Efficiency and Redundancy in Deep Learning Models: Theoretical Considerations and Practical Applications

Pierre Stock

Numéro National de Thèse : 2021LYSEN008

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée par
l'École Normale Supérieure de Lyon

École Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : **Informatique**

Soutenue publiquement le 09/04/2021, par :

Pierre STOCK

Efficiency and Redundancy in Deep Learning Models: Theoretical Considerations and Practical Applications

Efficience et redondance dans les modèles d'apprentissage profond : considérations théoriques et applications pratiques

---

Devant le jury composé de :

Raquel URTASUN, Uber ATG Chief Scientist, Université de Toronto	Rapporteure
François MALGOUYRES, Professeur des Universités, Université Paul Sabatier	Rapporteur
Julie DELON, Professeure des Universités, Université Paris Descartes	Examinatrice
Gabriel PEYRÉ, Directeur de Recherche, CNRS/DMA/ENS ULM	Examinateur
Aline ROUMY, Directrice de Recherche, INRIA	Examinatrice
Patrick PÉREZ, Scientific Director of Valeo AI	Examinateur
Rémi GRIBONVAL, Directeur de Recherche, INRIA/ENS de Lyon	Directeur de thèse
Hervé JÉGOU, Research Director, Facebook AI Research	Co-directeur de thèse

Abstract

Deep Neural Networks led to major breakthroughs in artificial intelligence. This unreasonable effectiveness is explained in part by a scaling-up in terms of computing power, available datasets and model size – the latter was achieved by building deeper and deeper networks. In this thesis, recognizing that such models are hard to comprehend and to train, we study the set of neural networks under the prism of their functional equivalence classes in order to group networks by orbits and to only manipulate one carefully selected representant. Based on these theoretical considerations, we propose a variant of the stochastic gradient descent (SGD) algorithm which amounts to inserting, between the SGD iterations, additional steps allowing us to select the representant of the current equivalence class that minimizes a certain energy. The redundancy of the network's parameters highlighted in the first part naturally leads to the question of the efficiency of such networks, hence to the question of their compression. We develop a novel method, iPQ, relying on vector quantization that drastically reduces the size of a network while preserving its accuracy. When combining iPQ with a new pre-conditioning technique called Quant-Noise that injects quantization noise in the network before its compression, we obtain state-of-the-art tradeoffs in terms of size/accuracy. Finally, willing to confront such algorithms to product constraints, we propose an application allowing anyone to make an ultra-low bandwidth video call that is deployed on-device and runs in real time.

Résumé

Les réseaux de neurones profonds sont à l'origine de percées majeures en intelligence artificielle. Ce succès s'explique en partie par un passage à l'échelle en termes de puissance de calcul, d'ensembles de données d'entrainement et de taille des modèles considérés – le dernier point ayant été rendu possible en construisant des réseaux de plus en plus profonds. Dans cette thèse, partant du constat que de tels modèles sont difficiles à appréhender et à entrainer, nous étudions l'ensemble des réseaux de neurones à travers leurs classes d'équivalence fonctionnelles, ce qui permet de les grouper par orbites et de ne manipuler qu'un représentant bien choisi. Ces considérations théoriques nous ont permis de proposer une variante de l'algorithme de descente de gradient stochastique qui consiste à insérer, au cours des itérations, des étapes permettant de choisir le représentant de la classe d'équivalence courante minimisant une certaine énergie. La redondance des paramètres de réseaux profonds de neurones mise en lumière dans ce premier volet amène naturellement à la question de l'efficience de tels réseaux, et donc de leur compression. Nous développons une nouvelle méthode de compression, appelée iPQ et reposant sur de la quantification vectorielle, prouvant qu'il est possible de réduire considérablement la taille d'un réseau tout en préservant sa capacité de prédiction. En combinant iPQ avec une procédure de pré-conditionnement appelée Quant-Noise qui consiste à injecter du bruit de quantification dans le réseau avant sa compression, nous obtenons des résultats état de l'art en termes de compromis taille/capacité de prédiction. Voulant confronter nos recherches à des contraintes de type produit, nous proposons enfin une application de ces algorithmes permettant un appel vidéo à très faible bande passante, déployée sur un téléphone portable et fonctionnant en temps réel.

Remerciements

Je tiens tout d'abord à remercier les deux rapporteurs, Raquel Urtasun et Francois Malgouyres, pour avoir pris le temps de relire ce manuscrit, pour leurs retours et leurs conseils. Merci également à l'ensemble des membres du jury pour leur lecture en profondeur du manuscrit et leurs questions. C'est un honneur de pouvoir soutenir ma thèse devant un panel de chercheurs aussi accomplis et renommés.

Merci à Hervé de m'avoir donné la chance de m'exprimer à travers le stage puis la thèse. La confiance immédiate et l'autonomie grisante que tu m'as accordés m'ont construit en tant que chercheur et m'ont indéniablement fait grandir. Un grand merci pour ton accompagnement et ton soutien décisif dans tous les moments clés de cette thèse, aussi bien au niveau scientifique, relationnel que personnel.

Merci à toi Rémi d'avoir soutenu mes velléités en toutes circonstances, qu'elles soient mathématiques ou plus appliquées. Merci d'avoir calmé mes doutes lorsqu'il le fallait et de m'avoir canalisé. Je retiendrai ta simplicité et ton accessibilité, ta volonté intransigeante de compréhension adossée à une conception du temps long en recherche et ta capacité à compiler, de tête, plusieurs pages de preuve.

Merci à Benjamin pour tes fulgurances qui ont bootstrapé ma thèse. Merci à Gabriel Peyré qui a été parmi les premiers à m'ouvrir les portes intrigantes et exaltantes de la recherche. Merci Angela d'avoir bien voulu former un duo de compression avec moi, sous l'égide d'Armand, Hervé et Edouard. Merci à Moustapha Cissé pour m'avoir montré qu'il était possible d'écrire un article en un temps record. Merci à Matthijs pour ta disponibilité et ton aide précieuse sur l'implémentation de kernels. Merci à la team FaceGen de m'avoir accueilli avec bienveillance il y a plus d'un an et demi et de m'avoir donné l'opportunité de construire une première démo. Merci à toi Maxime, j'ai énormément appris lors de nos interactions quasi-quotidiennes et je retiens avant tout ta confiance et ton écoute. Merci à Camille, Daniel et Onur : vous vous reconnaitrez sans doute dans quelques illustrations de cette thèse.

Merci à Facebook et à toute l'équipe de FAIR Paris. Merci au PhD Squad et en particulier à Neil, Pauline, Alexis, Guillaume, Timothée, Alexandre, Alexandre, Léonard, Nicolas, Louis Martin et les autres pour leur soutien par temps de deadline, pour les parties endiablées de baby-foot agrémentées de gamelles et autres snakes, et pour les moments de respiration hors du temps à Yosemite, en offsite ou en conférence. Merci à Pauline Luc de m'avoir accompagné pour ma première conférence au pays des *biergarten*. Merci à Timothée Lacroix et Alexandre Sablayrolles pour nos escapades enseignantes et culinaires au Rwanda. Merci à Ezéchiel et Clément pour nos discus-

sions lyriques sur la science en général. Merci à Marianne d'avoir fait irruption au début de ma thèse, merci pour ta joie de vivre et pour tous nos gouters sur le rooftop qui me manquent déjà. Merci enfin à Yana pour ton amitié indéfectible, pour toutes nos discussions et nos sessions de télétravail agrémentées de crêpes.

Merci à tous mes professeurs de mathématiques et de physique du lycée et de prépa pour m'avoir montré ce que la raison est capable de construire et pour m'avoir transmis la passion des sciences. Mon parcours est indissociable de ces rencontres et cette thèse n'aurait pas existé sans votre enseignement.

Merci à ma famille et en particulier à mes parents, qui m'ont tout donné pour que je m'épanouisse. Merci à Clara et Arthur pour leur soutien et pour leur écoute patiente lorsque je leur résumais un article. Merci à Margaux d'avoir illuminé ma thèse et d'avoir partagé mes joies et mes peines, tu es mon ancrage dans ce monde.

Une pensée particulière pour ma grand-mère maternelle, qui voulait que je sois préfet, et à qui je dédie cette thèse.

*

* *

"Crois et tu comprendras ; la foi précède, l'intelligence suit." Saint Augustin.

Contents

Notation

We first define a neural network using the formalism of graph theory to disentangle the architecture from the values of the weights. Unless mention of the contrary, we consider networks with ReLU non-linearities defined as $\sigma(x) = \max(x, 0)$ for $x \in \mathbb{R}$.

Architecture

A neural network architecture can be represented as a particular directed acyclic graph $G = (V, E)$. We denote a *neuron* by $\nu \in V$ and a *connection* by $e = \nu \to \mu \in E$. Each neuron ν belongs to a *layer* $\ell(\nu) \in [\![0, L]\!]$.

- If $\ell(\nu) = 0$ then ν belongs to the *input layer*.
- If $0 < \ell(\nu) < L$ then ν belongs to one of the $L - 1$ *hidden layers*.
- If $\ell(\nu) = L$ then ν belongs to the *output layer*.

G is such that two connected neurons necessarily belong to consecutive layers. We denote $H \subset V$ the set of all hidden neurons. Note that two neurons μ and ν in consecutive layers may be connected or not. We denote the neurons of layer ℓ by $N_\ell \triangleq \{\nu \mid \ell(\nu) = \ell\}$.

A *full path* p is a sequence of connected neurons $p = (\nu_0, \dots, \nu_L)$ where $\nu_\ell \in N_\ell$, $0 \leq \ell \leq L$ and $\nu_{\ell-1} \to \nu_\ell \in E$ for $1 \leq \ell \leq L$. We say that a connection e belongs to $p = (\nu_0, \dots, \nu_L)$ if there exists ℓ such that $e = \nu_{\ell-1} \to \nu_\ell$. We may write $p \cap H$ to denote the hidden neurons $(\nu_1, \dots, \nu_{L-1})$ belonging to the path p. We denote by $\mathcal{P}(G)$ the set of all full paths connecting some input neuron to some output neuron. We also define a *partial path* $q = (\nu_\ell, \dots, \nu_L)$ as a sequence of connected neurons between any hidden layer ℓ with $0 \leq \ell \leq L$ and the output layer. We finally denote by *path segment* a sequence of connected neurons $(\nu_\ell, \dots, \nu_{\ell'})$ where $0 \leq \ell \leq L$ and $0 \leq \ell' \leq L$. We denote $\mathcal{Q}(G)$ the set of such partial paths and we have $\mathcal{P}(G) \subset \mathcal{Q}(G)$. We may omit the dependency of the underlying graph G when it is obvious and simply write $\mathcal{P}$ and $\mathcal{Q}$. We may simply denote full paths by *paths* when the context is clear.

For any neuron ν, we define

$$\text{prev}(\nu) \triangleq \{\mu \in V \mid \mu \to \nu \in E\} \tag{1}$$

$$\text{next}(\nu) \triangleq \{\mu \in V \mid \nu \to \mu \in E\} \tag{2}$$

and for a set of neurons V, $\text{prev}(V) = \cup_{\nu \in V} \text{prev}(\nu)$. We denote by $\text{Parents}(\nu)$ the set of all parent neurons of ν

$$\text{Parents}(\nu) \triangleq \bigcup_{\ell} \text{prev}^{\ell}(\{\nu\}) \tag{3}$$

where prev^{ℓ} denotes the composition of the operator prev with itself. We similarly define $\text{Children}(\nu)$ the set of all children neurons of ν. We also introduce the notation $\bullet \to \nu$ to denote any edge $e \in E$ that has the form $\mu \to \nu$ for some $\mu \in \text{prev}(\nu)$ and similarly for the notation $\nu \to \bullet$.

Weights and Biases

Throughout the manuscript, we will manipulate quantities involving the weights and biases of the network, and find it cleaner to index them all using the connections of the network or its neurons. The graph $G = (V, E)$ is valued with the *weights* of the network. The weights can be represented:

- At the *connection level* by $w_e \in \mathbb{R}$ with $e \in E$;
- At the *layer level* by $W^{(\ell)} \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$ where $W^{(\ell)} = (w_{\nu \to \mu})_{\nu, \mu \in N_{\ell-1} \times N_\ell}$ and $\ell \in [\![1, L]\!]$;
- At the *network level* by $w \in \mathbb{R}^E$.

In order to define weights at the layer level, we write by convention $w_{\nu \to \mu} = 0$ if $\nu \to \mu \notin E$. Besides, given a fixed architecture G, we allow the case where some weights $w_e, e \in E$ are zero. Similarly to the weights, the biases can be represented:

- At the *neuron level* by $b_\nu \in \mathbb{R}$ for any hidden neuron $\nu \in H \cup N_L$;
- At the *layer level* by $b^{(\ell)} \in \mathbb{R}^{N_\ell}$ where $b^{(\ell)} = (b_\nu)_{\nu \in N_\ell}$ and $\ell \in [\![1, L]\!]$;
- At the network level by $b \in \mathbb{R}^{H \cup N_L}$.

We denote the global network parameterization by $\theta = (w, b)$ and refer to elements of θ as *parameters* of the network. Networks with at least one hidden layer are such that $L \geq 2$. The case $L = 1$ corresponds to a linear layer without any non-linearity. Note that a network has L affine layers and $L - 1$ hidden layers. Finally, we define useful support and sign sets as follows:

$$\begin{aligned} \text{supp}(w) &\triangleq \{e \in E \mid w_e \neq 0\} \subset E \\ \text{supp}(b) &\triangleq \{\nu \in H \cap N_L \mid b_\nu \neq 0\} \subset H \cap N_L \\ \text{supp}(\theta) &\triangleq \text{supp}(w) \cup \text{supp}(b) \subset E \cup (H \cap N_L) \end{aligned}$$

We further define the extended sign operator as follows. For $x \in \mathbb{R}, \text{sign}(x) = 1$ if $x > 0$, $\text{sign}(x) = 0$ if $x = 0$ and $\text{sign}(x) = -1$ if $x < 0$. When applied to a vector or a matrix, sign is taken pointwise. We finally define

$$\text{Sign}_w \triangleq \{w' \in \mathbb{R}^E \mid \text{sign}(w') = \text{sign}(w)\}$$
$$\text{Supp}_w \triangleq \{w' \in \mathbb{R}^E \mid \text{supp}(w') \subseteq \text{supp}(w)\}$$

Similarly, we define Sign_b and Supp_b and denote $\text{Sign}_\theta \triangleq \text{Sign}_w \times \text{Sign}_b$ and we denote $\text{Supp}_\theta \triangleq \text{Supp}_w \times \text{Supp}_b$

Function

We will also need to manipulate the output function or intermediary functions implemented by the network. Let G be a fixed architecture G valued with θ. Recall that we denote by σ the ReLU non-linearity. We define:

- *Layer-wise functions.* A neural network can be recursively implemented using intermediary row vector functions $y^{(\ell)}(\theta) : \mathbb{R}^{N_0} \to \mathbb{R}^{N_\ell}$ for $\ell \in [\![0, L]\!]$. We define $y^{(0)}(\theta, x) = x$ and, for $\ell \in [\![1, L-1]\!]$,

$$y^{(\ell)}(\theta, x) = \sigma\left(y^{(\ell-1)}(x)W^{(\ell)} + b^{(\ell)}\right). \tag{4}$$

- *Output functions.* The function implemented by the whole network is

$$y^{(L)}(\theta, x) = y^{(L-1)}(\theta, x)W^{(L)} + b^{(L)} \tag{5}$$

  i.e. the last layer is an affine function of the previous one. We use the notation $R_{G|\theta} = y^{(L)}(\theta)$ and we call $R_{G|\theta}$ the realization of the network architecture G given the parameters θ. We write $R_{G|\theta}(x)$ to denote the evaluation of the defined function at any input $x \in \mathbb{R}^{N_0}$. When the dependency on the graph G is obvious, we may simply write R_θ and $R_\theta(x)$. With a slight abuse of notation and for the sake of clarity, we may also write $R(\theta, x)$.

- *Neuron functions.* Given a neuron ν belonging to layer ℓ, we denote the function implemented by ν *before* the non-linearity as $y_\nu(\theta) : \mathbb{R}^{N_0} \to \mathbb{R}$ such that

$$y_\nu(\theta) = y_\nu^{(\ell)}(\theta). \tag{6}$$

Given a fixed architecture G valued with respective weights θ or θ', we say that θ

and θ' are *functionally equivalent* if the realizations R_θ and $R_{\theta'}$ are the same, *i.e.*, if for all $x \in R^{N_0}$, $R_\theta(x) = R_{\theta'}(x)$.

Useful Quantities

We denote the *value* of a full path by $v_p(\theta) = w_{\nu_0 \to \nu_1} \dots w_{\nu_{L-1} \to \nu_L}$ and we define the activation status of a full path p given the parameters θ and the input x as

$$a_p(\theta, x) \triangleq \prod_{\nu \in p \cap H} \mathbf{1}(y_\nu(\theta, x) > 0). \tag{7}$$

We naturally extend the notion of partial path value and the notion of partial path activation status. As the value of a full or partial path only depends on the weights w and not on the biases, we may write indifferently $v_p(\theta)$ or $v_p(w)$ for any full or partial path. For any path segments $q = (\nu_\ell, \dots, \nu_{\ell'})$, $q' = (\mu_{\ell'}, \dots, \mu_{\ell''})$ such that $\nu_{\ell'} = \mu_{\ell'}$, we denote the concatenation of q and q' as $q + q' = (\nu_\ell, \dots, \nu_{\ell'}, \mu_{\ell'+1}, \dots, \mu_{\ell''})$.

Algebraic Tools

We will rely on algebraic and geometric interpretation to understand the action of rescaling operations on a parameterization θ. To this end, we represent the mapping between edges and paths by the linear operator $\mathbf{P} : \mathbb{R}^E \to \mathbb{R}^{\mathcal{P}}$ such that for every connection $e \in E$ and every path $p \in \mathcal{P}$,

$$\left(\mathbf{P}\delta_e\right)_p \triangleq \begin{cases} 1 & \text{if } e \in p \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

where $\delta_e \in \mathbb{R}^E$ is the dirac vector for edge e. We denote by $\mathcal{D}(N)$ the set of diagonal matrices $D \in \mathbb{R}^{N \times N}$ such that, for all i, $d_i = D_{i,i}$ is strictly positive.

Admissible parameterizations

We say that the network parameterization $\theta = (w, b)$ is admissible if, for every hidden neuron $\nu \in H$, there exists a full path $p \in \mathcal{P}$ going through ν such that $v_p(\theta) > 0$. Equivalently, every hidden neuron $\nu \in H$ is connected to some input and some output neuron through a path with non-zero weights. As the notion of admissibility only depends on the weights w and not on the biases b, we may indifferently mention an admissible parameterization or admissible weights.

Chapter 1

Introduction

Computer Science has shaped our modern society in a tremendous way, in part since the seminal work of Alan Turing, who invented an abstract computer called the *Turing Machine* in 1936. Since then, this concept has materialized in the form of processors and chips with an extremely wide range of applications. In a fast-paced search for performance, the building blocks of modern computers called *transistors* have become smaller and smaller according to Moore's law, which states that the number of transistors on a chip doubles after a short, constant period of time[1]. Thanks to this miniaturization, the ubiquity of interconnected portable computers, prophesied by Silicon Valley entrepreneurs (Gates and Ottavino, 1995) or even by french writer Marguerite Duras[2] in 1985 is now a reality.

Such powerful and portable devices, including smartphones or virtual/augmented reality headsets, constitute a fertile ground for a particular class of algorithms called *Deep Neural Networks* (DNNs). These models are programmed to *learn* to perform specific tasks – hence the name *Deep Learning* – and belong to the more general field of Artificial Intelligence, also pioneered by Turing[3]. While DNNs are increasingly powerful for detecting persons in images or understanding and translating speech or text for instance, they still lack efficiency in terms of size and speed.

Hence, after the miniaturization of the computers themselves, the miniaturization or *compression* of DNNs is now a key challenge to deploy them on-device and in real

[1]Every 18 or 24 months according to a majority of the estimates. However, it is uncertain that this empirical law will hold in the future: with a characteristic scale of 5 nanometers of 2020 down to 3 and 2 nanometers in the next years, the transistors are now so small that they begin to experience quantum tunneling effects perturbing their functioning.

[2]`https://www.ina.fr/video/I04275518`, television interview in French by Michel Drucker.

[3]The Turing Award, usually considered as the highest distinction in Computer Science, was awarded to Yoshua Bengio, Geoffrey Hinton, and Yann LeCun in 2018 for their "conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing".

time. It would offload the servers that currently run such models, reduce the latency and promote more privacy since personal data would be analyzed directly on-device. This is particularly relevant after the European Union's new data privacy and security law entered into force in 2016, the General Data Protection Regulation (GDPR)[4].

In this Chapter, we first detail the general topics of efficiency and redundancy in deep learning and the underlying reasons motivating our work in Section 1.1. We then briefly summarize the interesting challenges that the Deep Learning community is facing on these areas in Section 1.2 and finally detail our contributions, both academic and product-oriented and the general organization of the manuscript in Section 1.3.

1.1 Motivation

We summarize here the main reasons that conducted us to study the redundancy and efficiency of Neural Networks both theoretically and in practice.

The Deep Learning Revolution. Following pioneering work by Rumelhart et al. (1986), LeCun et al. (1989) or (LeCun et al., 1998b), the inception of AlexNet by Krizhevsky et al. (2012) marked a turning point in the development of Deep Learning. Back in 2012, a heavy neural network, trained with stochastic gradient descent (SGD) and properly regularized, surpassed all existing methods by a large margin on the ImageNet competition (Deng et al., 2009). Since then, the concept of "ImageNet moment" was transposed to various domains where Deep Learning percolated with an unreasonable effectiveness, from Natural Language Processing (Ruder, 2019) a few years ago to protein folding as we write these lines (Jumper et al., 2020). Strong empirical evidence of deep learning approaches was also shown for extremely various tasks such as symbolic mathematics (Lample and Charton, 2019), fast MRI reconstruction (Zbontar et al., 2018) or quantum physics (Schütt et al., 2017).

Bigger, Hence Better Models. Fueled by this tremendous success on numerous tasks, a fast-paced search for performance is occurring in the research community. Since one straightforward way to improve expressivity – hence performance – is to act on depth by stacking more layers (Telgarsky, 2016; Raghu et al., 2017), researchers are considering the biggest possible networks they can train (Simonyan and Zisserman, 2014; He et al., 2015b), now up to 175 billion parameters (Rajbhandari et al., 2020;

[4] `https://gdpr-info.eu/`. It is a substantial update of the European Data Protection Directive passed in 1995 by the EU.

Rasley et al., 2020; Brown et al., 2020). Training such networks requires huge amounts of data (labelled or not), energy and performant distributed computing infrastructure relying on GPGPUs or TPUs[5].

Redundancy and Efficiency in Deep Learning Models. Then, as the research community produces bigger networks, the question of their redundancy and efficiency naturally arises in order to manipulate such networks more easily. Roughly speaking, *redundancy*[6] refers to a certain structure in the parameters of a neural network where some weights or groups of weights carry similar information (LeCun et al., 1990; Denil et al., 2013). On the other hand, *efficiency*[7] refers to Pareto efficiency (Fudenberg and Tirole, 1991) of a model given set of metrics[8], traditionally in terms of model size, model accuracy and inference time (Wang et al., 2018a). Thus, studying redundancy in deep learning models may help improve their efficiency, at least on the model size axis. Note that various setups – especially in terms of hardware – and constraints on these metrics may lead to distinct Pareto optima.

Making the Best Models Available to Everyone. Studying redundancy in deep learning to produce more efficient models is thus crucial for deploying the best models both on servers and on mobile devices. On the server side, the objective is to produce more parsimonious models in terms of parameters (Radosavovic et al., 2020; Tan and Le, 2019) or training data (Touvron et al., 2020), which may lead to a faster training time[9] , energy savings, or help consider even bigger models to train. On the mobile side, deploying models on embedded devices such as smartphones, autonomous vehicles or virtual/augmented reality headsets[10] opens up to numerous applications. Having such models on-device instead of performing the inference on a remote server reduces the latency and the network congestion, works offline and is compatible with privacy-preserving machine learning where the data stays on the user's device (Knott et al., 2020), at the cost of exposing the embedded model to various attacks. It also enables federated learning (Konečný et al., 2017) where a centralized model is trained while training data remains distributed over a large number of client devices with unreliable network connections.

[5]Respectively General Purpose Graphics Processing Units and Tensor Processing Units.
[6]More details in Sections 2.1 and 2.2.
[7]More details in Section 2.3.
[8]Such metrics are not independent: larger models are generally more accurate but slower.
[9]For instance, AlphaGo (Silver et al., 2017) took 40 days to train on a vast infrastructure.
[10]Such as Oculus Quest 2 for VR (Virtual Reality) or HoloLens 2 for AR (Augmented Reality).

1.2 Challenges

Redundancy is intimately related to the concept of over-parameterization, which is a key characteristic of modern neural networks (Srivastava et al., 2014) – see Subsection 2.1.1 for definitions and discussions. Over-parameterized networks have a high ability to fit training data, yet they are challenging to train, they are difficult to comprehend from a theoretical point of view and they pose significant hurdles to real-time applications on embedded devices. We briefly summarize these challenges in this Section and refer the reader to Chapter 2 for a more extensive discussion.

Theoretical Considerations on Redundancy. Parameter redundancy of deep learning models is a well-known fact (Denil et al., 2013) and has clear drawbacks – first and foremost, the model size. However, reducing this redundancy is challenging and indirectly provides hints on the *benefits* of redundant, over-parameterized models. At training time, researchers study the interplay between over-parameterization and SGD (Li and Liang, 2018; Sankararaman et al., 2019) to help mitigate overfitting. Taking advantage of this redundancy may lead to more efficient or performant training procedures. For instance, grouping networks that behave similarly and implicitly performing SGD in a reduced or quotiented space may help, see Chapters 3 and 4.

Redundancy in Practice for Efficient DNNs. While training happens once, the trained model is subsequently used numerous times for inference. For instance, at Facebook, deep learning models analyze trillions of bits of content per day. Then, the challenge is to compress the network – or more generally, to make it more efficient – without losing too much predictive performance or accuracy, sometimes referred to as the size/accuracy tradeoff, see Chapters 5 and 6. Another challenge is to select the best compression algorithm – or combination thereof – among a large set of methods that are not entirely orthogonal, given a target size/accuracy tradeoff. For instance, is it better to compress a large, high-performing network instead of a mobile-efficient architecture that is smaller but has a slightly degraded predictive performance?

On-device and Real Time Deployment. While less redundant networks are generally faster at inference, redundancy is only a part of the story. Indeed, compression algorithms with an excellent compression ratio could necessitate to *decompress* the network before inference instead of performing the prediction in the compressed domain. Hence, a good compression algorithm also depends on the task and hardware constraints – generally, inference has to be performed in real-time without draining

the battery and overloading the RAM[11], see Chapter 7. This involves low-level considerations on the type of hardware used for inference, see Subsection 2.3.6 for details.

1.3 Contributions

We detail the general organization of the manuscript and then enumerate our academic contributions as well as the applications and technology transfers derived directly from our published papers.

1.3.1 Outline

We present here our contributions in ascending order of applicability, measured as the closeness to production. We first start by presenting theoretical contributions on functional equivalence classes, then explore the compression of deep learning models down to the on-device deployment of such efficient models.

Equivalence Classes of Neural Networks. In Chapter 3, we study functional equivalence classes of ReLU neural networks. We first show that such classes contain orbits generated by the action of rescalings and permutations of hidden neurons for networks with arbitrary depth. We then characterize functional equivalence classes for one-hidden-layer networks under some *non-degeneracy* conditions and investigate the case with many hidden layers by designing algebraic tools to study the problem locally. Leveraging these theoretical considerations, we develop an alternative to the Stochastic Gradient Descent (SGD) algorithm in Chapter 4. Our variant, called Equi-Normalization or ENorm, alternates between standard SGD steps and *balancing* steps amounting to change the representant of the current functional equivalence class by selecting the one that minimizes a given energy function. Balancing steps preserve the output – hence the accuracy – of the network by definition but modify the gradients of the next SGD step, hence the learning trajectory. In other words, ENorm takes advantage of the redundancy in the parameter space by operating the optimization in the quotient space induced by the functional equivalence relation.

Neural Network Compression. Studying the redundancy of the network's parameters in the first part of the thesis naturally leads to the question of the efficiency

[11]Random Access Memory or RAM is generally used to rapidly store and retrieve working data.

of such networks, hence to the question of their compression. In Chapter 5, we develop a novel method, called Iterative Product Quantization or iPQ, that relies on vector quantization in order to drastically reduce the size of a network while almost preserving its accuracy. The proposed approach iteratively quantizes the layers and then finetunes them. The quantization step is performed by splitting the layer's weight matrix into a set of vectors that are clustered into a common codebook. In order to boost the obtained size/accuracy tradeoffs, we develop a pre-conditioning method that injects carefully selected quantization noise when training the network *before* its compression. The method, called Quant-Noise, is described in Chapter 6 and has proven to be effective for both iPQ and for traditional scalar quantization such as `int8` or `int4`. Quant-Noise is effective for a variety of tasks and quantization methods and thus reconciles pre-training for both scalar and vector quantization.

Ultra-Low Bandwidth Generative Video Chat. While the compressed size of the network is a significant indicator of the quality of the quantization, other metrics such as inference time and battery usage are also relevant, especially for on-device, real time applications. Hence, we investigated potential applications for iPQ and Quant-Noise. Among them, we design a method, called FaceGen, to perform ultra-low bandwidth video chats in Chapter 7. FaceGen streams compressed facial landmarks from the sender's phone to the receiver, and uses a generative adversarial network to reconstruct the sender's face based on the stream of landmarks plus one identity embedding sent once at the beginning of the call. The stream of landmarks is compressed to less than 10 kbits/s, and the networks are quantized to a total size of less than 2 MB and run at 20+ frames per second on an iPhone 8.

1.3.2 Publications

The work presented in this manuscript was also published in the following papers, that were written during the thesis.

- **Pierre Stock**, Benjamin Graham, Rémi Gribonval and Hervé Jégou. Equi-normalization of Neural Networks. *Published at ICLR 2019* (Stock et al., 2019a). Source code: `https://github.com/facebookresearch/enorm`.
- **Pierre Stock**, Armand Joulin, Rémi Gribonval, Benjamin Graham and Hervé Jégou. And the Bit Goes Down: Revisiting the Quantization of Neural Networks. *Published at ICLR 2020* (Stock et al., 2019b). Source code: `https://github.com/facebookresearch/kill-the-bits`.

- **Pierre Stock***, Angela Fan*, Benjamin Graham, Edouard Grave, Rémi Gribonval, Hervé Jégou and Armand Joulin. Training with Quantization Noise for Extreme Model Compression. *Published at ICLR 2021* (Stock et al., 2020). Source code: `https://github.com/pytorch/fairseq/tree/master/examples/quant_noise`

- Maxime Oquab*, **Pierre Stock***, Oran Gafni, Daniel Haziza, Tao Xu, Peizhao Zhang, Onur Celebi, Yana Hasson, Patrick Labatut, Bobo Bose-Kolanu, Thibault Peyronel, Camille Couprie. Low Bandwidth Video-Chat Compression using Deep Generative Models. *Under review, 2021* (Oquab et al., 2020).

The following paper was written during an internship at Facebook AI Research and will not be discussed in this manuscript.

- **Pierre Stock**, Moustapha Cisse. ConvNets and ImageNet Beyond Accuracy: Explanations, Bias Detection, Adversarial Examples and Model Criticism. *Published at ECCV 2018* (Stock and Cisse, 2018).

1.3.3 Technology Transfers

The research presented in this manuscript led to various technology transfers, which are briefly summarized here. Seeking for internal applications and collaborations and delivering product impact was the main focus of the last year of my PhD.

- The iPQ technique described in Chapter 5 is currently used to design fast CPU kernels for quantized models for both server side and mobile side, relying on the `fbgemm`[12] library for quantized matrix multiplication.

- The pre-conditioning technique Quant-Noise (Chapter 6) was used to deploy a quantized `int8` model for internal dogfooding. The model aims at detecting harmful content in conversations on-device on real time.

- The low-bandwidth generative video chat method described in Chapter 7 is currently under productionization and led to a US patent application P201451US00.

[12] `https://engineering.fb.com/2018/11/07/ml-applications/fbgemm/`

Chapter 2

Related Work

In this Chapter, we review the lines of work addressing the question of parameter redundancy and efficiency in Deep Learning. In Section 2.1, we first discuss the benefits of depth for neural networks in terms of expressivity and capacity to fit training data. We enumerate the main practical training challenges along with the methods and tools to mitigate them, in particular in terms of normalization layers. This line of work is related to our contribution in Chapter 4, where we re-normalize the network's weights after each training step while preserving the function implemented by the network. Then, we review the theoretical studies aiming at characterizing functional equivalence classes of neural networks in Section 2.2. Such equivalence classes allow to aggregate networks that behave identically to only manipulate one representant per class, thus effectively operating in the quotient space. This work is related to our contribution detailed in Chapter 3 and shows that under some assumptions, the equivalence classes only encompass permutations and rescalings of neurons (see 2.2.1). Finally, we study network compression in Section 2.3. We review the methods aiming at reducing the redundancy in the set of the network's parameters while maintaining a competitive accuracy and inference speed, in particular scalar and vector quantization. This section is related to our contributions on Iterative Product Quantization and Quantization Noise that are detailed in Chapters 5 and 6. More specifically, the subsection about on-device deployment is related to our Ultra-low Bandwidth Generative Video Chat contribution detailed in Chapter 7.

2.1 Over-Parameterization: a Double-Edged Sword

The inception of AlexNet (Krizhevsky et al., 2012) demonstrated that a deep neural network surpassed existing computer vision techniques by a good margin[1] on the ImageNet classification task (Deng et al., 2009). This success is conditioned on proper training techniques, the availability of large datasets, as well as huge computing capabilities. Since then, a part of the research community has focused on scaling the architectures, the datasets and the training techniques in the search of better performance (Brown et al., 2020). This fast-paced practical search for deeper networks, followed by a more theoretical analysis of the benefits of depth, is reviewed in Subsection 2.1.1. Then, we briefly enumerate the training challenges posed by such networks in Subsection 2.1.2 along with the tools to mitigate them in Subsection 2.1.3, including various normalization layers. Subsections 2.1.1 and 2.1.2 do not aim to be exhaustive but act rather as motivating illustrations for the remainder of this chapter.

2.1.1 With Greater Depth Comes Greater Expressivity

We briefly review and discuss the notion of over-parameterization, followed by a more theoretical analysis of the benefits of depth in terms of expressivity. Here, we do not aim at a comprehensive review but rather focus on a few illustrative examples.

Over-parameterization

Over-parameterized networks are primarily characterized by their large number of parameters[2] with respect to the number of training samples in the Deep Learning literature (Sagun et al., 2018; Allen-Zhu et al., 2019; Li and Liang, 2018). For instance, AlexNet has 60 million parameters, which is an order of magnitude larger than 1.2 million train images of ImageNet (Deng et al., 2009). Note that a more rigorous definition would take into account various factors such as the sample size[3], the architecture G or even the data itself (Sagun et al., 2018).

Next, we briefly focus on a few illustrative examples acknowledging the parameter redundancy in neural networks in practice. First, the fact that some parameters can

[1]The ILSVRC-2012 test top-5 error rate was 15.3% for AlexNet compared to 26.2% for the second best entry of the competition.

[2]The number of parameters of a neural network with architecture G is the number of connections in G (except for the biases). Sometimes authors consider the number of neurons rather than the number of connections (Gribonval et al., 2019).

[3]Fitting N training samples in a low dimension input space would require less parameters than fitting N training images of standard size $3 \times 224 \times 224$ for instance.

be deleted or *pruned* without harming the accuracy is well-known (LeCun et al., 1990), as discussed in Subsection 2.3.1. More recently, Denil et al. (2013) demonstrate that there is a significant redundancy in the network parameters of several deep learning models by accurately predicting 95% of the parameters based only on the remaining 5%, with a minor drop in accuracy. The motivation behind this technique is the fact that the first layer features of a convolutional neural network trained in natural images (*e.g.* ImageNet (Deng et al., 2009)) tend to be locally smooth with local edge features, similar to local Gabor filters[4]. Given this structure, representing the value of each pixel separately is redundant as the value of one pixel is highly correlated with its neighbors. Denil et al. (2013) propose to take advantage of this type of structure to factor the weight matrix. Similar approaches that learn a basis of low-rank filters are explored by Jaderberg et al. (2014).

Depth and Expressivity

The relation between the network's depth and its expressivity is widely studied (Pascanu et al., 2013; Montúfar et al., 2014; Eldan and Shamir, 2015; Telgarsky, 2016; Raghu et al., 2017; Gribonval et al., 2019). For instance, Montúfar et al. (2014) study the number of linear regions defined by a given architecture. As defined more formally in Subsection 2.2.2 in the case of ReLU networks, linear regions are the areas of the input space on which the gradient of the function implemented by the network $\nabla_x R_\theta$ is constant. The number of linear regions is connected to the complexity of the function implemented by the network: more linear regions means that the network is able to fit more complex training data. The authors derive a lower bound on the maximal number of linear regions and show in particular that, for architectures with fixed widths, the maximal number of linear regions – hence the expressivity of the network – grows exponentially in the number of layers.

While the expressivity of a network somehow grows exponentially with its depth, another approach to obtain a network with the same number of parameters is to increase its width when keeping the number of layers fixed. This strategy generally results in less expressive networks, as shown by Eldan and Shamir (2015). The authors exhibit a simple radial function φ in $\mathbb{R}^d$ that is implementable by a two-hidden-layer network but cannot be tightly approximated by any one-hidden-layer network, unless its width grows exponentially in the input dimension d. The authors conclude that "depth can be exponentially more valuable than width" for feedforward networks.

[4]See Figure 10 of Krizhevsky et al. (2012).

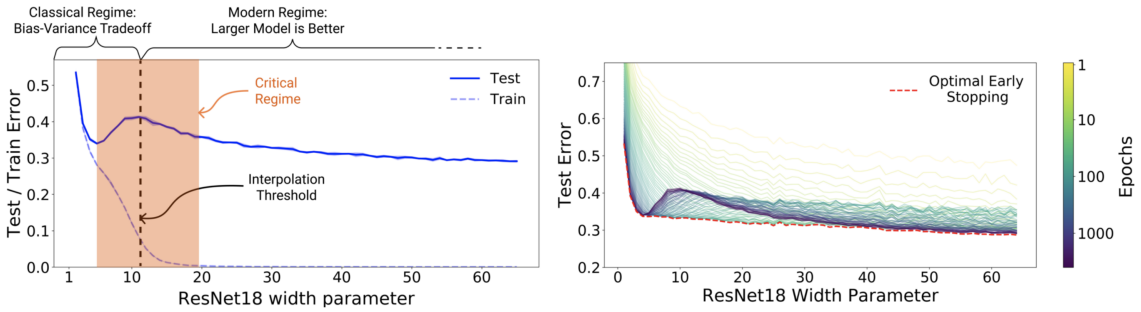


Figure 2-1: The double-descent phenomenon in Deep Learning as illustrated by Nakkiran et al. (2019). Here the complexity of the considered model (a ResNet-18) is measured as its width parameter, where higher width denotes a network with a larger number of parameters.

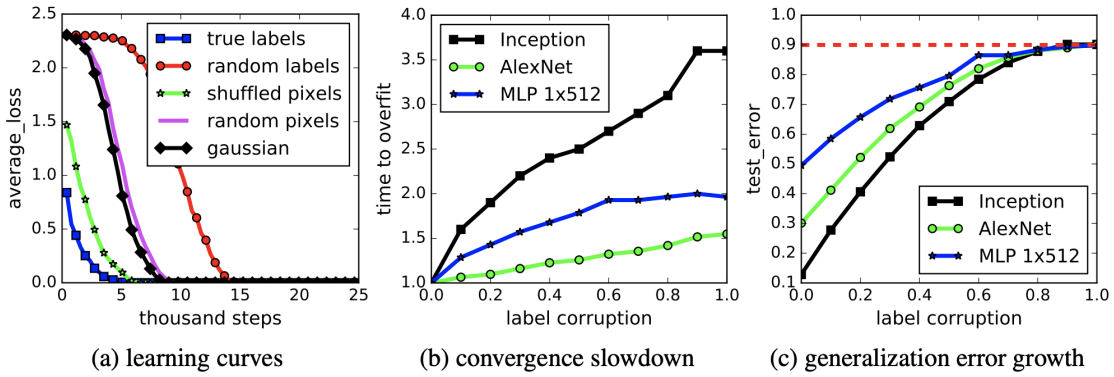
2.1.2 Deeper Networks Present Harder Training Challenges

In this Subsection, we briefly review the main training challenges of neural networks that arise as the networks go deeper in the search for more and more expressivity, as mentioned in Subsection 2.1.1. We refer the reader to the cited work below for a more exhaustive overview.

Overfitting

Overfitting is traditionally approached in Machine Learning through the Bias-Variance trade-off, as explained for instance by Hastie et al. (2009). According to this principle, as the model complexity – measured with its number of parameters or with more elaborate tools such as the VC dimension (Vapnik, 1998) or the Rademacher complexity (Bartlett and Mendelson, 2003) – increases, the training error decreases while the test error follows a U-shaped curve. Models with low complexity underfit and suffer from high bias whereas models with high complexity exhibit a high variance, suggesting that an intermediate model complexity reaches the optimal trade-off. However, recent work by Belkin et al. (2018), followed by Nakkiran et al. (2019); Mei and Montanari (2019); d'Ascoli et al. (2020) uncovered a *double-descent* behavior for deep learning models. After a critical regime, the test error goes down again as illustrated in Figure 2-1. This finding is consistent across architectures, optimizers and tasks (Nakkiran et al., 2019), suggesting, as found in practice, that deeper models and more data lead to better performance (Krizhevsky et al., 2012).

As stated before, given a fixed training dataset, overfitting is traditionally measured using the number of parameters of the considered model. In the search for more elaborate model complexity measures, Zhang et al. (2016) propose a protocol



(a) learning curves (b) convergence slowdown (c) generalization error growth

Figure 2-2: Fitting random labels and random pixels on CIFAR10 as illustrated by Zhang et al. (2016).

for understanding the effective capacity of machine learning models. The authors train several architectures on a copy of the training data where either (1) the original labels were replaced by random labels; or (2) the pixels were shuffled; or (3) the pixels were drawn randomly from gaussian noise. They observe that the training error goes down to zero provided that the number of epochs is large enough, as depicted in Figure 2-2, and conclude that "deep neural networks easily fit random labels". Or course, if the amount of randomization is larger, the time taken to overfit is longer. Such work paves the way for more formal complexity measures to explain the generalization ability of neural networks.

Vanishing and Exploding Gradients

In the 90's, feedforward neural networks – convolutional or fully connected – were a few layers deep (LeCun et al., 1989) and were trained through back-propagation. However, Bengio et al. (1994) reported difficulties to train *recurrent* networks in order to learn long-term dependencies – for instance in a sequence of words. The principle of recurrent networks is to apply the same weight matrix iteratively on the input sequence while updating a hidden state. Recurrent networks are usually trained with back-propagation through time (BPTT, Williams and Zipser (1995); Rumelhart et al. (1986); Werbos (1988)), where the network is *unfolded* for a fixed sequence length and trained using standard back-propagation. When working on an improved version of recurrent networks named LSTMs (Long Short-Term Memory networks, Hochreiter and Schmidhuber (1997)), Hochreiter and Bengio (2001) identify two undesirable behaviors of the gradients flowing backward in time: such gradients either blow up or vanish, resulting in training instabilities.

As feedforward networks went deeper, similar vanishing or exploding gradient problems arise Mishkin and Matas (2016) He et al. (2015c). Such training instabilities were related in part to the magnitude of the weights. The research community proposed tools to mitigate this undesirable training behavior as explained in Subsection 2.1.3. Failure modes preventing the training from properly starting were also theoretically studied by Hanin (2018).

2.1.3 The Deep Learning Training Toolbox

The training challenges of deep neural networks mentioned in Subsection 2.1.2 are overcome by various methods designed along the years. We briefly enumerate the main strategies that constitute the toolbox of every researcher and practitioner, in particular in terms of normalization layers. This line of work is related to our contribution detailed in Chapter 4.

Initialization Schemes

Properly initializing the weights of a neural network before training it alleviates the vanishing or exploding gradient problem in the first training iterations and allows stochastic gradient descent algorithms to find a suitable minimum, starting from this initialization. By studying the distribution of activations and gradients, Glorot and Bengio (2010) designed an initialization scheme to preserve the variance of activations and gradients across layers for networks with symmetrical activation functions like the sigmoid or the hyperbolic tangent. Following this idea, (Mishkin and Matas, 2016) and He et al. (2015c) designed initialization schemes for networks with Rectified Linear Units (ReLUs). This leads to a popular weight initialization technique

$$W^\ell \sim \mathcal{N}\left(0, \sqrt{2/N_{\ell-1}}\right)$$

where $N_{\ell-1}$ is the number of input features[5]. Finally, failure modes that prevent the training from starting have been theoretically studied by Hanin and Rolnick (2018).

Data Augmentation

Data augmentation is widely used to easily generate additional data to improve machine learning systems in various areas (Krizhevsky et al., 2012; Huang et al., 2016; Wu et al., 2019c) and to reduce overfitting. Traditionally, for object classification, at

[5]For convolutions, $N_{\ell-1} = K^2C$ where K is the kernel size and C the number of input channels.

training time, a random resized crop[6] is applied to the image which is then flipped horizontally with a probability 0.5. Since then, many more augmentation techniques were designed. For instance, Zhang et al. (2017a) train a network on convex combinations of pairs of examples and their labels, while (Cubuk et al., 2018) automatically search for improved data augmentation policies with a method called AutoAugment. We refer the reader to Cubuk et al. (2019) for a survey of data augmentation techniques. On a side note, the random resized crops used at training time involve a rescaling of the input image, in contrast to the center crops used at test time. Thus, the network is generally presented with larger objects at training time than at test time. This train-test resolution discrepancy is addressed by (Touvron et al., 2019) with a method called FixResNets.

Architectures

As networks are getting deeper, two major architectural changes are introduced to prevent the gradients to vanish. Rectified Linear Units (ReLUs) defined as $\sigma(x) = \max(0, x)$ are applied pointwise on the activations. Krizhevsky et al. (2012) were among the first to successfully employ such non-saturating activation functions to Convolutional Neural Networks, as opposed to traditional saturating functions like the sigmoid. Moreover, to allow the information to flow better up and down the network, He et al. (2015a) introduced skip-connections between blocks. More formally, if f is a building block[7], adding a skip connection amounts to output $f(x) + x$ after the block instead of $f(x)$ for any input activation x. Skip connections are now ubiquitous in deep learning architectures such as the Transformers in NLP (Vaswani et al., 2017).

Optimization

Neural networks are originally trained with Stochastic Gradient Descent (SGD)[8], generally with momentum LeCun et al. (1989). Denoting θ_t the parameters at time step t and $\mathcal{L}$ the loss function, one possible set of equations for SGD writes[9]:

$$\begin{cases} \theta_{t+1} = \theta_t - \eta v_{t+1} \\ v_{t+1} = \mu v_t + \nabla_\theta \mathcal{L} \end{cases}$$

[6]The input image is cropped with a random size and a random aspect ratio, and finally resized to the input size.

[7]For instance, two convolutions interleaved with one ReLU.

[8]Iterating over mini-batches of data, not single elements of the train set.

[9]There is also a Nesterov version (Nesterov, 1983) as well as the possibility to apply the learning rate directly to the gradient term $\nabla_\theta \mathcal{L}$ instead of the velocity term v_{t+1}.

where η is the learning rate and μ the momentum coefficient, generally set to 0.9. This remains the main training recipe for Image Classification problems (Goyal et al., 2017; Wu and He, 2018). The learning rate is generally following a *schedule*, meaning that $\eta = \eta_t$ depends on the current epoch or iteration. A classical schedule starts with a warm-up phase followed by a decay phase[10] (He et al., 2015a). Interestingly, the cosine schedule is gaining traction both in Vision and in NLP (Radosavovic et al., 2020). However, choosing a proper learning rate along with its schedule is expensive. Therefore, Duchi et al. (2011) designed a first-order gradient method named Adagrad that accounts for the anisotropic relation between the network's parameters and the loss function. More precisely, given a default learning rate η_0 usually set to 0.01 and a small constant for numerical stability ε,

$$\begin{cases} w_{e,t+1} = w_{e,t} - \dfrac{\eta_0}{\sqrt{g_{e,t+1} + \varepsilon}} \nabla_{w_e} \mathcal{L} \\ g_{e,t+1}^2 = g_{e,t}^2 + \left(\nabla_{w_e} \mathcal{L}\right)^2 . \end{cases}$$

In other words, each weight w_e is updated with an adaptive learning rate that depends on the sum of the past gradients with respect to this weight. Other adaptive algorithms were derived, or instance by Kingma and Ba (2014) which proposed a variant called Adam, frequently used in NLP. Intuitively, whereas SGD with momentum can be seen as a ball running down a surface, Adam behaves like a heavy ball with friction. For information about second-order methods or convergency considerations, we refer the reader to the manuscript by Bottou et al. (2016).

Weight Decay

Weight decay is a regularization method that is widely used in Deep Learning (Krogh and Hertz, 1992). It adds a penalty term to the traditional loss, for instance the cross-entropy loss $\mathcal{C}$ in image classification. The training loss writes

$$\mathcal{L} = \mathcal{C} + \lambda \|\theta\|_2^2$$

where θ is the network's parameters (weights and biases) and λ an hyper-parameter to cross-validate[11]. Weight decay is well suited for SGD as it amounts to add a term $2\lambda\theta$ to the gradients computed by back-propagation.

[10] For instance, given a default value η_0, use the learning rate $\eta_0/10$ during the first 5 epochs (warm-up). Then, set the learning rate back to η_0 and decay it a factor 10 every 30 epochs (decay).

[11] Generally, $\lambda = 10^{-4}$ or 10^{-5}.

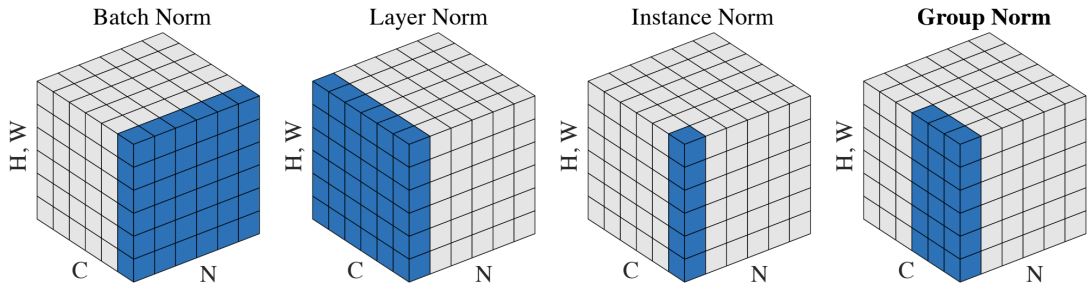


Figure 2-3: The effect of various normalization layers as illlustrated by Wu and He (2018). N denotes the batch size, C the channel dimension and H and W the spatial axes. The colored pixels are normalized with the same mean and standard deviation.

Dropout

Dropout (Hinton et al., 2012b) is a technique that randomly drops neurons[12], weights or more important structures (Ghiasi et al., 2018) during training time with a fixed small probability p[13]. This prevents the network from overfitting given the large variety of internal states it has to operate on. Dropout serves other purposes, for example, it helps pruning entire layers at test time (Fan et al., 2019) and is also related to our Quant-Noise contribution in Chapter 6.

Normalization Layers

Normalization procedures take an important part in the development of neural networks. While the inputs are almost always preprocessed[14] (LeCun et al., 1989), researchers began to normalize inner features as the networks were deeper. After successful attempts on whitening or local response normalization (Krizhevsky et al., 2012), Batch Normalization (BN) (Ioffe and Szegedy, 2015) is now a standard normalization layer. Let us detail BN in the context of computer vision. Denote the activations after some layer ℓ in the network by $x \in \mathbb{R}^{N \times C \times H \times W}$, where N is the batch size, C the number of channels, H and W the respective height and width of the activations, sometimes called *spatial dimensions*. For instance, an input image generally has shape $1 \times 3 \times 224 \times 224$ for three color channels (RGB) and a size of 224×224. For simplicity, we often flatten the last two dimensions so that $x \in \mathbb{R}^{N \times C \times HW}$ as depicted in Figure 2-3.

[12]When dropping a neuron $\nu \in H$ during a training iteration, we set $w_{\bullet \to \nu} = 0$ and $w_{\nu \to \bullet} = 0$ and we do not update $w_{\bullet \to \nu} = 0$ and $w_{\nu \to \bullet}$ during the backward pass. In other words, we detach all the incoming and outcoming connections of ν from the forward and backward passes.

[13]Generally $p \in [0.1, 0.3]$.

[14]For images, per-channel normalization of the pixel values within $[-1, 1]$ is a standard technique.

First, BN normalizes x *per-channel* into $\widehat{x}$. More formally, denoting $c \in [\![1, C]\!]$ a given channel, we have

$$\widehat{x}[:, c, :, :] = \frac{x[:, c, :, :] - \mu_c}{\sqrt{\sigma_c^2 + \varepsilon}}$$

where ε is a small constant for numerical stability and where μ_c and σ_c are the sample mean and (biased) standard deviation defined as

$$\mu_c = \frac{1}{NHW} \sum_{n,h,w} x[n, c, h, w]$$
$$\sigma_c^2 = \frac{1}{NHW} \sum_{n,h,w} (x[n, c, h, w] - \mu_c)^2$$

Second, the BN layer learns an affine transform of $\widehat{x}$ on the channel dimension:

$$y = \gamma \widehat{x} + \beta$$

where γ and β are learnt parameters of size C. More formally, for any batch, channel and spatial indexes n, c, h, w, we have

$$y_{n,c,h,w} = \gamma_c \widehat{x}_{n,c,h,w} + \beta_c. \tag{2.1}$$

Since the normalization statistics μ_c and σ_c depend on the batch, at test time BN is switched to *evaluation mode* and uses fixed statistics $\overline{\mu}_c$ and $\overline{\sigma}_c$ that are estimated with an exponential moving average of μ_c and σ_c during training time. Thus, at test time, BN is an affine layer.

While extremely effective in standard setups, Batch Normalization suffers from known shortcomings. In particular, BN only works well with sufficiently large batch sizes (Ioffe and Szegedy, 2015; Wu and He, 2018). For batch sizes below 16 or 32, the batch statistics μ_c and σ_c have a high variance and the test error increases significantly. Since then, variants of this technique such as Layer, Instance or Group Normalization (Ba et al., 2016; Ulyanov et al., 2017; Wu and He, 2018) were successfully introduced to circumvent the batch dependency, see Figure 2-3 for an illustration. For instance, Transformers (Vaswani et al., 2017) rely on Layer Normalization whereas Generative Adversarial Networks (GANs) use other variants such as the SPADE block (Park et al., 2019). This line of work is related to our Equi-normalization contribution in Chapter 4 where we re-normalize the weights – not the activations – in order to minimize the global L_2 norm of the network to ease the training.

Hyperparameter Tuning

One of the main difficulties of neural network training lies in finding the proper set of hyperparameter values[15] in the high-dimensional space of all the possible aforementioned techniques. For instance, Lample et al. (2017) found it extremely beneficial to add a dropout rate of 0.3 in some part of their architecture and Carion et al. (2020) underline the crucial importance of having two different learning rates for the two main components of their architecture. While the traditional cartesian grid search remains the main investigation tool, it requires a lot of computing power. For instance, with some PhD colleagues, we estimated that the energy consumed on average to produce one deep learning article has the same order of magnitude as the energy required to heat an average household during one year. Some more efficient techniques were developed, such as the gradient-free optimization platform Nevergrad (Rapin and Teytaud, 2018).

2.2 Equivalence Classes of Neural Networks

As demonstrated in Section 2.1, appropriate training techniques allow to train deeper and deeper networks in a fast-paced search for performance. Such networks are constructed by stacking elementary layers or more complex building blocks, which amounts to iterative function composition. For instance, the deepest ResNets (He et al., 2015a) have more than 100 layers. Although single-hidden-layer networks are well understood in terms of capacity to approximate functions presenting certain regularity properties[16] (Cybenko, 1989; Hornik, 1991), deeper networks remain difficult to comprehend despite numerous fructuous attempts (Eldan and Shamir, 2015; Cohen and Shashua, 2016). For instance, Mhaskar and Poggio (2016) prove matching direct and converse approximation theorems of complexity measurement for Gaussian Networks but not for ReLU networks[17]. In this section, we review theoretical studies

[15]This traditionally includes the learning rate and learning rate schedule, the optimizer, the momentum, the weight decay, the batch size or the dropout rate to name a few.

[16]A known result, proved independently by Cybenko (1989); Hornik (1991) states that networks with a single hidden layer with the sigmoid non-linearity can approximate with arbitrary precision any compactly supported continuous function. This result is known as the "Universal Approximation Property" and was extended to ReLU non-linearities for instance (Leshno et al., 1993).

[17]A Gaussian network has $x \mapsto \exp(-x^2)$ as activation function. The claim states as follows. (1) For a function f with a given smoothness, there exists a gaussian network g that approximates f, the quality of the approximation being controlled by a complexity measure of g. (2) Reciprocally, if any function f is approximated by a gaussian network g of given complexity, then the speed at which the approximation error decreases with respect to the complexity of g provides information about the smoothness of f.

studying deep networks under the light of their functional equivalence classes (Sussmann, 1992; Fefferman, 1994). For ReLU networks, functional equivalence classes are usually described in the parameter domain using so-called permutation and rescaling operations, that we review in Subsection 2.2.1. To better account for the duality between the function implemented by the network (which is abstract to manipulate) and its parameters (which allow for more concrete formulations), we study the Linear Regions (Pascanu et al., 2013) defined in the input space in Subsection 2.2.2. This helps to specify *irreducibility* or *non-degeneracy* conditions under which the functional equivalence classes are easily described by the sole rescaling and permutation operations. We review the main known results along with their limitations in Subsection 2.2.3. This work is related to our contribution detailed in Chapter 3. We end this section by discussing some adjacent work in Subsection 2.2.4 and by showcasing some practical applications in Subsection 2.2.5.

2.2.1 Permutations and Rescalings

After introducing some notations, we state the main definitions of rescaling-equivalent parameters in the literature and reconcile them under the condition that the considered parameterizations are admissible – recall that for an admissible parameterization, every hidden neuron is connected to at least one input and one output neuron through a path of non-zero weights (see Notations). We formally show that for ReLU non-linearities, rescalings preserve the function implemented by the network. We then introduce permutation equivalent parameters, that preserve the function implemented by the network for any activation function – and in particular for ReLU networks. In this Section, we fix the network architecture G. Recall that the function implemented by the parameterization θ is denoted by R_θ.

Rescaling Equivalent Parameters

We state three definitions for rescaling-equivalent parameters that we found in the literature for ReLU networks, namely at the layer, neuron or path level and provide an illustration in Figure 2-4. We next show that these definitions are equivalent under the condition that the considered parameterizations are admissible. Such a restriction to admissible parameterization is natural as explained in Subsection 2.2.2 for the case of "dead neurons". The fact that rescaling equivalent parameters preserve the function implemented by the network heavily relies on the homogeneity of the

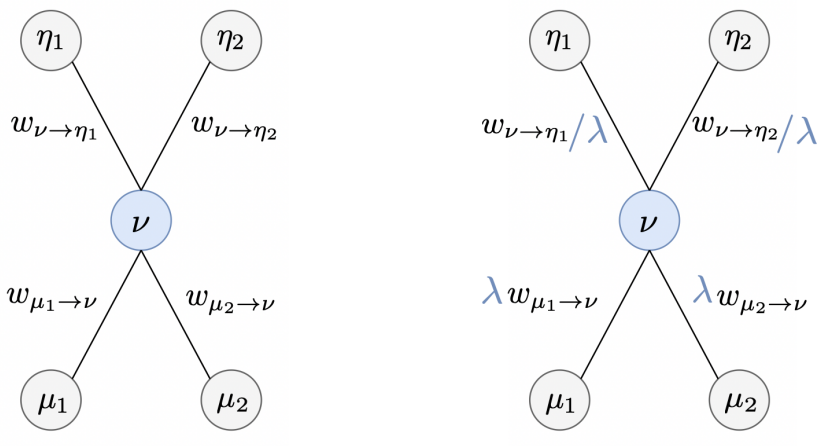


Figure 2-4: Illustration of the rescaling operations on one neuron, ν. Left: original network. Right: after applying the rescaling with $\lambda > 0$. For ReLU non-linearities, the function implemented by the network remains the same.

ReLU function σ (see Proposition A.1.1): for any $x \in \mathbb{R}$ and $\lambda > 0$,

$$\sigma(\lambda x) = \lambda \sigma(x). \tag{2.2}$$

The layer-wise equivalence is present for instance in the work of Nagel et al. (2019), Meller et al. (2019), Yuan and Xiao (2019) or Rolnick and Kording (2019).

Definition 2.2.1. *We denote by $\mathcal{D}(N)$ the set of diagonal matrices $D \in \mathbb{R}^{N\times N}$ such that, for all i, the diagonal element $d_i = D_{i,i}$ is strictly positive.*

Recall that $\theta = (w, b)$ can be also viewed as $\theta = (W^{(1)}, \dots, W^{(L)}, b^{(1)}, b^L)$.

Definition 2.2.2 (Layer rescaling equivalence)**.** *Two parameterizations θ and θ' are rescaling equivalent if, for all $\ell \in [\![1, L-1]\!]$, there exists $D^{(\ell)} \in \mathcal{D}(|N_\ell|)$ such that, for all $\ell \in [\![1, L]\!]$,*

$$W'^{(\ell)} = \left(D^{(\ell-1)}\right)^{-1} W^{(\ell)} D^{(\ell)} \qquad \text{and} \qquad b'^{(\ell)} = b^{(\ell)} D^{(\ell)} \tag{2.3}$$

with the conventions $D^{(0)} = I_{N_0}$ and $D^{(L)} = I_{N_L}$.

The neuron-wise equivalence is for instance used by Neyshabur et al. (2015). Let $\nu \in H$ be a neuron in some hidden layer and let $\lambda_\nu > 0$. A neuron-wise scaling is

defined as $s_{\nu,\lambda_\nu} : \theta = (w, b) \mapsto \theta' = (w', b')$ where for every connection $e \in E$,

$$w'_e = \begin{cases} \lambda_\nu w_e & \text{if } e = \bullet \to \nu \\ \frac{1}{\lambda_\nu} w_e & \text{if } e = \nu \to \bullet \\ w_e & \text{otherwise.} \end{cases} \tag{2.4}$$

and where, for every $\nu \in H$, $b'_\nu = \lambda_\nu b_\nu$. Let $\mathcal{S}$ be the set of neuron-wise scalings. We observe that two neuron-wise rescalings commute and that every neuron-wise rescaling is invertible, the inverse of s_{ν,λ_ν} being $s_{\nu,1/\lambda_\nu}$. Let $\langle \mathcal{S} \rangle$ be the commutative group generated by $\mathcal{S}$. Every $s \in \langle \mathcal{S} \rangle$ can be uniquely represented as the composition

$$s = \bigcirc_{\nu \in H} s_{\nu,\lambda_\nu} \tag{2.5}$$

where the λ_ν are strictly positive. Note that in this representation, every hidden neuron ν is associated to exactly one neuron-wise rescaling λ_ν.

Definition 2.2.3 (Neuron rescaling equivalence)**.** *Two parameterizations θ and θ' are rescaling equivalent if there exists $s \in \langle \mathcal{S} \rangle$ such that $\theta' = s(\theta)$.*

Definition involving paths in the networks are for instance employed by Meng et al. (2018) or Yi et al. (2019). We define the extended sign operator as, for $x \in \mathbb{R}$

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2.6}$$

When applied to a vector or a matrix, sign is taken entry-wise.

Definition 2.2.4 (Path rescaling equivalence)**.** *Two parameterizations θ and θ' are rescaling equivalent if:*

(i) $\text{sign}(\theta) = \text{sign}(\theta')$

(ii) For every full path $p \in \mathcal{P}$, $v_p(\theta) = v_p(\theta')$

(iii) For every partial path $q \in \mathcal{Q}$, $b_{q_0} v_q(\theta) = b'_{q_0} v_q(\theta')$.

Recall that a path $p = (\nu_0, \ldots, \nu_L) \in \mathcal{P}$ is a sequence of connected neurons from the input to the output layer, whereas a partial path $q \in \mathcal{Q}$ connects any hidden neuron to the output layer. The value of a path p is $v_p(\theta) = w_{\nu_0 \to \nu_1} \ldots w_{\nu_{L-1} \to \nu_L}$ and similarly for any partial path $q \in \mathcal{Q}$ (see Notations).

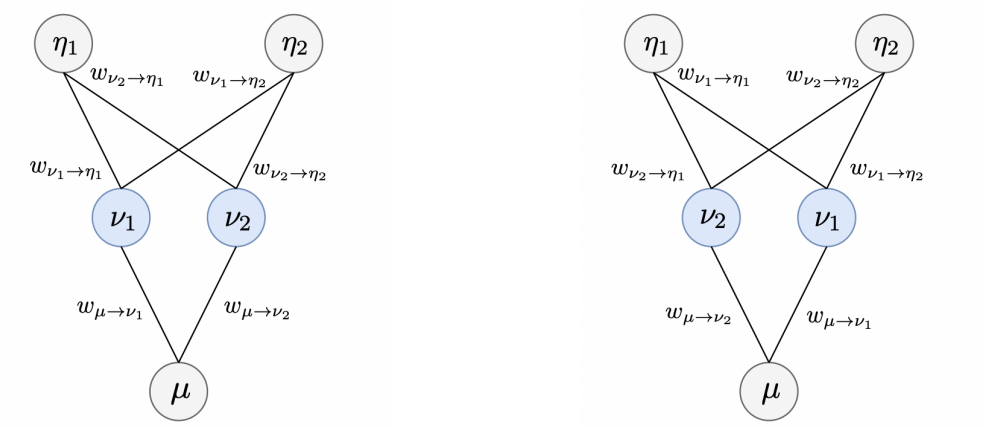


Figure 2-5: Illustration of the permutation operations on two neurons, ν_1 and ν_2. Right: original network. Left: after applying the permutation. The function implemented by the network remains the same for any activation function, and in particular for ReLU non-linearities.

Our first contribution is to reconcile these definitions on the layer, neuron and path level for *admissible* parameterizations as proven in Proposition 3.1.1. From now on, we denote by $\sim_S$ the rescaling equivalence relation between admissible parameterizations.

Permutation Equivalent Parameters

We provide a definition of permutation equivalent parameterizations (see Figure 2-5).

Definition 2.2.5. *We denote by* $\Pi(N)$ *the set of permutations* $\pi : [\![1, N]\!] \mapsto [\![1, N]\!]$. *With a slight abuse of notation,* $\pi \in \Pi(N)$ *also denotes the permutation matrix* $\pi \in \mathbb{R}^{N\times N}$ *such that* $\pi_{i,j} = 1$ *if* $j = \pi(i)$ *and* 0 *otherwise.*

Definition 2.2.6. *Two parameterizations* $\theta = (w, b)$ *and* $\theta' = (w', b')$ *are permutation equivalent if, for all* $\ell \in [\![1, L-1]\!]$, *there exists* $\pi \in \Pi(|N_\ell|)$ *such that, for all* $\ell \in [\![1, L]\!]$,

$$W'^{(\ell)} = \left(\pi^{(\ell-1)}\right)^{-1} W^{(\ell)}\pi^{(\ell)} \qquad \text{and} \qquad b'^{(\ell)} = b^{(\ell)}\pi^{(\ell)} \tag{2.7}$$

with the conventions $\pi^{(0)} = I_{N_0}$ *and* $\pi^{(L)} = I_{N_L}$. *We write* $\theta \sim_P \theta'$ *to denote the permutation equivalence between two parameterizations* θ *and* θ'.

As illustrated intuitively in Figure 2-5, permutations preserve the function implemented by the network for any non-linearity function, and in particular for ReLU non-linearities. We prove this result more formally in Proposition A.1.2.

Rescaling-permutation Equivalent Parameters

We define permutation-rescaling equivalent and rescaling-equivalent parameters.

Definition 2.2.7. *We call two parameterizations θ and θ' permutation-rescaling equivalent and we write $\theta \sim_{PS} \theta'$ if there exists a parameterization θ'' such that $\theta \sim_P \theta''$ and $\theta'' \sim_S \theta'$. We similarly define rescaling-permutation equivalent parameterizations and we denote $\theta \sim_{SP} \theta'$ if there exists a parameterization θ'' such that $\theta \sim_S \theta''$ and $\theta'' \sim_P \theta'$.*

Next, we show that the rescaling and permutation operations commute in Proposition A.1.3, thus both definitions coincide. From now on, we will denote $\sim_{PS}$ the permutation-rescaling equivalence. Such transformations preserve the realization of any ReLU network (Proposition 3.1.2).

Sign Flips

We will compare to existing work considering non-ReLU activation functions[18] in Subsection 2.2.3. Such activation functions include for instance the hyperbolic tangent, which is not homogeneous. Hence, we define *sign flips* by adapting Definition 2.2.3. Let $\mathcal{S}_\pm$ be the set of neuron-wise scalings restricted for every hidden neuron $\nu \in N$ to $\lambda_\nu \in \{-1, 1\}$. Let $\langle \mathcal{S}_\pm \rangle$ be the commutative group generated by $\mathcal{S}_\pm$. Every $s \in \langle \mathcal{S}_\pm \rangle$ can be uniquely represented as the composition

$$s = \bigcirc_{\nu \in H} s_{\nu, \lambda_\nu} \tag{2.8}$$

where $\lambda_\nu = \pm 1$. This operation amounts to flip signs of the weights and associated biases on the parameters.

