeepLab [47]–[49]: also uses the dilated convolutions in the same way, but modifies the last stage using "pyramid pooling" [50]: various convolutions, with different dilation rates, are applied to the same input and the outputs are summed together. This principle is perfected in DeepLabV3. Then, DeepLabV3+ elaborates its final decoder by introducing a U-Net-like re-injection of the encoder's secondary outputs.

- DANet [51]: adds a complex decoder, at the end of a dilated encoder, that uses an attention mechanism to compute inter-pixel and inter-channel interdependence in order to improve segmentation.

- HRNet [22], [52], [53]: instead of relying on an encoder designed for classification, HRNet is fully built for segmentation and applies the principle of pyramid pooling, similarly to DeepLab. However, instead of doing this using dilated convolutions, HRNet instead duplicates the intermediate representations at different resolutions, processes each duplicate independently and fuses together the obtained information. This design is applied all throughout the network, instead of involving only a decoder or the end of the network, as for DeepLab. HRNet can also be fitted with various kinds of decoders, such as HRNet-OCR [54].

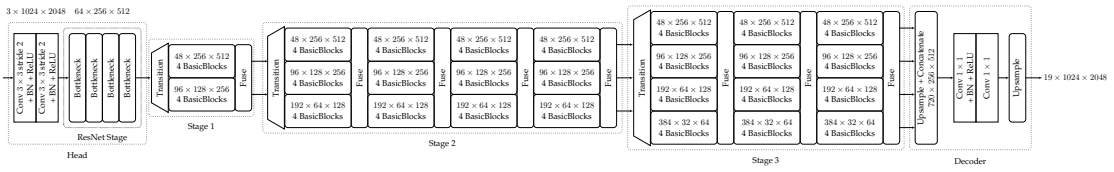
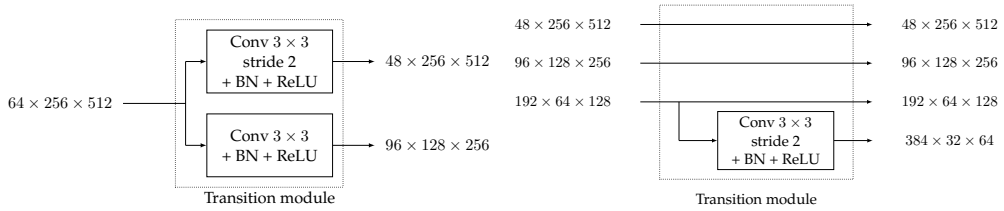


Figure 13: Overall architecture of HRNet-48. *Bottleneck* and *BasicBlocks* are described in Section 2.1.4.2 and Figure 12. Transition modules are illustrated in Figure 14. Fusion modules are illustrated in Figure 15.



(a) Case A **(b)** Case B

Figure 14: Two different transition modules: case A happens at the beginning of the first HRNet stage, where the output of the head is splitted into two branches of different resolutions; case B happens at the transition between the second and third stages, where the output of the three branches are forwarded without modification and a fourth one is created from the last of the three previous ones.

2.1.4.4 HRNet

Similarly to ResNets, HRNets are composed of three parts: 1) a head, composed of a few ResNet-like blocks, 2) a central part composed of multiple stages and 3) a small decoder at the end. The overall architecture is illustrated in Figure 13.

As for ResNets, the first layers of HRNet's head reduce the resolution of the input, using two strided convolutions. These two convolutions are then followed by four *Bottleneck* blocks, with a width of 64 channels whatever the type of HRNet: 18, 32 or 48.

The central part of HRNet is composed of three stages. Each stage is composed of a transition module (Figure 14) and a given number of "High Resolution Modules". Each HR-module is characterized by a number of branches and a width. Each branch operates on a different resolution and is composed of four *BasicBlocks*. After the branches, the module applies a fusion module (Figure 15). The transition module is responsible for generating the right number of inputs for the branches of the ensuing module. The fusion modules mix information coming from all branches and inject it into the input of each of the next module's branches.

Each of the three available types of HRNets are designed the same way: three stages, containing respectively one, four and three modules with two, three and four branches. The only difference between the three models is the base width of the *BasicBlocks* in their HR-modules: either 18, 32 or 48 channels.

Finally, the decoder upsamples three of the four outputs of the last stage, so that they all have the same resolution, in this case the highest of the four. Then, it applies two 11 convolutions. The output of the last convolution contains as many channels as classes in the dataset. Finally, the output is upsampled to the original resolution of the input.

2.1.5 Training

Now that we have all the information we need about the tasks, the datasets and the networks, we finally need to define how to train the said networks on the said datasets. First we will review some general information on neural networks training and then we will review the specific hyperparameters for each specific network-dataset couple.

2.1.5.1 General Training Hyperparameters and Algorithm

To train a neural network, one needs multiple elements:

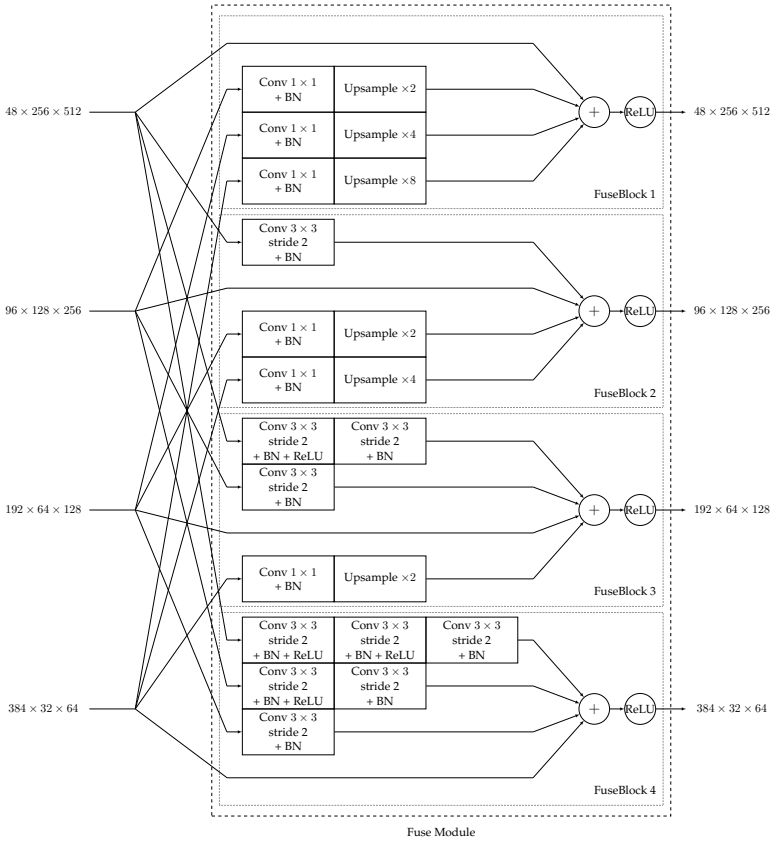


Figure 15: Example of a fusion module, in the case of a stage with four branches. The principle is the same for any number of branches.

- A dataset $\mathcal{D}$ that contains N image/label pairs $(\mathbf{X}, \mathbf{y})$, loaded by batches of b. We notate a batch $\left(\mathbf{X}^{(b)}, \mathbf{y}^{(b)}\right)$.
- A network $\mathcal{N}$ to train.
- A number of epochs E, i.e. the number of iterations over the whole dataset, the dataset being shuffled for each new iteration; the whole training therefore lasts during $\lfloor (N/b) \rfloor * E$ steps s.
- A loss function $\mathcal{L}$.
- A base learning rate λ.
- A scheduling function Sch for the learning rate, depending on λ and the current step s. Usually, the learning rate is instead updated for each new epoch e.
- An optimizer algorithm Op, with its own set of hyperparameters, to update the weights of $\mathcal{N}$ depending on its gradient $\mathbf{g}$ and a learning rate $Sch(\lambda, e)$.

Training a neural network usually involves iterating over the dataset, computing the output $\mathcal{N}(\mathbf{X}^{(b)})$, then the gradient of the loss function of this input $\mathbf{g} = \frac{\delta \mathcal{L}}{\delta \mathbf{W}}(\mathcal{N}(\mathbf{X}^{(b)}), \mathbf{y}^{(b)})$, and then updating the weights $\mathbf{W}$ of the network, using the optimizer depending on $\mathbf{g}$ and the current learning rate $Sch(\lambda, e)$. We can sum the training algorithm as Algorithm 1:

The goal of training is to minimize the loss function $\mathcal{L}$, which sets the objective toward which the network should converge. Actually, this objective function can be a mix of different functions, such as a specific criterion $\mathcal{F}$, that computes the error between the

Algorithm 1: Training of a neural network

Data: $\mathcal{N}$, b, $\mathcal{D}$, E, λ, Op, Sch
for $e \leftarrow 0$ *to* E **do**
  Shuffle $\mathcal{D}$;
  foreach $\left(\mathbf{X}^{(b)}, \mathbf{y}^{(b)}\right) \in \mathcal{D}$ **do**
    $\mathbf{g} \leftarrow \frac{\delta \mathcal{L}}{\delta \mathbf{W}}(\mathcal{N}(\mathbf{X}^{(b)}), \mathbf{y}^{(b)})$;
    $\mathbf{W} \leftarrow Op(\mathbf{W}, \mathbf{g}, Sch(\lambda, e))$

output $\mathcal{N}(\mathbf{X})$ and the intended label $\mathbf{y}$, and various types of secondary objectives or regularization functions.

One widespread regularization function is called weight decay [55], which can be summed up as a $\mathcal{L}_2$ penalty of the magnitude of the network's weights. The importance of this weight decay is defined by a coefficient μ.

For a given criterion $\mathcal{F}$ and a weight decay μ, the objective of training can be written as minimizing $\mathcal{L}$ for a given value of $\mathbf{W}$, accordingly to Equation 1:

$$\min_{\mathbf{W}} \mathcal{L}(\mathcal{N}(\mathbf{X}), \mathbf{y}), \text{with } \mathcal{L}(\mathcal{N}(\mathbf{X}), \mathbf{y}) = \mathcal{F}(\mathcal{N}(\mathbf{X}), \mathbf{y}) + \mu \left\|\mathbf{W}\right\|_2 \tag{1}$$

Finally, one last terminology to know, concerning training, is the principle of "fine-tuning", which generally consists in additional training epochs at the lowest learning rate used during training. For example, fine-tuning can be performed by pursuing training without updating the scheduler. Fine-tuning is often used to improve performance or reduce a performance degradation after a given post-training alteration of the network, such as pruning.

2.1.5.2 ResNet for CIFAR-10

For most of the ResNets we trained on CIFAR-10, we used the following hyperparameters:

- The batch-size, during training, is of 256.
- We trained during 300 epochs.
- The cross-entropy is the criterion, such as defined in the Pytorch [41] framework.
- The initial learning-rate is 0.1.
- The scheduling function follows the *MultiStepLR* function, configured so that the learning rate is divided by 10 at the milestones epochs 100 and 200.
- The optimizer is the standard Stochastic Gradient Descent algorithm, defined as SGD in Pytorch. The weight-decay is set at $5 \cdot 10^{-4}$, the momentum at 0.9 and the Nesterov momentum is disabled.

During training and testing, all images are normalized. During training, data augmentation consists in: 1) random crop, with a padding of 4, into images of size 32×32; 2) random horizontal flip. Hyperparameters of this data-augmentation were chosen in conformity with standard practice in the literature.

The standard metric to measure performance on the CIFAR datasets is the Top-1 accuracy. ResNet-20, with a base width of 16, usually achieves around 92.5% in Top-1

accuracy on CIFAR-10. With a width of 64, it achieves 95.5%. ResNet-56 with a width of 16 achieves 94.5%.

2.1.5.3 ResNet for ImageNet ILSVRC2012

For the ResNet-50 we trained on ImageNet ILSVRC 2012, we used the following hyperparameters:

- Because of technical limitations, the batch-size is only of 170 to fit the available memory of the GPUs we used during training.
- We trained during 90 epochs.
- The cross-entropy is the criterion, such as defined in the Pytorch [41] framework.
- The initial learning-rate is 0.01.
- The scheduling function follows the *MultiStepLR* function, configured so that the learning rate is divided by 10 at the milestones epochs 30 and 60.
- The optimizer is the standard Stochastic Gradient Descent algorithm, defined as SGD in Pytorch. The weight-decay is set at $1 \cdot 10^{-4}$, the momentum at 0.9 and the Nesterov momentum is disabled.

During training and testing, all images are normalized. During training, data augmentation consists in: 1) random resized crop, with the default hyperparameters of torchvision, into images of size 224×224; 2) random horizontal flip. Hyperparameters of this data-augmentation were chosen in conformity with standard practice in the literature.

The standard metrics to measure performance on the ImageNet dataset are the Top-1 and Top-5 accuracy. ResNet-50, with a base width of 64, usually achieves around 76% in Top-1 accuracy on ImageNet.

2.1.5.4 HRNet for Cityscapes

For the HRNets we trained on Cityscapes, we used the following hyperparameters:

- Because of technical limitations, the batch-size is only of 10 to fit the available memory of the GPUs we used during training.
- We trained during 200 epochs.
- The RMI loss [56] is the criterion we used.
- The initial learning-rate is 0.01.
- The scheduling function follows the *Poly* policy, with an exponent of 2, which means that the learning rate is reduced by $(1 - \frac{e}{E})^2$ at each epoch.
- The optimizer is the standard Stochastic Gradient Descent algorithm, defined as SGD in Pytorch. The weight-decay is set at $5 \cdot 10^{-4}$, the momentum at 0.9 and the Nesterov momentum is disabled.

During training and testing, all images are normalized. During training, data augmentation consists in: 1) random resized crop: a subpart of the image is cropped, from the whole 1024×2048 image to a smaller 256×512 one, which is then resized into a 512×1024 image; 2) random horizontal flip; 3) random Gaussian blur and 4) color jittering. Hyperparameters of this data-augmentation were chosen in conformity with state of the art contributions in the literature of semantic segmentation on Cityscapes [22].

The standard metric to measure performance on the Cityscapes dataset is the mean Intersection over Union (mIoU). This metric is usually computed only on pixels that belong to one of the 19 classes; the others are ignored. Our implementation of HRNet-48 achieves around 77% in mIoU on Cityscapes.

2.2 Compression Methods

Section 2.1 allowed reviewing all the important notions to know about the tasks, datasets, networks and training methods before dealing with the topic of neural networks compression and the rest of this manuscript. This section will be first dedicated to expose the general domain of compression, as well as its main methods. Pruning will then be tackled separately in Section 2.2.2.

2.2.1 Other Compression Methods

2.2.1.1 Considerations about Compression in General

The overall goal of compression is to produce networks that are both achieving good performance for their task and efficient enough not to exceed a certain capacity in computation power, memory requirement or energy consumption. To phrase it differently: compression involves finding the pareto optimum between various types of performances and various types of costs; *e.g.* the Top-1 accuracy of a classification network dependending on its number of parameters, the number of operations, the latency or the energetic consumption.

This objective contains a fundamental ambiguity between minimizing the size of a performing network and maximizing the performance of a cheap one. Even though the term "compression" would more intuitively point out to the first case, this ambiguity is still to be found in many methods.

Indeed, if neural architecture search can be seen as looking for the best available network in a given search space—thus leaning for the second case—, pruning, even though it is more explicitly a compression method, can also be seen as a form of architecture search. Distillation, which involves using an expensive and performing network to train another smaller one, can both be seen as transferring the knowledge of the first one into the smaller one—which would be compression—and as improving the performance of the second one—which is the second case.

Another aspect that adds to the confusion is the need to choose an adequate network to compress. Indeed, it is not always guaranteed that a compressed network performs better than a normal one that is smaller from the start [57]; therefore, compression implies beforehand the search for an already optimized architecture—that are themselves more difficult to compress.

Also, even different variants from a same family of methods can be considered as corresponding to either of the two cases. Indeed, as compression can be seen as a form of constraint applied to the network and its training, it is either possible to train normally a network and compress it afterward or to apply the said penalty during or even before training—in which case the goal is to maximize performance under this constraint.

It therefore appears that the very notion of compressing neural networks actually leads to tackle subjects that are way wider that simply reducing the cost of a well performing network.

On top of that, two more aspects complete the entanglement of the whole domain of compression. First: not only does every method bring different types of gains, which implies that they should be combined, but their combination can have effects that are sometimes difficult to predict, as they can each degrade performance differently or make the network more difficult to compress for other methods. This means that combining different compression methods is not only necessary, but also highly combinatorial and subject to a certain trade-off, where it may not be possible to reduce every type of cost at once.

The second aspect is that there are some theoretical links between different families of compression methods. If pruning neurons can be seen as a form of architecture search, pruning weights can be seen as imposing a constraint on the values of weights, alike quantization or clustering.

Therefore, speaking solely of only one domain of compression in isolation can only be an assumed, but arguably necessary bias for the sake of intelligibility. This is why this thesis focused solely on neural networks pruning and on studying in depth every of its aspects, in order to draw as much knowledge as possible out of this field. Yet, it does not prevent acknowledging that there are countless bridges with many other domains and that the notions that will be tackled in this manuscript can be useful for aspects of deep learning that are very far from pruning.

2.2.1.2 Neural Architecture Search

Sections 2.1.3 and 2.1.4.1 provided an overview of the different operators and practices involved in designing efficient and performing neural networks. However, most of the cited networks are designed by hand and the field of Neural Architecture Search (NAS) aims at providing methods to produce such architectures automatically.

NAS methods can be considered as the combination of three elements: a search space, a search strategy and a performance estimation strategy [58].

The search space defines the family of architectures among which to find the best suited for a task. NAS methods first tackled search spaces involving only simple, straightforward networks in which the only available degrees of liberty were the depth, the types of layers and the hyperparameters of these layers [59]. They then extended the search space to multi-branch architectures including ResNet-like or DenseNet-like residual connections [60]. Instead of searching for whole architectures, it is also possible instead to generate blocks, that can be repeated to produce architectures that can be scaled more easily to fit various datasets [61].

The search strategy defines how to explore this space. The literature has tested many different methods, such as reinforcement learning [59], neuro-evolutionary algorithms [62] or bayesian optimization [63]. The performance estimation defines the objective to optimize while crawling the search space. Usually, the metric to maximize is the test accuracy of the network. However, as it takes too much time to fully train a network and get its accuracy, multiple proxies have been used, such as extrapolating accuracy from a shorter or sub-optimal training [64] or predicting it directly from the architecture [65]. Another possibility is to consider the search space as the set of the subgraphs of a single big network [66], which allows training only one model and testing separately its subgraphs. This last approach is more reminiscent of pruning, especially structured pruning.

2.2.1.3 Quantization

The goal of quantization is to reduce the number of bits with which the weights of a networks are represented. The interest is twofold: first, the weights occupy less memory space and require fewer memory accesses; second, they may involve simpler operations, which reduces both the latency and the energy dissipation. Quantization involves mainly three aspects: the number of bits to quantize weights, whether or not to also quantize intermediate representations and whether to quantize after training or to do a quantization-aware training.

Weights are usually encoded as *float32*, but it is very simple to convert them to *float16* or even *bfloat16* without decreasing too much performance. Fixed-point or integers are also possible, but if the new encoding struggles too much to represent the original data, it may be preferable to apply instead various kinds of non-uniform, asymmetric or scaled quantization [67].

However, when aiming for low-bit precision, quantization-aware training yields better results than post-training quantization. For example, *BinaryConnect* [68] produces networks with weights having only two possible values, -1 or 1, using the straight-through estimator [69] to allow training directly the binarized network.

It is possible to quantize only weights, which allows reducing their memory occupation, but does not allow simplifying the operations of the network. This is why it may be preferable to quantize also intermediate representations (or "activations") [67], [70].

2.2.1.4 Clustering

Clustering is a more marginal compression method, that involves finding a representation that allows reconstructing an approximation of a network's weights. Many methods [71]–[73] are based on product quantization (PQ) [74], [75], that consists in splicing the weights' tensor into column and performing k-means clustering on each of them, so that an approximation of each column can be reconstructed from the centroids. In general, clustering reduces the memory occupation of weights, but do not reduce the number or the complexity of the operations they are involved with.

2.2.1.5 Distillation

The principle of knowledge distillation [76] is very different from the other compression methods. The basic idea of distillation is that the soft labels of a properly trained network may actually be a better classification than the usual one-hot labels of classification datasets, since these soft labels include information about inter-class similarities [77]. Therefore, the output of a "teacher" network can be used as the groundtruth label to train a "student" network.

Because of this principle, and even though the most usual case is the one where the student is smaller than the teacher, it is very possible for the student and the teacher to be of the same architecture, which can bring a slight improvement in accuracy [78].

Most of the time, the teacher's knowledge is used as a sort of regularization, conjointly to the original one-hot labels of the labels, during training. Many contributions have extended the principle of distillation not only to the teacher's output, but also to its intermediate representations [79]–[81].

Distillation has numerous uses outside of compression, for example for cross-modal learning [12].

2.2.2 Pruning

Now that we have reviewed the overall domain of compression, we will present the main aspects to be aware of before delving into the details of neural network pruning.

2.2.2.1 Principle

The basic idea of pruning is to reduce various types of costs of a network by removing some of its parts. This straightforward and intuitive principle raises three questions:

- What types of parts can be removed from the network and how does removing them impact its cost?
- How to tell which parts are the most relevant to remove to reduce performance the least?
- What is the best way to remove these parts?

Each of these questions leads to a different aspect of pruning, respectively: the pruning structure, the pruning criterion and the pruning method.

2.2.2.2 Structures

The pruning structure is a very important aspect of pruning, since it defines:

- what type of cost pruning will reduce;
- how easy it will be to leverage pruning;
- what is the granularity of pruning and how easy it will be to prune a lot without impacting performance.

Indeed, the most basic type of pruning, which also yields the best performance-to-parameters trade-off and is very simple to implement, involves pruning weights in a non-structured way [26]—hence the name "non-structured pruning". However, this type of pruning has serious drawbacks: 1) it produce sparse weights tensors that are difficult to leverage properly [82] and 2) it does not reduce the size of weights tensors, which means that the size of the intermediate representations is not reduced either.

This is why many papers prune whole convolution filters [83], which is called "structured pruning"—rather improperly, as it is not the only existing type of structures that can be pruned. Even though this coarser granularity leads to a less favorable performance-to-parameters trade-off, it has the clear advantage of: 1) actually reducing the size of the weights tensor, which can be leveraged on any hardware and framework and 2) reducing the size of the intermediate representations, which saves runtime memory usage.

2.2.2.3 Criteria

Once the pruning structure has been chosen, the pruning criterion defines which parameters or which filters are the most relevant to prune. Usually, the underlying idea is to remove parts that are the least useful, *i.e.* those that contribute the least to the network's accuracy. However, since manually measuring the impact of the removal of each parameter individually is impossible for networks that count millions of parameters, pruning methods instead use metrics that are meant to predict the importance of a parameter. The two main types of pruning criteria are those based on the magnitude

of a weight (*i.e.* its absolute value) [26], [84] and those based on the magnitude of its gradient [85], [86].

The other important aspect of pruning criteria, besides the metric, is how to adapt it to groups of weights. Mainly two tactics are used to adapt the aforementioned metrics to filters: 1) considering the norm of the metric over the whole filter [83] or 2) using a proxy, such as the multiplicative learned weight of the following batch-normalization layer, whose importance we assume to be proportional to that of the corresponding filter upstream [87].

Finally, the last aspect of pruning criteria is how to distribute pruning across the layers. Indeed, it is possible to prune all layers the same way or to define manually the pruning rate of each layer [83] as well as it is possible to set a global target [26]; these two versions can be called "local" and "global" pruning. Also, because of the varying size of filters between different layers, it is also necessary to choose a way to define the actual pruning rate, since different papers will either consider the proportion of pruned filters or the resulting number of remaining parameters.

2.2.2.4 Methods

The pruning method defines how to actually remove the weights or filters targeted by the pruning criterion. Pruning method mainly combine two aspects: 1) finding the best way to remove a weight or a filter and 2) defining when it is the best to prune relatively to the training schedule. These two aspects cannot always be clearly separated, even though we will for the sake of convenience.

The removal strategy involves how to actually remove a weight, whether it is definitely or if it can regrow. Most simple methods imply setting weights definitely to zero [26] but others either imply an explicit regrowth strategy [88], a dynamic reparameterization of a mask [89] or even removal through a continuous and soft penalty [1].

The training schedule defines when to prune relatively to training. A basic example would be to train a network and then iterate between pruning and fine-tuning steps [26], but other methods prune progressively throughout training [90] or replace fine-tuning by re-training, while reinitializing the value of weights [91] or not [92].

2.2.2.5 Contributions

Finally, this thesis has brought several contributions. Since they are sometimes used conjointly, we will review them briefly to help understanding their principle before their accurate description in the dedicated sections.

- Selective Weight Decay (SWD) [1]: a pruning method, that removes weights all throughout training using a growing penalty on the targeted weights until their value reaches 0. SWD has the advantage to be easily adaptable to any pruning structure and criterion. Its continuous aspect, combined with its ability to adapt its pruning target during training, allows for an efficient pruning even at very high compression rates.
- Dimensional Clear-Out (DCO) [3]: structured pruning, by altering the input/output dimensions of layers, imply many dependencies between layers. These interactions can make counting the actual count of remaining parameters difficult and can prevent inference because of some broken operations; *e.g.*, additions in residual blocks break when the two tensors to sum are not of the same dimensions

anymore. To solve these problems, DCO identifies all dependencies between layers and operations, removes all the remaining weights that were disconnected in the wake of pruning and adapts all the operations, that are susceptible to break, in order to allow the inference of any structured pruning distribution.

- Measurement of the gains of structured pruning, using DCO, on GPU [2]. This study highlighted how the cost metric influences the perceived efficiency of pruning and how imbalance in pruning criteria may harm its performance on hardware.

2.3 Recapitulation

In this chapter, we introduced all the background necessary to understand fully the next chapters. We reviewed what are image classification and semantic segmentation and which datasets we used in out experiments. We also showed what was a classification or segmentation network in general, how the history of their respective literature brought practices that are still used today and can be found in the networks we actually use in this manuscript. We also introduced all the types of layers involved in these networks and how to train them.

Finally we reviewed the domain of neural networks compression in general, its principle, its main methods and what is the one we focus on in this thesis: pruning. We presented what is pruning and its main notions and we briefly presented what are the contributions that this thesis brought.

In the following chapters, we will expand on the same aspects of pruning we mentioned: pruning structures, pruning criteria and pruning methods—not necessarily in this order, but rather the one that allows the best to articulate the different contributions together. In each chapter, we will first review in depth the related works and either discuss it or present our corresponding contribution.

Chapter 3

Pruning Criteria

Contents

This chapter will be dedicated to pruning criteria, i.e. metrics and strategies to identify the weights to prune in a given network. We will thematically tackle different aspects of criteria, such as the metrics to evaluate the importance of a parameter, the strategies to extend them to groups of parameters or the distribution of the pruning rate across layers. Finally, we will expose another way of representing the effect of pruning on the performance of the network, which will evidence one completely different way of considering the question of pruning criteria.

3.1 Identifying Unimportant Weights

In this section, we will review the main types of importance metrics for individual weights in neural networks. We will first tackle the two most widespread ones—weight and gradient magnitude—and detail their dedicated literature, and then we will tackle other, more exotic criteria.

3.1.1 Gradient Magnitude, a.k.a. "Saliency"

Even though the gradient magnitude criterion is not as widespread as the weight magnitude one, it is useful to present this one first to better understand the other. We regroup under the term "gradient magnitude criteria" all criteria, to be found in the literature, that somehow use the loss gradient, multiplied or not with the value of weights themselves—in this case, the criterion is usually called "saliency".

This type of criterion is one of the most ancient ones; Reed *et al.* [93] cited in 1993 already three influential works by Mozer and Smolensky [94], Le Cun *et al.* [85] and Karnin [95]. Multiple works, in the following years or even some more recent ones, built upon this basis or reused the same notations and understanding [27], [96]–[98]. We will now review these different ways of understanding the gradient magnitude criterion, that were presented in these papers. We will then synthesise them and tackle some eventual problems that must be discussed, in Section 3.1.1.7.

3.1.1.1 Skeletonization and SNIP

In 1988, Mozer and Smolensky presented a pruning method, called "Skeletonization"; to our knowledge, this is the first occurrence of gradient-based estimation of weights importance. The starting point of their reasoning is that one might want to remove preferentially weights of least "relevance"; they mention that the magnitude of weights is not necessarily faithful to its relevance, as *the effect of these connections may cancel each other out* (direct citation)—for reasons that are made more or less obsolete now by the use of batch-normalization [31]. This relevance ρ is defined as the difference in the network's error before and after the removal of a weight w, which can be roughly noted (we re-adapt the original notations to be more consistent with our own): $\rho_w = \mathcal{L}_{\setminus w} - \mathcal{L}$. Evaluating this relevance would require doing a test on the whole dataset for every weight w in the network, which would be unfeasible; this is why they proposed a method to evaluate a weight's relevance.

The first step of their reasoning is to suppose the existence of coefficients $\boldsymbol{\alpha}$ that represent the *attentional strength* of weights $\mathbf{w}$ (notation as a vector instead of tensor); in the case of a purely linear layer with an output $\mathbf{y}$, an input $\mathbf{x}$ and an activation function f, we have: $\mathbf{y} = f((\mathbf{w} \odot \boldsymbol{\alpha}) \cdot \mathbf{x})$. Values α in $\boldsymbol{\alpha}$ range from 0 (deactivated) to 1 (functional). The relevance of a weight w of attentional strength α can then be redefined as $\rho_w = \mathcal{L}_{\alpha=0} - \mathcal{L}_{\alpha=1}$.

Using this notation, the importance of ρ_w is then approximated using the derivative of the error $\mathcal{L}$ with respect to α:

$$\lim_{\gamma \to 1} \frac{\mathcal{L}_{\alpha=\gamma} - \mathcal{L}_{\alpha=1}}{\gamma - 1} = \left.\frac{\partial \mathcal{L}}{\partial \alpha}\right|_{\alpha=1}$$

Assuming that this holds approximately for $\gamma = 0$, Mozer and Smolensky then notate:

$$\frac{\mathcal{L}_{\alpha=0} - \mathcal{L}_{\alpha=1}}{-1} \approx \left.\frac{\partial \mathcal{L}}{\partial \alpha}\right|_{\alpha=1}, \text{ therefore } \rho_w \approx -\left.\frac{\partial \mathcal{L}}{\partial \alpha}\right|_{\alpha=1} \tag{2}$$

Then, Mozer and Smolensky state that *this derivative can be computed using an error propagation procedure very similar to that used in adjusting the weights with back propagation* (direct citation), and since all α are assumed to be 1, they do not need to be inserted as real parameters—they are only for notational convenience.

Even though Mozer and Smolensky give further details on how they exactly compute this relevance—using a linear error function instead of quadratic and accumulating relevance across batches as an "exponentially-decaying time average" to cope with the strong fluctuations of the derivative—, what we presented above is all we need to know in the scope of this section.

Lee *et al.* [99] use almost exactly the same reasoning for their method called SNIP, separating the value of a weight and its indicator variable (that they notate c instead); Lee *et al.* justify the need to separate the two with the following arguments: *this formulation ($\partial\mathcal{L}/\partial c_j$) can be viewed as perturbing the weight w_j by a multiplicative factor δ and measuring the change in loss. [...] Furthermore, $\partial\mathcal{L}/\partial c_j$ is not to be confused with the gradient with respect to the weights ($\partial\mathcal{L}/\partial w_j$), where the change in loss is measured with respect to an additive change in weight w_j* (direct citation [99]).

3.1.1.2 Optimal Brain Damage

Starting from the same motivation as Mozer and Smolensky [94], Le Cun *et al.* [85] use a slightly different approach to approximate the contribution of a parameter w to the error function. To this mean, Le Cun *et al.* approximate the variation $\delta\mathcal{L}$, consecutive to a modification $\delta\mathbf{w}$, using a Taylor series:

$$\delta\mathcal{L} = \sum_i g_i \delta w_i + \frac{1}{2}\sum_i h_{ii}\delta w_i^2 + \frac{1}{2}\sum_{i \neq j} h_{ij}\delta w_i \delta w_j + \mathcal{O}(\|\delta\mathbf{w}\|^3) \tag{3}$$

$$\text{with } g_i = \frac{\partial \mathcal{L}}{\partial w_i} \text{ and } h_{ij} = \frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}$$

This Equation 3 introduces two new components: the gradient tensor $\mathbf{G}$ and the Hessian $\mathbf{H}$ of $\delta\mathcal{L}$ with respect to $\mathbf{w}$. Because of the complexity of computing $\mathbf{H}$, Le Cun *et al.* propose three approximations to simplify the equation; discussing these approximations will be the focus of some papers afterwards. These are the three approximations (mostly direct citations from the original paper [85]):

- *The "diagonal" approximation: the $\delta\mathcal{L}$ caused by deleting several parameters is the sum of the $\delta\mathcal{L}$'s caused by deleting each parameter individually; cross terms are neglected, so third term of the right hand side of Equation 3 is discarded.* As highlighted by Hassibi and Stork [96], this assumption, despite its convenience, is questionable; in the case of the linear operations used in neural networks, there is no reason for cross-terms not to be as significant as diagonal ones.

- *The "extremal" approximation: parameter deletion will be performed after training has converged. The parameter vector is then at a (local) minimum of $\mathcal{L}$ and the first term of the right hand side of Equation 3 can be neglected. Furthermore, at a local minimum, all the h_{ii}'s are non-negative, so any perturbation of the parameters will cause $\mathcal{L}$ to increase or stay the same.*
- *The "quadratic" approximation: the cost function is nearly quadratic so that the last term in the equation can be neglected.*

These three approximations allow simplifying Equation 3 as:

$$\delta\mathcal{L} = \frac{1}{2}\sum_i h_{ii}\delta w_i^2$$

Therefore, Le Cun *et al.* [85] define the saliency of weights as:

$$s_k = \frac{h_{kk}w_k^2}{2}, \tag{4}$$

so that weights of least saliency can be removed. They also propose a way of quickly computing the second derivative h_{kk} [100], which we will not detail.

3.1.1.3 Optimal Brain Surgeon

Hassibi and Stork [96] noted that, in opposition to the diagonal approximation, *Hessians for every problem (they) have considered are strongly non-diagonal*. Since they keep the extremal approximations and ignore terms beyond the second order, they actually consider the following equation (with a slightly rearranged notation compared to Le Cun *et al.* [85]):

$$\delta\mathcal{L} = \frac{1}{2}\delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w}, \text{ with } \mathbf{H} = \frac{\partial^2\mathcal{L}}{\partial\mathbf{w}^2} \tag{5}$$

They then define the unit vector $\mathbf{e}_k$ (to designate the weight to prune) and formalize the pruning problem as:

$$\min_k \left\{ \min_{\delta\mathbf{w}} \left\{ \frac{1}{2}\delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w} \right\} \text{ such that } \mathbf{e}_k^T \cdot \delta\mathbf{w} + w_k = 0 \right\}$$

From this equation, they deduce a saliency metric defined as:

$$s_k = \frac{1}{2}\frac{w_k^2}{[\mathbf{H}^{-1}]_{kk}}, \tag{6}$$

so that this saliency gives *the increase in error that results when the weight is eliminated*, which means that *if this candidate error increase is much smaller than $\mathcal{L}$, then the k^{th} weight should be deleted* (direct citation [96]). Hassibi and Stork also provide a way to update all weights $\mathbf{w}$, for a given pruned k, in order to compensate the modification, using the inverse Hessian. They also provide a method to compute the inverse Hessian. This method is used in some modern papers [101].

3.1.1.4 Early Brain Damage

Tresp *et al.* [102] propose to instead question the extremal approximation, notably in order to apply it when using *early stopping*. While this could sound specific, since many pruning method apply their criterion while training (cf. Chapter 4), studying the case

where the network has not fully converged yet is actually relevant. Tresp *et al.* denote $\mathbf{w}^*$ as the converged $\mathbf{w}$ and define the saliency:

$$s_k = \underbrace{\frac{1}{2}\frac{\partial^2 \mathcal{L}}{\partial {w^*}_k^2}{w^*}^2_k}_{\text{OBD}} = \underbrace{\frac{1}{2}\frac{\partial^2 \mathcal{L}}{\partial w_k^2}w_k^2}_{A_k} + \underbrace{\left(-\frac{\partial \mathcal{L}}{\partial w_k}w_k\right)}_{B_k} + \underbrace{\frac{1}{2}\left(\frac{\partial^2 \mathcal{L}}{\partial w_k^2}^{-1}\right)\left(\frac{\partial \mathcal{L}}{\partial w_k}\right)^2}_{C_k} \tag{7}$$

This Equation 7 approximates before convergence the saliency of Le Cun *et al.* [85] (cf. Equation 4, alias "OBD"), that is computed after convergence, using the second order derivative (similarly to OBD) A_k, the first order derivative B_k (similarly to Mozer and Smolensky [94], cf. Equation 2), because of the absence of the extremal approximation and finally a new term C_k. The motivation behind this term C_k is that $A_k + B_k = \mathcal{L}_{w_k} - \mathcal{L}_{w_k=0}$ while $A_k + B_k + C_k = \mathcal{L}_{w_k^*} - \mathcal{L}_{w_k=0}$. This way, the saliency defined in Equation 7 is supposed to predict if a given weight will be relevant after convergence.

