université PARIS-SACLAY

# Spotting expressivity bottlenecks in neural networks and fixing them by optimal architecture growth

*Sur le manque d'expressivité des réseaux de neurones et leur résolution de manière optimale par accroissement d'architecture*

**Thèse de doctorat de l'université Paris-Saclay**

École doctorale n°580 : sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique mathématique.
Graduate School : Informatique et sciences du numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée dans les unités de recherche **Laboratoire Interdisciplinaire des Sciences du Numérique et Inria Saclay-Île-de-France** (Université Paris-Saclay, Inria), sous la direction de Sylvain CHEVALLIER, professeur, et le co-encadrement de Guillaume CHARPIAT, chargé de recherche.

**Thèse soutenue à Paris-Saclay, le 28 mars 2025, par**

**Manon VERBOCKHAVEN**

## Composition du jury

Membres du jury avec voix délibérative

| | |
|---|---|
| **Aurélie NÉVÉOL**<br>Directrice de recherche, CNRS, Université Paris-Saclay | Présidente |
| **Remi GRIBONVAL**<br>Directeur de recherche, INRIA, ENS Lyon | Rapporteur & Examinateur |
| **Hervé LUGA**<br>Professeur, Université Toulouse Jean Jaurès | Rapporteur & Examinateur |
| **Guillaume LECUÉ**<br>Professeur, ESSEC | Examinateur |
| **David PICARD**<br>Directeur de recherche, École des Ponts ParisTech | Examinateur |

THESE DE DOCTORAT

NNT : 2025UPASG022

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

**Sur le manque d'expressivité des réseaux de neurones et la résolution du problème de manière optimale**

**Mots clés: apprentissage profond, réseau de neurones, accroissement d'architecture en un coup, accroissement d'architecture à partir du gradient, optimisation sous contrainte, méthode d'approximation de rang faible**

**Résumé:** Cette thèse propose une stratégie originale d'accroissement d'architecture de réseaux de neurones en un coup, c'est-à-dire qui optimise conjointement l'architecture du réseau et ses paramètres. Cette méthode s'appuie sur une nouvelle métrique appelée le *Manque d'Expressivité*, qui associe à un emplacement de l'architecture du réseau actuel, son *incapacité* à suivre sa dérivée fonctionnelle. Il est montré que cette incapacité, ou Manque d'Expressivité, peut être résolu de manière optimale par l'ajout de neurones appropriés. Ce faisant, la résolution du manque d'expressivité fournit des outils et des propriétés pour développer une architecture à partir d'un très petit nombre de neurones. Ces travaux prouve, théoriquement et empiriquement, que les techniques de croissance basées sur cette métrique convergent vers une expressivité totale, et que si le rôle du gradient fonctionnel est explicite et majeur dans l'expression du Manque d'Expressivité, il est montré que ce rôle est aussi, implicitement, important dans d'autres stratégies récentes de recherche d'architectures neuronales.

**Title: Spotting expressivity bottlenecks in neural networks and fixing them optimally**
**Keywords : deep learning, neural architecture search, one-shot techniques, gradient-based method, constrained optimization, low-rank approximation**

**Abstract:** This thesis introduces a one-shot Neural Architecture Search strategy that jointly optimizes a network architecture and its weight using a new metric named the *Expressivity Bottleneck*. This metric associates a location of a network architecture to its lack of expressivity by quantifying the ability of the network to follow its functional gradient. It is shown that this lack of expressivity can be solved optimally by adding suitable neurons, hence providing tools and properties to develop an architecture starting with a very small number of neurons. This works proves, theoretically and empirically, that growing techniques based on this metric converge to full expressivity and that if the functional gradient plays an explicit and important role in the expressivity bottleneck metric, it is shown that it also plays, implicitly, an important role in other recent neural architecture search strategies.

# Remerciements

L'évidence et la matérialité des derniers épisodes de ma thèse n'aura jamais été aussi tangible qu'au moment au j'écris cette page des remerciements, et qui temporellement vient cloturer mon doctorat. Il faut dans ces derniere lignes, et j'aimerai terminer ma thèse sur ces mots, remercier mes collègues, mes amis et ma famille, qui ne mesurent que trop peu l'importance du temps et du bonheur qu'ils m'ont donné dans cette pèriode de ma vie.

En premier, je remercie Guillaume Charpiat pour le temps précieux et riche qu'il me donna, et pour la liberté et l'autonomie qu'il me laissa exercer tout au long de ma thèse. Je dois dire que je devinais déjà le caractère libre et détendu de mon futur gagne-pain quand il me recrutait comme stagiaire puis doctorante sur la base d'un simple appel vidéo, et sur recommandation de François Landes, ce dernier ayant cependant refusé ma candidature par email. Ensuite, je remercie Syvlain Chevallier qui me rejoingnait à mi-chemin de thèse. Je le remercie pour son calme et ses discussions appaisés qui m'ont permis de re-structurer et poser mes écrits, et en particulier après que notre paper fut plusieurs fois refusé par les grandes confrences.

Vient ensuite tous mes collèges de bureau, les petits thésards comme les permanants du batiment 660 qui n'ont eu de cesse de démêler ma pensée et d'éclairer ma caverne "ha ! Tu utilises encore `torch.matmul` ... Xfig ? Mais c'etait il y a 15 ans ça, tu sais maintenant ... Ha oui Lnsckape ...". En particulier, je garde precieusement le souvenir de mes boites à dejeuner avec Antoine Szatkownik, de la tablette de chocolat de Theo Rudkiewick, et du verre de boisson ici comme ailleurs, avec Nicolas Bereux.

Pour terminer, je remercie toute ma famille et mes amis proches qui sont venus ici aujourd'hui, et qui ont bravé le bus et le formalisme des sciences pour venir m'écouter et célébrer avec moi ce moment important.

Peu sur beaucoup et beaucoup sur rien ...

# Contents

# List of Figures

# List of Tables

# 1 Introduction

« On the basis of [...] the speed with which the research in [Heuristic Problem Solving] is progressing, we are willing to make the following predictions, to be realized within the ten years:
That within ten years, a digital computer will be the world's chess champion unless the rules bar it from competition.
That within ten years, a digital computer will discover and prove a new important mathematical theorem.
That within ten years, a digital computer will write music that will be accepted by critics as possessing considerable aesthetic value.
That within ten years, most theories in psychology theory will take the form of computer programs, or of quantitative statements about the characteristics of computer programs. »

Herbert A. Simon and Allen Newell, *Heuristic problem solving* (1957)

On November 14, 1957, at the banquet of the Twelfth National Meeting of the Operations Research Society of America, Herbert A. Simon and Allen Newell predicted well the technological leap that was about to start with Artificial Intelligence. The predictions of their paper *Heuristic problem solving* [Simon and Newell, 1958], as stated in the introduction of this thesis, would come true starting from 1996 as the supercomputer *DeepBlue* defeats the world chess champion. Since that date, research in Artificial Intelligence has been enjoying growing enthusiasm, moving from symbolic AI to statistical learning or Machine Learning (ML) techniques, which have permitted to tackle more abstract and complex tasks such as image and music generation, reasoning in games, etc.

Among all existing ML models, the neural networks stand out as they can be applied to a wide range of problems. Its story begins in 1958, one year after this citation, in an article proposed by psychologist F. Rosenblatt on the basis of the human brain. At this early stage, a neural network is mainly defined by its "*architecture*", and its performance is enhanced by training through an "*optimization*

*process*". Over the years, the definitions and settings of each of those two concepts have become so complex that they have become an important research field by themselves. In particular, considering the now usual optimization process of the network parameters, that is, gradient descent, it is acknowledged that the final performance of the network is tightly related to the network *architecture*, which has to be carefully chosen when considering one task or another. Indeed, novel and ingenious architecture designs often enhance the trainability of networks and better incorporate physics and human knowledge in task modeling, leading in either case to improvement in the network final performance. Constructing suitable architectures has long been to engineers and researchers a painful occupation, but new techniques that remove the human from this building loop are taking shape. This thesis falls into this new research field, that is, Neural Architecture Search (NAS).

For the non-expert readers that are more curious about NAS, I propose here an unambiguous analogy :

Suppose that God's apprentices would like to have a personal chef and decide to create it. To do so, they need to *assemble parts of the human body* into a human puppet, which will be *trained* to cook. Considering all possible layouts of the body sections, they try a first idea and place the hands on top of the head. After several hours of training, they observe, quite embarrassed, that their puppet is incapable of cooking the simplest recipes. One apprentice suggests that this failure might be linked to the puppet's inability to see its actions while using its hands. Considering this remark, they decide to revise the joining of the puppet and restart the training...

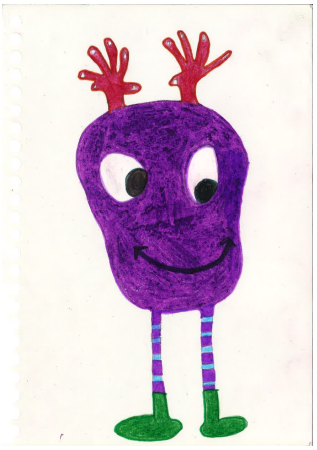


Figure 1.1: The chef by the apprentices.

Replace the action "*assemble the parts of the human body*" by *construct the architecture of a neural network*, and the apprentices become a computer program of some researchers in the NAS field.

In this analogy, we recognize a well-known optimization process called trial and error that, in NAS, is the repetition of the following : build an architecture, train it, and evaluate its performance. However, this way of proceeding distinguishes the search of the architecture from the training of the architecture itself, and is by construction time demanding because it requires training multiple architectures. An alternative would be to jointly optimize the architecture and its weights, that is, we would construct the chef and train it to cook at the same time. Such NAS techniques exist and they are called *one-shot* methods. In fact, they are called like that because, inversely to the latter optimization procedure, only one

architecture is trained during the overall search. There exist few such techniques in the literature, and most of them have been designed in the past five years. Within this manuscript, we complete this family of *one-shot* techniques with an original method and also prove that two existing ones fall into this formalism.

The NAS technique presented in this manuscript achieves the joint optimization of a network architecture and its weights using a new metric, the *Expressivity Bottleneck*. This metric associates a part of the network architecture to its lack of expressivity and naturally yields updates of parameters as well as architecture expansions to reduce such lack of expressivity. The technique avoids redundancy in the network by proposing extensions of architecture that are orthogonal to the achievable function space defined by the current network, and this, using the by-product of the gradient descent. This latter property allows this growing technique to be performed at the same time as the standard gradient descent training of the network. Each architecture modification, as well as their potential utility for the network, can be computed cheaply and in a close form from backpropagation, hence providing tools and properties to develop an architecture starting with a very small number of neurons. Growing techniques based on this metric are successful as we prove, theoretically and empirically, that they converge to full expressivity for any given dataset.

This thesis is composed of six chapters where the first and the last ones are, respectively, the introduction and the conclusion of this manuscript. The literature review is in Chapter 2 and has been divided into two main sections; in the first part, we present the NAS field, and in the second part, we introduce methods to evaluate the expressivity of a network given its architecture. The methods from the second part of the literature review might be empirical or mathematical metrics, and are to provide a baseline comparison for our metric *Expressivity Bottleneck*, as much on its mathematical definition than as on its ability to be computable.

In Chapter 3, we bridge the gap between the concept presented in the latter chapter by introducing the *Expressivity Bottleneck* metric, its properties, and how it applies to the NAS field.

In Chapter 4, we revisit two NAS techniques with my formalism.

Finally, in Chapter 5, we construct a naive NAS strategy using the *Expressivity Bottleneck* metric, to show on multiple reference datasets that such strategy matches large neural network accuracy, with competitive training time, while removing the need for standard architectural hyper-parameter search.

A part of this manuscript has been published in the TMLR journal [Verbockhaven et al., 2024]. The corresponding sections are indicated with the icon  next to their titles.

# 2 Literature review

**Contents**

In this chapter, we present the literature review on Neural Architecture Search (NAS) and on the notion of expressivity for neural networks.

In the first section, we present NAS, a research field that focuses on the construction of the architecture of neural networks; we start by defining its main concepts, which are the search space, the search strategy, and the evaluation strategy (Section 2.1.1), then, we go into more details for each of those concepts by listing the different types of search spaces and search strategies found in the literature (respectively Section 2.1.2 and Section 2.1.3).

In the second section, we provide several definitions of the notion *expressivity* and link it with the usual definition of performance. In the first part (Section 2.2.1), we present in more detail the evaluation strategy, which will have been defined in the previous section, and which regroups the empirical methods to evaluate the performance of a network without using the usual gradient descent; then, in a second subsection (Section 2.2.2), we present the mathematical tools known in the literature to define *expressivity*. In particular, we present the Vapnik–Chervonenkis dimension, the Rademacher complexity, the information bottleneck, and the Kolmogorov complexity, and how they apply to neural networks.

## 2.1 Neural architecture Search (NAS)

Neural Architecture Search (NAS) first arises as a subfield of AutoML, which aims to automate all pipeline steps in Machine Learning processes. In particular, it shares many of its concepts and definitions with hyper-parameter optimization (HPO) if we consider the network architecture as a hyper-parameter itself. However, HPO and NAS techniques are rather different as they optimize variables of different kinds. In HPO, the optimized variables lie within well-defined, low-dimensional, and non-complex spaces, making its algorithms ill-adapted to the optimization of the complex and high-dimensional space that is the space of architectures.

The research in NAS started a long time ago with genetic and evolutionary methods, but recent work starting from 2015-2017 gave a fresh impetus to the field, leading to an increasing number of papers per year (cf. Figure 2.1). Different formulations with their inner logic have been developed, and we will present and summarize those in the following sections.

We start by introducing the main definitions and concepts in Section 2.1.1, and we go into more detail for each of those concepts in Section 2.1.2.

### 2.1.1 Definitions

Formally, NAS can be formulated as follows : consider a task $\mathcal{T}$ with the dataset $\mathcal{X}$ separated into the training and the validation sets, and a set of architectures $\mathbb{A}$,

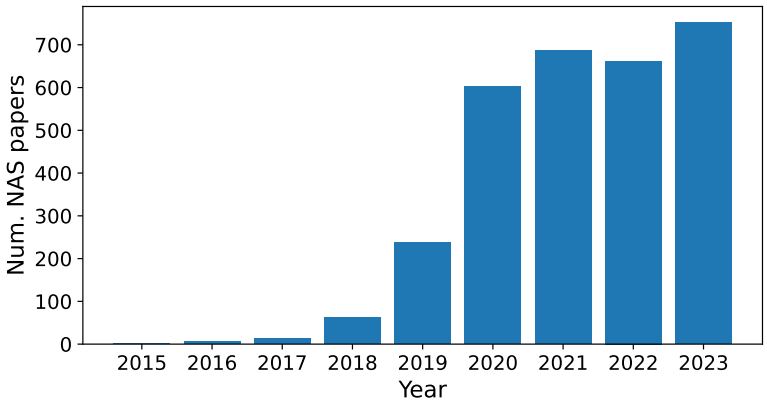



Figure 2.1: Number of NAS papers by year [Deng and Lindauer, 2023]

we would like to solve :

$$\underset{\mathcal{A}(.)\in\mathbb{A},\ \theta\in\mathbb{R}^{d(\mathcal{A})}}{\arg\min}\ \mathcal{L}_{train}(\mathcal{A}(\theta)) \tag{2.1}$$

where the loss $\mathcal{L}_{train}$ evaluates the performance of the network $\mathcal{A}(\theta)$ on the training set and $d(\mathcal{A})$ is the dimension of the parameters of the architecture $\mathcal{A}$. Usually, this optimization problem is reformulate as a two-step optimization problem where we search for the best architecture in $\mathbb{A}$ while the value of its parametrization $\theta^*$ has been fixed by minimizing the training loss for that specific architecture. Formally, Equation (2.1) is equivalent to :

$$\underset{\mathcal{A}\in\mathbb{A}}{\arg\min}\ \mathcal{L}_{train}(\mathcal{A}(\theta^*)) \qquad s.t. \quad \theta^* := \underset{\theta}{\arg\min}\ \mathcal{L}_{train}(\mathcal{A}(\theta)) \tag{2.2}$$

However, as the training set is used twice, first to estimate the best parametrization $\theta^*$ and the optimization of the architecture within $\mathbb{A}$, the solution of such optimization problem would overfit the training set, and one would observe a drop in accuracy when measuring the performance on the validation test. This overfit can be mitigated either by increasing the training dataset or by performing one of those optimization problems on the validation set. For example, instead of solving Equation (2.2), one could solve the following minimization problem:

$$\underset{\mathcal{A}\in\mathbb{A}}{\arg\min}\ \mathcal{L}_{val}(\mathcal{A}(\theta^*)) \qquad s.t. \quad \theta^* := \underset{\theta}{\arg\min}\ \mathcal{L}_{train}(\mathcal{A}(\theta)) \tag{2.3}$$

where the loss $\mathcal{L}_{val}$ evaluates the performance of the network $\mathcal{A}(\theta)$ on the validation set. This formulation can be slightly modified to direct the search toward a preferred subspace of $\mathbb{A}$, for example, by adding constraints on the size of $\mathcal{A}$ or a limit of time to compute the search.

Depending on the type of knowledge is used to solve Equation (2.3), we differentiate three types of NAS : the *from scratch*, the meta-learning and the transfer-learning methods. If a method solves Equation (2.3) using knowledge from matching duos of (dataset, architecture) that are not $\mathcal{T}$, we are in the meta-learning or transfer-learning setting. Reversely, if the construction of the architecture is specific to the task $\mathcal{T}$ and is performed using only information from task $\mathcal{T}$ (for example its associated dataset), we are instead in the *from scratch* setting. We can remark that the strict distinction between the from scratch and meta or transfer-learning is in fact incorrect because a from scratch method always implicitly uses knowledge from tasks other than $\mathcal{T}$. For example, it might focus its search on the attributes of known and workable architectures (convolutional layer, normalization layer, max-pooling). However, we consider that such use of exterior knowledge is minor compared to what can be done in meta-learning or transfer-learning where, for example, a neural network with multiple layers can be first trained to solve a task $\tilde{\mathcal{T}}$, and then partially retrained for task $\mathcal{T}$. For the rest of this thesis, we focus on the from-scratch search, as is our technique, and set aside the meta-learning and transfer-learning search.

We identify three main concepts which distinguish NAS techniques from one another: **the search space**, **the search strategy** and the **evaluation strategy** [Elsken et al., 2019, White et al., 2023]. We first give a general definition of those concepts and specify their diversity in the next sections.

- $\triangleq$ the search space defines the set of possible architectures $\mathbb{A}$ in which the search will be performed. It is chosen with prior knowledge of the given task and dataset to save time and energy searching in what is widely acknowledged as well-functioning architectures. In general, a large search space is more likely to give a better solution for Equation (2.3) but also tends to increase the search time within $\mathbb{A}$.

- $\triangleq$ the search strategy defines the exploration within $\mathbb{A}$ toward the solution of Equation (2.3). We distinguish three types of search: the random method, the informed random search (reinforcement learning, Bayesian optimization, neuro-evolutionary), and a more recent category, the one-shot methods. The informed random search differ from the random ones because they use *knowledge* to orient the search within $\mathbb{A}$. This *knowledge* is usually a concatenation of all the returns of the evaluation strategy that is called at each increment in $\mathbb{A}$. The last category of one-shot methods, to which our method belongs, does not distinguish the evaluation strategy from the search strategy: it performs architecture modifications while optimizing the current architecture. This joint optimization often induces a shorter time of search within $\mathbb{A}$.

- $\triangleq$ the evaluation strategy estimates the performance of a given architecture $\mathcal{A}$, that is $\mathcal{L}_{val}(\mathcal{A}(\theta^*))$ of Equation (2.3). This information is then sent and processed by the search strategy to continue and guide the search within $\mathbb{A}$. A straightforward and classic evaluation strategy is to train the overall

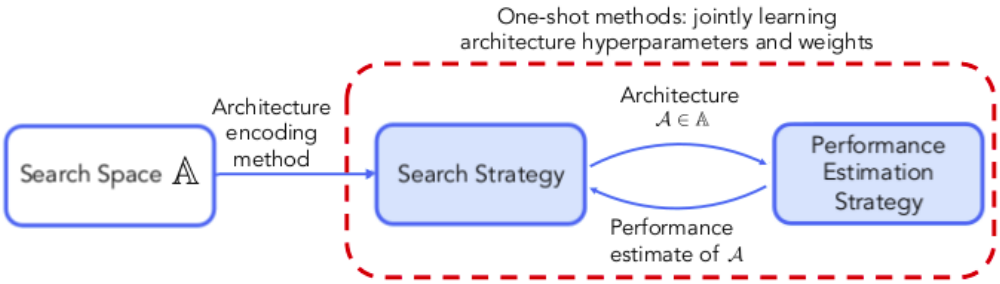



Figure 2.2: Graph of operation of NAS methods White et al. [2023]. A set of possible architectures $\mathbb{A}$ is chosen. A starting architecture $\mathcal{A}$ is drawn from $\mathbb{A}$ to encode a neural network $\mathcal{N}$. The performance of $\mathcal{N}$ is then evaluated and is processed by the search strategy, which might draw another architecture $\mathcal{A}$ from $\mathbb{A}$.

architecture $\mathcal{A}$ until convergence by usual gradient descent. Although techniques exist to shorten the computational time of such evaluation, this step is always the most costly operation in NAS. As stated above, for one-shot methods, the evaluation strategy is either non-existent or estimates a proxy of this quantity without further training.

In the following sections, we will present in more detail the search space and the search strategy. Although the evaluation strategy is tied to those two concepts, its presentation has been grouped with the literature review on expressivity (Section 2.2.1), as it can be seen as an empirical technique to estimate the expressivity of a network.

### 2.1.2 The search spaces : *The chain-structured, the cell-based and the hierarchical search space*

The chain-structured set of architectures is the simplest and historically the oldest one to be found in the literature. Each structure in the set is organized as a file of ordered layers, where a layer is a composition of a linear application and a non-linear function. The layer at position $k+1$ in the line takes its inputs from a subset of the outputs from the previous layers. In this setting, the set of architectures $\mathbb{A}$ is firstly defined by the possible variations of the *macro-structure*, which is the number of layers in the line and the set of indices from which each layer takes its inputs from [Kandasamy et al., 2018b], then, by the variations of the *micro-structure*, that is the type of operation at each layer [Ramachandran et al., 2018], [Chollet, 2017] and the hyperparameters of each operation [Wenzel et al., 2020]. Remark that searching for the micro-structure can be performed exhaustively as the set of possibilities for the operations (relu, soft step, hyperbolic

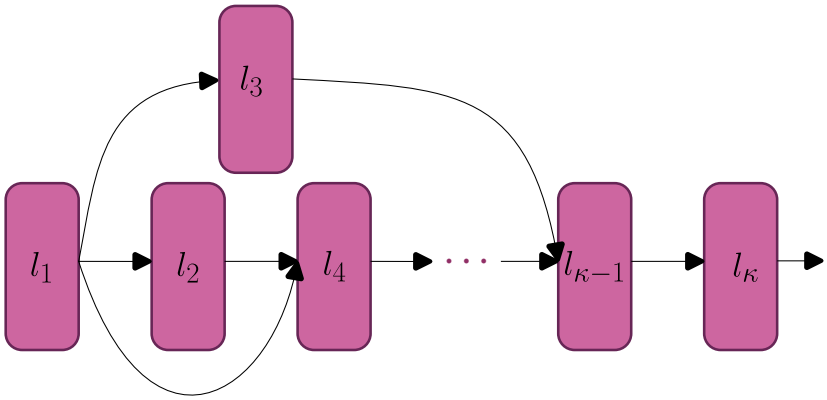



Figure 2.3: Example of a chain-structured architecture. Each layer $l_j$ takes its inputs from the incoming arrows.

tangent, etc.) and the settings of most of the hyper-parameters (kernel size, strides, presence of the bias, etc.) is of finite and relatively small cardinality while searching for the best macro-structure requires to optimize a combinatorial problem and is an issue at the very heart of the NAS research field.

Another popular and more recent search space is the cell-based search space where, as before, two optimization levels can be defined, *micro* / *macro* structure, but they are of another kind. The micro-architecture is instead called a cell and is a chained-structured architecture, while the macrostructure is the number of cells and the set of connections from one cell to another. Usually, the search focuses on the construction of the cell, while the macro-structure is fixed in advance and is often a simple graph. By construction, this type of architecture has recurring motifs, which have proved to be a nice property in many human design architectures such as ResNet, U-net, Transformers, etc. To reduce the time of the search for large and complex datasets, a set of best cells is pre-selected with a small and task-similar dataset such as CIFAR-10 for vision, or TIMIT for speech recognition [Liu et al., 2019, Zoph et al., 2018, Mehrotra et al., 2021], and only those selected cells are evaluated on the final dataset. However, this two-level optimization process mainly focuses the search on the micro-level, implying supplementary human bias or important amount of by-hand optimization work for the macro-level. This extra work can be avoided by the last category of search space, which is the hierarchical search space.

The hierarchical search, first introduced by Liu et al. [2018], enlarges the binary vision macro vs. microstructure by considering a spectrum of levels that are defined recursively. In the papers [Liu et al., 2018, Chrostoforidis et al., 2021], each cell of a specific level is a graph of cells from the preceding level. In the paper [Ru et al., 2020], the same logic applies with the small difference that the most nested level defines the hyper-parameters of the basic operations. Remark that other

hyper-parameters being equal, this hierarchical search space covers the cell-based search space as soon as more than two levels are considered.

### 2.1.3 The search strategy

Considering a task $\mathcal{T}$ associated to the dataset $\mathcal{D}$, we note $f^*$ the function such that for all elements $(\boldsymbol{x}, \boldsymbol{y})$ following the law distribution of $\mathcal{D}$, we have $f^*(\boldsymbol{x}) = \boldsymbol{y}$ (supposing a deterministic task and no labeling error). A simple search strategy to approximate $f^*$ with a neural network would be to take a random but huge feed-forward network and train it by gradient descent. Indeed, such a network enjoys two nice properties regarding its generalization error and the optimization of its parameters. The first aspect is provided by the universal approximation theorem by Hornik et al. [1989] and indicates that, upon some weak assumptions, the functional space described by the variation of the parameters of such an architecture is large enough to be close to any function $f^*$. The second property comes from the NTK theory by Jacot et al. [2018] and ensures that following the gradient descent will guide the parameters of that network toward a good minimum in the functional space. Those two properties are illustrated in Figure 2.4. While choosing a reasonable order of magnitude to translate the quantification "huge architecture", this method is always feasible. Furthermore, it is common practice to reduce the size of such a network once it has been trained, using pruning techniques that iteratively remove the useless parts of the network according to a mask, or other compression techniques (tensorization, quantization, distillation). However, for a new and complex task, the quantification of "huge" is often incorrect, and without any ingenious searching technique, the architecture search rapidly resembles a trial and error process which is memory inefficient and time-consuming, especially when dealing with high-dimensional data (images, videos, songs, text). In those settings, we might use more complex techniques that belong to the NAS search strategy field, which permit us to always work with small and manageable architectures. In the next sections, we describe those techniques that we have organized into two main categories : probabilistic techniques and one-shot strategies. We briefly summarize the probabilistic techniques, and we go into more detail for the one shot, which is the category this thesis belongs to.

#### Probabilistic techniques : from brute force to informed random searches

We name the probabilistic methods the strategies that use random distributions as the main indicators to search and move within $\mathbb{A}$. Among the literature, we distinguish three different methods in this category: the neuro-evolutionary, the reinforcement learning, and the sampling and Bayesian optimization methods. In each of those, there exists a continuum of techniques going from random and brute force approaches to more sophisticated and clever ones, named here informed random approaches. Regarding all other search strategies, the random methods are the most straightforward and, in research papers, are considered a weak baseline

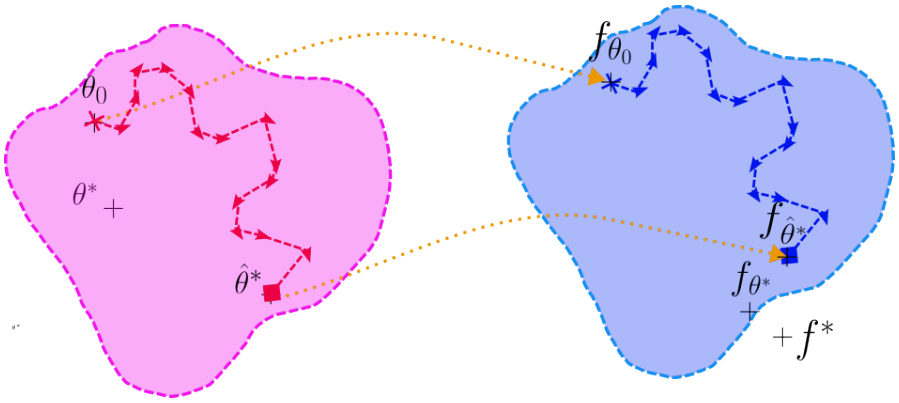



Figure 2.4: Optimization of the parameters of a network with a fixed and "huge" architecture $\mathcal{A}$. The pink shape defines the space of parameters of the architecture $\mathcal{A}$, and the blue shape represents the corresponding functional space. Each point in the pink shape can be paired to a point in the blue shape. Note that many different parameters $\theta$ yield the same $f_\theta$, for example by considering some permutations of $\theta$. The pink arrows indicate the path followed by the parameters during the gradient descent, and the blue arrows describe the induced functional changes. The parameter $\theta_0$ is the starting parameter of the gradient descent procedure, and $f_{\theta_0}$ is the corresponding neural network function. At the end of the gradient descent procedure, the parameters of the network are equal to $\hat{\theta}^*$, and using the approximation theorem and the NTK, we have the property that under certain assumptions the corresponding function $f_{\hat{\theta}^*}$ is close to $f_{\theta^*}$ which is itself close to $f^*$.

with which to be compared. This does not hold true for informed random methods, which are more subtle and challenging to implement as they use conditional random distributions where the knowledge of past explorations is somehow proceeded while exploring $\mathbb{A}$.

**Neuro-evolutionary.** Historically, neuro-evolutionary methods were the first ones to be developed to search for neural network architectures and, at a time when computing the gradient was costly Zhang et al. [1993], its inner logic was also used to update the weights of the network. Neuro-evolutionary methods operate on representation (genotype), which can be mapped to a neural network (phenotype) to be evaluated on some task to derive its *fitness*. A *fitness function* summarizes the performance of a network into one real number and guides the neuro-evolutionary algorithm toward the solution. The evolution process is as follows: take a population of genotypes, i.e., model - use sample parents from this population to get offspring by applying mutations to the parents. Those models are then trained, evaluated, and added to the population (cf Algorithm 1). Neuro-evolutionary methods differ from each other in how they encode the genotype, perform the mutations, sample parents, update the population, and evaluate the fitness. The encoding of a genotype can be direct [Yao and Liu, 1997, Stanley, 2007], that is, each connection and weight value can be mapped from the genotype to the network, or indirect [Stanley et al., 2009, Miikkulainen et al., 2017] that is the encoding is rather a program which indicates how to build the topology of the network and initialize its weights. If the direct encoding is straightforward and easy to implement, indirect encoding is compact and leads to an alternative exploration of $\mathbb{A}$ as the set of possible mutations and crossovers is highly preconditioned by the encoding. Usually, the cardinality of the population stays constant during the process, and some individuals are *killed* or *discarded*. They are often the worst-performing individuals, but they can also be the oldest to favor diversity and the spread of newer individuals [Real et al., 2018]. There exist other techniques to favor diversity in the population, as in the paper Stanley [2007], where individuals with similar genomes share their fitness payoff, which allows different structures a chance to optimize in their own niches. Neuro-evolutionary methods, like reinforcement learning methods, are very costly when the data lies in a high-dimensional space but seem to perform better than random search on the classic vision datasets MNIST, CIFAR-10 [da Silveira Bohrer et al., 2020] and ImageNet [Real et al., 2019].

**Reinforcement Learning (RL).** In reinforcement learning techniques, an agent explores and exploits its environment by taking actions that change its state and make it pocket a reward. The different actions are chosen according to the policy $\pi$, that is, the probability of performing an action knowing the current state. The objective is to find the policy $\pi^*$, which maximizes the expected reward of the agent, that is, the weighted sum of the expected future rewards. Different modeling

and optimization processes enable RL to answer NAS optimization problem as formulated in Equation (2.3). Most of the time, the agent is rather called a controller and is a recurrent neural network that generates new architectures by a mapping from a fixed length vector [Baker et al., 2017] or transforms an existing network by either increasing its depth or the size of an existing layer [Cai et al., 2017]. The reward is often equal to the validation accuracy of the sampled network after it has been (partially) trained. The architecture generation is enhanced by updating the parameters of the controller by gradient descent, which is in the RL jargon named the direct policy search strategy, using the Q-learning logic [Zhong et al., 2017, Baker et al., 2017, Cai et al., 2017] or the proximal policy optimization [Schulman et al., 2017] (8-10 days with 10 GPUs for CIFAR-10 dataset). Compared to the other strategies, reinforcement learning applied to NAS is a recent practice. In fact, solving Equation (2.3) in terms of RL is so resource-demanding that these strategies have only been considered with the emergence and mainstream use of large-scale parallelization.

**Sampling and Bayesian optimization.** Sampling techniques consist in generating architectures and evaluating their performances until an architecture with sufficient performance is found. Most sampling techniques come from the HPO domain and, when they are naive, are ill-conditioned and time-consuming for searching into the space of architecture. Nonetheless, for the low-complex tasks and rather small datasets, those naive techniques achieve competitive results, as in Yu et al. [2020] and Li and Talwalkar [2019], which reach almost state-of-the-art results on CIFAR-10 dataset using a grid-search technique for sampling the architecture (within 9.7 GPU days for Li and Talwalkar [2019]). Sampling techniques can be upgraded by exploiting the knowledge of the performance of past explorations when sampling new architectures; in that case, one speaks about Bayesian Optimization (BO). Those techniques interpolate performance for unseen architectures, assuming regularities in their function form, and in doing so, they propose further interesting potential designs to sample. The function form of the performance is given by the acquisition function $\phi$, which makes a trade-off between the exploration of the unexplored subspace of $\mathbb{A}$ and the exploitation of well-working and known designs. This function is approximated by a surrogate with the current population of visited architectures. The arg max of the surrogate is chosen as the next architecture to sample; it will be trained and added to the population, enhancing the next estimations of $\phi$ (cf. Algorithm 2). BO techniques differ from one another by the nature of the acquisition function, how they choose the surrogate, and how they encode the space of architectures. The most popular acquisition function is the expected improvement, and although other function forms exist, White et al. [2019] have shown that changing that criterion will not affect much the final performance of the search. The surrogate is often a Gaussian process [Bergstra et al., 2011, Kandasamy et al., 2018a] or a Bayesian neural network [Springenberg et al., 2016] which predicts a confidence interval for the

**Algorithm 1:** Pseudo-code for neuro-evolutionary search

```
Data: Data ≜ {x_i, y_i}_i
Randomly sample an initial population;
for individual in the initial population do
    Train the individual and evaluate its fitness;
end
for t=1, ..., T: do
    Selection: Individuals are selected for reproduction;
    Crossover: The selected individuals mate and produce offspring;
    Mutation: The offspring undergoes random mutations;
    Train the new individuals and evaluate their fitnesses;
end
```

**Algorithm 2:** Pseudo-code for Bayesian optimization search

```
Data: Data ≜ {x_i, y_i}_i, an acquisition function φ(.), a surrogate
Randomly sample a set of architectures ;
for A in the initial set of architectures do
    Train A and evaluate its performance;
end
for t=1, ..., T : do
    Train the surrogate model to fit φ with the current population ;
    Select the architecture A that maximizes the surrogate ;
    Train A and add it to the population;
end
```

performance of each point of $\mathbb{A}$. The encoding of the architecture modifies the writing and the estimation of the surrogate as it transforms the topology of the architecture space as being the input of the surrogate function. It can be defined by the adjacency matrix which indicates if node $i$ is connected to node $j$ or the path adjacent matrix [White et al., 2019] or a projection on a low-dimensional space with an auto-encoder [Luo et al., 2018]. BO methodology can be mixed with other approaches as in Cai et al. [2017], which designs a pseudo-metric "OTMANN" to estimate a Gaussian process surrogate and uses evolutionary search to mutate and explore $\mathbb{A}$ starting from the architecture with the highest surrogate value.

**One-shot and gradient-based.** One-shot methods, unlike black-box optimization search, use only one model and perform only one training over the search in the space of architectures $\mathbb{A}$. They develop a unique architecture and enhance its performance simultaneously, removing the distinction between optimizing in $\mathbb{A}$ and optimizing the setting of a specific $\mathcal{A}$ in $\mathbb{A}$. Their double optimization processes usually rely on the information extracted from backpropagation; hence, they are also nicknamed gradient-based methods. Compared to the previous section, each method has its inner logic, and instead of defining subcategories, we instead describe each technique from the furthest to the closest one of our method. For each of those techniques, I created a little scheme which identifies the method.

$\boxed{\begin{pmatrix}\pm 1\\ \cdot\\ \pm 1\end{pmatrix}\longmapsto \pm 1}$ **Faithful representation (not gradient-based, no training)**
For sign vector inputs and one-dimensional sign vector outputs, the authors of paper Mezard and Nadal [1989] propose a method to add layers and they prove that this method converges to a zero system error on the training set. Each layer is built brick by brick, adding neurons until the internal representation of the inputs at this layer is said to be faithful (▲); that is, any couple of inputs with distinct outputs have distinct internal representations. Once the layer $L$ is faithful, they propose a nice initialization using the pocket algorithm, which, coupled with this property, construct the units of the next layer, $L+1$, whose first unit produces at least one fewer error than the first unit of last layer $L$. If the error is non-zero, the size of layer $L+1$ is again increased until it is faithful, restarting the process in (▲). Repeating this logic, they achieve zero error in the training after a finite number of steps. There is no generalization property for this algorithm.

$\boxed{\begin{pmatrix}\{0,1\}\\ \cdot\\ \{0,1\}\end{pmatrix}\longmapsto \begin{pmatrix}\{0,1\}\\ \cdot\\ \{0,1\}\end{pmatrix}}$ **Universal neural Net (not gradient-based).**
For binary inputs with multidimensional binary outputs, the paper Chang and Abdel-Ghaffar [1992] proposes a method to increase the size of a one-hidden layer network by adding iteratively and one by one new hidden neurons. The method is proved to converge to zero loss on the training set and can be performed independently of gradient descent, in the sense that this convergence property still holds if gradient descent steps are performed between each neuron addition. The

initialization of each added neuron is such that it divides by four the loss value for one dimension of the output of the worst example while not changing much the prediction and (a fortiori) the associated loss for the other examples and dimensions. The neuron addition multiplies the overall loss by at least a constant factor equal to $1 - \frac{1}{2PM}$ where $M$ is the dimension of the output, and $P$ is the size of the training set. This result rests on the monotony and boundedness of the activation functions and the binary aspect of the input /output; hence, the extension of this method to float inputs does not hold. The proof of convergence of this method does not use any gradient-based argument but rather the loss value of the examples, however, the paper concludes with a note on a gradient-based proposition, which is a forerunner for the GradMax method [Evci et al., 2022], another one-shot strategy that is described below. It proves that initializing the fan-in weight of the new neuron with their initialization while setting its out-fan weights to zero does not change the overall loss while boosting the gradient descent as the gradient according to the zero-out-fan weights is strictly different from zero. Certainly, applying gradient descent after this process permits a decrease in the loss, but this gradient-based argument does not prove the convergence to zero loss. As the previous method, there is no generalization property for this algorithm.

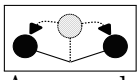
We now describe four recent and gradient-based methods that can be applied to float vector inputs and outputs. Compared to the previous methods, they do not create the overall architecture and are not proven to converge to zero loss on the training set. Nonetheless, they achieve competitive results on two major visual recognition tasks, which are CIFAR-100 [Krizhevsky et al., 2009] and ImageNet [Deng et al., 2009].

ℂ≕ℂ **DARTS** an example of supernetwork

We start this section by presenting the DARTS methodology [Liu et al., 2019] that is based on supernetworks. A supernetwork is a huge network containing all the architectures of the search space $\mathbb{A}$, and where each specific architecture of the search space can be retrieved by selecting a sub-graph of the supernetwork. The methods defined with a supernetwork aim at finding an appropriate sub-graph by eliminating the unnecessary connections of the supernetwork [Saxena and Verbeek, 2016, Bender et al., 2018, Liu et al., 2019]. The main advantage of using a supernetwork is that its training evaluates all the architectures of the search space at the same time and that such a training time depends linearly on its number of parameters; while training each architecture of the search space independently has a training complexity that grows exponentially with the total number of parameters of the biggest architecture [1]. The most well-known example of a supernetwork

[1] Consider a search space where each architecture can be obtained by choosing $k$ connections among $n$. If an architecture with $k$ connections has a training time complexity proportional to $k$, then, training all the architectures of the search space has a time complexity of $\sum_k \binom{n}{k} k =$

is DARTS, whose methodology is now detailed. In DARTS, the search space is cell-based, and the strategy is to learn the connections and activation functions between pre-constructed cells by gradient descent. We note that the learning of the connections and the activation functions is not differentiated, as the activation functions are searched in a finite set, which includes the null function. The method starts from a fixed skeleton of cells, where the connections between the cells are multiple, and each cell takes as input a weighted sum of the different activation functions whose weights are learned by gradient descent. At the end of training, the smaller weights of the activation functions are set to zero, defining, therefore, the final architecture of the network.



**Firefly**

A second one-shot and gradient-based method is the firefly gradient descent from paper Wu et al. [2020]. This technique starts from a tiny architecture that outputs the function $f$, and at each loop of the search strategy, it replaces the current architecture with an augmented one whose output function is $\|\varepsilon\|$ close to $f$, where $\varepsilon$ is a vector. This new architecture is searched inside a set of functions, $\partial(f, \varepsilon)$, where each function of the set resembles the current architecture but where its neurons have been divided into two neurons, and new neurons as well as skip connections have been added to the structure. Let $\delta_{split}$ and $\delta_{new}$ be some vectors and $\varepsilon = (\varepsilon_{split}, \varepsilon_{new})$, then the separation of a neuron defined by $n : \boldsymbol{x} \to \sigma(\theta^T \boldsymbol{x})$, is the replacement of such a neuron with the function $n_{split} : \boldsymbol{x} \to \frac{1}{2}\left(\sigma((\theta + \varepsilon_{split}\delta_{split})^T \boldsymbol{x}) + \sigma((\theta - \varepsilon_{split}\delta_{split})^T \boldsymbol{x})\right)$; the addition of one neuron is defined as the concatenation of a new neuron whose function is $n_{new} : \boldsymbol{x} \to \varepsilon_{new}\sigma(\delta_{new}^T \boldsymbol{x})$ to a specific layer. The addition of skip connections is defined using the same logic as the addition of neurons. The optimization inside $\partial(f, \varepsilon)$ is done first by an optimization on $(\varepsilon, \delta)$ with a $\ell_2$ constraint on the vector $\varepsilon$, then it is followed by an optimization on $\varepsilon$ with a $\ell_0$ constraint on $\varepsilon$. The constraints $\ell_2$ and $\ell_1$ on $\varepsilon$ through the optimizations ensure that the new architecture is $\varepsilon$ close to the previous one. The global search strategy is then defined by the alternation between the classical optimization of the parameters of the current architecture with the usual gradient descent and the replacement of the current architecture with an $\varepsilon$ close architecture.

We now present two methods that expand a network by increasing the size of its current layers. Those methods start with a well-known architecture, for example, a ResNet, but with fewer neurons by layer, then they iteratively increase the layer sizes by adding new neurons. In those two methods, the input weights or the output weights of the added neurons are initialized to zero to keep the output function of the network unchanged, while the remaining weights are initialized ingeniously. We point out that the inner logic of those two methods is a first step

$n2^{n-1}$, while training the largest architecture is of time complexity $n$.

into the understanding of our technique and that more details will be given in Chapter 4.

$0 \mid \|\nabla\mathcal{L}\|^2$ **GradMax**

GradMax method Evci et al. [2022] grows architectures by adding blocks of neurons to existing layers. Its strategy is to ensure that the norm of the gradient with respect to the new parameters is as large as possible because the first-order development of the loss with one usual gradient step is proportional to that norm. Indeed considering $f_\theta$ a neural network with parameters $\theta$, and performing at time $t$ the update of parameter $\theta \leftarrow \theta + \delta\theta$ with $\delta\theta = -\eta\nabla_\theta\mathcal{L}(f_\theta)$, where $\mathcal{L}$ is the loss function and $\eta \geq 0$ is the learning rate, it follows that :

$$\mathcal{L}^{t+1} \approx \mathcal{L}^t + \langle \nabla_\theta \mathcal{L}^t, \delta\theta \rangle \tag{2.4}$$

$$\approx \mathcal{L}^t - \eta \|\nabla_\theta \mathcal{L}\|^2 \tag{2.5}$$

When adding neurons to layer $l$, they set the fan-in weights of the new neurons to zero and compute the close form for the gradient norm of the new parameters. Using the notation of the right scheme of Figure 2.5, it follows that :

$$\left\| \frac{\partial \mathcal{L}^t}{\partial \boldsymbol{W}_l^{\text{new}}} \right\|^2 \propto \left\| \boldsymbol{W}_{l+1}^{\text{new}T} \mathbb{E}\left[ \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_{l+1}} \boldsymbol{h}_{l-1}^T \right] \right\|^2 \tag{2.6}$$

$$\left\| \frac{\partial \mathcal{L}^t}{\partial \boldsymbol{W}_{l+1}^{\text{new}}} \right\|^2 = 0 \tag{2.7}$$

| Dataset | Archi. | Reference | GradMax |
|---|---|---|---|
| CIFAR10 | WRN-28-1 | $92.09 \pm 0.2$ | $91.1 \pm 0.1$ |
| | VGG-11 | $86.6 \pm 0.3$ | $84.4 \pm 0.4$ |
| CIFAR100 | WRN-28-1 | $69.3 \pm 0.1$ | $66.8 \pm 0.2$ |
| ImageNet | Mobilenet-V1 | $70.8 \pm 0.0$ | $68.6 \pm 0.2$ |

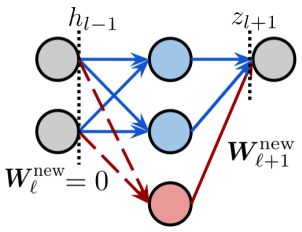



Figure 2.5: Left : Performance of GradMax on academic datasets from the table 1 of the original paper. Right : Adding one neuron at layer $l$ with GradMax method, figure from the original paper [Evci et al., 2022].

The fan-out weights of the new neurons are then set as the solution of the following minimization problem :

$$\underset{\boldsymbol{W}_{l+1}^{\text{new}}}{\arg\max} \left\| \boldsymbol{W}_{l+1}^{\text{new}T} \mathbb{E}\left[ \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_{l+1}} \boldsymbol{h}_{l-1}^T \right] \right\|^2 \quad \text{s.t.} \quad \left\| \boldsymbol{W}_{l+1}^{\text{new}} \right\| \leq c \tag{2.8}$$

where $\boldsymbol{h}_{l+1}$ and $\boldsymbol{z}_{l+1}$ are respectively the post-activation of layer $l-1$ and the pre-activation of layer $l+1$ and $c$ is a positive constant. Note that this initialization method might add neurons that are redundant with those of the current architecture, mathematically speaking: there is no guarantee that the function span of the new neurons and the function span of the current neurons are linearly independent, and this, because there is no such constraint in the minimization problem of Equation (2.8). We will see in Section 4.1.1 that such collinearity happens, especially when the gradient descent procedure has not yet converged (i.e. $\|\nabla_\theta \mathcal{L}\| \neq 0$).

In the original paper of GradMax [Evci et al., 2022], the authors propose a simple strategy on top of that initialization, which adds neurons in the order of the layers at regular time intervals of the standard training of the overall architecture. This growth method is tested on several academic datasets, and we note that the performance of the resulting architectures is, on average, 2.2 points of accuracy below the performance of the same architectures re-trained from scratch (the column Reference of left Table of Figure 2.5).

$\boxed{\mathcal{X}_\perp \mid 0}$ **NORTH or Random projection (RandomProj)**.

Like the previous method, the NORTH method of paper Maile et al. [2022] adds neurons by blocks to existing layers. Its initialization method is such that each increase of architecture expands some functional spaces defined by the network. They propose two initialization strategies where, in both, the fan-out weights are initialized to zero while the fan-in weights are set to random values, which are orthogonal either to the linear span of the pre-activation on a given minibatch or to the weight matrix. For each architecture increase, between 100 and 1000 neuron candidates are sampled from the associated kernel, and only the best one is added to the architecture. Along with that initialization, they propose two strategies for neuron additions. Each strategy relies on the use of a metric, which is computed after each gradient step at each layer. Each metric is itself related to a definition of the orthogonality of the current network, and neurons are added when that metric is high. The hypothesis is, that if the metric is low because of redundancy in a layer then, the neurons might benefit from differentiation via gradient descent before any architecture increase; while if the metric is high, there may be additional useful features to use, so neurons are added. In the paper, the methodology is tested over different academic datasets and, contrary to the GradMax method, the performances of the architectures found by such strategy (fig. 2.6) are equivalent to the performances of the same architectures retrained from scratch. This might be linked with the ability of the NORTH strategy to avoid redundancy between the added neurons and the current architecture.

### 2.1.4 Performances for gradient-based methods

We finish this literature review on NAS by presenting in Figure 2.6 the performances of the four recent one-shot and gradient-based methods (DARTS, Firefly,

GradMax, and NORTH) on three classic visual tasks (CIFAR-10, CIFAR-100 and ImageNet). For the Firefly, GradMax, and NORTH methods, we indicates the architecture as those methods do not construct the overall architecture but rather increase the size of existing layers; hence, the final architecture is similar to the one indicated in the architecture column of Figure 2.6.

We make the following remarks for each indicator $C_0$ the number of parameters of the network at the beginning of the search, $C_\infty$ the final number of parameters, $T$ the time of search and $P$ the accuracy on the test set with the final model :

- DARTS architecture final complexity ($C_\infty$) is not to be compared with the ones from the other methods because the search space of DARTS (cell-based) is different from the one used in Firefly, GradMax and NORTH approaches (chain-structured), resulting on highly dissimilar architectures. This variable alone is not sufficient to make a fair comparison between those, and an indicator such as the number of FLOPS or non-parallelized operations at test time would have been more suited.

- Comparison between the performances ($P$) and the complexity ($C_0$ and $C_\infty$) of GradMax and NORTH are consistent as both methods share a lot in terms of methodology, growth process, and neuron initialization. Nonetheless, we assume GradMax search time complexity $T_{GradMax}$ can be roughly estimated with $T_{NORTH}$. Indeed, NORTH generates either 100 or 1000 random candidate new neurons before each addition, and each generation time complexity is as $T_{GradMax}$. Therefore, one could divide NORTH search time complexity by 1000 and 100 to have rough estimates of $T_{GradMax}$.

- Although DARTS outperforms all methods in terms of accuracy, we note that its search time, $T$, is quite large (200 longer than NORTH), although we did not take into account the time searching for the basic cells on the CIFAR-10 dataset.

For the NORTH method, we used a by-hand average performance of the different initializations and strategies using the figure 4 of their paper.