Definition 2.2.8. *Two parameterizations θ and θ' are sign-flip equivalent if there exists $s \in \langle \mathcal{S}_\pm \rangle$ such that $\theta' = s(\theta)$.*

Since this Definition will be used only to compare to existing work in Subsection 2.2.3, we do not explore its variants as we did with the rescalings[19]. Since tanh is an odd function, sign flips preserve the function implemented by tanh networks but not by ReLU networks.

[18] Such related work is in general anterior to Krizhevsky et al. (2012) that were the first to introduce the ReLU non-linearity and prove its effectiveness.

[19] For instance, Definition 2.2.4 is not readily adaptable to sign flips.

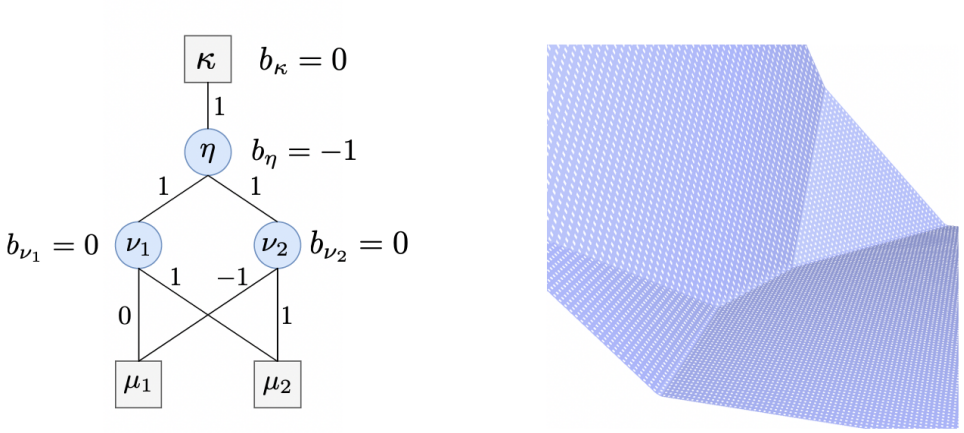


Figure 2-6: Left: A two-hidden layer network valued with a parameterization θ. Weights are indicated on the connections between neurons: for instance, $w_{\mu_1 \to \nu_1} = 0$. Biases are specified next to the neurons: for instance $b_\kappa = 0$. Input and output neurons are depicted as squares whereas hidden neurons are displayed as circles. Note that the input space is two-dimensional: $x = (x_{\mu_1}, x_{\mu_2})$ and the output is scalar: $|N_L| = 1$. Right: 3D visualization of the network's realization $x \mapsto R_\theta(x)$ which is piecewise affine.

2.2.2 Activation and Linear Regions

In this Subsection, we focus on ReLU networks. Such networks implement piecewise affine realizations $x \mapsto R_\theta(x)$ (Pascanu et al., 2013), as depicted for one particular example in Figure 2-6. Thus, studying the regions where the gradient is constant – the Linear Regions – yields valuable information about the network's architecture and weights, as uncovered by (Pascanu et al., 2013; Montúfar et al., 2014; Raghu et al., 2017). The main difficulty lies both in the dimensionality of the input space and in the number of layers L.

Activation vs. Linear Regions

We first define activation and linear regions of ReLU networks and leverage the work of (Hanin and Rolnick, 2019) that precisely explain the distinction between both. As explained in the Notations, for any hidden layer $\ell \in [\![1, L-1]\!]$, the intermediary function implemented by the layer *after* the non-linearity σ writes

$$y^{(\ell)}(\theta, x) = \sigma\left(y^{(\ell-1)}(x)W^{(\ell)} + b^{(\ell)}\right). \tag{2.9}$$

Recall that the function implemented by a single neuron $\nu \in H$ is denoted as $y_\nu(\theta, x) = y^{(\ell)}(\theta, x)_\nu$. Activation regions are defined by the activation pattern of

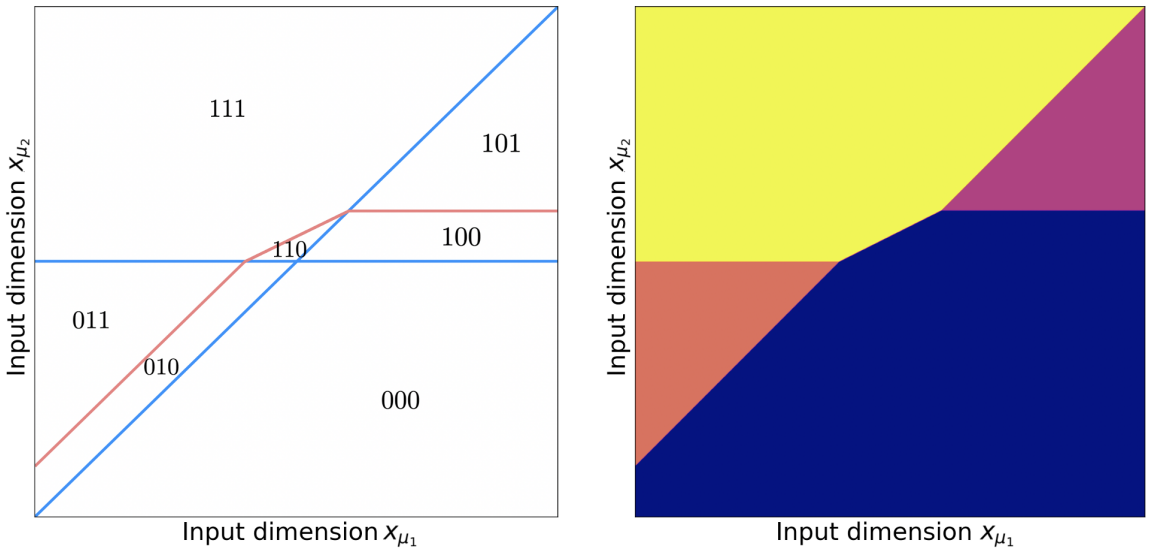


Figure 2-7: Activation and linear regions for the two-hidden-layer network defined in Figure 2-6. Recall that the input $x = (x_{\mu_1}, x_{\mu_2})$ is two-dimensional, *i.e.* $|N_0| = 2$ and the output is scalar, *i.e.* $|N_3| = 1$. Left: activation regions labeled with the activation pattern $\rho(x)$ for neurons (ν_1, ν_2, η) given the input x. For instance, $\rho(x) = 011$ means that the neuron ν_1 is *off* whereas both neurons ν_2 and η are *on* for input x. The two blue lines represent the separating hyperplanes for neurons ν_1 and ν_2 whereas the orange line denotes the *additional* gradient discontinuities introduced by neuron η. Right: linear regions implemented by R_θ. There are 7 activation regions and 4 linear regions. The figures are generated with our own PyTorch script available at `https://github.com/pierrestock/linear-regions/blob/main/partition.ipynb`.

all the hidden neurons for any input x. Indeed, denoting the ReLU function by σ, a neuron $\nu \in H$ is either *on*[20] when $y_\nu(\theta, x) > 0$ or *off* when $y_\nu(\theta, x) = 0$. For instance, as depicted in the example in Figure 2-7, a neuron ν belonging to the first hidden layer of the network defines a separating hyperplane Γ_ν in the input space, defined as the set of input points for which the output function of the neuron becomes strictly positive, or equivalently, where the gradient is discontinuous.

Definition 2.2.9 (Bent Separating Hyperplanes)**.** *For every $\nu \in H$, we denote*

$$\Gamma_\nu = \left\{ x \in \mathbb{R}^{N_0} \mid x \mapsto \nabla_\theta y_\nu(\theta, x) \text{ is discontinuous at } x \right\}.$$

For admissible parameterizations and for any neuron $\nu \in N_1$, Γ_ν is a hyperplane since $y_\nu(\theta) = \max\left(\langle x, w_{\bullet \to \nu} \rangle + b_\nu, 0\right)$[21] and since $w_{\bullet \to \nu} \neq 0$ (admissibility). This corresponds to the two blue lines on the left panel of Figure 2-7. For deeper hidden

[20] By analogy with the brain, we might say that a neuron *fires*.
[21] We denote by $w_{\bullet \to \nu}$ the vector $(w_{\eta \to \nu})_{\eta \in N_0}$.

neurons $\eta \in N_\ell$, $\ell > 1$, Γ_η is in general the union of (1) some gradient discontinuities due to the lower layers and (2) some gradient discontinuities that are induced by η. For instance, on Figure 2-7, the neuron on the second hidden layer (1) keeps a part of the gradient discontinuities induced by the two neurons of the first hidden layer, namely the lines between the orange and yellow regions, as well as the line between the purple and the yellow regions appearing in the right panel and (2) introduces new gradient discontinuities, namely the orange line depicted on the left panel. The sets Γ_ν are often referred to as "bent separating hyperplane" (Hanin and Rolnick, 2019). The main motivation for this name is that the new gradient discontinuities introduced by one neuron belonging to layer ℓ bend on the bent separating hyperplanes of lower layers $\ell' < \ell$ as demonstrated by Hanin and Rolnick (2019): for instance, on the left panel of Figure 2-7, the orange line bends on the two blue lines. However, without any further assumptions, the dimension of Γ_ν might be lower than $|N_0| - 1$[22]. Thus, the term "hyperplane" is not necessarily used rigorously in the literature but facilitates greater understanding and we choose to adopt it.

Definition 2.2.10 (Activation Regions)**.** *Let* $\Gamma = \cup_{\nu \in H} \Gamma_\nu$. *The activation regions are the connected components of* $\mathbb{R}^{N_0} \backslash \Gamma$.

Activation regions provide intimate information about each of the network's hidden neurons. One alternative definition proposed by Montúfar et al. (2014) is to assign to each input x a vector $\rho(x) \in 2^{|H|}$ where, for any hidden neuron $\nu \in H$, $\rho(x)_\nu = 1$ if $y_\nu(\theta, x) > 0$ and 0 otherwise. Then, ρ is piecewise constant and represents the activation pattern of all the hidden neurons for a given input, as depicted in Figure 2-7: for instance, the pattern $\rho(x) = (0, 1, 1)$ or "011" means that the neuron ν_1 is *off* whereas both neurons ν_2 and η are *on* for input x (see Figure 2-6 for the architecture and name of the neurons).

Definition 2.2.11 (Activation Regions bis)**.** *The activation regions are the areas of the input space where* $x \mapsto \rho(x)$ *is constant.*

Both definitions coincide (Hanin and Rolnick, 2019, Lemma 2). On the other hand, linear regions are areas of the input space for which the global function implemented by the network $x \mapsto R_\theta(x)$, or *realization*, has distinct locally affine behavior. In the remainder of this manuscript and with a slight abuse of notation, *we will interchangeably refer to affine or linear functions.*

[22]Hanin and Rolnick (2019) show that Γ_ν is a piecewise linear surface of codimension 1 for all but a measure-zero set of networks with respect to the Lebesgue measure.

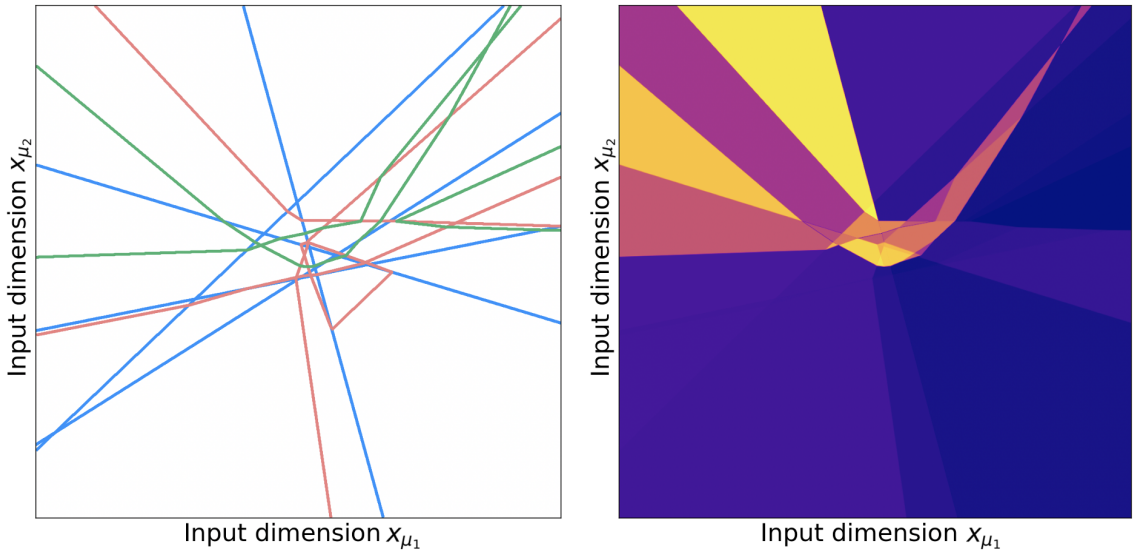


Figure 2-8: Activation and linear regions for a three hidden layer neural network with $|N_1| = 5$ neurons on the first hidden layer and $|N_2| = 4$ neurons on the second hidden layer and $|N_3| = 3$ neurons on the third hidden layer. The input x is two-dimensional, *i.e.* $|N_0| = 2$ and the output is scalar, *i.e.* $|N_4| = 1$. Left: activation regions. Right: linear regions. All the weights and biases were initialized randomly. The figures are generated with our own PyTorch script available at `https://github.com/pierrestock/linear-regions/blob/main/partition.ipynb`.

Definition 2.2.12 (Linear regions)**.** *For every $\nu \in H$, we denote*

$$B = \left\{ x \in \mathbb{R}^{N_0} \mid x \mapsto \nabla_x R_\theta \text{ is discontinuous at } x \right\}.$$

Then, the linear regions are the connected components of $\mathbb{R}^{N_0} \backslash B$.

Activation regions are convex and there are always at least as many activation regions than linear regions (Hanin and Rolnick, 2019, Lemma 3). For instance, the two-hidden-layer network defined in Figure 2-7 has 4 linear regions (see right panel), but 7 activation regions (see left panel). In general, for deeper networks, the number of activation and linear regions grows both exponentially as illustrated in Figure 2-8.

Parameters-Realization Duality

Rolnick and Kording (2019) prove, under certain assumptions detailed in Subsection 2.2.3, that it is possible to recover the network's parameters and architecture from the linear regions[23]. However, it is sometimes not the case. Let us focus on

[23]More precisely, from (1) linear regions and (2) each affine function implemented by the network on each linear region. Note that knowing (1) + (2) is equivalent to knowing the realization R_θ.

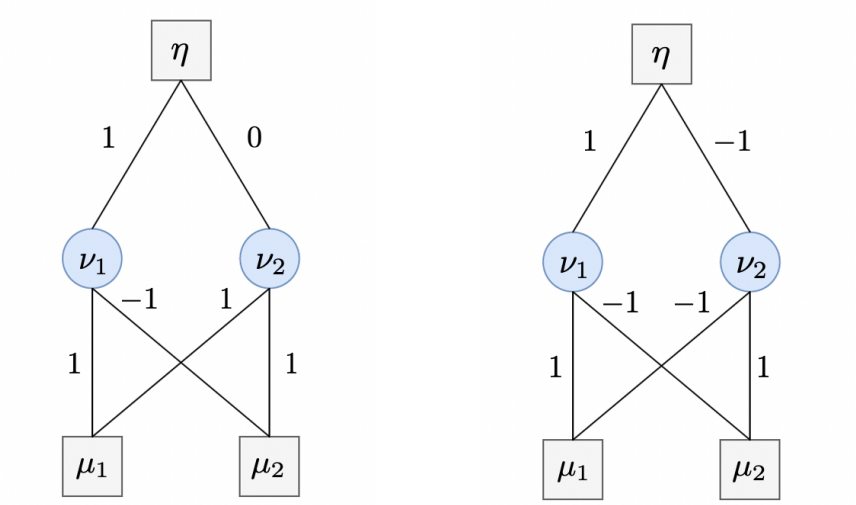


Figure 2-9: Left: example of a dead neuron, ν_2. Right: example of twin neurons, ν_1 and ν_2. Squares denote input or output neurons whereas circles denote hidden neurons. Weights are specified on the connections between two neurons and we set all biases to zero (not represented in the figures).

two toy examples, as illustrated in Figure 2-9, to better grasp the duality between the parameters and the realization. We focus on one-hidden-layer networks with $N_0 = \{\mu_1, \mu_2\}$, $N_1 = \{\nu_1, \nu_2\}$ and $N_2 = \{\eta\}$ and no biases.

- **Dead neuron.** We set $(w_{\mu_1\to\nu_1}, w_{\mu_2\to\nu_1}) = (1, -1)$, $(w_{\mu_1\to\nu_2}, w_{\mu_2\to\nu_2}) = (1, 1)$ and $(w_{\nu_1\to\eta}, w_{\nu_2\to\eta}) = (1, 0)$. Then, $R_\theta(x) = \max(0, x_{\mu_1} - x_{\mu_2})$ and R_θ only has two affine regions separated by a line $x_{\mu_1} = x_{\mu_2}$. Hence, the second neuron is *dead* (see Definition 3.2.1) and does not participate in the realization (since $w_{\nu_2\to\eta} = 0$) and the network cannot be recovered from the realization[24]. There are 4 activation regions but the dead neuron prevents them from showing up in the linear regions.

- **Twin neurons.** We set $(w_{\mu_1\to\nu_1}, w_{\mu_2\to\nu_1}) = (1, -1)$, $(w_{\mu_1\to\nu_2}, w_{\mu_2\to\nu_2}) = (-1, 1)$ and $(w_{\nu_1\to\eta}, w_{\nu_2\to\eta}) = (1, -1)$. Then, $R_\theta(x) = x_{\mu_1} - x_{\mu_2}$ is a hyperplane that has one linear region. Since the two neurons ν_1 and ν_2 have the same separating hyperplane, they are called *twin* neurons (see Definition 3.2.2) and the network cannot be recovered from the realization[25]. There are two activation regions separated by the hyperplane $x_{\mu_1} = x_{\mu_2}$.

Note that for the dead neuron example, the parameterization is *not* admissible since

[24] For instance, the weight $w_{\mu_2\to\nu_2}$ can take an arbitrary value, more details in Section 3.2.
[25] For instance, we can add any ε to $w_{\nu_1\to\eta}$ and subtract it from $w_{\nu_2\to\eta}$, see Section 3.2.

neuron ν_2 is not connected to any output neuron through a path of non-zero weights. Reciprocally, as shown in Chapter 3, if a parameterization θ is not admissible, then at least one hidden neuron is dead, where a dead neuron is formally defined in 3.2.1). Thus, it is natural to focus our attention to admissible parameterizations in our work. However, the admissibility condition does not exclude all possible degenerated cases as illustrated with the twin neurons example. In the two (degenerate) above cases, the activation regions cannot be recovered from the network realization.Hence, the functional equivalence classes of such examples are more vast than the sole rescalings and permutations (see Chapter 3 for details). We state precise non-degeneracy conditions in Subsection 2.2.3.

2.2.3 Functional Equivalence Classes

In this Subsection, we review related work aiming at characterizing functional equivalence classes under some precise conditions, sometimes called *non-degeneracy* or *irreducibility* conditions. We enumerate the known results along with their scope, the formulation of their assumptions and their limitations. Note that some results may not encompass the ReLU networks since they were proved before the recent introduction of the ReLU. In this case, other activation functions might necessitate sign flips instead of rescalings as defined in Subsection 2.2.1. The main difficulty is to properly identify the non-degeneracy conditions under which it is possible to easily describe the equivalence classes as rescalings (or sign flips) and permutations.

Definition 2.2.13. *Let G be a network architecture valued with the parameterizations θ and θ'. Then, θ and θ' are said to be functionally equivalent if, for all $x \in \mathbb{R}^{N_0}$, $R_\theta(x) = R_{\theta'}(x)$. We denote by $\sim_F$ the resulting equivalence relation.*

In this definition, we assume that θ and θ' share a common network architecture G. A more general definition would allow for two different architectures G and G' on the only conditions that G and G' share the same set of input neurons N_0 and output neurons[26]. As explained below, for the one-hidden-layer case, under proper *irreducibility conditions*, two functionally equivalent networks automatically share the same architecture: equivalently, two functionally equivalent one-hidden-layer networks valued with irreducible parameterizations θ and θ' have the same number of hidden neurons N_1. The question is more complex for deeper networks.

[26]In particular, without any constraint on the number of layers of G and G' or their sizes.

One Hidden Layer

We detail three successive approaches for the one-hidden-layer case that were designed in the early 90's, for activation functions satisfying various conditions[27]. These three results do not encompass the ReLU case as shown in Table 2.1. Since the considered activation functions are not homogeneous[28], rescalings are replaced by sign flips. All authors consider networks with one output node. Recall that a parameterization is *admissible* if for every hidden neuron $\nu \in H$ there exists a path $p \in \mathcal{P}$ going through ν such that $v_p(\theta) > 0$ as explained in the Notations. We denote by $w_{\bullet\to\nu}$ the vector $(w_{\mu\to\nu})_{\mu\in N_0}$. Thus, for a given architecture G valued with $\theta = (w, b)$, denoting the output neuron η, the function implemented by the network writes:

$$R_\theta(x) = \sum_{\nu\in N_1} w_{\nu\to\eta}\sigma\left(\langle w_{\bullet\to\nu}, x\rangle + b_\nu\right) + b_\eta.$$

All theorems below consider two parameterizations θ and θ' on two architectures G and G' having the same input and output neurons but *a priori* not the same number of hidden neurons. As summarized in Table 2.1, the results specify (1) a family of activation functions under study; (2) irreducibility conditions on θ and θ'; such that functional equivalence implies that the number of hidden neurons is the same and that θ' and θ are permutation sign-flip equivalent.

Sussmann (1992) considers only the tanh activation function, and specifies the irreducibility condition in terms of *dead* and *twin* neurons (See Subsection 2.2.2). Informally, a dead neuron does not participate to the output function of the network (for instance when all its incoming or outgoing weights are set to zero), whereas twin neurons can be collapsed to a single node without altering the input-output map. We study these notions more formally in Chapter 3. Albertini et al. (1993) consider any odd activation function σ that satisfies the *independence property*: for every positive integer p, for every $c_1, \ldots, c_p \in \mathbb{R}^*$ and every $\gamma_1, \ldots, \gamma_p \in \mathbb{R}$ such that $(c_i, \gamma_i) \neq \pm(c_j, \gamma_j)$ for all $i \neq j$ the functions

$$x \mapsto 1,\ x \mapsto \sigma(c_1 x + \gamma_1),\ \ \ldots,\ x \mapsto \sigma(c_p x + \gamma_p) \tag{2.10}$$

are linearly independent. Following earlier work by Kůrková and Kainen (1993), Kainen et al. (1994) consider two types of activation functions: (1) the gaussian function defined as $\sigma(x) = \exp(-x^2)$ and (2) even or odd, asymptotically constant,

[27] At that time, analogies to the human nervous system did not suggest that the ReLU non-linearity was a good candidate. It was considered later by Krizhevsky et al. (2012).

[28] f is homogeneous (of degree 1) if for every $x \in \mathbb{R}$ and every $\lambda > 0$, $f(\lambda x) = \lambda f(x)$.

Author	Activation function	Irreducibility conditions
Sussmann (1992)	tanh	(S.1) θ admissible (S.2) For all $\nu_1, \nu_2 \in N_1$, $\nu_1 \neq \nu_2$, $\lvert y_{\nu_1}(\theta)\rvert \neq \lvert y_{\nu_2}(\theta)\rvert$
Albertini et al. (1993)	Odd functions σ that satisfies the *independence property* (2.10)	(A.1) θ admissible (A.2) For all $\nu_1, \nu_2 \in N_1$, $\nu_1 \neq \nu_2$, $(w_{\bullet\to\nu_1}, b_{\nu_1}) \neq \pm(w_{\bullet\to\nu_2}, b_{\nu_2})$
Kainen et al. (1994)	Even or odd, asymptotically constant, non-polynomial rational functions. Extension to gaussian functions	(K.1) θ admissible (K.2) For all $\nu_1, \nu_2 \in N_1$, $\nu_1 \neq \nu_2$, $w_{\bullet\to\nu_1} \neq w_{\bullet\to\nu_2}$

Table 2.1: Specific functional equivalence results for one single hidden layer. All statements read as follows: for the family of activation functions, under the irreducibility conditions, the input-output map of a network uniquely determines its architecture and its parameters up to permutations and sign flips (see Subsection 2.2.1).

non-polynomial rational functions.

The irreducibility conditions summarized in Table 2.1 may differ in appearance from the original notations and statements on the papers. However, we have simply reformulated the assumptions to make them more consistent and readable and refer to the original papers for further details.

For the sake of clarity, own stand-alone proof for single-hidden-layer ReLU networks found independently, is available in Chapter 3, Proposition 3.2.1. The irreducibility condition reads: there are no dead hidden neurons and no twin hidden neurons as defined in Definitions 3.2.1 and 3.2.2. Note that this proof includes the case of multiple output neurons, *i.e.* $|N_2| > 1$, unlike the results detailed in Table 2.1.

Many Hidden Layers

Unfortunately, the proof techniques used in the single-hidden-layer case rely on geometrical arguments and are not readily adaptable for deeper networks. We detail below the known results and summarize them in Table 2.2.

Author	Activation function	Irreducibility conditions
Fefferman (1994)	tanh (permutations and sign flips)	(F.1) Non-nullity (F.2) Non-equality
Rolnick and Kording (2019)	ReLU (permutations and rescalings)	(R.1) Linear Regions (R.2) For any edge $e = \mu \to \nu$ ($\mu, \nu \in H$), $\Gamma_\mu \cap \Gamma_\nu \neq \emptyset$ (R.3) θ does not belong to certain a measure-zero set
Phuong and Lampert (2019)†	ReLU (permutations and rescalings)	(P.1) General Network (P.2) Transparent Network (P.3) Eligible Architecture

Table 2.2: Specific functional equivalence results for multiple hidden layers. All statements read as follows: for the family of activation functions, under the irreducibility conditions, the input-output map of a network uniquely determines its architecture (including depth) and its parameters up to symmetries. The symmetries depend on the activation function and are specified in the same column in parenthesis. Symmetries always include permutations and include either rescalings or sign flips (see 2.2.1). †The result of Phuong and Lampert (2019) writes as follows: given a bounded non-empty domain $Z \subset \mathbb{R}^{N_0}$, for eligible architectures (with non-increasing layer widths), there exists a parameterization θ such that the knowledge of the realization R_θ on Z determines the network's parameters uniquely up to permutations and scalings among general and transparent networks.

The first known exact result for networks with many hidden layers is due to Fefferman (1994) *in the case of the* tanh *non-linearity.* Indeed, before the introduction of the ReLU non-linearity (Krizhevsky et al., 2012), analogies to the human nervous system suggested to use a sigmoid-like activation function $\sigma(x)$ asymptotic to constants as $x \to \pm\infty$. The non-degeneracy conditions are specified as follows:

(F.1) Non-nullity. $b_\nu \neq 0$ for all $\nu \in H$ and $w_e \neq 0$ for all $e \in E$.

(F.2) Non-equality.

(i) For all $\nu \neq \nu' \in H$, $|b_\nu| \neq |b_{\nu'}|$.

(ii) For all $\ell \in [\![1, L]\!]$, for all $\nu \neq \nu' \in N_\ell$, for all $\mu \in N_{\ell-1}$, the ratio $w_{\mu\to\nu}/w_{\mu\to\nu'}$ is not equal to any fraction of the form p/q, with p, q integers and $1 \leq q \leq 100|N_\ell|^2$.

The statement reads as follows: under the two conditions above, perfect knowledge of the input-output map of a network uniquely specifies both the architecture (including depth) and the parameters, up to permutations and sign flips – we consider sign flips since the non-linearity is the hyperbolic tangent, see Definition 2.2.8. The idea of the proof is to reduce to the scalar case where $|N_0| = |N_L| = 1$ and to continue the realization $f(x) = R_\theta(x)$ analytically to a function f of a single, complex variable. Then, the qualitative geometry of the poles of f determines the network architecture and the asymptotics of f near its singularities determines the weights. The proof is not readily adaptable to ReLU networks since it heavily relies on the disposition of the poles of the meromorphic function $z \mapsto \tanh(z/2)$[29].

Rolnick and Kording (2019) propose to reverse-engineer deep ReLU networks by reading both the architecture of the network (including depth) and its parameters from the sole knowledge of the linear regions and each affine function implemented in these regions by the network (see Subsection 2.2.2 for definitions). The authors propose a constructive algorithm that samples network realizations for carefully chosen input points to deduce the architecture of the network and its parameters, up to rescalings and permutations. The authors make two crucial assumptions:

(R.1) Linear Regions Assumption. "Each [activation] region represents a maximal connected component of input space on which the piecewise linear function $x \mapsto R_\theta(x)$ is given by a single linear function" (see Definition 2.2.10).

(R.2) For any edge $e = \mu \to \nu$, $\Gamma_\mu \cap \Gamma_\nu \neq \emptyset$, *i.e.* two connected hidden neurons have bent separating hyperplanes that intersect, see Definition 2.2.9.

(R.3) The parameters θ do not belong to a certain measure-zero set.

Let us discuss assumption R.1 further. Recall that activation regions (Definition 2.2.10) are the areas of the input space that correspond to constant activation patterns of all the hidden neurons, whereas linear regions (Definition 2.2.12) correspond to the maximally connected areas of the input space for which the realization R_θ is an affine function. Assumption R.1 states that activation regions and linear regions coincide,

[29]The poles of $z \mapsto \tanh(z/2)$ are the points $(2m+1)i\pi$ with $m \in \mathbb{Z}$.

which is not always the case (see Subsection 2.2.2). For instance, the network considered in Figure 2-7 has 7 activations regions but only 4 linear regions[30], and does not satisfy R.1.

The authors prove that their algorithm terminates[31] except for a measure-zero [32] set of networks as stated in assumption R.3. In other words, under their assumptions, two functionally equivalent networks are permutation-rescaling equivalent. While their algorithm is constructive, the two assumptions are made on the linear regions and not on the network's parameters themselves, which might hide the underlying causes of such situations. Moreover, some precisions could have been made about the measure-zero set of network: which probability measure is used, and what are canonical examples of networks in this set.

Similarly, Phuong and Lampert (2019) propose to use linear regions to identify the network's parameters. They make the following assumptions that we cite below.

(P.1) General Network.

(i) For any $\nu \in H$, the local extrema of $x \mapsto y_\nu(\theta, x)$ do not have value zero.

(ii) For all k, ℓ such that $0 < k \leq \ell < L$ and all diagonal matrices $(I^{(k)}, \ldots, I^{(\ell)})$ with entries in $\{0, 1\}$ and with $I^{(\ell')} \in \mathbb{R}^{|N_{\ell'}|}$,

$$\operatorname{rank}\left(W^{(k)} I^{(k)} \cdots W^{(\ell)} I^{(\ell)}\right) = \min\left(|N_{k-1}|, \operatorname{rank}\left(I^{(k)}\right), \ldots, \operatorname{rank}\left(I^{(\ell)}\right)\right).$$

(iii) For all $\mu \neq \nu \in H$, any linear regions $A_1 \neq A_2$ of R_θ, the linear functions implemented by neuron ν on A_1 and neuron μ on A_2 are not multiple of each other.

(P.2) Transparent Network. For any input x, there exists at least one hidden neuron ν such that $y_\nu(\theta, x) > 0$.

(P.3) Eligible architecture. The network architecture G satisfies $|N_L| = 1$ and $|N_0| \geq |N_1| \geq \cdots \geq |N_{L-1}| \geq 2$.

Assumption P.3 is quite restrictive in terms of considered networks. The authors prove a much weaker result than the previous ones that states as follows: given a

[30] Activation regions labelled 010, 110, 100 and 000 belong to the same linear region.

[31] The complexity of the algorithm is not extensively studied by the authors for deep networks.

[32] Although the underlying measure is not explicitly stated, we may assume that the authors consider a measure-zero set with respect to the Lebesgue measure. This measure-zero set is not explicitly described by the authors.

bounded non-empty domain $Z \subset \mathbb{R}^{N_0}$, for eligible architectures (with non-increasing layer widths), there exists a parameterization θ such that the knowledge of the realization R_θ on Z determines the network's parameters uniquely up to permutations and scalings among general and transparent networks. Thus, given any parameterization θ, the result does *not* state that its parameters are uniquely determined up to permutations and scalings, even if θ is general with an eligible architecture. The assumptions however allow the authors to prove that *almost* all ReLU networks are general: "In other words, a sufficient condition for a [random] ReLU network to be general with probability one is that its weights are sampled from a distribution with a density". The authors consider only the scalar output case and do not formulate theorems for non-eligible architectures.

2.2.4 Relaxed Equivalence Classes

For many applications, for instance image classification, the quantity of interest it not the function implemented by the network R_θ or the loss, but rather the classification decision. In the ImageNet setup (Krizhevsky et al., 2012), there are 1000 classes, thus $N_L = 1000$ and the predicted class c for an input x is

$$c = \underset{\nu \in N_L}{\operatorname{argmax}}(R_\theta(x)_\nu).$$

Thus, various parameterizations θ and θ' of the same network might not be functionally equivalent, in the sense that $R_\theta = R'_\theta$, but may output exactly the same classification decisions for a given set of inputs, usually if their realizations are *close enough*. From this point of view, the strict concept of functional equivalence classes defined in Subsection 2.2.3 might be relaxed to preserve – approximately or exactly – a higher-level quantity of interest, such as the loss or the classification error, for the whole input space $\mathbb{R}^{N_0}$ or for a given distribution or set of test inputs.

In this section, we focus on the concept of inverse stability (Petersen et al., 2018; Berner et al., 2019). Consider a fixed ReLU architecture valued with two parameterizations θ and θ' and the following question.

"If the realizations R_θ and $R_{\theta'}$ are close, are the parameterizations θ and θ' close?"

This question is ill-posed and fundamentally connected to understanding the redundancies in the parameterizations of neural networks, precisely because of the rescaling and permutation operations defined in Subsection 2.2.1 that preserve the function implemented by the network. For instance, assume that R_θ and $R_{\theta'}$, as well as θ and θ',

are close. We might rescale θ' into θ'' such that $R_{\theta'} = R_{\theta''}$ and such that θ' and θ'' are arbitrarily far from each other. More precisely, following Definition 2.2.3, consider one single hidden neuron ν and define, for any $\lambda_\nu > 0$, $s = s_{\nu,\lambda_\nu}$ and $\theta'' = s(\theta')$. Further assume that the biases are all set to zero. Then,

$$\begin{aligned}\|\theta'' - \theta'\|_2^2 &= \sum_{\mu \in \text{prev}(\nu)} (\lambda_\nu w_{\mu\to\nu} - w_{\mu\to\nu})^2 + \sum_{\eta \in \text{next}(\nu)} (w_{\nu\to\eta}/\lambda_\nu - w_{\nu\to\eta})^2 \\ &= (\lambda_\nu - 1)^2 \sum_{\mu \in \text{prev}(\nu)} w_{\mu\to\nu}^2 + (1/\lambda_\nu - 1)^2 \sum_{\eta \in \text{next}(\nu)} w_{\nu\to\eta}^2.\end{aligned}$$

Hence, as $\lambda_\nu \to +\infty$, $\|\theta'' - \theta'\|_2^2 \to +\infty$ and we still have $R_{\theta'} = R_{\theta''}$ with θ' and θ'' far from each other. Since R_θ and $R_{\theta'}$ are close, R_θ and $R_{\theta''}$ are also close. Finally, since θ and θ' are close, θ and θ'' are far from each other. Thus, Berner et al. (2019) propose to reformulate the question as follows.

"If the realizations R_θ and $R_{\theta'}$ are close, can we find a parameterization θ'' that is permutation-rescaling equivalent to θ' such that θ and θ'' close?"

This question is addressed by Berner et al. (2019) in the one-hidden-layer case with no biases and scalar output. They exclude degenerate cases as follows.

(B.1) For all $\nu_1, \nu_2 \in N_1$ such that $\|w_{\bullet\to\nu_1}\|_\infty > 0$ and $\|w_{\bullet\to\nu_2}\|_\infty > 0$,

$$\frac{w_{\bullet\to\nu_1}}{\|w_{\bullet\to\nu_1}\|_\infty} \neq \frac{w_{\bullet\to\nu_2}}{\|w_{\bullet\to\nu_2}\|_\infty}.$$

(B.2) For all $\nu \in N_1$, $\|w_{\nu\to\bullet}\|_\infty = \|w_{\bullet\to\nu}\|_\infty$.

(B.3) There exists $\mu_1, \mu_2 \in N_0$ such that for all $\nu \in N_1$, $w_{\mu_1\to\nu} > 0$ and $w_{\mu_2\to\nu} > 0$.

Assumption B.1 is similar to assumptions S.2, A.2 and K.2. Assumption B.2 refers to the notion of *balanced* network (see Subsection 2.2.5 and Chapter 4). Assumption B.3 seems more restrictive. The statement is the following (Berner et al., 2019, Theorem 3.3) . Let G be a fixed, one-hidden layer architecture. Let θ and θ' be non-degenerate parameterizations on G, *without biases*. Then, there exists a non-degenerate parameterization θ'' such that $R_{\theta'} = R_{\theta''}$ and

$$\|\theta'' - \theta\|_\infty \leq 4|R_{\theta''} - R_\theta|_{W^{1,\infty}}^{\frac{1}{2}}.$$

Note that the distance between realizations is controlled with a Sobolev *semi*-norm. Thus, $|R_{\theta''} - R_\theta|_{W^{1,\infty}}^{\frac{1}{2}} = 0$ does not necessarily imply that $R_{\theta''} = R_\theta$. We refer the reader to Petersen et al. (2018) for a counter-example of this notion of stability.

Finally, we list below two interesting papers that closely relate to our investigations in Section 3.4.4 where we focus on local identifiability.

- Malgouyres and Landsberg (2018) consider so called *Deep Structured Neural Network* of arbitrary depth, without biases. Such *linear* networks are simply described by a product of weight matrices $M_i(h_i)$, where each matrix depends linearly on parameters $h_i \in \mathbb{R}^S$ for a given $S \in \mathbb{N}$. As explained in the paper, such architectures present strong connections with feedforward *non-linear* networks using the ReLU (for a fixed input x). Next, the authors define the rescaling equivalence by considering only *inter-layer* rescalings (Equation 3.2 in the paper), whereas we consider more generic *per-neuron* rescalings in this manuscript, see for instance Definition 2.2.3. The authors seek to establish sufficient and necessary conditions on deep structured linear neural networks guaranteeing local stability. More formally, let d be a metric taking into account the inter-layer rescaling, $x \in \mathbb{R}^{N_0}$ and $y \in \mathbb{R}^{N_L}$ an input/output pair and θ and θ' two parameterizations such that the considered network is indeed a deep structured network. Denote $\delta = \|y - R_\theta(x)\|$ and $\eta = \|y - R_{\theta'}(x)\|$. Then local stability writes as $d(\theta, \theta') \leq C(\delta + \eta)$ where $C \geq 0$ (Informal Theorem 1.1 in the paper). Here, we are interested in the case $\delta = \eta = 0$, *i.e.* local identifiability. In Section 6, the authors provide sufficient and necessary conditions for local identifiability by studying complex algebraic varieties leveraging the Segre embedding of the h_i. We refer to the paper for further details.

- Malgouyres (2020) also consider deep structured linear networks, without biases, under some sparsity constraints on the parameters $h_i \in \mathbb{R}^S$. The authors formalize the stability guarantees slightly differently than Malgouyres and Landsberg (2018), as stated in the Informal Theorem 1 in the paper. Let d be a metric taking into account the *inter-layer* rescaling of the weights. Define the empirical risk on a training set $(x_i, y_i)_{i=1..n}$ for a parameterization θ as $E(\theta) = \sum_i \|y_i - R_\theta(x_i)\|$. Then, the stability property writes: there exists $C \geq 0$ such that, for η sufficiently small and for any θ, θ' such that $E(\theta) \leq \eta$ and $E(\theta') \leq \eta$, we have $d(\theta, \theta') \leq C\eta$. In this work, the authors provide sharp conditions on the network architecture as well as the training set such that the parameters obtained *after* optimization in the training set are uniquely defined, up to *layer-wise* rescalings, again instead of neuron-wise rescalings.

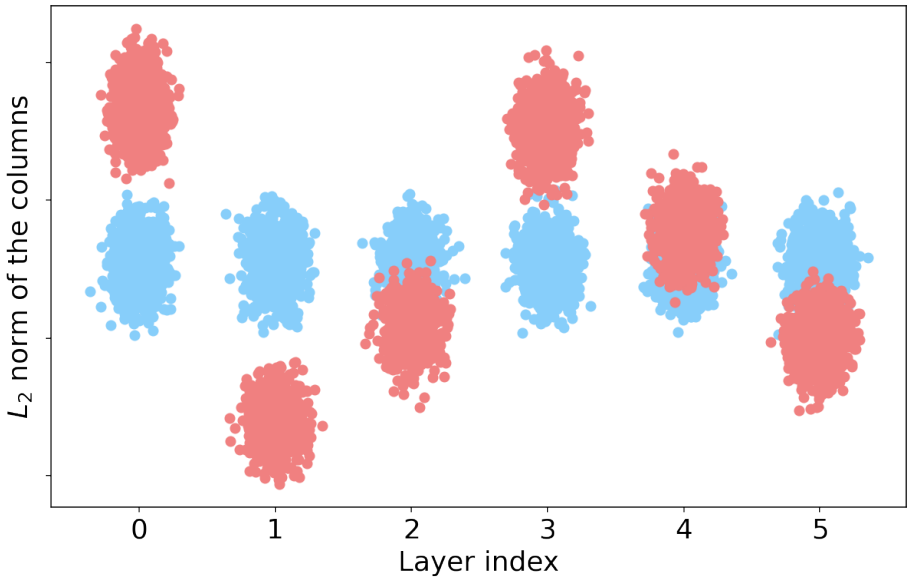


Figure 2-10: We compare a properly balanced network (blue) and an unbalanced network (red). Each dot represents the L_2 norm of one column in the layer's weight matrix. The dots are randomly spread around their layer index on the x-axis for readability. Both networks (balanced and unbalanced) implement the same input-output function.

2.2.5 Applications

As seen in the previous Subsection 2.2.3 for any activation function but more extensively for ReLU networks, functional equivalence classes are described by the sole permutation and rescaling operations, except for degenerate cases. In practice, researchers implicitly assume that the degenerate cases appear rarely, or even for a measure-zero set of networks ((Rolnick and Kording, 2019)). Moreover, since permutations are discrete operations[33], the authors mainly focus on the rescaling operations to describe functional equivalence classes. Since various representants of the same equivalence classes may react differently to optimization (Neyshabur et al., 2015) or scalar quantization (Nagel et al., 2019), the main difficulty lies in identifying a possible *canonical* representant, if any, for each equivalence class given the task at hand. In the following, we say that a network is *balanced* if all the weights have roughly the same magnitude, and *unbalanced* if this not the case, for instance when rescaling with large coefficients as explained in Subsection 2.2.4. An illustration of balanced and unbalanced networks implementing the same realization is given in Figure 2-10.

[33]Thus not readily amenable to back-propagation.

Quotient SGD

As observed by (Neyshabur et al., 2015), the balancedness of the network plays an important role in the optimization process. More precisely, given an architecture G, we compare a parameterization θ initialized with standard methods (as described in Subsection 2.1.3) and one rescaled, unbalanced parameterization θ' that yields the same realization as θ, as illustrated in Figure 2-10. It has been observed that, when initialized in an unbalanced fashion, neural networks are slower to train and can even converge to a poor local minimum[34].

Hence, Meng et al. (2018) propose to study the optimization problem over the space of realizations of neural networks rather than the space of parameters. The practical idea is to *re-balance* the network with a rescaling-equivalent network after each SGD step: this operation does not change the realization implemented by the network but will arguably change the future gradients and the learning trajectory[35]. More formally, this two-step optimization process amounts to quotient the parameter space with the rescaling equivalence relation $\sim_S$, implicitly assimilated to the functional equivalence relation $\sim_F$, where both notations are defined in Subsection 2.2.1. This line of work is related to our contribution in Chapter 4 where we derive a balancing algorithm that provably converges to a canonical representant for linear and convolutional networks and leverage it to improve the training. As stated in Chapter 4, the proposed algorithm – hence the obtained canonical representant – is justified by its practical efficiency. However, we make no theoretical claim supporting the fact that the proposed algorithm leads to the *right* canonical representant. Indeed, there may be alternative definitions leading to distinct canonical representants and we still miss a criterion to define "how good" a representant is.

Weight Equalization

Inspired by our contribution in Chapter 4, Nagel et al. (2019) and Meller et al. (2019) concurrently use our Equi-Normalization algorithm to balance networks and therefore make *them* more amenable to 8-bit scalar quantization (see Subsection 2.3.5 for details about quantization). Nagel et al. (2019) conjecture that the performance of trained models after quantization can be improved by adjusting the weights of each channel such that their ranges are more similar. As the authors empirically demonstrate, the quantization error induced on a single layer by traditional methods

[34] Figure 1(a) of Neyshabur et al. (2015).
[35] See Chapter 3 for a formal derivation.

such as Quantization-Aware Training (QAT, Krishnamoorthi (2018)) may induce a drift in the distribution of the activations of successive layers with potential damaging effects on the network's accuracy. Such methods are discussed in Subsection 2.3.5.

2.3 Compression of Deep Learning Models

Deep Learning models are able to perform multiple tasks such as image recognition, language understanding or speech enhancement thanks to a very active research community. Productionizing such over-parameterized – hence heavy – models, either on the server side or on the mobile side, poses challenges in terms of latency, memory and energy. In this Section, we detail the main compression techniques to produce models with fast inference speed, small memory footprint and energy efficiency. One of the most straightforward methods to compress a neural network is to prune or delete some of its parts (Subsection 2.3.1). More structured sparsity leads to the design of Structured Efficient Layers detailed in Subsection 2.3.2. This work in turns conducts the research community to construct or search for new efficient architectures (Subsection 2.3.3). Networks with these architectures are either learnt from scratch or using a more advanced teacher network with distillation techniques (Subsection 2.3.4). Moreover, reducing the precision at which to store single weights or subvectors leads to the fields of scalar and vector quantization (Subsection 2.3.5). Finally, we discuss hardware and metrics considerations in Subsection 2.3.6.

2.3.1 Pruning and Sparsity

Pruning amounts to remove some part of a network – such as a connection or a neuron – according to some importance criterion (LeCun et al., 1990; Hassibi and Stork, 1993). Generally, researchers start from a pretrained network and iteratively prune and finetune the network until the desired size/accuracy tradeoff is reached.

Importance criteria

The importance criteria are designed to quantify the effect of deleting a given part of the network on its accuracy. Then, the parts with the lowest importance are removed (or equivalently, freezed to zero[36]). For the sake of clarity, let us write the most

[36]We consider the weight w_e with $e \in E$. Then, to prune w_e, we simply remove e from the graph G. Equivalently, we set $w_e = 0$ and we do not update w_e during the backward pass. It amounts to detach the edge e from the computational graph during the forward and backward passes.

common importance criteria when pruning connections (*i.e.*, individual weights) in a network – sometimes called unstructured pruning. These methods generalize to larger structures such as neurons, filters or layers as explained in the references below.

Let us denote the weights of the network by $w = (w_e)_{e \in E}$, where E is the set of edges or connections of the network's architecture. We aim at defining an importance criterion $C \colon E \mapsto \mathbb{R}$ that will be used to rank the weights. One of the most straightforward implementation of C is to consider the magnitude of the weights:

$$C(e) = |w_e|. \tag{2.11}$$

There is a large line of work considering weight magnitude pruning (Han et al., 2015; Li et al., 2016; Gale et al., 2019) with extensions to the L_p norm when considering larger structures such as filters[37]. The intuition behind this criterion is that large weights (in absolute value) have a large influence on the loss function $\mathcal{L}$ whereas small weights have a negligible influence[38].

In line with the Optimal Brain Damage approach of LeCun et al. (1990), it is possible to construct a local model of the loss function $\mathcal{L}$ to predict the effect of perturbing the parameters by δw. Denoting $\nabla \mathcal{L}(w)$ the gradient of $\mathcal{L}$ with respect to w and $H(w)$ the Hessian, the Taylor approximation writes

$$\delta \mathcal{L} = \sum_e \nabla \mathcal{L}(w)_e \delta w_e + \frac{1}{2} \sum_e H(w)_{ee} \delta w_e^2 + \frac{1}{2} \sum_{e \neq e'} H(w)_{ee'} \delta w_e \delta w_{e'} + O(\|\delta w\|^3). \tag{2.12}$$

Pruning a single weight w_e is equivalent to applying a perturbation δw to w such that $\delta w_{e'} = -w_{e'}$ if $e' = e$ and 0 otherwise. Let us examine how to derive first and second order importance criteria.

- **First order.** If we neglect all the second-order terms in Equation (2.12), following Molchanov et al. (2016); Yu et al. (2018b), we obtain $\delta \mathcal{L} = \sum_e \nabla \mathcal{L}(w)_e \delta w_e$. Then, during training, we wish to prune a single weight w_e that will result in the minimum absolute loss variation $|\delta L|$. Using perturbations defined earlier, we are left to find e such that $|\nabla \mathcal{L}(w)_e w_e|$ is minimum. This leads to the criterion

$$\mathcal{C}(e) = |\nabla \mathcal{L}(w)_e w_e| \,. \tag{2.13}$$

[37] The importance ranking of filters may differ when considering the L_1 or the L_2 norm for instance.

[38] This claim implicitly assumes that the network is *balanced*, *i.e.* that the weights have the same order of magnitude. For the effect of rescalings on weight magnitude, see Chapter 4 and Figure 4-1.

Pruning with this criterion is sometimes referred to as gradient magnitude pruning. See Figure 2-11 for a comparison.

- **Second order.** Following LeCun et al. (1990), we assume that the training has converged at a local minimum, thus $\nabla\mathcal{L}(w) = 0$. We also assume that the Hessian is diagonal, thus we neglect the cross-error terms of the form $H(w)_{ee'}\delta w_e \delta w_{e'}$ for $e \neq e'$. Based on these two assumptions and the fact that the Hessian is positive semidefinite at a local minimum, we deduce that $H(w)_{ee} \geq 0$. Then Equation (2.12) boils down to $\delta\mathcal{L} = \frac{1}{2}\sum_e H(w)_{ee}\delta w_e^2$, and we wish to prune a single weight w_e that will result in the minimum absolute loss increase δL, which leads to the criterion

$$\mathcal{C}(e) = H(w)_{ee} w_e^2. \tag{2.14}$$

  However, computing second-order information involves additional code and relies on some approximations on the Hessian[39].

Such first- and second-order criteria are sometimes referred to as weight saliencies. In contrast to the weight magnitude, these criteria are data-dependent, and the gradients or the (approximated) Hessian are generally computed on a single batch of training data. In practice, we prune a proportion p of the weights at the same time, neglecting the influence of the deletion of one parameter to the saliency of other parameters. Note that *biases are generally not pruned*, since they represent a small proportion of the networks' parameters, although the same reasoning could apply. Finally, there is an impressive amount of importance criteria in the literature, for instance based in filter statistics, entropy or mutual information. Moreover, the scale at which to apply these criteria – single layer or entire network – may vary. See He et al. (2017b); Huang et al. (2018b); Mittal et al. (2018); Luo et al. (2017); He et al. (2018a); Evci et al. (2019); Louizos et al. (2017b) for pruning variants. For extensive references, we refer the reader to the surveys by Liu et al. (2018) or Blalock et al. (2020). Finally, the recent work by Frankle and Carbin (2018) constitute an interesting direction to extract performant subnetworks[40] that train efficiently (Caron et al., 2020).

[39]The Hessian has dimension: number of network parameters × number of network parameters, and is therefore a huge matrix, but it can be approximated to by a diagonal matrix, in which case it is no bigger than the gradient (Ollivier, 2015).

[40]Related work around the so-called "Lottery Tickets" will not be discussed here.

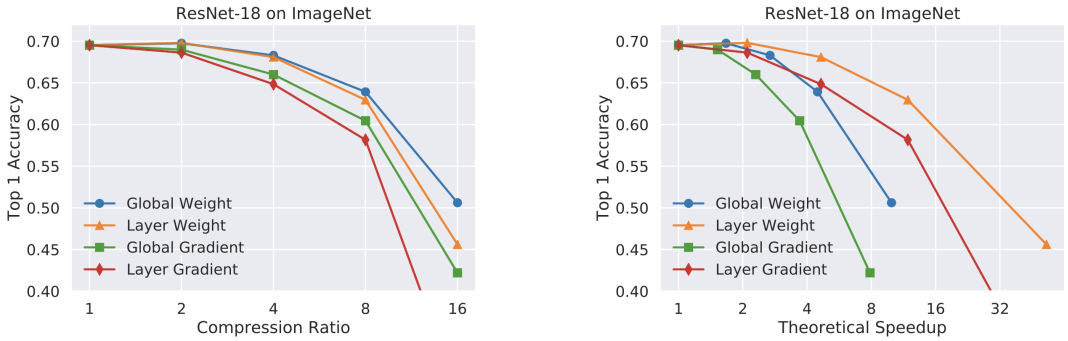


Figure 2-11: Accuracy for several levels of compression (left) and theoretical speedup (right) for a ResNet-18 on ImageNet reported by Blalock et al. (2020). Weight and Gradient refer respectively to weight magnitude pruning and gradient magnitude pruning. If these criteria are applied *per layer*, the method is prefixed with Layer. If the criteria are applied to the *whole network*, the method is prefixed with Global.

Pruning in practice

Pruning is a very common technique for network compression and is for instance implemented directly in PyTorch[41]. Users can perform random or magnitude pruning or even define their own criterion to prune connections, neurons or channels. On a side note, a technique for pruning[42] a given set of entire layers at test time called Stochastic Depth was developed and by Huang et al. (2016) for Vision architectures and extended and modified by Fan et al. (2019) in Natural Language Processing tasks with Transformers (Vaswani et al., 2017).

Pruning standard networks like a ResNet-18 (He et al., 2015a) on ImageNet with pruning baselines in Equations (2.11) and (2.13) generally leads to a 2 to 4× compression ratio at the cost of a few percentage point in test top-1 accuracy, as depicted in Figure 2-11. We observe a set of empirical Pareto Curves: the more we compress the network, the more the accuracy decreases. Note that considering the criteria (weight magnitude or gradient magnitude) for the whole network[43] yields better accuracy than considering them per layer[44].

[41] `https://pytorch.org/tutorials/intermediate/pruning_tutorial`.

[42] More precisely, a "pruned" layer ℓ is replaced by the identity layer so that the output of the previous layer $y^{(\ell-1)}$ is directly fed to layer $\ell+1$.

[43] Rank all the weights $w_e, e \in E$ and prune them according to the ranking.

[44] Rank the weights $w_{\mu\to\nu}, \ell(\nu) = \ell$ per-layer and prune them according to the per-layer ranking.

2.3.2 Structured Efficient Layers

In this Subsection, we seek to replace a dense weight matrix by a matrix or a product or matrices that have fewer parameters while being faster at inference time. The main constraint is that the parameters must be learnt – we do not seek to approximate a dense trained matrix but rather to directly learn the structured matrix, so the factorization must be compatible with back-propagation[45]. We briefly review the main ideas below and point the interested reader to the survey by Cheng et al. (2017). In this subsection we restrict ourselves to the $2D$ case and seek to approximate a weight matrix $W \in \mathbb{R}^{n \times n}$ for simplicity.

Low Rank

One of the most common ways to approximate W while saving parameters is to create a bottleneck in the network by writing $W = UV$ where $U \in \mathbb{R}^{n \times p}$ and $V \in \mathbb{R}^{p \times n}$ where p is smaller than n. This is equivalent to replacing the fully connected $n \times n$ layer by two stacked fully-connected layers of dimensions $n \times p$ and $p \times n$, respectively. The compression ratio is $2p/n$. This layer is generally a strong baseline, as the resulting network will be forced to find a compressed representation of the information, which is the idea at the core of the auto-encoders (Bengio, 2009). The Low Rank (or bottleneck) paradigm was extremely fruitful. For instance, generalizations of the low-rank decomposition such as Tensor Train Oseledets (2011) or T-Net (Kossaifi et al., 2019); some Transformer architectures using a low-rank approximation or the self-attention mechanism such as the Linformer (Wang et al., 2020), and approximations of the softmax[46] (Grave et al., 2016) are all inspired by the low-rank paradigm.

Structured Sparse

Using a random sparse matrix instead of a dense one, with a proportion α of non-zero coefficients amounts to prune a proportion $1 - \alpha$ of the weights before the training (Molchanov et al., 2016). For such matrices, sparse calculations rely on optimized kernels [47]. However, more structured operations may be faster. For instance, a structured layer with a block-diagonal matrix, a permutation of the features and another

[45]More precisely, the structured layers could be initialized with an approximation of the dense matrix instead of being randomly initialized, and then finetuned. In practice, finetuning is necessary to get good performance, therefore authors generally skip this initialization step (when it is computationally tractable).

[46]The softmax function $f \colon \mathbb{R}^d \mapsto \mathbb{R}^d$ is defined as $f(x)_j = \frac{e^{x_j}}{\sum_i e^{x_i}}$.

[47]Such as the cuSPARSE kernel on Nvidia's GPUs: `https://developer.nvidia.com/cusparse`.

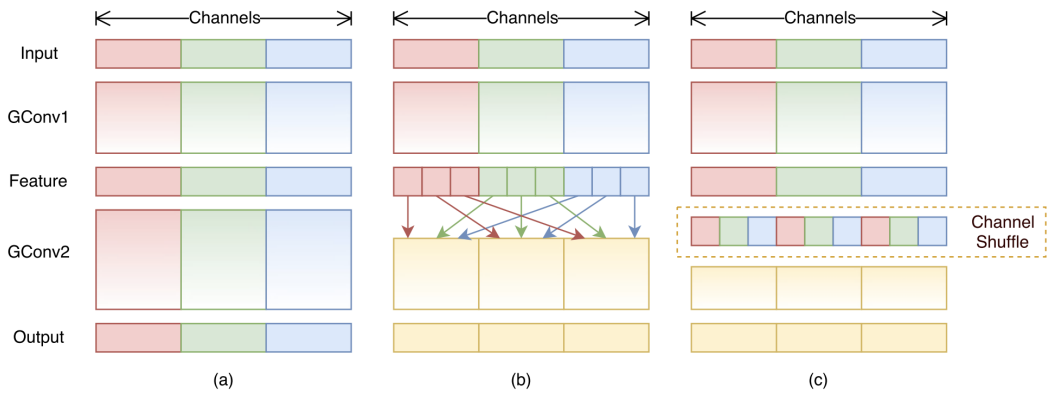


Figure 2-12: ShuffleNet block (Zhang et al., 2017b). (a) no shuffle, (b) channel shuffle and (c) an equivalent implementation of (b).

block-diagonal matrix. This parameterization relies on the batch-multiplication operation. We define a block-diagonal matrix with k blocks M_i of size $p \times p$ as

$$B = \begin{pmatrix} M_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & M_k \end{pmatrix}.$$

Then, the structured layer writes $W = B_1 P B_2$ where B_1 and B_2 are learnt block-diagonal matrices and P is a permutation matrix (which is not learnt but fixed). The idea is to combine local linear interactions in B_1 and B_2 interleaved by a permutation to add some diversity. The compression ratio is $2kp^2/n^2$. Zhang et al. (2017b) use this idea to design an efficient network based on group convolutions – the convolutional equivalent of the block-diagonal matrix B – and permutations called the ShuffleNet, depicted in Figure 2-12.

Fourier and Hadamard

A few methods rely on the discrete Fourier or Hadamard transforms, such as Fast-Food (Le et al., 2014), ACDC (Moczulski et al., 2015) and their variants (Arjovsky et al., 2015; Choromanska et al., 2015). these methods rely on the Fast Fourier or Hadamard Transform and its $O(n \log n)$ implementation using the Cooley-Tukey algorithm (Cooley et al., 1967).

Let us give some insights explaining why such transforms are used, besides their computational advantage. We rely on the theory of Fast Random Projections developed by Ailon and Chazelle (2006). The goal of this approach is to perform random

projections that are computationally cheap in time and space, that approximately preserve the metric information and that are *not learnt*. In a random projection, the basic operation takes the form $y = Wx$, where W is a random matrix. As stated by the famous Johnson-Leidenstrauss Lemma (Johnson and Lindenstrauss, 1984), the obtained embeddings approximately preserve metric information[48] when W is a Gaussian matrix (a matrix whose entries are drawn independently from $\mathcal{N}(0, 1)$). Ailon and Chazelle (2006) introduced an alternative approach that mimics the properties of random Gaussian matrices but that can be stored very efficiently. In particular, they proposed the PHD transform: $y = PHDx$ where P is a sparse random matrix with Gaussian entries, H is the Hadamard matrix and D a diagonal matrix with entries in $\{-1, +1\}$ drawn independently with probability $1/2$.

The transformations that we will detail now (ACDC and FastFood) can be seen as an alternative to this fast random projection where some of the parameters are *learnt*. The idea is to keep the approximate metric preservation properties while letting the network adapt to this particular layer. Denoting diagonal matrices of learnt parameters with the letter D, fixed permutations matrices with P, the Hadamard transform with H and the Fourier transform with F, the FastFood transform writes $D_1HD_2PHD_3$ whereas the ACDC transform corresponds to $D_1FD_2F^{-1}$.

Circulant Matrices and their Variants

We now review some classical learnt structured matrices whose storage complexity is linear (Sindhwani et al., 2015). The circulant matrix of size n is defined with $(u_0, \ldots, u_{n-1}) \in \mathbb{R}^n$ and this vector is successively shifted to generate the next columns. Similar matrices are the Vandermonde and Cauchy matrices. The structure of these matrices can be exploited for faster linear algebraic operations such as matrix-vector multiplication, inversion and factorization. In particular, the matrix-vector multiplication with circulant matrix is computable in in $O(n \log n)$ time using the Fourier Transform. Vandermonde and Cauchy matrix-vector operations take $O(n \log^2 n)$ time (Sindhwani et al., 2015). Note that there exist infinite possibilities of designing structured matrices and combining them to obtain structured layers (see Choromanska et al. (2015)) for further examples.

[48] On finite sets, provided that the dimension of y is at least logarithmic in the size of the set.

Kronecker

Jose et al. (2018) consider parameterizing the matrix W as a Kronecker product of k learnt matrices called Kronecker factors. This representation is flexible as we can choose the number and the size of the Kronecker factors, which allows us to design models with the appropriate trade-off between computational efficiency and accuracy. The authors also introduced what they call a soft unitary constraint (Cisse et al., 2017) which helps conditioning the matrix, especially in a recurrent network. A strict unitary constraint (*i.e.* projecting the matrix at each iteration so that it is orthogonal) would be computationally expensive and may deter the performances of the network. Kronecker products (or sum of Kronecker products) as also useful in the context of dictionary learning, Dantas et al. (2017, 2019).

2.3.3 Architecture Design for Fast Inference

Techniques designed in previous Subsections 2.3.1 and 2.3.2 led the research community to focus on more efficient architectures that are trained from scratch. Such architectures are either designed by hand – following some ground principle – or using more advanced frameworks (architecture search or graph neural networks).

Efficient Architectures

One of the first deep neural networks, AlexNet (Krizhevsky et al., 2012), contained 60 million parameters, including 10 million for the last three fully connected layers. The research community first searched for deeper and more parameter-efficient networks to ease the training on GPUs, using only convolutions with kernel sizes of 3×3 and 1×1 respectively[49] (except for the first layer) and one single fully connected layer acting as a classifier. Such popular networks include the VGG (Simonyan and Zisserman, 2014), Inception (Szegedy et al., 2014), ResNets and ResNeXts (He et al., 2015a; Xie et al., 2017) models that are intensively used for various benchmarks.

Guided by the possible mobile applications, researchers then focused on networks that are able to run in real-time on a CPU at inference time. Such models, among which MobileNets (Howard et al., 2017; Sandler et al., 2018a), ShuffleNets (Zhang et al., 2017b) or SqueezeNets (Hu et al., 2018) typically rely on a combination of

[49] A traditional building block generally includes a 3×3 convolution (*i.e.*, a convolution with kernel size 3×3) interleaved with two 1×1 convolutions (He et al., 2015a).

Operation	Kernel Size	Parameters	Multiplications
Standard Convolution	3×3	$9C_{\text{in}}C_{\text{out}}$	$O(9HWC_{\text{in}}C_{\text{out}})$
Point-wise Convolution	1×1	$C_{\text{in}}C_{\text{out}}$	$O(HWC_{\text{in}}C_{\text{out}})$
Depth-wise Convolution	3×3	$9C_{\text{out}}$	$O(9HWC_{\text{out}})$

Table 2.3: Respective number of parameters and multiplications for various types of convolutions. The input activations have size $1 \times C_{\text{in}} \times H \times W$. For depth-wise convolutions, we assume that $C_{\text{in}} = C_{\text{out}}$. Thus, sequentially combining one point-wise convolution with C_{in} input channels and C_{out} output channels, one depth-wise convolution and one point-wise convolution with C_{out} input channels and C_{in} output channels only necessitates a fraction $C_{\text{out}}/C_{\text{in}}(2/9 + 1/C_{\text{in}})$ of the parameters and operations required by a single standard 3×3 convolution with C_{in} input channels and C_{in} output channels.

depth-wise[50] and point-wise[51] convolutional filters that require less parameters and floating-point operations than a traditional 3×3 convolution (see Table 2.3). These efficient architectures inspired the development of more efficient Transformers in NLP (Vaswani et al., 2017) such as MobileBERT (Sun et al., 2020) as well as architecture adaptations by Wu et al. (2019b); Zhang et al. (2018a) and Sukhbaatar et al. (2019a).

Finally, the EfficientNet (Tan and Le, 2019) and RegNet (Radosavovic et al., 2020) network families bridge the gap between GPU-efficient, yet heavy models and mobile-capable networks by proposing a range of models for various operating points. Their common denominator is to heavily rely almost exclusively on group convolutions and pointwise convolutions with a small – yet carefully tuned – number of channels that grows with depth. As depicted in Figure 2-13, the smallest networks have over 4 million parameters for a test top-1 error on ImageNet around 25%, to be compared with a ResNet-50[52] that has 24 million parameters for a test top-1 error around 24%.

Feature Reuse and Wiring Patterns

Guided by the use of residual connections (He et al., 2015a), Huang et al. (2017a) designed a densely connected convolutional network (DenseNet) where each layer takes as input the output features of all the preceding layers. Such architectures are more parameter-efficient and yield good performance, in part since the skip-connections strongly encourage feature reuse through the upper layers, and allow us to easily train very deep networks (more than 100 stacked layers). For instance, a

[50]Group convolutions with the number of groups equal to the number of input features, generally with a kernel size of 3×3.

[51]Standard 1×1 convolutions.

[52]The suffix 50 stands for the number of layers in the network.

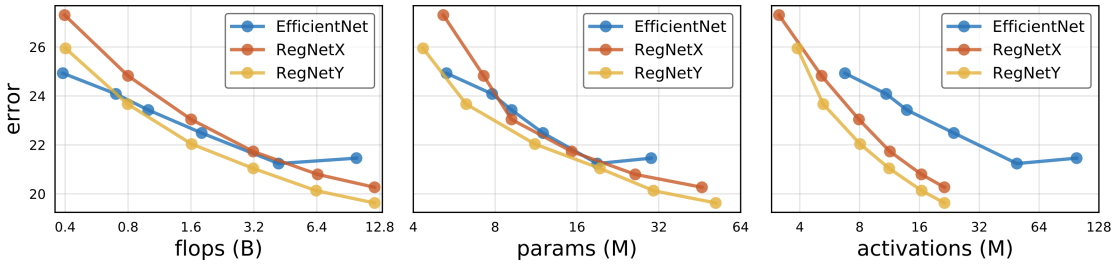


Figure 2-13: Tradeoffs in terms of FLOPs, number of parameters and total number of activations during a forward pass for EfficientNets and RegNets as exposed by Radosavovic et al. (2020). Error is test top-1 error on ImageNet object classification.

DenseNet-201 achieves 1.5% less top-1 error than a ResNet-50 for the same number of parameters. In the search for more efficiency, (Huang et al., 2018a) propose to replace the standard 3×3 convolutions used in the DenseNet by learnt 3×3 group convolutions in a network called the CondenseNet. Indeed, group convolutions are more parameter efficient and mobile-friendly than their traditional counterparts[53].

Feature reuse also powers the feature extractor of the Faster-R-CNN object detection model (Ren et al., 2015), which is a Feature Pyramid Network, allowing to extract information from the input image at different scales. We refer the reader to (Lin et al., 2017a) for further details.

Weight Sharing

Weight sharing is a simple technique assigning different parts of the network with the same value, for instance when two layers are assigned with the same weight matrix. During back-propagation, the gradients are summed[54] so that the updated shared value remains the same. This method is used among others by Dehghani et al. (2019) or Lan et al. (2019) to produce parameter-efficient Transformers.

A natural extension of weight sharing is the Siamese Networks (Bromley et al., 1993). The idea is to share the weights or two entire blocks or networks, each of them being given a different type of input, in order to mutualize knowledge. Siamese Networks are used successfully to compare image patches (Zagoruyko and Komodakis, 2015), for object tracking (Bertinetto et al., 2016) or for joint learning of speaker and phonetic similarities (Zeghidour et al., 2016).