3.1.1.5 First Order Saliency

While previous presented papers dated from the 80's or 90's, some more recent papers either build upon these methods or propose a new way of thinking them. Molchanov *et al.* [86], [97], similarly to Le Cun *et al* [85], use a Taylor expansion to define the saliency of a parameter. Let's dwell into their reasoning.

Actually, instead of dealing with weights themselves, Molchanov *et al.* [86] actually consider the output h_i produced by each weight w_i—which is rather equivalent, but let's stick to their own formalism. Their motivation is to approximate $|\Delta\mathcal{L}(h_i)| = |\mathcal{L}(h_i = 0) - \mathcal{L}(h_i)|$, assuming that all parameters are independent. They then remind what is the Taylor expansion of a function $f(x)$ in the proximity of $x = a$:

$$f(x)\big|_{x=a} = \sum_{p=0}^{P} \frac{f^{(p)}(a)}{p!}(x-a)^p + R_p(x), \tag{8}$$

which, at the first order, gives:

$$f(x)\big|_{x=a} = f(a) + \frac{\delta f(a)}{\delta x}(x-a) + R_1(x)$$

They then use this expansion to get:

$$\mathcal{L}(h_i = 0) = \mathcal{L}(h_i) - \frac{\delta \mathcal{L}}{\delta h_i}(h_i)h_i + R_1(h_i = 0),$$

which they finally re-inject in their definition of $|\Delta\mathcal{L}(h_i)|$ to get:

$$|\Delta\mathcal{L}(h_i)| = \left|\mathcal{L}(h_i) - \frac{\delta \mathcal{L}}{\delta h_i}(h_i)h_i - \mathcal{L}(h_i)\right| = \left|\frac{\delta \mathcal{L}}{\delta h_i}(h_i)h_i\right| \tag{9}$$

The remainder $R_1(h_i)\big|_{h_i=0}$ was ignored because it mainly involved second order derivative while *the widely-used ReLU activation function encourages a smaller second order term* (direct citation [86]).

Molchanov *et al.* [86] then provide a justification to the use of gradient while a network, that has converged enough, should have zero or near-zero gradient. Indeed, they affirm that while $\mathbb{E}\left(\frac{\delta \mathcal{L}}{\delta h_i}(h_i)h_i\right) = 0$, the absolute value involves that $\mathbb{E}\left(\left|\frac{\delta \mathcal{L}}{\delta h_i}(h_i)h_i\right|\right) = \sigma\sqrt{2}/\sqrt{\pi}$, with σ the standard deviation. Therefore, their definition of saliency actually measures the variance of $y = \frac{\delta \mathcal{L}}{\delta h_i}(h_i)h_i$: *while y tends to zero, the expectation of $|y|$ is proportional to the variance of y, a value which is empirically more informative as a pruning criterion* (direct citation [86]).

3.1.1.6 Pruning at Initialization

Finally, the last group of papers that rely on the gradient criterion belongs to the literature of pruning at initialization (which we will tackle in Chapter 4). Tanaka *et al.* [98] list multiple definitions of saliency from different papers, including those we already mentioned, and give this list: Skeletonization [94] $-\frac{\partial \mathcal{L}}{\partial w} \odot w$, SNIP [99] $\left|\frac{\partial \mathcal{L}}{\partial w} \odot w\right|$, GraSP [103] $-\left(H \frac{\partial \mathcal{L}}{\partial w} \odot w\right)$ (even though the actual GraSP paper does not use the second order derivative H and explicitly cite Molchanov *et al.* [86] as a direct influence), "Taylor-FO" [97] $\left(\frac{\partial \mathcal{L}}{\partial w} \odot w\right)^2$ (based on Molchanov *et al.* [86] too) and Optimal Brain Damage [85] $diag(H)w \odot w$.

We already studied each of these saliencies and described the theoretical reasoning behind each of them. We can now say that we identify mainly three seminal and parallel demonstrations for using gradient in the evaluation of the importance of a weight: Mozer and Smolensky [94] (a.k.a. "Skeletonization"), Le Cun *et al.* [85] (a.k.a. "Optimal Brain Damage" or OBD) and Molchanov *et al.* [86].

3.1.1.7 Synthesis and Discussion

We just reviewed multiple papers, each providing their own way of estimating the saliency of a network; this saliency always revolves around both a weight's magnitude and the first or second order derivative of the error with respect to the said weight. It is possible now to synthesize all these theoretical demonstrations to explain what seem to be the main arguments in favor to this saliency metric and what may be its pitfalls.

The starting point of all these discussions is that the weights that we want to remove are those whose removal induce the least difference in the network's performance (we do no consider cases where pruning may improve performance):

$$s(w_i) = |\Delta \mathcal{L}(w_i)| = |\mathcal{L}(\mathbf{w} \setminus w_i) - \mathcal{L}(\mathbf{w})| \tag{10}$$

The problem is that measuring empirically such a saliency would require measuring $\mathcal{L}(\mathbf{w} \setminus w_i)$ for each w_i, which is prohibitive; moreover, as weights are heavily dependent, one would have to recompute the saliency of all weights after each removal of one weight. This is why it is absolutely necessary to instead define a criterion that is able to reliably predict a weight's importance.

Two strategies to approximate this $\Delta \mathcal{L}(w_i)$ co-exist:

First Strategy: We introduce an importance factor $\boldsymbol{\alpha}$, which is virtually binary between 0 and 1, such that we instead consider $\mathcal{L}(\boldsymbol{\alpha} \odot \mathbf{w})$. Therefore, we want to approximate $s(w_i) = |\mathcal{L}(\alpha_i = 0) - \mathcal{L}(\alpha_i = 1)|$. This difference is then approximated using the derivative:

$$\mathcal{L}(\alpha_i = 0) - \mathcal{L}(\alpha_i = 1) \approx -\frac{\partial \mathcal{L}}{\partial \alpha_i}$$

Intuitively, the derivative $\partial \mathcal{L} / \partial \alpha_i$ can be seen as the slight variation $\delta \mathcal{L}$ induced by a slight variation $\delta \alpha_i$. Since the other, more commonplace, strategy does not introduce any $\boldsymbol{\alpha}$, and since, after all, it is expected for α_i and w_i to have very analogous behaviors, let's see more in details what this gradient actually equates to. Let's consider $f = \mathcal{L}$ and $g(\alpha) = w_i \alpha_i$, we have:

$$\frac{\partial \mathcal{L}}{\partial \alpha_i} = (f \circ g)'(\alpha_i) = g'(\alpha_i) \cdot f'(g(\alpha_i)) = w_i \cdot f'(g(\alpha_i)),$$

and since all α equal 1, we have $g(1) = w_i$ and:

$$w_i \cdot f'(g(\alpha_i))\big|_{\alpha_i=1} = w_i \frac{\partial \mathcal{L}}{\partial w_i}$$

We can notice that this definition gives exactly the definition that Tanaka *et al.* [98] gave of Skeletonization [94]. Now that we have defined this saliency without the use of $\boldsymbol{\alpha}$, it is easier to interpret.

This whole demonstration indicates that we implicitly did this approximation:

$$s(w_i) = |\mathcal{L}(\mathbf{w} \setminus w_i) - \mathcal{L}(\mathbf{w})| \approx \left| w_i \frac{\partial \mathcal{L}}{\partial w_i}\bigg|_{w_i} \right|,$$

which supposes that, at least partially, the slight variation $\delta\mathcal{L}$ induced by a slight variation δw_i evaluated at w_i is a good approximation of the whole variation $\Delta\mathcal{L}$ induced by the variation from w_i to 0. In the case of a strictly linear model, the derivative at one point gives the slope of the whole function, so we literally have $\mathcal{L}(w_i) - \mathcal{L}(w_i = 0) = w_i \cdot \delta\mathcal{L}/\delta w_i$; networks that use ReLU activation functions indeed behave as a piecewise linear function, which means that we indeed have $\mathcal{L}(w_i + \delta w_i) - \mathcal{L}(w_i) = \delta w_i \cdot \partial\mathcal{L}/\partial w_i$. However, as soon as δw_i gets too big, $\mathcal{L}$ cannot be considered linear anymore and we cannot know if the derivative at w_i can be a good approximation at all.

Therefore, we can tell that:

$$s(w_i) = \left| w_i \frac{\partial \mathcal{L}}{\partial w_i}\bigg|_{w_i} \right|, \text{ as long as } w_i < \epsilon, \tag{11}$$

which means that this criterion is not expected to give relevant results when the pruning rate gets too high and the magnitude of weights gets too large. Reciprocally, as long as the pruning rate stays low, since this saliency tends to favor pruning weights with low magnitude first, it happens by luck that this definition stays true.

Another aspect that mitigates the relevance of this reasoning is that it implicitly supposes that all weights are independent; as we will see when detailing the second strategy, this dependence is something to discuss and that can be neglected or not.

Second Strategy: This strategy involves introducing a Taylor expansion, detailed in Equation 8, to make some derivative appear, that could be computed using gradient descent. In order to make all kinds of dependencies between weights appear in this expansion, we will consider $\mathcal{L}$ depending on the whole weight vector $\mathbf{w}$ instead. We do the Taylor expansion at the vicinity of a certain point $\mathbf{a}$, and we only expand up to the second-order derivative. Also, we do not approximate $\mathcal{L}(\mathbf{w})$ by $\mathcal{L}(\mathbf{a})$ but the reverse:

$$\begin{aligned}\mathcal{L}(\mathbf{a})\big|_{\mathbf{w}=\mathbf{a}} &= \mathcal{L}(\mathbf{w}) + \frac{\partial \mathcal{L}}{\partial \mathbf{w}}(\mathbf{a} - \mathbf{w}) + \frac{1}{2}(\mathbf{a} - \mathbf{w})^T \frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}^2}(\mathbf{a} - \mathbf{w}) + R_2(\mathbf{w} = \mathbf{a}) \\ &= \mathcal{L}(\mathbf{w}) + \sum_i \frac{\partial \mathcal{L}}{\partial w_i}(a_i - w_i) + \frac{1}{2}\sum_i \sum_j \frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}(a_i - w_i)(a_j - w_j) + R_2(\mathbf{w} = \mathbf{a})\end{aligned} \tag{12}$$

When considering $\mathbf{a} = \mathbf{w} + \delta\mathbf{w}$, we find back Equations 3 and 5. However, whatever the point $\mathbf{a}$ we choose, we have the same problem: this expansion only works in the vicinity of that point, which is exactly the same problem as Strategy 1.

Therefore, let's pose this condition: we consider a case where we want to remove some weights w_k with $k \in K$ a list of indices, and these weights have a magnitude close enough to 0 so that putting them to zero can be seen as a slight change, so that $\mathbf{w}_{w_k=0,\ k\in K} = \mathbf{w} + \delta\mathbf{w}$, since $w_k < \epsilon$. Thanks to this condition, we can inject this development back into Equation 10, so that the $\mathcal{L}(\mathbf{w})$ terms cancel each other—which is only possible because we consider $\mathbf{w}$ to be in the vicinity of $\mathbf{w}_{w_k=0,\ k\in K}$. We get, for any k in K:

$$\forall k \in K,\ s(w_k) = \left| -\frac{\partial \mathcal{L}}{\partial w_k} w_k + \frac{1}{2} \sum_{i \in K} \frac{\partial^2 \mathcal{L}}{\partial w_k \partial w_i} w_k w_i + R_2(w_i = 0,\ i \in K) \right|$$

We can see that, in the second-order term, only weights that are to be modified appear, which means that if the modification only involves one weight, they are virtually independent. If several weights are to be pruned at the same time, all possible combinations of indices K are to be measured, which can be expensive.

However, one thing which may allow simplifying the equation is that, as we already mentioned, ReLU networks are locally linear, which means that for a small enough $\delta\mathbf{w}$, second-order terms (and further) can be neglected, which gives:

$$s(w_i) = \left| \frac{\partial \mathcal{L}}{\partial w_i} w_i \right|,$$

that is exactly the same as Equation 11, with the same proximity condition.

Conclusion: Our two strategies, that take slightly different paths, both converge toward the same metric with the same constraints for similar reasons. Indeed, approximating any $\Delta\mathcal{L}$ with a $\delta\mathcal{L}$ requires proximity conditions for three reasons: 1) obviously, it is easier to make this approximation if $\Delta\mathcal{L}$ is actually a $\delta\mathcal{L}$, 2) for small enough modifications, the network is linear (in the case of ReLU activations) and approximation through gradient becomes exact and 3) for modifications that are too large, not only this approximation is not exact anymore, but all previous guarantees become obsolete and we cannot possibly ignore second-order (or even further) derivatives; furthermore, Taylor expansion, by definition, is only valid in the vicinity of a point, so using it in another context can only be dubious. All of this means that such a criterion only works for pruning rates that are small enough to only target weights that are close to zero; beyond a certain threshold (that is undetermined for now), its fundamental assumptions become false.

Another big question to answer is: what happens when the network is at the optimum, where the gradient should be zero? Of course, it would be absurd for a zero gradient of all weights to signify that all weights can be removed without any change in the loss. Here two cases are possible: either the optimum is only a point and the zero gradient is just a limit, while any non-zero change δw introduces a non-zero change $\delta\mathcal{L}$; or the optimum is a whole plateau that, if it includes values up to zero for some of the weights, means that such weights can effectively be removed without moving from the optimum. However, in this last case, the zero gradient would put the saliency of all weights to zero, without distinction between those that are close enough for the proximity constraint to hold true or those that are too large to expect that putting them to zero stays in that optimum plateau. Once again, the proximity constraint is necessary for the saliency to make sense.

However, in practice it is very rare to attain and stabilize over the absolute optimum of the function, or even any local minimum, in such a way that the gradient stays

zero—partly because of momentum. Nonetheless, even if the optimum of the error is reached, two aspects prevent the gradient to be perfectly zero: 1) weight-decay forces weights, whose non-zero value is necessary, to have a non-zero gradient to compensate its penalty and 2) since the gradient is, most often, evaluated in a stochastic way, this means that it is actually the aggregation of the gradient over an ensemble of mini-batches, for each of which the corresponding $\mathcal{L}$ is likely to be slightly different, so that the overall optimum of the problem may not exactly coincide with that of the current batch, which therefore gives a non-zero gradient.

After this extensive study of the literature surrounding the saliency criterion, we will review the much shorter literature on the weight magnitude criterion.

3.1.2 Weight Magnitude

The principle of the weight magnitude criterion is extremely simple: pruning weights whose value $|w_i|$ is the smallest. This criterion is as ancient as the saliency one but has received way less theoretical interest so that, instead of attempts to provide formal demonstrations, we instead have to hunt evasive clues in various papers and formulate some hypotheses.

The earliest papers to be analogous to magnitude pruning are actually those that revolve around the idea of weight decay to either sparsify or, at least, minimize the "energy" of layers and activations [84]. One of the core ideas is that unimportant weights have an error gradient that is always small, so that this gradient is negligible compared to that of weight decay, which therefore drives unnecessary weights towards zero [104]. This is why these methods mostly propose new penalties to enforce sparsity [84], [105]–[109].

The more contemporary approach of simply pruning weights of least magnitude and fine-tuning the network afterward, that was introduced back by Han *et al.* [26] in 2015, could be found in some rare papers of this early era of pruning, such as that of Janowsky *et al.* [110] in 1989. However, none of these papers seem to bring theoretical justification to back up the magnitude criterion.

Nonetheless, an interesting work from Segee and Carter [111] does a comparison between saliency and magnitude, as soon as 1991, and find out that both are very correlated, and that this correlation is very stable before and after pruning. This is consistent with some of our observation in the previous section, where we found out that the saliency criterion made mostly sense for weights that are already small.

Finally, mainly three arguments come to mind to justify magnitude pruning, and their simplicity is surely a reason why many of these papers did not bother to justify their method:

- As already mentioned, unimportant weights have a small gradient that is negligible compared to weight decay. Therefore, unimportant weights become very small, but that does not mean that all small weights are unimportant; maybe the saliency metric could allow discriminating appart the false positives of this magnitude criterion.
- More simply, providing that weights and activations all contribute the same way—which is surely the "isotropic approximation" mentioned by Hassibi *et al.* [96], that is made acceptable partly thanks to the use of batch-normalization—, the $\mathcal{L}_2$ norm (or "energy", as mentioned by Chauvin *et al.* [84]) of output activations and weights are directly correlated, and therefore, pruning the smallest weights first

induces the least difference in $\mathcal{L}_2$ norm in both the weights and their output. This notion of square difference before and after pruning can be found in some papers, such as that of He *et al.* [112].

- Finally, the simplest reason is that the smallest weights are those for which being put to zero can be seen as a small, possibly non-significant modification; in this respect, this recalls the proximity constraint of saliency.

This magnitude criterion is now very widespread and almost considered as the default criterion. Blalock *et al.* [27] noted that magnitude-based methods tend to be, on average, more accurate than saliency-based ones, with some exceptions, notably for low compression rates—while performing worse wor high ones, which is consistent with our speculation that saliency could tell apart true and false positives among small weights.

3.1.3 Other Criteria

Even though the saliency and magnitude criteria are the two most widespread ones, with the latter being considered a standard, many criteria have been tested in the past decades. We will briefly review a non-exhaustive list of some notable examples here.

Pre-2015 papers:

- Sietsma and Dow [113], [114] remove "units" that either produce a constant output or whose output is either redundant with another unit or not correlated enough with the problem. Chung and Lee [115] have a rather similar approach, that involves manually defining a set of rules to tell how a weight can be of least importance.
- Many papers use genetic algorithm to choose which weights to remove [62], [116]–[118].
- Karnin [95] propose a variant around the idea of Mozer and Smolensky [94] by not considering only the derivative of the final trained network, but instead the accumulation of this saliency over all epochs, in order to have a more relevant estimate of the actual landscape of $\mathcal{L}$ and of the individual impact of each parameter.

Post-2015 papers:

- Average percentage of zeros in activations, by Hu *et al.* [119]; such a criterion is arguably made obsolete by the use of batch-normalization. Indeed, batch-normalization tends to drastically reduce the probability of a neuron to repeatedly have activations that are entirely empty; and concerning the proportion of zeroes in non-empty activations, it can be entirely predicted by the values of the parameters of the post-normalization affine function in batch-normalization layers.
- Multiple papers revolve around the idea of training separate pruning agents [120], for example using reinforcement learning [121] or attention statistics [122], [123].
- Many methods have a way to identify weights to prune that is inseparable from the method to remove them; for this reason we will review them in Chapter 4 instead.

- Some methods also use criteria that are pretty close to some used in the early days of pruning. For example, Anwar *et al.* use evolutionary methods [124] or even purely random masks [125]. Srinivas and Babu [126] intend to remove redundant neurons, which is reminiscent of Sietsma and Dow.
- Luo *et al.* [127] removes filters one by one, measuring each time which is the filter (among the remaining ones) that induces the least difference in the output of the following layer.

Post-2015, most of works centered around pruning criteria focus instead on how to generalize criteria to groups of weights for structured pruning—some of the aforementioned criteria are already intended to be applied to neurons/filters instead of weights. This is the topic we will tackle more specifically in Section 3.2.1.

3.2 Generalizing the Use of Criteria

The previous section discussed how to tell the importance of one weight. However, pruning almost systematically involves pruning multiple weights at once. Moreover, we need to know when to measure these criteria, relatively to training. Therefore, in this section, we will review the other three main aspects of pruning criteria: how to generalize them to structured pruning, how to distribute the pruning rate across layers and the relevance of pruning criteria throughout training.

3.2.1 Extending Criteria to Structures

As we will see in Chapter 5, many methods focus on pruning groups of weights, such as filters/neurons. In section 3.1.3 we already mentioned some criteria specifically designed to identify filters to prune, but they are rather marginal in the literature. Actually, the two most notable methods to design criteria for filters are either regrouping the criteria of the weights to group under a certain norm or to use a gate, such as the multiplicative weight in the affine part of batch-normalization layers, to characterize the importance of the whole channel. We will review the dedicated literature of both approaches in the following sections.

3.2.1.1 Norm-Based Approaches

Li *et al.* [83], one of the first influential works on structured pruning, already pruned filters, within each layer, on the basis of their $\mathcal{L}_1$ norm (*i.e.* the sum of their absolute value). Since then, many papers have evaluated the importance of a filter on the basis of the norm of the magnitude of its weights, using either the $\mathcal{L}_1$ norm [83], [128] or the $\mathcal{L}_2$ norm [129]–[133] (often through the use of weight-decay-like regularization). One advantage of regularization applied to structured pruning is that it allows regrouping weights into groups and then penalizing the norm of these groups, which has not the same behavior than penalizing all weights separately. Structured pruning through regularization of groups is an idea to be found as soon as 2016 in works such as that of Wen *et al.* [130].

Ye *et al.* [134] raise doubts about the relevance of norms and regularizaion to prune filters: *it has been an established practice to use tractable norm to regularize the parameters in optimizing a model and pick the important ones by comparing their norms after training. However, this assumption is not unconditional. By using Lasso or ridge regression to select important predictors in linear models, one always has to first normalize each predictor variable. Otherwise, the result might not be explanatory. [...] Such normalization condition for the right*

use of regularization is often unsatisfied for nonconvex learning (direct citation). Indeed, Ye *et al.* focus on the case of sparsity through regularization, and give, as examples, cases where the sparsifying effect of regularization is nullified by simply scaling the values of weights or using batch-normalization. However, the problem of norm-based methods goes beyond that.

Indeed, not only do filters count varying numbers of parameters, depending on the layer, but the average magnitude of parameters in a weight tends to depend on the dimensions of their layer, as pointed out by Tanaka *et al.* [98]. This means that it is very difficult to compare filters across layers on the basis of their norm. This difficulty to find a relevant metric to tell the importance of a filter solely based on the value of its weights is what led to the gate-based approaches.

3.2.1.2 Gate-Based Approaches

Gate-based methods [112], [134]–[138], instead of relying on a norm, approximate the importance of a filter through the magnitude of a multiplicative weight applied to the whole output of the involved filter. Either this gate is a distinct layer or directly the multiplicative weight in the affine part of batch-normalization layers, as done by Liu *et al.* [87]. Interestingly, Kruschke *et al.* [139] introduce "gains" as soon as 1988, to characterize the importance of a whole neuron, which is very similar.

The advantage of gates is that it negates multiple problems:

- Whatever their actual size, every filter is characterized by only one gate weight.
- Especially in the case of batch-normalization weights, the distribution of data right before this gate is supposedly following a centered-reduced Gaussian distribution, so that the average magnitude of weights in the filters is not expected to impact the range of the gate values.
- Such gates are very easy to insert anywhere, even at the end of residual blocks, which, as we will see in Chapter 5, can be used to give the importance of the same channel in multiple branches at once [87].

However, even though this eliminates many problems that introduced imbalance between layers, we still have no guarantee that they are perfectly comparable, especially since the conservation law of Tanaka *et al.* [98] remains. Some preliminary experiments of ours showed us that it was far easier to get a rather balanced global pruning when using gates than when using any type of norm. To conclude, let's mention that Kang *et al.* [138] mention that only considering the multiplicative weight in batch-normalization layers and not their bias too is problematic, while they include in their own method a metric that consider both; indeed, if we do the link with the percentage of zeros in activations, as we mentioned previously, both additive and multiplicative coefficients are necessary to predict the actual distribution of values in activations after the ReLU function.

3.2.2 Distribution of Sparsity Across Layers

We already briefly mentioned "local" or "global" pruning. This is a pretty important notion on which some discussion can be made. Very simply, "local" pruning denotes when each layer of a network is pruned either independently, or at least with its own pruning rate, while "global" pruning refers to the case where all layers are pruned using the same criterion, with a global pruning rate. A simple example to make this distinction more obvious: it is possible to prune 50% of the smallest weights of each

layer, which is local pruning; however, setting the same threshold for all weights in a network, so that 50% of all weights are removed is global pruning.

Obviously, there are some cases that are somewhat in-between these two cases. For example, many structured pruning methods struggle to find criteria that are balanced enough to apply them globally; however, they are perfectly conscious that setting the same pruning rate everywhere is not optimal. Therefore, some papers manually set a different pruning rate for each layer [83], which is technically speaking local pruning, even though it is not uniform. It is also possible to use a distinct metric to give the pruning rate of each layer, without it being global or manually set [140].

Another interesting nuance is that some layers can be pruned globally and others locally for a same pruning method. One relevant example, that we will show more deeply in Chapter 5, is the case where, to avoid introducing some discrepancies in the input/output dimensions of layers, the last layer of each residual block in a same stage (as well as the shortcut layer in the residual branch of the first block) are pruned exactly the same way (which is an even stronger constraint that simply uniform pruning) while other layers can be pruned globally [141]–[144].

One last aspect to tackle is the relevance of uniform pruning, compared to non-uniform local or global pruning. Indeed, uniform pruning or manually-set pruning rates, especially in the case of structured pruning, makes the resulting architecture predictable (if we do not take into account the non-trivial indexations that can be added by pruning, as we will see in Chapters 5 and 6). This means that such an architecture can be trained from scratch, without having to resort to pruning. The problem is that the ability of pruning to perform better than regular smaller baseline models is still debated. Liu *et al.* [57] affirm that baselines perform better, and that the interest of pruning is to produce non-trivial, non-predictable and more efficient architectures—which makes it fundamentally a NAS method. However, other works point to the contrary. We can think, notably, of the work of Renda *et al.* [92] which compares regular magnitude pruning, lottery ticket [91] and their own way of re-training pruned networks—which we will describe more into details in Chapter 4. To sum up what is interesting for us right here: their experiment show that, with the right retraining, pruning performs better than training from scratch a masked network (or, therefore, a smaller baseline), which is the opposite conclusion than that of Liu *et al.* [57].

Concerning the ability of global pruning to produce efficient architectures, this time it is our own works [2], [3], presented in Chapter 6, that raise questions. Indeed, as we will see, because of some behaviour of global structured pruning on gates, the pruned networks yield a better parameters-to-accuracy ratio, but with networks that are actually denser in term of operations. The problem is that operations are more correlated with latency and energy consumption than the raw number of parameters, which means that, in a technical point of view, these networks are actually less efficient than simply narrower baseline networks.

3.2.3 Criteria Throughout Training

The final aspect to review concerning pruning criteria is when to apply them, relatively to training. Indeed, most of the demonstrations and arguments we mentioned suppose that the network has converged. However, as we will see in Chapter 4, many methods prune the network during training or, sometimes, even before training [98], [99]. The problem is that, the further the network is from convergence when the criterion is applied, the less relevant it is likely to be; in other words: weights pruned early during

training are unlikely the right ones. We will see in Chapter 4 that many papers propose methods to cope with this problem, while pruning during or even before training.

Obviously, the core problem of all this question is that pruning criteria consider the loss $\mathcal{L}$ in the locality of $\mathbf{w}$, while training involves crawling more largely the landscape of $\mathcal{L}$ in order to find its global optimum. Since pruning equates putting some weights to zero, then it involves displacing the point $\mathbf{w}$, so that $\mathcal{L}$ is not expected to behave the same in this new proximity, and the proximity of the post-retraining $\mathbf{w}$ is expected to be also different.

We need to explore this problem more deeply, and this is what we will do in Section 3.3.

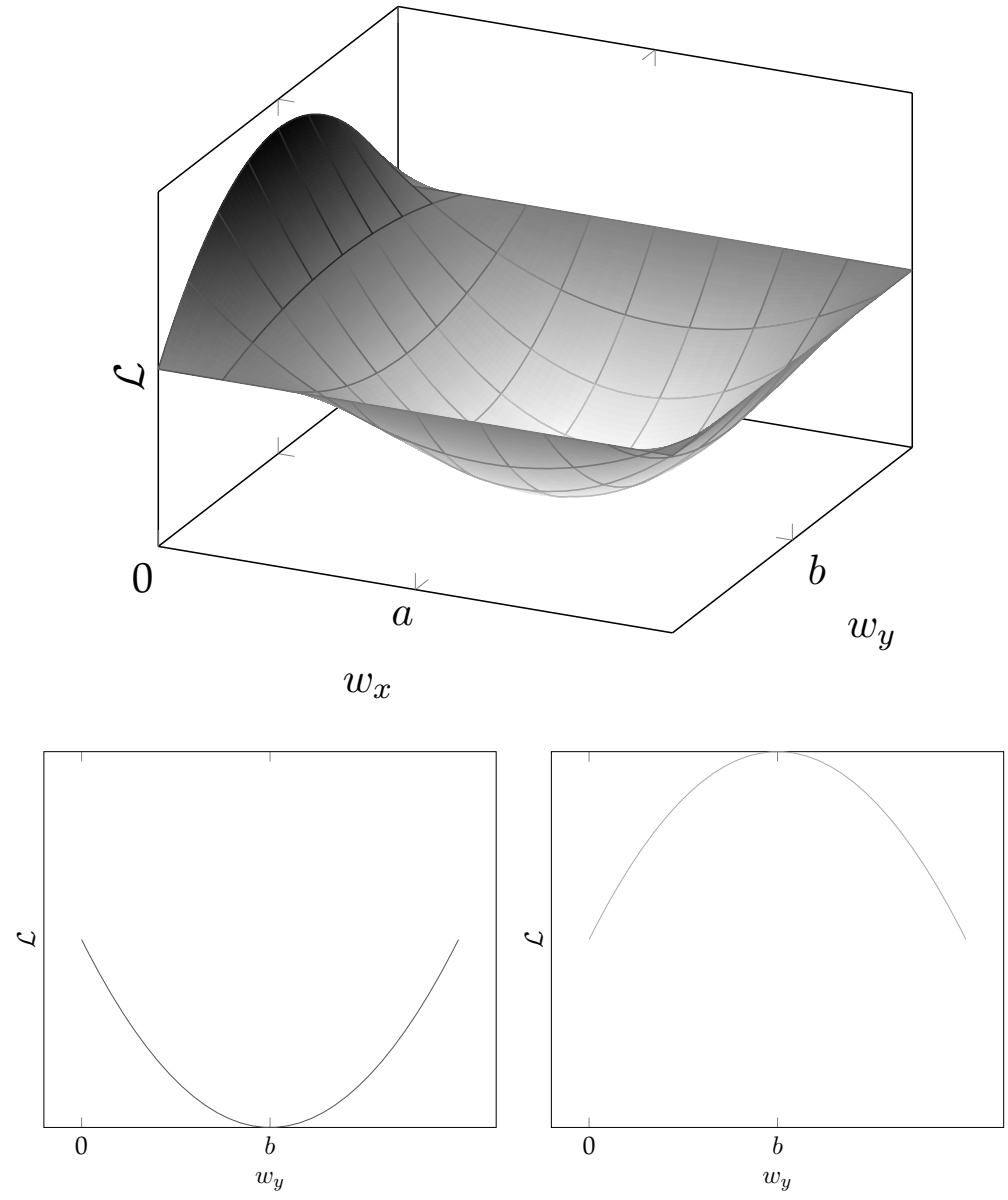
3.3 The Effect of Pruning on the Error Function

Our previous discussions concerning magnitude, gradient and training highlights the need to investigate more carefully the actual impact of pruning on the loss $\mathcal{L}$. Pruning can be seen as setting weights to zero, and this is no magical transformation as it basically sums up to displacing the point $\mathbf{w}$ in the definition space of $\mathcal{L}$. As obvious as it may seem, it is easy to forget this principle if one views pruning as "removing" a parameter.

This displacement of $\mathbf{w}$ in the landscape of $\mathcal{L}$ is enough to explain many behaviors of pruning, as well as what may be the pitfalls of current pruning criteria or even methods. Let's consider one toy example in Figure 16: here we consider a very simple network counting only two weights w_x and w_y, so that $\mathcal{L}$ reaches an optimum at $w_x = a$ and $w_y = b$. In the case we want to prune w_x, let's study the behavior of w_y: for $w_x = a$, $w_y = b$ is indeed at the minimum but for $w_x = 0$ it is at a maximum, which means that, from the point of view of the remaining w_y, the optimum got displaced.

This very simple example shows something that has very strong implications on pruning: the interdependence of parameters weights in a network means that the projected $\mathcal{L}_{w_q=0}$ is not expected to behave in any way like the original function $\mathcal{L}$. Therefore, the optimum of $\mathcal{L}_{w_q=0}$ is not expected to coincide with that of $\mathcal{L}$, as can be seen in Figure 16; moreover, the value of this new optimum is unlikely the same and, by definition, is superior or equal: we have $min(\mathcal{L}_{w_q=0}) \geq min(\mathcal{L})$. Also, because of how wildly different $\mathcal{L}$ can be in different points, there is no reason to think that there is any real correlation between the value or place of this minimum and any property of any weight in the point $\mathbf{w}$ obtained after convergence on the original $\mathcal{L}$: it seems unlikely to provide information on the whole function $\mathcal{L}$ just from a single point $\mathbf{w}$. The only thing that mitigates this problem is that, providing that $\mathcal{L}$ is at least locally smooth (which is sound in the case of piecewise linear networks), then a very small displacement of $\mathbf{w}$ should bring the least difference in behavior between $\mathcal{L}$ at $\mathbf{w}$ and at $\mathbf{w} + \delta\mathbf{w}$. Therefore, pruning the smallest weights brings the smallest possible displacement through pruning to $\mathbf{w}$, so that $\mathbf{w} \setminus w_q$ is expected to be still close to optimum in $\mathcal{L}_{w_q=0}$. This is consistent with the explanations we brought in Section 3.1.2.

This perspective also highlights one major flaw of how pruning and fine-tuning are usually depicted: indeed, if the displacement is not small enough to stay in the vicinity of the previous optimum, then fine-tuning is not here just to slightly re-adjust the weights but to find the whole new optimum of the function, that is not guaranteed to coincide with the previous one at all. This also means that the difference in loss between before and after pruning, once the displacement is too large to stay in a given proximity, has no guarantee to coincide with that between before pruning and after fine-tuning. To



(a) $w_x = a$ **(b)** $w_x = 0$

Figure 16: Illustrative example of how pruning modifies the behavior of non-pruned weights: at $w_x = a$, w_y is at a minimum, but at a maximum when $w_y = 0$. Therefore, pruning w_x completely changes the behavior of w_y, hence the need for retraining or fine-tuning.

sum up: rigorously speaking, the goal should not be to find the best $\mathcal{L}(\mathbf{w}_{w_q=0})$, but the best $min(\mathcal{L}_{w_q=0})$, even if the temporary degradation of performance is worse. Indeed, it is possible to imagine counter-intuitive cases where, for example, we have $w_x \leq w_y$ but $min(\mathcal{L}_{w_x=0}) \geq min(\mathcal{L}_{w_y=0})$, in which case we should rather prune w_y than w_x, even though the magnitude criterion tells the contrary.

The last aspect that needs investigation is the reasons for which pruning would possibly yield better or worse performance than retraining the same sparse architecture from scratch. Indeed, presenting the problem as we did highlights that, whatever the value of $\mathbf{w}$, the overall landscape of $\mathcal{L}$ stays fundamentally the same. Therefore, under the hypothesis of a perfect training method that reaches inevitably the optimum, whether it is from some random initialization or from a pruned model $\mathbf{w}_{w_q=0}$, the best possible performance is exactly the same in both cases. Once again: there is no magical transformation of $\mathcal{L}$ through pruning. Therefore, it appears that the benefit of pruning, for the same sparse architecture, can only be of providing an initialization that is less susceptible of getting stuck in a irrelevant local minima.

To conclude, we can now formalize what should be the two real missions of pruning:

- Let's consider all possible subsets $\boldsymbol{\omega}$ of $\mathbf{w}$, of a given cardinal corresponding to the intended pruning rate; the goal of pruning is:

$$\min_{\boldsymbol{\omega}} \left(min(\mathcal{L}_{\boldsymbol{\omega}=0}) \right), \tag{13}$$

  which means that we want to set to zero all weights in the subset $\boldsymbol{\omega}$ that allows for the best possible existing optimum in the corresponding projection of $\mathcal{L}$.

- The other goal of pruning is to produce a post-pruning initialization $\mathbf{w} \setminus \boldsymbol{\omega}$ that is able to reach the optimum through the chosen optimization methods.

3.4 Recapitulation

In this chapter, we reviewed and discussed the main aspects of pruning criteria: how to define the importance of a weight or of a group of weights, how to define the pruning rate of each layer and when to measure these criteria relatively to training.

