Growing Tiny Networks: Spotting Expressivity Bottlenecks and Fixing Them Optimally

Manon Verbockhaven, Sylvain Chevallier, Guillaume Charpiat

GROWING TINY NETWORKS: SPOTTING EXPRESSIVITY BOTTLENECKS AND FIXING THEM OPTIMALLY

Manon Verbockhaven, Sylvain Chevallier, Guillaume Charpiat
TAU team, Université Paris-Saclay, CNRS, Inria, LISN, 91405, Orsay, France
firstname.name@inria.fr

ABSTRACT

Machine learning tasks are generally formulated as optimization problems, where one searches for an optimal function within a certain functional space. In practice, parameterized functional spaces are considered, in order to be able to perform gradient descent. Typically, a neural network architecture is chosen and fixed, and its parameters (connection weights) are optimized, yielding an architecture-dependent result. This way of proceeding however 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. Costly architectural hyper-parameter optimization is often performed to compensate for this. Instead, we propose to adapt the architecture on the fly during training.

We show that the information about desirable architectural changes, due to expressivity bottlenecks when attempting to follow the functional gradient, can be extracted from backpropagation. To do this, we propose a mathematical definition of expressivity bottlenecks, which enables us to detect, quantify and solve them while training, by adding suitable neurons when and where needed. Thus, while the standard approach requires large networks, in terms of number of neurons per layer, for expressivity and optimization reasons, we are able to start with very small neural networks and let them grow appropriately. As a proof of concept, we show results on the CIFAR dataset, matching large neural network accuracy, with competitive training time, while removing the need for standard architectural hyper-parameter search.

1 INTRODUCTION

Issues with the fixed-architecture paradigm. Universal approximation theorems such as (Hornik et al., 1989; Cybenko, 1989) are historically among the first theoretical results obtained on neural networks, stating the family of neural networks with arbitrary width as a good candidate for a parameterized space of functions to be used in machine learning. However the current common practice in neural network training consists in choosing a fixed architecture, and training it, without any possible architecture modification meanwhile. This inconveniently prevents the direct application of these universal approximation theorems, as expressivity bottlenecks that might arise in a given layer during training will not be able to be fixed. There are two approaches to circumvent this in daily practice. Either one chooses a (very) large width, to be sure to avoid expressivity and optimization issues (Hanin & Rolnick, 2019b; Raghu et al., 2017), to the cost of extra computational power consumption for training and applying such big models; to mitigate this cost, model reduction techniques are often used afterwards, using pruning, tensor factorization, quantization (Louizos et al., 2017) or distillation (Hinton et al., 2015). Or one tries different architectures and keeps the most suitable one (in terms of performance-size compromise for instance), which multiplies the computational burden by the number of trials. This latter approach relates to the Auto-DeepLearning field (Liu et al., 2020), where different exploration strategies over the space of architecture hyper-parameters (among other ones) have been tested, including reinforcement learning (Baker et al., 2017; Zoph & Le, 2016), Bayesian optimization techniques (Mendoza et al., 2016), and evolutionary approaches (Miller et al., 1989; Stanley et al., 2009; Miikkulainen et al., 2017; Bennet et al., 2021), that all rely on random tries and consequently take time for exploration. Within that line, Net2Net (Chen et al., 2015), AdaptNet (Yang et al., 2018) and MorphNet (Gordon et al., 2018) propose different strategies to explore possible variations of a given architecture, possibly guided by model size con-

straints. Instead, we aim at providing a way to locate precisely expressivity bottlenecks in a trained network, which might speed up neural architecture search significantly. Moreover, based on such observations, we aim at modifying the architecture *on the fly* during training, in a single run (no re-training), using first-order derivatives only, while avoiding neuron redundancy. Related work on architecture adaptation while training includes probabilistic edges (Liu et al., 2019) or sparsifying priors (Wolinski et al., 2020). Yet the training is done on the largest architecture allowed, which is resource-consuming. On the opposite we aim at starting from the simplest architecture possible.

Optimization properties. An important reason for common practice to choose wide architectures is the associated optimization properties: sufficiently larger networks are proved theoretically and shown empirically to be better optimized than small ones (Jacot et al., 2018). Typically, small networks exhibit issues with spurious local minima, while wide ones find good nearly-global minima. One of our goals is to train small networks without suffering from such optimization difficulties.

Neural architecture growth. A related line of work consists in growing networks neuron by neuron, by iteratively estimating the best possible neurons to add, according to a certain criterion. For instance, approaches such as (Wu et al., 2019) or Firefly (Wu et al., 2020) aim at escaping local minima by adding neurons that minimize the loss under neighborhood constraints. These neurons are found by gradient descent or by solving quadratic problems involving second-order derivatives. Other approaches (Causse et al., 2019; Bashtova et al., 2022), including GradMax (Evci et al., 2022), seek to minimize the loss as fast as possible and involve another quadratic problem. However the neurons added by these approaches are possibly redundant with existing neurons, especially if one does not wait for training convergence to a local minimum (which is time consuming) before adding neurons, therefore producing larger-than-needed architectures.

Redundancy. To our knowledge, the only approach tackling redundancy in neural architecture growth adds random neurons that are orthogonal in some sense to the ones already present (Maile et al., 2022). More precisely, the new neurons 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 latter formulation is close to the one of GradMax, in that both study first-order loss variations and use the same pre-activation matrix, with an important difference though: GradMax optimally decreases the loss without caring about redundancy, while the other one avoids redundancy but picks random directions instead of optimal ones. In this paper we bridge the gap between these two approaches, picking optimal directions that avoid redundancy in the pre-activation space.

Notions of expressivity. Several concepts of expressivity or complexity exist in the Machine Learning literature, ranging from Vapnik-Chervonenkis dimension (Vapnik & Chervonenkis, 1971) and Rademacher complexity (Koltchinskii, 2001) to the number of pieces in a piecewise affine function (as networks with ReLU activations are) (Serra et al., 2018; Hanin & Rolnick, 2019a). Bottlenecks have been also studied from the point of view of Information Theory, through mutual information between the activities of different layers (Tishby & Zaslavsky, 2015; Dai et al., 2018); this quantity is difficult to estimate though. Also relevant and from Information Theory, the Minimum Description Length paradigm and Kolmogorov complexity (Kolmogorov, 1965; Li et al., 2008) enable to define trade-offs between performance and model complexity.

In this article, we aim at measuring lacks of expressivity 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 one that can be realized by a parameter update. Intuitively, differences arise when a layer does not have sufficient expressive power to realize the desired variation. Our main contributions are that we:

- take a functional analysis viewpoint over gradient descent on neural networks, suggesting to attempt to follow the functional gradient. We optimize not only the weights of the current architecture, but also the architecture itself on the fly, in order to progressively move towards more suitable parameterized functional spaces. This removes the optimization issues (local minima) that are due to thin architectures;
- properly define and quantify the notion of expressivity bottlenecks, globally at the neural network output as well as at each layer, and this in an easily computable way. This allows to localize expressivity bottlenecks in a neural network;

- mathematically define the best possible neurons to add to a given layer to decrease lacks of expressivity as a quadratic problem; compute them and their associated expressivity gain;
- automatically adapt the architecture to the task at hand by making it grow where needed, and this in a single run, in competitive computational complexity with respect to classically training a large model just once. To remove any need for layer width hyper-optimization, one could define a target accuracy and stop adding neurons when it is reached.

2 MAIN CONCEPTS

2.1 NOTATIONS

Let $\mathcal{F}$ be a functional space, e.g. $L_2(\mathbb{R}^p \to \mathbb{R}^d)$, and a loss function $\mathcal{L} : \mathcal{F} \to \mathbb{R}$ defined on it, of the form $\mathcal{L}(f) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\ell(f(\boldsymbol{x}), \boldsymbol{y})\right]$, where ℓ is the per-sample loss, assumed to be differentiable, and where $\mathcal{D}$ is the sample distribution, from which the dataset $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), ..., (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$ is sampled, with $\boldsymbol{x}_i \in \mathbb{R}^p$ and $\boldsymbol{y}_i \in \mathbb{R}^d$.

For the sake of simplicity we consider a feedforward neural 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 σ_l which satisfies $\sigma_l(0) = 0$. The network parameters are then $\theta := (\boldsymbol{W}_l)_{l=1...L}$. The network iteratively computes:

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

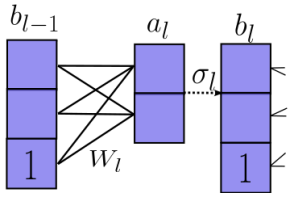

Figure 1: Notations

To any vector-valued function noted $\boldsymbol{t}(\boldsymbol{x})$ and any batch of inputs $\boldsymbol{X} := [\boldsymbol{x}_1, ..., \boldsymbol{x}_n]$, we associate the concatenated matrix $\boldsymbol{T}(\boldsymbol{X}) := (\boldsymbol{t}(\boldsymbol{x}_1) \quad ... \quad \boldsymbol{t}(\boldsymbol{x}_n)) \in \mathbb{R}^{|\boldsymbol{t}(.)| \times n}$. The matrices of pre-activation and post-activation activities at layer l over a minibatch $\boldsymbol{X}$ are thus respectively: $\boldsymbol{A}_l(\boldsymbol{X}) = (\boldsymbol{a}_l(\boldsymbol{x}_1) \quad ... \quad \boldsymbol{a}_l(\boldsymbol{x}_n))$ and $\boldsymbol{B}_l(\boldsymbol{X}) = (\boldsymbol{b}_l(\boldsymbol{x}_1) \quad ... \quad \boldsymbol{b}_l(\boldsymbol{x}_n))$.

NB: convolutions can also be considered, with appropriate representations (cf matrix $\boldsymbol{b}_l^c(\boldsymbol{x})$ in 13).

2.2 APPROACH

Functional gradient descent. We take a functional perspective on the use of neural networks. Ideally in a machine learning task, one would search for a function $f : \mathbb{R}^p \to \mathbb{R}^d$ that minimizes the loss $\mathcal{L}$ by gradient descent: $\frac{\partial f}{\partial t} = -\nabla_f \mathcal{L}(f)$ for some metric on the functional space $\mathcal{F}$ (typically, $L_2(\mathbb{R}^p \to \mathbb{R}^d)$), where ∇_f denotes the functional gradient and t denotes the evolution time of the gradient descent. The descent direction $\boldsymbol{v}_{\text{goal}} := -\nabla_f \mathcal{L}(f)$ is a function of the same type as f and whose value at $\boldsymbol{x}$ is easily computable as $\boldsymbol{v}_{\text{goal}}(\boldsymbol{x}) = -\left(\nabla_f \mathcal{L}(f)\right)(\boldsymbol{x}) = -\nabla_{\boldsymbol{u}} \ell(\boldsymbol{u}, \boldsymbol{y}(\boldsymbol{x}))\big|_{\boldsymbol{u}=f(\boldsymbol{x})}$ (see Appendix A.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 $\mathcal{L}$.

Parametric gradient descent reminder. However in practice, to represent functions and to compute gradients, the infinite-dimensioned functional space $\mathcal{F}$ 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_θ that can be represented with such a network for any parameter value θ. Under standard weak assumptions (see Appendix A.2), the gradient descent is of the form:

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

Using the chain rule, these parameter updates yield a functional evolution :

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

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.

Optimal move direction. We name $\mathcal{T}_{\mathcal{A}}^{f_\theta}$, or just $\mathcal{T}_{\mathcal{A}}$, the tangent space of $\mathcal{F}_{\mathcal{A}}$ at f_θ, that is, the set of all possible infinitesimal variations around f_θ 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 space is a first-order approximation of the neighborhood of f_θ 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).

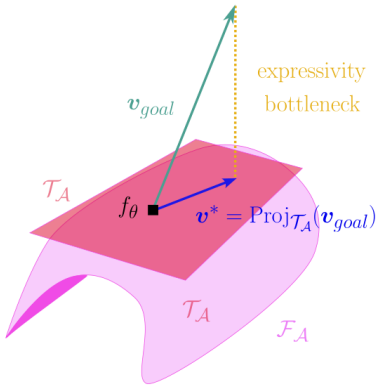

Figure 2: Expressivity bottleneck

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 θ 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:

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

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

$$\delta\theta^* = \mathop{\arg\min}_{\delta\theta\in\Theta_{\mathcal{A}}} \left\| \frac{\partial f_\theta}{\partial \theta}\,\delta\theta - \boldsymbol{v}_{\text{goal}} \right\|^2 = \mathop{\arg\min}_{\delta\theta\in\Theta_{\mathcal{A}}} \left\{ D_\theta\mathcal{L}(f_\theta)(\delta\theta) + \frac{1}{2}\left\| \frac{\partial f_\theta}{\partial \theta}\,\delta\theta \right\|^2 \right\} =: -\nabla_\theta^{\mathcal{T}_{\mathcal{A}}}\mathcal{L}(f_\theta)\,.$$

Lack of expressivity. 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 (see Appendix A.4 for an example). 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 Eq. 1).

2.3 Generalizing to all layers

Ideal updates. 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 an sample $(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}$, standard backpropagation already iteratively computes $\boldsymbol{v}_{\text{goal}}^l(\boldsymbol{x}) := -\left(\nabla_{\boldsymbol{a}_l}\mathcal{L}(f_\theta)\right)(\boldsymbol{x}) = -\nabla_{\boldsymbol{u}}\ell\left(\sigma_L(\boldsymbol{W}_L\,\sigma_{L-1}(\boldsymbol{W}_{L-1}\ldots\sigma_l(\boldsymbol{u}))),\,\boldsymbol{y}\right)|_{\boldsymbol{u}=\boldsymbol{a}_l(\boldsymbol{x})}$, 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 $\nabla_{\boldsymbol{W}_l}\ell(f_\theta(\boldsymbol{x}),\boldsymbol{y}) = \frac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial \boldsymbol{W}_l}\,\nabla_{\boldsymbol{a}_l}\ell(f_\theta(\boldsymbol{x}),\boldsymbol{y})$.

$\boldsymbol{v}_{\text{goal}}^l(\boldsymbol{x}) := -\left(\nabla_{\boldsymbol{a}_l}\mathcal{L}(f_\theta)\right)(\boldsymbol{x})$ 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 θ-parameterized family of functions $\boldsymbol{a}_l$ is not expressive enough.

Activity update resulting from a parameter change. 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: $\delta\tilde{\theta} = -\eta \sum_{(\boldsymbol{x},\boldsymbol{y})\in\text{minibatch}} \nabla_{\tilde{\theta}} \ell(f_\theta(\boldsymbol{x}), \boldsymbol{y})$ for some learning rate η), the impact of the parameter update $\delta\tilde{\theta}$ on the pre-activations $\boldsymbol{a}_l$ at layer l at order 1 in $\delta\tilde{\theta}$ is $\boldsymbol{v}^l(\boldsymbol{x}_i, \delta\tilde{\theta}) := \dfrac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial \tilde{\theta}}\, \delta\tilde{\theta}$.

Note: given a learning rate η, in the sequel we will rather consider $\boldsymbol{v}^l_{\text{goal}}(\boldsymbol{x}) := -\eta \nabla_{\boldsymbol{a}_l} \mathcal{L}(f_\theta)(\boldsymbol{x})$.