[53] The depth-wise convolution in Table 2.3 is a particular case of group convolution.
[54] Or averaged.

Neural Architecture Search

Feedforward Neural Networks are defined as directed acyclic graphs. In light of this generic definition, it appears that the traditional architectures represent a negligible proportion of all possible feedforward networks. Instead of relying on some based principles and designing the network essentially by hand, requiring some in-domain knowledge, Zoph and Le (2016) propose to use reinforcement learning to generate the model descriptions of neural networks that maximize the reward – here the validation accuracy on a dataset of interest, followed by Liu et al. (2017a); Pham et al. (2018); Liu et al. (2019a) and Luo et al. (2018). Such approaches conducted Zoph et al. (2017) to produce the NASNet models, Howard et al. (2019) to produce MobileNet-v3, Tan and Le (2019) to produce the aforementioned EfficientNet seed network or Chen et al. (2020a) to produce the AdaBERT models.

Following this recent progress, Xie et al. (2019) propose to loosen the constraints on the search space of wiring patterns by relying on various random graph models from graph theory. Going further, Radosavovic et al. (2019) and Radosavovic et al. (2020) propose to design *design spaces* combining the advantages of Neural Architecture Search (NAS) and manual design, leading to the RegNet network families. Finally, traditional NAS approaches require to train (at least partially) the generated network to get the reward *at each time step* of the reinforcement learning algorithm, which is extremely computationally intensive[55]. To alleviate this issue, Wu et al. (2019a) propose to train a stochastic *super-net* that contains a set parallel blocks at each layer, from which one given is sampled during every forward pass.

On a side note, Press et al. (2020) design a "Sandwich Transformer" by testing some random reordering of the sublayers of a Transformer Network and demonstrate gains on machine translation compared to the standard interleaved transformer.

2.3.4 Distillation: Learning Form a Teacher

Distillation techniques boost the accuracy of a small *student* network by training it to mimic the output of a larger *teacher network* instead of training it from scratch (Bucila et al., 2006; Ba and Caruana, 2013; Hinton et al., 2015).

[55] See in particular Table 3 of Wu et al. (2019a).

KL Divergence

Assume the student is producing a vector of probabilities $q = (q_1, \dots, q_n)$ and the teacher a vector of probabilities $p = (p_1, \dots, p_n)$[56]. Here, the student network will not be trained with a classical cross-entropy loss (defined in Equation (2.16)) but with a distillation loss measuring the distance between the two output distributions:

$$\mathcal{L}_D(q, p) = D_{KL}(p||q) = \sum_i p_i \log\left(\frac{p_i}{q_i}\right). \tag{2.15}$$

Intuitively, say the task is ImageNet classification for both the teacher and the student, and the input image is a picture of an apple and a banana, labeled as apple. When training the student network from scratch without distillation, the cross-entropy loss will indicate that only a banana is present in the picture since the supervision comes from a one-hot encoding of the true class. However, when trained with distillation, the student network supervised with the vector of probabilities of the teacher network for this particular image that contains richer information. In particular, the class banana will be assigned with a small yet non-zero probability. Note that the cross-entropy loss is a particular case of the KL Divergence when the teacher probability vector corresponds to the one-hot encoding of the true class indexed by j: if $p_i = 0$ for $i \neq j$ and $p_j = 1$, then[57]

$$\mathcal{L}_C(q, j) = D_{KL}(p||q) = \log\left(\frac{1}{q_j}\right) = -\log(q_j). \tag{2.16}$$

Sometimes, the student is trained with a combination $\mathcal{L}$ of the traditional cross-entropy loss $\mathcal{L}_C$, and of the distillation loss $\mathcal{L}_D$. Denoting $\lambda > 0$ a regularization parameter, $\mathcal{L}(q, p, j) = \mathcal{L}_C(q, j) + \lambda\mathcal{L}_D(q, p)$. The teacher network is trained beforehand and stays fixed – it only provides the probability vectors – during the distillation.

Distillation at Every Layer

Following the distillation paradigm, Romero et al. (2014) propose to train a student network using not only the outputs but also the intermediate representations $y^{(\ell)}(x)$ generated by the teacher network. Since the size of the intermediate representations – or *features* – of the student and the teacher or even the number of layers may not be equal, the authors introduce additional learnt parameters to map these rep-

[56] Such vectors are obtained by applying the softmax function to the output of the network.
[57] With the convention that $0 \log(0) = 0$.

resentations and minimize the L_2 norm between them. Experiments conclude that guiding the training of the student network with intermediate "hints" provides faster and better convergence, as demonstrated subsequently for BERT model training by (Sanh et al., 2019) or (Jiao et al., 2019) and even for Generative Adversarial Networks (GAN) by (Li et al., 2020).

Semi-Supervised Training

When training a student network with the plain distillation loss $\mathcal{L}_D$, there is no need for labeled data, provided a trained teacher network is available. However, the teacher network might be trained with labeled data for particular tasks such as image classification. Yalniz et al. (2019) exploit this idea and propose to train a large-capacity teacher network on a labeled dataset. The teacher is then used to label a large collection of unlabelled images (up to 1 billion) that are use to pre-train the student model, which is finally finetuned on the labelled dataset.

2.3.5 Scalar and Vector Quantization

Quantization generally refers to the idea of storing parameters or groups of parameters at a lower precision than the standard floating-point `fp32` format[58] – recall that `fp32` numbers are stored over 32 bits or 4 bytes[59]. The cornerstone of quantization is the k-means algorithm (Steinhaus, 1956; MacQueen et al., 1967; Lloyd, 1982) used to cluster the parameters or groups of parameters by assigning every member of one cluster with the value of its centroid or codeword. This discretization helps reduce the network size and improve the inference time with specialized kernels and/or hardware as explained in Subsection 2.3.6, at the cost of a reduced accuracy. In what follows we work with a weight matrix $W = (w_{k,l}) \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}}}$ and focus on the main training techniques for quantizing while maintaining a desired level of accuracy.

Floating-point Arithmetics

Before exposing the quantization methods per se, we briefly review the floating-point *half-precision* format or `fp16`[60], which is similar to the `fp32` format except that the numbers are stored over 16 bits. This format, that halves the size of the network, is

[58] Also called `float 32` or *single-precision* as opposed to `fp64` *double-precision*.

[59] The IEEE 754 `fp32` format has 1 sign bit, 8 exponent bits and 23 mantissa bits to represent numbers in absolute value between 1×10^{-38} and 3×10^{38} with 6 to 9 significant decimal digits.

[60] The IEEE 754 `fp16` format has 1 sign bit, 5 exponent bits and 10 mantissa bits to represent numbers in absolute value between 6×10^{-8} and 65,504 with 3 to 4 significant decimal digits.

becoming standard for storing and even for training Deep Learning models (Micikevicius et al., 2018). Indeed, training with half-precision allows us to consider bigger networks or to train existing networks faster as detailed in Subsection 2.3.6. Since the range allowed by `fp16` is smaller than `fp32`, the authors propose to stabilize the training (1) by dynamically scaling the loss so that the gradients of the activations fall within the acceptable range; (2) by keeping a master `fp32` copy[61] of the network's parameters for the optimization step and (3) by leaving the BatchNorm layers in `fp32` format. Many frameworks such as APEX[62] now propose seamless integration of mixed precision training with no loss of accuracy compared to `fp32`.

Scalar Quantization

Fixed-point scalar quantization is a generic method that replaces the floating-point weights by N bit fixed-point numbers (Gupta et al., 2015). Fixed-point operations generally take less energy and area on chips than their floating-point counterparts (Han et al., 2016a; Lian et al., 2019). Setting $N = 8$ defines the popular `int8` quantization scheme while having $N = 1$ encompasses the extreme case of binarization (Courbariaux et al., 2015). More precisely, the parameters are rounded to one of 2^N possible codewords. Let us detail the case of *uniform* scalar quantization below. Here, the codewords correspond to bins evenly spaced by a scale factor s and shifted by a bias z. Each weight w_{kl} is mapped to its nearest codeword c by successively quantizing with $z \mapsto \text{round}(w_{kl}/s + z)$ and dequantizing with the inverse operation:

$$c = (\text{round}(w_{kl}/s + z) - z) \times s, \tag{2.17}$$

where we compute the scale and bias from the trained `fp32` weight statistics:

$$s = \frac{\max W - \min W}{2^N - 1} \quad \text{and} \quad z = \text{round}(\min W/s). \tag{2.18}$$

This *calibration step* varies according to the quantization methods and will be discussed in the remainder of this Section. Note that the uniform rounding scheme in Equation (6.2) allows for fixed-point arithmetic with implementations in PyTorch and Tensorflow (see Subsection 2.3.6). The compression rate is $32/N$ and the activations $y^{(\ell)}(x)$ are also generally rounded to N-bit fixed-point numbers. In what follows, we

[61] This occupies additional memory in the GPU. However, the number of training activations is generally much higher than the number of weights in standard training regimes with large batches. Hence, this additional cost is minor.

[62] `https://github.com/NVIDIA/apex`.

first focus on the widely used case of `int8` with $N = 8$ and then detail compression schemes with lower precisions. Due to its wide range of applications, there is an extremely large body of literature on `int8` quantization. For the sake of conciseness, we mention the salient trends and refer the reader to the two recent surveys (Guo, 2018; Cheng et al., 2017) for a comprehensive overview.

- **Post-training Quantization.** One straightforward baseline is that of post-training quantization, where the scale s and the zero-point z are calibrated[63] as follows.
  - The weight quantization parameters are calibrated offline as the weights are fixed, and *per tensor* as in Equation (2.18). There is also the possibility of quantizing *per channel* where we calibrate one scale and zero-point per channel, which diminishes the quantization error while presenting a small memory overhead[64], although it requires specialized kernels, see Subsection 2.3.6.
  - The activation quantization parameters are calibrated using an exponential moving average (EMA) of the current quantization parameters when forwarding a few batches of data and maintaining the weights fixed. Since there may be some outliers in the activation values, considering the minimum and maximum values of the activations for calibration as detailed in Equation (2.18) may decrease the accuracy. Hence, there exist more sophisticated *observers*[65] minimizing the quantization error measured as the L_2 norm between the quantized and non-quantized tensors.

- **Quantization-Aware Training (QAT).** This very strong baseline was designed by Krishnamoorthi (2018). The idea is to emulate `int8` quantization of the weights and/or the activations *when training the network* to prepare the post-training quantization. The main under-the-hood take-away is to use the Straight-Through Estimator (STE) during the backward pass (Bengio et al., 2013; Courbariaux and Bengio, 2016). With STE, the gradients are obtained by replacing the non-quantized weights by their quantized counterparts during the backward pass (see 6.4.1).

- **Learnable Scale and Zero-Point.** Instead of calibrating the scale and zero-point as explained in Equation (2.18), one possibility is to learn them through back-propagation. The main difficulty is to make the quantization operation dif-

[63] `https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html`.
[64] For a square matrix of size $|N_\ell| \times |N_\ell|$, it requires $2|N_L|$ calibration parameters instead of 2.
[65] `https://github.com/pytorch/pytorch/blob/master/torch/quantization/observer.py`.

ferentiable with respect to w and z, either by modeling the quantization noise[66] (Nagel et al., 2019) or by approximating the gradients (Bhalgat et al., 2020).

- **Non-uniform Schemes.** The common idea of non-uniform quantization schemes (Choi et al., 2018; Li et al., 2019) is to take advantage of the bell-shaped and long-tailed distribution of the weights and activations (Han et al., 2016b). Indeed, uniform quantization is optimal when the weights or activations follow a uniform distribution, which is not the case in practice. Here, by having adaptive bin sizes, the authors observe less degradation in the test accuracy. However, the methods are less prone to hardware acceleration, in contrast to uniform quantization.

- **Mixed precision.** There are extensive studies of scalar quantization to train networks with lower precision weights and activations, for instance (Courbariaux et al., 2015; Courbariaux and Bengio, 2016; Zhu et al., 2016; Zhou et al., 2016; Rastegari et al., 2016; McDonnell, 2018). Note that precisions can be mixed (Wu et al., 2018) depending on the layer's sensitivity to quantization[67] (Gluska and Grobman, 2020), or by considering 4-bit weights and 8-bit activations for instance.

Vector Quantization

Vector Quantization (VQ) and Product Quantization (PQ) have been extensively studied in the context of nearest-neighbor search (Jegou et al., 2011; Ge et al., 2014; Norouzi and Fleet, 2013). The idea is to decompose the original high-dimensional space into a product of subspaces that are quantized separately with a joint codebook. To our knowledge, Gong et al. (2014) were the first to introduce these quantizers for neural network quantization, followed by Carreira-Perpiñán and Idelbayev (2017).

Compressing W with Product Quantization (PQ) amounts to evenly split each column into m subvectors of dimension d and to learn a codebook on the resulting set of mC_{out} subvectors[68]. Then, each column of W is quantized by mapping each of its subvectors to its nearest codeword in the codebook $\mathcal{C} = \{c_1, \ldots, c_k\}$ as illustrated in Figure 2-14. When m is set to 1, PQ is equivalent to vector quantization (VQ) and when m is equal to C_{in}, it is the scalar k-means algorithm. The main benefit of PQ is its expressivity: each column is mapped to a vector in the product $\mathcal{C} = \mathcal{C} \times \cdots \times \mathcal{C}$, thus PQ generates an implicit codebook of size k^m.

[66] For instance, model the noise as a normal distribution $\mathcal{N}(0, \sigma)$ and compute the gradient wrt σ.

[67] Measure for instance by observing the loss degradation with quantizing only this particular layer.

[68] For simplicity, we assume that C_{in} is a multiple of m, *i.e.* that all the subvectors have the same dimension $d = C_{\text{in}}/m$.

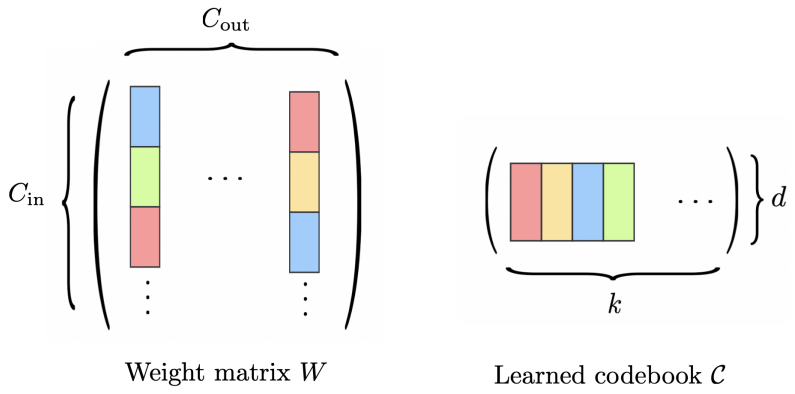


Figure 2-14: Performing Product Quantization on a weight matrix W.

As we will see in Chapters 5 and 6, employing this discretization off-the-shelf does not optimize the right objective function, and leads to a catastrophic drift of performance for deep networks. However, using a proper objective function and a careful finetuning of the codebook (Han et al., 2016b) yields state-of-the-art performance as explained in the aforementioned Chapters.

2.3.6 Hardware and Metrics

While the metrics of network size (measured in MB) and predictive performance (measured with some test accuracy) enjoy a wide consensus for measuring the compression efficiency, the inference time is subject to various metrics or proxys that depend on the target hardware. Here, we provide a few pointers to such metrics along with their (dis)advantages. We focus on inference (test time) for two reasons. First, deep learning models are currently mostly deployed to embedded devices or production servers for inference. Second, it is currently not possible to train a quantized neural network below half-precision: for instance, QAT (see Subsection 2.3.5) only *emulates* quantization[69] to prepare the post-training, *true* quantization. Therefore, training with low-precision ML[70] is a future challenge (Kung et al., 2020; Jeon et al., 2020).

Kernels

We first focus give a very high-level overview of the concept of *kernels*. Roughly speaking, a kernel is a hardware-dependent set of low-level instructions that perform standard linear algebra operations such as matrix multiplication or convolution in an

[69] By successively quantizing and dequantizing weights and activations.
[70] Low-precision Machine Learning.

optimized way. The first baseline for accelerating neural network inference is to *fuse* kernels. For instance, at inference time, the sequence convolution-ReLU-BatchNorm[71] is fusable into a single kernel and saves in/out memory writing of the intermediary activations. This same logic is followed by TorchScript[72], which is an intermediate, compiled representation of a PyTorch model that runs in high-performance environments such as C++. Finally, `int8` libraries such as `fbgemm`[73] provide speedups compared to their float counterparts by performing the operations – for instance convolutions – in `int8`. Accumulation is performed in `int32` and the weights are pre-packed so that their access in memory during the inner loop are adjacent.

Hardware

Fixed-point quantization methods such as `int8` benefit from specialized hardware to also improve the runtime during inference (Vanhoucke et al., 2011). Researchers then attempted to incorporate these hardware constraints when designing their compression algorithms, in particular in terms of latency: for instance, using an emulator of the target hardware (Wang et al., 2018a; Yang et al., 2016), or a proxy of the hardware inference time such as FLOPs (see next), or with the framework of reinforcement learning (Howard et al., 2019). Since specialized hardware is generally more performant but allows for less flexibility in the choice of operations and is less widespread, there is a tradeoff depending on the target inference time, the target devices and the development time. Among the popular hardware present in embedded devices, we name a few, including: the CPU[74], the DSP[75], the GPU[76], or dedicated accelerators for deep learning inference such as Apple's Neural Engine. Some modern chips include all these components, such as the SnapDragon 855[77]. Each computation unit comes with its own development toolkit and libraries[78].

FLOPS and Other Metrics

Since measuring the hardware latency requires a consequent engineering effort as seen in the previous paragraph, some work consider proxy metrics such as the FLOPs or

[71] `https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html`.
[72] `https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html`.
[73] `https://github.com/pytorch/FBGEMM`.
[74] Central Processing Unit, the most widespread but the less specialized computation unit.
[75] Digital Signal Processor, usually used to process real-world time series.
[76] Present in most modern smartphones, offloads the CPU for image processing and ML inference.
[77] Which also has a dedicated module for fast `int8` inference.
[78] For instance, models running on Apple's phone GPUs are converted into the CoreML format.

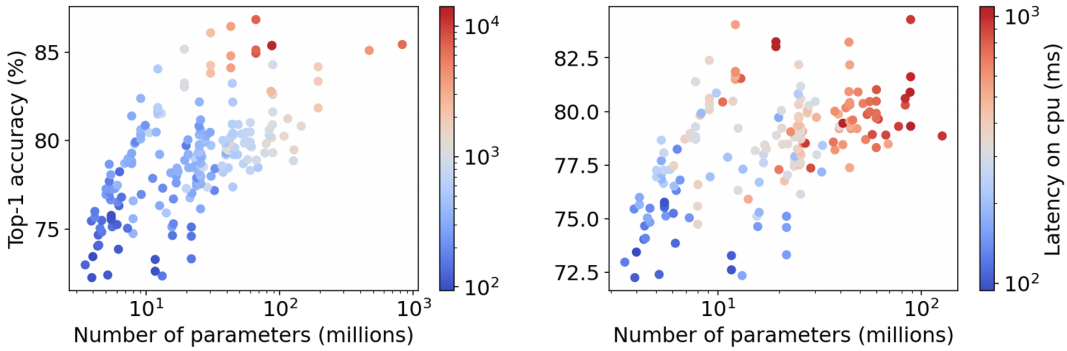


Figure 2-15: Left: Pareto curve of more than 250 image classification models on ImageNet with no added compression methods. Right: zoom on the bottom-left corner. Higher top-1 accuracy, lower latency and lower number of parameters is better. Timings were computed on a Quad-Core Intel Core i7, on a single thread.

FLoating-point Operations[79] required for performing one forward pass. As others have noted (Lebedev et al., 2014; Figurnov et al., 2016; Yang et al., 2016; Louizos et al., 2017a; He et al., 2018b), these metrics are far from perfect. Indeed, parameter count and FLOPs are a loose proxy for real-world latency, throughput, memory usage (RAM), and power consumption (battery drain). In other words, not all FLOPs are equal (Dudziak et al., 2020). Symmetrically, evaluating the latency of a deep learning model based *only* on the number of operations it is able to perform per second is considered harmful Sze et al. (2020). As an illustration, we give an overview of the current empirical Pareto curve[80] of computer vision models in terms of number of parameters, accuracy and latency in Figure 2-15 with no particular optimization (torchscript, `int8`) so only the relative value of the displayed numbers matters.

The Hardware Lottery

Finally, we close this Subsection with the concept of *Hardware Lottery* (Hooker, 2020). In this essay, the author underlines the interdependency between existing hardware and fruitful research directions. Based on examples from computer science history, the author argues that hardware lotteries can delay research progress, especially given the advent of specialized hardware that increases the frictions when testing ideas that do not rely on this hardware.

[79]FLOPs denotes a number of floating point operations, whereas FLOPS denotes a number of floating point operations *per second*. Converting FLOPs to FLOPS requires a framerate, *i.e.* how many forward passes are required within one second.

[80]Obtained using the framework: `https://github.com/rwightman/pytorch-image-models`.

Smaller Models, Bigger Biases?

Hooker et al. (2020b) explored the impact of compression on image recognition models' ability to perform accurately across various human groups. In earlier work, Hooker et al. (2020a) showed that compressed image recognition models, although they maintained their accuracy overall, had trouble identifying classes that were rare in their training data. To learn whether that shortcoming translates into bias against underrepresented human types, the researchers trained models to recognize a particular class (people with blond hair), compressed them, and measured the differences in their accuracy across different types of people. This enabled them to point out some gaps in performance between compressed and uncompressed models with respect to underrepresented groups.

2.4 Conclusion

In this Chapter, we studied the over-parameterization of Neural Networks from the point of view of the deep learning practitioner, the engineer and the mathematician. While this parameter redundancy is at the core of Deep Learning and helps optimization, it presents training challenges that can be alleviated by introducing proper regularization and normalization techniques. Another way to take advantage of this redundancy is to group networks that behave identically and work in the quotient space. This idea relies on the concept of functional equivalence classes, that are in general surprisingly small and are generated only by permutations and rescalings of hidden neurons. Studying the redundancy of the network's parameters naturally leads to the question of the efficiency of such networks, hence to the question of their compression. We reviewed various compression methods, along with the success metrics, especially in terms of latency on a given target hardware. Although not completely orthogonal – for instance pruning can be viewed as architecture search and learned group convolutions as sparsity patterns – these methods can be combined and the question of selecting the right compression method (or combination thereof) for a given task and given target metrics is still an open problem.

Chapter 3

Group and Understand: Functional Equivalence Classes

In this Chapter, we aim at characterizing functional equivalence classes of neural networks. As illustrated with our contribution in Chapter 4, a better understanding of the structure of such classes would allow us to conveniently operate in the space of neural network parameterizations quotiented by the functional equivalence relation. We restrict ourselves to feedforward ReLU neural networks. We first list various definitions of the rescaling operation found in the literature and prove that they are all equivalent under the condition that each hidden neuron is connected to at least one input neuron and one output neuron. We then provide a stand-alone result for the one-hidden-layer case where we prove that two functionally equivalent parameterizations are permutation-rescaling equivalent under some *irreducibility* conditions. Next, we investigate the case with many hidden layers by relying on the algebraic and geometric properties of the rescaling operation and by considering a local version of the rescaling equivalence. Finally, noticing that the case with deeper networks is not entirely solved, we list potential future directions and motivate them with the one-hidden-layer case.

3.1 Introduction

In this work, we study the over-parameterization of neural networks from the perspective of their functional equivalence classes. More precisely, we denote R_θ the function implemented by a network parameterized with θ and call it the *realization* of the network. Then, two parameterizations θ and θ' are *functionally equivalent* if both realizations R_θ and $R_{\theta'}$ are equal. As explained in Section 2.2.5, members of the same functional equivalence class behave differently under the action of common methods

used extensively in the Deep Learning field, for instance optimization or quantization algorithms. We argue that a better understanding of such classes may lead for instance to refined or novel optimization or quantization schemes, see Chapter 4.

We begin by recalling various definitions of rescaling equivalence for ReLU networks found in the literature as listed in Subsection 2.2.1, namely at the layer (Definition 2.2.2), neuron (Definition 2.2.3) and path level (Definition 2.2.4). Note that we introduce later in this Chapter our own definition of rescaling equivalence that we call trajectory-wise (Definition 3.3.4). Our first contribution is to reconcile these four definitions for *admissible* parameterizations, as proven in Appendix A.1.1. More formally, we denote by $\sim_{\text{Layer}}$ (resp. $\sim_{\text{Neuron}}$, $\sim_{\text{Path}}$, $\sim_{\text{Trajectory}}$) the equivalence relation induced by the layer (resp. neuron, path, trajectory) definition, and state the result in the following Proposition.

Proposition 3.1.1. *Let θ be an admissible parameterization. Then, for every θ', admissible or not, and every $a, b \in \{\text{Layer}, \text{Neuron}, \text{Path}, \text{Trajectory}\}$, we have $\theta \sim_a \theta' \iff \theta \sim_b \theta'$.*

The admissibility assumption on θ is not used to prove that the layer-wise and neuron-wise definitions are equivalent, but is used to show all the other equivalences. We end this Section by recalling and proving a well-known result (Neyshabur et al., 2015), which states that functional equivalence classes contain orbits generated by the rescaling and permutation operations (proven in Appendix A.1.2).

Proposition 3.1.2. *Two parameterizations that are permutation-rescaling equivalent are functionally equivalent.*

In the remainder of this Chapter, we investigate the reciprocal of Proposition 3.1.2: we aim at identifying *irreducibility* conditions under which two parameterizations that are functionally equivalent are also permutation-rescaling equivalent.

3.2 Reciprocal for One Hidden Layer

We focus first on the one hidden layer case, for which we derive sufficient conditions such that functional equivalence implies permutation-rescaling equivalence and investigate their necessity. The considered general architecture is depicted in Figure 3-1.

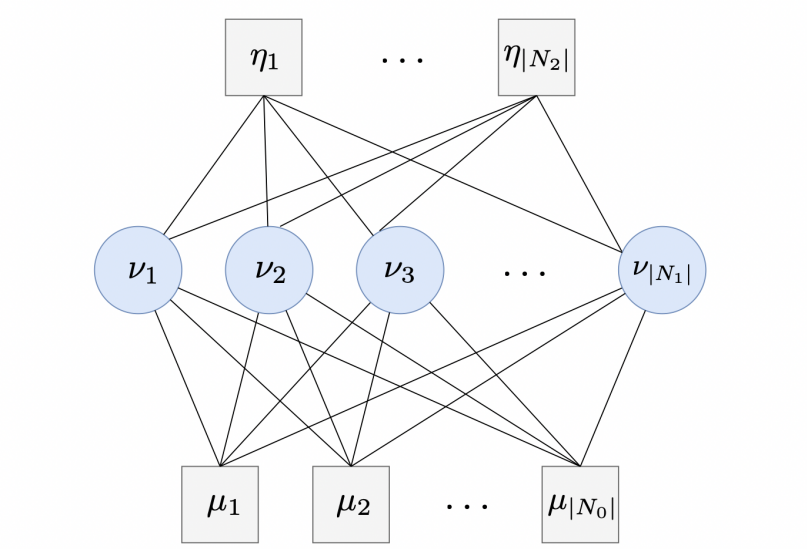


Figure 3-1: One hidden layer network architecture. Hidden neurons are depicted as circles whereas input and output neurons are displayed as squares.

3.2.1 Irreducible Parameterizations

We first define *irreducible* parameterizations by introducing the concepts of dead, twin and co-dependant neurons for one hidden layer architectures.

Definition 3.2.1 (Dead Neuron)**.** *Let G be a one hidden layer architecture valued with θ. Any hidden neuron $\nu \in H$ is said to be* dead *if either $w_{\bullet \to \nu} = 0$ or $w_{\nu \to \bullet} = 0$. Remark that θ is admissible if, and only if, there are no dead neurons.*

Intuitively, a dead neuron does not participate in the realization R_θ of the network. For θ admissible, we denote, for any $\nu \in H$, Γ_ν the affine hyperplane

$$\Gamma_\nu = \{x \in \mathbb{R}^{N_0} \mid \langle w_{\bullet \to \nu}, x \rangle + b_\nu = 0\}.$$

Note that Γ_ν is well-defined since $w_{\bullet \to \nu} \neq 0$ (θ is admissible).

Definition 3.2.2 (Twin Neurons)**.** *Let G be a one hidden layer architecture valued with θ admissible. Any hidden neurons $\nu_1 \neq \nu_2 \in H$ are said to be* twins *if there exists $d \neq 0$ such that $w_{\bullet \to \nu_1} = d w_{\bullet \to \nu_2}$ and $b_{\nu_1} = d b_{\nu_2}$. Equivalently, the separating hyperplanes are equal: $\Gamma_{\nu_1} = \Gamma_{\nu_2}$.*

Intuitively, two twin neurons either (1) *fire* in the same half-space or (2) *fire* in complementary half-spaces. Note that we can properly define twin neurons in Definition 3.2.4 since there are no dead neurons, hence θ is admissible.

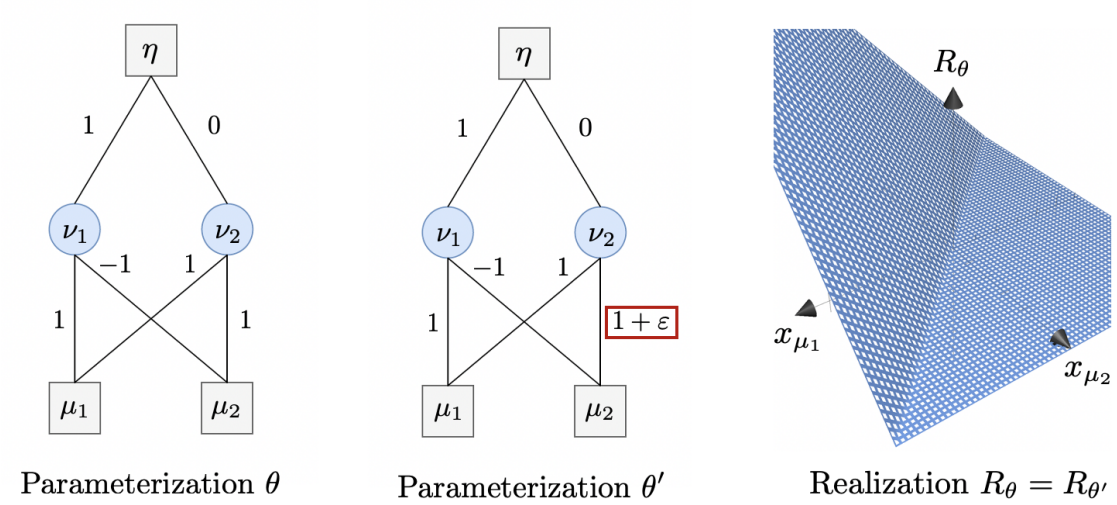


Figure 3-2: Left: example of a *dead* neuron, ν_2. Intuitively, ν_2 does not participate in the realization of the network R_θ. Center: alternative parameterization θ' such that θ and θ' are not permutation-rescaling equivalent, with $\varepsilon > 0$. Right: the same realization is implemented by θ and θ'. Squares denote input or output neurons whereas circles denote hidden neurons. Weights are specified on the connections between two neurons and all biases are set to zero (not represented here).

Definition 3.2.3 (Co-dependant Neurons)**.** *Let G be a one hidden layer architecture valued with θ admissible. For any $\eta \in N_2$, define $\mathcal{I}_\eta = \{\nu \in N_1 \mid w_{\nu\to\eta} \neq 0\}$. A set of hidden neurons $\mathcal{J} \subset \mathcal{I}_\eta$ is said to contain* co-dependant *neurons if*

$$\sum_{\nu\in\mathcal{J}} w_{\nu\to\eta} w_{\bullet\to\nu} = 0. \tag{3.1}$$

Definition 3.2.4 (Irreducible Parameterization)**.** *Let G be a one hidden layer architecture valued with θ. We say that θ is* irreducible *if there are no dead, no twin and no co-dependent (hidden) neurons in the network.*

3.2.2 Main Result

We now state our result, which is proven in Appendix A.1.3. In contrast to previous work summarized in Table 2.1 (Kůrková and Kainen, 1993; Sussmann, 1992; Albertini et al., 1993), we focus on the ReLU case and derive geometrical – hence easy to understand and to remember – irreducibility conditions.

Proposition 3.2.1. *Let G be a one-hidden layer architecture valued with θ and θ'. Assume that θ and θ' are irreducible. Then, $R_\theta = R_{\theta'}$ implies that $\theta \sim_{PS} \theta'$.*

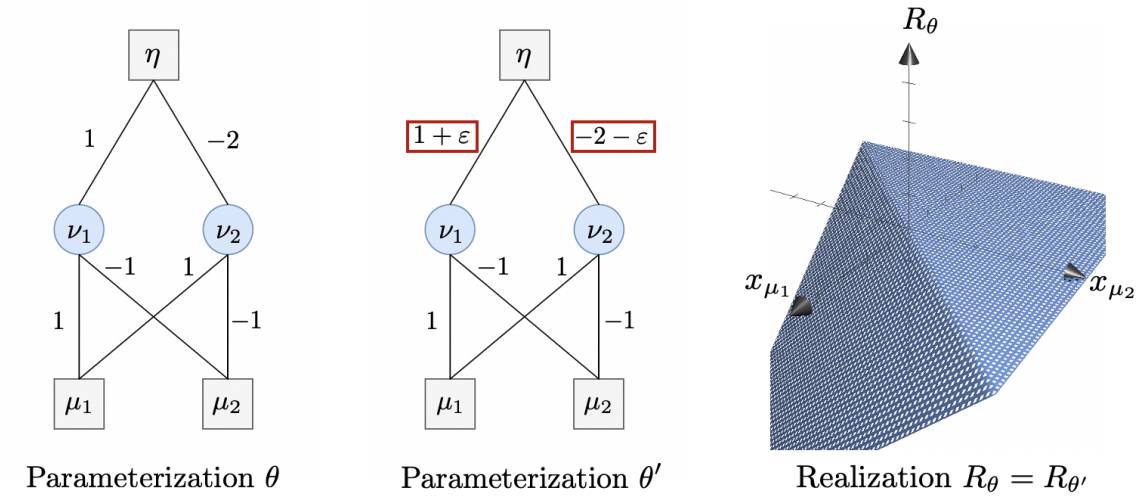


Figure 3-3: Left: example of *siamese neurons*, ν_1 and ν_2. Intuitively, ν_1 and ν_2 fire in the same half-space. Center: alternative parameterization θ' such that θ and θ' are not permutation-rescaling equivalent. Right: the same realization is implemented by θ and θ', with $\varepsilon > 0$. Squares denote input or output neurons whereas circles denote hidden neurons. Weights are specified on the connections between two neurons and all biases are set to zero (not represented here).

Proposition 3.2.1 provides sufficient conditions (irreducibility) such that functional equivalence implies permutation-rescaling equivalence. We now investigate the subtleties of non-irreducible parameters.

3.2.3 Subtleties of Non-Irreducible Parameters

We denote by ✓ the cases where it is possible to construct θ' from θ such that $R_\theta = R_{\theta'}$ but θ and θ' are not permutation-rescaling equivalent, and by ✗ the cases where it is not possible. We construct θ' from θ by setting $\theta' = \theta$ except for some edges or neurons as specified in the cases below. For the sake of conciseness and clarity, we focus on examples to illustrate our point.

✓ **Dead Neurons.** Let us assume that θ has a dead hidden neuron ν_2 as illustrated in Figure 3-2. Let $\varepsilon > 0$. Then, we set $w'_{\mu_2 \to \nu_2} = w_{\mu_2 \to \nu_2} + \varepsilon$. This modification does not change the realization $R_{\theta'} = R_\theta$, and is such that $\theta' \not\sim_{PS} \theta$.

Twin Neurons. We specify the different types of twin neurons as follows:

- ✓ **Siamese.** The separating hyperplanes are oriented in the same direction. Let us assume that θ has two siamese neurons ν_1 and ν_2 as illustrated in Figure 3-3.

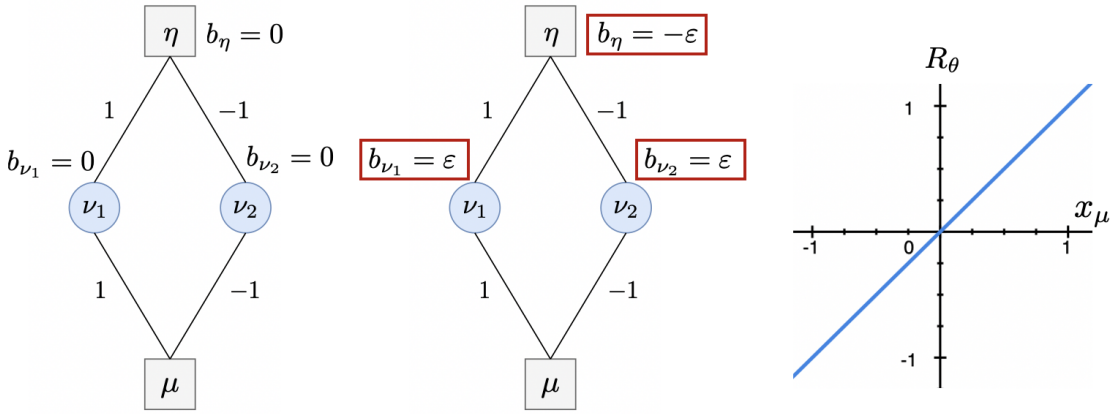


Figure 3-4: Left: example of *invisible neurons*, ν_1 and ν_2. Intuitively, ν_1 and ν_2 are such that their common breakpoint is not visible on the realization R_θ. Center: alternative parameterization θ' such that θ and θ' are not permutation-rescaling equivalent. Right: the same realization is implemented by θ and θ', with $\varepsilon > 0$. Squares denote input or output neurons whereas circles denote hidden neurons. Weights are specified on the connections between two neurons and biases are explicitly specified.

Let $\varepsilon > 0$. Since the functions y_{ν_1} and y_{ν_2} are proportional, we set $w'_{\nu_1 \to \eta} = w_{\nu_1 \to \eta} + \varepsilon$ and $w'_{\nu_2 \to \eta} = w_{\nu_2 \to \eta} - \varepsilon$ such that $w'_{\nu_1 \to \eta} + w'_{\nu_1 \to \eta} = w_{\nu_1 \to \eta} + w_{\nu_1 \to \eta}$. Then, $R_{\theta'} = R_\theta$ and θ' and θ are nor permutation-rescaling equivalent.

- **Anti-Siamese.** The separating hyperplanes are oriented in opposite directions.
  - ✓ **Invisible.** The breakpoint is not visible on the realization R_θ. In other words, there is no visible gradient change in the realization R_θ, although such changes are visible on some intermediate realizations y_ν for hidden neurons $\nu \in H$. Let us assume that θ has two invisible neurons ν_1 and ν_2 as illustrated in Figure 3-4. Let $\varepsilon > 0$. Since the breakpoint is not visible, we can translate the functions y_{ν_1} and y_{ν_2} on the horizontal axis (offsets $b'_{\nu_1} = b_{\nu_1} + \varepsilon$ and $b'_{\nu_2} = b_{\nu_2} + \varepsilon$) and translate $R_{\theta'}$ on the vertical axis (offset $b'_\eta = b_\eta - \varepsilon$) to compensate. Then, $R_\theta = R_{\theta'}$ but θ and θ' are not permutation-rescaling equivalent.
  - **Visible** The breakpoint is visible on the realization R_θ.
    - ✓ **Cousin.** There exists a third hidden neuron with a separating hyperplane that is parallel to $\Gamma_{\nu_1} = \Gamma_{\nu_2}$. Let us assume that θ has three cousin neurons ν_1, ν_2 and ν_3 as illustrated in Figure 3-5. Here, neurons

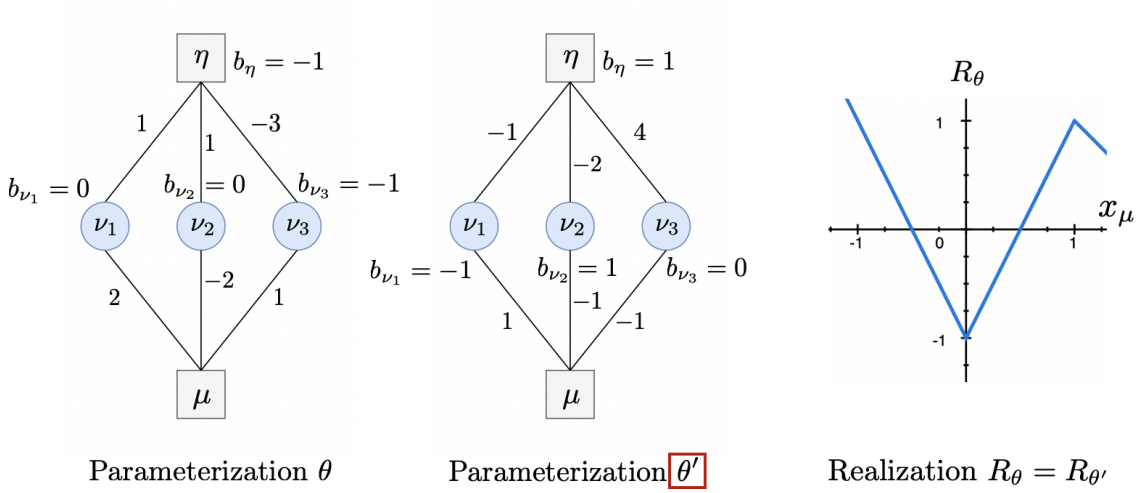


Figure 3-5: Left: example of *cousin neurons*, ν_1, ν_2 and ν_3. Intuitively, ν_1, ν_2 and ν_3 are such that two of them can ether implement the leftmost or the rightmost breakpoint (see right). Center: alternative parameterization θ' such that θ and θ' are not permutation-rescaling equivalent. Right: the same realization is implemented by θ and θ', with $\varepsilon > 0$. Squares denote input or output neurons whereas circles denote hidden neurons. Weights are specified on the connections between two neurons and biases are explicitly specified.

ν_1 and ν_2 implement the same breakpoint $x_\mu = 0$ and ν_3 the breakpoint $x_\mu = 1$. We define θ' such that ν_1 and ν_2 implement the same breakpoint $x_\mu = 1$ and nu_3 the breakpoint $x_\mu = 0$. The realization is the same and the parameters are not permutation-rescaling equivalent.

- ✗ **Non-Cousin.** All the other hidden neurons have a separating hyperplane that is not parallel to $\Gamma_{\nu_1} = \Gamma_{\nu_2}$. In that case, it is not possible to construct θ' such that $R_\theta = R_{\theta'}$ and such that θ and θ' are not permutation-rescaling equivalent. We do not provide a proof for the general case case and state this example for illustrative purposes.

The Notion of Locality. As illustrated in Figures 3-2, 3-3 and 3-4 for dead, siamese and invisible neurons, it is possible to construct θ' arbitrarily close to θ such that $R_\theta = R'_\theta$ but θ and θ' are not permutation-rescaling equivalent. This is not the case for cousin neurons. In the next sections, we will focus on local changes as they allow to manipulate the realization in a more algebraic way.

3.3 Algebraic and Geometric Tools

We now wish to state irreducibility conditions for deeper networks. However, the proof of Proposition 3.2.1 relies on geometrical arguments that are not readily adaptable for many hidden layers. Hence, we present in Section 3.3 some algebraic and geometrical to tackle the (local) case of deeper networks in Section 3.4.

3.3.1 An Algebraic Expression of R_θ and its Consequences

We express the realization of the ReLU network R_θ in a more algebraic way than the traditional definition relying on iterative composition of layer-wise functions of the type $z \mapsto \sigma(zW + b)$ (see Equation (2.9)). This allows us to formally state the well-known piecewise affine nature of $x \mapsto R_\theta(x)$ and the lesser-known piecewise polynomial nature of $\theta \mapsto R_\theta(x)$. A similar formula[1] is stated without taking the biases into account in Meng et al. (2018) whereas Balduzzi et al. (2018) (Lemma A.2) perform analogous computations for gradient computations, still without biases.

As defined in the Notations, we denote $\mathcal{P}$ the set of full paths connecting any input neuron to any output neuron and $\mathcal{Q}$ the set of partial paths from any hidden neuron to any output neuron. For simplicity, we will denote a full path $p = (\nu_0, \dots, \nu_L) \in \mathcal{P}$ by $p = (p_0, \dots, p_L)$. We denote $\mathcal{Q}_\ell = \{(p_\ell, \dots, p_L) \mid p \in \mathcal{P}\}$ the set of partial paths starting from hidden layer $0 < \ell < L$. With a sight abuse of notation, we denote by q_0 the first neuron of the path q. Finally, recall that $a_q(\theta, x)$ is the activation status of the full or partial path $q \in \mathcal{P} \cup \mathcal{Q}_1 \cup \cdots \cup \mathcal{Q}_L$ given θ and x:

$$a_q(\theta, x) \triangleq \prod_{\nu \in q \cap H} \mathbf{1}(y_\nu(\theta, x) > 0), \tag{3.2}$$

where we set $\prod_\emptyset = 1$. We now state the formula in Proposition 3.3.1 and provide a complete derivation in Appendix A.2.1.

Proposition 3.3.1. *For any parameterization $\theta = (w, b)$, any $x \in \mathbb{R}^{N_0}$ and any output neuron $\nu \in N_L$,*

$$R_\theta(x)_\nu = \sum_{\substack{p \in \mathcal{P} \\ p_L = \nu}} x_{p_0} v_p(\theta) a_p(\theta, x) + \sum_{\ell=1}^{L} \sum_{\substack{q \in \mathcal{Q}_\ell \\ q_\ell = \nu}} b_{q_0} v_q(\theta) a_q(\theta, x). \tag{3.3}$$

In Equation 3.3, the ReLU activation function is accounted for in the activation

[1] A proof does not seem to be present in the original paper.

statuses $a_p(\theta, x)$ and $a_q(\theta, x)$, and the path-sum formulation allows us to better disentangle the dependency on the input x and the dependency on the parameters θ in the terms $x_{p_0} v_p(\theta)$. A necessary (but not sufficient) condition for the slope of the piecewise affine function $x \mapsto R_\theta(x)$ to change is that at least one of the $a_p(\theta, x), p \in \mathcal{P} \cup \mathcal{Q}$ toggles. We prevent such cases by restricting ourselves to the domain $\mathcal{X}_\theta$ defined below. Similarly, we will only consider the piecewise polynomial function $\theta \mapsto R_\theta(x)$ for fixed inputs $x \in \mathcal{X}_\theta$. The proof can be found in Appendix A.2.1. Such a formula is useful for computing the gradient of $\theta \mapsto R_\theta(x)$ as explained in Proposition A.2.1.

Proposition 3.3.2. *Let $\mathcal{U}(x, \theta)$ be the set of all neighborhoods $U \subset \mathbb{R}^{N_0} \times \mathbb{R}^{E \cup H}$ of (x, θ). We define, for any parameterization θ,*

$$\mathcal{X}_\theta \triangleq \{x \in \mathbb{R}^{N_0} \mid \exists U \in \mathcal{U}(x, \theta), \forall p \in \mathcal{P} \cup \mathcal{Q}, \forall (x', \theta') \in U, a_p(\theta, x) = a_p(\theta', x')\}. \quad (3.4)$$

Then, $\mathcal{X}_\theta$ is open. Moreover, let θ be a parameterization and $x \in \mathcal{X}_\theta$. Then, for each full or partial path $p \in \mathcal{P} \cup \mathcal{Q}$, the function $x' \mapsto a_p(\theta, x')$ (resp. $\theta' \mapsto a_p(\theta', x)$) is locally constant in the neighborhood of x (resp. in the neighborhood of θ).

Such a set if useful for instance for specifying where the gradient of $\theta \mapsto R_\theta(x)$ is defined (see Proposition A.2.1). We next state a well-known result (Pascanu et al., 2013; Montúfar et al., 2014) about the affine piece-wise nature of $x \mapsto R_\theta(x)$ which we prove in Appendix A.2.1. We refer the reader to Subsection 2.2.2 for illustrations.

Proposition 3.3.3. *The function $x \mapsto R_\theta(x)$ is piecewise affine continuous and locally affine in the neighborhood of any $x \in \mathcal{X}_\theta$.*

We state a lesser-known result about the piece-wise polynomial nature of the function $\theta \mapsto R_\theta(x)$ which we prove in Appendix A.2.1.

Proposition 3.3.4. *For any $x \in \mathcal{X}_\theta$, the function $\theta \mapsto R_\theta(x)$ is piecewise polynomial continuous and locally polynomial in the neighborhood of any θ.*

3.3.2 Parameterizations of the Form $\theta' = \theta \odot e^\gamma$

In this Subsection, we investigate multiplicative changes of the form $\theta' = \theta \odot e^\gamma$ in the parameter space, where $\odot$ denotes pointwise multiplication, with the goal to study:

(1) The effect on the realization $R_{\theta'}$ (in this Subsection);

(2) The conditions on γ such that the perturbed parameterization θ' is rescaling equivalent to θ (in the next Subsection 3.3.3).

We restrict ourselves to a network with scalar output and no output bias for the sake of simplicity. Before this, we first prove[2] in Appendix A.2.2 that γ is uniquely defined for parameterizations θ' having the same sign as θ – we consider the extended sign as defined in the Notations which can take values in $\{-1, 0, 1\}$ – provided that we impose[3] the constraint $\gamma \in \mathrm{Supp}_\theta$. We argue that considering such multiplicative changes instead of additive ones helps addressing both goals (1) and (2).

Proposition 3.3.5. *Let θ be a parameterization. For all θ' in Sign_θ, there exists a unique $\gamma \in \mathrm{Supp}_\theta$ such that $\theta' = \theta \odot e^\gamma$.*

Next, we wish to simply express the realization $R_{\theta'}$ when $\theta' = \theta \odot e^\gamma$ is close enough to θ. We begin by defining matrices indicating the presence or absence of a given edge e in a full path p or partial path q.

Definition 3.3.1. *We define $\mathbf{P} \in \mathbb{R}^{\mathcal{P} \times E}$ such that, for every edge $e \in E$ and full path $p \in \mathcal{P}$, $(\mathbf{P})_{p,e} \triangleq \mathbf{1}(e \in p)$. Similarly, we define $\mathbf{Q} \in \mathbb{R}^{\mathcal{Q} \times E}$ such that, for every edge $e \in E$ and partial path $q \in \mathcal{Q}$. $(\mathbf{Q})_{q,e} \triangleq \mathbf{1}(e \in q)$.*

In order to consider only the non-zero terms in the two sums – full paths and partial paths – in Equation (3.3), we define $\mathbf{P}_\theta$ and $\mathbf{Q}_\theta$ as follows.

Definition 3.3.2. *Given $\theta = (w, b)$, we define*

$$\mathbf{P}_\theta \triangleq \mathrm{Diag}\left(\mathbf{1}(v_p(\theta) \neq 0)\right)_{p \in \mathcal{P}} \mathbf{P}. \tag{3.5}$$

$$\mathbf{Q}_\theta \triangleq \mathrm{Diag}\left(\mathbf{1}(b_{q_0} v_q(\theta) \neq 0)\right)_{q \in \mathcal{Q}} \mathbf{Q}. \tag{3.6}$$

Note that the term b_{q_0} only appears in $\mathbf{Q}_\theta$.

We now wish to decouple the update applied to the weights – a simple multiplicative modification of the form $\exp(\alpha)$ – to the update applied to the biases which has the form $\exp(\mathbf{S}\beta)$ to make the bridge between the weights – the space E of edges – and the biases – the space H of hidden neurons. The proof is located in Appendix A.2.2. Reasons for treating weights and biases differently will become clearer in Subsection 3.3.3.

Proposition 3.3.6. *We define the linear map $\mathbf{S}\colon \ker \mathbf{P} \to \mathbb{R}^H$ such that*

$$(\mathbf{S}\beta)_\nu \triangleq -\sum_{e \in q} \beta_e \tag{3.7}$$

[2]Using quantities defined in the remainder of this Subsection.

[3]Otherwise, we can multiply any zero weight or bias with any number without modifying it.

where q is the partial path going from ν to the output neuron η. Then,

1. $\mathbf{S}$ *is well-defined (i.e., independent of the choice of q);*
2. $\mathbf{S}$ *is an isomorphism and its inverse $\mathbf{S}^{-1} : \mathbb{R}^H \to \ker \mathbf{P}$ is such that for any $\lambda \in \mathbb{R}^H$ we have $\mathbf{S}^{-1}\lambda = \beta$ where for any edge $e = \mu \to \nu \in E$,*

$$\beta_e \triangleq \begin{cases} -\lambda_\mu & \text{if } \mu \in N_{L-1} \text{ (and } \nu = \eta) \\ \lambda_\nu - \lambda_\mu & \text{if } \mu \in N_\ell,\ 1 \leq \ell \leq L-2 \\ \lambda_\nu & \text{if } \mu \in N_0. \end{cases} \tag{3.8}$$

Now, consider $\theta = (w, b) \in \mathbb{R}^E \times \mathbb{R}^H$ a parameterization. We now refine our definition of γ given in Proposition 3.3.5 by expressing it either as $\gamma \in \mathrm{Supp}_\theta$ or by $\gamma = (\alpha, \mathbf{S}\beta)$ where $\mathbf{S}$ is defined in Proposition3.3.6, with $(\alpha, \beta) \in V_\theta$, where

$$V_\theta \triangleq \mathrm{Supp}_w \times \mathbf{S}^{-1}(\mathrm{Supp}_b) \subset \mathbb{R}^E \times \ker \mathbf{P} \subset \mathbb{R}^E \times \mathbb{R}^E. \tag{3.9}$$

Finally, we establish an explicit and local formula to describe trajectories near θ in the direction γ, which we prove in Appendix A.2.2. We use the set $\mathcal{X}_\theta$ defined in (3.4) and start from the path-sums formula (3.3).

Proposition 3.3.7. *Consider θ a parametrization and $B_\theta \triangleq \{\gamma \in \mathrm{Supp}_\theta \mid \|\gamma\|_2 \leq 1\}$. Then, for each $x \in \mathcal{X}_\theta$, there exists $\varepsilon > 0$ such that, for every $(\gamma, t) \in B_\theta \times]-\varepsilon, \varepsilon[$, the following result holds:*

$$R_\theta(\theta \odot e^{\gamma t}, x) = \sum_{p \in \mathcal{P}} x_{p_0} v_p(\theta) e^{t\mathbf{P}(\alpha)_p} a_p(\theta, x) + \sum_{q \in \mathcal{Q}} b_{q_0} v_q(\theta) e^{t\mathbf{Q}(\alpha-\beta)_q} a_q(\theta, x). \tag{3.10}$$

where we recall that $\gamma = (\alpha, \mathbf{S}\beta)$.

Recalling Definition 3.3.2, we note that the terms $\mathbf{P}\alpha$ and $\mathbf{Q}(\alpha - \beta)$ can be replaced in Equation (3.10) by $\mathbf{P}_\theta(\alpha)$ and $\mathbf{Q}_\theta(\alpha - \beta)$ respectively without changing the result since $v_p(\theta) = 0$ implies that $P_\theta(\alpha)_p = 0$ for any full path $p \in \mathcal{P}$ and similarly for any partial path $q \in \mathcal{Q}$.

3.3.3 Algebraic Characterization of Rescaling Equivalence

Given an *admissible* parameterization $\theta = (w, b)$, we characterize the set of multiplicative changes e^γ with respect to θ that preserve the rescaling equivalence when

applied to θ, leading to a more geometrical definition of rescaling equivalence in Definition 3.3.4, which we illustrate in Subsection A.2.4.

Definition 3.3.3. *Let θ be a parameterization. The vector $\gamma = (\alpha, \mathbf{S}\beta) \in \mathrm{Supp}_\theta$ is said to be* θ-compatible *or* compatible *when the dependency on θ is obvious if*

$$\alpha \in \ker(\mathbf{P}_\theta) \quad \textit{and} \quad \alpha - \beta \in \ker(\mathbf{Q}_\theta). \tag{3.11}$$

We denote by Comp_θ the set of θ-compatible vectors γ. With a slight abuse of notation, we might also say that $(\alpha, \beta) \in V_\theta$ is θ-compatible.

Remark 3.3.1. *We will constantly distinguish between the case without biases and the case with biases for the rest of this section.*

- *If $b = 0$, the condition $\alpha - \beta \in \ker(\mathbf{Q}_\theta)$ is always satisfied using Definition 3.3.2.*
- *If $b \neq 0$, the condition $\alpha - \beta \in \ker(\mathbf{Q}_\theta)$ ensures that the rescaling coefficients corresponding to non-zero biases b_ν are the same as the rescaling coefficients obtained for the weights for neuron ν.*

We now define rescaling equivalence using the previous definition 3.3.3. We prove that this definition 3.3.4 is equivalent to the three definitions of layer-, neuron- and path-wise rescaling equivalence in Proposition 3.1.1 under the condition that the considered parameterizations are admissible.

Definition 3.3.4 (Trajectory rescaling equivalence)**.** *θ and θ' are rescaling equivalent if there exists $\gamma \in \mathrm{Comp}_\theta$ such that $\theta' = \theta \odot e^\gamma$.*

Remark 3.3.2. *Let θ be an* admissible *parameterization. Consider $\theta' \in \mathrm{Sign}_\theta$ and γ the unique perturbation associated with θ such that $\theta' = \theta \odot e^\gamma$ (Proposition 3.3.5).*

1. *Using Proposition 3.1.1, we have $\theta' \sim_S \theta$ if, and only if, $\gamma \in \mathrm{Comp}_\theta$.*
2. *If $\gamma \in \mathrm{Comp}_\theta$, $\theta' \sim_S \theta$ and using Proposition 3.1.2, we have $R_{\theta \odot e^\gamma} = R_\theta$.*

We finally define the set $\mathrm{Sincomp}_\theta$ of strongly incompatible vectors γ for reasons that will become clearer in Section 3.4.

Definition 3.3.5. *The vector $\gamma = (\alpha, \mathbf{S}\beta) \in \mathrm{Supp}_\theta$ is said to be* strongly θ-incompatible *or simply* strongly incompatible *if*

$$\alpha \in \ker(\mathbf{P}_\theta)^\perp \quad \textit{and} \quad \|\gamma\|_2 = 1. \tag{3.12}$$

We denote by $\mathrm{Sincomp}_\theta$ the set of strongly θ-incompatible vectors γ.

We prove the following properties on Sincomp_θ in Appendix A.2.3.

Proposition 3.3.8. *We denote by $\overline{Comp}_\theta$ the complementary set of $Comp_\theta$. Then, the two following properties hold for $Sincomp_\theta$.*

(1) $Sincomp_\theta \subset \overline{Comp}_\theta$ for any admissible *parameterization θ.*

(2) $Sincomp_\theta$ is compact for any parameterization θ.

We provide an illustration of the tools designed in this Section in Appendix A.2.4 for particular networks (Stem and Sawtooth) to showcase their usefulness.

3.4 Locally Identifiable Parameterizations

We now investigate a possible reciprocal of Proposition 3.1.2 using the tools designed in Section 3.3. We study parameterizations for which the property that functional equivalence implies rescaling equivalence holds *locally*. We first give definitions of locally identifiable and non-identifiable parameterizations in Subsection 3.4.1 and derive a necessary condition as well as a sufficient condition for local identifiability in Subsections 3.4.2 and 3.4.3.

3.4.1 Definition of Locally Identifiable Parameterizations

We first give an intuitive definition of locally identifiable parameters for which the property that functional equivalence implies rescaling equivalence holds *locally*.

Definition 3.4.1. *Consider θ an admissible parameterization.*

- *θ is* locally identifiable *if there exists a neighborhood Ω of θ such that for all $\theta' \in \Omega$, $R_{\theta'} = R_\theta$ implies $\theta' \sim_S \theta$.*
- *θ is* locally non-identifiable *if it is not locally identifiable,* i.e. *for all neighborhood Ω of θ, there exists $\theta' \in \Omega$ such that $R_{\theta'} = R_\theta$ and $\theta' \not\sim_S \theta$.*

Let us now study the notions of locally non-identifiable and restricted locally non-identifiable parameterizations in order to leverage our geometrical definition of rescaling equivalence in Definition 3.3.3 that uses multiplicative changes defined in Proposition 3.3.5. Indeed, multiplicative changes only operate at constant sign.

Definition 3.4.2. *Consider θ an admissible parameterization. θ is* restricted locally non-identifiable *if for all neighborhood Ω of θ, there exists $\theta' \in \Omega \cap \text{Sign}_\theta$ such that $R_{\theta'} = R_\theta$ and $\theta' \not\sim_S \theta$.*

Proposition 3.4.1. *Let θ be restricted locally non-identifiable. Then, θ is locally non-identifiable. Moreover, if θ has full support,* i.e. $\mathrm{supp}(\theta) = \mathbb{R}^E$ *and if* Sign_θ *is open, then θ is locally non-identifiable if, and only if, it is restricted locally non-identifiable.*

Note that a locally non-identifiable parameterization is not necessarily restricted locally non-identifiable: consider an architecture with two input neurons μ_1, μ_2, two hidden neurons ν_1, ν_2, two output neurons η_1, η_2 and set: $w_{\mu_1\to\nu_1} = 1, w_{\mu_1\to\nu_2} = 1$, $w_{\mu_2\to\nu_1} = -1, w_{\mu_2\to\nu_2} = -1$ and $w_{\nu_1\to\eta_1} = 1, w_{\nu_1\to\eta_2} = 0$ and $w_{\nu_2\to\eta_1} = 0, w_{\nu_2\to\eta_2} = 1$. Here, θ is locally non-identifiable. Indeed, neurons ν_1 and ν_2 are twins (see the example of Figure 3-3). However, θ is not restricted locally non-identifiable since we cannot get θ' such that: θ' has the same support as θ, $\theta' \not\sim_S \theta$ and $R_\theta = R_{\theta'}$.

We now provide an equivalent definition of restricted locally non-identifiable parameterizations in terms of strongly incompatible multiplicative changes (see Definition 3.3.5). The proof is given in Appendix A.3.1.

Proposition 3.4.2. *Consider $\theta = (w, b)$ an* admissible *parameterization. The following properties are equivalent:*

1. *θ is restricted locally non-identifiable.*

2. *for all $\varepsilon > 0$, there exists $\tau \in]0, \varepsilon[$ and $\gamma \in Sincomp_\theta$ such that $R_{\theta \odot e^{\gamma\tau}} = R_\theta$.*

3.4.2 Sufficient Condition for Restricted Local Identifiability

We now provide a necessary condition for an admissible parameterization θ to be a *restricted* locally non-identifiable parameterization, *i.e.* a sufficient condition for θ to be restricted locally identifiable. The proof can be found in Appendix A.3.2.

Proposition 3.4.3. *Let $\theta = (w, b)$ be an admissible, restricted locally non-identifiable parameterization. Then, there exists $\gamma \in Sincomp_\theta$ such that, for all $x \in \mathcal{X}_\theta$,*

$$\langle \nabla_\theta R_\theta(x), \theta \odot \gamma \rangle = 0. \tag{3.13}$$

Next, let us contextualize Proposition 3.4.3 for the one-hidden-layer case. Recall that our result in Proposition 3.2.1) states that, for irreducible parameterizations, functional equivalence implies permutation-rescaling equivalence. In particular, irreducible parameterizations are locally identifiable. In particular, as proven in Appendix A.3.2, restricted locally non-identifiable parameterizations are non irreducible.

Proposition 3.4.4. *Let G be a one hidden layer architecture with scalar output valued with an admissible parameterization $\theta = (w, b)$ such that $b = 0$ (no biases). We assume that θ is restricted locally non-identifiable. Then there exists at least two twin hidden neurons as defined in Definition 3.2.2.*

3.4.3 Sufficient Condition for Local Identifiability

We now adopt a pure geometrical point of view to provide another sufficient condition for Locally Identifiable (but non necessarily restricted) parameterizations in Proposition 3.4.7. Here again, we focus on the scalar output case.

Definition 3.4.3. *Let θ be a network parameterization and $x \in \mathcal{X}_\theta$. We define $u(\theta) \in \mathbb{R}^{\mathcal{P} \cup \mathcal{Q}}$ such that, for all $q \in \mathcal{P} \cup \mathcal{Q}$,*

$$u_q(\theta) = \begin{cases} v_q(\theta) & \text{if } q \in \mathcal{P} \\ b_{q_0} v_q(\theta) & \text{if } q \in \mathcal{Q}. \end{cases} \tag{3.14}$$

For each $x \in \mathcal{X}_\theta$, we further define $c(\theta, x) \in \mathbb{R}^{\mathcal{P} \cup \mathcal{Q}}$ such that, for all $q \in \mathcal{P} \cup \mathcal{Q}$,

$$c_q(\theta, x) = \begin{cases} x_{q_0} a_q(\theta, x) & \text{if } q \in \mathcal{P} \\ a_q(\theta, x) & \text{if } q \in \mathcal{Q}. \end{cases} \tag{3.15}$$

With these notations and recalling that we focus on the scalar output case, we have, for all $x \in \mathbb{R}$, $R_\theta(x) = \langle u(\theta), c(\theta, x) \rangle$.

We now define useful sets to only manipulate finitely many input points, as proven in Appendix A.3.3.

Proposition 3.4.5. *Let θ be a parameterization. There exists a neighborhood Ω_θ of θ and a finite set of points $\mathcal{Z}_\theta \subset \mathcal{X}_\theta$ that satisfy the following properties:*

- *The following equality holds*

$$\operatorname*{Span}_{z \in \mathcal{Z}_\theta}(c(\theta, z)) = \operatorname*{Span}_{x \in \mathcal{X}_\theta}(c(\theta, x)). \tag{3.16}$$

- *For all $\theta' \in \Omega_\theta$ and for all $z \in \mathcal{Z}_\theta$, $c(\theta', z) = c(\theta, z)$.*

We now express functional equivalence for two close parameterizations in pure geometrical terms in the space of full and partial paths, as proven in Appendix A.3.3.

Proposition 3.4.6. *Let θ be a parameterization, and Ω_θ, $\mathcal{Z}_\theta$ given as in Proposition 3.4.5. Denote by $\mathcal{C}_\theta$ the vector space*

$$\mathcal{C}_\theta = \operatorname*{Span}_{x \in \mathcal{X}_\theta}(c(\theta, x)). \tag{3.17}$$

Then, for any $\theta' \in \Omega_\theta$, if $R_\theta = R_{\theta'}$, then $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$.

Recall that in Definition 2.2.4, we expressed rescaling equivalence between θ and θ' as $u(\theta) = u(\theta')$ (using notations from Definition 3.4.3). We are now ready to state a sufficient condition ensuring that θ is locally identifiable relying on geometrical considerations in the space of the full and partial paths. The corresponding Proposition is proven in Appendix A.3.3.

Proposition 3.4.7. *Let θ be a parameterization. We assume that there exists a neighborhood Ω of θ such that, for all $\theta' \in \Omega$, $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$ implies that $u(\theta) = u(\theta')$. Then, θ is locally identifiable.*

A simple corollary of Proposition 3.4.7 is given below and proven in Appendix A.3.3.

Proposition 3.4.8. *Let θ be an admissible parameterization such that $\mathcal{C}_\theta^\perp = \{0\}$. Then, θ is locally identifiable.*

Next, we leverage our standalone proof for the one-hidden-layer case showing that, for irreducible parameterizations, functional equivalence implies permutation-rescaling equivalence (Proposition 3.2.1) to check for consistency. We apply Proposition 3.4.7 to the one hidden layer case (see Proposition 3.2.1) to verify that any irreducible parameterization is indeed locally identifiable. We first prove the following Proposition in Appendix A.3.3.

Proposition 3.4.9. *Let G be a one hidden layer architecture with scalar output valued with an admissible parameterization θ. We further assume that there are no twin neurons,* i.e. *that θ is irreducible as defined in 3.2.4. Then, $\mathcal{C}_\theta^\perp = \{0\}$.*

Using Proposition 3.4.8, we deduce that the parameterization θ given in Proposition 3.4.9 is locally identifiable. Moreover, taking a closer look at the proofs, we conjecture that if would be possible to obtain a *characterization* of locally identifiable parameterizations with the condition $\mathcal{C}_\theta^\perp = \{0\}$ for one-hidden-layer networks.

3.4.4 Current Limitations and Discussions

We list below the limitations of our current approach and formulate various conjectures that deserve further interest. First and foremost, Propositions 3.4.7 and 3.4.3 only encompass the case of *local* identifiability. Hence, even if we were to properly *characterize* locally identifiable parameterizations, we still wouldn't take the permutations into account or *non-local degenerate* cases as listed in Subsection 3.2.3. We argue that studying *global* identifiability is out of the scope of this work.

Next, we list a few directions that would improve our local understanding of local identifiability, namely we wish to (1) work in the *path space* and only manipulate $\mathrm{Im}(u)$ in Proposition 3.4.7 and (2) sample a finite set of input points $\mathcal{Z}_\theta$ such that, if θ and θ' are close enough and coincide on $\mathcal{Z}_\theta$, they coincide everywhere, *i.e.* $R_\theta = R_{\theta'}$. Using this fact, we would have a necessary and sufficient condition for local identifiability in Proposition 3.4.7. Before jumping into the details, we mention a similar line of work in this direction (Malgouyres and Landsberg, 2018; Malgouyres, 2020), where the authors consider structured linear networks, without biases and without activation functions as well as per-layer (instead of neuron-wise) rescalings. See Section 2.2.4 in the Related Work for details. In particular Malgouyres and Landsberg (2018) also manipulate the quantities close to $u(\theta)$ and $\mathrm{Im}(u)$.