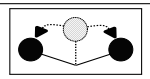
| Method | symbol | Indicators | | | | | Dataset |
|---|---|---|---|---|---|---|---|
| | | Architecture | $C_0$ | $C_\infty$ | T | P | |
| DARTS | $\mathcal{C} \backsimeq \mathcal{C}$ | × | × | $3.3e6$ | $1.5 + 2.5$ | $97.0 \pm 0.1$ | CIFAR-10 |
| | | × | × | $3.3e6$ | $4 + 2.5$ | $97.8 \pm 0.1$ | CIFAR-10 |
| | | × | × | $4.7e6$ | 4 | 73.3 | ImageNet |
| Firefly |  | VGG-19 | $1.4e6$ | $1.4e7$ | × | $\sim 94$ | CIFAR-10 |
| | | Mobilnet-V1 | × | $5.8e5$ | × | $\sim 71$ | CIFAR-100 |
| GradMax | $0 \mid \lVert\nabla\mathcal{L}\rVert^2$ | VGG-11 | $1e6$ | $9e6$ | × | $84.4 \pm 0.4$ | CIFAR-10 |
| | | WRN-28-1 | $1e5$ | $4e5$ | × | $91.1 \pm 0.1$ | |
| | | WRN-28-1 | $1e5$ | $4e5$ | × | $66.8 \pm 0.2$ | CIFAR-100 |
| | | Mobilenet-V1 | $1e6$ | $4.2e6$ | × | $68.6 \pm 0.2$ | ImageNet |
| NORTH | $\mathcal{X}_\perp \mid 0$ | VGG-11 | $1e6$ | $3e7$ | 3.3 | $\sim 76.2$ | CIFAR-10 |
| | | WRN-28-6 | $3e4$ | $3e5$ | 0.4 | $\sim 82.5$ | |
| | | VGG-11 | $1e6$ | $3e7$ | 1.6 | $\sim 55.2$ | CIFAR-100 |

Figure 2.6: Architecture : reference architecture, C : final complexity number as the number of parameters, T : GPU days, P : accuracy on test (%) by-hand mean on all tested strategy, see figure 4 of paper Maile et al. [2022] for more precise results.

## 2.2 Expressivity and Complexity

In this last part of the literature review, we present some NAS evaluation performance strategies (2.1.1) and the standard mathematical tools quantifying a network's complexity or expressivity. In fact, those two share the same objective, that is, estimating the performance of a network given its architecture. However, while they are very alike, the evaluation performance strategy is empirically based, while the mathematical tools are, by definition, theoretically well grounded. As a matter of fact, evaluation performance strategies have used empirical observations rather than mathematical concepts because, in the context of NAS, they should be scalable to high-dimensional data and compatible with the usual optimization procedure. Yet, it turns out that all existing theories, either do not combine with gradient descent (that is, the classical optimization procedure), either demonstrate performance properties on infinite networks, which is not an affordable setting in practice when the memory and the computational power are limited , and finally, they are not applicable within a reasonable time. Nonetheless, even if such concepts have not been quite used in this specific subfield of NAS, they had an impact elsewhere in the field of Machine learning; let us only cite Jacot et al. [2018] on generalization with the Neural Tangent Kernel, which has changed our bias on well-functioning architectures, justifying the consideration of huge designs for a preferred search space for all applications.

We start this chapter by presenting the classical evaluation performance strategies of NAS; then, in a second part, we present the theoretical tools that link a **finite** neural network architecture to its ability to perform well on a dataset. While those mathematical tools are yet not used in NAS, they are instead an inspiration for future works and a comparison for our new metric *Expressivity Bottleneck*. Note that this second section does not present any theoretical asymptotical results, as, so far and by definition, they do not provide any guidance to search within the space of architectures $\mathbb{A}$ (in particular thin ones), nor do they provide a metric to compare architectures between each other.

### 2.2.1 Empirical methods

As defined in Section 2.1.1, evaluation performance strategies provide feedback to the search strategy by evaluating the performance of the sampled architectures. It is clear that training the sampled architecture by gradient descent provides a nice estimate of its performance, but this procedure becomes rapidly exhaustive and unfeasible when dealing with complex datasets, as they require large architectures with high memory and computational cost when training. For that reason, there exist evaluation performance strategies that estimate the final performance of an architecture without this costly operation. We separate them into three main categories: performance interpolation, lower estimates, and speed-up techniques.

Speed-up techniques are all the methods accelerating the training of a network; they can be coupled with another evaluation performance strategy. They are optimization tricks rather than tools for estimating an architecture performance. Thus, we consider them as out of the scope of this thesis and do not mention them. An overview of such speed-up techniques can be found in the following surveys: Elsken et al. [2019], White et al. [2023].

**Performance interpolation.** Those methods apply a classic Machine Learning model to predict the performance of unseen architectures. One method consists in training shortly some architectures and in interpolating the rest of their learning curves by fitting a chosen parametric model [Domhan et al., 2015], or a Bayesian neural network [Klein et al., 2017]. Another technique that has already been discussed in the Bayesian search strategy Section 2.1.3 is to use a surrogate model. In this setting, the next architecture is given by maximizing the surrogate, that is, maximizing the expectation of the next sampled network's performance.

**Lower estimate.** Instead of predicting the network's final performance with all the available information and a complete optimization procedure, the lower estimate strategies use degraded information or a partial optimization process. One strategy consists of optimizing the network on a reduced dataset; the points selected for that training can be randomly extracted from the original dataset, smartly chosen Na et al. [2021], or generated by a Generative Teaching Network Such et al. [2020]. Another strategy is to downgrade the quality of the input (by averaging the pixels for images or lowering the registering frequency for a sound) and to perform training on such a degraded dataset. A third strategy, known as early stopping, evaluates the performance of a network after a short time of classical training while the learning curve has not yet converged Zhong et al. [2018].

### 2.2.2 Theoretical methods

In this section, we present the literature review on the concept of *expressivity* and *complexity*, which are formal metrics that link a finite neural network architecture and its ability to perform well on some datasets. Those measures are of great interest because they are closely related to the generalization properties of networks, and in particular, they upper-bound the generalization error, also known as the out-of-sample error or the *true* error of the model. Still, those metrics are difficult to evaluate, and for that reason, they are never computed nor used in practice.

We start the section by introducing some notations that will be used to define each metric; then, we describe the VC-dimension, the Rademacher complexity, the information bottleneck, and the Kolmogorov complexity.

### Notations

Definition 2.2.1 (Loss function). We note the loss function $\mathcal{L} : \left(\mathbb{R}^d\right)^2 \to \mathbb{R}^+$. For a fixed dataset $\mathcal{D} := \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^n$ and a function $f$, we note $\mathcal{L}(f(\mathcal{D})) := \frac{1}{n}\sum_i \ell(f(\boldsymbol{x}_i), \boldsymbol{y}_i)$, where $\ell$ is the loss function for one sample. When there is no confusion, we might use $\mathcal{L}(f) := \mathcal{L}(f(\mathcal{D}))$.

Definition 2.2.2 (Random Variable). We note $\boldsymbol{x} \sim \mathcal{D}$, the random vector variable that follows the distribution law of $\mathcal{D}$.

Definition 2.2.3 (Feed-forward network). We note $f_\theta : \mathbb{R}^p \to \mathbb{R}^d$ a feed-forward network with $L$ hidden layers, the application which maps $\boldsymbol{x} \in \mathbb{R}^p$ to $f_\theta(\boldsymbol{x})$ and which consists in the iterative application of affine layers with weights $\boldsymbol{W}_l$ followed by a activation function $\sigma_l$. Formally, the network parameters are $\theta := (\boldsymbol{W}_l)_{l=1...L}$, and the function $f_\theta$ iteratively computes :

$$
\begin{aligned}
\boldsymbol{b}_0(\boldsymbol{x}) &= \begin{pmatrix} \boldsymbol{x} \\ 1 \end{pmatrix} \\
\forall l \in [1, L], \quad &\begin{cases} \boldsymbol{a}_l(\boldsymbol{x}) &= \boldsymbol{W}_l\, \boldsymbol{b}_{l-1}(\boldsymbol{x}) \\ \boldsymbol{b}_l(\boldsymbol{x}) &= \begin{pmatrix} \sigma_l(\boldsymbol{a}_l(\boldsymbol{x})) \\ 1 \end{pmatrix} \end{cases} \\
f_\theta(\boldsymbol{x}) &= \sigma_L(\boldsymbol{a}_L(\boldsymbol{x}))
\end{aligned}
$$

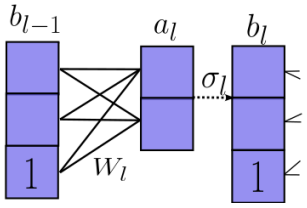



Figure 2.7: Notations

### The Vapnik–Chervonenkis (VC-dimension)

The VC-dimension is defined for a set of binary functions and can be seen as the ability of this set to fit random noise. Formally, consider the set of functions $\mathcal{C} := \{f : \mathbb{R}^p \mapsto \{0, 1\} \mid f \in \mathcal{F}\}$ for some $\mathcal{F}$, the VC-dimension of $\mathcal{F}$, $\mathrm{VC}(\mathcal{C})$, is defined as the natural number $n$, such that, there exists a set $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\} \in \{\mathbb{R}^p\}^n$ such that for any binary set $\{b_1, ..., b_n\} \in \{0, 1\}^n$ there exists a function $f \in \mathcal{C}$ which satisfies $f(\boldsymbol{x}_i) = b_i$. That is, there exists a $n$ set of inputs such that, whatever the labels of those inputs, there exists a function in $\mathcal{C}$ that maps those inputs to such labels. We give an example of this definition in Figure 2.8 for the set of linear separators in a plane.

Considering a neural network, a straightforward application of that definition is to consider a fixed architecture $\mathcal{A}$ and the set of functions described by the different initialization of $\mathcal{A}$, ie $\mathcal{C} = \{f_\theta \mid \theta \in \Theta\}$ for some $\Theta$. Saying that $\mathrm{VC}(\mathcal{C}) = n$, is equivalent to state that there exists $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\} \in \{\mathbb{R}^p\}^n$ such that, for any dataset of size $n$ noted $\{\tilde{\boldsymbol{x}}_i, y_i\}_{i=1}^n$ where $y_i \in \{0, 1\}$, there exists a $\theta^{\mathcal{D}} \in \Theta$, for which the prediction of the network $f_{\theta^{\mathcal{D}}}$ is exact if one considers a morphism between the

input data $\{\tilde{\boldsymbol{x}}_1, ..., \tilde{\boldsymbol{x}}_n\}$ and the points $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$. By doing so, we completely overfit the dataset.

We now state a result demonstrated in 1989 by Baum [1988] on the estimation of the VC dimension for multilayer neural networks :

**Proposition 2.2.4.** *The class of functions computed by multilayer neural networks with binary activation functions and $\rho$ weights has VC dimension in $O\big(\rho log(\rho)\big)$.*

This bound is tight in the sense that there exists configurations of multilayer perceptron for which the VC-dimension is proportional to $\rho log(\rho)$ (Maass [1994], Sakurai [1995]).

We now study the link between the VC-dimension and the generalization property of networks. For a given dataset $\mathcal{D}$ and a set of neural networks $\mathcal{C}$, the quantity of interest for generalization is the difference between the minimizer of the loss on the true distribution of the dataset as:

$$f^* := \arg\min_{f \in \mathcal{C}} \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}[\ell(f(\boldsymbol{x}), \boldsymbol{y})] \tag{2.9}$$

And the minimizer of the loss on the dataset $\mathcal{D}$ that is :

$$\hat{f} := \arg\min_{f \in \mathcal{C}} \mathcal{L}(f(\mathcal{D})). \tag{2.10}$$

We are interested in the excess of risk, that is :

$$l(f^*, f) := \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\ell(f(\boldsymbol{x}), \boldsymbol{y})\right] - \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\ell(f^*(\boldsymbol{x}), \boldsymbol{y})\right] \tag{2.11}$$

In statistics this error is decoupled into two terms, the estimation and approximation errors, also known as the bias-variance decomposition :

$$l(f^*, f) = \overbrace{l(f^*, f) - l(f^*, \mathcal{C})}^{\text{estimation error}} + \overbrace{l(f^*, \mathcal{C})}^{\text{approximation error}} \tag{2.12}$$

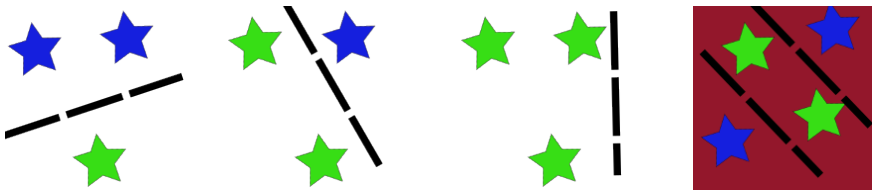


Figure 2.8: Consider a classification model on points in a two-dimensional plane. The line should separate positive data points from negative data points. There exist sets of 3 points that can indeed be correctly classified using this model (any 3 points that are not collinear can be correctly classified). However, no set of 4 points can be correctly classified for any possible labeling. Thus, the VC-dimension of this set of functions is 3.

where $l(f^*, \mathcal{C}) := \min_{f \in \mathcal{C}} l(f^*, f)$.

The approximation error is how the set $\mathcal{C}$ approximates the task of dataset $\mathcal{D}$, while the estimation error quantifies how far our estimator $\hat{f}$ is close to the ideal estimator in $\mathcal{C}$, that is $f^*$. For neural networks, and especially when the number of parameters is large, the approximation error goes toward 0 [Hornik et al., 1989]. Still, we keep this term in the equation and now study the estimator error that is equal to :

$$l(f^*, f) - l(f^*, \mathcal{C}) \tag{2.13}$$

We finish this subpart by an upper-bound on the generalization error (Vapnik-Chervonenkis inequality of section 3 of Boucheron et al. [2005]):

**Proposition 2.2.5.** *Let $\mathcal{C}$ a function set and $\mathcal{D}$ a dataset of size $n$, then*

$$l(f^*, f) - l(f^*, \mathcal{C}) \leq \kappa \sqrt{\frac{\mathrm{VC}(\mathcal{C})}{n}} \tag{2.14}$$

*with $\kappa$ a universal constant.*

Before commenting on this last property, we should present the Rademacher complexity, which is a notion close to VC-dimension and which appears in a proposition whose formulation is close to Equation (2.14).

### Rademacher complexity

Rademacher complexity is a more modern notion that is, in comparison with the VC dimension, input distribution dependent[2] and defined for any class real-valued function. Like the VC-dimension, the Rademacher complexity is defined for a set of function $\mathcal{C}$, and it measures the expected noise-fitting-ability of $\mathcal{C}$ over a distribution of the dataset $\mathcal{D}$. Formally, let $\{\boldsymbol{x}_i, y_i\}_{i=1}^n \overset{iid}{\sim} \mathcal{D} =$ and $\sigma_i \sim \mathcal{B}(\frac{1}{2})$[3], then the Rademacher complexity is defined as :

$$\mathcal{R}(\mathcal{C}) := \mathbb{E}_{\mathcal{D}} \left[ \frac{1}{n} \mathbb{E}_{\sigma_1, \ldots, \sigma_n} \left[ \sup_{f \in \mathcal{C}} \sum_{i=1}^{n} \sigma_i f(\boldsymbol{x}_i) \middle| \mathcal{D} \right] \right] \tag{2.15}$$

where the second expectation is over the distribution of the dataset $\mathcal{D}$, that is a random variable. This complexity can be seen as the sup over the scalar product between the vectors of output $f(\boldsymbol{X}) := (f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_n))^T$ and the vector variable $\boldsymbol{\sigma} := (\sigma_1, ..., \sigma_n)^T$, such as :

$$\mathcal{R}(\mathcal{C}) := \mathbb{E}_{\mathcal{D}} \left[ \frac{1}{n} \mathbb{E}_{\boldsymbol{\sigma}} \left[ \sup_{f \in \mathcal{C}} \boldsymbol{\sigma}^T f(\boldsymbol{X})) \middle| \mathcal{D} \right] \right] \tag{2.16}$$

[2] its value depends on the considered task

[3] $\mathcal{B}$ is the Bernoulli distribution

Consider the set of functions $\mathcal{C}$ defined by feed forward neural networks $f : \mathbb{R}^m \to \mathbb{R}$ with weights $\boldsymbol{W}_l$ bounded by $\omega$, having $\lambda-$Lipschitz activation functions and $L$ depths, the following proposition holds:

**Proposition 2.2.6.** *Suppose that the inputs $\boldsymbol{x}$ are such that $\|\boldsymbol{x}\|_\infty := \sup_{i=1,\dots,p} x[i]$ is bounded by 1 and note $n$ the size of the dataset, then*

$$\mathcal{R}(\mathcal{C}) \leq \frac{1}{\sqrt{n}} \left( \omega + 2\omega^2\lambda \sum_{k=0}^{L-3} (2\omega\lambda)^k + 2\omega(2\omega\lambda)^{L-2} + \sqrt{2\log(2m)} \right) \tag{2.17}$$

The demonstration can be found in Rebeschini [2022] proposition 3.6. Furthermore, the Rademacher complexity upper bounds the risk minimization error as ( see Arlot [2020] section 3.7.4)

$$l(f^*, f) - l(f^*, \mathcal{C}) \leq 2\mathcal{R}(\ell(\mathcal{C})) \tag{2.18}$$

where $\ell(\mathcal{C}) := \{\ell(f(.), .) : (\boldsymbol{x}, \boldsymbol{y}) \mapsto \ell(f(\boldsymbol{x}), \boldsymbol{y}) \mid f \in \mathcal{C}\}$.

Although being differently defined, the VC-dimension and the Rademacher complexity capture the ability of a set of functions to fit noise. However, while a set of functions might be very expressive or complex (with a relatively large VC-dimension or Rademacher complexity), no general algorithm nor initialization method exists to reach or even approach the most expressive function inside this set. The limits of Equation (2.14) and Equation (2.18) do not depend on the labels $y$ and ensure good generalization properties if the number of examples is comparable or greater than the complexity of the model. According to such modeling, the model should be sufficiently large to ensure that the approximation error of **??** is zero but not too large compared to the size of the dataset $\mathcal{D}$ to ensure a good approximation error.

We now define the bottleneck as defined in information theory.

### Information Bottleneck and information theory

The bottleneck metric defines an inverse of expressivity by quantifying the *inability* of the network to extract relevant information from its inputs. This metric uses the conditional mutual information function, noted $I$, which is defined with tools and definitions from information theory (the Shannon entropy, the mutual information, and the conditional mutual information). This function $I$ can be computed between two random variables $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$, we note it $I(\boldsymbol{x}; \hat{\boldsymbol{x}})$, and it calculates the reduction in the uncertainty of $\boldsymbol{x}$ due to the knowledge of $\hat{\boldsymbol{x}}$[4]. Considering a neural network $f$ and a dataset $\mathcal{D} = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_n$, the information bottleneck metric, $IB$, is a trade off between the mutual information of the input $\boldsymbol{x}$ and its internal

[4]The mutual information is constructed on the Shannon entropy $H(\boldsymbol{x})$ that is also referred as the *self-information* of a random variable $\boldsymbol{x}$. Formally $I(\boldsymbol{x}; \hat{\boldsymbol{x}}) := H(\boldsymbol{x}) - H(\boldsymbol{x}|\hat{\boldsymbol{x}})$.

representation $\hat{\boldsymbol{x}}$, and the mutual information of the internal representation $\hat{\boldsymbol{x}}$ and the desired output $\boldsymbol{y}$. Mathematically speaking :

$$IB(f) := I(\boldsymbol{x}, \hat{\boldsymbol{x}}) - \beta I(\hat{\boldsymbol{x}}, \boldsymbol{y}) \tag{2.19}$$

where $\beta$ is a fixed hyper parameter. The article Tishby and Zaslavsky [2015], which introduces it, proposes to minimize such criterion instead of the conventional loss because a well-functioning neural network would efficiently compress and reduce the information of $\boldsymbol{x}$ (equivalently reducing $I(\boldsymbol{x}, \hat{\boldsymbol{x}})$) while selecting the relevant information to predict $\boldsymbol{y}$ (which is equivalent to maximize $I(\hat{\boldsymbol{x}}, \boldsymbol{y})$). While such criterion might be a good quantification of the *inexpressivity* of a network and, it was also proved to be an effective optimization procedure when $\boldsymbol{y}$ is a categorical data, as $\exp(-I(\hat{\boldsymbol{x}}, \boldsymbol{y}))$ upper bounds the probability of misclassifying $\boldsymbol{x}$ and $I(\boldsymbol{x}, \hat{\boldsymbol{y}})$ plays the role of a regularization term [Shamir et al., 2010].

### Kolmogorov Complexity

We finish this section with a note on Kolmogorov complexity, a notion of complexity that was invented in the sixties and named after its creator. The Kolmogorov complexity of a mathematical object is its algorithm complexity, which is the length of the shortest binary computer program that describes that object. In the context of NAS and for a given dataset, one can search for a neural network that correctly maps all the inputs of such dataset to their outputs but whose Kolmogorov complexity is relatively low. Such a network should less overfit, following the Solomonoff's prior which consider that the best possible scientific model is the shortest algorithm. However, suppose the Kolmogorov complexity would ingeniously guide a NAS search strategy; the fact remains that such complexity is impossible to calculate and even less differentiable, making its use impossible in a NAS search strategy.

An alternative to the Kolmogorov complexity, especially in machine learning, is to consider the Minimum Description Length (MDL). Compared to Kolmogorov complexity, the MDL is a training criterion which regroups in one expression the error of the model and its complexity. A well-known MDL criterion is the Bayesian information criterion (BIC) that quantifies the model complexity by the number of its parameters and the size of the dataset. Minimizing the BIC criterion during training would enhance the model prediction but also orient the search toward a network with "few" parameters, and consequently that would have a short binary encoding that, in this context, often induces a small architecture and a short execution time[5]. In fact, those properties (compactness and generalization) are

[5]The Kolmogorov complexity of a standard feed-forward network with ReLU activation function is at most proportional to the memory of its parameters because the operations computed through its layers are identical and can be encoded by a single function whose Kolmogorov complexity is fixed and independent of the network configuration. Note, that when the weights of the network are shared through its layers, this statement is not true as coding the repetition of the same operation has a small BIC but a long execution time.

not straightforward in NAS because classical theories, such as the approximation theorem and the Neural Tangent Kernel [Hornik et al., 1989, Jacot et al., 2018], indicate that we would prefer to search for large architectures because they are more likely to generalize well and achieve full expressivity. Note that the MDL theory does not care about the optimization properties of the model training, it just provides a criterion to optimize.

## Conclusion

As stated in the introduction, the mathematical tools evaluating neural networks' complexity or expressivity are hardly usable. Firstly, because none of them is easily computable; second, because they do not provide any useful guidance or information on what type of architecture should be used or sampled at the next step of the search strategy. In fact, those metrics can be used to compare the expressivity or the bottleneck of different architectures, but, without a trial and error process, they do little to propose what architecture modifications should be performed to enhance the performances.

# The problem statement

Starting from the beginning, we saw that there exists multiple NAS techniques and that most of them rely on the trial and error method, which is embodied by the back and forth between the search and evaluation performance strategy. This time-consuming process mainly stems from the decoupling between the search into the space of architectures $\mathbb{A}$ and the space of parameters in Equation (2.3), as it requires sampling and training multiple architectures. However, a new category of methods, called the one-shot techniques, avoids such costly processes by constructing and training a unique architecture during the search. In fact, those methods achieve, in a certain way, the joint optimization of the network architecture and its parameters, transforming the two-step optimization procedure of Equation (2.3) into a unique and conventional optimization problem. In particular, the GradMax and the NORTH methods can be applied to any task, and they can grow architecture indefinitely.

The GradMax method initializes its new neurons by maximizing the norm of those new neurons parameters, while the NORTH method initializes them by expanding the functional span of the network, paying attention to redundancy within the current architecture. By construction, but also according to Section 2.1.4, the GradMax approach is the fastest. However, its added neurons might be redundant with the current architecture, which might explain why the method obtains lower performance for a fixed target architecture (Figure 2.5). On the other hand, the NORTH method is slower because its procedure still relies on random trials of neuron additions (without further training), but the final performances of its constructed networks are equal to the performances of the same architectures retrained from scratch. This observation gives rise to two remarks. First, adding neurons greedily to the network is an effective way of growing its architecture; second, to achieve high performance with a small network we might need to care about redundancy within the architecture. Ideally, we would like to construct an adding strategy that is as fast as the GradMax method and also as effective in performance as the NORTH method.

To achieve this successful combination, we aim at constructing a metric measuring the lack of expressivity of a network, which ranks among the existing theoretical metrics but that is easily computable and scalable to high dimensional datasets.

*For the next chapters I continue using the pronoun "we". It should be clear to the reader that those next chapters reflect my research and my approach on the subject; however, its realization would have been impossible without the active collaboration of my directors and my collaborators, which are, partly and implicitly, included in that pronoun "we".*

### Contributions

In this thesis, we aim at bridging the gap between the GradMax and the NORTH method, but also between theory and application, by constructing a new metric of expressivity that relies on mathematical tools and that is easily computable. We define the metric **Expressivity Bottleneck** as the difference between what the backpropagation asks for and what can be done by a small parameter update (such as a gradient step), that is, between the desired variation for each activation in each layer (for each sample) and the best variation that can be realized by a parameter update. We show that this metric can be easily computed from backpropagation, which is an inexpensive and straightforward computation. We show, using an optimization trick, that on top of quantifying lack of expressivity, this metric can also solve a network bottleneck by adding suitable neurons that are orthogonal to the current architecture, and this, during the architecture training. On this basis, we define a novel NAS search method and propose an incremental algorithm that iteratively grows the network until an appropriate architecture is found.

The contributions of this manuscript are manifold and could be described as follows. First, considering a neural network architecture, we adopt a functional analysis perspective on gradient descent in this architecture. We not only optimize the weights of the current architecture but also dynamically adjust the architecture itself to progress towards suitable parameterized functional spaces. This approach mitigates optimization challenges like local minima due to thin architectures.
For this, we properly define and quantify the concept of expressivity bottlenecks, both globally at the neural network output and locally at individual layers, in a computationally accessible manner. This methodology enables the localization of expressivity bottlenecks within a neural network.
Then, we formally define as a quadratic problem the best possible neurons to add to a given layer to decrease the lack of expressivity, solve it, and compute the associated expressivity gain.
Also, we provide tools to adapt the architecture to the specific task by expanding it where necessary within a single run, maintaining competitive computational complexity compared to training a large model once. To remove the need for hyper-optimization of layer width, one could specify a target accuracy and stop neuron additions when reached.
Furthermore, we prove (empirically and theoretically) that a simple one-shot strat-

egy on the basis of that new metric reaches zero error on the training set.
Moreover, we compare the expressivity bottleneck method with GradMax and NORTH and prove, without any additional hypothesis, that those two one-shot strategies can be seen as a particular application of the Expressivity Bottleneck method.
Finally, we implemented a public module named **TINYpub** which starts from a thin architecture and grows linear and convolutional layers on the basis of that theory.

Those contributions are presented in different parts of the manuscript, from Chapter 3 to Chapter 5.

In Chapter 3, we define the expressivity bottleneck metric as the minimization of a quadratic problem. In particular, we show that, for a neural network, solving this minimization problem naturally determines updates of parameters and increases of architecture, which are orthogonal to each other. We prove that a naive strategy on the basis of such an update of architecture converges to zero loss on the training set. In Chapter 4, we reformulate GradMax and NORTH methods with our formalism to clarify and understand the «philosophy » of each method from the expressivity bottleneck point of view. Finally, in Chapter 5, we construct a naive NAS strategy on the basis of the expressivity bottleneck metric and apply such strategy to academic datasets.

A major part of the following sections has been published in the TMLR journal [Verbockhaven et al., 2024]. The corresponding sections are indicated with the icon  on their titles.

# 3 Expressivity Bottleneck

**Contents**

In this Chapter 3, we define the expressivity bottleneck metric and provide tools to construct a search strategy on the basis of such metric. We start in Section 3.1, by recalling the notations and providing some intuitions on the expressivity bottleneck metric; then, in Section 3.2, we properly define the expressivity bottleneck metric and show how it can be used to compute the best update of the current architecture and the new neurons to add. This chapter ends with the Section 3.3, which provides some convergence properties for NAS search strategies using the architecture updates of the previous section.

## 3.1 Definitions and Intuitions

In this section, we start by recalling the notations, the main definitions, and the assumptions of this thesis. In a second part Section 3.1.2, we propose to merge the two-step optimization procedure of Equation (2.3) with a change of scalar product. We show in Section 3.1.3 that this change naturally defines the projected functional gradient, which is the starting point of our expressivity bottleneck metric.

### 3.1.1 Notations

For the sake of simplicity, the objects of the main sections are defined only for linear feed-forward networks, while the equivalence for convolutional layers is indicated with a reference to the annex. Let $\mathcal{F}$ be a functional space, e.g. $L_2(\mathbb{R}^p \to \mathbb{R}^d)$, and $\mathcal{L} : \mathcal{F} \to \mathbb{R}^+$ a loss function defined on it, and of the form $\mathcal{L}(f) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\ell(f(\boldsymbol{x}), \boldsymbol{y})\Big]$, where $\ell$ is the per-sample loss that is assumed to be differentiable, and where $\mathcal{D}$ is the sample distribution, from which the dataset $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ is sampled, with $\boldsymbol{x}_i \in \mathbb{R}^p$ and $\boldsymbol{y}_i \in \mathbb{R}^d$.

Let consider a network $f_\theta : \mathbb{R}^p \to \mathbb{R}^d$ with $L$ hidden layers, each of which consisting of an affine layer with weights $\boldsymbol{W}_l$ followed by a differentiable activation function $\sigma_l$ which satisfies $\sigma_l(0) = 0$. The network parameters are then $\theta := (\boldsymbol{W}_l)_{l=1\dots L}$, and the network iteratively computes:

$$
\begin{aligned}
\boldsymbol{b}_0(\boldsymbol{x}) &= \begin{pmatrix} \boldsymbol{x} \\ 1 \end{pmatrix} \\
\forall l \in [1, L], \quad &\begin{cases} \boldsymbol{a}_l(\boldsymbol{x}) &= \boldsymbol{W}_l\, \boldsymbol{b}_{l-1}(\boldsymbol{x}) \\ \boldsymbol{b}_l(\boldsymbol{x}) &= \begin{pmatrix} \sigma_l(\boldsymbol{a}_l(\boldsymbol{x})) \\ 1 \end{pmatrix} \end{cases} \\
f_\theta(\boldsymbol{x}) &= \sigma_L(\boldsymbol{a}_L(\boldsymbol{x}))
\end{aligned}
$$

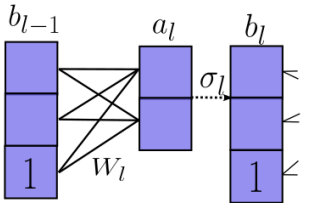



Figure 3.1: Notations

To any vector-valued function noted $\boldsymbol{t}(\boldsymbol{x})$ and any batch of inputs $\boldsymbol{X} := [\boldsymbol{x}_1, ..., \boldsymbol{x}_n]$, is associated the concatenated matrix $\boldsymbol{T}(\boldsymbol{X}) := \begin{pmatrix} \boldsymbol{t}(\boldsymbol{x}_1) & \dots & \boldsymbol{t}(\boldsymbol{x}_n) \end{pmatrix} \in$

$\mathbb{R}^{|\boldsymbol{t}(.)|\times n}$. The matrices of pre-activation and post-activation activities at layer $l$ over a minibatch $\boldsymbol{X}$ are thus respectively: $\boldsymbol{A}_l(\boldsymbol{X}) = \begin{pmatrix}\boldsymbol{a}_l(\boldsymbol{x}_1) & ... & \boldsymbol{a}_l(\boldsymbol{x}_n)\end{pmatrix}$ and $\boldsymbol{B}_l(\boldsymbol{X}) = \begin{pmatrix}\boldsymbol{b}_l(\boldsymbol{x}_1) & ... & \boldsymbol{b}_l(\boldsymbol{x}_n)\end{pmatrix}$.

Convolutions can also be considered, with appropriate representations (cf matrix $\boldsymbol{b}_l^c(\boldsymbol{x})$ in Theorem C.1.1).

### 3.1.2 Changing scalar product

As a starting point, let first recall the problem statement of NAS as defined in Equation (2.3). For a given dataset, NAS can be formulated as the following optimization problem :

$$\arg\min_{\mathcal{A}\in\mathbb{A}} \mathcal{L}_{val}(\mathcal{A}(\theta^*)) \qquad s.t. \quad \theta^* := \arg\min_{\theta} \mathcal{L}_{train}(\mathcal{A}(\theta)) \tag{3.1}$$

where $\mathcal{L}_{val}$ is the loss on the validation set and $\mathcal{L}_{train}$ is the loss on the train set. Ideally, we would jointly optimize the architecture $\mathcal{A}$ and its weights $\theta$ by defining a unique optimization procedure. This would remove the performance evaluation strategy from the search process, saving both time and computational resources. To find such a procedure, we draw inspiration from the well-known gradient-descent algorithm that is performed when one wants to find a local minimum of a differentiable loss function $\mathcal{L} : U \to \mathbb{R}$ on $U$ where $(U, \langle .\ ,.\ \rangle_\dagger)$ is an Hilbert space. The algorithm starts from a random $u^0 \in U$ and increments it $(t \to t+1)$ using the update rule :

$$u^{t+1} = u^t - \eta \nabla_u^\dagger \mathcal{L}(u)_{|u=u^t} \tag{3.2}$$

where $\eta > 0$ is the gradient step and $\nabla_\cdot^\dagger \mathcal{L}(.)$ is the gradient of $\mathcal{L}$ that is defined with the scalar product $\langle .\ ,.\ \rangle_\dagger$ and which satisfies for $u \in U$ (Lecué [2016]):

$$\delta u \in U, \quad \mathcal{L}(u+\delta u) = \mathcal{L}(u) + \left\langle \nabla_u^\dagger \mathcal{L}(u), \delta u \right\rangle_\dagger + o(\|\delta u\|_\dagger) \tag{3.3}$$

$||.||_\dagger$ being the norm associated to the scalar product $\langle ., . \rangle_\dagger$. In fact, the gradient descent objective, that is minimizing the loss, can be inferred again from Equation (3.3) as choosing $\delta u = -\eta \nabla_u^\dagger \mathcal{L}(u)$ is equivalent to minimizing the first order development of $\mathcal{L}$ in $u$ [1].

In practice, when trying to fit a network on a task by minimizing the loss function $\mathcal{L}$, the network architecture is fixed, and one choose to optimize $\mathcal{L}$ on the architecture weight $\Theta$, that is, one chose $U = \Theta$ and $\left\langle \delta\theta^A, \delta\theta^B \right\rangle := \sum_i \delta\theta_i^A \delta\theta_i^B$ for $\delta\theta^A, \delta\theta^B \in \Theta$. However, this way of proceeding forces the evolution of the function during training to lie within the realm of what is expressible with the chosen architecture and prevents any optimization across architectures. To address this

[1] For a fixed norm of $\delta u$, the scalar product $\left\langle \nabla_u^\dagger \mathcal{L}(u), \delta u \right\rangle_\dagger$ is minimal when $\delta u$ aligns with $-\nabla_u^\dagger \mathcal{L}(u)$.

issue, we change the space of interest and consider $U = L_2(\mathbb{R}^p \to \mathbb{R}^d)$ as the square-integrable function and the scalar product defined as $\langle f_1, f_2 \rangle_{L_2} := \int_{x \in \mathbb{R}} [f_1 \cdot f_2]$. Doing so, we take a functional perspective on the use of neural networks and search for a function $f : \mathbb{R}^p \to \mathbb{R}^d$ that minimizes the loss $\mathcal{L}$ by gradient descent: $\frac{\partial f}{\partial t} = -\nabla_f^{L_2} \mathcal{L}(f)$, where $\nabla_f^{L_2}$ denotes the functional gradient for the scalar product $\langle ., . \rangle_{L_2}$ and $t$ denotes the evolution time of the gradient descent. we define this direction as $\boldsymbol{v}_{\text{goal}} := -\nabla_f^{L_2} \mathcal{L}(f)$ that is a function of the same type as $f$ and whose value at $\boldsymbol{x}$ is easily computable as $\boldsymbol{v}_{\text{goal}}(\boldsymbol{x}) = -\left(\nabla_f^{L_2} \mathcal{L}(f)\right)(\boldsymbol{x}) = -\nabla_{\boldsymbol{u}} \ell(\boldsymbol{u}, \boldsymbol{y}(\boldsymbol{x}))\big|_{\boldsymbol{u}=f(\boldsymbol{x})}$ (see Appendix B.1 for more details). This direction $\boldsymbol{v}_{\text{goal}}$ is the best infinitesimal variation in $\mathcal{F}$ to add to $f$ to decrease the loss $\ell$, ie for all $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}$, performing $f_\theta \leftarrow f_\theta + \eta \boldsymbol{v}_{\text{goal}}$ is such that :

$$\ell\left((f_\theta + \eta \boldsymbol{v}_{\text{goal}})(\boldsymbol{x})\right) = \ell(f_\theta(\boldsymbol{x})) - \eta \left\| \nabla_{\boldsymbol{u}} \ell(\boldsymbol{u})_{|\boldsymbol{u}=f_\theta(\boldsymbol{x})} \right\|^2 + o(\eta) \tag{3.4}$$

where $||.||$ is the Euclidean norm. One can deduce that, for a fixed dataset $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, that incremental $\boldsymbol{v}_{\text{goal}}$ decreases the averaged loss as follows :

$$\mathcal{L}(f_\theta + \eta \boldsymbol{v}_{\text{goal}}) := \frac{1}{n} \sum_{i=1}^n \ell((f_\theta + \eta \boldsymbol{v}_{\text{goal}})(\boldsymbol{x}_i)) \tag{3.5}$$

$$= \mathcal{L}(f_\theta) - \frac{\eta}{n} \left( \sum_{i=1}^n \left\| \nabla_{\boldsymbol{u}} \ell(\boldsymbol{u})_{|\boldsymbol{u}=f_\theta(\boldsymbol{x}_i)} \right\|^2 \right) + o(\eta) \tag{3.6}$$

while the usual gradient $(t \to t+1)$ with $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(f_\theta)$ and starting from Equation (3.3) decreases the loss differently:

$$\mathcal{L}(f_{\theta - \eta \nabla_\theta \mathcal{L}(f_\theta)}) = \mathcal{L}(f_\theta) - \eta \left\| \nabla_\theta \left( \frac{1}{n} \sum_{i=1}^n \ell(f_\theta(\boldsymbol{x}_i)) \right) \right\|^2 + o(\eta) \ . \tag{3.7}$$

For the following sections, we will use the abuse of notation that $\|\boldsymbol{u}\|$ is either the norm in $L_2$ or the usual Euclidean norm, depending on the type of $\boldsymbol{u}$.

### 3.1.3 Parametric gradient descent reminder and optimal move direction.

However, in practice, to represent functions and to compute gradients, the infinite-dimensioned functional space $\mathcal{F}$, to which $\boldsymbol{v}_{\text{goal}}$ belongs, has to be replaced with a finite-dimensioned parametric space of functions, which is usually done by choosing a particular neural network architecture $\mathcal{A}$ with weights $\theta \in \Theta_{\mathcal{A}}$. The associated parametric search space $\mathcal{F}_{\mathcal{A}}$ then consists of all possible functions $f_\theta$ that can be represented with such a network for any parameter value $\theta$. For the usual gradient descent and under standard weak assumptions (see Appendix B.2), the gradient descent is of the form:

$$\frac{\partial \theta}{\partial t} = -\nabla_\theta \mathcal{L}(f_\theta) = -\mathbb{E}_{(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}} \Big[ \nabla_\theta \ell(f_\theta(\boldsymbol{x}), \boldsymbol{y}) \Big] \ . \tag{3.8}$$

Using the chain rule (on $\frac{\partial f_\theta}{\partial t}$ then on $\nabla_\theta \ell(f_\theta(\boldsymbol{x}), \boldsymbol{y})$), these parameter updates yield a functional evolution:

$$\boldsymbol{v}_{\text{GD}} \;:=\; \frac{\partial f_\theta}{\partial t} \;=\; \frac{\partial f_\theta}{\partial \theta}\,\frac{\partial \theta}{\partial t} \;=\; \frac{\partial f_\theta}{\partial \theta}\,\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\frac{\partial f_\theta}{\partial \theta}^T(\boldsymbol{x})\;\boldsymbol{v}_{\text{goal}}(\boldsymbol{x})\right] \tag{3.9}$$

which significantly differs from the original **functional** gradient descent. We will aim to augment the neural network architecture so that parametric gradient descents can get closer to the functional one.

Let $\mathcal{T}_{\mathcal{A}}^{f_\theta}$, or just $\mathcal{T}_{\mathcal{A}}$, the tangent space of $\mathcal{F}_{\mathcal{A}}$ at $f_\theta$, that is, the set of all possible infinitesimal variations around $f_\theta$ under small parameter variations:

$$\mathcal{T}_{\mathcal{A}}^{f_\theta} := \left\{ \frac{\partial f_\theta}{\partial \theta}\,\delta\theta \;\middle|\; \text{s.t. } \delta\theta \in \Theta_{\mathcal{A}} \right\}.$$

This linear[2] functional space is a first-order approximation of the neighborhood of $f_\theta$ within $\mathcal{F}_{\mathcal{A}}$. The direction $\boldsymbol{v}_{\text{GD}}$ obtained above by gradient descent is actually not the best one to consider within $\mathcal{T}_{\mathcal{A}}$. Indeed, the best move $\boldsymbol{v}^*$ would be the orthogonal projection of the desired direction $\boldsymbol{v}_{\text{goal}} := -\nabla_{f_\theta}\mathcal{L}(f_\theta)$ onto $\mathcal{T}_{\mathcal{A}}$. This projection is what a (generalization of the notion of) natural gradient would compute [Ollivier, 2017]. Indeed, the parameter variation $\delta\theta^*$ associated to the functional variation $\boldsymbol{v}^* = \frac{\partial f_\theta}{\partial \theta}\,\delta\theta^*$ is the gradient $-\nabla_\theta^{\mathcal{T}_{\mathcal{A}}}\mathcal{L}(f_\theta)$ of $\mathcal{L}\circ f_\theta$ w.r.t. parameters $\theta$ when considering the $L_2$ metric on *functional* variations $\|\frac{\partial f_\theta}{\partial \theta}\,\delta\theta\|_{L_2(\mathcal{T}_{\mathcal{A}})}$, not to be confused with the usual gradient $\nabla_\theta\mathcal{L}(f_\theta)$, based on the $L_2$ metric on *parameter* variations $\|\delta\theta\|_{L_2(\mathbb{R}^{|\Theta_{\mathcal{A}}|})}$. This can be seen in a proximal formulation as:

$$\boldsymbol{v}^* \;=\; \underset{\boldsymbol{v}\in\mathcal{T}_{\mathcal{A}}}{\arg\min}\; \|\boldsymbol{v} - \boldsymbol{v}_{\text{goal}}\|^2 \;=\; \underset{\boldsymbol{v}\in\mathcal{T}_{\mathcal{A}}}{\arg\min}\left\{ D_f\mathcal{L}(f)(\boldsymbol{v}) + \frac{1}{2}\|\boldsymbol{v}\|^2 \right\} \tag{3.10}$$

where $D$ is the directional derivative (see details in Appendix B.3), or equivalently as:

$$\delta\theta^* \;\triangleq\; -\nabla_\theta^{\mathcal{T}_{\mathcal{A}}}\mathcal{L}(f_\theta) \;= \underset{\delta\theta\,\in\,\Theta_{\mathcal{A}}}{\arg\min} \left\| \frac{\partial f_\theta}{\partial \theta}\,\delta\theta - \boldsymbol{v}_{\text{goal}} \right\|^2 \tag{3.11}$$

$$= \underset{\delta\theta\,\in\,\Theta_{\mathcal{A}}}{\arg\min} \left\{ D_\theta\mathcal{L}(f_\theta)(\delta\theta) + \frac{1}{2}\left\| \frac{\partial f_\theta}{\partial \theta}\,\delta\theta \right\|^2 \right\}. \tag{3.12}$$

When $\boldsymbol{v}_{\text{goal}}$ does not belong to the reachable subspace $\mathcal{T}_{\mathcal{A}}$, there is a lack of expressivity, that is, the parametric space $\mathcal{A}$ is not rich enough to follow the ideal functional gradient descent. This happens frequently with small neural networks. The expressivity bottleneck is then quantified as the distance $\|\boldsymbol{v}^* - \boldsymbol{v}_{\text{goal}}\|$ between the functional gradient $\boldsymbol{v}_{\text{goal}}$ and the optimal functional move $\boldsymbol{v}^*$ given the architecture $\mathcal{A}$ (in the sense of Equation (3.10)).

[2] $\frac{\partial f_\theta}{\partial \theta}\,\delta\theta : \mathbb{R}^p \to \mathbb{R}^d,\;\; \boldsymbol{x} \mapsto \frac{\partial f_\theta(\boldsymbol{x})}{\partial \theta}\,\delta\theta$

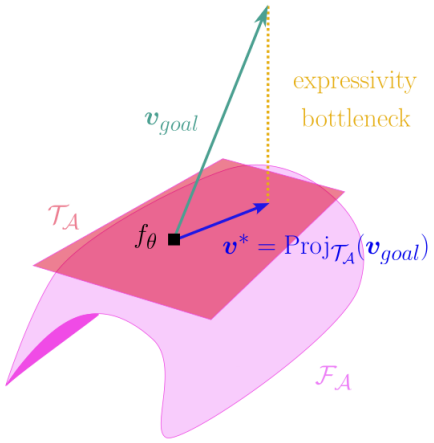



Figure 3.2: Expressivity bottleneck

**Example.** Suppose one tries to estimate the function $y = f_{\text{true}}(x) = 2\sin(x) + x$ with a linear model $f_{\text{predict}}(x) = ax + b$. Consider $(a, b) = (1, 0)$ and the square loss $\mathcal{L}$ . For the dataset of inputs $(x_0, x_1, x_2, x_3) = (0, \frac{\pi}{2}, \pi, \frac{3\pi}{2})$, there exists no parameter update $(\delta a, \delta b)$ that would improve prediction at $x_0, x_1, x_2$ and $x_3$ simultaneously, as the space of linear functions $\{f : x \to ax + b \mid a, b \in \mathbb{R}\}$ is not expressive enough. To improve the prediction at $x_0, x_1, x_2$ **and** $x_3$, one should look for another, more expressive functional space such that for $i = 0, 1, 2, 3$ the functional update $\Delta f(x_i) := f^{t+1}(x_i) - f^t(x_i)$ goes into the same direction as the functional gradient $\boldsymbol{v}_{\text{goal}}(x_i) := -\nabla_{f(x_i)}\mathcal{L}(f(x_i), y_i) = -2(f(x_i) - y_i)$ where $y_i = f_{\text{true}}(x_i)$.

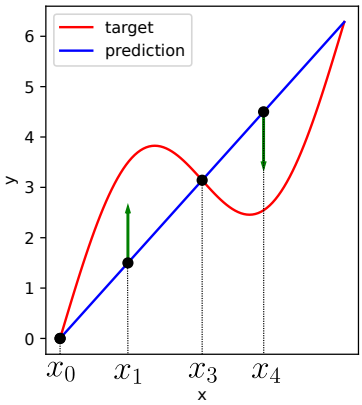



Figure 3.3: Linear interpolation

### Generalizing to all layers

The same reasoning can be applied to the pre-activations $\boldsymbol{a}_l$ at each layer $l$, seen as functions $\boldsymbol{a}_l : \boldsymbol{x} \in \mathbb{R}^p \mapsto \boldsymbol{a}_l(\boldsymbol{x}) \in \mathbb{R}^{d_l}$ defined over the input space of the neural network. The optimal parameter update for a given layer $l$ then follows the projection of the desired update $-\nabla_{\boldsymbol{a}_l}\mathcal{L}(f_\theta)$ of the pre-activation functions $\boldsymbol{a}_l$ onto the linear subspace $\mathcal{T}_{\mathcal{A}}^{\boldsymbol{a}_l}$ of pre-activation variations that are possible with the architecture, as we will detail now.

Given a sample $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{D}$, standard backpropagation already iteratively computes :

$$\boldsymbol{v}^l_{\text{goal}}(\boldsymbol{x}) := -\left(\nabla_{\boldsymbol{a}_l}\mathcal{L}(f_\theta)\right)(\boldsymbol{x}) \tag{3.13}$$

$$= -\nabla_{\boldsymbol{u}}\ell\left(\sigma_L(\boldsymbol{W}_L\,\sigma_{L-1}(\boldsymbol{W}_{L-1}\ldots\sigma_l(\boldsymbol{u}))),\,\boldsymbol{y}\right)|_{\boldsymbol{u}=\boldsymbol{a}_l(\boldsymbol{x})} \tag{3.14}$$

which is the derivative of the loss $\ell(f_\theta(\boldsymbol{x}), \boldsymbol{y})$ with respect to the pre-activations $\boldsymbol{u} = \boldsymbol{a}_l(\boldsymbol{x})$ of each layer. This is usually performed in order to compute the gradients w.r.t. model parameters $\boldsymbol{W}_l$, as $\nabla_{\boldsymbol{W}_l}\ell(f_\theta(\boldsymbol{x}), \boldsymbol{y}) = \frac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial W_l}\,\nabla_{\boldsymbol{a}_l}\ell(f_\theta(\boldsymbol{x}), \boldsymbol{y})$. $\boldsymbol{v}^l_{\text{goal}}(\boldsymbol{x}) := -\left(\nabla_{\boldsymbol{a}_l}\mathcal{L}(f_\theta)\right)(\boldsymbol{x})$ indicates the direction in which one would like to change the layer pre-activations $\boldsymbol{a}_l(\boldsymbol{x})$ in order to decrease the loss at point $\boldsymbol{x}$. However, given a minibatch of points $(\boldsymbol{x}_i)$, most of the time, no parameter move $\delta\theta$ is able to induce this progression for each $\boldsymbol{x}_i$ simultaneously because the $\theta$-parameterized family of functions $\boldsymbol{a}_l$ is not expressive enough. Given a subset of parameters $\tilde{\theta}$ (such as the ones specific to a layer: $\tilde{\theta} = \boldsymbol{W}_l$), and an incremental direction $\delta\tilde{\theta}$ to update these parameters (e.g. the one resulting from a gradient descent: $\delta\tilde{\theta} = -\sum_{(\boldsymbol{x},\boldsymbol{y})\in\text{minibatch}}\nabla_{\tilde{\theta}}\ell(f_\theta(\boldsymbol{x}), \boldsymbol{y})$, the impact of the parameter update $\gamma\delta\tilde{\theta}$ on the pre-activations $\boldsymbol{a}_l$ at layer $l$ at order 1 in $\gamma$, where $\gamma$ an amplitude factor, is as $\boldsymbol{v}^l(\boldsymbol{x}, \gamma\delta\tilde{\theta}) := \gamma\frac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial\tilde{\theta}}\,\delta\tilde{\theta}$. The factor $\gamma$ is the analog of the learning rate and is considered to be small.

## 3.2 Expressivity bottlenecks

In this section, we mathematically defined the expressivity bottleneck metric; how it can be applied to update the parameters of a network, and also, how it can be used to expand its architecture without redundancy.

### 3.2.1 The expressivity bottleneck

We quantify expressivity bottlenecks at any layer $l$ as the distance between the desired activity update $\boldsymbol{v}^l_{\text{goal}}(.)$ and the best realizable one $\boldsymbol{v}^l(.)$ (cf Figure Figure 3.2):

Definition 3.2.1 (Lack of expressivity). For a neural network $f_\theta$ and a minibatch of points $\boldsymbol{X} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, the lack of expressivity at layer $l$ is defined as how far the

desired activity update $\boldsymbol{V}^l_{\text{goal}} = (\boldsymbol{v}^l_{\text{goal}}(\boldsymbol{x}_1), \boldsymbol{v}^l_{\text{goal}}(\boldsymbol{x}_2), ...)$ is from the closest possible activity update $\boldsymbol{V}^l = (\boldsymbol{v}^l(\boldsymbol{x}_1), \boldsymbol{v}^l(\boldsymbol{x}_2), ...)$ realizable by a parameter change $\delta\theta$:

$$\Psi^l := \min_{\boldsymbol{v}^l \in \mathcal{T}_{\mathcal{A}}^{a_l}} \frac{1}{n} \sum_{i=1}^{n} \left\| \boldsymbol{v}^l(\boldsymbol{x}_i) - \boldsymbol{v}^l_{\text{goal}}(\boldsymbol{x}_i) \right\|^2 \tag{3.15}$$

$$= \min_{\delta\theta} \frac{1}{n} \left\| \boldsymbol{V}^l(\boldsymbol{X}, \delta\theta) - \boldsymbol{V}^l_{\text{goal}}(\boldsymbol{X}) \right\|^2_{\text{Tr}} \tag{3.16}$$

where $||.||$ stands for the usual Euclidean norm, $||.||_{\text{Tr}}$ for the Frobenius norm, and $\boldsymbol{V}^l(\boldsymbol{X}, \delta\theta)$ is the activity update resulting from parameter change $\delta\theta$ as defined in the previous section. In the two following parts, the minibatch $\boldsymbol{X}$ is fixed and the notations are simplified accordingly by removing the dependency on $\boldsymbol{X}$.