3 Expressivity bottlenecks

We now 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 2):

Definition 3.1 (Lack of expressivity)**.** *For a neural network f_θ and a minibatch of points $\boldsymbol{X} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, we define the lack of expressivity at layer l as how far the desired activity update $\boldsymbol{V}^l_{goal} = (\boldsymbol{v}^l_{goal}(\boldsymbol{x}_1), \boldsymbol{v}^l_{goal}(\boldsymbol{x}_2), ...)$ is from the closest possible activity update $V^l = (\boldsymbol{v}^l(\boldsymbol{x}_1), \boldsymbol{v}^l(\boldsymbol{x}_2), ...)$ realizable by a parameter change $\delta\theta$:*

$$\Psi^l := \min_{\boldsymbol{v}^l \in \mathcal{T}_{\mathcal{A}}^{\boldsymbol{a}_l}} \frac{1}{n} \sum_{i=1}^{n} \left\| \boldsymbol{v}^l(\boldsymbol{x}_i) - \boldsymbol{v}^l_{\text{goal}}(\boldsymbol{x}_i) \right\|^2 = \min_{\delta\theta} \frac{1}{n} \left\| \boldsymbol{V}^l(\boldsymbol{X}, \delta\theta) - \boldsymbol{V}^l_{\text{goal}}(\boldsymbol{X}) \right\|^2_{\text{Tr}} \tag{2}$$

where $||.||$ stands for the L_2 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 previous section. In the two following parts we fix the minibatch $\boldsymbol{X}$ and simplify notations accordingly by removing the dependency on $\boldsymbol{X}$.

3.1 Best move without modifying the architecture of the network

Let $\delta\boldsymbol{W}_l^*$ be the solution of 2 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:

$$\delta\boldsymbol{W}_l^* = \arg\min_{\delta\mathbf{W}_l} \frac{1}{n} \left\| \boldsymbol{V}^l(\delta\mathbf{W}_l) - \boldsymbol{V}^l_{\text{goal}} \right\|^2_{\text{Tr}} \tag{3}$$

Proposition 3.1. *The solution of Problem (3) is:*

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

where P^+ denotes the generalized inverse of matrix P.

This update $\delta\boldsymbol{W}_l^*$ is not equivalent to the usual gradient descent update, whose form is $\delta\boldsymbol{W}_l^{\text{GD}} \propto \boldsymbol{V}^l_{\text{goal}} \boldsymbol{B}_{l-1}^T$. 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 $\frac{1}{n}\boldsymbol{B}_{l-1}^T (\frac{1}{n}\boldsymbol{B}_{l-1}\boldsymbol{B}_{l-1}^T)^+ \boldsymbol{B}_{l-1}$. 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 ($\boldsymbol{V}^l_{\text{goal}} \boldsymbol{B}_{l-1}^T$) and of a (kind of) natural gradient only for the last part (projector), as we consider metrics in the pre-activation space.

3.2 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 Figure 3). We have the following expansions by concatenation : $\boldsymbol{W}_{l-1}^T \leftarrow \begin{pmatrix} \boldsymbol{W}_{l-1}^T & \boldsymbol{\alpha}_1 & ... & \boldsymbol{\alpha}_K \end{pmatrix}$ and $\boldsymbol{W}_l \leftarrow \begin{pmatrix} \boldsymbol{W}_l & \boldsymbol{\omega}_1 & ... & \boldsymbol{\omega}_K \end{pmatrix}$.

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

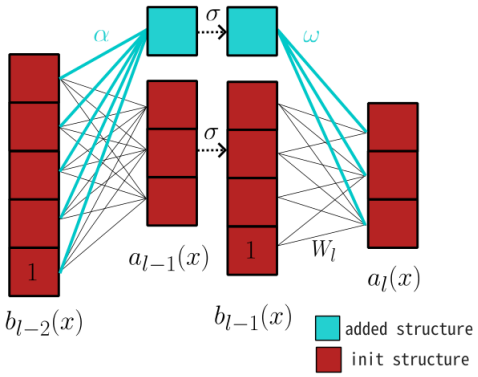

Figure 3: Adding one neuron to layer l in cyan ($K = 1$), with connections in cyan. Here, $\boldsymbol{\alpha} \in \mathrm{R}^5$ and $\boldsymbol{\omega} \in \mathrm{R}^3$.

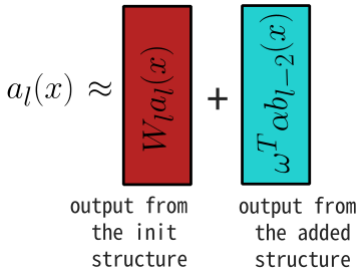

Figure 4: Sum of functional moves

The added neurons could be chosen randomly, as in 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. θ would be enriched from the new parameters. Another option is to solve a optimization problem as in the previous section with the modified structure $\theta \leftarrow \theta \oplus \theta_{\leftrightarrow}^K$ and jointly search for both the optimal new parameters $\theta_{\leftrightarrow}^K$ and the optimal variation $\delta \mathbf{W}_l$ of the old ones:

$$\underset{\theta_{\leftrightarrow}^K,\, \delta \mathbf{W}_l}{\arg\min} \left\| \boldsymbol{V}^l(\delta \mathbf{W}_l \oplus \theta_{\leftrightarrow}^K) - \boldsymbol{V}_{\text{goal}}^l \right\|_{\text{Tr}}^2 \tag{4}$$

As shown in figure 4, the displacement $\boldsymbol{V}^l$ at layer l is actually a sum of the moves induced by the neurons already present ($\delta \mathbf{W}_l$) and by the added neurons ($\theta_{\leftrightarrow}^K$), our problem rewrites as :

$$\underset{\theta_{\leftrightarrow}^K,\, \delta \mathbf{W}_l}{\arg\min} \left\| \boldsymbol{V}^l(\theta_{\leftrightarrow}^K) + \boldsymbol{V}^l(\delta \mathbf{W}_l) - \boldsymbol{V}_{\text{goal}}{}^l \right\|_{\text{Tr}}^2 \tag{5}$$

with $\boldsymbol{v}^l(\boldsymbol{x}, \theta_{\leftrightarrow}^K) := \sum_{k=1}^K \boldsymbol{\omega}_k \, (b_{l-2}(\boldsymbol{x})^T \boldsymbol{\alpha}_k)$ (See A.5). We choose $\delta \mathbf{W}_l$ as the best move of already-existing parameters as defined in Proposition 3.1 and we note $\boldsymbol{V}_{\text{goal}_{proj}}^l := \boldsymbol{V}_{\text{goal}}^l - \boldsymbol{V}^l(\delta \mathbf{W}_l{}^*)$. We are looking for the solution $\left(K^*, \theta_{\leftrightarrow}^{K*}\right)$ of the optimization problem :

$$\underset{K,\, \theta_{\leftrightarrow}^K}{\arg\min} \left\| \boldsymbol{V}^l(\theta_{\leftrightarrow}^K) - \boldsymbol{V}_{\text{goal}_{proj}}^l \right\|_{\text{Tr}}^2 . \tag{6}$$

This quadratic optimization problem can be solved thanks to the low-rank matrix approximation theorem (Eckart & Young, 1936), using matrices $\boldsymbol{N} := \frac{1}{n} \boldsymbol{B}_{l-2} \left(\boldsymbol{V}_{\text{goal}_{proj}}^l \right)^T$ and $\boldsymbol{S} := \frac{1}{n} \boldsymbol{B}_{l-2} \boldsymbol{B}_{l-2}^T$. As $\boldsymbol{S}$ is semi-positive definite, let its truncated SVD be $\boldsymbol{S} = \boldsymbol{U} \Sigma \boldsymbol{U}^T$, and define $S^{-\frac{1}{2}} := \boldsymbol{U} \sqrt{\Sigma}^{-1} \boldsymbol{U}^T$, with the convention that the inverse of 0 eigenvalues is 0. Finally, consider the truncated SVD of matrix $S^{-\frac{1}{2}} \boldsymbol{N} = \sum_{k=1}^R \lambda_k \boldsymbol{u}_k \boldsymbol{v}_k^T$, where R is the rank of the matrix $S^{-\frac{1}{2}} \boldsymbol{N}$. Then:

Proposition 3.2. *The solution of Problem (6) is:*

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

Moreover for any number of neurons $K \leqslant R$, and associated weights $\theta_{\leftrightarrow}^{K,*}$, the expressivity gain and the first order in η 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 singular values λ_k:

$$\Psi_{\theta \oplus \theta_{\leftrightarrow}^{K,*}}^l = \Psi_\theta^l - \sum_{k=1}^K \lambda_k^2 \qquad \textit{and} \qquad \mathcal{L}(f_{\theta \oplus \theta_{\leftrightarrow}^{K,*}}) = \mathcal{L}(f_\theta) + \frac{\sigma_l'(0)}{\eta} \sum_{k=1}^K \lambda_k^2 + o(||\theta_{\leftrightarrow}^{K,*}||^2)$$

Proposition 3.3. *If $\boldsymbol{S}$ is positive definite, then solving (5) 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 λ 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 η, 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.*

Note: Problems (5) and (6) are generally not equivalent, though similar (cf C.4).
Note 2: Minimizing the distance (4), ie the distance between $\boldsymbol{V}^l_{\text{goal}}$ and $\boldsymbol{V}^l$, is equivalent to minimizing the loss $\mathcal{L}$ at order one in $\boldsymbol{V}^l$:

$$\mathcal{L}(f_{\theta \oplus \theta^K_{\leftrightarrow}}) \approx \mathcal{L}(f_\theta) - \frac{\sigma'_{l-1}(0)}{\eta\, n} \left\langle \boldsymbol{V}^l_{\text{goal}},\ \boldsymbol{V}^l \right\rangle_{\text{Tr}} \tag{7}$$

The family $\{\boldsymbol{V}^{l+1}((\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k))\}_{k=1}^K$ of pre-activity variations induced by adding the neurons $\theta^{K,*}_{\leftrightarrow}$ 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 7, but without avoiding redundancy (see Appendix B.1 for more details).

Addition of new neurons. In practice before adding new neurons (α, ω), we multiply them by an amplitude factor γ found by a simple line search (see Appendix D.3), i.e. we add $(\sqrt{\gamma}\alpha, \sqrt{\gamma}\omega)$. The addition of each neuron k has an impact on the loss of the order of $\gamma\lambda_k^2$ provided γ is small. This performance gain could be used in a selection criterion realizing a trade-off with computational complexity. A selection based on statistical significance of singular values can also be performed. The full algorithm and its complexity are detailed in Appendices D.4 and D.5.

4 About greedy growth sufficiency

One might wonder whether a greedy approach on layer growth might get stuck in a non-optimal state. We derive the following series of propositions in this regard. Since in this work we add neurons layer per layer independently, we study here the case of a single hidden layer network, to spot potential layer growth issues. For the sake of simplicity, we consider the task of least square regression towards an explicit continuous target f^*, defined on a compact set. That is, we aim at minimizing the loss:

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

where $f(x)$ is the output of the neural network and $\mathcal{D}$ is the training set.

Proposition 4.1 (Greedy completion of an existing network)**.** *If f is not f^* yet, then 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. Furthermore:

Proposition 4.2 (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.*

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. One can express a lower bound on how much the loss has improved (for the best such neuron), but it is not a very good bound without further assumptions on f. Furthermore, finding the optimal neuron to add is actually NP-hard (Bach, 2017), so we will not necessarily search for the optimal one.

Proposition 4.3 (Greedy completion by one infinitesimal neuron)**.** *The neuron in the previous proposition can be chosen to 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 (6) will not find neurons to add. Yet following Prop. 4.3 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 ε 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, i.e. $\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}[\boldsymbol{\alpha}\cdot\boldsymbol{x}] = 0$, the quadratic term $(\boldsymbol{\alpha}\cdot\boldsymbol{x})^2$, or higher-order ones otherwise, might be correlated. 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 (5). Note that one could consider other function bases than the polynomials from Taylor expansion. 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 & Carbin, 2018). The difference with usual training is that we would perform such computationally-costly searches only when and where relevant, exploiting all simple information first (linear correlations in each layer).

5 RESULTS

5.1 COMPARISON WITH GRADMAX ON CIFAR-100

The closest growing method to TINY is GradMax Evci et al. (2022), as it solves a quadratic problem similar to 6. By construction, the objective of GradMax is to decrease the loss as fast as possible considering an infinitesimal increment of new neurons. In fact, this method is equivalent to ours, with the main difference that GradMax does not take into account the expressivity of the current architecture as TINY does in 6 by projecting v_{goal}. In-depth details about the difference between the GradMax and TINY are provided in B.1.

In this section, we show on the CIFAR-100 dataset that solving (6) instead of B.1(defined by GradMax) to grow a network allows better final performance and almost full expressivity power. To do so, we have re-implemented the GradMax method and mimicked its growing process which consists in increasing the architecture of a shallow ResNet18$_s$ until it reaches the architecture of the usual ResNet18. This process is described in the pseudo code 1, where two parameters can be chosen : the starting architecture of the model of the usual ResNet18 architecture 3 ($s = 1/4$ or $s = 1/64$) and the time of training between each neurons addition ($\Delta t = 1$ or $\Delta t = 0.25$). Then the number of parameters and the performance of the growing network are evaluated at regular intervals to plot Figure 5.

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). We have summarized the final performance in table 1. We also added the column Reference, which gives the performance of a ResNet18 trained from scratch by usual gradient descent with all its neurons. We do not expect TINY or GradMax to achieve the performance of the reference as its architecture and optimisation process have been optimised for years.

The details of the protocol can be found in the annexes E.1, as well as other technical details such as the dynamic of the learning batch size D.2, the number of examples used to solve the expressivity bottleneck 6 and the complexity of the algorithms D.5. For both methods, all the latter apply so that the main differences between GradMax and TINY is the mathematical definition of the new neurons.

For $s = 1/64$, we observe a significant difference in performance between TINY and GradMax methods. While TINY models almost achieve the reference's performance, they remain stuck 10 points below with GradMax. It suggests that starting with an architecture far from full expressvity,

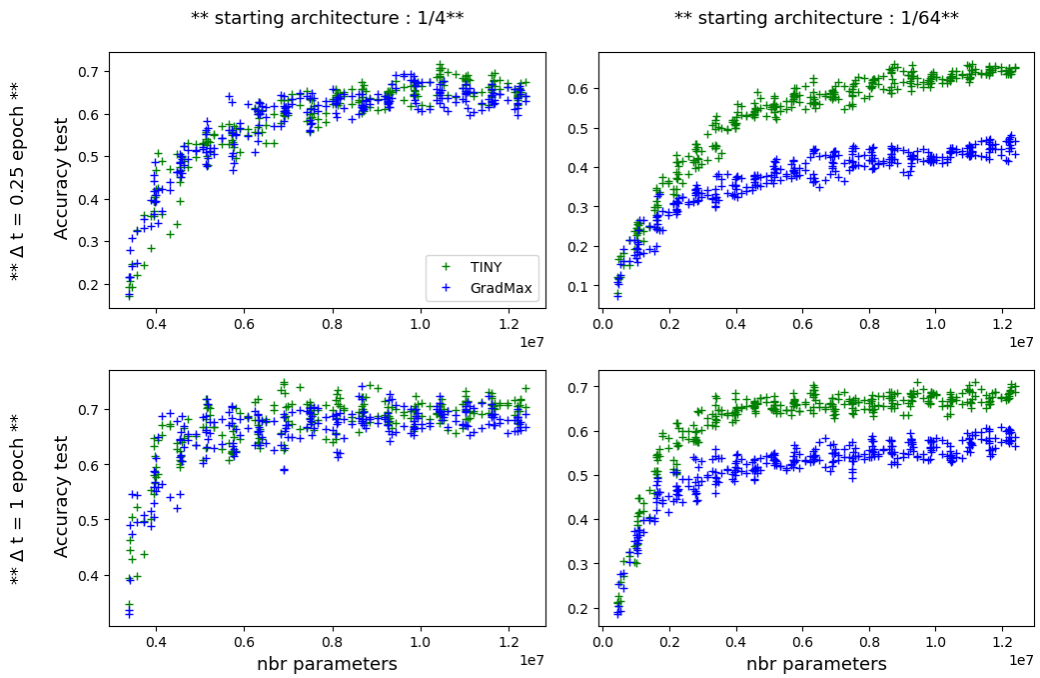

Figure 5: Accuracy on test of as a function of the number of parameters during architecture growth from ResNet$_s$ to ResNet18. The left (resp. right) column is for the starting architecture ResNet$_{1/4}$(resp. ResNet$_{1/64}$). The upper (resp. lower) row is for Δt equal to 0.25 (resp. 1) epoch.