First, assuming that a θ satisfies a certain condition $(*)$, we adopt a more geometrical view about the manifold $\mathrm{Im}(u) - u(\theta)$ (in the space of paths $\mathbb{R}^{\mathcal{P}}$) relatively to $\mathcal{C}_\theta$ in the neighborhood of $u(\theta)$. The proof is given in Appendix A.3.4. Note that the knowledge of $u(\theta)$ allows to recover θ up to rescalings, see Definition 2.2.4.

Proposition 3.4.10. *Assume that θ satisfies the following property*

$$\forall \varepsilon > 0, \exists \eta > 0, \forall \theta', \|u(\theta') - u(\theta)\| < \eta \implies \exists \theta'', u(\theta'') = u(\theta'), \|\theta'' - \theta\| < \varepsilon \qquad (*)$$

Then, the two following properties are equivalent:

(i) There exists $\varepsilon > 0$ such that, for all $\theta' \in B(\theta, \varepsilon)$ such that $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$, we have $u(\theta') = u(\theta)$.

(ii) There exists $\eta > 0$ such that $(\mathrm{Im}(u) - u(\theta)) \cap B_\infty(0, \eta) \cap \mathcal{C}_\theta^\perp = \{0\}$.

Next, we obtain necessary and sufficient condition on local identifiability, provided that our sampling conjecture (item (1) in the following Proposition) holds. The Proposition is proven in Appendix A.3.4.

Proposition 3.4.11. *Let assume that θ satisfies $(*)$. Let us assume further that there exists $\varepsilon, r > 0$ and a finite set $\mathcal{Z}_\theta \subset \mathcal{X}_\theta$ such that $\tilde{\mathcal{Z}}_\theta \triangleq \cup_{z \in \mathcal{Z}_\theta} B(z, r) \subset \mathcal{X}_\theta$ and such that, for all $\theta' \in B(\theta, \varepsilon)$,*

(1) If for all $z \in \tilde{\mathcal{Z}}_\theta$, $R_\theta(z) = R_{\theta'}(z)$, then $R_\theta = R_{\theta'}$.

(2) For all $z \in \tilde{\mathcal{Z}}_\theta$, $c(\theta', z) = c(\theta, z)$

Then the two following properties are equivalent:

(i) θ is locally identifiable.

(ii) There exists $\eta > 0$ such that $(\mathrm{Im}(u) - u(\theta)) \cap B_\infty(0, \eta) \cap \mathcal{C}_\theta^\perp = \{0\}$.

We believe that Property $(*)$ holds for any parameterization θ. However, it is still a conjecture at this time. We provide below concrete examples of parameterizations θ that satisfy $(*)$, as proven in Appendix A.3.4.

Proposition 3.4.12. *If all coefficients of $\theta = (w, b)$ are strictly positive, then $(*)$ defined in Proposition 3.4.10 holds.*

We conjecture that assumption (1) of Proposition 3.4.11 holds for any parameterization θ. On the other hand, assumption (2) is already proven in Proposition 3.4.5.

3.5 Conclusion

In this Chapter, we studied functional equivalence classes of ReLU neural networks. We first reconciled the various definitions of rescaling equivalence found in the literature under the condition that the underlying parameterizations are admissible. We then provided sufficient conditions – namely no dead or twin hidden neurons – in the one-hidden-layer case such that functional equivalence implies permutation-rescaling equivalence. Next, to investigate the case of deeper networks, we designed algebraic and geometrical tools to characterize the rescaling equivalence with multiplicative changes in the parameter space. Finally, we provide a sufficient condition for restricted local identifiability and another sufficient condition for local identifiability. Acknowledging that the case with deeper network is not entirely solved, we motivate and contextualize these conditions with particular networks and list interesting directions for further research.

Chapter 4

Learning to Balance the Energy with Equi-normalization

In this Chapter, we propose an application of the theoretical considerations on functional equivalence classes in Chapter 3. We restrict ourselves to describing such classes with the sole rescalings (see for instance Definition 2.2.2) for practicality. As proven in Proposition A.1.1 and illustrated in Figure 4-1, rescalings preserve the function implemented by the network and allow us to manipulate networks by considering their equivalence classes or *orbits*. Inspired by the Sinkhorn-Knopp algorithm, we introduce a fast iterative method for selecting, within one equivalence class, the representant that minimizes the L_2 norm of its weights. It provably converges to a unique solution. Interleaving our algorithm with SGD during training improves the test accuracy and amounts to perform SGD in the set of parameters quotiented by the rescaling equivalence relation. For small batches, our approach offers an alternative to batch and group normalization on CIFAR-10 and ImageNet with a ResNet-18.

4.1 Introduction

Deep Neural Networks (DNNs) have achieved outstanding performance across a wide range of empirical tasks such as image classification (Krizhevsky et al., 2012), image segmentation (He et al., 2017a), speech recognition (Hinton et al., 2012a), natural language processing (Collobert et al., 2011) or playing the game of Go (Silver et al., 2017). These successes have been driven by the availability of large labeled datasets such as ImageNet (Deng et al., 2009), increasing computational power and the use of deeper models (He et al., 2015a).

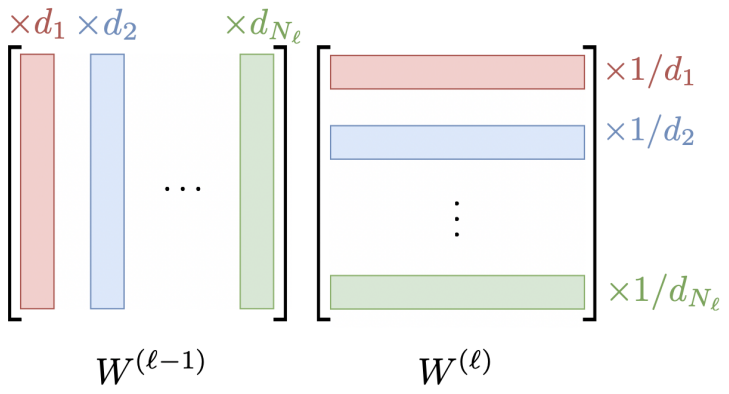


Figure 4-1: Matrices $W^{(\ell)}$ and $W^{(\ell+1)}$ are updated by multiplying the columns of the first matrix with rescaling coefficients. The rows of the second matrix are inversely rescaled to ensure that the product of the two matrices is unchanged. The rescaling coefficients are strictly positive to ensure functional equivalence when the matrices are interleaved with ReLUs. This rescaling is applied iteratively to each pair of adjacent matrices. In this Chapter, we address the more complex cases of biases, convolutions, max-pooling or skip-connections to be able to balance modern CNN architectures. This Figure is the layer analogous of neuron-wise rescalings depicted in Figure 2-4.

Although the expressivity of the function computed by a neural network grows exponentially with depth (Pascanu et al., 2013; Telgarsky, 2016), in practice deep networks are vulnerable to both over- and underfitting (Glorot and Bengio, 2010; Krizhevsky et al., 2012; He et al., 2015a). Widely used techniques to prevent DNNs from overfitting include regularization methods such as weight decay (Krogh and Hertz, 1992), Dropout (Hinton et al., 2012b) and various data augmentation schemes (Krizhevsky et al., 2012; Simonyan and Zisserman, 2014; Szegedy et al., 2014). Underfitting can occur if the network gets stuck in a local minima, which can be avoided by using stochastic gradient descent algorithms (Bottou, 2010; Duchi et al., 2011; Sutskever et al., 2013; Kingma and Ba, 2014), sometimes along with carefully tuned learning rate schedules (He et al., 2015a; Goyal et al., 2017).

Training deep networks is particularly challenging due to the vanishing/exploding gradient problem. It has been first studied for Recurrent Neural networks (RNNs) (Hochreiter and Bengio, 2001) as well as standard feedforward networks (He et al., 2015c; Mishkin and Matas, 2016). After a few iterations, the gradients computed during backpropagation become either too small or too large, preventing the optimization scheme from converging. This is alleviated by using non-saturating activation functions such as rectified linear units (ReLUs) (Krizhevsky et al., 2012) or better initialization schemes preserving the variance of the input across layers (Glorot and Bengio, 2010; Mishkin and Matas, 2016). Failure modes that prevent the training

from converging have been theoretically studied by Hanin and Rolnick (2018).

Two techniques in particular were key for vision models to achieve "super-human" accuracy. Batch Normalization (BN) was developed to train Inception networks (Ioffe and Szegedy, 2015). It introduces intermediate layers that normalize the features by the mean and variance computed within the current batch. BN is effective in reducing training time, provides better generalization capabilities after training and diminishes the need for a careful initialization. Network architectures such as ResNet (He et al., 2015a) and DenseNet (Huang et al., 2017a) use skip connections along with BN to improve the information flow during both the forward and backward passes.

However, BN has some limitations. In particular, BN only works well with sufficiently large batch sizes (Ioffe and Szegedy, 2015; Wu and He, 2018). For sizes below 16 or 32, the batch statistics have a high variance and the test error increases significantly. This prevents the investigation of higher-capacity models because large, memory-consuming batches are needed in order for BN to work in its optimal range. In many use cases, including video recognition (Carreira and Zisserman, 2017) and image segmentation (He et al., 2017a), the batch size restriction is even more challenging because the size of the models allows for only a few samples per batch. Another restriction of BN is that it is computationally intensive, typically consuming 20% to 30% of the training time. Variants such as Group Normalization (GN) (Wu and He, 2018) cover some of the failure modes of BN.

In this Chapter, we introduce a novel algorithm to improve both the training speed and generalization accuracy of networks by using their over-parameterization to regularize them. In particular, we focus on neural networks that are positive-rescaling equivalent (Neyshabur et al., 2015), *i.e.* whose weights are identical up to positive scalings and matching inverse scalings. The main principle of our method, referred to as *Equi-normalization* (ENorm), is illustrated in Figure 4-1 for the fully-connected case. We rescale two consecutive matrices in order to minimize the joint p-norm of these two matrices under the constraint of preserving the function implemented by the network. We conjecture that this particular choice of rescaling coefficients ensures a smooth propagation of the gradients during training.

4.2 Related Work

This section reviews methods improving neural network training and compares them with ENorm. Since there is a large body of literature in this research area, we focus on the works closest to the proposed approach and refer the reader to Section 2.1.

From early works, researchers have noticed the importance of normalizing the input of a learning system, and by extension the input of any layer in a DNN (LeCun et al., 1998a). Such normalization is applied either to the weights or to the activations. On the other hand, several strategies aim at better controlling the geometry of the weight space with respect to the loss function. Note that these research directions are not orthogonal. For example, explicitly normalizing the activations using BN has smoothing effects on the optimization landscape (Santurkar et al., 2018b).

Normalizing Activations. Batch Normalization (Ioffe and Szegedy, 2015) normalizes the activations by using statistics computed along the batch dimension. As stated in the introduction, the dependency on the batch size leads BN to underperform when small batches are used. Batch Renormalization (BR) (Ioffe, 2017) is a follow-up that reduces the sensitivity to the batch size, yet does not completely alleviate the negative effect of small batches. Several batch-independent methods operate on other dimensions, such as Layer Normalization (channel dimension) (Ba et al., 2016) and Instance-Normalization (sample dimension) (Ulyanov et al., 2017). Parametric data-independent estimation of the mean and variance in every layer is investigated by Arpit et al. (2016). However, these methods are inferior to BN in standard classification tasks. More recently, Group Normalization (GN) (Wu and He, 2018), which divides the channels into groups and normalizes independently each group, was shown to effectively replace BN for small batch sizes on vision tasks.

Normalizing Weights. Early weight normalization techniques only served to initialize the weights before training (Glorot and Bengio, 2010; He et al., 2015c). These methods aim at keeping the variance of the output activations close to one along the whole network, but the assumptions made to derive these initialization schemes may not hold as training evolves. More recently, Salimans and Kingma (2016) propose a polar-like re-parametrization of the weights to disentangle the direction from the norm of the weight vectors. Note that Weight Norm (WN) does require mean-only BN to get competitive results as well as a greedy layer-wise initialization.

Optimization Landscape. Generally, in the parameter space, the loss function moves quickly along some directions and slowly along others. To account for this anisotropic relation between the parameters of the model and the loss function, *natural gradient* methods have been introduced (Amari, 1998). They require storing and inverting the $N \times N$ *curvature matrix*, where N is the number of network parameters.

Several works approximate the inverse of the curvature matrix to circumvent this problem (Pascanu and Bengio, 2013; Marceau-Caron and Ollivier, 2016; Martens and Grosse, 2015). Another method called Diagonal Rescaling (Lafond et al., 2017) proposes to tune a particular re-parametrization of the weights with a block-diagonal approximation of the inverse curvature matrix. Qi et al. (2020) propose to enforce *near isometric* weights for both initialization and training. Finally, Neyshabur et al. (2015) propose a rescaling invariant path-wise regularizer and use it to derive Path-SGD, an approximate steepest descent with respect to the path-wise regularizer.

Positioning. Unlike BN, Equi-normalization focuses on the weights and is independent of the concept of batch. Like Path-SGD, our goal is to obtain a balanced network ensuring a good back-propagation of the gradients, but our method explicitly re-balances the network using an iterative algorithm instead of using an implicit regularizer. Moreover, ENorm can be readily adapted to the convolutional case whereas Neyshabur et al. (2015) restrict themselves to the fully-connected case. In addition, the theoretical computational complexity of our method is much lower than the overhead introduced by BN or GN (see Section 4.5). Besides, WN parametrizes the weights in a polar-like manner, $w = g \times v/|v|$, where g is a scalar and v are the weights, thus it does not balance the network but individually scales each layer. Finally, Huang et al. (2017b) and Yuan and Xiao (2019) use a different projection technique to select the representant based on Riemannian gradients while Bernstein et al. (2020) derive a multiplicative version of the Adam optimizer.

4.3 Equi-normalization

We first define Equi-normalization in the context of simple feed forward networks that consist of two operators: linear layers and ReLUs. The algorithm is inspired by Sinkhorn-Knopp and is designed to balance the energy of a network, *i.e.*, the euclidian p-norm of its weights, while preserving its function. As shown in Theorem 4.3.1, the algorithm converges to a unique canonical network parametrization that minimizes the euclidian p-norm of its weights among equivalent networks. When not ambiguous, we may denote by *network* a parametrization θ of a given network architecture G.

4.3.1 Notation and Definitions

As defined in the Notations, we consider a ReLU network with L linear layers, taking as input a row vector $x \in \mathbb{R}^{N_0}$. For the sake of exposition, we omit a bias term at this

stage. We rely on the layer definition of rescalings (see 2.2.2) since it is closer to the implementation of the proposed algorithm. Recall that rescaling equivalence implies functional equivalence as shown in Proposition A.1.1. As detailed in Chapter 3, a functional equivalence class is not entirely described by rescaling operations. For example, permutations of neurons inside a layer also preserve functional equivalence, but do not change the gradient.

In what follows our objective is to seek a canonical parameter vector that is rescaling equivalent to a given parameter vector. The same objective under a *functional equivalence* constraint is beyond the scope of this Chapter, as there exist degenerate cases where functional equivalence does not imply rescaling equivalence, even up to permutations, as investigated in Chapter 3.

4.3.2 Objective Function: Canonical Representation

Given a network parameterization θ and $p > 0$, we define the global p-norm of its weights as $L_p(\theta) = \sum_{\ell=1}^{L} \|W^{(\ell)}\|_p^p$, where $W^{(\ell)} \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$. We are interested in minimizing L_p inside an equivalence class of neural networks in order to exhibit a unique canonical element per equivalence class. Recalling that N_ℓ denotes all the neurons of layer $\ell \in [\![0, L]\!]$, we denote the *rescaling coefficients* as $d_\ell \in [0, +\infty]^{N_\ell}$ or as diagonal matrices[1] $D^{(\ell)} = \mathrm{Diag}(d_\ell) \in \mathcal{D}(|N_\ell|)$ for $\ell \in [\![1, L-1]\!]$. Fixing the weights $W^{(\ell)}$, we refer to $\left(D^{(\ell-1)}\right)^{-1} W^{(\ell)} D^{(\ell)}$ as the *rescaled weights*, and seek to minimize their euclidian p-norm as a function of the rescaling coefficients:

$$\varphi(d) = \sum_{\ell=1}^{L} \left\| \left(D^{(\ell-1)}\right)^{-1} W^{(\ell)} D^{(\ell)} \right\|_p^p. \tag{4.1}$$

4.3.3 Coordinate Descent: ENorm Algorithm

We formalize the ENorm algorithm using the framework of block coordinate descent. We denote[2] by $W[:, j]$ (resp. $W^{(\ell)}[i, :]$) the j^{th} column (resp. i^{th} row) of a matrix $W^{(\ell)}$. In what follows we assume that θ is admissible [3]. Note that this is equivalent to saying that all the rows and columns of the hidden weight matrices $W^{(\ell)}$ are non-zero, as well as the columns (resp. rows) or W_1 (resp. W_q). ENorm generates a sequence of rescalings $d^{(r)}$ by successively considering all the hidden layers $\ell \in [\![1, L-1]\!]$ to re-balance them as explained below.

[1] By convention, as explained in Definition 2.2.2, $D^{(0)} = I_{N_0}$ and $D^{(L)} = I_{N_L}$.

[2] This corresponds respectively to the incoming and outgoing weights of a given neuron ν.

[3] Each hidden neuron is connected to at least one input and one output neuron.

Algorithm 1: Pseudo-code of Equi-normalization

Input: Current layer weights $W^{(1)}, \dots, W^{(L)}$, number of cycles T, norm p
Output: Balanced layer weights
`// Perform T ENorm cycles`
for $t = 1 \dots T$ **do**
  `//` $D^{(0)}$ `and` $D^{(L)}$ `are never updated, see Definition 2.2.2`
  for $\ell = 1 \dots L-1$ **do**
    $L[j] \longleftarrow \left\| W^{(\ell)}[:, j] \right\|_p$ for all $j \in \mathbb{R}^{N_\ell}$
    $R[i] \longleftarrow \left\| W^{(\ell+1)}[i, :] \right\|_p$ for all $i \in \mathbb{R}^{N_\ell}$
    $D^{(\ell)} \longleftarrow \text{Diag}\sqrt{R/L}$
    $W^{(\ell)} \longleftarrow W^{(\ell)} D^{(\ell)}$
    $W^{(\ell+1)} \longleftarrow \left(D^{(\ell)}\right)^{-1} W^{(\ell+1)}$

(1) **Initialization**. Define $d^{(0)} = (1, \dots, 1)$.

(2) **Iteration**. At iteration r, set ℓ such that $\ell - 1 \equiv r \bmod L - 1$ and define

$$\begin{cases} d_{\ell'}^{(r+1)} = d_{\ell'}^{(r)} \text{ if } \ell' \neq \ell \\ d_{\ell}^{(r+1)} = \underset{t \in [0,+\infty]^{N_\ell}}{\text{argmin}} \ \varphi\left(d_1^{(r)}, \dots, d_{\ell-1}^{(r)}, t, d_{\ell+1}^{(r)}, \dots, d_{L-1}^{(r)}\right). \end{cases}$$

Denoting uv the coordinate-wise product of two vectors and u/v for the division,

$$d_\ell^{(r+1)}[i] = \sqrt{\frac{\left\| W_{\ell+1}[i, :] d_{\ell+1}^{(r)} \right\|_p}{\left\| \ W_\ell[:, i] / d_{\ell-1}^{(r)} \ \right\|_p}}. \tag{4.2}$$

Algorithm and pseudo-code. Algorithm 1 gives the pseudo-code of ENorm. By convention, one ENorm cycle balances the entire network once from $\ell = 1$ to $\ell = L-1$. See Appendix B.1 for illustrations showing the effect of ENorm on network's weights.

4.3.4 Convergence

We now state our main convergence result for Equi-normalization. The proof relies on a coordinate descent Theorem by Tseng (2001) and can be found in Appendix B.2.

Theorem 4.3.1. *Let $p > 0$ and $(d^{(r)})_{r \in \mathbb{N}}$ be the sequence of rescaling coefficients generated by ENorm from the starting point $d^{(0)}$ as described in Section 4.3.3. We assume that the considered parameterization θ is admissible. Then,*

(1) ***Convergence***. *The sequence of rescaling coefficients $d^{(r)}$ converges to d^* as $r \to +\infty$. As a consequence, the sequence of rescaled weights also converges;*

(2) ***Minimum norm***. *The rescaled weights after convergence minimize the global p-norm among all rescaling equivalent weights;*

(3) ***Uniqueness***. *The minimum d^* does not depend on the starting point $d^{(0)}$.*

4.3.5 Gradients & Biases

From now on, we denote the rescaled quantities (weights, biases, activations) with a tilde. In the presence of biases, as explained in Definition 2.2.2, we define matched rescaling equivalent biases $\widetilde{b}^{(\ell)} = b^{(\ell)} D^{(\ell)}$. Thus, we have, for all $\ell \in [\![0, L]\!]$,

$$\widetilde{y}^{(\ell)} = y^{(\ell)} D^{(\ell)} \tag{4.3}$$

We also compute the effect of applying ENorm on the gradients in Appendix B.1.1.

4.3.6 Asymmetric Scaling

Equi-normalization is easily adapted to introduce a depth-wise penalty on each layer. We consider the weighted loss $L_{p,(c_1,\dots,c_L)}(\theta) = \sum_{\ell=1}^{L} c_\ell \|W^{(\ell)}\|^p$. This amounts to modifying the rescaling coefficients as

$$d_\ell^{(r+1)}[i] \leftarrow d_\ell^{(r+1)}[i] \, (c_{\ell+1}/c_\ell)^{1/2p} \, . \tag{4.4}$$

In Section 4.6, we explore two natural ways of defining c_ℓ: $c_\ell = c^{p(L-\ell)}$ (**uniform**) and $c_\ell = 1/(N_{\ell-1}N_\ell)$ (**adaptive**). In the uniform setup, we penalize layers exponentially according to their depth: for instance, values of c larger than 1 increase the magnitude of the weights at the end of the network. In the adaptive setup, the loss is weighted by the size of the matrices.

4.4 Extension to CNNs

We now extend ENorm to CNNs, by focusing on the typical ResNet architecture. We briefly detail how we adapt ENorm to convolutional or max-pooling layers, and then how to update an elementary block with a skip-connection. We refer the reader to

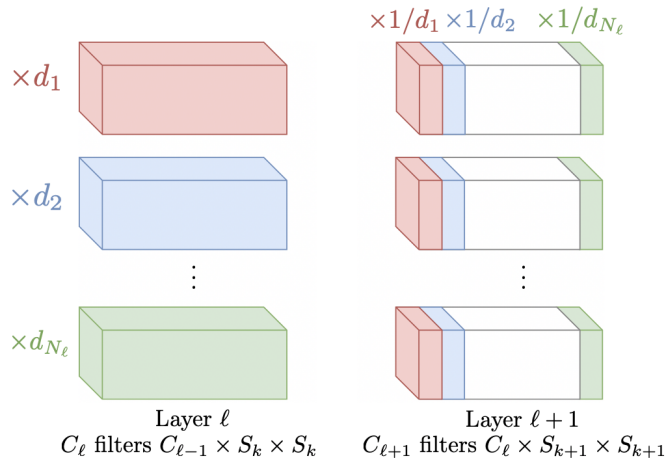


Figure 4-2: Rescaling the weights of two consecutive convolutional layers that preserves the function implemented by the CNN. Layer ℓ scales channel number i of the input activations by γ_i and layer $\ell+1$ cancels this scaling with the inverse scalar so that the activations after layer $\ell+1$ are unchanged.

Appendix B.3 for a more extensive discussion. Sanity checks of our implementation are provided in Appendix B.5.1.

4.4.1 Convolutional Layers

Figure 4-2 explains how to rescale two consecutive convolutional layers. As detailed in Appendix B.3, this is done by first properly reshaping the filters to 2D matrices, then performing the previously described rescaling on these matrices, and then reshaping the matrices back to convolutional filters. This matched rescaling does preserve the function implemented by the composition of the two layers, whether they are interleaved with a ReLU or not. It can be applied to any two consecutive convolutional layers with various stride and padding parameters. Note that when the kernel size is 1 in both layers, we recover the fully-connected case of Figure 4-1.

4.4.2 Max-Pooling

The MaxPool layer operates per channel by computing the maximum within a fixed-size kernel. We use Equation (4.3) to the convolutional case where the rescaling matrix $D^{(\ell)}$ is applied to the channel dimension of the activations $y^{(\ell-1)}$. Then,

$$\max\left(\widetilde{y}^{(\ell)}\right) = \max\left(y^{(\ell)}D^{(\ell)}\right) = \max\left(y^{(\ell)}\right)D^{(\ell)}. \tag{4.5}$$

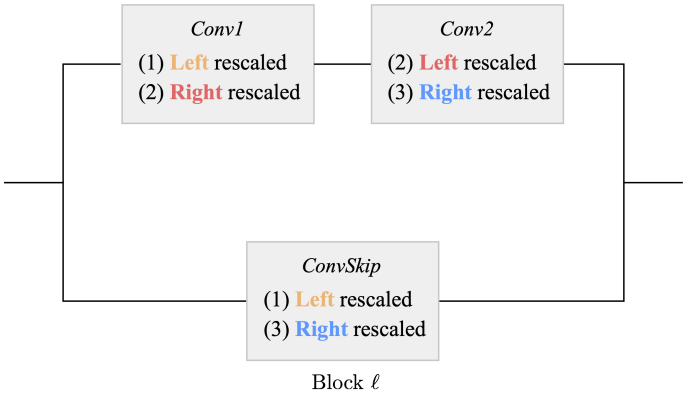


Figure 4-3: Rescaling an elementary block within a ResNet-18 consists of 3 steps. **(1)** Conv1 and ConvSkip are left-rescaled using the rescaling coefficients between blocks $k-1$ and k; **(2)** Conv1 and Conv2 are rescaled as two usual convolutional layers; **(3)** Conv2 and ConvSkip are right-rescaled using the rescaling coefficients between blocks k and $k+1$. Identical colors denote the same rescaling coefficients D. Coefficients between blocks are rescaled as detailed in Section B.3.2.

Thus, the activations before and after the MaxPool layer have the same scaling and the functional equivalence is preserved.

4.4.3 Skip Connections

We now consider an elementary block of a ResNet-18 architecture as depicted in Figure 4-3. In order to maintain functional equivalence, we only consider ResNet architectures of type C as defined in (He et al., 2015a), where all shortcuts are learned 1×1 convolutions. As detailed in Appendix B.3, rescaling two consecutive blocks requires (a) to define the structure of the rescaling process, *i.e.* where to insert the rescaling coefficients and (b) a formula for computing these rescaling coefficients.

4.5 Training with Equi-normalization & SGD

ENorm & SGD. As detailed in Algorithm 2, we balance the network periodically after updating the gradients. By design, this does not change the function implemented by the network but will yield different gradients in the next SGD iteration. Because this re-parameterization performs a jump in the parameter space, we update the momentum as described in Appendix B.1.1 and the same matrices $D^{(\ell)}$ as these used for the weights. The number of ENorm cycles after each SGD step is an hyperparameter and by default we perform one ENorm cycle after each SGD step. In

Algorithm 2: Training with Equi-normalization

Input: Initialized network
Output: Trained network
for iteration $= 1 \dots N$ **do**
  Update learning rate η
  Compute forward pass
  Compute backward pass
  Perform SGD step and update weights
  Perform one ENorm cycle using matrices $D^{(\ell)}$
  Update momentum buffers with the same $D^{(\ell)}$

Appendix B.4, we also explore a method to jointly learn the rescaling coefficients and the weights with SGD, and report corresponding results.

Computational advantage over BN and GN. Table 4.1 provides the number of elements (weights or activations) accessed when normalizing using various techniques. Assuming that the complexity (number of operations) of normalizing is proportional to the number of elements and assuming all techniques are equally parallelizable, we deduce that our method (ENorm) is theoretically 50 times faster than BN and 3 times faster than GN for a ResNet-18. In terms of memory, ENorm requires no extra-learnt parameters, but the number of parameters learnt by BN and GN is negligible (4800 for a ResNet-18 and 26,650 for a ResNet-50). We implemented ENorm using a tensor library. Taking full advantage of the theoretical reduction in compute would require to design an optimized CUDA kernel.

Model	ENorm	BN (B=256)	GN (B=16)
ResNet-18	12	636	40
ResNet-50	30	2,845	178

Table 4.1: Number of elements that are accessed during normalization (in million of activations/parameters, rounded to the closest million). For BN and GN, we choose the typical batch size B used for training.

4.6 Experiments

We analyze our approach by carrying out experiments on the MNIST and CIFAR-10 datasets and on the more challenging ImageNet dataset. ENorm will refer to Equi-normalization with $p = 2$.

4.6.1 MNIST Autoencoder

Training. Input data is normalized by subtracting the mean and dividing by standard deviation. The encoder has the structure FC(784, 1000)-ReLU-FC(1000, 500)-ReLU-FC(500, 250)-ReLU-FC(250, 30) and the decoder has the symmetric structure. We use He's initialization for the weights. We select the learning rate in $\{0.001, 0.01, 0.1\}$ and decay it linearly to zero. We use a batch size of 256 and SGD with no momentum and a weight decay of 0.001. For path-SGD, our implementation closely follows the original paper (Neyshabur et al., 2015) and we set the weight decay to zero. For GN, we cross-validate the number of groups among $\{5, 10, 20, 50\}$. For WN, we use BN as well as a greedy layer-wise initialization as in the original paper.

Results. While ENorm alone obtains competitive results compared to BN and GN, ENorm + BN outperforms all other methods, including WN + BN. Note that here ENorm refers to Enorm using the adaptive c parameter as described in Subsection 4.3.6, whereas for ENorm + BN we use the uniform setup with $c = 1$. We perform a parameter study for different values and setups of the asymmetric scaling (uniform and adaptive) in Appendix B.5.2. Without BN, the adaptive setup outperforms all other setups, which may be due to the strong bottleneck structure of the network. With BN, the dynamic is different and the results are much less sensitive to the values of c. Results without any normalization and with Path-SGD are not displayed because of their poor performance.

4.6.2 CIFAR-10 Fully Connected

Training. We first experiment with a basic fully-connected architecture that takes as input the flattened image of size 3072. Input data is normalized by subtracting mean and dividing by standard deviation independently for each channel. The first linear layer is of size 3072×500. We then consider p layers 500×500, p being an architecture parameter for the sake of the analysis. The last classification is of size 500×10. The weights are initialized with He's scheme. We train for 60 epochs using SGD with no momentum, a batch size of 256 and weight decay of 10^{-3}. Cross validation is used to pick an initial learning rate in $\{0.0005, 0.001, 0.005, 0.01, 0.05, 0.1\}$. Path-SGD, GN and WN are learned as detailed in Section 4.6.1. All results are the average test accuracies over 5 training runs.

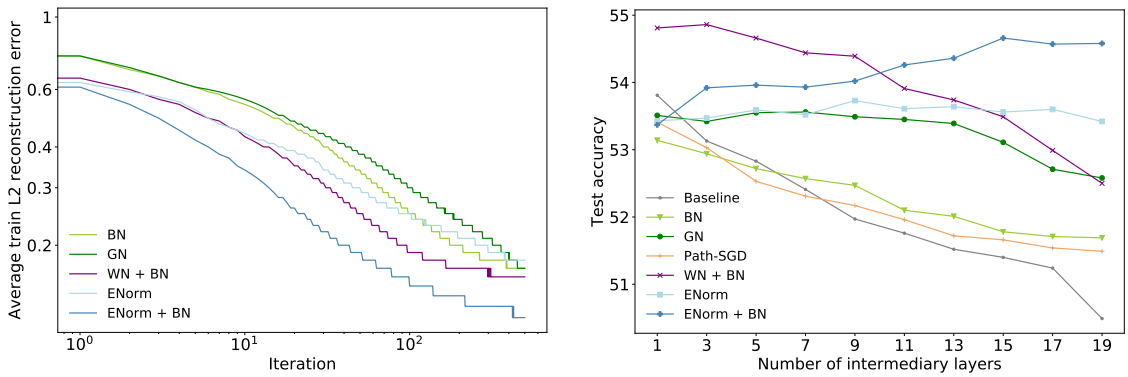


Figure 4-4: Left: MNIST auto-encoder results (lower is better). Right: CIFAR-10 fully-connected results (higher is better).

Results. ENorm alone outperforms both BN and GN for any depth of the network. ENorm + BN outperforms all other methods, including WN + BN, by a good margin for more than $p = 11$ intermediate layers. Note that here ENorm as well as ENorm + BN refers to ENorm in the uniform setup with $c = 1.2$. The results of the parameter study for different values and setups of the asymmetric scaling are similar to these of the MNIST setup, see Appendix B.5.2.

4.6.3 CIFAR-10 Fully Convolutional

Training. We use the CIFAR-NV architecture as described by Gitman and Ginsburg (2017). Images are normalized by subtracting mean and dividing by standard deviation independently for each channel. During training, we use 28×28 random crops and randomly flip the image horizontally. At test time, we use 28×28 center crops. We split the train set into one training set (40,000 images) and one validation set (10,000 images). We train for 128 epochs using SGD and an initial learning rate cross-validated on a held-out set among $\{0.01, 0.05, 0.1\}$, along with a weight decay of 0.001. The learning rate is then decayed linearly to 10^{-4}. We use a momentum of 0.9. The weights are initialized with He's scheme. In order to stabilize the training, we employ a BatchNorm layer at the end of the network after the FC layer for the Baseline and ENorm cases. For GN we cross-validate the number of groups among $\{4, 8, 16, 32, 64\}$. All results are the average test accuracies over 5 training runs.

Results. See Table 4.6.3. ENorm + BN outperforms all other methods, including WN + BN, by a good margin. Note that here ENorm refers to ENorm in the uniform

Method	Average train L_2 error
Baseline	0.542
BN	0.171
GN	0.171
WN + BN	0.162
ENorm	0.179
ENorm + BN	**0.102**

Method	Test top 1 accuracy
Baseline	88.94
BN	90.32
GN	90.36
WN + BN	90.50
ENorm	89.31
ENorm + BN	**91.35**

Table 4.2: Left: MNIST auto-encoder results (lower is better). Right: CIFAR-10 fully convolutional results (higher is better).

setup with the parameter $c = 1.2$ whereas ENorm + BN refers to the uniform setup with $c = 1$. A parameter study for different values and setups of the asymmetric scaling can be found in Appendix B.5.2.

4.6.4 ImageNet

This dataset contains 1.3M training images and 50,000 validation images split into 1000 classes. We use the ResNet-18 model with type-C learnt skip connections as described in Section 4.4.

Training. Our experimental setup closely follows that of GN (Wu and He, 2018). We train on 8 GPUs and compute the batch mean and standard deviation per GPU when evaluating BN. We use the Kaiming initialization for the weights (He et al., 2015c) and the standard data augmentation scheme of **?**. We train our models for 90 epochs using SGD with a momentum of 0.9. We adopt the linear scaling rule for the learning rate (Goyal et al., 2017) and set the initial learning rate to $0.1B/256$ where the batch size B is set to $32, 64, 128$, or 256. As smaller batches lead to more iterations per epoch, we adopt a similar rule and adopt a weight decay of $w = 10^{-4}$ for $B = 128$ and 256, $w = 10^{-4.5}$ for $B = 64$ and $w = 10^{-5}$ for $B = 32$. We decay the learning rate quadratically (Gitman and Ginsburg, 2017) to 10^{-5} and report the median error rate on the final 5 epochs. When using GN, we set the number of groups G to 32 and did not cross-validate this value as prior work (Wu and He, 2018) reports little impact when varying G from 2 to 64. In order for the training to be stable and faster, we added a BatchNorm at the end of the network after the FC layer for the Baseline and ENorm cases. The batch mean and variance for this additional BN are shared across GPUs. Note that the activation size at this stage of the network is

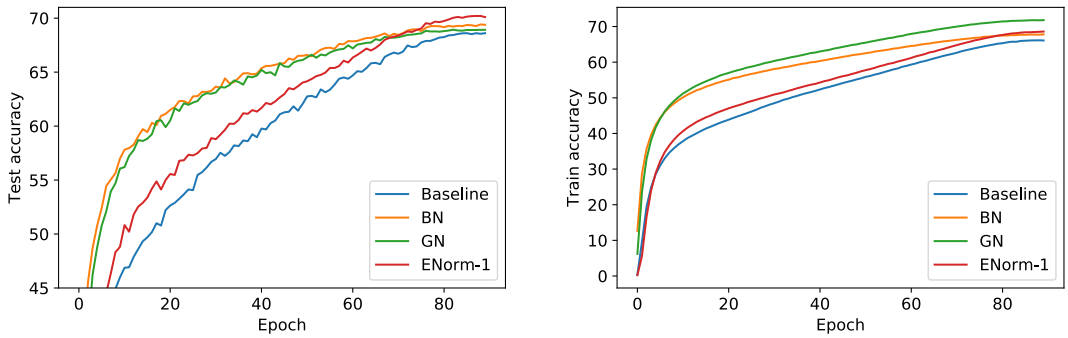


Figure 4-5: ResNet-18 results on the ImageNet dataset, batch size 64.

$B \times 1000$, which is a negligible overhead (see Table 4.1).

Results. We compare the Top 1 accuracy on a ResNet-18 when using no normalization scheme (Baseline), when using BN, GN and ENorm (our method). In both the Baseline and ENorm settings, we add a BN at the end of the network as described in Section 4.6.3. The results are reported in Table 4.3. The performance of BN decreases with small batches, which concurs with prior observations (Wu and He, 2018). Our method outperforms GN and BN for batch sizes ranging from 32 to 128. GN presents stable results across batch sizes. Note that values of c different from 1 did not yield better results. The training curves for a batch size of 64 are presented in Figure 4-5. While BN and GN are faster to converge than ENorm, our method achieves better results after convergence in this case. Note also that ENorm overfits the training set less than BN and GN, but more than the Baseline case.

Batch size	32	64	128	256
Baseline	66.20	68.60	69.20	69.58
BN	68.01	69.38	70.83	**71.37**
GN	68.94	68.90	70.69	70.64
ENorm-1 (ours)	**69.70**	**70.10**	**71.03**	71.14

Table 4.3: ResNet-18 results on the ImageNet dataset (test accuracy).

4.6.5 Limitations

We applied ENorm to a deeper (ResNet-50), but obtained unsatisfactory results. We observed that learnt skip-connections, even initialized to identity, make it harder to train without BN, even with careful layer-wise initialization or learning rate warmup. This would require further investigation.

4.7 Conclusion

We presented Equi-normalization, an iterative method that balances the energy of the weights of a network while preserving the function it implements. ENorm provably converges towards a unique equivalent network that minimizes the p-norm of its weights and it can be applied to modern CNN architectures. Using ENorm during training adds a much smaller computational overhead than BN or GN and leads to competitive performances in the FC case as well as in the convolutional case.

While optimizing an unbalanced network is hard (Neyshabur et al., 2015), the criterion we optimize to derive ENorm– minimizing the L_2 norm among the equivalence class – is likely not optimal regarding convergence or training properties. This limitation suggests that further research is required in this direction.

Chapter 5

Compressing Networks with Iterative Product Quantization

In this Chapter, we address the problem of reducing the memory footprint of convolutional network architectures. We introduce a vector quantization method that aims at preserving the quality of the reconstruction of the network outputs rather than its weights. The principle of our approach is that it minimizes the loss reconstruction error for *in-domain* inputs. Our method only requires a set of *unlabelled* data at quantization time and allows for efficient inference on CPU by using byte-aligned codebooks to store the compressed weights. We validate our approach by quantizing a high performing ResNet-50 model to a memory size of 5 MB (20× compression factor) while preserving a top-1 accuracy of 76.1% on ImageNet object classification and by compressing a Mask R-CNN with a 26× factor.

5.1 Introduction

There is a growing need for compressing the best convolutional networks (ConvNets) to support embedded devices for applications like robotics and virtual or augmented reality. Indeed, the performance of ConvNets on image classification has steadily improved since the introduction of AlexNet (Krizhevsky et al., 2012). This progress has been fueled by deeper and richer architectures such as the ResNets (He et al., 2015a) and their variants ResNeXts (Xie et al., 2017) or DenseNets (Huang et al., 2017a). these models particularly benefit from the recent progress made with weak supervision (Yalniz et al., 2019; Mahajan et al., 2018). Compression of ConvNets has been an active research topic in the recent years, leading to networks with a 71% top-1 accuracy on ImageNet object classification that fit in 1 MB (Wang et al., 2018a).

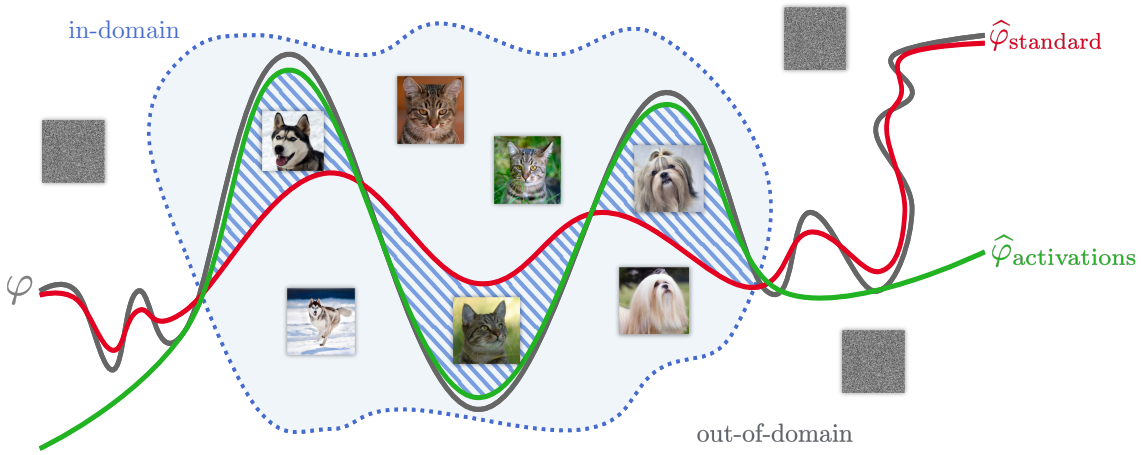


Figure 5-1: We approximate a binary classifier φ that labels images as dogs or cats by quantizing its weights. **Standard method**: quantizing φ with the standard objective function (5.1) promotes a classifier $\widehat{\varphi}_{\text{standard}}$ that tries to approximate φ over the entire input space and can thus perform badly for in-domain inputs. **Our method**: quantizing φ with our objective function (5.2) promotes a classifier $\widehat{\varphi}_{\text{activations}}$ that performs well for in-domain inputs. Images lying in the hatched area of the input space are correctly classified by $\varphi_{\text{activations}}$ but incorrectly by $\varphi_{\text{standard}}$.

In this work, we propose a compression method particularly adapted to ResNet-like architectures. Our approach takes advantage of the high correlation in the convolutions by the use of a structured quantization algorithm, Product Quantization (PQ) (Jegou et al., 2011). More precisely, we exploit the spatial redundancy of information inherent to standard convolution filters (Denton et al., 2014). Besides reducing the memory footprint, we produce compressed networks allowing efficient inference on CPU, as opposed to entropy decoders (Han et al., 2016b).

Our approach departs from traditional scalar quantizers (Han et al., 2016b) and vector quantizers (Gong et al., 2014; Carreira-Perpiñán and Idelbayev, 2017) by focusing on the accuracy of the activations rather than the weights. This is achieved by leveraging a weighted k-means technique. To our knowledge this strategy is novel in this context. The closest work we are aware of is the one by Choi et al. (2016), but the authors use a different objective (their weighted term is derived from second-order information) along with a different quantization technique (scalar quantization). Our method targets a better *in-domain* reconstruction, as depicted by Figure 5-1.

Finally, we compress the network *sequentially* to account for the dependency of our method to the activations at each layer. To prevent the accumulation of errors across layers, we guide this compression with the activations of the uncompressed network on unlabelled data: training by distillation (Hinton et al., 2015) allows for

both an efficient layer-by-layer compression procedure and a global fine-tuning of the codewords. Thus, we only need a set of unlabelled images to adjust the codewords. As opposed to recent works by Mishra and Marr (2017), Lopes et al. (2017), our distillation scheme is sequential and the underlying compression method is different. Similarly, Wu et al. (2016) use use Vector Quantization (VQ) instead PQ, do not finetune the learned codewords and do not compress the classifier's weights.

We show that applying our approach to the semi-supervised ResNet-50 of Yalniz et al. (2019) leads to a 5 MB memory footprint and a 76.1% top-1 accuracy on ImageNet object classification (hence 20× compression vs. the original model). Moreover, our approach generalizes to other tasks such as image detection. As shown in Section 5.4.3, we compress a Mask R-CNN (He et al., 2017a) with a size budget around 6 MB (26× compression factor) while maintaining a competitive performance.

5.2 Related work

Here, we review the works closest to ours and refer the reader to the Related Work section 2.3 for a more extensive discussion.

Low-precision Training. Since early works like these of Courbariaux et al. (2015), researchers have developed various approaches to train networks with low precision weights. these approaches include training with binary or ternary weights (Shayer et al., 2017; Zhu et al., 2016; Li and Liu, 2016; Rastegari et al., 2016; McDonnell, 2018), learning a combination of binary bases (Lin et al., 2017b) and quantizing the activations (Zhou et al., 2016, 2017; Mishra et al., 2017). Some of these methods assume the possibility to employ specialized hardware that speed up inference and improve power efficiency by replacing most arithmetic operations with bit-wise operations. However, the back-propagation has to be adapted to the case where the weights are discrete using accumulation or projection techniques and binary networks suffer from a significant drop in accuracy despite noticeable progress.

Quantization. Vector Quantization (VQ) and Product Quantization (PQ) have been extensively studied in the context of nearest-neighbor search (Jegou et al., 2011; Ge et al., 2014; Norouzi and Fleet, 2013). The idea is to decompose the original high-dimensional space into a cartesian product of subspaces that are quantized separately with a joint codebook. To our knowledge, Gong et al. (2014) were the first to introduce these stronger quantizers for neural network quantization, followed by

Carreira-Perpiñán and Idelbayev (2017). As we will see in the remainder of this Chapter, employing this discretization off-the-shelf does not optimize the right objective function, and leads to a catastrophic drift of performance for deep networks.

Pruning. Network pruning amounts to removing connections according to an importance criteria (typically the magnitude of the weight associated with this connection) until the desired model size/accuracy tradeoff is reached (LeCun et al., 1990). A natural extension of this work is to prune structural components of the network, for instance by enforcing channel-level (Liu et al., 2017b) or filter-level (Luo et al., 2017) sparsity. However, these methods alternate between pruning and re-training steps and thus typically require a long training time.

Dedicated Architectures. Architectures such as SqueezeNet (Iandola et al., 2016), NASNet (Zoph et al., 2017), ShuffleNet (Zhang et al., 2017b; Ma et al., 2018), MobileNets (Sandler et al., 2018b) and EfficientNets (Tan and Le, 2019) are designed to be memory efficient. As they typically rely on a combination of depth-wise and point-wise convolutional filters, sometimes along with channel shuffling, they are less prone than ResNets to structured quantization techniques such as PQ. These architectures are either designed by hand or using the framework of architecture search (Howard et al., 2019). For instance, the respective model size and test top-1 accuracy of ImageNet of a MobileNet are 13.4 MB for 71.9%, to be compared with a vanilla ResNet-50 with size 97.5 MB for a top-1 of 76.2%. Moreover, larger models such as ResNets can benefit from large-scale weakly- or semi-supervised learning to reach better performance (Mahajan et al., 2018; Yalniz et al., 2019).

Summary. Combining some of the mentioned approaches yields high compression factors as demonstrated by Han et al. (2016b) with Deep Compression (DC) or more recently by Tung and Mori (2018). Moreover and from a practical point of view, the process of compressing networks depends on the type of hardware on which the networks will run. Recent work directly quantizes to optimize energy-efficiency and latency time on a specific hardware (Wang et al., 2018a). Finally, the memory overhead of storing the full activations is negligible compared to the storage of the weights for two reasons. First, in realistic real-time inference setups, the batch size is almost always equal to one. Second, a forward pass only requires to store the activations of the *current* layer –which are often smaller than the size of the input– and not the whole activations of the network.

5.3 Our approach

In this section, we describe our strategy for network compression and we show how to extend our approach to quantize a modern ConvNet architecture. The specificity of our approach is that it aims at a small reconstruction error for the outputs of the layer rather than the layer weights themselves. We first describe how we quantize a single fully connected and convolutional layer. Then we describe how we quantize a full pre-trained network and finetune it. We call the proposed algorithm iPQ for Iterative Product Quantization.

5.3.1 Quantization of a Fully-connected Layer

We consider a fully-connected layer with weights $W \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}}}$ and, without loss of generality, we omit the bias since it does not impact reconstruction error.

Product Quantization (PQ). When applying the PQ algorithm to the columns of W, we evenly split each column into m contiguous subvectors and learn a codebook on the resulting mC_{out} subvectors. Then, a column of W is quantized by mapping each of its subvector to its nearest codeword in the codebook. For simplicity, we assume that C_{in} is a multiple of m. Hence, all the subvectors have the same dimension $d = C_{\text{in}}/m$.

More formally, the codebook $\mathcal{C} = \{c_1, \dots, c_k\}$ contains k codewords of dimension d. Any column w_j of W is mapped to its quantized version $q(w_j) = (c_{i_1}, \dots, c_{i_m})$ where i_1 denotes the index of the codeword assigned to the first subvector of w_j, and so forth. The codebook is then learned by minimizing the following objective:

$$\|W - \widehat{W}\|_2^2 = \sum_j \|w_j - q(w_j)\|_2^2, \tag{5.1}$$

where $\widehat{W}$ denotes the quantized weights. This objective can be efficiently minimized with k-means. When m is set to 1, PQ is equivalent to vector quantization (VQ) and when m is equal to C_{in}, it is the scalar k-means algorithm. The main benefit of PQ is its expressivity: each column w_j is mapped to a vector in the product $\mathcal{C} = \mathcal{C} \times \cdots \times \mathcal{C}$, thus PQ generates an implicit codebook of size k^m.

Our algorithm. PQ quantizes the weight matrix of the fully-connected layer. However, in practice, we are interested in preserving the output of the layer, not its weights. This is illustrated in the case of a non-linear classifier in Figure 5-1: preserving the weights a layer does not necessarily guarantee preserving its output. In

other words, the Frobenius approximation of the weights of a layer is not guaranteed to be the best approximation of the output over some arbitrary domain (in particular for *in-domain* inputs). We thus propose an alternative to PQ that directly minimizes the *reconstruction error* on the output activations obtained by applying the layer to in-domain inputs. More precisely, given a batch of B input activations $x \in \mathbb{R}^{B \times C_{\text{in}}}$, we are interested in learning a codebook $\mathcal{C}$ that minimizes the difference between the output activations and their reconstructions:

$$\|y - \widehat{y}\|_2^2 = \sum_j \|x(w_j - q(w_j))\|_2^2, \tag{5.2}$$

where $y = xW$ is the output and $\widehat{y} = x\widehat{W}$ its reconstruction. Our objective is a re-weighting of the objective in Equation (5.1). We can thus learn our codebook with a weighted k-means algorithm. First, we unroll x of size $B \times C_{\text{in}}$ into $\widetilde{x}$ of size $(B \times m) \times d$ *i.e.* we split each row of x into m subvectors of size d and stack these subvectors. Next, we adapt the EM algorithm as follows.

(1) **E-step (cluster assignment).** Recall that every column w_j is divided into m subvectors of dimension d. Each subvector v is assigned to the codeword

$$c_j = \underset{c \in \mathcal{C}}{\operatorname{argmin}} \|\widetilde{x}(c - v)\|_2^2. \tag{5.3}$$

This step is performed by exhaustive exploration. Our implementation relies on broadcasting to be computationally efficient.

(2) **M-step (codeword update).** Let $c \in \mathcal{C}$. We denote $(v_p)_{p \in I_c}$ the subvectors that are currently assigned to c. Then, we update $c \leftarrow c^\star$, where

$$c^\star = \underset{c \in \mathbb{R}^d}{\operatorname{argmin}} \sum_{p \in I_c} \|\widetilde{x}(c - v_p)\|_2^2. \tag{5.4}$$

This step explicitly computes the solution of the least-squares problem[1]. Our implementation performs the computation of the pseudo-inverse of $\widetilde{x}$ before alternating between the Expectation and Minimization steps as it does not depend on the learned codebook $\mathcal{C}$.

We initialize the codebook $\mathcal{C}$ by uniformly sampling k vectors among these we wish to quantize. After performing the E-step, some clusters may be empty. To resolve this

[1]Denoting $\widetilde{x}^+$ the Moore-Penrose pseudo-inverse of $\widetilde{x}$, we obtain $c^* = \frac{1}{|I_c|}\widetilde{x}^+\widetilde{x}\left(\sum_{p \in I_c} v_p\right)$

issue, we iteratively perform the following additional steps for each empty cluster of index i. (1) Find codeword c_0 corresponding to the most populated cluster ; (2) define new codewords $c_0' = c_0 + e$ and $c_i' = c_0 - e$, where $e \sim \mathcal{N}(0, \varepsilon)$ and (3) perform again the E-step. We proceed to the M-step after all the empty clusters are resolved. We set $\varepsilon = 1\text{e}{-8}$ and we observe that its generally takes less than 1 or 2 E-M iterations to resolve all the empty clusters. Note that the quality of the resulting compression is sensitive to the choice of x.

5.3.2 Convolutional Layers

Despite being presented in the case of a fully-connected layer, our approach works on any set of vectors. As a consequence, our approach can be applied to a convolutional layer if we split the associated 4D weight matrix into a set of vectors. There are many ways to split a 4D matrix in a set of vectors and we are aiming for one that maximizes the correlation between the vectors since vector quantization based methods work the best when the vectors are highly correlated.

Given a convolutional layer, we have C_{out} filters of size $K \times K \times C_{\text{in}}$, leading to an overall 4D weight matrix $W \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K \times K}$. The dimensions along the output and input coordinate have no particular reason to be correlated. On the other hand, the spatial dimensions related to the filter size are by nature very correlated: nearby patches or pixels likely share information. As depicted in Figure 5-2, we thus reshape the weight matrix in a way that lead to spatially coherent quantization. More precisely, we quantize W spatially into subvectors of size $d = K \times K$ using the following procedure. We first reshape W into a 2D matrix of size $(C_{\text{in}} \times K \times K) \times C_{\text{out}}$. Column j of the reshaped matrix W_{r} corresponds to the j^{th} filter of W and is divided into C_{in} subvectors of size $K \times K$. Similarly, we reshape the input activations x accordingly to x_{r} so that reshaping back the matrix $x_{\text{r}} W_{\text{r}}$ yields the same result as $x * W$. In other words, we adopt a dual approach to the one using bi-level Toeplitz matrices to represent the weights. Then, we apply our method exposed in Section 5.3.1 to quantize each column of W_{r} into $m = C_{\text{in}}$ subvectors of size $d = K \times K$ with k codewords, using x_{r} as input activations in (5.2). We also quantize with larger subvectors, for example subvectors of size $d = 2 \times K \times K$, see Section 5.4 for details.

In our implementation, we adapt the reshaping of W and x to various types of convolutions and refer the reader to the code for more details.

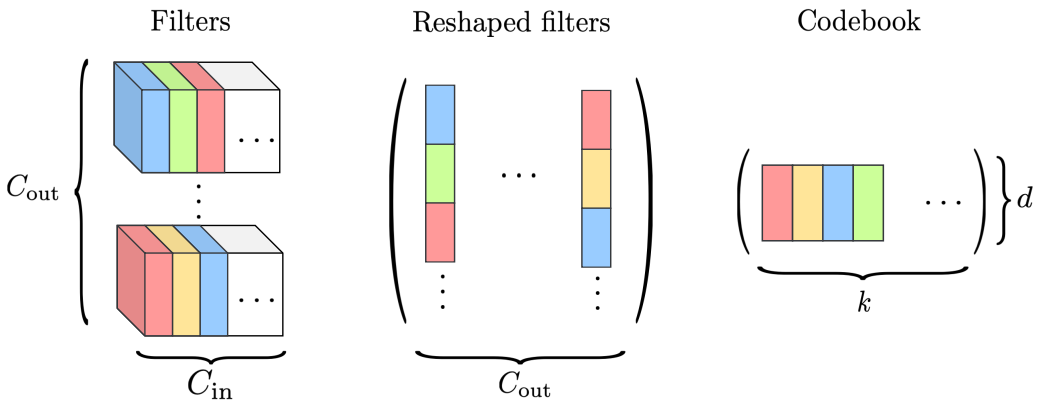


Figure 5-2: We quantize C_{out} filters of size $C_{\text{in}} \times K \times K$ using a subvector size of $d = K \times K$. In other words, we spatially quantize the convolutional filters to take advantage of the redundancy of information in the network. Similar colors denote subvectors assigned to the same codewords.

5.3.3 Network Quantization

In this section, we describe our approach for quantizing a neural network. We quantize the network sequentially starting from the lowest layer to the highest layer, and guide the compression of the student network by the non-compressed teacher network.

Learning the codebook. We recover the *current* input activations of the layer, *i.e.* the input activations obtained by forwarding a batch of images through the *quantized* lower layers, and we quantize the current layer using these activations. Experimentally, we observed a drift in both the reconstruction and classification errors when using the activations of the non-compressed network instead.

Finetuning the codebook. We finetune the codewords by distillation (Hinton et al., 2015) using the non-compressed network as the teacher network and the compressed network (up to the current layer) as the student network. Denoting y_{t} (resp. y_{s}) the output probabilities of the teacher (resp. student) network, the loss we optimize is the Kullback-Leibler divergence $\mathcal{L} = \text{KL}(y_{\text{s}}, y_{\text{t}})$. Finetuning on codewords is done by averaging the gradients of each subvector assigned to a given codeword. More formally, after the quantization step, we fix the assignments once for all. Then, denoting $(b_p)_{p \in I_c}$ the subvectors that are assigned to codeword c, we perform the SGD update with a learning rate η

$$c \leftarrow c - \eta \frac{1}{|I_c|} \sum_{p \in I_c} \frac{\partial \mathcal{L}}{\partial b_p}. \tag{5.5}$$

Experimentally, we find the approach to perform better than finetuning on the target of the images as demonstrated in Table 5.3. Moreover, this approach does not require any labelled data as it relies on distillation.

5.3.4 Global Finetuning

In a final step, we globally finetune the codebooks of all the layers to reduce any residual drifts and we update the running statistics of the BatchNorm layers: We empirically find it beneficial to finetune *all* the centroids after the whole network is quantized. The finetuning procedure is exactly the same as described in Section 5.3.3, except that we additionally switch the BatchNorms to the training mode, meaning that the learnt coefficients are still fixed but that the batch statistics (running mean and variance) are still being updated with the standard moving average procedure.

We perform the global finetuning using the standard ImageNet training set for 9 epochs with an initial learning rate of 0.01, a weight decay of 10^{-4} and a momentum of 0.9. The learning rate is decayed by a factor 10 every 30 epochs. As demonstrated in the ablation study in Table 5.3, finetuning on the true labels performs worse than finetuning by distillation. A possible explanation is that the supervision signal coming from the teacher network is richer than the one-hot vector used traditionally.

5.4 Experiments

5.4.1 Experimental Setup

We quantize vanilla ResNet-18 and ResNet-50 architectures pretrained on the ImageNet dataset (Deng et al., 2009). Unless explicit mention of the contrary, the pretrained models are taken from the PyTorch model zoo[2]. We run our method on a 16 GB Volta V100 GPU. Quantizing a ResNet-50 with our method (including all finetuning steps) takes about one day on 1 GPU. We detail our experimental setup below. Our code and the compressed models are open-sourced.

Compression Regimes. We explore a *large block sizes* (resp. *small block sizes*) compression regime by setting the subvector size of 3×3 convolutions to $d=9$ (resp. $d=18$) and the subvector size of pointwise convolutions to $d=4$ (resp. $d=8$). For ResNet-18, the block size of pointwise convolutions is always equal to 4. The number

[2] `https://pytorch.org/docs/stable/torchvision/models`.

of codewords or centroids is set to $k \in \{256, 512, 1024, 2048\}$ for each compression regime. We clamp the number of centroids to $\min(k, C_{\text{out}} \times m/4)$ for stability. For instance, the first layer of the first stage of the ResNet-50 has size $64\times$ $64\times$ 1 $\times 1$, thus we always use $k = 128$ centroids with a block size $d = 8$. For a given number of centroids k, small blocks lead to a lower compression than large blocks.

Sampling the Input Activations. Before quantizing each layer, we randomly sample a batch of 1024 training images to obtain the input activations of the current layer and reshape it as described in Section 5.3.2. Before each iteration (E+M step) of our method, we randomly sample $10,000$ rows from these reshaped activations.

Hyperparameters. We quantize each layer while performing 100 steps of our method (sufficient for convergence in practice). We finetune the centroids of each layer on the standard ImageNet training set during 2,500 iterations with a batch size of 128 (resp 64) for the ResNet-18 (resp. ResNet-50) with a learning rate of 0.01, a weight decay of 10^{-4} and a momentum of 0.9. For accuracy and memory reasons, the classifier is always quantized with a block size $d = 4$ and $k = 2048$ (resp. $k = 1024$) centroids for the ResNet-18 (resp., ResNet-50). Moreover, the first convolutional layer of size 7×7 is not quantized, as it represents less than 0.1% (resp., 0.05%) of the weights of a ResNet-18 (resp. ResNet-50).

Metrics. We focus on the tradeoff between accuracy and memory. The accuracy is the top-1 error on the standard validation set of ImageNet. The memory footprint is calculated as the indexing cost plus the overhead of storing the centroids in float16. As an example, quantizing a layer of size $128 \times 128 \times 3 \times 3$ with $k = 256$ centroids (1 byte per subvector) and a block size of $d = 9$ leads to an indexing cost of 16 kB for $m = 16,384$ blocks plus the cost of storing the centroids of 4.5 kB.

5.4.2 Image Classification Results

We report below the results of our method applied to various ResNet models. First, we compare our method with the state of the art on the standard ResNet-18 and ResNet-50 architecture. Next, we show the potential of our approach on a competitive ResNet-50. Finally, an ablation study validates the pertinence of our method.

Vanilla ResNet-18 and ResNet-50. We evaluate our method on the ImageNet benchmark for ResNet-18 and ResNet-50 architectures and compare our results to

Model (original top-1)	Compression	Size ratio	Model size	Top-1 (%)
ResNet-18 (69.76%)	Small blocks	29x	1.54 MB	**65.81** ±0.04
	Large blocks	43x	1.03 MB	**61.10** ±0.03
ResNet-50 (76.15%)	Small blocks	19x	5.09 MB	**73.79** ±0.05
	Large blocks	31x	3.19 MB	**68.21** ±0.04

Table 5.1: Results for vanilla ResNet-18 and ResNet-50 for $k = 256$ centroids.

the following methods: Trained Ternary Quantization (TTQ) (Zhu et al., 2016), LR-Net (Shayer et al., 2017), ABC-Net (Lin et al., 2017b), Binary Weight Network (XNOR-Net or BWN) (Rastegari et al., 2016), Deep Compression (DC) (Han et al., 2016b) and Hardware-Aware Automated Quantization (HAQ) (Wang et al., 2018a). We report the accuracies and compression factors in the original papers and/or in the two surveys (Guo, 2018; Cheng et al., 2017) for a given architecture when the result is available. We do not compare our method to DoReFa-Net (Zhou et al., 2016) and WRPN (Mishra et al., 2017) as these approaches also use low-precision activations and hence get lower accuracies, e.g., 51.2% top-1 accuracy for a XNOR-Net with ResNet-18. The results are presented in Figure 5.4.2. For better readability, some results for our method are also displayed in Table 5.1. We report the average accuracy and standard deviation over 3 runs. Our method significantly outperforms state of the art papers for various operating points. For instance, for a ResNet-18, our method with large blocks and $k = 512$ centroids reaches a larger accuracy than ABC-Net ($M = 2$) with a compression ratio that is 2x larger. Similarly, on the ResNet-50, our compressed model with $k = 256$ centroids in the large blocks setup yields a comparable accuracy to DC (2 bits) with a compression ratio that is 2x larger.

The work by Tung and Mori (2018) is likely the only one that remains competitive with ours with a 6.8 MB network after compression, with a technique that prunes the network and therefore implicitly changes the architecture. The authors report the delta accuracy for which we have no direct comparable top-1 accuracy, but their method is arguably complementary to ours.

Semi-supervised ResNet-50. Recent works (Mahajan et al., 2018; Yalniz et al., 2019) have demonstrated the possibility to leverage a large collection of unlabelled images to improve the accuracy. In particular, Yalniz et al. (2019) use the publicly available YFCC-100M dataset (Thomee et al., 2015) to train a ResNet-50 that reaches 79.3% top-1 accuracy on the standard validation set of ImageNet. In the following, we use this particular model and refer to it as semi-supervised ResNet-50. In the low

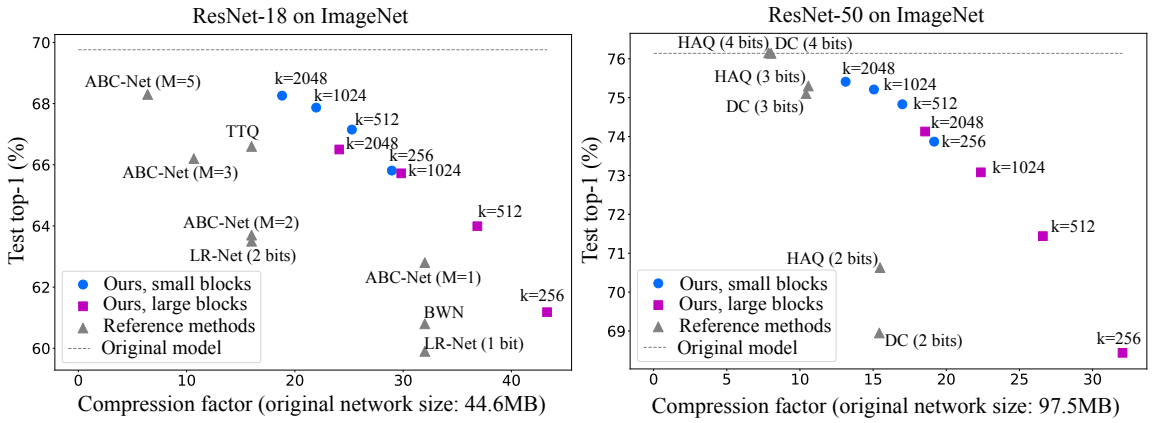


Figure 5-3: Compression results for ResNet-18 and ResNet-50 architectures. We explore two compression regimes as defined in Section 5.4.1: small block sizes (block sizes of $d = 4$ and 9) and large block sizes (block sizes $d = 8$ and 18). The results of our method for $k = 256$ centroids are of practical interest as they correspond to a byte-compatible compression scheme.

Size budget	Best previous published method	Ours
∼1 MB	**70.90%** (HAQ , MobileNet v2)	64.01% (vanilla ResNet-18)
∼5 MB	71.74% (HAQ, MobileNet v1)	**76.12%** (semi-sup. ResNet-50)
∼10 MB	75.30% (HAQ, ResNet-50)	**77.85%** (semi-sup. ResNet-50)

Table 5.2: Best test top-1 on ImageNet for a given size (no architecture constraint).

compression regime (block sizes of 4 and 9), with $k = 256$ centroids (practical for implementation), our compressed semi-supervised ResNet-50 reaches **76.12% top-1 accuracy**. In other words, the model compressed to 5.20 MB has the performance of a vanilla, non-compressed ResNet50 (vs. 97.5 MB for the non-compressed ResNet-50).

Comparison for a Given Size Budget. To ensure a fair comparison, we compare our method for a given model size budget against the reference methods in Table 5.2. It should be noted that our method can further benefit from advances in semi-supervised learning to boosts the performance of the non-compressed and hence of the compressed network.

Ablation Study. We perform an ablation study on the vanilla ResNet-18 to study the respective effects of quantizing using the activations and finetuning by distillation (here, finetuning refers both to the per-layer finetuning and to the global finetuning after the quantization described in Section 5.3). We refer to our method as Act + Distill. First, we still finetune by distillation but change the quantization: instead of

Compression	k	No act + Distill	Act + Labels	**Act + Distill (ours)**
Small blocks	256	64.76	65.55	**65.81**
	512	66.31	66.82	**67.15**
	1024	67.28	67.53	**67.87**
	2048	67.88	67.99	**68.26**
Large blocks	256	60.46	61.01	**61.18**
	512	63.21	63.67	**63.99**
	1024	64.74	65.48	**65.72**
	2048	65.94	66.21	**66.50**

Table 5.3: Ablation study on ResNet-18 (test top-1 accuracy on ImageNet).

quantizing using our method (see Equation (5.2)), we quantizing using the standard PQ algorithm and do not take the activations into account, see Equation (5.1). We refer to this method as No act + Distill. Second, we quantize using our method but perform a standard finetuning using the image labels (Act + Labels). The results are displayed in Table 5.3. Our approach consistently yields significantly better results. As a side note, quantizing all the layers of a ResNet-18 with the standard PQ algorithm and without any finetuning leads to top-1 accuracies below 25% for all operating points, which illustrates the drift in accuracy occurring when compressing deep networks with standard methods (as opposed to our method).

5.4.3 Image Detection Results

To demonstrate the generality of our method, we compress the Mask R-CNN architecture used for image detection in many real-life applications (He et al., 2017a). We compress the backbone (ResNet-50 FPN) in the *small blocks* compression regime and refer the reader to the open-sourced compressed model for the block sizes used in the various heads of the network. Results are displayed in Table 5.4.3. We argue that this provides an interesting point of comparison for future work aiming at compressing such detection architectures.

