Asymmetrical Scaling Layers for Stable Network Pruning

Pierre Wolinski
pierre.wolinski@u-psud.fr

Guillaume Charpiat
guillaume.charpiat@inria.fr

Yann Ollivier
yol@fb.com

April 7, 2020

Abstract

We propose a new training setup, called *ScaLa*, and a new pruning algorithm based on it, called *ScaLP*.

The new training setup ScaLa is designed to make standard Stochastic Gradient Descent resilient to layer width changes. It consists in adding a fixed well-chosen scaling layer before each linear or convolutional layer. This results in an overall learning behavior that is more independent of the layer widths, especially with respect to optimal learning rates, which stay close to 1.

Beyond the usual choice of scaling each input by the factor $1/\sqrt{\#\text{fan-in}}$, we also propose a family of *asymmetric* scaling factors: this promotes learning some neurons faster than others. The pruning algorithm ScaLP is a combination of ScaLa with asymmetrical scaling, and weight penalties. With ScaLP, the final pruned architecture is roughly independent of the layer widths in the initial network.

1 Introduction

Neural network architectures are becoming bigger and bigger. From hand-tuned (VGG [25], GoogLeNet [26], ResNets [13]) to automatically generated architectures (NAS [29], hypernetworks [3]), they tend to become deeper, structurally more complex, and wider. As a consequence, the number of weights increases dramatically, and so does the training time.

The widths of the layers in a neural network are critical hyperparameters, impacting final accuracy as well as computational cost both at training and test time. In practice, grid search over the layer sizes is often required. It would be nice to have an algorithm that works well just by initializing a network to a large width, in the hope that, if good performance can be achieved by a small network, it can also be achieved by a larger network that does not use its extra width. Such an algorithm would also be useful for pruning, by shaving off the extra neurons at the end or even during training.

Such a training algorithm has other hyperparameters such as the learning rate; one could wish these hyperparameters to be resilient to width changes. Otherwise, a new

hyperparameter search would be necessary (notably for the learning rate) each time a new layer width is tested, whether while pruning neurons or for each new experiment in a grid search.

In particular, we show that the standard Stochastic Gradient Descent (SGD) is not resilient to a change of widths in a given neural network: the learning rate has to be adapted accordingly. Consequently, in a network with layers of varied widths, the SGD learning rate cannot be adapted to all layers simultaneously. To solve these problems, we introduce the *ScaLa* (**Sca**ling **La**yer) trick: every layer is preceded by a fixed diagonal scaling layer. Instead of training a layer $\mathbf{w} \in \mathcal{M}_{n_{\text{in}},n_{\text{out}}}$, we train the tensor $\tilde{\mathbf{w}}$ defined by:

$$\mathbf{w} = S\tilde{\mathbf{w}},$$

where $S = \text{Diag}(\sigma_1, \cdots, \sigma_{n_{\text{in}}})$. We show that training $\tilde{\mathbf{w}}$ instead of $\mathbf{w}$ is more resilient to layer width changes.

Such a rescaling with $\sigma_k = 1/\sqrt{n_{\text{in}}}$ is often used in theoretical analyses of neural networks [6]. We emphasize that such layers S can be chosen asymmetrical, i.e., each neuron or channel in the network is scaled with a different factor. This promotes learning various neurons or channels at different speeds. Based on this, we propose *ScaLP* (**Sca**ling **L**ayer **P**runing) to perform either neuron pruning in fully connected layers, or channel pruning in convolutional layers. It consists in using ScaLa with a penalty and an asymmetrical scaling layer; this is provably equivalent to enforcing an *asymmetrical penalty* over the set of computing units (i.e., neurons or channels) in each layer. This way, the most useful units are preserved while the least useful ones are pushed even more strongly towards 0.

For instance, possible setups for S are:

$$\text{ScaLa}: \qquad \sigma_k = \frac{1}{\sqrt{n_{\text{in}}}}, \qquad \text{ScaLP}: \qquad \sigma_k \propto \frac{1}{\sqrt{k}\log k}.$$

In both cases, we show that fine-tuning the learning rate becomes unnecessary when changing layer widths. Moreover, ScaLP combined with a penalization leads to final layer weight matrices in which some rows are close to zero; such layers are consequently easy to prune.

We first provide (Section 3) a theoretical justification for scaling layers, that is based both on an analysis of variance similar to the standard $\frac{1}{\sqrt{n_{\text{in}}}}$ initialization [8], and on a novel analysis of the stability of the SGD step (since we would like to obtain stability of training, not only of initialization).

In Section 4 we define ScaLP, a pruning algorithm based on scaling layers.

ScaLa and ScaLP are tested in Section 5. First, we test the learning rate stability provided by ScaLa: it turns out that a learning rate of 1 performs well over a wide range of network widths. Next, we compare ScaLP (with fixed learning rate 1) to other pruning techniques. We compare the final accuracy and the final number of parameters. Finally, we test whether various pruning techniques find the same pruned architecture size when the initial width is changed. Ideally, either for pruning or for standard training, it should be safe to start with a too-wide network.

2 Related Work

Several approaches has been developed to reduce the computational cost of neural network models at test time. Large neural networks can be compressed in many ways: approximate large neural networks by small ones [2], decrease the rank of large matrices in linear layers [5], share or quantize the weights inside matrices [4, 9]

Making the layers more sparse is one of the most common options. One way to achieve sparsity is to prune connections or neurons during or after training. The first pruning technique, *Optimal Brain Damage*, was based on the second-order derivative of the loss with respect to the weights [16, 11]. This was shown to provide better results than simply pruning the smallest weights. Despite this observation, pruning the weights according to their magnitude is still used [10], and the resulting pruned networks keep roughly their initial accuracy.

A second group of pruning methods focuses on pruning neurons, which is more difficult than pruning weights. The idea of most neuron pruning techniques is to use group Lasso-like penalties, studied in [24]: group Lasso itself [24], sparse-group Lasso [1] or $\ell_{\infty,1}$-norm [22]. It is also possible to prune entire channels in a Convolutional Neural Network (CNN) by evaluating and ranking the $\mathcal{L}^1$-norm of the channels, without regularization during training [18].

Another type of approach recently proposed [19] consists in introducing a multiplicative scaling factor just after each channel or neuron, that is learned and penalized by a $\mathcal{L}^1$-norm. This simple trick allows direct neuron pruning, and can be generalized to remove groups of convolutions or entire layers [14]. Our work also makes use of scaling factors after each neuron, although they play a different role: in our setup, the scaling layer is not learned. Both methods behave differently in our experiments.

Our treatment of width independence for network training is based on intuitions and results for infinitely wide neural networks (Section 3). Interest for the latter has recently increased, continuing the seminal work of Neal [23], who pointed out a connection with Gaussian processes. In [21], infinitely wide deep neural networks are proven to behave like Gaussian processes, while [17] and [15] study the dynamics of such networks, which are called *Neural Tangent Kernels* (NTK). These works indicate that scaling the weights is a necessary step in order to build a training algorithm resilient to any width change: the weights of finitely wide layers are supposed to be scaled by the same factor $1/\sqrt{n_{\text{in}}}$ (where n_{in} is the number of inputs), in order to ensure convergence in the infinitely-wide limit. In Section 3, we prove that a whole family of scaling factors, beyond the standard choice $1/\sqrt{n_{\text{in}}}$, can be used to achieve such resilience. In particular, our algorithm ScaLa may use non-uniform scalings: this preserves individual neurons in the infinite-width limit, whereas the classical $1/\sqrt{n_{\text{in}}}$ scaling produces neurons with infinitesimal individual influence in the limit.

3 ScaLa: Scaling the Weights for Width-Independent Training

In this section, we introduce ScaLa, a technique designed to make SGD resilient to any change of a layer width in the trained model. We will later use this technique for the

pruning algorithm ScaLP (Section 4).

Indeed, one strategy to find the optimal size of a neural network layer consists in starting from a intentionally too wide layer, and pruning it iteratively during training according to some well-suited method. For this, we will ensure that the training technique behaves correctly in a very-large-width setup; ideally, the training of the neural network should be asymptotically independent of the network width for large widths. This leads to considering the infinite-width limit: How should we learn and prune an infinitely wide neural network?

3.1 Two Problems with Infinitely Wide Layers

In a neural network where the layers can be arbitrarily wide, one has to check at least that the first forward pass and the first update do not lead to any divergence.

Problem 1: the forward pass. The forward pass should output activations with bounded variance for arbitrary network sizes. With the Glorot initialization [8], weight variance is set to $1/n_{\text{in}}$, with n_{in} the number of inputs. This preserves the variances of the activations from one layer to the next. Thanks to this, the layer's output does not diverge if n_{in} tends to infinity.

We extend this classical analysis to non-uniform initializations.

Problem 2: the gradient step. Initialization is not the only problem: one has to ensure that the training mechanism is stable as well with large network widths. The output of one neuron should not diverge after a few updates when the number of inputs tends to infinity.

In fact, the number of inputs affects gradient computation: with a fixed learning rate, if n_{in} tends to infinity, the outputs are likely to diverge after even one update. It follows that a simple Glorot-like initialization is not sufficient to solve this problem (Remark 2 below).

We show that replacing a layer with weights w, with a layer with weights $\tilde{w}$ preceded by a *scaling layer* S, is sufficient to deal simultaneously with the two problems above.

In the next sections, the weights w denote the original network parameters; while the weights $\tilde{w}$ are called *scaled weights* and are those we learn via SGD.

3.2 Training a Layer with an Infinite Number of Inputs

To face the problem of learning arbitrarily wide neural networks, let us first understand how a single neuron can deal with an arbitrarily large number N of inputs. We focus here on the simple case where a neuron just computes its pre-activation y and its activation a from its input vector $\mathbf{x} = (x_k)_{1 \leq k \leq N}$:

$$a = \phi\left(\sum_{k=1}^{N} w_k x_k + b\right) = \phi\left(\mathbf{w}^T \mathbf{x} + b\right) = \phi(y),$$

where $\mathbf{w} = (w_k)_{1 \leq k \leq N} \in \mathbb{R}^N$ is the vector of weights of the neuron, $b \in \mathbb{R}$ its bias and ϕ its activation function.