Concerning criteria themselves, we mainly reviewed the gradient-based saliency metric and weight magnitude, and gave our own hindsight about them: gradient-based strategies suffer from a severe oversight concerning the theoretical necessity to consider only weights that are very close to zero, which makes them redundant with the weight magnitude criterion.

When reviewing the other aspects, we saw that structured pruning criteria still struggle with both the unbalance between layers for a same criteria and the necessity to cope with constraints due to architectures, as well as we saw that the fact that most pruning method do not wait for full convergence to apply these criteria harms the theoretical justifications backing most pruning criteria.

Finally, we proposed a new simple way to consider the impact of pruning, which highlighted what is the main problem of current criteria: extrapolating from a point the behavior of the whole error function, while pruning is itself the combination of a displacement and a projection. This allowed us to figure out what should be—at least theoretically—the two real goals of pruning: finding the projection that allows for the best possible minimum and providing a relevant initialization to reach it through optimization methods.

Now that we have seen how to identify the weights to prune in a neural network, we will see how to remove them in Chapter 4.

Chapter 4

Removal Methods

Contents

This chapter will be dedicated to removal methods, i.e. everything that encompasses how and when to remove weights. We call it "removal method" to avoid the ambiguity when "pruning method" refers to the whole method proposed by a paper, which includes the choice of a criterion and of a structure.

Removing weights supposes that the criterion (Chapter 3) and the structure (Chapters 5 and 6) have already been chosen and properly defined. Therefore, except when a given pruning method presupposes a specific criterion or structure, we will notate by default the subset of the weights to remove by the generic notation $\mathbf{W}^*$.

During this chapter, we will first review some methods in the literature, present what stakes motivated them and, finally, show one of our contributions: Selective Weight Decay (SWD) [1]. This will be the occasion to discuss each family of method as well as our own result and to put them in perspective with the notion we develop in the other chapters of this manuscript. We will also expose the shortcomings of some methods and show the link between different papers in the literature to deconstruct some non-trivial topics such as the lottery ticket hypothesis or the ability of pruning to produce relevant architectures.

4.1 Basic Framework and Considerations

In 2015, the work of Han *et al.* [26] put pruning back under the spotlight, after the general drought in the domain of neural networks in the 2000s and once its rebirth in the early 2010s made it obvious that reducing their cost would be a major stake of the following years. Its very simple method can still be considered the basics of pruning in general, so that many methods can actually be defined by how they stray away from this method. Therefore, presenting it in details allows highlighting many stakes that motivated the birth of many methods afterward. This is why we will detail this method and use it to present some general considerations on removal methods.

4.1.1 Train, Prune and Fine-Tune: the Basic Framework

The method of Han *et al.* [26] involves pruning isolated weights (non-structured pruning) on the basis of their magnitude (weights magnitude criterion). The network $\mathcal{N}$ is first trained normally, then pruned by setting weights of lowest magnitude definitely to zero; the network is then fine-tuned. Actually, these last two steps can be iterated while increasing the pruning rate each time.

Let t be a pruning rate and n the number of aforementioned iterations. The method can be coarsely summed up as Algorithm 2:

Algorithm 2: Method of Han *et al.* [26]

Data: $\mathcal{N}$, $\mathbf{W}$, t, n
Train $\mathcal{N}$;
for $i \leftarrow 0$ *to* n **do**
    Increase t;
    Define $\mathbf{W}^*$ according to t;
    Prune $\mathbf{W}^*$ from $\mathcal{N}$;
    Fine-tune $\mathcal{N}$;

This train-prune-retrain framework is the basis of many pruning methods or papers in the field. However, it already raises some questions:

- How many iterations are required and how should the pruning rate increase between each, precisely?
- Is abruptly putting weights to zero, without any possibility to recover, the best way for the network to learn good sparse solutions?
- Is it really the best choice, to remove weights only once the network has been fully trained without any constraint of sparsity?

In the original paper, Han *et al.* [26] applied five iterations, without telling how to increase the pruning rate between each. We will detail the influence of iterations and how the literature tackled these questions in the following sections.

4.1.2 The Importance of Iterations

Multiple works [26], [87] have acknowledged the importance of iterations and the gain in performance it allowed in opposition to purely one-shot pruning. Tanaka *et al.* [98] in particular have proposed an explanation of the ability of iterations to prevent *layer collapse*, that is the propensity of pruning methods to prune entire layers, which tends to destroy (or at least to harm severely) the network.

Tanaka *et al.* [98] propose two theorems: the *neuron-wise conservation of synaptic saliency* and the *network-wise conservation of synaptic saliency*. To sum it up roughly: with the saliency defined as the product between a weight value and that of the loss function gradient depending on said weight, the sum of the saliency of all weights in a layer is constant across layers (more detailed explanations are to be found in the original publication). This conservation, said to have been noted in other papers or literature [145]–[150], means that the saliency of weights tends to decrease for larger layers, which implies that pruning on the basis of the magnitude of this saliency tends to prune larger layers—therefore pruning iteratively prevents this phenomenon, since the magnitude of saliency in pruned layers therefore increases between iterations.

Tanaka *et al.* [98] then expands this observation to magnitude pruning, observing that there is a correlation between the evolution of the magnitude of weights during training and said saliency, which means that said magnitude tends to observe the same type of conservation. Therefore, iterating pruning and re-training (or fine-tuning) allows for larger layers, whose weights are smaller and then more pruned, to re-increase the magnitude of their remaining weights after pruning, which makes them less prone to be pruned. Tanaka *et al.* [98] used methods centered around the *Lottery Ticket Hypothesis* [91], [151] to illustrate this principle, but nothing prevents it from being generalized to the type of iterations to be found in Han *et al.* [26].

4.1.3 On the Abruptness of Manual Pruning

Another interest of dividing pruning into multiple iterations is to make each pruning step smaller, and ultimately to make pruning as smooth as possible. This effort to make pruning smooth underlies a large portion of the literature. Even though the reason why pruning abruptly and manually is a problem seems intuitive, it deserves being more formally presented.

Indeed, in which way does pruning exactly disturb training? Training a network can be roughly summed up as finding the best $\mathbf{W}$ to minimize $\mathcal{L}(\mathcal{N}_{\mathbf{W}}(\mathbf{X}), \mathbf{Y})$ (cf. Section 2.1.5).

It is possible to simplify this as finding the global minimum of the $\mathcal{L}_{\mathcal{N}}(\mathbf{W})$ function. However, we showed in Chapter 3 that pruning acts as a displacement of $\mathbf{W}$ and a projection of $\mathcal{L}$. The behavior of $\mathcal{L}$ in the vicinity of the point represented by the pruned $\mathbf{W}$, from the point of view of the remaining parameters, is likely not to be any similar to the one they had before pruning. Therefore, while training tries to crawl smoothly the landscape of $\mathcal{L}$ to find its optimum, pruning throws it seemingly randomly somewhere else, where the landscape is completely different. Moreover, the reduced dimensionality of the pruned $\mathbf{W}$ may make each post-pruning training steps more difficult—although it may be also possible to view it, on the other hand, as a type of regularization [85], [93], [94], [152].

To sum up: each time the network is pruned, the remaining parameters have to solve a completely different problem. Formulating all this problem as pruning being a redefinition makes easier to intuit that the solution is to redefine pruning. Indeed, the overall solution that underlies the literature on pruning methods is to make pruning a part of the problem to solve, to redefine what is an optimization problem under the constraint of sparsity. Mainly three ways of achieving this can be found in the literature:

- One solution is to divide pruning into as many infinitesimal steps as possible, which implicitly assumes that, if $\mathcal{L}$ is smooth enough, a small modification of $\mathbf{W}$ does not change too much the behavior of the remaining parameters—because we stay in the vicinity of the previous point. This can be achieved through iterations [26] or even through a progressive pruning throughout training [90], [153]; another method is to relax pruning as a penalty on weights magnitude, so that this penalty either converges toward the hard constraint of pruning while growing in importance, or at least helps unimportant weights to be even less important, which makes them less harmful to prune [1], [112], [128], [130], [132], [154]–[156]. Other methods, without mentioning explicitly this topic, also split pruning into smaller, less definitive steps by allowing some types of regrows [88], [140], [157]–[159].

- Another solution is to make pruning derivable, in order to make it a part of the problem to optimize. However, pruning is often formalized as a constraint on the $\mathcal{L}_0$ norm of $\mathbf{W}$, which is not differentiable [89]. This problem is usually either relaxed into a smoother, differentiable penalty [1], [154] or circumvented using the straight-through estimator [69], similarly to Binary-Connect [68], in order to learn progressively a mask using gradient descent [89], [160]–[163].

- Finally, one last solution is, on the contrary, to completely split pruning and training into two completely different problems by first finding a relevant architecture before training it, whether it is done using Lottery Ticket [91], [151], [164]–[167], pruning at initialization [98], [99], [168], [169] or simply by turning fine-tuning into a complete, proper retraining, using the previous trained weights as the new initialization [92].

Presenting these three types of solutions already allowed to present what are the main families of methods to be found in the literature. We will review them individually in the following section before presenting our own work: Selective Weight Decay (SWD) [1].

4.2 Removal Methods in the Literature

4.2.1 Progressive Pruning and Sparse Training

The most simple and straightforward way of making the basic method smoother is to increase the number of iterations. This logic can be pushed further until the whole training is made of infinitesimal steps. This principle is applied in the "gradual pruning" method [90], that also allows for masked weight to regrow by masking weights of smallest magnitude while letting them being updated during gradient descent, so that masked weights can overcome the magnitude threshold; the masking rate grows during training. This method proved to yield competitive results [153], even when facing more sophisticated methods [89], [156].

Letting the possibility to regrow weights is a principle to be found in other methods, that often share the property of performing pruning all throughout pruning. This is understandable, as pruning criteria meant to work well at the end of training are not guaranteed to be relevant before—letting weights regrow allows to compensate some premature or abusive pruning. One way, called "soft pruning" [157] simply consists in setting periodically weights to prune to zero, but without preventing them from being updated afterward and without masking them.

Another family of method, based on this regrowing principle, is called "sparse training" [88]. It consists in training the network whose a portion of weights is masked from the start at its definitive pruning rate; the initial pruning mask is random. Periodically, a portion of non-pruned weights is pruned—for example, the smallest ones— and, on the other hand, an equivalent portion of pruned weights is brought back. Either these revived weights are brought back randomly [88] or using a given criterion (usually based on momentum or gradient) [140], [159]; this revival rate can be the same for each layer or global [158].

Even though these methods yield satisfying improvements over the original method, the main theoretical problem is that they still abruptly put weights to zero, even though they try putting the fewest possible weights to zero at once every time. While such a method effectively mitigates the problem, it does not make the pruning-induced displacements continuous either.

4.2.2 Pruning at Initialization

Pruning at initialization, in theory, has multiple advantages: 1) it allows to separate pruning and training, 2) it makes training less expensive, as the networks to train are smaller, 3) it does not require any particular process after training, while pruning during training can bring additional complexity and the need for fine-tuning, which makes training longer. However, as we will see, this type of pruning encounters many practical and theoretical problems. Such types of pruning can be divided into two families: pruning at initialization per se and Lottery Ticket.

Methods such as SNIP [99], GraSP [169] or SynFlow [98] all have the same goal: to find a criterion that functions well on a network that is not trained at all. Since weights are generated randomly, it would not make any sense to prune them on the basis of their magnitude before training. This is why these methods usually revolve around criteria based on the loss gradient, or its "saliency". Unfortunately, Frankle *et al.* [170] demonstrated that these three methods each perform as well if the mask of each of their layer is shuffled. This means that, at most, these criteria only find relevant pruning rates for each layer, while the masks themselves could as well be perfectly random. Thus,

the performance of these methods come from two things: 1) their ability to avoid layer collapse and produce smaller networks that can properly be trained, and 2) the fact that training from scratch or retraining a pruned network seems to bring improvements compared to mere fine-tuning [92].

The other family of methods we mentioned revolves around the "Lottery Ticket Hypothesis" [91], formulated by Frankle *et al.* as: *A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—–when trained in isolation—–it can match the test accuracy of the original network after training for at most the same number of iterations* (direct citation from the original publication [91]). This hypothesis was drawn from a certain observation: when training a network, pruning it, restoring the network to its original initialization, applying it the post-training pruning mask and retraining it, it appears that, even at significant pruning rates, the resulting networks can reach an accuracy similar to that of the full, non-sparse network. This observation seemed to work well at least on CIFAR-10 and MNIST; on ImageNet, it required not rewinding at the initialization, but at a few epochs after it [151].

This Lottery Ticket hypothesis caused a craze in the literature: it birthed a large literature concerning it directly and more or less deeply inspired an impressive number of papers—some papers even mention the Lottery Ticket in their title while being barely related to it. Papers of the first type studied multiple properties of Lottery Ticket, such as:

- What makes Lottery Ticket masks relevant, under which conditions do they work and how tightly linked to a given initialization they are [164], [171].
- When exactly should the masks be generated or to which point should the network be rewinded [151], [172].
- The supermasks, i.e. the ability of pruned non-trained networks to have above random guess performance [164], [166].
- The applicability of Lottery Ticket masks to transfer learning and their ability to keep their performance on datasets that are different from the one used to generate it [165], [173]–[175].
- Theoretically justifying the possibility of Lottery Ticket, using the theory of dynamical systems [167].

However, some doubts remain about how hasty of an extrapolation the Lottery Ticket Hypothesis may be to its experiments, whose results are nonetheless interesting. The sometime impressive performance of methods that prune at initialization already hinted that getting good performance from networks that are sparse from the start is not a miracle. Moreover, most of the experiments conducted on Lottery Ticket only involve simpler datasets, such as CIFAR-10; making it work on ImageNet requires to rewind to a network that has already started to be trained, and is thus arguably closer to its final state before rewinding—which would help the mask to stay relevant. Therefore, if the possibility of obtaining good performance from a sub-network seems very plausible, attributing this good performance to some special property of a given initialization and sub-network requires more evidence—especially since multiple works, cited previously, hint that the relationship between an initialization, a mask and the task to solve is rather loose.

Finally, the work of Renda *et al.* [92] compares three methods: 1) simple magnitude pruning, 2) the lottery ticket experiment and 3) Learning Rate Rewinding (alias LR Rewinding or LRR) [92] that gets rid of the rewinding of the Lottery Ticket and only

rewinds the learning rate—thus, it consists in replacing fine-tuning with a warm restart. In these experiments, lottery ticket performed better than magnitude pruning but LRR performed better than lottery ticket. This result would hint that, actually, the rewinding principle of lottery ticket actually harms the performance and that the gains observed when applying it only came from the post-pruning re-training under an increased learning rate. If this hypothesis confirms to be true in the future, it would potentially mean that the hypothetical special properties of the lottery ticket masks were actually a red herring from the start. However, whatever what LRR may mean for the lottery ticket hypothesis, its performance allows it to be a relevant alternative to fine-tuning, which makes it a very useful addition to the literature of pruning methods.

One last aspect to tackle, concerning this domain, is that pruning at initialization (or even LRR) revolves around the idea that pruning is able to produce relevant architectures that perform better than simply smaller architectures. The work of Liu *et al.* [57] expresses doubts about the ability of pruning either to work better than pruned networks trained from scratch or than smaller architectures. Notably, it seems that producing, through pruning, the architecture of a smaller network does not work better than training said network from scratch—which reduces drastically the interest of structured pruning where all layers are pruned at the same rate. Therefore, the main interest of, especially, structured pruning is to produce better architectures. Even though the performance of LRR allows to put into perspective the aforementioned statement, that pruned networks do not perform better than baselines, our own work, exposed in Chapter 6, shows that the ability of pruning to produce relevant architectures is not guaranteed [2]. Finally, on the topic of whether or not fine-tuned (or retrained) networks should perform better than pruned architectures trained from scratch: we already saw in Section 3.3 that both cases try to solve the same problem and that the role of pruning, between the two, is to provide a more relevant initialization. If both the random initialization or the one produced through pruning are able to find the optimum, they should perform exactly the same in the end.

4.2.3 Learning Masks Through Auxiliary Parameters

As mentioned before, one theoretical way to improve pruning would be to include it directly within the problem to solve, i.e. in the loss function itself. However, we also mentioned that pruning is often formalized as a penalty on the $\mathcal{L}_0$ norm of $\mathbf{W}$, which is not differentiable and, thus, not compatible with stochastic gradient descent. While some methods choose to solve this problem by relaxing said $\mathcal{L}_0$ constraint, as can be seen in Section 4.2.4, the methods we will review here adopt another strategy: keeping a binary mask on parameters, mask that is itself produced using another set of auxiliary masks.

Let $\mathbf{M}$ be the set of auxiliary parameters and f the function to produce the mask; inference of the network to prune is then performed using not $\mathbf{W}$ but $\mathbf{W} \odot f(\mathbf{M})$. Therefore, the stake, in such methods, is to learn both $\mathbf{W}$ and $\mathbf{M}$, providing that f is differentiable. Since $f(\mathbf{M})$ is supposed to be a binary mask, made only of zeros and ones, f is not expected to be differentiable. However, such a problem is often eluded using the straight-through estimator [69], in a way that is very similar to the Binary-Connect method [68].

Courbariaux *et al.* [68] originally proposed two ways of defining f: either f is a rounding of a $\mathbf{M}$ clipped between 0 and 1 or f generates a random mask, with each element being 0 or 1 following a Bernouilli law parameterized by the corresponding value in $\mathbf{M}$ (so that these auxiliary parameters are actually the probability of masking the cor-

responding weight in $\mathbf{W}$). To sum it up, there are two variants: $f(\mathbf{M}) = \lfloor \mathbf{M} \rceil$ (deterministic) and $f(\mathbf{M}) = \mathcal{B}(\mathbf{M})$ (stochastic). In both cases, values of $\mathbf{M}$ are always clipped between 0 and 1 after each update.

Srinivas *et al.* [161] directly applies this stochastic variant to pruning. Xiao *et al.* [162] instead uses the deterministic variant; however, instead of completely integrating pruning into the loss function, it instead applies an iterative (even citing Han *et al.* [26]) bi-level optimization [176] where $\mathcal{L}$ is split in two optimization processes $\min_{\mathbf{W}} \mathcal{L}_1$, trained on the training set, and $\min_{\mathbf{M}} \mathcal{L}_2$, trained on the validation set (a subset of the training set) and containing a penalty on $\mathbf{M}$. Also, $\mathbf{M}$ is not updated solely according to its gradient, since it is actually modified (using the gradient, values and sign of $\mathbf{W}$) before update; this modification is meant to bring some desirable properties to $\mathbf{M}$.

Similarly to Srinivas *et al.* [161], Louizos *et al.* [89] uses the stochastic variant of BinaryConnect. However, treating parameters $\mathbf{M}$ as probabilities of a Bernouilli function poses problems that are solved by using a reparameterization trick [177], [178] so that $\mathbf{M}$ instead parameterizes a differentiable function that is applied to a parameter-free noise; this way, the differentiable parameters and the stochastic noise are separated.

Comparing these three methods allows to see how this literature tends to revolve around the same idea while proposing new ways of improving it. To conclude, let's mention the work of Savarese *et al.* [163], that is a bit different, as it gets rid of the non-differentiability of pruning masks—that required the use of the "Straight Through Estimator"—by instead using a smooth approximation of the Heavyside function over $\mathbf{M}$, whose temperature rises during training until it converges toward said Heavyside function. This type of relaxation is rather related with the underlying idea of penalty-based method (cf. Section 4.2.4).

4.2.4 Penalty-Based Methods

Penalty-based methods all revolve around the same idea of relaxing the $\mathcal{L}_0$ constraint into a differentiable penalty. Multiple variations around this idea can be found in the literature.

The first version consists in applying to the whole network some type of penalty to encourage sparsity. The LASSO [179] and group-LASSO [180] regression methods were used by different pruning methods [112], [130], [132] to introduce sparsity of individual weights or groups. Other methods, similarly, apply other types of global penalties over the entirety of weights—which can be seen as modifying weight decay to enforce sparsity [105], [181].

Another variation is called "Variational Dropout" [182]. Dropout [183] is originally a regularization technique that involves multiplying activations or weights with a Bernouilli mask $\mathcal{B}(p)$, with p the dropout rate; it was later found out that a Gaussian Dropout [184], that instead adds a Gaussian noise $\mathcal{N}(0, \frac{p}{1-p})$, could achieve a similar behavior. Kingma *et al.* [182] noticed that such a way of noising weights was very reminiscent of the variational paradigm that can be used to train auto-encoders using variational inference [177]. This paradigm allows considering the variance $\alpha = \frac{p}{1-p}$ as a learnable parameter; therefore, each weight can even have its own variance. Molchanov *et al.* [156] then noticed that the Kullback–Leibler divergence term of the variational lower bound, to maximize during variational inference, tended to favor large values of α, and therefore the equivalent of a large dropout rate of parameters. When the α of a given weight tends to infinity, then the corresponding weight can be considered as pruned. Therefore, variational dropout can be considered as a pruning method, based on a type of

regularization that is not weight decay but dropout. Multiple papers have then focused on adapting this pruning method to structured pruning [185], [186].

Finally, the last family of penalty-based methods uses selective penalties, i.e. applying a penalty only to weights susceptible to be pruned. Most of them consist in periodically identifying weights to prune, using the magnitude [1], [136], [154], [155] or gradient (or saliency) [103] criteria; then selected weights are penalized, mainly using a $\mathcal{L}_2$ penalty. Choi *et al.* [155] penalizes smallest weights with an additional $\mathcal{L}_2$ penalty whose importance is learned during training; Ding *et al.* [103] removes the gradient of error from the update of weights of smallest saliency, so that only weight decay remains and shrinks them; Carreira-Perpiñán [154] penalizes weights under a given threshold with a $\mathcal{L}_2$ penalty whose importance grows toward infinity during training; our own contribution, SWD [1] generalizes this same principle to structured pruning and studies the importance of the growth of the penalty.

One last sub-group of pruning methods using selective penalty revolves around the Alternating Direction Method of Multipliers (ADMM) [187] to learn and optimize separately the weights, their mask and how to penalize the difference between the two. Many papers have then used ADMM to study pruning or to generalize it to structured pruning [82], [188]–[193].

4.3 Selective Weight Decay

In the previous sections, we presented the literature around pruning methods as the various attempts at smoothing the displacement of $\mathbf{W}$ in $\mathcal{L}$ through pruning. Since penalty-based methods, instead of putting in succession small batches of weights abruptly to zero, penalize smoothly the magnitude of weight themselves, they appeared to us as a particularly relevant direction to explore further.

In this section we will present a contribution of ours: Selective Weight Decay (SWD) [1], a penalty-based removal method. We will present its principle and then the various experiments that justified its paradigm and showed its performance.

4.3.1 Principle

SWD aims at pruning as smoothly as possible by applying a $\mathcal{L}_2$ penalty on the subset $\mathbf{W}^*$ of weights to prune from $\mathbf{W}$. In Chapter 2, we already exposed the standard optimization problem, with weight decay, in Equation 1. We update this equation to add Selective Weight Decay in Equation 14:

$$\min_{\mathbf{W}} \mathcal{L}(\mathcal{N}(\mathbf{X}), \mathbf{y}), \text{with } \mathcal{L}(\mathcal{N}(\mathbf{X}), \mathbf{y}) = \mathcal{F}(\mathcal{N}(\mathbf{X}), \mathbf{y}) + \mu \left\|\mathbf{W}\right\|_2 + a\mu \left\|\mathbf{W}^*\right\|_2 \tag{14}$$

This $\mathcal{L}_2$ penalty, whose importance is defined by the multiplier a, serves as a relaxation of a $\mathcal{L}_0$ constraint. With the increase of a, this penalty should converge toward this constraint. This principle is directly inspired by Lagrangian smoothing [194]. Figure 17 illustrates the landscape of the combined penalty of weight decay and SWD. Besides this simple principle and motivation, two things remain to be defined: $\mathbf{W}^*$ and a.

Definition of $\mathbf{W}^*$ As $\mathbf{W}^*$ represents the subset of weights to prune, its definition depends on the chosen pruning structure and criterion, which is orthogonal to the definition of SWD, that is a removal method. However, one important thing to notice is that, depending on any criterion or structure, $\mathbf{W}^*$ can be updated at any moment

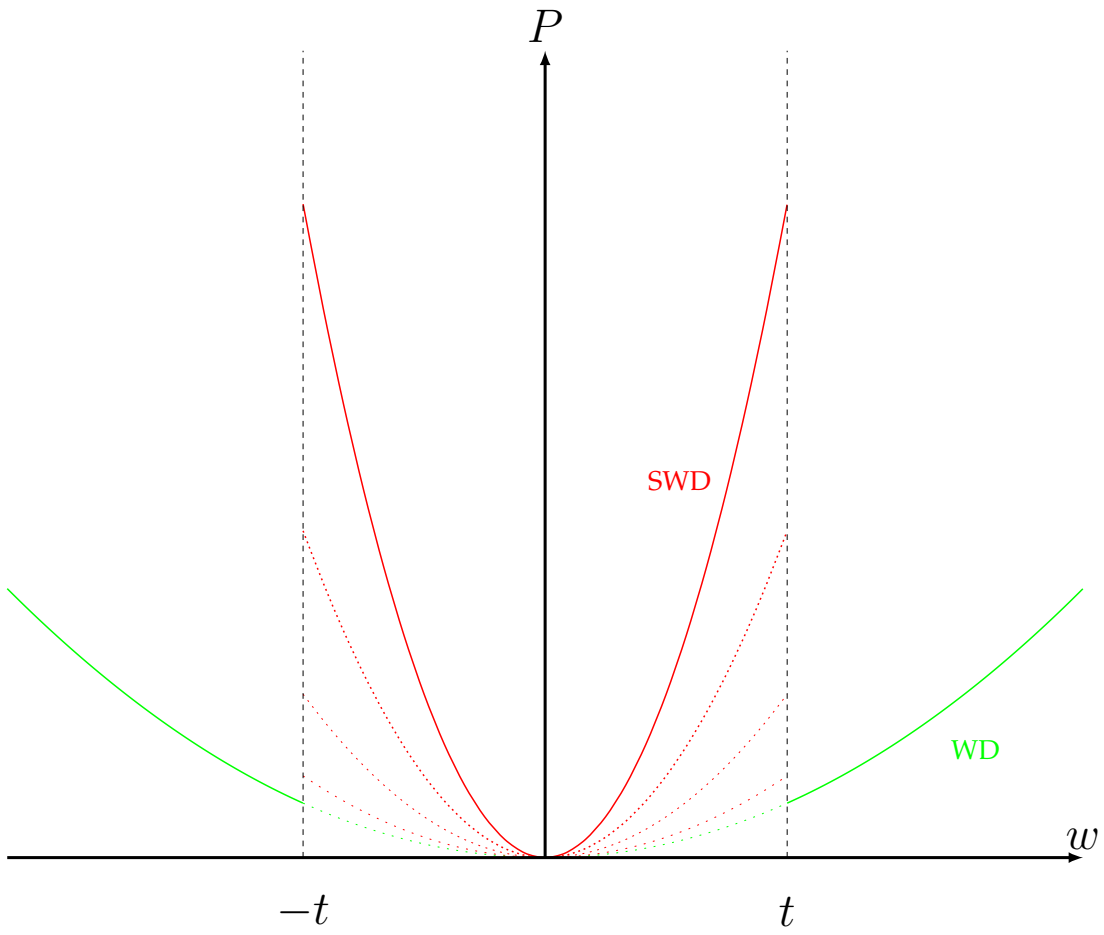


Figure 17: In the case of non-structured magnitude pruning, SWD behaves as such: all weights w of $\mathbf{W}$ are applied a penalty P, that combines weight decay and SWD. Since SWD is only applied to weights below a given threshold t, it is like P applies a stronger penalty on weights below t and a smaller one above t. Another way to present this is to consider SWD as a trap that progressively subjects weights below a certain threshold to an increasingly strong penalty, that drives them to zero, thus pruning them.

during training, as Equation 14 does not force $\mathbf{W}^*$ to be the same all throughout training. This is why we actually update $\mathbf{W}^*$ at each training step. Depending on the cost of computing $\mathbf{W}^*$, it is theoretically possible to update it less often; in another of our contributions [2], that we present in Chapter 6, we only update $\mathbf{W}^*$ at each epoch.

Definition of a The starting and final values of a are extremely important to the performance of SWD. Indeed, the idea is to start training by letting the network learn normally and to end training under a penalty that is strong enough to reduce targeted weights to zero (or at least, close enough to zero so that their removal does not affect the performance). However, one cannot simply make a grow indefinitely, as the gradient of a given w, from $\mathbf{W}^*$, is $\frac{\delta \mathcal{L}}{\delta w}(\mathcal{N}(\mathbf{X}), \mathbf{y}) = \frac{\delta \mathcal{F}}{\delta w}(\mathcal{N}(\mathbf{X}), \mathbf{y}) + 2\mu(1+a)w$, which means that, besides the supervised training term, the value of w is penalized by $-2\lambda\mu(1+a)w$ (with λ the learning rate—we simplify our explanation by eluding specific properties of optimizers and their momentum), that may actually make the value of $|w|$ diverge if $2\lambda\mu(1+a)w > w$. This is why we control both starting and final values of a. We chose to interpolate these two extreme values using an exponential function, as exposed in Equation 15:

$$a(s) = a_{min} \left(\frac{a_{max}}{a_{min}} \right)^{\frac{s}{s_{final}}}, \tag{15}$$

with a_{min} the starting value, a_{max} the final one, s the current training step and s_{final} the total number of pruning steps during training.

We now have defined the necessary background and terms concerning SWD. We will now present which criteria and structures we used with SWD during our experiments and what methods we compared it to; we will also present the experimental conditions.

4.3.2 Experimental Conditions

In our publication dedicated to SWD [1], we experimented with two variants of SWD:

- Non-structured SWD: isolated weights are pruned on the basis of their magnitude.
- Structured SWD: channels are pruned on the basis of the magnitudes of their weights in batch-normalization layers; an additional batch-normalization layer is applied to the output tensor of each residual block, so that pruning it prunes the same channels in both branches (the same trick was used by Liu *et al.* [87]).

In both cases, the magnitude threshold is global (the same for all layers) and defined so that $\mathbf{W}^*$ contains the needed number of parameters to reach the intended pruning rate.

We also compared SWD to other removal methods, for the same structures:

- Han *et al.* [26]: train, prune and fine-tune, with the last two steps iterated 5 times—the pruning rate grows linearly between iterations.
- Liu *et al.* [87]: train, prune and fine-tune once, with a smooth-$\mathcal{L}_1$ penalty on batch-normalization weights during training.
- Renda *et al.* [92]: train, prune and re-train, once, the re-training being a warm restart. When applied to structured pruning, we keep the smooth-$\mathcal{L}_1$ from Liu *et al.* [87].

Because of what we mentioned in Section 4.1, it would have been more rigorous to keep the same number of iterations for every reference method—something we were

not aware of at the time these experiments were conducted. Our motivation at the time, besides sticking to how methods were described in the original papers, was to avoid excessive computation times, as fine-tuning and retraining have a significant cost, especially on ImageNet.

We made experiments on three datasets, the values for hyper-parameters were chosen empirically through preliminary experiments:

- CIFAR-10 and CIFAR-100: fine-tuning iterations last during 15 epochs, except the last one that lasts 50 epochs. The smooth-$\mathcal{L}_1$ penalty is set to 10^{-4}. For non-structured SWD, we set $a_{min} = 10^{-1}$ and $a_{max} = 10^5$; for structured SWD, we set $a_{min} = 10^2$ and $a_{max} = 10^7$. We do no fine-tuning after training with SWD.
- ImageNet ILSVRC2012: for Han *et al.* [26], fine-tuning iterations last during 5 epochs, except the last one that lasts 15 epochs; for Liu *et al.* [87], fine-tuning iterations last during 15 epochs, except the last one that lasts 40 epochs. The smooth-$\mathcal{L}_1$ penalty is set to 10^{-5}. For non-structured SWD, we set $a_{min} = 10^{-1}$ and $a_{max} = 10^5$; for structured SWD, we set $a_{min} = 10$ and $a_{max} = 10^4$. In order to get results more consistent with the state of the art, structured SWD on ImageNet was combined with LRR [92] to maximize performance.
- Cora [195]: in order to verify that SWD could still be applied in the case of a very different type of task, we chose to do some tests on a Graph Convolutional Network from Kipf and Welling [196]. This GCN, with 16 hidden units, was trained with the Adam optimizer, weight decay set to $5 \cdot 10^{-4}$, a learning rate of 0.01 and a dropout rate of 50%. In order to make training with SWD less unstable, we increased the number of training steps from 200 to 2000 epochs. Fine tuning lasts 200 epochs, or 2000 for the last one. Non-structured SWD is set to $a_{min} = 0.1$ and $a_{max} = 10^6$. We tested no structured pruning on this network.

One last aspect to mention is that we did all our experiments in a deterministic setting, with the same random seed. Indeed, we could not afford multiplying experiments to reach statistical significance, so we instead made sure all experiments of a same type were conducted in the exact same settings, so that we can isolate the contribution of the pruning rate/method and that of some random initialization or order of batches. We now have all the necessary information to present the results of our experiments.

4.3.3 Ablation Test: the Importance of a

The first experiments we will show were conducted not only to find the best a_{min}/a_{max} pairs to use for our experiments, but also to show the sensitivity of SWD to these hyper-parameters and, ultimately, to justify the use of an increasing penalty during training. Figure 18 shows the results of this grid-search and provides two types of information: what is the most efficient a_{min}/a_{max} pair from a pruning perspective and how sufficient SWD is at pruning weights, without manually pruning targeted weights afterward—because if SWD is not strong enough, weights to prune may not decrease enough to nullify their impact on the network.