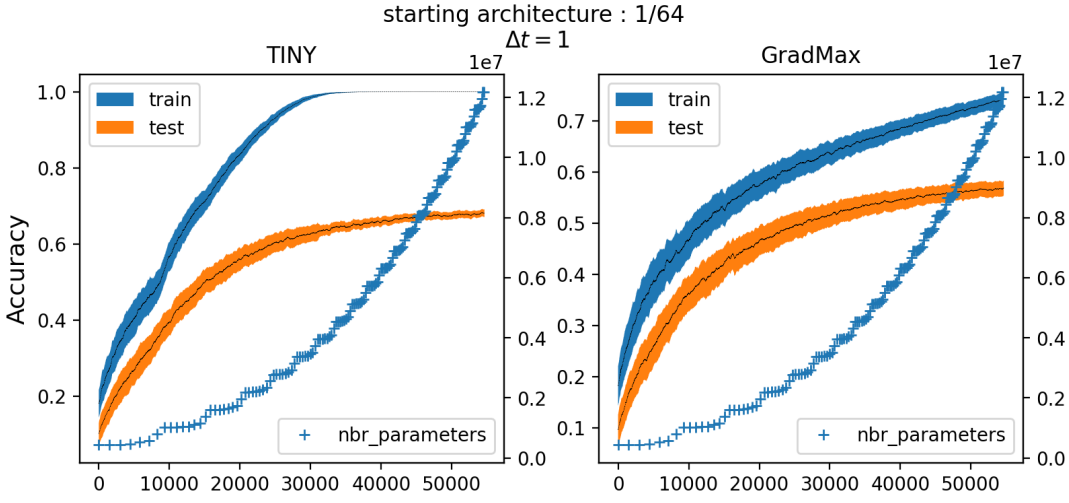

Figure 6: Evolution of accuracy and number of parameters as a function of gradient step for the setting $\Delta t = 1$, $s = 1/64$ for TINY and GradMax, mean and standard deviation over two runs. Other settings in the annexes 13

	TINY		GradMax		Reference
s \ Δt	0.25	1	0.25	1	
1/4	67.0 ± 0.1	71.0 ± 0.1	65.0 ± 0.1	69.0 ± 0.1	72.9 ± 0.1 ^{5*}
1/4	70.0 ± 0.2 ^{5*}	$\mathbf{71.0 \pm 0.2}$ ^{5*}	67.0 ± 0.2 ^{5*}	69.0 ± 0.1 ^{5*}	
1/64	66.0 ± 0.1	68.0 ± 0.4	45.0 ± 0.2	57.0 ± 0.2	
1/64	69.0 ± 0.1 ^{5*}	$\mathbf{69.0 \pm 0.6}$ ^{5*}	57.0 ± 0.3 ^{10*}	59.0 ± 0.1 ^{10*}	

Table 1: Final accuracy on test of ResNet18 after the architecture growth (*grey*) and after convergence (*black*). The number of start indicated grossly 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 66.0 ± 0.1on test after its growth and it reaches 69.0 ± 0.1 ^{5*}after $5* := 5 \times 50$ epochs (examples of training curves for the extra training 14). Mean and standard deviation performed on 2 runs for each setting.

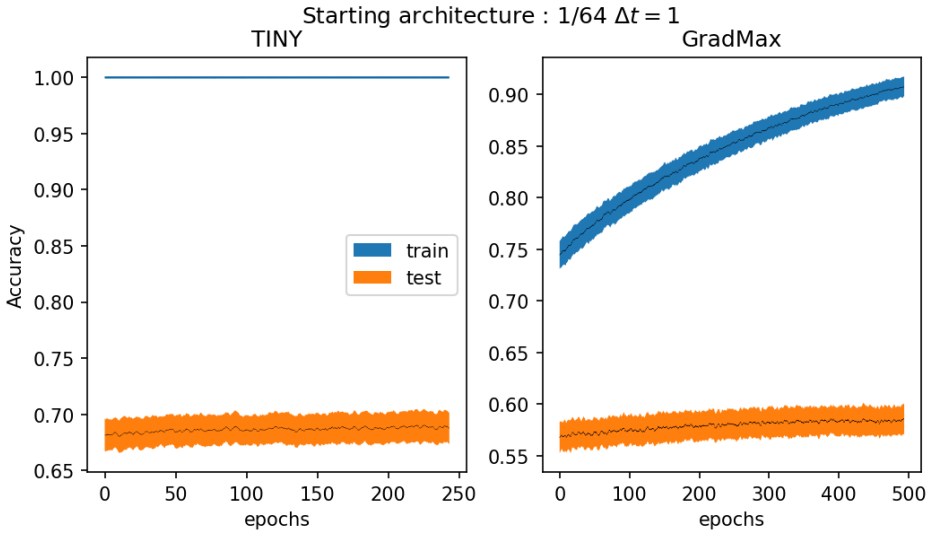

Figure 7: Evolution of accuracy and number of parameters as a function of gradient step for the setting $\Delta t = 1$, $s = 1/64$ during extra training for TINY and GradMax, mean and standard deviation over two runs. Other settings in the annexes 15 and 14.

ie $\text{ResNet}_{1/64}$, the theory proposed by GradMax is not sufficient and TINY is more robust. As for the setting $s = 1/4$, both methods seem equivalent in terms of final performance and achieve full expressivity.

The curves on Figure 6, which are extracted from Figure 13 in the annexes, show that TINY models have converged at the end of the growing process, while GradMax ones do not. This latter effects contrast with GradMax formulation which is to accelerate the gradient descent as fast as possible by adding neurons. Furthermore GradMax need extra training to achieve full expressivity as for the particular setting $s = 1/64, \Delta t = 1$, the extra training time of GradMax is twice as high as TINY's, as shown in Figure 7. This need for extra training also appear in Table 1 where for all settings the difference in performance after and before extra training goes up to 20 % ($s = 1/64, \Delta t = 0.25$, resp. 6% for TINY).

5.2 Comparison with Random on CIFAR-100 : initialisation impact

In this section, we focus on the impact of the new neurons' initialization. To do so, we consider as a baseline the Random method, which initializes the new neurons according to a Gaussian distribution: $(\alpha_k^*, \omega_k^*)_{k=1}^K \sim \mathcal{N}(0, Id)$. Also, when adding new neurons, instead of normalizing them as previously, we search for the best scaling using a line-search on the loss. Thus, we perform the

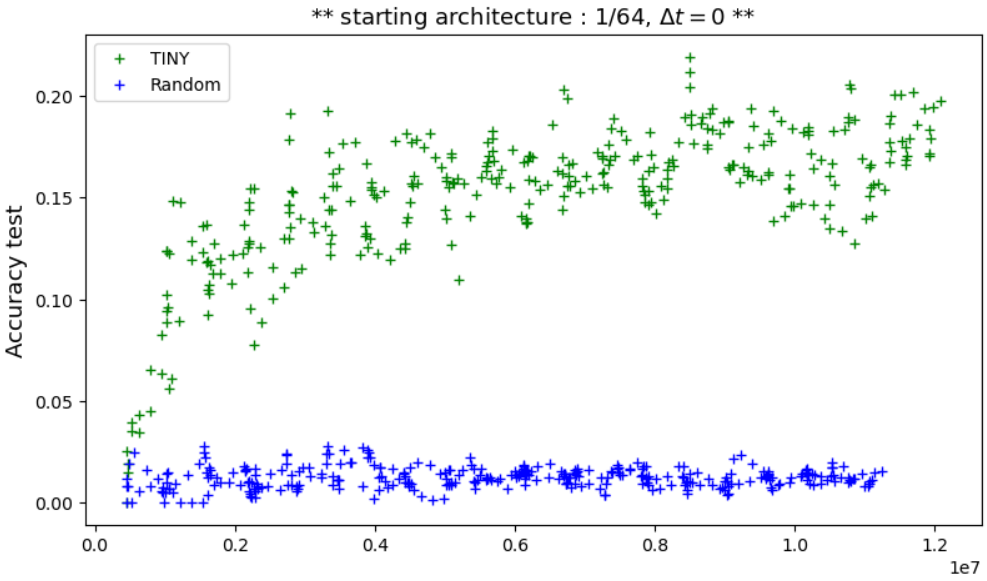

Figure 8: Accuracy on test as a function of the number of parameters during architecture growth from $\text{ResNet}_{1/64}$ to ResNet_{18}.

operation $\theta^K_{\leftrightarrow} \leftarrow \gamma^* \theta^K_{\leftrightarrow}$ 1, with the amplitude factor $\gamma^* \in \mathbb{R}$ defined as :

$$\gamma^* := \underset{\gamma \in [-L,L]}{\arg\min} \sum_i \mathcal{L}(f_{\theta \oplus \gamma \theta^K_{\leftrightarrow}}(\boldsymbol{x}_i), \boldsymbol{y}_i) \qquad \text{with} \quad \gamma \theta^{K^*}_{\leftrightarrow} = (\gamma \alpha^*_k, \gamma \omega^*_k)^K_k \tag{8}$$

with L a positive constant. More details can be found in D.3.2. With such an amplitude factor, one can measure the quality of the directions generated by TINY and Random by quantifying the decrease of loss.

To better measure the impact of the initialisation method, and to distinguish it from the optimization process, we do not preform any gradient descent. This contrasts with the previous section where long training time after architecture 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 Section D.3.1).

With these two modifications to the protocol of previous section, we obtain Figure 8. We see the crucial impact of TINY initialization over the Random method. Indeed, the random method does not learn through the growing process and it can be quantified as follows. In the random setting, we models $\boldsymbol{v}(X), \boldsymbol{v}_{\text{goal}}(X)$ as independent Gaussian variable following $\mathcal{N}(0_d, I_d \frac{1}{\sqrt{d}})$ where d approximate the dimension of $\boldsymbol{v}_{\text{goal}}$ and $\boldsymbol{v}$. From equation 7, 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 of the expected decrease of loss after each architecture growth. This quantity can be approximated by its standard deviation, ie $\frac{1}{\sqrt{d}}$ and therefore making the expected gain of loss comparable to to $\frac{1}{\sqrt{64}}$ for the first layer and $\frac{1}{\sqrt{512}}$ for the last layer.

A first supplementary remarks is that the search interval of equation 8 for can be shrunk to $[0, L]$ as the first order development of the loss in equation 7 is positive. This property is the direct consequence of the definition of $\boldsymbol{V}^*$ as the minimizer of the expressivity bottleneck equation 6. One can also note that we do not include GradMax in Figure 8, because its protocol initializes the on-going weight to zero ($\alpha_k \leftarrow 0$) and imposes a small norm on its out-going weights ($||\omega_k|| = \varepsilon$). Those two aspects make the amplitude factor γ^* meaningless and the impact of the new neurons initialization invisible without gradient descent.

The code and the DEMO-notebook are available at `https://gitlab.inria.fr/mverbock/tinypub`.

6 CONCLUSION

We 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 the neurons where needed and yields a fully trained architecture at the end. Our 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 architectures at the same time.

The method is generic for all architectures and is instantiated for linear and convolutional layers. Extension to self-attention mechanism (transformers) is part of future works. Although the common architectures consist of a succession of layers, a new research direction is to develop tool handling general computational graphs (U-net, Inception, Dense-Net).

Another possible development would be to study the statistics reliability of the TINY method, for instance using tools borrowed from random matrix theory. Indeed statistical tests can be applied on intermediate computations to obtain the new neurons. An interesting byproduct of this approach would define a threshold to select neurons found by 3.2, based on statistical significance.

REFERENCES

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.

Kateryna Bashtova, Mathieu Causse, Cameron James, Florent Masmoudi, Mohamed Masmoudi, Houcine Turki, and Joshua Wolff. Application of the topological gradient to parsimonious neural networks. In *Computational Sciences and Artificial Intelligence in Industry*, pp. 47–61. Springer, 2022.

Pauline Bennet, Carola Doerr, Antoine Moreau, Jeremy Rapin, Fabien Teytaud, and Olivier Teytaud. Nevergrad: black-box optimization platform. *ACM SIGEVOlution*, 14(1):8–15, 2021.

Mathieu Causse, Cameron James, Mohamed Slim Masmoudi, and Houcine Turki. Parsimonious neural networks. In *Cesar Conference*, 2019. URL https://www.cesar-conference.org/wp-content/uploads/2019/10/s5_p1_21_1330.pdf.

Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

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

Bin Dai, Chen Zhu, Baining Guo, and David Wipf. Compressing neural networks using the variational information bottleneck. In *International Conference on Machine Learning*, pp. 1135–1144. PMLR, 2018.

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.

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.

Harley Flanders. Differentiation under the integral sign. *The American Mathematical Monthly*, 80 (6):615–627, 1973. ISSN 00029890, 19300972. URL http://www.jstor.org/stable/2319163.

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.

Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1586–1595, 2018.

Boris Hanin and David Rolnick. Complexity of linear regions in deep networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2596–2604. PMLR, 09–15 Jun 2019a. URL https://proceedings.mlr.press/v97/hanin19a.html.

Boris Hanin and David Rolnick. Deep relu networks have surprisingly few activation patterns. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019b. URL https://proceedings.neurips.cc/paper/2019/file/9766527f2b5d3e95d4a733fcfb77bd7e-Paper.pdf.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

K. Hornik, M. Stinchcombe, and H. 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 (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/5a4be1fa34e62bb8a6ec6b91d2462f5a-Paper.pdf.

Andrei N Kolmogorov. Three approaches to the quantitative definition ofinformation'. *Problems of information transmission*, 1(1):1–7, 1965.

Vladimir Koltchinskii. Rademacher penalties and structural risk minimization. *IEEE Transactions on Information Theory*, 47(5):1902–1914, 2001.

Ming Li, Paul Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*, volume 3. Springer, 2008.

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.

Zhengying Liu, Zhen Xu, Shangeth Rajaa, Meysam Madadi, Julio CS Jacques Junior, Sergio Escalera, Adrien Pavao, Sebastien Treguer, Wei-Wei Tu, and Isabelle Guyon. Towards automated deep learning: Analysis of the autodl challenge series 2019. In *NeurIPS 2019 Competition and Demonstration Track*, pp. 242–252. PMLR, 2020.

Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. *Advances in neural information processing systems*, 30, 2017.

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.

Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (eds.), *Proceedings of the Workshop on Automatic Machine Learning*, volume 64 of *Proceedings of Machine Learning Research*, pp. 58–65, New York, New York, USA, 24 Jun 2016. PMLR. URL https://proceedings.mlr.press/v64/mendoza_towards_2016.html.

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.

Geoffrey F. Miller, Peter M. Todd, and Shailesh U. Hegde. Designing neural networks using genetic algorithms. In *ICGA*, 1989.

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.

Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 2847–2854. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/raghu17a.html.

Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. Bounding and counting linear regions of deep neural networks. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4558–4566. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/serra18b.html.

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.

Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. *CoRR*, abs/1503.02406, 2015. URL http://dblp.uni-trier.de/db/journals/corr/corr1503.html#TishbyZ15.

V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971. doi: 10.1137/1116025. URL http://link.aip.org/link/?TPR/16/264/1.

P. Wolinski, G. Charpiat, and O. Ollivier. Asymmetrical scaling layers for stable network pruning. *OpenReview Archive*, 2020.

Lemeng Wu, Dilin Wang, and Qiang Liu. Splitting steepest descent for growing neural architectures. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/3a01fc0853ebeba94fde4d1cc6fb842a-Paper.pdf.

Lemeng Wu, Bo Liu, Peter Stone, and Qiang Liu. Firefly neural architecture descent: a general approach for growing neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 22373–22383. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/fdbe012e2e11314b96402b32c0df26b7-Paper.pdf.

Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 285–300, 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2016. URL http://arxiv.org/abs/1611.01578. cite arxiv:1611.01578.

Appendix outline

- Section A details the theoretical approach of TINY.
- Section B compares the theoretical approach of TINY with other approaches .
- Section C proves the propositions of the paper.
- Section D provides the hyper parameters for learning.
- Section E gives additional graphics associated to the result part.

We provide additional details, following the same order as the sections in the paper.

A THEORETICAL DETAILS FOR PART 2

A.1 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:

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

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 in which we would like to update f, following an infinitesimal step (of size ε), 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 $\langle , \rangle$ 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^∞ 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 ℓ-loss differentiability and integrability, we get, for any function update direction $v \in \mathcal{F}$ and infinitesimal step size $\varepsilon \in \mathbb{R}$:

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

using the usual gradient of function ℓ 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:

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

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:

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

which simplifies into:

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

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:

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

and

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

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 minibatch 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).

A.2 DIFFERENTIATION UNDER THE INTEGRAL SIGN

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

Then, for all $x \in X$,

$$\frac{d}{dx} \int_\Omega f(x, \omega)\, d\omega = \int_\Omega f_x(x, \omega)\, d\omega$$

See proof and details :Flanders (1973).

A.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 θ, 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 θ 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:

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

where $\| \quad \|_P$ is the norm associated to the chosen inner product. Changing the inner product obviously 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:

$$\begin{aligned} \left\| \delta\theta - \left(-\nabla_\theta \widetilde{\mathcal{L}}(\theta) \right) \right\|^2 &= \|\delta\theta\|^2 + 2\, \delta\theta \cdot \nabla_\theta \widetilde{\mathcal{L}}(\theta) + \left\| \nabla_\theta \widetilde{\mathcal{L}}(\theta) \right\|^2 \\ &= 2 \left(\frac{1}{2} \|\delta\theta\|^2 + D_\theta \widetilde{\mathcal{L}}(\theta)(\delta\theta) \right) + K \end{aligned}$$

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 our case of study, for the norm over the space $\Theta_{\mathcal{A}}$ of parameter variations, we consider a norm of in the space of associated functional variations, 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.

We consider 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. We aim at getting rid of it in order to follow the functional gradient descent as closely as possible.

Searching for

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

or equivalently for:

$$\delta\theta^* \;=\; \arg\min_{\delta\theta\in\Theta_{\mathcal{A}}} \left\| \frac{\partial f_\theta}{\partial\theta}\,\delta\theta - \boldsymbol{v}_{\text{goal}} \right\|^2 \;=\; \arg\min_{\delta\theta\in\Theta_{\mathcal{A}}} \left\{ D_\theta\mathcal{L}(f_\theta)(\delta\theta) + \frac{1}{2}\left\| \frac{\partial f_\theta}{\partial\theta}\,\delta\theta \right\|^2 \right\} =: -\nabla_\theta^{\mathcal{T}_{\mathcal{A}}}\mathcal{L}(f_\theta)$$

then appears as a natural goal.

A.4 Example of 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 $y_i = f_{\text{true}}(x_i)$.

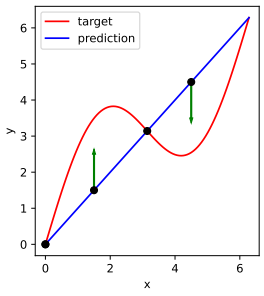

Figure 9: Linear interpolation

A.5 Problem formulation and choice of pre-activities

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

Suppose one wishes to add K neurons $\theta^K_{\leftrightarrow} := (\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)_{k=1}^K$ 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 nul weights, or nul input weights $\boldsymbol{\alpha}_k$ and non-nul output weights $\boldsymbol{\omega}_k$, or the opposite, or both non-nul 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 $\theta^K_{\leftrightarrow} := (\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)_{k=1}^K$ changes the activities $\boldsymbol{a}_l$ of the (next) layer by

$$\delta\boldsymbol{a}_l \;=\; \sum_{k=1}^K \boldsymbol{\omega}_k\, \sigma(\boldsymbol{b}_{l-2}(\boldsymbol{x})^T\boldsymbol{\alpha}_k) \tag{9}$$

Small weights approximation Under the hypothesis of small input weights α_k, the activity variation 9 can be approximated by:

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

at first order in $\|\boldsymbol{\alpha}_k\|$. We will drop the constant $\sigma'(0)$ in the sequel.

This quantity is linear both in α_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 we have:

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

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 $\theta^K_{\leftrightarrow} = (\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)$.

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 nul quantities are added) and also without changing the gradient with respect to the parameters, thus not improving expressivity. Indeed, the added quantity (Eq. 9) involves 0×0 multiplications, and consequently the derivative $\frac{\partial \boldsymbol{a}_l(\boldsymbol{x})}{\partial \theta^K_{\leftrightarrow}}\Big|_{\theta^K_{\leftrightarrow}=0}$ 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 as 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 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 search for the best tangent space for the $\boldsymbol{\alpha}_k$, while symmetrically optimizing w.r.t. the $\boldsymbol{\alpha}_k$ is equivalent to find the best projection on the tangent space defined by the $\boldsymbol{\omega}_k$.

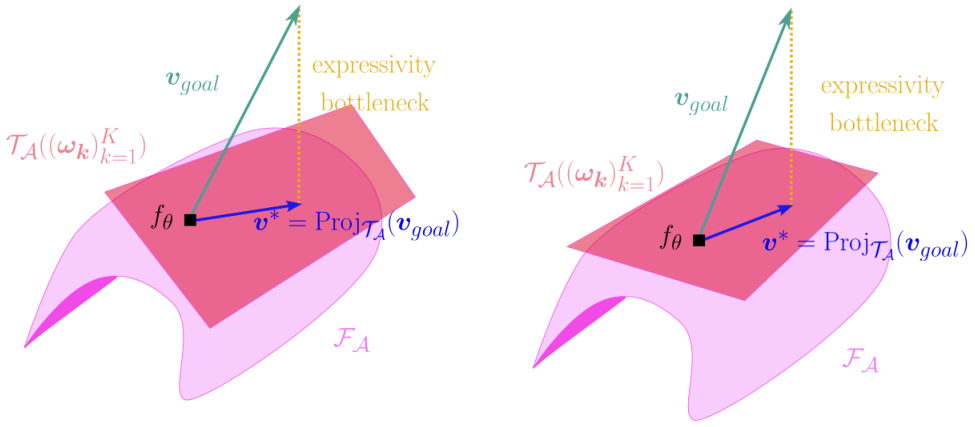

Figure 10: Changing the tangent space with different values of $(\boldsymbol{\omega}_k)_{k=1}^K$.

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 we compute first-order variation quantities.

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 of the non-linear activation function σ added in front of all quantities (while in the case of pre-activations, quantity 9 is linear in $\boldsymbol{\omega}_k$ and thus does not require 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).

A.6 Adding convolutional neurons

To add a convolutional neuron at layer $l-1$, one should add a kernel at layer $l-1$ and expand one dimension to all the kernels in layer l to match the new dimension of the post-activity.

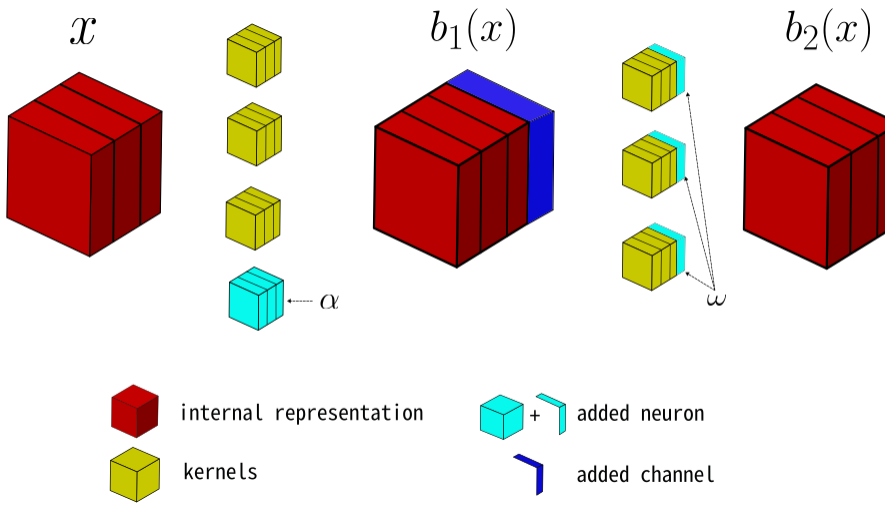

Figure 11: Adding one convolutional neuron at layer one for a input with tree channels.

B Theoretical comparison with other approaches

B.1 GradMax method

The theoretical approach of GradMax is to add neurons with zero fan-in and choose the fan-out that will decrease the loss as much as possible after one gradient step. At time t, we perform such neuron addition and we note Ω the fan-out of the new neurons. After one gradient step, ie $t \to t+1$, we have the approximation :

$$\mathcal{L}^{t+1} \approx \mathcal{L}^t - ||\nabla_\theta \mathcal{L}||^2 - ||\nabla_\Omega \mathcal{L}||^2$$

The solution of GradMax is then :

$$\begin{aligned}
(\boldsymbol{\omega}_1^*, ..., \boldsymbol{\omega}_K^*) := \Omega^* &= \arg\max_\Omega ||\nabla_\Omega \mathcal{L}||^2 && s.t.\ \ ||\Omega||_F^2 < c \\
&= \arg\max_\Omega ||\sum_i \boldsymbol{b}_{l-2}(\boldsymbol{x}_i) \boldsymbol{v}_{\text{goal}}^{l+1\,T}(x_i)\Omega||_F^2 && s.t.\ \ ||\Omega||_F^2 < c \\
&:= \arg\max_\Omega ||\boldsymbol{B}_{l-2} \boldsymbol{V}_{\text{goal}}^{l+1\,T} \Omega||_F^2 && s.t.\ \ ||\Omega||_F^2 < c \\
&:= \arg\max_\Omega \text{Tr}\left(\Omega^T \tilde{\boldsymbol{N}}^T \tilde{\boldsymbol{N}} \Omega\right) && s.t.\ \ ||\Omega||_F^2 < c \qquad (10)
\end{aligned}$$

Where c is a normalisation constant. On the other hand, TINY is equivalent to the following optimisation problem :

$$\begin{aligned}
(\boldsymbol{\omega}_1^*, ..., \boldsymbol{\omega}_K^*) = \Omega^* &= \arg\max_{\Omega} ||\sum_i \boldsymbol{b}_{l-2}(x_i) {\boldsymbol{v}_{\text{goal}}}_{proj}^{l+1\ T}(x_i)\Omega||_F^2 && s.t.\ \ ||\boldsymbol{B}_{l-1}\Omega||_F^2 < c \\
&= \arg\max_{\Omega} ||\boldsymbol{B}_{l-2}{\boldsymbol{V}_{\text{goal}}}_{proj}^{l\ T}\Omega||_F^2 && s.t.\ \ ||\boldsymbol{B}_{l-1}\Omega||_F^2 < c \\
&= \arg\max_{\hat{\Omega}} \text{Tr}\left(\tilde{\Omega}^T \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{N}^T \boldsymbol{N} \boldsymbol{S}^{-\frac{1}{2}} \tilde{\Omega}\right) && s.t.\ \ ||\tilde{\Omega}||_F^2 < c,\ \tilde{\Omega} := \boldsymbol{S}^{\frac{1}{2}}\Omega \\
&= \arg\max_{\Omega} \text{Tr}\left(\Omega \boldsymbol{S}^{-1} \boldsymbol{N}^T \boldsymbol{N} \boldsymbol{S}^{-1} \Omega\right) && s.t.\ \ ||\Omega||_F^2 < \tilde{c}
\end{aligned}$$

One can note three differences between those optimization problems:

- 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.
- Second, the constraint lies in the weight space for GradMax method while it lies in the pre-activation space in our case. 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 from the parameterization. In practice we do observe that the "natural" gradient direction improves the loss better than the usual L2 gradient.
- Third, our fan-in weights are not set to 0 but directly to their optimal values (at first order).

B.2 NORTH PREACTIVATION

In paper Maile et al. (2022), fan-out weights are initialized to 0 while fan-in weights are initialized as $\boldsymbol{\alpha}_i = \boldsymbol{S}^{-1}\boldsymbol{B}_{l-1}\mathbf{V}_{\mathbf{Z}_l}^T r_i$ where r_i is a random vector and $\mathbf{V}_{\mathbf{Z}_l} \in \mathcal{M}(|Ker(\boldsymbol{B}_{l-1}^T)|, |\boldsymbol{b}_{l-1}(\boldsymbol{x})|)$ is a matrix consisting of the orthogonal vectors of the kernel of pre-activations $\boldsymbol{B}_l$, i.e $\{z \mid \boldsymbol{B}_l^T z = 0\}$. In our paper fan-in weights are initialized as $\boldsymbol{\alpha}_i = \boldsymbol{S}^{-1}\boldsymbol{B}_{l-1}{\boldsymbol{V}_{\text{goal}}}_{proj}^{T}\boldsymbol{v}_i = \boldsymbol{S}^{-1}\boldsymbol{B}_{l-1}{\boldsymbol{V}_{\text{goal}}}^T\mathbf{V}_{\mathbf{Z}_l}\mathbf{V}_{\mathbf{Z}_l}^T\boldsymbol{v}_i$, where the $\boldsymbol{v}_i$ are right eigenvectors of the matrix $\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N}$.

The main difference is thus that we use the backpropagation to find the best $\boldsymbol{v}_i$ or r_i directly, while the NORTH approach tries random directions r_i to explore the space of possible neuron additions.