### 3.2.2 Best move without modifying the architecture of the network

Let $\delta\boldsymbol{W}^*_l$ be the solution of Equation (3.16) when the parameter variation $\delta\theta$ is restricted to involve only layer $l$ parameters, i.e. $\boldsymbol{W}_l$. This move is sub-optimal in that it does not result from an update of all architecture parameters but only of the current layer ones:

$$\delta\boldsymbol{W}^*_l = \arg\min_{\delta\boldsymbol{W}} \frac{1}{n} \left\| \boldsymbol{V}^l(\delta\boldsymbol{W}) - \boldsymbol{V}^l_{\text{goal}} \right\|^2_{\text{Tr}} \tag{3.17}$$

**Proposition 3.2.2.** *The solution of Problem (Equation (3.17)) is:*

$$\delta\boldsymbol{W}^*_l = \frac{1}{n} \boldsymbol{V}^l_{goal} \boldsymbol{B}^T_{l-1} (\frac{1}{n} \boldsymbol{B}_{l-1} \boldsymbol{B}^T_{l-1})^+ \tag{3.18}$$

*where $P^+$ denotes the generalized inverse of matrix $P$.*
*Proof in Appendix C.1*

This update $\delta\boldsymbol{W}^*_l$ is not equivalent to the usual gradient descent update, whose form is $\delta\boldsymbol{W}^{\text{GD}}_l \propto \boldsymbol{V}^l_{\text{goal}} \boldsymbol{B}^T_{l-1}$. In fact, the associated activity variation, $\delta\boldsymbol{W}^*_l \boldsymbol{B}_{l-1}$, is the projection of $\boldsymbol{V}^l_{\text{goal}}$ on the post-activation matrix of layer $l-1$, that is to say onto the span of all possible post-activation directions, through the projector $\frac{1}{n}\boldsymbol{B}^T_{l-1}(\frac{1}{n}\boldsymbol{B}_{l-1}\boldsymbol{B}^T_{l-1})^+\boldsymbol{B}_{l-1}$. To increase expressivity if needed, we will aim at increasing this span with the most useful directions to close the gap between this best update and the desired one. Note that the update $\delta\boldsymbol{W}^*_l$ consists of a standard gradient ($\boldsymbol{V}^l_{\text{goal}} \boldsymbol{B}^T_{l-1}$) and of a (kind of) natural gradient only for the last part (projector), as we consider metrics in the pre-activation space.

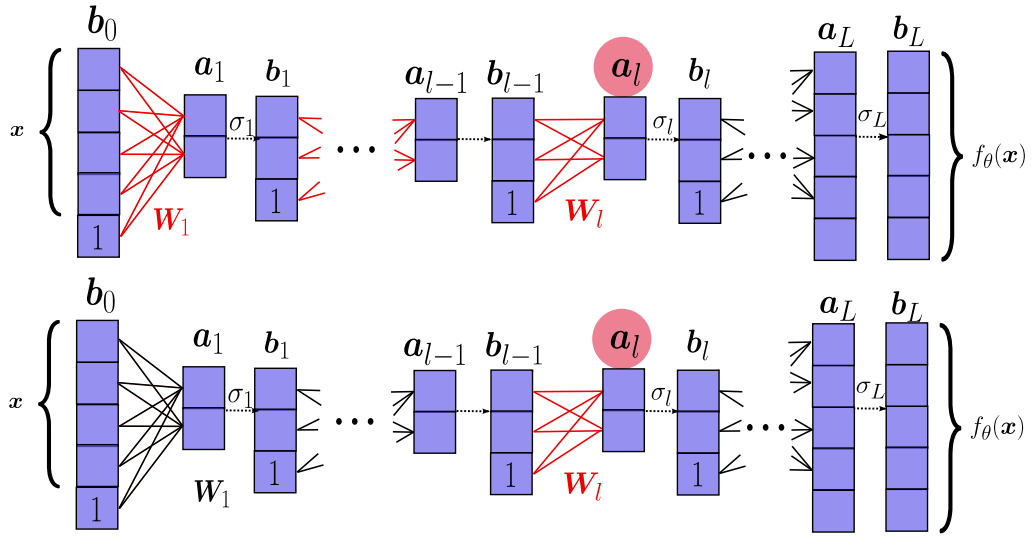



Figure 3.4: Feed-forward networks, in red the connection where modification $\delta\theta$ might apply when solving the expressivity bottleneck at layer $l$. The top figure is for the original formulation Equation (3.16) while the bottom figure is for the suboptimal formulation Equation (3.17).

### 3.2.3 Reducing expressivity bottleneck by modifying the architecture

To get as close as possible to $\boldsymbol{V}^l_{\text{goal}}$ and to increase the expressive power of the current neural network, we modify each layer of its structure. At layer $l-1$, we add $K$ neurons $n_1, ..., n_K$ with input weights $\boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_k$ and output weights $\boldsymbol{\omega}_1, ..., \boldsymbol{\omega}_K$ (cf Figures 3.5 and 3.7). The following expansions by concatenation is performed : $\boldsymbol{W}^T_{l-1} \leftarrow \begin{pmatrix} \boldsymbol{W}^T_{l-1} & \boldsymbol{\alpha}_1 & ... & \boldsymbol{\alpha}_K \end{pmatrix}$ and $\boldsymbol{W}_l \leftarrow \begin{pmatrix} \boldsymbol{W}_l & \boldsymbol{\omega}_1 & ... & \boldsymbol{\omega}_K \end{pmatrix}$.

This architecture modification is noted $\theta \leftarrow \theta \oplus \theta^K_{\leftrightarrow}$ where $\oplus$ is the concatenation sign and $\theta^K_{\leftrightarrow} := (\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)_{k=1}^K$ are the $K$ added neurons.

The added neurons could be chosen randomly, as in the usual neural network initialization, but this would not yield any guarantee regarding the impact on the system loss. Another possibility would be to set either input weights $(\boldsymbol{\alpha}_k)_{k=1}^K$ or output weights $(\boldsymbol{\omega}_k)_{k=1}^K$ to 0, so that the function $f_\theta(.)$ would not be modified, while its gradient w.r.t. $\theta$ would be enriched from the new parameters. Another option is to solve an optimization problem as in the previous section with the modified structure $\theta \leftarrow \theta \oplus \theta^K_{\leftrightarrow}$ and jointly search for both the optimal new parameters $\theta^K_{\leftrightarrow}$ and the optimal variation $\delta\boldsymbol{W}$ of the old ones:

$$\underset{\theta^K_{\leftrightarrow},\, \delta\boldsymbol{W}}{\arg\min} \frac{1}{n} \left\| \boldsymbol{V}^l(\delta\boldsymbol{W} \oplus \theta^K_{\leftrightarrow}) - \boldsymbol{V}^l_{\text{goal}} \right\|^2_{\text{Tr}} \tag{3.19}$$

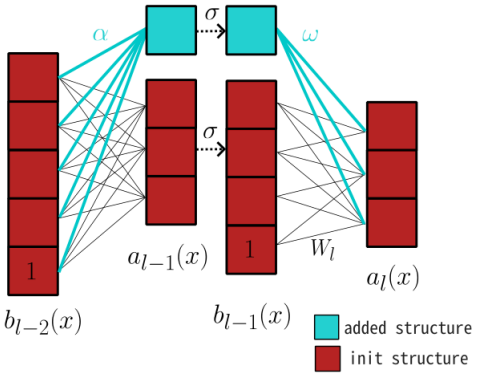



Figure 3.5: Adding one neuron to layer $l$ in cyan ($K = 1$), with connections in cyan. Here, $\boldsymbol{\alpha} \in \mathrm{R}^5$ and $\boldsymbol{\omega} \in \mathrm{R}^3$.

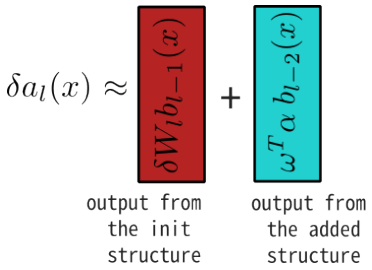



Figure 3.6: Sum of functional moves

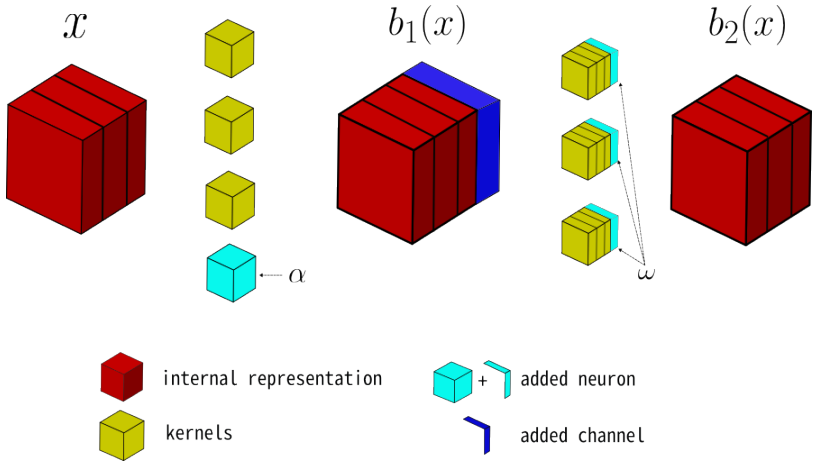



Figure 3.7: Adding one convolutional neuron at layer one for an input with three channels.

As shown in figure Figure 3.6, the displacement $\boldsymbol{V}^l$ at layer $l$ is actually a sum of the moves induced by the neurons already present ($\delta \boldsymbol{W}$) and by the added neurons ($\theta^K_{\leftrightarrow}$), Equation (3.19) rewrites as :

$$\underset{\theta^K_{\leftrightarrow},\, \delta \boldsymbol{W}}{\arg\min} \frac{1}{n} \left\| \boldsymbol{V}^l(\theta^K_{\leftrightarrow}) + \boldsymbol{V}^l(\delta \boldsymbol{W}) - {\boldsymbol{V}_{\text{goal}}}^l \right\|^2_{\text{Tr}} \tag{3.20}$$

with $\boldsymbol{v}^l(\boldsymbol{x}, \theta^K_{\leftrightarrow}) := \sum_{k=1}^K \boldsymbol{\omega}_k \; (b_{l-2}(\boldsymbol{x})^T \boldsymbol{\alpha}_k)$ (See Appendix B.4). Choosing $\delta \boldsymbol{W}$

as the best move of already-existing parameters as defined in Theorem 3.2.2 and noting $\boldsymbol{V}^l_{\text{goal}_{proj}} := \boldsymbol{V}^l_{\text{goal}} - \boldsymbol{V}^l(\delta \boldsymbol{W}^*)$, we search for the solution $\left(K^*, \theta^{K*}_{\leftrightarrow}\right)$ of the optimization problem :

$$\underset{K,\, \theta^K_{\leftrightarrow}}{\arg\min} \frac{1}{n} \left\| \boldsymbol{V}^l(\theta^K_{\leftrightarrow}) - \boldsymbol{V}^l_{\text{goal}_{proj}} \right\|^2_{\text{Tr}} . \tag{3.21}$$

One should note that Equation (3.20) and Equation (3.21) are generally not equivalent, though similar (cf. Appendix B.5).
This quadratic optimization problem can be solved thanks to the low-rank matrix approximation theorem [Eckart and Young, 1936], using matrices $\boldsymbol{N} := \frac{1}{n}\boldsymbol{B}_{l-2}\left(\boldsymbol{V}^l_{\text{goal}_{proj}}\right)^T$ and $\boldsymbol{S} := \frac{1}{n}\boldsymbol{B}_{l-2}\boldsymbol{B}^T_{l-2}$. As $\boldsymbol{S}$ is positive semi-definite, let its SVD be $\boldsymbol{S} = \boldsymbol{O}\boldsymbol{\Sigma}\boldsymbol{O}^T$, and define $\boldsymbol{S}^{-\frac{1}{2}} := \boldsymbol{O}\sqrt{\boldsymbol{\Sigma}}^{-1}\boldsymbol{O}^T$, with the convention that the inverse of 0 eigenvalues is 0. Finally, consider the SVD of matrix $\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N} = \sum_{k=1}^{R} \lambda_k \boldsymbol{u}_k \boldsymbol{v}_k^T$, where $R$ is the rank of the matrix $\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N}$. Then:

**Proposition 3.2.3.** *The solution of Problem (Equation (3.21)) is:*

- *optimal number of neurons:* $K^* = R$
- *their optimal weights:* $\theta^{K^*}_{\leftrightarrow} = (\boldsymbol{\alpha}^*_k, \boldsymbol{\omega}^*_k)_{k=1}^{K^*} = \left(\sqrt{\lambda_k}\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{u}_k,\ \sqrt{\lambda_k}\boldsymbol{v}_k\right)_{k=1}^{K^*}$

*Moreover for any number of neurons $K \leqslant R$, and associated optimal weights $\theta^{K,*}_{\leftrightarrow}$ consisting of the first $K$ neurons of $\theta^{K^*}_{\leftrightarrow}$, the expressivity gain can be quantified very simply as a function of the singular values $\lambda_k$:*

$$\Psi^l_{\theta \oplus \theta^{K,*}_{\leftrightarrow}} = \Psi^l_\theta - \sum_{k=1}^{K} \lambda_k^2 \tag{3.22}$$

*where $\Psi^l_\theta$ is the expressivity bottleneck (defined in Equation (3.16)). For convolutional layers instead of fully connected ones, Equation (3.22) becomes an inequality ($\leqslant$).*
*Proof in Appendix C.2.*

In practice before adding new neurons $(\alpha, \omega)$, we multiply them by an amplitude factor $\gamma$ found by a simple line search, i.e. we add $(\sqrt{\gamma}\alpha, \sqrt{\gamma}\omega)$ (we will discuss this amplitude factor in more detail in Section 4.2.1). The addition of each neuron $k$ has an impact on the bottleneck of the order of $\gamma\lambda_k^2$ provided $\gamma$ is small. We observe the same phenomenon with the loss as stated in the next proposition :

**Proposition 3.2.4.** *For $\gamma > 0$, solving (Equation (3.21)) using $\boldsymbol{V}_{goal_{proj}} = \boldsymbol{V}_{goal} - \boldsymbol{V}(\gamma\delta\boldsymbol{W}^*)$ is equivalent to minimizing the loss $\mathcal{L}$ at order one in $\gamma\boldsymbol{V}^l$. Furthermore, performing an architecture update with $\gamma\delta\boldsymbol{W}^*$ (Equation (3.17)) and a neuron*

*addition with* $\gamma\theta^{K,*}_{\leftrightarrow}$ *(Theorem 3.2.3) has an impact on the loss at first order in* $\gamma$ *as :*

$$\begin{aligned}\mathcal{L}(f_{\theta\oplus\gamma\theta^{K,*}_{\leftrightarrow}}) &:= \frac{1}{n}\sum_{i=1}^{n}\ell(f_{\theta\oplus\gamma\theta^{K,*}_{\leftrightarrow}}(\boldsymbol{x}_i),\boldsymbol{y}_i)\\ &= \mathcal{L}(f_\theta)-\gamma\left(\sigma'_{l-1}(0)\Delta_{\theta^{K,*}_{\leftrightarrow}}+\Delta_{\delta\boldsymbol{W}^*}\right)+o(\gamma)\end{aligned} \tag{3.23}$$

*with*

$$\Delta_{\theta^{K,*}_{\leftrightarrow}} := \frac{1}{n}\left\langle \boldsymbol{V}^l_{goal_{proj}},\ \boldsymbol{V}^l(\theta^{K,*}_{\leftrightarrow})\right\rangle_{\mathrm{Tr}} = \sum_{k=1}^{K}\lambda_k^2 \tag{3.24}$$

$$\Delta_{\delta\boldsymbol{W}^*} := \frac{1}{n}\left\langle \boldsymbol{V}^l_{goal},\ \boldsymbol{V}^l(\delta\boldsymbol{W}^*)\right\rangle_{\mathrm{Tr}} \geqslant 0\,. \tag{3.25}$$

*Proof in Appendix C.3*

The last two propositions are linked. While the complete proof for linear and convolutional layers can be found in the annexes, we now provide some intuitions to go from Equation (3.21) to Theorems 3.2.3 and 3.2.4 for linear feedforward network.

*Sketch of proof for Theorem 3.2.3:*
By linearizing the activation function $\sigma_{l-1}$ (cf Appendix B.4), we obtain $\boldsymbol{V} = \boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}$. The expressivity bottleneck squared (Equation (3.21) squared) becomes :

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \left\|\boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}-\boldsymbol{V}_{\mathrm{goal}_{proj}}\right\|^2 \tag{3.26}$$

By developing the norm using the scalar product, we remark that :

$$\left\|\boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}\right\|^2 = \left\|\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}\boldsymbol{\Omega}^T\right\|^2 \quad \text{and} \quad \left\langle \boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}, \boldsymbol{V}_{\mathrm{goal}_{proj}}\right\rangle = \left\langle \mathbf{S}^{\frac{1}{2}}\boldsymbol{A}\boldsymbol{\Omega}^T, \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\rangle$$

Where $\boldsymbol{S} := \boldsymbol{B}\boldsymbol{B}^T$ is the covariance matrix of $\boldsymbol{B}$, $\boldsymbol{N} := \boldsymbol{B}\boldsymbol{V}^T_{\mathrm{goal}_{proj}}$ is the covariance matrix between $\boldsymbol{B}$ and $\boldsymbol{V}_{\mathrm{goal}_{proj}}$ and $\mathbf{S}^{\frac{1}{2}} := \boldsymbol{O}\sqrt{\Sigma}\boldsymbol{O}^T$ is defined using the SVD of $\boldsymbol{S} = \boldsymbol{O}\Sigma\boldsymbol{O}^T$. Thus, we infer that there is an equivalence between solving Equation (3.26) and the following minimization problem :

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \left\|\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}\boldsymbol{\Omega}^T-\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 \tag{3.27}$$

We solve this last equation on variable $\boldsymbol{Q} := \mathbf{S}^{\frac{1}{2}}\boldsymbol{A}\boldsymbol{\Omega}^T$. The solution $\boldsymbol{Q}^*$ is equal to the matrix $\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}$ and we choose a particular solution for the couple

$\left(\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}^*, \boldsymbol{\Omega}^*\right)$ that is given by the SVD of $\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} = \boldsymbol{U}\Lambda\boldsymbol{V}^T$, ie

$$\boldsymbol{Q}^* = \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} = \boldsymbol{U}\Lambda\boldsymbol{V} \tag{3.28}$$

$$(\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}^*, \boldsymbol{\Omega}^*) = (\boldsymbol{U}\sqrt{\Lambda}, \boldsymbol{V}\sqrt{\Lambda}) \tag{3.29}$$

.

*NB: There are an infinite number of solutions for the couple* $\left(\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}^*, \boldsymbol{\Omega}^*\right)$.

*Sketch of proof for Theorem 3.2.4:*
This proposition suggests that when adding the new neurons, there is a link between the decrease of loss $\mathcal{L}$ and the decrease of the expressivity bottleneck $\Psi$. This is a direct consequence of the first order development of the loss in $\boldsymbol{a}_l$ when performing $\boldsymbol{a}_l \leftarrow \boldsymbol{a}_l + \gamma\boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*)$ at $t \to t+1$ is such that :

$$\mathcal{L}^{t+1} = \mathcal{L}^t - \frac{\gamma}{n}\left\langle \boldsymbol{V}_{\text{goal}}, \boldsymbol{V}\right\rangle + o(\gamma) \tag{3.30}$$

The final result can be recovered by replacing each element by considering $(\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}^*, \boldsymbol{\Omega}^*) = (\boldsymbol{U}\sqrt{\Lambda}, \boldsymbol{V}\sqrt{\Lambda})$ (plus some tricks ...).

One should also note that the family $\{\boldsymbol{V}^{l+1}((\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k))\}_{k=1}^K$ of pre-activity variations induced by adding the neurons $\theta_{\leftrightarrow}^{K,*}$ is orthogonal for the trace scalar product. We could say that the added neurons are orthogonal to each other (and to the already-present ones) in that sense. Interestingly, the GradMax method Evci et al. [2022] also aims at minimizing the loss Equation (3.23), but without avoiding redundancy (more precision will be given in Section 4.1.1).

The $\lambda_k$ could be used in a selection criterion, realizing a trade-off with computational complexity. A selection based on the statistical significance of singular values can also be performed. The full algorithm to compute the new neurons and its complexity is detailed in Section 5.1 and the python module is presented in Appendix D.

We finish this section by some additional propositions and remarks.

**Proposition 3.2.5.** *If $\boldsymbol{S}$ is positive definite, then solving Equation (3.21) is equivalent to taking $\boldsymbol{\omega}_k = \boldsymbol{N}\boldsymbol{\alpha}_k$ and finding the $K$ first eigenvectors $\boldsymbol{\alpha}_k$ associated to the $K$ largest eigenvalues $\lambda$ of the generalized eigenvalue problem :*

$$\boldsymbol{N}\boldsymbol{N}^T\boldsymbol{\alpha}_k = \lambda\boldsymbol{S}\boldsymbol{\alpha}_k$$

**Corollary 1** *For all integers $m, m'$ such that $m + m' \leqslant R$, at order one in $\boldsymbol{V}$, adding $m + m'$ neurons simultaneously according to the previous method is equivalent to adding $m$ neurons then $m'$ neurons by applying successively the previous method twice.*

## 3.3 Convergence to zero training loss

One might wonder whether a greedy approach on layer growth might get stuck in a non-optimal state. In this case, *greedy* means that every added neuron has to decrease the loss. Since in this work, neurons are added layer per layer independently, lets study here the case of a single hidden layer network, to spot potential layer growth issues. For the sake of simplicity, lets consider the task of least square regression towards an explicit continuous target $f^*$, defined on a compact set. That is, the objective is to minimize the loss:

$$\inf_f \sum_{\boldsymbol{x}\in\mathcal{D}} \|f(\boldsymbol{x}) - f^*(\boldsymbol{x})\|^2 \tag{3.31}$$

where $f(\boldsymbol{x})$ is the output of the neural network and $\mathcal{D}$ is the training set. Proofs and supplementary propositions are deferred to Appendix C.6, in particular C.6.4 and C.6.7.

First, if one allows only adding neurons but no modification of already existing ones:

**Proposition 3.3.1.** *It is possible to decrease the loss exponentially fast with the number $t$ of added neurons, i.e. as $\gamma^t \mathcal{L}(f)$, towards 0 training loss, and this in a greedy way, that is, such that each added neuron decreases the loss. The factor $\gamma$ is $\gamma = 1 - \frac{1}{n^3 d'} \left(\frac{d_m}{d_M}\right)^2$, where $d_m$ and $d_M$ are quantities solely dependent on the dataset geometry, $d'$ is the output dimension of the network, and $n$ is the dataset size.*

In particular, there exists no situation where one would need to add many neurons simultaneously to decrease the loss: it is always feasible with a single neuron.

TINY might get stuck when no correlation between inputs $\boldsymbol{x}_i$ and desired output variations $f^*(\boldsymbol{x}_i) - f(\boldsymbol{x}_i)$ can be found anymore. To prevent this, one can choose an auxiliary method to add neurons in such cases, for instance, random neurons (with a line search over their amplitude, cf. Appendix C.6.3), or locally optimal neurons found by gradient descent, or solutions of higher-order expressivity bottleneck formulations using further developments of the activation function. Lets name *completed-TINY* the completion of TINY by any such auxiliary method.

Now, if we also update already existing weights when adding new neurons, a stronger result is obtained :

**Proposition 3.3.2.** *Under certain assumptions (full-batch optimization, updating already existing parameters, and, more technically: polynomial activation function of order $\geqslant n^2$), completed-TINY reaches 0 training error in at most $n$ neuron additions almost surely.*

Hence we see the importance of updating existing parameters on the convergence speed. This optimization protocol is actually the one that is followed in practice when training neural networks with TINY (except when compared with other methods using their protocol).

Note that TINY approach shares similarity with gradient boosting Friedman [2001] somehow, as the architecture is grown based on the gradient of the loss. Note also that finding the optimal neuron to add is actually NP-hard [Bach, 2017], but that new neuron optimality is not needed to converge to 0 training error.

### Conclusion

In this chapter, we have provided the theoretical principles of TINY, a method to grow the architecture of a neural net while training it; starting from a very thin architecture, TINY adds neurons where needed and yields a fully trained architecture at the end. This method relies on the functional gradient to find new directions that tackle the expressivity bottleneck, even for small networks, by expanding their space of parameters. This way, we combine in the same framework gradient descent and neural architecture search, that is, optimizing the network parameters and its architecture at the same time, and this, in a way that guarantees convergence to 0 training error, thus escaping expressivity bottlenecks indeed.

While transfer learning works well on ordinary tasks, for it to succeed, it needs to fine-tune and use large architectures at deployment in order to extract and manipulate common knowledge. TINY methodology has the advantage of being generic and could also produce smaller models as it adapts the architecture to a single task.

# 4 Comparison with other approaches

**Contents**

In this chapter, we compare TINY's neurons initialization (Theorem 3.2.3) with GradMax from paper Evci et al. [2022] and NORTH from paper Maile et al. [2022] by adapting the expressivity bottleneck formalism to their frameworks. This is feasible as those methods also study first-order loss variations and use the same pre-activation matrix, but with an important difference: GradMax optimally decreases the loss without caring about redundancy, while NORTH avoids redundancy but picks random directions instead of optimal ones.

While sharing some similarities with TINY, GradMax and NORTH methods also single out themselves by initializing one part of their new neurons to zero. Such initialization allows for increases of architecture without modifying the output function $f_\theta$; however, it naturally makes any performance improvement impossible. In fact, to observe an enhancement, one would need to update the zero part of the new neurons with the usual gradient descent. On the other hand, TINY does not initialize any part of its new neurons to zero, and it can improve its performance without using gradient descent. In fact, TINY can also accelerate its training (in computation step units) compared to the usual gradient descent by computing an amplitude factor $\gamma$ on the best update of Theorem 3.2.2 as $\delta \boldsymbol{W}^* \leftarrow \gamma \delta \boldsymbol{W}^*$, and on the new neurons of Theorem 3.2.3 as $(\boldsymbol{\alpha}, \boldsymbol{\omega}) \leftarrow (\sqrt{\gamma}\boldsymbol{\alpha}, \sqrt{\gamma}\boldsymbol{\omega})$. This amplitude factor can be computed with a simple line search and allows the architecture to be modified with non-infinitesimal updates ($\gamma \gg 0$). While accelerating the learning, such amplitude factor might also be a better proxy of the usefulness of the new neurons compared to the eigenvalues of Theorem 3.2.5.

In Section 4.1 we redefine GradMax and NORTH methods with the expressivity bottleneck formalism and compare the computed new neurons of each method with the ones defined in Theorem 3.2.3; then, in Section 4.2, we define the amplitude factor and provide a theoretical intuition as well as a toy experiment to justify its use in a search strategy.

## 4.1 Revisiting other NAS approaches with Expressivity Bottleneck

In that section, we compare GradMax [Evci et al., 2022] and NORTH [Maile et al., 2022] new neurons initialization with the initialization of Theorem 3.2.3. In particular, we show that the fan-out weights of Theorem 3.2.5 resemble the ones of GradMax while the fan-in weights of Theorem 3.2.5 resemble the NORTH ones. We refer to each method with a little symbol that portrays the method, of the shape "fan-in | fan-out weights", the position of the zero indicating which part of the new neurons is initialized to zero.

### 4.1.1 $\boxed{0 \mid \|\nabla \mathcal{L}\|^2}$ GradMax

In this subsection, we start by recalling the GradMax paper methodology and its optimization problem.

The theoretical approach of GradMax is to add neurons with zero fan-in weights and choose the fan-out weights that would decrease the loss as much as possible after one gradient step. Let $\mathbf{\Omega}$ be the fan-out weights of such neurons and perform the addition at layer $l-1$ at time $t$. After one gradient step, i.e. $t \to t+1$, and considering a learning rate equal to 1, the decrease of loss is :

$$\mathcal{L}^{t+1} \approx \mathcal{L}^t + \langle \nabla_\theta \mathcal{L}, \delta\theta \rangle + \langle \nabla_{\mathbf{\Omega}} \mathcal{L}, \delta\mathbf{\Omega} \rangle + \langle \nabla_{\mathbf{A}} \mathcal{L}, \delta\mathbf{A} \rangle$$

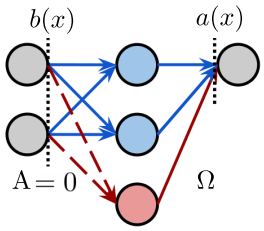



Figure 4.1: One neuron addition with GradMax method with the notations of Figure 2.7.

Taking the direction of the usual gradient descent, i.e. $\delta\theta = -\nabla_\theta \mathcal{L}$, $\delta\mathbf{A} = -\nabla_{\mathbf{A}} \mathcal{L}$ and $\delta\mathbf{\Omega} = -\nabla_{\mathbf{\Omega}} \mathcal{L} = 0$ because $\mathbf{A} = 0$, we have :

$$\mathcal{L}^{t+1} \approx \mathcal{L}^t - ||\nabla_\theta \mathcal{L}||^2 - ||\nabla_{\mathbf{A}} \mathcal{L}||^2 \tag{4.1}$$

To maximize the first-order decrease of the loss with such neuron addition, the output weights of the new neurons, as formulated in the original paper [Evci et al., 2022] at eq (11), are the solution of :

$$(\boldsymbol{\omega}_1^*, ..., \boldsymbol{\omega}_K^*) := \mathbf{\Omega}^* = \arg\max_{\mathbf{\Omega}} ||\nabla_{\mathbf{A}} \mathcal{L}||^2 \qquad s.t. \;\; ||\mathbf{\Omega}||^2 \leq c \tag{4.2}$$

for some fixed $c$.
Using Theorem C.5.10, we can re-write this derivative with the TINY notation :

$$\begin{aligned} \|\nabla_{\mathbf{A}} \mathcal{L}\|^2 &= \left\| \sum_i \boldsymbol{b}(\boldsymbol{x}_i) {\boldsymbol{v}_{\text{goal}}}^T(\boldsymbol{x}_i) \mathbf{\Omega} \right\|^2 \\ &= \left\| \boldsymbol{B} {\boldsymbol{V}_{\text{goal}}}^T \mathbf{\Omega} \right\|^2 \\ &= \left\| \tilde{\boldsymbol{N}} \mathbf{\Omega} \right\|^2 \qquad\qquad \tilde{\boldsymbol{N}} := \boldsymbol{B} {\boldsymbol{V}_{\text{goal}}}^T \end{aligned}$$

It follows that the fan-out weights of the neurons are the solution of :

$$\mathbf{\Omega}^* := \arg\max_{\mathbf{\Omega}} \left\| \tilde{\boldsymbol{N}} \mathbf{\Omega} \right\| \quad s.t. \;\; \|\mathbf{\Omega}\|^2 \leq c \tag{4.3}$$

To make TINY comparable to GradMax, we reformulate our minimization problem using the scalar product with the following proposition :

**Proposition 4.1.1.** $\forall \boldsymbol{D} \in \mathbb{R}^{p,q}, \boldsymbol{B} \in \mathbb{R}^{k,q}$,

$$\exists \tilde{c} \in \mathbb{R} \quad s.t, \quad \arg\min_{\boldsymbol{H}} \|\boldsymbol{D} - \boldsymbol{H}\boldsymbol{B}\|^2 = \arg\max_{\boldsymbol{H},\ \|\boldsymbol{H}\boldsymbol{B}\|^2 \leq \tilde{c}} \langle \boldsymbol{D}, \boldsymbol{H}\boldsymbol{B} \rangle$$

*The proof can be found in 4.*

Taking $\boldsymbol{V}_{\text{goal}}$ as $\boldsymbol{D}$ and $\boldsymbol{V} = \boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}$ as $\boldsymbol{H}\boldsymbol{B}$, we can reformulate TINY optimization problem 3.21 as :

$$\boldsymbol{A}^*, \boldsymbol{\Omega}^* = \arg\max_{\boldsymbol{A},\boldsymbol{\Omega}} \langle \boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}), \boldsymbol{V}_{\text{goal}_{proj}} \rangle \qquad s.t. \quad \|\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega})\|^2 \leq \tilde{c} \tag{4.4}$$

We now re-write the scalar product $\left\langle \boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}), \boldsymbol{V}_{\text{goal}_{proj}} \right\rangle$ and the norm of the constrain $\|\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega})\|$. We remark that

$$\begin{aligned}
\langle \boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}), \boldsymbol{V}_{\text{goal}_{proj}} \rangle &= \langle \boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}, \boldsymbol{V}_{\text{goal}_{proj}} \rangle \\
&= \text{Tr}(\boldsymbol{B}^T\boldsymbol{A}\boldsymbol{\Omega}^T\boldsymbol{V}_{\text{goal}_{proj}}) \\
&= \text{Tr}(\boldsymbol{A}\boldsymbol{\Omega}^T\boldsymbol{V}_{\text{goal}_{proj}}\boldsymbol{B}^T) \\
&= \langle \boldsymbol{\Omega}\boldsymbol{A}^T, \boldsymbol{V}_{\text{goal}_{proj}}\boldsymbol{B}^T \rangle \\
&= \left\langle \boldsymbol{A}\boldsymbol{\Omega}^T, \boldsymbol{N} \right\rangle \qquad\qquad \boldsymbol{N} := \boldsymbol{B}\boldsymbol{V}_{\text{goal}_{proj}}^T \qquad (4.5) \\
&= \langle \tilde{\boldsymbol{A}}\boldsymbol{\Omega}^T, \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \rangle \qquad \tilde{\boldsymbol{A}} := \boldsymbol{S}^{\frac{1}{2}}\boldsymbol{A},\ \boldsymbol{S} = \boldsymbol{B}\boldsymbol{B}^T \qquad (4.6)
\end{aligned}$$

By performing the same change of variable $\tilde{\boldsymbol{A}} := \boldsymbol{S}^{\frac{1}{2}}\boldsymbol{A}$ for the constraint on $\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega})$, it follows that :

$$\begin{aligned}
\|\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega})\|^2 &= \left\|\boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}\right\|^2 \\
&= \text{Tr}(\boldsymbol{A}\boldsymbol{\Omega}^T\boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{S}) \\
&= \left\|\boldsymbol{\Omega}(\mathbf{S}^{\frac{1}{2}}\boldsymbol{A})^T\right\| \\
&= \left\|\boldsymbol{\Omega}\tilde{\boldsymbol{A}}^T\right\|
\end{aligned}$$

TINY optimization problem is now equivalent to :

$$\tilde{\boldsymbol{A}}^*, \boldsymbol{\Omega}^* = \arg\max_{\tilde{\boldsymbol{A}},\boldsymbol{\Omega}} \langle \tilde{\boldsymbol{A}}\boldsymbol{\Omega}^T, \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \rangle \qquad s.t. \quad \left\|\boldsymbol{\Omega}\tilde{\boldsymbol{A}}^T\right\|^2 \leq c \tag{4.7}$$

To maximize the scalar product, lets choose $\tilde{\boldsymbol{A}}\boldsymbol{\Omega}^T = \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}$. A solution for $(\tilde{\boldsymbol{A}}, \boldsymbol{\Omega})$ is the (left, right) eigenvectors of the matrix $\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}$. It implies that :

$$\boldsymbol{\Omega}^* := \arg\max_{\boldsymbol{\Omega}} \left\|\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\boldsymbol{\Omega}\right\|^2 \quad s.t. \|\boldsymbol{\Omega}\| \leq \tilde{c} \tag{4.8}$$

This last equation could have been directly induced by Theorem 3.2.3, nonetheless the latter reasoning enables to determine what are the theoretical similarities and differences between GradMax and TINY optimization problems :

| GradMax | TINY |
|---|---|
| $\arg\max_{\boldsymbol{\Omega}} \left\|\nabla_{\boldsymbol{A}} \mathcal{L}(f_\theta)\right\|^2 \quad s.t. \quad \|\boldsymbol{\Omega}\| \leq c$ | $\arg\max_{\boldsymbol{A},\boldsymbol{\Omega}} \left\langle \boldsymbol{A}\boldsymbol{\Omega}^T, \boldsymbol{N} \right\rangle \quad s.t. \quad \left\|\boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B}\right\|^2 \leq c$ |
| $\Downarrow$ | $\Downarrow$ |
| $\arg\max_{\boldsymbol{\Omega}} \left\|\tilde{\boldsymbol{N}}\boldsymbol{\Omega}\right\|^2 \quad s.t. \quad \|\boldsymbol{\Omega}\|^2 \leq c$ | $\arg\max_{\boldsymbol{\Omega}} \left\|\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\boldsymbol{\Omega}\right\|^2 \quad s.t. \quad \|\boldsymbol{\Omega}\| \leq \tilde{c}$ |

Table 4.1: Optimization problems defining the fan-out weights of the new neurons for GradMax method (left) and TINY method (right).

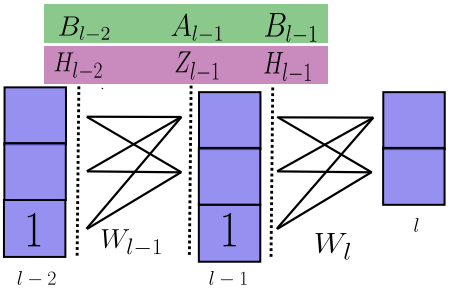



Figure 4.2: In green the TINY notations, in purple the notations of Maile et al. [2022].

First, the matrix $\tilde{\boldsymbol{N}}$ is not defined using the projection of the desired update $\boldsymbol{V}_{\text{goal}_{proj}}^{l+1}$. As a consequence, GradMax does not take into account redundancy and, on the opposite, will actually try to add new neurons that are as redundant as possible with the part of the goal update that is already feasible with already-existing neurons. Such redundancy is avoided when $\tilde{\boldsymbol{N}} = \boldsymbol{N}$, that is, when the best update is 0. We note that there is an equivalence between the best update is equal to 0 and the usual gradient is equal to 0 (cf Theorem C.5.11).

Second, the constraint lies in the weight space for the GradMax method, while it lies in the pre-activation space in TINY. The difference is that GradMax relies on the Euclidean metric in the space of parameters, which arguably offers less meaning that the Euclidean metric in the space of activities. Essentially, this is the same difference as between the standard L2 gradient w.r.t. parameters and the natural gradient, which takes care of parameter redundancy and measures all quantities in the output space in order to be independent of the parameterization. In practice, we do observe that the "natural" gradient direction improves the loss better than the usual L2 gradient.

Third, TINY fan-in weights are not set to 0 but directly to their optimal values (at first order).

### 4.1.2 $\boxed{\mathcal{X}_\perp \mid 0}$ NORTH/ RandomProj

In the paper Maile et al. [2022], when adding neurons at layer $l-1$, the fan-out weights are initialized to 0, while the fan-in weights are picked within the *kernel* (preimage of $\{0\}$) of an application describing already existing neurons. Two such applications are proposed, respectively the matrix of fan-in weights and the pre-activation matrix, yielding two different notions of orthogonality. The first orthogonality definition is on the fan-in weights, hence the new neurons are initialized to expand the span of the lines of $\boldsymbol{W}_{l-1}$. Formally, the fan-in weights of the new neurons are initialized as :

$$\boldsymbol{\alpha}^w = \mathcal{P}_{\mathrm{Ker}(\boldsymbol{W}_{l-1})}\, \boldsymbol{r} \quad \boldsymbol{r} \sim \mathcal{N}(0, I) \tag{4.9}$$

where the index $w$ stands for weights, and where the function

$$\mathcal{P}_{\mathrm{Ker}(.)} : \boldsymbol{W} \mapsto \left(I - \boldsymbol{W}^T (\boldsymbol{W}\boldsymbol{W}^T)^+ \boldsymbol{W}\right) \tag{4.10}$$

is the projection onto the kernel of the lines of the matrix $\boldsymbol{W}$. In the original paper, this function is approximated with a concatenation of vectors that are orthogonal to the columns of the matrix $\boldsymbol{W}_{l-1}$, noted $\boldsymbol{V}_{\mathrm{Ker}(\boldsymbol{W}_{l-1}^T)}$. The second orthogonality metric is on the functional space of pre-activity at layer $l-1$. To increase the space of such functional space, the fan-in weights of the new neurons are initialized as :

$$\boldsymbol{\alpha}^{act} = \boldsymbol{S}^+ \boldsymbol{B}_{l-2} \mathbf{V}_{\mathrm{Ker}(\boldsymbol{A}_{l-1})} \boldsymbol{r} \quad \text{with} \quad \boldsymbol{r} \sim \mathcal{N}(0, I) \tag{4.11}$$

where the index *act* stands for activation, and where $\mathbf{V}_{\mathrm{Ker}(\boldsymbol{A}_{l-1})} \in \mathbb{R}^{n, n-M_{l-1}}$ is the concatenation of $n - M_{l-1}$ vectors that are orthogonal to the lines of the matrix $\boldsymbol{A}_{l-1} \in \mathbb{R}^{M_{l-1}, n}$, where $M_{l-1}$ is the size of $\boldsymbol{a}_{l-1}$ and is also a proxy of the rank of $\boldsymbol{A}_{l-1}$ when $n$ is large. Here, $\boldsymbol{S}$ is, as before, the covariance matrix of $\boldsymbol{B}_{l-2}$, ie $\boldsymbol{S} = \frac{1}{n}\boldsymbol{B}_{l-2}\boldsymbol{B}_{l-2}^T$.

On the other hand, fan-in weights in TINY method are initialized as the left eigenvectors of the matrix :

$$\begin{aligned}
\boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{B}_{l-2} \boldsymbol{V}_{\mathrm{goal}_{proj}}^T &= \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{B}_{l-2} \boldsymbol{V}_{goal_{proj}}^T && (4.12)\\
&= \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{B}_{l-2} (\boldsymbol{V}_{goal} - \boldsymbol{V}_{goal} \boldsymbol{B}_{l-1}^T \left(\boldsymbol{B}_{l-1} \boldsymbol{B}_{l-1}^T\right)^+ \boldsymbol{B}_{l-1})^T && (4.13)\\
&= \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{B}_{l-2}\, \mathcal{P}_{\mathrm{Ker}(\boldsymbol{B}_{l-1})}\, \boldsymbol{V}_{goal}^T && (4.14)
\end{aligned}$$

Considering $\boldsymbol{v}_i$ the right eigenvectors of the matrix $\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}$, then, TINY initialization is, up to a multiplication factor, equal to :

$$\begin{aligned}
\boldsymbol{\alpha}_i^{\mathrm{TINY}} &= \mathbf{S}^{-\frac{1}{2}} \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{B}_{l-2}\, \mathcal{P}_{\mathrm{Ker}(\boldsymbol{B}_{l-1})}\, \boldsymbol{V}_{goal}^T \boldsymbol{v}_i && (4.15)\\
&= \boldsymbol{S}^+ \boldsymbol{B}_{l-2}\, \mathcal{P}_{\mathrm{Ker}(\boldsymbol{B}_{l-1})}\, \boldsymbol{V}_{goal}^T \boldsymbol{v}_i && (4.16)
\end{aligned}$$

A first difference is that NORTH bases its theory on the space of pre-activation at layer $l-1$ while TINY focuses on the pre-activation at layer $l$. Then the other main difference is that TINY uses the backpropagation to find the best $\boldsymbol{v}_i$ directly, while the NORTH approach tries random directions $\boldsymbol{r}$ to explore the space of possible neuron additions. For that reason, we rather call the NORTH method the RandomProjection or RandomProj method.

| NORTH | TINY |
|---|---|
| $\boldsymbol{S}^{+}\boldsymbol{B}_{l-2}\mathbf{V}_{\mathrm{Ker}(\boldsymbol{A}_{l-1})}\boldsymbol{r}$ | $\boldsymbol{S}^{+}\boldsymbol{B}_{l-2}\mathcal{P}_{\mathrm{Ker}(\boldsymbol{B}_{l-1})}\boldsymbol{V}_{goal}^{T}\boldsymbol{v}_i$ |

Table 4.2: Fan-in weights initialization for NORTH method (left) and TINY method (right).

#### 4.1.3 TINY $\approx \boxed{\mathcal{X}_{\perp} \mid 0} + \boxed{0 \mid \|\nabla\mathcal{L}\|^2}$ ?

The neurons of TINY method, as defined in Theorem 3.2.3, share similarity with the RandomProj's ones for the fan-in weights and with the GradMax's ones for the fan-out weights. However, by initializing no side of the layer to zero, TINY new neurons have an effect on the loss with no need for a gradient step, while GradMax and RandomProj need this procedure to observe a decrease in loss. In practice, before adding the new neurons, we multiplied them by an amplitude factor $\gamma$ found by a simple line search, i.e. we add $(\sqrt{\gamma}\alpha, \sqrt{\gamma}\omega)$. We discuss the amplitude factor in the next section.

## 4.2 Moving away from first-order approximation

While TINY methodology relies on first-order approximations ($\boldsymbol{v}_{\text{goal}} = -\nabla_a\mathcal{L}$, $\boldsymbol{v} := \frac{\partial a}{\partial\theta}\delta\theta$) to estimate the neurons to add, we leave this setting in this section by considering a non-infinitesimal amplitude factor on the new neurons. This would benefit the growth from the empirical and theoretical point of view as explained in this section. Note that such amplitude factor is useless for GradMax and NORTH initializations as one part of their neurons is set to zero.

### 4.2.1 Definition of the amplitude factor

Once the new neurons are computed using Theorem 3.2.3, they are added to the architecture with an amplitude factor $\gamma$, i.e. $(\boldsymbol{\alpha}_i, \boldsymbol{\omega}_i) \leftarrow (\sqrt{\gamma}\boldsymbol{\alpha}_i, \sqrt{\gamma}\boldsymbol{\omega}_i)$. When this factor is chosen arbitrarily small, the first order approximation of Theorem 3.2.5 holds, but the impact on the loss is hardly noticeable as being infinitesimal in $\gamma$. In practice, we would rather choose a relatively large $\gamma$ and leave the first approximation setting to observe a clear impact on the loss and to accelerate the training. To choose $\gamma$ correctly, we perform a line search by looking for the value of $\gamma$, which decreases the loss the most on a mini-batch $\boldsymbol{X}_\gamma$ which might be different from the mini-batch used to estimate the new neurons and the best update. By doing so, the neuron addition pipeline becomes a three-steps procedure; first compute the new neurons using Theorem 3.2.3 and choose a maximum number of neurons to add $n_d \leq K^*$, then normalize the fan-in weights $\{\boldsymbol{\alpha}_i\}_{i=1}^{n_d}$ and fan-out weights

$\{\boldsymbol{\omega}_i\}_{i=1}^{n_d}$of the new neuron as:

$$\boldsymbol{\alpha}_k^* \leftarrow \boldsymbol{\alpha}_k^* \times \frac{1}{\sqrt{||(\boldsymbol{\alpha}_j^*)_{j=1}^{n_d}||_2^2/n_d}} \qquad \boldsymbol{\omega}_k^* \leftarrow \boldsymbol{\omega}_k^* \times \frac{1}{\sqrt{||(\boldsymbol{\omega}_j^*)_{j=1}^{n_d}||_2^2/n_d}} \tag{4.17}$$

Finally, multiply them by the amplitude factor $\gamma^*$ :

$$\boldsymbol{\alpha}_k^*, \ \boldsymbol{\omega}_k^* \ \leftarrow \boldsymbol{\alpha}_k^*\gamma^*, \ \boldsymbol{\omega}_k^*\gamma^* \qquad \text{s.t.} \qquad \gamma^* := \underset{\gamma\in[0,u]}{\arg\min} \sum_{\boldsymbol{x}_i\in\boldsymbol{X}_\gamma} \mathcal{L}(f_{\theta\oplus\gamma\theta_{\leftrightarrow}^{n_d}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

where $u > 0$, $\gamma\theta_{\leftrightarrow}^{n_d} = (\sqrt{\gamma}\boldsymbol{\alpha}_k^*, \sqrt{\gamma}\boldsymbol{\omega}_k^*)_k^{n_d^*}$ and the operation $\theta \oplus \gamma\theta_{\leftrightarrow}^K$ is the concatenation of the neural network with the new amplified neurons. In practice the interval $[0, u]$ is approximated by the set of values $\{h^k \mid k \in \mathbb{Z},\ 1e^{-8} < h^k < u\}$ for some chosen $h$.

With the same logic, let us define the amplitude factor for an update of architecture $\delta\theta$ as the positive factor to multiply $\delta\theta$ with before adding it to the architecture. In this case, after normalizing $\delta\theta$ by its norm as $\delta\theta \leftarrow \delta\theta/\left\|\delta\theta\right\|$, the amplitude factor $\gamma^*$ is defined as :

$$\delta\theta \leftarrow \gamma^*\delta\theta \qquad \text{s.t.} \qquad \gamma^* := \underset{\gamma\in[0,u]}{\arg\min} \sum_{\boldsymbol{x}_i\in\boldsymbol{X}_\gamma} \mathcal{L}(f_{\theta+\gamma\delta\theta}(\boldsymbol{x}_i), \boldsymbol{y}_i) \tag{4.18}$$

In particular, the amplitude factor will be used for the best update, i.e. $\delta\theta = \delta\boldsymbol{W}^*$.