Model	Size	Box AP	Mask AP
Non-compressed	170 MB	37.9	34.6
Compressed	6.65 MB	33.9	30.8

Table 5.4: Compression results for Mask R-CNN (backbone ResNet-50 FPN) for $k = 256$ centroids (compression factor 26×).

5.5 Conclusion

We presented a quantization method based on Product Quantization that gives state of the art results on ResNet architectures and that generalizes to other architectures such as Mask R-CNN. Our compression scheme does not require labeled data and the resulting models are byte-aligned, allowing for efficient inference on CPU.

A current limitation of our method is that it takes advantage of the natural spatial redundancy in the traditional 3×3 filters and performs worse on 1×1 filters (pointwise convolutions), where the redundancy, if any, is less observable. This makes iPQ less suited for mobile-efficient architectures such as MobileNets or EfficientNets (Sandler et al., 2018a; Tan and Le, 2019). We developed a pre-conditioning technique called Quant-Noise that is explained in the next Chapter 6.

Chapter 6

Pre-conditioning Network Compression with Quant-Noise

As in Chapter 5, we tackle the problem of producing compact models, maximizing their accuracy for a given model size. To that end, we develop a pre-conditioning technique called Quant-Noise that injects a carefully chosen quantization noise when training the uncompressed network before compressing it with iPQ. As a result we establish new state-of-the-art compromises between accuracy and model size both in natural language processing and image classification. For example, applying our method to state-of-the-art Transformer and ConvNet architectures, we can achieve 82.5% accuracy on MNLI by compressing RoBERTa to 14 MB and 80.0% top-1 accuracy on ImageNet by compressing an EfficientNet-B3 to 3.3 MB. We also show the potential of Quant-Noise for scalar `int8` and `int4` quantization over the standard Quantization Aware Training (Jacob et al., 2018) procedure. Finally, we combine product and `int8` scalar quantization to take advantage of (i) the speedup provided by `int8` and the high compression ratio of iPQ.

6.1 Introduction

Many of the best performing neural network architectures in real-world applications have a large number of parameters. For example, the current standard machine translation architecture, Transformer (Vaswani et al., 2017), has layers that contain millions of parameters. Even models that are designed to jointly optimize the performance and the parameter efficiency, such as EfficientNets (Tan and Le, 2019), still require dozens to hundreds of megabytes, which limits their applications to domains like robotics or virtual assistants.

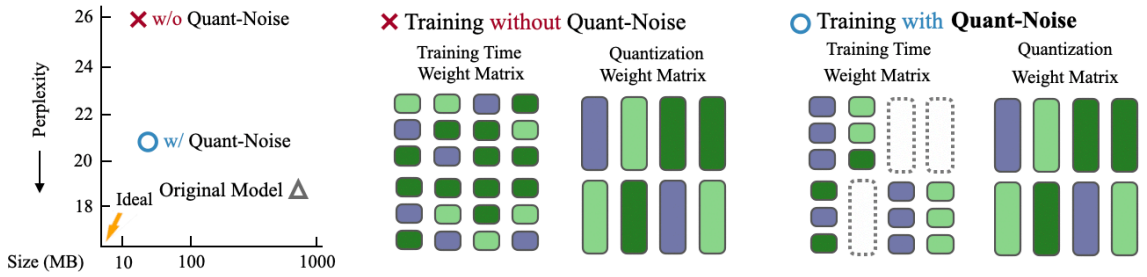


Figure 6-1: **Quant-Noise** trains models to be resilient to quantization by mimicking the effect of the quantization method during training time. This allows for extreme compression rates without much loss in accuracy on a variety of tasks and benchmarks.

Model compression schemes reduce the memory footprint of overparametrized models. Pruning (LeCun et al., 1990) and distillation (Hinton et al., 2015) remove parameters by reducing the number of network weights. In contrast, quantization focuses on reducing the bits per weight. This makes quantization particularly interesting when compressing models that have already been carefully optimized in terms of network architecture. Whereas deleting weights or whole hidden units will inevitably lead to a drop in performance, we demonstrate that quantizing the weights can be performed with little to no loss in accuracy.

Popular postprocessing quantization methods, like scalar quantization, replace the floating-point weights of a trained network by a lower-precision representation, like fixed-width integers (Vanhoucke et al., 2011). These approaches achieve a good compression rate with the additional benefit of accelerating inference on supporting hardware. However, the errors made by these approximations accumulate in the computations operated during the forward pass, inducing a significant drop in performance as explained in Chapter 5.

A solution to address this drifting effect is to directly quantize the network during training. This raises two challenges. First, the discretization operators have a null gradient — the derivative with respect to the input is zero almost everywhere. This requires special workarounds to train a network with these operators. The second challenge that often comes with these workarounds is the discrepancy that appears between the train and test functions implemented by the network. Quantization Aware Training (QAT) (Jacob et al., 2018) resolves these issues by quantizing all the weights during the forward and using a straight through estimator (STE) (Bengio et al., 2013) to compute the gradient. This works when the error introduced by STE is small, like with `int8` quantization, but does not suffice in compression regimes where the approximation made by the compression is more severe.

In this work, we show that quantizing only a subset of weights instead of the entire network during training is more stable for high compression schemes. Indeed, by quantizing only a random fraction of the network at each forward, most the weights are updated with unbiased gradients. Interestingly, we show that our method can employ a simpler quantization scheme during the training. This is particularly useful for quantizers with trainable parameters, such as Product Quantizer (PQ), for which our quantization proxy is not parametrized. Our approach simply applies a quantization noise, called Quant-Noise, to a random subset of the weights, see Figure 6-1. We observe that this makes a network resilient to various types of discretization methods: it significantly improves the accuracy associated with (a) low precision representation of weights like `int8`; and (b) state-of-the-art iPQ. Further, we demonstrate that Quant-Noise can be applied to existing trained networks as a post-processing step, to improve the performance network after quantization. In summary,

- We introduce the Quant-Noise technique to learn networks that are more resilient to a variety of quantization methods such as `int4`, `int8`, and iPQ;
- Adding Quant-Noise to iPQ leads to state-of-the-art trade-offs between accuracy and model size. For instance, for natural language processing (NLP), we reach 82.5% accuracy on MNLI by compressing RoBERTa to 14 MB. Similarly for computer vision, we report 80.0% top-1 accuracy on ImageNet by compressing an EfficientNet-B3 to 3.3 MB;
- By combining iPQ and `int8` to quantize weights and activations for networks trained with Quant-Noise, we obtain extreme compression with fixed-precision computation and achieve 79.8% top-1 accuracy on ImageNet and 21.1 perplexity on WikiText-103.

6.2 Related Work

Here, we review the works closest to ours and refer the reader to the Related Work section 2.3 for a more extensive discussion.

Model compression. Many compression methods focus on efficient parameterization, via weight pruning (LeCun et al., 1990; Li et al., 2016; Huang et al., 2018a; Mittal et al., 2018), weight sharing (Dehghani et al., 2019; Turc et al., 2019; Lan et al., 2019) or with dedicated architectures (Tan and Le, 2019; Zhang et al., 2017b; Howard et al.,

2019). Weight pruning is implemented during training (Louizos et al., 2017b) or as a fine-tuning post-processing step (Han et al., 2015, 2016b). Many pruning methods are unstructured, i.e., remove individual weights (LeCun et al., 1990; Molchanov et al., 2017). On the other hand, structured pruning methods follow the structure of the weights to reduce both the memory footprint and the inference time of a model (Li et al., 2016; Luo et al., 2017; Fan et al., 2019). We refer the reader to Liu et al. (2018) for a review of different pruning strategies. Other authors have worked on lightweight architectures, by modifying existing models (Zhang et al., 2018a; Wu et al., 2019b; Sukhbaatar et al., 2019a) or developing new networks, such as MobileNet (Howard et al., 2019), ShuffleNet (Zhang et al., 2017b), and EfficientNet (Tan and Le, 2019). Finally, knowledge distillation (Hinton et al., 2015) has been applied to sentence representation (Turc et al., 2019; Sanh et al., 2019; Zhao et al., 2019; Jiao et al., 2019) and to reduce the size of a BERT model (Devlin et al., 2018).

Quantization. There are extensive studies of scalar quantization to train networks with low-precision weights and activations (Courbariaux et al., 2015; Courbariaux and Bengio, 2016; Rastegari et al., 2016; McDonnell, 2018). These methods benefit from specialized hardware to also improve the runtime during inference (Vanhoucke et al., 2011). Other quantization methods such as Vector Quantization (VQ) and PQ (Jegou et al., 2011) quantize blocks of weights simultaneously to achieve higher compression rate (Gong et al., 2014; Joulin et al., 2016; Carreira-Perpiñán and Idelbayev, 2017). Closer to our work, several works have focused at simultaneously training and quantizing a network (Jacob et al., 2018; Krishnamoorthi, 2018; Gupta et al., 2015; Dong et al., 2019). Gupta et al. (2015) assigns weights to a quantized bin stochastically which is specific to scalar quantization, but permits training with fixed point arithmetic. Finally, our method can be interpreted as a form of Bayesian compression (Louizos et al., 2017b), using the Bayesian interpretation of Dropout (Gal and Ghahramani, 2016). However, we select our noise to match the weight transformation of a quantization method without restricting it to a scale mixture prior.

6.3 Quantizing Neural Networks

In this section, we present the principles of quantization, several standard quantization methods, and describe how to combine scalar and product quantization. For clarity, we focus on the case of a fixed real matrix $W \in \mathbb{R}^{n \times p}$. We suppose that this

matrix is split into $m \times q$ blocks b_{kl} as follows:

$$W = \begin{pmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mq} \end{pmatrix}, \tag{6.1}$$

where the nature of these blocks is determined by the quantization method. A codebook is a set of K vectors, i.e., $\mathcal{C} = \{c[1], \ldots, c[K]\}$. Quantization methods compress the matrix W by assigning to each block b_{kl} an index that points to a codeword c in a codebook $\mathcal{C}$, and storing the codebook $\mathcal{C}$ and the resulting indices (as the entries I_{kl} of an index matrix I) instead of the real weights. During the inference, they reconstruct an approximation $\widehat{W}$ of the original matrix W such that $\widehat{b}_{kl} = c[I_{kl}]$.

We distinguish scalar quantization, such as `int8`, where each block b_{kl} consists of a single weight, from vector quantization, where several weights are quantized jointly.

6.3.1 Fixed-point Scalar Quantization

Fixed-point scalar quantization methods replace floating-point number representations by low-precision fixed-point representations. They simultaneously reduce a model's memory footprint and accelerate inference by using fixed-point arithmetic on supporting hardware.

Fixed-point scalar quantization operates on blocks that represent a single weight, i.e., $b_{kl} = W_{kl}$. Floating-point weights are replaced by N bit fixed-point numbers (Gupta et al., 2015), with the extreme case of binarization where $N = 1$ (Courbariaux et al., 2015). More precisely, the weights are rounded to one of 2^N possible codewords. These codewords correspond to bins evenly spaced by a scale factor s and shifted by a bias z. Each weight W_{kl} is mapped to its nearest codeword c by successively quantizing with $z \mapsto \text{round}(W_{kl}/s + z)$ and dequantizing with:

$$c = (\text{round}(W_{kl}/s + z) - z) \times s, \tag{6.2}$$

where we compute the scale and bias as:

$$s = \frac{\max W - \min W}{2^N - 1} \quad \text{and} \quad z = \text{round}(\min W/s).$$

We focus on this uniform rounding scheme instead of other non-uniform schemes (Choi et al., 2018; Li et al., 2019), because it enables fixed-point arithmetic with implemen-

tations in PyTorch and Tensorflow (see Appendix). The compression rate is $\times 32/N$. The activations are also rounded to N-bit fixed-point numbers. With `int8` for instance, this leads to $\times 2$ to $\times 4$ faster inference on dedicated hardware. In this work, we consider both `int4` and `int8` quantization.

6.3.2 Product Quantization

Several quantization methods work on groups of weights, such as vectors, to benefit from the correlation induced by the structure of the network. In this work, we focus on Product Quantization for its good performance at extreme compression ratio as explained in Chapter 5.

Traditional PQ. In vector quantization methods, the blocks are predefined groups of weights instead of single weights. The codewords are groups of values, and the index matrix I maps groups of weights from the matrix W to these codewords. In this section, we present the Product Quantization framework as it generalizes both scalar and vector quantization. We consider the case where we apply PQ to the *columns* of W and thus assume that $q = p$.

Traditional vector quantization techniques split the matrix W into its p columns and learn a codebook on the resulting p vectors. Instead, Product Quantization splits each column into m subvectors and learns the same codebook for each of the resulting $m \times p$ subvectors. Each quantized vector is subsequently obtained by assigning its subvectors to the nearest codeword in the codebook. Learning the codebook is traditionally done using k-means with a fixed number K of centroids, typically $K = 256$ to store the index matrix I using `int8`. Thus, the objective function is written as:

$$\|W - \widehat{W}\|_2^2 = \sum_{k,l} \|b_{kl} - c[I_{kl}]\|_2^2. \tag{6.3}$$

PQ shares representations between subvectors, yielding higher compression rates than `int8` or `int4` (respectively $\times 4$ and $\times 8$ with respect to the non-compressed model).

iPQ. When quantizing a full network rather than a single matrix, extreme compression with PQ induces a quantization drift as reconstruction error accumulates as explained in Chapter 5. Indeed, subsequent layers take as input the output of preceding layers, which are modified by the quantization of the preceding layers. This creates a drift in the network activations, resulting in large losses of performance.

A solution proposed in Chapter 5, called **iterative PQ** (iPQ), is to quantize layers sequentially from the lowest to the highest, and finetune the upper layers as the lower layers are quantized, under the supervision of the uncompressed (teacher) model. Codewords of each layer are finetuned by averaging the gradients of their assigned elements with gradient steps:

$$c \leftarrow c - \eta \frac{1}{|J_c|} \sum_{(k,l) \in J_c} \frac{\partial \mathcal{L}}{\partial b_{kl}}, \tag{6.4}$$

where $J_c = \{(k, l) \mid c[I_{kl}] = c\}$, $\mathcal{L}$ is the loss function and $\eta > 0$ is a learning rate. This adapts the upper layers to the drift appearing in their inputs, reducing the impact of the quantization approximation on the overall performance.

6.3.3 Combining Fixed-Point with Product Quantization

Fixed-point quantization and Product Quantization are often regarded as competing choices, but can be advantageously combined. Indeed, PQ/iPQ compresses the network by replacing vectors of weights by their assigned centroids, but these centroids are in floating-point precision. Fixed-point quantization compresses both activations and weights to fixed-point representations. Combining both approaches means that the vectors of weights are mapped to centroids that are compressed to fixed-point representations, along with the activations. This benefits from the extreme compression ratio of iPQ and the finite-precision arithmetics of `intN` quantization.

More precisely, for a given matrix, we store the `int8` representation of the K centroids of dimension d along with the $\log_2 K$ representations of the centroid assignments of the $m \times p$ subvectors. The `int8` representation of the centroids is obtained with Eq. (6.2). The overall storage of the matrix and activations during a forward pass with batch size 1 (recalling that the input dimension is n) writes

$$M = 8 \times Kd + \log_2 K \times mp + 8 \times n \text{ bits}. \tag{6.5}$$

In particular, when $K = 256$, the centroid assignments are also stored in `int8`, which means that every value required for a forward pass is stored in an `int8` format. We divide by 4 the `float32` overhead of storing the centroids, although the storage requirement associated with the centroids is small compared to the cost of indexing the subvectors for standard networks. In contrast to iPQ alone where we only quantize the weights, we also quantize the activations using `int8`.

6.4 Method

Deep networks are not exposed to the noise caused by the quantization drift during training, leading to suboptimal performance. A solution to make the network robust to quantization is to introduce it during training. Quantization Aware Training (QAT) (Jacob et al., 2018) exposes the network during training by quantizing weights during the forward pass. This transformation is not differentiable and gradients are approximated with a straight through estimator (STE) (Bengio et al., 2013; Courbariaux and Bengio, 2016). STE introduces a bias in the gradients that depends on level of quantization of the weights, and thus, the compression ratio. In this section, we propose a simple modification to control this induced bias with a stochastic amelioration of QAT, called Quant-Noise. The idea is to quantize a randomly selected fraction of the weights instead of the full network as in QAT, leaving some unbiased gradients flow through unquantized weights. Our general formulation can simulate the effect of both quantization and of pruning during training.

6.4.1 Training Networks with Quantization Noise

We consider the case of a real matrix W as in Section 6.3. During the training of a network, our proposed Quant-Noise method works as follows: first, we compute blocks b_{kl} related to a target quantization method. Then, during each forward pass, we randomly select a subset of these blocks and apply some distortion to them, and we compute gradients for all the weights, using STE for the distorted weights.

More formally, given a set of tuples of indices $J \subset \{(k, l)\}$ for $1 \leq k \leq m$, $1 \leq l \leq q$ and a *distortion* or *noise* function φ acting on a block, we define an operator $\psi(\cdot \mid J)$ such that, for each block b_{kl}, we apply the following transformation:

$$\psi(b_{kl} \mid J) = \begin{cases} \varphi(b_{kl}) & \text{if } (k, l) \in J, \\ b_{kl} & \text{otherwise.} \end{cases} \tag{6.6}$$

The noise function φ simulates the change in the weights produced by the target quantization method (see Section 6.4.2 for details). We replace the matrix W by the noisy matrix W_{noise} during the forward pass to compute a noisy output y_{noise} as

$$W_{\text{noise}} = (\psi(b_{kl} \mid J))_{kl} \qquad \text{and} \qquad y_{\text{noise}} = xW_{\text{noise}} \tag{6.7}$$

where x is an input vector. During the backward pass, we apply STE, which amounts

to replacing the distorted weights W_{noise} by their non-distorted counterparts. Note that our approach is equivalent to QAT when J contains all the tuples of indices. However, an advantage of Quant-Noise over QAT is that unbiased gradients continue to flow via blocks unaffected by the noise. As these blocks are randomly selected for each forward, we guarantee that each weight regularly sees gradients that are not affected by the nature of the function φ. As a side effect, our quantization noise regularizes the network in a similar way as DropConnect (Wan et al., 2013) or LayerDrop (Fan et al., 2019).

Composing quantization noises. As noise operators are compositionally commutative, we can make a network robust to a combination of quantization methods by composing their noise operators:

$$\psi(b_{kl} \mid J) = \psi_1 \circ \psi_2(b_{kl} \mid J). \tag{6.8}$$

This property is particularly useful to combine quantization with pruning operators during training, as well as combining scalar quantization with product quantization.

6.4.2 Adding Noise to Specific Quantization Methods

In this section, we propose several implementations of the noise function φ for the quantization methods described in Section 6.3.

Fixed-point scalar quantization. In `intN` quantization, the blocks are atomic and weights are rounded to their nearest neighbor in the codebook. The function φ replaces weight W_{kl} with the output of the rounding function defined in Eq. (6.2)

$$\varphi_{\texttt{intN}}(w) = (\text{round}(w/s + z) - z) \times s, \tag{6.9}$$

where s and z are updated during training. In particular, the application of Quant-Noise to `int8` scalar quantization is a stochastic amelioration of QAT.

Product quantization. As opposed to `intN`, codebooks in PQ require a clustering step based on weight values. During training, we learn codewords online and use the resulting centroids to implement the quantization noise. More precisely, the noise

Quantization Scheme	Language Modeling 16-layer Transformer Wikitext-103			Image Classification EfficientNet-B3 ImageNet-1k		
	Size	Compression	PPL	Size	Compression	Top-1
Uncompressed model	942	× 1	18.3	46.7	× 1	81.5
`int4` quantization	118	× 8	39.4	5.8	× 8	45.3
- with QAT	118	× 8	34.1	5.8	× 8	59.4
- with Quant-Noise	118	× 8	21.8	5.8	× 8	67.8
`int8` quantization	236	× 4	19.6	11.7	× 4	80.7
- with QAT	236	× 4	21.0	11.7	× 4	80.8
- with Quant-Noise	236	× 4	18.7	11.7	× 4	80.9
iPQ	38	× 25	25.2	3.3	× 14	79.0
- with QAT	38	× 25	41.2	3.3	× 14	55.7
- with Quant-Noise	38	× 25	20.7	3.3	× 14	80.0
iPQ & `int8` + QNoise	38	× 25	21.1	3.1	× 15	79.8

Table 6.1: **Comparison of different quantization schemes with and without Quant-Noise** on language modeling and image classification. For language modeling, we train a Transformer on the Wikitext-103 benchmark and report perplexity (PPL) on test. For image classification, we train a EfficientNet-B3 on the ImageNet-1k benchmark and report top-1 accuracy on validation and use our re-implementation of EfficientNet-B3. The original implementation of Tan and Le (2019) achieves an uncompressed Top-1 accuracy of 81.9%. For both settings, we report model size in megabyte (MB) and the compression ratio compared to the original model.

function φ_{PQ} assigns a selected block b to its nearest codeword in $\mathcal{C}$:

$$\varphi_{\text{PQ}}(v) = \text{argmin}_{c \in \mathcal{C}} \|b - c\|_2^2. \tag{6.10}$$

Updating the codebooks online works well. However, empirically, running k-means once per epoch is faster and does not noticeably modify the resulting accuracy. Note that computing the exact noise function for PQ is computationally demanding. We propose a simpler and faster alternative approximation φ_{proxy} to the operational transformation of PQ and iPQ. The noise function simply zeroes out the subvectors of the selected blocks, i.e., $\varphi_{\text{proxy}}(v) = 0$. As a side note, we considered other alternatives, for instance one where the subvectors are mapped to the mean subvector. In practice, we found that these approximations lead to similar performance, see Section C.2. This proxy noise function is a form of Structured Dropout and encourages correlations between the subvectors. This correlation is beneficial to the subsequent clustering involved in PQ/iPQ.

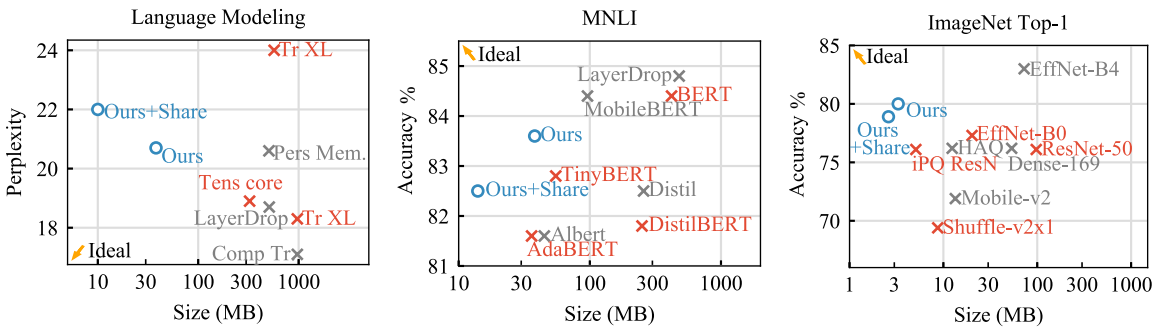


Figure 6-2: **Performance as a function of model size.** We compare models quantized with PQ and trained with the related Quant-Noise to the state of the art. **(a)** Test perplexity on Wikitext-103 **(b)** Dev Accuracy on MNLI **(c)** ImageNet Top-1 accuracy. Model size is shown in megabytes on a log scale. Red and gray coloring indicates existing work, with different colors for visual distinction.

Adding pruning to the quantization noise. The specific form of quantization noise can be adjusted to incorporate additional noise specific to pruning. We simply combine the noise operators of quantization and pruning by composing them following Eq. (6.8). We consider the pruning noise function of Fan et al. (2019) where they randomly drop predefined structures during training. In particular, we focus on *LayerDrop*, where the structures are the residual blocks of highway-like layers (Srivastava et al., 2015), as most modern architectures, such as ResNet or Transformer, are composed of this structure. More precisely, the corresponding noise operator over residual blocks v is $\varphi_{\text{LayerDrop}}(v) = 0$. For pruning, we do not use STE to backpropagate the gradient of pruned weights, as dropping them entirely during training has the benefit of speeding convergence (Huang et al., 2016). Once a model is trained with LayerDrop, the number of layers kept at inference can be adapted to match computation budget or time constraint.

6.5 Experiments

We demonstrate the impact of Quant-Noise on the performance of several quantization schemes in a variety of settings (see Appendix - Sec. C.3).

6.5.1 Improving Compression with Quant-Noise

Quant-Noise is a regularization method that makes networks more robust to the target quantization scheme or combination of quantization schemes during training. We show the impact of Quant-Noise in Table 1 for a variety of quantization methods.

	Language modeling			RoBERTa			Image Classification		
	Comp.	Size	PPL	Comp.	Size	Acc.	Comp.	Size	Acc.
Unquantized									
Original	× 1.0	942	18.3	× 1.0	480	84.8	× 1.0	46.7	81.5
+ Sharing	× 1.8	510	18.7	× 1.9	250	84.0	× 1.4	34.2	80.1
+ Pruning	× 3.7	255	22.5	× 3.8	125	81.3	× 1.6	29.5	78.5
Quantized									
iPQ	× 24.8	38	25.2	× 12.6	38	82.5	× 14.1	3.3	79.0
+ Quant-Noise	× 24.8	38	20.7	× 12.6	38	83.6	× 14.1	3.3	80.0
+ Sharing	× 49.5	19	22.0	× 34.3	14	82.5	× 18.0	2.6	78.9
+ Pruning	× 94.2	10	24.7	× 58.5	8	78.8	× 20.0	2.3	77.8

Table 6.2: **Decomposing the impact of the different compression schemes. (a)** we train Transformers with Adaptive Input and LayerDrop on Wikitext-103 **(b)** we pretrain RoBERTA base models with LayerDrop and then finetune on MNLI **(c)** we train an EfficientNet-B3 on ImageNet. We report the compression ratio w.r.t. to the original model ("Comp.") and the resulting size in MB.

We experiment in 2 different settings: a Transformer network trained for language modeling on WikiText-103 and a EfficientNet-B3 convolutional network trained for image classification on ImageNet-1k. Our quantization noise framework is general and flexible — Quant-Noise improves the performance of quantized models for every quantization scheme in both experimental settings. Importantly, Quant-Noise only changes model training by adding a regularization noise similar to dropout, with no impact on convergence and very limited impact on training speed ($< 5\%$ slower).

This comparison of different quantization schemes shows that Quant-Noise works particularly well with high performance quantization methods, like iPQ, where QAT tends to degrade the performances, even compared to quantizing as a post-processing step. In subsequent experiments in this section, we focus on applications with iPQ because it offers the best trade-off between model performance and compression, and has little negative impact on FLOPS.

Fixed-Point Product Quantization. Combining iPQ and `int8` as described in Section 6.3.3 allows us to take advantage of the high compression rate of iPQ with a fixed-point representation of both centroids and activations. As shown in Table 6.1, this combination incurs little loss in accuracy with respect to iPQ + Quant-Noise. Most of the memory footprint of iPQ comes from indexing and not storing centroids, so the compression ratios are comparable.

Complementarity with Weight Pruning and Sharing. We analyze how Quant-Noise is compatible and complementary with pruning ("+Prune") and weight sharing ("+Share"), see Appendix for details on weight sharing. We report results for Language modeling on WikiText-103, pre-trained sentence representations on MNLI and object classification on ImageNet-1k in Table 6.2. The conclusions are remarkably consistent across tasks and benchmarks: Quant-Noise gives a large improvement over strong iPQ baselines. Combining it with sharing and pruning offers additional interesting operating points of performance vs size.

6.5.2 Comparison with the State of the Art

We now compare our approach on the same tasks against the state of the art. We compare iPQ + Quant-Noise with 6 methods of network compression for Language modeling, 8 state-of-the-art methods for Text classification, and 8 recent methods evaluate image classification on ImageNet with compressed models. These comparisons demonstrate that Quant-Noise leads to extreme compression rates at a reasonable cost in accuracy. We apply our best quantization setup on competitive models and reduce their memory footprint by $\times 20 - 94$ when combining with weight sharing and pruning, offering extreme compression for good performance.

Natural Language Processing. In Figure 6-2, we examine the trade-off between performance and model size. Our quantized RoBERTa offers a competitive trade-off between size and performance with memory reduction methods dedicated to BERT, like TinyBERT, MobileBERT, or AdaBERT.

Image Classification. We compress EfficientNet-B3 from 46.7Mb to 3.3Mb ($\times 14$ compression) while maintaining high top-1 accuracy (78.5% versus 80% for the original model). As shown in Figure 6-2, our quantized EfficientNet-B3 is smaller and more accurate than architectures dedicated to optimize on-device performance with limited size like MobileNet or ShuffleNet. We further evaluate the beneficial effect of Quant-Noise on ResNet-50 to compare directly with the results of Chapter 5.

Incorporating pruning noise into quantization is also beneficial. For example, with pruning iPQ+Quant-Noise reduces size by $\times 25$ with only a drop of 2.4 PPL in language modeling. Further, pruning reduces FLOPS by the same ratio as its compression factor, in our case, $\times 2$. By adding sharing with pruning, in language modeling, we achieve an extreme compression ratio of $\times 94$ with a drop of 6.4 PPL

Language Modeling	PPL	RoBERTa	Acc.
Train without Quant-Noise	25.2	Train without Quant-Noise	82.5
+ Finetune with Quant-Noise	20.9	+ Finetune with Quant-Noise	83.4
Train with Quant-Noise	20.7	Train with Quant-Noise	83.6

Table 6.3: **Quant-Noise: Finetuning vs training.** We report performance after iPQ quantization. We train with the ϕ_{proxy} noise and finetune with Quant-Noise, and use it during the transfer to MNLI for each RoBERTa model.

with FLOPS reduction from pruning entire shared chunks of layers. For comparison, our 10 MB model has the same performance as the 570 MB Transformer-XL base.

6.5.3 Finetuning with Quant-Noise

We explore taking existing models and post-processing with Quant-Noise instead of training from scratch. For language modeling, we train for 10 additional epochs. For RoBERTa, we train for 25k additional updates. Finetuning with Quant-Noise incorporates the benefits and almost matches training from scratch (Table 6.3). In language modeling, there is only a 0.2 PPL difference.

6.6 Conclusion

We show that quantizing a random subset of weights during training maintains performance in the high quantization regime. We validate that Quant-Noise works with a variety of different quantization schemes on several applications in text and vision. Our method can be applied to a combination of iPQ and `int8` to benefit from extreme compression ratio and fixed-point arithmetic. Finally, we show that Quant-Noise can be used as a post-processing step to prepare already trained networks for subsequent quantization, to improve the performance of the compressed model.

Chapter 7

Compressing Faces for Ultra-Low Bandwidth Video Chat

To unlock video chat for hundreds of millions of people hindered by poor connectivity or unaffordable data costs, we propose to authentically reconstruct faces on the receiver's device using facial landmarks extracted at the sender's side and transmitted over the network. In this context, we discuss and evaluate the benefits and disadvantages of several deep learning approaches. In particular, we explore quality and bandwidth trade-offs for approaches based on static landmarks, dynamic landmarks or segmentation maps. We design a mobile-compatible architecture based on the first order animation model of Siarohin et al. (2019). In addition, we leverage SPADE blocks (Park et al., 2019) to refine results in important areas such as the eyes and lips. We compress the networks down to about 3 MB, allowing models to run in real time on iPhone 8 (CPU). This approach enables video calling at a few kbits per second, an order of magnitude lower than currently available alternatives.

7.1 Introduction

For many smartphone users around the world, video-calling remains unavailable or unaffordable. These users are driven out of this fundamental connectivity experience by the prohibitive cost of data plans or because they depend on outdated technologies and infrastructures. For instance, networks might suffer from congestion, poor coverage, power fluctuations and datarate limits – 2G networks allow for a maximum of 30 kbits/s. However, with current technologies, an acceptable video-call quality requires at least a stable 200 kbits/s connection.

Meanwhile, the research in generative models has now come to a point where the

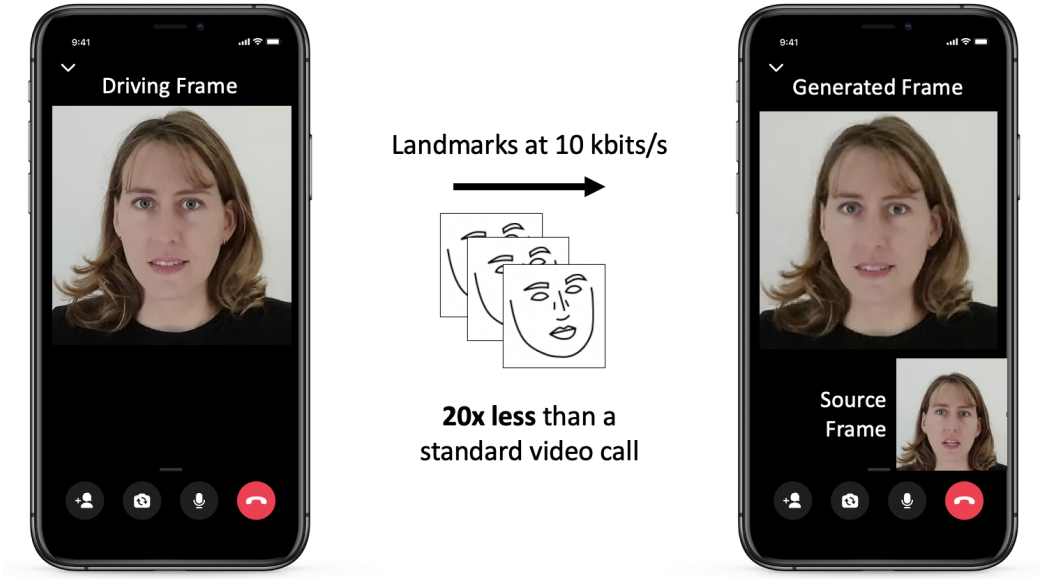


Figure 7-1: We propose to authentically reconstruct faces in real-time on mobile devices using a stream of compressed facial landmarks extracted from *driving* or target frames. The identity of the sender is transmitted one-shot to the receiver at the beginning of the call through a *reference* or *source* frame. This approach is compatible with end-to-end encryption (E2EE).

quality of synthetic faces are sometimes indistinguishable from real videos (Dolhansky et al., 2019). To name a few, we may cite Deep video portraits (Kim et al., 2018), X2Face (Wiles et al., 2018), FSGAN (Nirkin et al., 2019), Neural Talking Heads (Zakharov et al., 2019), the Bilayer model (Zakharov et al., 2020) and the First Order Model (Siarohin et al., 2019). This unprecedented performance can now be exploited to the benefit of higher quality video calls. However, there remain important challenges to address before generative models can offer ultra-low data-rate video-calling. In particular, to unlock duplex video-calling for users with last-mile connectivity issues or limited data plans, the models need to be light enough to run on mobile handsets. Generating the user's face in real time on device is compatible with end-to-end encryption. In addition, to deliver a more seamless and authentic experience, the models should adapt to the current appearance of the user without additional training. In this work, we focus on identifying the best generative strategy compatible with real-time inference on device. We discuss the following approaches:

- The Neural Talking Heads model (Zakharov et al., 2019), which requires sending a stream of landmarks in addition to an initial face embedding.
- The Bilayer model (Zakharov et al., 2020), where the face is reconstructed from

a stream of landmarks and a reference frame sent once.

- The SegFace model, a novel architecture based on SPADE (Park et al., 2019), adapted to face animation, which requires sending an initial face embedding and a stream of semantic segmentation maps.
- The First Order Model (FOM) (Siarohin et al., 2019), which requires sending ten landmarks, their associated motion matrices, and one reference frame.

Analyzing the FOM in depth, we observe that only sending the landmarks compressed with Huffman coding (no motion matrices) achieves sufficient quality and leads to an outstanding data-rate reduction. Compared to other approaches, this model allows for good identity and background preservation. In summary, our contributions are the following:

- We provide a comparative analysis of leading generative approaches for the specific use-case of enabling ultra-low data-rate video calling.
- We develop a strong baseline leveraging the SPADE architecture and the segmentation maps.
- We propose a warping based approach leveraging SPADE blocks to refine important face attributes such as eyes and lips.
- While previous approaches were tested on specialized hardware (servers, mobile GPU), we provide first real-time results on mobile CPU.

7.2 Related Work

7.2.1 Face Compression Before Deep Learning

The idea of face-specific video compression is not novel and appeared with classical computer vision tools, for instance morphings using Delaunay triangulations, Eigenfaces, or 3D models. The first reference we found on the topic is the work of Lopez and Huang (1995) that proposes to encode only pose parameters of a 3D head model, which is projected to reproduce a video sequence.

Previous work (Koufakis and Buxton, 1999) use PCA to model the current frame as a linear combination of 3 basis frames sent prior to the call. The authors rely on known control points on the face boundaries and landmarks. The principal drawback of the approach is the presence of triangulation artifacts, even when a large number of control points is used. The achieved bandwidth is 1500 bits/frame. Similar usage

of Eigenspaces are suggested in (Tuceryan and Flinchbaugh, 2000; Torres and Prado, 2002; Söderström, 2006). Among these proposals using eigenspaces, one claims an extremely low bit-rates achievement of 100 bits/s (Son et al., 2006). However the proposed solution is hard to scale, as it requires storing personal galleries of face images to reconstruct videos at the receiver side.

7.2.2 Deep Compression

The emergence of Generative Adversarial Networks (GANs) stimulated the application of deep learning to video compression. Super-resolution has been an active field of research leveraging GANs for image and video compression. There have been a number of research works tackling this problem (Chen et al., 2018; Ustinova and Lempitsky, 2017; Bulat and Tzimiropoulos, 2018). However, for compressing faces, these reconstructions methods are limited to restoring personal traits from low level images and only work well for limited upscaling factors (around $2\times$ in resolution). The power of GANs for lossy image compression started to be demonstrated in the Generative compression work of Santurkar et al. (2018a), using an auto-encoder combined with adversarial training. The state-of-the-art has since improved with the Extreme Learned Image Compression work of Agustsson et al. (2019), thanks to a multi-scale architecture and the usage of semantic segmentation information, among other tricks used by the authors. The work of Liu et al. (2020) surveys deep learning-based approaches for general purpose video compression. Among them, Learned Video Compression (Rippel et al., 2019) demonstrates for the first time the superior capacity of an end-to-end machine learning approach over standard codecs. By focusing on faces only, we can lower the bandwidth, improve the quality and compress models compared to using more generic methods. Therefore, we review next deep learning approaches and their adequacy to video chat compression.

7.2.3 Deep Talking Head Approaches

3D based approaches produce realistic avatars which can be animated in real-time (Cao et al., 2016). However, such methods require to capture a set of images of the user (a few dozens) to build their personal face model. PAGAN (Nagano et al., 2018) generates key face expression textures that can be deformed and blended in real-time on mobile from a single frame. However, the reconstruction of certain features, notably the hair, is still problematic in 3D model-based approaches. Deep video portraits (Kim et al., 2018) is handling this issue using a rendering-to-video trans-

lation network, but the approach needs about a thousand images per subject for training. Stimulated by advancements in face swapping pipelines (Korshunova et al., 2017; Wiles et al., 2018), a number of deep generative re-enactment approaches arose. Contrary to warping based re-enactment (Averbuch-Elor et al., 2017), learning faces reconstructions enables extra robustness in presence of large head angles. The Face Swapping GAN (Nirkin et al., 2019) relies on several steps: landmarks extraction, segmentation, interpolation and inpainting. This complex pipeline may result in robustness issues and limited bandwidth gain due to the need of sending both compressed segmentations and landmarks. Similarly, the vid2vid approach ((Wang et al., 2018b), (Wang et al., 2019b)) requires sending a "sketch" (edge map) for each frame in order to re-enact a face, which has a relatively high bandwidth cost.

7.3 Generative Models

In this section we describe in-depth several recent face animation algorithms that we have implemented and studied. We share our understanding of these works and present our two model contributions, namely SegFace and Hybrid Motion-SPADE. An overview of these different models appears in Figure 7-2. For this self-reenactment task, unless mentioned otherwise, the goal of all these approaches is to generate a frame based on (i) one fixed *reference* or *source* frame and (ii) position information (e.g. landmarks) from a stream of *driving* or *target* frames (see Figure 7-1). Implementation details are located in the original paper of Oquab et al. (2020).

7.3.1 Talking Heads (NTH) and Bilayer Model

The "Talking Heads" work of Zakharov et al. (2019) learns to synthesize videos of people from facial landmarks given one reference image. It follows an encoder-decoder architecture with a style transfer component. A set of style parameters is computed for the set of reference images. Then, facial landmarks are plotted as an image and processed by an encoder network. The resulting code is decoded with style transfer, using Adaptive Instance Normalization (Huang and Belongie, 2017) layers, adjusting the mean and standard deviation of each feature map with the style parameters.

The networks are trained end-to-end with adversarial and perceptual losses on a dataset of videos. The best results are achieved by performing a fine-tuning training phase on the generator to match the reference frames as precisely as possible. This fine-tuning phase requires several minutes on a modern server GPU. Without fine-

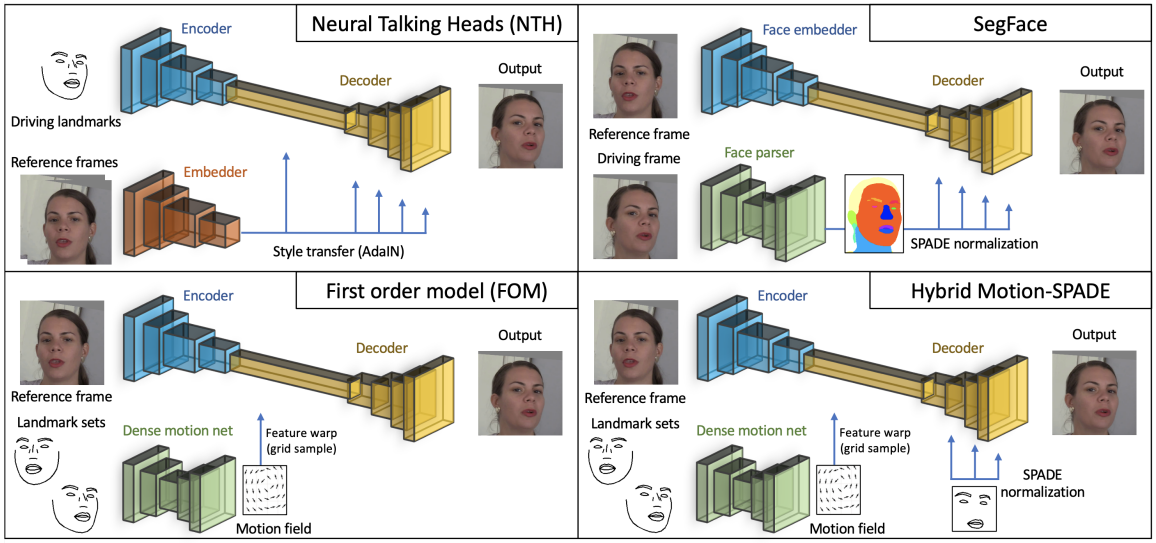


Figure 7-2: Scheme of principle for the different deep generative approaches discussed in this study. In particular, we detail two novel architectures, SegFace and Hybrid Motion-SPADE (right) and compare them to existing NTH (Zakharov et al., 2019) and FOM (Siarohin et al., 2019) models (left). For all models, we assume the generation is performed by the encoder-decoder pair on the receiver device, while the emitter sends a reference frame (or several) at the beginning of inference, and streams a series of landmarks or segmentation maps.

tuning, the identity is not preserved as well in the generated frames. In practice, a few hundreds of frames would have to be sent at the beginning of the call.

This work was further improved in the Bilayer Synthesis approach (Zakharov et al., 2020), where the fine-tuning step is not required anymore, and leads to visually appealing and sharp results. In our observations (see Figure 7-3), the identity preservation suffers from a stronger uncanny valley effect. In terms of bandwidth, the NTH and Bilayer approaches require sending 68 compressed landmarks.

7.3.2 First Order Model for Image Animation (FOM)

The "First Order Model" approach of Siarohin et al. (2019) deforms a reference source frame to follow the motion of a driving video. While this method works on various types of videos (Tai-chi, cartoons), we focus here on the face animation application. FOM follows an encoder-decoder architecture with a motion transfer component. Fist, a landmark extractor is learned using an equivariant loss, without explicit labels. Then, two sets of ten learned landmarks are computed for the source and driving frames, and a dense motion network uses the landmarks and the source frame to produce a dense motion field and an occlusion map. In parallel, the encoder encodes

the source frame. Next, the resulting feature map is warped using the dense motion field (through a grid-sample operation (Jaderberg et al., 2015)), then multiplied with the occlusion map. Finally, the decoder generates an image from the warped map.

The networks are trained end-to-end on video frames, using perceptual losses, and are then optionally fine-tuned with an adversarial discriminator. The self-supervised landmarks do not necessarily match precise locations of the face. Instead, they correspond to point coordinates that are optimized to achieve the best deformation of the source frame. Siarohin et al. (2019) describes how to improve motion approximation in landmark areas by estimating Jacobian matrices to model motion in their neighborhood. In our observations (see Table 7.3), this approach preserves identities better than NTH and is at least on par with the follow-up Bilayer synthesis approach. Next, we study variants of this approach.

Variants First, our implementation does not use the Jacobian component, as we do not observe a strong influence on the results. We refer to the resulting model as "Motion Net (MN-10)" as it no longer uses first order approximation and employs a set of ten landmarks. Second, we explore using off-the-shelf facial landmarks extraction to complement the unsupervised landmarks. In this case, we only stream 20 or 68 compressed landmarks. Third, we explore a combined strategy employing both 10 self-supervised landmarks and 20 supervised ones, that we note MN-10+20. We will introduce a fourth variant in 7.3.4, after detailing our SegFace approach below.

7.3.3 SegFace

This novel approach builds upon Park et al. (2019). Unlike MaskGAN (Lee et al., 2020), we propose to use a face descriptor computed on a source frame, and decode it by conditioning on face segmentation maps from a driving frame. It follows an encoder-decoder architecture: a face descriptor is computed on a source frame and given to a decoder network, that applies SPADE normalization blocks at each layer using the face segmentation maps of the driving frame, ensuring all parts of the face are correctly placed. The decoder network is trained using VGGFace2 face embeddings (Cao et al., 2018), and segmentation maps from Yu et al. (2018a) as inputs. Its objective during training is to reconstruct the same source frame. The optimization is done using losses from Park et al. (2019), and the face perceptual loss from Gafni et al. (2019). This method operates on independent frames, and thus allows us to use high-resolution training data, leading to high quality results. Training is achieved using CelebA (Liu et al., 2015) and Flickr-Faces-HQ datasets.

Bandwidth The model requires a segmentation map labeled for 15 categories (eyes, hairs, ears etc.). Sending compressed segmentation maps would require 18/25 kbits/s at resolutions 48×/64×, knowing that there is a trade-off between the resolution of the transmitted segmentation maps and the quality of the generated faces. We do not build on this method further for low-bandwidth video-chat because the cost of running a face parser inference step and the bandwidth requirements are too high. The SegFace implementation, however, allows us to observe that the generated images respect the segmentation map labels almost perfectly, consistently with the conclusions of Lee et al. (2020). We will build on this property in the next subsection with our Hybrid Motion-SPADE approach.

7.3.4 Hybrid Motion-SPADE Model

Important quality criteria for compressed video-chat include a good synchronization between the lips and the speech, and a good rendering of the eyes and eyebrows; therefore, it is crucial to generate these facial parts precisely. We propose an improvement over the FOM-based Motion Net method, by adding SPADE normalization layers in the upsampling blocks of the decoder network (in the last step of the FOM approach). We draw polygons for the eyes, eyebrows, lips and inner mouth using 60 extracted face landmarks, and use these as semantic maps for SPADE.

The dense motion network receives (i) a downsampled reference frame with (ii) the positions of N landmarks for that frame, and (iii) the positions of the same landmarks for a driving frame. It outputs a motion field M and an occlusion map O. The encoder network outputs a feature map F_s. The decoder warps F_s with the result of the dense motion network M and multiplies it element-wise with the occlusion map O, to obtain F_w. Then, F_w is processed by a stack of five residual blocks and three upsampling blocks that apply the SPADE normalization using a set of 60 landmarks.

Bandwidth The necessary segmentation maps are obtained by plotting the polygons of the facial landmarks extracted using a landmark detector (see Figure 7-2), rather than running a face segmentation network. Moreover, landmark coordinates are inexpensive to transmit, while rasterized segmentation maps are more difficult to compress, especially at higher resolutions. In terms of bandwidth, this approach requires sending $N + 60$ compressed landmarks. We experiment with $N = 10, 20, 30$.

Model variant	Inputs	FPS	Params	FLOPS	int8 size	BW
Motion Net	10U	18	2.9 M	1411 M	3.1 MB	1.4 kbits/s
Motion Net	20L	19	2.3 M	1293 M	2.5 MB	2.2 kbits/s
Motion Net	10U + 20L	14	3.0 M	1505 M	3.4 MB	3.6 kbits/s
Motion SPADE	10U	16	2.9 M	1198 M	3.2 MB	8.0 kbits/s
Motion SPADE	20L	19	2.3 M	1029 M	2.5 MB	8.8 kbits/s
Motion SPADE	10U + 20L	13	3.0 M	1292 M	3.4 MB	10.2 kbits/s

Table 7.1: Comparison of our approaches running on mobile in terms of compression for both model size and stream."10U" (resp. "20L") means that 10 unsupervised keypoints (resp. 20 facial landmarks) are used as inputs to the dense motion network. SPADE variants require 60 extra facial landmarks to draw the facial label maps. Notes: the "int8 size" is the full combined size of the models. The number of frames per second (FPS) is measured for the whole int8-quantized pipeline running on an iPhone 8, including landmark detection, grid-samples and face alignment. The #FLOPS count is for the dense motion, decoder, and unsupervised keypoint extractor networks. The bandwidth (BW) is measured at 25 FPS with Huffman encoding.

7.4 Compression

In this section, we explain different strategies to make architectures – and in particular our novel hybrid Motion-SPADE – compatible with low-bandwidth video calls on mobile. We first detail the architectures and then the compression aspects for the models and the bandwidth. Results are displayed in Table 7.1.

7.4.1 Mobile Architectures

Base blocks We rely on the open-source FbNet family of architectures (Wan et al., 2020; Dai et al., 2020) to design mobile-capable models for our Motion Net and Motion-SPADE approaches. These networks typically build on blocks combining 1×1 pointwise and 3×3 depthwise convolutions (Sandler et al., 2018a) that require less floating-point operations than traditional 3×3 convolutions in residual blocks.

Mobile SPADE normalization blocks When applicable, we perform a SPADE normalization after the last 1×1 pointwise convolution, with kernel sizes of 1×1, and 32 hidden channels. We have found these parameters to provide a good tradeoff between speed and quality while preserving the fidelity of the SPADE approach.

7.4.2 Landmark Stream Compression

We compress the landmarks with Huffman encoding (Huffman, 1952). In this approach, the landmark displacements are first binarized into 32 bins plus one bit sign, and we encode the bin index with Huffman coding. This compression leads to an average rate of 130 bits/frame for 20 landmarks, hence 3 kbits/s at 25 FPS (see Table 7.1 for details). For reference, bandwidth requirements for audio are around 10 kbits/s, while the AV1 video codec (not widely hardware-supported to date) aims at 30 kbits/s (Citron, 2020). Therefore, we did not explore other variants such as Arithmetic Coding (Rissanen and Langdon, 1979) since the audio part takes most of the bandwidth of a call with the proposed approach.

7.4.3 Model Quantization

We rely on `int8` post-training quantization. As detailed in Section 2.3.5 of the Related Work, this technique consists in uniformly quantizing both weights and activations over 8 bits, reducing the model size by a factor 4. Moreover, `int8` models traditionally benefit from a $\times 2$ or $\times 3$ speed-up compared to their `fp32` counterparts for both server and mobile CPUs. The scale and zero-point [1] are calibrated after training using a few batches of training data. When not properly calibrated, we found that the decoder generates an image with a small amount of noise resulting in a loss of visual quality.

7.5 Experiments

7.5.1 Quantitative Evaluation

We evaluate the models using the perceptual LPIPS (Zhang et al., 2018b) and multi-scale LPIPS-like metrics employed in Siarohin et al. (2019), that we name msVGG. Second, as argued in Chen et al. (2020b), the cosine similarity CSIM computed between features of the pre-trained face embedding network ArcFace (Deng et al., 2019) is one of the most effective metric to assess quality of talking heads models, we therefore report it. Finally, we quantify facial landmarks mismatch by running a landmark detector on the true and generated videos and computing the Mean Square Error between each pair of landmarks. This metric is classically referred to as the Normalized Mean Error (NME) of head pose (Bulat and Tzimiropoulos, 2017). All the generative

[1] The affine transform coefficients that converts an 8-bit quantized tensor (integer-valued in $[0, 255]$) to its floating-point counterpart.

approaches considered in this work are trained using different alignments and close-ups (see Figure 7-3), so we compute our metrics using the optimal modified videos for each method as ground truth. Ablation studies are displayed in Appendix D.1.1 and the quantitative evaluation of the models is displayed in Table D.2.

Figure 7-3: Comparison of different results using Seg2Face (48×48), NTH, Bilayer, and FOM adv. The model generates the face using the fixed source frame and the facial information (such as keypoints) of the driving frame. We pasted ground truth backgrounds to have a fair evaluation. The last column showcases the results using our Mobile Motion-SPADE that runs at 18 FPS on an iPhone 8, whereas the other models run on server, have at least $10\times$ more parameters and are not necessarily compatible with low-bandwidth video calling. Note that the alignment procedure may differ between the models, hence the head is not centered the same on the generated faces (last 5 columns).

7.5.2 Qualitative Evaluation and Human Study

Figure 7-3 compares results obtained using SegFace, Bilayer, NTH and FOM with adversarial finetuning. We observe skin tones differences between targets and SegFace results and distortions of personal traits. The Bilayer, NTH and FOM models are qualitatively better. In the last column, we observe a side by side comparison with a Mobile Motion-SPADE model.

Table 7.3 provides a quality assessment of different models by human raters. Participants are asked to rate images produced by the different models by comparing

	LPIPS ↓	NME ↓	CSIM ↑
Dense MN-10 U	0.221	0.59	0.83
Dense MN-20 L	0.242	0.50	0.80
Dense MN-68 L	0.240	0.49	0.81
Mob MN-10 U	0.225	0.52	0.79
Mob MN-20 L	0.244	0.48	0.78
Mob MN-10 U + 20 L	0.218	0.46	0.80
Mob M-SPADE-10 U	0.217	0.47	**0.81**
Mob M-SPADE-20 L	0.242	**0.44**	0.79
Mob M-SPADE-10 U + 20 L	**0.215**	0.46	**0.81**

Table 7.2: Evaluation results for Motion Net approaches without adversarial fine-tuning on the VoxCeleb2-28 video subset. Mob : Mobile models. Dense models (64×64 latent space) are trained on VoxCeleb. Mobile models (32×32) are trained on the DFDC aligned dataset. U: unsupervised keypoints; L: facial landmarks.

them in terms of identity and expression preservation, on a scale from 1 to 5. In a first round of evaluations, we display side by side the four main dense models results, and in the second round, Motion Net and Motion-SPADE results using six different mobile architectures. We collect in each case 500 pairwise evaluations, each from 5 different participants. Dense approaches results present a large variability and human scores seem to agree with metrics. Mobile models results differences are more subtle, but the Hybrid Motion-SPADE using 10 landmarks model seems preferred.

Figure D-1 illustrates the quality performance reached on mobile. The quality of results degrades in presence of large head rotations. Still, the Motion-SPADE results are visually close to the targets, particularly it renders lips and teeth better. The H264 compression results are displayed given a bandwidth of 9 kbit/s, to be compared to the ones of the Motion-SPADE 20 model that runs the fastest on mobile. At this bandwidth, video transmission is hardly possible using standard codecs.

7.5.3 On-device Real-time Inference

We develop a standalone app that is able to perform a real-time call between two participants. In order to do so, we converted the models to int8, then to TorchScript and run them on the phone's CPU. To compress the Motion based models, we only rely on `int8` since the non-compressed models are already small. We use WebRTC to also feature the sound during the call to make the experience more immersive. Screenshots of the app are displayed in Figure 7-4.

Dense models		
model	human identity score	human expression score
NTH	3.71±0.041	3.77±0.040
Bilayer	3.70±0.046	3.62±0.046
SegFace	3.11±0.047	3.00±0.051
FOM adv	**3.99**±0.042	**4.00**±0.041

Overall human ratings of Mobile models			
	MN-10	MN-20	MN-10+20
no SPADE	$3.44_{\pm 0.034}$	$3.40_{\pm 0.034}$	$3.46_{\pm 0.034}$
with SPADE	$\mathbf{3.50}_{\pm 0.034}$	$3.46_{\pm 0.035}$	$3.45_{\pm 0.034}$

Table 7.3: Quality assessment of different dense models on DFDC-50 - Human study. Average scores (Higher is better) with confidence intervals.

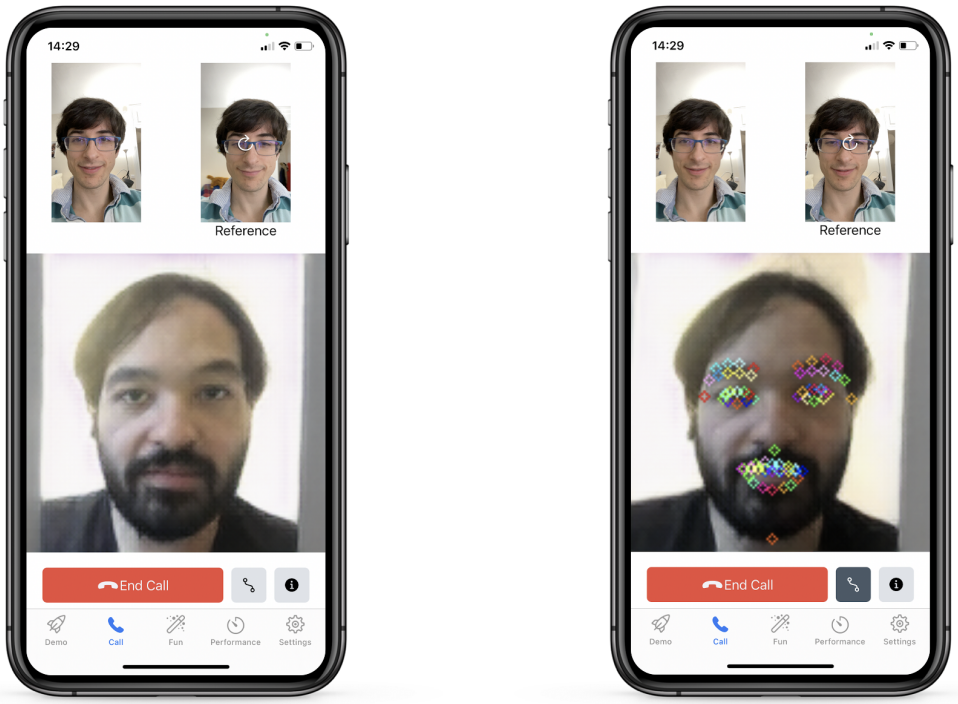


Figure 7-4: Daniel (with glasses) is calling Maxime on our iPhone app. Left: Daniel's phone when calling Maxime. The reference frame of Daniel (top right) of Daniel is updated periodically. Daniel sees the face of Maxime reconstructed in real time. Right: on the generated frame, we display the set of landmarks used to perform the inference. The generation runs at 20 FPS on an iPhone 8 and the call uses an average video bandwidth of 5 kbits/s.

7.6 Conclusion

Our exploration of state-of-the art animation models led us to the following observations: The Neural Talking head results are qualitatively satisfactory, but the fine-tuning step requirement makes the approach complex to implement in practice. Using a full face segmentation approach seems unfit to a low bandwidth application. The Bilayer approach and the FOM methods perform best towards reaching a correct low-bandwidth/quality trade-off on mobile. Our human study shows that FOM results are preferred in terms of identity and expression preservation. Focusing on this best candidate approach, we design a novel hybrid architecture taking advantage of the high fidelity to the target thanks to the warping principle, and enhancing the quality of important attributes with SPADE blocks. Only exploiting polygons induced segments with this approach improves quality without high transmission cost. The obtained image quality is close to the one reached by dense models while running in real-time on Mobile CPU. The bandwidth required to send a video is lower than the one required for sending audio. There are a number of interesting challenges to tackle next to improve quality of the generations, *e.g.* generating large head rotation movements, hands, or using pupils tracking.

Chapter 8

Discussion

In this Chapter, we briefly summarize our contributions and sketch some interesting future research directions.

8.1 Summary of Contributions

In this manuscript, we presented our contributions on the redundancy and efficiency of Neural Networks in ascending order of applicability.

8.1.1 Equivalence Classes

In Chapter 3, we studied the functional equivalence classes of Neural Networks from a local perspective in the space of parameters for networks with arbitrary depth, relying on our standalone characterization for one-hidden-layer networks along with algebraic and geometrical tools that we designed. This allows us to group networks that behave similarly by orbits under the action of the rescaling the permutation operations. Using these theoretical considerations, we developed an alternative to SGD that operates in the quotient space in Chapter 4, called ENorm. The method selects the representant of the current orbit that minimizes a certain energy after each SGD step. This work aiming at properly balancing the energy of the network was later leveraged by Nagel et al. (2019) to design a robust `int8` quantization scheme that *equalizes* the weights before compression.

8.1.2 Neural Network Compression

In Chapter 5, we designed a product quantization method, called iPQ, that allows us to drastically reduce the size of a network while almost preserving its accuracy. This

method was improved by Martinez et al. (2020), who also learn an adequate permutation of the weights to enforce better redundancy when considering large quantization blocks. Next, aiming at applying iPQ to any architecture, we developed a preconditioning method called Quant-Noise that injects carefully selected quantization noise when training the network before its compression in Chapter 6. Quant-Noise yields competitive results for scalar quantization techniques such as `int8`.

8.1.3 Low-Bandwidth Video Chat

The compressed size of the network is a significant indicator of the quality of the quantization. However, other metrics such as inference time and battery usage are also relevant, especially for on-device, real time applications. In order to confront our findings to real-world constraints, we developed a low-bandwidth generative video chat called FaceGen in Chapter 7. FaceGen generates the face on the receiver side using a GAN conditioned on a stream of landmarks plus an identity embedding sent at the beginning of the call. FaceGen models weight a total of less than 2 MB, run in real-time on an iPhone 8 and compress the video bandwidth to less than 10 kbits/s.

8.2 Future Directions

We list below some directions for further research on the short and longer run.

8.2.1 Equivalence Classes

Equivalence classes allow us to conveniently operate in the quotient space of neural networks, provided that we know the right representant to manipulate.

Canonical Representant. To properly benefit from the theoretical advances regarding the functional equivalence classes, the question of the choice of the adequate representant is key. For instance, with ENorm in Chapter 4, we interleave SGD updates with a step that selects the representant of the current orbit that minimizes the L_2 global norm of its weights. While this particular choice leads to a fast iterative algorithm and presents empirical evidence of its pertinence, this criterion is not theoretically grounded. One possibility would be to select the adequate representant such that the next SGD step is equivalent to a second order natural gradient update.

Non-local Characterization. On the more theoretical side on the characterization of the functional equivalence classes, we wish to reconcile the sufficient condition for restricted local identifiability and the necessary condition for local identifiability in Chapter 3 to properly characterize locally identifiable parameterizations. Next, we argue that having a global characterization would necessitate to take the permutations into account and to design a more general way to modify θ as follows: keep the multiplicative updates of the form we^{γ} for non-zero weights w or biases b and default back to additive updates of the form $w + \delta$ for null parameters.

8.2.2 Neural Network Compression

Compression and quantization is a major challenge for the research industry in order to deploy deep learning models on-device and for more privacy.

Deployment of iPQ. First, we hope that our iPQ + `int8` implementation and open-sourced code (see Chapters 5 and 6) will unlock on-device model deployment for models with 10 to 100 million parameters. Such models would enable model deployment off-the-shelf as follows. First, compress then model with iPQ + `int8` and send it on-device. Then, decompress the iPQ part on-device to default back to a plain `int8` model to benefit from the speedup of `int8`. Further research directions include designing our own kernels to perform the forward pass in the iPQ + `int8` domain. This would imply to pre-compute the dot products between activations and the codebook to build and store Look-Up Tables (LUTs) in low-level cache memory and then lookup and sum the results. Larger block sizes, fewer codewords and larger underlying matrices would help reduce the latency.

Compression as a Service. One of the current limitations of iPQ is that the number of codewords and the block size has to be manually tuned using in-domain knowledge. For instance, we observed that the classifiers in convolutional networks are harder to compress and therefore use a block size of 4 instead of 8 and/or more codewords for this particular layer. Conversely, some methods (see Subsection 2.3.5) for mixed scalar quantization propose to learn the precision – the number of bits – per layer in a fully differentiable way. One interesting direction would be to apply such a paradigm to iPQ to foster its adoption. One orthogonal direction would be to derive heuristics or methods to select the right compression method for a specific task, target metrics and hardware. Currently, as explained in Section 2.3, the current process is to tailor one of the many available methods listed in Section 2.3 – or a

combination thereof – to a specific task by trials-and-errors. For instance, distillation and then iPQ proved to work well in that order. Adaptability would be a determinant factor for the broad adoption of compression methods.

Training Bigger Models. The next generation of Nvidia's GPUs, the A100, now natively supports `int8` and `int4` additionally to half-precision training in `fp16`. To benefit from this theoretical speedup, we could train models with more parameters than currently done, provided that we are able to design a training recipe to preserve the accuracy of the network. Hence, advances in both the hardware and the training techniques could help taking neural networks to the next level.

8.2.3 Low-Bandwidth Video Chat

While incremental changes in the current method would make face generation more convincing, such methods also pave the way for 3D face generation.

Incremental changes. There are many possible incremental ameliorations to the current setup of Chapter 7. For instance, we could improve the data loaders to sample more large head rotations since the model performs currently poorly on these. We could also be more strict when sampling the reference frame and have a neutral expression with a person facing the camera. In the search for more efficiency, we could also incorporate new blocks such as FBNetV3 blocks (Dai et al., 2020) and optimize some operations performed in the Dense Motion network such as the grid sample.

3D Face Reconstruction. On the longer term, similar insights could be applied to 3D face reconstruction, with or without the low bandwidth constraint. The idea is to leverage (1) the growing availability of augmented and virtual reality devices and (2) the desire of people to stay close to their relative ones that live remotely. On traditional devices, the 3D face could be projected back to 2D images, while on more advanced VR/AR headsets such as the Oculus Quest 2, this could provide an immersive and realistic experience of remote presence.

Appendix A

Proofs for Functional Equivalence Classes

A.1 Permutation-Rescaling Equivalence

A.1.1 Reconciling Definitions

Proof of Proposition 3.1.1

Proposition. *Let θ be an admissible parameterization. Then, for every θ', admissible or not, and every $a, b \in \{Layer, Neuron, Path, Trajectory\}$, $\theta \sim_a \theta' \iff \theta \sim_b \theta'$.*

Proof. We prove successive implications.

Layer $\implies$ Neuron. If $\theta' \sim_{\text{Layer}} \theta$, let us define $\gamma \in \mathbb{R}^H$ such that, for every hidden layer $1 \leq \ell \leq L-1$,

$$D^{(\ell)} = \text{Diag}(\gamma_\nu)_{\nu \in N_\ell}. \tag{A.1}$$

By assumption on the matrices $D^{(\ell)}$, we have that $\gamma_\nu > 0$ for every $\nu \in H$. Let ν be a neuron in layer $\ell = 1$. Since $W'^{(1)} = W^{(1)} D^{(1)}$, we have for any $\mu \in \text{prev}(\nu)$

$$w'_{\mu \to \nu} = w_{\mu \to \nu} \gamma_\nu. \tag{A.2}$$

Next, let ν be a neuron in layer $2 \leq \ell \leq L-1$ and $\mu \in \text{prev}(\nu)$. Since $\left(D^{(\ell-1)}\right)^{-1} W^{(\ell)} D^{(\ell)}$, the following holds

$$w'_{\mu \to \nu} = w_{\mu \to \nu} \gamma_\nu / \gamma_\mu. \tag{A.3}$$

Finally, let μ be a neuron in the last hidden layer $L-1$ and $\nu \in \text{next}(\mu)$. Since

$W^L = \left(D^{(L-1)}\right)^{-1} W^{(L)}$, we have

$$w'_{\mu\to\nu} = w_{\mu\to\nu}/\gamma_\mu. \tag{A.4}$$

Finally, regarding the biases, $b'^{(\ell)} = b^{(\ell)}D^{(\ell)}$ for every layer $1 \leq \ell \leq L$ gives, for every hidden neuron $\nu \in H$, $b'_\nu = \gamma_\nu b_\nu$. Thus, defining s as in representation (2.8), we obtain $s(\theta) = \theta'$. Hence, $\theta \sim_{\text{Neuron}} \theta'$.