Instead of learning $\mathbf{w}$ directly, we apply the following variable change:

$$\mathbf{w} = S\tilde{\mathbf{w}}, \tag{1}$$

where $S = S_{(N)} = \mathrm{Diag}(\sigma_{(N)1}, \cdots, \sigma_{(N)N}) \in \mathcal{M}_{N,N}$ is a fixed scaling matrix whose coefficients depend only on N, and where $\tilde{\mathbf{w}} = (\tilde{w}_k)_{1 \leq k \leq N}$. The criterion to be optimized is not seen as a function of $\mathbf{w}$ anymore, but as a function of S and $\tilde{\mathbf{w}}$. As S is fixed, we perform SGD on $\tilde{\mathbf{w}}$. This is the *ScaLa* algorithm.

We now provide a necessary and sufficient condition on the scaling S and on the initialization variance of $\tilde{w}$, to ensure that the output of a layer stays bounded, both at initialization and after one SGD step, in the limit of infinitely many inputs to a layer.

Notation and Conditions:

- let $(x_k)_{1 \leq k \leq N}$ be the vector of inputs, whose coordinates are independent random variables with mean 0, variance 1 and finite order-4 momentum;
- we assume, as in [8], that $\frac{\partial L}{\partial y}$ is a random variable of mean 0 and non-zero finite variance, independent of the $(x_k)_k$;
- the weights $(\tilde{w}_k)_{1 \leq k \leq N}$ are randomly initialized, i.i.d. of mean 0 and common variance $\tau^2_{(N)}$ (it will typically set to 1 later);
- the bias b is drawn from a distribution of mean 0 and variance τ_b^2, independently from $(\tilde{w}_k)_k$.

Proposition 1. *We denote by $y_{(N)}$ and $y'_{(N)}$ respectively the initial pre-activation of the neuron and its pre-activation after one SGD step over $\tilde{\mathbf{w}} \in \mathbb{R}^N$. The learning rate η is fixed and independent from N.*

With the assumptions above, and assuming that $\left(\sum_{k=1}^N \sigma^2_{(N)k}\right)_N$ is weakly monotonic and does not admit a subsequence that converges to 0, the following equivalence holds:

$$\begin{cases} \lim_{N\to\infty} \mathrm{Var}(y_{(N)}) & < \infty \\ \lim_{N\to\infty} \mathrm{Var}(y'_{(N)} - y_{(N)}) & < \infty \end{cases}$$
$$\Leftrightarrow \begin{cases} \lim_{N\to\infty} \sum_{k=1}^N \sigma^2_{(N)k} & < \infty \\ \lim_{N\to\infty} \sum_{k=1}^N \sigma^4_{(N)k} & < \infty \\ \lim_{N\to\infty} \tau_{(N)} & < \infty \end{cases}. \tag{2}$$

Thus, luckily, when the conditions of Proposition 1 are satisfied, the neural network is resilient to any width change, *at fixed learning rate.*

Remark 2 (The Glorot initialization leads to an infinite first step)**.** *Importantly, the standard setup of working with the original weights w does not have such properties: it leads to infinite variance after the first SGD step.*

Indeed, w_k is usually initialized with $\mathrm{Var}(w_k) = 1/N$. This is equivalent to setting $\tau^2_{(N)} = 1/N$ and $\sigma_{(N)k} = 1$ (namely, $S = \mathrm{Id}$, no rescaling) in our setup. Since

$$\lim_{N\to\infty} \sum_{k=1}^{N} \sigma^2_{(N)k} = \infty$$
$$\lim_{N\to\infty} \mathrm{Var}(y_{(N)}) < \infty,$$

it follows from Proposition 1 that:

$$\lim_{N\to\infty} \mathrm{Var}(y'_{(N)} - y_{(N)}) = \infty.$$

Therefore, just initializing the weights for bounded activation variance does not implies a bounded SGD update. Hence the interest of the scaling layer S.

In the following, we set $\tau^2_{(N)} = 1$.

Link with classical initialization strategies. Still, it is possible to reproduce Glorot initialization with a *finite* first step. We choose the *uniform scaling* $\sigma_{(N)k} = 1/\sqrt{N}$ (for all k) and $\tau^2 = 1$, which fulfills the conditions (2):

$$\begin{cases} \lim_{N\to\infty} \sum_{k=1}^{N} \left(\frac{1}{\sqrt{N}}\right)^2 & = 1 & < \infty \\ \lim_{N\to\infty} \sum_{k=1}^{N} \left(\frac{1}{\sqrt{N}}\right)^4 & = \frac{1}{N} & < \infty \\ \lim_{N\to\infty} 1 & = 1 & < \infty \end{cases}.$$

This choice corresponds to the well-known initialization: $w_k \sim \mathcal{N}(0, 1/N)$. Still, ScaLa with uniform scaling $\sigma_{(N)k} = 1/\sqrt{N}$ differs from classical SGD: the reparameterization leads to a change of the update rule. Indeed, the update rule for $\tilde{\mathbf{w}}$ can be interpreted as an update rule for $\mathbf{w}$ with scaled learning rates:

$$\begin{cases} \tilde{\mathbf{w}}_{t+1} & = \tilde{\mathbf{w}}_t - \eta \frac{\partial L}{\partial y} (S\mathbf{x})^T \\ S & = \mathrm{Diag}\left(\frac{1}{\sqrt{N}}, \cdots, \frac{1}{\sqrt{N}}\right) \end{cases} \qquad \Rightarrow \qquad W_{t+1} = W_t - \frac{\eta}{N} \frac{\partial L}{\partial y} \mathbf{x}^T.$$

Thus, in the case where the chosen scaling is uniform ($\sigma_{(N)k} = 1/\sqrt{N}$), the variable change can be simply seen as a scaling for the learning rate, inversely proportional to N. Experimentally, this will result in learning rates and learning speeds that are roughly independent of N (Figures. 3a, 3b).

Infinite number of inputs. In Proposition 1, we impose a condition over $(\sigma_{(N)k})_{N,k}$, in order to obtain a consistent behavior of $y_{(N)}$ and $y'_{(N)}$ as the number N of inputs grows.

In the classical treatment of infinitely wide networks, the variances are $1/N$: asymptotically, each individual neuron "disappears".

Here, it is possible to choose values of $\sigma_{(N)k}$ that do not depend on N. This allows a direct treatment of effectively infinite networks, without vanishing neurons. It could also be useful in situations when N changes during training, such as pruning (Section 4). This can be expressed as follows.

Corollary 3. *Under the same conditions as in Proposition 1, and assuming that* $\sigma_{(N)k} = \sigma_k$ *and* $\tau_{(N)} = \tau$*, the outputs of a layer at initialization and after one ScaLa SGD step stay bounded if and only if the sum of scaling factors converges:*

$$\sum_{k=1}^{\infty} \sigma_k^2 < \infty \Leftrightarrow \begin{cases} \lim_{N\to\infty} \mathrm{Var}(y_{(N)}) & < \infty \\ \lim_{N\to\infty} \mathrm{Var}(y'_{(N)} - y_{(N)}) & < \infty \end{cases} . \tag{3}$$

This corollary imposes a constraint over the sequence of scaling factors $(\sigma_k)_k$ in a given layer. Possibilities include

$$\sigma_k \propto \frac{1}{k}, \qquad \sigma_k \propto \frac{1}{\sqrt{k}\log(k)}$$

the latter being one of the slowest-decreasing sequences that still satisfies the finite variance condition.

3.3 Neurons and Convolutional Filters with ScaLa

The initialization rule and the update rule have in common the decomposition of a standard neuron into an unlearned *scaling layer* and a normalized weight layer. So, from now, we use this general model of a neuron, which is easily customizable to fit convolutional filters in CNNs.

Simple neuron. A simple neuron which computes $a = \phi(y) = \phi(\mathbf{w}^T\mathbf{x} + b)$ is transformed into $a = \phi(y) = \phi(\tilde{\mathbf{w}}^T S\mathbf{x}+b)$, where S is a fixed diagonal matrix and $\tilde{\mathbf{w}}$ the matrix of learnable parameters. In the case where $\phi = \mathrm{ReLU}$, the matrix S should satisfy (see Equation (10)):

$$\sum_{k=1}^{N} \sigma_k^2 + \tau_b^2 = 2. \tag{4}$$

This model is represented in Figure 1, where S is the scaling layer.

Remark 4. *If we substitute* $\mathbf{w}$ *for* $\tilde{\mathbf{w}}$*, as proposed in equation* (1)*, then one can write:*

$$y = \mathbf{w}^T\mathbf{x} + b = \tilde{\mathbf{w}}^T(S\mathbf{x}) + b.$$

Therefore, the substitution can be seen in two ways: either the weights are a scaled vector of the original weights and the inputs are as they are (zero-mean and variance 1), or the weights are as they are and the inputs are scaled when they enter the neuron.

Convolutional filter. Let $\mathcal{F} \in \mathbb{R}^{N\times c\times c}$ be a convolutional filter, where N is the number of masks and $c \times c$ is the size of each 2D-mask, and $\mathbf{x} \in \mathbb{R}^{N\times p\times q}$ be a tensor containing N images of size $p \times q$. The filter computing $A = \phi(\mathcal{F}\mathbf{x})$ is transformed into $A = \phi(\tilde{\mathcal{F}}\mathcal{S}\mathbf{x})$, where $\mathcal{S}$ is an operator multiplying each subtensor $X_i \in \mathbb{R}^{p\times q}$ by a factor σ_k, for $k \in \{1, \cdots, N\}$. $\tilde{\mathcal{F}}$ is the tensor of learnable parameters. In the case where $\phi = \mathrm{ReLU}$, we should have:

$$c^2 \cdot \sum_{k=1}^{N} \sigma_k^2 + \tau_b^2 = 2. \tag{5}$$

In this case, $\mathcal{S}$ is the scaling layer.