C PROOFS OF PART 3 AND 4

C.1 PROPOSITION 3.1

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

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

Proof
Consider the function

$$g(\delta\mathbf{W}_l) = ||{\boldsymbol{V}_{\text{goal}}}^l - \delta\mathbf{W}_l\boldsymbol{B}_{l-1}||_{\text{Tr}}^2 \tag{11}$$

then:

$$\begin{aligned} g(\delta \mathbf{W}_l + \boldsymbol{H}) &= ||{\boldsymbol{V}_{\text{goal}}}^l - \delta \mathbf{W}_l \boldsymbol{B}_{l-1} - \boldsymbol{H}\boldsymbol{B}_{l-1}||^2_{\text{Tr}} \\ &= g(\delta \mathbf{W}_l) - 2\langle {\boldsymbol{V}_{\text{goal}}}^l - \delta \mathbf{W}_l \boldsymbol{B}_{l-1}, \boldsymbol{H}\boldsymbol{B}_{l-1}\rangle_{\text{Tr}} + o(||\boldsymbol{H}||) \\ &= g(\delta \mathbf{W}_l) - 2\,\text{Tr}\left(\left({\boldsymbol{V}_{\text{goal}}}^l - \delta \mathbf{W}_l \boldsymbol{B}_{l-1}\right)^T \boldsymbol{H}\boldsymbol{B}_{l-1}\right) + o(||\boldsymbol{H}||) \\ &= g(\delta \mathbf{W}_l) - 2\,\text{Tr}\left(\boldsymbol{B}_{l-1}\left({\boldsymbol{V}_{\text{goal}}}^l - \delta \mathbf{W}_l \boldsymbol{B}_{l-1}\right)^T \boldsymbol{H}\right) + o(||\boldsymbol{H}||) \\ &= g(\delta \mathbf{W}_l) - 2\langle \left({\boldsymbol{V}_{\text{goal}}}^l - \delta \mathbf{W}_l \boldsymbol{B}_{l-1}\right)\boldsymbol{B}_{l-1}^T, \boldsymbol{H}\rangle_{\text{Tr}} + o(||\boldsymbol{H}||) \end{aligned}$$

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

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

Using the definition of the generalized inverse of M^+, we get:

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

as one solution. For convolutional layer, we definied as b_i^c the input associated to the activation $a_l(X_i) \in \mathbb{R}^{k,p,p}$, such that for any convolution layer $Conv$ with parameter W, we have :

$$Conv(a_l(X_i)) = B_i^c \; vect(W) \tag{12}$$

Example : considering the kernel of $Conv$ to be $(2, 2)$, then :

$$\boldsymbol{b}_i^k = \begin{pmatrix} b_i^{1,k} & b_i^{2,k} & . & b_i^{p,k} \\ b_i^{p+1,k} & b_i^{p+2,k} & . & b_i^{2p,k} \\ . & . & . & . \\ b_i^{p(p-1)+1,k} & . & . & b_i^{p^2,k} \end{pmatrix} \tag{13}$$

$$\boldsymbol{B}_i^c = \begin{pmatrix} b_i^{1,1} & b_i^{2,1} & b_i^{p+1,1} & b_i^{p+2,1} & b_i^{1,2} & b_i^{2,2} & b_i^{p+1,2} & b_i^{p+2,2} & b_i^{1,3} & . \\ b_i^{2,1} & b_i^{3,1} & b_i^{p+2,1} & b_i^{p+3,1} & b_i^{2,2} & b_i^{3,2} & b_i^{p+2,2} & b_i^{p+3,2} & b_i^{2,3} & . \\ . & . & . & . & . & . & . & . & . & . \end{pmatrix}$$

Then the function to minimize is

$$g(\delta \mathbf{W}_l) = ||{\boldsymbol{V}_{\text{goal}}}^l - \boldsymbol{B}^c \delta \mathbf{W}_l||^2_{\text{Tr}}$$

where $\boldsymbol{B}^c := \begin{pmatrix} \boldsymbol{B}_1^c & ... & \boldsymbol{B}_n^c \end{pmatrix}$. □

C.2 Proposition 3.2

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

Proposition C.1 (3.2)**.** *The solution of (5) can be written as:*

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

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

$$\mathcal{L}(f_{\theta \oplus \theta_{\leftrightarrow}^{K,*}}) = \mathcal{L}(f_\theta) + \frac{\sigma_l'(0)}{\eta}\sum_{k=1}^K \lambda_k^2 + o(||\theta_{\leftrightarrow}^{K,*}||^2)$$

Proof

We first compute the proof for a linear layer.
The optimal neurons $n_1, ..., n_K$ are defined by $n_i := (\boldsymbol{\alpha}_i, \boldsymbol{\omega}_i)$ and are the solution of the optimization problem :

$$\underset{\delta \boldsymbol{W}_l, \delta\theta^K_{l-1\leftrightarrow l}}{\arg\min} \; || \overbrace{\boldsymbol{V}^l_{\text{goal}} - \delta \boldsymbol{W}_l \boldsymbol{B}_{l-1}}^{\boldsymbol{V}^l_{\text{goal}_{proj}}} - \boldsymbol{\Omega} \boldsymbol{A}^T \boldsymbol{B}_{l-2} ||^2_{\text{Tr}}$$

$$\text{where } \boldsymbol{\Omega} := \begin{pmatrix} \boldsymbol{\omega}_1 & ... & \boldsymbol{\omega}_K \end{pmatrix} \text{ and } \boldsymbol{A} := \begin{pmatrix} \boldsymbol{\alpha}_1 & ... & \boldsymbol{\alpha}_K \end{pmatrix}$$

This is equivalent to :

$$\underset{\boldsymbol{C} = \boldsymbol{\Omega}\boldsymbol{A}^T}{\arg\min} || \overbrace{\boldsymbol{V}^l_{\text{goal}} - \delta \boldsymbol{W}_l \boldsymbol{B}_{l-1}}^{\boldsymbol{V}^l_{\text{goal}_{proj}}} - \boldsymbol{C} \boldsymbol{B}_{l-2} ||^2_{\text{Tr}} \tag{14}$$

Then $\boldsymbol{C}^* = \boldsymbol{S}^+ \boldsymbol{N}$. Taking $R = rank(\boldsymbol{S}^+ \boldsymbol{N})$ and a family $(\boldsymbol{\alpha}_k, \boldsymbol{\omega}_k)_{1,...,K}$ such that $\boldsymbol{\Omega}\boldsymbol{A}^T = \sum_k \boldsymbol{\omega}_k \alpha_k^T = \boldsymbol{S}^+ \boldsymbol{N}$ is a optimal solution. But what if we decide to only add $K < R$ neurons ?

$$\begin{aligned}
\underset{\theta^K_{\leftrightarrow}}{\arg\min} \left\{ \frac{1}{n} || {\boldsymbol{V}_{\text{goal}}}^l_{proj} - \boldsymbol{V}^l(\theta^K_{\leftrightarrow}) ||^2_{\text{Tr}} \right\} &= \underset{\theta^K_{\leftrightarrow}}{\arg\min} \left\{ -\frac{2}{n} \left\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}, \boldsymbol{V}^l(\theta^K_{\leftrightarrow}) \right\rangle_{\text{Tr}} + \frac{1}{n} || \boldsymbol{V}^l(\theta^K_{\leftrightarrow}) ||^2_{\text{Tr}} \right\} \\
&= \underset{\theta^K_{\leftrightarrow}}{\arg\min} \frac{1}{n} g(\theta^K_{\leftrightarrow})
\end{aligned}$$

We have

$$\begin{aligned}
\frac{1}{n} g(\theta^K_{\leftrightarrow}) &= -\frac{2}{n} \sum_i^n \sum_k {\boldsymbol{v}_{\text{goal}}}^l_{proj}(\boldsymbol{x}_i)^T \Big(\boldsymbol{\alpha}_k^T \boldsymbol{b}_{l-2}(\boldsymbol{x}_i) \Big) \boldsymbol{\omega}_k \\
&\qquad + \frac{1}{n} \sum_{k,j}^K \sum_i^n \Big(\boldsymbol{\alpha}_k^T \boldsymbol{b}_{l-2}(\boldsymbol{x}_i) \Big) \boldsymbol{\omega}_k^T \boldsymbol{\omega}_j \Big(\boldsymbol{\alpha}_j^T \boldsymbol{b}_{l-2}(\boldsymbol{x}_i) \Big) \\
&= -2 \sum_k^K \boldsymbol{\alpha}_k^T \Big(\frac{1}{n} \sum_i^n \boldsymbol{b}_{l-2}(\boldsymbol{x}_i) {\boldsymbol{v}_{\text{goal}}}^l_{proj}(\boldsymbol{x}_i)^T \Big) \boldsymbol{\omega}_k \\
&\qquad + \sum_{k,j}^K \boldsymbol{\omega}_k^T \boldsymbol{\omega}_j \boldsymbol{\alpha}_k^T \Big(\frac{1}{n} \sum_i^n \boldsymbol{b}_{l-2}(\boldsymbol{x}_i) \boldsymbol{b}_{l-2}(\boldsymbol{x}_i)^T \Big) \boldsymbol{\alpha}_j \\
&= -2 \sum_k^K \boldsymbol{\alpha}_k^T \boldsymbol{N} \boldsymbol{\omega}_k + \sum_{k,j}^K \boldsymbol{\omega}_k^T \boldsymbol{\omega}_j \boldsymbol{\alpha}_k^T \boldsymbol{S} \boldsymbol{\alpha}_j
\end{aligned}$$

with $\boldsymbol{N} := \frac{1}{n} \boldsymbol{B}_{l-2} \big({\boldsymbol{V}_{\text{goal}}}^l_{proj} \big)^T$ and $\boldsymbol{S} := \frac{1}{n} \boldsymbol{B}_{l-2} \boldsymbol{B}_{l-2}^T$.
Consider the SVD of $\boldsymbol{S} = \boldsymbol{U} \Sigma \boldsymbol{U}^T$. Define $\boldsymbol{S}^{\frac{1}{2}} := \boldsymbol{U} \sqrt{\Sigma} \boldsymbol{U}$ and $\boldsymbol{S}^{-\frac{1}{2}} := \boldsymbol{U} \sqrt{\Sigma}^{-1} \boldsymbol{U}^T$.
Consider also the SVD of $\boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{N} = \sum_{r=1}^R \lambda_r \boldsymbol{v}_r \boldsymbol{e}_r^T$.
Note also $\boldsymbol{\beta}_k := \boldsymbol{S}^{\frac{1}{2}} \boldsymbol{\alpha}_k$. Then :

$$\begin{aligned}
-\sum_{k=1}^K \boldsymbol{\alpha}_k^T N \boldsymbol{\omega}_k &= -\sum_k \boldsymbol{\beta}_k^T \boldsymbol{S}^{-\frac{1}{2}} N \boldsymbol{\omega}_k \\
&= -\operatorname{Tr} \Big(\sum_k \sum_r \lambda_r \Big(\boldsymbol{\beta}_k^T \boldsymbol{v}_r \boldsymbol{e}_r^T \Big) \boldsymbol{\omega}_k \Big)
\end{aligned}$$

Using the linearity of the Trace and that $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$, we have :

$$\begin{aligned}
-\sum_{k=1}^{K} \boldsymbol{\alpha}_k^T N \boldsymbol{\omega}_k &= -\mathrm{Tr}\Big(\sum_k \sum_r \lambda_r \boldsymbol{\omega}_k \boldsymbol{\beta}_k^T \boldsymbol{v}_r \boldsymbol{e}_r^T\Big) \\
&= -\mathrm{Tr}\Big(\sum_k \boldsymbol{\omega}_k \boldsymbol{\beta}_k^T \sum_r \lambda_r \boldsymbol{v}_r \boldsymbol{e}_r^T\Big) \\
&= -\Big\langle \sum_k \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T, \sum_r \lambda_r \boldsymbol{v}_r \boldsymbol{e}_r^T \Big\rangle_{\mathrm{Tr}} \text{ with } \langle \boldsymbol{A}, \boldsymbol{B}\rangle_{\mathrm{Tr}} = \mathrm{Tr}(\boldsymbol{A}^T \boldsymbol{B})
\end{aligned}$$

For the second sum :

$$\begin{aligned}
\sum_{k,j}^{K} \boldsymbol{\omega}_k^T \boldsymbol{\omega}_j \boldsymbol{\alpha}_k^T S \boldsymbol{\alpha}_j &= \sum_{k,j} \Big(\boldsymbol{\omega}_k^T \boldsymbol{\omega}_j\Big)\Big(\boldsymbol{\beta}_j^T \boldsymbol{\beta}_k\Big) \\
&= \mathrm{Tr}\Big(\sum_{k,j} \Big((\boldsymbol{\omega}_k^T \boldsymbol{\omega}_j)\boldsymbol{\beta}_j^T\Big)\boldsymbol{\beta}_k\Big)\Big) \\
&= \mathrm{Tr}(\Big(\sum_{k,j} \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T \boldsymbol{\omega}_j \boldsymbol{\beta}_j^T\Big) \\
&= ||\sum_k \boldsymbol{\omega}_k \boldsymbol{\beta}_k^T||_{\mathrm{Tr}}^2 \text{ with } ||\boldsymbol{A}||_{\mathrm{Tr}} = \sqrt{\mathrm{Tr}(\boldsymbol{A}^T \boldsymbol{A})} \\
&= ||\sum_k \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T||_{\mathrm{Tr}}^2
\end{aligned}$$

Then we have :

$$\underset{K,\theta_{\leftrightarrow}}{\arg\min}\ \frac{1}{n} g(\boldsymbol{\alpha}, \boldsymbol{\omega}) \ = \underset{K, \boldsymbol{\alpha} = \boldsymbol{S}^{-1/2}\boldsymbol{\gamma}, \boldsymbol{\omega}}{\arg\min} \Big\| \boldsymbol{S}^{-1/2}\boldsymbol{N} - \sum_{k=1}^{K} \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T \Big\|_{\mathrm{Tr}}^2$$

The solution of such problems is given by the paper Eckart & Young (1936). For any $K \leq R := \mathrm{rank}(\boldsymbol{S}^{-1/2}\boldsymbol{N})$, the best option is given by $\sum_{k=1}^{K} \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T = \sum_{r=1}^{K} \lambda_r \boldsymbol{v}_r \boldsymbol{e}_r^T$. Thus we have that $\sum_k^R \boldsymbol{\omega}_k \boldsymbol{\alpha}_k^T = \boldsymbol{S}^{-1/2} \sum_k^R \lambda_k \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T = \boldsymbol{S}^{-1}\boldsymbol{N}$.

