université
PARIS-SACLAY

NNT : 2020UPASS026

Thèse de doctorat

Structural Learning of Neural Networks

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique

Unité de recherche : Laboratoire de Recherche en Informatique (LRI, UMR8623)
Thèse présentée et soutenue à Gif-sur-Yvette, le 06/03/2020, par

PIERRE WOLINSKI

Composition du Jury :

Florent Krzakala Full Professor, Sorbonne Universités & École Normale Supérieure (SPHINX team)	Président
Stéphane Canu Professeur des universités, Insa Rouen (Laboratoire d'Informatique, du Traitement de l'Information et des Systèmes)	Rapporteur
Mathieu Salzmann Senior Researcher, École Polytechnique Fédérale de Lausanne (Computer Vision Laboratory)	Rapporteur
Florence d'Alché-Buc Full Professor, Télécom ParisTech (DigiCosme)	Examinateur
Guillaume Charpiat Chargé de recherche, Inria (Laboratoire de Recherche en Informatique)	Directeur de thèse
Yann Ollivier Research Scientist, Facebook	Co-directeur de thèse

Résumé

La structure d'un réseau de neurones détermine dans une large mesure son coût d'entraînement et d'utilisation, ainsi que sa capacité à apprendre. Ces deux aspects sont habituellement en compétition : plus un réseau de neurones est grand, mieux il remplira la tâche qui lui a été assignée, mais plus son entraînement nécessitera des ressources en mémoire et en temps de calcul. Trouver une structure de réseau de taille raisonnable et qui remplisse néanmoins son rôle est donc l'un des problèmes majeurs à résoudre dans ce domaine.

L'objectif de la thèse est d'apporter des solutions à plusieurs problèmes qui se posent dans la recherche d'une bonne structure de réseau de neurones. La première contribution est une méthode d'entraînement de réseau qui fonctionne dans un vaste périmètre de structures de réseaux et de tâches à accomplir, sans nécessité de régler le taux d'apprentissage. La deuxième contribution est une technique d'entraînement et d'élagage de réseau, conçue pour être insensible à la largeur initiale de celui-ci. La dernière contribution est principalement un théorème qui permet de traduire une pénalité d'entraînement empirique en *a priori* bayésien, théoriquement bien fondé.

Taux d'apprentissage variés. La première contribution est la description et le test d'une nouvelle technique d'entraînement de réseau de neurones, baptisée *Alrao* (*All Learning Rates At Once*, « tous les taux d'apprentissage à la fois »).

La méthode standard d'entraînement de réseau de neurones est l'application d'une *Descente de Gradient Stochastique* (SGD) sur ses paramètres. En alternant les cycles d'envoi de données d'entraînement dans le réseau et d'application de la SGD, le réseau s'améliore dans la résolution de la tâche qui lui a été assignée. Or, la SGD ne fonctionne efficacement que si son *taux d'apprentissage* est bien réglé : un taux d'apprentissage trop grand déstabilise le processus d'entraînement et peut faire diverger les poids du réseau (qui devient alors inutilisable), et un taux d'apprentissage trop petit ralentit le processus d'apprentissage. Le taux d'apprentissage est donc une *hyperparamètre* qu'il faut régler *a priori*. La façon la plus simple de le régler consiste à faire une *grid search*, c'est-à-dire lancer plusieurs entraînements de réseau avec différents taux d'apprentissage, puis sélectionner celui qui aboutit au meilleur réseau de neurones. Cette méthode est donc coûteuse en temps de calcul : il faut procéder à autant d'entraînements qu'il y a de taux d'apprentissage à tester !

Avec Alrao, il est possible d'éviter ce surcoût. Au lieu de fixer un taux d'apprentissage bien précis et identique pour l'apprentissage de tous les neurones, Alrao consiste

à attribuer à chaque neurone un taux d'apprentissage différent, initialement tiré aléatoirement dans un intervalle assez large. Les résultats expérimentaux montrent que, pour l'entraînement d'un même réseau, Alrao et la SGD avec un taux d'apprentissage optimisé aboutissent à des réseaux aux performances voisines, et ce pour un ensemble de tâches et d'architectures de réseau très varié. Même si la meilleure performance est souvent atteinte avec la SGD munie d'un taux d'apprentissage optimisé plutôt qu'avec Alrao, ce dernier est certainement utile pour le test rapide de nouvelles architectures.

Élagage de réseaux de taille arbitrairement grande. La deuxième contribution est la description de deux techniques nouvelles, *ScaLa* et *ScaLP*, respectivement *Scaling Layers* (« couches de mise à l'échelle ») et *Scaling Layers for Pruning* (« couches de mise à l'échelle pour élagage »).

Pour résoudre une tâche donnée, le choix d'une architecture de réseau de neurones en particulier résulte le plus souvent d'un compromis entre deux critères : la performance du réseau une fois entraîné au regard de la tâche à accomplir, et le coût en temps de calcul de son entraînement et de son utilisation. Une méthode de construction de réseaux de neurones à l'architecture plus légère consiste à *élaguer* un réseau de neurones de grand taille (i.e. supprimer des neurones) au cours de son entraînement.

ScaLP est une technique d'élagage de réseau vérifiant certains critères de stabilité : le réseau de neurones élagué est toujours le même lors de la reproduction d'une même expérience, et reste le même si l'on augmente le nombre de neurones par couche dans le réseau initial. Pour construire ScaLP, la technique ScaLa a été créée. ScaLa consiste à intercaler une *scaling layer* entre chaque couche du réseau, de sorte que la dynamique du début de l'entraînement soit stable lorsque la taille des couches tend vers l'infini.

Interprétation bayésienne de la pénalité dans le cadre de l'inférence variationnelle. La troisième contribution décrit comment, dans le cadre de l'inférence variationnelle, une pénalité peut être interprétée comme un *a priori* bayésien.

L'inférence bayésienne consiste à estimer les paramètres d'un modèle par des distributions de probabilité *a posteriori*, à partir de données d'entraînement et d'une distribution *a priori* sur les paramètres. Cette façon de procéder provient directement de la formule de Bayes, qui permet de calculer la distribution des paramètres sachant les données, à partir de la distribution des données sachant les paramètres. Outre le fait d'être mathématiquement justifiée, l'inférence bayésienne offre la possibilité de calculer, non seulement les paramètres optimaux d'un modèle, mais aussi l'incertitude liée à cette estimation.

Pour des modèles aussi complexes que des réseaux de neurones, l'inférence bayésienne est bien souvent inutilisable, et il faut se contenter d'une approximation, par exemple l'*inférence variationnelle*. L'inférence variationnelle consiste à fixer une

famille de distributions *a posteriori*, et à sélectionner au sein de cette famille la distribution qui approche le plus l'*a posteriori* bayésien.

Dans cette contribution, on se place dans un cadre général où l'on cherche à optimiser une distribution sur les paramètres d'un réseau de neurones, et où l'on veut favoriser certaines distributions via une *pénalité*. On établit une condition sous laquelle cette pénalité peut être interprétée comme un *a priori* bayésien dans le cadre de l'inférence variationnelle ; et, si c'est le cas, une formule permettant de calculer cette distribution *a priori* est proposée. Ces résultats sont utiles pour comprendre quelle incertitude sur les paramètres est implicitement encouragée par une pénalité donnée.

Remerciements

Je remercie mes directeurs de thèse Guillaume Charpiat et Yann Ollivier, qui ont accepté de superviser ma thèse. Leurs points de vue très différents sur les réseaux de neurones m'ont été extrêmement utiles dans mes recherches. Je leur suis également reconnaissants d'avoir été disponibles tout au long de ma thèse. Enfin, je les remercie de la confiance qu'ils m'ont témoignée en me laissant explorer mon sujet de recherche.

Je remercie Mathieu Salzmann, Stéphane Canu, Florence d'Alché-Buc et Florent Krzakala d'avoir accepté de faire partie de mon jury de thèse. En particulier, je les remercie pour leurs remarques et les conseils qu'ils m'ont apportés pour finaliser mon manuscrit.

Je remercie Marc Schoenauer et Michèle Sebag ainsi que toute l'équipe Tau pour l'organisation des séminaires, des groupes de travail, ainsi que pour les discussions techniques ou philosophiques sur l'apprentissage automatique. Tous ces moments furent pour moi instructifs et formateurs, et ont contribué à rendre vivant le travail de recherche.

Je remercie évidemment ma famille, mes amis, mes camarades. La thèse est une période éprouvante, et leur présence à mes côtés fut indispensable pour soutenir cet effort dans la durée.

Enfin, je souhaite émettre quelques remerciements supplémentaires : à Yann, qui m'a transmis le point de vue de Solomonoff sur l'apprentissage automatique ; à Victor, avec qui j'ai eu de fructueuses discussions sur Bayes ; à Michèle et Zhengying, pour le groupe de travail sur AutoML ; à Léonard, pour la rédaction de l'article sur Alrao ; à Jean, pour les discussions philosophiques sur de nombreux sujets ; à mes parents et à mon frère, à Pierre, à Pierre-Yves, Salim, Aurore, Julien, Raphaël, Virgile, à Claire, à Anaïs, pour leur soutien moral.