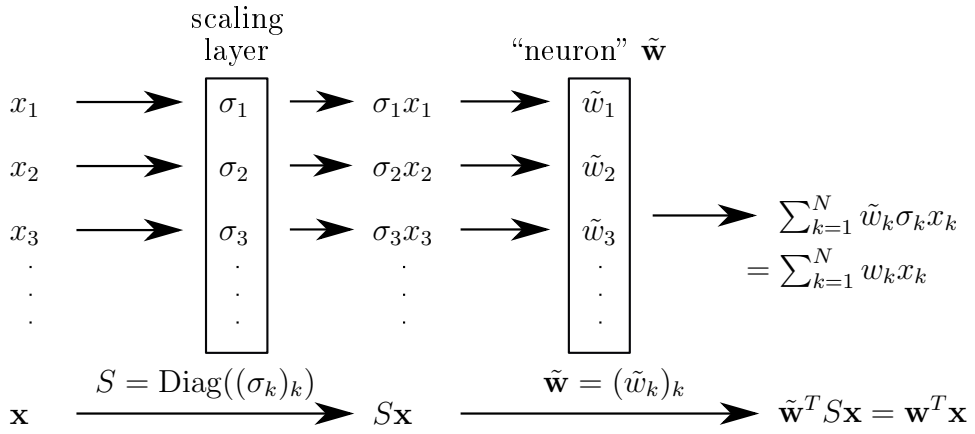


Figure 1: The new model for the simple neuron.

3.4 Normalizing the Activations

We recall that, according to Proposition 1, the inputs of a neuron are assumed to be of mean zero and variance 1. This is easily achieved if the inputs are taken from the dataset: it is sufficient to make an affine operation over the dataset.

In order to keep this assumption true for all neurons of a neural network, we propose to add a batch-norm layer before of after each weight layer.

4 ScaLP: Pruning with Non-Uniform Weight Scaling

In this section, we introduce ScaLP, a pruning method combining ScaLa with a non-uniform scaling and a penalty pushing the rescaled weights $\tilde{w}$ to 0. Thanks to the non-uniform scaling, the penalty on the original weights w becomes non-uniform: some neurons are more strongly pushed to 0 than others, making them easier to prune. The network will more easily train neurons with weaker penalties, and thus consider more penalized neurons only if necessary. This intuitively should lead to automatic adaptation of the layer size, according to the complexity of the task.

4.1 The $\mathcal{L}^2$ Penalty for ScaLP

For the sake of simplicity, we first consider the $\mathcal{L}^2$-squared penalty. Instead of penalizing the weights w directly, we penalize the underlying parameters $\tilde{w}$. This choice results from a consideration about their magnitude: since the parameters $\tilde{w}$ are initially i.i.d., they have initially the same order of magnitude. Then, they contribute equally to the loss and can be learned with the same learning rate. Recalling that "$w = \sigma\tilde{w}$", this penalty can equivalently be seen as a $\mathcal{L}^2$-squared penalization of w modulated by $1/\sigma$. Thus, the level of penalization of w can be tuned through σ.

Definition of the penalty. We denote by $\mathbf{w}$ the vector of all weights, by $\mathbf{w}_k^l$ the k-th neuron in the l-th layer, and by w_{ki}^l its i-th input weight. We denote by $\tilde{\mathbf{w}}_k^l$ and $\tilde{w}_{ki}^l$ their respective underlying parameters.

Proposition 5. *Let $L+1$ be the number of layers of the neural network, where the layer at index 0 is the input of the network. For $l \in \{0, \cdots, L\}$, let n_l be the number of computing units (neurons or convolutional filters) in the layer l. We denote by $\mathbf{w}_k^l$ the k-th computing unit in the layer l and by σ_{lk} the k-th scaling factor in layer l. Thus the entire penalty can be written:*

$$\text{pen}(\mathbf{w}) := \sum_{l=1}^{L} \sum_{k=1}^{n_l} \sum_{i} (\tilde{w}_{ki}^l)^2 = \sum_{l=1}^{L} \sum_{k=1}^{n_l} \sum_{i} \left(\frac{w_{ki}^l}{\sigma_{l,k}} \right)^2 = \sum_{l=0}^{L-1} \sum_{k=1}^{n_l} \left[\frac{1}{\sigma_{l+1,k}^2} \sum_{w \in \mathbf{w}_{k\rightarrow}^l} w^2 \right],$$

where $\mathbf{w}_{k\rightarrow}^l$ is the set of "output weights" of $\mathbf{w}_k^l$. In other words, w belongs to $\mathbf{w}_{k\rightarrow}^l$ if, and only if, the output of $\mathbf{w}_k^l$ is multiplied by the weight w in the next layer.

Since the meaning and the effect of this penalty are not easy to see, we illustrate the rearrangement of the terms. For this purpose, we define respectively the penalty term associated to a neuron w_k^l and its *norm*, denoted by m:

$$\text{pen}(\mathbf{w}_k^l) = \sum_i \left(\tilde{w}_{ki}^l\right)^2 \qquad \text{and} \qquad m(\mathbf{w}_k^l)^2 = \sum_{w \in \mathbf{w}_{k\rightarrow}^l} w^2.$$

The norm $m(\mathbf{w}_k^l)^2$ can be interpreted as the utility of the neuron $\mathbf{w}_k^l$ from the point of view of the next layer. Then, we can reformulate Proposition 5:

$$\text{pen}(\mathbf{w}) = \sum_{l=1}^{L} \sum_{k=1}^{n_l} \text{pen}(\mathbf{w}_k^l) = \sum_{l=0}^{L-1} \sum_{k=1}^{n_l} \frac{m(\mathbf{w}_k^l)^2}{\sigma_{l+1,k}^2}. \tag{6}$$

The right-hand side of this equality can be interpreted as an asymmetrical penalization of the usage of each neuron in a layer: for each neuron $\mathbf{w}_k^l$, we evaluate its "utility" through $m(\mathbf{w}_k^l)^2$, then we scale it by $1/\sigma_{l+1,k}^2$ to obtain the final penalty term.

In practice, the role of the σ_{lk} is clearer if we sort the sequence $(\sigma_{lk})_k$ for each layer l in descending order: the higher is the index k, the smaller σ_{lk} is, the more is penalized the usage of the k-th neuron in the preceding layer. Briefly, the higher is the index of a neuron, the less it is likely to be used.

From now, the sequences $(\sigma_{lk})_k$ are supposed to be non-increasing for all l.

Reordering the neurons. We recall that, in most neural networks, two neurons of a layer can be swapped without changing the function computed by the neural network, if the corresponding weights in the next layer are swapped too. We show not only how to do this in neural networks with scaling layers, but also how it helps decreasing the penalty in that case.

Proposition 6. *Two neurons $\mathbf{w}_i^l$ and $\mathbf{w}_j^l$ can be swapped without changing the operation made by the neural network through the following procedure:*

$$\mathbf{w}_i^l \longleftrightarrow \mathbf{w}_j^l \qquad \textit{and} \qquad \forall k, \begin{cases} \tilde{w}_{ki}^{l+1} & \longleftarrow & \tilde{w}_{kj}^{l+1} \frac{\sigma_{l+1,j}}{\sigma_{l+1,i}} \\ \tilde{w}_{kj}^{l+1} & \longleftarrow & \tilde{w}_{ki}^{l+1} \frac{\sigma_{l+1,i}}{\sigma_{l+1,j}} \end{cases},$$

where $\sigma_{l+1,i}$ and $\sigma_{l+1,j}$ are respectively the i-th and the j-th scaling factor in layer $l+1$.

Remark 7. *Swapping the neurons* $\mathbf{w}_i^l$ *and* $\mathbf{w}_j^l$ *causes a swapping of* $m(\mathbf{w}_i^l)$ *and* $m(\mathbf{w}_j^l)$ without change.

In neural networks with scaling layers, the penalty depends on the association between the $(\mathbf{w}_k^l)_k$ and the $(\sigma_{l+1,k})_k$ at fixed l (equation (6)). The following proposition indicates how to permute the neurons to minimize the penalty.

Proposition 8. *The order of the neurons* $(\mathbf{w}_i^l)_i$ *in a given layer* l *provides the minimal loss if, and only if the sequence* $m(\mathbf{w}_i^l)_i$ *is non-increasing, that is:*

$$\forall i < j, \qquad m(\mathbf{w}_i^l) \geq m(\mathbf{w}_j^l).$$

Since this reordering does not change the output of the network, its benefit may not appear clearly. A potential issue that might arise if no reordering is performed during training is that, by chance, neurons of initially high index might become useful (i.e. their norm may increase). Because of this high indices and thus heavy penalization, the optimization might get stuck and prevent further learning. The proposed reordering gets rid of this potential issue, thus we apply it periodically.

Other penalties. The results above can be extended to other penalties, such as the $\mathcal{L}^1$ penalty or a modified group-Lasso penalty:

- Lasso $\mathcal{L}^1$: choosing the $\mathcal{L}^1$ penalty changes only the definition of the norm:

$$\sum_{l,k} \text{pen}(\mathbf{w}_k^l) = \sum_{l=0}^{L-1} \sum_{k=1}^{n_l} \frac{m(\mathbf{w}_k^l)}{\sigma_{l+1,k}} \qquad \text{with:} \qquad m(\mathbf{w}_k^l) = \sum_{w \in \mathbf{w}_{k\rightarrow}^l} |w|;$$

- group-Lasso $\ell_{2,1}$: unlike the usual version of the group-Lasso penalty applied to neural networks [24], the weights of a layer are put in the same group if they are connected to the same neuron in the preceding layer:

$$\sum_{l,k} \text{pen}(\mathbf{w}_k^l) = \sum_{l=0}^{L-1} \sum_{k=1}^{n_l} \frac{m(\mathbf{w}_k^l)}{\sigma_{l+1,k}} \qquad \text{with:} \qquad m(\mathbf{w}_k^l) = \sqrt{\sum_{w \in \mathbf{w}_{k\rightarrow}^l} w^2}\,.$$

  This norm m is exactly the same as the one defined in equation (6). The resulting parsimony differs slightly from the usual Lasso: instead of pushing the input weights of each neuron towards zero, this group-Lasso penalty pushes its output weights towards zero. This choice is compatible with the pruning rule presented in section 4.2.

4.2 ScaLP Pruning Rule

Pruning neurons from a trained network, that is, removing certain neurons from the network, is expected to cause an accuracy drop, as this operation changes the function computed by the network. Therefore, we divided the learning into two phases: A) training and pruning phase; B) fine-tuning phase. This trick is widely used in pruning literature [18, 24, 19], in order to achieve better performance.