We now consider the matrix $\boldsymbol{S}^{-1/2}\boldsymbol{N}$. The minimization also yields the following properties at the optimum:

$$\textbf{for } k \neq j, \quad \langle \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T, \boldsymbol{\beta}_j \boldsymbol{\omega}_j^T \rangle_{\mathrm{Tr}} = 0$$

$$\begin{aligned}
||\boldsymbol{S}^{-1/2}\boldsymbol{N} - \sum_{k=1}^{K} \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T||_{\mathrm{Tr}}^2 &= \sum_{r=K+1}^{R} \lambda_r^2 \\
&= ||\boldsymbol{S}^{-1/2}\boldsymbol{N}||_{\mathrm{Tr}}^2 - ||\sum_{k=1}^{K} \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T||_{\mathrm{Tr}}^2
\end{aligned}$$

Furthermore :

$$\begin{aligned}
\frac{1}{n}||{\boldsymbol{V}_{\text{goal}}}_{proj}^{l} - \boldsymbol{V}(\theta_{\leftrightarrow}^{K,*})||_{\mathrm{Tr}}^2 &= \frac{1}{n}||{\boldsymbol{V}_{\text{goal}}}_{proj}^{l}||_{\mathrm{Tr}}^2 + ||\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N} - \sum_k \boldsymbol{\beta}_k \boldsymbol{\omega}_k^T||^2 - ||\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N}||_{\mathrm{Tr}}^2 \\
&= \sum_{r=K+1}^{R} \lambda_r^2 + \frac{1}{n}||{\boldsymbol{V}_{\text{goal}}}_{proj}^{l}||_{\mathrm{Tr}}^2 - ||\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N}||_{\mathrm{Tr}}^2 \\
&= -\sum_{r=1}^{K} \lambda_r^2 + \frac{1}{n}||{\boldsymbol{V}_{\text{goal}}}_{proj}^{l}||_{\mathrm{Tr}}^2
\end{aligned}$$

We note ${\boldsymbol{V}_{\text{goal}}}^l_{proj}(\delta \boldsymbol{W}_l^*) := {\boldsymbol{V}_{\text{goal}}}^l_{proj}$. Suppose that $\boldsymbol{B}_{l-1}$ and $\boldsymbol{B}_{l-2}$ are orthogonal for the trace scalar product, then when adding the new neurons, the impact on the global loss is :

$$\mathcal{L}(f_{\theta \oplus \theta^K_{\leftrightarrow}}) = \frac{1}{n}\sum_{i=1}^{n} L(f_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i) - \frac{\gamma}{\eta}\frac{1}{n}\sigma_l'(0)\Big\langle \boldsymbol{V}^l(\theta^{K,*}_{\leftrightarrow}), {\boldsymbol{V}_{\text{goal}}}^l_{proj} \Big\rangle_{\text{Tr}} + o(1)$$

We also have the following property :

$$\begin{aligned}
&\arg\min_{\theta^K_{\leftrightarrow}} \Big\{ \frac{1}{n} ||{\boldsymbol{V}_{\text{goal}}}^l_{proj} - \boldsymbol{V}^l(\theta^K_{\leftrightarrow})||^2_{\text{Tr}} \Big\} \\
&= \arg\min_{H \geq 0} \quad \arg\min_{\theta^K_{\leftrightarrow}, ||\boldsymbol{V}^l(\theta^K_{\leftrightarrow})||_{\text{Tr}} = H} \Big\{ \frac{1}{n} ||{\boldsymbol{V}_{\text{goal}}}^l_{proj} - \boldsymbol{V}^l(\theta^K_{\leftrightarrow})||^2_{\text{Tr}} \Big\} \\
&= \arg\min_{H \geq 0} \quad \arg\min_{\theta^K_{\leftrightarrow}, ||\boldsymbol{V}^l(\theta^K_{\leftrightarrow})||_{\text{Tr}} = H} \Big\{ -\frac{2}{n} \Big\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}, \boldsymbol{V}^l(\theta^K_{\leftrightarrow}) \Big\rangle_{\text{Tr}} + \frac{1}{n} ||\boldsymbol{V}^l(\theta^K_{\leftrightarrow})||^2_{\text{Tr}} \Big\} \\
&= \arg\min_{H \geq 0} \quad \arg\min_{\theta^K_{\leftrightarrow}, ||\boldsymbol{V}^l(\theta^K_{\leftrightarrow})||_{\text{Tr}} = H} \Big\{ -\frac{2}{n} \Big\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}, \boldsymbol{V}^l(\theta^K_{\leftrightarrow}) \Big\rangle_{\text{Tr}} + \frac{1}{n} H^2 \Big\} \\
&= \arg\min_{H \geq 0} \quad \arg\min_{\theta^K_{\leftrightarrow}, ||\boldsymbol{V}^l(\theta^K_{l\leftrightarrow})^*||_{\text{Tr}} = 1} \Big\{ -H \Big\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}, \boldsymbol{V}^l(\theta^K_{l\leftrightarrow})^* \Big\rangle_{\text{Tr}} + \frac{1}{2} H^2 \Big\}
\end{aligned}$$

with $\boldsymbol{V}^l(\theta^K_{\leftrightarrow})^*$ the solution of the second $\arg\min$ (ie for $H = 1$).
Then the norm minimizing the first argmin is given by :

$$H^* = \Big\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}, \boldsymbol{V}^l(\theta^K_{\leftrightarrow})^* \Big\rangle_{\text{Tr}}$$

Furthermore

$$\min_{\theta^K_{l\leftrightarrow l}} \Big\{ \frac{1}{n} ||{\boldsymbol{V}_{\text{goal}}}^l_{proj} - \boldsymbol{V}^l(\theta^K_{l\leftrightarrow l})||^2_{\text{Tr}} \Big\} = -\sum_{r=1}^{K} \lambda_r^2 + \frac{1}{n} ||{\boldsymbol{V}_{\text{goal}}}^l_{proj}||^2_{\text{Tr}} \tag{15}$$

$$\min_{\theta^K_{\leftrightarrow}} \Big\{ \frac{1}{n} ||{\boldsymbol{V}_{\text{goal}}}^l_{proj} - \boldsymbol{V}^l(\theta^K_{l\leftrightarrow l})||^2_{\text{Tr}} \Big\} = -\frac{1}{n} {H^*}^2 + \frac{1}{n} ||{\boldsymbol{V}_{\text{goal}}}^l_{proj}||^2_{\text{Tr}} \tag{16}$$

$$\implies H^* = \Big\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}, \boldsymbol{V}^l(\theta^K_{\leftrightarrow})^* \Big\rangle_{\text{Tr}} = \sqrt{\sum_{r=1}^{K} \lambda_r^2} \times \sqrt{n} \tag{17}$$

$$\boldsymbol{V}^l(\theta^{K,*}_{\leftrightarrow}) = H^* \boldsymbol{V}^l(\theta^K_{\leftrightarrow})^* \tag{18}$$

$$\Big\langle \boldsymbol{V}^l(\theta^{K,*}_{\leftrightarrow}), {\boldsymbol{V}_{\text{goal}}}^l_{proj} \Big\rangle_{\text{Tr}} = H^* \times \Big\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}, \boldsymbol{V}^l(\theta^K_{\leftrightarrow})^* \Big\rangle_{\text{Tr}} = {H^*}^2 \tag{19}$$

where the last equality is given by the optimisation of $||\boldsymbol{S}^{-\frac{1}{2}}\boldsymbol{N} - \sum_{k=1}^{K} \boldsymbol{u}_k \boldsymbol{\omega}_k^T||^2_{\text{Tr}}$. So minimizing the scalar product $-\Big\langle {\boldsymbol{V}_{\text{goal}}}^l_{proj}(\delta \boldsymbol{W}_l^*), \boldsymbol{V}^l(\theta^K_{\leftrightarrow})^* \Big\rangle_{\text{Tr}}$ for fixed norm of $\boldsymbol{V}^l(\theta^K_{\leftrightarrow})$ is equivalent to minimizing the norm $||{\boldsymbol{V}_{\text{goal}}}^l_{proj}(\delta \boldsymbol{W}_l^*) - \boldsymbol{V}^l(\theta^K_{\leftrightarrow})||^2_{\text{Tr}}$.

$$\mathcal{L}(f_{\theta \oplus \theta^K_{\leftrightarrow}}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i) - \frac{1}{\eta}\sigma_l'(0)\sum_{r=1}^{K} \lambda_k^2 + o(1)$$

For convolutional layers, the same reasoning can be applied. Consider one adds one convolution layer, ie $K = 1$. Use $\boldsymbol{B}_i^c$ defined in the first proof and $\boldsymbol{T}_j$ the linear application selecting the

activities for the $j-$pixel , then one has to minimize the expression :

$$\begin{aligned}
g(\theta^1_{\leftrightarrow}) &= \overset{\text{out channels}}{\sum_m} \overset{\text{examples}}{\sum_i} \overset{\text{preactivity size}}{\sum_{j=1}} (\boldsymbol{\omega}_m^T \boldsymbol{T}_j \boldsymbol{B}^c \boldsymbol{\alpha} - {\boldsymbol{V}_{\text{goal}}}^i_{j,m})^2 \\
&= \sum_m \sum_i \sum_{j=1} (\boldsymbol{\omega}_m^T \boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha})^2 - 2\boldsymbol{\omega}_m^T \boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha} {\boldsymbol{V}_{\text{goal}}}^i_{j,m} + C \\
&= \sum_m \sum_i \sum_{j=1} \text{Tr}(\boldsymbol{\omega}_m^T \boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha})^2 - 2\boldsymbol{\omega}_m^T \boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha} {\boldsymbol{V}_{\text{goal}}}^i_{j,m} + C \\
&= \sum_m \sum_i \sum_{j=1} \text{Tr}(\boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha} \boldsymbol{\omega}_m^T)^2 - 2\boldsymbol{\omega}_m^T \boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha} {\boldsymbol{V}_{\text{goal}}}^i_{j,m} + C
\end{aligned}$$

for some constant C. We have the property that $\langle \boldsymbol{T}_j^T, \boldsymbol{B}_i^c \boldsymbol{\alpha} \boldsymbol{\omega}_m^T \rangle^2_{\text{Tr}} = \text{Tr}(\boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha} \boldsymbol{\omega}_m^T)^2 = ||\boldsymbol{T}_j \boldsymbol{B}_i^c \boldsymbol{\alpha} \boldsymbol{\omega}_m^T||^2_{\text{Tr}}$. Ignoring the constant C:

$$\begin{aligned}
g(\theta^1_{\leftrightarrow}) &= \sum_m \text{Tr}(\boldsymbol{\omega}_m \boldsymbol{\alpha}^T \Big(\sum_i \boldsymbol{B}_i^{cT} \sum_j \boldsymbol{T}_j^T \boldsymbol{T}_j \boldsymbol{B}_i^c \Big) \boldsymbol{\alpha} \boldsymbol{\omega}_m^T) - 2\boldsymbol{\omega}_m^T \sum_{i,j} \boldsymbol{T}_j \boldsymbol{B}_i^c {\boldsymbol{V}_{\text{goal}}}^i_{j,m} \boldsymbol{\alpha} \\
&= \sum_m \boldsymbol{\alpha}^T \Big(\sum_i \boldsymbol{B}_i^{cT} \sum_j \boldsymbol{T}_j^T \boldsymbol{T}_j \boldsymbol{B}_i^c \Big) \boldsymbol{\alpha} \boldsymbol{\omega}_m^T \boldsymbol{\omega}_m - 2\boldsymbol{\omega}_m^T \sum_{i,j} \boldsymbol{T}_j \boldsymbol{B}_i^c {\boldsymbol{V}_{\text{goal}}}^i_{j,m} \boldsymbol{\alpha} \\
&= \boldsymbol{\alpha}^T \Big(\sum_i \boldsymbol{B}_i^{cT} \sum_j \boldsymbol{T}_j^T \boldsymbol{T}_j \boldsymbol{B}_i^c \Big) \boldsymbol{\alpha} \boldsymbol{\omega}^T \boldsymbol{\omega} - 2 \sum_m \boldsymbol{\omega}_m^T \sum_{i,j} \boldsymbol{T}_j 1^T_{full} {\boldsymbol{V}_{\text{goal}}}^i 1_{j,m} \boldsymbol{B}_i^c \boldsymbol{\alpha} \\
&= \boldsymbol{\alpha}^T \boldsymbol{S} \boldsymbol{\alpha} \boldsymbol{\omega}^T \boldsymbol{\omega} - 2 \sum_m \boldsymbol{\omega}_m^T \sum_i \boldsymbol{F}_i^m \boldsymbol{B}_i^c \boldsymbol{\alpha} \\
&= \boldsymbol{\alpha}^T \boldsymbol{S} \boldsymbol{\alpha} \boldsymbol{\omega}^T \boldsymbol{\omega} - 2\boldsymbol{\omega} \boldsymbol{N} \boldsymbol{\alpha}
\end{aligned}$$

with $T_j = \begin{pmatrix} \overbrace{0 \ . \ 0}^{j-1+\lfloor j/(p-1) \rfloor} & 1 & 0 & . & . & . & . \\ \overbrace{0 \ . \ 0}^{j-1+\lfloor j/(p-1) \rfloor} & 0 & 1 & . & . & . & . \\ \overbrace{0 \ . \ 0}^{j-1+\lfloor j/(p-1) \rfloor} & \overbrace{0 \ . \ 0}^{31} & 1 & 0 & . & . & . \\ \overbrace{0 \ . \ 0}^{j-1+\lfloor j/(p-1) \rfloor} & \overbrace{0 \ . \ 0}^{31} & 0 & 1 & 0 & . & . \end{pmatrix}$ for a kernel size equal to $(2, 2)$. □

C.3 PROPOSITION AND REMARK 3.3

Suppose $\boldsymbol{S}$ is semi definite, we note $\boldsymbol{S} = \boldsymbol{S}^{\frac{1}{2}} \boldsymbol{S}^{\frac{1}{2}}$. Solving (7) is equivalent to find the K first eigenvectors $\boldsymbol{\alpha}_k$ associated to the K largest eigenvalues λ of the generalized eigenvalue problem :

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

Proof

This is equivalent to maximizing the following generalized Rayleigh quotient (which is solvable by the LOBPCG technique):

$$\begin{aligned}
\boldsymbol{\alpha}^* &= \max_x \frac{\boldsymbol{\alpha}^T \boldsymbol{N}\boldsymbol{N}^T \boldsymbol{\alpha}}{\boldsymbol{\alpha}^T \boldsymbol{S} \boldsymbol{\alpha}} \\
\boldsymbol{p}^* &= \max_{\boldsymbol{p} = \boldsymbol{S}^{1/2} \boldsymbol{\alpha}} \frac{\boldsymbol{p}^T \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{N}\boldsymbol{N}^T \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{p}}{\boldsymbol{p}^T \boldsymbol{p}} \\
\boldsymbol{p}^* &= \max_{||\boldsymbol{p}||=1} ||\boldsymbol{N}^T \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{p}|| \\
\boldsymbol{\alpha}^* &= \boldsymbol{S}^{-\frac{1}{2}} \boldsymbol{p}^*
\end{aligned}$$

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

We prove second corollary 3.2 by induction. For $m = m' = 1$:

$$\boldsymbol{a}_l(\boldsymbol{x})^{t+1} = \boldsymbol{a}_l(\boldsymbol{x})^t + \boldsymbol{V}(\theta_{\leftrightarrow}^{1,*}, \boldsymbol{x})\gamma + o(\gamma)$$
$$\boldsymbol{v}_{\text{goal}}{}^{l,t+1}(\boldsymbol{x}) = \boldsymbol{v}_{\text{goal}}{}^{l,t}(\boldsymbol{x}) + \nabla_{a_l(\boldsymbol{x})}\mathcal{L}(f_\theta(\boldsymbol{x}), \boldsymbol{y})^T \boldsymbol{v}(\theta_{\leftrightarrow}^{1,*}, \boldsymbol{x})\gamma + o(\gamma)$$

Adding the second neuron we obtain the minimization problem:

$$\arg\min_{\boldsymbol{\alpha}_2, \boldsymbol{\omega}_2} ||\boldsymbol{V}_{\text{goal}}{}^{l,t} - \boldsymbol{V}^l(\boldsymbol{\alpha}_2, \boldsymbol{\omega}_2)||_{\text{Tr}} + o(1)$$

□

C.4 About equivalence of quadratic problems

Problems 6 and 5 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:

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

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.