### 4.2.2 Theoretical justification of the amplitude factor : A fair comparison of the loss decrease $\sum_i \lambda_i^2$

Apart from accelerating the training, using a relatively large $\gamma^*$ has a theoretical advantage if one wants to choose at which layer neurons should be added first. At first sigh, the eigenvalues of Theorem 3.2.5, as quantifying the first-order impact of the new neurons on the loss, define a natural criterion to select such a layer, for example, we could choose the layer whose new neurons have the highest eigenvalues. However, this criterion is empirically less effective than the naive method, which adds neurons in the order of the layers (Section 4.2.3), and it can be theoretically explained by comparing the desired update between two successive layers. Indeed, the desired update at layer $l$ is actually a function of the desired

update at layer $l+1$ as:

$$
\begin{aligned}
{\boldsymbol{v}_{\text{goal}}}^{l}(\boldsymbol{x}) &= -\nabla_{\boldsymbol{a}^l(\boldsymbol{x})}\ell(\boldsymbol{x}) && (4.19)\\
&= -\frac{\partial \boldsymbol{a}^{l+1}(\boldsymbol{x}))}{\partial \boldsymbol{a}^l(\boldsymbol{x})}^T \nabla_{a^{l+1}(\boldsymbol{x})}\ell(\boldsymbol{x}) && (4.20)\\
&= -\left(\frac{\partial\left(\sigma_{l+1}(\boldsymbol{W}_l\boldsymbol{a}_l(\boldsymbol{x}))\right)}{\partial \boldsymbol{a}^l(\boldsymbol{x})}\right)^T \nabla_{a^{l+1}(\boldsymbol{x})}\ell(\boldsymbol{x}) && (4.21)\\
&= -\left(\frac{\partial\sigma(\boldsymbol{u})}{\partial \boldsymbol{u}}_{|\boldsymbol{u}=\boldsymbol{W}\boldsymbol{a}_l(\boldsymbol{x})}\boldsymbol{W}_l\right)^T \ell(\boldsymbol{x}) && (4.22)\\
&= \boldsymbol{K}(\boldsymbol{x}){\boldsymbol{v}_{\text{goal}}}^{l+1}(\boldsymbol{x}) && (4.23)
\end{aligned}
$$

where $\boldsymbol{K}(\boldsymbol{x}) := \left(\frac{\partial\sigma(\boldsymbol{u})}{\partial \boldsymbol{u}}_{|\boldsymbol{u}=\boldsymbol{W}\boldsymbol{a}_l(\boldsymbol{x})}\boldsymbol{W}_l\right)^T$. Considering independently the best neurons to add at layer $l$ and $l+1$ of Theorem 3.2.3, their associated updates, i.e. $\boldsymbol{v}^l$ and $\boldsymbol{v}^{l+1}$, and then using Equation (3.24), one can deduce that comparing the eigenvalues of Theorem 3.2.3 is equivalent to compare $\frac{1}{n}\left\langle \boldsymbol{V}^l, {\boldsymbol{V}_{\text{goal}}}^l_{proj}\right\rangle$ to $\frac{1}{n}\left\langle \boldsymbol{V}^{l+1}, {\boldsymbol{V}_{\text{goal}}}^{l+1}_{proj}\right\rangle$. We transform those two scalar products to make them comparable :

$$
\begin{aligned}
&\left\langle \boldsymbol{V}^l, {\boldsymbol{V}_{\text{goal}}}^l_{proj}\right\rangle && \text{and} && \left\langle \boldsymbol{V}^{l+1}, {\boldsymbol{V}_{\text{goal}}}^{l+1}_{proj}\right\rangle && (4.24)\\
&= \left\langle \boldsymbol{V}^l, {\boldsymbol{V}_{\text{goal}}}^l \mathcal{P}_{\text{Ker}(\boldsymbol{B}_{l-1})}\right\rangle && && = \left\langle \boldsymbol{V}^{l+1}, {\boldsymbol{V}_{\text{goal}}}^{l+1} \mathcal{P}_{\text{Ker}(\boldsymbol{B}_{l})}\right\rangle && (4.25)\\
&= \left\langle \boldsymbol{V}^l, \boldsymbol{K}^T{\boldsymbol{V}_{\text{goal}}}^{l+1} \mathcal{P}_{\text{Ker}(\boldsymbol{B}_{l-1})}\right\rangle && && = \left\langle \boldsymbol{V}^{l+1}\mathcal{P}^T_{\text{Ker}(\boldsymbol{B}_{l})}, {\boldsymbol{V}_{\text{goal}}}^{l+1}\right\rangle && (4.26)\\
&= \text{Tr}\left(\mathcal{P}_{\text{Ker}(\boldsymbol{B}_{l-1})}{\boldsymbol{V}^l}^T\boldsymbol{K}^T{\boldsymbol{V}_{\text{goal}}}^{l+1}\right) && && && (4.27)\\
&= \left\langle \boldsymbol{K}\boldsymbol{V}^l\mathcal{P}_{\text{Ker}(\boldsymbol{B}_{l-1})}, {\boldsymbol{V}_{\text{goal}}}^{l+1}\right\rangle && && && (4.28)
\end{aligned}
$$

Where Equation (4.26) is obtained with the property $\text{Tr}(\boldsymbol{AB}) = \text{Tr}(\boldsymbol{BA})$ and Equation (4.25) is obtained using Equation (4.13). From the last equation, on can see that the left scalar product is highly dependent on the matrix $\boldsymbol{K} = \frac{\partial\sigma(\boldsymbol{u})}{\partial \boldsymbol{u}}_{|\boldsymbol{u}=\boldsymbol{W}\boldsymbol{a}_l(\boldsymbol{x})}\boldsymbol{W}_l$, which is itself highly amplitude dependent on the weight matrices $\boldsymbol{W}_l$. It implies, that without any assumption on the weight matrices of the network, the amplitude of the previous scalar products might be artificially amplified, making the comparison between those two (and doing so between their associated eigenvalues) meaningless. A nicer way to choose where to add the new neurons would be to estimate the non-linear impact of those new neurons on the loss and it seems that performing a line search on $\gamma$ would estimate a nice proxy of this impact.

### 4.2.3 Empirical justification of the use of the amplitude factor

In this section, we show empirically that choosing where to perform neuron addition with the amplitude factor $\gamma$ is more relevant than performing such choice on the eigenvalues of Theorem 3.2.3. To do so, we increase the architecture of networks one neuron by one neuron, and compare the performances of the growing networks with different addition strategies; the *Naive* strategy adds neurons in the order of the layer index; the *MaxEigenvalue* strategy adds neurons where the first eigenvalue of Theorem 3.2.3 is the highest; the *AmplitudeFcator* strategy adds neurons where the decrease of loss $\Delta := \mathcal{L}_(f_\theta) - \mathcal{L}_(f_{\theta \oplus \gamma^* \theta^1_\leftrightarrow})$ is the highest.

The experiment is performed on a simulated and balanced dataset $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{500}$ with $\boldsymbol{x} \sim \mathcal{N}(0, I_{50})$ and $\boldsymbol{y} = \arg\max_{k=1,\dots,10} \left\{ \sin(k \sum_{j=1}^{50} \boldsymbol{x}[j] \right\}$, where $\boldsymbol{x}[j]$ is the $j-th$ coordinate of $\boldsymbol{x}$. To plot Figure 4.3, a two linear layer network is initialized with one neuron by hidden layer and its architecture is grown for a given strategy $s$. The growing process is the repetition of the following iteration: select a depth $d$ with the strategy $s$, increase the network at depth $d$ with one neuron, then update all the network layers with its best update. For all the strategies the new neurons and the best updates are added to the architecture with an amplitude factor and no training is performed between each modification of architecture. The full algorithm to grow the network architecture is described in Algorithm 3, in which the number of iterations $T$ has been set differently depending on the strategy. It is equal to 100 for the Naive and the Amplitude factor strategy, but for the eigenvalue strategy, which is slower to converge to full expressivity, this number of iteration has been set to 200.

The best updates, the new neurons, and the amplitude factors are performed on the overall dataset. The accuracy of Figure 4.3 is also evaluated on the overall dataset and after each update of architecture (neurons addition or best update), which defines the step unit of the top plots.

In the figure Figure 4.3, we observe that the network whose growing strategy is based on the amplitude factor, $\gamma^*$, and its decrease of loss $\Delta_{\gamma^*}$, have the most interesting curve on all the plots. On the top left plot, its accuracy is the highest at any step, and its Pareto front shows a better ratio between the performance and the number of parameters. On the other hand, we observe that a network whose architectures is increased with the eigenvalues criterion has the worst performances. Indeed, on the top left figure, its accuracy curve is the lowest at any step of the growing process; on the top right figure, its complexity is the highest at the beginning of the experiment; and, in the bottom plot, it has the worst Pareto front: accuracy versus the number of parameters (bottom plot). Nonetheless, we observe that its complexity velocity significantly decreases around 250 steps (top right plot) while its accuracy increases (top left plot). Those two tendencies impact its Pareto front, which almost catches up with the curves of the other strategies. However, it still needs extra-growing steps to achieve full expressivity, as shown in the top figures. We can explain this change of dynamic in the Pareto front for the eigenvalues strategy as so: first, let us remark that the dimension of the input (50) is greater than the dimension of the output (10), making a neuron

addition at the first layer more expansive in terms of parameters than a neuron addition at the second layer. Then, regarding the top right plot, we can deduce that networks grown with this strategy add more neurons at the first layer at the beginning of the experiment (the red curve being above the blue curve). Those neuron additions are less interesting in terms of performance and complexity, but they enhance the expressivity of the first layer in the sense that they increase considerably its dimension and potentially the available information usable for neurons at the second layer. In the middle of the experiment (300 steps on the top plots), it is likely that the information of the dataset passed the first layer and is now available to be transformed by neurons at the second layer as the method only adds neurons at the second layer. This change of layer to perform the neuron addition slows the evolution of the complexity, and the available information at layer one greatly enhances the performance at each neuron addition at layer two. However, this remark should be taken with a grain of salt as it might greatly depend on the setting of the considered experiment, such as the dimension of the input and the output, the number of layers, and the starting number of neurons per layer.

## Conclusion

TINY has a lot in common with the GradMax and the NORTH methods, and in some sense, it bridges the gap between the two. Furthermore, TINY can accelerate its training using an amplitude factor $\gamma$, which is also a more informative indicator to construct a search strategy compared to the eigenvalues of theorem 3.2.3. Indeed, those eigenvalues can be artificially amplified by the weights matrices, making any comparison of eigenvalues between layers meaningless. To construct a strategy of adding, instead of focusing only on the impact of the new neurons, one could also compare the contribution of the new neurons at layer $l$ ($\sum_k \lambda_i^2$) with the first-order impact of updating the current weight matrices at layer $l$ ($\|\nabla_{\boldsymbol{W}^l}\mathcal{L}\|^2$). As such comparison would be done at the same layer, it would not suffer from any amplifying phenomenon, and it would define a search strategy that makes a trade-off between the possible enhancement due to the current network and the potential enhancement due to the increase of architecture.

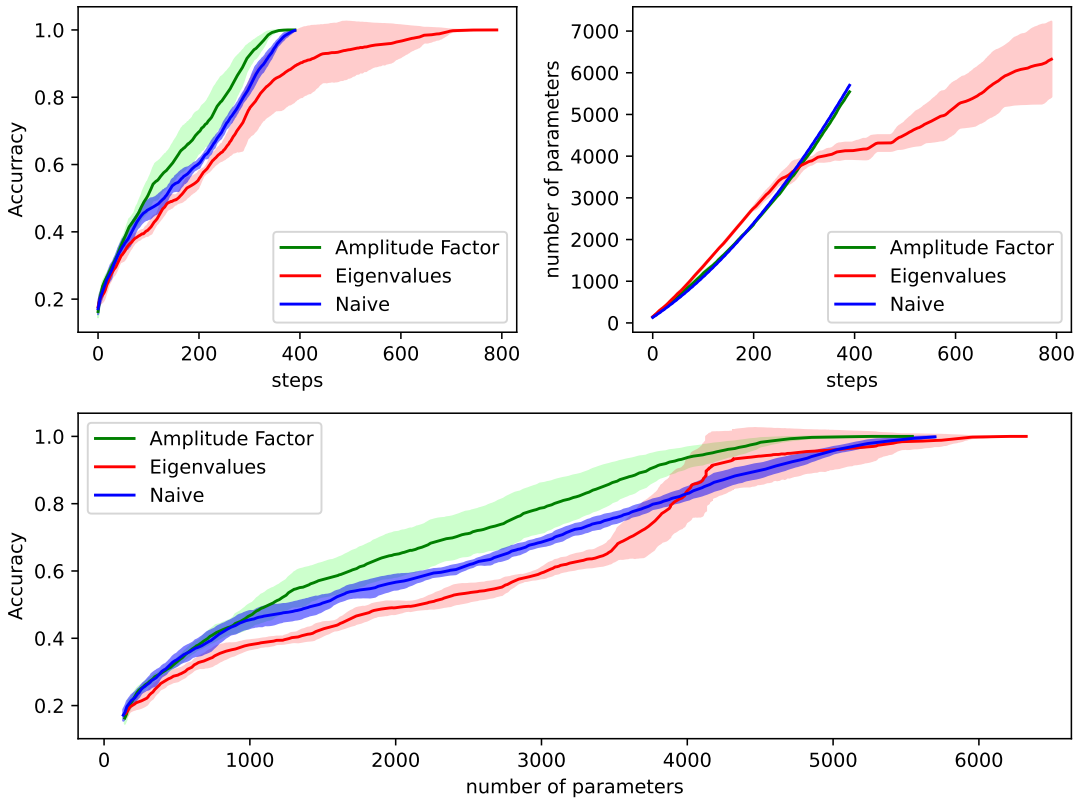



Figure 4.3: Accuracy and number of parameters of growing networks on a simulated dataset with different addition strategies.

**Algorithm 3:** Grow a network with the strategy $s$

```
Data: s : Strategy
Result: Neural Network N
Initialize a two linear layers network with one neuron per layer;
for t in [1, ..., T] do
    depth = s(t);
    Compute the first neuron α_1, ω_1 of Theorem 3.2.3 at depth;
    Compute the amplitude factor γ* for this new neuron;
    Multiply the neuron with its amplitude factor;
    Add it to the architecture;
    Evaluate the performances of the network;
    for d in [1, 2, 3] do
        Compute the best update at layer d and its amplitude factor;
        Multiply the best update with its amplitude factor;
        Update the architecture with this best update;
        Evaluate the performances of the network;
    end
end
```

**Algorithm 4:** Select depth to grow the network

```
Data: s : Strategy, t : iteration
Result: depth where to add a neuron
if s = Naive then
    depth = np.mod(t, 2) + 1;
else if s = MaxEigenValue then
    for d in [1, 2] do
        Compute the first eigenvalue λ_1^d of Theorem 3.2.3 at d;
    end
    depth = arg max_{d=1,2}{λ_1^d};
else if s = AmplitudeFactor then
    for d in [1, 2] do
        Compute the first neuron α_1, ω_1 of Theorem 3.2.3 at d;
        Compute the amplitude factor γ* for this new neuron;
        Evaluate the decrease of loss Δ_d if the architecture is expanded
          with (√γ* α_1, √γ* ω_1)
    end
    depth = arg max_{d=1,2}{Δ_d};
return depth;
```

# 5 TINYpub and experiments

**Contents**

## Introduction

In this chapter, we present the main functions of the python module `TINYpub`, which constructs the best update of the current parameters of the network as defined in Theorem 3.2.2 and the new neurons as defined in Theorem 3.2.3. We show that the best update and the new neurons can be obtained very similarly with one forward and one backward pass and that their computations complexities are comparable with the usual gradient procedure. We apply our module TINYpub to search for architecture on three academic datasets MNIST [LeCun et al., 1998], CIFAR10 and CIFAR100 [Krizhevsky et al., 2009], and show, empirically, that a naive strategy on top of Theorems 3.2.2 and 3.2.3 can grow neural network architectures to zero training loss within fewer time steps than the theoretical bound of Section 3.3.

We start this chapter by presenting the main functions of `TINYpub` and their complexities. Then, we provide proofs of concept for our search strategy on MNIST and CIFAR10 datasets, and, finally, on CIFAR100 dataset, we compare our growing technique with other existing methods. All our code and experiments are available at `https://gitlab.inria.fr/mverbock/tinypub/thesis`.

## 5.1 Core strategies and associated complexities

When growing a network as in Chapter 3, we aim at reducing the expressivity bottleneck at all the layers $l$ of the network architecture. This is done by computing, at each layer, the best update $\delta \boldsymbol{W}^*$ at layer $l-1$ (Theorem 3.2.2), and then, by computing the new neurons $\{\boldsymbol{\alpha}_i, \boldsymbol{\omega}_i\}_k^K$ at layer $l-1$ (Theorem 3.2.3). Those two quantities are very alike and can be calculated with the same logic, as, by construction, they are symmetric in the covariance matrix of post-activities, i.e. $\boldsymbol{S} := \boldsymbol{B}\boldsymbol{B}^T$ and in the covariance matrix of post-activities with desired update $\boldsymbol{N} := \boldsymbol{B}{\boldsymbol{V}_{\text{goal}}}^T$. This property is illustrated by the following lemma :

**Lemma 5.1.1.** *Considering a batch of size $n$ to construct the post activation and the desired update matrices, then for any depth $l$ and $z \in \{1, 2\}$, noting $\boldsymbol{S}_{-z} := \frac{1}{n}\boldsymbol{B}_{l-z}\boldsymbol{B}_{l-z}^T$ and $\boldsymbol{N}_z := \frac{1}{n}\boldsymbol{B}_{l-z}\big(I_d - \mathbb{1}_{z=2}\boldsymbol{B}_{l-1}^T\boldsymbol{S}_{-1}^+\boldsymbol{B}_{l-1}\big){\boldsymbol{V}_{goal}}_l^T$, we have that*

$$\delta \boldsymbol{W}_l^* = F_{BU}^l(\boldsymbol{S}_{-1}, \boldsymbol{N}_{-1}) \tag{5.1}$$

$$\{\boldsymbol{\alpha}_i^*, \boldsymbol{\omega}_i^*\}_k^K = F_{NN}^l(\boldsymbol{S}_{-2}, \boldsymbol{N}_{-2}) \tag{5.2}$$

*where the functions $F_{BU}^l$ and $F_{NN}^l$ are defined respectively in Theorem 3.2.2 and Theorem 3.2.3 and their implementations are given in Algorithm 5 and Algorithm 6.*

The functions $F_{BU}$ and $F_{NN}$ perform similar operations, which are: inverting or performing of $\boldsymbol{S}_{-z}$ (which equivalent in terms of basic operations) and finally

multiplying $\boldsymbol{S}_{-z}$ with $\boldsymbol{N}_{-z}$. Consequently, the time complexity of $F_{NN}^l$ is the same as the function $F_{BU}^l$, which will be detailed now. Define the function :

$$F_{BU} \ : \ l \in \{1, ..., L\} \mapsto \delta \boldsymbol{W}^* \tag{5.3}$$

The following lemma holds :

**Lemma 5.1.2.** *Note $\mathcal{C}(F)$ the computational complexity of the function $F$ as the number of basic operations needed to compute $F(l)$ for any $l \in \{1, ..., L\}$ with a batch of size $n$, then, using the notations of Figure 5.1 and assuming that all layers have the same width or kernel, it follows that, for a batch of size $n$ :*

$$\mathcal{C}\left(F_{BU}\right) = \mathcal{C}\left(F_{GD}\right) + O\left(nP(SW)^2 + (SW)^3\right) \tag{5.4}$$

*Where the function $F_{GD}$ computes the usual gradient update at one layer.*

*Proof of Theorem 5.1.2 :*
We assume that all layers have the same width (W) or kernel (S) and we use the notations of Figure 5.1. Furthermore, for a linear layer, we have $S = P = 1$. We take a batch of size $n$ and perform a forward and backward pass to obtain the matrices ${\boldsymbol{V}_{\text{goal}}}^l$ and $\boldsymbol{B}_{l-1}$.
For linear layers, $\boldsymbol{B}_{l-1}$ and ${\boldsymbol{V}_{\text{goal}}}^l$ are in $\mathbb{R}^{W,n}$, and, for convolutional layers, $\boldsymbol{B}_{l-1}$ and ${\boldsymbol{V}_{\text{goal}}}^l$ are respectively in $\mathbb{R}^{nP,SW}$ and $\mathbb{R}^{nP,W}$ (see Theorem C.1.1). Hence computing the associated matrices $\boldsymbol{S}_{-1}$ and $\boldsymbol{N}_{-1}$, has a cost in $O(nW^2) = O(n\overbrace{P}^{=1}(\overbrace{S}^{=1}W)^2)$ for linear layers and in $O(nP(SW)^2 + nPW^2) = O(nP(SW)^2)$ for convolutional layers.
At this point, the matrices $\boldsymbol{S}_{-1}$ and $\boldsymbol{N}_{-1}$ are in $\mathbb{R}^{SW,SW}$ and $\mathbb{R}^{SW,W}$ for both the linear and convolutional cases. We invert the matrix $\boldsymbol{S}_{-1}$, that is an operation at cost $(SW)^3$ and finally we multiply $\boldsymbol{S}_{-1}^+$ with $\boldsymbol{N}_{-1}$ which is in $(SW)^2W$ operations.
In total, updating the architecture with the best update instead of the gradient descent requires an extra computational cost of $O\left(nP(SW)^2 + (SW)^3\right)$.

Once the best update and the new neurons are computed, a line search is performed on those quantities as defined in Section 4.2.1. This operation is as costly as doing a forward pass, as the loss is evaluated for a specific batch at different values of $\gamma$, and $\gamma^*$ is chosen as the one that decreases the loss the most.

We now compare this computational cost with the complexity of a standard update with a gradient descent considering a batch of size $n_{GD}$, where $GD$ stands for gradient descent. Performing the forward and backward pass has a complexity of $n_{GD}W^2SPL$. Considering adding neurons at all the layers, the relative added complexity w.r.t. the standard training part is thus:

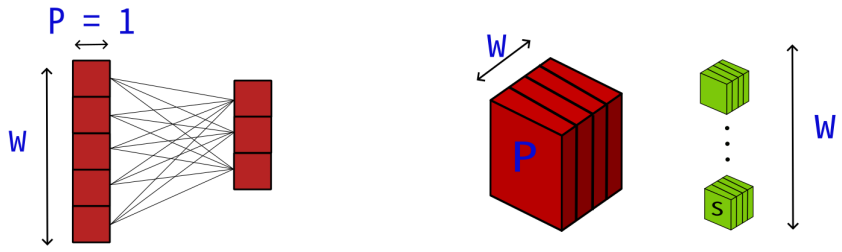



Figure 5.1: Notation and size for convolutional and linear layers, P : number of pixels by channel, S : kernel size, W : number of filter or neurons. For linear layers, we take the convention that $S = P = 1$.

$$\frac{L\left(nP(SW)^2 + (SW)^3\right)}{n_{GD}W^2SPL} = \overbrace{\frac{nS}{n_{GD}}}^{\substack{\text{Computing} \\ \boldsymbol{S} \text{ and } \boldsymbol{N}}} + \overbrace{\frac{S^2W}{n_{GD}P}}^{\substack{\text{Performing} \\ \text{SVD}}} \tag{5.5}$$

In the fully connected network case, $S = 1$, $P = 1$, and the relative cost of the SVD is then $W/n_{GD}$. It is then negligible, as layer width $W$ is usually much smaller or comparable than $n_{GM}$, which is typically $10^2$, for instance. In the convolutional case, $S = 9$ for $3 \times 3$ kernels, and $P \approx 1000$ for CIFAR, $P \approx 100000$ for ImageNet, so the SVD cost is negligible as long as layer width $W \ll 10000$ or 1 000 000 respectively. So, one needs no worrying about SVD cost. On the contrary, estimating the matrices (to which SVD is applied) can be more resource-demanding. The factor $nS/n_{GD}$ can be large if the minibatch size $n$ needs to be large for statistical significance reasons. One can show that an upper bound to the value required for $n$ to ensure estimator precision (appendix D.4) is $(SW)^2/P$. In that case, if $W > \sqrt{n_{GD}P/S^3}$, these matrix estimations will get costly. In the fully connected network case, this means $W > \sqrt{n_{GD}} \approx 10$ for $n_{GD} = 100$. In the convolutional case, this means $W > \sqrt{n_{GD}P/S^3} \approx 60$ for CIFAR and $\approx 600$ for ImageNet. It is possible to work on finer variance estimation and on other types of estimators to decrease $n$ and, consequently, this cost. Actually $(SW)^2/P$ is just an upper bound on the value required for $n$, which might be much lower, depending on the rank of computed matrices.

**In practice.** In our experiments the cost of a full training with TINY architecture growth approach is similar (sometimes a bit faster, sometimes a bit slower) than a standard gradient descent training using the final architecture from scratch. This is great as the right comparison should take into account the number of different architectures to try in the classical neural architecture search approach. Therefore, it is possible to get layer width hyper-optimization for free.

**Algorithm 5:** BestUpdate

Data: $l$ : index of a layer, n : batch size
Result: Best update at $l$
Take a minibatch $\mathbf{X}$ of size $n$;
Compute $(\mathbf{S}, \mathbf{N})$ with MatrixSN($l$, $-1$, $n$);
$\delta \mathbf{W}_l = \boldsymbol{N}^T \boldsymbol{S}^{-1}$;

**Algorithm 6:** NewNeurons

Data: $l$ : layer to add neurons, $\delta \boldsymbol{W}$ : best update at $l+1$, $n$ : batch size
Result: Best neurons at $l$
$\boldsymbol{S}, \boldsymbol{N} = \text{MatrixSN}(l+1, -2, n, \delta \boldsymbol{W} = \delta \boldsymbol{W})$;
Compute the SVD of $\boldsymbol{S} := \boldsymbol{Q} \Sigma \boldsymbol{Q}^T$;
Compute the SVD of $\boldsymbol{Q} \sqrt{\Sigma}^{-1} \boldsymbol{Q} \boldsymbol{N} := \tilde{\boldsymbol{A}} \Lambda \boldsymbol{\Omega}^T$;
$\boldsymbol{A} = \boldsymbol{Q} \Sigma^{-\frac{1}{2}} \boldsymbol{Q}^T \tilde{\boldsymbol{A}}$;

**Algorithm 7:** MatrixSN

Data: $l$ : layer, $z$ : index, n : batch size, $\delta \boldsymbol{W}$ = None : best update
Result: Matrices $\boldsymbol{S}$ and $\boldsymbol{N}$
Take a minibatch $\mathbf{X}$ of size $n$;
Propagate$\mathbf{X}$ and backpropagate ;
Compute $\boldsymbol{V}_{\text{goal}}$ at $l$, ie $-\frac{\partial \mathcal{L}^{tot}}{\partial \boldsymbol{A}_l}$;
if $\delta \boldsymbol{W} \neq$ None then
  $\mathbf{V}_{goal}$ $-= \delta \boldsymbol{W} \boldsymbol{B}_{l-1}$
end
$\mathbf{S}, \mathbf{N} = \frac{1}{n} \boldsymbol{B}_{l-z} {\boldsymbol{B}_{l-z}}^T, \frac{1}{n} \boldsymbol{B}_{l-z} {\boldsymbol{V}_{\text{goal}}}^T$;

**Algorithm 8:** Amplitude Factor

Data: $\delta \boldsymbol{W} = None$ : best update, $\{\boldsymbol{\alpha}_i, \boldsymbol{\omega}_i\}_{i=1}^K = None$ : new neurons ,
  $s = 1e-4$ : threshold, $exp = 2$ : speed, $M = 0$ : batch size
Result: Best amplitude factor $\gamma^*$
Take a minibatch $\mathbf{X}$ of size $M$;
Evaluate the loss $\mathcal{L}_{ref}$ on $\mathbf{X}$;
Normalize $\delta \boldsymbol{W}$ or $\{\boldsymbol{\alpha}_i, \boldsymbol{\omega}_i\}_{i=1}^K$ by its norm (4.17);
for $\gamma \in [exp^k$ for $k$ in range$(\lfloor -8 \log_{exp}(10) \rfloor, 1)$ do
  Scale $\delta \boldsymbol{W}$ or $\{\boldsymbol{\alpha}_i, \boldsymbol{\omega}_i\}_{i=1}^K$ with $\gamma$ and update the architecture with it;
  Evaluate the loss $\mathcal{L}_\gamma$ on $\mathbf{X}$;
  Undo the actions of the line in purple;
end
$\gamma^* = \arg\min_\gamma \mathcal{L}_\gamma$;
if $\frac{\mathcal{L}_{ref} - \mathcal{L}_{\gamma^*}}{\mathcal{L}_{ref}}$ < s then
  $\gamma^* = 0$;
end

## 5.2 Proof of concept

In this section, we design a naive search strategy on top of the module `TINYpub`, and show, on MNIST and CIFAR10 datasets, that we can reach full expressivity on the training set within fewer steps than the theoretical upper bound of Theorem 3.3.2. To achieve such objective and to prove that this property is only the consequence of the reduction of the expressivity bottleneck, we remove the usual gradient descent from the optimization process and only update neural network architecture with the best update from Theorem 3.2.2 and the addition of the new neurons from Theorem 3.2.3, both scaled with an amplitude factor. To remove the variation in performance estimation and to ensure the strict monotonicity of the performance, the new neurons, the best updates, and the amplitude factors are estimated on the whole training set.

### 5.2.1 MNIST

This section focuses on the academic dataset MNIST. All plots are averaged over three independent runs and have been performed on four CPUs. The associated notebooks can be found in the folder `thesis/MNIST/` of the module `TINYpub`.

To plot Figure 5.2, we initialize randomly a feed-forward network with two hidden layers with *selu* activation function and one neuron per hidden layer. The network is then grown by iteratively looping 10 times on both layers, and at each layer $l$, increasing the architecture with the $10^{th}$ first neurons of Theorem 3.2.3. After each neurons addition, the architecture is updated with the best update at all its layers. Each computed quantity ($\{\boldsymbol{\alpha}_i, \boldsymbol{\omega}_i\}_{i=1}^{10}$ or $\delta \boldsymbol{W}^*$) is scaled with an amplitude factor (Section 4.2.1) before it is added to the architecture. As stated, the best update, the new neurons, and the amplitude factor are estimated on the full training to remove variation in performance estimation. The algorithm for this experiment is described in Algorithm 9.

The results are plotted on Figure 5.2 where four variables are evaluated on all training or test sets as a function of time in seconds. In all the figures, a black vertical line has been drawn to indicate the time at which we start to observe overfitting.

The top figure of Figure 5.2 is the accuracy and the number of neurons per layer ($n_{layer}$) through time. The accuracy increases rapidly during the first 50 seconds, achieving 91.7 % on train and 90.4 on test at the black line; then, it increases very slowly and eventually almost overfits the training set with an architecture of 90 neurons per layer, which is far less than the theoretical upper bound of 50 000 of Theorem 3.3.2.

The middle plot is the ratio between the sum of the $10^{th}$ first eigenvalues $\lambda_k^2$ of Theorem 3.2.3 and the squared norm of the projected desired update. This ratio can be interpreted with Equation (3.22) as the ratio between the expressivity bottleneck solved by the new neurons ($\sum_k \lambda_k^2$) over the expressivity bottleneck which had to be solved ($\hat{\mathbb{E}}\left[||\boldsymbol{v}_{\text{goal}}(\boldsymbol{x})^l||^2\right] := \Psi_\theta^l$). This ratio goes quickly to 0 while

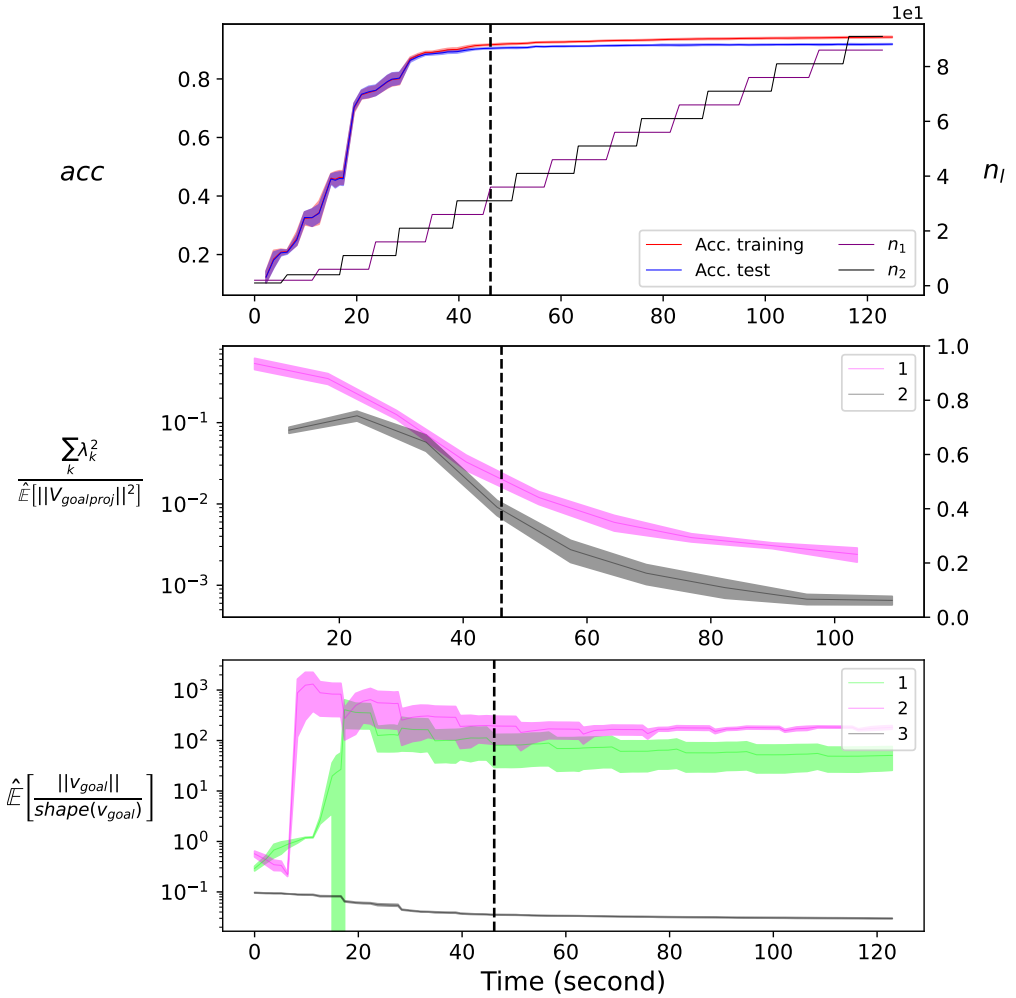



Figure 5.2: Experiments on MNIST for three independent runs. Top plot : Accuracy on full training and test set and number of parameters as a function of time; middle plot : ratio of expressivity bottleneck solved when adding neurons at layer $l$ as a function of time; bottom plot : norm of desired update divided by its shape at all the layers as a function of time. The $y$-axis is the same for all plots and is the time in seconds.

**Algorithm 9:** NaiveArchitectureGrowth

```
Data: 𝒜: starting architecture;
      𝒟_tr : training dataset, 𝒟_te : test dataset;
      T = 10 : number of iterations ;
Result: network N
Initialize network N with the architecture 𝒜;
L = depth(N) // number of layers of N
n = len(𝒟_tr) // size of the batch to estimate
              // the best update, the new neurons
              // and the amplitude factor
for j in range(T) do
    for l in {1, ..., L-1} do
        δW*_{l+1} = BestUpdate(l + 1, n);
        {α_i, ω_i}_{i=1}^K = NewNeurons(l, n, δW*_{l+1});
        take only the 10^th first neurons, ie K = 10;
        γ* = AmplitudeFactor({α_i, ω_i}_{i=1}^10, M = n);
        if γ* > 0 then
            {α_i, ω_i}_{i=1}^10 ← {√γ* α_i, √γ* ω_i}_{i=1}^10;
            Update the architecture with those neurons;
        end
        Compute the accuracy a_tr, a_tr on 𝒟_tr and 𝒟_te;
        for d in {1, ..., L} do
            δW*_d = BestUpdate(d, n);
            γ* = AmplitudeFactor(δW*_d, M = n);
            δW*_d ← γ* δW*_d;
            update the architecture with δW*_d as W_d ← W_d + δW*_d;
            Compute the accuracy a_tr, a_te on 𝒟_tr and 𝒟_te;
        end
    end
end
```

we observe overfitting, which indicates that the added neurons are not significantly decreasing the expressivity bottleneck and might be unnecessary to the growth.

The bottom plot is the evolution of the norm of the desired update divided by its shape during the experiment. At the beginning of the experiments, while the network architecture is thin, we remark that the norm of that variable is close to zero. It is because no information can go through the network, as the loss function might not depend on $\boldsymbol{a}_1$, making the functional derivative $\nabla_{\boldsymbol{a}_1}\mathcal{L}$ almost null. This phenomenon is highly visible when using the relu activation functions on thin architectures as the post-activity $\boldsymbol{b}_1$ (after the relu) might return 0 for almost every $\boldsymbol{x}$ inducing $\nabla_{\boldsymbol{a}_1}\mathcal{L} = 0$. However, the norm of the desired update increases as the network grows and as relevant information in pre-activity $\boldsymbol{a}_l$ could be used to decrease the loss. At the end of the experiment, the norm of the desired update stagnates, and as explained in Section 4.2.1, we might not compare the norm of the desired update between layers as the weight matrices artificially modify the magnitude of ${\boldsymbol{v}_{\text{goal}}}^l$ from one layer to another.

The line colors are matched according to the layers of interest, that is, the number of neurons in layer $l$ for the top plot is of the same color as the evaluation of the ratio when adding neurons at $l$ for the middle plot, that is itself of the same color as the evaluation of the desired update at layer $l+1$ for the bottom plot, as neurons are added at $l$ to reduce the expressivity bottleneck at $l+1$.

### 5.2.2 CIFAR10

In this section, the same type of experiment is performed but for the CIFAR10 dataset. As before, the objective is to overfit the training set in a few steps. All the curves are averaged over three independent runs and are performed on four CPUs, and all the plots and scripts can be found in the folder `thesis/CIFAR10` of the module TINYpub. The growth protocol is the same as for the MNIST experiment and is described in Algorithm 9.

While only one starting architecture was tested in the previous section, here, three types of starting architectures are considered. Lets note `kC-k'L` a feed-forward architecture with $k$ convolutional layers followed by $k'$ fully connected layers, all with the *selu* activation function. Using this notation, we performed the same experiment as in Section 5.2.1 but with the starting architecture `2C-2L`, `5C-1L` and `1C-5L` with one neuron per hidden layer. The performances are summarized on Table 5.1 and the details for each starting architecture are in the Figures 5.3 to 5.5. The variables of the figures are the same as for Figure 5.2, but the first plot has been separated into two parts with the accuracy in the top plot and the number of neurons below.

At the end of each experiment on Figure 5.3 (`2C-2L`) and Figure 5.4 (`5C-1L`), we notice that the relative bottleneck solved order themselves according to the depths, in decreasing order for the middle plot and in increasing order for the last plot. In Table 5.1, we note that the networks `2C2L` and `5C1L` achieve full expressivity on the training set but also exhibit strong overfitting, as their accuracy on the

| archi. | Accuracy | | Final Complexity | | $T^*$ (seconds) |
|---|---|---|---|---|---|
| | Train | Test | Parameters | Operations | |
| `2C2L` | $1.0 \pm 0.0$ | $0.37 \pm 0.00$ | $3.8\ 10^5$ | $4.8\ 10^6$ | $3.5\ 10^4$ |
| `5C1L` | $1.0 \pm 0.0$ | $0.30 \pm 0.00$ | $1.6\ 10^5$ | $2.2\ 10^6$ | $1.2\ 10^5$ |
| `1C5L` | $0.7 \pm 0.1$ | $0.22 \pm 0.02$ | $3.8\ 10^5$ | $9.6\ 10^5$ | $8.0\ 10^4$ |

Table 5.1: Accuracy, memory, and time complexity of growing networks starting from the architectures `2C2L`, `5C1L` and `1C5L` on CIFAR10 dataset for two independent runs. Accuracy and complexity are evaluated at $t = T^*$ where $T^*$ is the time of search in seconds at which the networks reach 99% accuracy on the training set with 4 CPUs. As full expressivity is not achieved for the architecture `1C5L`, its $T^*$ is set to the duration of the overall experiment. The accuracy is evaluated on the overall train and test dataset, and the complexity is measured as the number of parameters and the number of basic operations performed at the test.

test set saturates around 30%. for all architectures. This overfitting was expected, as the CIFAR-10 dataset is more complex than the MNIST dataset. To achieve better generalization, one would need to use a more advanced architecture, such as ResNet or VGG. Additionally, we observe that the search time $T^*$ in Table 5.1 is quite large (over 10 hours!). However, this could have been significantly reduced. First, by stopping the experiments earlier—for instance, when there is no further improvement in test set accuracy. Second, we chose to estimate new neurons and determine the best update using the entire dataset. Instead, we could have used a smaller subset for these estimations, thereby reducing the computational cost of each step in the growth process.

## 5.3 CIFAR100

In this section, we grow a neural network on CIFAR100 dataset and compare our new neurons initialization strategy with the GradMax and the random approaches.

### 5.3.1 Comparison with GradMax

In this section, TINY method is compared with the GradMax method of Evci et al. [2022]. While the theoretical comparison with that method has already been done in Section 4.1.1, here an empirical comparison is provided. To ensure an instructive comparison with the original paper of GradMax, the experiment of their paper have been reproduced but with various settings. In particular, the architecture of a ResNet18 is grown starting from either a quarter or from a sixty-fourth of its usual number of neurons, and, between each neuron addition, the network

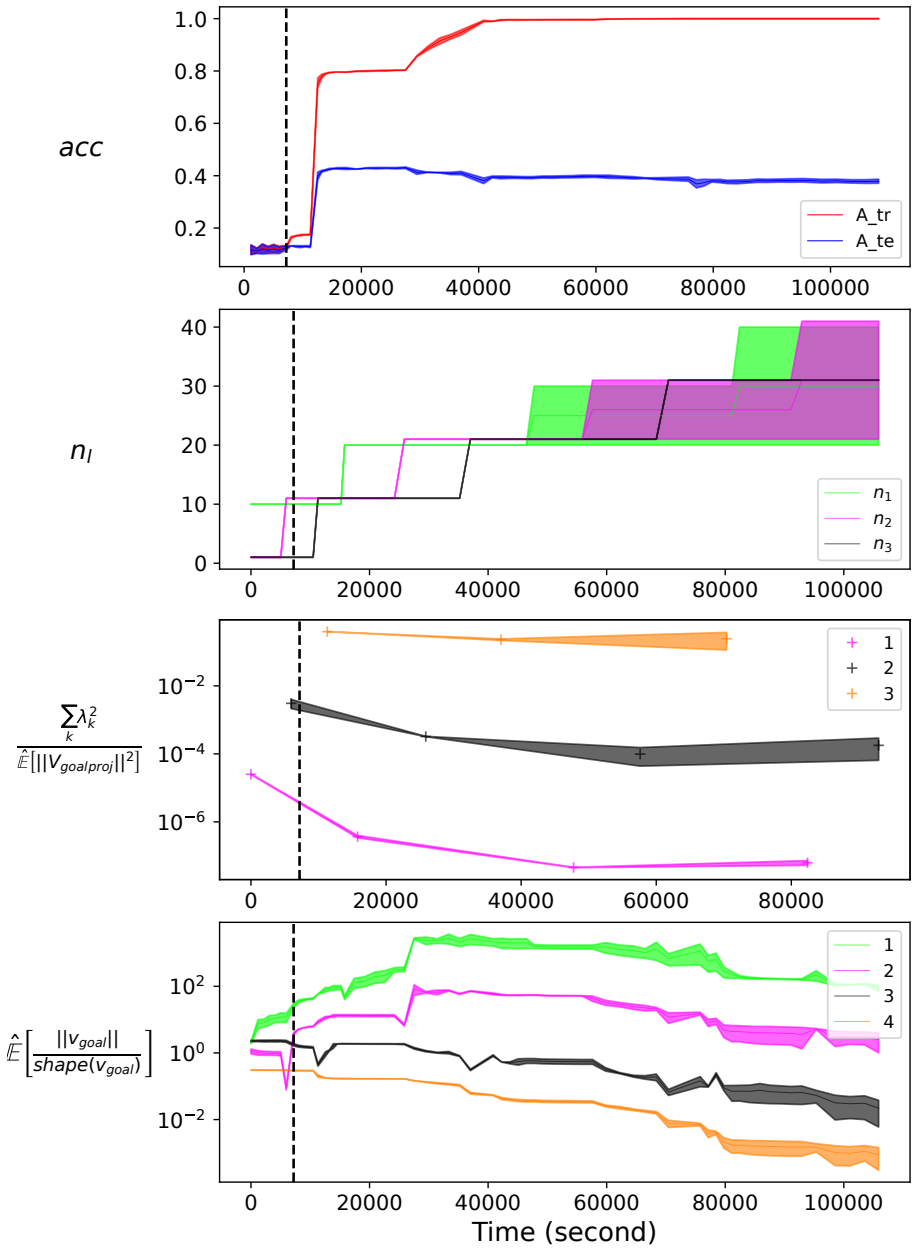



Figure 5.3: Experiments on CIFAR10 for two independent runs starting with the architecture 2C2L. Top plot : Accuracy on full training and test set; second plot : number of parameters as a function of time; third plot : ratio of expressivity bottleneck solved when adding neurons at layer $l$ as a function of time; bottom plot : norm of desired update divided by its shape at all the layers as a function of time. The $y$ axis is the same for all plots and is the time in seconds.

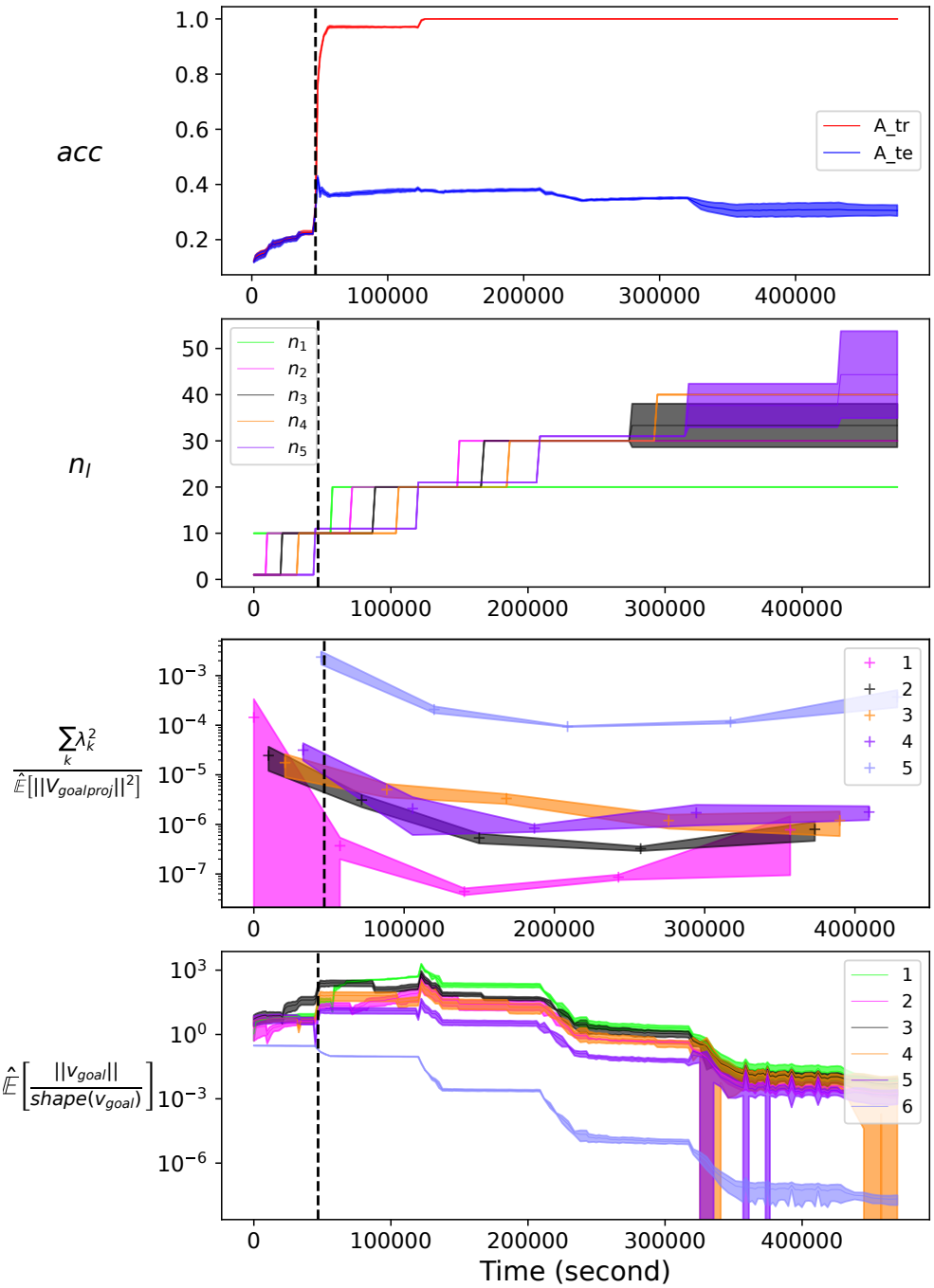



Figure 5.4: Same description as Figure 5.3 but for the starting architecture `5C1L`

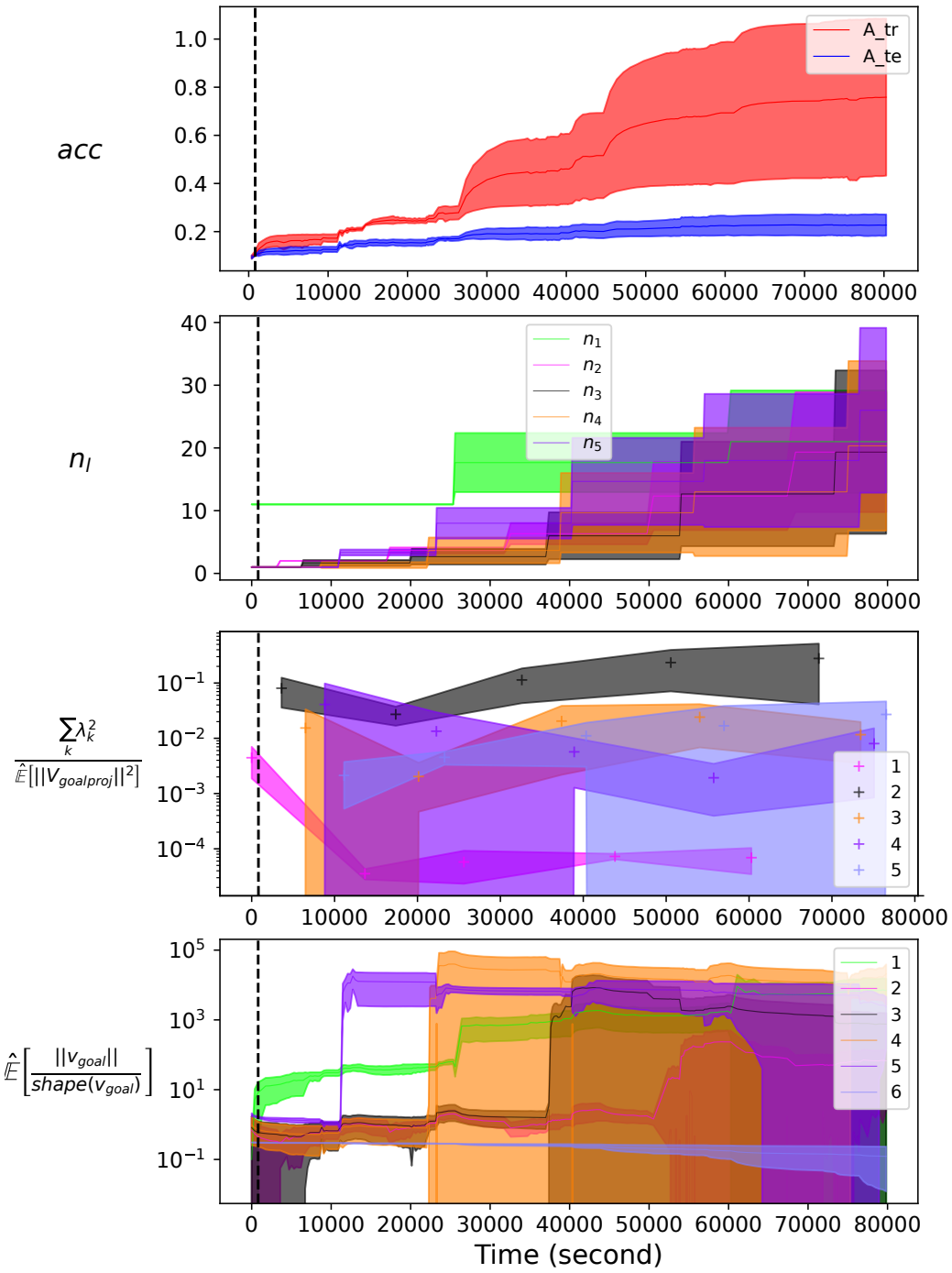



Figure 5.5: Same description as Figure 5.3 but for the starting architecture `1C5L`

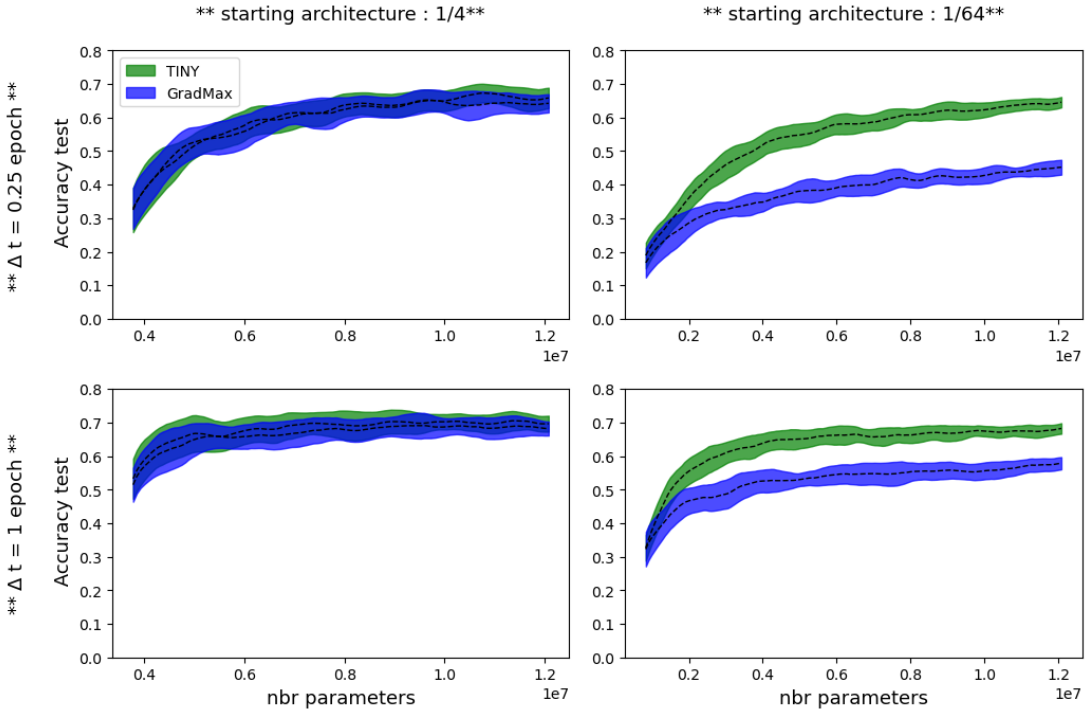



Figure 5.6: Test accuracy as a function of the number of parameters during architecture growth from $\text{ResNet}_s$ to ResNet18. The left (resp. right) column is for the starting architecture $\text{ResNet}_{1/4}$(resp. $\text{ResNet}_{1/64}$). The upper (resp. lower) row is for $\Delta t$ equal to 0.25 (resp. 1) epoch. Each line corresponds to an independent run; 4 runs are performed for each setting.

is trained for a quarter of an epoch or for a full epoch. In the experiment of the original paper of GradMax, the network starts with one-fourth of its architecture, and it is trained until convergence between each neuron addition.