Neuron $\implies$ Path. If $\theta \sim_2 \theta'$, there exists s as in representation (2.8) such that $\theta' = s(\theta)$. First, we deduce that $\text{sign}(\theta) = \text{sign}(\theta')$. Next, let $p = (\nu_0, \dots, \nu_L) \in \mathcal{P}$ be a full path. Then,

$$\begin{aligned} v_p(\theta') &= w'_{\nu_0\to\nu_1} w'_{\nu_1\to\nu_2} \dots w'_{\nu_{L-1}\to\nu_L} && \text{(A.5)}\\ &= \left(\gamma_{\nu_1} w_{\nu_0\to\nu_1}\right)\left(\frac{\gamma_{\nu_2}}{\gamma_{\nu_1}} w_{\nu_1\to\nu_2}\right) \dots \left(\frac{1}{\gamma_{\nu_L}} w'_{\nu_{L-1}\to\nu_L}\right) && \text{(A.6)}\\ &= w_{\nu_0\to\nu_1} w_{\nu_1\to\nu_2} \dots w_{\nu_{L-1}\to\nu_L} && \text{(A.7)}\\ &= v_p(\theta). && \text{(A.8)} \end{aligned}$$

Similarly, we show that for every partial path $q \in \mathcal{Q}$, $b_{q_0} v_q(\theta) = b'_{q_0} v_q(\theta')$.

Path $\implies$ Trajectory. If $\theta \sim_{\text{Path}} \theta'$, since $\text{sign}(\theta') = \text{sign}(\theta)$, using Proposition 3.3.5, there exists a unique $\gamma = (\alpha, \mathbf{S}\beta) \in \text{Supp}_\theta$ such that $\theta' = \theta \odot e^\gamma$. Let us show that γ is θ-rescaling-compatible. Let us show first that $\alpha \in \ker(\mathbf{P}_\theta)$. Since $(\mathbf{P}_\theta \alpha)_p = \mathbf{1}(v_p(\theta) \neq 0)\mathbf{P}\alpha$, it suffices to show that, for every full path $p \in \mathcal{P}$ such that $v_p(\theta) \neq 0$, we have $(\mathbf{P}\alpha)_p = 0$. We compute

$$v_p(\theta') = v_p(\theta) \prod_{e\in p} \exp(\alpha_e) = v_p(\theta) \exp\left(\sum_{e\in p} \alpha_e\right) = v_p(\theta) \exp((\mathbf{P}\alpha)_p). \tag{A.9}$$

Using the fact that $v_p(\theta') = v_p(\theta)$, we deduce that $(\mathbf{P}\alpha) = 0$. Thus, $\alpha \in \ker(\mathbf{P}_\theta)_p$. Next, let us similarly show that $\alpha - \beta \in \ker(\mathbf{Q}_\theta)$. Since $(\mathbf{Q}_\theta \alpha)_q = \mathbf{1}(b_{q_0} v_q(\theta) \neq 0)\mathbf{P}\alpha$, it suffices to show that, for every partial path $q \in \mathcal{Q}$ such that $b_{q_0} v_q(\theta)$, we have $(\mathbf{Q}\alpha)_q = 0$. We compute

$$b'_{q_0} v_q(\theta') = b_{q_0} v_q(\theta) \exp(\mathbf{S}\beta_{q_0}) \exp\left(\sum_{e\in q} \alpha_e\right).$$

Recalling that the definition of $\mathbf{S}\beta_{q_0}$ is independent of the choice of the path going

from q_0 to any output neuron (see Proposition 3.3.6 since $\beta \in \ker \mathbf{P}$),

$$\begin{aligned} b'_{q_0} v_q(\theta') &= b_{q_0} v_q(\theta) \exp\left(-\sum_{e \in q} \beta_e\right) \left(\sum_{e \in q} \alpha_e\right) \\ &= b_{q_0} v_q(\theta) \exp\left(\sum_{e \in q} (\alpha_e - \beta_e)\right) \\ &= b_{q_0} v_q(\theta) \exp\left(\left(\mathbf{Q}(\alpha - \beta)\right)_q\right). \end{aligned}$$

Using the fact that $b'_{q_0} v_p(\theta') = b_{q_0} v_q(\theta)$, we deduce that $\left(\mathbf{Q}(\alpha - \beta)\right)_q = 0$. Thus, $\alpha\beta \in \ker\left(\mathbf{Q}_\theta\right)$. Thus, we have proven that $\theta \sim_{\text{Trajectory}} \theta'$.

Trajectory $\implies$ Layer. If $\theta \sim_{\text{Trajectory}} \theta'$, there exists a θ-rescaling-compatible $\gamma = (\alpha, \mathbf{S}\beta) \in \text{Supp}_\theta$ such that $\theta' = \theta \odot e^\gamma$. Let us define

$$\delta = \exp(\mathbf{S}\alpha) \in \mathbb{R}^H. \tag{A.10}$$

Note that δ is well-defined since $\alpha \in \ker\left(\mathbf{P}_\theta\right)$. Let $\nu \in H$ such that $b_\nu \neq 0$ and let q be a partial path going from ν to the output neuron such that $v_q(\theta) \neq 0$, possible since θ is admissible. Let us show that $(\mathbf{S}\beta)_\nu = (\mathbf{S}\alpha)_\nu$. Recalling that $\alpha - \beta \in \ker\left(\mathbf{Q}_\theta\right)$,

$$b_{q_0} v_q(\theta) \sum_{e \in q} \alpha_e = b_{q_0} v_q(\theta) \sum_{e \in q} \beta_e. \tag{A.11}$$

Thus, since $b_{q_0} v_q(\theta) \neq 0$, we deduce $(\mathbf{S}\beta)_\nu = (\mathbf{S}\alpha)_\nu$. Next, let us define for every hidden layer $1 \leq \ell \leq L-1$,

$$D^{(\ell)} = \text{Diag}(\delta_\nu)_{\nu \in N_\ell}. \tag{A.12}$$

Next, recall that $w' = w \cdot \exp(\alpha)$. Let $e = \mu \to \nu \in E$.

- If $\mu \in N_0$, let us show that $w'_e = w_e \delta_\nu$ for every $e \in E$ such that $w_e \neq 0$. First, We have $w'_e = w_e \exp(\alpha_e)$. Moreover,

  $$(\mathbf{S}\alpha)_\nu = \sum_{e' \in q} \alpha_{e'} \tag{A.13}$$

  where q is any partial path going from ν to the output neuron such that $v_q(\theta) \neq 0$. Since $p = (\mu) + q$ (where $+$ denotes path concatenation) is also such that

$v_p(\theta) \neq 0$, and since $\alpha \in \ker \mathbf{P}_\theta$, we have

$$0 = \sum_{e' \in p} \alpha_{e'} = \alpha_e + \sum_{e' \in q} \alpha_{e'} = \alpha_e - (\mathbf{S}\alpha)_\nu \tag{A.14}$$

Thus, $(\mathbf{S}\alpha)_\nu = \alpha_e$ and $w'_e = w_e \exp(\alpha_e) = w_e \exp((\mathbf{S}\alpha)_\nu) = w_e \delta_\nu$.

- If $\nu \in N_L$, let us show that $w'_e = w_e/\delta_\mu$ $e \in E$ such that $w_e \neq 0$. Since $q = (\mu, \nu)$ is a path going from μ to the output neuron such that $v_q(\theta) \neq 0$, we have $(\mathbf{S}\alpha)_\mu = -\alpha_e$. Thus, $w'_e = w_e \exp(\alpha_e) = w_e \exp((\mathbf{S}\alpha)_\nu) = w_e/\delta_\nu$.

- Otherwise, let us show that $w'_e = w_e \delta_\nu/\delta_\mu$ for every $e \in E$ such that $w_e \neq 0$. Let q be a partial path going from neuron ν to the output neuron such that $v_q(\theta) \neq 0$ and let $\overline{q} = (\mu) + q$. We observe that $v_{\overline{q}}(\theta) \neq 0$ and that

$$\alpha_e = \sum_{e' \in \overline{q}} \alpha_{e'} - \sum_{e' \in q} \alpha_{e'} = (\mathbf{S}\alpha)_\nu - (\mathbf{S}\alpha)_\mu \tag{A.15}$$

Thus, $w'_e = w_e \exp(\alpha_e) = w_e \exp((\mathbf{S}\alpha)_\nu - (\mathbf{S}\alpha)_\mu) = w_e \delta_\nu/\delta_\mu$.

Thus, for all $1 \leq \ell \leq L$, $W'^{(\ell)} = \left(D^{(\ell-1)}\right)^{-1} W^{(\ell)} D^{(\ell)}$ and $b'^{(\ell)} = b^{(\ell)} D^{(\ell)}$. Therefore, $\theta' \sim_{\text{Layer}} \theta$, which concludes the proof. □

A.1.2 Link with Functional Equivalence

We formally prove separately that the permutation and rescaling operations preserve the function implemented by the network. Next, we show that permutation and rescaling operations commute, which allows us to define permutation-rescaling equivalent parameters. Finally, we show that permutation-rescaling equivalence implies functional equivalence.

Proposition A.1.1. *Two rescaling equivalent parameterizations θ and θ' yield the same realization,* i.e. *$R_\theta(x) = R_{\theta'}(x)$ for all x.*

Proposition A.1.2. *Two permutation equivalent parameterizations θ and θ' yield the same realization,* i.e. *$R_\theta(x) = R_{\theta'}(x)$ for all x.*

Proposition A.1.3. *Two parameter vectors θ and θ' are permutation-rescaling equivalent iff they are rescaling-permutation equivalent.*

Proof of Proposition A.1.1

Proposition. *Two rescaling equivalent parameterizations θ and θ' yield the same realization,* i.e. $R_G(\theta, x) = R_G(\theta', x)$ *for all* x.

Proof. We show by induction that

$$y^{(\ell)}(\theta', x) = y^{(\ell)}(\theta, x)D^{(\ell)} \tag{A.16}$$

for all $\ell \in [\![0, L]\!]$. We have $y^{(0)}(\theta', x) = x = xI_{N_0} = y^{(0)}(\theta, x)D^{(0)}$ and, if we assume Equation (A.16) valid for some $\ell \in [\![0, L-2]\!]$, then using Definition 2.2.2,

$$\begin{aligned}
y^{(\ell+1)}(\theta', x) &= \sigma\left(y^{(\ell)}(\theta', x)W'^{(\ell+1)} + b'^{(\ell+1)}\right) && \text{(A.17)}\\
&= \sigma\left(y^{(\ell)}(\theta, x)D^{(\ell)}\left(D^{(\ell)}\right)^{-1}W^{(\ell+1)}D^{(\ell+1)} + b^{(\ell+1)}D^{(\ell+1)}\right) && \text{(A.18)}\\
&= \sigma\left(y^{(\ell)}(\theta, x)W^{(\ell+1)} + b^{(\ell+1)}\right)D^{(\ell+1)} && \text{(A.19)}\\
&= y^{(\ell+1)}(\theta, x)D^{(\ell+1)}. && \text{(A.20)}
\end{aligned}$$

For the last layer, as $D^{(L)} = I_{N_L}$, $y^{(L)}(\theta', x) = y^{(L-1)}(\theta', x)W'^{(L)} + b'^{(L)}$ similarly yields $y^{(L)}(\theta', x) = y^{(L)}(\theta, x)$ thus $R_G(\theta) = R_G(\theta')$. □

Proof of Proposition A.1.2

Proposition. *Two permutation equivalent parameterizations θ and θ' yield the same realization,* i.e. $R_G(\theta, x) = R_G(\theta', x)$ *for all* x.

Proof. We show by induction that

$$y^{(\ell)}(\theta', x) = y^{(\ell)}(\theta, x)\pi^{(\ell)} \tag{A.21}$$

for all $\ell \in [\![0, L]\!]$. We have $y^{(0)}(\theta', x) = x = xI_{N_0} = y^{(0)}(\theta, x)\pi^{(0)}$ and, if we assume Equation (A.21) valid for some $\ell \in [\![0, L-2]\!]$, then by Definition 2.2.6,

$$\begin{aligned}
y^{(\ell+1)}(\theta', x) &= \sigma\left(y^{(\ell)}(\theta', x)W'^{(\ell+1)} + b'^{(\ell+1)}\right) && \text{(A.22)}\\
&= \sigma\left(y^{(\ell)}(\theta, x)\pi^{(\ell)}\left(\pi^{(\ell)}\right)^{-1}W^{(\ell+1)}\pi^{(\ell+1)} + b^{(\ell+1)}\pi^{(\ell+1)}\right) && \text{(A.23)}\\
&= \sigma\left(y^{(\ell)}(\theta, x)W^{(\ell+1)} + b^{(\ell+1)}\right)\pi^{(\ell+1)} && \text{(A.24)}\\
&= y^{(\ell+1)}(\theta, x)\pi^{(\ell+1)}. && \text{(A.25)}
\end{aligned}$$

For the last layer, as $\pi^{(L)} = I_{N_L}$, $y^{(L)}(\theta', x) = y^{(L-1)}(\theta', x)W'^{(L)} + b'^{(L)}$ similarly yields $y^{(L)}(\theta', x) = y^{(L)}(\theta, x)$ thus $R_G(\theta) = R_G(\theta')$. □

Proof of Proposition A.1.3

Proposition. *Two parameterizations θ and θ' are permutation-rescaling equivalent iff they are rescaling-permutation equivalent.*

Proof. Let θ and θ' be permutation-rescaling equivalent parameterizations, with no biases. There exists a parameterization θ'' such that $\theta \sim_P \theta''$ and $\theta'' \sim_S \theta'$. Thus, for every $\ell \in [\![1, L]\!]$,

$$W''^{(\ell)} = \left(\pi^{(\ell-1)}\right)^{-1} W^{(\ell)} \pi^{(\ell)} \tag{A.26}$$

$$W'^{(\ell)} = \left(D^{(\ell-1)}\right)^{-1} W''^{(\ell)} D^{(\ell)} \tag{A.27}$$

$$= \left(D^{(\ell-1)}\right)^{-1} \left(\pi^{(\ell-1)}\right)^{-1} W^{(\ell)} \pi^{(\ell)} D^{(\ell)} \tag{A.28}$$

$$= \left(\pi^{(\ell-1)}\right)^{-1} \left(\widetilde{D}^{(\ell-1)}\right)^{-1} W^{(\ell)} \widetilde{D}^{(\ell)} \pi^{(\ell)} \tag{A.29}$$

where, for every $\ell \in [\![0, L]\!]$,

$$\widetilde{D}^{(\ell)} = \pi^{(\ell)} D^{(\ell)} \left(\pi^{(\ell)}\right)^{-1} \tag{A.30}$$

We verify that $\widetilde{D}^{(\ell)}$ is a diagonal matrix with strictly positive entries, $D^{(\ell)} \in \mathcal{D}(|N_\ell|)$. We define

$$W'''^{(\ell)} = \left(\widetilde{D}^{(\ell-1)}\right)^{-1} W^{(\ell)} \widetilde{D}^{(\ell)} \tag{A.31}$$

Using Equation (A.29), we have

$$W'^{(\ell)} = \left(\pi^{(\ell-1)}\right)^{-1} W'''^{(\ell)} \pi^{(\ell)} \tag{A.32}$$

Thus, there exists a parameterization θ''' such that $\theta \sim_S \theta'''$ and $\theta''' \sim_P \theta'$, *i.e.* θ and θ' are rescaling-permutation equivalent. The reciprocal and the case with biases is similar and is not displayed here. □

Proof of Proposition 3.1.2

Proposition. *Two parameterizations that are permutation-rescaling equivalent are functionally equivalent.*

Proof. Let θ and θ' be permutation-rescaling equivalent parameterizations. There exists a parameterization θ'' such that $\theta \sim_P \theta''$ and $\theta'' \sim_S \theta'$. Using Propositions

A.1.2, we have $R_\theta = R_{\theta''}$ and $R_{\theta''} = R_{\theta'}$. Thus, $R_\theta = R_{\theta''}$. □

A.1.3 One Hidden Layer Case

Proof of Proposition 3.2.1

Proposition. *Let G be a one-hidden layer architecture valued with θ and θ'. Assume that θ and θ' are irreducible. Then, $R_\theta = R'_\theta$ implies that $\theta \sim_{PS} \theta'$.*

Proof. Let θ and θ' be two irreducible parameterizations such that $R_\theta = R_{\theta'}$. Recall that $x \mapsto R_\theta(x)$ and $x \mapsto R_{\theta'}(x)$ are functions from $\mathbb{R}^{N_0}$ to $\mathbb{R}^{N_2}$. Let us show that $\theta \sim_{PS} \theta'$. Since $R_\theta = R_{\theta'}$, for any output neuron η and $x \in \mathbb{R}^{N_0}$,

$$\underbrace{\sum_{\nu \in N_1} w_{\nu\to\eta}\sigma(\langle w_{\bullet\to\nu}, x\rangle + b_\nu) + b_\eta}_{\triangleq \varphi_\eta(x)} = \underbrace{\sum_{\nu \in N_1} w'_{\nu\to\eta}\sigma(\langle w'_{\bullet\to\nu}, x\rangle + b'_\nu) + b'_\eta}_{\triangleq \psi_\eta(x)}. \tag{A.33}$$

The proof follows four steps:

1. We reduce to the scalar case by taking $x = ut + r$, $t \in \mathbb{R}$ in Equation (A.33) for carefully chosen sets of directions $\mathcal{U} \subset \mathbb{R}^{N_0}$ and of offsets $\mathcal{R}_u \subset \mathbb{R}^{N_0}$ (note that $\mathcal{R}_u$ depends on $u \in \mathcal{U}$). We choose these sets such that, for all $u \in \mathcal{U}$ and $r \in \mathcal{R}_u$, and for any output neuron $\eta \in N_2$, the breakpoints of the continuous piecewise affine scalar function $t \mapsto \varphi_\eta(ut+r)$ are distinct, and similarly for ψ_η.

2. We fix one output neuron η and we fix $u \in \mathcal{U}, r \in \mathcal{R}_u$. We use the identifiability lemma A.1.1 on the functions $t \mapsto \varphi_\eta(ut + r)$ and $t \mapsto \psi_\eta(ut + r)$ (that depend on both η and u) to (partially) identify the weights and biases of θ and θ'.

3. We prove that the (partial) identification of the weights and biases of θ and θ' obtained in Step 2 is independent from u and r. It allows to define the rescalings and the permutation of the hidden neurons with respect to the output neuron η. Note that such rescalings and permutation still depend *a priori* on η.

4. We prove that the definitions of the rescalings and the permutation obtained in Step 2 are independent of the considered output neuron η, allowing to show that θ and θ' are indeed permutation and rescaling equivalent.

Step 1. We first construct $\mathcal{U}$ and then, for every $u \in \mathcal{U}$, we construct $\mathcal{R}_u$.

- Recall that none of the vectors $w_{\bullet\to\nu}$ and $w'_{\bullet\to\nu}$ equals zero for $\nu \in N_1$. Then, we define $\mathcal{U}_0$ as $\mathbb{R}^{N_0}$ minus a finite union of hyperplanes as follows:

$$\mathcal{U}_0 = \mathbb{R}^{N_0} \setminus \left(\bigcup_{\nu \in N_1} w_{\bullet\to\nu}^{\perp} \cup w'^{\perp}_{\bullet\to\nu} \right) \tag{A.34}$$

where $w_{\bullet\to\nu}^{\perp} = \{u \in \mathbb{R}^{N_0} \mid \langle u, w_{\bullet\to\nu} \rangle = 0\}$. In particular, for all $u \in \mathcal{U}_0$, we have

$$\forall \nu \in N_1, \langle w_{\bullet\to\nu}, u \rangle \neq 0, \tag{A.35}$$

$$\forall \nu \in N_1, \langle w'_{\bullet\to\nu}, u \rangle \neq 0. \tag{A.36}$$

Next, let $\eta \in N_2$ and define $\mathcal{I}_\eta = \{\nu \in N_1 \mid w_{\nu\to\eta} \neq 0\}$. Let $\mathcal{J} \subset \mathcal{I}_\eta$ and define

$$H_{\mathcal{J},\eta} = \left\{ u \in \mathbb{R}^{N_0} \middle| \left\langle u, \sum_{\nu \in \mathcal{J}} w_{\nu\to\eta} w_{\bullet\to\nu} \right\rangle = 0 \right\}. \tag{A.37}$$

Since θ is irreducible, there are no co-dependant hidden neurons (Definition 3.2.3), hence $\sum_{\nu\in\mathcal{J}} w_{\nu\to\eta} w_{\bullet\to\nu} \neq 0$, hence $H_{\mathcal{J},\eta}$ is a hyperplane of dimension $N_0 - 1$. Since there are finitely many sets $\mathcal{J}$, we define

$$\mathcal{U}_1 = \mathbb{R}^{N_0} \setminus \left(\bigcup_{\substack{\eta \in N_2 \\ \mathcal{J} \subset \mathcal{I}_\eta}} H_{\mathcal{J},\eta} \right). \tag{A.38}$$

Hence, for all $u \in \mathcal{U}_1$, we have

$$\forall \eta \in N_2, \forall \mathcal{J} \subset \mathcal{I}_\eta, \left\langle u, \sum_{\nu \in \mathcal{J}} w_{\nu\to\eta} w_{\bullet\to\nu} \right\rangle \neq 0. \tag{A.39}$$

We finally define $\mathcal{U} = \mathcal{U}_0 \cup \mathcal{U}_1$.

- We fix $u \in \mathcal{U}$ and construct $\mathcal{R}_u$ such that, for all $\nu_1 \neq \nu_2 \in N_1$ and $r \in \mathcal{R}_u$,

$$\frac{b_{\nu_1} + \langle w_{\bullet\to\nu_1}, r \rangle}{\langle w_{\bullet\to\nu_1}, u \rangle} \neq \frac{b_{\nu_2} + \langle w_{\bullet\to\nu_2}, r \rangle}{\langle w_{\bullet\to\nu_2}, u \rangle}, \tag{A.40}$$

$$\frac{b'_{\nu_1} + \langle w'_{\bullet\to\nu_1}, r \rangle}{\langle w'_{\bullet\to\nu_1}, u \rangle} \neq \frac{b'_{\nu_2} + \langle w'_{\bullet\to\nu_2}, r \rangle}{\langle w'_{\bullet\to\nu_2}, u \rangle}. \tag{A.41}$$

Equation (A.40) writes $\langle r, \alpha_{\nu_1,\nu_2}\rangle + \beta_{\nu_1,\nu_2} \neq 0$, where

$$\alpha_{\nu_1,\nu_2} = w_{\bullet\to\nu_1}\langle w_{\bullet\to\nu_2}, u\rangle - w_{\bullet\to\nu_2}\langle w_{\bullet\to\nu_1}, u\rangle \tag{A.42}$$

$$\beta_{\nu_1,\nu_2} = b_{\nu_1}\langle w_{\bullet\to\nu_2}, u\rangle - b_{\nu_2}\langle w_{\bullet\to\nu_1}, u\rangle. \tag{A.43}$$

Since θ is irreducible, there are no twin neurons (Definition 3.2.2). Define, for any $\nu_1 \neq \nu_2 \in N_1$, the set $H_{\nu_1,\nu_2} = \left\{r \in \mathbb{R}^{N_0} \mid \langle r, \alpha_{\nu_1,\nu_2}\rangle + \beta_{\nu_1,\nu_2} = 0\right\}$ which is either the empty set, a hyperplane, or the whole space $\mathbb{R}^{N_0}$. Let us show that H_{ν_1,ν_2} is not equal to $\mathbb{R}^{N_0}$. For the sake of contradiction, let us assume that $H_{\nu_1,\nu_2} = \mathbb{R}^{N_0}$. Then, both $\alpha_{\nu_1,\nu_2} = 0$ and $\beta_{\nu_1,\nu_2} = 0$. Using Equations (A.42) and (A.43), we deduce that[1] $w_{\bullet\to\nu_1} = dw_{\bullet\to\nu_2}$ and $b_{\nu_1} = db_{\nu_2}$ with $d = \langle w_{\bullet\to\nu_1}, u\rangle / \langle w_{\bullet\to\nu_2}, u\rangle \neq 0$, which is equivalent to $\Gamma_{\nu_1} = \Gamma_{\nu_2}$ in Definition 3.2.2, which is a contradiction since there are no hidden twin neurons in θ. Thus, we have shown that H_{ν_1,ν_2} is either an affine hyperplane, or the empty set. We similarly define H'_{ν_1,ν_2} and proceed to the same reasoning with θ'.

Finally, we define $\mathcal{R}_u$ as a $\mathbb{R}^{N_0}$ minus a finite union of hyperplanes as follows.

$$\mathcal{R}_u = \mathbb{R}^{N_0} \setminus \left(\bigcup_{\nu_1 \neq \nu_2 \in N_1} H_{\nu_1,\nu_2} \cup {H'}_{\nu_1,\nu_2} \right). \tag{A.44}$$

Note that by construction, for all $\nu_1 \neq \nu_2 \in N_1$ and $r \in \mathcal{R}_u$, Equations (A.40) and (A.41) are satisfied.

Step 2. We fix an output neuron $\eta \in N_2$ and define $\mathcal{I} = \{\nu \in N_1 \mid w_{\nu\to\eta} \neq 0\}$ and $\mathcal{I}' = \{\nu \in N_1 \mid w'_{\nu\to\eta} \neq 0\}$. Next, we fix $u \in \mathcal{U}$ and $r \in \mathcal{R}_u$. Using Equation (A.33), we have that for all $t \in \mathbb{R}$, $\varphi_\eta(ut + r) = \psi_\eta(ut + r)$. Then, for all $\nu \in N_1$, we define

$$s_\nu = \operatorname{sign}(\langle w_{\bullet\to\nu}, u\rangle) \tag{A.45}$$

$$a_\nu = w_{\nu\to\eta}\langle w_{\bullet\to\nu}, u\rangle s_\nu \tag{A.46}$$

$$t_\nu = -(b_\nu + \langle w_{\bullet\to\nu}, r\rangle)/\langle w_{\bullet\to\nu}, u\rangle \tag{A.47}$$

$$c = b_\eta \tag{A.48}$$

and we similarly define a'_ν, s'_ν, t'_ν and c'_ν. Note that these coefficients depend on u and r, although we omit to state the dependency explicitly for the sake of clarity. We verify that the following three conditions are met.

[1] Recall that $\langle w_{\bullet\to\nu_1}, u\rangle \neq 0$ and $\langle w_{\bullet\to\nu_2}, u\rangle \neq 0$ using Equation (A.35).

- For all $\nu \in \mathcal{I}$, $a_\nu \neq 0$. Indeed, by definition of $\mathcal{I}$, $w_{\nu\to\eta} \neq 0$ and by Equation (A.35), we have that $\langle w_{\bullet\to\nu}, u\rangle \neq 0$. Similarly, for all $\nu \in \mathcal{I}'$, we have $a'_\nu \neq 0$.

- For all $\nu_1 \neq \nu_2 \in \mathcal{I}$, $t_{\nu_1} \neq t_{\nu_2}$. This comes from Equations (A.40) and (A.41). Similarly, for all $\nu_1 \neq \nu_2 \in \mathcal{I}'$, $t'_{\nu_1} \neq t'_{\nu_2}$.

- For all $\mathcal{J} \subset \mathcal{I}$, $\sum_{\nu\in\mathcal{J}} a_\nu s_\nu \neq 0$. This comes from Equation (A.39) and from the fact that $a_\nu s_\nu = w_{\nu\to\eta}\langle w_{\bullet\to\nu}, u\rangle s_\nu^2 = w_{\nu\to\eta}\langle w_{\bullet\to\nu}, u\rangle$.

We are now ready to apply the identification Lemma A.1.1 stated below.

Lemma A.1.1. *Let $\mathcal{I}$, $\mathcal{I}'$ two finite sets of indices, empty or not*[2]*. For $i \in \mathcal{I}$, let $a_i \in \mathbb{R}^*$, $s_i \in \{-1, +1\}$, $t_i \in \mathbb{R}$ such that the t_i are distinct and $c \in \mathbb{R}$. Define φ as*

$$\varphi : t \mapsto \sum_{i\in\mathcal{I}} a_i \sigma(s_i(t - t_i)) + c. \tag{A.49}$$

Similarly, we define a'_i, s'_i, t'_i for $i \in \mathcal{I}'$, c' and the function ψ. We further assume that, for all $\mathcal{J} \subset \mathcal{I}$,

$$\sum_{i\in\mathcal{J}} a_i s_i \neq 0. \tag{A.50}$$

We assume $\forall t, \varphi(t) = \psi(t)$. Then, $c = c'$ and up to a re-numbering, $\mathcal{I} = \mathcal{I}'$ and for all $i \in \mathcal{I}$, $t_i = t'_i$, $s_i = s'_i$ and $a_i = a'_i$.

We get $b_\eta = b'_\eta$ and $\mathcal{I} = \mathcal{I}'$ up to a re-numbering. We distinguish between two cases. If $\mathcal{I} = \emptyset$ and $\mathcal{I}' = \emptyset$, we directly proceed to Step 3. Otherwise, $\mathcal{I} \neq \emptyset$ and $\mathcal{I}' \neq \emptyset$. We explicitly denote $\pi : \mathcal{I} \mapsto \mathcal{I}'$ the re-numbering of $\mathcal{I}'$. Then, for all $\nu \in \mathcal{I}$,

$$s'_{\pi(\nu)} = \operatorname{sign}(\langle w'_{\bullet\to\pi(\nu)}, u\rangle) = \operatorname{sign}(\langle w_{\bullet\to\nu}, u\rangle) = s_\nu \tag{A.51}$$

$$w'_{\pi(\nu)\to\eta}\langle w'_{\bullet\to\pi(\nu)}, u\rangle s'_{\pi(\nu)} = w_{\nu\to\eta}\langle w_{\bullet\to\nu}, u\rangle s_\nu \tag{A.52}$$

$$-(b'_{\pi(\nu)} + \langle w'_{\bullet\to\pi(\nu)}, r\rangle)/\langle w'_{\bullet\to\pi(\nu)}, u\rangle = -(b_\nu + \langle w_{\bullet\to\nu}, r\rangle)/\langle w_{\bullet\to\nu}, u\rangle \tag{A.53}$$

Then, using Equation (A.51) with Equation (A.52) and the fact that both s_ν and $s_{\pi(\nu)}$ are non-zero, we deduce that for all $\nu \in \mathcal{I}$,

$$w'_{\pi(\nu)\to\eta}\langle w'_{\bullet\to\pi(\nu)}, u\rangle = w_{\nu\to\eta}\langle w_{\bullet\to\nu}, u\rangle. \tag{A.54}$$

[2]We use the convention: $\sum_\emptyset = 0$.

Step 3. We still fix an output neuron $\eta \in N_2$. Let us show that π is independent from the considered vectors $u \in \mathcal{U}$ and $r \in \mathcal{R}_u$. To this end, we explicitly state the *a priori* dependency of π on u and r by writing $\pi = \pi_{u,r}$. Let us first fix $u \in \mathcal{U}$ and show that the vectors $w'_{\bullet\to\pi_{r,u}(\nu)}$ and $w_{\bullet\to\nu}$ are collinear for all $r \in \mathcal{R}_u$. Let $r \in \mathcal{R}_u$.

- First, since $\mathcal{R}_u$ is open, there exists a neighborhood V of r such that $V \subset \mathcal{R}_u$.

- Second, $\pi_{u,r}$ is entirely determined by the relation $t'_{\pi_{u,r}(\nu)} = t_\nu$ for all $\nu \in N_1$, hence by the relative ordering of the t_ν and t'_ν. Since the t_ν (resp. t'_ν) are distinct and since t_ν and t'_ν depend continuously on r, there exists a neighborhood V' of r such that, for all $r' \in V'$, the relative ordering of the t_ν and t'_ν is preserved, hence $\pi_{u,r'} = \pi_{u,r}$.

- We deduce from the first two points and from Equation (A.53) applied in r and to any $r' \neq r \in V \cap V'$ that

$$\langle w'_{\bullet\to\pi_{u,r}(\nu)}, \tau\rangle = d_{\nu\to\eta,u,r}\langle w_{\bullet\to\nu}, \tau\rangle \tag{A.55}$$

  where $\tau = r' - r$ and $d_{\nu\to\eta,u,r} = \langle w'_{\bullet\to\pi_{u,r}(\nu)}, u\rangle / \langle w_{\bullet\to\nu}, u\rangle \neq 0$. Since Equation (A.55) is valid for any small enough direction τ, we deduce that

$$w'_{\bullet\to\pi_{u,r}(\nu)} = d_{\nu\to\eta,u,r} w_{\bullet\to\nu}. \tag{A.56}$$

  which proves that $w'_{\bullet\to\pi_{u,r}(\nu)}$ and $w_{\bullet\to\nu}$ are collinear. Note that thanks to Equation (A.54), we also have

$$d_{\nu\to\eta,u,r} = \frac{w_{\nu\to\eta}}{w'_{\pi_{u,r}(\nu)\to\eta}}. \tag{A.57}$$

Next, using Equations (A.53) and (A.56), we deduce that

$$b'_{\pi_{u,r}(\nu)} = d_{\nu\to\eta,u,r} b_\nu. \tag{A.58}$$

Then, using Equations (A.56) and (A.58), for any $r' \neq r \in \mathcal{R}_u$ and for all $x \in \mathbb{R}^{N_0}$,

$$\langle x, w'_{\bullet\to\pi_{u,r'}(\nu)}\rangle + b'_{\pi_{u,r'}(\nu)} = d_{\nu\to\eta,u,r'}\left(\langle x, w_{\bullet\to\nu}\rangle + b_\nu\right) \tag{A.59}$$

$$= \frac{d_{\nu\to\eta,u,r'}}{d_{\nu\to\eta,u,r}}\left(\langle x, w'_{\bullet\to\pi_{u,r}(\nu)}\rangle + b'_{\pi_{u,r}(\nu)}\right). \tag{A.60}$$

Note that $d_{\nu\to\eta,u,r'}/d_{\nu\to\eta,u,r} \neq 0$. This is equivalent to $\Gamma_{\pi_{u,r'}(\nu)} = \Gamma_{\pi_{u,r}(\nu)}$ in Definition 3.2.2. Since there are no hidden twin neurons in θ', we have $\pi_{u,r'}(\nu) = \pi_{u,r}(\nu)$, hence $\pi_{u,r} = \pi_u$ does not depend on $r \in \mathcal{R}_u$. With a similar argument, we deduce that π_u does not depend on u either and we default back to the notation π.

Equation (A.56) implies that $d_{\nu\to\eta,u,r} = d_{\nu\to\eta}$ does not depend on u and r. Because both $w'_{\pi(\nu)\to\eta} \neq 0$ and $w_{\nu\to\eta} \neq 0$, we have $d_{\nu\to\eta} \neq 0$. Moreover, $d_{\nu\to\eta} > 0$ because Equations A.51 and A.54 imply $\operatorname{sign}(w_{\nu\to\eta}) = \operatorname{sign}(w'_{\pi(\nu)\to\eta})$. To summarize, for $\nu \in \mathcal{I}$, we have

$$w'_{\pi(\nu)\to\eta} = w_{\nu\to\eta}/d_{\nu\to\eta} \tag{A.61}$$

$$w'_{\bullet\to\pi(\nu)} = w_{\bullet\to\nu}d_{\nu\to\eta} \tag{A.62}$$

$$b'_{\pi(\nu)} = b_\nu d_{\nu\to\eta} \tag{A.63}$$

The first relation comes from Equation (A.57), the second from Equation (A.56) and the third from Equation (A.58).

Step 4. We now prove that θ and θ' are permutation-rescaling equivalent. We explicitly denote by $\mathcal{I}_\eta$ and $\mathcal{I}'_\eta$ the sets constructed in Step 2 and similarly for the re-numbering π_η to show the dependency on the output neuron η. Let us show that the definitions of the rescaling coefficients $d_{\nu\to\eta}$ and the permutations π_η in Step 2 are independent of the choice of η. Consider $\eta_1 \neq \eta_2$ two distinct output neurons.

- Regarding the permutation, we first show that if $\nu \in \mathcal{I}_{\eta_1} \cap \mathcal{I}_{\eta_2}$,

$$\pi_{\eta_1}(\nu) = \pi_{\eta_2}(\nu). \tag{A.64}$$

  Equations (A.62) and (A.63) show that, for all $x \in \mathbb{R}^{N_0}$,

$$\langle w'_{\bullet\to\pi_{\eta_1}(\nu)}, x\rangle + b'_{\pi_{\eta_1}(\nu)} = d_{\nu\to\eta_1}\left(\langle w_{\bullet\to\nu}, x\rangle + b_\nu\right) \tag{A.65}$$

$$= \frac{d_{\nu\to\eta_1}}{d_{\nu\to\eta_2}}\left(\langle w'_{\bullet\to\pi_{\eta_2}(\nu)}, x\rangle + b'_{\pi_{\eta_2}(\nu)}\right) \tag{A.66}$$

  Note that $d_{\nu\to\eta_1}/d_{\nu\to\eta_2} > 0$. Since θ' is irreducible, there are no twin hidden neurons (Definition 3.2.2). Hence, we deduce that $\pi_{\eta_1}(\nu) = \pi_{\eta_2}(\nu)$, which in turn implies that

$$d_{\nu\to\eta_1} = d_{\nu\to\eta_2}. \tag{A.67}$$

  Then, for all hidden neuron $\nu \in N_1$, there exists η such that $\nu \in \mathcal{I}_\eta$ since there are no dead neurons. We define $\pi(\nu) = \pi_\eta(\nu)$.

- Regarding the rescalings, Equation (A.67) implies that $d_{\nu\to\eta}$ does not depend on η in the sense that, if $\nu \in \mathcal{I}_{\eta_1} \cap \mathcal{I}_{\eta_2}$, $d_{\nu\to\eta_1} = d_{\nu\to\eta_2}$. Then, for all hidden neuron $\nu \in N_1$, there exists η such that $\nu \in \mathcal{I}_\eta$ since there are no dead neurons. We then define $d_\nu = d_{\nu\to\eta}$.

Finally, we have shown that θ and θ' are permutation-rescaling equivalent according to Definitions 2.2.3 and 2.2.7.

□

We finally prove Lemma A.1.1.

Lemma. *Let $\mathcal{I}$, $\mathcal{I}'$ two finite sets of indices, empty or not*[3]*. For $i \in \mathcal{I}$, let $a_i \in \mathbb{R}^*$, $s_i \in \{-1, +1\}$, $t_i \in \mathbb{R}$ such that the t_i are distinct and $c \in \mathbb{R}$. Define φ as*

$$\varphi : t \mapsto \sum_{i\in\mathcal{I}} a_i \sigma(s_i(t - t_i)) + c. \tag{A.68}$$

Similarly, we define a'_i, s'_i, t'_i for $i \in \mathcal{I}'$, c' and the function ψ. We further assume that, for all $\mathcal{J} \subset \mathcal{I}$,

$$\sum_{i\in\mathcal{J}} a_i s_i \neq 0. \tag{A.69}$$

We assume $\forall t, \varphi(t) = \psi(t)$. Then, $c = c'$ and up to a re-numbering, $\mathcal{I} = \mathcal{I}'$ and for all $i \in \mathcal{I}$, $t_i = t'_i$, $s_i = s'_i$ and $a_i = a'_i$.

Proof. We first show that $|\mathcal{I}| = |\mathcal{I}'|$. The function φ is piecewise affine and has exactly $|\mathcal{I}|$ distinct breakpoints because all the t_i are distinct and the s_i are non-zero. We denote this set of breakpoints $\mathcal{T} = \{t_i\}_{i\in\mathcal{I}}$. Similarly, the function ψ has exactly $|\mathcal{I}'|$ breakpoints. We denote this set of breakpoints $\mathcal{T}' = \{t'_i\}_{i\in\mathcal{I}'}$. Note that $\mathcal{T}$ may be empty since $\mathcal{I}$ may be empty, and similarly for $\mathcal{T}'$.

Then, because $\varphi = \psi$, $|\mathcal{I}| = |\mathcal{I}'|$ and $\mathcal{T} = \mathcal{T}'$, otherwise there would exist one point where one function would be differentiable and the other not. Thus, up to a re-numbering of $\mathcal{I}'$, we assume $\mathcal{I} = \mathcal{I}'$.

The re-ordering of the numbers t_i and t'_i and the fact that $\mathcal{T} = \mathcal{T}'$ implies that $\forall i \in \mathcal{I}, t_i = t'_i$. Let us show that $\forall i \in \mathcal{I}, a_i = a'_i$. We compute the derivative of φ in a neighbourhood Ω of t_i such that $\Omega \cap \mathcal{T} = \{t_i\}$, which is possible because the t_i are distinct. For $t \in \Omega\backslash\{t_i\}$,

$$\varphi'(t) = a_i s_i \mathbf{1}(s_i(t - t_i)) + f(t) \tag{A.70}$$

[3]We use the convention: $\sum_\emptyset = 0$.

where the function f is constant on Ω and where $\mathbf{1}(x) = 1$ if $x > 0$ and 0 otherwise. Similarly, for $t \in \Omega \backslash \{t_i\}$, $\psi'(t) = a'_i s'_i \mathbf{1}(s'_i(t - t'_i)) + g(t)$. Then, we use the equality $\varphi'(t_i^+) - \varphi'(t_i^-) = \psi'(t_i^+) - \psi'(t_i^-)$ to obtain

$$a_i s_i(\mathbf{1}(s_i) - \mathbf{1}(-s_i)) + f(t_i^+) - f(t_i^-) = a'_i s'_i(\mathbf{1}(s'_i) - \mathbf{1}(-s'_i)) + g(t_i^+) - g(t_i^-)$$

Using the facts that $f(t_i^+) = f(t_i^-)$ and $g(t_i^+) = g(t_i^-)$, this simplifies to

$$a_i s_i(\mathbf{1}(s_i) - \mathbf{1}(-s_i)) = a'_i s'_i(\mathbf{1}(s'_i) - \mathbf{1}(-s'_i)), \tag{A.71}$$

which finally yields $a_i s_i^2 = a'_i s'^2_i$. Since $s_i \in \{-1, +1\}$, we deduce that $s_i^2 = 1$, and similarly $s'^2_i = 1$. Hence, $a_i = a'_i$ and there exists $\varepsilon_i \in \{-1, +1\}$ such that $s'_i = \varepsilon_i s_i$. We then re-write the equality $\varphi(t) = \psi(t)$ as follows:

$$\sum_{i \in \mathcal{I}} a_i \Big[\sigma(s_i(t - t_i)) - \sigma(\varepsilon_i s_i(t - t_i)) \Big] + c - c' = 0. \tag{A.72}$$

Next, we observe that

$$\sigma(s_i(t - t_i)) - \sigma(\varepsilon_i s_i(t - t_i)) = \begin{cases} 0 & \text{if } \varepsilon_i = 1 \\ s_i(t - t_i) & \text{if } \varepsilon_i = -1 \end{cases} \tag{A.73}$$

Let us denote $\mathcal{J} = \{i \in \mathcal{I} \mid \varepsilon_i = -1\}$ and let us show that $\mathcal{J} = \emptyset$. Using Equation (A.73), we re-write Equation (A.72) as follows:

$$\sum_{i \in \mathcal{J}} a_i s_i(t - t_i) + c - c' = 0 \tag{A.74}$$

Since this is valid for all $t \in \mathbb{R}$ and since we assume that $\sum_{i \in \mathcal{J}} a_i s_i \neq 0$, we obtain that $\mathcal{J} = \emptyset$, hence $s'_i = s_i$ for all $i \in \mathcal{I}$. Finally, evaluating Equation (A.72) at any point t yields $c = c'$, which concludes the proof. □

A.2 Algebraic and Geometric tools

A.2.1 An Algebraic Expression of R_θ and its Consequences.

Proof of Proposition 3.3.1

Proposition. *For any parameterization $\theta = (w, b)$, any $x \in \mathbb{R}^{N_0}$ and any output neuron $\nu \in N_L$,*

$$R_\theta(x)_\nu = \sum_{\substack{p \in \mathcal{P} \\ p_L = \nu}} x_{p_0} v_p(\theta) a_p(\theta, x) + \sum_{\ell=1}^{L} \sum_{\substack{q \in \mathcal{Q}_\ell \\ q_\ell = \nu}} b_{q_0} v_q(\theta) a_q(\theta, x). \tag{A.75}$$

Proof. We prove the result by induction on the number of layers L.

- If $L = 0$, the input and output layers are the same and $R_\theta(x) = x$. On the other hand, $\mathcal{P} = \{(\mu)\}_{\mu \in N_0}$. Since there are no non-linearities, $a_p(\theta, x) = 1$ for all p, x. We also have $v_p(\theta) = 1$ for all p. It follows that

$$\sum_{\substack{p \in \mathcal{P} \\ p_L = \nu}} x_{p_0} v_p(\theta) a_p(\theta, x) = \sum_{\substack{p \in \mathcal{P} \\ p_L = \nu}} x_{p_0} = x_\nu \tag{A.76}$$

- If $L = 1$, there exists W and b such that $R_\theta(x) = xW + b$. On the other hand, $\mathcal{P} = \{(\mu, \nu)\}_{\mu \in N_0, \nu \in N_1}$ and $\mathcal{Q}_1 = \{(\nu)\}_{\nu \in N_1}$. Since there are no non-linearities, $a_q(\theta, x) = 1$ for all $q \in \mathcal{P} \cup \mathcal{Q}_{1 \to 1}$. We also have $v_p(\theta) = w_{\mu \to \nu}$ for all $p = (\mu, \nu)$. It follows that

$$\sum_{\substack{p \in \mathcal{P} \\ p_L = \nu}} x_{p_0} v_p(\theta) a_p(\theta, x) = \sum_{\mu \in N_0} x_{p_0} w_{\mu \to \nu} = (xW)_\nu \tag{A.77}$$

  and

$$\sum_{\ell=1}^{L} \sum_{\substack{q \in \mathcal{Q}_\ell \\ q_\ell = \nu}} b_{q_0} v_q(\theta) a_q(\theta, x) = \sum_{\substack{q \in \mathcal{Q}_1 \\ q_0 = \nu}} b_{q_0} = b_\nu \tag{A.78}$$

- If $L \geq 2$, for any $x \in \mathbb{R}^{N_0}$ and any $\nu \in N_L$,

$$R_\theta(x)_\nu = \sum_{\mu \in N_{L-1}} w_{\mu \to \nu} y_\mu(\theta, x) + b_\nu \tag{A.79}$$

$$= \sum_{\mu \in N_{L-1}} w_{\mu \to \nu} \sigma\left(\widetilde{R}_\theta(x)_\mu\right) + b_\nu \tag{A.80}$$

where $\widetilde{R}_\theta(x)$ denotes the output of layer $L-1$ of the network G valued with θ, evaluated *before* the non-linearity. Using the induction hypothesis, we get, for $\mu \in N_{L-1}$,

$$\widetilde{R}_\theta(x)_\mu = \sum_{\substack{\tilde{p}\in\widetilde{\mathcal{P}} \\ \tilde{p}_{L-1}=\mu}} x_{\tilde{p}_0} v_{\tilde{p}}(\theta) a_{\tilde{p}}(\theta, x) + \sum_{\ell=1}^{L-1} \sum_{\substack{\tilde{q}\in\widetilde{\mathcal{Q}}_\ell \\ \tilde{q}_{L-1}=\mu}} b_{\tilde{q}_0} v_{\tilde{q}}(\theta) a_{\tilde{q}}(\theta, x) \tag{A.81}$$

where $\widetilde{\mathcal{P}} = \{(p_0, \ldots, p_{L-1}) \mid p \in \mathcal{P}\}$ and $\widetilde{\mathcal{Q}}_\ell = \{(p_\ell, \ldots, p_{L-1}) \mid p \in \mathcal{P}\}$. We then observe that

$$\sigma\left(\widetilde{R}_\theta(x)_\mu\right) = \mathbf{1}(y_\mu(\theta, x) > 0)\widetilde{R}_\theta(x)_\mu. \tag{A.82}$$

We use both this fact and Equation (A.81) to rewrite Equation (A.80) as

$$\begin{aligned} R_\theta(x)_\nu &= \sum_{\mu\in N_{L-1}} w_{\mu\to\nu}\mathbf{1}(y_\mu(\theta, x) > 0)\widetilde{R}_\theta(x)_\mu + b_\nu & \text{(A.83)} \\ &= \sum_{\mu\in N_{L-1}} \sum_{\substack{\tilde{p}\in\widetilde{\mathcal{P}} \\ \tilde{p}_{L-1}=\mu}} x_{\tilde{p}_0} w_{\mu\to\nu} v_{\tilde{p}}(\theta)\mathbf{1}(y_\mu(\theta, x) > 0) a_{\tilde{p}}(\theta, x) \\ &+ \sum_{\mu\in N_{L-1}} \sum_{\ell=1}^{L-1} \sum_{\substack{\tilde{q}\in\widetilde{\mathcal{Q}}_\ell \\ \tilde{q}_{L-1}=\mu}} b_{\tilde{q}_0} w_{\mu\to\nu} v_{\tilde{q}}(\theta)\mathbf{1}(y_\mu(\theta, x) > 0) a_{\tilde{q}}(\theta, x) \\ &+ b_\nu & \text{(A.84)} \end{aligned}$$

We next simplify Equation (A.84) using the following remarks. Recall that we denote path concatenation with $+$. Let $q = \tilde{q} + \mu$ and $p = \tilde{p} + \mu$. First, $w_{\mu\to\nu} v_{\tilde{p}}(\theta) = v_p(\theta)$ by definition of $v_p(\theta)$ and similarly for q. Second, $\mathbf{1}(y_\mu(\theta, x) > 0) a_{\tilde{p}}(\theta, x) = a_p(\theta, x)$ by definition of $a_p(\theta, x)$ and similarly for q. Finally, $\{\tilde{p} + (\nu) \mid \tilde{p} \in \widetilde{\mathcal{P}} \text{ and } \tilde{p}_{L-1} = \mu\}_{\mu\in N_{L-1}} = \{p \in \mathcal{P} \mid p_L = \nu\}$ and similarly, $\{\tilde{q} + (\nu) \mid \tilde{q} \in \widetilde{\mathcal{Q}}_\ell \text{ and } \tilde{q}_{L-1} = \mu\}_{\mu\in N_{L-1}} = \{q \in \mathcal{Q}_{\ell\to L} \mid q_L = \nu\}$. Thus,

$$\begin{aligned} R_\theta(x)_\nu &= \sum_{\substack{p\in\mathcal{P} \\ p_L=\nu}} x_{p_0} v_p(\theta) a_p(\theta, x) + \sum_{\ell=1}^{L-1} \sum_{\substack{q\in\mathcal{Q}_{\ell\to L} \\ q_L=\nu}} b_{q_0} v_q(\theta) a_q(\theta, x) + b_\nu \\ &= \sum_{\substack{p\in\mathcal{P} \\ p_L=\nu}} x_{p_0} v_p(\theta) a_p(\theta, x) + \sum_{\ell=1}^{L} \sum_{\substack{q\in\mathcal{Q}_{\ell\to L} \\ q_L=\nu}} b_{q_0} v_q(\theta) a_q(\theta, x) \end{aligned}$$

□

Proof of Proposition 3.3.2

Proposition. *Let $\mathcal{U}(x,\theta)$ be the set of all neighborhoods $U \subset \mathbb{R}^{N_0} \times \mathbb{R}^{E\cup H}$ of (x,θ). We define, for any parameterization θ,*

$$\mathcal{X}_\theta \triangleq \{x \in \mathbb{R}^{N_0} \mid \exists U \in \mathcal{U}(x,\theta), \forall q \in \mathcal{P}\cup\mathcal{Q}, \forall (x',\theta') \in U, a_p(\theta,x) = a_p(\theta',x')\}. \tag{A.85}$$

Then, $\mathcal{X}_\theta$ is open. Moreover, let θ be a parameterization and $x \in \mathcal{X}_\theta$. Then, for each full or partial path $p \in \mathcal{P} \cup \mathcal{Q}$, the function $x' \mapsto a_p(\theta,x')$ (resp. $\theta' \mapsto a_p(\theta',x)$) is locally constant in the neighborhood of x (resp. in the neighborhood of θ).

Proof. To show that $\mathcal{X}_\theta$ is open, consider $x \in \mathcal{X}_\theta$. By definition there exists $U \in \mathcal{U}(x,\theta)$ such that, for all $p \in \mathcal{P}\cup\mathcal{Q}$ and for all $(x',\theta') \in U$, $a_p(\theta,x) = a_p(\theta',x')$. Since U is a neighborhood of (x,θ), there exists two open neighborhoods V, W of x (resp. of θ) such that $V \times W \subset U$. Considering $x' \in V$ we now show that $x' \in \mathcal{X}_\theta$. This will imply that $\mathcal{X}_\theta$ is open as claimed. First, $(x',\theta) \in U$, hence $a_p(\theta,x) = a_p(\theta,x')$. Second, by definition of $x \in \mathcal{X}_\theta$, we have $a_p(\theta,x) = a_p(\theta'',x'')$ for all $\theta'',x'' \in U$. Hence, noting that $U \in \mathcal{U}(x',\theta)$ since V is a neighborhood of x', we deduce that for all $\theta'',x'' \in U$, we have $a_p(\theta,x') = a_p(\theta'',x'')$.

Next, let us show that the function $\theta' \mapsto a_p(\theta',x)$ is locally constant. Let $x \in \mathcal{X}_\theta$. Then, there exists $U \in \mathcal{U}(x,\theta)$ such that, for all $p \in \mathcal{P}\cup\mathcal{Q}$ and for all $(x',\theta') \in U, a_p(\theta,x) = a_p(\theta',x')$. As before, there are open neighborhoods V,W such that $V\times W \subset U$ where $V \subset R^{N_0}$ is a neighborhood of x and $W \subset R^{E\cup H}$ is a neighborhood of θ. Since $x \in V$, we have that for all $\theta' \in W$, $a_p(\theta,x) = a_p(\theta',x)$. Thus, the function $\theta \mapsto a_p(\theta,x)$ is locally constant. The proof that the function $x \mapsto a_p(\theta,x)$ is locally constant is similar. □

Proof of Proposition 3.3.3

Proposition. *The function $x \mapsto R_\theta(x)$ is piecewise affine continuous and locally affine in the neighborhood of any $x \in X_\theta$.*

Proof. The piecewise affine nature comes from Proposition 3.3.1, where we have

$$\begin{aligned} R_\theta(x)_\nu &= \sum_{\substack{p\in\mathcal{P}\\ p_L=\nu}} x_{p_0} v_p(\theta) a_p(\theta,x) + \sum_{\ell=1}^{L} \sum_{\substack{q\in\mathcal{Q}_{\ell\to L}\\ q_L=\nu}} b_{q_0} v_q(\theta) a_q(\theta,x) \\ &= \sum_{\mu\in N_0} x_\mu A_\mu(x) + B(x) \end{aligned}$$

with the following coefficients

$$A_\mu(x) = \sum_{\substack{p\in\mathcal{P}\\ p_0=\mu\\ p_L=\nu}} v_p(\theta)a_p(\theta,x)$$
$$B(x) = \sum_{\ell=1}^{L}\sum_{\substack{q\in\mathcal{Q}_{\ell\to L}\\ q_L=\nu}} b_{q_0}v_q(\theta)a_q(\theta,x).$$

We notice that $x \mapsto A_\mu(x)$ and $x \mapsto B(x)$ are piecewise constant using Proposition 3.3.2, hence the result. The continuity comes from Equation 2.9 which shows that $x \mapsto R_\theta(x)$ is the composition of continuous functions. □

Proof of Proposition 3.3.4

Proposition. *For any $x \in \mathcal{X}_\theta$, the function $\theta \mapsto R_\theta(x)$ is piecewise polynomial continuous and locally polynomial in the neighborhood of any θ.*

Proof. The piecewise polynomial nature comes from Proposition 3.3.1, where we have

$$\begin{aligned}R_\theta(x)_\nu &= \sum_{\substack{p\in\mathcal{P}\\ p_L=\nu}} x_{p_0}v_p(\theta)a_p(\theta,x) + \sum_{\ell=1}^{L}\sum_{\substack{q\in\mathcal{Q}_{\ell\to L}\\ q_L=\nu}} b_{q_0}v_q(\theta)a_q(\theta,x)\\ &= \sum_{\substack{p\in\mathcal{P}\\ p_L=\nu}} v_p(\theta)A_p(\theta) + \sum_{\ell=1}^{L}\sum_{\substack{q\in\mathcal{Q}_{\ell\to L}\\ q_L=\nu}} v_q(\theta)B_q(\theta)\end{aligned}$$

with the following coefficients

$$A_p(\theta) = x_{p_0}a_p(\theta,x)$$
$$B(x) = b_{q_0}a_q(\theta,x).$$

We notice that $x \mapsto A_\mu(x)$ and $x \mapsto B(x)$ are piecewise constant using Proposition 3.3.2, hence the result. The continuity comes from Equation 2.9 which shows that $\theta \mapsto R_\theta(x)$ is the composition of continuous functions. □

Proof of Proposition A.2.1

Proposition A.2.1. *Let $e \in E$. Then, for all $x \in \mathcal{X}_\theta$,*

$$w_e\left(\frac{\partial R_\theta(x)}{\partial w_e}\right)_\nu = \sum_{\substack{p\in\mathcal{P}\\ p_L=\nu\\ e\in p}} x_{p_0}v_p(\theta)a_p(\theta,x) + \sum_{\ell=1}^{L}\sum_{\substack{q\in\mathcal{Q}_\ell\\ q_{L-\ell}=\nu\\ e\in q}} b_{q_0}v_q(\theta)a_q(\theta,x). \tag{A.86}$$

Proof. Let θ be a parameterization and $x \in \mathcal{X}_\theta$. Using Proposition 3.3.2, we deduce that the functions $\theta' \mapsto a_p(\theta', x)$ are locally constant in the neighborhood of θ for any full or partial path $p \in \mathcal{P} \cup \mathcal{Q}$. Hence,

$$\frac{\partial a_p(\theta, x)}{\partial w_e} = 0 \tag{A.87}$$

Moreover, recalling the definition of a path value in the Notations,

$$w_e \frac{v_p(\theta)}{\partial w_e} = \begin{cases} 0 & \text{if } e \notin p \\ v_p(\theta) & \text{if } e \in p. \end{cases} \tag{A.88}$$

We use Proposition 3.3.1 to conclude. □

A.2.2 Trajectories in the Parameter Space

Proof of Proposition 3.3.6

Proposition. *We define the linear map* $\mathbf{S}\colon \ker \mathbf{P} \to \mathbb{R}^H$ *such that*

$$(\mathbf{S}\beta)_\nu \triangleq -\sum_{e \in q} \beta_e \tag{A.89}$$

where q *is the partial path going from* ν *to the output neuron* η*. Then,*

1. $\mathbf{S}$ *is well-defined (i.e., independent of the choice of* q*);*
2. $\mathbf{S}$ *is an isomorphism and its inverse* $\mathbf{S}^{-1} : \mathbb{R}^H \to \ker \mathbf{P}$ *is such that for any* $\lambda \in \mathbb{R}^H$ *we have* $\mathbf{S}^{-1}\lambda = \beta$ *where for any edge* $e = \mu \to \nu \in E$*,*

$$\beta_e \triangleq \begin{cases} -\lambda_\mu & \text{if } \mu \in N_{L-1} \text{ (and } \nu = \eta) \\ \lambda_\nu - \lambda_\mu & \text{if } \mu \in N_\ell,\ 1 \leq \ell \leq L-2 \\ \lambda_\nu & \text{if } \mu \in N_0. \end{cases} \tag{A.90}$$

Proof. Note that $\mathbf{S}$ is linear. The proof follows three steps.

S is well-defined. Let $\alpha \in \ker \mathbf{P}$. We wish to show that if q and q' are two partial paths going from ν to the output neuron η, then

$$\sum_{e \in q} \alpha_e = \sum_{e \in q'} \alpha_e.$$

Let $\overline{q}$ be a path going from some input neuron to ν and define $p = \overline{q}+q$ and $p' = \overline{q}+q'$ (recall that + denotes path concatenation). As $\alpha \in \ker \mathbf{P}$, we have

$$\sum_{e\in\overline{q}} \alpha_e + \sum_{e\in q} \alpha_e = \sum_{e\in p} \alpha_e = (\mathbf{P}\alpha)_p = 0 = (\mathbf{P}\alpha)_{p'} = \sum_{e\in\overline{q}} \alpha_e + \sum_{e\in q'} \alpha_e.$$

Thus, $\sum_{e\in q} \alpha_e = \sum_{e\in q'} \alpha_e$ and $\mathbf{S}$ is well-defined.

S is injective. Let $\alpha \in \ker \mathbf{P}$ such that $\mathbf{S}\alpha = 0$. Considering any edge $e = \mu \to \nu \in E$ we show that $\alpha_e = 0$, hence $\alpha = 0$. Let $\overline{q}$ be a partial path going from ν to the output neuron and $q = \{\mu\} + \overline{q}$. Then,

$$0 = (\mathbf{S}\alpha)_\nu - (\mathbf{S}\alpha)_\mu = -\sum_{e'\in\overline{q}} \alpha_{e'} + \sum_{e'\in q} \alpha_{e'} = \alpha_e.$$

S is surjective. Let $\gamma \in \mathbb{R}^H$ and $\alpha \in \mathbb{R}^E$ be defined for any edge e as in (A.90). Let $q = (q_0, \dots, q_L)$ be a partial path going from some neuron $\mu = q_0 \in N_0 \cup H$ to the output neuron $\eta = q_k \in N_\ell$. If $\mu \in N_0$ then

$$\sum_{e\in q} \alpha_e = -\gamma_{q_{k-1}} + \sum_{j=2}^{k-1} (\gamma_{q_j} - \gamma_{q_{j-1}}) + \gamma_{q_1} = 0$$

hence $\alpha \in \ker \mathbf{P}$. If $\nu \in H$ then similarly

$$\sum_{e\in q} \alpha_e = -\gamma_{q_{k-1}} + \sum_{j=1}^{k-1} (\gamma_{q_j} - \gamma_{q_{j-1}}) = \gamma_{q_0} = \gamma_\mu$$

hence $(\mathbf{S}\alpha)_\mu = \gamma_\mu$. This shows that $(\mathbf{S}\alpha) = \gamma$. □

Proof of Proposition A.2.2

Proposition A.2.2. *Consider $\theta = (w, b) \in \mathbb{R}^E \times \mathbb{R}^H$ a parameterization and define*

$$V_\theta \triangleq \mathrm{Supp}_w \times \mathbf{S}^{-1}(\mathrm{Supp}_b) \subset \mathbb{R}^E \times \ker \mathbf{P} \subset \mathbb{R}^E \times \mathbb{R}^E.$$

For any $(\alpha, \beta) \in V_\theta$, we write $\gamma = (\alpha, \mathbf{S}\beta) \in \mathrm{Supp}_w \times \mathrm{Supp}_b = \mathrm{Supp}_\theta$ and

$$\varphi_\theta(\alpha, \beta) \triangleq \theta \odot e^\gamma = (w \odot \exp(\alpha), b \odot \exp(\mathbf{S}\beta))$$

where $\odot$ denotes pointwise multiplication. Then,

1. *The function $\varphi_\theta \colon V_\theta \to \mathrm{Sign}_\theta$ is a $\mathcal{C}^\infty$-diffeomorphism.*

2. *For $\theta' \in \mathrm{Sign}_\theta$ we have $\varphi_\theta^{-1}(\theta') = (\alpha, \beta) \in V_\theta$ where for any edge $e \in E$ and hidden neuron $\nu \in H$*

$$\alpha_e \triangleq \begin{cases} \log\left(\frac{w'_e}{w_e}\right) & \text{if } w_e \neq 0 \\ 0 & \text{if } w_e = 0. \end{cases} \qquad (\mathbf{S}\beta)_\nu \triangleq \begin{cases} \log\left(\frac{b'_\nu}{b_\nu}\right) & \text{if } b_\nu \neq 0 \\ 0 & \text{if } b_\nu = 0. \end{cases} \tag{A.91}$$

Proof. Since $\mathrm{sign}(\varphi_\theta(\alpha, \beta)) = \mathrm{sign}(\theta)$ for any (α, β), we have $\varphi_\theta(\alpha, \beta) \in \mathrm{Sign}_\theta$. Let us show that φ_θ is both surjective and injective with the given expression of φ_θ^{-1}. The fact that φ_θ and φ_θ^{-1} are $\mathcal{C}^\infty$ is clear from their expressions.

φ_θ is injective. Consider $(\alpha, \beta), (\tilde{\alpha}, \tilde{\beta}) \in V_\theta$ and assume that $\varphi_\theta(\alpha, \beta) = \varphi_\theta(\tilde{\alpha}, \tilde{\beta})$.

Consider $e \in E$. If $w_e = 0$ then, since $\mathrm{supp}(\alpha) \subset \mathrm{supp}(w)$ and $\mathrm{supp}(\tilde{\alpha}) \subset \mathrm{supp}(w)$, we get $\alpha_e = \tilde{\alpha}_e = 0$; if $w_e \neq 0$ then $w_e \exp(\alpha_e) = \varphi_\theta(\alpha, \beta)_e = \varphi_\theta(\tilde{\alpha}, \tilde{\beta})_e = w_e \exp(\tilde{\alpha}_e)$ implies $\alpha_e = \tilde{\alpha}_e$. Overall we obtain $\alpha = \tilde{\alpha}$.

Consider now $\nu \in H$. As $\beta, \tilde{\beta} \in \mathbf{S}^{-1}(\mathrm{Supp}_b)$, we have $\mathbf{S}\beta, \mathbf{S}\tilde{\beta} \in \mathrm{Supp}_b$. When $b_\nu = 0$ this implies $(\mathbf{S}\beta)_\nu = 0 = (\mathbf{S}\tilde{\beta})_\nu$; when $b_\nu \neq 0$, as $b_\nu \exp((\mathbf{S}\beta)_\nu) = \varphi_\theta(\alpha, \beta)_\nu = \varphi_\theta(\tilde{\alpha}, \tilde{\beta})_\nu = b_\nu \exp((\mathbf{S}\tilde{\beta})_\nu)$ we get $(\mathbf{S}\beta)_\nu = (\mathbf{S}\tilde{\beta})_\nu$; in both cases we obtain $\mathbf{S}\beta = \mathbf{S}\tilde{\beta}$. Since $\mathbf{S}$ is an isomorphism, we deduce that $\beta = \tilde{\beta}$.

φ_θ is surjective. Let $\theta' = (w', b') \in \mathrm{Sign}_\theta$. Let us show that Equations (A.91) define $(\alpha, \beta = \mathbf{S}^{-1}(\gamma)) \in V_\theta$ such that $\theta' = \varphi_\theta(\alpha, \beta)$.

Since $\theta' \in \mathrm{Sign}_\theta$, w'_e and w_e have the same sign for every edge $e \in E$. In particular, when $w_e \neq 0$ we get $w'_e / w_e > 0$, hence α_e is well defined, and $\alpha \in \mathrm{Supp}_w$.

Similarly, γ_ν is well-defined for every hidden neuron $\nu \in H$, and $\gamma \in \mathrm{Supp}_b$, hence $\beta = \mathbf{S}^{-1}\gamma$ satisfies $\beta \in \mathbf{S}^{-1}(\mathrm{Supp}_b)$. We observe that $\theta' = \varphi_\theta(\alpha, \beta)$ to conclude. □

Proof of Proposition 3.3.7

Proposition. *Consider θ a parametrization and $B_\theta \triangleq \{\gamma \in \mathrm{Supp}_\theta \mid \|\gamma\|_2 \leq 1\}$. Then, for each $x \in \mathcal{X}_\theta$, there exists $\varepsilon > 0$ such that, for every $(\gamma, t) \in B_\theta \times]-\varepsilon, \varepsilon[$, the following result holds:*

$$R_\theta(\theta \odot e^{\gamma t}, x) = \sum_{p \in \mathcal{P}} x_{p_0} v_p(\theta) e^{t\mathbf{P}(\alpha)_p} a_p(\theta, x) + \sum_{q \in \mathcal{Q}} b_{q_0} v_q(\theta) e^{t\mathbf{Q}(\alpha-\beta)_q} a_q(\theta, x). \tag{A.92}$$

Proof. The proof follows two steps.

Neighborhood. Let $x \in X_\theta$. Let us show that there exists a neighborhood Ω of $t = 0$ such that, for every full or partial path $q \in \mathcal{P} \cup \mathcal{Q}$ and for every $(\alpha, \beta, t) \in B_\theta \times \Omega$, we have $a_q(\theta \odot e^{\gamma t}, x) = a_q(\theta, x)$.

Let $q \in \mathcal{P} \cup \mathcal{Q}$ be a full path or partial path. First we show that there exists a neighborhood Ω_q of $t = 0$ such that, for every $(\alpha, \beta, t) \in B_\theta \times \Omega_q$, $a_q(\theta \odot e^{\gamma t}, x) = a_q(\theta, x)$. We define the following continuous function:

$$\begin{aligned} f_\theta \colon V_\theta \times \mathbb{R} &\longrightarrow \mathbb{R}^{E \cup H} \\ (\alpha, \beta, t) &\mapsto \theta \odot e^{\gamma t}. \end{aligned}$$

Denote, for any set $Z \in \mathbb{R}^{E \cup H}$, $f_\theta^{-1}(Z) = \{(\alpha, \beta, t) \in V_\theta \times \mathbb{R} \mid f_\theta(\alpha, \beta, t) \in Z\}$. By Proposition 3.3.2, there exists an open neighborhood[4] $Z \subset \mathbb{R}^{E \cup H}$ of θ such that, for all $\theta' \in Z$, $a_q(\theta', x) = a_q(\theta, x)$. Define

$$U \triangleq f_\theta^{-1}(Z \cap \mathrm{Sign}_\theta) \subset V_\theta \times \mathbb{R}. \tag{A.93}$$

Since f_θ is continuous and since Z is an open set in Sign_θ, U is an open set in $V_\theta \times \mathbb{R}$. Moreover, U is a neighborhood of $(0, 0, 0) \in V_\theta \times \mathbb{R}$ since $f_\theta(0, 0, 0) = \theta \in Z \cap \mathrm{Sign}_\theta$. Define $B_\theta(\varepsilon) \triangleq \{\gamma \in \mathrm{Supp}_\theta \mid \|\gamma\|_2 \leq \varepsilon\}$. Hence, there exists $\varepsilon_q > 0$ and a neighborhood $\widetilde{\Omega}_q \subset \mathbb{R}$ of $t = 0$ such that

$$B_\theta(\varepsilon_q) \times \widetilde{\Omega}_q \subset U. \tag{A.94}$$

We observe that, for any $\varepsilon > 0$, $f(\cdot, \cdot, t) = f(\cdot/\varepsilon, \cdot/\varepsilon, \varepsilon t)$. Hence, defining the rescaled neighborhood $\Omega_q = \varepsilon_q \widetilde{\Omega}_q$, we have

$$f_\theta(B_\theta(1) \times \Omega_q) = f_\theta(B_\theta(\varepsilon_q) \times \widetilde{\Omega}_q) \subset f_\theta(U) = Z \cap \mathrm{Sign}_\theta \,. \tag{A.95}$$

Thus, for every $(\alpha, \beta, t) \in B_\theta \times \Omega_q$, we have $\theta \odot e^{\gamma t} \in Z \cap \mathrm{Sign}_\theta$ and therefore $a_q(\theta \odot e^{\gamma}, x) = a_q(\theta, x)$. We finally define $\Omega = \cap_q \Omega_q$. Since there are finitely many partial paths, Ω is also a neighborhood of $t = 0$.