From these results, we can already draw multiple interesting observations:

- Worst final performance are obtained for low a_{start} and low a_{end}—it is also the range of values for which we have the most severe degradation after manual pruning. This means that, for this range of values, SWD is not strong enough and the final zeroing of selected values harms the network.

a_{end} \ a_{start}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}	10^{4}
10^{1}	67.07	71.88	53.68	83.89	91.0	94.54	94.21	93.92	94.08	94.44
10^{2}	89.87	92.4	93.99	94.55	95.15	94.65	94.0	93.91	94.13	94.52
10^{3}	94.72	94.81	95.27	95.29	94.96	94.72	94.0	93.5	94.23	94.57
10^{4}	95.22	95.14	95.29	95.37	94.73	94.5	94.49	93.85	94.17	94.61
10^{5}	95.19	95.3	95.07	95.24	94.78	94.39	94.39	93.54	94.45	94.37

(a) Top-1 accuracy after removal (%)

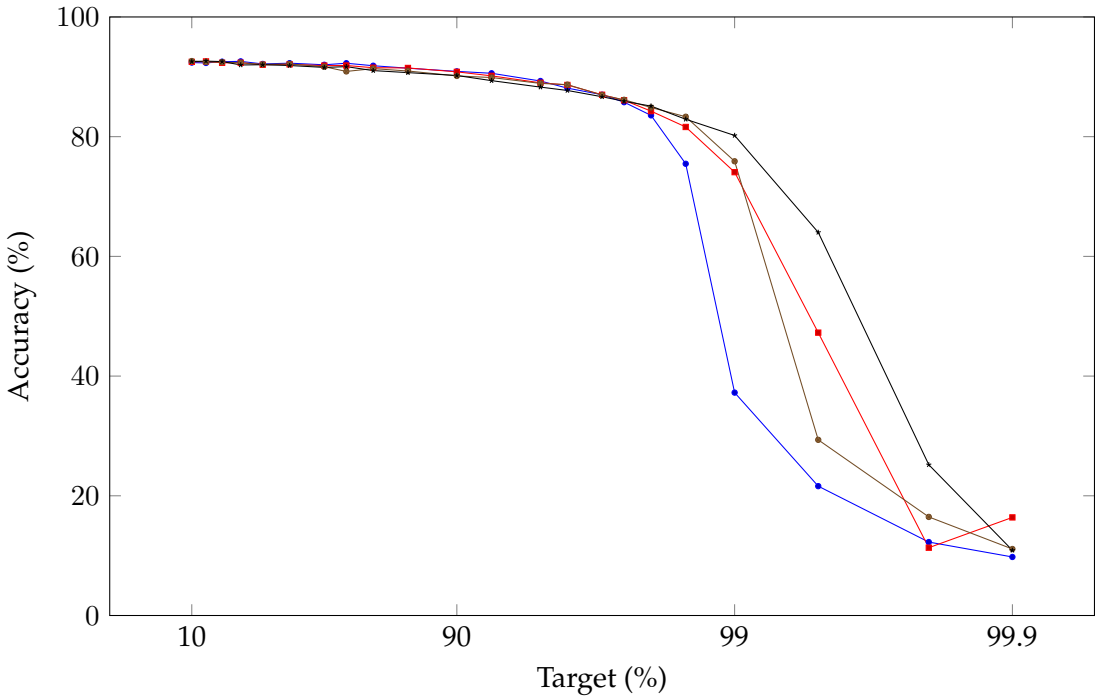
a_{end} \ a_{start}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}	10^{4}
10^{1}	-28.36	-23.53	-41.72	-11.49	-4.33	0.06	0	-0.01	0	-0.1
10^{2}	-4.74	-2.37	-0.99	-0.45	-0.01	0.1	-0.05	-0.01	-0.09	0.06
10^{3}	0.11	-0.01	0.05	-0.06	-0.01	0.02	0	-0.01	0.03	-0.01
10^{4}	0.08	-0.02	0.05	0.02	-0.03	0.02	0.01	0.01	-0.07	-0.02
10^{5}	-0.1	0.01	-0.01	-0.02	-0.01	0	0	0.02	0.04	0.01

(b) Change in accuracy through removal (%)

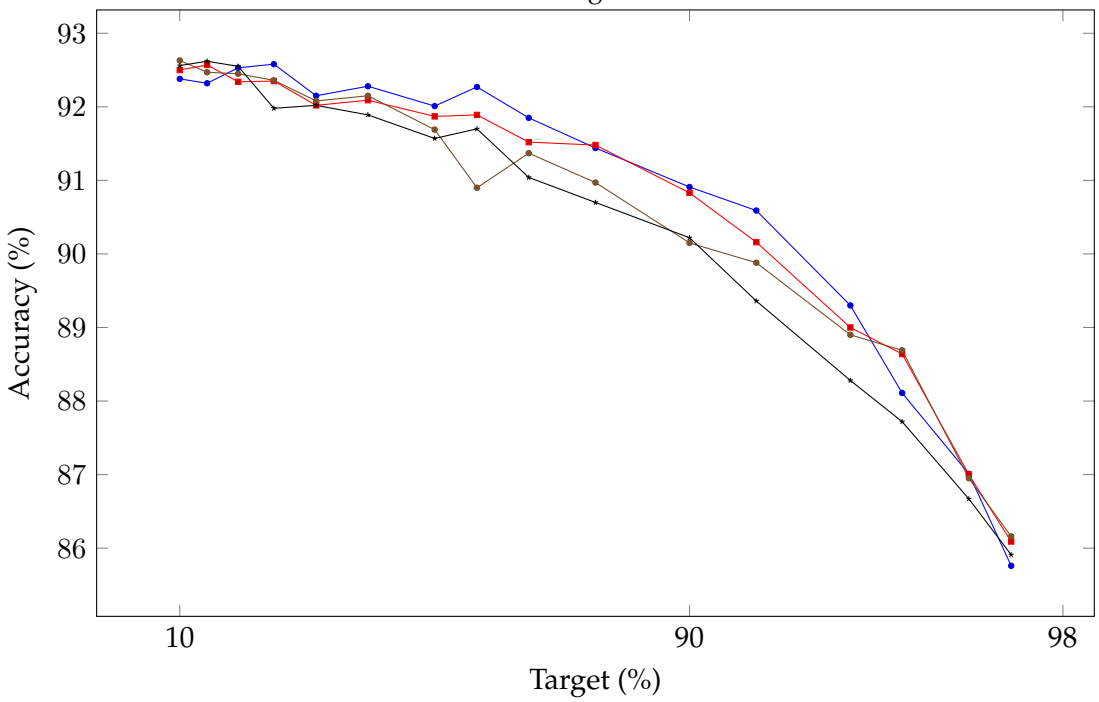
Figure 18: Hyper-parameter grid-search: ResNet-20 (64 base channels) on CIFAR-10; non-structured pruning rate of 90%. We tested several values of a_{min} and a_{max} (here renamed respectively a_{start} and a_{end} to test decreasing cases. We report two measurements: a) the final accuracy after pruning and b) the difference in accuracy between before and after the final removal of weights (because SWD shrinks targeted weights to values close to 0, but not perfectly zero; setting them really to zero can have an impact on accuracy).

- Best final performance are obtained for low a_{start} and high a_{end}, which is consistent with our initial motivation to first let the network learn without constraint and to end with a penalization strong enough to prune weights.
- As soon as a_{start} is big enough (above 10^{0}), SWD seems to reach sub-optimal yet rather good performance, whatever the value of a_{end}. In term of post-removal accuracy drop, it seems that a threshold on both a_{start} and a_{end} independently and abruptly makes SWD shift into a regime where it seems to succeed at pruning. We do not know yet the reason for such a radical behavior. A plausible explanation would be that crushing a weight early makes it less prone to recover, even if the penalty is smaller further during training.

Most importantly, two capital observations can be drawn: 1) these results validate our initial assumption on the role of the increase of a and on the choice of its initial and final values and 2) for well chosen values of these hyper-parameters, there is no difference before and after removing weights, which means that SWD succeeded at effectively pruning them—which allows to say that SWD is effectively a removal method and not just a regularization that helps pruning.



(a) Whole figure



(b) Close-up on points below 98% of pruning.

$a_{min} = 0.1/\ a_{max} = 10^4$ $a_{min} = 0.1/\ a_{max} = 5 \cdot 10^4$
$a_{min} = 1/\ a_{max} = 10^4$ $a_{min} = 1/\ a_{max} = 5 \cdot 10^4$

Figure 19: ResNet-20 (16 channels) on CIFAR-10, non-structured SWD. The x-axis is the percentage of pruned parameters and the y-axis the accuracy. Each curve corresponds to a different a_{min}/a_{max} pair.

We did other grid-searches in the original publication [1], in order to study how this behavior depends on the architecture, dataset and pruning rate. It turned out that the frontier, that separates regimes for which SWD functions properly or not, tends to move slightly depending on these factors—notably, structured pruning seems much more punitive—, but the overall behavior remains the same. Therefore we thought them not to be especially interesting to report in this manuscript and we invite readers, if interested, to check the original paper instead.

However, another experiment that is interesting to show is how values of a_{min}/a_{max} influence the overall parameters/accuracy trade-off. Figure 19 shows these results.

Multiple observations can be drawn from Figure 19:

- Figure 19 seems to be divided into two regimes below and above 98% of pruning. The blue curve, that looks to be consistently the best below 98% under-performs above this threshold; respectively, the black curve, that was the worst, looks to be the best for very high pruning rates. The difference between the two is that the blue curve has both the smallest a_{min} and a_{max} while the black one has the highest ones. However, the green curve (high a_{min} and low a_{max}) almost always performs worse than the red curve (low a_{min} and high a_{max}). These tendencies seem to suggest a certain rule of thumb to choose the right hyper-parameters pair for a given pruning ratio: when the pruning rate increases, first increase a_{max} and then a_{min}. This also suggests that: 1) a too strong penalty harms the network for low pruning rate but is necessary to make pruning work for high rates and 2) increasing the initial penalty, as also suggested by Figure 18, helps enforcing sparsity from the start, which makes pruning easier afterwards.
- Some points are visible outliers, for example $0.1/5\cdot10^4$ for a pruning rate of 99.8%. This sudden drop in accuracy reveals some instability, whether it is in SWD or in the ability of the network to handle pruning. Yet, the most plausible explanation, that can be raised when considering some posterior works of ours [2], [3], is that some variance in the distribution of sparsity may have produced some layer collapse [98], which makes the real number of active parameters considerably drop while we did not pay attention to such a possibility in our measurements back then.

To summarize the overall conclusions of our experiments: with the right choice of hyper-parameters, SWD is able to effectively prune networks, so that removing these weights does not impact the network. Choosing the right hyper-parameters, depending on the pruning rate, may be a bit more delicate; but is seems that it is better to increase both a_{min} and a_{max} for high levels of pruning, for example above 98% for example—the original publication also shows that structured pruning tends to require higher penalty values too.

We will now present experiments that show the performance of SWD, compared to some reference methods.

4.3.4 Experiments

We show results on ImageNet (Table 1), on CIFAR-10 (Figure 20 and 21) and Cora (Figure 22). We already detailed the experimental conditions and compared methods in Section 4.3.2.

Our motivation behind these choices of experiments is that ImageNet allows to get at-scale results, but is very expensive to compute; CIFAR-10 allows much more fine-

grained explorations, so we present extensive trade-off curves, first for an over parameterized, good performing and easy to prune network (ResNet-20, when its initial width is of 64 channels) and one much more parsimonious and punitive to prune (ResNet-20, when its initial width is of 16 channels); finally Cora allows to test SWD on a completely different type of task, which, in this case, is node classification. Chapter 6 also presents results on Cityscapes (semantic segmentation in urban environment), but they are extracted from completely different papers [2], [3].

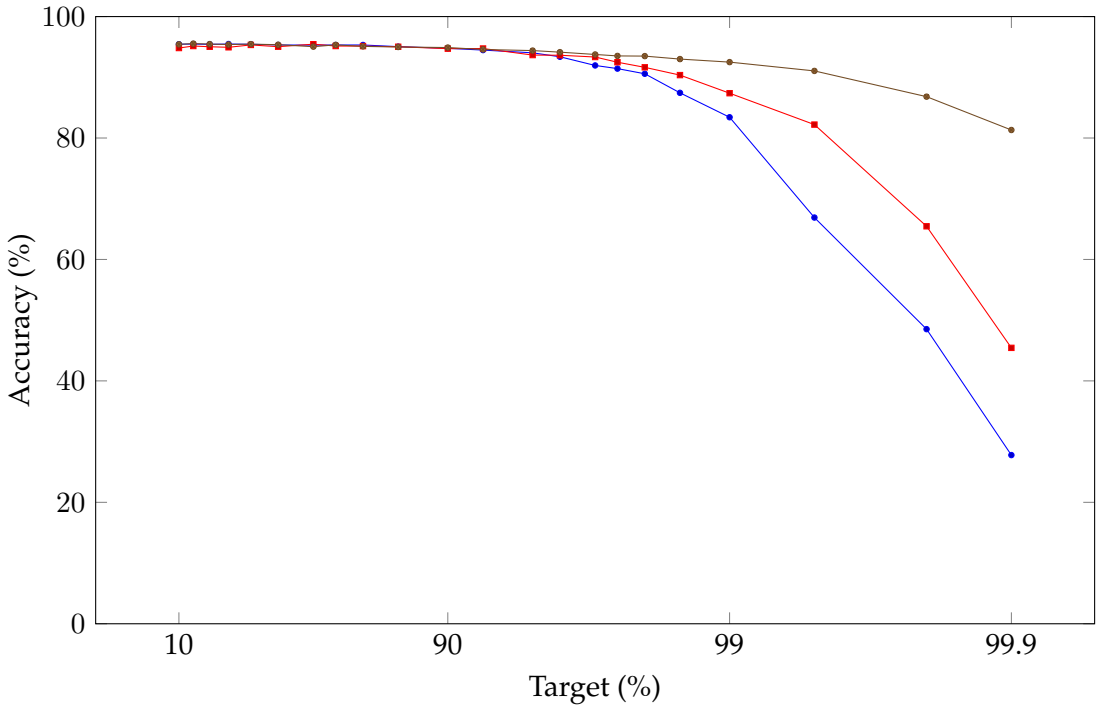
Experiments on ImageNet ILSVRC2012						
Non-structured pruning						
Target (%)	Han et al. [26]		+LRR [92]		SWD	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
50	74.9	92.2	58.4	82.1	**75.0**	**92.2**
10	71.1	90.5	54.6	79.6	**73.1**	**91.3**
2.5	47.2	73.2	34.8	61.54	**67.8**	**88.4**
Structured pruning						
Target (%)	Liu et al. [87]		+LRR [92]		SWD (+LRR)	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
90	**74.7**	**92.2**	56.1	80.7	74.2	91.9
75	73.4	91.6	51.1	77.1	**73.5**	**91.5**
50	63.6	85.7	40.0	66.2	**69.0**	**88.8**
20	0.1	0.5	0.1	0.5	**69.0**	**88.7**

Table 1: Non-structured and structured pruning on ResNet-50 on ImageNet ILSVRC2012, comparison between one base method (Han *et al.* [15] or Liu *et al.* [87]), its LRR variant [98] and SWD.

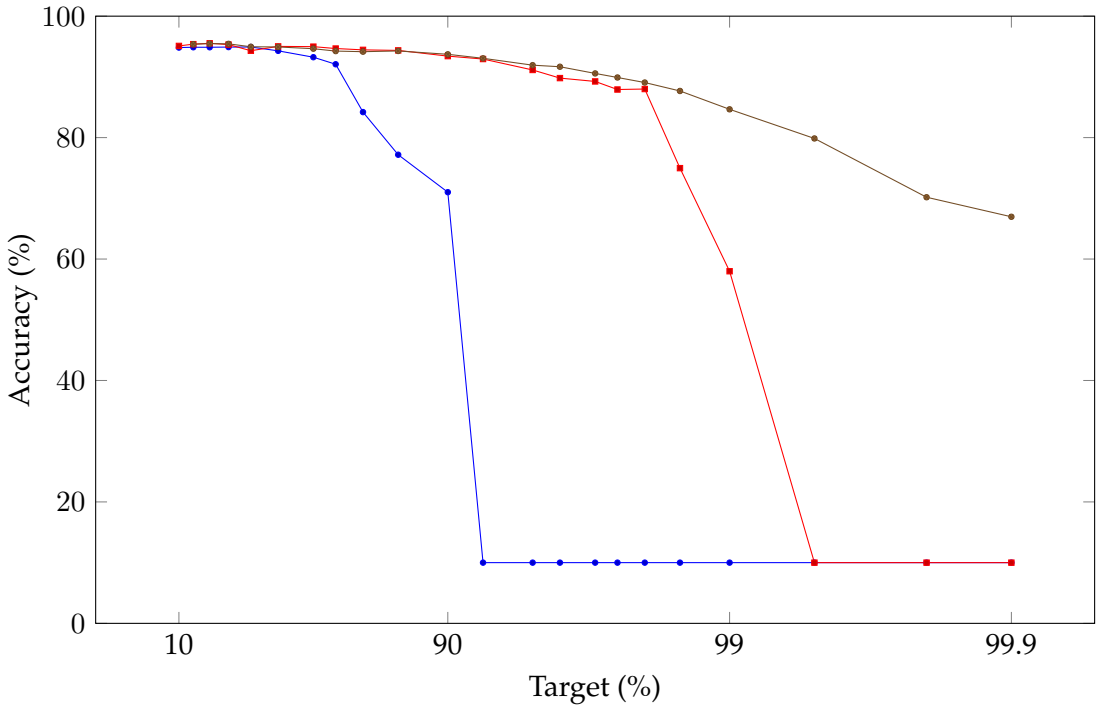
Table 1 shows performance on ImageNet, for both non-structured and structured pruning. SWD shows to yield almost systematically the best performance—even though we must mention that, for low pruning rates, the margin between the first reference method and SWD is not very significant, even when SWD lies second. However, two very clear tendencies can be noticed: 1) SWD performs substantially better for high pruning rates and achieves non-trivial performance where other methods destroy the network, and 2) LRR [92] surprisingly degrades the performance of the first reference method (while improving that of SWD), which is not a behavior to be seen in the other experiments. The reason why LRR under-performs in this specific situation while improving SWD (or improving the reference in the other experiments) is unknown to us; yet it would be possible to speculate on the respective ability of the reference and of SWD to produce efficient architectures that, during retraining, have different learning capabilities (however that would not explain why pruned and fine-tuned networks would perform better than the same network, pruned and warm-restarted).

Figure 20 and 21 show performance on CIFAR-10, using the same set of methods (except that SWD never uses fine-tuning or LRR here). The relative performance of each method seems more consistent with expectations in these experiments than in the case of ImageNet, and the more extensive number of points allows for more subtle observations:

- In the four displayed cases (non-structured/structured, 64 channels/16 channels), the same relative behavior between methods can be observed: 1) for low



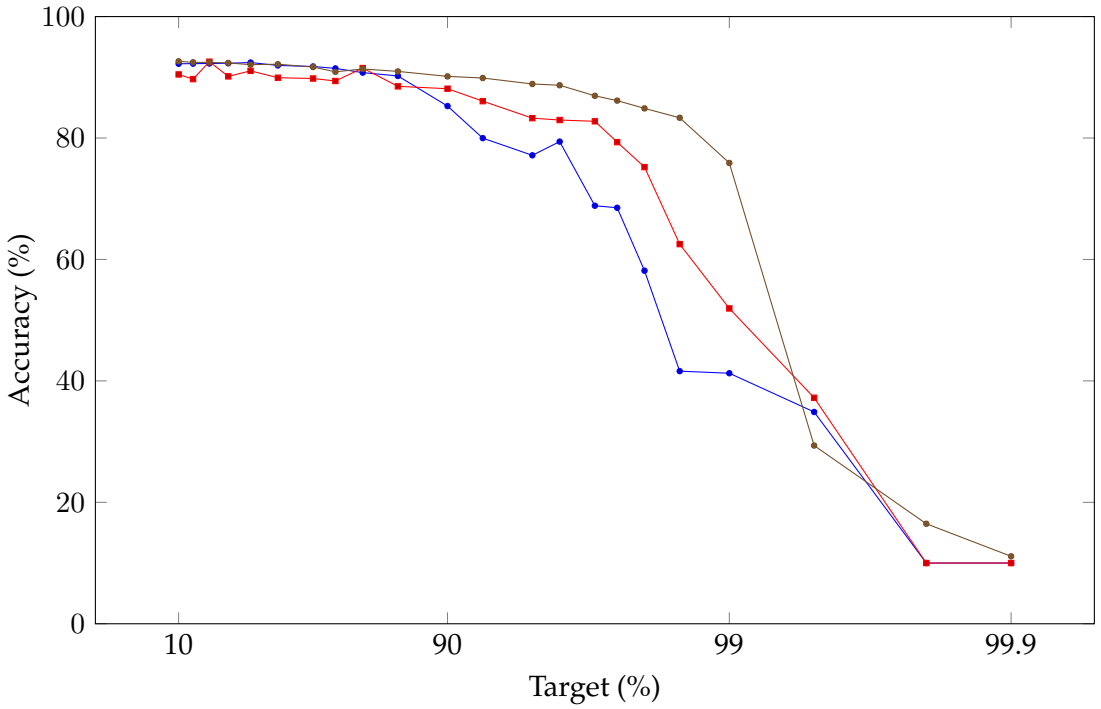
(a) Non-structured pruning



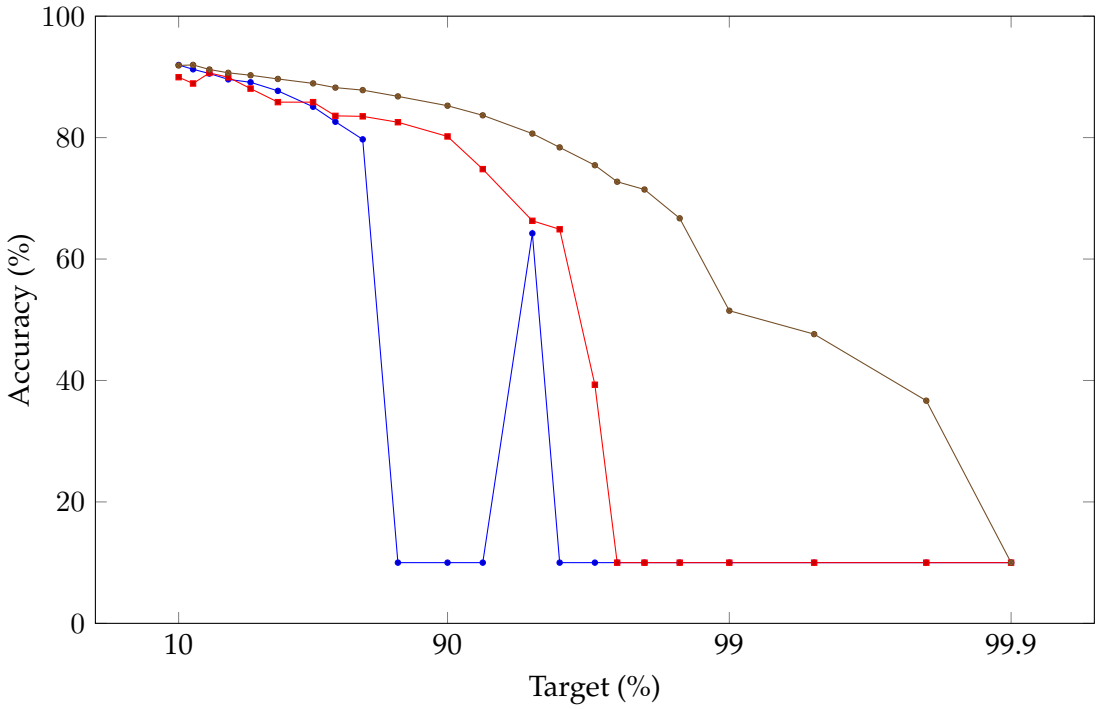
(b) Structured pruning

Magnitude pruning LR-Rewinding SWD

Figure 20: Non-structured and structured pruning on ResNet-20 (64 initial channels) on CIFAR-10, comparison between one base method (Han *et al.* [15] or Liu *et al.* [87]), its LRR variant [98] and SWD.



(a) Non-structured pruning



(b) Structured pruning

Magnitude pruning — LR-Rewinding — SWD

Figure 21: Non-structured and structured pruning on ResNet-20 (16 initial channels) on CIFAR-10, comparison between one base method (Han *et al.* [15] or Liu *et al.* [87]), its LRR variant [98] and SWD.

pruning rates, all methods tend to perform the same, with a notable under performance of LRR for 16 channels; 2) for high pruning rates, SWD performs notably better (with very rare exceptions), LRR lies clearly as second and magnitude pruning as last.

- ResNet-20, with 16 channels, is clearly more difficult to prune than with 64 channels: even SWD, that manages to stay at non-trivial performance for 99.9% pruning with 64 channels, falls to random-guess for the same pruning rate with 16 feature maps. Also, curves for 64 channels are much smoother, which means that pruning with 16 channels is much more unstable and may degrade performance in much more abrupt bursts.
- Structured pruning looks much more difficult than non-structured pruning. However, this must be put in perspective with multiple facts:
    - Structured pruning is done here in one-shot, while non-structured has 5 iterations; there is no doubt that structured pruning would perform better with iterations.
    - SWD also has more trouble with structured pruning, but we cannot know how the choice of hyper-parameters (that is made more difficult because of structured pruning) influences this aspect.
    - One-shot pruning, especially in the case of structured pruning, is quick to bring layer-collapse, which plausibly explains why networks pruned this way are quick to reach random-guess.
    - However, one last major aspect that puts all of this in perspective again is that, because our experiments were deterministic and because structured pruning is one-shot, points between magnitude pruning and LRR only differ by the way they are fine-tuned or retrained: in both cases we have the same network, with the same weights and pruned the same way. This means that, in some cases, fine-tuning is completely unable to save networks that are recovered through retraining, which means that they were not completely destroyed. This observation is very interesting, since it means that retraining is not just some sort of "better fine-tuning" and that the difference between their respective behavior is deeper than that; it also raises questions about what kind of inescapable local minima those pruned networks may have fallen into so that fine-tuning could not recover them, while retraining could.
- There is one point, for structured magnitude pruning on ResNet-20 with 16 channels that is a clear outlier: why does it reach non-random performance, while it was the case of the three points right before? Let's first assume that this point is not an error on our part. Even though Chapters 5 and 6 will show how the count of pruned parameters, in the case of structured pruning, can be deceptive, the fact that our experiments, for structured pruning, were deterministic and one-shot guarantees that the real count of pruned parameters should decrease monotonously with the increase in pruning rate: it is not possible for that outlier to secretly have more parameters than points right before. Moreover, the same point with LRR, that is pruned exactly the same way, is inserted in a much more consistent curve. This means that this weird behavior is not due to an error in the count of pruned parameters. Similarly, since the same pruning criterion is used, weights pruned for points right before are still pruned for this outlier, plus some additional ones. Considering all of these factors, we can speculate that, in

this particular case, it is possible that pruning first got the network stuck in an inescapable local minimum where some filters were detrimental to learning, and these filters were those pruned in the outlier, which let the fine-tuned network reach performance on-par with its LRR counterpart. We cannot affirm anything without further dedicated investigations, but, once more, this highlights the unpredictable influence of pruning relatively to the landscape of $\mathcal{L}$, as mentioned in Section 4.1.3.

- Finally, while SWD tended to out-perform other methods rather quickly on ImageNet, they all separate themselves much later: the tendency looks much clearer after 90% pruning. That separation appears a bit earlier with 16 channels or structured pruning, which seems to imply that the more difficult the problem is, the more the removal method has a significant impact.

To sum it up: despite some questionable methodological choices, SWD clearly shows better performance for high-pruning rates. However, the relative behaviors of fine-tuning and retraining, while it was not the original intended focus of these experiments, raise many interesting questions about the impact of pruning on training.

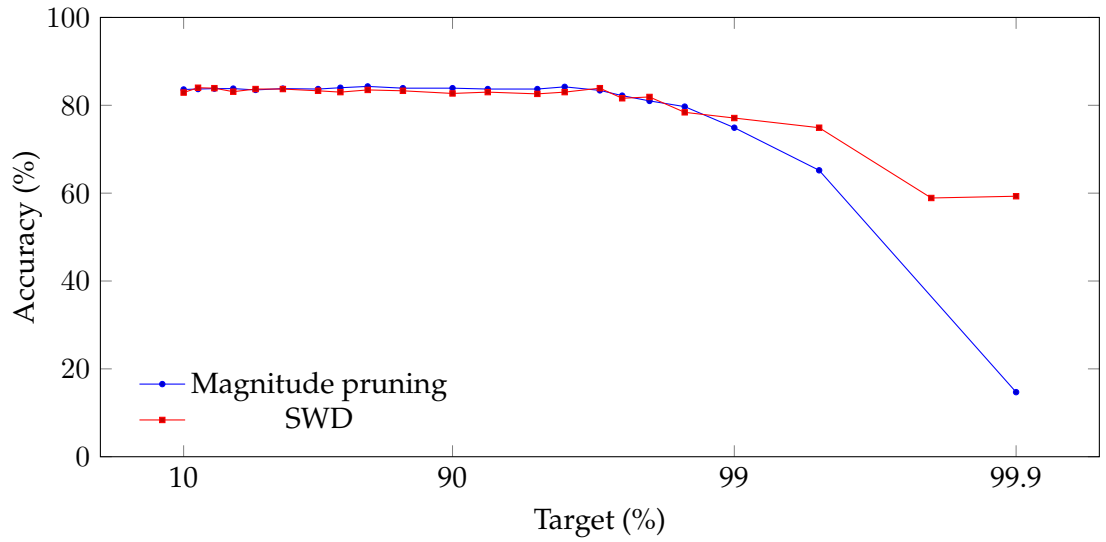


Figure 22: Non-structured magnitude pruning [26] and SWD applied on a Graph Convolutional Network[196] on the Cora dataset [195].

Figure 22 provides some results on the Cora dataset [195]. Once again, for low pruning rates, SWD and magnitude pruning seem to perform roughly the same (with SWD looking to be a very close second) while SWD performs significantly better for high pruning rates. Also, the two methods look clearly separated only after 99% pruning, which is very late—and consistent with our results on ImageNet and CIFAR-10, that seem to show that the more complex the task, the sooner methods are separated.

Finally, Table 2 shows how our results and experiments compare with the rest of the literature. Despite our efforts to select papers, where we can find the same dataset/architecture pairs that we used for our own experiments, we can see that not only do the different methods differ in so many aspects that they are barely comparable, but even the same networks, for the same datasets, yield different baseline accuracy each time. Furthermore, papers rarely compare for the same compression rates, or rarely provide enough data points to compare easily.

Method	Structure	Dataset	Network	Comp.	Accuracy
Liu et al. [197]	Weights	ImageNet	AlexNet	×22.6	56.82% (+0.24%)
Zhu et al. [90]	Weights	ImageNet	InceptionV3	×8	74.6% (−3.5%)
Zhu et al. [90]	Weights	ImageNet	MobileNet	×10	61.8% (−8.8%)
Xiao et al. [162]	Weights	ImageNet	ResNet50	×2.2	74.50% (−0.40%)
SWD (ours) *	Weights	ImageNet	ResNet50	×2	75.0% (−0.7%)
SWD (ours) *	Weights	ImageNet	ResNet50	×10	73.1% (−1.8%)
SWD (ours) *	Weights	ImageNet	ResNet50	×40	67.8% (−7.1%)
Liu et al. [87] *	Filters	ImageNet	ResNet50	×2	63.6% (−12.1%)
Luo et al. [127]	Filters	ImageNet	ResNet50	× 2.06	72.03% (−3.27%)
Luo et al. [127]	Filters	ImageNet	ResNet50	× 2.95	68.17% (−7.13%)
Molchanov et al. [97]	Filters	ImageNet	ResNet50	×1.59	74.5% (−1.68%)
Molchanov et al. [97]	Filters	ImageNet	ResNet50	×2.86	71.69% (−4.49%)
SWD (ours) *	Filters	ImageNet	ResNet50	×1.33	74.7% (−1.0%)
SWD (ours) *	Filters	ImageNet	ResNet50	×2	73.9% (−1.8%)
Liu et al. [87]	Filters	CIFAR10	DenseNet40	×2.87	94.35% (+0.46%)
Liu et al. [87]	Filters	CIFAR10	ResNet164	×1.54	94.73% (+0.15%)
Ye et al. [134]	Filters	CIFAR10	ResNet20−16	×1.6	90.9% (−1.1%)
Ye et al. [134]	Filters	CIFAR10	ResNet20−16	×3.1	88.8% (−3.2%)
SWD (ours) *	Filters	CIFAR10	ResNet20−16	×1.42	91.22% (−1.15%)
SWD (ours) *	Filters	CIFAR10	ResNet20−16	×3.33	88.93% (−3.44%)
Liu et al. [87] *	Filters	CIFAR10	ResNet20−64	×2	94.92% (−0.75%)
SWD (ours) *	Filters	CIFAR10	ResNet20−64	×2	94.96% (−0.71%)
SWD (ours) *	Filters	CIFAR10	ResNet20−64	×50	89.07% (−6.5%)

Table 2: Comparison between results from our experiments (marked with a star) and results indicated in the original papers. We compare both structured and unstructured methods, for ImageNet or CIFAR-10 and various architectures, including ResNets with an initial embedding of either 16 or 64 channels. We report the compression rate (in term of parameters), the final Top-1 accuracy and the overall accuracy degradation.

This means that this kind of table, apart from giving an initial overview of whether or not the results of one's experiment are up to the standards of the literature, does not provide much usable information. This is a problem that has already been discussed by Blalock *et. al.* [27], and motivated our way to conceptually dividing pruning into three different aspects.

Now that we have reviewed these experiments, that demonstrated the relevance of SWD as a pruning method, we can conclude this chapter.

Recapitulation

In this chapter, we first exposed what was a removal method, what was the most basic one in the literature and what were its problems—notably how its abruptness disturbed training. Exposing the various ways to tackle these problems, that are related to some fundamental aspects of how to train neural networks, allowed us to present thematically what other removal methods could be found in the literature. This in turn allowed us to present the motivations behind our own method, SWD [1].

We then defined SWD, showed our experimental conditions and reference methods. We then presented two sets of experiments: first, ablation tests to demonstrate the relevance of the definition of SWD and to study the impact of its hyper-parameters; then, performance tests that showed the benefits of SWD as well as some interesting behaviors from reference methods, which raised some more fundamental questions. Indeed, we saw that not only do removal methods tend to actually behave the same when the pruning rate is low—which is understandable because, in such a case, the abrupt transformation, to smoothen through using a given method, is already small enough—but also that the ability of a pruned network to recover performance through fine-tuning or retraining is still difficult to predict and understand, which deserves further investigations.

To conclude, the literature concerning removal methods is one of the most dense and hazy aspects of pruning. It counts many very different methods and families of methods that, unfortunately, are likely to behave mostly the same (as evidenced by works such as that of Gale *et al.* [153] or even our own results, presented in this chapter). Furthermore, the difficulty of comparing pruning methods, because of confusing methodologies (as shown by Blalock *et al.* [27]) or of not distinguishing the separate influence of pruning criteria and structures or of hyper-parameters such as the number of pruning iterations—which we highlighted while discussing our experiments—, could imply that creating new methods, with the motivation of improving upon the state of the art, may be a bit futile as long as we do not have more rigorous ways of comparing and studying removal methods (or pruning methods in general). All the discussions in this chapter, as well as the way we chose to present the literature, give possible hints about which aspects to dwell into in future works, in order to help developing a more objective way of tackling this whole topic, as well as to identify which are the most fundamental problems to truly tackle to improve pruning.

Now that the most theoretical aspects of pruning—the criteria and the method—have been dealt with, we will explore the more practical considerations of pruning: pruning structures, in Chapter 5 and their impact on the energy consumption of pruned networks on embedded hardware in Chapter 6.

Chapter 5

Pruning Structures

Contents

This chapter will tackle the topic of pruning structures, *i.e.* the type of elements to remove from a neural network to reduce its cost. The choice of the pruning structure depends on multiple aspects: the aim of the pruning method, the type of cost to reduce or the ability of the targeted hardware to handle sparsity. On the other hand, the type of structure also influences the difficulty of pruning, as we saw in Chapter 4, as well as the architectural modifications it may imply can introduce various types of discrepancies, which means that all kinds of structured pruning or distributions of pruning may not be possible. Thus, the topic of pruning structures involves two aspects: one more related to hardware and implementation, the other more related to how the pruning structure impacts the overall pruning methodology. This is why we separated the two topics: Chapter 6 will tackle implementation-related questions while this chapter will tackle more methodological aspects.

In this chapter, we will first review the specificity of each type of pruning structure. We will then expose the type of discrepancies they may introduce and what are the types of sparsity distribution, to be found in the literature, meant to avoid these discrepancies.

5.1 Types of Structures

We will review here the various types of structures to be found in the literature, from the finest to the coarsest, starting with "non-structured" pruning. As we will see, different types of structures bring different types of gains, and therefore can be combined. However, the finer the structure, the more complex the implementation tends to be, or rather: many larger types of structures were initially explored to avoid the technical difficulty implied by how a fine structure, by definition, impacts more elementary operations of networks and, therefore, requires more low-level and technically challenging implementations. Even though filter pruning is a type of structured pruning that is so widespread that it is considered as the default kind of "structured pruning"—which is also the reason why we will mainly focus on this type in this chapter and Chapter 6—, we will present it, in this section, the same way as the others.

5.1.1 Non-Structured Pruning

Non-structured (or sometimes "unstructured") pruning is, as its name indicates, considered as the default type of pruning. It involves removing isolated weights [26], which is the finest possible pruning granularity. Because of this, it generally allows a better parameters-to-accuracy trade-off, but since it is more difficult to leverage [82], [83], many methods instead favor structured pruning. The various strategies to be found in the literature about how to leverage non-structured pruning will be reviewed in Chapter 6.

Many papers still use non-structured pruning nowadays [167], [198], [199], despite multiple works advocating in favor of structured pruning against non-structured pruning [82]. One reason for this is simple: many pruning papers do not bother being exploitable on hardware and focus instead on more theoretical aspects of pruning. In such cases, worrying about hardware or structures can be a major obstacle to some studies; not only because structured pruning brings its own problems that non-structured pruning does not encounter, **cf.** Section 5.2, but also because some pruning methods [156] are so difficult to adapt to structured pruning that it may be the topic of whole distinct contributions [185], [186].

Therefore, non-structured pruning is often used in cases where one may not want to be encumbered by the limitations of structured pruning. This is the reason why most

papers whose approach is more theoretical or fundamental use non-structured pruning [28], [92], [98], while those that tackle the actual hardware acceleration of pruned networks mostly use structured pruning [121], [200], [201].

Before moving on to the next section, we must anticipate a little bit on Chapter 6 and remind that various implementation strategies of non-structured pruning exist and can bring different types of gains—for example, some may save energy consumption while not providing speed-up. As we will see later, each type of pruning structure can bring its own distinct gains, and since non-structured pruning, depending on the implementation, can bring some, it can be combined with the others—even though it would be insufficient, for example, to reduce the size of intermediate representations. We had to clarify this point in order not to leave the impression that non-structured pruning was just a sort of oversimplified pruning, relegated to theoretical works only.

5.1.2 Constrained Sparsity

The finest types of structured sparsity involve producing sparse tensors whose distribution of sparsity allows some optimization, while not allowing an actual rearrangement of the weights—unlike filter shape pruning, shift layers or filter pruning. This is why we call it "constrained sparsity", as it involves introducing some rules in the way of sparsifying tensors instead of really pruning larger structures.