By construction, the objective of GradMax is to decrease the loss as fast as possible considering an infinitesimal increment of new neurons. The main difference is that GradMax does not take into account the expressivity of the current architecture as TINY does in Equation (3.21) by projecting $v_{\text{goal}}$ (see Section 4.1.1). In the following experiment, it is shown on the CIFAR-100 dataset that solving Equation (3.21) instead of Equation (4.3) (defined by GradMax) to grow a network using a naive strategy allows better final performance and almost full expressivity power. To do so, the GradMax method has been re-implemented and its growing process, which consists of increasing the architecture of a thin ResNet18 until it reaches the architecture of the usual ResNet18, has been mimicked. This process is described in the pseudo-code Algorithm 11, where two parameters can be chosen : the relative thinness $s$ of the starting architecture, w.r.t. the usual ResNet18 architecture Table D.2 ($s = 1/4$ or $s = 1/64$), and the amount of training time

between consecutive neuron additions ($\Delta t = 1$ or $\Delta t = 0.25$ epochs). The function `NewNeuron` is also redefined in Algorithm 10 to compute the new neurons for the GradMax and TINY methods. The number of parameters and the performance of the growing network are evaluated at regular intervals to plot Figure 5.6. We note that both methods reach their final accuracy within less than one GPU day, outperforming, by far, other NAS search methods in their category (cf Figure 2.6).

Once the models have reached the final architecture ResNet18, they are trained for 250 epochs (or 500 epochs if they have not converged on the training set). the final performances are summarized in Table 5.2. We added Table 5.3, which gives the performance of a ResNet18 trained from scratch by the usual gradient descent with all its neurons,. It is not expected that TINY or GradMax achieve the performance of the reference, as its architecture and optimization process have been optimized for years. The column small references corresponds to the training of the architecture $\text{ResNet18}_s$ for $s = 1/4$ and $s = 1/64$ using standard gradient descent without any increase in architecture.

The details of the protocol can be found in the annexes (Appendix D.3), as well as other technical details such as the dynamic of the training batch size (Appendix D.4.1) and the number of examples used to estimate and solve the expressivity bottleneck (Appendix D.4). For both methods, all the latter apply so that the main difference between GradMax and TINY in this experiment is the mathematical definition of the new neurons.

| | | TINY | | GradMax | |
|---|---|---|---|---|---|
| | $\Delta t$ | 0.25 | 1 | 0.25 | 1 |
| s | 1/4 | $67.2 \pm 0.1$ | $70.4 \pm 0.2$ | $65.1 \pm 0.2$ | $68.6 \pm 0.1$ |
| | 1/4 | $70.3 \pm 0.2$ $^{5*}$ | $70.9 \pm 0.2$ $^{5*}$ | $67.0 \pm 0.2$ $^{5*}$ | $69.0 \pm 0.2$ $^{5*}$ |
| | 1/64 | $65.8 \pm 0.1$ | $68.1 \pm 0.5$ | $45.0 \pm 0.4$ | $56.8 \pm 0.2$ |
| | 1/64 | $69.5 \pm 0.2$ $^{5*}$ | $68.7 \pm 0.6$ $^{5*}$ | $57.0 \pm 0.4$ $^{10*}$ | $58.4 \pm 0.2$ $^{10*}$ |

Table 5.2: Final accuracy on test of ResNet18 after the architecture growth (*grey*) and after convergence (*blue*). The number of stars indicates the multiple of 50 epochs needed to achieve convergence. With the starting architecture $\text{ResNet}_{1/64}$ and $\Delta t = 0.25$, the method TINY achieves $65.8 \pm 0.1$ on test after its growth and it reaches $69.5 \pm 0.2$ $^{5*}$after $5* := 5 \times 50$ epochs (examples of training curves for the extra training in Figure D.3). Mean and standard deviation are estimated on 4 runs for each setting.

For $s = 1/64$, that is, thin start, we observe a significant difference in performance between TINY and GradMax methods. While TINY models almost achieve the reference's performance, GradMax remains stuck 10 points below. This suggests that the framework proposed by GradMax is not sufficient to be able to start

**Algorithm 10:** NewNeurons

```
Data: l : layer to add neurons, δW : best update at l + 1
Result: Best neurons at l
if method == TINY then
    δW = BestUpdate(l+1)
else
    δW = None
end
S, N = MatrixSN(l + 1, −2, δW = δW);
Compute the SVD of S = UΣU^T;
Compute the SVD of U√Σ^{-1}UN = AΛΩ;
Use the columns of A, the lines of Ω and the diagonal of Λ to construct
 the new neurons of Prop.Theorem 3.2.3;
if method == GradMax then
    A = 0
end
```

**Algorithm 11:** Algorithm to plot Figure 5.6 and Figure 5.9.

```
for each method [TINY, MethodToCompareWith] do
    Start from neural network N with initial structure s ∈ {1/4, 1/64};
    while N architecture does not match ResNet18 width do
        for d in {depths to grow} do
            θ^{K*}_↔ = NewNeurons(d, method) ;
            Normalize θ^{K*}_↔ according to Appendix D.4.2;
            Add the neurons at layer d ;
            Train N for Δt epochs ;
            Save model N and its performance ;
        end
    end
end
```

| Small Reference $s = 1/64$ | Small Reference s = 1/4 | Reference |
|---|---|---|
| 63.4± 0.3 [5*] | 68.6± 0.4 [5*] | 72.8± 0.3 [5*] |

Table 5.3: Accuracy for the references architecture trained from scratch

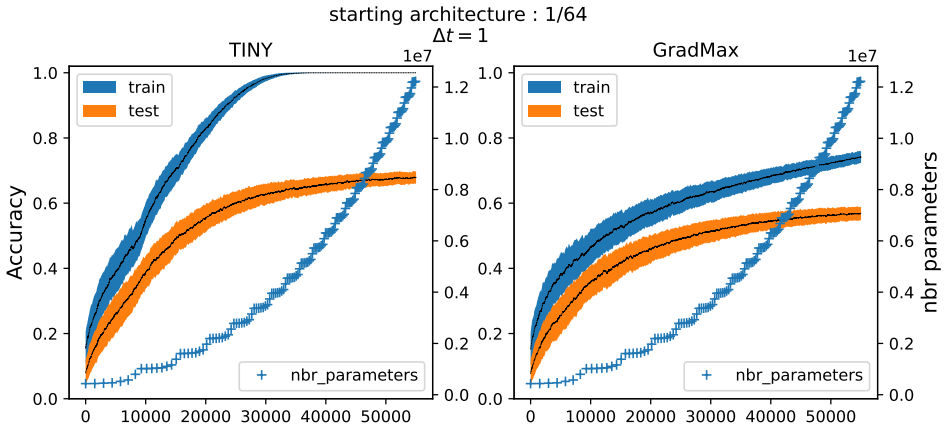



Figure 5.7: Evolution of accuracy and number of parameters as a function of the gradient step for the setting $\Delta t = 1$, $s = 1/64$ for TINY and GradMax, mean and standard deviation over four runs. Other settings in the annexes Figure D.2

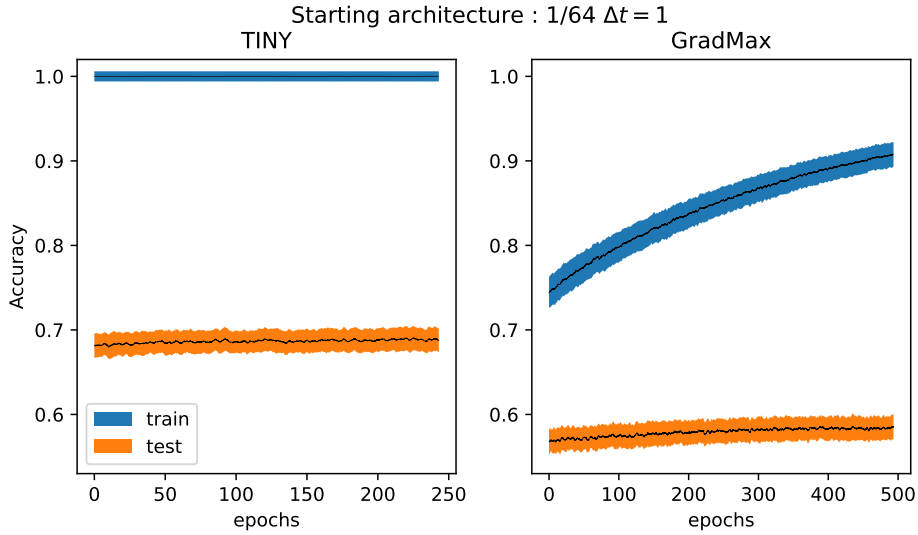



Figure 5.8: Evolution of accuracy and number of parameters as a function of the gradient step for the setting $\Delta t = 1$, $s = 1/64$ during extra training for TINY and GradMax, mean and standard deviation over four runs. Other settings in the annexes Figure D.3.

| Method | | Indicators | | | Dataset |
|---|---|---|---|---|---|
| | | Arch. | Time | Acc. | |
| GradMax$^{\dagger}$ | $\Delta t = 0.25, s = 1/64$ | ResNet-18 | 0.12 | $45.0 \pm 0.4$ | CIFAR-100 |
| | $\Delta t = 1, s = 1/64$ | ResNet-18 | 0.26 | $56.8 \pm 0.2$ | |
| TINY$^{\dagger}$ | $\Delta t = 0.25, s = 1/64$ | ResNet-18 | 0.13 | $65.8 \pm 0.1$ | |
| | $\Delta t = 0.1, s = 1/64$ | ResNet-18 | 0.28 | $68.1 \pm 0.5$ | |

Table 5.4: *Time*: GPU days spent to search for the architecture and to train it. *Acc.*: accuracy on test set (%). $\dagger$ : estimated with our implementation.

with an architecture far from full expressivity, i.e. $\text{ResNet}_{1/64}$, while TINY is able to handle it. As for the setting $s = 1/4$, both methods seem equivalent in terms of final performance and achieve full expressivity.

The curves on Figure 5.7, which are extracted from Figure D.2 in the appendix, show that TINY models have converged at the end of the growing process, while GradMax ones have not. This latter effect contrasts with the GradMax formulation, which is to accelerate the gradient descent as fast as possible by adding neurons. Furthermore, GradMax needs extra training to achieve full expressivity: for the particular setting $s = 1/64, \Delta t = 1$, the extra training time required by GradMax is twice as high as TINY's, as shown in Figure 5.8. This need for extra training also appears for all settings in Table 5.2. In particular, for $s = 1/64, \Delta t = 0.25$, the difference in performance after and before extra training goes up to 20 % above the initial performance while it is only of 6% for TINY.

### 5.3.2 Comparison with Random on CIFAR-100 : initialisation impact

This section focuses on the impact of the new neurons' initialization. We consider as a baseline the Random method, which initializes the new neurons according to a Gaussian distribution: $(\alpha_k^*, \omega_k^*)_{k=1}^K \sim \mathcal{N}(0, I_d)$ or a uniform distribution $\mathcal{U}[-1, 1]$. Also, when adding new neurons, the best scaling is searched using a line search on the loss. Thus, the operation $\theta_{\leftrightarrow}^K \leftarrow \gamma^* \theta_{\leftrightarrow}^K$, is performed with the amplitude factor $\gamma^* \in \mathbb{R}$ defined in Equation (4.18) and that is recalled here:

$$\gamma^* := \underset{\gamma \in [-L, L]}{\arg\min} \sum_i \ell(f_{\theta \oplus \gamma \theta_{\leftrightarrow}^K}(\boldsymbol{x}_i), \boldsymbol{y}_i) \qquad \text{with } \gamma \theta_{\leftrightarrow}^{K^*} = (\sqrt{\gamma}\alpha_k^*, \sqrt{\gamma}\omega_k^*)_{k=1}^K \quad (5.6)$$

with $L$ a positive constant, which is given in Algorithm 11. With such an amplitude factor, one can measure the quality of the directions generated by TINY and Random by quantifying the maximal decrease of loss in these directions.

No gradient descent step is performed in order to better measure the impact of the initialization method and distinguish it from the optimization process. This contrasts with the previous section where long training time after architecture

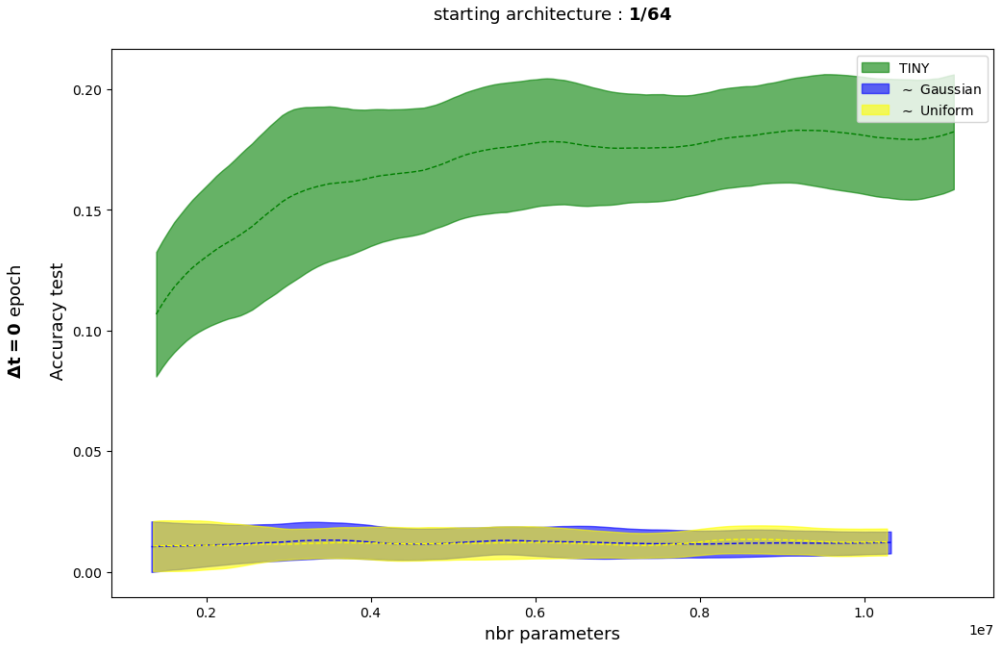



Figure 5.9: Test accuracy as a function of the number of parameters during pure architecture growth (no gradient steps performed, only neuron addition with a given initialization) from $\text{ResNet}_{1/64}$ to $\text{ResNet}_{18}$, averaged over four independent runs.

growth was modifying the direction of the added neurons, dampening initialization impact with training time, especially as they were added with a small amplitude factor (cf Appendix D.4.2).

With these two modifications to the protocol of the previous section, the Figure 5.9 can be plotted. The crucial impact of TINY initialization can be seen compared to the Random one. Indeed, TINY reaches more than 17% accuracy just by adding neurons (without any further update), which accounts for about one-quarter of the total accuracy with the full method (69% in Table 5.2 using in addition gradient descent). On the opposite, the Random initialization does not contribute to the accuracy through the growing process (just about 1%); this can be explained and quantified as follows.

To study the random setting, one can model $\boldsymbol{v}(X)$ and $\boldsymbol{v}_{\text{goal}}(X)$ as independent variables where $\boldsymbol{v}_{\text{goal}} \sim \mathcal{N}\left(0_d, \frac{1}{d} I_d\right)$ and $\boldsymbol{v}$ either $\sim \mathcal{N}\left(0_d, \frac{1}{d} I_d\right)$ or $\sim \mathcal{U}[-\frac{1}{d}, \frac{1}{d}]$, $d$ being the dimension of $\boldsymbol{v}_{\text{goal}}$ and $\boldsymbol{v}$. From Equation (3.23), the scalar product $\langle \boldsymbol{V}(X), \boldsymbol{V}_{\text{goal}}(X) \rangle := \frac{1}{n} \sum_i \boldsymbol{v}_{\text{goal}}(\boldsymbol{x}_i)^T \boldsymbol{v}(\boldsymbol{x}_i)$ is a proxy for the expected decrease of loss after each architecture growth. This quantity can be approximated by its standard deviation, ie $\frac{1}{\sqrt{n\,d}}$, which makes the expected relative gain of loss (for a gradient step) of the order of magnitude of $\frac{1}{\sqrt{64}}$ for the first layer and $\frac{1}{\sqrt{512}}$ for the

last layer when compared to the true gradient, and consequently when compared to TINY. Furthermore, one can take into account the effect of a line search over the random direction: in that case, the expected relative loss gain is quadratic in the angle between the directions and, therefore, of the order of magnitude of $\frac{1}{64}$ or $\frac{1}{512}$ respectively (see Appendix Appendix C.6.3).

Note that the search interval of Equation (5.6) can be shrunk to $[0, L]$ with TINY initialization, as the first order development of the loss in Equation (3.23) is positive. This property is the direct consequence of the definition of $\boldsymbol{V}^*$ as the minimizer of the expressivity bottleneck (Equation (3.21)). Also, note that GradMax was not included in Figure 5.9 because its protocol initializes the ongoing weights to zero ($\alpha_k \leftarrow 0$) and imposes a small norm on its outgoing weights ($||\omega_k|| = \varepsilon$). Those two aspects make the amplitude factor $\gamma^*$ meaningless and the impact of the new neuron initialization invisible without gradient descent.

## Conclusion

In this experiment section, we have shown that reducing the expressivity bottleneck is a self-sufficient objective for constructing a neural network architecture. The proposed method achieves better results than the random and the GradMax optimization processes, and is able to start from a very thin architecture, contrary to GradMax. In the module `TINYpub`, architecture growth optimization is already instantiated for linear and convolutional layers; extension to self-attention mechanisms (transformers) is part of future works. Although common architectures consist of a succession of layers, another research direction would be to develop tools for handling general computational graphs (such as U-net, Inception, and Dense-Net), which offers the possibility to let the architecture graph grow and bypass manual design.

# 6 Conclusion

In this manuscript, we have properly defined the expressivity bottleneck metric, which quantifies the lack of expressivity of a network at a given layer. This metric is defined as the minimal distance between what is asked by the backpropagation and what can be done by a change of parameter. In fact, it measures the amplitude of the directions that are needed by the current network but that are not yet expressed by its architecture. While quantifying that lack of expressivity, the metric also defines the best infinitesimal parameters move and the best infinitesimal neurons to add at each layer, both in close form, to reduce this lack of expressivity. This parameter move and extension of architecture naturally define the building blocks of a mathematically well-defined and easily computable NAS strategy.

The mathematical expression of the new neurons' parameters, defined by the expressivity bottleneck metric, is comparable to the initialization of the new neurons of two similar one-shot strategies: namely, the GradMax and the NORTH methods. GradMax initializes its neurons with the objective of decreasing the loss as fast as possible, and the NORTH method initializes its neurons randomly while avoiding redundancy within the current architecture. In fact, TINY initialization strategy bridges the gap between those two by initializing its new neurons with the aim of decreasing the loss as much as possible and by avoiding redundancy of its new neurons with the current ones.

For each new neuron defined with TINY, we have determined its first-order impact on the loss as a closed-form expression. This first-order impact ranks the new neurons according to their relevance when considering the expressivity bottleneck at a single layer. However, experiments and theory indicate that this first-order impact is no longer self-sufficient when one has to choose across the layers and decide at which depth neurons should be added first. In that case, it seems that estimating a non-linear impact of those new neuron additions on the loss is actually a more promising criterion, for instance with a line search as done in Section 4.2.1.

With a naive strategy on top of the expressivity bottleneck metric, we have shown, theoretically and empirically, that decreasing the expressivity bottleneck is a self-sufficient objective to achieve zero-loss on the training set, and that this objective can easily be coupled with the usual gradient descent process. Using the

Python module `TINYpub`, we have implemented a naive search strategy, and we have shown that the method can provide competitive results on academic datasets compared to other one-shot strategies on reference architectures.

## Future works

In this manuscript, we have proposed the theory and the application of the expressivity bottleneck for linear and convolutional layers; one could propose to extend the theory to attention layers and to graph neural networks. With these extensions, one could grow unusual architecture with various types of layers, hence studying novel and perhaps more adapted network designs for standard and non-standard tasks.

For general Directed Acyclic Graph (DAG), we can remark that adding a new layer in the graph is equivalent to growing an empty layer, which is compatible with the theory of Chapter 3 by considering that the output function of that empty layer is the null function. However, with a DAG, there exists configurations for which the best new neuron to add, as defined in Theorem 3.2.3, is none. Such phenomenon happens as soon as two activities are connected twice through two types of connections, forming altogether a triangle: let's consider a DAG where each post-activation is called a node, and each node is connected to another node through a linear application. Then, consider the nodes $a, b$ and $c$ connected in a triangle with $a$ connected to $b$ ($a \to b$), and $a$ connected to $c$, which is itself connected to $b$ ($a \to c \to b$). In that case and considering the first order approximation setting of Chapter 3, the functional space induced by an update of parameters of the connection between nodes $a$ and $b$ is equal to the functional space induced by a neuron addition at node $c$, and both are equal to $Span(a)$. It is to say that no neuron addition can expand the tangent space of achievable function of the current network, and therefore no neuron addition is needed (at first order). To ensure that those two functional spans are different, one can take into account the non-linearity of the activation function at node $b$ and search for the best neurons to add by reducing the expressivity bottleneck. While this problem has been shown to be NP-hard [Bach, 2017] with the ReLU activation function, it is still possible to approximate the solution by a local minimum using standard optimization methods. Those extensions have already been proposed in the paper Douka et al. [2025] (accepted at the ESANN conference and of which I am a co-author), where layers are added to a neural network by minimizing the expressivity bottleneck with the usual gradient descent and by taking into account the non-linearity of the activation function.

Another research direction would be to study the statistical reliability of the TINY method, for instance, using tools borrowed from the random matrix theory. Indeed, statistical tests can be applied to the intermediate computations from which the new neurons are obtained. An interesting byproduct of this approach would be to define a threshold to select neurons found by Theorem 3.2.3, based

on statistical significance. This selection could be the starting point of a search strategy at a given layer, and could be mixed with a strategy based on the amplitude factor to construct a strategy across the layers. Although the TINY method is self-sufficient in the sense that it can train and grow a neural network without the usual update of the gradient descent, one could also search to genuinely combine the usual gradient descent optimization of the parameters and the updates of the TINY method. The idea would be to exploit the computational efficiency of gradient descent and estimate when training should be preferred over growth.

These research directions are in progress with the implementation of a pytorch module called GrowMo (for Growing Module), that is developed in the TAU team.

# Glossary

**Bayesian neural network** A Bayesian neural network has the same structure as the standard neural network, but each of its weights is a random variable. 28, 38

**brute force** A brute force method is a method that checks all possibilities until the correct one is found. 25

**Gaussian process** A Gaussian process is a collection of random variables indexed by time or space such that every finite collection of those random variables has a multivariate normal distribution. 28, 30

**generalization** The generalization property of a model defines its capability to adapt and react properly to previously unseen, new data, which has been drawn from the same distribution as the one used to build the model. 25, 30, 31, 37, 38

**Hilbert space** A Hilbert space is a vector space equipped with an inner product that induces a distance function for which the space is a complete metric space. 51

**pocket algorithm** The pocket algorithm is a variant of the Perceptron learning algorithms where, at each iteration, one saves the model and its number of misclassified examples. At the end of the training iteration, the algorithm returns the solution that has the lowest number of misclassified examples. 30

**surrogate** A surrogate model is an engineering method used when an outcome of interest cannot be easily measured or computed, so an approximate mathematical model of the outcome is used instead. 28

**the Solomonoff's prior** The Solomonoff's theory of inductive inference defines the best possible scientific model as the shortest algorithm that generates the empirical data under consideration. 43

**trial and error** The trial and error process is a fundamental method of problem-solving characterized by repeated, varied attempts which are continued until success, or until the practicer stops trying. 16, 25, 44, 45

# Bibliography


Sylvain Arlot. Fondamentaux de l'apprentissage statistique. https://www.imo.universite-paris-saclay.fr/~sylvain.arlot/enseign/2020-21_M2/1-fondamentaux.pdf, 2020.

Francis Bach. Breaking the curse of dimensionality with convex neural networks. *The Journal of Machine Learning Research*, 18(1):629–681, 2017.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=S1c2cvqee.

Eric B Baum. On the capabilities of multilayer perceptrons. *Journal of complexity*, 4(3):193–215, 1988.

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 550–559. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/bender18a.html.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011. URL https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

Stéphane Boucheron, Olivier Bousquet, and Gábor Lugosi. Theory of classification: A survey of some recent advances. *ESAIM: probability and statistics*, 9:323–375, 2005.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Reinforcement learning for architecture search by network transformation. *CoRR*, abs/1707.04873, 2017. URL http://arxiv.org/abs/1707.04873.

T.-S. Chang and K.A.S. Abdel-Ghaffar. A universal neural net with guaranteed convergence to zero system error. *IEEE Transactions on Signal Processing*, 40 (12):3022–3031, 1992. doi: 10.1109/78.175745.

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

Aristeidis Chrostoforidis, George Kyriakides, and Konstantinos G. Margaritis. A novel evolutionary algorithm for hierarchical neural architecture search. *CoRR*, abs/2107.08484, 2021. URL https://arxiv.org/abs/2107.08484.

Jonas da Silveira Bohrer, Bruno Iochins Grisci, and Márcio Dorn. Neuroevolution of neural network architectures using codeepneat and keras. *CoRR*, abs/2002.04634, 2020. URL https://arxiv.org/abs/2002.04634.

Difan Deng and Marius Lindauer. Literature list on Neural Architecture Search, 2023. URL https://www.automl.org/automl/literature-on-neural-architecture-search/.

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

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 3460–3468. AAAI Press, 2015. ISBN 9781577357384.

Stella Douka, Manon Verbockhaven, Théo Rudkiewicz, Stéphane Rivaud, François P Landes, Sylvain Chevallier, and Guillaume Charpiat. Growth strategies for arbitrary dag neural architectures, 2025. URL https://arxiv.org/abs/2501.12690.

Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, September 1936. ISSN 1860-0980. doi: 10.1007/BF02288367. URL https://doi.org/10.1007/BF02288367.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019. URL http://jmlr.org/papers/v20/18-598.html.

Utku Evci, Bart van Merrienboer, Thomas Unterthiner, Fabian Pedregosa, and Max Vladymyrov. Gradmax: Growing neural networks using gradient information. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=qjN4h_wwUO.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018. URL http://arxiv.org/abs/1803.03635.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Arthur Jacot, Franck Gabriel, and Clement Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/5a4be1fa34e62bb8a6ec6b91d2462f5a-Paper.pdf.

Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. Neural architecture search with bayesian optimisation and optimal transport. *CoRR*, abs/1802.07191, 2018a. URL http://arxiv.org/abs/1802.07191.

Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018b. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf.

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=S11KBYclx.

Alex Krizhevsky, Geoffrey Hinton, et al. Object classifcation experiments. In *Learning multiple layers of features from tiny images*. Toronto, ON, Canada, 2009.

Jean-François Le Gall. *Intégration, Probabilités et Processus Aléatoires*. 2006.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998.

Guillaume Lecué. Approximations locales d une fonction differentiable de plusieurs variables, 2016. URL https://lecueguillaume.github.io/assets/fcts_plusieures_variables.pdf.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *CoRR*, abs/1902.07638, 2019. URL http://arxiv.org/abs/1902.07638.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJQRKzbA-.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=S1eYHoC5FX.

Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/933670f1ac8ba969f32989c312faba75-Paper.pdf.

Wolfgang Maass. Neural Nets with Superlinear VC-Dimension. *Neural Computation*, 6(5):877–884, 09 1994. ISSN 0899-7667. doi: 10.1162/neco.1994.6.5.877. URL https://doi.org/10.1162/neco.1994.6.5.877.

Kaitlin Maile, Emmanuel Rachelson, Hervé Luga, and Dennis George Wilson. When, where, and how to add new neurons to ANNs. In *First Conference on Automated Machine Learning (Main Track)*, 2022. URL https://openreview.net/forum?id=SWOg-arIg9.

Abhinav Mehrotra, Alberto Gil C. P. Ramos, Sourav Bhattacharya, Łukasz Dudziak, Ravichander Vipperla, Thomas Chau, Mohamed S Abdelfattah, Samin Ishtiaq, and Nicholas Donald Lane. {NAS}-bench-{asr}: Reproducible neural architecture search for speech recognition. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=CUOAPx9LMaL.

Marc Mezard and Jean-Pierre Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A: Mathematical and General*, 22:2191, 01 1989. doi: 10.1088/0305-4470/22/12/019.

Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017. URL http://arxiv.org/abs/1703.00548.

Byunggook Na, Jisoo Mok, Hyeokjun Choe, and Sungroh Yoon. Accelerating neural architecture search via proxy data. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 2848–2854. International Joint Conferences on Artificial Intelligence Organization, 8 2021. doi: 10.24963/ijcai.2021/392. URL https://doi.org/10.24963/ijcai.2021/392. Main Track.

Yann Ollivier. True asymptotic natural gradient optimization, 2017.

Allan Pinkus. Approximation theory of the mlp model in neural networks. *ACTA NUMERICA*, 8:143–195, 1999.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2018. URL https://openreview.net/forum?id=SkBYYyZRZ.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018. URL http://arxiv.org/abs/1802.01548.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4780–4789, Jul. 2019. doi: 10.1609/aaai.v33i01.33014780. URL https://ojs.aaai.org/index.php/AAAI/article/view/4405.

Patrick Rebeschini. Rademacher Complexity. Examples. https://www.stats.ox.ac.uk/~rebeschi/teaching/AFoL/22/material/lecture03.pdf, 2022.

Robin Ru, Pedro Esperança, and Fabio Maria Carlucci. Neural architecture generator optimization. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12057–12069. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/8c53d30ad023ce50140181f713059ddf-Paper.pdf.

Akito Sakurai. On the vc-dimension of depth four threshold circuits and the complexity of boolean-valued functions. *Theoretical Computer Science*, 137(1):109–127, 1995. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(94)00163-D. URL https://www.sciencedirect.com/science/article/pii/030439759400163D.

Shreyas Saxena and Jakob Verbeek. Convolutional Neural Fabrics. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://papers.nips.cc/paper_files/paper/2016/hash/07811dc6c422334ce36a09ff5cd6fe71-Abstract.html.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Ohad Shamir, Sivan Sabato, and Naftali Tishby. Learning and generalization with the information bottleneck. *Theoretical Computer Science*, 411 (29):2696–2711, 2010. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2010.04.006. URL https://www.sciencedirect.com/science/article/pii/S030439751000201X. Algorithmic Learning Theory (ALT 2008).

Herbert Simon and Allen Newell. Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1):1–10, 1958. URL https://EconPapers.repec.org/RePEc:inm:oropre:v:6:y:1958:i:1:p:1-10.

Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust bayesian neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/a96d3afec184766bfeca7a9f989fc7e7-Paper.pdf.

Kenneth O Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8:131–162, 2007.

Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212, 2009. doi: 10.1162/artl.2009.15.2.15202.

Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth Stanley, and Jeffrey Clune. Generative teaching networks: Accelerating neural architecture search by learning to generate synthetic training data. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9206–9216. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/such20a.html.

Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *2015 IEEE Information Theory Workshop (ITW)*, pages 1–5, 2015. doi: 10.1109/ITW.2015.7133169.

Manon Verbockhaven, Théo Rudkiewicz, Guillaume Charpiat, and Sylvain Chevallier. Growing tiny networks: Spotting expressivity bottlenecks and fixing them optimally. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=hbtG6s6e7r.

Florian Wenzel, Jasper Snoek, Dustin Tran, and Rodolphe Jenatton. Hyperparameter ensembles for robustness and uncertainty quantification. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6514–6527. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/481fbfa59da2581098e841b7afc122f1-Paper.pdf.

Colin White, Willie Neiswanger, and Yash Savani. BANANAS: bayesian optimization with neural architectures for neural architecture search. *CoRR*, abs/1910.11858, 2019. URL http://arxiv.org/abs/1910.11858.

Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers, 2023. URL https://arxiv.org/abs/2301.08727.

Lemeng Wu, Bo Liu, Peter Stone, and Qiang Liu. Firefly Neural Architecture Descent: a General Approach for Growing Neural Networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 22373–22383. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/hash/fdbe012e2e11314b96402b32c0df26b7-Abstract.html.

X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, 1997. doi: 10.1109/72.572107.

Kaicheng Yu, Christian Sciuto, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1loF2NFwr.

Byoung-Tak Zhang, Heinz Muhlenbein, et al. Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex systems*, 7(3):199–220, 1993.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Sy8gdB9xx.

Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *CoRR*, abs/1708.05552, 2017. URL http://arxiv.org/abs/1708.05552.

Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2423–2432, 2018. doi: 10.1109/CVPR.2018.00257.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

# Appendix

## Appendix outline and general remarks

- Appendix B details the theoretical approach of TINY.
- Appendix C proves the propositions of Chapter 3.
- Appendix D details the python module TINYpub and gives additional graphics associated to the result part.

Apart from Appendix B where we need to identify scalar products between each other, we use the trace scalar product and its associated norm as the default scalar product, and we note $\langle\ .\ ,\ .\ \rangle := \langle\ .\ ,\ .\ \rangle_{\mathrm{Tr}}$ and $||\ .\ || := ||\ .\ ||_{\mathrm{Tr}}$. One should remark that $||\ .\ || = ||\ .\ ||_{\mathrm{Tr}} = ||\ .\ ||_2 == ||\ .\ ||_F$ where $||\ .\ ||_2$ is the usual Euclidean norm and $= ||\ .\ ||_F$ is the Frobenius norm.

# A Résumé

Cette thèse propose une stratégie originale d'accroissement d'architecture de réseaux de neurones en un coup, c'est-à-dire qui optimise conjointement l'architecture du réseau et ses paramètres. Pour ce faire, cette approche utilise l'outil d'optimisation classique qu'est la descente de gradient ainsi qu'une nouvelle métrique appelée manque d'expressivité, qui associe à un emplacement de l'architecture du réseau actuel son incapacité à suivre sa dérivée fonctionnelle. Cette métrique est définie comme la projection du gradient fonctionnel du réseau sur son espace tangent et s'estime en un temps comparable à celui d'une « passe aller-retour » dans le réseau. Elle définit une mesure d'expressivité du réseau peu coûteuse en temps de calcul, notamment en comparaison du temps nécessaire au calcul d'autres mesures de complexité comme la complexité de Rademacher, la dimension VC ou encore la complexité de Kolmogorov.

Ce manuscrit s'attache à augmenter l'architecture du réseau de sorte à minimiser cette nouvelle métrique. Bien que le type d'accroissement d'architecture considéré et pris en exemple dans ce manuscrit soit l'ajout de neurones à des couches préexistantes pour un réseau de profondeur fixée, l'ensemble des propositions présentées ici restent vraies et applicables à l'accroissement de réseaux en graphes directs et acycliques, avec ajout de nouvelles couches.

Dans ce manuscrit, il est démontré que l'incapacité, ou manque d'expressivité, peut être résolue au premier ordre de manière optimale par l'ajout de neurones appropriés et que la forme de la solution optimale pour ces nouveaux neurones dépend de la décomposition en valeurs singulières de matrices de covariances qui sont naturellement sauvegardées lors des « passes aller-retour » dans le cadre de l'entraînement du réseau, i.e. les matrices de post-activités et les matrices de poids du réseau. Bien que la décomposition en valeurs singulières soit habituellement considérée comme une opération coûteuse en machine learning, ce calcul est ici négligeable au regard du coût des « passes aller-retour » dans le réseau, car les matrices de covariances sur lesquelles cette décomposition est effectuée sont de petite taille.

Ensuite, il est démontré qu'il existe une équivalence entre la réduction du manque d'expressivité du réseau et la maximisation de la décroissance de la perte globale à l'ordre un. En fait, l'impact de chaque nouveau neurone défini par la min-

imisation du manque d'expressivité réduit non seulement le manque d'expressivité mais également la perte globale, proportionnellement aux valeurs singulières des matrices de covariances évoquées. Cette propriété permet d'établir un classement de ces nouveaux neurones au sein d'une même couche selon l'amplitude de ces valeurs singulières, et ainsi de construire une stratégie d'ajout basée sur la métrique du manque d'expressivité. Il est clair que la résolution du manque d'expressivité fournit des outils et des propriétés pour développer une architecture à partir d'un très petit nombre de neurones, et ce, pendant l'entraînement classique du réseau. Avec une stratégie naïve fondée sur la métrique du manque d'expressivité, il est montré, théoriquement et empiriquement, que la minimisation de cette métrique est un objectif autosuffisant pour atteindre une perte nulle sur l'ensemble d'apprentissage. De plus, cet objectif peut être facilement couplé au processus habituel de descente de gradient. En utilisant le module Python TINYpub, nous avons mis en œuvre une stratégie de recherche naïve et montré que la méthode peut fournir des résultats compétitifs sur des ensembles de données académiques par rapport à d'autres stratégies en un coup appliquées à des architectures de référence.

Bien que la notion de gradient fonctionnel ne soit pas explicitement nommée dans d'autres techniques d'accroissement d'architecture en un coup, ce manuscrit montre que cet objet joue un rôle implicite et important dans d'autres stratégies récentes de recherche d'architectures neuronales en un coup. En particulier, l'expression mathématique des paramètres des nouveaux neurones, définie par la métrique du manque d'expressivité, est comparable à l'initialisation des nouveaux neurones dans deux stratégies similaires, à savoir les méthodes GradMax et NORTH présentées dans les travaux [Evci et al., 2022, Maile et al., 2022]. GradMax initialise ses neurones dans l'objectif de diminuer la perte aussi rapidement que possible, tandis que la méthode NORTH initialise ses neurones de manière aléatoire tout en évitant la redondance au sein de l'architecture actuelle. En fait, la stratégie d'initialisation TINY établit un lien entre ces deux méthodes, en initialisant ses nouveaux neurones de manière à minimiser la perte tout en évitant la redondance avec les neurones existants.

# B Theoretical approach

## B.1 The functional gradient

The functional loss $\mathcal{L}$ is a functional that takes as input a function $f \in \mathcal{F}$ and outputs a real score:

$$\mathcal{L} : \ f \in \mathcal{F} \ \mapsto \ \mathcal{L}(f) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\ell(f(\boldsymbol{x}),\boldsymbol{y})\Big] \ \in \mathbb{R} \ .$$

The function space $\mathcal{F}$ can typically be chosen to be $L_2(\mathbb{R}^p \to \mathbb{R}^d)$, which is a Hilbert space. The directional derivative (or Gateaux derivative, or Fréchet derivative) of functional $\mathcal{L}$ at function $f$ in direction $v$ is defined as:

$$D\mathcal{L}(f)(v) = \lim_{\varepsilon\to 0} \frac{\mathcal{L}(f+\varepsilon v) - \mathcal{L}(f)}{\varepsilon}$$

if it exists. Here $v$ denotes any function in the Hilbert space $\mathcal{F}$ and stands for the direction of change for the function $f$, following an infinitesimal step (of size $\varepsilon$), resulting in a function $f + \varepsilon v$.

If this directional derivative exists in all possible directions $v \in \mathcal{F}$ and moreover is continuous in $v$, then the Riesz representation theorem implies that there exists a unique direction $v^* \in \mathcal{F}$ such that:

$$\forall v \in \mathcal{F}, \ \ D\mathcal{L}(f)(v) = \langle v^*, v\rangle \ .$$

This direction $v^*$ is named the *gradient* of the functional $\mathcal{L}$ at function $f$ and is denoted by $\nabla_f \mathcal{L}(f)$.

Note that while the inner product $\langle\cdot,\cdot\rangle$ considered is usually the $L_2$ one, it is possible to consider other ones, such as Sobolev ones (e.g., $H^1$). The gradient $\nabla_f \mathcal{L}(F)$ depends on the chosen inner product and should consequently rather be denoted by $\nabla_f^{L_2}\mathcal{L}(f)$, for instance.

Note that continuous functions from $\mathbb{R}^p$ to $\mathbb{R}^d$, as well as $C^\infty$ functions, are dense in $L_2(\mathbb{R}^p \to \mathbb{R}^d)$.

Let us now study properties specific to our loss design: $\mathcal{L}(f) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\ell(f(\boldsymbol{x}),\boldsymbol{y})\Big]$. Assuming sufficient $\ell$-loss differentiability and integrability, it follows that, for any function update direction $v \in \mathcal{F}$ and infinitesimal step size $\varepsilon \in \mathbb{R}$:

$$\mathcal{L}(f+\varepsilon v) - \mathcal{L}(f) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\ell(f(\boldsymbol{x})+\varepsilon v(\boldsymbol{x}),\boldsymbol{y}) - \ell(f(\boldsymbol{x}),\boldsymbol{y})\Big]$$

$$= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\nabla_{\boldsymbol{u}}\ell(\boldsymbol{u},\boldsymbol{y})\big|_{\boldsymbol{u}=f(\boldsymbol{x})} \cdot \varepsilon v(\boldsymbol{x}) + O(\varepsilon^2\|v(\boldsymbol{x})\|^2)\Big]$$

using the usual gradient of function $\ell$ at point $(\boldsymbol{u} = f(\boldsymbol{x}), \boldsymbol{y})$ w.r.t. its first argument $\boldsymbol{u}$, with the standard Euclidean dot product $\cdot$ in $\mathbb{R}^p$. Then the directional derivative is:

$$D\mathcal{L}(f)(v) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\nabla_{\boldsymbol{u}}\ell(\boldsymbol{u},\boldsymbol{y})\big|_{\boldsymbol{u}=f(\boldsymbol{x})}\cdot v(\boldsymbol{x})\Big] = \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\Big[\mathbb{E}_{\boldsymbol{y}\sim\mathcal{D}|\boldsymbol{x}}\Big[\nabla_{\boldsymbol{u}}\ell(\boldsymbol{u},\boldsymbol{y})\big|_{\boldsymbol{u}=f(\boldsymbol{x})}\Big]\cdot v(\boldsymbol{x})\Big]$$

and thus the functional gradient for the inner product $\langle v, v'\rangle_{\mathbb{E}} := \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\Big[v(\boldsymbol{x})\cdot v'(\boldsymbol{x})\Big]$ is the function:

$$\nabla^{\mathbb{E}}_f\mathcal{L}(f) : \boldsymbol{x} \mapsto \mathbb{E}_{\boldsymbol{y}\sim\mathcal{D}|\boldsymbol{x}}\Big[\nabla_{\boldsymbol{u}}\ell(\boldsymbol{u},\boldsymbol{y})\big|_{\boldsymbol{u}=f(\boldsymbol{x})}\Big]$$

which simplifies into:

$$\nabla^{\mathbb{E}}_f\mathcal{L}(f) : \boldsymbol{x} \mapsto \nabla_{\boldsymbol{u}}\ell(\boldsymbol{u},\boldsymbol{y}(\boldsymbol{x}))\big|_{\boldsymbol{u}=f(\boldsymbol{x})}$$

if there is no ambiguity in the dataset, i.e. if for each $\boldsymbol{x}$ there is a unique $\boldsymbol{y}(\boldsymbol{x})$.

Note that by considering the $L_2(\mathbb{R}^p \to \mathbb{R}^d)$ inner product $\int v \cdot v'$ instead, one would respectively get:

$$\nabla^{L_2}_f\mathcal{L}(f) : \boldsymbol{x} \mapsto p_{\mathcal{D}}(\boldsymbol{x})\mathbb{E}_{\boldsymbol{y}\sim\mathcal{D}|\boldsymbol{x}}\Big[\nabla_{\boldsymbol{u}}\ell(\boldsymbol{u},\boldsymbol{y})\big|_{\boldsymbol{u}=f(\boldsymbol{x})}\Big]$$

and

$$\nabla^{L_2}_f\mathcal{L}(f) : \boldsymbol{x} \mapsto p_{\mathcal{D}}(\boldsymbol{x})\,\nabla_{\boldsymbol{u}}\ell(\boldsymbol{u},\boldsymbol{y}(\boldsymbol{x}))\big|_{\boldsymbol{u}=f(\boldsymbol{x})}$$

instead, where $p_{\mathcal{D}}(\boldsymbol{x})$ is the density of the dataset distribution at point $\boldsymbol{x}$. In practice, one estimates such gradients using a mini batch of samples $(\boldsymbol{x},\boldsymbol{y})$, obtained by picking uniformly at random within a finite dataset, and thus the formulas for the two inner products coincide (up to a constant factor).

## B.2 Differentiation under the integral sign

Let $X$ be an open subset of $\mathbb{R}$, and $\Omega$ be a measure space. Suppose $f : X \times \Omega \to \mathbb{R}$ satisfies the following conditions:

- $f(x,\omega)$ is a Lebesgue-integrable function of $\omega$ for each $x \in X$.

- For almost all $\omega \in \Omega$ , the partial derivative $\frac{\partial}{\partial x} f$ of $f$ according to $x$ exists for all $x \in X$.
- There is an integrable function $\theta : \Omega \to \mathbb{R}$ such that $|\frac{\partial}{\partial x}(x, \omega)| \leq \theta(\omega)$ for all $x \in X$ and almost every $\omega \in \Omega$.

Then, for all $x \in X$,

$$\frac{\partial}{\partial x} \int_{\Omega} f(x, \omega)\, d\omega = \int_{\Omega} \frac{\partial}{\partial x} f(x, \omega)\, d\omega \tag{B.1}$$

See proof and details :Le Gall [2006].

## B.3 Gradients and proximal point of view

Gradients with respect to standard variables such as vectors are defined the same way as functional gradients above: given a sufficiently smooth loss $\widetilde{\mathcal{L}} : \theta \in \Theta_{\mathcal{A}} \mapsto \widetilde{\mathcal{L}}(\theta) = \mathcal{L}(f_\theta) \in \mathbb{R}$, and an inner product $\cdot$ in the space $\Theta_{\mathcal{A}}$ of parameters $\theta$, the gradient $\nabla_\theta \widetilde{\mathcal{L}}(\theta)$ is the unique vector $\boldsymbol{\tau} \in \Theta_{\mathcal{A}}$ such that:

$$\forall \delta\theta \in \Theta_{\mathcal{A}}, \quad \boldsymbol{\tau} \cdot \delta\theta = D_\theta \widetilde{\mathcal{L}}(\theta)(\delta\theta)$$

where $D_\theta \widetilde{\mathcal{L}}(\theta)(\delta\theta)$ is the directional derivative of $\widetilde{\mathcal{L}}$ at point $\theta$ in the direction $\delta\theta$, defined as in the previous section. This gradient depends on the inner product chosen, which can be highlighted by the following property. The opposite $-\nabla_\theta \widetilde{\mathcal{L}}(\theta)$ of the gradient is the unique solution of the problem:

$$\underset{\delta\theta \in \Theta_{\mathcal{A}}}{\arg\min} \left\{ D_\theta \widetilde{\mathcal{L}}(\theta)(\delta\theta) + \frac{1}{2} \|\delta\theta\|_P^2 \right\}$$

where $\| \quad \|_P$ is the norm associated to the chosen inner product. Changing the inner product changes the way candidate directions $\delta\theta$ are penalized, leading to different gradients. This proximal formulation can be obtained as follows. For any $\delta\theta$, its distance to the gradient descent direction is:

$$\left\| \delta\theta - \left( -\nabla_\theta \widetilde{\mathcal{L}}(\theta) \right) \right\|^2 = \|\delta\theta\|^2 + 2\, \delta\theta \cdot \nabla_\theta \widetilde{\mathcal{L}}(\theta) + \left\| \nabla_\theta \widetilde{\mathcal{L}}(\theta) \right\|^2$$

$$= 2 \left( \frac{1}{2} \|\delta\theta\|^2 + D_\theta \widetilde{\mathcal{L}}(\theta)(\delta\theta) \right) + K$$

where $K$ does not depend on $\delta\theta$. For the above to hold, the inner product used has to be the one from which the norm is derived. By minimizing this expression with respect to $\delta\theta$, one obtains the desired property.

In this case of study, for the norm over the space $\Theta_{\mathcal{A}}$ of parameter variations, a norm in the space of associated functional variations is considered, i.e.:

$$\|\delta\theta\|_P := \left\| \frac{\partial f_\theta}{\partial \theta} \delta\theta \right\|$$

which makes more sense from a physical point of view, as it is more intrinsic to the task to solve and depends as little as possible on the parameterization (i.e. on the architecture chosen). This results in a functional move that is the projection of the functional one to the set of possible moves given the architecture. On the opposite, the standard gradient (using Euclidean parameter norm $\|\delta\theta\|$ in parameter space) yields a functional move obtained not only by projecting the functional gradient but also by multiplying it by a matrix $\frac{\partial f_\theta}{\partial \theta}\frac{\partial f_\theta}{\partial \theta}^T$ which can be seen as a strong architecture bias over-optimization directions.

It is supposed here, that the loss $\mathcal{L}$ to be minimized is the real loss that the user wants to optimize, possibly including regularizers to avoid overfitting, and since the architecture is evolving during training, possibly to architectures far from usual manual design and never tested before, one cannot assume architecture bias to be desirable. The objective is to get rid of it in order to follow the functional gradient descent as closely as possible.

Searching for

$$\boldsymbol{v}^* = \underset{\boldsymbol{v}\in\mathcal{T}_{\mathcal{A}}}{\arg\min} \|\boldsymbol{v} - \boldsymbol{v}_{\text{goal}}\|^2 = \underset{\boldsymbol{v}\in\mathcal{T}_{\mathcal{A}}}{\arg\min} \left\{ D\mathcal{L}(f)(\boldsymbol{v}) + \frac{1}{2}\|\boldsymbol{v}\|^2 \right\} \tag{B.2}$$

or equivalently for:

$$\delta\theta^* = \underset{\delta\theta\in\Theta_{\mathcal{A}}}{\arg\min} \left\| \frac{\partial f_\theta}{\partial \theta} \delta\theta - \boldsymbol{v}_{\text{goal}} \right\|^2 \tag{B.3}$$

$$= \underset{\delta\theta\in\Theta_{\mathcal{A}}}{\arg\min} \left\{ D_\theta \mathcal{L}(f_\theta)(\delta\theta) + \frac{1}{2} \left\| \frac{\partial f_\theta}{\partial \theta} \delta\theta \right\|^2 \right\} =: -\nabla_\theta^{\mathcal{T}_{\mathcal{A}}} \mathcal{L}(f_\theta) \tag{B.4}$$

then appears as a natural goal.