Contents

Chapter 1

Introduction

Over the last decade, deep neural networks have progressively beaten state of the art models in many fields: in computer vision, in automatic translation, and even in problems like winning at the go game against the world's best players, which was considered very difficult.

In all these tasks, deep neural networks have the same elementary brick: the artificial *neuron*. The neuron has a very simple design: it takes some inputs, performs an operation by using its own trainable parameters (its *weights*), and outputs a result that can be sent to another neuron. However, the ways to arrange them, that is the *architecture* of the network, are unlimited: this may refer to the high-level design of a deep neural network (does the network contain cycles?) as well as the low-level design (what is the elementary operation performed by the neurons? Are they "fully connected" of "convolutional"?). And between them, it may refer to the number of neurons, the depth of the network, the connectivity of each neuron, the existence or absence of links between layers of neurons, etc.

When tackling a problem, the choice of architecture determines in a large extent the final performance of the neural network. For instance, it has been empirically proven that convolutional neurons are adapted to problems of image classification, while the Long Short-Term Memory (LSTM) units are very useful to problems of text prediction. The number of neurons impacts the performance too: large neural networks tend to perform better than smaller ones. Besides, the architecture also determines the necessary amount of computational resources: the larger a neural network is, the more computational power is needed to train and use it. Therefore, selecting an architecture is about a trade-off between performance and computational cost.

Architecture search strategies. In order to find the architecture that leads to the best results at minimal cost, several strategies have been developed. As a first step, a standard empirical architecture search can be performed: a human expert proposes, trains and tests a set of architectures, and selects the best one. Despite the performance and the wide use of the resulting architectures (VGG,

Inception networks, ResNets), this strategy is not fully satisfactory. First, it requires much computational time, since each proposed architecture should be independently trained and tested. Second, experts' knowledge is implicitly embedded in the search, which prevents from automating it.

As a second step, compressing neural networks during or after training was envisaged. According to this strategy, an expert proposes an architecture, and, at some point, modifies it in order to reduce its size. For instance, the user can prune useless neurons, approximate tensors of weights by low-rank tensors, or quantize the weights (that is approximating them by values lying in a very small set). As a result, the final architecture is less computationally intensive than the initial one, while being complex enough to perform roughly as well. However, experts' knowledge is still needed, since the resulting architecture is very close to the initial one.

In order to perform a wider architecture search, a more sophisticated strategy is being developed: instead of fixing an initial architecture to reduce, the space of architectures is automatically explored by adding or removing neurons, connections, layers, etc. In this category, the main difficulties lie in the structure of the space of architectures: it is impossible to perform an extensive search because of its size, while its discreteness makes impossible a direct gradient descent. Therefore, each proposed algorithm relies on its own topology defined on this space: each new tested architecture is supposed to be a good proxy to estimate the performance of "close" architectures. This way, exploration is guided and computationally feasible. Among existing techniques, there are genetic algorithms, where individuals are neural networks and mutations are slight architecture changes, or a meta-approach where a recurrent neural network which learns to generate architectures.

Present approach of the problem. In this third strategy, one point is computationally critical: each generated architecture should be trained and tested in order to rate it and then refine the direction of exploration of the architecture space. Moreover, they should be trained with their optimal learning rate in order to be compared them fairly. In order to address this problem, we propose the Alrao method, which is designed to train any given neural network without fine-tuning the learning rate.

However, usual techniques designed according to the third strategy are very difficult to study theoretically. Therefore, we focused two works on neuron pruning. In the first one, we propose theoretically well-founded techniques for training and pruning a neural network. Notably, we claim that they are insensitive to the initial width of the network, which is not the case for usual training and pruning procedures.

The second work on neuron pruning is a theoretical study of the constraint (namely the *penalty*) used to push the parameters of the neurons towards zero, in order to prune them. We establish a link between this constraint, which is usually empirically fixed, and the concept of *prior distribution* used in the well-founded framework of Bayesian inference. The link between these is very general, and can be used to find heuristics for the penalty hyperparameters.

Outline. In the introduction, I recall the fundamental concepts used in the present work and expose the main contributions. In the next chapters, I develop them along three axes. Finally, I summarize them and propose further research topics in the conclusion.

1.1 Feedforward Neural Networks

Initially developed to model biological neurons, artificial neurons are nowadays one fundamental brick in the edifice of machine learning. On one side, they are far more simple than their biological version, which makes their implementation easy and allows us to manipulate millions of them; on the other side, they are also sufficiently complex to be combined to complete a large variety of tasks.

One of the most popular way to combine neurons consists in stacking alternatively layers of neurons and non-linear functions, to form a multi-layer perceptron. In this design, each *layer* is composed of a list of neurons taking the same vector as input, and outputting a vector of *pre-activations* whose coordinates are the output of each neuron. This vector is then coordinate-wise transformed by the *activation function* to become the vector of *activations*, which is finally taken as input by the next layer of neurons. Despite its rough design, a multi-layer perceptron can be seen as a universal approximator: with a well-chosen activation function, an arbitrarily wide two-layer perceptron is able to approximate any continuous function [12].

This result shows that any continuous function *can* be approximated by such neural network, but does not show *how* to tune it to fit a given function. Hopefully, the algorithm of backward propagation of the error, called *backpropagation*, can be used to compute the influence of each network parameter over the network output [54]. Thus, it is possible to tune them in order to make the network fit some user-defined property, as approximating a target function.

For instance, let us consider the vector of weights $\mathbf{w}$ of a neural network $f_{\mathbf{w}}$. We want to optimize $\mathbf{w}$ such that, for a given input x, the network outputs y^*. To perform such optimization, we can define a function L to minimize with respect to $\mathbf{w}$, the *loss* function:

$$L(\mathbf{w}) = \|f_{\mathbf{w}}(x) - y^*\|^2.$$

Thus, the vector of weights $\mathbf{w}$ can be tuned in order to minimize the loss L by *Stochastic Gradient Descent* (SGD) [87], that is repeating the following step until $\mathbf{w}$ converges:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \partial_{\mathbf{w}} L, \tag{1.1}$$

where $\eta > 0$ and decreases over time, and $\partial_{\mathbf{w}} L$ is computed by backpropagation. In theory, the loss function L should be convex to ensure the convergence of $\mathbf{w}$ to a global minimizer of L. In the case of neural networks, this condition is not verified. Still, SGD leads empirically to acceptable results.

1.2 Initialization of a Neural Network

Now that a training procedure is defined for neural networks, arises the question of its initialization. In fact, naive initialization procedures make the initial network very hard to train at the beginning. For example, initializing all weights to zero prevents from transmitting any information of the input to the output. Thus, the last layers are initially trained independently from the input, which is a waste of time. More generally, all layers should be initialized such that the global output of the network does not explode and depends on the input data.

Therefore, some heuristics have been found and are widely used to initialize neural networks. An usual initialization heuristic, proposed by Glorot et al. [22], is based on the transmission of the information across the network in both directions. Glorot proposes to build an initialization distribution such that, given random inputs and outputs: the variance of the activations remains the same along the network during propagation, and the variance of the computed gradients remains the same during backpropagation. That is, respectively, and for an activation function with average slope 1, the weights $w_{\cdot}^{l}$ of each layer l should be randomly drawn such that (see Appendix A.1):

$$n_{l-1} \operatorname{Var}(w_{\cdot}^{l}) = 1 \tag{1.2}$$

$$n_{l} \operatorname{Var}(w_{\cdot}^{l}) = 1, \tag{1.3}$$

where $\operatorname{Var}(w_{\cdot}^{l})$ is the variance of the initialization for a weight $w_{\cdot}^{l}$ in the l-th layer, n_{l-1} is the number of weights in one neuron of the l-th layer, and n_l its number of outputs. Since both conditions are incompatible, Glorot proposes a compromise to fix the variance:

$$\operatorname{Var}(w_{\cdot}^{l}) = \frac{2}{n_{l-1} + n_l},$$

and also proposes to draw the weights from a uniform distribution on $[-b, b]$, where b is chosen such that its variance is the same as above. This initialization procedure is known as the "Glorot's initialization" or "Xavier initialization".

Most initialization procedures are also based on considerations on the variance of the activations or the backpropagated gradients, or both.

Example 1.2.1 (Glorot's initialization with uniform distribution)**.** *In the seminal paper, the proposed initialization distribution is:*

$$w_{\cdot}^{l} \sim \mathcal{U}\left(-\frac{6}{n_{l-1} + n_l}, \frac{6}{n_{l-1} + n_l}\right),$$

so that $\operatorname{Var}(w_{\cdot}^{l}) = \frac{2}{n_{l-1}+n_l}$.

Example 1.2.2 (He's initialization with normal distribution)**.** *A refinement of this technique has been found by considering the slope of the activation function: in*

Glorot's initialization, the slope of the activation function is assumed to be 1. *When initializing a network according to Glorot, if an activation function increases very slowly, then the variance of its outputs would be very small. Thus, the output of a deep network composed of such stacked layers would have a variance close to zero. Therefore, the slope of the activation function has to be taken into account [30].*

For instance, if the activation function is the Rectified Linear Unit (ReLU): $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$*, then the variance of the initialization distribution has to be scaled by a factor* 2. *Therefore, the normal He initialization for deep ReLU networks can be written (see Appendix A.1):*

$$w_{\cdot}^{l} \sim \mathcal{N}\left(0, \frac{2}{n_{l-1}}\right),$$

in order to preserve the variance of the activations along the network in the propagation phase.