We define a pruning criterion based on the norm m of each neuron, that is, the norm of its *output weights*. As neurons do not all have the same number of outputs, for a fair treatment we choose to balance the norm m by it. We thus introduce the *average norm* $\bar{m}$ of a neuron $\mathbf{w}_k^l$:

$$m(\mathbf{w}_k^l)^2 = \sum_{w \in \mathbf{w}_{k\to}^l} w^2 \qquad \Rightarrow \qquad \bar{m}(\mathbf{w}_k^l)^2 = \frac{1}{\#[\mathbf{w}_{k\to}^l]} \sum_{w \in \mathbf{w}_{k\to}^l} w^2$$
$$m(\mathbf{w}_k^l) = \sum_{w \in \mathbf{w}_{k\to}^l} |w| \qquad \Rightarrow \qquad \bar{m}(\mathbf{w}_k^l) = \frac{1}{\#[\mathbf{w}_{k\to}^l]} \sum_{w \in \mathbf{w}_{k\to}^l} |w| \,,$$

where $\#[\mathbf{w}_{k\to}^l]$ denotes the number of outputs weights $\mathbf{w}_{k\to}^l$.

Phase A: iterative training and pruning. A pruning threshold $\epsilon > 0$ is fixed. The neural network is trained using SGD to minimize the penalized loss. At the end of each epoch, the following steps are sequentially performed on each layer l, from the output to the input, as illustrated in Figure 2:

1. perform a reordering in layer l, as described in section 4.1;
2. establish the list of neurons to prune in layer l. We prune every neuron verifying:
$$\bar{m}(\mathbf{w}_k^l) < \epsilon \; ; \tag{7}$$
3. recompute the $(l+1)$-th scaling layer, in order to maintain the same theoretical pre-activation variance s^2 throughout pruning. For example, if the scaling factors were initially chosen such that:
$$\sigma_{l+1,k} \propto \frac{1}{k} \qquad \text{and} \qquad \sum_{k=1}^{n_l} \sigma_{l+1,k}^2 = s^2,$$
then, after pruning K neurons in layer l, they are redefined such that:
$$\sigma'_{l+1,k} \propto \frac{1}{k} \qquad \text{and} \qquad \sum_{k=1}^{n_l - K} \sigma'^2_{l+1,k} = s^2.$$
Meanwhile, following Proposition 6, the weights are also recomputed so that the global output of the network remains the same:
$$w_k^l \leftarrow w_k^l \frac{\sigma_{lk}}{\sigma'_{lk}}.$$

Selecting the neurons to prune is very easy to perform, since the neurons are regularly reordered by descending order of norms. For k from n_l to 1, i.e. from the lowest norm to the highest, we prune each visited neuron until $\bar{m}(\mathbf{w}_k^l)$ reaches ϵ.

Phase B: fine-tuning. The penalty is removed from the loss, and a final training is performed (without pruning).

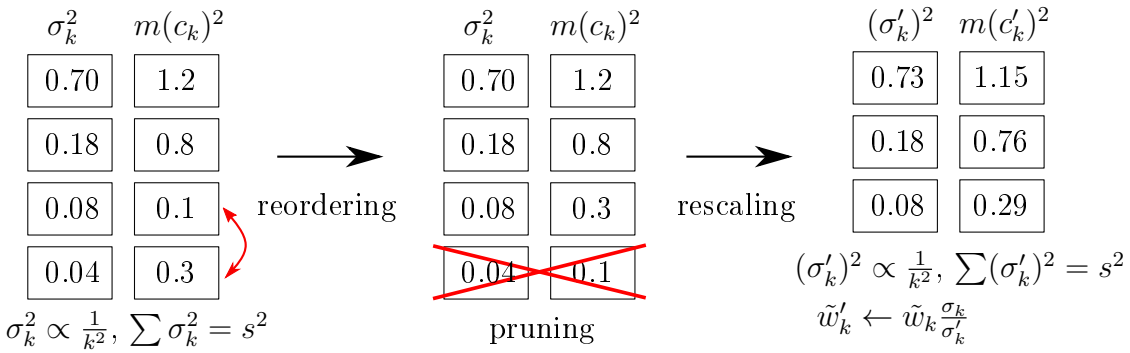


Figure 2: Steps of the pruning phase for one layer: 1) the neurons are reordered by descending order of norm; 2) the neurons with a norm lower than a fixed threshold (e.g. $m(\mathbf{w})^2 \leq \epsilon = 0.2$) are pruned; 3) in the next layer, the scaling layer is recomputed to fit equation (8), and its weights are modified so as not to alter its behavior.

Algorithm 1 Pseudo-code for phase A (training with penalty + pruning)

epoch $\leftarrow 1$
while loss or number of neurons has decreased in the last T epochs **do**
  $\tilde{\mathbf{w}} \leftarrow \tilde{\mathbf{w}} - \eta \left[\frac{\partial \ell}{\partial \tilde{\mathbf{w}}} + \frac{\partial \text{penalty}}{\partial \tilde{\mathbf{w}}}\right]$ ▷ update rule
  for all l from L to 1 **do**
    sort the neurons $(\mathbf{w}_k^l)_k$ of l by decreasing norm $(m(\mathbf{w}_k^l))_k$ ▷ step 1
    $k \leftarrow n_l$
    while $\bar{m}(\mathbf{w}_k^l) < \epsilon$ **do** ▷ step 2
      prune neuron k in layer l
      $k \leftarrow k - 1$
    recompute the scaling layer $(\sigma_{(l+1)k})_k$ of layer $l+1$ ▷ step 3
  epoch $\leftarrow$ epoch $+ 1$

4.3 Choice of $(\sigma_k)_k$

As shown in equation (6), the choice of the sequence $(\sigma_{lk})_k$ determines the penalty, thus the behavior of the network during training. This hyperparameter is closely linked to the expected final sparsity of the network.

Let us consider any layer and drop the index l for simplicity, hence denoting σ_{lk} by σ_k. Supposing that the activation function is $\phi = \text{ReLU}$, the sequence $(\sigma_k)_k$ should verify equation (4) in the case of a fully connected layer, or equation (5) in the case of a convolutional layer. In both cases, the condition can be written:

$$\sum_{k=1}^{N} \sigma_k^2 = s^2, \tag{8}$$

where s is a constant depending only on the activation function [12].

Case $l \geq 1$: hidden layer. We recall that we want to train and prune a neural network, such that the width of the layers of the resulting network will be approximately the same, however large they were at initialization. To handle arbitrarily large widths, we pick a infinite sequence $(\sigma_k)_{k\in[1,\infty[}$ such that $\sum_{k=1}^{\infty} \sigma_k^2$ is finite, and, for a given layer width N, we consider the normalized subsequence $(\sigma_k)_{k\in[1,N]}$, i.e. consider $(\alpha\,\sigma_k)_{k\in[1,N]}$ with $\alpha = s^2/\|(\sigma_k)_{k\in[1,N]}\|_2$, to satisfy Equation (8). Note that such a choice of $(\sigma_k)_k$ satisfies Corollary 3 and that we can pick for instance:

- $\sigma_k \propto \frac{1}{\sqrt{k}\log(k)}$. This sequence just fulfills the preceding condition, introducing few asymmetry between the neurons of the layer in the penalty;
- $\sigma_k \propto \frac{1}{k}$. This sequence is sharper and the neuron penalization is more heterogeneous.

In general, the faster the sequence $(\sigma_k^2)_k$ decreases, the more the first neurons are privileged: they can reach higher values and they learn faster, thus the first neurons are more likely to be useful.

Case $l = 0$: input layer. In the special case $l = 0$, $m(\mathbf{w}_k^0)$ measures the norm of the inputs of the network. If we do not have any prior knowledge about the usefulness of these inputs, the most reasonable choice for the sequence $(\sigma_{0k})_k$ is the constant sequence:

$$\sigma_{0k}^2 = \frac{s^2}{N}.$$

5 Experiments

In this section, we evaluate the pruning technique developed above. We consider an image classification task, with either fully connected or convolutional neural networks.

First, we verify that ScaLa addresses efficiently the Problems 1 and 2 defined in Section 3, that is, the training with SGD of a standard neural network should not diverge when its width tends to infinity. For this, we compare the performance of a simple neural network with different layer widths, with and without scaling layer.

Second, we check the stability of ScaLP. On the one hand, the initial width of the layers should not impact the structure of the resulting neural network, provided that they are wide enough. On the other hand, when performing several runs of a same experiment with same hyperparameters, such as the learning rate or the penalty factor, we would like results to be stable, that is, the final accuracy and the final structure not to vary much.

Third, we compare ScaLP to other pruning algorithms. Moreover, we also test our setup with other common penalties (Lasso, group-Lasso).

In all experiments, the datasets are split into a training set, a validation set and a test set. The test set is only used to test the final version of the pruning technique.

5.1 Influence of the Variable Change $\mathbf{w} \to \tilde{\mathbf{w}}$

In this section, we illustrate the difference between ScaLa and standard SGD, that is, given the variable change $\mathbf{w} = S\tilde{\mathbf{w}}$, we compare SGD over $\tilde{\mathbf{w}}$ to SGD over $\mathbf{w}$.

We first illustrate the training of a neural network with or without ScaLa in the simplest possible setup: a fully connected *linear* neural network $[1024, N, 10]$ on CIFAR-10. We also test the same neural network $[1024, N, 10]$ with ReLU activation functions after the two first layers.

- with ScaLa: the learning rate η is chosen *once and for all*, in order to achieve the best performance for $N = 300$. In practice, this yields $\eta = 1$. Once η is fixed, the neural network is trained for each N with this same learning rate. We have tested three different types of scaling layers $S = \text{Diag}(\sigma_1, \sigma_2, \cdots, \sigma_N)$, defined by:

$$\begin{array}{rll} \text{Uniform:} & \sigma_k = \frac{1}{\sqrt{N}} & \\ \frac{1}{\sqrt{k}\log(k)}\text{:} & \sigma_k = C\frac{1}{\sqrt{k+1}\log(k+1)} & \text{with } C \text{ s.t. } \sum_{k=1}^{N} \sigma_k^2 = 1 \\ \frac{1}{k}\text{:} & \sigma_k^2 = C'\frac{1}{k} & \text{with } C' \text{ s.t. } \sum_{k=1}^{N} \sigma_k^2 = 1 \end{array} \tag{9}$$

- without ScaLa: for each N, we trained the neural network with all learning rates η in $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$.