## B.4 Problem formulation and choice of pre-activities

There are several ways to design the problem of adding neurons, which is discussed now, in order to explain TINY choice of the pre-activities to express expressivity bottlenecks.

Suppose one wishes to add $K$ neurons $\theta_{\leftrightarrow}^K := (\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)_{k=1}^K$ to layer $l-1$, which impacts the activities $\boldsymbol{a}_l$ at the next layer, in order to improve its expressivity. These neurons could be chosen to have only null weights, or null input weights $\boldsymbol{\alpha}_k$ and non-null output weights $\boldsymbol{\omega}_k$, or the opposite, or both non-null weights.

Searching for the best neurons to add for each of these cases will produce different optimization problems.

Let us remind first that adding such $K$ neurons with weights $\theta_{\leftrightarrow}^K := (\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)_{k=1}^K$ changes the activities $\boldsymbol{a}_l$ of the (next) layer by

$$\delta \boldsymbol{a}_l \quad = \quad \sum_{k=1}^{K} \boldsymbol{\omega}_k \, \sigma(\boldsymbol{\alpha}_k^T \boldsymbol{b}_{l-2}(\boldsymbol{x})) \tag{B.5}$$

**Small weights approximation** Under the hypothesis of small input weights $\alpha_k$, the activity variation B.5 can be approximated by:

$$\sigma'(0) \, \sum_{k=1}^{K} \boldsymbol{\omega}_k \boldsymbol{\alpha}_k^T \boldsymbol{b}_{l-2}(\boldsymbol{x}) \tag{B.6}$$

at first order in $\|\boldsymbol{\alpha}_k\|$. The constant $\sigma'(0)$ is dropped in the sequel.

This quantity is linear both in $\alpha_k$ and $\boldsymbol{\omega}_k$, therefore the first-order parameter-induced activity variations are easy to compute:

$$\boldsymbol{v}^l(\boldsymbol{x}, (\boldsymbol{\alpha}_k)_{k=1}^K) = \frac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial(\ (\boldsymbol{\alpha}_k)_{k=1}^K\ )_{|(\boldsymbol{\alpha}_k)_{k=1}^K=0}} (\boldsymbol{\alpha}_k)_{k=1}^K = \sum_{k=1}^{K} \boldsymbol{\omega}_k \boldsymbol{b}_{l-2}(\boldsymbol{x})^T \boldsymbol{\alpha}_k$$

$$\boldsymbol{v}^l(\boldsymbol{x}, (\boldsymbol{\omega}_k)_{k=1}^K) = \frac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial(\ (\boldsymbol{\omega}_k)_{k=1}^K\ )_{|(\boldsymbol{\omega}_k)_{k=1}^K=0}} (\boldsymbol{\omega}_k)_{k=1}^K = \sum_{k=1}^{K} \boldsymbol{\omega}_k \boldsymbol{b}_{l-2}(\boldsymbol{x})^T \boldsymbol{\alpha}_k$$

so with a slight abuse of notation, it follows that :

$$\boldsymbol{v}^l(\boldsymbol{x}, \theta_{\leftrightarrow}^K) = \sum_{k=1}^{K} \boldsymbol{\omega}_k \boldsymbol{\alpha}_k^T \boldsymbol{b}_{l-2}(\boldsymbol{x})$$

Note also that technically the quantity above is first-order in $\boldsymbol{\alpha}_k$ and in $\boldsymbol{\omega}_k$ but second-order in the joint variable $\theta_{\leftrightarrow}^K = (\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)$.

**Adding neurons with 0 weights (both input and output weights).** In that case, one increases the number of neurons in the layer, but without changing the function (since only null quantities are added) and also without changing the gradient with respect to the parameters, thus not improving expressivity. Indeed, the added quantity (Eq. B.5) involves $0 \times 0$ multiplications, and consequently the derivative $\left.\frac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial \theta_{\leftrightarrow}^K}\right|_{\theta_{\leftrightarrow}^K=0}$ w.r.t. these new parameters, that is, $\boldsymbol{b}_{l-2}(\boldsymbol{x})^T \boldsymbol{\alpha}_k$ w.r.t. $\boldsymbol{\omega}_k$ and $\boldsymbol{\omega}_k \, \boldsymbol{b}_{l-2}(\boldsymbol{x})^T$ w.r.t. $\boldsymbol{a}_k$ is 0, as both $\boldsymbol{a}_k$ and $\boldsymbol{\omega}_k$ are 0.

**Adding neurons with non-0 input weights and 0 output weights or the opposite.** In these cases, the addition of neurons will not change the function (because of multiplications by 0), but just the gradient. One of the 2 gradients (w.r.t. $\boldsymbol{a}_k$ or w.r.t $\boldsymbol{\omega}_k$) will be non-0, as the variable that is 0 has non-0 derivatives.

The question is then how to pick the best non-0 variable, ($\boldsymbol{a}_k$ or $\boldsymbol{\omega}_k$) such that the added gradient will be the most useful. The problem can then be formulated similarly to what is done in the paper.

**Adding neurons with small yet non-0 weights.** In this case, both the function and its gradient will change when adding the neurons. Fortunately, Proposition 3.2.3 states that the best neurons to add in terms of expressivity (to get the gradient closer to the variation desired by the backpropagation) are also the best neurons to add to decrease the loss, i.e. the function change they will imply goes into the right direction.

For each family $(\boldsymbol{\omega}_k)_{k=1}^K$, the tangent space in $\boldsymbol{a}_l$ restricted to the family $(\boldsymbol{\alpha}_k)_{k=1}^K$, ie $\mathcal{T}_{\mathcal{A}}^{\boldsymbol{a}_l} := \{\frac{\partial \boldsymbol{a}_l}{\partial (\boldsymbol{\alpha}_k)_{k=1}^K}_{|(\boldsymbol{\alpha}_k)_{k=1}^K=0}(.)(\boldsymbol{\alpha}_k)_{k=1}^K | (\boldsymbol{\alpha}_k)_{k=1}^K \in \left(\mathbb{R}^{|\boldsymbol{b}_{l-2}(\boldsymbol{x})|}\right)^K\}$ varies with the family $(\boldsymbol{\omega}_k)_{k=1}^K$, ie $\mathcal{T}_{\mathcal{A}}^{a_l} := \mathcal{T}_{\mathcal{A}}^{a_l}((\boldsymbol{\omega}_k)_{k=1}^K)$. Optimizing w.r.t. the $\boldsymbol{\omega}_k$ is equivalent to searching for the best tangent space for the $\boldsymbol{\alpha}_k$, while symmetrically optimizing w.r.t. the $\boldsymbol{\alpha}_k$ is equivalent to finding the best projection on the tangent space defined by the $\boldsymbol{\omega}_k$.

**Pre-activities vs. post-activities.** The space of pre-activities $\boldsymbol{a}_l$ is a natural space for this framework, as they are formed with linear operations, and first-order variation quantities can be computed. Considering the space of post-activities $\boldsymbol{b}_l = \sigma(\boldsymbol{a}_l)$ is also possible, though computing variations will be more complex. Indeed, without first-order approximation, the obtained problem is not manageable because the non-linear activation function $\sigma$ added in front of all quantities (while in the case of pre-activations, quantity B.5 is linear in $\boldsymbol{\omega}_k$ and thus does not require an approximation in $\boldsymbol{\omega}_k$, which allow considering large $\boldsymbol{\omega}_k$), and, with first-order approximation, it would add the derivative of the activation function, taken at various locations $\sigma'(\boldsymbol{a}_l)$ (while in the previous case, the derivatives of the activation function were always taken at 0).

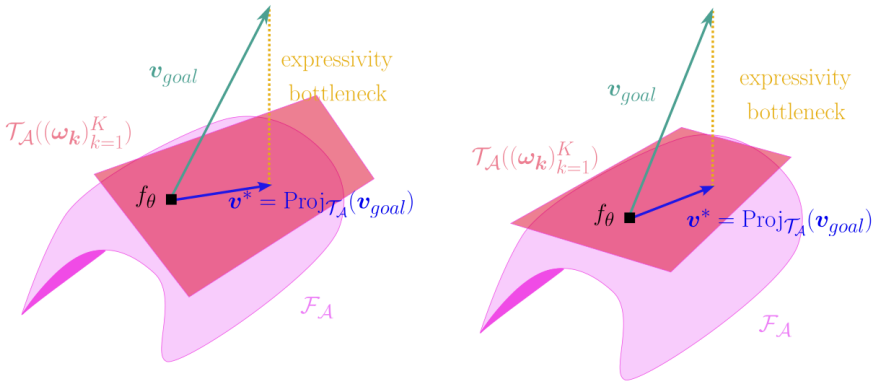



Figure B.1: Changing the tangent space with different values of $(\boldsymbol{\omega}_k)_{k=1}^K$.

## B.5 About equivalence of quadratic problems

Problems 3.21 and 3.20 are generally not equivalent but might be very close, depending on layer sizes and number of samples. The difference between the two problems is that, in one case, one minimizes the quadratic quantity:

$$\left\| \boldsymbol{V}^l(\theta_{\leftrightarrow}^K) + \boldsymbol{V}^l(\delta \mathbf{W}_l) - \boldsymbol{V}_{\text{goal}}{}^l \right\|^2$$

w.r.t. $\delta \mathbf{W}_l$ and $\theta_{\leftrightarrow}^K$ **jointly**, while in the other case the problem is first minimized w.r.t. $\delta \mathbf{W}_l$ and then w.r.t. $\theta_{\leftrightarrow}^K$. The latter process, being greedy, might thus provide a solution that is not as optimal as the joint optimization.

This two-step process is chosen as it intuitively relates to the spirit of improving upon a standard gradient descent: the objective is to add neurons that complement what the other ones have already done. This choice is debatable, and one could solve the joint problem using the same techniques.

The topic of this section is to check how close the two problems are. To study this further, note that $\boldsymbol{V}^l(\delta \mathbf{W}_l) = \delta \boldsymbol{W}_l \boldsymbol{B}_{l-1}$ while $\boldsymbol{V}^l(\theta_{\leftrightarrow}^K) = \sum_{k=1}^K \boldsymbol{\omega}_k \boldsymbol{B}_{l-2}^T \boldsymbol{\alpha}_k$. The rank of $\boldsymbol{B}_{l-1}$ is $\min(n_S, n_{l-1})$ where $n_S$ is the number of samples and $n_{l-1}$ the number of neurons (post-activities) in layer $l-1$, while the rank of $\boldsymbol{B}_{l-2}$ is $\min(n_S, n_{l-2})$ where $n_{l-2}$ is the number of neurons (post-activities) in layer $l-2$. Note also that the number of degrees of freedom in the optimization variables $\delta \boldsymbol{W}_l$ and $\theta_{\leftrightarrow}^K = (\boldsymbol{\omega}_k, \boldsymbol{\alpha}_k)$ is much larger than these ranks.

**Small sample case.** If the number $n_S$ of samples is lower than the number of neurons $n_{l-1}$ and $n_{l-2}$ (which is potentially problematic, see Section D.4), then it is possible to find suitable variables $\delta \boldsymbol{W}_l$ and $\theta_{\leftrightarrow}^K$ to form any desired $\boldsymbol{V}^l(\delta \mathbf{W}_l)$ and $\boldsymbol{V}^l(\theta_{\leftrightarrow}^K)$. In particular, if $n_S \leqslant n_{l-1} \leqslant n_{l-2}$, one can choose $\boldsymbol{V}^l(\theta_{\leftrightarrow}^K)$ to be

$\boldsymbol{V}_{\text{goal}}{}^{l} - \boldsymbol{V}^{l}(\delta \mathbf{W}_l)$ and thus cancel any effect due to the greedy process in two steps. The two problems are then equivalent.

**Large sample case.** On the opposite, if the number of samples is very large (compared to the number of neurons $n_{l-1}$ and $n_{l-2}$), then the lines of matrices $\boldsymbol{B}_{l-1}$ and $\boldsymbol{B}_{l-2}$ become asymptotically uncorrelated, under the assumption of their independence (which is debatable, depending on the type of layers and activation functions). Thus the optimization directions available to $\boldsymbol{V}^{l}(\delta \mathbf{W}_l)$ and $\boldsymbol{V}^{l}(\theta^{K}_{\leftrightarrow})$ become orthogonal, and proceeding greedily does not affect the result, the two problems are asymptotically equivalent.

In the general case, matrices $\boldsymbol{B}_{l-1}$ and $\boldsymbol{B}_{l-2}$ are not independent, though not fully correlated, and the number of samples (in the minibatch) is typically larger than the number of neurons; the problems are then different.

Note that technically the ranks could be lower in the improbable case where some neurons are perfectly redundant, or, e.g., if some samples yield exactly the same activities.

# C proofs of 3

## C.1 proof of Proposition 3.2.2

Denoting by $\delta\mathbf{W}_l{}^+$ the generalized (pseudo-)inverse of $\delta\mathbf{W}_l$, it follows that:

$$\delta \boldsymbol{W}_l^* = \frac{1}{n}\boldsymbol{V}_{\text{goal}}{}^l \boldsymbol{B}_{l-1}^T \left(\frac{1}{n}\boldsymbol{B}_{l-1}\boldsymbol{B}_{l-1}^T\right)^+ \text{ and } \boldsymbol{V}_0^l = \frac{1}{n}\boldsymbol{V}_{\text{goal}}{}^l \boldsymbol{B}_{l-1}^T \left(\frac{1}{n}\boldsymbol{B}_{l-1}\boldsymbol{B}_{l-1}^T\right)^+ \boldsymbol{B}_{l-1}$$

*Proof:* **Fully connected layers**
Consider the function

$$g(\delta \boldsymbol{W}) := \left\| \boldsymbol{V}_{\text{goal}}{}^l - \delta \boldsymbol{W}\boldsymbol{B}_{l-1} \right\|^2 \tag{C.1}$$

then:

$$\begin{aligned} g(\delta \boldsymbol{W} + \boldsymbol{H}) &= ||\boldsymbol{V}_{\text{goal}}{}^l - \delta \boldsymbol{W}\boldsymbol{B}_{l-1} - \boldsymbol{H}\boldsymbol{B}_{l-1}||^2 && \text{(C.2)} \\ &= g(\delta \boldsymbol{W}) - 2\left\langle \boldsymbol{V}_{\text{goal}}{}^l - \delta \boldsymbol{W}\boldsymbol{B}_{l-1}, \boldsymbol{H}\boldsymbol{B}_{l-1}\right\rangle + o(||\boldsymbol{H}||) && \text{(C.3)} \\ &= g(\delta \boldsymbol{W}) - 2\left\langle \left(\boldsymbol{V}_{\text{goal}}{}^l - \delta \boldsymbol{W}\boldsymbol{B}_{l-1}\right)\boldsymbol{B}_{l-1}^T, \boldsymbol{H}\right\rangle + o(||\boldsymbol{H}||) && \text{(C.4)} \end{aligned}$$

By identification $\nabla_{\delta \boldsymbol{W}} g(\delta \boldsymbol{W}) = -2\left(\boldsymbol{V}_{\text{goal}}{}^l - \delta \boldsymbol{W}\boldsymbol{B}_{l-1}\right)\boldsymbol{B}_{l-1}^T$, and thus

$$\nabla_{\delta \boldsymbol{W}} g(\delta \boldsymbol{W}) = 0 \implies \boldsymbol{V}_{\text{goal}}{}^l \boldsymbol{B}_{l-1}^T = \delta \boldsymbol{W}\boldsymbol{B}_{l-1}\boldsymbol{B}_{l-1}^T.$$

Using that $g$ is convex and the definition of the generalized inverse, then:

$$\delta \boldsymbol{W}_l^* = \frac{1}{n}\boldsymbol{V}_{\text{goal}}{}^l \boldsymbol{B}_{l-1}^T \left(\frac{1}{n}\boldsymbol{B}_{l-1}\boldsymbol{B}_{l-1}^T\right)^+.$$

**Convolutional layers**
For convolutional layers, the minimization problem is (the index $l-1$ has been dropped for readability):

$$\arg\min_{\delta \boldsymbol{W}} \|\boldsymbol{V}_{\text{goal}} - \mathbf{Conv}_{\delta \boldsymbol{W}}(\boldsymbol{B})\| \tag{C.5}$$

This can be converted in a linear regression by transforming the convolution in a matrix multiplication. Let $H$ and $W$ the height and width of an internal representation of $\boldsymbol{B}$, $C$ the number of channels and $(d, d)$ the size of the kernel. In square brackets is indicated the index position of those quantity with respect to the reference layer, which is here $l-1$. Doing so, $C[+1]$ is the number of channels at layer $l-1+1$. For the figure fig. 5.1, one can remark that $HW = P$ and $S = d^2$. Using those notations, the matrices $\delta \boldsymbol{W}$ and $\boldsymbol{V}_{\text{goal}}$ are reshaped and permuted :

- $\delta \boldsymbol{W} \in (C[+1], C, d[+1], d[+1])$ is transformed in $\delta \boldsymbol{W}_F \in (C[+1], Cd[+1]d[+1])$
- $\boldsymbol{V}_{\text{goal}} \in (n, C[+1], H[+1], W[+1])$ is transformed in $\boldsymbol{V}_{\text{goal}_F} \in (nH[+1]W[+1], C[+1])$

Definition C.1.1 ($\boldsymbol{B}^c$). Let $\boldsymbol{B}^c \in (n, Cd[+1]d[+1], H[+1]W[+1])$ and its reshaped version $\boldsymbol{B}^c_F \in (nH[+1]W[+1], Cd[+1]d[+1])$ satisfying $\boldsymbol{B}^c_F \delta \boldsymbol{W}_F^T \in (nH[+1]W[+1], C[+1])$ and that is a reshaped version of $\mathbf{Conv}_{\delta \boldsymbol{W}}(\boldsymbol{B})$. ($\boldsymbol{B}^c$ can be easily computed using `torch.Tensor.Unfold`.)

Using such definition, eq. (C.5) becomes:

$$\arg\min_{\delta \boldsymbol{W}_F} \left\|\boldsymbol{V}_{\text{goal}_F} - \boldsymbol{B}^c_F \delta \boldsymbol{W}_F^T\right\| \tag{C.6}$$

Using the same reasoning that for fully connected layers an optimal solution is then :

$$(\delta \boldsymbol{W}_F^*)^T = \frac{1}{n}\left((\boldsymbol{B}^c_F)^T \boldsymbol{B}^c_F\right)^+ \frac{1}{n}(\boldsymbol{B}^c_F)^T \boldsymbol{V}_{\text{goal}_F} \tag{C.7}$$

■

## C.2 Proof of proposition 3.2.3

We define the matrices $\boldsymbol{N} := \frac{1}{n}\boldsymbol{B}_{l-2}\left(\boldsymbol{V}_{\text{goal}^l_{proj}}\right)^T$ and $\boldsymbol{S} := \frac{1}{n}\boldsymbol{B}_{l-2}\boldsymbol{B}_{l-2}^T$. Let us denote its SVD by $\boldsymbol{S} = \boldsymbol{O}\Sigma\boldsymbol{O}^T$, and note $\boldsymbol{S}^{-\frac{1}{2}} := \boldsymbol{O}\sqrt{\Sigma}^{-1}\boldsymbol{O}^T$ and consider the SVD of the matrix $\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N} = \sum_{k=1}^{R} \lambda_k \boldsymbol{u}_k \boldsymbol{v}_k^T$ with $\lambda_1 \geq ... \geq \lambda_R \geq 0$, where $R$ is the rank of the matrix $\boldsymbol{N}$. Then:

**Proposition C.2.1 (3.2.3).** *The solution of equation 3.21 can be written as:*

- *optimal number of neurons:* $K^* = R$
- *their optimal weights:* $(\boldsymbol{\alpha}_k^*, \boldsymbol{\omega}_k^*) = (\sqrt{\lambda_k} S^{-\frac{1}{2}} \boldsymbol{u}_k, \sqrt{\lambda_k}\boldsymbol{v}_k)$ *for* $k = 1, ..., R$.

*Moreover for any number of neurons $K \leqslant R$, and associated scaled weights $\theta_{\leftrightarrow}^{K,*}$, the expressivity gain and the first order in $\eta$ of the loss improvement due to the addition of these $K$ neurons are equal and can be quantified very simply as a function of the eigenvalues $\lambda_k$:*

$$\Psi^l_{\theta \oplus \theta^{K,*}_{\leftrightarrow}} = \Psi^l_\theta - \sum_{k=1}^{K} \lambda_k^2$$

*for fully-connected layers, with an inequality instead ($\leq$) for convolutional layers.*

---

To facilitate reading the layer index of each quantity has been removed, i.e. $\boldsymbol{B} := \boldsymbol{B}_{l-2}, \boldsymbol{V}^l(\boldsymbol{A}, \boldsymbol{\Omega}) := \boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega})$ and $\boldsymbol{V}_{\text{goal}^l_{proj}} := \boldsymbol{V}_{\text{goal}_{proj}}$. Then, we have fixed $n$ and $\boldsymbol{x}_1, ...., \boldsymbol{x}_n$ on which the expressivity bottleneck formula is solved.

To solve this problem, consider the input of the incoming connections $\boldsymbol{B}$ and the desired change in the output of the outgoing connections $\boldsymbol{V}_{\text{goal}_{proj}}$. Hence if $L(\boldsymbol{A})$ and $L(\boldsymbol{\Omega})$ are the additional connections of the expanded representation and $\sigma$ the non linearity, the following proxy problem is optimized:

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n} \left\| (L(\boldsymbol{\Omega}) \circ \sigma \circ L(\boldsymbol{A}))(\boldsymbol{B}) - \boldsymbol{V}_{\text{goal}_{proj}} \right\| \tag{C.8}$$

This problem is solved at first order by linearizing the non linearity $\sigma$. Let $\mathbf{Lin}_{(a,b)}(W)$ the fully connected layer with input size $a$, output size $b$ and weight matrix $\boldsymbol{W}$. Let $C[+1]$ and $C[-1]$ the layer width at layer $l+1$ and $l-1$ with the convention that C[0] is the dimension of the input $\boldsymbol{x}$. With those notations, for fully connected layers, it follows that for the additions of $K$ neurons :

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n} \left\| \mathbf{Lin}_{(C[+1],K)}(\boldsymbol{\Omega})(\mathbf{Lin}_{(K,C[-1])}(\boldsymbol{A})(\boldsymbol{B})) - \boldsymbol{V}_{\text{goal}_{proj}} \right\| \tag{C.9}$$

With the same notations, for convolutional layers, it follows that for the additions of $K$ intermediate channels:

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n} \left\| \mathbf{Conv}_{(C[+1],K)}(\boldsymbol{\Omega})(\mathbf{Conv}_{(K,C[-1])}(\boldsymbol{A})(\boldsymbol{B})) - \boldsymbol{V}_{\text{goal}_{proj}} \right\| \tag{C.10}$$

Let $\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega})$ the result of $\boldsymbol{B}$ after applying the layers parametrized by $\boldsymbol{A}$ and $\boldsymbol{\Omega}$, in both cases the minimization problem is:

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n} \left\| \boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}) - \boldsymbol{V}_{\text{goal}_{proj}} \right\| \tag{C.11}$$

First, for linear layers, we will show that solving C.9 is equivalent to solve :

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \left\| \mathbf{S}^{\frac{1}{2}} \boldsymbol{A}\boldsymbol{\Omega}^T - \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\| \tag{C.12}$$

where $\boldsymbol{S}$ depends of $\boldsymbol{B}$ and $\boldsymbol{N}$ of $\boldsymbol{B}$ and $\boldsymbol{V}_{\text{goal}_{proj}}$.

Then, for convolutional layers, we will provide the exact solution and also, we will upper bound the solution of C.10 as :

$$\frac{1}{n} \left\| \boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}) - \boldsymbol{V}_{\text{goal}_{proj}} \right\|^2 \leq \left\| \mathbf{S}^{\frac{1}{2}} \boldsymbol{A}_F \boldsymbol{\Omega}_F - \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 + \frac{1}{n} \left\| \boldsymbol{V}_{\text{goal}_{\text{proj}}} \right\|^2 \tag{C.13}$$

where, as for linear layers, $\boldsymbol{S}$ depends of $\boldsymbol{B}$ and $\boldsymbol{N}$ of $\boldsymbol{B}$ and $\boldsymbol{V}_{\text{goal}_{proj}}$, and, $\boldsymbol{A}_F$ and $\boldsymbol{\Omega}_F$ are reshaped versions of $\boldsymbol{A}$ and $\boldsymbol{\Omega}$.

### C.2.1 Fully connected layers

For a fully connected layer, then :

$$\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}) = \mathbf{Lin}_{(C[+1],K)}(\boldsymbol{\Omega})(\mathbf{Lin}_{(K,C[-1])}(\boldsymbol{A})(\boldsymbol{B})) = \boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B} \tag{C.14}$$

We now use the following lemma that we shall prove in 6.

**Lemma C.2.2.** *Let* $\boldsymbol{Y} \in \mathbb{R}(t, n), \boldsymbol{X} \in \mathbb{R}(s, n), \boldsymbol{C} \in \mathbb{R}(t, s)$
*We define:*

$$\boldsymbol{S} := \frac{1}{n} \boldsymbol{X}\boldsymbol{X}^T \in \mathbb{R}(s, s) \tag{C.15}$$

$$\boldsymbol{N} := \frac{1}{n} \boldsymbol{X}\boldsymbol{Y}^T \in \mathbb{R}(s, t) \tag{C.16}$$

$$\frac{1}{n} \|\boldsymbol{C}\boldsymbol{X} - \boldsymbol{Y}\|^2 = \left\| \boldsymbol{C}\mathbf{S}^{\frac{1}{2}} - \boldsymbol{N}^T\mathbf{S}^{-\frac{1}{2}} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 + \frac{1}{n} \|\boldsymbol{Y}\|^2 \tag{C.17}$$

Hence, using theorem C.2.2 with $\boldsymbol{C} \leftarrow \boldsymbol{\Omega}\boldsymbol{A}^T$, $\boldsymbol{Y} \leftarrow \boldsymbol{V}_{\text{goal}_{proj}}$ and $\boldsymbol{B} \leftarrow \boldsymbol{X}$, then:

$$\begin{gathered} \frac{1}{n} \left\| \mathbf{Lin}_{(C[+1],K)}(\boldsymbol{\Omega})(\mathbf{Lin}_{(K,C[-1])}(\boldsymbol{A})(\boldsymbol{B})) - \boldsymbol{V}_{\text{goal}_{proj}} \right\|^2 \\ = \\ \frac{1}{n} \left\| \boldsymbol{\Omega}\boldsymbol{A}^T\boldsymbol{B} - \boldsymbol{V}_{\text{goal}_{proj}} \right\|^2 \\ = \\ \left\| \boldsymbol{\Omega}\boldsymbol{A}^T\mathbf{S}^{\frac{1}{2}} - \boldsymbol{N}^T\mathbf{S}^{-\frac{1}{2}} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 + \frac{1}{n} \left\| \boldsymbol{V}_{\text{goal}_{proj}} \right\|^2 \end{gathered} \tag{C.18}$$

With:

$$\boldsymbol{S} := \frac{1}{n}\boldsymbol{B}\boldsymbol{B}^T \in \mathbb{R}(C[-1], C[-1]) \tag{C.19}$$

$$\boldsymbol{N} := \frac{1}{n}\boldsymbol{B}{\boldsymbol{V}_{\text{goal}_{proj}}}^T \in \mathbb{R}(C[-1], C[+1]) \tag{C.20}$$

Doing so we have that the following equivalence between the two optimization problems :

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}) - \boldsymbol{V}_{\text{goal}_{proj}}\right\| = \arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \left\|\boldsymbol{\Omega}\boldsymbol{A}^T\mathbf{S}^{\frac{1}{2}} - \boldsymbol{N}^T\mathbf{S}^{-\frac{1}{2}}\right\| \tag{C.21}$$

Before continuing the proof, we now prove the same type of equivalence for convolutional layers.

### C.2.2 Convolutional connected layers

Let $\boldsymbol{A} \in \mathbb{R}(K, C[-1], d, d)$ and $\boldsymbol{\Omega} \in \mathbb{R}(C[+1], K, d[+1], d[+1])$ where $d, d[+1]$ is the kernel size at $l$ and $l+1$. Let $\boldsymbol{A}_F$ the flatten and transposed version of $\boldsymbol{A}$ of shape $(C[-1]dd, K)$ and $\boldsymbol{\alpha}_k := \boldsymbol{A}_F[:, k] \in (C[-1]dd, 1)$. Let $\boldsymbol{\Omega}$ with the last order flatten *i.e.* $\boldsymbol{\Omega} \in \mathbb{R}(C[+1], K, d[+1]d[+1])$. Let $\boldsymbol{\omega}_{k,m} := \boldsymbol{\Omega}[m, k] \in (d[+1]d[+1], 1)$. Using this, let $\boldsymbol{\Omega}[m]_F := \begin{pmatrix} \boldsymbol{\omega}_{1,m}^T \\ \vdots \\ \boldsymbol{\omega}_{K,m}^T \end{pmatrix} \in (K, d[+1]d[+1])$ and $\boldsymbol{\Omega}_F := \begin{pmatrix} \boldsymbol{\Omega}[1]_F & \cdots & \boldsymbol{\Omega}[m]_F \end{pmatrix} \in (K, C[+1]d[+1]d[+1])$.

Let $\boldsymbol{T}$ the tensor such that for a pixel $j$ of the output of the convolutional layer, $\boldsymbol{T}_j$ is a linear application that select the pixels of the input of the convolutional layer that are used to compute the pixel $j$ of the output in a flatten version image (flatten only on the space not on the channels). $\boldsymbol{T} \in \mathbb{R}(H[+1]W[+1], d[+1]d[+1], HW)$ where $H$ and $W$ are the height and width of the intermediate image and $H[+1]$ and $W[+1]$ are the height and width of the output image.

As previously, $\boldsymbol{B}^c$ the unfolded version of $\boldsymbol{B}$ is such that $\boldsymbol{B}^c \in \mathbb{R}(n, C[-1]dd, HW)$ satisfies $\mathbf{Conv}(\boldsymbol{B}_i)$ is equal, with the correct reshape, to $\boldsymbol{A}\boldsymbol{B}_i^c$.

In addition, let $j$ an index on the space of pixel instead of having a couple $h, w$ for height and width. With those notations, it follows that :

$$\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega})[i, m, j] = \mathbf{Conv}_{(C[+1],K)}(\boldsymbol{\Omega})(\mathbf{Conv}_{(K,C[-1])}(\boldsymbol{A})(\boldsymbol{B}_i))[m, j] \tag{C.22}$$

$$= \sum_k^K \boldsymbol{\omega}_{m,k}^T \boldsymbol{T}_j (\boldsymbol{B}_i^c)^T \boldsymbol{\alpha}_k \tag{C.23}$$

In the following for simplicity, let $\boldsymbol{B}_{i,j}^t := \boldsymbol{T}_j(\boldsymbol{B}_i^c)^T$. To find the best neurons

to add, let consider the expressivity bottleneck as :

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n} \sum_i \sum_j \sum_m \left\| \boldsymbol{V}_{\text{goal}_{\text{proj}_i}}^{(j,m)} - \sum_{k=1}^{K} \boldsymbol{\omega}_{m,k}^T \boldsymbol{B}_{i,j}^t \boldsymbol{\alpha}_k \right\|^2 \tag{C.24}$$

$$\tag{C.25}$$

Using the properties of the trace, it follows that :

$$\sum_{i,j,m} \left\| \boldsymbol{V}_{\text{goal}_{\text{proj}_i}}^{(j,m)} - \sum_{k=1}^{K} \boldsymbol{\omega}_{m,k}^T \boldsymbol{B}_{i,j}^t \boldsymbol{\alpha}_k \right\|^2 = \sum_{i,j,m} \left\| \boldsymbol{V}_{\text{goal}_{\text{proj}_i}}^{(j,m)} - \sum_k \text{tr}\left( \boldsymbol{B}_{i,j}^t \boldsymbol{\alpha}_k \boldsymbol{\omega}_{m,k}^T \right) \right\|^2 \tag{C.26}$$

$$= \sum_{i,j,m} \left\| \boldsymbol{V}_{\text{goal}_{\text{proj}_i}}^{(j,m)} - \text{tr}\left( \boldsymbol{B}_{i,j}^t \overbrace{\sum_k \boldsymbol{\alpha}_k \boldsymbol{\omega}_{m,k}^T}^{\boldsymbol{F}_m} \right) \right\|^2 \tag{C.27}$$

$$= \sum_{i,j,m} \left\| \boldsymbol{V}_{\text{goal}_{\text{proj}_i}}^{(j,m)} - flat(\boldsymbol{B}_{i,j}^t)^T flat(\boldsymbol{F}_m) \right\|^2 \tag{C.28}$$

$$= \sum_{i,j} \left\| \boldsymbol{V}_{\text{goal}_{\text{proj}_i}}^{(j)} - flat(\boldsymbol{B}_{i,j}^t)^T \boldsymbol{F} \right\|^2 \tag{C.29}$$

$$\tag{C.30}$$

With $\boldsymbol{F} := \begin{pmatrix} flat(\boldsymbol{F}_1) & \dots & flat(\boldsymbol{F}_{C[+1]}) \end{pmatrix}$.

It follows that $\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega})$ is a linear function of the matrix $\boldsymbol{F}$ which implies that the solution of C.24 is the same as for linear layer. Replacing $\boldsymbol{\Omega A}$ by $\boldsymbol{F}$ in C.14 and following the same reasoning as for linear layer, it follows that C.24 is equivalent to :

$$\arg\min_{\boldsymbol{F}} \left\| \mathbf{S}^{\frac{1}{2}} \boldsymbol{F} - \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\| \tag{C.31}$$

with $\boldsymbol{S} := \sum_{i,j} flat(\boldsymbol{B}_{i,j}^t) flat(\boldsymbol{B}_{i,j}^t)^T$ and $\boldsymbol{N} := \sum_{i,j} \boldsymbol{V}_{\text{goal}_{\text{proj}}}{}^j flat(\boldsymbol{B}_{i,j}^t)^T$.

However, the dimension of $\boldsymbol{S} \in \mathbb{R}(C[-1]d[+1]^2 d^2, C[-1]d[+1]^2 d^2)$ is quite large and that computing the SVD of such matrix is costly. To avoid expensive computation, C.24 is approximated by defining the matrix $\boldsymbol{S}$ and $\boldsymbol{N}$ as C.32 and C.34. It is now proved that 3.2.3, 3.2.5 and eq. (3.23) still hold with such new definitions of $\boldsymbol{S}$ and $\boldsymbol{N}$.

**Lemma C.2.3.** *Let* $r := \min(\boldsymbol{B}^t_{1,1}.shape)$*, we define:*

$$\boldsymbol{S} := \frac{r}{n}\sum_{i=1}^{n}\sum_{j=1}^{H[+1]W[+1]} (\boldsymbol{B}^t_{i,j})^T(\boldsymbol{B}^t_{i,j}) \in (C[-1]dd, C[-1]dd) \tag{C.32}$$

$$\boldsymbol{N}_m := \frac{1}{n}\sum_{i,j}^{n,H[+1]W[+1]} \boldsymbol{V}_{goal_{proj}i,j,m}(\boldsymbol{B}^t_{i,j})^T \in (C[-1]dd, d[+1]d[+1]) \tag{C.33}$$

$$\boldsymbol{N} := \left(\boldsymbol{N}_1 \cdots \boldsymbol{N}_{C[+1]}\right) \in (C[-1]dd, C[+1]d[+1]d[+1]) \tag{C.34}$$

*We have:*

$$\frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}) - \boldsymbol{V}_{goal_{proj}}\right\|^2 \leq \left\|\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}_F\boldsymbol{\Omega}_F - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 - \left\|\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 + \frac{1}{n}\left\|\boldsymbol{V}_{goal_{proj}}\right\|^2 \tag{C.35}$$

The proof in 8. We now continue the general proof of the main theorem for convolutional and linear layers using the following lemma :

**Lemma C.2.4.** *For* $\boldsymbol{S} \in \mathbb{R}(s,s), \boldsymbol{N} \in \mathbb{R}(s,t), \boldsymbol{A} \in \mathbb{R}(s,K), \boldsymbol{\Omega} \in \mathbb{R}(K,t)$*.*

*Let* $\boldsymbol{U}\Lambda\boldsymbol{V}$ *the singular value decomposition of* $\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}$ *and* $\boldsymbol{U}_K$ *the first* $K$ *columns of* $\boldsymbol{U}$*,* $V_K$ *the first* $K$ *lines of* $\boldsymbol{V}$*,* $\Lambda_K$ *the first* $K$ *singular values of* $\Lambda$ *and* $\Lambda_{K+1:}$ *the other singular values of* $\Lambda$*.*

*Let :*

$$\boldsymbol{A}^* := \mathbf{S}^{-\frac{1}{2}}\boldsymbol{U}_K\sqrt{\Lambda_K} \tag{C.36}$$

$$\boldsymbol{\Omega}^* := \sqrt{\Lambda_K}\boldsymbol{V}_K \tag{C.37}$$

$$\min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}) - \boldsymbol{V}_{goal_{proj}}\right\|^2 \leq \frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A}^*,\boldsymbol{\Omega}^*) - \boldsymbol{V}_{goal_{proj}}\right\|^2 = -\left\|\Lambda_K\right\|^2 + \frac{1}{n}\left\|\boldsymbol{V}_{goal_{proj}}\right\|^2 \tag{C.38}$$

*with equality for the linear case.*

$$\Psi^l_{\theta\oplus\theta^{K*}_{\leftrightarrow}} \leq \Psi^l_\theta - \sum_{k=1}^{K}\lambda_k^2 \tag{C.39}$$

*Proof:* Using theorem C.2.2 and theorem C.2.3:

$$\frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}) - \boldsymbol{V}_{\text{goal}_{proj}}\right\|^2 \leq \left\|\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}\boldsymbol{\Omega} - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 - \left\|\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 + \frac{1}{n}\overbrace{\left\|\boldsymbol{V}_{\text{goal}_{proj}}\right\|^2}^{\Psi^l_\theta} \tag{C.40}$$

Hence, the second term of the right term is minimized :

$$\arg\min_{\boldsymbol{A},\boldsymbol{\Omega}} \left\| \mathbf{S}^{\frac{1}{2}}\boldsymbol{A}\boldsymbol{\Omega} - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \right\| \tag{C.41}$$

It is assumed that $S$ is invertible, let consider the change of variable $\widetilde{\boldsymbol{A}} = \mathbf{S}^{\frac{1}{2}}\boldsymbol{A}$, it follows that :

$$\min_{\boldsymbol{A},\boldsymbol{\Omega}} \left\| \mathbf{S}^{\frac{1}{2}}\boldsymbol{A}\boldsymbol{\Omega} - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \right\| = \min_{\widetilde{\boldsymbol{A}},\boldsymbol{\Omega}} \left\| \widetilde{\boldsymbol{A}}\boldsymbol{\Omega} - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \right\| \tag{C.42}$$

The solution of such problems is given by the paper Eckart and Young [1936] and is:

$$\widetilde{\boldsymbol{A}}^* = \boldsymbol{U}_K\sqrt{\Lambda_K} \tag{C.43}$$

$$\boldsymbol{\Omega}^* = \sqrt{\Lambda_K}\boldsymbol{V}_K \tag{C.44}$$

To recover $\boldsymbol{A}^*$ one has to multiply by $\mathbf{S}^{-\frac{1}{2}}$ on the left side of $\widetilde{\boldsymbol{A}}^*$. By definition of the SVD and the construction of $(\boldsymbol{A}^*, \boldsymbol{\Omega}^*)$, it follows that :

$$\left\| \mathbf{S}^{\frac{1}{2}}\boldsymbol{A}^*\boldsymbol{\Omega}^* - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \right\|^2 = \|\Lambda_{K+1:}\|^2 - \|\Lambda\|^2 = -\|\Lambda_K\|^2 \tag{C.45}$$

Using this and eq. (C.40) the desired equation eq. (C.38) is straightforward. To conclude, the bottleneck expression can be re-written as follows :

$$\Psi_{\theta\oplus\theta^K_{\leftrightarrow}} := \min_{\boldsymbol{A},\boldsymbol{\Omega}} \frac{1}{n} \left\| \boldsymbol{V}(\boldsymbol{A},\boldsymbol{\Omega}) - \boldsymbol{V}_{\text{goal}_{proj}} \right\|^2 \leq \Psi^l_\theta - \sum_{k=1}^{K} \lambda_k^2 \tag{C.46}$$

■

## C.3 Proof of 3.2.4

For $\gamma > 0$, solving (3.21) using $\boldsymbol{V}_{goal_{proj}} = \boldsymbol{V}_{goal} - \boldsymbol{V}(\gamma\delta\boldsymbol{W}^*)$ is equivalent to minimizing the loss $\mathcal{L}$ at order one in $\gamma\boldsymbol{V}^l$. Furthermore, performing an architecture update with $\gamma\delta\boldsymbol{W}^*$ (3.17) and a neuron addition with $\gamma\theta^{K,*}_{\leftrightarrow}$ (3.2.3) has an impact on the loss at first order in $\gamma$ as :

$$\begin{aligned} \mathcal{L}(f_{\theta\oplus\gamma\theta^{K,*}_{\leftrightarrow}}) &:= \frac{1}{n}\sum_{i=1}^{n} \ell(f_{\theta\oplus\gamma\theta^{K,*}_{\leftrightarrow}}(\boldsymbol{x}_i), \boldsymbol{y}_i) \\ &= \mathcal{L}(f_\theta) - \gamma\left(\sigma'_{l-1}(0)\Delta_{\theta^{K,*}_{\leftrightarrow}} + \Delta_{\delta\boldsymbol{W}^*}\right) + o(\gamma) \end{aligned} \tag{C.47}$$

with

$$\Delta_{\theta^{K,*}_{\leftrightarrow}} := \frac{1}{n}\left\langle \boldsymbol{V}^l_{\text{goal}_{proj}},\ \boldsymbol{V}^l(\theta^{K,*}_{\leftrightarrow}) \right\rangle = \sum_{k=1}^{K}\lambda_k^2 \tag{C.48}$$

$$\Delta_{\delta\boldsymbol{W}^*} := \frac{1}{n}\left\langle \boldsymbol{V}^l_{\text{goal}},\ \boldsymbol{V}^l(\delta\boldsymbol{W}^*) \right\rangle \geqslant 0\,. \tag{C.49}$$

To prove such proposition the following lemma is used :

**Lemma C.3.1.** *We note $\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega})$ the result of $\boldsymbol{B}$ after applying the layers parameterized by $\boldsymbol{A}$ and $\boldsymbol{\Omega}$. We note $\boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*)$ where $\boldsymbol{A}^*$ and $\boldsymbol{B}^*$ as define in 3.2.3*

$$\frac{1}{n}\left\langle \boldsymbol{V}_{goal_{proj}}, \boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*)\right\rangle = \|\Lambda_K\|^2 \tag{C.50}$$

*Proof:* Starting from theorem C.2.4

$$\frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*) - \boldsymbol{V}_{\text{goal}_{\text{proj}}}\right\|^2 = -\|\Lambda_K\|^2 + \frac{1}{n}\left\|\boldsymbol{V}_{\text{goal}_{\text{proj}}}\right\|^2 \tag{C.51}$$

Hence by developing the norm, we have:

$$\frac{1}{n}\|\boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*)\|^2 - \frac{2}{n}\left\langle \boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*), \boldsymbol{V}_{\text{goal}_{\text{proj}}}\right\rangle = -\|\Lambda_K\|^2 \tag{C.52}$$

Moreover by construction we have $\frac{1}{n}\|\boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*)\|^2 = \|\Lambda_K\|^2$ and therefore we get:

$$-\frac{2}{n}\left\langle \boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*), \boldsymbol{V}_{\text{goal}_{\text{proj}}}\right\rangle = -2\|\Lambda_K\|^2 \tag{C.53}$$

which conclude the proof. ∎

The main proposition is now proved. Suppose that each quantity is added to the architecture with an amplitude factor $\gamma$ *i.e.* the best update is then $\gamma\,\delta\boldsymbol{W}^*$ and the new neurons are $\{\sqrt{\gamma}\boldsymbol{\alpha}_i^*, \sqrt{\gamma}\boldsymbol{\omega}_i^*\}_i$.

Using the Fréchet derivative on $\gamma$,it follows that:

$$\mathcal{L}(\boldsymbol{a}^l + \gamma\,\delta\boldsymbol{a}^l) = \mathcal{L}(\boldsymbol{a}^l) + \left\langle \nabla_{\boldsymbol{a}^l}\mathcal{L}, \gamma\,\delta\boldsymbol{a}^l\right\rangle + o(\gamma) \tag{C.54}$$

On one hand, performing an update of architecture, ie $\boldsymbol{W}^* \leftarrow \boldsymbol{W} + \gamma\,\delta\boldsymbol{W}^*$, changes the activation function $\boldsymbol{a}^l$ by $\gamma\,\delta\boldsymbol{a}_u^l := \boldsymbol{V}(\gamma\,\delta\boldsymbol{W}^*)$. Then, as explained in Appendix B.4, adding neurons $(\boldsymbol{A}^*, \boldsymbol{\Omega}^*)$ at layer $l-1$ changes the activation function $\boldsymbol{a}^l$ by :

$$\gamma\,\delta\boldsymbol{a}_a^l = \sigma'_{l-1}(0)\,\gamma\,\boldsymbol{V}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*) + o(\gamma)\ . \tag{C.55}$$

It is supposed that $\delta\boldsymbol{a}_u^l \neq -\delta\boldsymbol{a}_a^l$ and perform a first order development in $\gamma$. Then combining Equations (C.54) and (C.55), it follows that :

$$\mathcal{L}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*) = \mathcal{L} + \left\langle \nabla_{\boldsymbol{a}^l}\mathcal{L}, \gamma\left(\delta\boldsymbol{a}_u^l + \delta\boldsymbol{a}_a^l\right)\right\rangle + o(\gamma)\ . \tag{C.56}$$

Using that $\boldsymbol{v}_{\text{goal}}(\boldsymbol{x}_i) := -\nabla_{\boldsymbol{a}^l(\boldsymbol{x}_i)}\ell(\boldsymbol{x}_i)$ and that $\mathcal{L} = \frac{1}{n}\sum_i \ell(\boldsymbol{x}_i)$,

it follows that :

$$\mathcal{L}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*) = \mathcal{L} - \frac{1}{n}\left\langle \boldsymbol{V}_{\text{goal}}, \gamma\left(\delta \boldsymbol{a}_u^l + \delta \boldsymbol{a}_a^l\right)\right\rangle + o(\gamma) \tag{C.57}$$

$$= \mathcal{L} - \frac{\gamma}{n}\Big( \left\langle \boldsymbol{V}_{\text{goal}}, \delta \boldsymbol{a}_u^l\right\rangle + \left\langle \boldsymbol{V}_{\text{goal}} - \boldsymbol{V}(\gamma \delta \boldsymbol{W}^*), \delta \boldsymbol{a}_a^l\right\rangle + \gamma \left\langle \delta \boldsymbol{a}_u^l, \delta \boldsymbol{a}_a^l\right\rangle \Big) + o(\gamma)\ . \tag{C.58}$$

Using C.3.1, it follows that :

$$\mathcal{L}(\boldsymbol{A}^*, \boldsymbol{\Omega}^*) = \mathcal{L} - \gamma\left(\sigma'_{l-1}(0)\sum_{k=1}^{K}\lambda_k^2 + \frac{1}{n}\left\langle \boldsymbol{V}_{\text{goal}}, \delta \boldsymbol{a}_u^l\right\rangle\right) + o(\gamma)\ . \tag{C.59}$$

**Note on the approximation for convolutional layer.** By developing the expression $||\boldsymbol{V} - \boldsymbol{V}_{\text{goal}_{proj}}||^2$, one can remark that minimizing $||\boldsymbol{V} - \boldsymbol{V}_{\text{goal}_{proj}}||^2$ over $\boldsymbol{V}$ is equivalent to maximizing $\langle \boldsymbol{V}, \boldsymbol{V}_{\text{goal}_{proj}}\rangle$ with a constraint on the norm of $\boldsymbol{V}$. This constraint lies in the functional space of the activities and can be reformulated in the parameter space with the matrix $\boldsymbol{S}$ as $||\boldsymbol{A}\boldsymbol{\Omega}^T||_{\boldsymbol{S}} = ||\boldsymbol{V}||$. By changing the matrix $\boldsymbol{S}$ for another positive semi-definite matrix $\boldsymbol{S}_{pseudo}$, the metric on $\boldsymbol{V}$ is modified and a pseudo-solution $\boldsymbol{S}_{pseudo}^{-1}\boldsymbol{N}$ is obtained.

## C.4 Theorem 3.2.5 and Section 3.2.3

(Theorem 3.2.5) Suppose $\boldsymbol{S}$ is semi definite, let $\boldsymbol{S} = \boldsymbol{S}^{\frac{1}{2}}\boldsymbol{S}^{\frac{1}{2}}$. Solving Equation (3.21) is equivalent to find the K first eigenvectors $\boldsymbol{\alpha}_k$ associated to the K largest eigenvalues $\lambda$ of the generalized eigenvalue problem :

$$\boldsymbol{N}\boldsymbol{N}^T\boldsymbol{\alpha}_k = \lambda \boldsymbol{S}\boldsymbol{\alpha}_k \tag{C.60}$$

(Section 3.2.3) For all integers $m, m'$ such that $m + m' \leqslant R$, at order one in $\eta$, adding $m + m'$ neurons simultaneously according to the previous method is equivalent to adding $m$ neurons then $m'$ neurons by applying successively the previous method twice while imposing an orthogonality constraint between the already $m$ added neurons and the next $m'$ neurons to add.

*Proof*

To prove Theorem 3.2.5, it is sufficient to prove that the solution of the generalized eigenvalue problem as stated in Equation (C.60) is collinear to the formula of Theorem 3.2.3.

Solving C.60 is equivalent to maximizing the following generalized Rayleigh quotient (which is solvable by the LOBPCG technique):

$$\boldsymbol{\alpha}^* = \arg\max_{\alpha} \frac{\boldsymbol{\alpha}^T \boldsymbol{N}\boldsymbol{N}^T\boldsymbol{\alpha}}{\boldsymbol{\alpha}^T \boldsymbol{S}\boldsymbol{\alpha}} \tag{C.61}$$

$$\boldsymbol{p}^* = \arg\max_{\boldsymbol{p}=\boldsymbol{S}^{1/2}\boldsymbol{\alpha}} \frac{\boldsymbol{p}^T \boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N}\boldsymbol{N}^T\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{p}}{\boldsymbol{p}^T\boldsymbol{p}} \tag{C.62}$$

$$\boldsymbol{p}^* = \arg\max_{||\boldsymbol{p}||=1} ||\boldsymbol{N}^T\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{p}|| \tag{C.63}$$

$$\boldsymbol{\alpha}^* = \boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{p}^* \tag{C.64}$$

Considering the SVD of $\boldsymbol{N}^T\boldsymbol{S}^{-\frac{1}{2}} = \sum_{r=1}^{R} \lambda_r \boldsymbol{e}_r \boldsymbol{f}_r^T$, then $\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N}\boldsymbol{N}^T\boldsymbol{S}^{-\frac{1}{2}} = \sum_{r=1}^{R} \lambda_r^2 \boldsymbol{f}_r \boldsymbol{f}_r^T$, because $i \neq j \implies \boldsymbol{e}_i^T\boldsymbol{e}_j = 0$ and $\boldsymbol{f}_i^T\boldsymbol{f}_j = 0$. Hence maximizing the first quantity is equivalent to $\boldsymbol{p}_k^* = \boldsymbol{f}_k$, then $\boldsymbol{\alpha}_k = \boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{f}_k$, which match the formula of proposition 3.2.3. The same reasoning can be applied on $\boldsymbol{\omega}_k$.