We chose this two-step process as it intuitively relates to the spirit of improving upon a standard gradient descent: we aim at adding neurons that complement what the other ones have already done. This choice is debatable and one could solve the joint problem instead, with 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.1), 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 $\boldsymbol{V}^l(\theta_{\leftrightarrow}^K)$ 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.5 Section *Theory behind Greedy Growth* with proofs

One might wonder whether a greedy approach on layer growth might get stuck in a non-optimal state. We derive the following series of propositions in this regard. Since in this work we add

neurons layer per layer independently, we study here the case of a single hidden layer network, to spot potential layer growth issues. For the sake of simplicity, we consider the task of least square regression towards an explicit continuous target f^*, defined on a compact set. That is, we aim at minimizing the loss:

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

where $f(x)$ is the output of the neural network and $\mathcal{D}$ is the training set.

Proposition C.2 (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 γ, and for any activation function σ provided it is not a polynomial, then there exists a neural network $\widehat{g}$ with one hidden layer (possibly quite large when γ is small) and with this activation function σ, such that

$$\forall x, \|g(x) - g^*(x)\| \leqslant \gamma$$

We apply this theorem 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 σ, that we will suppose to be continuous for the sake of simplicity. We will suppose 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). We choose $\gamma = \frac{1}{10}\|f^* - f\|_{L2}$, where $\langle a|\ b\rangle_{L2} = \frac{1}{|\boldsymbol{\omega}|}\int_{x \in \boldsymbol{\omega}} a(x)\, b(x)\, dx$. This way we obtain a one-hidden-layer neural network g with activation function σ such that:

$$\forall x \in \boldsymbol{\omega},\ \ -\gamma \leqslant g(x) - g^*(x) \leqslant \gamma$$

$$\forall x \in \boldsymbol{\omega},\ \ g(x) = f^*(x) - f(x) + a(x)$$

with $\forall x \in \boldsymbol{\omega},\ |a(x)| \leqslant \gamma$.

Then:

$$\forall x \in \boldsymbol{\omega},\ \ f^*(x) - (f(x) + g(x)) = -a(x)$$

$$\forall x \in \boldsymbol{\omega},\ \ (f^*(x) - h(x))^2 = a^2(x) \tag{20}$$

with h being the function corresponding to a neural network consisting in concatenating the hidden layer neurons of f and g, and consequently summing their outputs.

$$\|f^* - h\|_{L2}^2 = \|a\|_{L2}^2$$

$$\|f^* - h\|_{L2}^2 \leqslant \gamma^2 = \frac{1}{100}\|f^* - f\|_{L2}^2$$

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 $\gamma = \frac{1}{10}\sqrt{\frac{1}{|\mathcal{D}|}\sum_{x \in \mathcal{D}} \|f(x) - f^*(x)\|^2}$, as Equation (20) then yields:

$$\sum_{x \in \mathcal{D}} (f^*(x) - h(x))^2 = \sum_{x \in \mathcal{D}} a^2(x) \leqslant \frac{1}{100}\sum_{x \in \mathcal{D}} (f^*(x) - f(x))^2$$

which proves the proposition as stated.

For more general losses, one can consider order-1 (linear) developpment of the loss and ask for a network g that is close to (the opposite of) the gradient of the loss.

□

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 σ 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 σ 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.3 (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 $L2$ 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 $L2$ regression). That is, denoting by $a_i\sigma(\boldsymbol{W}_i \cdot \boldsymbol{x})$ these N neurons:

$$\big\langle \sum_{i=1}^{N} a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x}) \;\big|\; g \big\rangle_{L2} = K < 0$$

i.e.

$$\sum_{i=1}^{N} \langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\; g \rangle_{L2} = K < 0$$

Now, by contradiction, if there existed no neuron i among these ones such that

$$\langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x}) \mid\; g \rangle_{L2} \leqslant \frac{1}{N}K$$

then we would have:

$$\forall i \in [1, N], \;\; \langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\; g \rangle_{L2} > \frac{1}{N}K$$

$$\sum_{i=1}^{N} \langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\; g \rangle_{L2} > K$$

hence a contradiction. Then necessarily at least one of the N neurons satisfies

$$\langle a_i\sigma(\boldsymbol{w}_i \cdot \boldsymbol{x})|\; g \rangle_{L2} \leqslant \frac{1}{N}K < 0$$

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, our greedy approach will not get stuck in a situation where one would need to add many neurons simultaneously to decrease the loss: it is always feasible by a single neuron. On can express a lower bound on how much the loss has improved (for the best such neuron), but it not a very good one without further assumptions on f.

Proposition C.4 (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 consequently manipulates infinitesimal quantities. Though we perform line search in a second step and consequently add non-infinitesimal neurons, our first optimization problem relies on the linearization of the activation function by requiring the added neuron to have infinitely small input weights, without which it would be much harder to solve. This proposition confirms that such neuron does exist indeed.

D TECHNICAL DETAILS

D.1 BATCH SIZE TO ESTIMATE THE NEW NEURON AND THE BEST UPDATE

In this section we study the variance of the matrices $\delta \mathbf{W}_l{}^*$and $\boldsymbol{S}^{-1/2}\boldsymbol{N}$ 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 (11) and in (14), as we are trying to regress the desired update noted Y onto the span of the activities noted X. We suppose we have the following setting :

$$Y \sim AX + \varepsilon, \;\; \varepsilon \sim \mathcal{N}(0, \sigma^2), \;\; \mathbb{E}[\varepsilon|X] = 0$$

where the (X_i, Y_i) are *i.i.d.* and A is the oracle for $\delta \mathbf{W}_l{}^*$ or matrix $\boldsymbol{S}^{-1/2}\boldsymbol{N}$. If Y is multidimensional, the the total variance of our estimator can be seen as the sum of the variances of the estimator on each dimension of Y.

We now suppose that $Y \in \mathbb{R}$. The estimator $\hat{A} := (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{Y}^T$ has variance $\mathrm{var}(\hat{A}) = \sigma^2(\mathbf{X}\mathbf{X}^T)^{-1}$. If n is large, and if matrix $\frac{1}{n}\mathbf{X}\mathbf{X}^T \to Q$, with Q non singular, then, asymptotically, we have $\hat{A} \sim \mathcal{N}(A, \sigma^2 \frac{Q^{-1}}{n})$, which is equivalent to $(\hat{A} - A)\sqrt{\frac{n}{\sigma}}Q^{1/2} \sim \mathcal{N}(0, I)$. Then $||(\hat{A} - A)\sqrt{\frac{n}{\sigma}}Q^{1/2}||^2 \sim \chi^2(k)$ where k is the dimension of X. It follows that $\mathbb{E}\left[||(\hat{A} - A)Q^{1/2}||^2\right] = \frac{k\sigma}{n}$ and as $Q^{1/2}{Q^{1/2}}^T$ is positive definite, we conclude that $\mathrm{var}(\hat{A}) \leqslant \frac{k\sigma}{n\lambda_{\min}(Q)}$.

In practice and to keep the variance of our estimators stable during architecture growth, for the estimation of the best neuron to add we use batch size

$$n \propto \frac{(SW)^2}{P},$$

with the notations defined in Figure 12, since the matrices we estimate have side size SW and that each input sample contains P values, i.e. P quantities that each play the role of X here.

D.2 BATCH SIZE FOR LEARNING

We adjust the batch size for gradient descent as follow : 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 (ie 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.3 NORMALIZATION

D.3.1 FIGURES 5 AND 16 : USUAL NORMALIZATION

For the GradMax method of figure 5 and 16, before adding the new neurons to the architecture, we normalize the out-going weight of the new neurons according to Evci et al. (2022), ie :

$$\alpha_k^* \leftarrow 0 \tag{21}$$

$$\text{for 5} \quad \omega_k^* \leftarrow \omega_k^* \times \frac{1e-3}{\sqrt{||\{\omega_j^*\}_{j=1}^{n_d}||_2^2/n_d}} \tag{22}$$

$$\text{for 16} \quad \omega_k^* \leftarrow \omega_k^* \times \sqrt{\frac{1e-3}{||\{\omega_j^*\}_{j=1}^{n_d}||_2^2/n_d}} \tag{23}$$

For TINY method of both figures, the previous normalization process is mimicked by normalizing the in and out going weights by theirs norms and multiplying them by $\sqrt{1e-3}$, ie :

$$\alpha_k \leftarrow \alpha_k^* \times \sqrt{\frac{1e-3}{||\{\alpha_j^*\}_{j=1}^{n_d}||_2^2/n_d}} \tag{24}$$

$$\omega_k \leftarrow \omega_k^* \times \sqrt{\frac{1e-3}{||\omega_j^*\}_{j=1}^{n_d}||_2^2/n_d}} \tag{25}$$

D.3.2 FIGURE 8 : AMPLITUDE FACTOR

For the Random and the TINY methods of figure 8, we first normalize the parameters as :

For the new neurons

$$\alpha_k^* \leftarrow \alpha_k^* \times \frac{1}{\sqrt{||\{\alpha_j^*\}_{j=1}^{n_d}||_2^2/n_d}}$$

$$\omega_k^* \leftarrow \alpha_k^* \times \frac{1}{\sqrt{||\{\omega_j^*\}_{j=1}^{n_d}||_2^2/n_d}}$$

For the best update

$$\boldsymbol{W}^* \leftarrow \boldsymbol{W}^* \times \frac{1}{\sqrt{||\boldsymbol{W}^*||_2^2/n_d}}$$

Then, we multiply them by the amplitude factor γ^* :

For the new neurons :

$$\alpha_k^*, \ \ \omega_k^* \ \ \leftarrow \alpha_k^*\gamma^*, \omega_k^*\gamma^*$$

$$\gamma^* := \underset{\gamma \in [-L,L]}{\arg\min} \sum_i \mathcal{L}(f_{\theta \oplus \gamma \theta_{\leftrightarrow}^K}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

For the best update :

$$\boldsymbol{W}_l^* \ \ \leftarrow \gamma^* \boldsymbol{W}_l$$

$$\gamma^* := \underset{\gamma \in [-L,L]}{\arg\min} \sum_i \mathcal{L}(f_{\theta + \gamma \boldsymbol{W}^*}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

Where the operation $\gamma\theta_{\leftrightarrow}^{K^*} = (\gamma\alpha_k^*, \gamma\omega_k^*)_k^K$ is the concatenation of the neural network with the new neurons and $\theta + \gamma \boldsymbol{W}^*$ is the update of one layer with its best update. The batch on which γ^* is computed is different from the one used to estimate the new parameters and its size is fixed to 1000 for all experiments.

D.4 FULL ALGORITHM

In this section we describe in detail the pseudo code to plot 5 and 8. The function NewNeurons(l), in Algorithm 2, computes the new neurons defined at Proposition 3.2 for layer l sorted by decreasing eigenvalues. The function BestUpdate(l), in Algorithm 4 computes the best update at Proposition 3.1 for layer l.

Algorithm 1: Algorithm to plot Figure 5 and 8.

```
for each method [TINY, MethodToCompareWith] do
    Start from neural network N with initial structure ι ∈ {1/4, 1/64};
    while N architecture doesn't match ResNet18_f do
        for d in {depths to growth} do
            θ_↔^{K*} = NewNeurons(d, method = method) ;
            Normalize θ_↔^{K*} according to D.3;
            Add the neurons at d ;
            Train N for Δt epochs ;
            Save model N and its performance ;
        end
    end
end
```

Algorithm 2: NewNeurons

Data: l, method $= TINY$
Result: Best neurons at l
if *method* $= TINY$ **then**
  $\delta \mathbf{W}_l =$ BestUpdate$(l+1)$;
  $\boldsymbol{S}, \boldsymbol{N} =$ MatrixSN$(l-1, l+1, \delta \mathbf{W}_l = \delta \mathbf{W}_l)$;
  Compute the SVD of $\boldsymbol{S} := U \Sigma U^T$;
  Compute the SVD of $\boldsymbol{U} \sqrt{\Sigma}^{-1} \boldsymbol{U} \boldsymbol{N} := \boldsymbol{A} \Lambda \boldsymbol{\Omega}$;
  Use the columns of $\boldsymbol{A}$, the ligns of $\boldsymbol{\Omega}$ and the diagonal of Λ to construct the new neurons of Prop. 3.2;
else if *method* $= GradMax$ **then**
  $\delta \mathbf{W}_l = None$;
  $\_, N =$ MatrixSN$(l-1, l+1, \delta \mathbf{W}_l = \delta \mathbf{W}_l)$;
  Compute the SVD of $N^T N$;
  Use the eigenvectors to define the new out-going weights ;
  Set the new in-going weight to 0;
else if *method* $= Random$ **then**
  $(\alpha_k, \omega_k)_{k=1}^{n_d} \sim \mathcal{N}(0, Id)$;
end

Algorithm 3: MatrixSN

Data: p_1, p_2 (layer indexes), $\delta \mathbf{W}_l =$ None
Result: Construct matrices $\boldsymbol{S}$ and $\boldsymbol{N}$
Take a minibatch $\mathbf{X}$ of size $\propto \frac{(SW)^2}{P}$;
Propagate and backpropagate $\mathbf{X}$;
Compute $\mathbf{V}_{goal}$ at p_2, ie $-\frac{\partial \mathcal{L}^{tot}}{\partial \mathbf{A_{p_2}}}$;
if $\delta \mathbf{W}_l \neq$ *None* **then**
  $\mathbf{V}_{goal} -= \delta \mathbf{W}_l \boldsymbol{B}_{p_1}$
end
$\mathbf{S}, \mathbf{N} = \boldsymbol{B}_{p_1} \boldsymbol{B}_{p_1}{}^T, \boldsymbol{B}_{p_1} \boldsymbol{V}_{\text{goal}}{}^T$;

Algorithm 4: BestUpdate

Data: l, index of a layer
Result: Best update at l
Take a minibatch $\mathbf{X}$ of size $\propto \frac{(SW)^2}{P}$;
Compute $(\mathbf{S}, \mathbf{N})$ with the function S_N(l, l);
$\delta \mathbf{W}_l = N^T S^{-1}$;

D.5 Computational complexity

We estimate here the computational complexity of the above algorithm for architecture growth.

Theoretical estimate. We use the following notations:

- number of layers: L
- layer width, or number of kernels if convolutions: W (assuming for simplicity that all layers have same width or kernels)
- number of pixels in the image: P ($P = 1$ for fully-connected)
- kernel filter size: S ($S = 1$ if fully-connected)
- minibatch size used for standard gradient descent: M
- minibatch size used for new neuron estimation: M'
- minibatch size used in the line-search to estimate amplitude factor: M''
- number of classical gradients steps performed between 2 addition tentatives: T

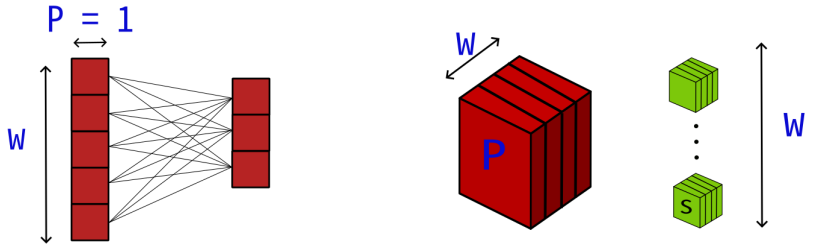

Figure 12: Notation and size for convolutional and linear layers

Complexity, estimated as the number of basic operations, cumulated over all calls of the functions:

- of the standard training part: $TMLW^2SP$
- of the computation of matrices of interest (function MatrixSN): $LM'(SW)^2P$
- of SVD computations (function NewNeurons): $L(SW)^3$
- of line-searches (function AmplitudeFactor): $L^2M''W^2SP$
- of weight updates (function BestUpdate): LSW

The relative added complexity w.r.t. the standard training part is thus:

$$M'S\,/\,TM \;+\; S^2W\,/\,TMP \;+\; M''L\,/\,TM \;+\; 1\,/\,WTMP.$$

SVD cost is negligible. The relative cost of the SVD w.r.t. the standard training part is $S^2W\,/\,TMP$. In the fully-connected network case, $S = 1$, $P = 1$, and the relative cost of the SVD is then W/TM. It is then negligible, as layer width W is usually much smaller than TM, which is typically 10×100 for instance. In the convolutional case, $S = 9$ for 3×3 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.