Such types of constrained sparsity often aim at arranging non-zero elements in tensors in a way that optimizes memory access, reutilization and cache usage [202]–[204]. Notably, multiple methods divide tensors into blocks that each contain a given number of non-zero elements, whether it is constant and defined [204] or simply bounded [203]; the distribution of sparsity inside these blocks stays random. This division into blocks has multiple advantages: 1) each block can be treated separately, which fits well the parallelization power of GPUs, 2) the workload of each of these parallel threads is more predictable thanks to the constraint on the number of non-zero elements and 3) the introduced regularity and parallelization helps avoiding expensive buffering operations.

Each of these papers [202]–[204] comes with its own proposal of implementation, which indicates how little generic this approach unfortunately is. Indeed, it highly depends on hardware and is non-standard, which means that using this approach requires a lot of engineering and a fitting hardware to be exploitable.

5.1.3 Filter Shapes

"Filter shapes" pruning involves pruning the same weight in the same kernel of every filter [82], [130]. The reason why such a type of sparsity can be leveraged comes from the fact that many widespread frameworks and libraries actually transform convolutions into GeMM using the *im2col* operation [17]. This transformation involves rearranging the weights tensor into a matrix, so that removing the same element in all kernels of a filter removes a column of the said matrix.

This method has multiple advantages: not only is *im2col* sufficiently widespread for this pruning method to be considered reasonably generic, but it also produces different types of gains. Indeed, not only is the unfolded weights matrix smaller, which means less memory occupation and fewer operations, but it also allows for a smaller temporarily dilated input matrix, which further reduces the runtime memory occupation.

This last point deserves some explanations: to unfold a convolution into a GeMM, the weights are simply rearranged while the input pixels are duplicated into a pseudo-

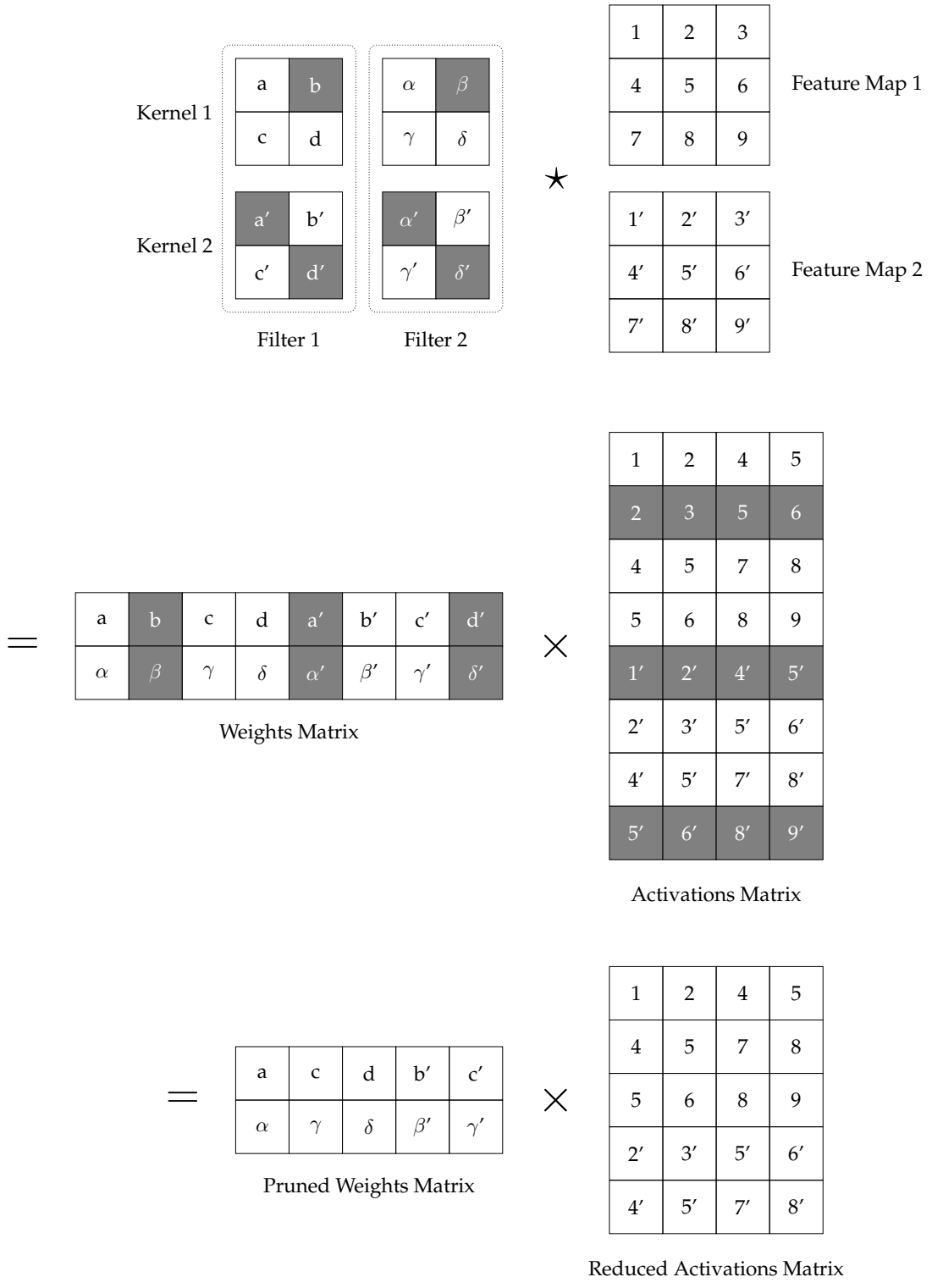


Figure 23: Pruning the same weight in the same kernel of each filter of a convolution layer allows pruning columns in the weight matrix after the *im2col* operation, that turns the convolution operation into a GeMM (General Matrix Multiplication, as we will see in Chapter 6). Removing columns in the weights matrix also allows removing rows in the activation matrix, which helps limiting the memory overhead of the *im2col* operation. Therefore, filter shape pruning allows reducing the memory usage and number of operations.

Toeplitz matrix. This means that this method requires extra memory space to store the dilated activations. Reducing the dimensions of the weights matrix means that the activation can be less duplicated, which means that the gain in memory usage is actually twofold. The principle of filter shape pruning is summed up in Figure 23.

5.1.4 Shift Layers

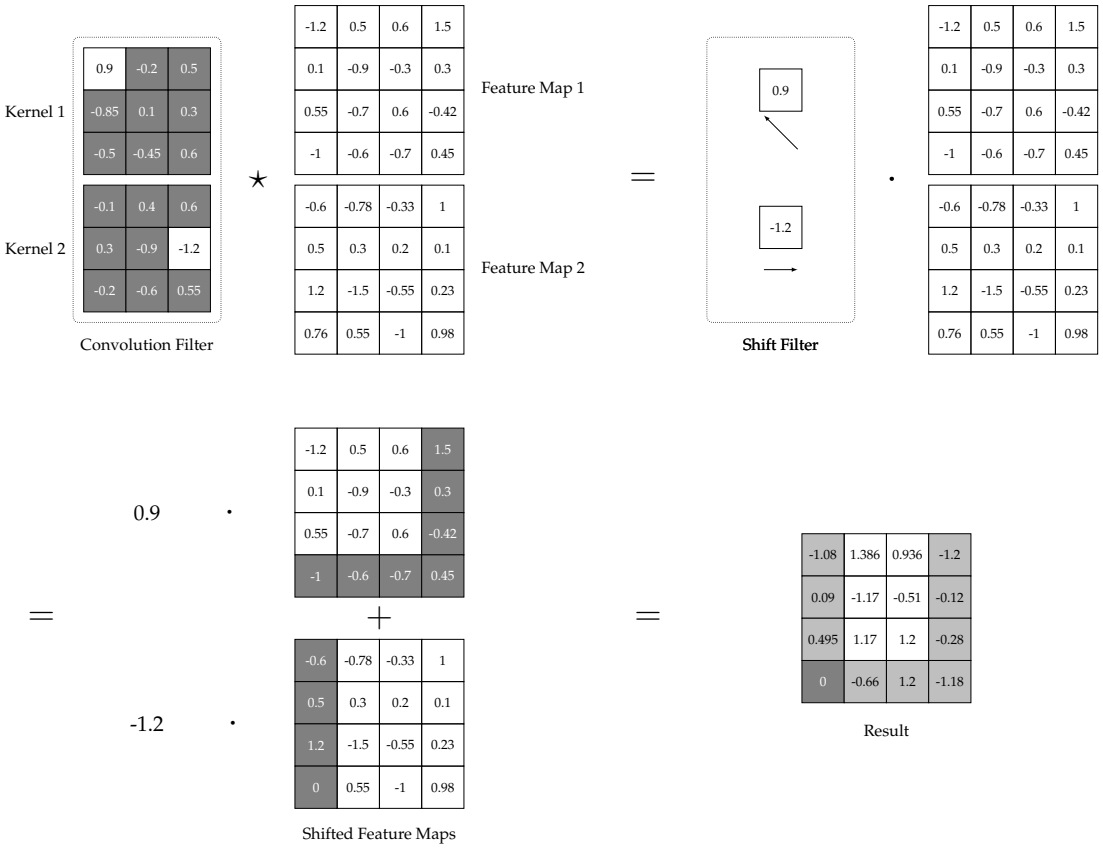


Figure 24: Pruning all weights but one in a convolution kernel allows transforming into the combination of a shifting operation and of a multiplication. The shifting itself can also be considered as a cropping.

When all parameters from a convolution layer but one in each kernel are pruned, the convolution can be transformed into the combination of a shift operation and a 1×1 convolution [123], [205]. Even though the shifting requires storing the direction of the shift, which is non negligible at such a compression rate, the resulting operation allows for efficient implementations [206], although non-standard. The principle of shift layer pruning is summed up in Figure 24.

5.1.5 Grouped Convolutions

Grouped convolutions already exist in multiple architectures, either to reduce the cost of a convolution operation or to produce depth-wise separable convolutions [37], [38]; their principle is explained in Section 2.1.3. The advantage of grouped convolutions is that they are perfectly well handled by any framework and offer another way of reducing easily the number of weights and operations of a network, without reducing its number of filters.

Surprisingly, pruning a convolution layer to produce a grouped convolution layer is a rare practice, despite its relevance [207].

5.1.6 Filter Pruning

The most common type of structured pruning involves removing whole convolution filters (or "neurons" in the case of linear layers). This type of pruning has unique advantages which make it particularly interesting despite its apparent coarseness:

- Removing a whole filter means one can directly reduce the dimensions of the weights tensor, without having to apply *im2col* beforehand; this means that any framework can leverage this reduction.
- Therefore, removing filters allows reducing both memory occupation of the weights and the number of operations.
- Since each filter produce one output feature map, this means that removing a filter also reduces the dimensions of the output; this allows to significantly reduce the memory usage of intermediate representations, which is not the case for other types of pruning.

The principle of removing whole neurons instead of isolated weights is as old as pruning itself [108], [115], [139], [208], but filter pruning has been mostly popularized by some papers such as those by Wen *et al.* [130] and Li *et al.* [83], even though some other papers came right before [119], [124], [126], [129]. Since then, many influential papers focus on filter pruning [87], [97], [112], [121], [157], [209], [210]. Filter pruning, thanks to its unique advantages and its popularity, is considered as the default "structured pruning"—so that this term is often used to talk about filter pruning [27], even though it could also apply to all other types.

Filter pruning can also be mentioned as "neuron pruning" (which is perfectly equivalent) or "channel pruning" (which is almost perfectly equivalent but emphasizes one subtlety that we will review in Section 5.2).

5.1.7 Whole layers

Another rare practice is to prune deliberately whole layers [130]. This is made possible thanks to residual connections, that allow removing whole blocks without destroying the network. This is the coarsest type of pruning and obviously leads to the most gains; however it is so coarse that it can be barely exploitable; moreover, due to *layer-collapse* [98], it tends to happen automatically when doing global filter pruning (i.e. setting an overall pruning target on the network, using the same pruning criterion on all layers, so that layers can be pruned at different rates).

5.1.8 Neural Architecture Search

In Chapter 2, we mentioned NAS as a compression method, distinct from pruning, meant to produce efficient architectures. Actually, many of them use principles that are so similar to that of pruning that they can be considered as just another kind of pruning. Notably, one-shot NAS [211] involves choosing sub-graphs of a unique "supernet" using various kinds of criteria, which makes this type of NAS almost perfectly a pruning method. Some zero-shot NAS methods [212], [213] even use criteria and methods directly borrowed from the pruning literature, such as SNIP [99], GraSP [169] or SynFlow [98].

Therefore, we can consider the various types of NAS search spaces as different types of pruning structures; for example, using NAS to define a block, to repeat to produce whole architectures, is analogous to prune the same layers in every block of the same type in a network.

5.1.9 Conclusion on pruning structures

Structure	**Reduces**				**Implementation**
	Memory			Operations	
	Parameters	Temporary	Outputs		
Weights	✓	✗	✗	?	Custom
Constrained	✓	✗	✗	✓	Custom
Shift Layers	✓	✓	✗	✓	Custom
Filter Shapes	✓	✓	✗	✓	*im2col*
Grouped Convolutions	✓	✓	✗	✓	Standard
Filters	✓	✓	✓	✓	Standard
Layers/Blocks	✓	✓	✓	✓	Standard

Table 3: Summary of pruning structures: whether or not they are able to reduce the memory occupation and movement of a) parameters, b) temporary representations of data during the execution of a layer or c) intermediate representations of data between layers, or to reduce the number of operations—which, in the case of weights is "?", as it heavily depends on the implementation and sometimes their energetic consumption but not their number. We also remind conditions to leverage each type of sparsity. "Standard" implementations correspond to cases where any framework, meant to run inference of neural networks, can leverage it, as these types of sparsity do not need new operators.

To conclude this section, Table 3 summarizes the respective gains and constraints of each type of pruning structure we reviewed. Since filter pruning is the most widespread type of structured pruning in the literature—and we can see now that it is because of both its easier implementation and its interesting gains while staying reasonably fine-grained—, this is the one type we focused onto, notably in Chapter 6 and in the following sections.

5.2 The Problems of Filter Pruning

We mentioned that removing filters reduces the size of layers, which any framework can leverage. Even though it is perfectly true, there are some problems that we did not mention and that have a significant impact on how to perform structured pruning. We will detail these problems in this section, while standard solutions will be presented in Section 5.3.

5.2.1 Dependencies Between Layers

The first problem of filter pruning is that the dimensions of layers in a network are inter-dependant: if a layer contains n filters, then the output tensor contains n feature-maps and every filter of the following layers, that take this tensor as their input, must contain n kernels. Therefore pruning a filter requires to remove one kernel in each filter of the following involved layers. For this reason, pruning a filter (and its bias) is equivalent to pruning a feature map (or "channel"), which is the reason why filter pruning can also be mentioned as "channel pruning" in the literature.

This type of dependencies is mentioned as soon as 2016 by Li *et al.* [83]. However, it was only shown in the context of a straightforward VGG-like network, in which such types of dependencies are very simple to identify. However, in the case of a ResNet-like network, the problem gets much more difficult.

Residual connections are ubiquitous in modern networks; unfortunately they also are the main source of problems for filter pruning. Indeed, not only do residual blocks sum together two tensors, that must be of the exact same dimensions (which is not guaranteed anymore after pruning), but also one of these two tensors come from a residual connection and not from one single identifiable layer. This means that the dimensions of this particular tensor cannot be deduced from the shape of a tensor, but of a complex combination of all layers upstream.

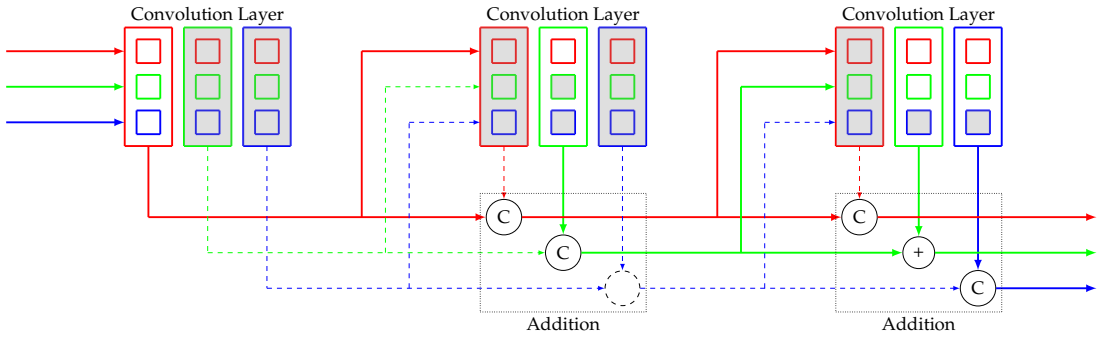


Figure 25: Example of problems introduced by pruning in the presence of residual connections.

Figures 25 illustrates the types of problem quickly encountered when pruning a network with residual connections. Detailed explanations: here are three convolution layers, with each three red-green-blue filters, each with three kernels of corresponding colors. The greyed-out filters are those that are pruned manually. Dashed arrows correspond to the pruned, missing feature maps. The output of the first layer has dimensions that match the pruning pattern of the said layer.

The second layer has greyed-out, pruned kernels to match the dimensions of its input. The second layer has also pruned filters, but not the same ones as the first layer. Indeed, the first layer had only the red channel remaining whereas the second layer only outputs a green channel. Therefore, both layers have only one output channels, but since they are not the same, they should not be summed together (like the default addition operation would do) but concatenated together (hence the "C" in the circles in the figure). This already raises two problems: first, the result of this concatenation contains two channels instead of one, which is not trivial to predict if one only looks at the dimensions of the layers; second, the addition operation of residual blocks would not have the right behavior, since it would add together the two single-channel tensors while a concatenation is needed—which already suggests that the addition operation

should be replaced to fit pruning.

The third layer has pruned kernels to match the input's dimensions and only one pruned filter. First problem: we already saw that the input's dimensions can be hard to predict, which means that the exact dimensions and number of parameters of this layer directly depend on one's ability to predict the said dimensions. Second problem: the green-blue output must be summed with the red-green input, which means that the addition must be replaced with a more complex mix of concatenations ("C") and additions ("+"). Not featured in the figure: the case where the two tensors to sum do not have the same number of channels, which a normal addition operator could not handle at all.

To conclude, the dimensions of intermediate representations and, consequently, the input dimensions of layers depend on the combination of all pruning patterns in the network. Therefore, all dependencies and combinations must be reliably identified in order to have the right number of parameters (by removing the right amount of kernels in filters to accommodate the actual dimensionality of the inputs) and to get a functioning network (because a mismatch between the dimensions of a layer and its input causes a fatal error on runtime). This last error can be circumvented when pruned parameters are put to 0 instead of being properly removed, which is sufficient when studying only the impact of pruning on the network's function.

5.2.2 Orphan Biases

Another problem that can occur when doing filter pruning is to forget removing biases when pruning away a filter. Indeed, if a filter is removed but not its bias, the said bias will keep producing a constant output. This has two consequences: 1) the resulting "orphan bias" is not trivial to implement and, because of its arguable uselessness, ultimately not worth the effort and 2) the produced uniform and constant channel prevents removing the corresponding kernels in the following layers. This second consequences has itself two consequences: 1) the compression rate is not as high as it could be, because of these useless remaining weights and 2) these weights cannot be removed because their constant contribution to the function, although useless, still impacts the distribution of intermediate representations, which means that removing them, while not hurting the learning capacity of the network, may still alter its performance if a fine-tuning is not performed.

Forgetting to remove biases can sound silly at first, but there are two aspects that can make it surprisingly commonplace:

1. Biases in convolution layers are nowadays mostly contained in the following batch-normalization layer, which means that pruning only the convolution layer is actually insufficient to really remove the whole filter. Preventing this issue makes the implementation of pruning a bit less trivial, as it implies finding the corresponding batch-normalization layer and pruning it the same way as the convolution layer.

2. When a whole layer is pruned, all kernels in the filters of the following layers are removed; more concretely the whole following layer is removed too. But since this removal is secondary, it is way more easy to forget removing the biases of this layer. Moreover, if all types of secondary removal are overlooked (by pruning filters but not following kernels), then the kernels do not contribute anymore to the function (because of the empty input) but the biases do. This means that the layer collapse [98] does not propagate properly and only the layers upstream are

removed while the whole branched should be pruned.

All these aspects—layers interdependence and orphan biases—make pruning properly a network much more difficult and subtle than it may appear first. In complex networks such as HRNet, the complexity can become overwhelming, which calls for an automatic and generic method to both identify all dependencies and replace all needed operators in order to make pruning easier to do properly. We will present such a method in Chapter 6.

Far from being a trivially technical question, this complexity made a wide part of the literature think that some types of filter pruning were not possible at all, as we will see in Section 5.3. Other papers also completely ignore this issues and therefore report wrong compression rates (even though this fact is sometimes acknowledged by the same papers).

5.3 Distribution Strategies

As previously mentioned, the dimensional dependencies between layers is something acknowledged by Li *et al.* [83], whose work is seminal in the field of filter pruning and whose observation is shared by many following papers.

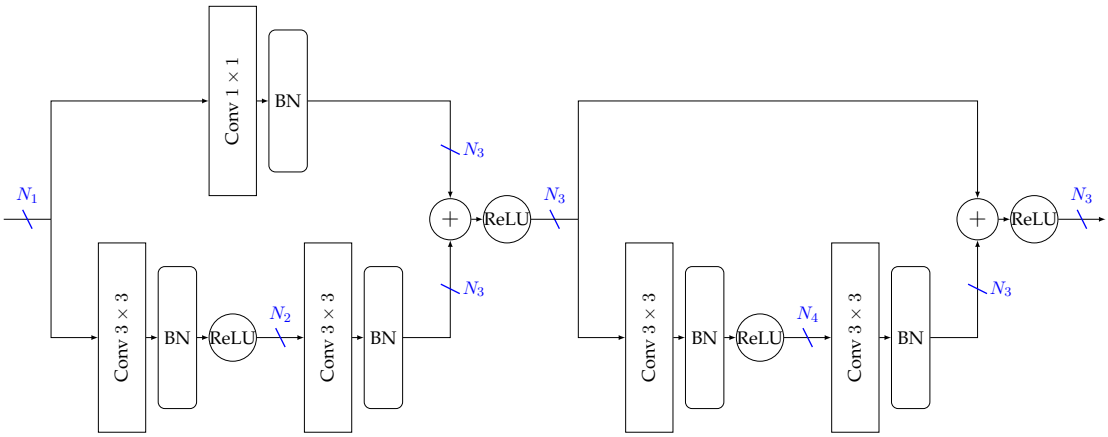
Therefore, Li *et al.* [83] prune filters and their corresponding kernels in VGG-like networks or in the first layers in residual BasicBlocks (or the first two ones in BottleNecks). But in the case of the last layers of residual blocks, Li *et al.* [83] say that it is necessary for both branches to be pruned the same way (which eliminates the previously mentioned problems). Because of this, the last layer of every block in a ResNet stage is pruned the same way: the way the upstream shortcut layer is pruned (i.e. the 1×1 convolution inserted in the residual connection to change the dimensions of the corresponding tensor each time the number of channels of ResNet is increased).

Such a choice implies that, in a ResNet made of BasicBlocks, actually half the layers in the network are completely constrained in the way they can be pruned. This type of distribution for filter pruning is still very widespread to the point of being almost standard, as it can be seen in many papers that propose methods meant to produce acceleration [141]–[144]. This strategy corresponds to Strategy A in Figure 26, in which we can see that it allows very few variations in the number of channels between layers—we used *BasicBlocks* (cf. Chapter 2) as an example.

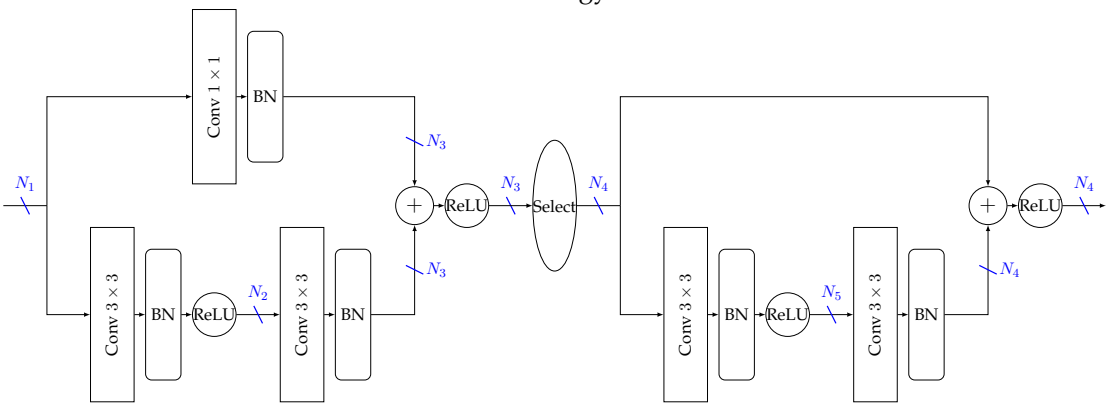
Another solution to eliminate the problem is to add a channel selection layer (or even a batch-normalization layer in cases where pruned filters are simply put to zero) at the end of the residual block, which makes the dimensional dependencies much shorter and easier to handle [1], [87]. Pruning this channel selection layer allows to prune very simply both branches at once, without the need to constrain the same dimensionality on all blocks in a stage. This strategy corresponds to Strategy B in Figure 26.

Finally some works propose solutions to accommodate dimensional mismatches with custom operators [112], [201], [214], [215], even though it often involves padding some channels with zeros to keep the dimensional consistency between tensors to sum. This strategy corresponds approximately to Strategy C in Figure 26.

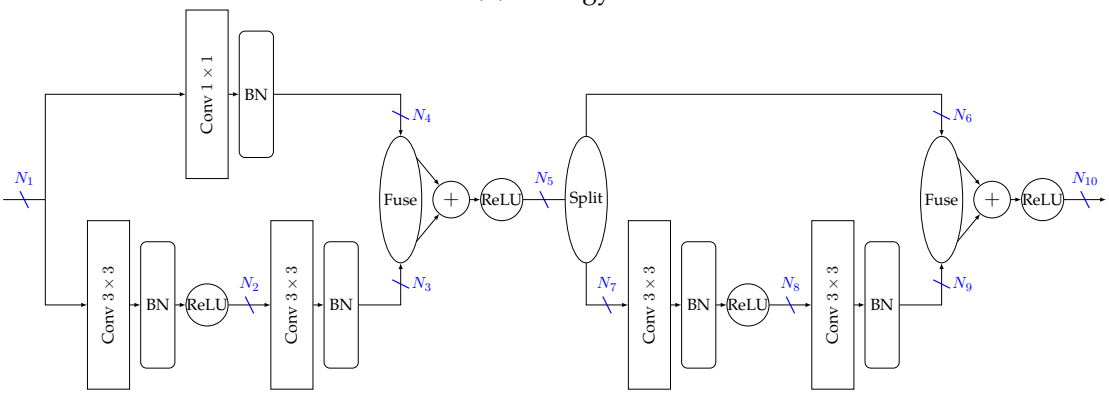
In Chapter 6, we will present another solution, akin to Strategy C. In this chapter, we refer to Strategies A and B as "constrained pruning", as they limit the degrees of freedom allowed by Strategy C—indeed, the number of degrees of freedom, in Figure 26, is



(a) Strategy A



(b) Strategy B



(c) Strategy C

Figure 26: Strategies to increase the number of degrees of freedom of filter pruning in residual blocks: A) heavily constrained to maintain dimensional consistency [83], B) added channel select to separate blocks [87] and C) completely dissociate the dimensionality of branches with custom select/scatter operations [112].

represented by the diversity of Ns. Since the goal of the method, presented in Chapter 6 is to leverage any kind of filter pruning, without having to pay attention to constraints required by Strategies A and B, we call it "unconstrained pruning".

5.4 Recapitulation

In this chapter, we reviewed the different types of structures in the domain of pruning, from the finest non-structured pruning to the the coarsest type possible, that is akin to practices in Neural Architecture Search. We showed that all the types bring different gains, so that they can be combined, but they also require more or less specific implementations.

We also saw that, because of numerous advantages, filter pruning is the most widespread type of "structured" pruning (in opposition to "non-structured" weight pruning). However, we saw that using filter pruning carelessly produces many problems, such as dimensional discrepancies and orphan biases.

Finally, we showed which strategies could be found in the literature to tackle these problems, which usually either involve heavily constraining pruning degrees of freedom or introducing custom layers or operations. Our own solution to this problem will be presented in Chapter 6.

Chapter 6

Leveraging Pruning on Hardware

Contents

In Chapter 5 we saw different types of pruning structures, that each bring different types of gains. In this chapter, we will dwell more into the details of how to implement neural networks on embedded hardware: which issues do we encounter when implementing structured pruning, what solutions did we developed to solve these problems and how pruning effectively reduces the cost of neural networks.

6.1 General Considerations

First, let's see what are the limits neural networks are confronted with when implemented on hardware. This will allow us to see how exactly the cost of networks can be reduced.

6.1.1 Key Metrics

Running neural networks on any hardware involves paying attention to different metrics that will define the overall performance of the system. Using the same definitions as Vivienne Sze [216], we can list different metrics and design objectives. Here are a short description of each of them (more details can be found in the original book):

- **Accuracy**: the ability of the network to solve the task. Measuring this performance depends on the test task and dataset; for example, ImageNet is a more difficult classification dataset than CIFAR-10 and semantic segmentation is more difficult than classification. Robustness is another aspect that can be taken into consideration. On this topic, we can cite the work of Hooker *et al.* [28] that highlights that simple Top-1 or Top-5 accuracy are rather naive metrics to describe the actual impact of compression on networks' performance and robustness; rather, it seems that the impact of compression highly depends on the complexity of the input and how out-of-distribution it is. This means that measuring the performance of networks and, in particular, of compressed networks is not as trivial as it may seem. However, this is out of our scope, and we focused on standard metrics such as Top-1 accuracy or mIoU (cf. Chapter 2.
- **Throughput/latency**: respectively, the amount of data processed by unit of time and the time spent to process each input. Both are crucial for real-time fluidity and responsiveness. They depend not only on the number of operation the hardware can execute per second, but also the number of processing units that can be used in parallel and how much they are all utilized. Therefore, obtaining a good throughput and latency involves not only minimizing the required number of operations, but also maximizing the utilization of processing units. Batch processing can improve throughput while degrading latency, with a possible optimum between the two depending on how well the amount of data processed maximizes the utilization.
- **Power consumption**: the quantity of energy dispersed to run the neural network. Actually, energy is consumed not only for each computation operation but also for each memory access, that are in fact way more costly than addition or multiplication operations—sometimes, by a factor of hundreds [217], [218]. Also, off-chip memory reads are even more expensive, which means that minimizing energy consumption not only involves reducing the number of operations or the quantity of data to handle, but also how memory is organized, in order to minimize how far data have to be moved. Making sure that all data fit on the chip's cache and avoiding accessing it too often is a crucial aspect on low-power embedded hardware [219].

- **Hardware cost/Flexibility/Scalability**: these metrics are more related to the domain of hardware design and are not relevant to the topic of this thesis. This is why we will not discuss them.

6.1.2 How to reduce the cost of neural networks?

We just saw which metrics are important to consider; however, when designing neural networks, we do not have the control over everything. Indeed, two aspects reduce our room for maneuver:

- Because of the cost of designing ASICs, designing hardware specifically for a neural network is very rare. In the context of this thesis, we had to deal with preexisting hardware—and this is the case of many papers in the literature. This means that the amount of available processing units, the organization of memory and the power supply are not aspects over which we have control.
- Similarly, different hardware come with their own firmware implementation or inference frameworks. These frameworks are what will define how data and memory are handled and how operations are dispatched or pipelined to maximize the hardware utilization.
- Finally, neural networks have to be stored under a certain format that can possibly represent a limited array of operations. Using operations that are non-standard means risking being poorly handled (or not handled at all) by aforementioned frameworks. For example: the ONNX format (that became a standard in the field as a non-framework-specific representation for neural networks) supports a limited set of operations, listed in "opsets". Some libraries, such as ONNX-Runtime or TensorRT, are only able to support opsets up to a certain version.

Since the topic of this thesis only involves the design of neural networks themselves, this means that we have instead to focus on which aspects of neural networks have an impact on the aforementioned metrics, for a given hardware, framework and format. Here are the elements over which we have control:

- **Input/Output data**: Even though this is not properly a part of a neural network, the size of the input is surely the first thing to consider to reduce the cost of a network. Although the resolution of input images depends on the dataset, cropping the input or reducing its resolution allows significantly reducing the number of operation and the size of intermediate representations. Reducing the resolution can degrade the performance of the network, but a certain Pareto optimum can be reached. Moreover, if parameters are only stored once on chip, input data have to be fetched off-chip, which involves the most energetically expensive type of memory read. Therefore, reducing their size can be crucial to minimize energy consumption per inference. Similarly, semantic segmentation networks are often designed to actually produce an output that is of lower resolution than the input, which does not impact the accuracy too much while reducing the overall cost. While this question is mostly out of the topic of compression, it is worth mentioning because of its efficiency and convenience.
- **Parameters**: The parameters are generally stored into tensors. Although they are involved in operations, which means that pruning parameters may involve pruning operations, their main cost is in term of memory. Since they are stored into tensors, putting weights to zero while not reducing the size of said tensors may thus not reduce their memory occupation. This is mainly the topic of leveraging unstructured pruning (Section 6.1.3) as well as the justification of the different

types of structured pruning (Chapter 5). Optimizing the storage and fetching of parameters depends mainly on the framework and hardware, but sometimes some of their properties may benefit from some design constraints; e.g.: some devices load convolution filters by batches, which means that some layers width are more optimal than others concerning hardware utilization.

- **Operations**: The number and cost of operations depend on four elements: 1) the size of the input, 2) the size of the parameters tensor, 3) the type of operator and 4) the precision of data representation. This last point belongs to the domain of quantization but the first two are directly impacted by structured pruning. Concerning the last two: additions tend to consume less energy than multiplications and operations on 32 bits float numbers cost more than those on 8bits integers—the quantization of data also directly impact the volume of data to store and move in memory, which further reduces the energy consumption. Since we mostly use standard operations—linear layers, convolutions, batch-normalization, etc—, we do not have much control over the proportions between multiplications and additions, nor do we control data representation since we do not study quantization. This means that our main degree if freedom involves the shape of the operations' operands.

- **Intermediate representations**: Each layer produces its own output, called an intermediate representation. The size of these intermediate representations may significantly exceed that of the input or output data of the network. When the input data are already significantly big, for example in semantic segmentation, the memory space required to store these intermediate representations may exceed vastly that of parameters. This means that their size is actually the main limiting factor on devices with limited available memory. Moreover, their size scales with that of input batches. Since the only way to reduce them is to remove whole filters or neurons, structured (or rather, filter) pruning may be essential in cases they are more of a limiting factor than parameters, for example.

- **Operators**: Some types of pruning may require the implementation of custom operators (for example, the shift layers mentioned in Chapter 5). We already mentioned that, depending on the used frameworks, some operations are more or less well handled. Moreover, the representation format of the network may constrain which operations can be used; e.g.: the ONNX format supports only a certain array of operators, that depends on the operation set (or opset)—not all frameworks support all the opset; for example: TensorRT 8.4.0 does not support the ONNX opset 16, which had consequences we mention in Section 6.3.2. Choosing the right operators depending on the format, the framework and the hardware can be a whole discussion for a single custom operator.

Now that we have listed what are our options to reduce the cost of a network, we will tackle the topic of both non-structured pruning and filter pruning. Concerning non-structured pruning, we will present what solutions can be found in the literature to leverage it. Since this thesis focused more on leveraging filter pruning, we will detail the solutions we came up with, as well as the results of experiments we did on embedded hardware.

6.1.3 Leveraging Non-Structured Pruning

Many papers have proposed solutions to leverage non-structured pruning or, rather, "non-structured sparsity" as the stake is to leverage the sparse tensors produced by

pruning. The multiple strategies developed in the literature can be divided into multiple categories:

6.1.3.1 Pruning as a Simplification of Weights Values

The ability of pruning to introduce many zeros in the weights tensor can allow other compression methods to better reduce its memory occupation. Indeed, one of the earliest attempts at leveraging pruning [15] simply reduces the required memory space to store weights by using Huffman coding, which can indeed take advantage of any kind of redundancy in tensors, such as that introduced by zeros. Although doing such does not reduce the number of operations, the gain in static storage can be crucial for low-power devices, more so than the number of operations. Clustering is another compression method that can benefit from pruning [220], without reducing operations either. As we already mentioned, even while not reducing the number of operations, reducing the number of memory accesses can be crucial from an energetic point of view [219].