The second corollary of Section 3.2.3 is proved by induction. Note that $\boldsymbol{v}(\theta_{\leftrightarrow}^{K,*}, \boldsymbol{x}) = o(\eta)$ because $\boldsymbol{v}_{\text{goal}} = o(\eta)$, then for $m = m' = 1$ :

$$\boldsymbol{a}_l(\boldsymbol{x})^{t+1} = \boldsymbol{a}_l(\boldsymbol{x})^t + \boldsymbol{v}(\theta_{\leftrightarrow}^{1,*}, \boldsymbol{x}) + o(\eta) \tag{C.65}$$

Remark that $\boldsymbol{v}_{\text{goal}}(\boldsymbol{x})$ is a function of $\boldsymbol{a}_l(\boldsymbol{x})$, ie $\boldsymbol{v}_{\text{goal}}(\boldsymbol{x}) := g(\boldsymbol{a}_l(\boldsymbol{x}))$. Then suppose that $\mathcal{L}(f(\boldsymbol{x}), \boldsymbol{y}))$ is twice differentiable in $\boldsymbol{a}_l(\boldsymbol{x})$. It follows that $g(\boldsymbol{a}_l(\boldsymbol{x}))$ is differentiable and :

$$\begin{aligned}
{\boldsymbol{v}_{\text{goal}}}^{t+1}(\boldsymbol{x}) &= g(\boldsymbol{a}_l^t(\boldsymbol{x}) + \boldsymbol{v}(\theta_{\leftrightarrow}^{1,*}, \boldsymbol{x})) && \text{(C.66)}\\
&= g(\boldsymbol{a}_l^t(\boldsymbol{x})) + \nabla_{\boldsymbol{a}_l^t(\boldsymbol{x})} g(\boldsymbol{a}_l^t(\boldsymbol{x}))^T \boldsymbol{v}(\theta_{\leftrightarrow}^{1,*}, \boldsymbol{x}) + o(\eta^2) && \text{(C.67)}\\
&= {\boldsymbol{v}_{\text{goal}}}^t(\boldsymbol{x}) + \eta \frac{\partial^2 \mathcal{L}(f_\theta(\boldsymbol{x}), \boldsymbol{y})}{\partial \boldsymbol{a}^l(\boldsymbol{x})^2} \boldsymbol{v}(\theta_{\leftrightarrow}^{1,*}, \boldsymbol{x}) + o(\eta^2) && \text{(C.68)}\\
&= {\boldsymbol{v}_{\text{goal}}}^t(\boldsymbol{x}) + o(\eta) && \text{(C.69)}
\end{aligned}$$

Adding the second neuron, the following minimization problem is obtained:

$$\arg\min_{\boldsymbol{\alpha}_2, \boldsymbol{\omega}_2} ||{\boldsymbol{V}_{\text{goal}}}^t - \boldsymbol{V}(\boldsymbol{\alpha}_2, \boldsymbol{\omega}_2)|| \tag{C.70}$$

## C.5 Lemmas

Proofs of the lemmas :

**Proposition C.5.1.** $\forall \boldsymbol{D} \in \mathbb{R}^{p,q}, \boldsymbol{B} \in \mathbb{R}^{k,q}$,

$$\exists c \in \mathbb{R} \quad s.t, \quad \arg\min_{\boldsymbol{H}} \|\boldsymbol{D} - \boldsymbol{H}\boldsymbol{B}\|^2 = \arg\max_{\boldsymbol{H},\ \|\boldsymbol{H}\boldsymbol{B}\|^2 \leq c} \langle \boldsymbol{D}, \boldsymbol{H}\boldsymbol{B} \rangle$$

*Proof:* Indeed,

$$\begin{aligned}
\arg\min_{\boldsymbol{H}} \|\boldsymbol{D} - \boldsymbol{H}\boldsymbol{B}\|^2 &= \arg\min_{\boldsymbol{H}} \|\boldsymbol{H}\boldsymbol{B}\|^2 - 2\langle \boldsymbol{D}, \boldsymbol{H}\boldsymbol{B} \rangle && \text{(C.71)} \\
&= \arg\min_{\boldsymbol{H}} \|\boldsymbol{H}\boldsymbol{B}\|^2 - 2\|\boldsymbol{H}\boldsymbol{B}\| \left\langle \boldsymbol{a}, \frac{\boldsymbol{H}\boldsymbol{B}}{\|\boldsymbol{H}\boldsymbol{B}\|} \right\rangle && \text{(C.72)} \\
&= \arg\min_{h\boldsymbol{U},\ \|\boldsymbol{H}\boldsymbol{B}\| = h,\, \boldsymbol{U} = \frac{\boldsymbol{H}\boldsymbol{B}}{\|\boldsymbol{H}\boldsymbol{B}\|}} h^2 - 2h\langle \boldsymbol{D}, \boldsymbol{U} \rangle && \text{(C.73)} \\
& && \text{(C.74)}
\end{aligned}$$

Let $\boldsymbol{U}^* := \arg\max_{\boldsymbol{U} = \frac{\boldsymbol{H}\boldsymbol{B}}{\|\boldsymbol{H}\boldsymbol{B}\|}} \langle \boldsymbol{D}, \boldsymbol{U} \rangle$ which depends on $\boldsymbol{B}$ and $\boldsymbol{D}$ but does not depends on $h$. Then :

$$\begin{aligned}
\arg\min_{\boldsymbol{H}} \|\boldsymbol{D} - \boldsymbol{H}\boldsymbol{B}\|^2 &= \arg\min_{h\boldsymbol{U}^*,\, h \geq 0} h^2 - 2h \langle \boldsymbol{D}, \boldsymbol{U}^* \rangle && \text{(C.75)} \\
&= h^* \boldsymbol{U}^* && \text{(C.76)}
\end{aligned}$$

With the convention that $\frac{0}{\|0\|} = 0$. ∎

**Lemma C.5.2.** *For* $\boldsymbol{S} \in \mathbb{R}(s,s), \boldsymbol{N} \in \mathbb{R}(s,t), \boldsymbol{C} \in \mathbb{R}(t,s)$.
*If* $\boldsymbol{N} = \mathbf{S}^{\frac{1}{2}}\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}$, *then :*

$$\left\langle \boldsymbol{C}^T, \boldsymbol{S}\boldsymbol{C}^T \right\rangle - 2\left\langle \boldsymbol{N}, \boldsymbol{C}^T \right\rangle = \left\| \mathbf{S}^{\frac{1}{2}}\boldsymbol{C}^T - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N} \right\|^2 \qquad \text{(C.77)}$$

*Proof:*

- For the first term ::

$$\begin{aligned}
\left\langle \boldsymbol{C}^T, \boldsymbol{S}\boldsymbol{C}^T \right\rangle &= \left\langle \boldsymbol{C}^T, \mathbf{S}^{\frac{1}{2}}\mathbf{S}^{\frac{1}{2}}\boldsymbol{C}^T \right\rangle && \text{(C.78)} \\
&= \left\langle \mathbf{S}^{\frac{1}{2}}\boldsymbol{C}^T, \mathbf{S}^{\frac{1}{2}}\boldsymbol{C}^T \right\rangle && \text{(C.79)} \\
&= \left\| \mathbf{S}^{\frac{1}{2}}\boldsymbol{C}^T \right\|^2 && \text{(C.80)}
\end{aligned}$$

- For the second term :

$$\left\langle \boldsymbol{N}, \boldsymbol{C}^T \right\rangle = \left\langle \mathbf{S}^{\frac{1}{2}} \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N}, \boldsymbol{C}^T \right\rangle \tag{C.81}$$

$$= \left\langle \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N}, \mathbf{S}^{\frac{1}{2}} \boldsymbol{C}^T \right\rangle \tag{C.82}$$

Hence it follows that :

$$\left\langle \boldsymbol{C}^T, \boldsymbol{S}\boldsymbol{C}^T \right\rangle - 2 \left\langle \boldsymbol{N}, \boldsymbol{C}^T \right\rangle = \left\| \mathbf{S}^{\frac{1}{2}} \boldsymbol{C}^T \right\|^2 - 2 \left\langle \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N}, \mathbf{S}^{\frac{1}{2}} \boldsymbol{C}^T \right\rangle + \left\| \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 \tag{C.83}$$

$$= \left\| \mathbf{S}^{\frac{1}{2}} \boldsymbol{C}^T - \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 \tag{C.84}$$

■

**Lemma C.5.3.** *Let* $\boldsymbol{Y} \in \mathbb{R}(t, n), \boldsymbol{X} \in \mathbb{R}(s, n), \boldsymbol{C} \in \mathbb{R}(t, s)$
*We define:*

$$\boldsymbol{S} := \frac{1}{n} \boldsymbol{X}\boldsymbol{X}^T \in \mathbb{R}(s, s) \tag{C.85}$$

$$\boldsymbol{N} := \frac{1}{n} \boldsymbol{X}\boldsymbol{Y}^T \in \mathbb{R}(s, t) \tag{C.86}$$

$$\frac{1}{n} \left\| \boldsymbol{C}\boldsymbol{X} - \boldsymbol{Y} \right\|^2 = \left\| \boldsymbol{C}\mathbf{S}^{\frac{1}{2}} - \boldsymbol{N}^T \mathbf{S}^{-\frac{1}{2}} \right\|^2 - \left\| \mathbf{S}^{-\frac{1}{2}} \boldsymbol{N} \right\|^2 + \frac{1}{n} \left\| \boldsymbol{Y} \right\|^2 \tag{C.87}$$

*Proof:* By developing the scalar product, it follows that:

$$\frac{1}{n} \left\| \boldsymbol{C}\boldsymbol{X} - \boldsymbol{Y} \right\|^2 = \frac{1}{n} \left\| \boldsymbol{Y} \right\|^2 - 2 \left\langle \boldsymbol{Y}, \frac{1}{n} \boldsymbol{C}\boldsymbol{X} \right\rangle + \frac{1}{n} \left\| \boldsymbol{C}\boldsymbol{X} \right\|^2 \tag{C.88}$$

$$= \frac{1}{n} \left\| \boldsymbol{Y} \right\|^2 - 2 \left\langle \boldsymbol{Y}, \frac{1}{n} \boldsymbol{C}\boldsymbol{X} \right\rangle + \frac{1}{n} \left\langle \boldsymbol{C}\boldsymbol{X}, \boldsymbol{C}\boldsymbol{X} \right\rangle \tag{C.89}$$

$$= \frac{1}{n} \left\| \boldsymbol{Y} \right\|^2 - 2 \left\langle \boldsymbol{Y}^T, \frac{1}{n} \left( \boldsymbol{C}\boldsymbol{X} \right)^T \right\rangle + \frac{1}{n} \left\langle \left( \boldsymbol{C}\boldsymbol{X} \right)^T, \left( \boldsymbol{C}\boldsymbol{X} \right)^T \right\rangle \tag{C.90}$$

$$= \frac{1}{n} \left\| \boldsymbol{Y} \right\|^2 - 2 \left\langle \frac{1}{n} \boldsymbol{X}\boldsymbol{Y}^T, \boldsymbol{C}^T \right\rangle + \left\langle \boldsymbol{C}^T, \frac{1}{n} \boldsymbol{X}\boldsymbol{X}^T \boldsymbol{C}^T \right\rangle \tag{C.91}$$

The two following lemma are used :

**Lemma C.5.4.** *Let* $\boldsymbol{Y} \in \mathbb{R}(t, n), \boldsymbol{X} \in \mathbb{R}(s, n)$ *and* $\boldsymbol{S} := \boldsymbol{X}\boldsymbol{X}^T \in \mathbb{R}(s, s)$.

$$\mathbf{S}^{\frac{1}{2}} \mathbf{S}^{-\frac{1}{2}} \boldsymbol{X}\boldsymbol{Y}^T = \boldsymbol{X}\boldsymbol{Y}^T \tag{C.92}$$

*Proof:* Let decompose $\boldsymbol{Y}$ on $\mathrm{Im}(\boldsymbol{X}^T) \oplus_\perp \ker(\boldsymbol{X})$: $\boldsymbol{Y} = \boldsymbol{X}^T I + \boldsymbol{K}$.

$$\boldsymbol{X}\boldsymbol{Y}^T = \boldsymbol{X}\boldsymbol{X}^T I + \boldsymbol{X}K = \boldsymbol{X}\boldsymbol{X}^T I = \boldsymbol{S}I$$

Hence $\boldsymbol{X}\boldsymbol{Y}^T \in \mathrm{Im}(\boldsymbol{S})$, hence as $\boldsymbol{S}_{|\mathrm{Im}(\boldsymbol{S})}$ is invertible, it follows that: $\mathbf{S}^{-\frac{1}{2}}\mathbf{S}^{\frac{1}{2}}\boldsymbol{X}\boldsymbol{Y}^T = \boldsymbol{X}\boldsymbol{Y}^T$. ■

Continuing the demonstration from eq. (C.91), by applying theorem C.5.2, it follows that:

$$\frac{1}{n}\left\|\boldsymbol{C}\boldsymbol{X} - \boldsymbol{Y}\right\|^2 = \frac{1}{n}\left\|\boldsymbol{Y}\right\|^2 - 2\left\langle \boldsymbol{N}, \boldsymbol{C}^T\right\rangle + \left\langle \boldsymbol{C}^T, \boldsymbol{S}\boldsymbol{C}^T\right\rangle \tag{C.93}$$

$$= \frac{1}{n}\left\|\boldsymbol{Y}\right\|^2 + \left\|\mathbf{S}^{\frac{1}{2}}\boldsymbol{C}^T - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 - \left\|\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 \tag{C.94}$$

$$= \frac{1}{n}\left\|\boldsymbol{Y}\right\|^2 + \left\|\boldsymbol{C}\mathbf{S}^{\frac{1}{2}} - \boldsymbol{N}^T\mathbf{S}^{-\frac{1}{2}}\right\|^2 - \left\|\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 \tag{C.95}$$

■

**Lemma C.5.5.** *Let* $r := \min(\boldsymbol{B}^t_{1,1}.shape)$*, we define:*

$$\boldsymbol{S} := \frac{r}{n}\sum_{i=1}^{n}\sum_{j=1}^{H[+1]W[+1]} (\boldsymbol{B}^t_{i,j})^T(\boldsymbol{B}^t_{i,j}) \in (C[-1]dd, C[-1]dd) \tag{C.96}$$

$$\boldsymbol{N}_m := \frac{1}{n}\sum_{i,j}^{n,H[+1]W[+1]} \boldsymbol{V}_{goal_{proj}i,j,m}(\boldsymbol{B}^t_{i,j})^T \in (C[-1]dd, d[+1]d[+1]) \tag{C.97}$$

$$\boldsymbol{N} := \left(\boldsymbol{N}_1 \cdots \boldsymbol{N}_{C[+1]}\right) \in (C[-1]dd, C[+1]d[+1]d[+1]) \tag{C.98}$$

*We have:*

$$\frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}) - \boldsymbol{V}_{goal_{proj}}\right\|^2 \leq \left\|\mathbf{S}^{\frac{1}{2}}\boldsymbol{A}_F\boldsymbol{\Omega}_F - \mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 - \left\|\mathbf{S}^{-\frac{1}{2}}\boldsymbol{N}\right\|^2 + \frac{1}{n}\left\|\boldsymbol{V}_{goal_{proj}}\right\|^2 \tag{C.99}$$

*Proof:* It follows that :

$$\frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}) - \boldsymbol{V}_{\mathrm{goal}_{proj}}\right\|^2 = \frac{1}{n}\left\|\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega})\right\|^2 - \frac{2}{n}\left\langle \boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}), \boldsymbol{V}_{\mathrm{goal}_{proj}}\right\rangle^2 + \frac{1}{n}\left\|-\boldsymbol{V}_{\mathrm{goal}_{proj}}\right\|^2 \tag{C.100}$$

The first two terms are now simplified separately.

### Norm simplification

**Lemma C.5.6.** *For any square matrix* $\boldsymbol{A} \in \mathbb{R}^{(n,n)}$,$\mathrm{tr}(\boldsymbol{A})^2 \leq \mathrm{rank}(\boldsymbol{A})\left\|\boldsymbol{A}\right\|^2$.

*Proof:* Using the truncated SVD we have $\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}$ with $\Sigma$ a diagonal and $\boldsymbol{U} \in \mathbb{R}^{(n,\text{rank}(\boldsymbol{A}))}, \boldsymbol{V} \in \mathbb{R}^{(\text{rank}(\boldsymbol{A}),n)}$ truncated orthonormal matrices.

We have:

$$\begin{aligned} \text{tr}(\boldsymbol{A})^2 &= \text{tr}(\boldsymbol{U}\Sigma\boldsymbol{V})^2 && \text{(C.101)} \\ &= \text{tr}(\boldsymbol{V}\boldsymbol{U}\Sigma)^2 && \text{(C.102)} \\ &= \langle \boldsymbol{V}\boldsymbol{U}, \Sigma \rangle^2 && \text{(C.103)} \\ \text{(Cauchy-Swarz)} \quad &\leq \|\boldsymbol{V}\boldsymbol{U}\|^2 \|\Sigma\|^2 && \text{(C.104)} \end{aligned}$$

As $\boldsymbol{U}, \boldsymbol{V}$ are truncated orthonormal matrices, we have:

$$\|\boldsymbol{V}\boldsymbol{U}\|^2 = \text{tr}(\boldsymbol{U}^T\boldsymbol{V}^T\boldsymbol{V}\boldsymbol{U}) = \text{tr}(\boldsymbol{U}^T\boldsymbol{U}) = \text{tr}(I_{\text{rank}(\boldsymbol{A})}) = \text{rank}(\boldsymbol{A})$$

Hence:

$$\text{tr}(\boldsymbol{A})^2 \leq \text{rank}(\boldsymbol{A}) \|\Sigma\|^2 \quad \text{(C.105)}$$

As $\boldsymbol{U}, \boldsymbol{V}$ are truncated orthonormal matrices, we have:

$$\|\Sigma\|^2 = \text{tr}(\Sigma^T\Sigma) = \text{tr}((\boldsymbol{V}\Sigma\boldsymbol{U})\boldsymbol{U}\Sigma\boldsymbol{V}) = \text{tr}(\boldsymbol{A}^T\boldsymbol{A}) = \|\boldsymbol{A}\|^2$$

We conclude that:

$$\text{tr}(\boldsymbol{A})^2 \leq \text{rank}(\boldsymbol{A}) \|\boldsymbol{A}\|^2 \quad \text{(C.106)}$$

■

**Lemma C.5.7.** *For $\delta\mathbf{W}_l \in (m,n), (\boldsymbol{u}_k)_{k\in[[K]]} \in (m)^K, (\boldsymbol{v}_k)_{k\in[[K]]} \in (n)^K$ and with $\boldsymbol{W} := \sum_{k\in[\![K]\!]} \boldsymbol{v}_k\boldsymbol{u}_k^T \in (n,m)$, it follows that:*

$$\left\| \sum_{k\in[\![K]\!]} \boldsymbol{u}_k^T \delta\mathbf{W}_l \boldsymbol{v}_k \right\|^2 = \text{tr}\left(\delta\mathbf{W}_l \boldsymbol{W}\right)^2 \quad \text{(C.107)}$$

*Proof:* Let $i \in I$:

$$\begin{aligned} \left\| \sum_{k\in[\![K]\!]} \boldsymbol{v}\boldsymbol{u}_k^T \delta\mathbf{W}_l \boldsymbol{v}_k \right\|^2 &= \left( \sum_{k\in[\![K]\!]} \boldsymbol{u}_k^T \delta\mathbf{W}_l \boldsymbol{v}_k \right)^2 && \text{(C.108)} \\ &= \text{tr}\left( \sum_{k\in[\![K]\!]} \boldsymbol{u}_k^T \delta\mathbf{W}_l \boldsymbol{v}_k \right)^2 && \text{(C.109)} \\ &= \text{tr}\left( \delta\mathbf{W}_l \sum_{k\in[\![K]\!]} \boldsymbol{v}_k\boldsymbol{u}_k^T \right)^2 && \text{(C.110)} \\ &= \text{tr}\left(\delta\mathbf{W}_l \boldsymbol{W}\right)^2 && \text{(C.111)} \end{aligned}$$

■

**Lemma C.5.8.** *For $(\delta \mathbf{W}_{li})_{i\in I} \in (m,n)^I$ such that $\forall i \in I, \operatorname{rank}(\delta \mathbf{W}_{li}\boldsymbol{W}) \leq H$ and with $\boldsymbol{W} \in (n,m)$, it follows that:*

$$\sum_{i\in I} \operatorname{tr}\left(\delta \mathbf{W}_{li}\boldsymbol{W}\right)^2 \leq \left\langle \boldsymbol{W}, H \sum_{i\in I} \delta \mathbf{W}_{li}^T \delta \mathbf{W}_{li}\boldsymbol{W} \right\rangle \tag{C.112}$$

*Proof:* Let $i \in I$:

Using theorem C.5.6 with $\boldsymbol{A} \leftarrow \delta \mathbf{W}_{li}\boldsymbol{W}$

$$\begin{aligned} \operatorname{tr}\left(\delta \mathbf{W}_{li}\boldsymbol{W}\right)^2 &\leq \operatorname{rank}(\delta \mathbf{W}_{li}\boldsymbol{W}) ||\delta \mathbf{W}_{li}\boldsymbol{W}||^2 && \text{(C.113)} \\ &\leq H \left\|\delta \mathbf{W}_{li}\boldsymbol{W}\right\|^2 \quad H := \min(\delta \mathbf{W}_{li}.shape) && \text{(C.114)} \end{aligned}$$

Hence, it follows that:

$$\begin{aligned} \sum_{i\in I} \left\| \sum_{k\in[\![K]\!]} \boldsymbol{u}_k^T \delta \mathbf{W}_{li} \boldsymbol{v}_k \right\|^2 &\leq H \sum_{i\in I} ||\delta \mathbf{W}_{li}\boldsymbol{W}||^2 && \text{(C.115)} \\ &= H \sum_{i\in I} \langle \delta \mathbf{W}_{li}\boldsymbol{W}, \delta \mathbf{W}_{li}\boldsymbol{W} \rangle && \text{(C.116)} \\ &= H \sum_{i\in I} \left\langle \boldsymbol{W}, \delta \mathbf{W}_{li}^T \delta \mathbf{W}_{li}\boldsymbol{W} \right\rangle && \text{(C.117)} \\ &= H \left\langle \boldsymbol{W}, \sum_{i\in I} \delta \mathbf{W}_{li}^T \delta \mathbf{W}_{li}\boldsymbol{W} \right\rangle && \text{(C.118)} \end{aligned}$$

■

Using first theorem C.5.7 with $\boldsymbol{u}_k \leftarrow \boldsymbol{\omega}_{k,m}$, $\boldsymbol{v}_k \leftarrow \boldsymbol{\alpha}_k$ and $\delta \mathbf{W}_l \leftarrow \boldsymbol{B}_{i,j}^t$, then:

$$\begin{aligned} \frac{1}{n} \left\|\boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega})\right\|^2 &= \frac{1}{n} \sum_{m}^{C[+1]} \sum_{i,j}^{n,H[+1]W[+1]} \left\| \sum_{k}^{K} \boldsymbol{\omega}_{k,m}^T (\boldsymbol{B}_{i,j}^t) \boldsymbol{\alpha}_k \right\|^2 && \text{(C.119)} \\ &= \frac{1}{n} \sum_{m}^{C[+1]} \sum_{i,j}^{n,H[+1]W[+1]} \operatorname{tr}\left( \boldsymbol{B}_{i,j}^t \sum_{k}^{K} \boldsymbol{\alpha}_k \boldsymbol{\omega}_{k,m}^T \right)^2 && \text{(C.120)} \\ &= \frac{1}{n} \sum_{m}^{C[+1]} \sum_{i,j}^{n,H[+1]W[+1]} \operatorname{tr}\left( \boldsymbol{B}_{i,j}^t \boldsymbol{A}_F \boldsymbol{\Omega}[m]_F \right)^2 && \text{(C.121)} \end{aligned}$$

Using theorem C.5.8 with $i \leftarrow (i,j)$, $\delta \mathbf{W}_{li} \leftarrow \boldsymbol{B}^t_{i,j}$ and $\boldsymbol{W} \leftarrow \boldsymbol{A}_F \boldsymbol{\Omega}[m]_F$:

$$\leq \sum_m^{C[+1]} \left\langle \boldsymbol{A}_F \boldsymbol{\Omega}[m]_F \ , \ \frac{r}{n} \sum_{i,j}^{n,H[+1]W[+1]} (\boldsymbol{B}^t_{i,j})^T (\boldsymbol{B}^t_{i,j}) \boldsymbol{A}_F \Omega[m]_F \right\rangle \tag{C.122}$$

$$= \sum_m^{C[+1]} \langle \boldsymbol{A}_F \boldsymbol{\Omega}[m]_F, \boldsymbol{S} \boldsymbol{A}_F \boldsymbol{\Omega}[m]_F \rangle \tag{C.123}$$

$$= \langle \boldsymbol{A}_F \boldsymbol{\Omega}_F, \boldsymbol{S} \boldsymbol{A}_F \boldsymbol{\Omega}_F \rangle \tag{C.124}$$

**Scalar product simplification**

**Lemma C.5.9.** *For $\delta \mathbf{W}_l \in (m,n), \boldsymbol{u} \in (m), \boldsymbol{v} \in (n)$, it follows that :*

$$\boldsymbol{u}^T \delta \mathbf{W}_l \boldsymbol{v} = \left\langle \boldsymbol{v}\boldsymbol{u}^T, \delta \mathbf{W}_l{}^T \right\rangle \tag{C.125}$$

*Proof:*

$$\boldsymbol{u}^T \delta \mathbf{W}_l \boldsymbol{v} = (\boldsymbol{u}^T \delta \mathbf{W}_l v)^T \tag{C.126}$$

$$= \boldsymbol{v}^T \delta \mathbf{W}_l{}^T \boldsymbol{u} \tag{C.127}$$

$$= \left\langle \boldsymbol{v}, \delta \mathbf{W}_l{}^T \boldsymbol{u} \right\rangle \tag{C.128}$$

$$= \left\langle \boldsymbol{v}\boldsymbol{u}^T, \delta \mathbf{W}_l{}^T \right\rangle \tag{C.129}$$

■

Then:

$$\frac{1}{n} \left\langle \boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}), \boldsymbol{V}_{\text{goal}_{proj}} \right\rangle^2 = \frac{1}{n} \sum_m^{C[+1]} \sum_{i,j}^{n,H[+1]W[+1]} \sum_k^K \boldsymbol{\omega}^T_{k,m} \boldsymbol{B}^t_{i,j} \boldsymbol{\alpha}_k \boldsymbol{V}_{\text{goal}_{proj}{}_{i,j,m}} \tag{C.130}$$

$$(\boldsymbol{V}_{\text{goal}_{proj}{}_{i,j,m}} \in (1)) \quad = \sum_m^{C[+1]} \sum_k^K \boldsymbol{\omega}^T_{k,m} \frac{1}{n} \sum_{i,j}^{n,H[+1]W[+1]} (\boldsymbol{B}^t_{i,j} \boldsymbol{V}_{\text{goal}_{proj}{}_{i,j,m}}) \boldsymbol{\alpha}_k \tag{C.131}$$

Using theorem C.5.9 with $\delta \mathbf{W}_l \leftarrow \sum_{i,j}^{n,H[+1]W[+1]} \boldsymbol{V}_{\text{goal}_{\text{proj}_{i,j,m}}} (\boldsymbol{B}_{i,j}^t)^T$

$$= \sum_m^{C[+1]} \left\langle \sum_k^K \boldsymbol{\alpha}_k \boldsymbol{\omega}_{k,m}^T, \frac{1}{n} \sum_{i,j}^{n,H[+1]W[+1]} \boldsymbol{V}_{\text{goal}_{proj}{}_{i,j,m}} (\boldsymbol{B}_{i,j}^t)^T \right\rangle \tag{C.132}$$

$$= \sum_m^{C[+1]} \langle \boldsymbol{A}_F \boldsymbol{\Omega}[m]_F, \boldsymbol{N}_m \rangle \tag{C.133}$$

$$= \langle \boldsymbol{A}_F \boldsymbol{\Omega}_F, \boldsymbol{N} \rangle \tag{C.134}$$

**Conclusion** In total :

$$\frac{1}{n} \left\| \boldsymbol{V}(\boldsymbol{A}, \boldsymbol{\Omega}) - \boldsymbol{V}_{\text{goal}_{proj}} \right\|^2 \leq \langle \boldsymbol{A}_F \boldsymbol{\Omega}_F, \boldsymbol{S} \boldsymbol{A}_F \boldsymbol{\Omega}_F \rangle - 2 \langle \boldsymbol{A}_F \boldsymbol{\Omega}_F, \boldsymbol{N} \rangle + \frac{1}{n} \left\| \boldsymbol{V}_{\text{goal}_{proj}} \right\|^2 \tag{C.135}$$

If it is supposed that $\boldsymbol{S}$ is invertible, one can apply theorem C.5.2 and get the result. ■

**Lemma C.5.10.** *For $f_\theta$ a linear feed-forward network with parameter $\theta$, when adding new neurons at layer $l-1$ with input weights $\boldsymbol{A} = 0$ and output weights $\boldsymbol{\Omega}$ with $\boldsymbol{A} \in \mathbb{R}(|\boldsymbol{b}_{l-2}(\boldsymbol{x})|, K)$ and $\boldsymbol{\Omega} \in \mathbb{R}(|\boldsymbol{a}_l(\boldsymbol{x})|, K)$, it follows that :*

$$\nabla_{\boldsymbol{A}} \ell(\boldsymbol{x}) = -\boldsymbol{b}_{l-2}(\boldsymbol{x}) \boldsymbol{v}_{goal}(\boldsymbol{x})^T \boldsymbol{\Omega} \tag{C.136}$$

*Proof:* for a input $\boldsymbol{x}$, it follows that :

$$\ell(\boldsymbol{x}) := \ell(f_{\theta \oplus (\boldsymbol{A}, \boldsymbol{\Omega})}(\boldsymbol{x})) \tag{C.137}$$

$$= \ell(\sigma_L(...(\boldsymbol{a}_l(\boldsymbol{x}) + \boldsymbol{\Omega} \sigma_{l-1}(\boldsymbol{A}^T \boldsymbol{b}_{l-2}(\boldsymbol{x}))) \tag{C.138}$$

For readability let $\boldsymbol{b} := \boldsymbol{b}_{l-2}(\boldsymbol{x})$ and $\boldsymbol{v}_{\text{goal}}{}^l(\boldsymbol{x}) := \boldsymbol{v}_{\text{goal}}$. It follows that for any $\boldsymbol{H} \in \mathbb{R}(|\boldsymbol{b}|, K)$ :

$$\ell(\boldsymbol{x}, \boldsymbol{H}) := \ell(f_{\theta \oplus (\boldsymbol{A} + \boldsymbol{H}, \boldsymbol{\Omega})}(\boldsymbol{x})) \tag{C.139}$$

$$= \ell(\sigma_L(...(\boldsymbol{a}_l(\boldsymbol{x}) + \boldsymbol{\Omega} \sigma_{l-1}(\boldsymbol{A}^T \boldsymbol{b} + \boldsymbol{H}^T \boldsymbol{b}))) \tag{C.140}$$

$$= \ell(\sigma_L(...(\boldsymbol{a}_l(\boldsymbol{x}) + \boldsymbol{\Omega} \sigma_{l-1}(\boldsymbol{A}^T \boldsymbol{b}) + \sigma'_{l-1}(\boldsymbol{A}^T \boldsymbol{b}) \boldsymbol{\Omega} \boldsymbol{H}^T \boldsymbol{b}))) \tag{C.141}$$

$$\tag{C.142}$$

As $\boldsymbol{A} = 0$, then $\sigma'_{l-1}(\boldsymbol{A}^T \boldsymbol{x}) = \sigma'_{l-1}(0)$, and it is supposed that it is equal to 1 without loss of generality.

$$\ell(\boldsymbol{x}, \boldsymbol{H}) = \ell(\sigma_L(...(\boldsymbol{a}_l(\boldsymbol{x}) + \boldsymbol{\Omega} \sigma_{l-1}(\boldsymbol{A}^T \boldsymbol{b}) + \boldsymbol{\Omega} \boldsymbol{H}^T \boldsymbol{b} + o(\|\boldsymbol{H}\|))) \tag{C.143}$$

$$= \ell(\boldsymbol{x}, 0) - \left\langle -\boldsymbol{v}_{\text{goal}}, \boldsymbol{\Omega} \boldsymbol{H}^T \boldsymbol{b} \right\rangle + o(\|\boldsymbol{H}\|) \tag{C.144}$$

Furthermore :

$$\begin{aligned}\left\langle \boldsymbol{v}_{\text{goal}}, \boldsymbol{\Omega}\boldsymbol{H}^T\boldsymbol{b}\right\rangle &= {\boldsymbol{v}_{\text{goal}}}^T\boldsymbol{\Omega}\boldsymbol{H}^T\boldsymbol{b} && \text{(C.145)}\\ &= \text{Tr}({\boldsymbol{v}_{\text{goal}}}^T\boldsymbol{\Omega}\boldsymbol{H}^T\boldsymbol{b}) && \text{(C.146)}\\ &= \text{Tr}(\boldsymbol{H}^T\boldsymbol{b}{\boldsymbol{v}_{\text{goal}}}^T\boldsymbol{\Omega}) && \text{(C.147)}\\ &= \left\langle \boldsymbol{H}, \boldsymbol{b}{\boldsymbol{v}_{\text{goal}}}^T\boldsymbol{\Omega}\right\rangle_{\text{Tr}} && \text{(C.148)}\end{aligned}$$

It follows that $\nabla_A \ell(\boldsymbol{x}) = \boldsymbol{b}{\boldsymbol{v}_{\text{goal}}}^T\boldsymbol{\Omega}$. ∎

**Lemma C.5.11.** *Consider the update of gradient descent $\boldsymbol{V}_{goal}\boldsymbol{B}^T$ and the best update $\boldsymbol{V}_{goal}\boldsymbol{B}^T\left(\boldsymbol{B}\boldsymbol{B}^T\right)^+$, then :*

$$\boldsymbol{V}_{goal}\boldsymbol{B}^T = 0 \iff \boldsymbol{V}_{goal}\boldsymbol{B}^T\left(\boldsymbol{B}\boldsymbol{B}^T\right)^+ = 0 \qquad \text{(C.149)}$$

*Proof:* The first implication $\boldsymbol{V}_{goal}\boldsymbol{B}^T = 0 \implies \boldsymbol{V}_{goal}\boldsymbol{B}^T\left(\boldsymbol{B}\boldsymbol{B}^T\right)^+ = 0$ is straightforward.
Suppose $\boldsymbol{V}_{goal}\boldsymbol{B}^T\left(\boldsymbol{B}\boldsymbol{B}^T\right)^+ = 0$. It implies that $\boldsymbol{V}_{goal}\boldsymbol{B}^T\left(\boldsymbol{B}\boldsymbol{B}^T\right)^+\boldsymbol{B} = 0$. Let $\boldsymbol{B} = \boldsymbol{U}\Sigma\boldsymbol{V}^T$ the SVD of $\boldsymbol{B}$. It follows that :

$$\begin{aligned}\left(\boldsymbol{B}\boldsymbol{B}^T\right)^+\boldsymbol{B} &= \boldsymbol{U}\Sigma^{-2}\boldsymbol{U}^T\boldsymbol{U}\Sigma\boldsymbol{V}^T && \text{(C.150)}\\ &= \boldsymbol{U}\Sigma^{-1}\boldsymbol{V}^T && \text{(C.151)}\end{aligned}$$

With the convention that $0^{-1} = 0^{-2}0$. It follows that :

$$\begin{aligned}\boldsymbol{V}_{goal}\boldsymbol{B}^T\left(\boldsymbol{B}\boldsymbol{B}^T\right)^+\boldsymbol{B} &= \boldsymbol{V}_{goal}\boldsymbol{V}\Sigma\boldsymbol{U}^T\boldsymbol{U}\Sigma^{-1}\boldsymbol{V}^T && \text{(C.152)}\\ &= \boldsymbol{V}_{goal}\boldsymbol{V}I_\Sigma\boldsymbol{V}^T && \text{(C.153)}\\ &= 0 && \text{(C.154)}\end{aligned}$$

With $I_\Sigma$ the identity matrix with a one at position $k,k$ only if $\Sigma_{k,k}$ is non-zero. On the same reasoning, let $I_{\bar{\Sigma}}$ the identity matrix with a one at position $k,k$ only if $\Sigma_{k,k}$ is zero. It follows that :

$$\begin{aligned}\boldsymbol{V}_{goal}\boldsymbol{B}^T &= \boldsymbol{V}_{goal}\boldsymbol{V}\Sigma\boldsymbol{U}^T && \text{(C.155)}\\ \boldsymbol{V}_{goal}\boldsymbol{B}^T &= \boldsymbol{V}_{goal}(\boldsymbol{V}I_\Sigma\boldsymbol{V}^T + \boldsymbol{V}I_{\bar{\Sigma}}\boldsymbol{V}^T)\boldsymbol{V}\Sigma\boldsymbol{U}^T && \text{(C.156)}\\ &= \overbrace{\boldsymbol{V}_{goal}\boldsymbol{V}I_\Sigma\boldsymbol{V}^T}^{=0}\boldsymbol{V}\Sigma\boldsymbol{U}^T + \boldsymbol{V}_{goal}\boldsymbol{V}\overbrace{I_{\bar{\Sigma}}\boldsymbol{V}^T\boldsymbol{V}\Sigma}^{=0}\boldsymbol{U}^T && \text{(C.157)}\\ &= 0 && \text{(C.158)}\end{aligned}$$

∎

## C.6 Section *About greedy growth sufficiency and TINY convergence* with more details and proofs

One might wonder whether a greedy approach on layer growth might get stuck in a non-optimal state. *Greedy* means that every neuron added has to decrease the loss. The following series of propositions are proposed in this regard. Since in this work, neurons are added layer per layer independently, here the case of a single hidden layer network is studied, to spot potential layer growth issues. For the sake of simplicity, let consider the task of least square regression towards an explicit continuous target $f^*$, defined on a compact set. That is, the objective is to minimize the loss:

$$\inf \sum_{\boldsymbol{x} \in \mathcal{D}} \|f(\boldsymbol{x}) - f^*(\boldsymbol{x})\|^2 \tag{C.159}$$

where $f(\boldsymbol{x})$ is the output of the neural network and $\mathcal{D}$ is the training set.

The section starts with an optional introductory section C.6.1 about greedy growth possibilities, then prepare lemmas in Sections C.6.2 and C.6.3 that will be used in Section C.6.4 to show that one can keep on adding neurons to a network (without modifying already existing weights) to make it converge exponentially fast towards the optimal function. Then in Section C.6.6 it is presented a growth method that explicitly overfits each dataset sample one by one, thus requiring only $n$ neurons, thanks to existing weights modification. Finally, more importantly, in Section C.6.7, it is shown that actually any reasonable growth method that follows a certain optimization protocol (this includes TINY completed by random neuron additions if necessary) will reach 0 training error in at most $n$ neuron additions.

### C.6.1 Possibility of greedy growth

**Proposition C.6.1 (Greedy completion of an existing network).** *If $f$ is not $f^*$ yet, there exists a set of neurons to add to the hidden layer such that the new function $f'$ will have a lower loss than $f$.*

One can even choose the added neurons such that the loss is arbitrarily well minimized.

*Proof:* The classic universal approximation theorem about neural networks with one hidden layer Pinkus [1999] states that for any continuous function $g^*$ defined on a compact set $\boldsymbol{\omega}$, for any desired precision $\gamma$, and for any activation function $\sigma$ provided it is not a polynomial, then there exists a neural network $g$ with one hidden layer (possibly quite large when $\gamma$ is small) and with this activation function $\sigma$, such that

$$\forall x, \|g(x) - g^*(x)\| \leqslant \gamma \tag{C.160}$$

This theorem is applied to the case where $g^* = f^* - f$, which is continuous as $f^*$ is continuous, and $f$ is a shallow neural network and as such is a composition of

linear functions and of the function $\sigma$, that is supposed to be continuous for the sake of simplicity. It is supposed that $f$ is real-valued for the sake of simplicity as well, but the result is trivially extendable to vector-valued functions (just concatenate the networks obtained for each output independently). Let $\gamma = \frac{1}{10}\|f^* - f\|_{L^2}$, where $\langle a|b\rangle_{L^2} = \frac{1}{|\boldsymbol{\omega}|}\int_{\boldsymbol{x}\in\boldsymbol{\omega}} a(\boldsymbol{x})\, b(\boldsymbol{x})\, d\boldsymbol{x}$. This way a one-hidden-layer neural network is obtained $g$ with activation function $\sigma$, and let $a(\boldsymbol{x}) = g(\boldsymbol{x}) - g^*(\boldsymbol{x})$ the error term, it follows that:

$$\forall \boldsymbol{x} \in \boldsymbol{\omega},\ \ -\gamma \leqslant g(\boldsymbol{x}) - g^*(\boldsymbol{x}) \leqslant \gamma \tag{C.161}$$

$$\forall \boldsymbol{x} \in \boldsymbol{\omega},\ \ g(\boldsymbol{x}) = f^*(\boldsymbol{x}) - f(\boldsymbol{x}) + a(\boldsymbol{x}) \tag{C.162}$$

with $\forall \boldsymbol{x} \in \boldsymbol{\omega},\ |a(\boldsymbol{x})| \leqslant \gamma$.

Then:

$$\forall \boldsymbol{x} \in \boldsymbol{\omega},\ \ f^*(\boldsymbol{x}) - (f(\boldsymbol{x}) + g(\boldsymbol{x})) = -a(\boldsymbol{x}) \tag{C.163}$$

$$\forall \boldsymbol{x} \in \boldsymbol{\omega},\ \ (f^*(\boldsymbol{x}) - h(\boldsymbol{x}))^2 = a^2(\boldsymbol{x}) \tag{C.164}$$

with $h$ being the function corresponding to a neural network consisting of concatenating the hidden layer neurons of $f$ and $g$, and consequently summing their outputs.

$$\|f^* - h\|_{L^2}^2 = \|a\|_{L^2}^2 \tag{C.165}$$

$$\|f^* - h\|_{L^2}^2 \leqslant \gamma^2 = \frac{1}{100}\|f^* - f\|_{L^2}^2 \tag{C.166}$$

and consequently the loss is reduced indeed (by a factor of 100 in this construction).

The same holds in expectation or sum over a training set, by choosing $\gamma = \frac{1}{10}\sqrt{\frac{1}{|\mathcal{D}|}\sum_{\boldsymbol{x}\in\mathcal{D}}\|f(\boldsymbol{x}) - f^*(\boldsymbol{x})\|^2}$, as Equation (C.164) then yields:

$$\sum_{\boldsymbol{x}\in\mathcal{D}} (f^*(\boldsymbol{x}) - h(\boldsymbol{x}))^2 = \sum_{\boldsymbol{x}\in\mathcal{D}} a^2(\boldsymbol{x}) \leqslant \frac{1}{100}\sum_{\boldsymbol{x}\in\mathcal{D}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}))^2 \tag{C.167}$$

which proves the proposition as stated. ∎

For more general losses, one can consider order-1 (linear) development of the loss and ask for a network $g$ that is close to (the opposite of) the gradient of the loss.

*Proof: (Proof of the additional remark)* The proof in Pinkus [1999] relies on the existence of real values $c_n$ such that the $n$-th order derivatives $\sigma^{(n)}(c_n)$ are not 0. Then, by considering appropriate values arbitrarily close to $c_n$, one can approximate the $n$-th derivative of $\sigma$ at $c_n$ and consequently the polynomial $c^n$ of order

$n$. This standard proof then concludes by density of polynomials in continuous functions.

Provided the activation function $\sigma$ is not a polynomial, these values $c_n$ can actually be chosen arbitrarily, in particular arbitrarily close to 0. This corresponds to choosing neuron input weights arbitrarily close to 0. ■

**Proposition C.6.2 (Greedy completion by one single neuron).** *If $f$ is not $f^*$ yet, there exists a neuron to add to the hidden layer such that the new function $f'$ will have a lower loss than $f$.*

*Proof:* From the previous proposition, there exists a finite set of neurons to add such that the loss will be decreased. In this particular setting of $L^2$ regression, or for more general losses if considering small function moves, this means that the function represented by this set of neurons has a strictly negative component over the gradient $g$ of the loss ($g = -2(f^* - f)$ in the case of the $L^2$ regression). That is, denoting by $a_i\sigma(\boldsymbol{W}_i \cdot \boldsymbol{x})$ these $N$ neurons:

$$\left\langle \sum_{i=1}^{N} a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x}) \;\middle|\; g \right\rangle_{L^2} = K < 0 \tag{C.168}$$

i.e.

$$\sum_{i=1}^{N} \langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\, g\rangle_{L^2} = K < 0 \tag{C.169}$$

It follows that:

$$0 > \frac{1}{N}K = \frac{1}{N}\sum_{i=1}^{N} \langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\, g\rangle_{L^2} \geq \min_{i=1}^{N} \langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\, g\rangle_{L^2} \tag{C.170}$$

Then necessarily at least one of the $N$ neurons satisfies

$$\langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\, g\rangle_{L^2} \leqslant \frac{1}{N}K < 0 \tag{C.171}$$

■

and thus decreases the loss when added to the hidden layer of the neural network representing $f$. Moreover this decrease is at least $\frac{1}{N}$ of the loss decrease resulting from the addition of all neurons.

As a consequence, there exists no situation where one would need to add many neurons simultaneously to decrease the loss: it is always feasible with a single neuron. Note that finding the optimal neuron to add is actually NP-hard [Bach, 2017], so the optimal one will not necessarily be searched for . A constructive lower bound on how much the loss can be improved will be given later in this section.

**Proposition C.6.3 (Greedy completion by one infinitesimal neuron).** *The neuron in the previous proposition can be chosen to have arbitrarily small input weights.*

*Proof:* This is straightforward, as, following a previous remark, the neurons found to collectively decrease the loss can be supposed to all have arbitrarily small input weights. ∎

This detail is important in that our approach is based on the tangent space of the function $f$ and thus manipulates infinitesimal quantities. Our optimization problem indeed relies on the linearization of the activation function by requiring the added neuron to have infinitely small input weights, to make the problem easier to solve. This proposition confirms that such neuron exists indeed.

**Correlations and higher orders.** Note that, as a matter of fact, our approach exploits linear correlations between inputs of a layer and desired output variations. It might happen that the loss is not minimized yet but there is no such correlation to exploit anymore. In that case the optimization problem (3.21) will not find neurons to add. Yet following Prop. C.6.3 there does exist a neuron with arbitrarily small input weights that can reduce the loss. This paradox can be explained by pushing further the Taylor expansion of that neuron output in terms of weight amplitude (single factor $\varepsilon$ on all of its input weights), for instance $\sigma(\varepsilon\boldsymbol{\alpha}\cdot\boldsymbol{x}) \simeq \sigma(0) + \sigma'(0)\varepsilon\boldsymbol{\alpha}\cdot\boldsymbol{x} + \frac{1}{2}\sigma''(0)\varepsilon^2(\boldsymbol{\alpha}\cdot\boldsymbol{x})^2 + O(\varepsilon^3)$. Though the linear term $\boldsymbol{\alpha}\cdot\boldsymbol{x}$ might be uncorrelated over the dataset with desired output variation $v(\boldsymbol{x})$, i.e. $\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[\boldsymbol{x}\, v(\boldsymbol{x})] = 0$, the quadratic term $(\boldsymbol{\alpha}\cdot\boldsymbol{x})^2$, or higher-order ones otherwise, might be correlated with $v(\boldsymbol{x})$. Finding neurons with such higher-order correlations can be done by increasing accordingly the power of $(\boldsymbol{\alpha}\cdot\boldsymbol{x})$ in the optimization problem (3.20). Note that one could consider other function bases than the polynomials from Taylor expansion, such as Hermite or Legendre polynomials, for their orthogonality properties. In all cases, one does not need to solve such problems exactly but just to find an approximate solution, i.e. a neuron improving the loss.

**Adding random neurons.** Another possibility to suggest additional neurons, when expressivity bottlenecks are detected but no correlation (up to order $p$) can be exploited anymore, is to add random neurons. The first $p$ order Taylor expansions will show 0 correlation with desired output variation, hence no loss improvement nor worsening, but the correlation of the $p+1$-th order will be non-0, with probability 1, in the spirit of random projections. Furthermore, in the spirit of common neural network training practice, one could consider brute force combinatorics by adding many random neurons and hoping some will be close enough to the desired direction [Frankle and Carbin, 2018]. The difference with the usual training is that one would perform such computationally costly searches only when and where relevant, exploiting all simple information first (linear correlations in each layer).

### C.6.2 Loss decreases with a line search on a quadratic energy

Let $\mathcal{L}$ be a quadratic loss over $\mathbb{R}^d$ and $g$ be a vector in $\mathbb{R}^d$. The loss $\mathcal{L}$ can be written as:

$$\mathcal{L}(g) = g^T Q g + v^T g + K \tag{C.172}$$

where $Q$ is a matrix that is supposed to be symmetric positive definite. This is to ensure that all eigenvalues of $Q$ are positive, hence modeling a local minimum without a saddle point. $v$ is a vector in $\mathbb{R}^d$ and $K$ is a real constant.

For instance, the mean square loss $\mathbb{E}_{x\in\mathcal{D}}\left[\|f(x) - f^*(x)\|_S^2\right]$, where $\mathcal{D}$ is a finite dataset of $N$ samples, $f^*$ a target function, and $S$ is a symmetric positive definite matrix used as a metric, fits these hypotheses, considering $g = (f(x_1), f(x_2), ...)$ as a vector. Indeed this loss rewrites as

$$\sum_{i=1}^{N} f(x_i)^T S f(x_i) - 2\sum_i f^{*T}(x_i) S f(x_i) + K \;=\; g^T Q\, g \;+\; v^T g \;+\; K \tag{C.173}$$

by flattening and concatenating the vectors $f(x_i)$ and considering $Q = S\otimes S\otimes S\otimes ...$ the tensor product of $N$ times the same matrix $S$, i.e. a diagonal-block matrix with $N$ identical blocks $S$. Note that for the standard regression with the $L^2$ metric, this matrix $Q$ is just the Identity.

Starting from point $g$, and given a direction $h \in \mathbb{R}^d$, the question is to perform a line search in that direction, i.e. to optimize the factor $\lambda \in \mathbb{R}$ in order to minimize $\mathcal{L}(g + \lambda h)$.