Then, we measured the best performance achieved by each trained neural network (according to the validation set), and the number of epochs needed to reach this performance.

Note that we made the setup harder for ScaLa: instead of exploring the most efficient learning rate η for each N, we fixed η after one grid search for $N = 300$. Thus, the learning rate we used is independent from N. On the contrary, without ScaLa, we made one grid search for every tested N.

Results. Figure 3 shows the performance and the number of training epochs according to the hidden layer size N, without and with a scaling layer. Without scaling layer, most learning rates ($\eta \in [10^{-4}, 10^{-1}]$) lead to unstable results when N changes: there exists at least one N for which each learning rate performs poorly. In the linear case, the only learning rate that works well for all N is 10^{-5}, but its number of training epochs is much larger than in the other cases, especially for small N. In the ReLU case, the only learning rate which works well without scaling layer is $\eta = 10^{-4}$, but, in terms of accuracy, this setup is dominated by the ScaLa algorithm with uniform scaling.

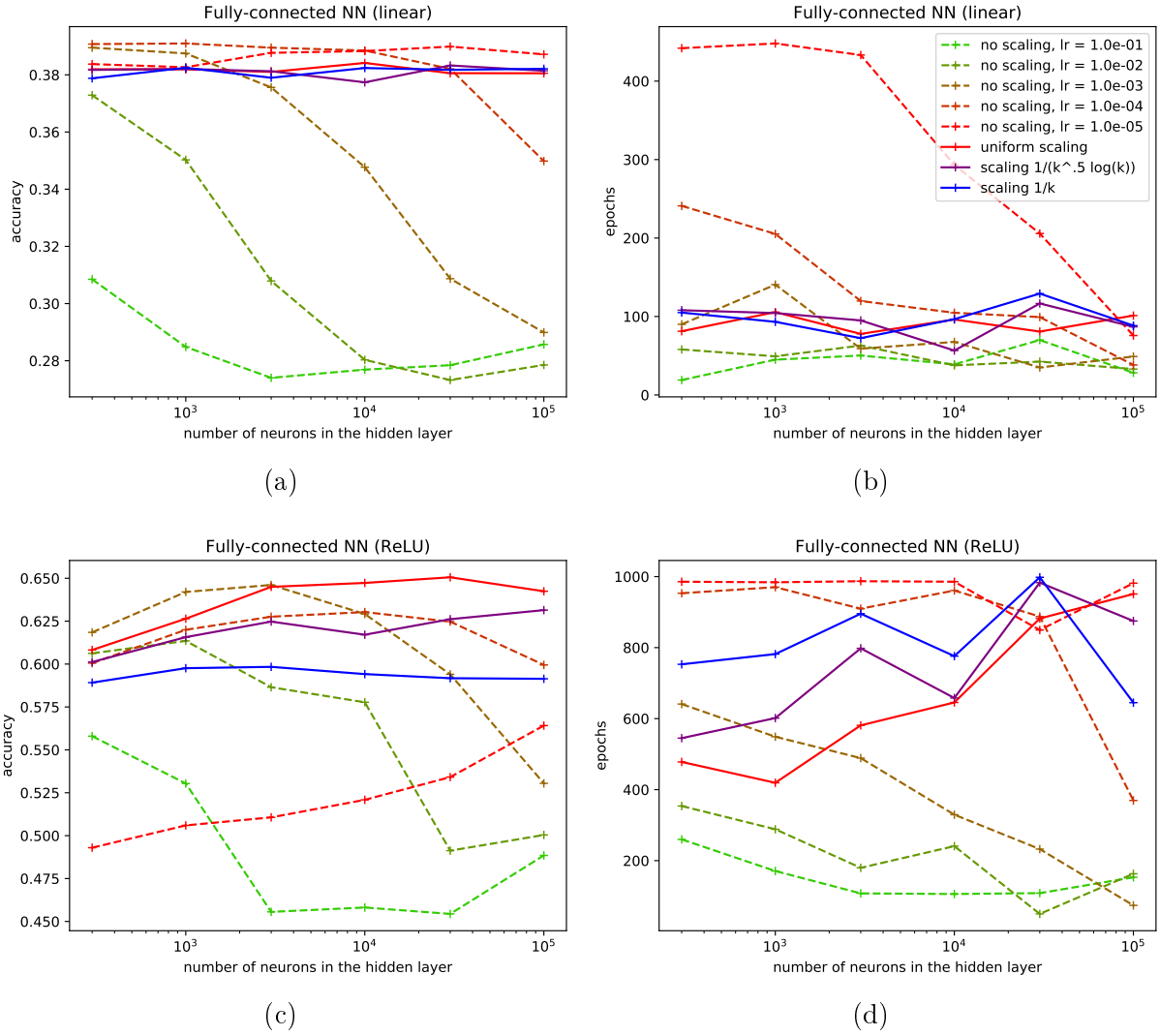


Figure 3: Results of the training of neural networks with one hidden layer of size $N \in [3 \cdot 10^2, 10^5]$ on CIFAR-10, with two setups of activations functions: identity (top) and ReLU (bottom). Figures (3a) and (3c) show their best accuracies on the validation set. Figures (3b) and (3d) show the number of epochs necessary to reach it, up to 1000 epochs. Continuous lines correspond to experiments with a scaling layer, and the dotted lines correspond to experiments without it. Each point corresponds to an average over 3 runs and each run is early stopped if no improvement has been made during 50 epochs.

Conversely, with a scaling layer and a fixed learning rate, the given results do not depend on N: the accuracy and the number of training epochs remain the same for the tested widths. Moreover, they are roughly the same for the three tested scaling layers: there exists a learning rate η such that for all N, the resulting network performs well (in our case $\eta = 1$). However, this stability is not always costless (see Figure 3a): in the linear network setup, for all N, there also exists η such that the resulting network, trained without scaling layer, performs slightly better (but finding it requires a grid search).

Moreover, we recall that the chosen learning rate for ScaLa is 1. Thus, it appears that the scaling layer leads not only to stable results, but also makes the optimal learning rate close to 1, which would make the learning rate search easier. This result is confirmed in the next section: with ScaLP over VGG19, the chosen learning rate is also 1.

Overall, we have shown that, even in a simple setup (training a single-hidden layer neural network), we observe a difference between ScaLa and standard SGD: using ScaLa leads to very stable results according to the width of the hidden layer, with the same learning rate η. However, standard SGD with learning rates fine-tuned separately for each width leads to slightly better results.

5.2 Pruning: Results and Comparison

In this section, we compare our algorithm with other pruning techniques. As pointed out in [20], pruning techniques can be split into two categories. Some, as in [18], need a predefined pruning rate (and consequently, predefined resulting architectures), while the other ones discover automatically the final architecture, as in [19]. Since we do not need any prior knowledge about the pruning rate, our method stands in the second category. Thus we compare it to other penalty-based methods.

5.2.1 Existing Penalties

The pruning techniques studied here depend on a penalty: adding to the loss a well-chosen penalty term is an efficient way to push neurons towards zero, allowing their removal. In this section, we recall some penalties commonly used in pruning. For convenience, we denote by $\mathbf{w}_k^l$ the vector of weights of the k-th neuron in layer l, where $l \in [1, L]$ and $k \in [1, n_l]$.

Lasso penalty. The loss with a Lasso penalty [27] can be written:

$$L(\mathbf{w}) = \ell(\mathbf{w}) + \lambda \sum_{l=1}^{L} \sum_{k=1}^{n_l} \|\mathbf{w}_k^l\|_1,$$

where ℓ is the error term and $\lambda > 0$ is a given constant. The Lasso penalty is likely to push towards 0 each weight with a constant force proportional to λ [27].

Group-Lasso penalty. The group-Lasso penalty [28] is designed to push groups of parameters towards 0. The loss writes itself:

$$L(\mathbf{w}) = \ell(\mathbf{w}) + \sum_{l=1}^{L} \sum_{k=1}^{n_l} \lambda_l \sqrt{\dim(\mathbf{w}_k^l)} \|\mathbf{w}_k^l\|_2,$$

where the $\lambda_l > 0$ are given constants. The scaling factor $\sqrt{\dim(\mathbf{w}_n^l)}$ ensures that the weights are uniformly penalized.

Sparse group-Lasso penalty. The sparse-group Lasso penalty [7] is a linear combination of the Lasso and the group-Lasso penalties. The aim of this penalty is to push simultaneously each weight and each group of weights towards 0.

$$L(\mathbf{w}) = \ell(\mathbf{w}) + \sum_{l=1}^{L} \sum_{k=1}^{n_l} \lambda_l \left[\alpha \|\mathbf{w}_k^l\|_1 + (1-\alpha)\sqrt{\dim(\mathbf{w}_k^l)} \|\mathbf{w}_k^l\|_2 \right],$$

where the $\lambda_l > 0$ are given constants and $\alpha \in [0, 1]$. The value of α is usually 0.5, as in [24, 1].

In the case of large neural networks, the efficiency of pruning techniques using the group-Lasso penalty or the sparse group-Lasso penalty highly depends on the hyperparameters λ_l, which are usually different from layer to layer [1].

BN-Lasso: Lasso penalty over scaling parameters of batch-norm layers. In addition to these penalties, [19] proposes to introduce learned scaling parameters γ, penalized by their $\mathcal{L}^1$-norm. These parameters γ are used as follows: the output of each neuron unit u is multiplied by γ_u. Thus, the size of γ_u indicates the usefulness of the neuron u. Then, the loss can be written as:

$$L(\mathbf{w}, \boldsymbol{\gamma}) = \ell(\mathbf{w}) + \lambda \sum_{\text{neurons } u} |\gamma_u|,$$

where $\lambda > 0$ is a given constant. In practice, instead of introducing new parameters, the authors propose to penalize the trained scaling parameter in each batch-norm layer. We refer to this penalty by the name *BN-Lasso*. Unlike simple Lasso, group-Lasso and sparse group-Lasso over the weights, BN-Lasso leads to stable results, with only one hyperparameter, and without fixing the final sparsity.