Computation. Let $x \in X_\theta$ and $(\alpha, \beta, t) \in \Omega$. We note $\theta' = \theta \odot e^{\gamma t}$. For any path or partial path $q \in \mathcal{P} \cup \mathcal{Q}$, the following holds:

- Using the first proof step, $a_q(\theta', x) = a_q(\theta, x)$.

[4] Recall that we restrict ourselves to a scalar output with no output biases, see Subsection 3.3.2.

- The expression of $v_q(\theta')$ yields $v_q(\theta') = v_q(\theta)\exp(t\sum_{e\in q}\alpha_e)$.
- By definition of $\mathbf{S}$ (Proposition 3.3.6), the biases associated with the parameterization θ' are equal to $b'_\nu = b_\nu \exp(t\mathbf{S}(\beta)_\nu)$ for every hidden neuron $\nu \in H$.

We explicitly compute $\theta \odot e^{\gamma t}$ as follows. Using Proposition 3.3.1, we have

$$\theta \odot e^{\gamma t} = \underbrace{\sum_{p\in\mathcal{P}} x_{p_0} v_p(\theta') a_p(\theta', x)}_{\rho_1(t)} + \underbrace{\sum_{q\in\mathcal{Q}} b'_{q_0} v_q(\theta') a_q(\theta', x)}_{\rho_2(t)}.$$

Using the above observations and Definition 3.3.1, the first term writes

$$\begin{aligned}\rho_1(t) &= \sum_{p\in\mathcal{P}} x_{p_0} v_p(\theta) \exp\left(t\sum_{e\in p}\alpha_e\right) a_p(\theta, x)\\ &= \sum_{p\in\mathcal{P}} x_{p_0} v_p(\theta) \exp\left(t(\mathbf{P}\alpha)_p\right) a_p(\theta, x).\end{aligned}$$

Similarly, the second term writes

$$\rho_2(t) = \sum_{q\in\mathcal{Q}} b_{q_0} \exp\left(t\mathbf{S}(\beta)_{q_0}\right) v_q(\theta) \exp\left(t\sum_{e\in q}\alpha_e\right) a_q(\theta, x).$$

In particular, as the definition of $(\mathbf{S}\beta)_{q_0}$ is independent of the choice of the partial path going from q_0 to the output neuron (see Proposition 3.3.6), we have

$$\begin{aligned}\rho_2(t) &= \sum_{q\in\mathcal{Q}} b_{q_0} \exp\left(-t\sum_{e\in q}\beta_e\right) v_q(\theta) \exp\left(t\sum_{e\in q}\alpha_e\right) a_q(\theta, x)\\ &= \sum_{q\in\mathcal{Q}} b_{q_0} \exp\left(-t\mathbf{Q}(\beta)_q\right) v_q(\theta) \exp\left(t\mathbf{Q}(\alpha)_q\right) a_q(\theta, x)\\ &= \sum_{q\in\mathcal{Q}} b_{q_0} v_q(\theta) \exp\left(t\mathbf{Q}(\alpha-\beta)_q\right) a_q(\theta, x).\end{aligned}$$

□

A.2.3 Algebraic Characterization of Rescaling Equivalence

Proof of Proposition 3.3.8

Proposition. *$\mathrm{Sincomp}_\theta$ is compact and $\mathrm{Sincomp}_\theta \subset \overline{\mathrm{Comp}_\theta}$ for any admissible parameterization θ.*

Proof. We prove both results successively.

$\mathbf{Sincomp}_\theta \subset \overline{\mathbf{Comp}}_\theta$. Assume that $\gamma = (\alpha, \mathbf{S}\beta) \in \mathrm{Sincomp}_\theta$. We distinguish between two cases.

- If $b = 0$, then $\beta = 0$ since $(\alpha, \beta) \in V_\theta$, hence $\beta \in \mathrm{Supp}_b = \{0\}$ and $\mathbf{S}\beta = 0$. The condition $\|\gamma\|_2 = 1$ therefore ensures that $\|(\alpha, 0)\|_2 = 1$ and $\alpha \neq 0$. Together with the condition $\alpha \in (\ker(\mathbf{P}_\theta))^\perp$, this implies that $\alpha \notin \ker(\mathbf{P}_\theta)$.

- If $b \neq 0$, we use $\alpha \in (\ker(\mathbf{P}_\theta))^\perp$ and distinguish again between two cases.

  - If $\alpha \neq 0$, we deduce that $\alpha \notin (\ker(\mathbf{P}_\theta))$.
  - If $\alpha = 0$, let us show that $\alpha - \beta \notin \ker(\mathbf{Q}_\theta)$. For the sake of contradiction, we assume that $\alpha - \beta = -\beta \in \ker(\mathbf{Q}_\theta)$, *i.e.* for all $q \in \mathcal{Q}$,

$$b_{q_0} v_q(\theta) \sum_{e \in q} \beta_e = 0 \tag{A.96}$$

    Recalling that $\beta \in \mathbf{S}^{-1}(\mathrm{Supp}_b) \subseteq \ker \mathbf{P}$ and recalling Proposition 3.3.6, we deduce that, for all $q \in \mathcal{Q}$,

$$b_{q_0} v_q(\theta)(\mathbf{S}\beta)_{q_0} = 0. \tag{A.97}$$

    Let us show that $(\mathbf{S}\beta)_\nu = 0$ for every hidden neuron $\nu \in H$. If $b_\nu = 0$, then $(\mathbf{S}\beta)_\nu = 0$ since $\mathbf{S}\beta \in \mathrm{Supp}_b$. If $b_\nu \neq 0$, since θ is admissible, there exists a path segment q going from ν to the output neuron such that $v_q(\theta) \neq 0$. Thus, since $b_\nu v_q(\theta)(\mathbf{S}\beta)_\nu = b_{q_0} v_q(\theta)(\mathbf{S}\beta)_{q_0} = 0$, we obtain $(\mathbf{S}\beta)_\nu = 0$ again.

  To summarize, we have $\mathbf{S}\beta = 0$. Since $\mathbf{S}$ is an isomorphism, we deduce that $\beta = 0$. Since $\alpha = 0$, this contradicts the condition $\|\gamma\|_2 = 1$. □

$\mathbf{Sincomp}_\theta$ is compact. By definition, $\mathrm{Sincomp}_\theta$ is a bounded subset of V_θ. Let us show that it is closed. Let $\gamma_n = (\alpha_n, \mathbf{S}\beta_n) \in \mathrm{Sincomp}_\theta$ be a sequence that converges to some $\gamma = (\alpha, \mathbf{S}\beta) \in \mathrm{Supp}_\theta$. For all $n \in \mathbb{N}$, γ_n satisfies:

(2) $\alpha_n \in (\ker \mathbf{P}_\theta)^\perp$.

(3) $\|\gamma_n\|_2 = 1$.

As the set $(\ker \mathbf{P}_\theta)^\perp$ is a finite-dimensional linear space, it is closed. Similarly the unit sphere of Supp_θ is a closed set, hence properties (1) and (2) are also satisfied for γ, showing that $\gamma \in \mathrm{Sincomp}_\theta$.

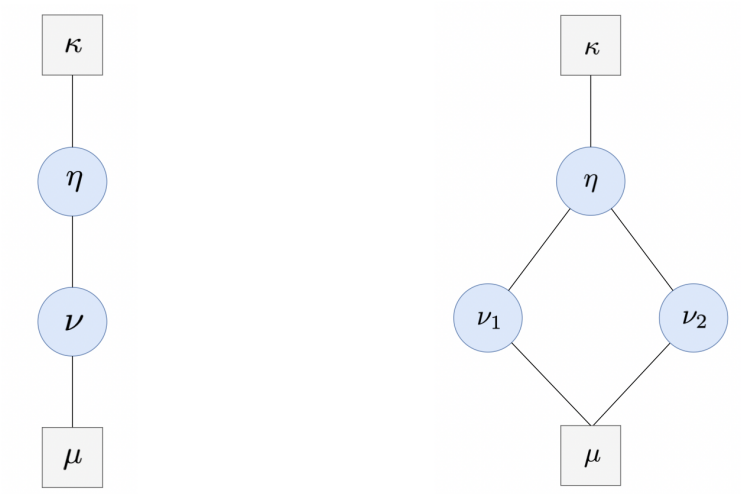


Figure A-1: Left: Stem network architecture. Right: Sawtooth network architecture. Hidden neurons are depicted as circles whereas input and output neurons as squares.

A.2.4 Illustrations on Particular Networks

We illustrate the main tools developed in Section 3.3 on two particular networks to showcase their usefulness.

Stem Network

We introduce a particular architecture G defined in Figure A-1 that we call the *Stem Network*. We enumerate the connections as $E = (\mu \to \nu, \nu \to \eta, \eta \to \kappa)$, the only full path as $\mathcal{P} = (\mu \to \nu \to \eta \to \kappa)$ and the partial paths as $\mathcal{Q} = (\nu \to \eta \to \kappa, \eta \to \kappa)$. Using Definition 3.3.1, we compute $\mathbf{P}$ and $\mathbf{Q}$ as follows:

$$\mathbf{P} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \text{ and } \mathbf{Q} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}. \tag{A.98}$$

Thus both $\mathbf{P}$ and $\mathbf{Q}$ have full rank and

$$\ker \mathbf{P} = \{\alpha \in \mathbb{R}^E \mid \alpha_{\mu\to\nu} + \alpha_{\nu\to\eta} + \alpha_{\eta\to\kappa} = 0\} \tag{A.99}$$

$$\ker \mathbf{Q} = \{\alpha \in \mathbb{R}^E \mid \alpha_{\nu\to\eta} = 0 \text{ and } \alpha_{\eta\to\kappa} = 0\} \tag{A.100}$$

We value G with $\theta = (w, b)$, where $w = (w_{\mu\to\nu}, w_{\nu\to\eta}, w_{\eta\to\kappa})$ and $b = (b_\nu, b_\eta)$. We assume that θ is admissible (note that this implies that $v_p(\theta) \neq 0$ for the only path p). In this particular case, this is equivalent to $w_e \neq 0$ for all $e \in E$. Let us assume

further that $b_\nu \neq 0$ and $b_\eta \neq 0$. Then, using the Notations,

$$\mathrm{Supp}_w = \{w' \in \mathbb{R}^E \mid \mathrm{supp}(w') \subseteq \mathrm{supp}(w)\} = \mathbb{R}^E \backslash \{0\}. \tag{A.101}$$

Similarly, $\mathrm{Supp}_b = \mathbb{R}^H \backslash \{0\}$. Using Propositions 3.3.6 and A.2.2, we deduce that

$$V_\theta = \mathrm{Supp}_w \times \mathbf{S}^{-1}(\mathrm{Supp}_b) = \mathbb{R}^E \backslash \{0\} \times \ker \mathbf{P} \backslash \{0\}. \tag{A.102}$$

Then, $\gamma \in \mathrm{Supp}_\theta$ writes $\gamma = (\alpha, \mathbf{S}\beta)$ with $(\alpha, \beta) \in V_\theta$ and the updated version of θ writes $\theta \odot e^\gamma = (w \odot e^\alpha, b \odot e^{\mathbf{S}\beta}) = (w', b')$, where

$$w' = (w_{\mu\to\nu} \exp(\alpha_{\mu\to\nu}), w_{\nu\to\eta} \exp(\alpha_{\nu\to\eta}), w_{\eta\to\kappa} \exp(\alpha_{\eta\to\kappa})) \tag{A.103}$$

$$b' = (b_\nu \exp(-\beta_{\nu\to\eta} - \beta_{\eta\to\kappa}), b_\eta \exp(-\beta_{\eta\to\kappa})) \tag{A.104}$$

To compute b', we used the definition of $\mathbf{S}$ in Propositions 3.3.6. Let us now compute the set of θ-compatible perturbations using Definition 3.3.3. First, using Definition 3.3.2, remark that $\mathbf{P} = \mathbf{P}_\theta$ and $\mathbf{Q} = \mathbf{Q}_\theta$ since $v_p(\theta) \neq 0$ for for only full path $p \in \mathcal{P}$ and since $b_\nu \neq 0$ for all $\nu \in H$. Then, γ is θ-compatible if, and only if,

$$\alpha \in \ker(\mathbf{P}_\theta) \text{ and } \alpha - \beta \in \ker(\mathbf{Q}_\theta). \tag{A.105}$$

Using Equations A.99 and A.100, we deduce that γ is θ-compatible if, and only if,

$$0 = \alpha_{\mu\to\nu} + \alpha_{\nu\to\eta} + \alpha_{\eta\to\kappa} \tag{A.106}$$

$$0 = \alpha_{\nu\to\eta} - \beta_{\nu\to\eta} \tag{A.107}$$

$$0 = \alpha_{\eta\to\kappa} - \beta_{\eta\to\kappa} \tag{A.108}$$

Hence, $-\alpha_{\eta\to\kappa} = -\beta_{\eta\to\kappa}$ and $-\beta_{\nu\to\eta} - \beta_{\eta\to\kappa} = -\alpha_{\nu\to\eta} - \alpha_{\eta\to\kappa} = \alpha_{\mu\to\nu}$. Recalling that $\gamma \in \mathrm{Supp}_\theta$ and using Equation (A.102), we also have $0 = \beta_{\mu\to\nu} + \beta_{\nu\to\eta} + \beta_{\eta\to\kappa}$, hence $\beta_{\mu\to\nu} = 0$. Plugging these results into Equation (A.104), and re-stating Equation (A.103) here for clarity, we express θ' only in terms of α as follows:

$$w' = (w_{\mu\to\nu} \exp(\alpha_{\mu\to\nu}), w_{\nu\to\eta} \exp(\alpha_{\nu\to\eta}), w_{\eta\to\kappa} \exp(\alpha_{\eta\to\kappa})) \tag{A.109}$$

$$b' = (b_\nu \exp(\alpha_{\mu\to\nu}), b_\eta \exp(-\alpha_{\eta\to\kappa})) \tag{A.110}$$

We now verify that the path-wise rescaling equivalence Definition 2.2.4 is satisfied. For the only path p, using Equation A.109,

$$v_p(\theta') = w_{\mu\to\nu}\exp(\alpha_{\mu\to\nu})w_{\nu\to\eta}\exp(\alpha_{\nu\to\eta})w_{\eta\to\kappa}\exp(\alpha_{\eta\to\kappa}) \tag{A.111}$$

$$= w_{\mu\to\nu}w_{\nu\to\eta}w_{\eta\to\kappa}\exp(\alpha_{\mu\to\nu}+\alpha_{\nu\to\eta}+\alpha_{\eta\to\kappa}) \tag{A.112}$$

$$= v_p(\theta)\exp(0) \tag{A.113}$$

$$= v_p(\theta) \tag{A.114}$$

where we used Equation (A.99). Similarly, we have $b_{q_0}v_q(\theta) = b'_{q_0}v_q(\theta')$ for both partial paths $q \in \mathcal{Q}$. Let us now give an interpretation of the parameter α. Recalling Definition 2.2.3, let us denote

$$\lambda = \exp(\mathbf{S}\alpha) \in \mathbb{R}^H. \tag{A.115}$$

We deduce that $\lambda_\nu = \exp(-\alpha_{\nu\to\eta} - \alpha_{\eta\to\kappa}) = \exp(\alpha_{\mu\to\nu})$ and $\lambda_\eta = \exp(-\alpha_{\eta\to\kappa})$. Then, we recover the neuron-wise rescaling equivalence. Indeed, $w_{\mu\to\nu}$ and b_ν are multiplied by λ_ν, $w_{\nu\to\eta}$ is multiplied by $\lambda_\eta/\lambda_\nu = \exp(-\alpha_{\eta\to\kappa}-\alpha_{\mu\to\nu}) = \exp(\alpha_{\nu\to\eta})$, b_η is multiplied by λ_η and $w_{\eta\to\kappa}$ is multiplied by $1/\lambda_\eta = \exp(\alpha_{\eta\to\kappa})$. Thus, this examples shows the duality between the weights and the biases to define rescaling-compatible perturbations, where the bridge between both spaces is implemented by $\mathbf{S}$. We also use this technique to prove the Trajectory $\implies$ Layer result Appendix A.1.1.

Sawtooth Network

We introduce an architecture G in Figure A-1 called the *Sawtooth Network*[5]. We enumerate the connections[6] as $E = (\mu\to\nu_1,\ \mu\to\nu_2,\ \nu_1\to\eta,\ \nu_2\to\eta,\ \eta\to\kappa)$, the full paths as $\mathcal{P} = (\mu\to\nu_1\to\eta\to\kappa,\ \mu\to\nu_2\to\eta\to\kappa)$ and the partial paths as $\mathcal{Q} = (\nu_1\to\eta\to\kappa, \nu_2\to\eta\to\kappa, \eta\to\kappa)$. Unlike with the previous example, we will not display all the calculations for the sake of concision. We compute

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \text{ and } \mathbf{Q} = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.116}$$

[5]When valued with a particular parameterization θ, this network is able to represent a triangular function $\mathbb{R}\to\mathbb{R}$, hence the name.

[6]In the BFS order.

Thus both $\mathbf{P}$ and $\mathbf{Q}$ have full rank and

$$\ker \mathbf{P} = \{\alpha \in \mathbb{R}^E \mid \alpha_{\mu\to\nu_1} + \alpha_{\nu_1\to\eta} + \alpha_{\eta\to\kappa} = 0 \text{ and } \alpha_{\mu\to\nu_2} + \alpha_{\nu_2\to\eta} + \alpha_{\eta\to\kappa} = 0\}.$$

We value G with $\theta = (w, b)$. We assume that θ is admissible (note that this implies that $v_p(\theta) \neq 0$ for both full paths). In this particular case, this is equivalent to $w_e \neq 0$ for all $e \in E$. Let us assume further that $b_{\nu_1} = b_{\nu_2} = b_\eta = 0$. Then,

$$\mathrm{Supp}_w = \{w' \in \mathbb{R}^E \mid \mathrm{supp}(w') \subseteq \mathrm{supp}(w)\} = \mathbb{R}^E \backslash \{0\}. \tag{A.117}$$

Similarly, $\mathrm{Supp}_b = \{0\}$. Using Propositions 3.3.6 and A.2.2, we deduce that

$$V_\theta = \mathrm{Supp}_w \times \mathbf{S}^{-1}(\mathrm{Supp}_b) = \left(\mathbb{R}^E \backslash \{0\}\right) \times \{0\}. \tag{A.118}$$

Then, $\gamma \in \mathrm{Supp}_\theta$ writes $\gamma = (\alpha, \mathbf{S}\beta)$ with $(\alpha, \beta) \in V_\theta$, hence $\gamma = (\alpha, 0)$ using Equation A.118. The perturbed version of θ writes

$$\theta \odot e^\gamma = \theta \odot e^\gamma = (w \odot e^\alpha, b \odot e^0) = (w \odot e^\alpha, b). \tag{A.119}$$

Let us now compute the set of θ-compatible perturbations using Definition 3.3.3. First, using Definition 3.3.2, remark that $\mathbf{P} = \mathbf{P}_\theta$ since $v_p(\theta) \neq 0$ for for both full paths. Then, using Definition 3.3.3, γ is θ-compatible if, and only if, $\alpha \in \ker(\mathbf{P}_\theta)$. This allows to verify that the path-wise rescaling equivalence Definition 2.2.4 is satisfied as in the previous example. Using the description of $\ker \mathbf{P}$, for the path $p_1 = \mu \to \nu_1 \to \eta \to \kappa$, we have $v_{p_1}(\theta) = v_{p_1}(\theta')$ and similarly for the other path $p_2 = \mu \to \nu_2 \to \eta \to \kappa$. As in the previous example, let us give an interpretation of the parameter α. Recalling Definition 2.2.3, let us denote

$$\lambda = \exp(\mathbf{S}\alpha) \in \mathbb{R}^H. \tag{A.120}$$

We deduce as in the previous example: $\lambda_{\nu_1} = \exp(\alpha_{\mu\to\nu_1})$, $\lambda_{\nu_2} = \exp(\alpha_{\mu\to\nu_2})$ and $\lambda_\eta = \exp(-\alpha_{\eta\to\kappa})$. Then, we recover the neuron-wise rescaling equivalence.

A.3 Locally Identifiable Parameterizations

A.3.1 Definition of Locally Identifiable Parameterizations

Proof of Proposition 3.4.2

Proposition. *Consider $\theta = (w, b)$ an* admissible *parameterization. The following properties are equivalent:*

1. *θ is restricted locally non-identifiable.*

2. *for all $\varepsilon > 0$, there exists $\tau \in]0, \varepsilon[$ and $\gamma \in Sincomp_\theta$ such that $R_{\theta \odot e^{\gamma\tau}} = R_\theta$.*

Proof. First we show 2 $\Rightarrow$ 1. Let Ω be a neighborhood of θ. Considering an arbitrary $\varepsilon > 0$, property 2 implies the existence of $\tau \in]0, \varepsilon[$ and $\gamma = (\alpha, \mathbf{S}\beta) \in \mathrm{Sincomp}_\theta$ such that $R_{\theta'} = R_\theta$ with $\theta' \triangleq \varphi_\theta(\tau\alpha, \tau\beta)$ where φ_θ is defined in Proposition A.2.2. Since θ is admissible, we can use the first point of Proposition 3.3.8 to deduce that $\gamma \in \overline{\mathrm{Comp}}_\theta$, hence $\tau\gamma \in \overline{\mathrm{Comp}}_\theta$. By the first point of Remark 3.3.2, it follows that $\theta' \not\sim_S \theta$. Since $\theta' \in \mathrm{Sign}_\theta$, and since $\theta' \in \Omega$ for ε small enough, we obtain that θ is restricted locally non-identifiable.

We now show 1 $\Rightarrow$ 2. Let θ be restricted locally non-identifiable, and consider an arbitrary $\varepsilon > 0$. Since the set $B(\varepsilon) \triangleq \{(\alpha, \beta) \in V_\theta,\ \|(\alpha, \beta)\|_2 < \varepsilon\}$ is open in the linear space V_θ and since the function $f \triangleq \varphi_\theta^{-1} : \mathrm{Sign}_\theta \to V_\theta$ from Proposition A.2.2 is continuous, the set $\varphi_\theta(B(\varepsilon)) = f^{-1}(B(\varepsilon)) \subset \mathrm{Sign}_\theta$ is open in Sign_θ, hence there exists an open set $\Omega_\varepsilon \subset \mathbb{R}^{E \cup H}$ such that $\Omega_\varepsilon \cap \mathrm{Sign}_\theta = \varphi_\theta(B(\varepsilon))$. As $\theta \in \Omega_\varepsilon$, Ω_ε is a neighborhood of θ. Since θ is restricted locally non-identifiable, there exists $\theta' \in \Omega_\varepsilon \cap \mathrm{Sign}_\theta$ such that $R_{\theta'} = R_\theta$ and $\theta' \not\sim_S \theta$. Denote $(\alpha, \beta) \triangleq \varphi_\theta^{-1}(\theta')$. By construction of Ω_ε, $(\alpha, \beta) \in V_\theta$ satisfies $\theta' = \varphi_\theta(\alpha, \beta)$ and

$$\|(\alpha, \beta)\|_2 < \varepsilon \tag{A.121}$$

Since $\theta' \in \mathrm{Sign}_\theta$ and $\theta' \not\sim_S \theta$, by the first point of Proposition 3.3.2 (remember that θ is admissible) we obtain that $\gamma = (\alpha, \mathbf{S}\beta) \in \overline{\mathrm{Comp}}_\theta$.

Moreover, $R_\theta = R_{\theta'}$. We now massage γ to get a *strongly* non θ-rescaling-compatible $\tilde{\gamma} = (\tilde{\alpha}, \mathbf{S}\tilde{\beta})$ and $\tau \in]0, \varepsilon[$ such that $R_{\theta \odot e^{\gamma\tau}} = R_\theta$ as claimed.

Step 1: projection. We write $(\alpha, \beta) = (\alpha^\perp, \beta^\perp) + (\alpha^\parallel, \beta^\parallel)$ where $(\alpha^\perp, \beta^\perp) \in V_\theta$ is non θ-rescaling-compatible, $\alpha^\perp \in (\ker(\mathbf{P}_\theta))^\perp$, and

$$R_{\varphi_\theta(\alpha^\perp, \beta^\perp)} = R_{\varphi_\theta(\alpha,\beta)} = R_\theta. \tag{A.122}$$

First, decompose $\alpha \in \mathrm{Supp}_w \subset \mathbb{R}^E$ as

$$\alpha = \alpha^\perp + \alpha^\parallel, \text{ where } \begin{cases} \alpha^\perp \in (\ker(\mathbf{P}_\theta))^\perp \\ \alpha^\parallel \in \ker(\mathbf{P}_\theta) \end{cases}$$

We next construct $\beta^\parallel$ as follows. We now extend the definition of $\mathbf{S} :: \ker \mathbf{P} \to \mathbb{R}^H$ to $\mathbf{T} \colon \ker \mathbf{P}_\theta \to \mathbb{R}^H$ (see Proposition 3.3.6) and similarly prove that $\mathbf{T}$ is well-defined (but $\mathbf{T}$ is not an isomorphism). Next, we define $\delta = \mathbf{T}(\alpha^\parallel) \in \mathbb{R}^H$. Next, define

$$\widetilde{\delta}_\nu \triangleq \begin{cases} \delta_\nu & \text{if } b_\nu \neq 0 \\ 0 & \text{if } b_\nu = 0 \end{cases} \tag{A.123}$$

Finally, define $\beta^\parallel \triangleq \mathbf{S}^{-1}(\widetilde{\delta}) \in \ker \mathbf{P}$ and $\beta^\perp \triangleq \beta - \beta^\parallel$. We now show that $(\alpha^\perp, \beta^\perp) \in V_\theta$ is non θ-rescaling-compatible.

- Let us show that $\alpha^\parallel, \alpha^\perp \in \mathrm{Supp}_w$. First, by definition,

  $$\alpha^\perp \in (\ker(\mathbf{P}_\theta))^\perp = \mathrm{Im}\left(\mathbf{P}^T D_{\mathcal{P}}(\theta)\right),$$

  where $D_{\mathcal{P}}(\theta)$ is a diagonal matrix such that $D_{\mathcal{P}}(\theta)_{p,p} = \mathbf{1}(v_p(\theta) \neq 0)$ Given $e \in E$ such that $w_e = 0$, we show that $\alpha^\perp_e = 0$. It suffices to show that for every path $p \in \mathcal{P}$, $\left(\mathbf{P}^T D_{\mathcal{P}}(\theta)\right)_{e,p} = 0$. This is achieved by treating two cases:

  - for p such that $e \notin p$, we have $\left(\mathbf{P}^T\right)_{e,p} = (\mathbf{P})_{p,e} = \mathbf{1}(e \in p) = 0$;
  - for p such that $e \in p$, since $w_e = 0$, $D_{\mathcal{P}}(\theta)_{p,p} = \mathbf{1}(v_p(\theta) \neq 0) = 0$.

  Both cases yield

  $$\left(\mathbf{P}^T D_{\mathcal{P}}(\theta)\right)_{e,p} = \left(\mathbf{P}^T\right)_{e,p} (D_{\mathcal{P}}(\theta))_{p,p} = 0, \tag{A.124}$$

  hence $\alpha^\perp_e = 0$. This shows that $\alpha^\perp \in \mathrm{Supp}_w$. As $\alpha \in \mathrm{Supp}_w$, it follows that $\alpha^\parallel = \alpha - \alpha^\perp \in \mathrm{Supp}_w$.

- Let us show that $\beta^{\|}, \beta^{\perp} \in \mathbf{S}^{-1}(\mathrm{Supp}_b)$. Using Equation (A.123), we deduce that $\widetilde{\delta} \in \mathrm{Supp}_b$. Hence, $\beta^{\|} \in \mathbf{S}^{-1}(\mathrm{Supp}_b)$. Since $\beta \in \mathbf{S}^{-1}(\mathrm{Supp}_b)$, we also have $\beta^{\perp} = \beta - \beta^{\|} \in \mathbf{S}^{-1}(\mathrm{Supp}_b)$.

- Let us show that $w^{\|} - \beta^{\|} \in \ker(\mathbf{Q}_\theta)$. It suffices to show that, for every partial path $q \in \mathcal{Q}$,

$$b_{q_0} v_q(\theta) \sum_{e \in q} (\alpha_e^{\|} - \beta_e^{\|}) = 0 \tag{A.125}$$

  We only need to show (A.125) for any partial path q such that $b_{q_0} \neq 0$ and $v_q(\theta) \neq 0$. Let q be such a partial path. Then,

$$\sum_{e \in q} \alpha_e^{\|} = -\mathbf{T}(\alpha^{\|})_{q_0} = -\delta_{q_0} \tag{A.126}$$

  and

$$\sum_{e \in q} \beta_e^{\|} = -\mathbf{S}(\beta^{\|})_{q_0} = -\widetilde{\delta}_{q_0} \tag{A.127}$$

  We conclude using the fact that $\delta_{q_0} = \widetilde{\delta}_{q_0}$.

As $\alpha^{\|} \in \ker \mathbf{P}_\theta$ and $\alpha^{\|} - \beta^{\|} \in \ker \mathbf{Q}_\theta$, we get that $(\alpha^{\|}, \beta^{\|})$ is θ-rescaling compatible. Since $(\alpha, \beta) = (\alpha^{\perp}, \beta^{\perp}) + (\alpha^{\|}, \beta^{\|})$ is non θ-rescaling compatible, this implies that $(\alpha^{\perp}, \beta^{\perp})$ is non θ-rescaling compatible as claimed. Using the above remarks, let us finally show that

$$R_{\varphi_\theta(\alpha,\beta)} = R_{\varphi_\theta(\alpha^{\perp},\beta^{\perp})} \tag{A.128}$$

Define $\theta_0 = \varphi_\theta(\alpha^{\perp}, \beta^{\perp})$. Let us show that $(\alpha^{\|}, \beta^{\|})$ is θ_0-rescaling-compatible. Since $(\alpha^{\|}, \beta^{\|})$ is θ-rescaling-compatible, if suffices to show that $D_{\mathcal{P}}(\theta) = D_{\mathcal{P}}(\theta_0)$ and $D_{\mathcal{Q}}(\theta) = D_{\mathcal{Q}}(\theta_0)$. This is indeed a trivial consequence of the fact that $\mathrm{sign}(\theta) = \mathrm{sign}(\theta_0)$, hence $(\alpha^{\|}, \beta^{\|})$ is θ_0-rescaling-compatible. Next, using Proposition 3.3.2, we deduce that

$$R_{\varphi_{\theta_0}(\alpha^{\|},\beta^{\|})} = R_{\theta_0} \tag{A.129}$$

We conclude by using the fact that $\varphi_{\theta_0}(\alpha^{\|}, \beta^{\|}) = \varphi_\theta(\alpha, \beta)$.

Step 2: normalization. Set $\tau \triangleq \|(\alpha^{\perp}, \beta^{\perp})\|_2$. As $(\alpha^{\perp}, \beta^{\perp})$ is non θ-rescaling-compatible, it is nonzero hence $\tau > 0$. We can thus define

$$\widetilde{\alpha} = \frac{1}{\tau}\alpha^{\perp} \qquad \widetilde{\beta} = \frac{1}{\tau}\beta^{\perp} \tag{A.130}$$

so that $\|(\widetilde{\alpha}, \widetilde{\beta})\|_2 = 1$. Since $\alpha^\perp \in (\ker \mathbf{P}_\theta)^\perp$, we have $\widetilde{\alpha} \in (\ker \mathbf{P}_\theta)^\perp$, showing that $(\widetilde{\alpha}, \widetilde{\beta})$ is *strongly* non θ-rescaling-compatible. In summary, there exists $\tau > 0$ and a joint perturbation direction $(\widetilde{\alpha}, \widetilde{\beta}) \in V_\theta$ that is strongly non θ-rescaling-compatible such that

$$R_{\varphi_\theta(\widetilde{\alpha}, \widetilde{\beta}, \tau)} = R_{\varphi_\theta(\alpha^\perp, \beta^\perp)} = R_{\varphi_\theta(\alpha, \beta)} = R_{\theta'} = R_\theta.$$

To conclude, we show that τ can be chosen arbitrarily small. By the definition of $\alpha^\perp$ as an orthogonal projection onto $(\ker \mathbf{P}_\theta)^\perp$, we have $\|\alpha^\perp\|_2 \leq \|\alpha\|_2$. Moreover, let us denote $\|f\| = \|f\|_{2,2}$ the operator norm of any linear map $f \colon E \to F$, defined as

$$\|f\| = \sup_{\substack{x \in E \\ x \neq 0}} \frac{\|f(x)\|_2}{\|x\|_2}. \tag{A.131}$$

Recalling that: $\beta^\parallel = \mathbf{S}^{-1}(\widetilde{\delta})$, $\|\widetilde{\delta}\|_2 \leq \|\delta\|_2$ and $\delta = \mathbf{T}(\alpha^\parallel)$, we get

$$\|\beta^\parallel\|_2 \leq \|\mathbf{S}^{-1}\| \|\widetilde{\delta}\|_2 \leq \|\mathbf{S}^{-1}\| \|\delta\|_2 \leq \|\mathbf{S}^{-1}\| \|\mathbf{T}\| \|\alpha^\parallel\|_2. \tag{A.132}$$

Next, since $\beta^\perp = \beta - \beta^\parallel$,

$$\|\beta^\perp\|_2 \leq \|\beta\|_2 + \|\beta^\parallel\|_2 \leq \|\beta\|_2 + \|\mathbf{S}^{-1}\| \|\mathbf{T}\| \|\alpha^\parallel\|_2 \tag{A.133}$$

Recalling that $\|(\alpha, \beta)\|_2 \leq \varepsilon$ with Equation (A.121), we deduce that $\|\alpha\|_2 \leq \varepsilon$ and $\|\beta\|_2 \leq \varepsilon$. Thus,

$$\|\beta^\perp\|_2 \leq \varepsilon(1 + \|\mathbf{S}^{-1}\| \|\mathbf{T}\|). \tag{A.134}$$

Combining the above estimates, we get that $\tau \leq \varepsilon(2 + \|\mathbf{S}^{-1}\| \|\mathbf{T}\|)$ can be chosen arbitrarily small by properly choosing ε. □

A.3.2 Sufficient Condition for Restricted Local Identifiability

Proof of Proposition 3.4.3

Proposition. *Let $\theta = (w, b)$ be an admissible, restricted locally non-identifiable parameterization. Then, there exists $\gamma \in Sincomp_\theta$ such that, for all $x \in \mathcal{X}_\theta$,*

$$\langle \nabla_\theta R_\theta(x), \theta \odot \gamma \rangle = 0. \tag{A.135}$$

Proof. Let $n \in \mathbb{N}$. Since θ is restricted locally non-identifiable parameterization, by Proposition 3.4.2 with $\varepsilon = 1/(n+1)$, there exists $\tau_n \in]0, 1/(n+1)[$ and $\gamma_n =$

$(\alpha_n, \mathbf{S}\beta_n) \in \mathrm{Sincomp}_\theta$ such that

$$R_{\theta \odot e^{\gamma_n \tau_n}} = R_\theta. \tag{A.136}$$

By Proposition 3.3.8, $\mathrm{Sincomp}_\theta$ is compact, hence there exists a subsequence of γ_n that converges to $\gamma = (\alpha, \mathbf{S}\beta)$. Without loss of generality and to make notations lighter, we assume that the sequence γ_n converges. Thus, $\lim_{n\to\infty} \alpha_n = \alpha$, $\lim_{n\to\infty} \beta_n = \beta$ and $\lim_{n\to\infty} \tau_n = 0$.

Denote $\rho_{\gamma,x}(t) = R_{\theta \odot e^{\gamma t}}(x)$ and consider $x \in \mathcal{X}_\theta$. By Proposition 3.3.7, there is a neighborhood Ω of $t = 0$ such that, for each $t \in \Omega$ and each $n \in \mathbb{N}$,

$$\rho_{\gamma_n,x}(t) = \underbrace{\sum_{p\in\mathcal{P}} x_{p_0} v_p(\theta) e^{t(\mathbf{P}\alpha_n)_p} a_p(\theta, x)}_{\rho_{1,n}(t)} + \underbrace{\sum_{q\in\mathcal{Q}} b_{q_0} v_q(\theta) e^{t(\mathbf{Q}(\alpha_n - \beta_n))_q} a_q(\theta, x)}_{\rho_{2,n}(t)} \tag{A.137}$$

where we used that $(\alpha_n, \beta_n) \in B_\theta$ since $\gamma_n \in \mathrm{Sincomp}_\theta$. In particular, we obtain that for every $n \in \mathbb{N}$, $t \mapsto \rho_{\gamma_n,x}(t)$ is $\mathcal{C}^\infty$ in Ω.

Consider n large enough such that $\tau_n \in \Omega$. Taylor-Lagrange's formula yields $\xi_n \in]0, \tau_n[\subset \Omega$ such that

$$\rho_{\gamma_n,x}(\tau_n) = \rho_{\gamma_n,x}(0) + \tau_n \rho'_{\gamma_n,x}(0) + \frac{\tau_n^2}{2} \rho''_{\gamma_n,x}(\xi_n). \tag{A.138}$$

Let us show that there exists $C \geq 0$ such that, for all n large enough,

$$|\rho''_{\gamma_n,x}(\xi_n)| \leq C. \tag{A.139}$$

Differentiating $\rho_{1,n}$ twice with respect to t and evaluating at $\xi_n \in \Omega$ yields

$$\rho''_{1,n}(\xi_n) = \sum_{p\in\mathcal{P}} x_{p_0} v_p(\theta) \left((\mathbf{P}\alpha_n)_p\right)^2 e^{\xi_n (\mathbf{P}\alpha_n)_p} a_p(\theta, x) \tag{A.140}$$

We denote $\|\mathbf{P}\| = \|\mathbf{P}\|_{2,2}$ the spectral norm of $\mathbf{P}$ and similarly for $\mathbf{Q}$. Recalling that $||\gamma_n\|_2 = 1$, we have $\|\alpha_n\|_2 \leq 1$. Then, for every path $p \in \mathcal{P}$,

$$|(\mathbf{P}\alpha_n)_p| \leq \|\mathbf{P}\alpha_n\|_2 \leq \|\mathbf{P}\| \|\alpha_n\|_2 \leq \|\mathbf{P}\|. \tag{A.141}$$

Moreover, since $0 < \xi_n < \tau_n \leq 1$, we use Equation (A.141) to deduce

$$e^{\xi_n(\mathbf{P}\alpha_n)_p} \leq e^{|\xi_n(\mathbf{P}\alpha_n)_p|} \leq e^{|(\mathbf{P}\alpha_n)_p|} \leq e^{\|\mathbf{P}\|}. \tag{A.142}$$

Thus, $\rho''_{1,n}(\xi_n)$ is bounded from above by a term that is independent of n, namely

$$\left|\rho''_{1,n}(\xi_n)\right| \leq \|\mathbf{P}\|^2 e^{\|\mathbf{P}\|} \sum_{p\in\mathcal{P}} \left|x_{p_0} v_p(\theta) a_p(\theta, x)\right|. \tag{A.143}$$

Similarly, we show that, for n large enough,

$$\left|\rho''_{2,n}(\xi_n)\right| \leq \|\mathbf{Q}\|^2 e^{2\|\mathbf{Q}\|} \sum_{q\in\mathcal{Q}} \left|b_{q_0} v_q(\theta) a_q(\theta, x)\right|. \tag{A.144}$$

Summing the right hand sides of (A.143)-(A.144) yields $C \geq 0$ (independent of n) such that (A.139) holds for n large enough. Since

$$\rho_{\gamma_n,x}(\tau_n) = R_G\left(\theta \odot e^{\gamma_n \tau_n}\right) \overset{(A.136)}{=} R_G(\theta) = \rho_{\gamma_n,x}(0),$$

we use Equation (A.138) to obtain

$$0 = \tau_n \rho'_{\gamma_n,x}(0) + \frac{\tau_n^2}{2} \rho''_{\gamma_n,x}(\xi_n),$$

hence, since $\tau_n \neq 0$,

$$\rho'_{\gamma_n,x}(0) = -\frac{\tau_n}{2} \rho''_{\gamma_n,x}(\xi_n).$$

Using Equation (A.139), we deduce that, for n large enough,

$$\left|\rho'_{\gamma_n,x}(0)\right| \leq \tau_n \frac{C}{2}. \tag{A.145}$$

Hence $\rho'_{\gamma_n,x}(0) \to_{n\to\infty} 0$. Since $\lim_{n\to\infty} \rho'_{\gamma_n,x}(0) = \rho'_{\gamma,x}(0)$, we deduce $\rho'_{\gamma,x}(0) = 0$.

Finally, differentiating the function $t \mapsto \rho_{\gamma,x}(t)$ and evaluating in $t = 0$ yields:

$$\rho'_{\gamma,x}(0) = \langle \nabla_\theta R_\theta(x), \theta \odot \gamma \rangle \tag{A.146}$$

which concludes the proof. □

Proof of Proposition 3.4.4

Proposition. *Let G be a one hidden layer architecture with scalar output valued with an admissible parameterization $\theta = (w, b)$ such that $b = 0$ (no biases). We assume that θ is restricted locally non-identifiable. Then there exists at least two twin hidden neurons as defined in 3.2.2.*

Proof. Using Proposition 3.4.3, there exists $\gamma = (\alpha, \mathbf{S}\beta) \in \text{Sincomp}_\theta$ such that, for all $x \in \mathcal{X}_\theta$, $\langle \nabla_\theta R_\theta(x), \theta \odot \gamma \rangle = 0$. Since $(\alpha, \beta) \in V_\theta = \text{Supp}_w \times \mathbf{S}^{-1}(\text{Supp}_b)$ (see Equation (A.102)) and since $b = 0$, we have $\mathbf{S}^{-1}(\text{Supp}_b) = \{0\}$ and $\beta = 0$. Then, using Equation (A.146), notice that

$$\langle \nabla_\theta R_\theta(x), \theta \odot \gamma \rangle = \rho'_{\gamma,x}(0) \tag{A.147}$$

where $\rho_{\gamma,x}$ is defined in the proof of Proposition 3.4.3. Then, using Proposition 3.3.7,

$$\begin{aligned} 0 &= \langle \nabla_\theta R_\theta(x), \theta \odot \gamma \rangle && \text{(A.148)} \\ &= \sum_{p \in \mathcal{P}} x_{p_0} v_p(\theta) \mathbf{P}_\theta(\alpha)_p a_p(\theta, x). && \text{(A.149)} \end{aligned}$$

Recalling that $\gamma \in \text{Sincomp}_\theta$ (Definition 3.3.5), we have $\alpha \in \ker(\mathbf{P}_\theta)^\perp$ and $\|\gamma\|_2 = 1$. In particular, $\gamma = (\alpha, 0) \neq 0$, hence $\alpha \neq 0$ and $\mathbf{P}_\theta(\alpha) \neq 0$. Moreover, since θ is admissible, we have $v_p(\theta) \neq 0$ for all full path p. Hence, the vector $\delta \in \mathbb{R}^{\mathcal{P}}$ such that $\delta_p = \mathbf{P}_\theta(\alpha)_p v_p(\theta)$ is non-zero. Next, for all $\mu \in N_0$, deriving Equation (A.148) with respect to x_μ, we get, for all $x \in \mathcal{X}_\theta$,

$$\sum_{\substack{p \in \mathcal{P} \\ p_0 = \mu}} \delta_p a_p(\theta, x) = 0 \tag{A.150}$$

Let $p \in \mathcal{P}$ be a full path. Since p is given by three neurons $\mu, \nu \in N_0 \times N_1$ and since the output is scalar (output neuron η), we can write $p = \mu \to \nu \to \eta$ with $\mu, \nu \in N_0 \times N_1$ and by definition, $a_p(\theta, x) = a_\nu(\theta, x)$ since there is only one hidden layer. Finally, we can label δ_p as $\delta_{\mu \to \nu}$. Hence, we re-write (A.150) as

$$\sum_{\nu \in N_1} \delta_{\mu \to \nu} a_\nu(\theta, x) = 0 \tag{A.151}$$

Let $\mu \in N_0$ such that the vector $(\delta_{\mu \to \nu})_{\nu \in N_1}$ is non-zero (possible since δ is non-zero). Hence, the functions $x \mapsto a_\nu(\theta, x) = \sigma(\langle w_{\bullet \to \nu}, x \rangle + b_\nu), \nu \in N_1$ are linearly dependent.

Hence, at least two separating hyperplanes Γ_{ν_1} and Γ_{ν_2} are the same, hence there are at least two twin neurons.

□

A.3.3 Sufficient Condition for Local Identifiability

Proof of Proposition 3.4.5

Proposition. *Let θ be a parameterization. There exists a neighborhood Ω_θ of θ and a finite set of points $\mathcal{Z}_\theta \subset \mathcal{X}_\theta$ that satisfy the following properties:*

- *The following equality holds*

$$\operatorname*{Span}_{z\in\mathcal{Z}_\theta}(c(\theta,z)) = \operatorname*{Span}_{x\in\mathcal{X}_\theta}(c(\theta,x)). \tag{A.152}$$

- *For all $\theta' \in \Omega_\theta$ and for all $z \in \mathcal{Z}_\theta$, $c(\theta', z) = c(\theta, z)$.*

Proof. First, $\operatorname{Span}_{z\in\mathcal{X}_\theta}(c(\theta,z))$ is a vector space of finite dimension, hence it is generated by a finite set of points $\mathcal{Z}_\theta = (z_1, \ldots, z_r)$. Moreover, since $Z_\theta \subset X_\theta$ and since X_θ is finite, using Proposition 3.3.2, there exists a neighborhood Ω_θ of θ such that $c(\theta', z) = c(\theta, z)$. □

Proof of Proposition 3.4.6

Proposition. *Let θ be a parameterization, and Ω_θ, $\mathcal{Z}_\theta$ given as in Proposition 3.4.5. Denote by $\mathcal{C}_\theta$ the vector space*

$$\mathcal{C}_\theta = \operatorname*{Span}_{x\in\mathcal{X}_\theta}(c(\theta,x)). \tag{A.153}$$

Then, for any $\theta' \in \Omega_\theta$, if $R_\theta = R_{\theta'}$, then $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$.

Proof. Let $\theta' \in \Omega_\theta$ such that $R_\theta = R_{\theta'}$. Using the algebraic expression of R_θ in Proposition 3.3.1 and Definition 3.4.3, we have that, for all x, $R_\theta(x) = \langle u(\theta), c(\theta, x)\rangle$. Then, $R_\theta = R_{\theta'}$ implies that, for all $x \in \mathbb{R}^{|N_0|}$,

$$\langle u(\theta), c(\theta, x)\rangle = \langle u(\theta'), c(\theta', x)\rangle. \tag{A.154}$$

We evaluate Equation (A.154) at $z \in \mathcal{Z}_\theta$ and using the second point of Proposition 3.4.5 to deduce that for all $z \in \mathcal{Z}_\theta$,

$$\langle u(\theta') - u(\theta), c(\theta, z) \rangle = 0. \tag{A.155}$$

Using the first point of Proposition 3.4.5, we deduce that $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$. □

Proof of Proposition 3.4.7

Proposition. *Let θ be a locally non-identifiable parameterization. Then, for all neighborhood Ω of θ, there exists $\theta' \in \Omega$ such that $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$ and $u(\theta) \neq u(\theta')$.*

Proof. Let Ω_θ given as in Proposition 3.4.5. Since θ is locally non-identifiable and since $\Omega \cap \Omega_\theta$ is a neighborhood of θ, there exists $\theta' \in \Omega \cap \Omega_\theta \subset \Omega$ such that $R_{\theta'} = R_\theta$ and $\theta' \not\sim_S \theta$. Using Proposition 3.4.6, $R_{\theta'} = R_\theta$ implies that $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$. Using Definition 2.2.4, since $\theta' \not\sim_S \theta$, we have $u(\theta) \neq u(\theta')$. □

Proof of Proposition 3.4.9

Proposition. *Let G be a one hidden layer architecture with scalar output valued with an admissible parameterization θ. We further assume that there are no twin neurons,* i.e. *that θ is irreducible as defined in 3.2.4, Then, $\mathcal{C}_\theta^\perp = \{0\}$.*

Proof. Assume that there are no twin neurons. Let us show that $\mathcal{C}_\theta^\perp = \{0\}$. Since G is a one hidden layer architecture with scalar output, any full path $p = \mu \to \nu \to \eta \in \mathcal{P}$ is given by an input neuron $\mu \in N_0$ and a hidden neuron $\nu \in N_1$. Indeed, η is the only output neuron. Using the Notations, recall that for any $\mu \in N_1$ and $\nu \in N_2$,

$$a_{\mu \to \nu \to \eta}(\theta, x) = a_\nu(\theta, x). \tag{A.156}$$

Similarly, any partial path $q = \nu \to \eta$ is given by a hidden neuron $\nu \in N_1$ and $a_{\nu \to \eta}(\theta, x) = a_\nu(\theta, x)$. Using Definition 3.4.3, for all $x \in \mathcal{X}_\theta$, $c(\theta, x)$ is the concatenation of $(x_\mu a_\nu(\theta, x))_{\mu \in N_0,\ \nu \in N_1}$ and of $(a_\nu(\theta, x))_{\nu \in N_1}$. For clarity, we will omit the dependency on θ and simply write $a_\nu(x)$. For all $\nu \in N_1$, let $d_\nu \in \mathbb{R}^{N_0}$ and $e_\nu \in \mathbb{R}$. Then, using Proposition 3.4.5, any vector $((d_\nu)_{\nu \in N_1}, (e_\nu)_{\nu \in N_1}) \in \mathcal{C}_\theta^\perp$ satisfies

$$\sum_{\nu \in N_1} a_\nu(x)(\langle d_\nu, x \rangle + e_\nu) = 0 \tag{A.157}$$

for all $x \in \mathcal{X}_\theta$. Let $\nu \in N_1$. Since there are no twin neurons, the separating hyperplanes $\Gamma_{\nu'}$ defined in 3.2.2 are all distinct for $\nu' \in N_1$. Hence, there exists $x \in \Gamma_\nu$ and a neighborhood $\Omega \in \mathbb{R}^{N_0}$ of x such that

$$\Omega \cap \left(\bigcup_{\nu' \in N_1} \Gamma_{\nu'} \right) = \Omega \cup \Gamma_\nu. \tag{A.158}$$

Define the half spaces $\Gamma_\nu^+ = \{x \in \mathbb{R}^{N_0} \mid \langle w_{\bullet \to \nu}, x \rangle + b_\nu > 0\}$ and similarly for Γ_ν^-. Let $x^+ \in \Omega \cup \Gamma_\nu^+$ and $x^- \in \Omega \cup \Gamma_\nu^-$. We have $a_\nu(x^+) = 1$ and $a_\nu(x^-) = 0$. Moreover, for all $\nu' \neq \nu$, we have $a_\nu(x^+) = a_\nu(x^-)$. We evaluate Equation (A.157) on x^+ and x^- and subtract both equations to get

$$\left\langle x^- - x^+, z \right\rangle = \left\langle d_\nu, x^+ \right\rangle + e_\nu \tag{A.159}$$

where z is a quantity that does not depend on the choice of x^+ and x^- defined as

$$z = \sum_{\substack{\nu' \in N_1 \\ \nu' \neq \nu}} a_{\nu'}(x^+) d_{\nu'}. \tag{A.160}$$

Let $\tau \in \mathbb{R}^{N_0}$ be any direction such that $x^+ + \tau \in \Omega \cup \Gamma_\nu^+$ and $x^- + \tau \in \Omega \cup \Gamma_\nu^-$. Evaluating Equation (A.159) at $x^+ + \tau$ and $x^- + \tau$ yields

$$\langle x^- - x^+, z \rangle - \langle d_\nu, x^+ \rangle - e_\nu = \langle d_\nu, \tau \rangle . \tag{A.161}$$

Since the left-hand side of Equation (A.159) does not depend on τ, we deduce that $d_\nu = 0$. Since this is valid for every $\nu' \in N_1$, we deduce from Equation (A.160) that $z = 0$. Then, Equation (A.159) implies that $e_\nu = 0$. We apply the same reasoning to every $\nu \in H$ to deduce that $\mathcal{C}_\theta^\perp = \{0\}$. □

Proof of Proposition 3.4.8

Proposition. *Let θ be an admissible parameterization such that $\mathcal{C}_\theta^\perp = \{0\}$. Then, θ is locally identifiable.*

Proof. Let us show that θ is locally identifiable. For the sake of contradiction, let us assume that θ is locally non-identifiable. Then, using Proposition 3.4.7, we must have both $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp = \{0\}$ and $u(\theta) \neq u(\theta')$, which is a contradiction. □

A.3.4 Current Limitations and Discussions

Proof of Proposition 3.4.10

Proposition. *Assume that θ satisfies the following property*

$$\forall \varepsilon > 0, \exists \eta > 0, \forall \theta', \|u(\theta') - u(\theta)\| < \eta \implies \exists \theta'', u(\theta'') = u(\theta'), \|\theta'' - \theta\| < \varepsilon \quad (*)$$

Then, the two following properties are equivalent:

(i) There exists $\varepsilon > 0$ such that, for all $\theta' \in B(\theta, \varepsilon)$ such that $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$, we have $u(\theta') = u(\theta)$.

(ii) There exists $\eta > 0$ such that $(\mathrm{Im}(u) - u(\theta)) \cap B_\infty(0, \eta) \cap \mathcal{C}_\theta^\perp = \{0\}$.

Proof. The implication $(ii) \implies (i)$ hold even without assuming $(*)$. Assume that (ii) holds for some $\eta > 0$. Since the function u is polynomial, it is locally Lipschitz and we have

$$L \triangleq \sup_{\theta' \in B(\theta,1), \theta' \neq \theta} \frac{\|u(\theta') - u(\theta)\|}{\|\theta' - \theta\|} < \infty. \quad (A.162)$$

Let $\varepsilon \triangleq \min(1, \eta/L)$. For all $\theta' \in B(\theta, \varepsilon)$ that satisfies $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$, we have $u(\theta') - u(\theta) \in (\mathrm{Im}(u) - u(\theta)) \cap B_\infty(0, \eta) \cap \mathcal{C}_\theta^\perp = \{0\}$. Hence, $u(\theta') = u(\theta)$.

Let us now prove $(i) \implies (ii)$ using $(*)$. Let $\varepsilon > 0$ satisfying (i), that we re-write $(u(B(\theta, \varepsilon)) - u(\theta)) \cap \mathcal{C}_\theta^\perp = \{0\}$. Using $(*)$, there exists $\eta > 0$ such that $(\mathrm{Im}(u) - u(\theta)) \cap B_\infty(0, \eta) \subset u(B(\theta, \varepsilon)) - u(\theta)$. We conclude that

$$(\mathrm{Im}(u) - u(\theta)) \cap B_\infty(0, \eta) \cap \mathcal{C}_\theta^\perp \subset= (u(B(\theta, \varepsilon)) - u(\theta)) \cap \mathcal{C}_\theta^\perp = \{0\}. \quad (A.163)$$

□

Proof of Proposition 3.4.11

Proposition. *Let assume that θ satisfies $(*)$. Let us assume further that there exists $\varepsilon, r > 0$ and a finite set $\mathcal{Z}_\theta \subset \mathcal{X}_\theta$ such that $\tilde{\mathcal{Z}}_\theta \triangleq \cup_{z \in \mathcal{Z}_\theta} B(z, r) \subset \mathcal{X}_\theta$ and such that, for all $\theta' \in B(\theta, \varepsilon)$,*

(1) If for all $z \in \tilde{\mathcal{Z}}_\theta$, $R_\theta(z) = R_{\theta'}(z)$, then $R_\theta = R_{\theta'}$.

(2) For all $z \in \tilde{\mathcal{Z}}_\theta$, $c(\theta', z) = c(\theta, z)$

Then the two following properties are equivalent:

(i) θ is locally identifiable.

(ii) There exists $\eta > 0$ such that $(\mathrm{Im}(u) - u(\theta)) \cap B_\infty(0, \eta) \cap \mathcal{C}_\theta^\perp = \{0\}$.

Proof. Let us show that $(ii) \implies (i)$. Since we assume $(*)$, we leverage Proposition 3.4.10 to deduce from (ii) that there exists $\varepsilon_1 > 0$ such that, for all $\theta' \in B(\theta, \varepsilon_1)$, if $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$, then $u(\theta') = u(\theta)$. Using Proposition 3.4.7, we deduce that θ is locally identifiable.

Let us show $(i) \implies (ii)$. Since θ is locally identifiable, there exists $\varepsilon_0 > 0$ such that, for all $\theta' \in B(\theta, \varepsilon > 0)$, $R_\theta = R_{\theta'}$ implies that $u(\theta) = u(\theta')$.

Let $\varepsilon' = \min(\varepsilon, \varepsilon_0)$ and $\theta' \in B(\theta, \varepsilon')$ such that $u(\theta') - u(\theta) \in \mathcal{C}_\theta^\perp$. Let us show that $u(\theta) = u(\theta')$. We will then be able to conclude with Proposition 3.4.10 since θ satisfies $(*)$.

For $z \in$, assumption (2) implies that $c(\theta', z) = c(\theta, z)$. Since $u(\theta') - u(\theta) \in\perp C_\theta$, we have

$$\begin{aligned} R_{\theta'}(z) - R_\theta(z) &= \langle u(\theta'), c(\theta', z)\rangle - \langle u(\theta), c(\theta, z)\rangle && \text{(A.164)} \\ &= \langle u(\theta'), c(\theta, z)\rangle - \langle u(\theta), c(\theta, z)\rangle && \text{(A.165)} \\ &= \langle u(\theta') - u(\theta), c(\theta, z)\rangle && \text{(A.166)} \\ &= 0 && \text{(A.167)} \end{aligned}$$

Using Assumption (1), we deduce that $R_\theta = R_{\theta'}$. Since $\theta' \in B(\theta, \varepsilon') \subset B(\theta, \varepsilon_0)$, we have $u(\theta) = u(\theta')$. □

Proof of Proposition A.3.1

Proposition A.3.1. *If all coefficients of $\theta = (w, b)$ are strictly positive, then $(*)$ defined in Proposition 3.4.10 holds.*

Proof. Using Definitions 3.3.1, define $\mathbf{L} : \mathbb{R}^{E \cup H} \to \mathbb{R}^{\mathcal{P} \cup \mathcal{Q}}$ as $\mathbf{L} = (\mathbf{P}, \mathbf{Q}_b)$ where $\mathbf{Q}_b$ is the matrix $\mathbf{Q}$ multiplied column-wise by b. Then, $\mathbf{L}$ is such that $u(\theta) = \exp(\mathbf{L} \log \theta)$.

Next, we decompose $\log \theta = f(\theta) + g(\theta)$ where f is the orthogonal projection of $\log \theta$ on $\ker \mathbf{L}$ and $g(\theta) = \mathbf{L}^\dagger \mathbf{L} \log \theta$. We have $u(\theta) = \exp(\mathbf{L} g(\theta))$ and $g(\theta) = \mathbf{L}^\dagger \log u(\theta)$.

Let $\varepsilon > 0$. Consider $0 < \eta \leq \min_{q \in \mathcal{P} \cup \mathcal{Q}} u(\theta)_q$. For all θ' such that $\|u(\theta') - u(\theta)\| < \eta$, since $u(\theta')$ has all its coefficients that are strictly positive, we have $u(\theta) = u(|\theta'|)$. Hence, without loss of generality, we assume that all the coefficients of θ' are strictly positive. Then, we have

$$u(\theta') = \exp(\mathbf{L} g(\theta')) = \exp(\mathbf{L}(f(\theta) + g(\theta'))) = u(\theta'') \qquad \text{(A.168)}$$

where $\log \theta'' \triangleq f(\theta) + g(\theta')$. We have $u(\theta'') = u(\theta')$. Finally, let us show that if η is small enough, we have $\|\theta'' - \theta\| < \varepsilon$. To do that, we combine the fact that $\log \theta'' - \log \theta = g(\theta') - g(\theta) = \mathbf{L}^\dagger(u(\theta') - u(\theta))$ with the continuity of the exponential function and of $\mathbf{L}^\dagger$. □

Appendix B

Proofs and Supplementary Results for Equi-normalization

B.1 Illustration of the Effect of Equi-normalization

We apply ENorm to one randomly initialized fully connected network with 20 intermediary layers. All the layers have a size 500×500 and are initialized following the Xavier scheme. The network has been artificially unbalanced as follows: all the weights of layer 6 are multiplied by a factor 1.2 and all the weights of layer 12 are multiplied by 0.8, see Figure B-1. We then iterate our ENorm algorithm on the network, without training it, to see that it naturally re-balances the network, see Figure B-2.

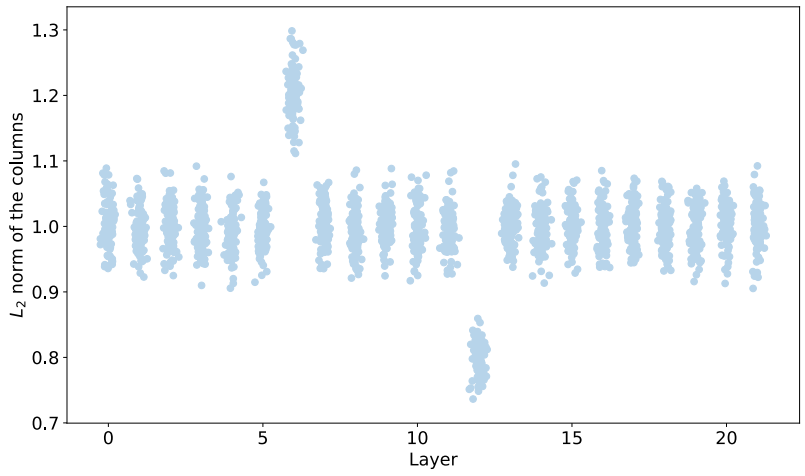


Figure B-1: Energy of the network (ℓ_2-norm of the weights), before ENorm. Each dot represents the norm of one column in the layer's weight matrix.

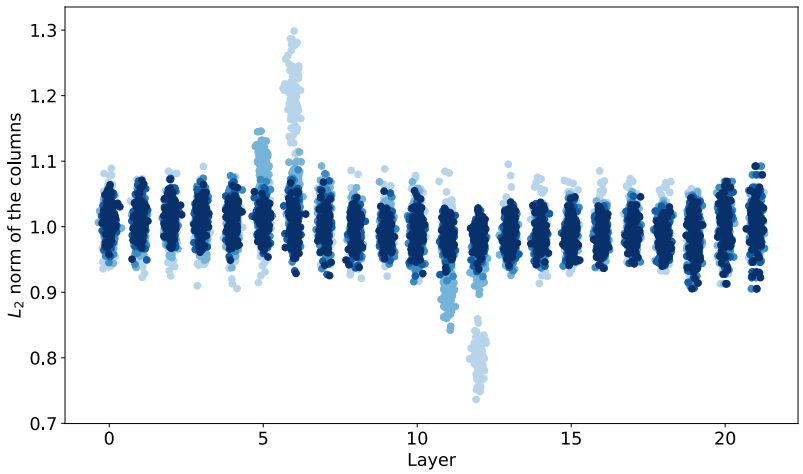


Figure B-2: Energy of the network through successive ENorm iterations (without training). One color denotes one iteration. The darker the color, the higher the iteration number.

B.1.1 Gradients & Biases

Denoting the rescaled weights and biases with a tilde, and denoting by $\mathcal{L}$ the loss of the network, for every layer $\ell \in [\![0, L]\!]$, we have

$$\frac{\partial \mathcal{L}}{\partial \widetilde{y}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial y^{(\ell)}} \left(D^{(\ell)}\right)^{-1}. \tag{B.1}$$

Similarly, we obtain

$$\frac{\partial \mathcal{L}}{\partial \widetilde{W}^{(\ell)}} = D^{(\ell-1)} \frac{\partial \mathcal{L}}{\partial W^{(\ell)}} \left(D^{\ell}\right)^{-1} \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \widetilde{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial b^{(\ell)}} \left(D^{(\ell)}\right)^{-1}. \tag{B.2}$$

Equation (B.2) will be used to update the momentum (see Section 4.5).

B.2 Proof of Convergence of Equi-normalization

We now prove Theorem 4.3.1. We use the framework of block coordinate descent and we first state a consequence of a theorem of Tseng (2001) [Theorem 4.1][1].

Theorem B.2.1. *Let $D \subset \mathbb{R}^n$ be an open set and $f : D \to \mathbb{R}$ a real function of B block variables $x_b \in \mathbb{R}^{n_b}$ with $\sum_{b=1}^{B} n_b = n$. Let $x^{(0)}$ be the starting point of the*

[1]Note that what Tseng denotes as *stationary point* in his paper is actually defined as a point where the directional derivative is positive along every possible direction, *i.e.* a local minimum.

coordinate descent algorithm and X the level set $X = \left\{x \mid f(x) \leq f\left(x^{(0)}\right)\right\}$. We make the following assumptions:

(1) f is differentiable on D ;

(2) X is compact ;

(3) for each $x \in D$, each block coordinate function $f_\ell : t \to f(x_1, \ldots, x_{\ell-1}, t, x_{\ell+1}, \ldots, x_B)$, where $2 \leq \ell \leq B-1$, has at most one minimum.

Under these assumptions, the sequence $(x^{(r)})_{r\in\mathbb{N}}$ generated by the coordinate descent algorithm is defined and bounded. Moreover, every cluster point of $(x^{(r)})_{r\in\mathbb{N}}$ is a local minimizer of f.

Step 1. We apply Theorem B.2.1 to the function φ. This is possible because all the assumptions are verified as shown below. Recall that

$$\varphi(d) = \sum_{\ell=1}^{L} \left\| \left(D^{(\ell-1)}\right)^{-1} W^{(\ell)} D^{(\ell)} \right\|_p^p = \sum_{\ell=1}^{L} \sum_{i,j} \left| \frac{d_\ell[j]}{d_{\ell-1}[i]} W^{(\ell)}[i,j] \right|^p . \tag{B.3}$$

Assumption (1). φ is differentiable on the open set $D = (0, +\infty)^n$.

Assumption (2). $\varphi \to +\infty$ when $\|d\| \to +\infty$. Let d such that $\varphi(d) < M^p$, $M > 1$. Let us show by induction that for all $\ell \in [\![1, L-1]\!]$, $\|d_\ell\|_\infty < (CM)^\ell$, where $C = \max(C_0, 1)$ and

$$C_0 = \max_{W^{(\ell)}[i,j]\neq 0} \left(\frac{1}{|W^{(\ell)}[i,j]|} \right) \tag{B.4}$$

- For the first hidden layer, index $\ell = 1$. By assumption, every hidden neuron is connected at least to one neuron in the input layer. Thus, for every j, there exists i such that $W^{(1)}[i,j] \neq 0$. Because $\varphi(d) < M^p$ and $d_0[i] = 1$ for all i,

$$(d_1[j])^p \left|W^{(1)}[i,j]\right|^p = \left(\frac{d_1[j]}{d_0[i]} \right)^p \left|W^{(1)}[i,j]\right|^p < M^p \tag{B.5}$$

  Thus $\|d_1\|_\infty < CM$.

- For any next hidden layer, index $\ell \in [\![2, L-1]\!]$. By assumption, every hidden neuron is connected at least to one neuron in the previous layer. Thus, for every j, there exists i such that $W^{(\ell)}[i,j] \neq 0$. Because $\varphi(d) < M$,

$$\left(\frac{d_k[j]}{d_{k-1}[i]} \right)^p \left|W^{(\ell)}[i,j]\right|^p < M^p \tag{B.6}$$

Using the induction hypothesis, we get $\|d_k\|_\infty < (CM)^\ell$.

Thus, $\|d\|_\infty < (MC)^L$ because $MC > 1$. By contraposition, $\varphi \to +\infty$.

Thus, there exists a ball B such that $d \notin B$ implies $\varphi(d) > \varphi(d^{(0)})$. Thus, by contraposition, $d \in X$ implies that $x \in B$. Thus, $X \subset B$ is bounded. Moreover, X is closed because φ is continuous thus X is compact. Then, Assumption (2) is satisfied.

Assumption (3). We next note that

$$\varphi_\ell(t) = \varphi\left(d_1^{(r)}, \dots, d_{\ell-1}^{(r)}, t, d_{\ell+1}^{(r)}, \dots, d_{L-1}^{(r)}\right) \tag{B.7}$$

has a unique minimum as shown in Section 4.3.3, see Equation (4.2). The existence and uniqueness of the minimum comes from the fact that each hidden neuron is connected to at least one input *and* one output neuron, thus all the row and column norms of the hidden weight matrices $W^{(\ell)}$ are non-zero, as well as the column (resp. row) norms or $W^{(1)}$ (resp. $W^{(L)}$).