Developing that expression, it follows that:

$$\mathcal{L}(g+\lambda h) \;=\; (g+\lambda h)^T Q\,(g+\lambda h) + v^T(g+\lambda h) + K \;=\; \lambda^2 h^T Q h + \lambda(2h^T Q g + v^T h) + \mathcal{L}(g) \tag{C.174}$$

which is a second-order polynomial in $\lambda$ with a positive quadratic coefficient. Note that the linear coefficient is $h^T \nabla_g \mathcal{L}(g)$, where $\nabla_g \mathcal{L}(g) = 2Qg + v$ is the gradient of $\mathcal{L}$ at point $g$. The unique minimum of the polynomial in $\lambda$ is then:

$$\lambda^* = -\frac{1}{2}\frac{h^T \nabla_g \mathcal{L}(g)}{h^T Q h} \tag{C.175}$$

which leads to

$$\min_\lambda \mathcal{L}(g + \lambda h) = \lambda^{*2} h^T Q h + \lambda^* h^T \nabla_g \mathcal{L}(g) + \mathcal{L}(g) \tag{C.176}$$

$$= \mathcal{L}(g) - \frac{1}{4}\frac{\left(h^T \nabla_g \mathcal{L}(g)\right)^2}{h^T Q h} \tag{C.177}$$

$$= \mathcal{L}(g) - \frac{1}{4}\left\langle \frac{h}{\|h\|_Q} \middle| \nabla_g^Q \mathcal{L}(g) \right\rangle_Q^2 . \tag{C.178}$$

Thus the loss gain obtained by a line search in a direction $h$ is quadratic in the angle between that direction and the gradient of the loss, in the sense of the $Q$ norm

(and it is also quadratic in the norm of the gradient). Note that inner products with the gradient do not depend on the metric, in the sense that $\langle h \,|\, \nabla_g \mathcal{L}(g) \rangle_{L^2} = \langle h \,|\, \nabla_g^S \mathcal{L}(g) \rangle_S \quad \forall h$ for any metric $S$, i.e. any symmetric definite positive matrix $S$, associated to the norm $\|h\|_S^2 = h^T S h$ and to the gradient $\nabla_g^S \mathcal{L}(g) = S^{-1} \nabla_g^{L^2} \mathcal{L}(g)$.

In the case of a standard $L^2$ regression this boils down to:

$$\min_\lambda \|g + \lambda h\|_{L^2}^2 = \|g\|^2 - \left\langle \frac{h}{\|h\|} \,\middle|\, g \right\rangle_{L^2}^2 \tag{C.179}$$

i.e. considering $\mathcal{L}(f) := \mathbb{E}_{x \in \mathcal{D}} \left[ \|f(x) - f^*(x)\|^2 \right]$ :

$$\min_\lambda \mathcal{L}(f + \lambda h) \;=\; \mathcal{L}(f) - \left\langle \frac{h}{\|h\|} \,\middle|\, f^* - f \right\rangle_{L^2}^2 \;=\; \mathcal{L}(f) - \frac{\mathbb{E}_{x \in \mathcal{D}} \left[ (f^* - f)\, h \right]^2}{\mathbb{E}_{x \in \mathcal{D}} \left[ \|h\|^2 \right]} \,. \tag{C.180}$$

A result that is useful in the next sections.

### C.6.3 Expected loss gain with a line search in a random direction

Using Appendix C.6.2 above, the loss gain when performing a line search on a quadratic loss is quadratic in the angle $\alpha = \left\langle \frac{\boldsymbol{V}(X)}{\|\boldsymbol{V}(X)\|} \,\middle|\, \frac{\boldsymbol{V}_{\text{goal}}(X)}{\|\boldsymbol{V}_{\text{goal}}(X)\|} \right\rangle_{L^2}$ between the random search direction $\boldsymbol{V}(X)$ and the gradient $\boldsymbol{V}_{\text{goal}}(X)$.

This angle has average 0 and is of standard deviation $\frac{1}{nd}$, as described in Section 5.3.2. The loss gain is thus of the order of magnitude of $\frac{1}{d}$ in the best case (single-sample minibatch).

### C.6.4 Exponential convergence to 0 training error

Considering a regression to a target $f^*$ with the quadratic loss, the function $f$ represented by the current neural network (fully-connected, one hidden layer, with ReLU activation function) can be improved to reach 0 loss by an addition of $n$ neurons $(h_i)_{1 \leqslant i \leqslant n}$, with $n$ is the dataset size, using Zhang et al. [2017]. Unfortunately there is no guarantee that if one adds each of these neurons one by one, the loss decreases each time. It will be proved that one of these neurons does decrease the loss, and it will be quantified by how much, relying on the explicit construction in Zhang et al. [2017]. This decrease will actually be a constant factor of the loss, thus leading to exponential convergence towards the target $f^*$ on the training set.

As in the proof of Proposition C.6.2 in Appendix C.6, at least one of the added neurons satisfies that its inner product with the gradient direction is at least $1/n$. While one could consequently hope for a loss gain in $O(\frac{1}{n})$, one has to see that this decrease would be the one of a gradient step, which is multiplied by a step size $\eta$, and asks for multiple steps to be done. Instead in TINY approach, a line search is performed over the direction of the new neuron. In both cases (line search or multiple small gradient steps), one has to take into account at least order-2 changes

of the loss to compute the line search or estimate suitable $\eta$ and/or its associated number of steps. Luckily in our case of least square regression, the loss is exactly equal to its second order Taylor development, and all following computations are exact.

Consider the mean square regression loss $\mathcal{L}(f) = \mathbb{E}_{x\in\mathcal{D}}\left[\|f(x) - f^*(x)\|_S^2\right]$, where $\mathcal{D}$ is a finite training dataset of $N$ samples. Its functional gradient $\nabla\mathcal{L}(f)$ at point $f$ is $2(f - f^*)$, which is proportional to the optimal change to add to $f$, that is, $f^* - f$. The $n$ neurons $(h_i)_{1\leqslant i\leqslant n}$ to be added to $f$ following Zhang et al. [2017] satisfy $\sum_i h_i = f^* - f = -\frac{1}{2}\nabla\mathcal{L}(f)$. Thus

$$\left\langle \sum_i h_i \middle| f^* - f \right\rangle_{L^2} = \|f^* - f\|_{L^2}^2 = \mathcal{L}(f)\,. \tag{C.181}$$

Then like in the proof of C.6.2 it is used that the maximum is greater or equal to the mean to get that there exists a neuron $h_i$ that satisfies:

$$\langle h_i|\ f^* - f\rangle_{L^2} \geqslant \mathcal{L}(f)/n\,. \tag{C.182}$$

By applying Appendix C.6.2 one obtains that the new loss after line search into the direction of $h_i$ yields:

$$\min_\lambda \mathcal{L}(f + \lambda h_i) = \mathcal{L}(f) - \frac{\langle h_i \mid f^* - f\rangle_{L^2}^2}{\|h_i\|^2} \leqslant \mathcal{L}(f) \times \left(1 - \frac{\mathcal{L}(f)}{n^2\|h_i\|^2}\right)\,. \tag{C.183}$$

From the particular construction in Zhang et al. [2017] it is possible to bound the square norm of the neuron $\|h_i\|^2$ by $n\,d'\left(\frac{d_M}{d_m}\right)^2\mathcal{L}(f)$, where $d_M$ is related to the maximum distance between 2 points in the dataset, $d_m$ is another geometric quantity related to the minimum distance, and $d'$ is the network output dimension. To ease the reading of this proof, the construction of this bound is deferred to the next section, Appendix C.6.5.

Then the loss at each neuron addition decreases by a factor which is at least $\gamma = 1 - \frac{1}{n^3 d'}\left(\frac{d_m}{d_M}\right)^2 < 1$. This factor is a constant, as it is a bound that depends only on the geometry of the dataset (not on $f$).

Thus it is possible to decrease the loss exponentially fast with the number $t$ of added neurons, i.e. $\mathcal{L}(f_t) \leqslant \gamma^t\mathcal{L}(f)$, towards 0 training loss, and this in a greedy way, that is, by adding neuron one by one, with the property that each neuron addition decreases the loss.

Note that, in the proof of Zhang et al. [2017], the added neurons could be chosen to have arbitrarily small input weights. This corresponds to choosing $a$ with small norm instead of unit norm in Equation C.184.

The number of neuron additions expected to reach good performance according to this bound is in the order of magnitude of $n^3$, which is to be compared to $n$ (number of neurons needed to overfit the dataset, without the constraint that

each addition decreases the loss). This bound might be improved using other constructions than Zhang et al. [2017], though with this proof the bound cannot be better than $n^2$ (supposing $\|h_i\|$ can be made not to depend on $n$).

Note also that with ReLU activation functions, all points that are on the convex hull of the dataset (which is necessarily the case of all points if the input dimension is higher that the number of points) can easily in turn be perfectly predicted (0 loss) by just one neuron addition each (without changing the outputs for the other points), by choosing an hyperplane that separates the current convex hull point from the rest of the dataset, and setting a ReLU neuron in that direction.

### C.6.5 Bound on the norm of the neurons

Here it is proved that the neurons obtained by Zhang et al. [2017] can be chosen so as to bound the square norm of any neuron $\|h_i\|^2$ by $n\,d'\left(\frac{d_M}{d_m}\right)^2\mathcal{L}(f)$, where $d_M$ is related to the maximum distance between 2 points in the dataset, and $d_m$ is another geometric quantity related to the minimum distance. For the sake of simplicity, let first consider the case where the output dimension is $d'=1$.

In Zhang et al. [2017], the $n$ neurons are obtained by solving $y=Aw$, where $y=(y_1,y_2...)$ is the target function (here $(f^*-f)$ at each $x_j$), $A$ is the matrix given by $A_{jk}=\mathrm{ReLU}(a\cdot x_j-b_k)$, representing neuron activations, and $a$ is any vector that separates the dataset points, i.e. $a\cdot x_j\neq a\cdot x_{j'}\ \forall j\neq j'$, that is, $a$ could be almost any vector in $\mathbb{R}^d$ (in the sense of random projections, that is, the set of vectors that do not satisfy this is of measure 0).

Here a particular unit direction $a$ is picked, one that maximizes the distance between any two samples after projection:

$$a\in\arg\max_{\|a\|=1}\ \min_{j,j'}|a\cdot(x_j-x_{j'})| \tag{C.184}$$

and let us denote $d'_m$ the associated value: $d'_m=\min_{j,j'}|a\cdot(x_j-x_{j'})|$ for that $a$. Note that $d'_m\leqslant\min_{j,j'}\|x_j-x_{j'}\|$ and that it depends only on the training set. The quantity $d'_m$ is likely to be also lower-bounded (over all possible datasets) by $\min_{j,j'}\|x_j-x_{j'}\|$ times a factor depending on the embedding dimension $d$ and the number of points $n$.

Now, let us sort the samples according to increasing $a\cdot x_j$, that is, let us re-index the samples such that $(a\cdot x_j)$ now grows with $j$. By definition of $a$, the difference between any two consecutive $a\cdot x_j$ is at least $d'_m$.

Choose biases $b_j=a\cdot x_j-d'_m+\varepsilon$ for some very small $\varepsilon$. The neurons are then defined as $h_k(x)=w_k\mathrm{ReLU}(a\cdot x-b_k)$. The induced activation matrix $A_{jk}=\mathrm{ReLU}(a\cdot x_j-b_k)$ then satisfies $\forall j<k; A_{jk}=0$ and $\forall j\geqslant k; A_{jk}\geqslant d'_m-\varepsilon$. The matrix $A$ is lower triangular with diagonal elements above $d_m:=d'_m-\varepsilon$, hence invertible. Recall that $y=Aw$.

Consequently, $w=A^{-1}y$, and hence $\|w\|^2\leqslant|||A^{-1}|||^2\,\|y\|^2$, that is,

$$\|w\|^2\leqslant\frac{1}{d_m^2}\,\mathcal{L}(f) \tag{C.185}$$

as the target $y$ is the vector $f^* - f$ in our case. Consequently, for any neuron $h_i$, one has:

$$w_i^2 \leqslant \frac{1}{d_m^2} \mathcal{L}(f) \,. \tag{C.186}$$

As the norm of the neuron is $\|h_i\|^2 = w_i^2 \sum_j A_{ji}^2$, one still has to bound the activities $A_{ji} = \mathrm{ReLU}(a \cdot x_j - b_i)$. As $a$ was chosen a unit direction, the values $a \cdot x_j$ span a domain smaller than the diameter of the dataset $\mathcal{D}$: $|a \cdot (x_j - x_{j'})| \leqslant \|x_j - x_{j'}\| \leqslant \mathrm{diam}(\mathcal{D})\ \forall j, j'$. Hence all values $\forall i, j, |A_{ij}| = |a \cdot x_i - b_j| = |a \cdot x_i - a \cdot x_j + d_m| < d_M := \mathrm{diam}(\mathcal{D}) + d_m$. Note that $d_M$ depends only the dataset geometry, as for $d_m$.

It follows that :

$$\|h_i\|^2 = w_i^2 \sum_j A_{ji}^2 \leqslant n \frac{d_M^2}{d_m^2} \mathcal{L}(f) \tag{C.187}$$

which ends the proof.

For higher output dimensions $d'$, one vector $w$ of output weights is estimated per dimension, independently, leading to the same bound for each dimension. The square norms of neurons are summed over all dimensions and thus multiplied by at most $d'$.

### C.6.6 Reaching 0 training error in $n$ neuron additions by overfitting each dataset sample in turn

If one allows updating already existing output weights at the same time as one adds new neurons, then it is possible to reach 0 training error in only $n$ steps (where $n$ is the size of the dataset) while decreasing the loss at each addition.

This scenario is closer to the one that is considered with TINY, as the optimal update of existing weights is computed inside the layer, as a byproduct of new neuron estimation, and apply them.

However the existence proof here follows a very different way to create new neurons, tailored to obtain a constructive proof, and inspired by the previous section. See Appendix C.6.7 for another, more generic proof, applicable to a wide range of growth methods.

Here one can consider the same approach as in Appendix C.6.5 above, but introducing neurons one by one instead of $n$ neurons at once. After computing $a$ and the biases $b_j$, thus forming the activity matrix $A$, only the last neuron $h_n$ is added. The activity of this neuron is 0 for all input samples $x_j$ except for the last one, for which it is $A_{nn} > 0$. Thus, the neuron $h_n$ separates the sample $x_n$ from the rest of the dataset, and it is easy to find $w_n$ so that the loss gets to 0 on that training sample, without changing the outputs for other samples.

Similarly, one can then add neuron $h_{n-1}$, which is active only for samples $x_{n-1}$ and $x_n$. However designing $w_{n-1}$ so that the loss becomes 0 at point $x_{n-1}$ disturbs the output for point $x_n$ (and for that point only). Luckily if one allows updating $w_n$ then there exists a (unique) solution $(w_{n-1}, w_n)$ to achieve 0 loss at both points. This is done exactly as previously, by solving $y = Aw$, but considering only the

last 2 lines and rows of $A$, leading to a smaller $2 \times 2$ system which is also lower-triangular with positive diagonal.

Proceeding iteratively this way adds neuron one by one in a way that sends each time one more sample to 0 loss. Thus adding $n$ neurons is sufficient to achieve 0 loss on the full training set, and this in a way that each time decreases the loss.

Note that updating existing output weights $w_i$ while adding a new neuron, to decrease optimally the loss, is actually what TINY does. However, the construction in this Appendix completely overfits each sample in turn, by design, without being able to generalize to new test points. On the opposite, TINY exploits correlations over the whole dataset to extract the main tendencies.

### C.6.7 TINY reaches 0 training error in $n$ neuron additions

It is now shown that the TINY approach, as well as any other suitable greedy growth method, implemented within the right optimization procedure, reaches 0 training error in at most $n$ steps (where $n$ is the size of the dataset), almost surely.

Before stating it formally, the optimization protocol is introduced, growth completion and a probability measure over activation functions.

**Optimization protocol.** For this let consider the following optimization protocol conditions, that has to be applied at least during the last, $n$-th addition step:

- a **full batch** approach,
- when adding new neurons, also compute and add the **optimal moves of already existing parameters** (i.e. of output weights $w$).

The first point is to ensure that all dataset samples will be taken into account in the loss during the $n$-th update. Otherwise, for instance if using minibatches instead, the optimization of output weights $w$ will not be able to overfit the training loss.

The second point is to make sure that, after update, the output weights $w$ will be optimal for the training loss. Note that in the mean square regression case, this is easy to do, as the loss is quadratic in $w$: the optimal move (leading to the global optimum $f^*$) can be obtained by line search over the natural gradient (which is obtained for free as a by-product of TINY's projection of $\boldsymbol{V}_{\text{goal}}$, and is proportional to $f^* - f$). This is precisely what TINY does in practice when training networks (except when comparing with other methods and using their own protocol).

**Growth completion.** For this proof to make sense, the growth method needs to be able to perform $n$ neuron additions, if it has not reached 0 training loss before. A counter-example would be a growth method that gets stuck at a place where the training loss is not 0 while being unable to propose new neuron to add. In the case of TINY, this can happen when no correlation between inputs $x_i$ and desired output variations $f^*(x_i) - f(x_i)$ can be found anymore. To prevent this,

one can choose any auxiliary method to add neurons in such cases, for instance random directions, solutions of higher-order expressivity bottleneck formulations using further developments of the activation function, or locally optimal neurons found by gradient descent. Some auxiliary methods are guaranteed to further decrease the loss by a neuron addition (cf. Appendices C.6.2, C.6.3, C.6.4), while any other one is guaranteed not to increase the loss if combined with a line search along that neuron direction.

Let name *completed-TINY* the completion of TINY by any such auxiliary method.

**Activation function.** For technical reasons, the result will stand *almost surely* only, depending on the invertibility of a certain matrix, namely, the activation matrix $A$, defined as $A_{ij} = \sigma(\boldsymbol{v}_j \cdot \boldsymbol{x}_i + b_j)$, indexed by samples $i$ and neurons $j$.

Generally speaking, kernels induced by neurons $k_j : \boldsymbol{x} \mapsto \sigma(\boldsymbol{v}_j \cdot \boldsymbol{x} + b_j)$ form free families, in the sense that they are linearly independent (to the notable exception of the linear kernel). This linear independency means that a linear combination of kernels cannot be equal, as a function, to another kernel with different parameters. Equality is to be understood as *for all possible points* $\boldsymbol{x}$ *ever*. However here the functions will only be evaluated at a finite number $n$ of points (the dataset samples), therefore linear independence will be considered among the rows of the activation matrix $A$. This notion of linear dependence is much weaker: kernels might form a free family as functions but be linearly dependent once restricted to the dataset samples, by mere chance. While this is not likely (over dataset samples), this is not impossible in general (though of measure 0), and it is difficult to express an explicit, simple condition on the activation function to be sure that the activation matrix $A$ is *always* invertible (up to slight changes of parameters). Thus instead results will be expressed *almost surely* over activation functions and neuron parameters.

For most activation functions in the space of smooth functions, the activation matrix $A$ will be invertible almost surely over all possible datasets. In the unlucky case where the matrix is not invertible, an infinitesimal move of the neurons' parameters will be sufficient to make it invertible. For some activation functions, however, such as linear or piecewise-linear ones (e.g., ReLU), the matrix might remain non-invertible over a wide range of parameter variations (unless further assumptions are made on the neurons added by the growth process). Yet, in such cases, slight perturbations of the activation function (i.e., choosing another, smooth, activation function, arbitrarily close to the original one) will yield invertibility.

To properly define "*almost surely*" regarding activation functions, let us restrict the activation function $\sigma$ to belong to the space $\mathfrak{P}$ of polynomials of order at least $n^2$, that is:

$$\sigma(x) \;=\; \sum_{k=0}^{K} \gamma_k \, x^k \tag{C.188}$$

with $n^2 \leqslant K < +\infty$, and non-0 highest-order amplitude $\gamma_K \neq 0$. This set $\mathfrak{P}$ is dense in the set of all continuous functions over the set $\Omega = [-r_M, r_M]^d$ which is a hypercube of sufficient radius $r_M$ to cover all samples from the given dataset. One can define probability distributions over $\mathfrak{P}$, for instance consider the density $p(\sigma) = \frac{\alpha}{K^2} \prod_{k=0}^{K} \frac{e^{-\gamma_k^2}}{\sqrt{2\pi}}$ with a factor $\alpha = \left( \frac{\pi^2}{6} - \sum_{k<n^2} \frac{1}{k^2} \right)^{-1}$ to normalize the distribution, and where $K$ is the order of the polynomial and thus depends on $\sigma$. This density is continuous in the space of parameters $\gamma_k$ (though not continuous in the usual functional metric spaces). Note that the decomposition of any $\sigma \in \mathfrak{P}$ as a finite-order polynomial is unique, as monomials of different orders are linearly independent.

Let now state the following lemma (that is proved later):

**Lemma C.6.4 (Invertibility of the activation matrix).** *Let $\mathcal{D} = \{x_i, 1 \leqslant i \leqslant n\}$ be a dataset of $n$ distinct points, and let $\sigma : \mathbb{R} \to \mathbb{R}$ be a function in $\mathfrak{P}$, that is, a polynomial of order at least $n^2$. Then with probability 1 over function and neuron parameters $(\gamma_k)$, $(\boldsymbol{v}_j)$ and $(b_j)$, the activity matrix $A$ defined by $A_{ij} = \sigma(\boldsymbol{v}_j \cdot \boldsymbol{x}_i + b_j)$ is full rank.*

and the following proposition:

**Proposition C.6.5 (Reaching 0 training error in at most $n$ neuron additions).** *Under the assumptions above (polynomial activation function of order $\geqslant n^2$, full-batch optimization and computation of the optimal moves of already existing parameters), completed-TINY reaches 0 training error in at most $n$ neuron additions almost surely.*

*Proof:* If the growth method reaches 0 training error before $n$ neuron additions, the proof is done. Otherwise, let us consider the $n$-th neuron addition. It will be shown in Lemma C.6.4 that the activity matrix $A$, defined by $A_{ij} = \sigma(\boldsymbol{v}_j \cdot \boldsymbol{x}_i + b_j)$, indexed by samples $i$ and neurons $j$, is invertible. Then there exists a unique $\boldsymbol{w} \in \mathbb{R}^n$ such that $A\boldsymbol{w} = f^*$, i.e. $\sum_j w_j \sigma(\boldsymbol{v}_j \cdot \boldsymbol{x}_i + b_j) = f^*(\boldsymbol{x}_i)$ for each point $\boldsymbol{x}_i$ of the dataset. This vector of output parameters $\boldsymbol{w}$ realizes the global minimum of the loss over already existing weights: $\inf_{\boldsymbol{w}} \mathcal{L}(f_{\boldsymbol{v},\boldsymbol{w}}) = \inf_{\boldsymbol{w}} \|A\boldsymbol{w} - f^*\|^2$. They are also the ones found by a natural gradient step over the loss (up to a factor 2, that can easily be found by line search as the loss is convex). Then after that update the training loss is exactly 0. ∎

Note: piecewise-linear activation functions such as ReLU are not covered by this proposition. However the result might still hold with further assumptions over the growth process. For instance, with the method in Zhang et al. [2017], the ReLU neurons are chosen in such a way that the matrix $A$ is full rank by construction.

*Proof: (Proof of Lemma C.6.4)* Let us first show that if, unluckily, for a given activation function $\sigma$ and given parameters $(\boldsymbol{v}_j, b_j)$, the matrix $A$ is not full rank, then upon infinitesimal variation of the parameters, the matrix $A$ becomes full rank.

Indeed, if all pre-activities $a_{i,j} := \boldsymbol{v}_j \cdot \boldsymbol{x}_i + b_j$ are not distinct for all $i, j$, then an infinitesimal variation of the vectors $\boldsymbol{v}_j$ can make them distinct. For this, one can see that the set of directions $\boldsymbol{v}_j$ on which any two dataset points $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i'}$ have the same projection is finite (since it has to be the direction of $\boldsymbol{x}_i - \boldsymbol{x}_{i'}$, for a given pair of dataset samples $(i, i')$) and thus of measure 0. As a consequence with probability 1 over neuron parameters $\boldsymbol{v}_j$ and $b_j$, all pre-activities are distinct.

Now, if the matrix $A$ is not invertible, as invertible matrices are dense in the space of matrices, one can easily find an infinitesimal change $\delta A$ to apply to $A$ to make it invertible. This corresponds to changing the activation function $\sigma$ accordingly at each of the $n^2$ distinct pre-activity values. Since $\sigma$ has more than $n^2$ parameters, this is doable. For instance, one can select the $n^2$ first parameters and search for a suitable variation $\boldsymbol{g} := (\delta\gamma_k)_{0 \leqslant k < n^2}$ of them by solving the linear system $S\,\boldsymbol{g} = \delta A$ where the $n^2 \times n^2$ matrix $S$ is defined by $S_{ij,k} = a_{i,j}^k = (\boldsymbol{v}_j \cdot \boldsymbol{x}_i + b_j)^k$. This matrix $S$ is invertible because any $\boldsymbol{g}$ such that $S\,\boldsymbol{g} = 0$ would induce:

$$\forall i, j, \ \sum_{k=0}^{n^2-1} \delta\gamma_k \; a_{i,j}^k = 0 \tag{C.189}$$

and thus the polynomial $P(x) = \sum_{k=0}^{n^2-1} \delta\gamma_k \; x^k$ has at least $n^2$ roots while being of order at most $n^2 - 1$. Thus $S\,\boldsymbol{g} = 0 \implies \boldsymbol{g} = 0$ and $S$ is invertible. Note that as $\delta A$ is infinitesimal, $\boldsymbol{g} = S^{-1}\,\delta A$ will be infinitesimal as well, and so is the change brought to the activation function $\sigma$.

Consequently, the set of activation functions $\sigma$ and neuron parameters $(\boldsymbol{v}_j, b_j)$ for which the matrix $A$ is full rank is dense in the set of polynomials $\mathfrak{P}$ of order at least $n^2$ and of neuron parameters $\mathfrak{N}$.

Now, the function $\det : \mathfrak{P} \times \mathfrak{N} \to \mathbb{R}$, $((\gamma_k)_k, (\boldsymbol{v}_j, b_j)_j) \mapsto \det A = \det\left(\sigma_\gamma(\boldsymbol{v}_j \cdot \boldsymbol{x}_i + b_j)\right)$ is smooth as a function of its input parameters (the determinant being a polynomial function of the matrix coefficients). As this continuous function is non-0 on a dense set of its inputs, the pre-image $\det^{-1}\{0\}$ is closed and contains no open subset. This is not yet sufficient to prove that this pre-image is of measure 0 (e.g., fat Cantor set).

For a fixed order $K$, one can see this function as a polynomial of its inputs $\gamma_k$ and $\boldsymbol{v}_j, b_j$, and conclude[1] that the set of its roots is of measure 0. As a consequence, the probability, over coefficients $\gamma_k$ or equivalently over polynomials $\sigma$ of order $K$, that $\det A$ is non-0, is 1. As this stands for all $K$, it follows that the probability

[1] See for instance a proof by recurrence that roots of a polynomial are always of measure 0: https://math.stackexchange.com/questions/1920302/the-lebesgue-measure-of-zero-set-of-a-polynomial-function-is-zero .

that the matrix $A$ is invertible is at least the mass of polynomials of all orders $K$, i.e. $\sum_{k \geqslant n^2} \frac{\alpha}{k^2} = 1$. Thus $A$ is invertible with probability 1. ∎

# D module description and technical details

## D.1 module description

This section presents the class TINY, a Python module that encodes a neural network whose architecture is to be expanded using Chapter 3.

## D.2 folder description

```
TINYpub/
├── TINY.py
├── SOLVE_EB.py
├── T_S_F_N.py
├── UTILS.py
├── GLOBALS.py
├── define_device.py
├── Loader_Data_Loader.py
├── mes_imports.py
└── settings
```

Figure D.1: Folder TINYpub

The repository can be found in https://gitlab.inria.fr/mverbock/tinypub and is encoded in Python using PyTorch as a basis. It contains three folders: 1) TINYpub, which contains the class and the theory; 2) DEMO, which contains notebooks that grows networks on academic datasets; and 3) Paper, which regroups experiments comparing TINY with classic training methods.

The class TINY is in `TINY.py` and has all the attributes of the class. The file `T_S_F_N.py` computes the matrices $\boldsymbol{S}$ and $\boldsymbol{N}$ of theorem 3.2.3. The file `SOLVE_EB.py` computes the best update and the new neurons using the matrix $\boldsymbol{S}$ and $\boldsymbol{N}$. The files `define_device.py`, `Loader_Data_Loader.py` and `mes_imports.py`, respectively, define the type of device (CPU or GPU), load the data and import the python packages.

The TINY class, as defined in algorithm 12, has three types of parameters. The first type of parameters defines the training setting and is explicit. The second category describes the starting architecture of the network with :

- `skeleton[i]`: the number of neurons at layer i.
- `layer_name[i]` : being L for a linear layer, C for a convolutional layer, and CB for a convolutional followed by a batch-norm layer.
- `skip_connection[s]` : the list of layers which are the starting point of a skip connection for s = 'in' or the ending point of a skip connection for s = 'out'
- `activation_functions[i]` : the activation function of layer i.
- `skip_functions[i]` : the activation function for the skip connection that starts at layer i.

The third category is the hyperparameters of the architecture growth :

- `T_j_depth` : the list of indexes $j$ at which to compute the $\boldsymbol{T}_j$ matrices of C.2.2 (only for convolutional layers).
- `lambda_method` : the amplitude factor of the new neurons, if set to 0, a line search will be performed to find the best amplitude factor.
- `lambda_method_NG` : the amplitude factor of the best update, if set to 0, a line search will be performed to find the best amplitude factor.
- `init_deplacement` : it is used when lambda_method = 0 and is the lowest value tested as amplitude factor when performing the line search on the new neurons.
- `init_deplacement_NG` : it is used when lambda_method_NG = 0 and is the lowest value tested as an amplitude factor when performing the line search on the best update.

**Algorithm 12:** TINY class

```
Class TINY(torch.nn.Module):
    def __init__(self ;
        // Training parameters
        batch_size: int = 128 ;
        lr: float = 1e-2 ;
        gradient_clip: Optional[float] = None ;
        scheduler: Optional[callable] = None ;
        len_train_dataset: int = 50000 ;
        len_test_dataset: int = 10000 ;
        loss: torch.nn.modules.loss = torch.nn.MSELoss() ;
        // Starting architecture
        skeleton: dict = None ;
        layer_name: dict = None ;
        activation_function: dict[str, torch.nn.Module] = None ;
        skip_connections: dict[str, tuple[int, int]] = None ;
        skip_functions: dict[int, torch.nn.Module] = None ;
        init_input_x_shape: tuple[int, int, int] = (3, 32, 32) ;
        // Expending the architecture
        T_j_depth: Optional[list[int]] = None;
        lambda_method: float = 0.;
        lambda_method_NG: float = 0.;
        init_deplacement: float = 1e-8;
        init_deplacement_NG: Optional[float] = None;
        accroissement_decay: float = 1e-3;
        accroissement_decay_NG: Optional[float] = None;
        exp: int = 2;
        ind_lmbda_shape: int = 1000;
        max_amplitude: float = 1.;
        rescale: str = 'DE';
        architecture_growth: str = 'Our';
        selection_neuron: callable = UTILS.selection_neuron_seuil;
        selection_NG: callable = UTILS.selection_NG_Id;
        ):
```

## D.3 Technical details of Figures 5.6 and 5.9

### D.3.1 settings and strategy of adding

The experiments were performed on 1 GPU. The optimizer is SGD($lr = 1e-2$) with the starting batch size 32 D.4.1. At each depth $l$, the number of neurons $n_l$ to be added at this depth is defined in D.1. These numbers do not depend on the starting architecture and have been chosen such that, for a given starting architecture, each depth will reach its final width with the same number of layer extensions. For the initial structure $s = 1/4$, resp. $1/64$, the number of layer extensions is set to 16, resp. 21, such that at depth 2 (named Conv2 in Table D.2), $n_2 = (\text{Size}_2^{final} - \text{Size}_2^{start})/\text{nb of layer extensions} = (64-16)/16 = (64-1)/21 = 3$. The initial architecture is described in Table D.2.

| depth $l$ | Conv2 | Conv3 | Conv5 | Conv6 | Conv8 | Conv9 | Conv11 | Conv12 |
|---|---|---|---|---|---|---|---|---|
| $n_l$ | 3 | 3 | 6 | 6 | 12 | 12 | 24 | 24 |

Table D.1: Number of neurons to add per layer. The depth is identified by its name on Table D.2.

## D.4 Batch size to estimate the new neuron and the best update

This section studies the variance of the matrices $\delta \mathbf{W}_l{}^*$and $\boldsymbol{S}^{-1}\boldsymbol{N}$ computed using a minibatch of $n$ samples, seeing the samples as random variables, and the matrices computed as estimators of the true matrices one would obtain by considering the full distribution of samples. Those two matrices are the solutions of the multiple linear regression problems defined in Equation (3.17) and in Equation (3.27), as we are trying to regress the desired update noted $Y$ onto the span of the activities noted $X$. The following setting is supposed :

$$Y \sim \boldsymbol{A}X + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2), \qquad \mathbb{E}[\varepsilon|X] = 0 \tag{D.1}$$

where the $(X_i, Y_i)$ are *i.i.d.* and $\boldsymbol{A}$ is the oracle for $\delta \mathbf{W}_l{}^*$ or matrix $\boldsymbol{S}^{-1}\boldsymbol{N}$. If $Y$ is multidimensional, the total variance of our estimator can be seen as the sum of the variances of the estimator on each dimension of $Y$.

It is now supposed that $Y \in \mathbb{R}$ and note $\hat{A} := \boldsymbol{Y}\boldsymbol{X}^T(\mathbf{X}\mathbf{X}^T)^+$ the solution of D.1. First, one can remark that $\hat{\boldsymbol{A}}\boldsymbol{X} = \boldsymbol{Y}\boldsymbol{P}$ with $\boldsymbol{P} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T)^+\boldsymbol{X} \in \mathbb{R}(n, n)$ . It follows that when $n \leq p$, almost surely it follows that $rk(\boldsymbol{P}) = n$ and $\boldsymbol{Y}\boldsymbol{P} = \boldsymbol{Y}$, resulting in a zero expressivity bottleneck for that specific mini-batch, *i.e.* $\boldsymbol{Y} = \hat{\boldsymbol{A}}\boldsymbol{X}$. In practice, $n < p$ is not considered as the solution ($\delta \boldsymbol{W}$ or $\boldsymbol{A}, \boldsymbol{\Omega}$)

would overfit a specific mini-batch and would increase the expressivity bottleneck for the rest of the dataset.

It is now supposed that $n > p$, almost surely it follows that $rk(\boldsymbol{P}) = p$ and $\boldsymbol{YP} \neq \boldsymbol{Y}$. For the variance of the estimator $\hat{\boldsymbol{A}} \in \mathbb{R}^p$, almost surely it follows that $\boldsymbol{XX}^T$ is invertible and its inverse is noted $(\boldsymbol{XX})^{-1}$. Taking the expectation on variable $\varepsilon$,

It follows that $\text{cov}(\hat{\boldsymbol{A}}) = \sigma^2(\mathbf{X}\mathbf{X}^T)^{-1}$. If $n$ is large, and if matrix $\frac{1}{n}\mathbf{X}\mathbf{X}^T \to \boldsymbol{Q}$, with $\boldsymbol{Q}$ non-singular, then, asymptotically, it follows that $\hat{\boldsymbol{A}} \sim \mathcal{N}(\boldsymbol{A}, \sigma^2\frac{\boldsymbol{Q}^{-1}}{n})$, which is equivalent to $(\hat{\boldsymbol{A}} - \boldsymbol{A})\frac{\sqrt{n}}{\sigma}\boldsymbol{Q}^{1/2} \sim \mathcal{N}(0, I)$. Then $||(\hat{\boldsymbol{A}} - \boldsymbol{A})\frac{\sqrt{n}}{\sigma}\boldsymbol{Q}^{1/2}||^2 \sim \chi^2(k)$ where $k$ is the dimension of $X$. It follows that $\mathbb{E}\left[||(\hat{\boldsymbol{A}} - \boldsymbol{A})\boldsymbol{Q}^{1/2}||^2\right] = \frac{k\sigma^2}{n}$ and as $\boldsymbol{Q}^{1/2}\boldsymbol{Q}^{1/2^T}$ is positive definite, as a conclusion $\text{(}\hat{\boldsymbol{A}}) \leqslant \frac{k\sigma^2}{n\lambda_{\min}(\boldsymbol{Q})}$.

In practice, one can aim at keeping the variance of our estimators stable during architecture growth. To ensure this, the batch size $n$ can be chosen to make the bound constant. With the notations defined in Figure 5.1, one can estimate a matrix of size $k \leftarrow (SW)^2$. For $n$ images, as each input sample contains $P$ quantities, and that each is a realization of the random variable $\boldsymbol{X}$ (total $nP$ variables), in total $n \leftarrow nP$ data points for the estimation of the best neuron. Hence to add new neurons with a (asymptotically) fixed variance, batch size

$$n \propto \frac{(SW)^2}{P}$$

is used.

For convolutional layers, $n = 0.001 \times \frac{(SW)^2}{P} \times 2^k$ (Figure 5.6) and $n = 0.01 \times \frac{(SW)^2}{P} \times 2^k$ (Figure 5.9) is chosen, where $k$ is equal to $\sqrt{\frac{32\times32}{P}}$, and this $2^k$ factor is found empirically to somehow account for the variances of the estimators even when the same input is used multiple times, as are the $\{\boldsymbol{B}^t_{i,j}\}_{j\in P}$ in Equation (C.23).

### D.4.1 Batch size for learning

The batch size for gradient descent is adjusted as follows: the batch size is set to $b_{t=0} = 32$ at the beginning of each experiment, and it is scheduled to increase as the square root of the complexity of the model (*i.e.* number of parameters). If at time $t$ the network has complexity $C_t$ parameters, then at time $t+1$ the training batch size is equal to $b_{t+1} = b_t \times \sqrt{\frac{C_{t+1}}{C_t}}$.

### D.4.2 Normalization for 5.6,5.7 and D.4

For the GradMax method of Figures 5.6 and D.4, before adding the new neurons to the architecture, the outgoing weight of the new neurons are normalized

according to Evci et al. [2022], *i.e.* :

$$\alpha_k^* \leftarrow 0 \tag{D.2}$$

$$\text{for Figures 5.6 and 5.7} \quad \omega_k^* \leftarrow \omega_k^* \times \frac{10^{-3}}{\sqrt{||(\omega_j^*)_{j=1}^{n_d}||_2^2/n_d}} \tag{D.3}$$

$$\text{for Figure D.4} \quad \omega_k^* \leftarrow \omega_k^* \times \sqrt{\frac{10^{-3}}{||(\omega_j^*)_{j=1}^{n_d}||_2^2/n_d}} \tag{D.4}$$

For TINY method of both figures, the previous normalization process is mimicked by normalizing the in and out going weights by their norms and multiplying them by $\sqrt{10^{-3}}$, *i.e.* :

$$\alpha_k \leftarrow \alpha_k^* \times \sqrt{\frac{10^{-3}}{||(\alpha_j^*)_{j=1}^{n_d}||_2^2/n_d}} \tag{D.5}$$

$$\omega_k \leftarrow \omega_k^* \times \sqrt{\frac{10^{-3}}{||(\omega_j^*)_{j=1}^{n_d}||_2^2/n_d}} \tag{D.6}$$

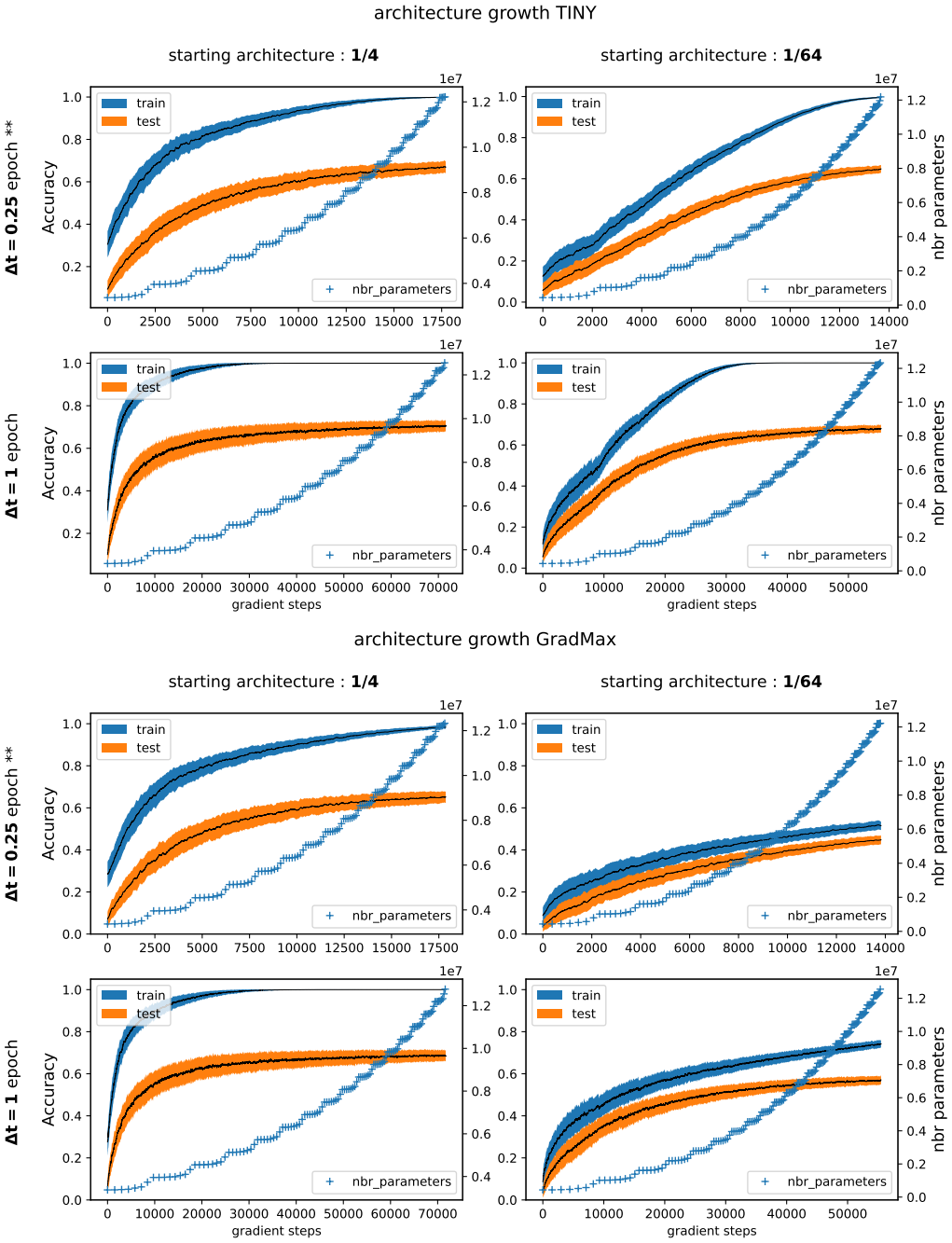



Figure D.2: Accuracy and number of parameters during architecture growth for methods TINY and GradMax as a function of the gradient step. Mean and standard deviation for four independent runs.

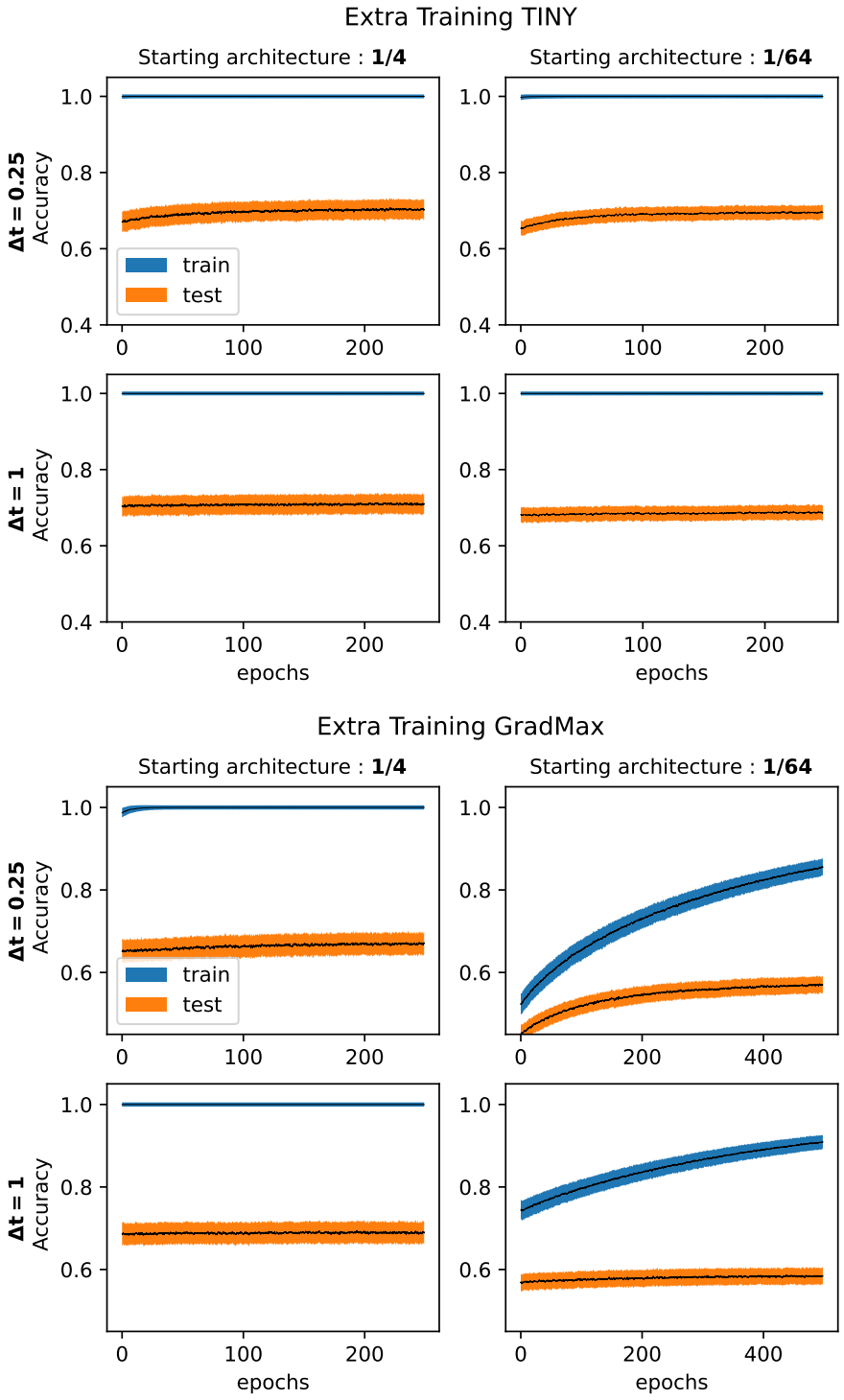



Figure D.3: Accuracy curves as a function of the number of epochs during extra training for TINY (top plot) and GradMax (bottom plot) on four independent runs.

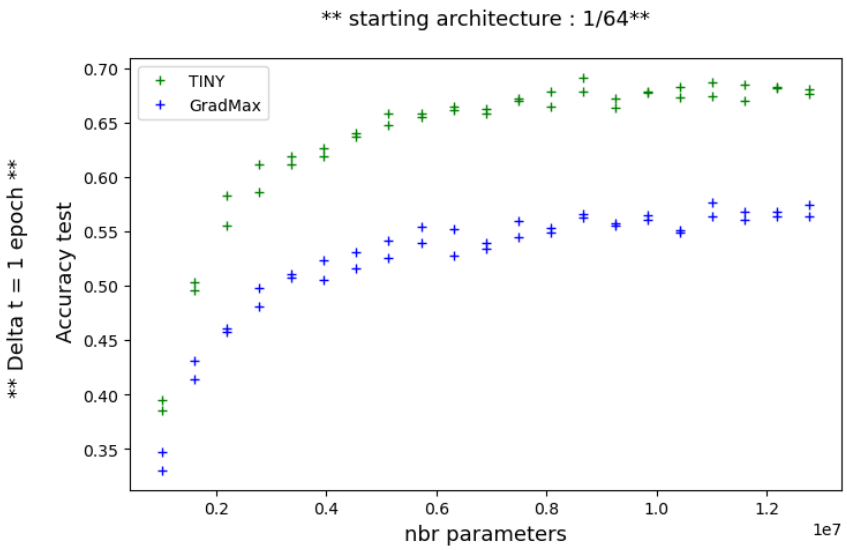



Figure D.4: Accuracy on test split of as a function of the number of parameters during architecture growth from $\text{ResNet}_{1/64}$ to ResNet18. **The normalization for GradMax is $\sqrt{10^{-3}}$.**

Table D.2: Initial and final architecture for the models of Figure 5.6. Numbers in color indicate where the methods were allowed to add neurons (middle of ResNet blocks). In blue the initial structure for the model 1/64 and in green the initial structure for the model 1/4, i.e., $\boxed{1|16}$ indicates that the model 1/64 started with 1 neuron at this layer while the model 1/4 started with 16 neurons at the same layer. In red are indicated the final number of neuron at this layer.

| ResNet18 | | |
|---|---|---|
| name | Output size | layers (kernel=(3,3), padd.=1) |
| Conv 1 | $32 \times 32 \times 64$ | $\boxed{3 \times 3,}$ |
| Conv 2 | $32 \times 32 \times 64$ | $\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & \boxed{1\|16} \rightarrow 64 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{1\|16} \rightarrow 64 \\ 3 \times 3, & 64 \end{bmatrix}$ |
| Conv 3 | $32 \times 32 \times 64$ | $\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & \boxed{1\|16} \rightarrow 64 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{1\|16} \rightarrow 64 \\ 3 \times 3, & 64 \end{bmatrix}$ |
| Conv 4 | $16 \times 16 \times 64$ | $\boxed{3 \times 3, 128}$ |
| Conv 5 | $16 \times 16 \times 128$ | $\begin{bmatrix} 3 \times 3, & 128 \\ 3 \times 3, & \boxed{2\|32} \rightarrow 128 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{2\|32} \rightarrow 128 \\ 3 \times 3, & 128 \end{bmatrix}$ |
| Conv 6 | $16 \times 16 \times 128$ | $\begin{bmatrix} 3 \times 3, & 128 \\ 3 \times 3, & \boxed{2\|32} \rightarrow 128 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{2\|32} \rightarrow 128 \\ 3 \times 3, & 128 \end{bmatrix}$ |
| Conv 7 | $8 \times 8 \times 256$ | $\boxed{3 \times 3, 256}$ |
| Conv 8 | $8 \times 8 \times 256$ | $\begin{bmatrix} 3 \times 3, & 256 \\ 3 \times 3, & \boxed{4\|64} \rightarrow 256 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{4\|64} \rightarrow 256 \\ 3 \times 3, & 256 \end{bmatrix}$ |
| Conv 9 | $8 \times 8 \times 256$ | $\begin{bmatrix} 3 \times 3, & 256 \\ 3 \times 3, & \boxed{4\|64} \rightarrow 256 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{4\|64} \rightarrow 256 \\ 3 \times 3, & 256 \end{bmatrix}$ |
| Conv 10 | $4 \times 4 \times 512$ | $\boxed{3 \times 3, 512}$ |
| Conv 11 | $4 \times 4 \times 512$ | $\begin{bmatrix} 3 \times 3, & 512 \\ 3 \times 3, & \boxed{8\|128} \rightarrow 512 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{8\|128} \rightarrow 512 \\ 3 \times 3, & 512 \end{bmatrix}$ |
| Conv 12 | $4 \times 4 \times 512$ | $\begin{bmatrix} 3 \times 3, & 512 \\ 3 \times 3, & \boxed{8\|128} \rightarrow 512 \end{bmatrix}$ $\begin{bmatrix} 3 \times 3, & \boxed{8\|128} \rightarrow 512 \\ 3 \times 3, & 512 \end{bmatrix}$ |
| AvgPool2d | $1 \times 1 \times 512$ | |
| FC 1 | 100 | $512 \times 100$ |
| SoftMax | 100 | |

| | $\Delta t = 1$ | | Baseline |
|---|---|---|---|
| | TINY | GradMax | |
| s = 1/64 | $68.1 \pm 0.5$<br>$68.7 \pm 0.6$ [5*] | $57.2 \pm 0.3$<br>$57.7 \pm 0.3$ [3*] | $72.8 \pm 0.3$ [5*] |

Table D.3: Final accuracy on test split of ResNet18 of D.4 after the architecture growth (*grey*) and after convergence (*blue*). The number of stars indicates the multiple of 50 epochs needed to achieve convergence. With the starting architecture $\text{ResNet}_{1/64}$ and $\Delta t = 1$ the method TINY achieves $68.1 \pm 0.5$ on test split after its growth and it reaches $68.7 \pm 0.6$ [5*]after $* := 5 \times 50$ epochs.