As with our algorithm, BN-Lasso is based on scaling factors, but, in our case, these factors are *fixed*, and we penalize the *weights*.

5.2.2 Pruning Experiments

Setup. We tested pruning with two neural networks: a small fully connected neural network trained on MNIST (which is named sFC, with architecture [1000, 1000, 10], i.e. two hidden layers of size 1000), and a VGG19 deep convolutional network (with only one output fully connected layer) trained on CIFAR-10. For each experiment, we retain the neural network at the epoch where it achieves the best validation accuracy: the reported test accuracy, final number of parameters... refer to this specific state.

We tested ScaLP (Section 4.2) with different scalings (9) and different penalties. The tested penalties are: $\mathcal{L}^2$, Lasso and group-Lasso (denoted by *GLasso*). We tested the penalty proposed in [19], to which we refer as *BN-Lasso*, with our pruning setup. As mentioned above, the learning rate was fixed once and for all to 1 for these methods (which was the best across the board, as can be expected with scaling layers).

We also tested standard pruning setup without scaling layers (with Lasso and Group-Lasso); in that case, the learning rate has to be retuned by grid search for each penalty factor λ.

Phase A (pruning) ends when the number of neurons and the best validation accuracy have not improved for 50 epochs. During phase B (fine tuning), the learning rate is decreased by a factor 10 each time the validation accuracy has not improved for 50 epochs, up to 2 times. The third time, training is stopped.

The baseline is the accuracy obtained with the same neural network, learned with SGD and weight decay, without pruning; its learning rate and weight decay constant are optimized for accuracy on the validation set. To obtain a similar setup, the training is also divided into two phases: in phase A, weight decay is applied, then removed in phase B. Moreover, we apply the same learning rate schedule and early stopping rule as in the other setups.

Results. In terms of performance, the non-pruned baseline performs very well. ScaLP with group-Lasso penalty and uniform scaling matches this baseline performance while dividing the number of parameters by 14.

To assess performance for various pruned network sizes, we look at the Pareto front, namely, the set of points corresponding to the best performance for a given target on pruned network size. Clearly (Fig. 4), no method sticks to the Pareto front for all target network sizes. On VGG, both BN-Lasso and ScaLP with $1/(k^{1/2}\log k)$ scaling are on the Pareto front for small network sizes. For pruned network sizes above 10^6, the best performance is with ScaLP with group-Lasso penalty and uniform scaling. On MNIST with a fully-connected network, the Pareto front is entirely made of ScaLP variants, while BN-Lasso is inferior (Fig. 5).

ScaLP with group-Lasso and uniform scaling provides the top performance in Figure 4, but only for a well-tuned λ. On the contrary, ScaLP group-Lasso with non-uniform scaling leads to more regular curves as λ varies. Indeed, in Figure 4, the brightest cyan line goes out of the plot: ScaLP with group-Lasso and uniform scaling is not usable with high λ. For small λ, on the other hand, it tends to overfit: with $\lambda = 0$ performance is below the baseline. (The same holds for BN-Lasso.)

Pruning without scaling layers, either with Lasso or group-Lasso penalties, leads to poor performance compared to ScaLP and BN-Lasso (Fig. 4 and Fig. 5). Moreover, the learning rate for these methods had to be tuned differently for each penalty factor λ (to compensate for the absence of scaling; otherwise their performance are substantially worse). This is in line with the results for such methods in [1], where the authors had to use one penalty factor λ *per layer* for good results.

Surprisingly, the $\mathcal{L}^2$ penalty combined with non-uniform scaling layers (dotted orange lines) leads to pruned neural networks with reasonable performance, although it is rarely used for pruning in standard setups. Still, the resulting networks are always below the Pareto front.

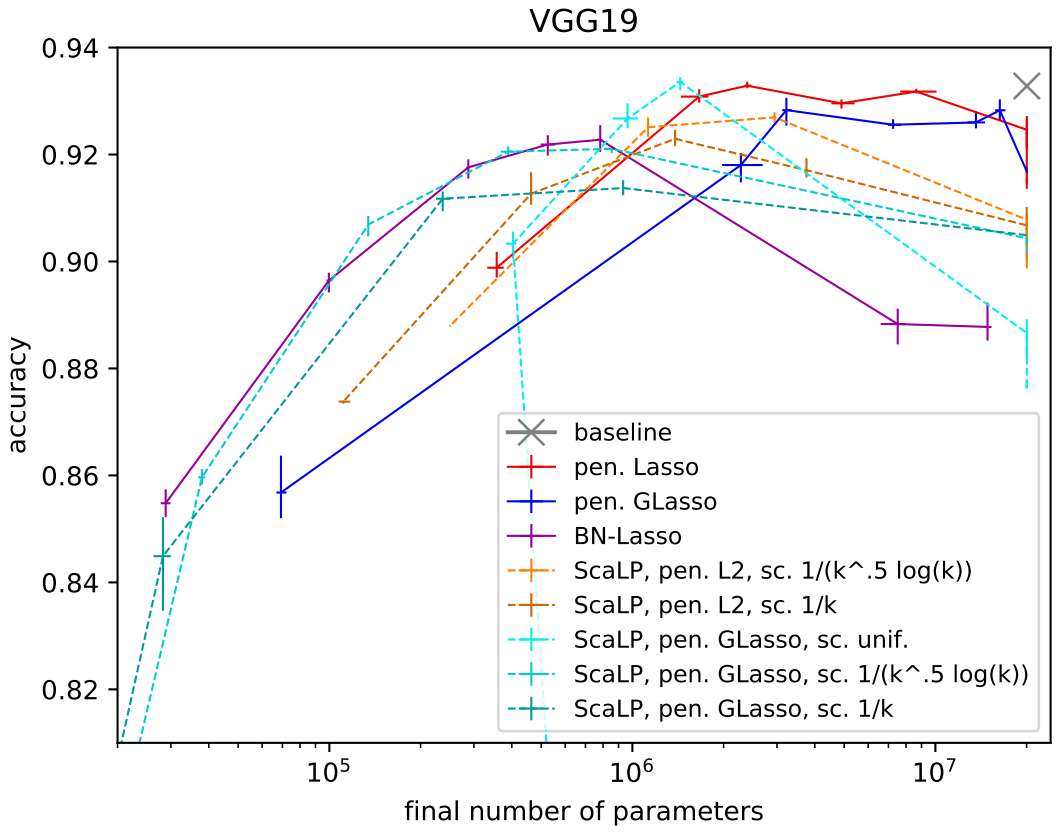


Figure 4: Comparison between ScaLP, BN-Lasso, and standard pruning setups without scaling: Lasso penalty (red) and group-Lasso penalty (blue). Performance of a VGG19 network trained on CIFAR-10. Each setup (pruning method + penalty + scaling) is represented by a line, and each point of a line corresponds to a different penalty factor λ. The bigger λ is, the more the resulting network appears on the left. Each point corresponds to an average over 3 runs, and is surrounded by a vertical bar and an horizontal bar, showing respectively the min/max final accuracy and the min/max final number of parameters.

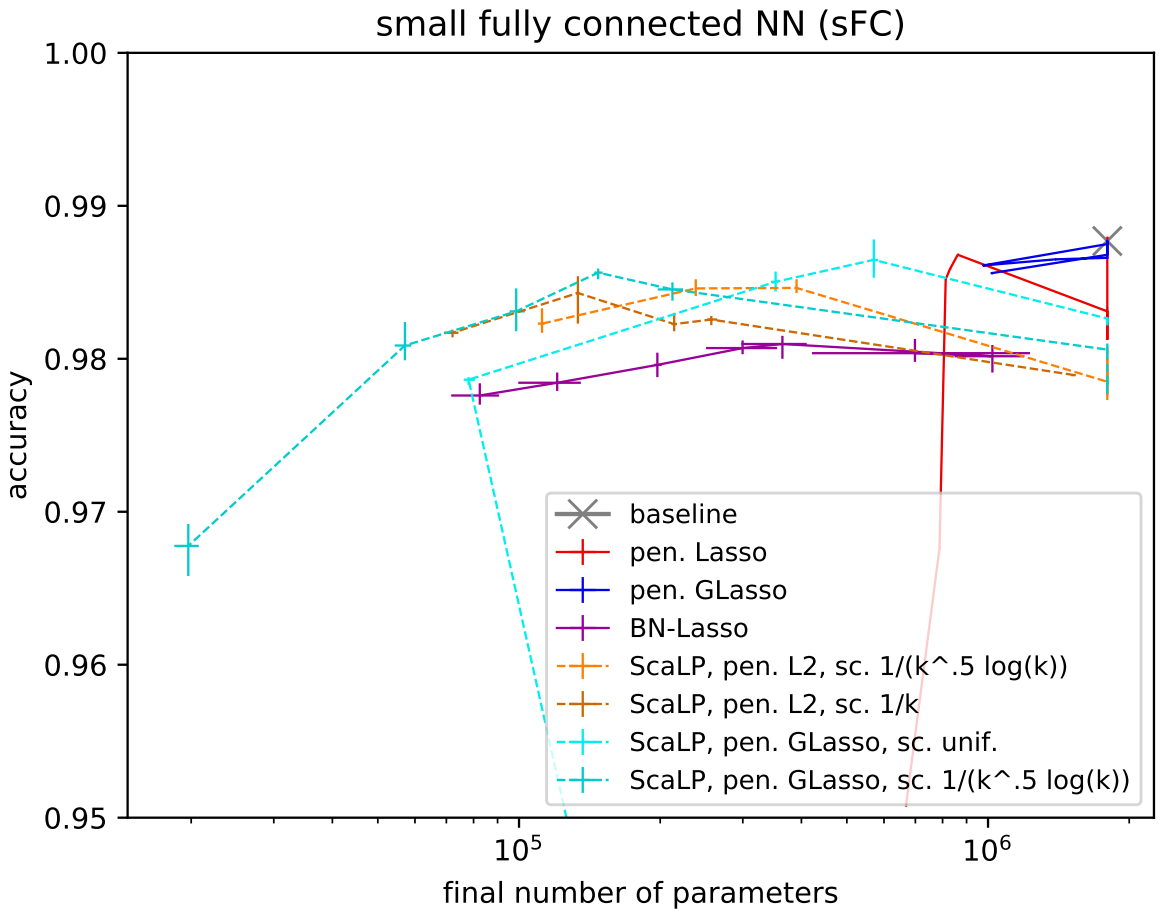
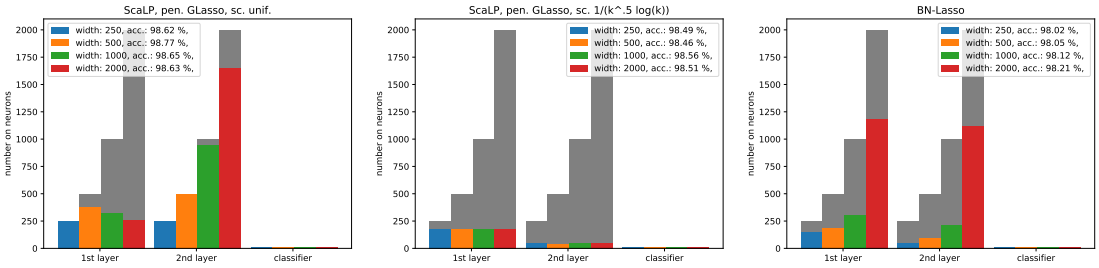


Figure 5: Final accuracy according to final number of parameters for different setups, for a fully connected network trained on MNIST.