Besides, He proves that considerations about the variance of the backpropagated gradient are superfluous when initializing weights as indicated above.

The heuristic for $\mathrm{Var}(w_{\cdot}^{l})$ given in Equation (1.2) is used in Chapters 3 and 4. In Chapter 3, we use the same considerations as Glorot: we assume that the weights of a neuron should be such that the variance of its output is the same as its inputs. Notably, we obtain more general results than the initialization procedure presented above: the weights may not be i.i.d. and their learning rates can be scaled with respect to their initial order of magnitude. In Chapter 4, we use Equation (1.2) to get a reasonable order of magnitude of $w_{\cdot}^{l}$ during training. This allows us to compute a distribution around which the weights are likely to lie, a kind of "default distribution" of the weights.

1.3 Variational Inference

We have seen that a neural network is a model, whose parameters are tuned according to training data in order to minimize a loss L. Therefore, we are able to provide a *minimizer* $\hat{\mathbf{w}}$, but we do not have further information about the *distribution* of possible $\hat{\mathbf{w}}$. In particular, when we want to store a trained neural network, we do not know the precision with which each component of $\hat{\mathbf{w}}$ should be stored.

Luckily, the framework of Bayesian inference allows us to compute a *posterior distribution*, which contains both information about the optimal parameter $\hat{\mathbf{w}}$ and about the error margin around it. Despite the intractability of the Bayesian posterior within the context of deep non-linear neural network, it is still possible to approximate it through *variational inference*. The following presentation is based on Graves' paper [24].

Variational inference is mainly used in Chapter 4, in order to better understand the meaning of empirically-defined penalized losses. However, the theoretical content of Chapter 3 can be interpreted in variational terms as well. For the sake of clarity, and since it is not necessary, I will not detail this relation.

1.3.1 General Framework

We introduce step by step the general framework of variational inference. We start from a simple parameter optimization, then we introduce Bayesian inference, and finally we show how to approximate the Bayesian posterior through variational inference.

Deterministic parameters. In a very general framework, we want to model a dataset $\mathcal{D}$ with a model $\mathcal{M}_{\mathbf{w}}$. The dataset is defined by:

$$\mathcal{D} = \{(x_1, y_1), \cdots, (x_n, y_n)\} \in (\mathcal{X} \times \mathcal{Y})^n,$$

where n is the size of the dataset.

The model $\mathcal{M}_{\mathbf{w}}$, parametrized by a vector $\mathbf{w} \in \mathbb{R}^N$, takes an input $x \in \mathcal{X}$ and returns a distribution over $\mathcal{Y}$, denoted by $p_{\mathbf{w}}(\cdot|x)$. Thus, given a pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$, $p_{\mathbf{w}}(y|x)$ measures the likelihood of y given x, according to the model $\mathbf{w}$. By abuse of notation, we will write:

$$p_{\mathbf{w}}(\mathcal{D}) = \prod_{i=1}^{n} p_{\mathbf{w}}(y_i|x_i).$$

Within this framework, it is very common to consider $\mathbf{w}$ as a deterministic vector of parameters, which has to be tuned to maximize the likelihood $p_{\mathbf{w}}(\mathcal{D})$, or minimize the negative log-likelihood:

$$L(\mathbf{w}) = -\ln p_{\mathbf{w}}(\mathcal{D}).$$

Equivalently, $L(\mathbf{w})$ is the loss of a minimization problem over $\mathbf{w}$.

Example 1.3.1 (Binary classification problem)**.** *Our goal is to use the model $\mathcal{M}_{\mathbf{w}}$ to classify inputs $x \in \mathcal{X}$ into classes of $\mathcal{Y} = \{0, 1\}$. For a given input x, the model outputs a distribution over the discrete set $\mathcal{Y}$, that is:*

$$p_{\mathbf{w}}(0|x) \qquad \textit{and} \qquad p_{\mathbf{w}}(1|x),$$

such that $p_{\mathbf{w}}(0|x) + p_{\mathbf{w}}(1|x) = 1$ with $p_{\mathbf{w}}(0|x) \geq 0$ and $p_{\mathbf{w}}(1|x) \geq 0$.

Thus, we can consider that the model classifies x in the class $y^* = \arg\max_y p_{\mathbf{w}}(y|x)$.

Example 1.3.2 (Regression problem)**.** *Our goal is to use $\mathcal{M}_{\mathbf{w}}$ to model a function $f : \mathcal{X} \to \mathcal{Y}$. For a given input x, we assume that the model outputs a Gaussian distribution $\mathcal{N}(\mu(x), 1)$. In practice, $\mathcal{M}_{\mathbf{w}}$ returns $\mu(x)$, which is then interpreted as the mean of a Gaussian distribution of variance 1.*

Thus, given an input x, we can consider that the model estimates $f(x)$ by $\mu(x)$.

Bayesian inference. From now, we suppose that both the vector of parameters $\mathbf{w}$ and any pair (x, y) of the training set $\mathcal{D}$ are random variables. Given x and $\mathbf{w}$, we model y by sampling it from the distribution $\mathcal{M}_{\mathbf{w}}(x) = p_{\mathbf{w}}(\cdot|x)$. Then, we can use Bayes' theorem to compute the distribution of $\mathbf{w}$ given (x, y). Applied to the whole dataset $\mathcal{D}$, this distribution can be expressed as follows:

$$\pi_{\mathcal{D}}(\mathbf{w}) = \frac{p_{\mathbf{w}}(\mathcal{D})\alpha(\mathbf{w})}{\mathbb{P}(\mathcal{D})},$$

where α is a known fixed distribution, and $\mathbb{P}$ is the distribution from which the dataset $\mathcal{D}$ is sampled. α is called the prior distribution of the random variable $\mathbf{w}$, and $\pi_{\mathcal{D}}$ is the posterior distribution of $\mathbf{w}$ given the dataset $\mathcal{D}$.

This formula allows us to compute the posterior distribution $\pi_{\mathcal{D}}$, which indicates which parameters $\mathbf{w}$ are the most likely to generate the dataset $\mathcal{D}$. The next step consists in finding an acceptable value for $\mathbf{w}$.

There are several ways to do this, notably sample a vector $\hat{\mathbf{w}}$ from the distribution $\pi_{\mathcal{D}}$ or compute the Maximum A Posteriori (MAP) estimator $\hat{\mathbf{w}}_{\text{MAP}}$, which is defined by: $\hat{\mathbf{w}}_{\text{MAP}} = \arg\max_{\mathbf{w}} \pi_{\mathcal{D}}(\mathbf{w})$. Equivalently, $\hat{\mathbf{w}}_{\text{MAP}}$ can be computed by minimizing the loss L_{MAP} defined by:

$$L_{\text{MAP}}(\mathbf{w}) = -\ln \pi_{\mathcal{D}}(\mathbf{w}) = -\ln p_{\mathbf{w}}(\mathcal{D}) - \ln \alpha(\mathbf{w}).$$

To summarize, the MAP estimator $\hat{\mathbf{w}}_{\text{MAP}}$ is the most likely value of $\mathbf{w}$, given the dataset $\mathcal{D}$.

Example 1.3.3 (linear model with Gaussian noise)**.** *We are looking for one real parameter $b \in \mathbb{R}$ such that the distribution of $y \in \mathbb{R}$ given $(b, x) \in \mathbb{R}^2$ has a density:*

$$p_b(y|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - bx)^2}{2\sigma^2}\right),$$

which corresponds to a linear model of y given x with centered Gaussian noise of given variance σ^2. In other words, according to this model, y is generated with the formula:

$$y = bx + \sigma\xi \qquad \textit{where} \qquad \xi \sim \mathcal{N}(0, 1).$$

We want to estimate b from a given dataset $\mathcal{D} = \{(x_1, y_1), \cdots, (x_n, y_n)\}$, where the data points (x_i, y_i) are supposed to be independent and identically distributed (i.i.d.). Then, the Bayes formula gives:

$$\pi_{\mathcal{D}}(b) = \frac{p_b(\mathcal{D})\alpha(b)}{\mathbb{P}(\mathcal{D})}.$$

It is sufficient to compute the numerator of the fraction, since the denominator does not depend on b, and the density $\pi_{\mathcal{D}}(b)$ has integral one. Then:

$$\pi_{\mathcal{D}}(b) \propto p_b(\mathcal{D})\alpha(b).$$

Since the data points are supposed to be i.i.d., we have:

$$\begin{aligned}\pi_{\mathcal{D}}(b) &\propto \prod_{i=1}^{n} p_b(y_i|x_i)\alpha(b) \\ &\propto (2\pi\sigma^2)^{-n/2} \prod_{i=1}^{n} \exp\left(-\frac{(y_i - bx_i)^2}{2\sigma^2}\right)\alpha(b) \\ &\propto (2\pi\sigma^2)^{-n/2} \exp\left(-\sum_{i=1}^{n} \frac{(y_i - bx_i)^2}{2\sigma^2}\right)\alpha(b)\end{aligned}$$

If we choose the prior over b as a product of centered Gaussian distributions, i.e.: $\alpha = \mathcal{N}(0, \sigma_b^2)$, then we have:

$$\pi_{\mathcal{D}}(b) \propto (2\pi\sigma^2)^{-n/2}(2\pi\sigma_b^2)^{-1/2} \exp\left(-\sum_{i=1}^{n} \frac{(y_i - bx_i)^2}{2\sigma^2} - \frac{b^2}{2\sigma_b^2}\right),$$

which achieves the computation of the posterior distribution given $\mathcal{D}$.