6.1.4 Operations skipping

Runtime operations skipping, or "Zero Value Clock Gating" [203], consist in skipping an operation if one of its operands is zero or near-zero [221]–[224]. Such a method is therefore able to leverage both pruned weights and any zero in the activations—which fits well with ReLU activation functions. Even though this method efficiently reduces power consumption, it has some drawbacks: 1) it is something to implement explicitly and not to be taken as granted in any framework, hardware or implementation; 2) it does not reduce memory usage, as zeros are still to be stored, since they are detected dynamically and 3) it tends to reduce hardware utilization [203]. Despite its drawbacks, this method can be used in conjunction with others, when possible, as it has the advantage of not adding any overhead.

6.1.5 Custom Operators

Even though some papers [130], [204] use libraries such as cuSPARSE [225] to accelerate pruned neural networks, the proposed solutions may not always be satisfying. For example, cuSPARSE aims at accelerating multiplications of matrices with a sparsity rate above 95%, and acceleration may only start at 91% [204]. Such sparsity rates are rarely aimed for in most of the literature, as they tend to severely damage the accuracy of the network. The Myrtle AU accelerator is another example of accelerator that is optimized for sparsity rates above 93.75% [226].

This is why multiple papers have instead proposed custom implementations to leverage sparse convolutions or sparse GEneral Matrix Multiplication (GEMM). These implementations generaly involve at least one indexing step, whether it is to collect non-zero operands beforehand [227] or to scatter afterward the results of the outer-product of all non-zero weights and activations [228]. Even though these methods allow a better hardware utilization, the indexation operations tend to represent a non-negligible overhead [203]. According to some papers, these overheads tend to make non-structured pruning less efficient than filter pruning [82].

6.2 Solving the Problems of Filter Pruning

In Chapter 5, we mentioned that global and unconstrained filter pruning can introduce many dimensional discrepancies between layers, that must be tackled to have a func-

tioning network. Since we wanted to precisely tackle this type of pruning, without resorting to the various types of constraints usually to be found in the literature, we developed a method to automatically identify all dependencies, remove useless parameters and adapt operators so that our freely pruned networks could be leveraged on embedded devices. This method is generic, as we tested it on different types of networks, and simple to use, as it is implemented in Pytorch and freely available (`https://github.com/HugoTessier-lab/Neural-Network-Shrinking`). Networks pruned using this method could be converted into ONNX, which makes them exploitable on multiple hardware and frameworks.

6.2.1 Clear-Out: Identifying Deactivated Weights

In order to have a method that could identify all disconnected weights, kernels and filters automatically while being agnostic of the network's architecture, we decided to use the network function gradient—backpropagation being well handled by any training framework. This is why we first designed a method, that we called *Clear-Out*, that identified as eliminated the weights whose gradient were zero whatever the input.

Justification: Let's consider a network $\mathcal{N}$ and one of its parameters w

$$Let\ \mathbf{X} \in \mathcal{X},\ \frac{\delta \mathcal{N}}{\delta w}(w, \mathbf{X}) = 0 \implies \mathcal{N}(w, \mathbf{X}) = \mathcal{N}(0, \mathbf{X})$$

In such a case, $\mathcal{N}(w, \mathbf{X})$ is constant considering w, which means that its value does not change the output depending on $\mathbf{X}$.

If the aforementioned condition is verified for any $\mathbf{X}$, then w can be safely removed from the network. Actually, since networks can have multiple outputs, this condition must be verified for all outputs at the same time.

Problem: Clear-Out has three issues:

1. The condition that the gradient of w must be zero for any input, which cannot be analytically deduced, implies that the input space must be sampled. Drawing samples from the training dataset could allow approximating relevantly the domain of definition of the network classes. However our experiments, meant to help figuring out exactly how many samples were needed exactly, turned out to raise many questions, many of which are out of the scope initially intended for Clear-Out.
2. The definition of Clear-Out means that are considered deactivated not only weights that are disconnected because of pruning, but also those that are made inactive because of the degradation of the network's function. Our experiments showed that this proportion of isolated disconnected weights can be very significant, and leveraging their pruning is of the domain of non-structured pruning (Section 6.1.3), which we did not tackle.
3. Clear-Out cannot solve the problem of orphan biases, since the weights downstream are involved in the network's function despite their arguable uselessness.

Because of these issues, we decided that this definition of Clear-Out did not fit our needs, that are focused solely on identifying weights that are irremediably disconnected because of pruning and dimensional dependencies between layers. This is why we propose a variant called Dimensional Clear-Out, that is specifically designed for filter pruning.

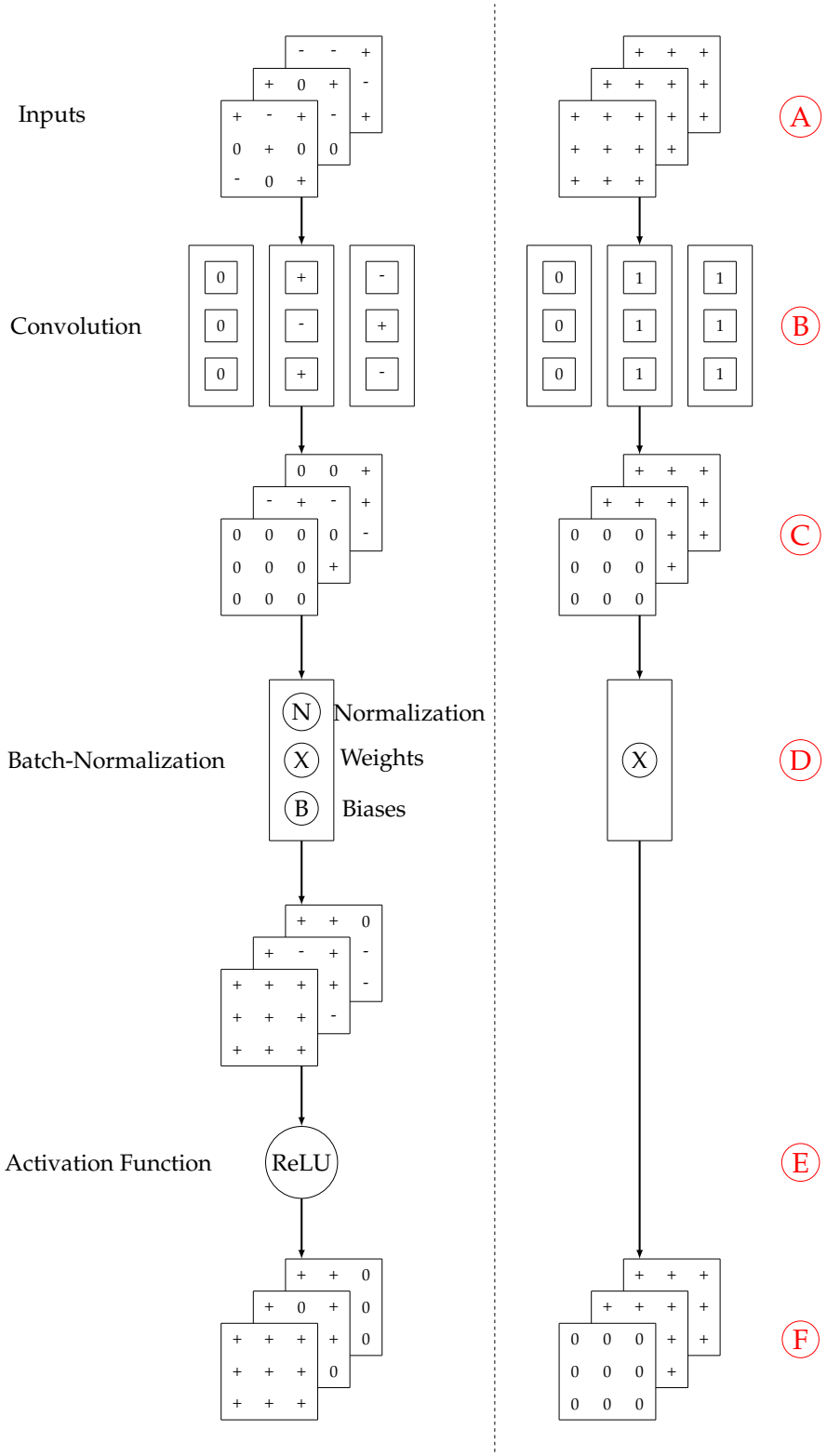


Figure 27: On the left: a regular convolution-batchnorm-ReLU block. On the right: the modified copy to suit DCO. Explanation of the modifications: A) the input can be generated, as long as it contains no zero values and all values are of the same sign; B) the weights of layers are replaced with their pruning mask; C) therefore the output of aforementioned layer contains only zeros in the case of a pruned filter; D) normalization and biases are removed; E) non-linear functions are removed; F) the output of the whole block contains only zeros for each pruned filter.

6.2.2 Dimensional Clear-Out

Since our concern only involves the architectural aspects of the network and not its actual function, we figured out that a way to solve the aforementioned problems was actually not to operate on the network itself, but on a modified copy that had some specific properties.

The modifications of the network, illustrated in Figure 27, are meant to transform the network into a purely linear network without biases—which means removing all non-linear functions and biases. Removing biases allows circumventing the orphan biases problem and removing non-linear functions prevents introducing unwanted zero values. Replacing the value of weights with the value of their pruning mask means that zero values can only be introduced by pruned weights. Removing normalization makes sure that these zero values stay zero.

Since the network is now purely linear, any weight that is involved in the network's function can only have a non-zero gradient as long as the network's input is itself non-zero. This has two consequences: 1) one single input suffices to identify reliably all weights that are possibly involved in the network's function and 2) weights can only have a zero gradient if they are structurally disconnected from the rest of the network because of structured pruning.

To conclude: the trick of using a modified network to perform the Clear-Out computation allows solving all three problems of Clear-Out, which makes this variant—Dimensional Clear-Out (DCO)— suitable for our needs while being also quick to compute and reliable.

6.2.3 The Problem of Addition Operations

DCO itself allows identifying all disconnections following structured pruning and thus allows guaranteeing the consistency of input/output dimensions of layers. However, one last problem remains: additions at the end of residual blocks still cannot handle tensors of different sizes. Even if, after global unconstrained filter pruning, the two tensors happen to be of the same dimensions, there is no guarantee that the same filters were pruned on each side, which means that summing the two together would sum channels that are not meant to go together.

To solve this problem, we defined a more generic operator, that we called the *indexation-addition* operation (that we could as well call a crossbar operation).

Definition of the Indexation-Addition Operation Let $\mathbf{a}$ and $\mathbf{b}$ be the tensors to sum, that contain respectively n^a and n^b channels. Let $\mathbf{i}^a$ and $\mathbf{i}^b$ be two lists of indices and $\mathbf{c}$ the output tensor, that contains n^c channels. The indexation-addition operation is defined as such:

$$\forall k \in [\![1; n^c]\!],\ \mathbf{c}_k = \begin{cases} \mathbf{a}_{\mathbf{i}^a_k}, & \text{if } \mathbf{i}^a_k \in [\![1; n^a]\!] \\ \varnothing, & \text{otherwise} \end{cases} + \begin{cases} \mathbf{b}_{\mathbf{i}^b_k}, & \text{if } \mathbf{i}^b_k \in [\![1; n^b]\!] \\ \varnothing, & \text{otherwise} \end{cases}$$

If $n^a = n^b$, $\mathbf{i}^a = [1, 2, \ldots, n^a]$ and $\mathbf{i}^b = \left[1, 2, \ldots, n^b\right]$, this *indexation-addition* operation is purely equivalent to an element-wise addition.

This operation allows configuring each addition to sum together the right channels among the two tensors to handle. This operator therefore allows leveraging any distribution of unconstrained filter pruning, as enforcing the same sparsity for the two branches of a residual block is not a necessity anymore.

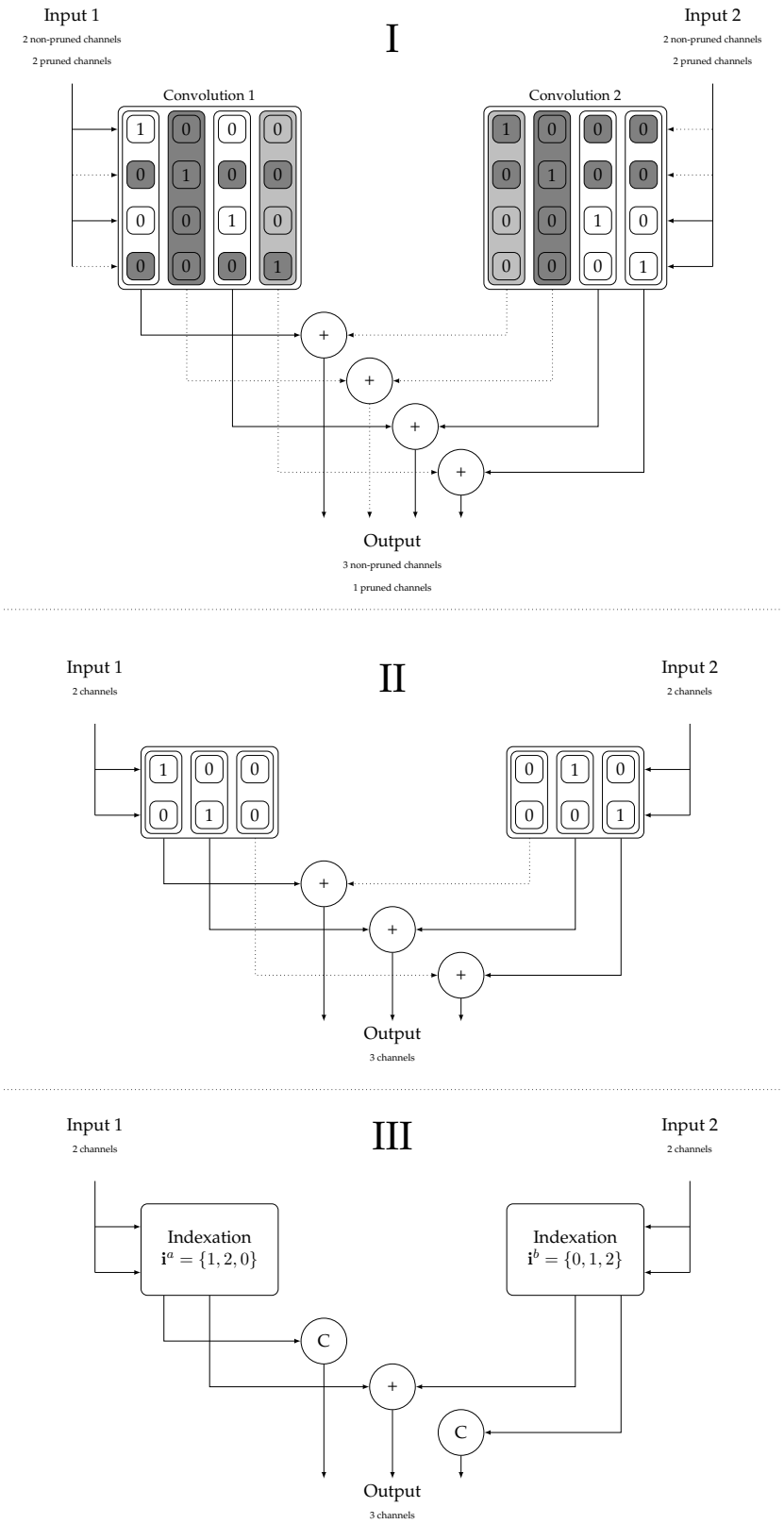


Figure 28: Creation of the indexation-addition operation. I) Identity convolutions are applied to each input. Grey-out parameters can be removed; the light-grey filters would be removed by DCO, but we keep them to for the next step (because the same filter in the other layer is not pruned). Dotted arrows represent the empty feature maps. II) The shrunken convolutions now are able to insert each of the two inputs into larger tensors that can be properly summed together. III) The indices $\mathbf{i}^a$ and $\mathbf{i}^b$ can be directly deduced from the reduced convolutions from step II.

However, this requires finding the right $\mathbf{i}^a$ and $\mathbf{i}^b$, which can be done automatically by another trick that exploits the properties of DCO.

Generating Automatically Indices The trick we used consisted in inserting two layers, one for each branch right before the addition. These layers behave like an identity—actually they are 1×1 convolutions whose weights match an identity matrix. While not modifying the tensor they are applied to, these layers are very useful, since, once inactive channels are identified using DCO, their weight matrices give exactly the indices $\mathbf{i}^a$ and $\mathbf{i}^b$ necessary to configure properly the indexation-addition operations. Figure 28 illustrates how the pruned identity layers allow generating the lists of indices.

To conclude: thanks to the combination of the DCO, of the indexation-addition operation and of the aforementioned trick to generate it automatically, we are able to make any distribution of filter pruning functional by enforcing the consistency of input/output dimensions of layers and by adapting additions so that they do not require a specific distribution of channel sparsity to work properly. Therefore, our work finally provides a simple way of making the study of the efficiency of such types of pruning on hardware possible.

This is why the next sections will precisely tackle the topic of measuring the gain in term of energy consumption when doing unconstrained filter pruning. We will see under which experimental conditions we ran our experiments, what were the problems to solve and how this study raised a whole new array of questions that call for further investigation in the future.

6.3 Implementation on an Embedded GPU

Dimensional Clear-Out [3] was initially developed to allow measuring the decrease in energy consumption when pruning neural networks [2]. This is why we will now focus on said measurements. First, we have to describe under which experimental conditions such measurements were conducted.

6.3.1 Experimental Conditions

Environment: We did our measurement on a NVIDIA Jetson AGX Xavier embedded GPU, configured in the "30W All" mode. We installed JetPack SDK 5.0, that goes along with CUDA 11.4.14, cuDNN 8.3.2 and TensorRT 8.4.0 EA. We also installed ONNX Runtime 1.12.0, compatible with GPU computing and able to use the TensorRT execution provider. This allows using the optimization and inference engine of TensorRT while having a better control of how to do the benchmarking. Energy consumption of the GPU was measured using the tegrastat utility every second while running a large number of inferences for each pruned network (tegrastat uses a chip on the board to measure the power consumption of the whole board or of individual componants, such as the CPU, the GPU or the memory—we only measure the consumption of the GPU). All networks, designed in Pytorch, are converted into the ONNX format before being fed to the TensorRT execution provider or *trtexec* utility.

Networks: All networks were trained in a standard way, as described in Section 2.1.5. All aspects proper to pruning are described in the next paragraph. On ImageNet we used ResNet-50 and on Cityscapes we used HRNet-18, 32 and 48.

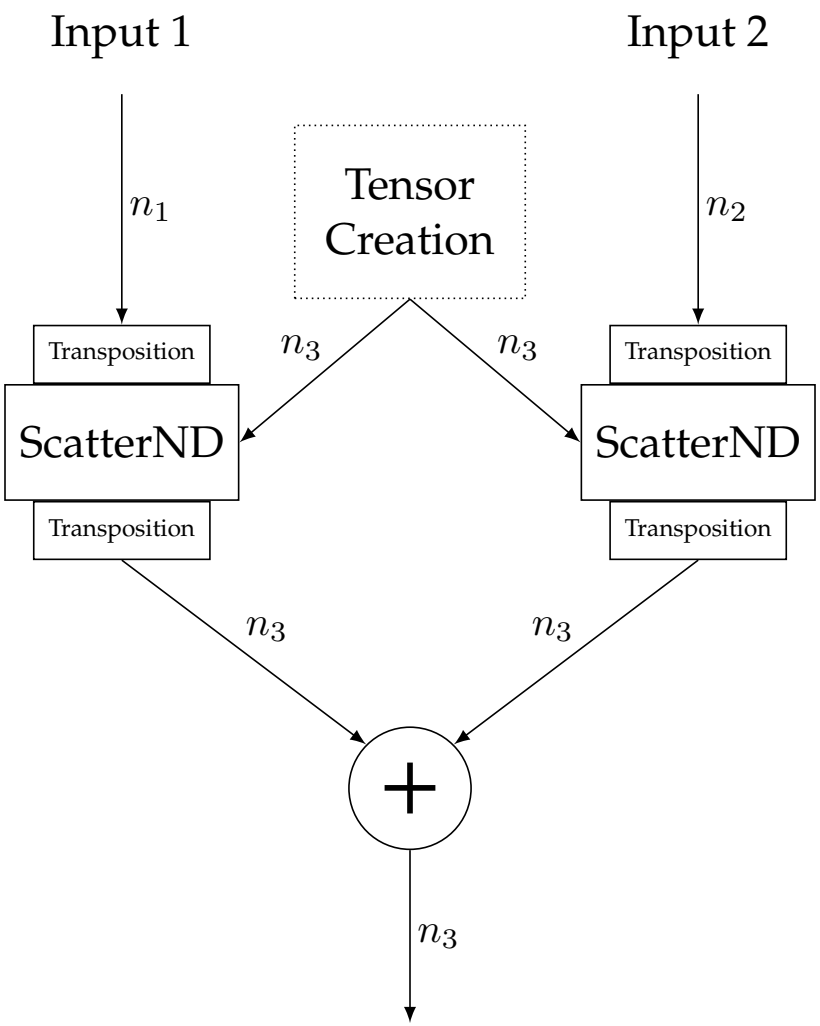


Figure 29: Schematic representation of our ONNX implementation of the indexation-addition operation. The tensor creation node actually contains many different operations, but its overall cost is negligible. The transposition nodes adapt the axis order in the input to fit the *ScatterND* operation, then restore the original order. The *ScatterND* operations inject the content of Input 1 and Input 2, that contain respectively n_1 and n_2 channels, into the dynamically created empty tensor of size n_3, to create two tensors of size n_3 that are then summed together.

Pruning Methods: In our experiments, we used two different pruning methods:

- **Slimming** [87]: We prune filters on the basis of the magnitude of their weight in the corresponding batch-normalization layer. Networks are pruned in three steps, with a linearly increasing pruning rate and fine-tuning (20 epochs for HRNet, 10 epochs for ResNet). For some experiments, after the last pruning step, the network is retrained using LR-Rewinding [92]. Following the original method, during training, batchnorm weights are penalized by a smooth-$\mathcal{L}_1$ norm ($\lambda = 10^{-6}$ for HRNet and $\lambda = 10^{-5}$ for ResNet).
- **SWD** [1] (cf. Section 4.3): We only used SWD with HRNet. We chose $a_{min} = 10^{-1}$ and $a_{max} = 10^{10}$. LR-Rewinding [92] is applied after pruning.

Depending on the experiments, the pruning target was either set using DCO for an exact estimation of the number of pruned parameters or using a naive estimation based on the proportion of pruned filters, whatever their actual size. Also, whatever the method, pruning is always performed globally and unconstrained. Therefore, we always apply DCO at the end and shrink networks while transforming, when needed, additions into indexation-addition operations.

6.3.2 Implementation of the indexation-addition operator

As described previously, we used the ONNX format[1] to store neural networks before feeding them to TensorRT. This format has no trouble representing all standard operations used in neural networks—convolution layers, ReLU, upsampling, etc. However, indexation-addition operations are non-standard. Therefore, we needed to design an ONNX implementation of this operator.

The indexation-addition mixes two aspects: 1) inserting the data of the two tensors to sum in another tensor that does not have the same dimension and 2) effectively summing them. We chose to perform the insertion using the *ScatterND* operation, that allows scattering the content of a given tensor into another tensor. While the *Scatter* operation is element-wise and therefore requires storing an index for every element of the tensor to scatter (which makes the operator resolution-specific and very costly in memory), *ScatterND* operates at the scale of a given axis, which allows only indexing the channels themselves, which is much lighter and more generic. The only constraint is that the behavior we wanted is only allowed for the first axis of a tensor, so that we have to transpose the tensor before and after the *ScatterND* operation (the cost of these transpositions is negligible).

In the 16th opset of ONNX, *ScatterND* has an argument that allows not to insert the data but instead to sum its values to those in the destination tensor. This would have allowed a much more optimized implementation by performing scattering and addition in one step, but unfortunately, TensorRT does not support ONNX operation sets beyond the 15th opset. This is why we instead had to insert each of the two tensors into a larger tensor, so that the two new tensors can be of the same dimensions and summed together. This larger tensor has to be instantiated dynamically, which involves many different operations; fortunately this tensor creation turned out to be of negligible cost, compared to that of *ScatterND* operations. Similarly to related works in the field [112], [201], [214], [215], this implementation involves, at least temporarily, inserting empty feature maps.

[1] `onnx.ai/`

This implementation, summed up in Figure 29 is the one we used in the following experiments [2], [3], that we will now present.

6.4 Experiments

The following experiences can be found in two papers we published during this thesis [2], [3]. They will be presented thematically.

6.4.1 DCO as a Tool for a More Reliable Study of Pruning

The existence of dependencies between layers and the necessity of acknowledging them for a reliable measurement of the compression rate is something that is often left ambiguous in many papers. Blalock *et al.* [27] even cited this lack of consistency, in the metrics reported to characterize the compression of a pruned network, among a large proportion of papers in the field.

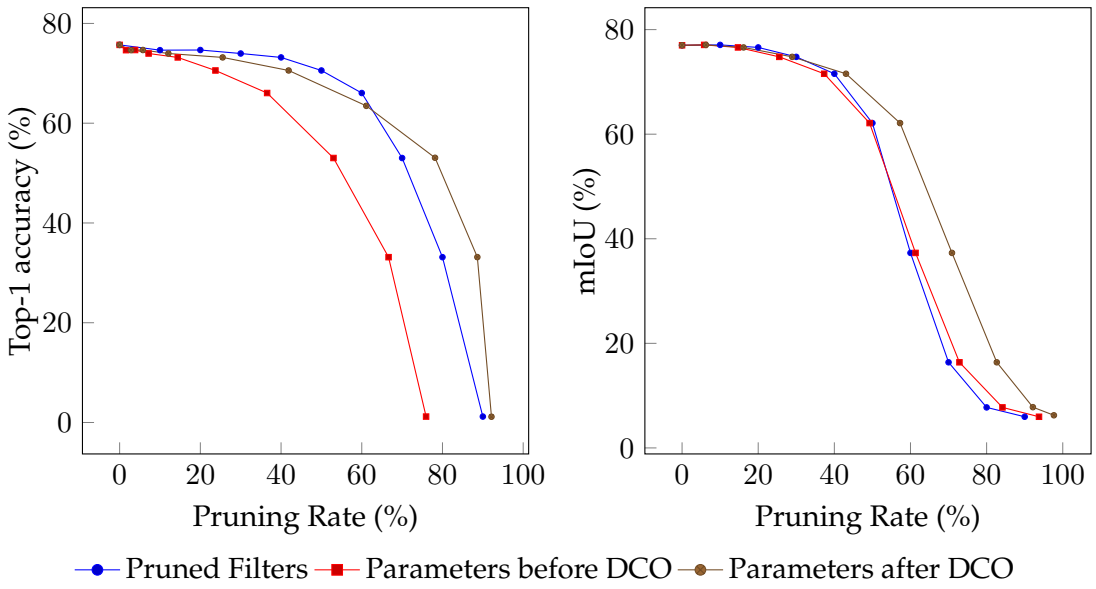


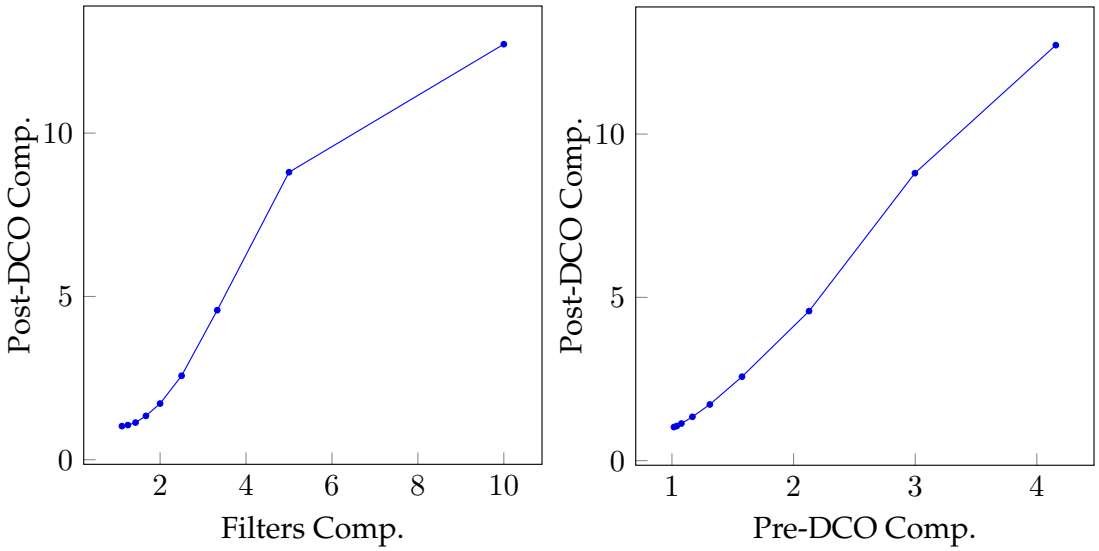
Figure 30: Trade-off between the pruning rate and the accuracy of ResNet-50 on ImageNet (left) or the mIoU of HRNet-48 on Cityscapes (right), depending on three different ways of estimating the pruning rate: 1) the proportion of pruned filters, 2) the remaining parameters after pruning the filters and 3) the remaining parameters once the pruned network is applied DCO.

This is why, when presenting our work on DCO [3], we illustrated its purpose with a comparison of the effect of different ways of calculating the pruning rate on the apparent trade-off between accuracy and pruning rate. We compare three different calculation methods:

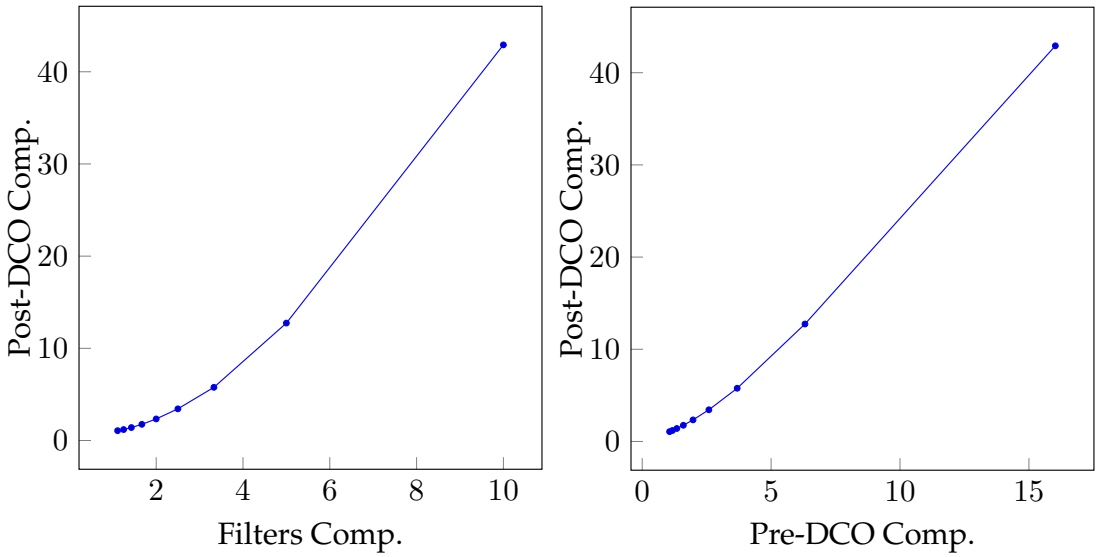
- A very naive method, that only counts the proportion of pruned filters.
- A more relevant method, that counts the remaining parameters after pruning, but without taking into account any dependencies between networks.
- The actual proportion of remaining parameters after DCO.

Figure 30 shows precisely this comparison. These experiments did not use LR-rewinding [92] and networks post-DCO were not fine-tuned, so some of them encountered a small

drop in accuracy because of the perturbation of the distribution of data in intermediate representations, due to the removal of orphan biases. Also, the pruning target, for each pruning iteration, was each time defined according to the proportion of pruned filters.



(a) ResNet-50 on ImageNet

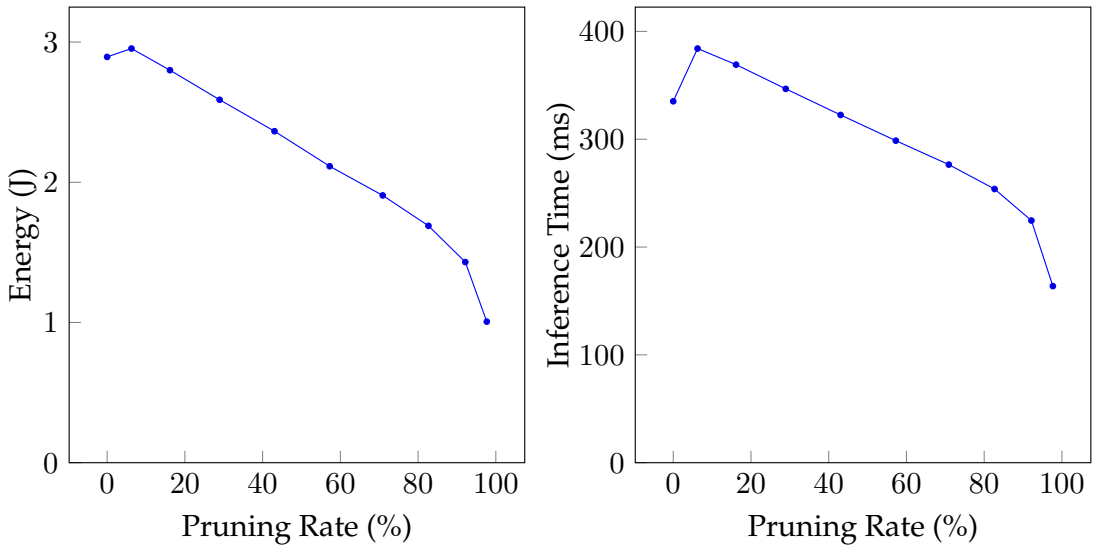


(b) HRNet-48 on Cityscapes

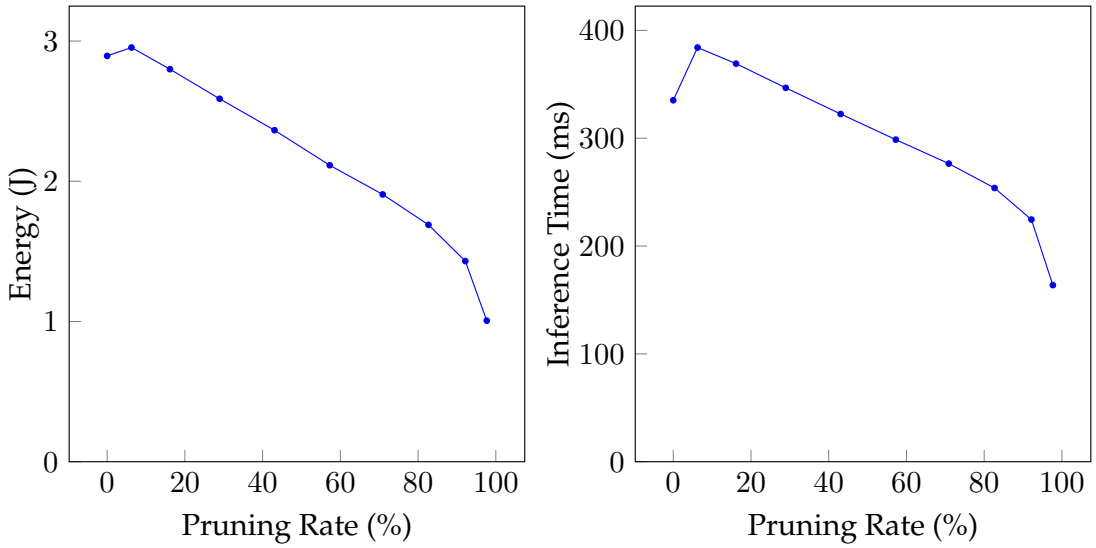
Figure 31: Comparison of the evaluated compression rate between either the proportion of pruned filters and post-DCO remaining parameters, or pre-DCO and post-DCO parameters.

This figure allows to make some interesting observations:

- For low pruning rate, the proportion of filters overestimates the actual count of parameters; but for high pruning rates it underestimates it.
- The relationship between the proportion of filters and the pre-DCO parameters count are completely different between ResNet and HRNet.
- The gap between before and after DCO is significant, but fairly less for HRNet than for ResNet.



(a) ResNet-50 on ImageNet



(b) HRNet-48 on Cityscapes

Figure 32: Energy consumption and inference time, depending on the post-DCO pruning rate. For ResNet-50, results are averaged over 10k inferences after 1k runs of warm-up, with inputs of size $(1 \times 3 \times 224 \times 224)$. For HRNet-48, results are averaged over 1k inferences after 100 runs of warm-up, with inputs of size $(1 \times 3 \times 512 \times 1024)$

All these observations highlight the same fact: all filters in networks do not have the same number of parameters (which is obvious) and pruning using the global criterion we used seems to favor pruning first filters that contain less parameters (which is more surprising). Indeed, at the beginning, pruning the first 10% of filters removes less than 10% of parameters, which means that smallest-than-average filters were removed. However, the tendency is reversed after a certain point.