Table 1: Final accuracy and final number of parameters. For each pruning setup (i.e. row), we selected the penalty factor λ that led to the best mean accuracy over 3 runs. We reported here this mean accuracy and the mean number of parameters at the end of training. Best results, either in terms of accuracy or final number of parameters have been highlighted (even if they are very close to others).

Model	VGG19 on CIFAR-10		sFC on MNIST	
	Acc(%)	# Params	Acc (%)	# Params
Baseline	93.28	$20M$	98.77	$1.8M$
BN-Lasso	92.28 ± 0.21	$\mathbf{783K \pm 14}$	98.10 ± 0.07	$364K \pm 48$
pen. Lasso	93.29 ± 0.05	$2.39M \pm 0.05$	98.72 ± 0.06	$1.8M \pm 0$
pen. GLasso	92.83 ± 0.22	$3.22M \pm 0.10$	$\mathbf{98.71 \pm 0.05}$	$1.8M \pm 0$
ScaLP, pen. $\mathcal{L}^2$, sc. $1/(\sqrt{k}\log k)$	92.69 ± 0.09	$2.95M \pm 0.02$	98.46 ± 0.05	$238K \pm 0.9$
ScaLP, pen. $\mathcal{L}^2$, sc. $1/k$	92.29 ± 0.12	$1.38M \pm 0.02$	98.43 ± 0.14	$\mathbf{134K \pm 1}$
ScaLP, pen. GLasso, sc. unif.	$\mathbf{93.36 \pm 0.11}$	$1.44M \pm 0.03$	98.65 ± 0.10	$571K \pm 9$
ScaLP, pen. GLasso, sc. $1/(\sqrt{k}\log k)$	92.11 ± 0.05	$855K \pm 29$	98.56 ± 0.03	$147K \pm 3$
ScaLP, pen. GLasso, sc. $1/k$	91.37 ± 0.11	$932K \pm 4$	-	-



(a) (b) (c)

Figure 6: Final number of neurons for different pruning setups and different widths of the initial networks. Grey bars show the initial number of neurons (250, 500, 1000 or 2000). The dataset is MNIST.

5.3 Stability of the Final Architecture with Respect to Initial Width

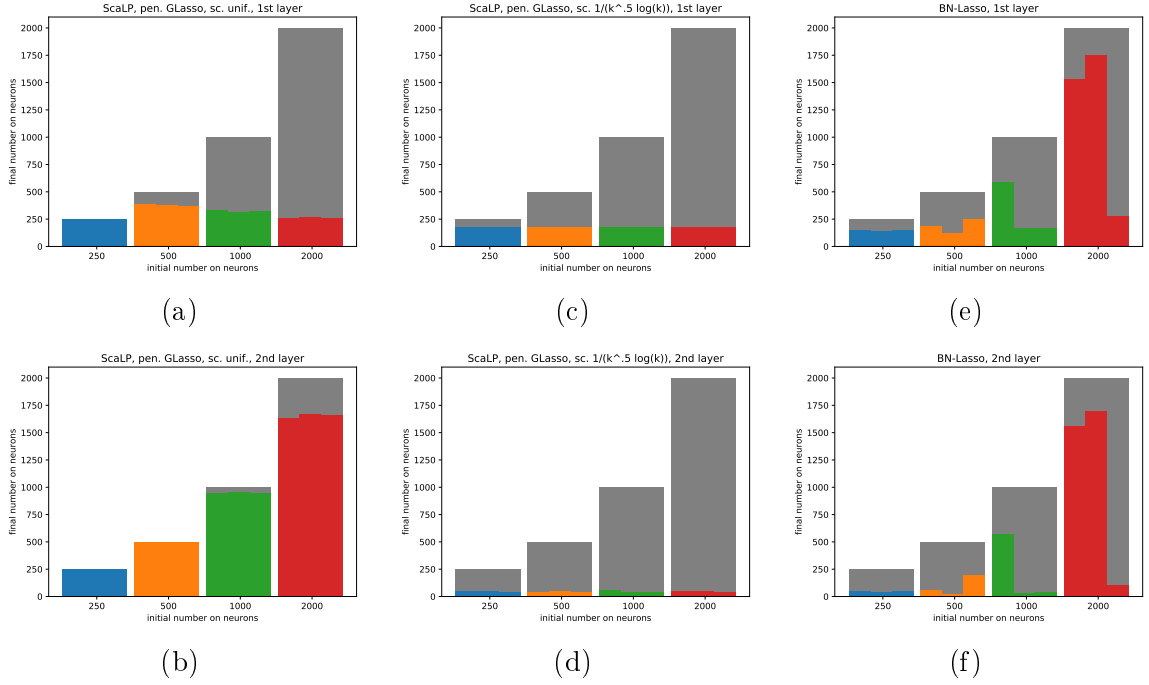
The main motivation for the presented pruning technique was that its behavior should be independent of the initial network width, provided the initial network is wide enough. To test this assumption, we ran the pruning methods with various initial numbers of neurons per layer.

We tested the sFC architecture on MNIST with $[N, N, 10]$ neurons, where $N \in \{250, 500, 1000, 2000\}$. We tested ScaLP with uniform scaling and $1/(k^{1/2}\log k)$ scaling, and BN-Lasso pruning. The results are given in Figure 6: each colored bar is the average final number of neurons, computed over 3 runs.

ScaLP with $1/(k^{1/2}\log k)$ scaling leads to the same network architecture, no matter the initial width. On the contrary, the other two methods return different architectures, depending on the initial width N.

While Figure 6 reports the average over three runs of the final architecture width, Figure 7 reports the individual values obtained for the three runs. We find that ScaLP

leads roughly to the same final neural network, either with a uniform scaling or with scaling $1/(k^{1/2}\log k)$. On the other hand, pruning with BN-Lasso, results in quite different architectures from run to run.



(a) (c) (e)

(b) (d) (f)

Figure 7: Variability of the final number of neurons. Each column corresponds to a pruning setup. For each setup, four initial layer widths were tested (one per color), and three experiments per initial layer width were run. In one plot, each colored bar shows the number of neurons at the end of one run. Figures 7a and 7b: ScaLP with uniform scaling and group-Lasso penalty. Figures 7c and 7d: ScaLP with scaling $1/(\sqrt{k}\log k)$ and group-Lasso penalty. Figures 7e and 7f: BN-Lasso.

5.4 Discussion

Choice of the scaling. With the use of ScaLa or ScaLP comes a new hyperparameter: the scaling $(\sigma_k)_k$. As shown in Table 2, the final neural network depends on the choice of the scaling layer: at fixed penalty factor λ, the more the sequence $(\sigma_k)_k$ decreases sharply, the more neurons are pruned. Therefore, the scaling $(\sigma_k)_k$ may be tuned by the user according to their needs in terms of accuracy and sparsity. Still, a "universal" setting like $1/(k^{1/2}\log k)$ provides a slow decrease while still satisfying the finiteness assumptions (3), and we would expect it to work in general situations.

Limitation in the choice of the penalty. Some choices of penalties may not make sense with ScaLP. The asymmetric scaling is more consistent with grouping output weights. More precisely (Section 4.1, Proposition 5), ScaLP with an asymmetrical scaling and a $\mathcal{L}^2$-squared penalty is equivalent to penalizing asymmetrically the neurons through their *output weights*. The same holds for a Lasso penalty.

Table 2: Final accuracy and final number of parameters. The model is VGG19 trained on CIFAR-10, and pruned with ScaLP combined with the group Lasso penalty with different penalty factor λ and different scalings.

penalty factor λ	sc. unif.		sc. $1/(\sqrt{k}\log k)$		sc. $1/k$	
	Acc(%)	# Params	Acc (%)	# Params	Acc(%)	# Params
$\lambda = 1.38 \cdot 10^{-5}$	90.33 ± 0.21	$404K \pm 14$	85.96 ± 0.12	$38K \pm 0.8$	84.49 ± 0.74	$28K \pm 1.5$
$\lambda = 1.38 \cdot 10^{-6}$	93.36 ± 0.11	$1.44M \pm 0.03$	92.05 ± 0.07	$389K \pm 17$	91.17 ± 0.17	$237K \pm 11$
$\lambda = 1.38 \cdot 10^{-7}$	87.71 ± 0.05	$20M \pm 0$	90.43 ± 0.11	$20M \pm 1K$	91.37 ± 0.11	$932K \pm 4$

On the other hand, a standard group Lasso penalty applied to the sets of input weights of each neuron cannot be interpreted in such way. Group Lasso with an asymmetric scaling would penalize all incoming weights of a given neuron in the same way. This is why we used group Lasso on the sets of *output* weights.