In order to compute the MAP estimator $\hat{b}_{\text{MAP}}$, it is sufficient to compute the maximum of the density $\pi_{\mathcal{D}}$, which is equivalent to compute the maximum of the log*-density:*

$$\ln \pi_{\mathcal{D}}(b) = -\sum_{i=1}^{n} \frac{(y_i - bx_i)^2}{2\sigma^2} - \frac{b^2}{2\sigma_b^2} + \text{constant}.$$

Then, it is easy to prove that this log*-density has exactly one maximum:*

$$\hat{b}_{\text{MAP}} = \frac{\sum_{i=1}^{n} x_i y_i}{\frac{\sigma^2}{\sigma_b^2} + \sum_{i=1}^{n} x_i^2}.$$

In this example, the model is simple enough to compute analytically the posterior. However, the posterior is intractable if the model is too complex, like in the case of deep non-linear neural networks. Therefore, variational inference has been developed in order to approximate the Bayesian posterior for complex models.

Variational inference. Variational inference is a technique of approximation of the posterior $\pi_{\mathcal{D}}$ by expressing the distance between $\pi_{\mathcal{D}}$ and any distribution β in an operable way (see appendix A.2):

$$\text{KL}(\pi_{\mathcal{D}} \| \beta) = -\mathbb{E}_{\mathbf{w}\sim\beta} \ln p_{\mathbf{w}}(\mathcal{D}) + \text{KL}(\alpha \| \beta) + \ln \mathbb{P}(\mathcal{D}), \tag{1.4}$$

where $\text{KL}(\alpha\|\beta)$ is the Kullback-Leibler divergence between α and β, defined by:

$$\text{KL}(\alpha \| \beta) = \mathbb{E}_{\mathbf{w}\sim\beta} \ln \frac{\beta(\mathbf{w})}{\alpha(\mathbf{w})}.$$

Since the Kullback-Leibler divergence between $\pi_{\mathcal{D}}$ and β is minimal when the tested distribution β and the target distribution $\pi_{\mathcal{D}}$ are equal, $\text{KL}(\pi_{\mathcal{D}}\|\beta)$ can

be seen as a loss to minimize with respect to β. Therefore, it is possible to approximate $\pi_{\mathcal{D}}$ by minimizing $\mathrm{KL}(\pi_{\mathcal{D}} \| \beta)$ over a parametrized family of distributions $\mathcal{B} = \{\beta_{\mathbf{u}} : \mathbf{u} \in \mathcal{U} \subset \mathbb{R}^P\}$. Then, equation (1.4) can be turned into a loss over $\mathbf{u}$:

$$L_{\mathrm{VI}}(\mathbf{u}) = -\mathbb{E}_{\mathbf{w} \sim \beta_{\mathbf{u}}} \ln p_{\mathbf{w}}(\mathcal{D}) + \mathrm{KL}(\alpha \| \beta_{\mathbf{u}}), \tag{1.5}$$

where the constant term $\ln \mathbb{P}(\mathcal{D})$ has been removed. In short, the Bayesian posterior $\pi_{\mathcal{D}}$ is approximated by the so-called *variational posterior* $\beta_{\mathbf{u}^*}$, where $\mathbf{u}^* = \arg\min_{\mathbf{u} \in \mathcal{U}} L_{\mathrm{VI}}(\mathbf{u})$.

1.3.2 Application to Neural Networks

In variational neural networks, the vector of weights $\mathbf{w}$ is indirectly trained by optimizing $\mathbf{u}$, the vector of parameters of the current posterior $\beta_{\mathbf{u}}$ of $\mathbf{w}$. Usually, this is done by applying the following procedure: each time data is sent to the network, the weights $\mathbf{w}$ are drawn from their current posterior distribution $\beta_{\mathbf{u}}$; then, a forward and a backward pass are performed; finally, the computed gradients $\partial_{\mathbf{w}}[-\ln(p_{\mathbf{w}}(\mathcal{D}))]$ are used to compute $\partial_{\mathbf{u}} L_{\mathrm{VI}}$, which is then used to update the parameters $\mathbf{u}$ by SGD.

In order to train such a network, one has to choose the family $\mathcal{B}$ of posterior distributions $\beta_{\mathbf{u}}$, the parametrization $\mathcal{U}$ of this family, and the prior distribution α. Moreover, one should provide a way to compute $\partial_{\mathbf{u}} L_{\mathrm{VI}}$ from the gradient $\partial_{\mathbf{w}}[-\ln(p_{\mathbf{w}}(\mathcal{D}))]$.

Family of variational posteriors and its parametrization. When the model $\mathcal{M}_{\mathbf{w}}$ is a neural network, the vector of weights and biases $\mathbf{w} \in \mathbb{R}^N$ is randomly sampled from a distribution $\beta_{\mathbf{u}}$. It is then necessary to choose the family of posterior distributions $\mathcal{B} = \{\beta_{\mathbf{u}} : \mathbf{u} \in \mathcal{U}\}$ and its parameter space $\mathcal{U}$.

Within the context of neural networks, the parameters are usually learned independently: for a given parameter w, w is individually learned by SGD:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}.$$

Therefore, it is reasonable to sample independently each component of the vector $\mathbf{w}$: each w_k is drawn from its own distribution β_{u_k} parametrized by a vector $u_k \in \mathbb{R}^q$. Thus, we can write:

$$\begin{aligned} \mathbf{w} &\sim \beta_{(u_1, \cdots, u_N)} = \beta_{u_1} \otimes \cdots \otimes \beta_{u_N} \\ \mathbf{u} &= (u_1, \cdots, u_N) \in \mathbb{R}^{N \times q}. \end{aligned}$$

To summarize, instead of learning each parameter w_k directly, each w_k is randomly drawn from a distribution β_{u_k} with learned vector of parameters $u_k \in \mathbb{R}^q$.

Since the distributions $\beta_{\mathbf{u}}$ over $\mathbf{w}$ are products of distributions over its components w_k, we say that $\beta_{\mathbf{u}}$ is a *diagonal* distribution.

Prior. Since we consider a family of diagonal posteriors, we propose a diagonal prior, that is:

$$\alpha = \alpha_1 \otimes \cdots \otimes \alpha_N.$$

This way, the KL-divergence term of (1.5) can be rewritten as a sum of independent terms:

$$\mathrm{KL}(\alpha \| \beta_{\mathbf{u}}) = \sum_{k=1}^{N} \mathrm{KL}(\alpha_k \| \beta_{u_k}),$$

that is a individual penalization of each parameter u_k.

Learning the parameters of the variational posterior. As such, the vector of parameters $\mathbf{u}$ is difficult to learn by SGD in a general framework. However, Kingma et al. proposed the *reparameterization trick* [49], which is designed to trace back easily $\partial_{\mathbf{u}} L_{\mathrm{VI}}$ from $\partial_{\mathbf{w}}[-\ln(p_{\mathbf{w}}(\mathcal{D}))]$. The latter can easily be computed backpropagation of the error through the network.

The reparameterization trick consists in drawing each weight w_k indirectly from its posterior distribution β_{u_k}: a random variable ξ_k is drawn from a fixed distribution, then ξ_k is transformed by a function f_{u_k} such that $f_{u_k}(\xi_k) \sim \beta_{u_k}$. Finally, w_k is set to $f_{u_k}(\xi_k)$. This way, $\partial_{u_k} L_{\mathrm{VI}}$ can easily be computed from $\partial_{w_k}[-\ln(p_{\mathbf{w}}(\mathcal{D}))]$:

$$\partial_{u_k} L_{\mathrm{VI}}(\mathbf{u}) = \mathbb{E}_{\boldsymbol{\xi}}\left[\partial_{w_k}[-\ln(p_{\mathbf{w}}(\mathcal{D}))] \cdot \partial_{u_k} f_{u_k}(\xi_k)\right] + \partial_{u_k} \mathrm{KL}(\alpha \| \beta_{\mathbf{u}}),$$

where $\boldsymbol{\xi} = (\xi_1, \cdots, \xi_N)$.