Likewise, the update of existing weights using the "optimal move" (already computed as a by-product) is computationally negligible, and the relative cost of the line searches is limited as long as the network is not extremely deep ($L < TM/M$").

On the opposite, the estimation of the matrices (to which SVD is applied) can be more ressource demanding. The factor $M'S/TM$ can be large if the minibatch size M' needs to be large for statistical significance reasons. One can show that an upper bound to the value required for M' to ensure estimator precision (see Appendix D.1) is $(SW)^2/P$. In that case, if $W > \sqrt{TMP/S^3}$, these matrix estimations will get costly. In the fully-connected network case, this means $W > \sqrt{TM} \approx 30$ for $T = 10$ and $M = 100$. In the convolutional case, this means $W > \sqrt{TMP/S^3} \approx 30$ for CIFAR and ≈ 300 for ImageNet. We are working on finer variance estimation and on other types of estimators to decrease M' and consequently this cost. Actually $(SW)^2/P$ is just an upper bound on the value required for M', which might be much lower, depending on the rank of computed matrices.

In practice. In practice the cost of a full training with our 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 we get layer width hyper-optimization for free.

E Additional experimental results and remarks

E.1 ResNet18 on CIFAR-100

Figures. In all plots the black line represents the average performance over two independent runs, and the colored regions indicate the confidence interval.

technical details of figure 5 and 8 The experiment were performed on 1 GPU. The optimizer is SGD($lr = 1e-2$) with the starting batch size 32 D.2. At each depth l we set the number n_l of neurons to be added at this depth 2. These numbers do not depend on the starting architecture and have been chosen such that each depth will reach its final width with the same number of layer extensions. For the initial structure $s = 1/4$, resp. $1/64$, we set the number of layer extensions to 16, resp. 21, such that at depth 2 (named Conv2 in Table 3), $n_2 = (\text{Size}_2^{final} - \text{Size}_2^{start})/\text{nb of layer extensions} = (64 - 16)/16 = (64 - 1)/21 = 3$. The initial architecture is described in Table 3.

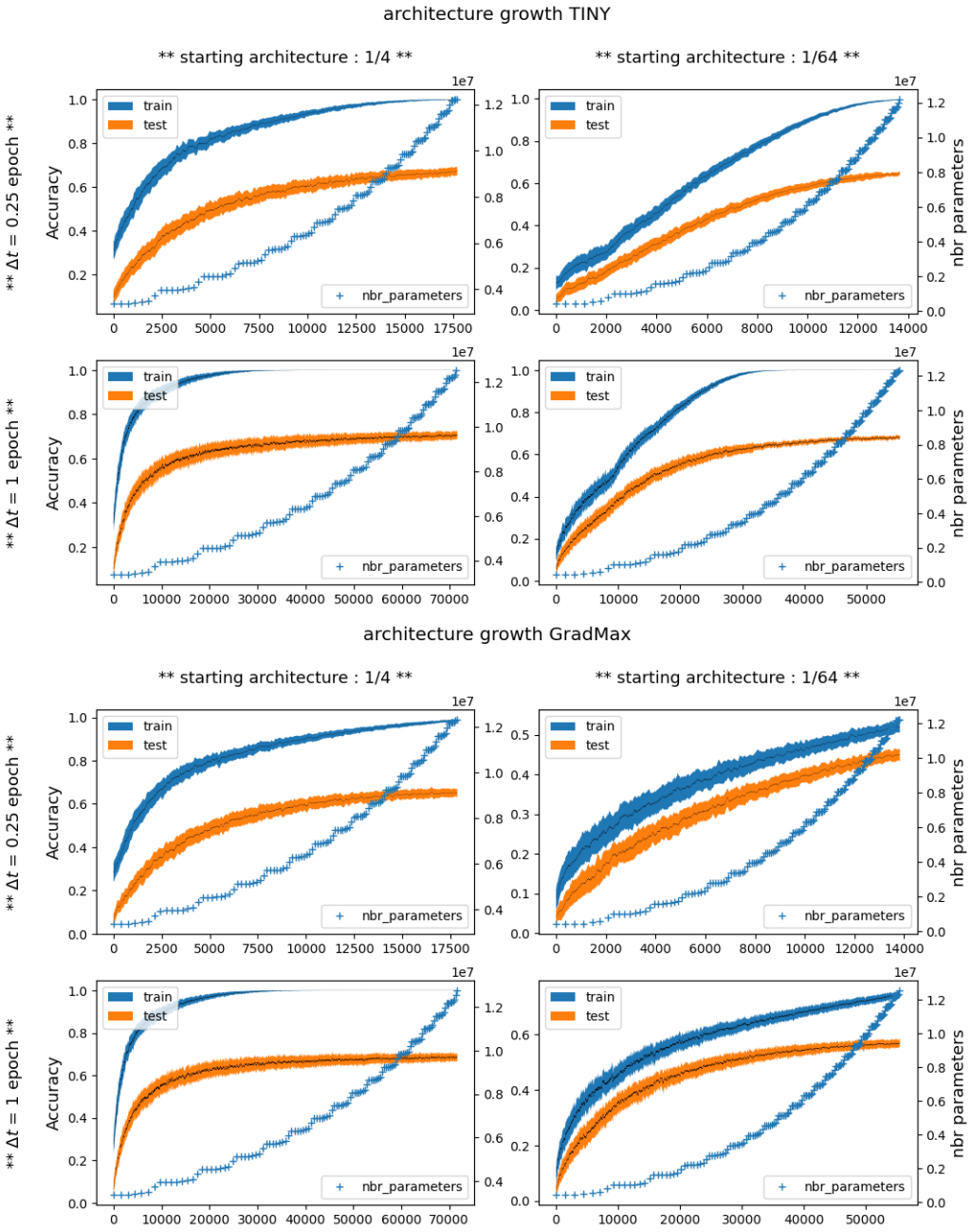

Figure 13: Accuracy and number of parameters during architecture growth for methods TINY and GradMax as a function of gradient step.

.

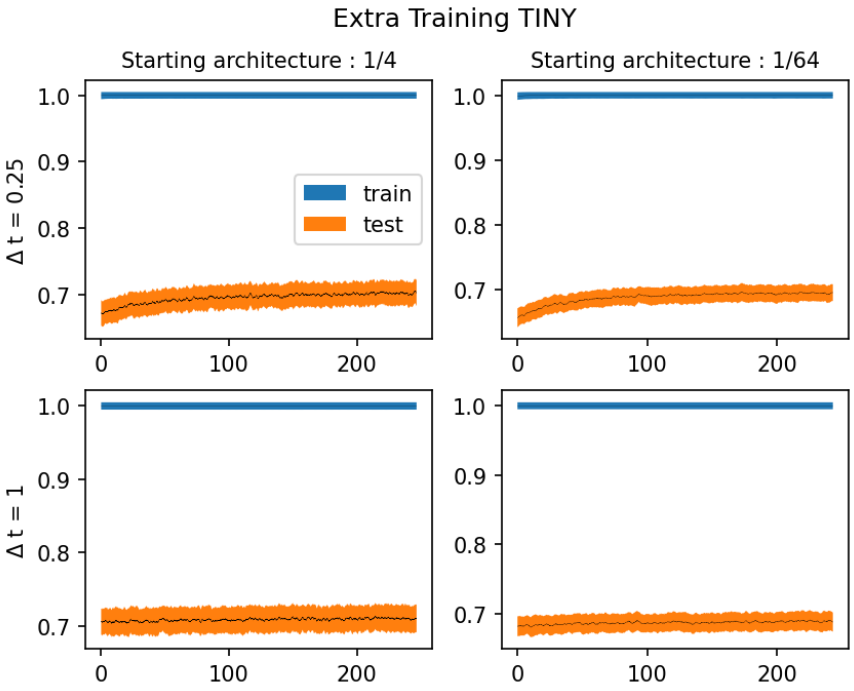

Figure 14: Accuracy as a function of the number of epochs during extra training for TINY.

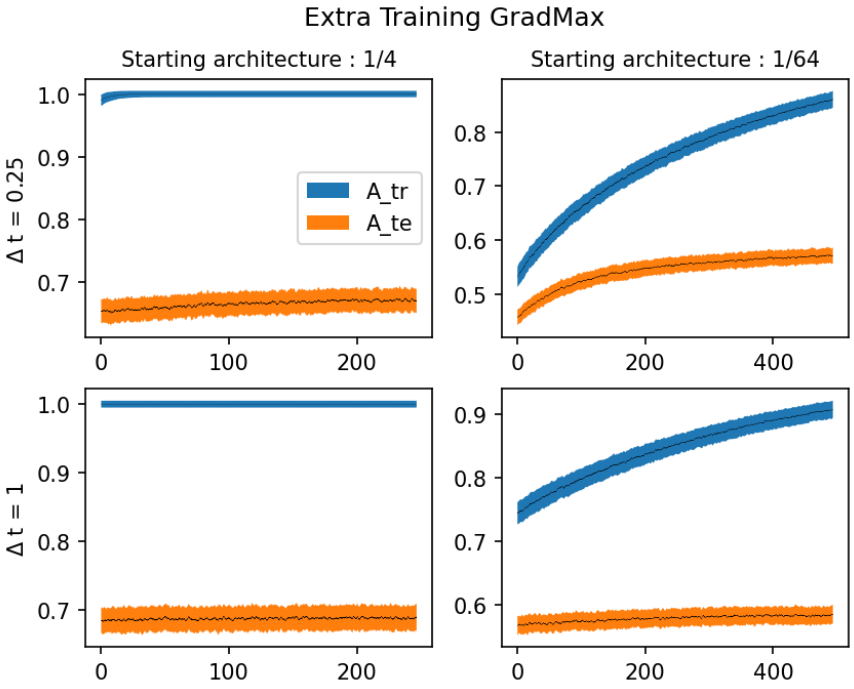

Figure 15: Accuracy curves as a function of the number of epochs during extra training for GradMax.

depth l	Conv2	Conv3	Conv5	Conv6	Conv8	Conv9	Conv11	Conv12
n_l	3	3	6	6	12	12	24	24

Table 2: Number of neurons to add per layer. The depth is identified by its name on tab 3

Table 3: Initial and final architecture for the models of Figure 5. 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 $1/4$, ie $1/16$ indicates that the model $1/64$ started with 1 neurons at this layer while the model $1/4$ started with 16 neurons at the same layer.

ResNet18			
Layer name	Output size	Initial layers (kernel=(3,3), padd.=1)	Final layers (end of Fig 5)
Conv 1	$32 \times 32 \times 64$	$\begin{bmatrix} 3 \times 3, \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 64 \end{bmatrix}$
Conv 2	$32 \times 32 \times 64$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 1/16 \end{bmatrix} \begin{bmatrix} 3 \times 3, 1/16 \\ 3 \times 3, 64 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix}$
Conv 3	$32 \times 32 \times 64$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 1/16 \end{bmatrix} \begin{bmatrix} 3 \times 3, 1/16 \\ 3 \times 3, 64 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix}$
Conv 4	$16 \times 16 \times 64$	$\begin{bmatrix} 3 \times 3, 128 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 128 \end{bmatrix}$
Conv 5	$16 \times 16 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 2/32 \end{bmatrix} \begin{bmatrix} 3 \times 3, 2/32 \\ 3 \times 3, 128 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix}$
Conv 6	$16 \times 16 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 2/32 \end{bmatrix} \begin{bmatrix} 3 \times 3, 2/32 \\ 3 \times 3, 128 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix}$
Conv 7	$8 \times 8 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 256 \end{bmatrix}$
Conv 8	$8 \times 8 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 4/64 \end{bmatrix} \begin{bmatrix} 3 \times 3, 4/64 \\ 3 \times 3, 256 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix}$
Conv 9	$8 \times 8 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 4/64 \end{bmatrix} \begin{bmatrix} 3 \times 3, 4/64 \\ 3 \times 3, 256 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix}$
Conv 10	$4 \times 4 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 512 \end{bmatrix}$
Conv 11	$4 \times 4 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 8/128 \end{bmatrix} \begin{bmatrix} 3 \times 3, 8/128 \\ 3 \times 3, 512 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix}$
Conv 12	$4 \times 4 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 8/128 \end{bmatrix} \begin{bmatrix} 3 \times 3, 8/128 \\ 3 \times 3, 512 \end{bmatrix}$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix}$
AvgPool2d	$1 \times 1 \times 512$		
FC 1	100	512×100	256×100
SoftMax	100		

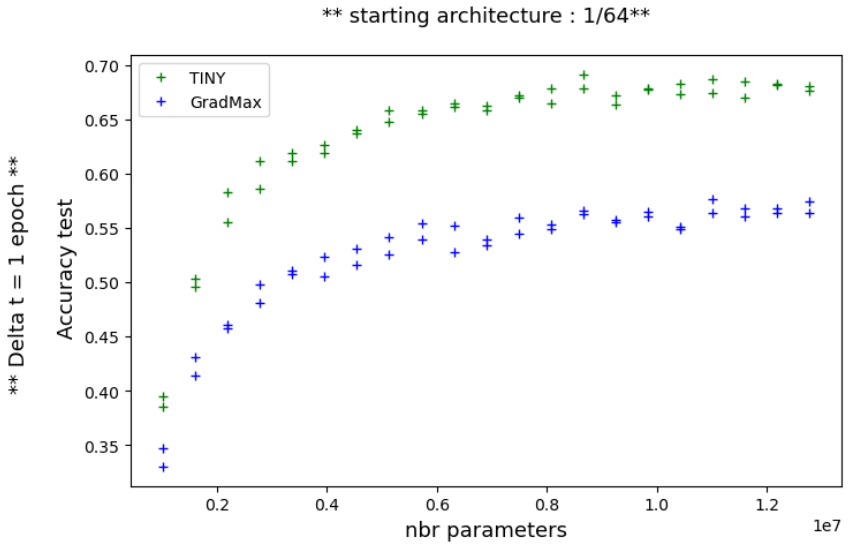

Figure 16: Accuracy on test of as a function of the number of parameters during architecture growth from $\text{ResNet}_{1/64}$ to ResNet18. **The normalization for GradMax is** $\sqrt{10^{-3}}$

.

	TINY	GradMax	Baseline
s \ Δt	1	1	
1/64	68.0 ± 0.4	57.2 ± 0.3	72.9 ± 0.1 ^{5*}
1/64	**69.0** $\pm$ **0.6** ^{5*}	57.7 ± 0.3 ^{3*}	

Table 4: Final accuracy on test of ResNet18 of 16 after the architecture growth (*grey*) and after convergence (*black*). The number of start indicated 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 68.0 ± 0.4on test after its growth and it reaches **69.0** $\pm$ **0.6** ^{5*}after $* := 5 \times 50$ epochs.