Since for high pruning rates, small differences in percentage are very significant—95% pruning means reducing by 20 while 99% pruning means reducing by 100, for only a difference of 4%—we decided to show the comparison of compression rates too in Figure 31. We define the compression rate as $\frac{100\%}{100\% - pruning\_rate\%}$ and compare either between the pre-DCO and post-DCO parameters count or between the proportion of filters and the post-DCO parameters count.

This figure highlights the huge gap in compression rate that is missed when not using DCO. Also, it looks like the relationship between the actual compression rate and the naive ones is rather complex and heavily depends on the architecture. This means that one cannot simply deduce the actual compression rate from a naive one: bothering to do the real measurement, using DCO for example, is a step that cannot be skipped.

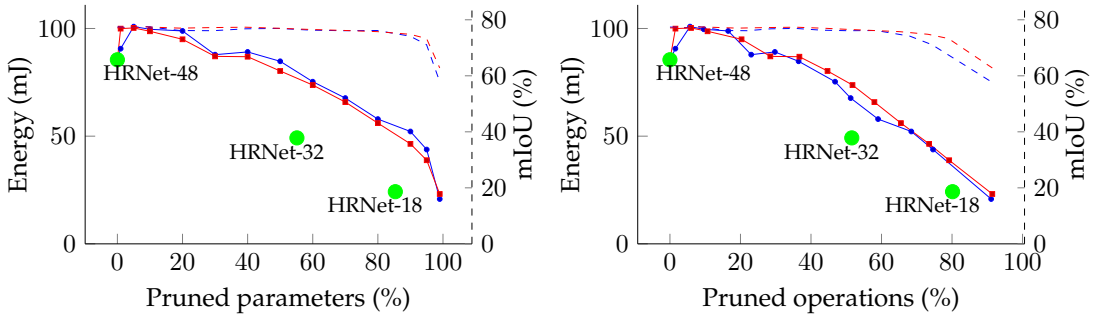
After having illustrated the methodological interest of DCO, we illustrated how DCO allowed producing networks that could be leveraged, despite the lack of precaution when distributing filter sparsity. Figure 32 shows briefly some energy consumption and latency measurement on ResNet-50 and HRNet-48. The decrease in cost proves that we effectively reduce the cost of networks in a way that impacts both energy and latency—that both seem very correlated for either ResNet-50 or HRNet-48. However, the initial increase in cost for the lowest pruning rates shows that something is costly in our implementation—which we will observe more in details in Section 6.4.2.

6.4.2 Energy Consumption Analysis

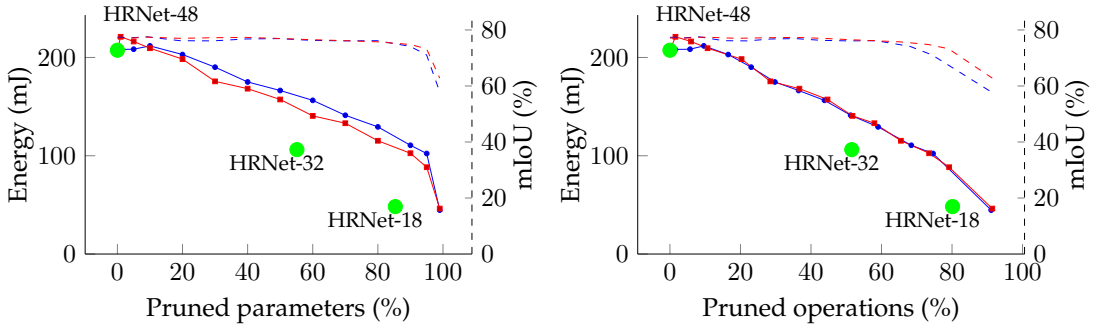
Our second contribution [2], using DCO, focused more specifically on measuring the energy consumption of pruned HRNet semantic segmentation networks trained on Cityscapes. In this work, we not only showed how we could reduce energy consumption even with unconstrained filter pruning, but we also studied how efficient was pruning compared to smaller baseline networks and highlighted the necessity of an efficient implementation of custom operations such as the indexation-addition operation.

Figure 33 shows the reduction in energy consumption depending on the count of remaining parameters or operations after DCO. Is also indicated the corresponding mIoU, to provide a reference. Each time we compare the pruned HRNet-48 with non-pruned HRNet-18, 32 and 48 and we made the experiments with the Slimming [87] and SWD [1] methods, as well as LR-Rewinding [92]. This figure already allows to draw multiple observations:

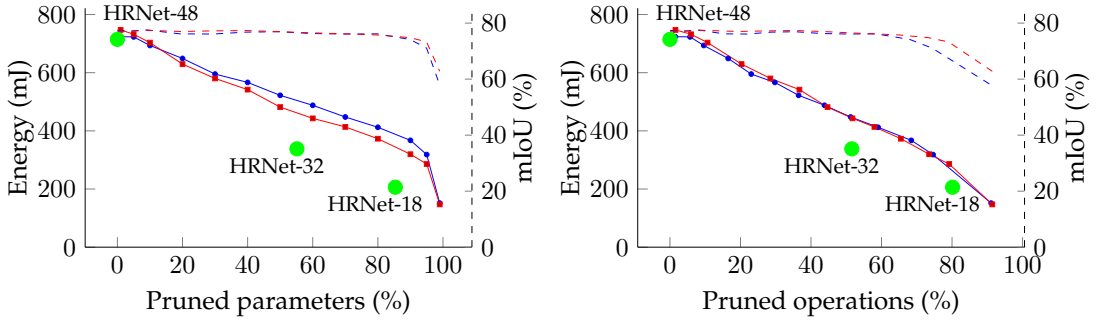
- The shape of the energy consumption curve depends on the size of the input: the bigger it is, the smaller is the initial overhead that makes the first pruning rates less efficient than the non-pruned initial network. As we will see, a large part of this overhead is due to the indexation-addition operations and, more precisely, the *ScatterND* operations. It seems that, for larger inputs, the cost of these operations increases less than that of convolutions, which makes the overhead appear smaller in comparison.
- When comparing to the rate of pruned parameters, results appear much more



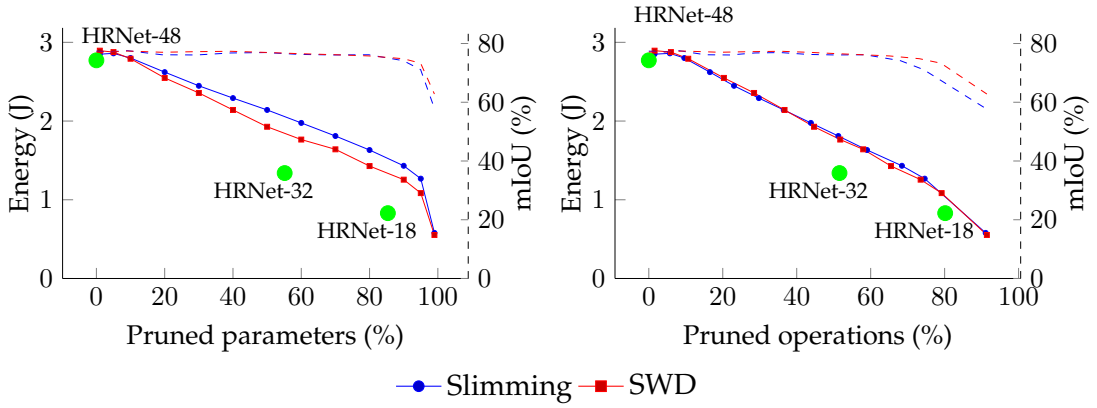
(a) Input size: $(1 \times 3 \times 64 \times 128)$



(b) Input size: $(1 \times 3 \times 128 \times 512)$



(c) Input size: $(1 \times 3 \times 256 \times 512)$



(d) Input size: $(1 \times 3 \times 512 \times 1024)$

Figure 33: Energy consumption, averaged over 1000 inferences (after 100 runs of warmup), of pruned HRNet-48 and non-pruned HRNet-32 and HRNet-18, trained on Cityscapes. Green dots provide non-pruned references. Dashed lines provide instead the mIoU of corresponding pruned HRNet-48. For plain or dashed lines, the x-axis is the same and corresponds to the pruning in term of post-DCO parameters or operations count. The position on the x axis of HRNet-32 and HRNet-18 reflects their count of parameters or operations compared to HRNet-48.

separated between the two pruning methods than when comparing to operations. Actually, all pruned networks seem to belong to the same curve, whatever the method. Networks were pruned to have a given number of parameters (using DCO for an exact count), with the same pruning rates chosen for both methods—this is why all pairs of pruning points are vertically aligned when comparing to parameters. However, this vertical alignment is not here anymore when looking at operations. It then appears that energy consumption is clearly correlated with the number of operations and that the difference in energy consumption for a same pruning rate between the two methods comes from the apparent tendency of said methods to produce networks with a varying number of operations for a same count of parameters. Figures 30 and 31 already highlighted that, since all filters are not of the same size in networks, a same pruning rate in parameters may not always impact the number of operations the same way, depending on the distribution of sparsity.

- Whether it is when comparing to parameters or operations, non-pruned smaller networks HRNet-32 and 18 appear to be more energetically efficient than pruned networks for a same number of parameters or operations. We will see that most of that gap can be explained by the overhead due to *ScatterND* operations. The fact that this gap is smaller for operations than for parameters hints that pruned networks are denser in parameters, for a same number of operations, than smaller non-pruned networks. This comparison is made even easier by the fact that these architectures only differ by their width and not their depth, which means that their architecture approximately equates an HRNet-48 that would have been uniformly pruned at the same rate for each layer.

To sum up: these results already hint what will be the three main observations of the following results: 1) in our case, the interest of pruning, compared to smaller baselines, is not guaranteed, 2) the cost of *ScatterND* operations is not negligible at all despite their apparent simplicity (especially when compared to convolution operations) and 3) sparsity is not uniformly distributed in pruned networks, and this distribution does not advantage pruned networks.

Figure 34 proposes a different perspective on these results, by comparing the mIoU to energy, operations or parameters. It also displays some pruned HRNet-18s. This figure brings again some new observations:

- Whether pruned networks appear more efficient or not than non-pruned smaller networks seems to highly depend on which trade-off ones look. When considering parameters, it would seem like pruned networks achieve a better accuracy than baselines—or, at least, they appear to be roughly located on the same Pareto optimum. However, this is absolutely not the case when looking at operations, for which non-pruned baselines can be far better. This confirms that, as mentioned previously, pruning here seems to prioritize pruning weights with fewer operations, so that for a same number of parameters, pruned networks are more costly in parameters and, ultimately, in energy. This is rather surprising, because the pruning criterion we used—the magnitude of batchnorm weights—is not explicitly designed to have such a behavior. In our preparatory experiments, we already observed that, when applied globally, filter pruning criteria tended to be unbalanced but we chose this one specifically because it tended to be the most balanced one. Such a behavior is especially problematic, because not only is it completely spontaneous and unwanted, but it arguably harms the interest of pruning. However, this allows not to conclude on the overall interest of pruning, because we

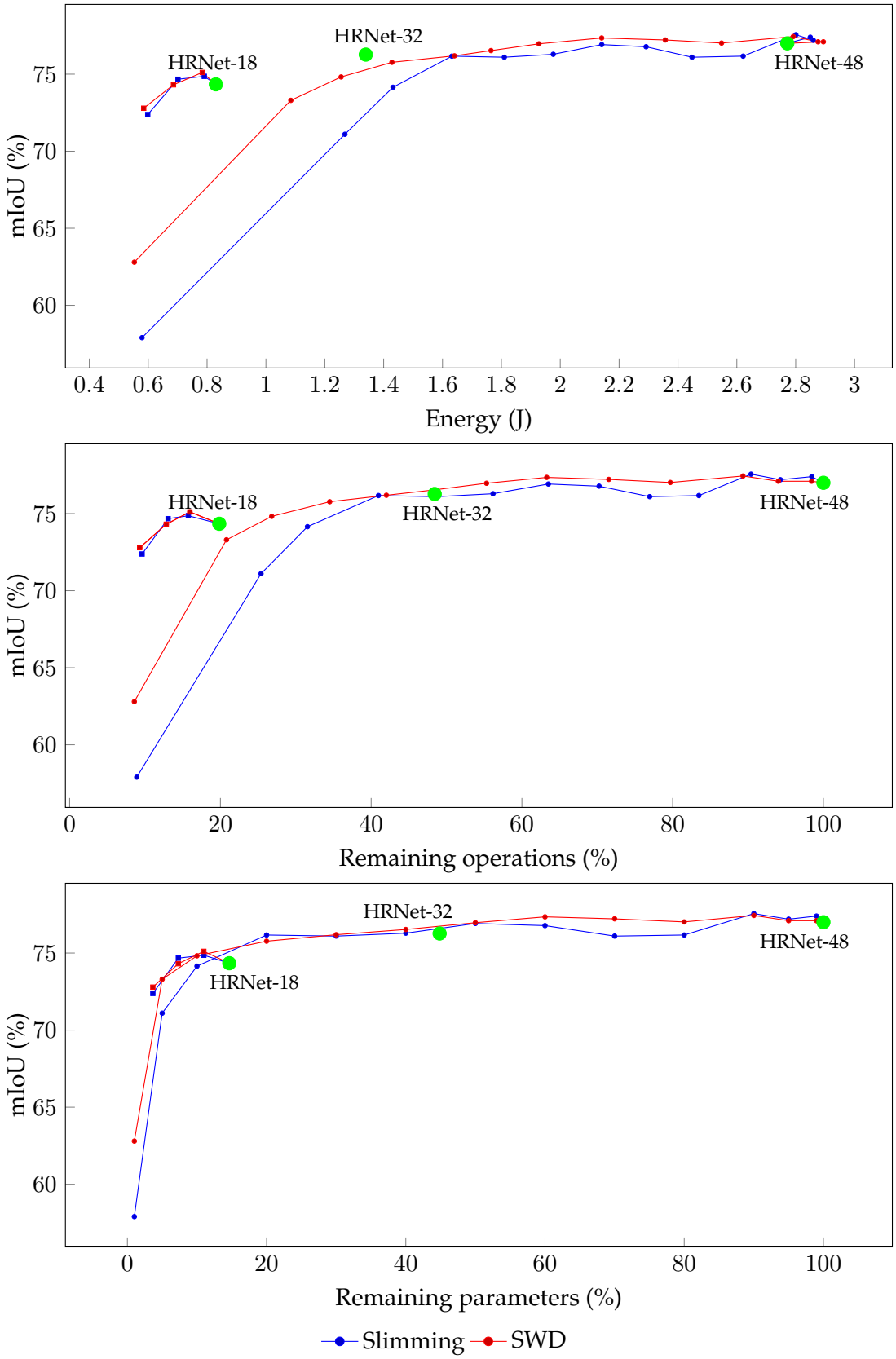


Figure 34: mIoU accuracy of pruned HRNet-48 (curves with filled circles) and HRNet-18 (curve with squares) and non-pruned HRNet-18, 32 and 48 (green dots), compared to energy consumption, number of operations and number of parameters, with two different pruning methods. The input size is $(1 \times 3 \times 512 \times 1024)$. For the number of parameters or operations, the maximum bound and reference is defined by non-pruned HRNet-48.

still do not know if, in the case a criterion that produced networks that are poor in operations existed, it would make pruning better than small non-pruned networks. Therefore, this whole topic is still open to further investigations and raises some fundamental questions on pruning criteria.

- The inefficiency of the implementation of our pruned networks seems to exaggerate the relative efficiency of non-pruned networks for energy consumption.

- Pruned HRNet-18s, at least in term of operations and energy, seem to be a lot more profitable than pruned HRNet-48s. This evokes two conflicting ideas: 1) it is more sound to compress a network that is already closer to the final cost target and 2) between the compressed version of a larger and better network and a smaller baseline, the originally better network should perform the best. As we mentioned before, pruning is likely sabotaged by an imbalance in its criterion—after all, when considering the trade-off between mIoU and parameters, all pruned networks look to be on the same Pareto optimum, whether they were originally HRNet-48 or HRNet-18. Also, the gain in mIoU for the first pruned HRNet-18 points hints that either the pruning process or LR-Rewinding brings some overall improvements to the accuracy; it is possible that, if non-pruned networks were as well trained as pruned ones, they would perfectly be on the same Pareto optimum as pruned networks, when comparing to the number of parameters. Therefore, it is not yet possible to conclude that the second idea is wrong or not. Finally, considering the problem of imbalance and excessive density of operations, it is normal that pruning HRNet-18 leads to more optimized networks.

- The fact that SWD [1] both produces networks with a better accuracy and, surprisingly, fewer parameters and operations (as can be seen in Figure 33), further amplifies the gap with Slimming [87]. If the advantage of SWD was not that obvious in Figure 33, it appears here clearly as more profitable in a very significant way. Surprisingly, while both use the same criterion, the fact that SWD achieves both a better mIoU and a lower density in operations seems rather counter-intuitive. Indeed, the fact that the criterion targets weights that are involved in fewer operations could be interpreted as such weights being less important; but this behavior of SWD hints that the relationship may actually be much more complex. Also: when comparing mIoU to parameters, SWD and Slimming look rather close; but when considering operations and energy, the difference is very significant. This hints that in our case, the efficiency of SWD is mostly due to this ability to produce networks that are poorer in operations. The fact that both methods used the exact same type of pruning structure and pruning criterion shows that not only the pruning method has a very significant impact on which weights are pruned and on the distribution of sparsity, but also that this impact on the distribution may actually be the main discriminating factor between methods—even more than the actual accuracy for a given target in parameters.

In conclusion, these experiments showed, besides the necessity of finding a better implementation for custom operators of pruned networks, that the pruning structure, the pruning criterion and the pruning method have very complex interactions that each impact the final distribution of sparsity and, thus, the efficiency of the produced architecture. These dependencies are both very important for the relevance of the field and complex enough to create many topics of study to investigate in the future. Considering the implementation of custom operators, Section 6.4.3 will detail further what are the problems that make it less efficient.

6.4.3 *ScatterND* and TensorRT: a pair that does not go together well

	Param.	Op.	Lat.	Lat.*	C. Lat.	Lat.	Lat.*	C. Lat.
			$(1 \times 3 \times 64 \times 128)$			$(1 \times 3 \times 512 \times 1024)$		
SWD	40.0%	48.2%	25.1ms	20.3ms	19.9ms	245.2ms	196.0ms	188.0ms
Slim.	30.0%	48.8%	22.5ms	17.9ms	17.5ms	248.1ms	199.8ms	191.9ms
HRN32	44.9%	48.4%	15.1ms	15.1ms	14.4ms	158.3ms	158.3ms	147.7ms

Table 4: Count of remaining parameters (Param.) and operations (Op.), overall latency (Lat.), latency with the exclusion of *ScatterND* operations (Lat.*) and latency of convolution layers only (C. Lat.), for two different sizes of inputs, of HRNet-48 pruned using SWD (SWD) or Slimming (Slim.) and a non-pruned HRNet-32 (HRN32). For HRNet-32, the percentage of parameters and operations gives a relative comparison to those of non-pruned HRNet-48. Measurements and profiling were performed using the *trtexec* utility. These networks were chosen to have the closest count of operation we could find.

When we saw how big of an overhead our custom indexation-addition operation seemed to bring, we made some measurements to quantify the cost of its operators. As described in Section 6.3.2, the indexation-addition operation involves mainly four types of operations:

- A tensor creation, that our preliminary tests showed to have a negligible cost.
- Transpositions, that are negligible too.
- An addition, that already exists in non-pruned network and that should cost less, since they operate on smaller tensors.
- The *ScatterND* operator.

Therefore, the *ScatterND* is the main cause of the overhead. To study it, we used the *trtexec* tool, from TensorRT, that does the optimization and some inference benchmark on a given network. Observing the verbose execution trace of *trtexec* allowed some interesting observations.

The first observation is summed up in Table 4, that isolates the cost of *ScatterND* operators (alias "Foreign Nodes" as we will explain after) and of convolution layers. This table allows to see that:

- Non-*ScatterND* and non-convolution operators have a marginal cost.
- *ScatterND* operations make up for a large portion of the gap in inference time between pruned and non-pruned networks.
- Even when considering only convolutions, despite the very similar number of operations and the reduced number of parameters, pruned networks have a significantly higher latency.

These results have to be put in perspective with the observation we made concerning TensorRT itself. Indeed, this framework has a certain way of optimizing networks:

- It splits the networks into nodes, that can encompass multiple layers (for example, performing convolutions, ReLU and additions together; BatchNorm layers were already fused with convolutions during the conversion to ONNX).

- Each node is then optimized separately and empirically by testing multiple strategies (that imply mostly lossless compression and operation rearrangement methods) and choosing the one that gives the best inference time.
- Most standard types of layers are then run using the cuDNN [17] library, that is specifically designed for neural networks inference on GPU.

However, we observed one things: *ScatterND* operations are not treated the same way as the others. They are instead turned into "Foreign Nodes" that are not handled by cuDNN but by Myelin, that is a part of cuBLAS. Also, dwelling into the documentation of TensorRT revealed that *Scatter*-like operations are only supported since TensorRT 8.0. All of this hints that TensorRT does not handle this operator well at all, which explains why, despite its theoretical simplicity, *ScatterND* operations cost this much. Yet, the increased inference time of convolution layers alone is still to explain.

Besides the purely technical aspect of this discussion, one thing that is interesting to notice is that, despite what may seem theoretically optimal, what makes an implementation efficient or not ultimately depends on the ability of the framework (or hardware in other cases) to support and handle it. A different implementation of the indexation-addition operation could have, instead, involved concatenating the two tensors together and then summing them using a 1×1 convolution. Theoretically, such an implementation would be far less optimal, because of the number of useless operations, but since it would involve only operations that are very standard in deep neural networks, TensorRT would have probably handled it much better.

6.5 Recapitulation

In this chapter, we saw that the implementation of deep neural networks on embedded hardware requires to bear in mind different types of metrics, such as accuracy, latency or energy. Since the scope of this thesis does not encompass designing dedicated hardware or inference frameworks, we instead focus on what elements of the networks themselves can be adapted to fit the capacity of a given hardware. We saw that pruning could allow reducing the number of parameters and operations and the size of intermediate representations, which helps reducing the cost of a network.

We saw that several methods to leverage non-structured pruning have been proposed in the literature, but our work does not focus on it. Instead we tackled the topic of filter pruning and how to leverage it in the case where pruning introduces discrepancies in the dimensions of layers. We proposed a method, Dimensional Clear-Out (DCO), to identify and solve all problems that can be encountered when performing filter pruning.

We then presented some experiments. First, we showed how DCO can be used for a more reliable study of how pruning can achieve a Pareto optimum between the number of parameters and accuracy. After that, we finally showed how pruning could reduce the cost of networks. However, these experiments, while raising doubts about the efficiency of pruning, in comparison with smaller non-pruned networks, also raises fundamental questions about the behavior of pruning methods and criterion regarding the distribution of sparsity. We also highlighted that leveraging filter pruning also requires an optimized implementation of some custom operators, which tends to be confronted with the ability of frameworks to handle them.

To conclude: the various problems we encountered and solved in order to do these final measurements, as well as the questions raised by these experiments, show that con-

fronting the technical aspects of pruning is actually a crucial part of identifying what are the most important fundamental questions to answer to get efficient pruned networks. The problem of inter-dependencies between layers is not something that leaps to mind when staying in a more theoretical paradigm, where pruned weights are simply put to zero and not removed. The primordial importance of the actual distribution of sparsity is only highlighted when comparing the energetic consumption and count of operations of pruned networks, using different methods, and non-pruned ones. All these questions call for further investigation in order to make sure pruning is profitable.

Chapter 7

Conclusion

Contents

In this manuscript, we reviewed the main aspects of neural networks pruning, discussed them, provided our own hindsight and finally exposed our contributions. In this conclusion chapter, we will recapitulate briefly the main elements to remember from each of them and explain how they answer the initial problem statement. This will then allow us to discuss the possible perspectives and avenues of exploration in the field and how our findings can lead to reconsider some fundamental aspects of pruning. Finally we will conclude on the personal outcomes of this thesis.

7.1 Summary

7.1.1 Reminder: Problem Statement

Convolutional Neural Networks (CNNs), and deep neural networks in general, have become a staple in many fields, including some that are crucial to industrial applications such as autonomous driving. However, these same industrial applications imply the use of embedded hardware, whose computation power cannot handle networks that are too big. Moreover, some applications such as autonomous driving also require a real-time responsiveness along with reliable enough performance. This means that CNNs have to be as accurate as possible while staying under a strict budget.

In order to improve this performance-to-cost trade-off, the field of neural networks compression has developed many techniques, with each their own extensive literature, fundamental questionings and epistemic issues. In order to provide a relevant hindsight, we chose to focus on one method and study it as thoroughly as possible, to finally tell how its different aspects answer or not the initial industrial problem.

The different aspects of compression, including pruning, each relate to these two symmetric principles: maximizing the performance at a given cost budget and reducing the said cost for a same performance budget. To sum it up: there is a continuity between the more theoretical and the more practical aspects of pruning.

7.1.2 Deep Neural Networks Compression and Pruning

Different compression methods focus on different aspects of the two principles mentioned above: distillation just improves the accuracy of a network without changing its architecture; quantization and clustering reduce the memory footprint of a same number of parameters; factorization, neural architecture search and pruning focus on producing efficient and/or sparse architectures.

Pruning involves removing parts, deemed unnecessary, of a network to reduce its cost. It implicitly involves answering first three questions: what type of parts to remove, how to identify the unnecessary ones and how to remove them; respectively, these three aspects are the pruning structures, criteria and removal methods. When adding the question "how to effectively implement these sparse networks for inference on embedded hardware", we get the four chapters of this manuscript, even though we presented them in a different order, to get a thematic progression from the most theoretical questions to the most hardware-related ones.

7.1.3 Pruning Criteria

First, we described two different criteria to tell the importance of a parameter: the gradient-based saliency metric and the weight magnitude criterion. The saliency metric has been theorized to predict the difference in the error function when setting a weight

to 0, however, when deconstructing the demonstrations backing up this argument, we saw that it implicitly required the involved weights to be very close to 0, which makes it redundant with the magnitude criterion. This weight magnitude criterion, for its part, has not received as much theoretical interest in the literature, even though it can be understood intuitively—the most intuitive reason being that pruning the smallest weights first is what, obviously, introduces the least difference in the value of the weights and, therefore, in the output of a layer.

We then presented different ways of extending criteria to larger structures, such as using the norm of said groups or inserting gates in the network. However, we saw that there are still no theoretical ways of ensuring that the value of such metrics can be comparable between layers, which harms the principle of global pruning; however, local pruning does not fare any better, as it produces, by definition, much more predictable architectures—especially in the case of structured pruning.

Finally, we deconstructed what was pruning fundamentally: a displacement and a projection on the error function $\mathcal{L}$, which highlights that the goal of pruning criteria should not be to minimize the performance degradation through putting weights to 0, but rather to predict how minimal can the new global optimum be after the projection. Arguably, there is no obvious way of predicting the optimum of the projection of a function $\mathcal{L}$ that is, by definition, too complicated to process analytically—because otherwise, we would not be using machine learning to solve it at all. However, this still shows that, in fact, there is not a real separation between crawling $\mathcal{L}$ during training, measuring the importance of weights and removing them—which also involves moving in $\mathcal{L}$. This also highlighted that, for a same resulting architecture, pruning or training from scratch should, theoretically, produce the same results, because in both cases, the same problem is solved.

The topic of pruning criteria, as we saw, mainly aims at finding the weights that, to be removed, allow for the best performance at a given parameters budget; therefore, it is more related to the performance/theoretical side of compression. However, in the case of global pruning, it also has a clear impact on the distribution of sparsity across layers, and since this sparsity does not have the same impact on the cost of the network depending on the layers, it also has a direct influence on the hardware/practical aspect of pruning.

7.1.4 Removal Methods and SWD

After we presented the basic train/prune/fine-tune method, we showed that all pruning methods aimed at making pruning smoother: indeed, setting a lot of weights to 0 at once equates to a huge displacement in $\mathcal{L}$, which may disrupt critically the training. After a review of each family of method, including the trendy "lottery ticket hypothesis" about which we raised doubts, we presented our own method, Selective Weight Decay, that showed convincing results when compared to more abrupt methods. Through ablation experiments, we showed that, with the right values of hyper-parameters, it could successfully remove parameters through a smooth penalty during training.

Our experiments showed that SWD could preserve non-random performance of networks even at very high pruning rates, which the other reference methods could not. However, we also showed that for a sufficiently low pruning rate, all pruning methods tend to behave the same, which anti-climatically questions the relevance of the whole dedicated literature. Finally, we showed that, in the end, we need a deeper theoretical understanding of pruning methods, as well as more rigorous practices to compare together different pruning methods in the literature.

Of course, the removal methods influence the final performance of the network, eventually the distribution of its sparsity (and thus the efficiency of the architecture) and the overall training time—because of all the possible retraining, fine-tuning, additional epochs or of an increased computational complexity at each training step. Therefore, removal methods are of clear theoretical interest but have some practical implications.

7.1.5 Pruning Structures

We showed that there is a wide array of different pruning structures, from weights to filters, passing by more or less constrained forms of sparsity. Each type of structure brings different benefits: for example, pruning filters allows reducing the number of parameters and operations as well as the size of the intermediate products of the network; however its impact on the network makes it reminiscent of neural architecture search. Pruning isolated weights does not reduce the size of the intermediate representations, and the actual reduction of parameters or operations depends on whether the implementation is able to handle sparse matrix multiplications; however, such a type of sparsity is difficult to get through the means of other methods than pruning. Globally, pruning structures can be classified according to the type of gains they theoretically allow and the specificity of the implementation required to leverage them. Therefore, they can be combined to get their respective benefits, while maximizing the performance—because the coarsest structures are easier to leverage and produce lots of gains, but also degrade the most the performance.

Concerning "structured" filter pruning, that we explored the most because of its efficiency, we showed that it could be disturbed by a problem of interdependence between the dimensions of layers. This problem is usually solved by heavily constraining pruning, which produces more predictable and possibly less efficient architectures.

The topic of pruning structures is of major interest, as it determines how exactly the cost of the network will be reduced and how easy it will be to leverage. Also, because of its impact on performance and on the choice of the pruning criteria, it also raises fundamental questions. The problem of interdependencies is crucial to solve, notably because it directly hinders the ability of pruning to produce efficient architectures.

7.1.6 Hardware Implementation and DCO

Finally, we dealt more specifically with the topic of hardware implementation. We saw that, when restricted only to the scope of neural networks themselves, our main degrees of freedom lie in the parameters (number, precision, memory management), the operations (number, complexity, hardware utilization) and the size and number of intermediate representations—that can end up having a larger memory footprint than parameters.

We briefly showed how the literature managed to leverage non-structured pruning before focusing on filter pruning. We presented a method, DCO, that could identify all the dependencies (exposed in the chapter before) and solve them—while adapting some operations when needed. This method allows to leverage any distribution of filter pruning, while previous methods had to rely on constraining pruning.

DCO allowed us then to study the efficiency of global filter pruning at reducing the cost of networks on hardware. While a clear reduction was observable, we saw that, compared to simply thinner baselines, pruned networks were energetically inefficient, while very efficient in term of parameters. We showed that this came from an imbalance in the used pruning criterion, that tended to target parameters that are responsi-

ble for few operations, thus creating networks that are poor in parameters but dense in operations. This showed once again that the theoretical question of pruning criteria, because of its impact on the distribution of sparsity, is actually crucial to solve the practical problem of producing energetically efficient neural networks architectures. Also, we showed that the removal method, because of its own impact on the distribution of sparsity, has its importance too.

Obviously, the topic of this chapter is directly related to the hardware-related side of the original problem statement. However, it also showed that there is not a clear separation between theory and practice in pruning, and that even the most theoretical questions, such as that of criteria or removal methods, actually have a decisive impact on the efficiency of produced architectures.

7.1.7 Answer to the Problem Statement

Finally, we showed that pruning, far from being just one single method that could be summarized easily, is actually a combination of multiple aspects that are not clearly separated but instead interwoven, so that they all have their importance for both the performance of the network and its cost in memory, latency or energy on embedded hardware. Ultimately, these observations justify our choice of focusing on one method only, as we showed that, far from being vain quibbles, such minute attentions revealed to be necessary to get any benefit at all from pruning, and even more to produce architectures that are actually efficient.

Indeed, in the end, the best possible conclusion about pruning is to say that, for a method that is extremely popular and supposedly efficient, we cannot say whether or not it is able to produce better results than trivial smaller networks trained from scratch at all. Being an entanglement of multiple theoretical or practical aspects, many of which are still not reliably formalized, there is no way to conclude the question of pruning before much further investigation.

Fortunately, not only does it imply that, despite the popularity of pruning, there is still a lot of room for progress; but also, because of the many fundamental aspects of deep learning it tackles and its many bridges with the domain of neural architecture search, the field of pruning, even if it means undergoing major changes in the future, still yields much interest, both from an epistemic and industrial point of view.

7.2 Perspectives in the Field

We just said that our work revealed that there was a lot of room for further investigation in the field of pruning. Let's review first what are the immediate avenues of explorations that come to mind for each chapter:

- Concerning pruning criteria, besides the last discussion that we will tackle later:
    - The most immediate topic of interest is the question of the distribution of sparsity across layers. Indeed, we figured out that imbalance in the pruning criteria could harm the efficiency of the produced architectures. We also saw that this imbalance may be partly due to some conservation rules between layers, which means that it is surely possible to look for some theoretical or mathematical clues to understand better this imbalance problem and eventually solve it.

  - This problem is also related to that of criteria applied to structured pruning: while inserting gates to prune filters is relatively easy, there is no such proxy in the case of the other types of structured pruning. It is problematic, as we already mentioned that pruning on the basis of the norm of groups can pose problems. This means that we should either investigate how to use norm in a more reliable way, or look for some more generic kind of proxy than gates.
  - Finally, we need tools to compare more deeply pruning criteria, especially since many of them, despite different theoretical backgrounds, tend to overlap. We should investigate the precise behavior of criteria, which weights do they target and for which reasons, and whether or not the targeted weights are really the most relevant to remove.
- Concerning removal methods:
  - We mentioned that, for low pruning rates, all removal methods tend to behave the same. This calls for a thorough, rigorous and synthetic comparison of the different families of pruning methods, from low to very high pruning rates.
  - We already mentioned that it could be worth testing different functions for the increase of the penalty of SWD during training. More generally, studying the impact of the pruning/penalization schedule during training could be important, as it impacts both the relevance of the pruning criteria and the ability of the network and the training to recover from the perturbation.
- Concerning structures:
  - Few papers bother to mix together different types of structures, while we mentioned that it could allow for a finer control over the performance and the cost of the network. How exactly to balance such a mix is still to be found.
  - Also, since each type of structure can be more or less punitive in term of performance, it would be interesting to study this fact more thoroughly—as well as the difference in performance between local and global pruning, for each type of structure, which however presupposes solving the theoretical problems with global pruning.
  - At a higher level, we mentioned that the largest possible pruning structures made pruning very related to neural architecture search. Obviously, it would be interesting to do a thorough synthesis of the common points between the two fields and of what they can bring to each other.
- Concerning hardware implementation:
  - We mentioned the necessity of finding a better implementation for the indexation-addition operation; we gave the example of 1×1 convolutions. However, if we replace additions with such layers, the logical continuity of this would be to train these 1×1 convolutions and just consider this modified ResNet as a new architecture, that replaces restrictive operations such as additions by permissive, pruning-friendly layers. Once again, designing an architecture that is explicitly made to be pruned is reminiscent of the supernets of neural architecture search.
  - We also mentioned the problem with the imbalance in pruning criteria between layers. Besides understanding the theoretical reasons behind such an

imbalance, one possibility could be of designing criteria that are especially made to bring energetic efficiency, which may not be completely equivalent to finding the best parameters-to-accuracy ratio.