Example 1.3.4 (Diagonal Gaussian posterior and prior distributions)**.** *A very common choice for the family of posterior distributions is the family of diagonal Gaussian distributions: each component w_k of $\mathbf{w}$ is independently drawn from its own Gaussian distribution $\beta_{\mu_k,\sigma_k^2} = \mathcal{N}(\mu_k, \sigma_k^2)$. Then, we can write:*

$$\begin{aligned} \mathbf{w} &\sim \beta_{(\mu_1,\sigma_1^2,\cdots,\mu_N,\sigma_N^2)} = \mathcal{N}(\mu_1, \sigma_1^2) \otimes \cdots \otimes \mathcal{N}(\mu_N, \sigma_N^2) \\ \mathbf{u} &= (\mu_1, \sigma_1^2, \cdots, \mu_N, \sigma_N^2) \in (\mathbb{R} \times \mathbb{R}_*^+)^N \subset \mathbb{R}^{2N}. \end{aligned}$$

That is:

$$\forall k \in \{1, \cdots, N\}, w_k \sim \mathcal{N}(\mu_k, \sigma_k^2),$$

where μ_k and σ_k^2 are learned parameters. Intuitively, μ_k and σ_k can be seen respectively as the theoretical value of w_k and the acceptable error margin between w_k and μ_k.

The prior distribution is also a diagonal Gaussian distribution:

$$\alpha = \mathcal{N}(0, \sigma_{0,1}^2) \otimes \cdots \otimes \mathcal{N}(0, \sigma_{0,N}^2),$$

where $(\sigma_{0,k}^2)_k$ *are fixed hyperparameters. Since we are agnostic about the sign of* w_k*, all the priors are centered in* 0*.*

Finally, the reparameterization trick consists in drawing each parameter w_k *this way:*

$$w_k = \mu_k + \sigma_k \cdot \xi_k, \qquad \textit{where} \qquad \xi_k \sim \mathcal{N}(0,1).$$

Then, the gradients with respect to μ_k *and* σ_k^2 *can easily be computed:*

$$\partial_{\mu_k} L_{\text{VI}} = \mathbb{E}_{\boldsymbol{\xi}} \partial_{w_k}[-\ln(p_{\mathbf{w}}(\mathcal{D}))] + \partial_{\mu_k} \text{KL}(\alpha \| \beta_{\mathbf{u}})$$
$$\partial_{\sigma_k^2} L_{\text{VI}} = \mathbb{E}_{\boldsymbol{\xi}} \left[\partial_{w_k}[-\ln(p_{\mathbf{w}}(\mathcal{D}))] \cdot \frac{\xi_k}{2\sigma_k} \right] + \partial_{\sigma_k^2} \text{KL}(\alpha \| \beta_{\mathbf{u}}).$$

1.4 Contributions

1.4.1 Facilitate Architecture Selection with Alrao

Outline. Usually, the learning rate is an hyperparameter to optimize each time a new combination task/neural network is tested. The same network is trained several times with different learning rates, and the best final neural network is selected, which is time-consuming. Therefore, I present in Chapter 2 the Alrao training technique, in which the learning rate has been removed from the set of hyperparameters to optimize. Notably, we claim that, for a wide range of neural networks and tasks, Alrao outputs *in a single run* a neural network that performs almost as well as the same network trained by SGD with optimized learning rate.

The intuition behind Alrao was to replace learning rate optimization by learning rate diversity inside each considered neural network. To summarize, each neuron is trained by SGD with its own learning rate, which is randomly sampled before training from some fixed distribution. This distribution is chosen such that it spans several orders of magnitude. So, for a large set of tasks and initial architectures, the training process is supposed to work at least for a fraction of the neurons.

Related work. Since Saxe [88], several works have been published around the importance of diversity inside the neural networks. Saxe studied *partially trained* neural networks made of an untrained convolutional hidden layer followed by a trained classifier. Surprisingly, the resulting neural networks achieved good results in relation to their size. Moreover, the author observed a performance correlation between these partially trained neural networks and their *fully trained* counterpart. He proposed to use it for fast architecture selection: in order to compare the relative performances of one-layered convolutional neural networks with different filter sizes, pooling sizes or filter strides, it is sufficient to compare their partially trained version, which are fast to obtain. Once the winning architecture is selected, all that remains is to fully train it.

This work of Saxe is highly interesting in its methodology, which can be summed up in the words: "diversity–proxy–selection". First, he showed empirically and proved theoretically that *diversity* in the weights is sufficient to compute meaningful features, that are easily usable for classification. Secondly, he showed that the performance of such raw networks is a good *proxy* for the performance of their fully trained counterpart. Finally, he used this observation to perform architecture *selection*. In a sense, we followed the same approach as Saxe with its random weights by proposing Alrao with random learning rates. But, unlike Saxe's method, Alrao is applicable to a far wider range of tasks and models, like deep and recurrent neural networks, and regression tasks.

More recently, Frankle et al. [20] showed that, given an untrained neural network, a subnetwork can be extracted and trained separately such that the resulting network performs closely to the trained whole network. This subnetwork is called the *winning ticket*, since its weights have been –by chance– well sampled. This observation can only be made if the whole network is initially composed of diverse neurons. Therefore, the authors propose to extract lottery tickets in order to get a better intuition of the working architectures, and then guide an architecture search.

With Alrao, we hope that, among the sampled learning rates, some are very well adapted to the architecture and will make the neural network learn to perform the task. Hence, it can be seen as an adaptation of [20], where the well sampled learning rates take the place of the well sampled neurons.

Personal contribution. This work has been made in collaboration with Léonard Blier and Yann Ollivier (facebook AI Research). My personal contribution lies mainly in the implementation of the main algorithm *Alrao* (in PyTorch), its front-end design, and experiments in the context of Natural Language Processing (NLP). Notably, I worked on the library design and I implemented the "switch" part of Alrao, which is the Bayesian ensemble method applied on the final layers of the network, in the case of classification and regression tasks, and for feedforward and recurrent neural networks. These parts were necessary in order to test Alrao in such many tasks and models, and thus ensure its robustness.

Moreover, I designed the code front-end to be easily used by other researchers, in a few lines of code. The final code is available on GitHub (`https://github.com/leonardblier/alrao`) and comes with a tutorial. This part ensures the reproducibility of our research and its diffusion in the Deep Learning community.

The experiments that I ran on NLP tasks with Recurrent Neural Networks are part of the main results of Alrao and contributed to demonstrate the stability of Alrao. In some extent, they were also useful to detect a limitation of Alrao: when performing word prediction, the computational cost of Alrao increases dramatically.

1.4.2 Stable Neural Network Training and Pruning with ScaLP

Outline. When training and pruning alternatively a neural network in order to find a good trade-off between final accuracy and number of parameters, the training algorithm (basically, the SGD) should be resilient to width change, since the width of the layers can change dramatically during the pruning phases. Moreover, the whole pruning strategy should output roughly the same network architecture at the end of each run. In the standard setup, I show that these stability issues are not addressed. Therefore, I present in Chapter 3 the techniques ScaLa and ScaLP, respectively to make the training algorithm resilient to width change, and to make the pruning strategy stable between runs.

Both ScaLa and ScaLP consist in inserting a fixed scaling layer before each layer of neurons: each of its inputs is scaled by a factor verifying a property that we detail later. In short, if this property is verified and the parameters are initialized according to $\mathcal{N}(0, 1)$, then we can ensure that the neural network will output a reasonable value at initialization and just after one update. In practice, we observe that ScaLa can be used to train neural networks of various widths *without changing the learning rate*, which is set to 1 in our experiments. In the same way, when pruning a neural network with ScaLP, the resulting network does not depend on the initial layer widths, provided that they are wide enough.

Related work and approach of the problem. The theoretical part of this work is mainly inspired by the work of Glorot about the initialization procedure for the weights of a neural network. According to Glorot, the weights should be initially drawn from a distribution with specific variance in order to preserve the variance of the activations along the network during the first forward pass. In the present work, this consideration is extended to the activations after one update and in the case where the width of each layer tends to infinity.

To some extent, the initial intuition about variance preservation is tested in a framework for which it has not been designed. This way, we are able to extract valuable information about the relevance of this intuition, and we discover that we can refine it with ScaLa. In the process, the new pruning setup ScaLP arises naturally, and can be used to perform stable pruning: each time a given neural network is pruned with ScaLP, the resulting network architecture is approximately the same.

1.4.3 Interpreting an Empirical Penalty as the Influence of a Prior in a Variational Inference Setup

Outline. In machine learning, the loss can usually be decomposed into two terms: the error term, which evaluates the quality of the model regarding the data, and the penalty term, which is usually set to make the model fit some user-defined property.

For instance, the user might want to restrain the number of non-zero parameters in the model, which can be done by designing a penalty that pushes towards zero the parameters.

Besides, the framework of variational inference provides a loss composed of two terms:

$$L_{\mathrm{VI}}(\mathbf{u}) = \underbrace{-\mathbb{E}_{\mathbf{w}\sim\beta_{\mathbf{u}}} \ln p_{\mathbf{w}}(\mathcal{D})}_{\text{error term}} + \underbrace{\mathrm{KL}(\beta_{\mathbf{u}} \| \alpha)}_{\text{penalty term}}.$$

The variational loss can easily be interpreted as a combination of an error term (the negative log-likelihood) and a penalty term (the influence of a prior distribution), which matches the empirical way to build a loss. Conversely, we ask the questions: "Given a penalty, is there always a way to interpret it as the influence of a prior? In that case, how can we compute this prior?"

Answering these theoretical questions for a given penalty has two implications. First, we are able to check whether a given optimization problem is in fact a way to approximate a Bayesian posterior. Second, if the penalty is parametrized by some hyperparameters, we are able to provide a necessary and sufficient condition over them in a dense case for the Bayesian interpretation of the penalty to be possible. The main theoretical result we provide is a formula allowing us to compute the corresponding prior α of a given penalty $r(\cdot)$.