Generic use of scaling layers. We have shown the pertinence of rescaling the inputs of the layers for training a simple linear network (Section 5.1), as well as for training and pruning more complex networks (Section 5.2 and 5.3). However, the uniform scaling $1/\sqrt{\#\text{fan-in}}$, which is already known and used into theoretical works, is not frequently used to train practical neural networks. This is probably due to the slight loss of performance we have observed with scaling layers (Fig. 3a), which we are not able to explain. Therefore, further study has to be made in order to understand why such theoretically assessed scaling does not lead to better results. This phenomenon may be due to a worsening of overfit when we use scaling: by assigning a well-chosen learning rate to each neuron, we remove noise during optimization, thus we regularize less.

6 Conclusion

The standard Glorot initialization provides finite variance of activities at initialization, independently of network width. However, the size of the resulting first SGD gradient step depends heavily on the network width, and learning rates must be adapted accordingly. We have identified the scaling layer trick, ScaLa, as a possible solution, together with theoretical conditions on the scaling factors. Experimentally, ScaLa works well and provides a unified learning rate, close to 1 whatever the widths of the layers in a network are. This provides both theoretical understanding and practical hyperparameter optimization advantage by reducing the size of the search space of learning rates.

Using ScaLa together with non-uniform scalings and penalties leads to the pruning method ScaLP, which is competitive with respect to comparable pruning methods. Interestingly, the final network size provided by ScaLP tends to be independent of the initial size used (though it still depends on a regularization constant), and also to be regular between runs. Neither is the case with other pruning methods.

Our methods are based on the principle that the behavior of network training should be independent from network width: it should be safe to just start with a large enough network, and also to change layer width during training (as happens for pruning). Our approach is based on analogies with the theory of infinitely wide networks (NTK). One

difference is that our non-uniform weights retain the individuality of neurons in the infinite-width limit. In further work, we plan to study how the NTK framework can be extended to our case.

References

[1] Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, pages 2270–2278, 2016.

[2] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[3] Andrew Brock, Theodore Lim, James Millar Ritchie, and Nicholas J Weston. Smash: One-shot model architecture search through hypernetworks. In *6th International Conference on Learning Representations*, 2018.

[4] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[5] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2148–2156. Curran Associates, Inc., 2013.

[6] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. 2019.

[7] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. A note on the group lasso and a sparse group lasso. *arXiv preprint arXiv:1001.0736*, 2010.

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

[9] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[10] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[11] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 304–320, 2018.

[15] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.

[16] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[17] Jaehoon Lee, Lechao Xiao, Samuel S Schoenholz, Yasaman Bahri, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. *arXiv preprint arXiv:1902.06720*, 2019.

[18] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[19] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 2755–2763. IEEE, 2017.

[20] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

[21] Alexander G de G Matthews, Jiri Hron, Mark Rowland, Richard E Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks. 2018.

[22] Kenton Murray and David Chiang. Auto-sizing neural networks: With applications to n-gram language models. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 908–916, 2015.

[23] Radford M Neal. *Bayesian learning for neural networks*. PhD thesis, University of Toronto, 1995.

[24] Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, 2017.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[26] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[27] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[28] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[29] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

A Proof of Proposition 1

We recall Proposition 1:

Proposition. *We assume that the $(x_k)_k$ are independent random variables with zero-mean, variance 1 and finite order 4 momentum. Moreover, we suppose that $\frac{\partial L}{\partial y}$ is a random variable of mean 0 and non-zero finite variance, independent from the $(x_k)_k$.*

We initialize the weights $(\tilde{w}_k)_k$ such that they are i.i.d. of mean 0 and variance $\tau_{(N)}^2$ ($\tau_{(N)}^2 = 1$ if not specified). The bias b is drawn from a distribution of mean 0 and variance τ_b^2, independently from $(\tilde{w}_k)_k$.

We denote by $y_{(N)}$ and $y'_{(N)}$ respectively the initial pre-activation of the neuron and its pre-activation after one SGD step over $\tilde{\mathbf{w}} \in \mathbb{R}^N$. The learning rate η is fixed and independent from N.

Assuming that $\left(\sum_{k=1}^N \sigma_{(N)k}^2\right)_N$ is weakly monotonic and does not admit a subsequence that converges to 0, the following equivalence holds:

$$\begin{cases} \lim_{N\to\infty} \sum_{k=1}^N \sigma_{(N)k}^2 & < \infty \\ \lim_{N\to\infty} \sum_{k=1}^N \sigma_{(N)k}^4 & < \infty \\ \lim_{N\to\infty} \tau_{(N)} & < \infty \end{cases}$$
$$\Leftrightarrow \begin{cases} \lim_{N\to\infty} \mathrm{Var}(y_{(N)}) & < \infty \\ \lim_{N\to\infty} \mathrm{Var}(y'_{(N)} - y_{(N)}) & < \infty \end{cases}.$$

Proof. • We assume that:

$$\begin{cases} \lim_{N\to\infty} \sum_{k=1}^N \sigma_{(N)k}^2 & < \infty \\ \lim_{N\to\infty} \sum_{k=1}^N \sigma_{(N)k}^4 & < \infty \\ \lim_{N\to\infty} \tau_{(N)} & < \infty \end{cases}.$$

Computation of $\mathrm{Var}(y_{(N)})$. We have:

$$y_{(N)} = \sum_{k=1}^N w_k \sigma_{(N),k} x_k + b.$$

Then:

$$\mathrm{Var}(y_{(N)}) = \tau_{(N)}^2 \sum_{k=1}^N \sigma_{(N),k}^2 + \tau_b^2. \tag{10}$$

Since the sequences $\left(\tau_{(N)}^2\right)_N$ and $\left(\sum_{k=1}^N \sigma_{(N),k}^2\right)_N$ converge, then $\left(\mathrm{Var}(y_{(N)})\right)_N$ converges.

Computation of $\mathrm{Var}(y'_{(N)} - y_{(N)})$**.** We denote by w'_k and b' the weight and the bias after one update. We have:

$$\begin{aligned} w'_k &= w_k - \eta \frac{\partial L}{\partial w_k} \\ &= w_k - \eta \frac{\partial L}{\partial y} \frac{\partial y}{\partial w_k} \\ &= w_k - \eta \sigma_{(N),k} x_k \frac{\partial L}{\partial y}. \end{aligned}$$

and:

$$\begin{aligned} b' &= b - \eta \frac{\partial L}{\partial b} \\ &= b - \eta \frac{\partial L}{\partial y} \end{aligned}$$

Then:

$$\begin{aligned} y'_{(N)} - y_{(N)} &= \sum_{k=1}^{N} w'_k \sigma_{(N),k} x_k + b' - \sum_{k=1}^{N} w_k \sigma_{(N),k} x_k - b \\ &= -\eta \frac{\partial L}{\partial y} \left[\sum_{k=1}^{N} \sigma^2_{(N),k} x_k^2 + 1 \right]. \end{aligned}$$

Therefore:

$$\begin{aligned} &\mathrm{Var}\left(y'_{(N)} - y_{(N)}\right) \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[\sum_{k=1}^{N} \sigma^4_{(N),k} \mathbb{E}(x_k^4) + \sum_{k \neq l} \sigma^2_{(N),k} \sigma^2_l \mathrm{Var}(x_k) \mathrm{Var}(x_l) + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} \mathrm{Var}(x_k) + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[3 \sum_{k=1}^{N} \sigma^4_{(N),k} + \sum_{k=1}^{N} \sigma^2_{(N),k} \sum_{l=1, l \neq k}^{N} \sigma^2_l + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[3 \sum_{k=1}^{N} \sigma^4_{(N),k} + \sum_{k=1}^{N} \sigma^2_{(N),k} \left(\sum_{l=1}^{N} \sigma^2_l - \sigma^2_{(N),k} \right) + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[3 \sum_{k=1}^{N} \sigma^4_{(N),k} + \left(\sum_{k=1}^{N} \sigma^2_{(N),k} \right)^2 - \sum_{k=1}^{N} \sigma^4_{(N),k} + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[2 \sum_{k=1}^{N} \sigma^4_{(N),k} + \left(\sum_{k=1}^{N} \sigma^2_{(N),k} \right)^2 + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right]. \end{aligned} \tag{11}$$

Using the assumptions, $\left(\mathrm{Var}(y'_{(N)} - y_{(N)})\right)_N$ converges.

- We assume that:

$$\begin{cases} \lim_{N \to \infty} \mathrm{Var}(y_{(N)}) & < \infty \\ \lim_{N \to \infty} \mathrm{Var}(y'_{(N)} - y_{(N)}) & < \infty \end{cases}$$

Convergence of $\left(\sum_{k=1}^{N}\sigma_{(N)k}^2\right)_N$ and $\left(\sum_{k=1}^{N}\sigma_{(N)k}^4\right)_N$. We recall equation (11):

$$\begin{aligned}&\mathrm{Var}\left(y'_{(N)}-y_{(N)}\right)\\&=\eta^2\mathrm{Var}\left(\frac{\partial L}{\partial y}\right)\left[2\sum_{k=1}^{N}\sigma_{(N)k}^4+\left(\sum_{k=1}^{N}\sigma_{(N)k}^2\right)^2+2\sum_{k=1}^{N}\sigma_{(N)k}^2+1\right].\end{aligned}$$

Then $\left(2\sum_{k=1}^{N}\sigma_{(N)k}^4+\left(\sum_{k=1}^{N}\sigma_{(N)k}^2\right)^2+2\sum_{k=1}^{N}\sigma_{(N)k}^2\right)_N$ converges. Moreover, the three terms are non-negative, thus $\left(\sum_{k=1}^{N}\sigma_{(N)k}^2\right)_N$ is bounded. Recalling that it is also weakly monotonic, this sequence converges, so the second term and the third term. Thus, the first term $\left(\sum_{k=1}^{N}\sigma_{(N)k}^4\right)_N$ converges as well.

Convergence of $(\tau_{(N)}^2)_N$. We recall equation (10):

$$\mathrm{Var}\left(y_{(N)}\right)=\tau_{(N)}^2\sum_{k=1}^{N}\sigma_{(N)k}^2+\tau_b^2.$$

Since we have proven that the sequence $\left(\sum_{k=1}^{N}\sigma_{(N)k}^2\right)_N$ converges, and we have assumed that it does not admit a subsequence that tends to 0, $(\tau_{(N)}^2)_N$ converges. □