Step 2. We prove that φ has at most one stationary point on D under the assumption that each hidden neuron is connected *either* to an input neuron *or* to an output neuron, which is weaker than the general admissibility assumption of Theorem 4.3.1 that assumes that each hidden neuron is connected to an input neuron *and* to an output neuron.

As defined in the Notations, we denote the set of all neurons in the network by V, the set of all hidden neurons by H. For each neuron ν, we define $\mathrm{prev}(\nu)$ as the neurons connected to ν that belong to the previous layer. We further denote by H the set of hidden neurons ν belonging to layers $\ell \in [\![1, L-1]\!]$. We define E_0 as the set of edges whose weights are non-zero, *i.e.*

$$E_0 = \{(\ell, i, j) \mid W_{i,j}^{(\ell)} \neq 0\}. \tag{B.8}$$

We now show that φ has at most one stationary point on D. Directly computing the gradient of φ and solving for zeros happens to be painful or even intractable. Thus, we define a change of variables as follows. We define h as

$$\begin{aligned} h\colon (0, +\infty)^H &\to \mathbb{R}^H \\ d &\mapsto \log(d) \end{aligned}$$

We next define the shift operator $S\colon \mathbb{R}^V \to \mathbb{R}^{E_0}$ such that, for every $x \in \mathbb{R}^V$,

$$S(x) = (\nu - \nu')_{\nu,\nu' \in V \text{ s.t. } \nu' \in \mathrm{prev}(\nu)}$$

and the padding operator P as

$$\begin{aligned} P\colon \mathbb{R}^H &\to \mathbb{R}^V \\ x &\mapsto y \text{ where } \begin{cases} y_\nu = 0 & \text{if } \nu \in V \backslash H; \\ y_\nu = x_\nu & \text{otherwise.} \end{cases} \end{aligned}$$

We define the extended shift operator $S_H = S \circ P$. We are now ready to define our change of variables. We define $\chi = \psi \circ S_H$ where

$$\begin{aligned} \psi\colon \mathbb{R}^{E_0} &\to \mathbb{R} \\ x &\mapsto \sum_{e \in E_0} \exp(p x_e) \, |w_e|^p \end{aligned}$$

and observe that

$$\varphi = \chi \circ h \tag{B.9}$$

so that its differential satisfies

$$[D\varphi](d) = [D\chi](h(d))[Dh](d). \tag{B.10}$$

Since h is a $\mathcal{C}^\infty$ diffeomorphism, its differential $[Dh](d)$ is invertible for any d. It follows that $[D\varphi](d) = 0$ if, and only if, $[D\chi](h(d)) = 0$. As χ is the composition of a strictly convex function, ψ, and a linear injective function, S_H (proof after Step 3), it is strictly convex. Thus χ has at most one stationary point, which concludes this step. Note the similarities of this step with Proposition 3.3.6 in Chapter 3.

Step 3. We prove that the sequence $d^{(r)}$ converges. Step 1 implies that the sequence $d^{(r)}$ is bounded and has at least one cluster point, as f is continuous on the compact X. Step 2 implies that the sequence $d^{(r)}$ has at most one cluster point. We then use the fact that any bounded sequence with exactly one cluster point converges to conclude the proof.

S is injective. Let $x \in \ker S_H$. Let us show by induction on the hidden layer index k that for every neuron ν at layer k, $x_\nu = 0$.

- $k = 1$. Let ν be a neuron at layer 1. Then, there exists a path coming from an input neuron to ν_0 through edge e_1. By definition, $P(x)_{\nu_0} = 0$ and $P(x)_\nu = x_\nu$, hence $S_H(x)_{e_1} = x_\nu - 0$. Since $S_H(x) = 0$ it follows that $x_\nu = 0$.
- $k \to k + 1$. Same reasoning using the fact that $x_{\nu_k} = 0$ by induction.

The case where the path goes from neuron ν to some output neuron is similar.

B.3 Extension of ENorm to CNNs

B.3.1 Convolutional Layers

Let us consider two consecutive convolutional layers ℓ and $\ell + 1$, without bias. Layer ℓ has C_ℓ filters of size $C_{\ell-1} \times S_\ell \times S_\ell$, where $C_{\ell-1}$ is the number of input features and S_ℓ is the kernel size. This results in a weight tensor T_ℓ of size $C_\ell \times C_{\ell-1} \times S_\ell \times S_\ell$. Similarly, layer $\ell + 1$ has a weight matrix $T_{\ell+1}$ of size $C_{\ell+1} \times C_\ell \times S_{\ell+1} \times S_{\ell+1}$. We then perform axis-permutation and reshaping operations to obtain the 2D matrices:

$$M_\ell \quad \text{of size } (C_{\ell-1} \times S_\ell \times S_\ell) \times C_\ell; \tag{B.11}$$

$$M_{\ell+1} \text{ of size } C_\ell \times (C_{\ell+1} \times S_{\ell+1} \times S_{\ell+1}). \tag{B.12}$$

For example, we first reshape T_ℓ as a 2D matrix by collapsing its last 3 dimensions, then transpose it to obtain M_ℓ. We then jointly rescale these 2D matrices using rescaling matrices $D_\ell \in \mathcal{D}(\ell)$ as detailed in Section 4.3 and perform the inverse axis permutation and reshaping operations to obtain a *right-rescaled* weight tensor $\widetilde{T}_\ell$ and a *left-rescaled* weight tensor $\widetilde{T}_{\ell+1}$. See Figure 4-2 for an illustration of the procedure. This matched rescaling does preserve the function implemented by the composition of the two layers, whether they are interleaved with a ReLU or not. It can be applied to any two consecutive convolutional layers with various stride and padding parameters. When the kernel size is 1 in both layers, we recover the fully-connected case of Figure 4-1.

B.3.2 Skip Connections

We now consider an elementary block of a ResNet-18 architecture as depicted in Figure 4-3. In order to maintain functional equivalence, we only consider ResNet architectures of type C as defined in (He et al., 2015a), where all shortcuts are learned 1×1 convolutions.

Structure of the rescaling process. Consider a ResNet block ℓ. We first left-rescale the *Conv1* and *ConvSkip* weights using the rescaling coefficients calculated between blocks $\ell - 1$ and ℓ. We then rescale the two consecutive layers *Conv1* and *Conv2* with their own rescaling coefficients, and finally right-rescale the *Conv2* and *ConvSkip* weights using the rescaling coefficients calculated between blocks ℓ and $\ell+1$.

Computation of the rescaling coefficients. Two types of rescaling coefficients are involved, namely these between two convolution layers inside the same block and these between two blocks. The rescaling coefficients between the *Conv1* and *Conv2* layers are calculated as explained in Section 4.4.1. Then, in order to calculate the rescaling coefficients between two blocks, we compute *equivalent block weights* to deduce rescaling coefficients.

We empirically explored some methods to compute the equivalent weight of a block using electrical network analogies. The most accurate method we found is to compute the equivalent weight of the *Conv1* and *Conv2* layers, *i.e.*, to express the succession of two convolution layers as only one convolution layer denoted as *ConvEquiv* (series equivalent weight), and in turn to express the two remaining parallel layers *ConvEquiv* and *ConvSkip* again as a single convolution layer (parallel equivalent weight). It is not possible to obtain series of equivalent weights, in particular when the convolution layers are interleaved with ReLUs. Therefore, we approximate the equivalent weight as the parallel equivalent weight of the *Conv1* and *ConvSkip* layers.

B.4 Implicit Equi-normalization

In Section 4.3, we defined an iterative algorithm that minimizes the global ℓ_p norm of the network

$$L_2(\theta, d) = \sum_{\ell=1}^{L} \left\| \left(D^{(\ell-1)}\right)^{-1} W^{(\ell)} D^{(\ell)} \right\|_p^p . \tag{B.13}$$

As detailed in Algorithm 2, we perform alternative SGD and ENorm steps during training. We now derive an implicit formulation of this algorithm that we call *Implicit Equi-normalization*. Let us fix $p = 2$. We denote by $\mathcal{C}(R_\theta(x), y)$ the cross-entropy loss for the training sample (x, y) and by $L_2(\theta, d)$ the weight decay regularizer (B.13). The loss function of the network writes

$$\mathcal{L}(\theta, d) = \mathcal{C}(R_\theta(x), y) + \lambda L_2(\theta, d) \tag{B.14}$$

where λ is a regularization parameter. We now consider both the weights *and* the rescaling coefficients as learnable parameters and we rely on automatic differentiation packages to compute the derivatives of $\mathcal{L}$ with respect to the weights and to the rescaling coefficients. We then train the network by performing iterative SGD steps and updating all the learnt parameters. By design, the derivative of $\mathcal{C}$ with respect to any rescaling coefficient is zero. Although the additional overhead of implicit ENorm is theoretically negligible, we observed an increase of the training time of a ResNet-18 by roughly 30% using PyTorch 4.0 (Paszke et al., 2017). We refer to Implicit Equi-normalization as ENorm-Impl and to Explicit Equi-normalization as ENorm.

We performed early experiments for the CIFAR10 fully-connected case. ENorm-Impl performs generally better than the baseline but does not outperform explicit ENorm, in particular when the network is deep. We follow the same experimental setup than previously, except that we additionally cross-validated λ. We also initialize all the rescaling coefficients to one. Recall that ENorm or ENorm denotes explicit Equi-normalization while ENorm-Impl denotes Implicit Equi-normalization. We did not investigate learning the weights and the rescaling coefficients at different speeds (*e.g.* with different learning rates or momentum). This may explain in part why ENorm-Impl generally underperforms ENorm in these early experiments.

B.5 Experiments

We perform sanity checks to verify our implementation and give additional results.

B.5.1 Sanity Checks

We apply our Equi-normalization algorithm to a ResNet architecture by integrating all the methods exposed in Section 4.4. We perform three sanity checks before proceeding to experiments. First, we randomly initialize a ResNet-18 and verify that it outputs the same probabilities before and after balancing. Second, we randomly initialize a ResNet-18 and perform successive ENorm cycles (without any training) and observe that the L_2 norm of the weights in the network is decreasing and then converging, as theoretically proven in Section 4.3, see Figure B-3.

We finally compare the evolution of the total ℓ_2 norm of the network when training it, with or without ENorm. We use the setup described in Subsection 4.6.2 and use $p = 3$ intermediary layers. The results are presented in Figure B-4. ENorm consistently leads to a lower energy level in the network.

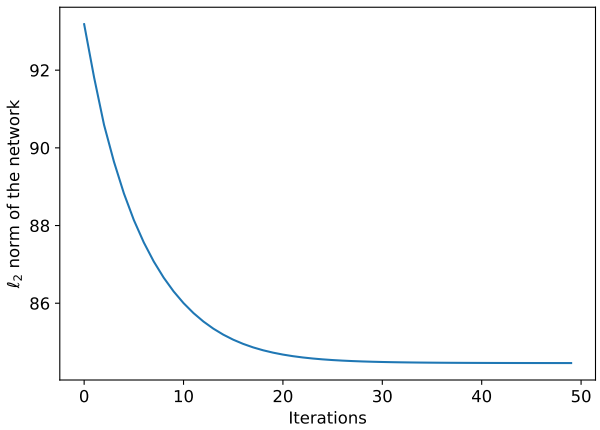


Figure B-3: ENorm cycles on a randomly initialized ResNet-18 with no training.

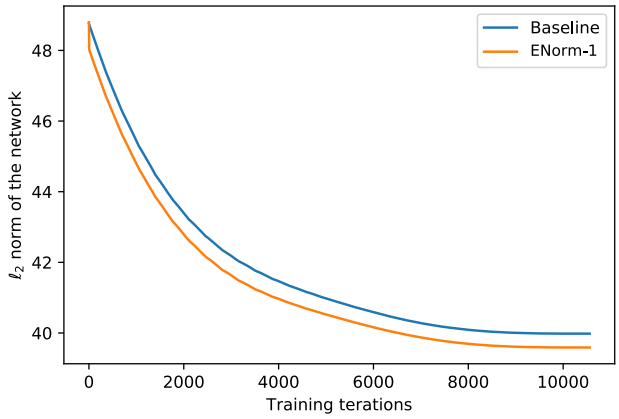


Figure B-4: Training a fully-connected network on CIFAR-10, with (ENorm-1) or without (Baseline) Equi-normalization.

B.5.2 Asymetric Scaling: Uniform vs. Adaptive

MNIST auto-encoder. For the uniform setup, we test for three different values of c, without BN: $c = 1$ (uniform setup), $c = 0.8$ (uniform setup), $c = 1.2$ (uniform setup). We also test the adaptive setup. The adaptive setup outperforms all other choices, which may be due to the strong bottleneck structure of the network. With BN, the dynamics are different and the results are much less sensitive to the values of c (see Figures B-5 and B-6).

CIFAR10 Fully Convolutional. For the uniform setup, we test for three different values of c, without BN: $c = 1$ (uniform setup), $c = 0.8$ (uniform setup), $c = 1.2$

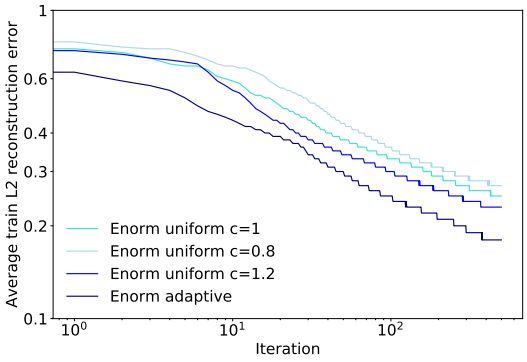


Figure B-5: Uniform vs adaptive scaling on MNIST, without BN.

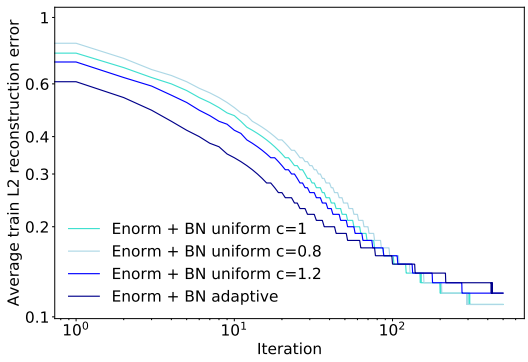


Figure B-6: Uniform vs adaptive scaling on MNIST, with BN.

Method	Test top 1 accuracy
ENorm uniform $c = 1$	86.98
ENorm uniform $c = 0.8$	79.88
ENorm uniform $c = 1.2$	89.31
ENorm adaptive	89.28
ENorm + BN uniform $c = 1$	91.85
ENorm + BN uniform $c = 0.8$	90.95
ENorm + BN uniform $c = 1.2$	90.89
ENorm + BN adaptive	90.79

Table B.1: Uniform vs adaptive scaling, CIFAR-10 fully convolutional

(uniform setup). We also test the adaptive setup (see Table B.1). Once again, the dynamics with or without BN are quite different. With or without BN, $c = 1.2$ performs the best, which may be linked to the fact that the ReLUs are cutting energy during each forward pass. With BN, the results are less sensitive to the values of c.

Appendix C

Supplementary Results for iPQ and Quant-Noise

C.1 Quantization of Additional Architectures

ResNet-50. We explore the compression of ResNet-50, a standard architecture used Computer Vision. In Table C.1, we compare Quant-Noise to iPQ Compression from Chapter 5 and show that Quant-Noise provide consistent additional improvement.

C.2 Ablations

In this section, we examine the impact of the level of noise during training as well as the impact of approximating iPQ during training.

C.2.1 Impact of Noise Rate

We analyze the performance for various values of Quant-Noise in Figure C-1 on a Transformer for language modeling. For iPQ, performance is impacted by high rates of quantization noise. For example, a Transformer with the noise function φ_{proxy} degrades with rate higher than 0.5, i.e., when half of the weights are passed through the noise function φ_{proxy}. We hypothesize that for large quantities of noise, a larger effect of using proxy rather than the exact PQ noise is observed. For `int8` quantization and its noise function, higher rates of noise are slightly worse but not as severe. A rate of 1 for `int8` quantization is equivalent to the Quantization Aware Training of (Krishnamoorthi, 2018), as the full matrix is quantized with STE, showing the potential benefit of partial quantization during training.

Setting	Model	Compression	Top-1 Accuracy
Small Blocks	Stock et al. (2019b)	19x	73.8
	Quant-Noise	19x	**74.3**
Large Blocks	Stock et al. (2019b)	32x	68.2
	Quant-Noise	32x	**68.8**

Table C.1: **Compression of ResNet-50 with Quant-Noise**. We compare to Stock et al. (2019b) in both the small and large blocks regime. For fair comparison, we hold the compression rate constant. Quant-Noise provides improved performance in both settings.

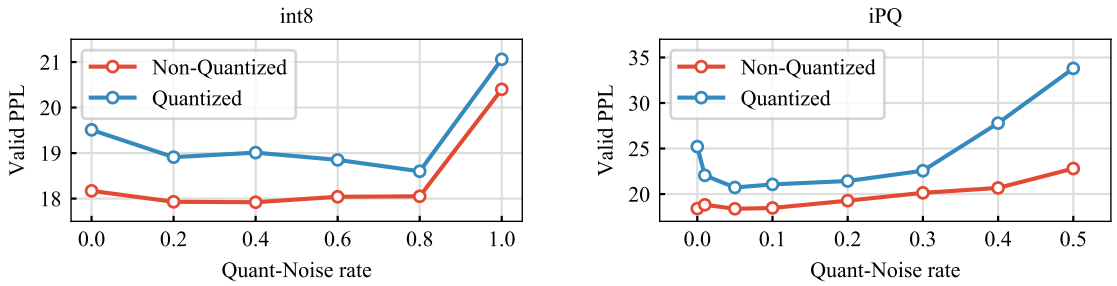


Figure C-1: **Effect of Quantization Parameters**. We report the influence of the proportion of blocks to which we apply the noise. We focus on Transformer for Wikitext-103 language modeling. We explore two settings: iPQ and `int8`. For iPQ, we use φ_{proxy}.

C.2.2 Impact of Approximating the Noise Function

We study the impact of approximating quantization noise during training. We focus on the case of iPQ with the approximation described in Section 6.4.2. In Table C.2, we compare the correct noise function for iPQ with its approximation φ_{proxy}. This approximate noise function does not consider cluster assignments or centroid values and simply zeroes out the selected blocks. For completeness, we include an intermediate approximation where we consider cluster assignments to apply noise within each cluster, but still zero-out the vectors. These approximations do not affect the performance of the quantized models. This suggests that increasing the correlation between subvectors that are jointly clustered is enough to maintain the performance of a model quantized with iPQ. Since PQ tends to work well on highly correlated vectors, such as activations in convolutional networks, this is not surprising. Using the approximation φ_{proxy} presents the advantage of speed and practicality. Indeed, one does not need to compute cluster assignments and centroids for every layer in the network after each epoch. Moreover, this approach is less involved in terms of code.

Noise	Blocks	PPL	Quant PPL
φ_{PQ}	Subvectors	18.3	21.1
φ_{PQ}	Clusters	18.3	21.2
φ_{proxy}	Subvectors	18.3	21.0
φ_{proxy}	Clusters	18.4	21.1

Table C.2: **Exact versus proxy noise function for different block selections with iPQ.** We compare exact ϕ_{PQ} and the approximation ϕ_{proxy} with blocks selected from all subvectors or subvectors from the same cluster.

C.3 Experimental Setting

We assess the effectiveness of Quant-Noise on competitive language and vision benchmarks. We consider Transformers for language modeling, RoBERTa for pre-training sentence representations, and EfficientNet for image classification. Our models are implemented in PyTorch (Paszke et al., 2017). We use `fairseq` (Ott et al., 2019) for language modeling and pre-training for sentence representation tasks and `Classy Vision` (Adcock et al., 2019) for EfficientNet.

Language Modeling. We experiment on the `Wikitext-103` benchmark (Merity et al., 2016) that contains 100M tokens and a vocabulary of 260k words. We train a 16 layer Transformer following Baevski and Auli (2018) with a LayerDrop rate of 0.2 (Fan et al., 2019). We report perplexity (PPL) on the test set.

Pre-Training of Sentence Representations. We pre-train the base BERT model (Devlin et al., 2018) on the `BooksCorpus + Wiki` dataset with a LayerDrop rate of 0.2. We finetune the pre-trained models on the MNLI task (Williams et al., 2018) from the GLUE Benchmark (Wang et al., 2019a) and report accuracy. We follow the parameters in Liu et al. (2019b) training and finetuning.

Image Classification. We train an EfficientNet-B3 model (Tan and Le, 2019) on the ImageNet object classification benchmark (Deng et al., 2009). The EfficientNet-B3 of `Classy Vision` achieves a Top-1 accuracy of 81.5%, which is slightly below than the performance of 81.9% reported by Tan and Le (2019).

C.3.1 Training Details

Language Modeling To handle the large vocabulary of Wikitext-103, we follow (Dauphin et al., 2017) and (Baevski and Auli, 2018). We use the adaptive softmax

(Grave et al., 2016) and adaptive input for computational efficiency. For both input and output embeddings, we use dimension size 1024 and three adaptive bands: 20K, 40K, and 200K. We use a cosine learning rate schedule (Baevski and Auli, 2018; Loshchilov and Hutter, 2016) and train with Nesterov's accelerated gradient (Sutskever et al., 2013). We set the momentum to 0.99 and renormalize gradients if the norm exceeds 0.1 (Pascanu et al., 2014). During training, we partition the data into blocks of contiguous tokens that ignore document boundaries. At test time, we respect sentence boundaries. We set LayerDrop to 0.2. We set Quant-Noise value to 0.05. During training time, we searched over the parameters (0.05, 0.1, 0.2) to determine the optimal value of Quant-Noise. During training, the block size is 8.

RoBERTa The base architecture is a 12 layer model with embedding size 768 and FFN size 3072. We follow (Liu et al., 2019b) in using the subword tokenization scheme from (Radford et al., 2019), which uses bytes as subword units. This eliminates unknown tokens. We train with large batches of size 8192 and maintain this batch size using gradient accumulation. We do not use next sentence prediction (Lample and Conneau, 2019). We optimize with Adam with a polynomial decay learning rate schedule. We set LayerDrop to 0.2. We set Quant-Noise value to 0.1. We did not hyperparameter search to determine the optimal value of Quant-Noise as training RoBERTa is computationally intensive. During training time, the block size of Quant-Noise is 8.

During finetuning, we hyperparameter search over three learning rate options (1e-5, 2e-5, 3e-5) and batchsize (16 or 32 sentences). The other parameters are set following (Liu et al., 2019b). We do single task finetuning, meaning we only tune on the data provided for the given natural language understanding task. We do not perform ensembling. When finetuning models trained with LayerDrop, we apply LayerDrop and Quant-Noise during finetuning time as well.

EfficientNet We use the architecture of EfficientNet-B3 defined in `Classy Vision` (Adcock et al., 2019) and follow the default hyperparameters for training. We set Quant-Noise value to 0.1. During training time, we searched over the parameters (0.05, 0.1, 0.2) to determine the optimal value of Quant-Noise. During training time, the block size of Quant-Noise is set to 4 for all 1×1 convolutions, 9 for depth-wise 3×3 convolutions, 5 for depth-wise 5×5 convolutions and 4 for the classifier. For sharing, we shared weights between blocks 9-10, 11-12, 14-15, 16-17, 19-20-21, 22-23 and refer to blocks that share the same weights as a *chunk*. For LayerDrop, we drop

Model	MB	PPL
Trans XL Large (Dai et al., 2019)	970	18.3
Compressive Trans (Rae et al., 2019)	970	17.1
GCNN (Dauphin et al., 2017)	870	37.2
4 Layer QRNN (Bradbury et al., 2016)	575	33.0
Trans XL Base (Dai et al., 2019)	570	24.0
Persis Mem (Sukhbaatar et al., 2019b)	506	20.6
Tensorized core-2 (Ma et al., 2019)	325	18.9
Quant-Noise	**38**	20.7
Quant-Noise + Share + Prune	10	24.2

Table C.3: **Performance on Wikitext-103.** We report test set perplexity and model size in megabytes. Lower perplexity is better.

the chunks of blocks defined previously with probability 0.2 and evaluate only with chunks 9-10, 14-15 and 19-20-21.

C.3.2 Scalar Quantization Details

We emulate scalar quantization by quantizing the weights and the activations. The scales and zero points of activations are determined by doing a few forward passes ahead of the evaluation and then fixed. We use the `Histogram` method to compute s and z, which aims at approximately minimizing the L_2 quantization error by adjusting s and z. This scheme is a refinement of the `MinMax` scheme. Per channel quantization is also discussed in Table C.7.

C.3.3 iPQ Quantization Details

Language Modeling We quantize FFN with block size 8, embeddings with block size 8, and attention with block size 4. We tuned the block size for attention between the values (4, 8) to find the best performance. Note that during training with apply Quant-Noise to all the layers.

RoBERTa We quantize FFN with block size 4, embeddings with block size 4, and attention with block size 4. We tuned the block size between the values (4, 8) to find the best performance. During training with apply Quant-Noise to all the layers.

EfficientNet We quantize blocks sequentially and end up with the classifier. The block sizes are 4 for all 1×1 convolutions, 9 for depth-wise 3×3 convolutions, 5

Model	MB	MNLI
RoBERTa Base + LD (Fan et al., 2019)	480	84.8
BERT Base (Devlin et al., 2018)	420	84.4
PreTrained Distil (Turc et al., 2019)	257	82.5
DistilBERT (Sanh et al., 2019)	250	81.8
MobileBERT* (Sun et al., 2020)	96	84.4
TinyBERT† (Jiao et al., 2019)	55	82.8
ALBERT Base (Lan et al., 2019)	45	81.6
AdaBERT† (Chen et al., 2020a)	36	81.6
Quant-Noise	38	83.6
Quant-Noise + Share + Prune	14	82.5

Table C.4: **Performance on MNLI.** We report accuracy and size in megabytes. * indicates distillation using BERT Large. † indicates training with data augmentation. Work from Sun et al. (2019) and Zhao et al. (2019) do not report results on the dev set. Cao et al. (2020) do not report model size. Higher accuracy is better.

for depth-wise 5×5 convolutions and 4 for the classifier. Note that during training with apply Quant-Noise to all the weights in InvertedResidual Blocks (except the Squeeze-Excitation subblocks), the head convolution and the classifier.

C.3.4 Details of Pruning and Layer Sharing

We apply the *Every Other Layer* strategy from Fan et al. (2019). When combining layer sharing with pruning, we train models with shared layers and then prune chunks of shared layers. When sharing layers, the weights of adjacent layers are shared in chunks of two. For a concrete example, imagine we have a model with layers A, B, C, D, E, F, G, H. We share layers A and B, C and D, E and F, G and H. To prune, every other chunk would be pruned away, for example we could prune A, B, E, F.

C.4 Numerical Results for Graphical Diagrams

We report the numerical values displayed in Figures 6-2 in Table C.3 for language modeling, Table C.4 for BERT, and Table C.5 for ImageNet.

Model	MB	Acc.
EfficientNet-B7 (Tan and Le, 2019)	260	84.4
ResNet-50 (He et al., 2015a)	97.5	76.1
DenseNet-169 (Huang et al., 2018a)	53.4	76.2
EfficientNet-B0 (Tan and Le, 2019)	20.2	77.3
MobileNet-v2 (Sandler et al., 2018a)	13.4	71.9
Shufflenet-v2 ×1 (Ma et al., 2018)	8.7	69.4
HAQ 4 bits (Wang et al., 2018a)	12.4	76.2
iPQ ResNet-50 (Stock et al., 2019b)	5.09	76.1
Quant-Noise	3.3	80.0
Quant-Noise + Share + Prune	2.3	77.8

Table C.5: **Performance on ImageNet.** We report accuracy and size in megabytes. Higher accuracy is better.

p	0	0.2	0.4	0.6	0.8	1
Top-1	80.66	80.83	80.82	80.88	80.92	80.64

Table C.6: **Effect of Quantization Parameters**. We report the influence of the Quant-Noise rate p with Scalar Quantization (`int8`). We focus on EfficientNet for ImageNet classification.

C.5 Further Ablations

C.5.1 Impact of Quant-Noise for the Vision setup

We provide another study showing the impact of the proportion of elements on which to apply Quant-Noise in Table C.6.

C.5.2 Impact of the number of centroids

We quantize with 256 centroids which represents a balance between size and representation capacity. The effect of the number of centroids on performance and size is shown in Figure C-2 (a). Quantizing with more centroids improves perplexity — this parameter could be adjusted based on the practical storage constraints.

C.5.3 Effect of Initial Model Size

Large, overparameterized models are more easily compressed. In Figure C-3, we explore quantizing both shallower and skinnier models. For shallow models, the

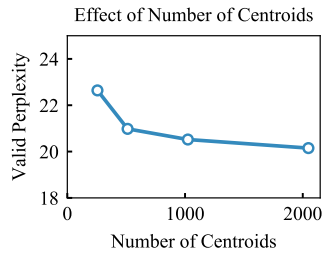


Figure C-2: **Quantizing with a larger number of centroids**. Results are shown on Wikitext-103 valid.

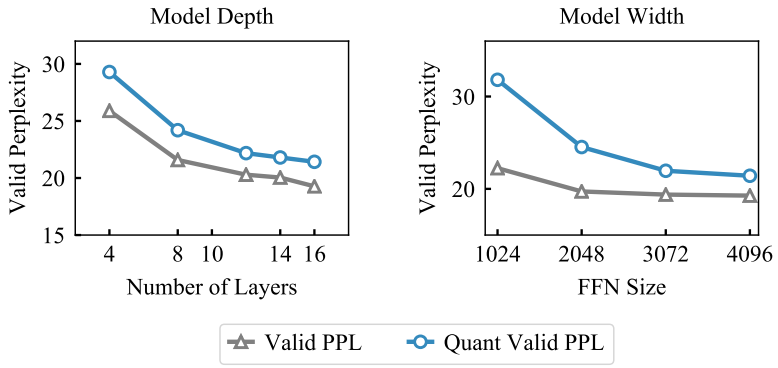


Figure C-3: **(a)** Effect of Initial Model Size for more shallow models **(b)** Effect of Initial Model Size more skinny models

gap between quantized and non-quantized perplexity does not increase as layers are removed (Figure C-3, left). In contrast, there is a larger gap in performance for models with smaller FFN (Figure C-3, right). As the FFN size decreases, the weights are less redundant and more difficult to quantize with iPQ.

C.5.4 Difficulty of Quantizing Different Model Structures

Quantization is applied to various portions of the Transformer architecture — the embedding, attention, feedforward, and classifier output. We compare the quantizability of various portions of the network in this section.

Is the order of structures important? We quantize specific network structures first — this is important as quantizing weight matrices can accumulate reconstruction error. Some structures of the network should be quantized last so the finetuning process can better adjust the centroids. We find that there are small variations in performance based on quantization order (see Figure C-4). We choose to quantize

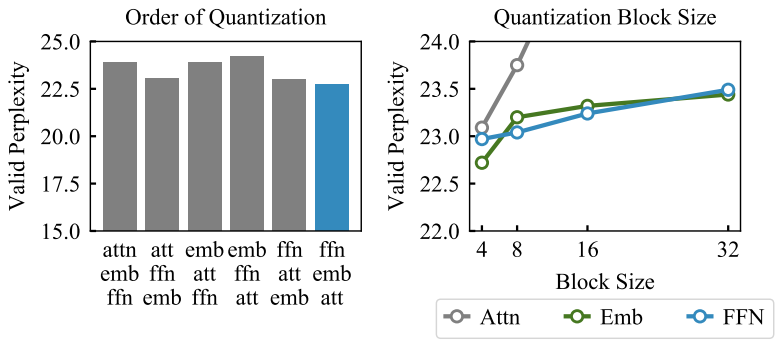


Figure C-4: **Effect of Quantization on Model Structures.** Results are shown on the validation set of Wikitext-103. **(a)** Quantizing Attention, FFN, and Embeddings in different order. **(b)** More Extreme compression of different structures.

FFN, then embeddings, and finally the attention matrices in Transformer networks.

Which structures can be compressed the most? Finally, we analyze which network structures can be most compressed. During quantization, various matrix block sizes can be chosen as a parameter — the larger the block size, the more compression, but also the larger the potential reduction of performance. Thus, it is important to understand how much each network structure can be compressed to reduce the memory footprint of the final model as much as possible. In Figure C-4, we quantize two model structures with a fixed block size and vary the block size of the third between 4 and 32. As shown, the FFN and embedding structures are more robust to aggressive compression, while the attention drastically loses performance as larger block sizes are used.

C.5.5 Approach to `intN` Scalar Quantization

We compare quantizing per-channel to using a histogram quantizer in Table C.7. The histogram quantizer maintains a running min/max and minimizes L2 distance between quantized and non-quantized values to find the optimal min/max. Quantizing per channel learns scales and offsets as vectors along the channel dimension, which provides more flexibility since scales and offsets can be different.

C.5.6 LayerDrop with STE

For quantization noise, we apply the straight through estimator (STE) to remaining weights in the backward pass. We experiment with applying STE to the backward

Quantization Scheme	**Language Modeling** 16-layer Transformer Wikitext-103			**Image Classification** EfficientNet-B3 ImageNet-1K		
	Size	Compress	Test PPL	Size	Compress	Top-1 Acc.
Uncompressed model	942	×1	18.3	46.7	×1	81.5
Int4 Quant Histogram	118	×8	39.4	5.8	×8	45.3
+ Quant-Noise	118	×8	21.8	5.8	×8	67.8
Int4 Quant Channel	118	×8	21.2	5.8	×8	68.2
+ Quant-Noise	118	×8	19.5	5.8	×8	72.3
Int8 Quant Histogram	236	×4	19.6	11.7	×4	80.7
+ Quant-Noise	236	×4	18.7	11.7	×4	80.9
Int8 Quant Channel	236	×4	18.5	11.7	×4	81.1
+ Quant-Noise	236	×4	18.3	11.7	×4	81.2

Table C.7: **Comparison of different approaches to `int4` and `int8` with and without Quant-Noise** on language modeling and image classification. For language modeling, we train a Transformer on the Wikitext-103 benchmark. We report perplexity (PPL) on the test set. For image classification, we train a EfficientNet-B3 on the ImageNet-1K benchmark. We report top-1 accuracy on the validation set. For both setting, we also report model size in megabyte (MB) and the compression ratio compared to the original model.

Model	**MB**	**PPL**
Quant-Noise + Share + Prune	10	24.2
Quant-Noise + Share + Prune with STE	10	24.5

Table C.8: **Performance on Wikitext-103 when using STE in the backward pass of the LayerDrop pruning noise.**

pass of LayerDrop's pruning noise. Results are shown in Table C.8 and find slightly worse results.

Appendix D

Supplementary Results for FaceGen

D.1 Additional Comparative Results

D.1.1 Quality Evaluation: Ablation Studies

For evaluation, we assembled 28 videos of diverse persons in terms of gender, age, skin color from the validation set of VoxCeleb2 (Chung et al., 2018), and a similar set of 50 videos from the validation split of the DFDC dataset (Dolhansky et al., 2019).

We begin our analysis of the FOM by computing the quality of reconstruction without first order motion approximation and with/without adversarial training in Table D.1. While it is clear that the adversarial fine-tuning boosts the performance, we experiment without it in the remaining of our ablation study around this model to reduce training time for each model. Removing the first order approximation only slightly degrades the LPIPS but not the msVGG perceptual metric. Interestingly, the CSIM metric which is the one supposed to best reflect the identify preservation, is slightly increased by dropping this component. A second observation is that the fidelity of facial landmarks to the target video is negatively affected by this removal. Since the drop of performance induced by discarding first order motion approximation leads to important bandwidth savings and limited loss in performance, we conduct our experiments without it. We refer to this approach as the Motion Net approach. Next, we explore the replacement of the self-supervised landmarks of the Motion Net approach by off-the-shelf landmarks from a state-of-the art detector. Results appear in Table D.2. Note that the results presented in this table are obtained by our re-implementation of the MotionNet approach, and are slightly better than these of

	FOM adv	FOM w/o adv	MN
msVGG ↓	**85.6**	87.5	87.9
LPIPS ↓	**0.226**	0.233	0.236
NME ↓	**0.51**	0.53	0.54
CSIM ↑	**0.83**	0.81	0.82

Table D.1: Ablation study for FOM on VoxCeleb2-28. MN: FOM without first order approximation nor adversarial fine-tuning.

	LPIPS ↓	NME ↓	CSIM ↑
Dense MN-10 U	0.221	0.59	0.83
Dense MN-20 L	0.242	0.50	0.80
Dense MN-68 L	0.240	0.49	0.81
Mob MN-10 U	0.225	0.52	0.79
Mob MN-20 L	0.244	0.48	0.78
Mob MN-10 U + 20 L	0.218	0.46	0.80
Mob M-SPADE-10 U	0.217	0.47	**0.81**
Mob M-SPADE-20 L	0.242	**0.44**	0.79
Mob M-SPADE-10 U + 20 L	**0.215**	0.46	**0.81**

Table D.2: Evaluation results for Motion Net approaches without adversarial fine-tuning on the VoxCeleb2-28 video subset. Mob : Mobile models. Dense models (64×64 latent space) are trained on VoxCeleb. Mobile models (32×32) are trained on the DFDC aligned dataset. U: unsupervised keypoints; L: facial landmarks.

Table D.1 obtained with the original code.

We compare in Table D.2 different variants of the Motion Net approach, using 20 input landmarks, 68 input landmarks, self-supervised landmarks with dense architectures and with mobile architectures. All these dense architecture employ a latent space of $256 \times 64 \times 64$, and were trained on VoxCeleb. We first note that using standard facial landmarks instead of unsupervised motion landmarks degrades the scores of all metrics. Using 68 landmarks is not improving the quality over 20. With mobile architectures, we reduce the latent space to $256 \times 32 \times 32$. In addition to using 10 motion landmarks or 20 landmarks alone, combining these two sets helps boost all quality metrics. Finally, we observe that adding the SPADE blocks preserves the perceptual quality and brings a large improvement in NME.

	NTH	Bilayer	SegFace	FOM	MS20L
msVGG* ↓	**56.3**	68.6	84.4	58.7	57.9
LPIPS* ↓	0.165	0.200	0.304	**0.153**	0.167
NME* ↓	**0.38**	0.55	0.55	0.50	0.44
CSIM* ↑	0.83	0.85	0.76	**0.87**	0.84
kbits/s ↓	9.7	9.7	18	**4.0**	8.8

Table D.3: Comparison of Bilayer, SegFace (48x48), FOM adv, NTH in terms of quality / bandwidth (kbits/s with 25 fps) trade-offs on VoxCeleb2-28. * : Metrics were computed using ground truth backgrounds. We also include our best mobile model, Motion-SPADE-20L (MS20L) in the comparison.

D.1.2 Quantitative Comparative Evaluation

We compare the quality/bandwidth trade-off of different dense face animation approaches in Table D.3. As the different models were trained using different data pre-processing (different crops, alignment), we evaluate each one in the setting allowing the best generations. This means that synthesized videos need to be compared to different source videos. Therefore, we paste the ground truth video background on the generated result so that the metrics focus on evaluating face differences only. We observe that the NTH results lead to better NME, and FOM to better LPIPS and CSIM metrics. SegFace numerical results lower, partly due to a color shift appearing in the results. Interestingly, our mobile results have better NME and msVGG scores than the original FOM dense approach.

Figure D-1: Qualitative results using Motion based variants on mobile architectures, using a $32 \times 32 \times 256$ latent space. The model generates the face given the fixed source frame and the landmarks of the target frame. All models run in real-time on an iPhone 8.

Bibliography

A. Adcock, V. Reis, M. Singh, Z. Yan, L. van der Maaten, K. Zhang, S. Motwani, J. Guerin, N. Goyal, I. Misra, L. Gustafson, C. Changhan, and P. Goyal. Classy vision. 2019.

Eirikur Agustsson, Michael Tschannen, Fabian Mentzer, Radu Timofte, and Luc Van Gool. Generative adversarial networks for extreme learned image compression. In *ICCV*, 2019.

N. Ailon and B. Chazelle. Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform. *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, 2006.

Francesca Albertini, Eduardo D. Sontag, and Vincent Maillot. Uniqueness of weights for neural networks. In *Artificial Neural Networks with Applications in Speech and Vision*, 1993.

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *ICML*, 2019.

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural Comput.*, 1998.

Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. *ICML*, 2015.

Devansh Arpit, Yingbo Zhou, Bhargava U. Kota, and Venu Govindaraju. Normalization propagation: A parametric technique for removing internal covariate shift in deep networks. *ICML*, 2016.

Hadar Averbuch-Elor, Daniel Cohen-Or, Johannes Kopf, and Michael F Cohen. Bringing portraits to life. *Transactions on Graphics*, 2017.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? *arXiv preprint arXiv:1312.6184*, 2013.

Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018.

David Balduzzi, Marcus Frean, Lennox Leary, JP Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. The shattered gradients problem: If resnets are the answer, then what is the question? *ICML*, 2018.

Peter L. Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *JMLR*, 2003.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine learning practice and the bias-variance trade-off. *PNAS*, 2018.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 1994.

Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2009.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Julius Berner, Dennis Elbrächter, and Philipp Grohs. How degenerate is the parametrization of neural networks with the relu activation function? *NeurIPS*, 2019.

Jeremy Bernstein, Jiawei Zhao, Markus Meister, Ming-Yu Liu, Anima Anandkumar, and Yisong Yue. Learning compositional functions via multiplicative weight updates. *NeurIPS*, 2020.

Luca Bertinetto, Jack Valmadre, João F. Henriques, Andrea Vedaldi, and Philip H. S. Torr. Fully-convolutional siamese networks for object tracking. *arXiv preprint arXiv:1606.09549*, 2016.

Yash Bhalgat, Jinwon Lee, Markus Nagel, Tijmen Blankevoort, and Nojun Kwak. Lsq+: Improving low-bit quantization through learnable offsets and better initialization. *arXiv preprint arXiv:2004.09576*, 2020.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *MLSys*, 2020.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*, 2010.

Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM*, 2016.

James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, 1993.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *NeurIPS*, 2020.

Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2006.

Adrian Bulat and Georgios Tzimiropoulos. How far are we from solving the 2d & 3d face alignment problem?(and a dataset of 230,000 3d facial landmarks). In *ICCV*, 2017.

Adrian Bulat and Georgios Tzimiropoulos. Super-fan: Integrated facial landmark localization and super-resolution of real-world low resolution faces in arbitrary poses with gans. In *CVPR*, 2018.

Chen Cao, Hongzhi Wu, Yanlin Weng, Tianjia Shao, and Kun Zhou. Real-time facial animation with image-based dynamic avatars. *Transactions on Graphics*, 35(4), 2016.

Qingqing Cao, Harsh Trivedi, Aruna Balasubramanian, and Niranjan Balasubramanian. Faster and just as accurate: A simple decomposition for transformer models. *arXiv preprint arXiv:2010.03688*, 2020.

Qiong Cao, Li Shen, Weidi Xie, Omkar M Parkhi, and Andrew Zisserman. Vggface2: A dataset for recognising faces across pose and age. In *International Conference on Automatic Face & Gesture Recognition*, 2018.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. *ECCV*, 2020.

Mathilde Caron, Ari Morcos, Piotr Bojanowski, Julien Mairal, and Armand Joulin. Pruning convolutional neural networks with self-supervision. *arXiv preprint arXiv:2001.03554*, 2020.

Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? A new model and the kinetics dataset. *CoRR*, 2017.

Miguel A. Carreira-Perpiñán and Yerlan Idelbayev. Model compression as constrained optimization, with application to neural nets. part ii: quantization. *arXiv preprint arXiv:1707.04319*, 2017.

Daoyuan Chen, Yaliang Li, Minghui Qiu, Zhen Wang, Bofang Li, Bolin Ding, Hongbo Deng, Jun Huang, Wei Lin, and Jingren Zhou. Adabert: Task-adaptive bert compression with differentiable neural architecture search. *arXiv preprint arXiv:2001.04246*, 2020a.

Lele Chen, Guofeng Cui, Ziyi Kou, Haitian Zheng, and Chenliang Xu. What comprises a good talking-head video generation?: A survey and benchmark. *arXiv preprint arXiv:2005.03201*, 2020b.

Yu Chen, Ying Tai, Xiaoming Liu, Chunhua Shen, and Jian Yang. Fsrnet: End-to-end learning face super-resolution with facial priors. In *CVPR*, 2018.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, 2017.

Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. *CoRR*, 2016.

Anna Choromanska, Krzysztof Choromanski, Mariusz Bojarski, Tony Jebara, Sanjiv Kumar, and Yann LeCun. Binary embeddings with structured hashed projections. *JMLR*, 2015.

Joon Son Chung, Arsha Nagrani, and Andrew Zisserman. Voxceleb2: Deep speaker recognition. In *INTERSPEECH*, 2018.

Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval networks: Improving robustness to adversarial examples. *ICML*, 2017.

Dave Citron. "four new google duo features to help you stay connected", April 2020. URL `https://blog.google/products/duo/4-new-google-duo-features-help-you-stay-connected/`.

Nadav Cohen and Amnon Shashua. Convolutional rectifier networks as generalized tensor decompositions. *JMLR*, 2016.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *JMLR*, 2011.

J. Cooley, P. Lewis, and P. Welch. Historical notes on the fast fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 1967.

Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, 2016.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, 2015.

Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. *CVPR*, 2018.

Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical automated data augmentation with a reduced search space. *CVPR*, 2019.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 1989.

Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, et al. Fbnetv3: Joint architecture-recipe search using neural acquisition function. *arXiv preprint arXiv 2006.02049*, 2020.

Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

C. F. Dantas, M. N. da Costa, and R. d. R. Lopes. Learning dictionaries as a sum of kronecker products. *IEEE Signal Processing Letters*, 2017.

C. F. Dantas, J. E. Cohen, and R. Gribonval. Learning tensor-structured dictionaries with application to hyperspectral image denoising. In *EUSIPCO*, 2019.

Stéphane d'Ascoli, Maria Refinetti, Giulio Biroli, and Florent Krzakala. Double trouble in double descent : Bias and variance(s) in the lazy regime. *ICML*, 2020.

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *ICML*, 2017.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *ICLR*, 2019.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In *CVPR*, 2019.

Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning. *NIPS*, 2013.

Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*. 2014.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Brian Dolhansky, Joanna Bitton, Ben Pflaum, Jikuo Lu, Russ Howes, Menglin Wang, and Cristian Canton Ferrer. The deepfake detection challenge (dfdc) dataset. *arXiv preprint arXiv:2006.07397*, 2019.

Yinpeng Dong, Renkun Ni, Jianguo Li, Yurong Chen, Hang Su, and Jun Zhu. Stochastic quantization for learning accurate low-bit deep neural networks. *International Journal of Computer Vision*, 2019.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 2011.

Lukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. BRP-NAS: Prediction-based NAS using GCNs. *NeurIPS*, 2020.

Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. *ICML*, 2015.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. *ICML*, 2019.

Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *ICLR*, 2019.

Charles Fefferman. Reconstructing a neural net from its output. *Revista Matemática Iberoamericana*, 1994.

Michael Figurnov, Maxwell D. Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. *CVPR*, 2016.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.

Drew Fudenberg and Jean Tirole. *Game Theory*. MIT Press, 1991.

Oran Gafni, Lior Wolf, and Yaniv Taigman. Live face de-identification in video. In *CVPR*, 2019.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *ICML*, 2016.

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

Bill Gates and John Ottavino. *Road Ahead*. HighBridge Company, 1995. ISBN 0453009204.

Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2014.

Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. Dropblock: A regularization method for convolutional networks. In *Advances in Neural Information Processing Systems*, pages 10727–10737, 2018.

Igor Gitman and Boris Ginsburg. Comparison of batch normalization and weight normalization algorithms for the large-scale image classification. *CoRR*, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

Shachar Gluska and Mark Grobman. Exploring neural networks quantization via layer-wise quantization analysis. *arXiv preprint arXiv:2012.08420*, 2020.

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Edouard Grave, Armand Joulin, Moustapha Cisse, David Grangier, and Herve Jegou. Efficient softmax approximation for gpus. *arXiv preprint arXiv:1609.04309*, 2016.

Rémi Gribonval, Gitta Kutyniok, Morten Nielsen, and Felix Voigtlaender. Approximation spaces of deep neural networks. 2019.

Yunhui Guo. A survey on methods and theories of quantized neural networks. *CoRR*, 2018.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, 2015.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NIPS*, pages 1135–1143, 2015.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH*, 2016a.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *ICLR*, 2016b.

Boris Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In *NIPS*, 2018.

Boris Hanin and David Rolnick. How to start training: The effect of initialization and architecture. *NIPS*, 2018.

Boris Hanin and David Rolnick. Deep relu networks have surprisingly few activation patterns. *NIPS*, 2019.

Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*. 1993.

T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 2015a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proc. of CVPR*, 2015b.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CVPR*, 2015c.

Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. Mask R-CNN. *ICCV*, 2017a.

Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. *CoRR*, abs/1808.06866, 2018a.

Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *CVPR*, 2017b.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. *CVPR*, 2018b.

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012a.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012b.

Sepp Hochreiter and Yoshua Bengio. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In *IEEE*, 2001.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 1997.

Sara Hooker. The hardware lottery. *arXiv preprint arXIv:2009.06489*, 2020.

Sara Hooker, Aaron Courville, Gregory Clark, Yann Dauphin, and Andrea Frome. What do compressed deep neural networks forget? *arxiv preprint arXiv:1911.05248*, 2020a.

Sara Hooker, Nyalleng Moorosi, Gregory Clark, Samy Bengio, and Emily Denton. Characterising bias in compressed models. *arXiv preprint arXIv:2010.03058*, 2020b.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 1991.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *arXiv e-prints*, 2019.

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

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Conference on Computer Vision and Pattern Recognition*, 2018.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017a.

Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *CVPR*, 2018a.

Lei Huang, Xianglong Liu, Bo Lang, and Bo Li. Projection based weight normalization for deep neural networks. *arXiv preprint arXiv:1710.02338*, 2017b.

Qiangui Huang, Shaohua Kevin Zhou, Suya You, and Ulrich Neumann. Learning to prune filters in convolutional neural networks. *CoRR*, abs/1801.07365, 2018b.

Xun Huang and Serge Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. In *ICCV*, 2017.

David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 1952.

Forrest Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size. *CoRR*, 2016.

Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. *CoRR*, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, 2015.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*, 2018.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *BMVA*, 2014.

Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *NeurIPS*, 2015.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *PAMI*, 2011.

Yongkweon Jeon, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Jeongin Yun, and Dongsoo Lee. BiQGEMM: Matrix multiplication with lookup table for binary-coding-based quantized DNNs. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling BERT for natural language understanding. *ACL*, 2019.

William B. Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Conference in modern analysis and probability*, 1984.

Cijo Jose, Moustapha Cisse, and Francois Fleuret. Kronecker recurrent units. 2018.

Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Kathryn Tunyasuvunakool, Olaf Ronneberger, Russ Bates, Augustin Žídek, Alex Bridgland, Clemens Meyer, Simon Kohl, Anna Potapenko, Andrew Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas

Adler, Trevor Back, Stig Petersen, David Reiman, Martin Steinegger, Michalina Pacholska, David Silver, Oriol Vinyals, Andrew Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. High accuracy protein structure prediction using deep learning. In *Fourteenth Critical Assessment of Techniques for Protein Structure Prediction (Abstract Book)*, 2020.

Paul Kainen, Vera Kurková, Vladik Kreinovich, and Ongard Sirisengtaksin. Uniqueness of network parameterizations and faster learning. *Preprint*, 1994.

Hyeongwoo Kim, Pablo Garrido, Ayush Tewari, Weipeng Xu, Justus Thies, Matthias Niessner, Patrick Pérez, Christian Richardt, Michael Zollhöfer, and Christian Theobalt. Deep video portraits. *Transactions on Graphics*, 2018.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2014.

B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXIv:1610.05492*, 2017.

Iryna Korshunova, Wenzhe Shi, Joni Dambre, and Lucas Theis. Fast face-swap using convolutional neural networks. In *ICCV*, 2017.

Jean Kossaifi, Adrian Bulat, Georgios Tzimiropoulos, and Maja Pantic. T-Net: Parametrizing fully convolutional nets with a single high-order tensor. *CVPR*, 2019.

Ioannis Koufakis and Bernard F Buxton. Very low bit rate face video compression using linear combination of 2D face views and principal components analysis. *Image and Vision computing*, 1999.

Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 2012.

Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems*. 1992.

Věra Kůrková and Paul C. Kainen. Functionally equivalent feedforward neural networks. *Neural Comput.*, 1993.

H. T. Kung, Bradley McDanel, and Sai Qian Zhang. Term revealing: Furthering quantization at run time on quantized DNNs, 2020.

Jean Lafond, Nicolas Vasilache, and Léon Bottou. Diagonal rescaling for neural networks. *CoRR*, 2017.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *ICLR*, 2019.

Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

Guillaume Lample, Neil Zeghidour, Nicolas Usunier, Antoine Bordes, Ludovic Denoyer, and Marc'Aurelio Ranzato. Fader networks: Manipulating images by sliding attributes. *NIPS*, 2017.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *ICLR*, 2019.

Quoc Viet Le, Tamas Sarlos, and Alexander Johannes Smola. Fastfood: Approximate kernel expansions in loglinear time. *ICML*, 2014.

Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. 2014.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1989.

Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *NIPS*, 1990.

Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Springer-Verlag, 1998a.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998b.

Cheng-Han Lee, Ziwei Liu, Lingyun Wu, and Ping Luo. Maskgan: Towards diverse and interactive facial image manipulation. In *CVPR*, 2020.

Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 1993.

Fengfu Li and Bin Liu. Ternary weight networks. *CoRR*, 2016.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Muyang Li, Ji Lin, Yaoyao Ding, Zhijian Liu, Jun-Yan Zhu, and Song Han. Gan compression: Efficient architectures for interactive conditional GANs. 2020.

Yuanzhi Li and Yingyu Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. 2018.

Yuhang Li, Xin Dong, and Wei Wang. Additive powers-of-two quantization: A non-uniform discretization for neural networks. *arXiv preprint arXiv:1909.13144*, 2019.

Xiaocong Lian, Zhenyu Liu, Zhourui Song, Jiwu Dai, Wei Zhou, and Xiangyang Ji. High-performance FPGA-based CNN accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP, 2019.

Tsung-Yi Lin, Piotr Dollar, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017a.

Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. *CoRR*, 2017b.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *CVPR*, 2017a.

Dong Liu, Yue Li, Jianping Lin, Houqiang Li, and Feng Wu. Deep learning-based video coding: A review and a case study. *Computing Surveys*, 2020.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. *ICLR*, 2019a.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019b.

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. *International Conference on Computer Vision*, 2017b.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *ICCV*, 2015.

Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 1982.

Raphael Gontijo Lopes, Stefano Fenu, and Thad Starner. Data-free knowledge distillation for deep neural networks, 2017.

Ricardo Lopez and Thomas S Huang. Head pose computation for very low bit-rate video coding. *International Conference on Computer Analysis of Images and Patterns*, 1995.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. *NIPS*, 2017a.

Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017b.

Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *ICCV*, 2017.

Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. *NIPS*, 2018.

Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet V2: practical guidelines for efficient CNN architecture design. *CoRR*, 2018.

Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Dawei Song, and Ming Zhou. A tensorized transformer for language modeling. *arXiv preprint arXiv:1906.09777*, 2019.

James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, 1967.

Dhruv Mahajan, Ross B. Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. *CoRR*, 2018.

Francois Malgouyres. On the stable recovery of deep structured linear networks under sparsity constraints. *Proceedings of Machine Learning Research*, 2020.

Francois Malgouyres and Joseph Landsberg. Multilinear compressive sensing and an application to convolutional linear networks. 2018.

Gaétan Marceau-Caron and Yann Ollivier. Practical riemannian neural networks. *CoRR*, 2016.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Julieta Martinez, Jashan Shewakramani, Ting Wei Liu, Ioan Andrei Bârsan, Wenyuan Zeng, and Raquel Urtasun. Permute, quantize, and fine-tune: Efficient compression of neural networks, 2020.

Mark D. McDonnell. Training wide residual networks for deployment using a single bit for each weight. *ICLR*, 2018.

Song Mei and Andrea Montanari. The generalization error of random features regression: Precise asymptotics and double descent curve. *arXiv preprint arXiv:1908.05355*, 2019.

Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. Same, same but different - recovering neural network quantization error through weight factorization. *ICML*, 2019.

Qi Meng, Shuxin Zheng, Huishuai Zhang, Wei Chen, Zhi-Ming Ma, and Tie-Yan Liu. $\mathcal{G}$-SGD: Optimizing relu neural networks in its positively scale-invariant space. *arXiv preprint arXiv:1802.03713*, 2018.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models. *arXiv*, abs/1609.07843, 2016.

Hrushikesh Mhaskar and Tomaso Poggio. Deep vs. shallow networks : An approximation theory perspective. *arXiv preprint arXIv:1608.03287*, 2016.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2018.

Dmytro Mishkin and Jiri Matas. All you need is a good init. *ICLR*, 2016.

Asit K. Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. *CoRR*, 2017.

Asit K. Mishra, Eriko Nurvitadhi, Jeffrey J. Cook, and Debbie Marr. WRPN: wide reduced-precision networks. *CoRR*, 2017.

Deepak Mittal, Shweta Bhardwaj, Mitesh M Khapra, and Balaraman Ravindran. Recovering from random pruning: On the plasticity of deep convolutional neural networks. In *WACV*, 2018.

Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. ACDC: A structured efficient linear layer. *arXiv preprint arXiv:1511.05946*, 2015.

Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *ICML*, 2017.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.

Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *NIPS*, 2014.

Koki Nagano, Jaewoo Seo, Jun Xing, Lingyu Wei, Zimo Li, Shunsuke Saito, Aviral Agarwal, Jens Fursund, and Hao Li. paGAN: real-time avatars using dynamic textures. In *SIGGRAPH Asia*, 2018.

Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. *CVPR*, 2019.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. *arXiv preprint arXiv:1912.02292*, 2019.

Y. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 1983.

Behnam Neyshabur, Ruslan Salakhutdinov, and Nathan Srebro. Path-SGD: Path-normalized optimization in deep neural networks. *arXiv preprint arXiv:1506.02617*, 2015.

Yuval Nirkin, Yosi Keller, and Tal Hassner. FSGAN: Subject agnostic face swapping and reenactment. In *CVPR*, 2019.

Mohammad Norouzi and David J Fleet. Cartesian k-means. In *Conference on Computer Vision and Pattern Recognition*, 2013.

Yann Ollivier. Riemannian metrics for neural networks I: feedforward networks. *arXiv preprint arXiv:1303.0818*, 2015.

Maxime Oquab, Pierre Stock, Oran Gafni, Daniel Haziza, Tao Xu, Peizhao Zhang, Onur Celebi, Yana Hasson, Patrick Labatut, Bobo Bose-Kolanu, Thibault Peyronel, and Camille Couprie. Low bandwidth video-chat compression using deep generative models. *arXiv preprint arXiv:2012.00328*, 2020.

Ivan Oseledets. Tensor-train decomposition. *SIAM J. Scientific Computing*, 2011.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic image synthesis with spatially-adaptive normalization. *CVPR*, 2019.

Razvan Pascanu and Yoshua Bengio. Natural gradient revisited. *CoRR*, 2013.

Razvan Pascanu, Guido Montufar, and Yoshua Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. 2013.

Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*, 2014.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NIPS*, 2017.

Philipp Petersen, Mones Raslan, and Felix Voigtlaender. Topological properties of the set of functions generated by neural networks of fixed size. *arXiv preprint arXiv:1806.08459*, 2018.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *ICML*, 2018.

Mary Phuong and Christoph H Lampert. Functional vs. parametric equivalence of relu networks. In *International Conference on Learning Representations*, 2019.

Ofir Press, Noah A. Smith, and Omer Levy. Improving transformer models by reordering their sublayers. *ACL*, 2020.

Haozhi Qi, Chong You, Xiaolong Wang, Yi Ma, and Jitendra Malik. Deep isometric learning for visual recognition. *ICML*, 2020.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *Preprint*, 2019.

Ilija Radosavovic, Justin Johnson, Saining Xie, Wan-Yen Lo, and Piotr Dollár. On network design spaces for visual recognition. *CVPR*, 2019.

Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces. *CVPR*, 2020.

Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

J. Rapin and O. Teytaud. Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad, 2018.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '20, 2020.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *NIPS*, 2015.

Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander Anderson, and Lubomir Bourdev. Learned video compression. In *ICCV*, 2019.

Jorma Rissanen and Glen G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979. doi: 10.1147/rd.232.0149.

David Rolnick and Konrad P. Kording. Reverse-engineering deep relu networks. *ICML*, 2019.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

Sebastian Ruder. NLP's ImageNet moment has arrived. https://physicstoday.scitation.org/doi/10.1063/PT.3.4164, 2019.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, chapter 8. 1986.

Levent Sagun, Utku Evci, V. Ugur Guney, Yann Dauphin, and Leon Bottou. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv preprint arXiv:1706.04454*, 2018.

Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *CoRR*, 2016.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018a.

Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, 2018b.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Karthik A. Sankararaman, Soham De, Zheng Xu, W. Ronny Huang, and Tom Goldstein. The impact of neural network overparameterization on gradient confusion and stochastic gradient descent. *ICML*, 2019.

Shibani Santurkar, David Budden, and Nir Shavit. Generative compression. In *Picture Coding Symposium*, 2018a.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? (no, it is not about internal covariate shift). *CoRR*, 2018b.

Kristof Schütt, Farhad Arbabzadah, Stefan Chmiela, Klaus-Robert Müller, and Alexandre Tkatchenko. Quantum-chemical insights from deep tensor neural networks. *Nature Communications*, 8, 01 2017. doi: 10.1038/ncomms13890.

Oran Shayer, Dan Levi, and Ethan Fetaya. Learning discrete weights using the local reparameterization trick. *CoRR*, 2017.

Aliaksandr Siarohin, Stéphane Lathuilière, Sergey Tulyakov, Elisa Ricci, and Nicu Sebe. First order motion model for image animation. In *NeurIPS*, 2019.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 2017.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Vikas Sindhwani, Tara N. Sainath, and Sanjiv Kumar. Structured transforms for small-footprint deep learning. *NIPS*, 2015.

Ulrik Söderström. *Very low bitrate facial video coding: based on principal component analysis*. PhD thesis, Tillämpad fysik och elektronik, 2006.

Le-Hung Son, Ulrik Söderström, and Haibo Li. Ultra low bit-rate video communication, 2006.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 15(1):1929–1958, 2014.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

Hugo Steinhaus. Sur la division des corps materiels en parties. *Springer*, 1956.

Pierre Stock and Moustapha Cisse. Convnets and imagenet beyond accuracy: Understanding mistakes and uncovering biases. *ECCV*, 2018.

Pierre Stock, Benjamin Graham, Rémi Gribonval, and Hervé Jégou. Equi-normalization of neural networks. *International Conference on Learning Representations (ICLR)*, 2019a.

Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the bit goes down: Revisiting the quantization of neural networks. *ICLR*, 2019b.

Pierre Stock, Angela Fan, Benjamin Graham, Edouard Grave, Remi Gribonval, Herve Jegou, and Armand Joulin. Training with quantization noise for extreme model compression. *ICLR*, 2020.

Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *arXiv preprint arXiv:1905.07799*, 2019a.

Sainbayar Sukhbaatar, Edouard Grave, Guillaume Lample, Herve Jegou, and Armand Joulin. Augmenting self-attention with persistent memory. *arXiv preprint arXiv:1907.01470*, 2019b.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. *EMNLP*, 2019.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. MobileBERT: a compact task-agnostic bert for resource-limited devices. *ACL*, 2020.

Héctor J. Sussmann. Uniqueness of the weights for minimal feedforward nets with a given input-output map. *Neural Networks*, 1992.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, 2013.

V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. How to evaluate deep neural network processors: Tops/w (alone) considered harmful. *IEEE Solid-State Circuits Magazine*, 2020.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CVPR*, 2014.

Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *ICML*, 2019.

Matus Telgarsky. Benefits of depth in neural networks. *JMLR*, 2016.

Bart Thomee, David A. Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. The new data and new challenges in multimedia research. *CoRR*, 2015.

Luis Torres and Daniel Prado. A proposal for high compression of faces in video sequences using adaptive eigenspaces. In *ICIP*, 2002.

Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. *NeurIPS*, 2019.

Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers and distillation through attention. *arXiv preprint arXiv:2012.12877*, 2020.

P. Tseng. Convergence of a block coordinate descent method for nondifferentiable minimization. *J. Optim. Theory Appl.*, 2001.

Mihran Tuceryan and Bruce E Flinchbaugh. Model based faced coding and decoding using feature detection and eigenface coding, March 28 2000. US Patent 6,044,168.

Frederick Tung and Greg Mori. Deep neural network compression by in-parallel pruning-quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: The impact of student initialization on knowledge distillation. *arXiv preprint arXiv:1908.08962*, 2019.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *ICCV*, 2017.

Evgeniya Ustinova and Victor S. Lempitsky. Deep multi-frame face hallucination for face identification. *arXiv*, 1709.03196, 2017.

Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. *Deep Learning and Unsupervised Feature Learning Workshop, NIPS*, 2011.

Vladimir Vapnik. *Statistical Learning Theory*. A Wiley-Interscience publication. Wiley, 1998.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthefw Yu, Tao Xu, Kan Chen, Peter Vajda, and Joseph E. Gonzalez. FBNetV2: Differentiable neural architecture search for spatial and channel dimensions. *CVPR*, 2020.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using DropConnect. In *ICML*, 2013.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. 2019a. ICLR.

Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: hardware-aware automated quantization. *arXiv preprint arXiv:1811.08886*, 2018a.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. Video-to-video synthesis. In *NeurIPS*, 2018b.

Ting-Chun Wang, Ming-Yu Liu, Andrew Tao, Guilin Liu, Jan Kautz, and Bryan Catanzaro. Few-shot video-to-video synthesis. In *NeurIPS*, 2019b.

Paul J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1988.

Olivia Wiles, A. Sophia Koepke, and Andrew Zisserman. X2face: A network for controlling face generation using images, audio, and pose codes. In *ECCV*, 2018.

Adina Williams, Nikita Nangia, and Samuel R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of NAACL-HLT*, 2018.

R. J. Williams and D. Zipser. Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity. In *Back-propagation:Theory, Architectures and Applications*. Lawrence Erlbaum Associates, 1995.

Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search. *CVPR*, 2019a.

Felix Wu, Angela Fan, Alexei Baevski, Yann Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. In *ICLR*, 2019b.

Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Yuxin Wu and Kaiming He. Group normalization. *ECCV*, 2018.

Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019c.

Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Conference on Computer Vision and Pattern Recognition*, 2017.

Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *CVPR*, 2019.

I. Zeki Yalniz, Hervé Jégou, Kan Chen, Manohar Paluri, and Dhruv Mahajan. Billion-scale semi-supervised learning for image classification. *arXiv e-prints*, 2019.

Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *CVPR*, 2016.

Mingyang Yi, Qi Meng, Wei Chen, Zhi ming Ma, and Tie-Yan Liu. Positively scale-invariant flatness of ReLU neural networks. *PNAS*, 2019.

Changqian Yu, Jingbo Wang, Chao Peng, Changxin Gao, Gang Yu, and Nong Sang. Bisenet: Bilateral segmentation network for real-time semantic segmentation. In *ECCV*, 2018a.

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. Nisp: Pruning networks using neuron importance score propagation. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2018b.

Q. Yuan and N. Xiao. Scaling-based weight normalization for deep neural networks. *IEEE Access*, 2019.

Sergey Zagoruyko and Nikos Komodakis. Learning to compare image patches via convolutional neural networks. *CVPR*, 2015.

Egor Zakharov, Aliaksandra Shysheya, Egor Burkov, and Victor Lempitsky. Few-shot adversarial learning of realistic neural talking head models. *ICCV*, 2019.

Egor Zakharov, Aleksei Ivakhnenko, Aliaksandra Shysheya, and Victor Lempitsky. Fast bi-layer neural synthesis of one-shot realistic head avatars. *ECCV*, 2020.

Jure Zbontar, Florian Knoll, Anuroop Sriram, Matthew J. Muckley, Mary Bruno, Aaron Defazio, Marc Parente, Krzysztof J. Geras, Joe Katsnelson, Hersh Chandarana, Zizhao Zhang, Michal Drozdzal, Adriana Romero, Michael Rabbat, Pascal Vincent, James Pinkerton, Duo Wang, Nafissa Yakubova, Erich Owens,

C. Lawrence Zitnick, Michael P. Recht, Daniel K. Sodickson, and Yvonne W. Lui. fastMRI: An open dataset and benchmarks for accelerated MRI. *ArXiv e-prints*, 2018.

Neil Zeghidour, Gabriel Synnaeve, Nicolas Usunier, and Emmanuel Dupoux. Joint learning of speaker and phonetic similarities with siamese networks. In *INTERSPEECH*, pages 1295–1299, 2016.

Biao Zhang, Deyi Xiong, and Jinsong Su. Accelerating neural transformer via an average attention network. *arXiv preprint arXiv:1805.00631*, 2018a.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *ICLR*, 2016.

Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *ICLR*, 2017a.

Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018b.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, 2017b.

Sanqiang Zhao, Raghav Gupta, Yang Song, and Denny Zhou. Extreme language model compression with optimal subwords and shared projections. *arXiv preprint arXiv:1909.11687*, 2019.

Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless CNNs with low-precision weights. *CoRR*, 2017.

Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, 2016.

Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. *CoRR*, 2016.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CVPR*, 2017.