As an application, given a penalty $r_\lambda(\cdot)$ parametrized by λ, we propose a way to compute a heuristic for λ in the case of neural networks. This heuristic is a combination of the formula mentioned above and Glorot's intuition about the variance of initialization distributions: assuming that a Bayesian prior should be usable to initialize the parameters of a neural network, we constrain its variance. So, if a parametrized penalty r_λ can be interpreted as the influence of a parametrized prior α_λ, and we impose a condition over the variance of α_λ, then we impose a condition over the penalty r_λ itself, that is over its parameter λ.

Related work and approach of the problem. In the Bayesian framework, the prior knowledge of the user, or equivalently its goal, is explicitly embedded into the prior distribution and nowhere else. On the contrary, in empirically designed training methods, the prior knowledge of the user is spread out over the penalty terms, its hyperparameters and other regularization tricks (as the *drop-out*). Therefore, interpreting a regularization method as the influence of a prior in a Bayesian framework is a way of isolating the prior knowledge, which is then expressed as a probability distribution.

Since the variational inference framework provides a standardized way to translate regularization methods in Bayesian terms, some work has already been done to interpret regularization methods in variational terms. For example, the *drop-out* method has been studied from a variational point of view [80], which led to several interesting results including a pruning technique. The theoretical results presented

here are a further investigation of the relation regularization–prior in the case where the regularization is in fact a penalization of the loss.

At this point, such reduction of several regularization techniques to the same object may appear as mostly theoretical and limited to a question of aesthetics. However, seeing a penalty as the influence of a prior distribution allows us to constrain it through unintended heuristics, as Glorot's condition. The latter is usually used to constrain the initialization distribution, but we can now use it to constrain the penalty through its corresponding prior distribution.

Chapter 2

Learning with Random Learning Rates

Based on a paper written in collaboration with Léonard Blier and Yann Ollivier. Published and presented at ECML PKDD 2019.

2.1 Introduction

Deep learning models require delicate hyperparameter tuning [113]: when facing new data or new model architectures, finding a configuration that enables fast learning requires both expert knowledge and extensive testing. This prevents deep learning models from working out-of-the-box on new problems without human intervention (AutoML setup, [25]). One of the most critical hyperparameters is the learning rate of the gradient descent [100, p. 892]. With too large learning rates, the model does not learn; with too small learning rates, optimization is slow and can lead to local minima and poor generalization [42, 53, 68, 98].

Efficient methods with no learning rate tuning are a necessary step towards more robust learning algorithms, ideally working out of the box. Many methods were designed to directly set optimal per-parameter learning rates [103, 17, 48, 90, 55], such as the popular Adam optimizer. The latter comes with default hyperparameters which reach good performance on many problems and architectures; yet fine-tuning and scheduling of its learning rate is still frequently needed [15], and the default setting is specific to current problems and architecture sizes. Indeed Adam's default hyperparameters fail in some natural setups (Section 2.6.2). This makes it unfit in an out-of-the-box scenario.

We propose *All Learning Rates At Once* (Alrao), a gradient descent method for deep learning models that leverages redundancy in the network. Alrao uses multiple learning rates at the same time in the same network, spread across several orders of magnitude. This creates a mixture of slow and fast learning units. Alrao departs from the usual philosophy of trying to find the "right" learning rates; instead we take advantage of the overparameterization of network-based models to produce a

diversity of behaviors from which good network outputs can be built. The width of the architecture may optionally be increased to get enough units within a suitable learning rate range, but surprisingly, performance was largely satisfying even without increasing width.

Our contributions are as follows:

- We introduce Alrao, a gradient descent method for deep learning models with no learning rate tuning, leveraging redundancy in deep learning models via a range of learning rates in the same network. Surprisingly, Alrao does manage to learn well over a range of problems from image classification, text prediction, and reinforcement learning.

- In our tests, Alrao's performance is always close to that of SGD with the optimal learning rate, without any tuning.

- Alrao combines performance with *robustness*: not a single run failed to learn with the default learning rate range we used. In contrast, our parameter-free baseline, Adam with default hyperparameters, is not reliable across the board.

- Alrao vindicates the role of redundancy in deep learning: having enough units with a suitable learning rate is sufficient for learning.

Acknowledgments. We would like to thank Corentin Tallec for his technical help and extensive remarks. We thank Olivier Teytaud for pointing useful references, Hervé Jégou for advice on the text, and Léon Bottou, Guillaume Charpiat, and Michèle Sebag for their remarks on our ideas.

2.2 Related Work

Redundancy in deep learning. Alrao specifically exploits the redundancy of units in network-like models. Several lines of work underline the importance of such redundancy in deep learning. For instance, dropout [95] relies on redundancy between units. Similarly, many units can be pruned after training without affecting accuracy [56, 26, 28, 92]. Wider networks have been found to make training easier [7, 33, 111], even if not all units are useful a posteriori.

The *lottery ticket hypothesis* [19, 20] posits that "large networks that train successfully contain subnetworks that—when trained in isolation—converge in a comparable number of iterations to comparable accuracy'. This subnetwork is the *lottery ticket winner*: the one which had the best initial values. In this view, redundancy helps because a larger network has a larger probability to contain a suitable subnetwork. Alrao extends this principle to the learning rate.

Learning rate tuning. Automatically using the "right" learning rate for each parameter was one motivation behind "adaptive" methods such as RMSProp [103], AdaGrad [17] or Adam [48]. Adam with its default setting is currently considered the default method in many works [109]. However, further global adjustment of the Adam learning rate is common [63]. Other heuristics for setting the learning rate have been proposed [90]; these heuristics often start with the idea of approximating a second-order Newton step to define an optimal learning rate [55]. Indeed, asymptotically, an arguably optimal preconditioner is either the Hessian of the loss (Newton method) or the Fisher information matrix [2]. Another approach is to perform gradient descent on the learning rate itself through the whole training procedure [40, 91, 74, 73, 77, 5]. Despite being around since the 80's [40], this has not been widely adopted, because of sensitivity to hyperparameters such as the meta-learning rate or the initial learning rate [18]. Of all these methods, Adam is probably the most widespread at present [109], and we use it as a baseline.

The learning rate can also be optimized within the framework of architecture or hyperparameter search, using methods from from reinforcement learning [113, 4, 61], evolutionary algorithms [96, 45, 86], Bayesian optimization [8], or differentiable architecture search [64]. Such methods are resource-intensive and do not allow us to find a good learning rate in a single run.

2.3 Motivation and Outline

We first introduce the general ideas behind Alrao. The detailed algorithm is explained in Section 2.4 and in Algorithm 1. We also release a Pytorch [84] implementation, including tutorials: http://github.com/leonardblier/alrao.

Different learning rates for different units. Instead of using a single learning rate for the model, Alrao samples once and for all a learning rate for each *unit* in the network. These rates are taken from a log-uniform distribution in an interval $[\eta_{\min}; \eta_{\max}]$. The log-uniform distribution produces learning rates spread over several order of magnitudes, mimicking the log-uniform grids used in standard grid searches on the learning rate.

A *unit* corresponds for example to a feature or neuron for fully connected networks, or to a channel for convolutional networks. Thus we build "slow-learning" and "fast-learning" units. In contrast, with per-parameter learning rates, every unit would have a few incoming weights with very large learning rates, and possibly diverge.

Intuition. Alrao is inspired by the fact that not all units in a neural network end up being useful. Our idea is that in a large enough network with learning rates sampled randomly per unit, a sub-network made of units with a good learning rate will learn well, while the units with a wrong learning rate will produce useless values

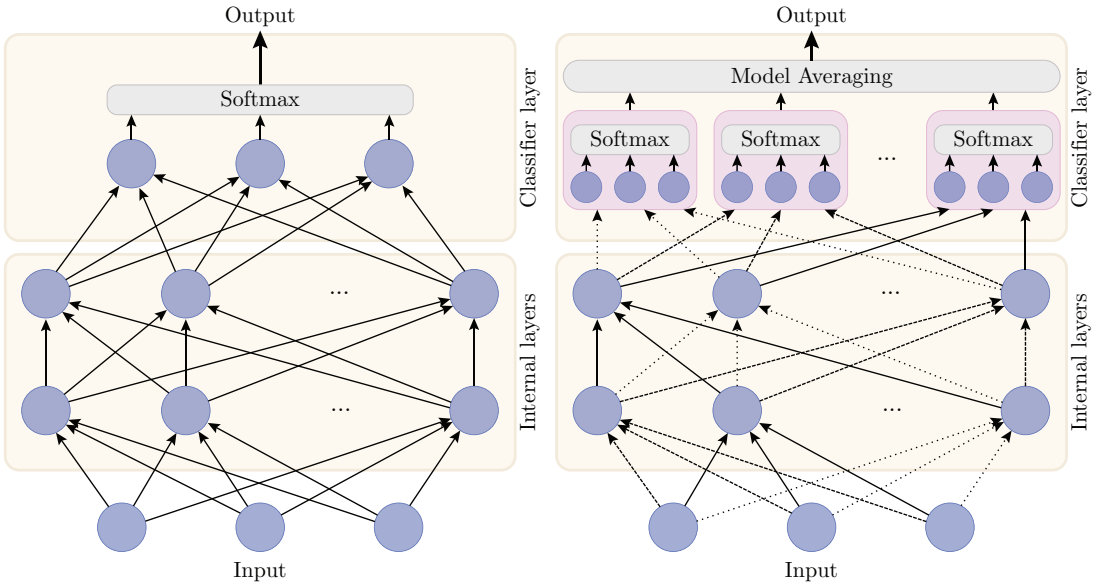


Figure 2.1 – Left: a standard fully connected neural network for a classification task with three classes, made of several internal layers and an output layer. Right: Alrao version of the same network. The single classifier layer is replaced with a set of parallel copies of the original classifier, averaged with a model averaging method. Each unit uses its own learning rate for its incoming weights (represented by different styles of arrows).

and just be ignored by the rest of the network. Units with too small learning rates will not learn anything and stay close to their initial values; this does not hurt training (indeed, even leaving some weights at their initial values, corresponding to a learning rate 0, does not hurt training). Units with a too large learning rate may produce large activation values, but those will be mitigated by subsequent normalizing mechanisms in the computational graph, such as sigmoid/tanh activations or BatchNorm.