However, multiple discussions we had have a scope that is more general than one isolated aspect of pruning. Here are some perspectives concerning pruning in general:

- First, the big question is whether or not pruning is able to produce architectures (or sparse networks) that are more efficient than trivial ones trained from scratch. This is a question that requires solving many fundamental problems with both criteria and removal methods for a given pruning structure: every aspect of pruning has to be tackled to answer this question. The fact that this essential question, that basically tells if pruning is worth anything at all from a practical point of view, is still unanswered and arguably impossible to answer, in the current state of our knowledge, says a lot about the state of this literature.
- Presenting pruning as a displacement and a projection of $\mathcal{L}$ highlights a new way of thinking both the criteria and the removal methods. Of course, dealing in term of global optimums instead of locally relevant weights is difficult, as we already said, but it also shows that the distribution of sparsity may actually be even more important than the specific weights to prune, especially for structured pruning, since neurons/filters are structurally interchangeable in a same layer—if we neglect the (very significant) importance of initialization in training. Overall, this avenue of exploration may lead to the fusion of the principle of criteria and of removal methods, because both imply finding the best way to move inside the landscape of $\mathcal{L}$ under a given constraint.
- Also, we saw that there is an intimate influence between theoretical questions, such as that of criteria, and the direct practical impact on the cost of networks on hardware. Indeed, structures have to be chosen depending on how well they can be leveraged, the criteria depend on the structure and the methods have an impact on the outcome of the criteria. Moreover, we showed that what is more efficient in practice, in term of implementation, is not always what looks better on paper (cf. using *ScatterND* operations for indexation-addition layers). This means that both worlds should not be too separated, as isolating them can lead to absurd outcomes—such as implementations that only get benefits counting from a sparsity level of 95%, or on the other hand pruning structures that may not be possible to leverage at all.

Finally, it is also possible to formulate expectations that may encompass beyond the sole field of pruning:

- While non-structured pruning produces sparse networks, that are rarely obtained through other methods than pruning, structured types of pruning are much more akin to NAS—after all, some pruning criteria are already used in NAS. This means that the two literature should not be too separated, since on the conceptual level, they have a significant overlap. While this thesis did not explore NAS, we advocate for a more thorough comparison of the two fields.
- Some papers, such as that of Hooker *et al.* [28], suggest that all compression methods do not have the same kind of impact on the robustness or performance of networks; we can extend this reasoning on even their cost on hardware. Unfortunately, such papers could only afford comparing prototypical and, therefore, imperfect versions of each types of methods—and we saw that the overall moral of this manuscript is that it is not always desirable to sum up a whole field with

a single strawman. Studying in depth the relative impact of fundamentally different compression methods on the same metrics, and even their relative impact on each other when applied conjointly, seems to be an essential topic of study, especially for industrial applications.

- We considered in this manuscript pruning only as a compression method to produce efficient architectures. However, there are some fields, such as federated learning, where pruning can be used to make networks lighter to transfer between clients (devices on which individual networks are trained on different data sources) and servers (where networks can be fused and then sent back to clients to update them) [229], [230].
- From a more theoretical point of view, pruning can raise interesting questions on the relation between the contribution of each parameter and convergence when crawling various projections of $\mathcal{L}$; after all, when considering that any network can be seen as the pruned version of a bigger one, we can consider that all the possible architectures to train on a same problem each correspond to a different projection of the same hypothetical super-$\mathcal{L}$. Similarly, pruning, that considers when and how to remove a parameter, implicitly raises the question of when to add a parameter and how it will affect the relative behaviors of all the others, while keeping in mind that all types of parameters are not expected to behave the same; for example, summing two different biases on the same channel at the same time is purely redundant, while adding a new filter is much more impactful—even though the order of these filters is not important, since channels are interchangeable.
- We can also wonder, when considering that the goal of neural networks is to approximate functions, if, once said function is approximated, neural networks are the best tool to keep representing such functions and if they could not be converted into much less expensive kinds of algorithms. Indeed, neural networks are renowned for their outstanding ability at converging through gradient descent on extremely complex problems, but that does not mean that the solutions, found using them, still have to be represented using a neural network. It is conceivable that trying to squeeze expensive networks on unfitting hardware is a chimera, because the answer from the start was that neural networks were not meant to be the final product of learning.

Bibliography

[1] H. Tessier, V. Gripon, M. Léonardon, M. Arzel, T. Hannagan, and D. Bertrand, "Rethinking weight decay for efficient neural network pruning", *Journal of Imaging*, vol. 8, no. 3, p. 64, 2022.

[2] H. Tessier, V. Gripon, M. Léonardon, M. Arzel, D. Bertrand, and T. Hannagan, "Energy consumption analysis of pruned semantic segmentation networks on an embedded gpu", in *International Conference on System-Integrated Intelligence*, Springer, 2023, pp. 553–563.

[3] ——, "Leveraging structured pruning of convolutional neural networks", in *2022 IEEE Workshop on Signal Processing Systems (SiPS)*, IEEE, 2022, pp. 1–6.

[4] G. B. Hacene, "Processing and learning deep neural networks on chip", Ph.D. dissertation, Ecole nationale supérieure Mines-Télécom Atlantique, 2019.

[5] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: concepts, cnn architectures, challenges, applications, future directions", *Journal of big Data*, vol. 8, no. 1, pp. 1–74, 2021.

[6] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.", *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] M. Minsky and S. A. Papert, *Perceptrons, Reissue of the 1988 Expanded Edition with a new foreword by Léon Bottou: An Introduction to Computational Geometry*. MIT press, 2017.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", *Advances in neural information processing systems*, vol. 25, 2012.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

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

[12] Y. Aytar, C. Vondrick, and A. Torralba, "Soundnet: learning sound representations from unlabeled video", *Advances in neural information processing systems*, vol. 29, 2016.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: pre-training of deep bidirectional transformers for language understanding", *arXiv preprint arXiv:1810.04805*, 2018.

[14] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with clip latents", *arXiv preprint arXiv:2204.06125*, 2022.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding", *arXiv preprint arXiv:1510.00149*, 2015.

[16] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, *et al.*, "Highly accurate protein structure prediction with alphafold", *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[17] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: efficient primitives for deep learning", *arXiv preprint arXiv:1410.0759*, 2014.

[18] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge", *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[19] J. Yu, Z. Wang, V. Vasudevan, L. Yeung, M. Seyedhosseini, and Y. Wu, "Coca: contrastive captioners are image-text foundation models", *arXiv preprint arXiv:2205.01917*, 2022.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *arXiv preprint arXiv:1409.1556*, 2014.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[22] K. Sun, Y. Zhao, B. Jiang, T. Cheng, B. Xiao, D. Liu, Y. Mu, X. Wang, W. Liu, and J. Wang, "High-resolution representations for labeling pixels and regions", *arXiv preprint arXiv:1904.04514*, 2019.

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: unified, real-time object detection", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[24] A. R. Vidal, H. Rebecq, T. Horstschaefer, and D. Scaramuzza, "Ultimate slam? combining events, images, and imu for robust visual slam in hdr and high-speed scenarios", *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 994–1001, 2018.

[25] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223.

[26] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network", *Advances in neural information processing systems*, vol. 28, 2015.

[27] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?", *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.

[28] S. Hooker, A. Courville, G. Clark, Y. Dauphin, and A. Frome, "What do compressed deep neural networks forget?", *arXiv preprint arXiv:1911.05248*, 2019.

[29] Y. Le Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard, "Handwritten digit recognition: applications of neural network chips and automatic learning", *IEEE Communications Magazine*, vol. 27, no. 11, pp. 41–46, 1989.

[30] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images", 2009.

[31] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift", in *International conference on machine learning*, PMLR, 2015, pp. 448–456.

[32] M. Lin, Q. Chen, and S. Yan, "Network in network", *arXiv preprint arXiv:1312.4400*, 2013.

[33] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features", in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, Ieee, vol. 1, 2001, pp. I–I.

[34] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[35] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size", *arXiv preprint arXiv:1602.07360*, 2016.

[36] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[37] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.

[38] 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", *arXiv preprint arXiv:1704.04861*, 2017.

[39] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: inverted residuals and linear bottlenecks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[40] M. Tan and Q. Le, "Efficientnet: rethinking model scaling for convolutional neural networks", in *International conference on machine learning*, PMLR, 2019, pp. 6105–6114.

[41] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: an imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[42] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[43] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation", in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1520–1528.

[44] O. Ronneberger, P. Fischer, and T. Brox, "U-net: convolutional networks for biomedical image segmentation", in *International Conference on Medical image computing and computer-assisted intervention*, Springer, 2015, pp. 234–241.

[45] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: a deep convolutional encoder-decoder architecture for image segmentation", *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[46] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions", *arXiv preprint arXiv:1511.07122*, 2015.

[47] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs", *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[48] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation", *arXiv preprint arXiv:1706.05587*, 2017.

[49] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation", in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.

[50] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition", *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015.

[51] J. Fu, J. Liu, H. Tian, Y. Li, Y. Bao, Z. Fang, and H. Lu, "Dual attention network for scene segmentation", in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 3146–3154.

[52] K. Sun, B. Xiao, D. Liu, and J. Wang, "Deep high-resolution representation learning for human pose estimation", in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 5693–5703.

[53] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, *et al.*, "Deep high-resolution representation learning for visual recognition", *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, no. 10, pp. 3349–3364, 2020.

[54] Y. Yuan, X. Chen, and J. Wang, "Object-contextual representations for semantic segmentation", in *European conference on computer vision*, Springer, 2020, pp. 173–190.

[55] A. Krogh and J. Hertz, "A simple weight decay can improve generalization", *Advances in neural information processing systems*, vol. 4, 1991.

[56] S. Zhao, Y. Wang, Z. Yang, and D. Cai, "Region mutual information loss for semantic segmentation", *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[57] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning", *arXiv preprint arXiv:1810.05270*, 2018.

[58] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: a survey", *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[59] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning", *arXiv preprint arXiv:1611.02167*, 2016.

[60] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[61] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2423–2432.

[62] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms.", in *ICGA*, vol. 89, 1989, pp. 379–384.

[63] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, "Neural architecture search with bayesian optimisation and optimal transport", *Advances in neural information processing systems*, vol. 31, 2018.

[64] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction", *arXiv preprint arXiv:1705.10823*, 2017.

[65] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search", in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 19–34.

[66] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: train one network and specialize it for efficient deployment", *arXiv preprint arXiv:1908.09791*, 2019.

[67] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: principles and empirical evaluation", *arXiv preprint arXiv:2004.09602*, 2020.

[68] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: training deep neural networks with binary weights during propagations", *Advances in neural information processing systems*, vol. 28, 2015.

[69] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation", *arXiv preprint arXiv:1308.3432*, 2013.

[70] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: parameterized clipping activation for quantized neural networks", *arXiv preprint arXiv:1805.06085*, 2018.

[71] G. Soulié, V. Gripon, and M. Robert, "Compression of deep neural networks on the fly", in *International Conference on Artificial Neural Networks*, Springer, 2016, pp. 153–160.

[72] P. Stock, A. Joulin, R. Gribonval, B. Graham, and H. Jégou, "And the bit goes down: revisiting the quantization of neural networks", *arXiv preprint arXiv:1907.05686*, 2019.

[73] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-means: re-training and parameter sharing with harder cluster assignments for compressing deep convolutions", in *International Conference on Machine Learning*, PMLR, 2018, pp. 5363–5372.

[74] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization", *arXiv preprint arXiv:1412.6115*, 2014.

[75] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search", *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[76] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: a survey", *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[77] G. Hinton, O. Vinyals, J. Dean, *et al.*, "Distilling the knowledge in a neural network", *arXiv preprint arXiv:1503.02531*, vol. 2, no. 7, 2015.

[78] T. Furlanello, Z. Lipton, M. Tschannen, L. Itti, and A. Anandkumar, "Born again neural networks", in *International Conference on Machine Learning*, PMLR, 2018, pp. 1607–1616.

[79] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: hints for thin deep nets", *arXiv preprint arXiv:1412.6550*, 2014.

[80] W. Park, D. Kim, Y. Lu, and M. Cho, "Relational knowledge distillation", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 3967–3976.

[81] C. Lassance, M. Bontonou, G. B. Hacene, V. Gripon, J. Tang, and A. Ortega, "Deep geometric knowledge distillation with graphs", in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2020, pp. 8484–8488.

[82] X. Ma, S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, S. H. Tan, Z. Li, D. Fan, X. Qian, *et al.*, "Non-structured dnn weight pruning–is it beneficial in any platform?", *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[83] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets", *arXiv preprint arXiv:1608.08710*, 2016.

[84] Y. Chauvin, "A back-propagation algorithm with optimal use of hidden units", *Advances in neural information processing systems*, vol. 1, 1988.

[85] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage", *Advances in neural information processing systems*, vol. 2, 1989.

[86] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference", *arXiv preprint arXiv:1611.06440*, 2016.

[87] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming", in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2736–2744.

[88] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science", *Nature communications*, vol. 9, no. 1, pp. 1–12, 2018.

[89] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through L_0 regularization", *arXiv preprint arXiv:1712.01312*, 2017.

[90] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression", *arXiv preprint arXiv:1710.01878*, 2017.

[91] J. Frankle and M. Carbin, "The lottery ticket hypothesis: finding sparse, trainable neural networks", *arXiv preprint arXiv:1803.03635*, 2018.

[92] A. Renda, J. Frankle, and M. Carbin, "Comparing rewinding and fine-tuning in neural network pruning", *arXiv preprint arXiv:2003.02389*, 2020.

[93] R. Reed, "Pruning algorithms-a survey", *IEEE transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, 1993.

[94] M. C. Mozer and P. Smolensky, "Skeletonization: a technique for trimming the fat from a network via relevance assessment", *Advances in neural information processing systems*, vol. 1, 1988.

[95] E. D. Karnin, "A simple procedure for pruning back-propagation trained neural networks", *IEEE transactions on neural networks*, vol. 1, no. 2, pp. 239–242, 1990.

[96] B. Hassibi and D. Stork, "Second order derivatives for network pruning: optimal brain surgeon", *Advances in neural information processing systems*, vol. 5, 1992.

[97] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 264–11 272.

[98] H. Tanaka, D. Kunin, D. L. Yamins, and S. Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow", *Advances in Neural Information Processing Systems*, vol. 33, pp. 6377–6389, 2020.

[99] N. Lee, T. Ajanthan, and P. H. Torr, "Snip: single-shot network pruning based on connection sensitivity", *arXiv preprint arXiv:1810.02340*, 2018.

[100] L. Yann, "Modeles connexionnistes de lapprentissage", Ph.D. dissertation, These de Doctorat, Universite Paris, 1987.

[101] X. Dong, S. Chen, and S. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon", *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[102] V. Tresp, R. Neuneier, and H.-G. Zimmermann, "Early brain damage", *Advances in neural information processing systems*, vol. 9, 1996.

[103] X. Ding, G. Ding, X. Zhou, Y. Guo, J. Liu, and J. Han, "Global sparse momentum sgd for pruning very deep neural networks", in *NeurIPS*, 2019.

[104] D. C. Plaut *et al.*, "Experiments on learning by back propagation.", 1986.

[105] S. Hanson and L. Pratt, "Comparing biases for minimal network construction with back-propagation", in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 1, Morgan-Kaufmann, 1988. [Online]. Available: `https://proceedings.neurips.cc/paper/1988/file/1c9ac0159c94d8d0cbedc973445af2c Paper.pdf`.

[106] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Generalization by weight-elimination applied to currency exchange rate prediction", in *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, IEEE, 1991, pp. 2374–2379.

[107] A. Weigend, D. Rumelhart, and B. Huberman, "Generalization by weight-elimination with application to forecasting", *Advances in neural information processing systems*, vol. 3, 1990.

[108] C. Ji, R. R. Snapp, and D. Psaltis, "Generalizing smoothness constraints from discrete samples", *Neural Computation*, vol. 2, no. 2, pp. 188–197, 1990.

[109] S. J. Nowlan and G. E. Hinton, "Simplifying neural networks by soft weight sharing", in *The Mathematics of Generalization*, CRC Press, 2018, pp. 373–394.

[110] S. A. Janowsky, "Pruning versus clipping in neural networks", *Physical Review A*, vol. 39, no. 12, p. 6600, 1989.

[111] B. E. Segee and M. J. Carter, "Fault tolerance of pruned multilayer networks", in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, IEEE, vol. 2, 1991, pp. 447–452.

[112] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks", in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1389–1397.

[113] J. Sietsma, "Neural net pruning-why and how", in *Proceedings of International Conference on Neural Networks, San Diego, CA, 1988*, vol. 1, 1988, pp. 325–333.

[114] J. Sietsma and R. J. Dow, "Creating artificial neural networks that generalize", *Neural networks*, vol. 4, no. 1, pp. 67–79, 1991.

[115] F.-L. Chung and T. Lee, "A node pruning algorithm for backpropagation networks", *International Journal of Neural Systems*, vol. 3, no. 03, pp. 301–314, 1992.

[116] D. Whitley, "The evolution of connectivity: pruning neural networks using genetic algorithm", in *Proceedings of IJCNN-90*, 1990, pp. 134–137.

[117] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: optimizing connections and connectivity", *Parallel computing*, vol. 14, no. 3, pp. 347–361, 1990.

[118] S. W. Stepniewski and A. J. Keane, "Pruning backpropagation neural networks using modern stochastic optimisation techniques", *Neural Computing & Applications*, vol. 5, no. 2, pp. 76–98, 1997.

[119] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: a data-driven neuron pruning approach towards efficient deep architectures", *arXiv preprint arXiv:1607.03250*, 2016.

[120] Q. Huang, K. Zhou, S. You, and U. Neumann, "Learning to prune filters in convolutional neural networks", in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, IEEE, 2018, pp. 709–718.

[121] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: automl for model compression and acceleration on mobile devices", in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 784–800.

[122] K. Yamamoto and K. Maeno, "Pcas: pruning channels with attention statistics for deep network compression", *arXiv preprint arXiv:1806.05382*, 2018.

[123] G. B. Hacene, C. Lassance, V. Gripon, M. Courbariaux, and Y. Bengio, "Attention based pruning for shift networks", in *2020 25th International Conference on Pattern Recognition (ICPR)*, IEEE, 2021, pp. 4054–4061.

[124] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks", *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–18, 2017.

[125] S. Anwar and W. Sung, "Compact deep convolutional neural networks with coarse pruning", *arXiv preprint arXiv:1610.09639*, 2016.

[126] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks", *arXiv preprint arXiv:1507.06149*, 2015.

[127] J.-H. Luo, J. Wu, and W. Lin, "Thinet: a filter level pruning method for deep neural network compression", in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

[128] X. Ding, G. Ding, J. Han, and S. Tang, "Auto-balanced filter pruning for efficient convolutional neural networks", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[129] V. Lebedev and V. Lempitsky, "Fast convnets using group-wise brain damage", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2554–2564.

[130] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks", *Advances in neural information processing systems*, vol. 29, 2016.

[131] J. M. Alvarez and M. Salzmann, "Learning the number of neurons in deep networks", *Advances in neural information processing systems*, vol. 29, 2016.

[132] S. Gao, X. Liu, L.-S. Chien, W. Zhang, and J. M. Alvarez, "Vacl: variance-aware cross-layer regularization for pruning deep residual networks", in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, IEEE Computer Society, 2019, pp. 2980–2988.

[133] Y. Li, S. Gu, C. Mayer, L. V. Gool, and R. Timofte, "Group sparsity: the hinge between filter pruning and decomposition for network compression", in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 8018–8027.

[134] J. Ye, X. Lu, Z. Lin, and J. Z. Wang, "Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers", *arXiv preprint arXiv:1802.00124*, 2018.

[135] C. K. Mummadi, T. Genewein, D. Zhang, T. Brox, and V. Fischer, "Group pruning using a bounded-ℓ_p norm for group gating and regularization", in *German Conference on Pattern Recognition*, Springer, 2019, pp. 139–155.

[136] N. Jiang, X. Zhao, C. Zhao, Y. An, M. Tang, and J. Wang, "Pruning-aware sparse regularization for network pruning", *arXiv preprint arXiv:2201.06776*, 2022.

[137] L. Liu, S. Zhang, Z. Kuang, A. Zhou, J.-H. Xue, X. Wang, Y. Chen, W. Yang, Q. Liao, and W. Zhang, "Group fisher pruning for practical network compression", in *International Conference on Machine Learning*, PMLR, 2021, pp. 7021–7032.

[138] M. Kang and B. Han, "Operation-aware soft channel pruning using differentiable masks", in *International Conference on Machine Learning*, PMLR, 2020, pp. 5122–5131.

[139] J. K. Kruschke and J. R. Movellan, "Benefits of gain: speeded learning and minimal hidden layers in back-propagation networks", *IEEE Transactions on systems, Man, and Cybernetics*, vol. 21, no. 1, pp. 273–280, 1991.

[140] T. Dettmers and L. Zettlemoyer, "Sparse networks from scratch: faster training without losing performance", *arXiv preprint arXiv:1907.04840*, 2019.

[141] M. Lin, R. Ji, Y. Zhang, B. Zhang, Y. Wu, and Y. Tian, "Channel pruning via automatic structure search", *arXiv preprint arXiv:2001.08565*, 2020.

[142] B. Li, B. Wu, J. Su, and G. Wang, "Eagleeye: fast sub-net evaluation for efficient neural network pruning", in *European conference on computer vision*, Springer, 2020, pp. 639–654.

[143] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun, "Metapruning: meta learning for automatic neural network channel pruning", in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 3296–3305.

[144] T. Zhuang, Z. Zhang, Y. Huang, X. Zeng, K. Shuang, and X. Li, "Neuron-level structured pruning using polarization regularizer", *Advances in neural information processing systems*, vol. 33, pp. 9865–9877, 2020.

[145] T. Liang, T. Poggio, A. Rakhlin, and J. Stokes, "Fisher-rao metric, geometry, and complexity of neural networks", in *The 22nd international conference on artificial intelligence and statistics*, PMLR, 2019, pp. 888–896.

[146] S. S. Du, W. Hu, and J. D. Lee, "Algorithmic regularization in learning deep homogeneous models: layers are automatically balanced", *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[147] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation", *PloS one*, vol. 10, no. 7, e0130140, 2015.

[148] K. Dhamdhere, M. Sundararajan, and Q. Yan, "How important is a neuron?", *arXiv preprint arXiv:1805.12233*, 2018.

[149] S.-K. Yeom, P. Seegerer, S. Lapuschkin, A. Binder, S. Wiedemann, K.-R. Müller, and W. Samek, "Pruning by explaining: a novel criterion for deep neural network pruning", *Pattern Recognition*, vol. 115, p. 107 899, 2021.

[150] H. Tanaka, A. Nayebi, N. Maheswaranathan, L. McIntosh, S. Baccus, and S. Ganguli, "From deep learning to mechanistic understanding in neuroscience: the structure of retinal prediction", *Advances in neural information processing systems*, vol. 32, 2019.

[151] J. Frankle, G. K. Dziugaite, D. M. Roy, and M. Carbin, "Stabilizing the lottery ticket hypothesis", *arXiv preprint arXiv:1903.01611*, 2019.

[152] B. Bartoldson, A. Morcos, A. Barbu, and G. Erlebacher, "The generalization-stability tradeoff in neural network pruning", *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 852–20 864, 2020.

[153] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks", *arXiv preprint arXiv:1902.09574*, 2019.

[154] M. A. Carreira-Perpinán and Y. Idelbayev, ""learning-compression" algorithms for neural net pruning", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8532–8541.

[155] Y. Choi, M. El-Khamy, and J. Lee, "Compression of deep convolutional neural networks under joint sparsity constraints", *arXiv preprint arXiv:1805.08303*, 2018.

[156] D. Molchanov, A. Ashukha, and D. Vetrov, "Variational dropout sparsifies deep neural networks", in *International Conference on Machine Learning*, PMLR, 2017, pp. 2498–2507.

[157] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks", *arXiv preprint arXiv:1808.06866*, 2018.

[158] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization", in *International Conference on Machine Learning*, PMLR, 2019, pp. 4646–4655.

[159] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: making all tickets winners", in *International Conference on Machine Learning*, PMLR, 2020, pp. 2943–2952.

[160] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns", *Advances in neural information processing systems*, vol. 29, 2016.

[161] S. Srinivas, A. Subramanya, and R. Venkatesh Babu, "Training sparse neural networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 138–145.

[162] X. Xiao, Z. Wang, and S. Rajasekaran, "Autoprune: automatic network pruning by regularizing auxiliary parameters", *Advances in neural information processing systems*, vol. 32, 2019.

[163] P. Savarese, H. Silva, and M. Maire, "Winning the lottery with continuous sparsification", *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 380–11 390, 2020.

[164] H. Zhou, J. Lan, R. Liu, and J. Yosinski, "Deconstructing lottery tickets: zeros, signs, and the supermask", *Advances in neural information processing systems*, vol. 32, 2019.

[165] A. Morcos, H. Yu, M. Paganini, and Y. Tian, "One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers", *Advances in neural information processing systems*, vol. 32, 2019.

[166] E. Malach, G. Yehudai, S. Shalev-Schwartz, and O. Shamir, "Proving the lottery ticket hypothesis: pruning is all you need", in *International Conference on Machine Learning*, PMLR, 2020, pp. 6682–6691.

[167] Z. Zhang, J. Jin, Z. Zhang, Y. Zhou, X. Zhao, J. Ren, J. Liu, L. Wu, R. Jin, and D. Dou, "Validating the lottery ticket hypothesis with inertial manifold theory", *Advances in Neural Information Processing Systems*, vol. 34, pp. 30 196–30 210, 2021.

[168] N. Lee, T. Ajanthan, S. Gould, and P. H. Torr, "A signal propagation perspective for pruning neural networks at initialization", *arXiv preprint arXiv:1906.06307*, 2019.

[169] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow", *arXiv preprint arXiv:2002.07376*, 2020.

[170] J. Frankle, G. K. Dziugaite, D. M. Roy, and M. Carbin, "Pruning neural networks at initialization: why are we missing the mark?", *arXiv preprint arXiv:2009.08576*, 2020.

[171] J. Frankle, G. K. Dziugaite, D. Roy, and M. Carbin, "Linear mode connectivity and the lottery ticket hypothesis", in *International Conference on Machine Learning*, PMLR, 2020, pp. 3259–3269.

[172] S. Yin, K.-H. Kim, J. Oh, N. Wang, M. Serrano, J.-S. Seo, and J. Choi, "The sooner the better: investigating structure of early winning lottery tickets", 2019.

[173] R. Mehta, "Sparse transfer learning via winning lottery tickets", *arXiv preprint arXiv:1905.07785*, 2019.

[174] R. Van Soelen and J. W. Sheppard, "Using winning lottery tickets in transfer learning for convolutional neural networks", in *2019 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2019, pp. 1–8.

[175] S. Desai, H. Zhan, and A. Aly, "Evaluating lottery tickets under distributional shifts", *arXiv preprint arXiv:1910.12708*, 2019.

[176] H. Liu, K. Simonyan, and Y. Yang, "Darts: differentiable architecture search", *arXiv preprint arXiv:1806.09055*, 2018.

[177] D. P. Kingma and M. Welling, "Auto-encoding variational bayes", *CoRR*, vol. abs/1312.6114, 2014.

[178] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic back-propagation and variational inference in deep latent gaussian models", *ArXiv*, vol. abs/1401.4082, 2014.

[179] R. Tibshirani, "Regression shrinkage and selection via the lasso", *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996. DOI: 10.1111/j.2517-6161.1996.tb02080.x.

[180] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables", *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 68, 2006.

[181] J. Chang and J. Sha, "Prune deep neural networks with the modified $L_{1/2}$ penalty", *IEEE Access*, vol. 7, pp. 2273–2280, 2019. DOI: 10.1109/ACCESS.2018.2886876.

[182] D. P. Kingma, T. Salimans, and M. Welling, "Variational dropout and the local reparameterization trick", *ArXiv*, vol. abs/1506.02557, 2015.

[183] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors", *ArXiv*, vol. abs/1207.0580, 2012.

[184] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting", *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: `http://jmlr.org/papers/v15/srivastava14a.html`.

[185] K. Neklyudov, D. Molchanov, A. Ashukha, and D. P. Vetrov, "Structured bayesian pruning via log-normal multiplicative noise", *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[186] C. Louizos, K. Ullrich, and M. Welling, "Bayesian compression for deep learning", *Advances in neural information processing systems*, vol. 30, 2017.

[187] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers", in *ECCV*, 2018.

[188] T. Zhang, S. Ye, K. Zhang, X. Ma, N. Liu, L. Zhang, J. Tang, K. Ma, X. Lin, M. Fardad, and Y. Wang, "Structadmm: a systematic, high-efficiency framework of structured weight pruning for dnns", *arXiv: Neural and Evolutionary Computing*, 2018.

[189] S. Ye, T. Zhang, K. Zhang, J. Li, J. Xie, Y. Liang, S. Liu, X. Lin, and Y. Wang, "A unified framework of dnn weight pruning and weight clustering/quantization using admm", *ArXiv*, vol. abs/1811.01907, 2018.

[190] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "Admm-nn: an algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers", *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[191] S. Ye, X. Feng, T. Zhang, X. Ma, S. Lin, Z. Li, K. Xu, W. Wen, S. Liu, J. Tang, M. Fardad, X. Lin, Y. Liu, and Y. Wang, "Progressive dnn compression: a key to achieve ultra-high weight pruning and quantization rates using admm", *ArXiv*, vol. abs/1903.09769, 2019.

[192] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, "Autocompress: an automatic dnn structured pruning framework for ultra-high compression rates", in *AAAI*, 2020.

[193] Z. Li, Y. Gong, X. Ma, S. Liu, M. Sun, Z. Zhan, Z. Kong, G. Yuan, and Y. Wang, "Ss-auto: a single-shot, automatic structured weight pruning framework of dnns with ultra-high efficiency", *ArXiv*, vol. abs/2001.08839, 2020.

[194] W. Murray and K.-M. Ng, "An algorithm for nonlinear optimization problems with binary variables", *Computational optimization and applications*, vol. 47, no. 2, pp. 257–288, 2010.

[195] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data", *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.

[196] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks", *arXiv preprint arXiv:1609.02907*, 2016.

[197] Z. Liu, J. Xu, X. Peng, and R. Xiong, "Frequency-domain dynamic pruning for convolutional neural networks", *Advances in neural information processing systems*, vol. 31, 2018.

[198] X. Yu, T. Serra, S. Zhe, and S. Ramalingam, "The combinatorial brain surgeon: pruning weights that cancel one another in neural networks", *arXiv preprint arXiv:2203.04466*, 2022.

[199] J. S. Rosenfeld, J. Frankle, M. Carbin, and N. Shavit, "On the predictability of pruning across scales", in *International Conference on Machine Learning*, PMLR, 2021, pp. 9075–9083.

[200] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, "Autocompress: an automatic dnn structured pruning framework for ultra-high compression rates", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 4876–4883.

[201] S. Lym, E. Choukse, S. Zangeneh, W. Wen, S. Sanghavi, and M. Erez, "Prunetrain: fast neural network training by dynamic sparse model reconfiguration", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–13.

[202] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast sparse convnets", in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 14 629–14 638.

[203] Z.-G. Liu, P. N. Whatmough, Y. Zhu, and M. Mattina, "S2ta: exploiting structured sparsity for energy-efficient mobile cnn acceleration", in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2022, pp. 573–586.

[204] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient dnn inference on gpu", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 5676–5683.

[205] B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer, "Shift: a zero flop, zero parameter alternative to spatial convolutions", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 9127–9135.

[206] G. Boukli Hacene, "Processing and learning deep neural networks on chip", Ph.D. dissertation, Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire, 2019.

[207] S. Zhong, G. Zhang, N. Huang, and S. Xu, "Revisit kernel pruning with lottery regulated grouped convolutions", in *International Conference on Learning Representations*, 2021.

[208] A. N. Burkitt, "Optimization of the architecture of feed-forward neural networks with hidden layers by unit elimination", *Complex Systems*, vol. 5, no. 4, pp. 371–380, 1991.

[209] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration", in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 4340–4349.

[210] M. Lin, R. Ji, Y. Wang, Y. Zhang, B. Zhang, Y. Tian, and L. Shao, "Hrank: filter pruning using high-rank feature map", in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 1529–1538.

[211] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing", in *International conference on machine learning*, PMLR, 2018, pp. 4095–4104.

[212] M. S. Abdelfattah, A. Mehrotra, Ł. Dudziak, and N. D. Lane, "Zero-cost proxies for lightweight nas", *arXiv preprint arXiv:2101.08134*, 2021.

[213] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training", in *International Conference on Machine Learning*, PMLR, 2021, pp. 7588–7598.

[214] A. Glinserer, "Autopruning with intel distiller and evaluation on a jetson xavier agx", Ph.D. dissertation, Wien, 2021.

[215] A. Glinserer, M. Lechner, and A. Wendt, "Automated pruning of neural networks for mobile applications", in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, IEEE, 2021, pp. 1–6.

[216] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks", *Synthesis Lectures on Computer Architecture*, vol. 15, no. 2, pp. 1–341, 2020.

[217] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: enabling in-memory computation", *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019.

[218] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky, "Dark memory and accelerator-rich system optimization in the dark silicon era", *IEEE Design & Test*, vol. 34, no. 2, pp. 39–50, 2016.

[219] H. Younes, H. L. Blevec, M. Léonardon, and V. Gripon, "Inter-operability of compression techniques for efficient deployment of cnns on microcontrollers", in *International Conference on System-Integrated Intelligence*, Springer, 2023, pp. 543–552.

[220] A. Frickenstein, C. Unger, and W. Stechele, "Resource-aware optimization of dnns for embedded applications", in *2019 16th Conference on Computer and Robot Vision (CRV)*, IEEE, 2019, pp. 17–24.

[221] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit", in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[222] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks", *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[223] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: enabling low-power, highly-accurate deep neural network accelerators", in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 267–278.

[224] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing", *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[225] *Nvidia cusparse*, `https://developer.nvidia.com/cusparse`, Accessed: 2022-07-14.

[226] M. Ashby, C. Baaij, P. Baldwin, M. Bastiaan, O. Bunting, A. Cairncross, C. Chalmers, L. Corrigan, S. Davis, N. van Doorn, *et al.*, *Exploiting unstructured sparsity on next-generation datacenter hardware*, 2019.

[227] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network", *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[228] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: an accelerator for compressed-sparse convolutional neural networks", *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.

[229] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas, "Model pruning enables efficient federated learning on edge devices", *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[230] X. Qiu, J. Fernandez-Marques, P. P. Gusmao, Y. Gao, T. Parcollet, and N. D. Lane, "Zerofl: efficient on-device training for federated learning with local sparsity", *arXiv preprint arXiv:2208.02507*, 2022.

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Élagage de réseaux de neurones convolutifs et son application aux systèmes embarqués de vision par ordinateur

Mots clés : Vision par ordinateur, Apprentissage profond, compression, élagage

Résumé : À l'état de l'art dans de nombreux domaines tels que la vision par ordinateur, les réseaux à convolution sont devenus indispensables pour de nombreux types d'applications industrielles, comme la conception de véhicules autonomes – qui est l'une des ambitions de Stellantis. Toutefois, les réseaux de neurones peuvent présenter une grande complexité algorithmique, couplée à une importante empreinte mémoire, ce qui les rend potentiellement inutilisables sur le type de matériel embarqué que l'on peut trouver dans ces véhicules. Afin de réduire cette complexité, tout en conservant la performance d'origine le domaine de la compression de réseaux de neurones a proposé plusieurs types de méthodes, comme l'élagage qui vise à simplifier les réseaux en retirant des parties jugées inutiles. Cependant, derrière ce principe simple se cache en réalité de nombreuses considérations beaucoup plus subtiles ayant chacune de lourdes implications sur l'efficacité d'une telle méthode. Afin de mettre au clair toute la complexité insoupçonnée de l'élagage et de répondre à la question de son efficacité réelle, ce manuscrit aborde chaque aspect de la méthode de façon thématique et en discute à la fois les fondements théoriques et les conséquences pratiques. Il détaille également les implications académiques et industrielles de plusieurs contributions de cette thèse, portant notamment sur la suppression de paramètres, les interdépendances entre couches et l'efficacité énergétique des réseaux élagués.

Title : Convolutional Neural Networks Pruning and its Application to Embedded Vision Systems

Keywords : Computer Vision, Deep Learning, Compression, Pruning

Abstract : Being at the state of the art in many domains, such as computer vision, convolutional neural networks became a staple for many industrial applications, such as autonomous vehicles—about which Stellantis have ambitions. However, neural networks can bear a great algorithmic complexity, as well as a large memory footprint, which makes them potentially unusable on embedded hardware such as those equipped on such vehicles. In order to reduce this complexity, while keeping the performance that said complexity is supposed to enable, the domain of neural networks compression proposed multiple families of methods, such as pruning that aims at simplifying networks by removing parts deemed unnecessary. Yet, the apparent simplicity of this principle actually hides many subtle implications that have a decisive impact on the efficiency of pruning. In order to clarify the unsuspected complexity of this method and to answer the question of its true efficiency, this manuscript tackles thematically each aspect of pruning and discusses both its theoretical foundations and its practical consequences. It also details the academical and industrial implications of various original contributions of this thesis about parameters supression, layers interdependencies and the energetic efficiency of pruned networks.