Alrao can be interpreted within the *lottery ticket hypothesis* [19]: viewing the per-unit learning rates of Alrao as part of the initialization, this hypothesis suggests that in a wide enough network, there will be a sub-network whose initialization (both values and learning rate) leads to good convergence.

Slow and fast learning units for the output layer. Sampling a learning rate per unit at random in the last layer would not make sense. For classification, each unit in the last layer represents a single category: using different learning rates for these units would favor some categories during learning. Moreover for scalar regression tasks there is only one output unit, thus we would be back to selecting a single learning rate.

The simplest way to obtain the best of several learning rates for the last layer,

without relying on heuristics to guess an optimal value, is to use *model averaging* over several copies of the output layer (Fig. 2.1), each copy trained with its own learning rate from the interval $[\eta_{\min}; \eta_{\max}]$. All these untied copies of the output layer share the same Alrao internal layers (Fig. 2.1). This can be seen as a smooth form of model selection or grid-search over the output layer learning rate; actually, this part of the architecture can even be dropped after a few epochs, as the model averaging quickly concentrates on one model.

Increasing network width. With Alrao, neurons with unsuitable learning rates will not learn: those with too large learning rates might learn no useful signal, while those with too small learning rates will learn too slowly. Thus, Alrao may reduce the *effective width* of the network to only a fraction of the actual architecture width, depending on $[\eta_{\min}; \eta_{\max}]$. This may be compensated by multiplying the width of the network by a factor γ. Our first intuition was that $\gamma > 1$ would be necessary; still Alrao turns out to work well even without width augmentation.

2.4 All Learning Rates At Once: Description

2.4.1 Notation

We now describe Alrao more precisely for deep learning models with softmax output, on classification tasks; the case of regression is similar.

Let $\mathcal{D} = \{(x_1, y_1), ..., (x_n, y_n)\}$, with $y_i \in \{1, ..., K\}$, be a classification dataset. The goal is to predict the y_i given the x_i, using a deep learning model Φ_θ. For each input x, $\Phi_\theta(x)$ is a probability distribution over $\{1, ..., K\}$, and we want to minimize the categorical cross-entropy loss ℓ over the dataset: $\frac{1}{n}\sum_i \ell(\Phi_\theta(x_i), y_i)$.

We denote $\log\text{-}\mathcal{U}(\cdot; \eta_{\min}, \eta_{\max})$ the *log-uniform* probability distribution on an interval $[\eta_{\min}; \eta_{\max}]$. Namely, if $\eta \sim \log\text{-}\mathcal{U}(\cdot; \eta_{\min}, \eta_{\max})$, then $\log \eta$ is uniformly distributed between $\log \eta_{\min}$ and $\log \eta_{\max}$. Its density function is $\log\text{-}\mathcal{U}(\eta; \eta_{\min}, \eta_{\max}) = \frac{1}{\eta} \frac{\mathbb{1}_{\eta_{\min} \leq \eta \leq \eta_{\max}}}{\log(\eta_{\max}) - \log(\eta_{\min})}$.

2.4.2 Alrao Architecture

Multiple Alrao output layers. A deep learning model Φ_θ for classification can be decomposed into two parts: first, *internal layers* compute some function $z = \phi_{\theta^{\text{int}}}(x)$ of the inputs x, fed to a final *output (classifier) layer* $C_{\theta^{\text{out}}}$, so that the overall network output is $\Phi_\theta(x) := C_{\theta^{\text{out}}}(\phi_{\theta^{\text{int}}}(x))$. For a classification task with K categories, the output layer $C_{\theta^{\text{out}}}$ is defined by $C_{\theta^{\text{out}}}(z) := \text{softmax}\left(W^T z + b\right)$ with $\theta^{\text{out}} := (W, b)$, and $\text{softmax}(u_1, ..., u_K)_k := e^{u_k} / \left(\sum_i e^{u_i}\right)$.

In Alrao, we build multiple copies of the original output layer, with different learning rates for each, and then use a model averaging method among them. The

Algorithm 1 Alrao-SGD for model $\Phi_\theta = C_{\theta^{\text{out}}}(\phi_{\theta^{\text{r}}}(\cdot))$ with N_{out} classifiers and learning rates in $[\eta_{\min}; \eta_{\max}]$

$a_j \leftarrow 1/N_{\text{out}}$ for each $1 \leq j \leq N_{\text{out}}$ ▷ Init. the N_{out} model averaging weights a_j
$\Phi_\theta^{\text{Alrao}}(x) := \sum_{j=1}^{N_{\text{out}}} a_j \, C_{\theta_j^{\text{out}}}(\phi_{\theta^{\text{int}}}(x))$ ▷ Define the Alrao architecture
for all layers l, **for all** unit i in layer l **do**
    Sample $\eta_{l,i} \sim \log\text{-}\mathcal{U}(.; \eta_{\min}, \eta_{\max})$. ▷ Sample a learning rate for each unit
for all Classifiers j, $1 \leq j \leq N_{\text{out}}$ **do**
    Define $\log \eta_j = \log \eta_{\min} + \frac{j-1}{N_{\text{out}}-1} \log \frac{\eta_{\max}}{\eta_{\min}}$. ▷ Set a learning rate per classifier
while Stopping criterion is False **do**
    $z_t \leftarrow \phi_{\theta^{\text{int}}}(x_t)$ ▷ Store the output of the last internal layer
    for all layers l, **for all** unit i in layer l **do**
        $\theta_{l,i} \leftarrow \theta_{l,i} - \eta_{l,i} \nabla_{\theta_{l,i}} \ell(\Phi_\theta^{\text{Alrao}}(x_t), y_t)$ ▷ Update the repr. netw. weights
    for all Classifier j **do**
        $\theta_j^{\text{out}} \leftarrow \theta_j^{\text{out}} - \eta_j \nabla_{\theta_j^{\text{out}}} \ell(C_{\theta_j^{\text{out}}}(z_t), y_t)$ ▷ Update the classifiers' weights
    $a \leftarrow \text{ModelAveraging}(a, (C_{\theta_i^{\text{out}}}(z_t))_i, y_t)$ ▷ Update the mdl. averaging weights
    $t \leftarrow t + 1 \bmod N$

averaged classifier and the overall Alrao model are:

$$C_{\theta^{\text{out}}}^{\text{Alrao}}(z) := \sum_{j=1}^{N_{\text{out}}} a_j \, C_{\theta_j^{\text{out}}}(z), \qquad \Phi_\theta^{\text{Alrao}}(x) := C_{\theta^{\text{out}}}^{\text{Alrao}}(\phi_{\theta^{\text{int}}}(x)), \tag{2.1}$$

where the $C_{\theta_j^{\text{out}}}$ are copies of the original classifier layer, with non-tied parameters, and $\theta^{\text{out}} := (\theta_1^{\text{out}}, ..., \theta_{N_{\text{out}}}^{\text{out}})$. The a_j are the parameters of the model averaging, with $0 \leq a_j \leq 1$ and $\sum_j a_j = 1$. The a_j are not updated by gradient descent, but via a model averaging method from the literature (see below).

Increasing the width of internal layers. As explained in Section 2.3, we may compensate the effective width reduction in Alrao by multiplying the width of the network by a factor γ. This means multiplying the number of units (or filters for a convolutional layer) of all internal layers by γ.

2.4.3 Alrao Update for the Internal Layers: A Random Learning Rate for Each Unit

In the internal layers, for each unit i in each layer l, a learning rate $\eta_{l,i}$ is sampled from the probability distribution $\log\text{-}\mathcal{U}(.; \eta_{\min}, \eta_{\max})$, once and for all at the beginning of training. [1]

[1] With learning rates resampled at each time, each step would be, in expectation, an ordinary SGD step with learning rate $\mathbb{E}\eta_{l,i}$, thus just yielding an ordinary SGD trajectory with more variance.

The incoming parameters of each unit in the internal layers are updated in the usual SGD way, only with per-unit learning rates (Eq. 2.2): for each unit i in each layer l, its incoming parameters are updated as:

$$\theta_{l,i} \leftarrow \theta_{l,i} - \eta_{l,i} \cdot \nabla_{\theta_{l,i}} \ell(\Phi_\theta^{\text{Alrao}}(x), y), \tag{2.2}$$

where $\Phi_\theta^{\text{Alrao}}$ is the Alrao loss (2.1) defined above.

What constitutes a *unit* depends on the type of layers in the model. In a fully connected layer, each component of a layer is considered as a unit for Alrao: all incoming weights of the same unit share the same Alrao learning rate. On the other hand, in a convolutional layer we consider each convolution filter as constituting a unit: there is one learning rate per filter (or channel), thus preserving translation-invariance over the input image. In LSTMs, we apply the same learning rate to all components in each LSTM cell (thus the vector of learning rates is the same for input gates, for forget gates, etc.).

We set a learning rate *per unit*, rather than per parameter. Otherwise, every unit would have some parameters with large learning rates, and we would expect even a few large incoming weights to be able to derail a unit. Having diverging parameters within every unit is hurtful, while having diverging units in a layer is not necessarily hurtful since the next layer can learn to disregard them.

2.4.4 Alrao Update for the Output Layer: Model Averaging from Output Layers Trained with Different Learning Rates

Learning the output layers. The j-th copy $C_{\theta_j^{\text{out}}}$ of the classifier layer is attributed a learning rate η_j defined by $\log \eta_j := \log \eta_{\min} + \frac{j-1}{N_{\text{out}}-1} \log\left(\frac{\eta_{\max}}{\eta_{\min}}\right)$, so that the classifiers' learning rates are log-uniformly spread on the interval $[\eta_{\min}; \eta_{\max}]$. Then the parameters θ_j^{out} of each classifier j are updated as if this classifier alone was the only output of the model:

$$\theta_j^{\text{out}} \leftarrow \theta_j^{\text{out}} - \eta_j \cdot \nabla_{\theta_j^{\text{out}}} \ell(C_{\theta_j^{\text{out}}}(\phi_{\theta^{\text{int}}}(x)), y), \tag{2.3}$$

(still sharing the same internal layers $\phi_{\theta^{\text{int}}}$). This ensures that classifiers with low weights a_j still learn, and is consistent with model averaging philosophy. Algorithmically this requires differentiating the loss N_{out} times with respect to the last layer, but no additional backpropagations through the internal layers.

Model averaging. To set the weights a_j, several model averaging techniques are available, such as Bayesian Model Averaging [107]. We use the *Switch* model averaging [104] (details provided in Section 2.9.1), a Bayesian method which is both simple, principled, and very responsive to changes in performance of the various models. After each mini-batch, the switch computes a modified posterior distribution (a_j) over the classifiers. This computation is directly taken from [104].

Additional experiments show that the model averaging method acts like a smooth model selection procedure: after only a few hundreds gradient steps, a single output layer is selected, with its parameter a_j very close to 1 (details provided in Section 2.9.2). Actually, Alrao's performance is unchanged if the extraneous output layer copies are thrown away when the posterior weight a_j of one of the copies gets close to 1.

Table 2.1 – Performance of Alrao, SGD with tuned learning rate, and Adam with its default setting. Three convolutional models are reported for image classification on CIFAR10, three others for ImageNet, one recurrent model for character prediction (Penn Treebank), and two experiments on RL problems. Four of the image classification architectures are further tested with a width multiplication factor $\gamma = 3$. Alrao learning rates are taken in a wide, a priori reasonable interval $[\eta_{\min}; \eta_{\max}] = [10^{-5}; 10]$, and the optimal learning rate for SGD is chosen in the set $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1., 10.\}$. Each experiment is run 10 times (CIFAR10 and RL), 5 times (PTB) or 1 time (ImageNet); the confidence intervals report the standard deviation over these runs. For RL tasks, the return has to be maximized, not minimized.

Model	SGD with optimal LR			Adam - Default		Alrao	
	LR	Loss	Top1 (%)	Loss	Top1 (%)	Loss	Top1 (%)
CIFAR10							
MobileNet	0.1	$0.37 \pm .01$	$90.2 \pm .3$	$1.01 \pm .95$	78 ± 11	$0.42 \pm .02$	$88.1 \pm .6$
MobileNet, $\gamma = 3$	0.1	$0.33 \pm .01$	$90.3 \pm .5$	$0.32 \pm .02$	$90.8 \pm .4$	$0.35 \pm .01$	$89.0 \pm .6$
GoogLeNet	0.01	$0.45 \pm .05$	$89.6 \pm 1.$	$0.47 \pm .04$	$89.8 \pm .4$	$0.47 \pm .03$	$88.9 \pm .8$
GoogLeNet, $\gamma = 3$	0.1	$0.34 \pm .02$	$90.5 \pm .8$	$0.41 \pm .02$	$88.6 \pm .6$	$0.37 \pm .01$	$89.8 \pm .8$
VGG19	0.1	$0.42 \pm .02$	$89.5 \pm .2$	$0.43 \pm .02$	$88.9 \pm .4$	$0.45 \pm .03$	$87.5 \pm .4$
VGG19, $\gamma = 3$	0.1	$0.35 \pm .01$	$90.0 \pm .6$	$0.37 \pm .01$	$89.5 \pm .8$	$0.381 \pm .004$	$88.4 \pm .7$
ImageNet							
AlexNet	0.01	2.15	53.2	6.91	0.10	2.56	43.2
Densenet121	1	1.35	69.7	1.39	67.9	1.41	67.3
ResNet50	1	1.49	67.4	1.39	67.1	1.42	67.5
ResNet50, $\gamma = 3$	-	-	-	1.99	60.8	1.33	70.9
Penn Treebank							
LSTM	1	$1.566 \pm .003$	$66.1 \pm .1$	$1.587 \pm .005$	$65.6 \pm .1$	$1.706 \pm .004$	$63.4 \pm .1$
RL		Return		Return		Return	
Pendulum	0.0001	-372 ± 24		-414 ± 64		-371 ± 36	
LunarLander	0.1	188 ± 23		155 ± 23		186 ± 45	

2.5 Experimental Setup

We tested Alrao on various convolutional networks for image classification (Imagenet and CIFAR10), on LSTMs for text prediction, and on reinforcement learning problems. We always use the same learning rate interval $[10^{-5}; 10]$, corresponding to the values we would have tested in a grid search, and 10 Alrao output layer copies, for every task.

We compare Alrao to SGD with an optimal learning rate selected in the set $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1., 10.\}$, and, as a tuning-free baseline, to Adam with its default setting ($\eta = 10^{-3}, \beta_1 = 0.9, \beta_2 = 0.999$), arguably the current default method [109].

The results are presented in Table 2.1. Fig. 2.2 presents learning curves for AlexNet and Resnet50 on ImageNet, and additional details are provided in Section 2.9.3.

2.5.1 Image Classification on ImageNet and CIFAR10

For image classification, we used the ImageNet [13] and CIFAR10 [51] datasets. The ImageNet dataset is made of 1,283,166 training and 60,000 testing data; we split the training set into a smaller training set and a validation set with 60,000 samples. We do the same on CIFAR10: the 50,000 training samples are split into 40,000 training samples and 10,000 validation samples.

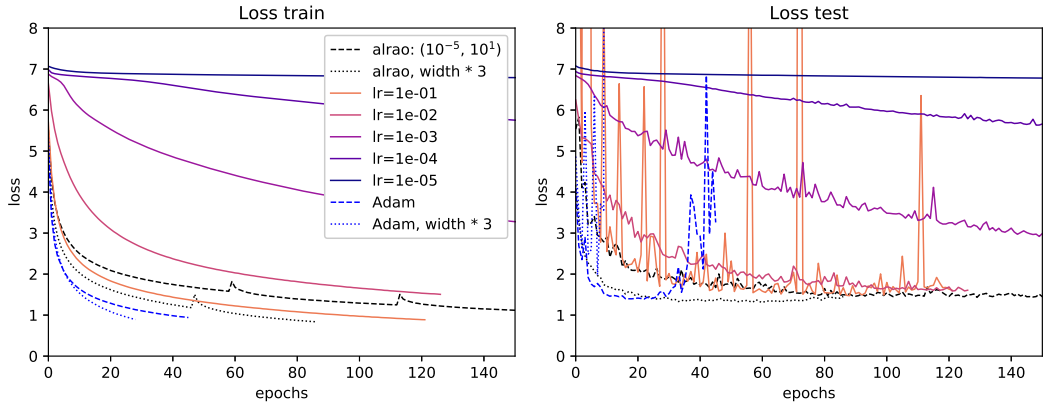
For each architecture, training was stopped when the validation loss had not improved for 20 epochs. The epoch with best validation loss was selected and the corresponding model tested on the test set. The inputs are normalized, and training used data augmentation: random cropping and random horizontal flipping. For CIFAR10, each setting was run 10 times: the confidence intervals presented are the standard deviation over these runs. For ImageNet, because of high computation time, we performed only a single run per experiment.

We tested Alrao on several standard architectures. On ImageNet, we tested Resnet50 [31], Densenet121 [37], and Alexnet [52], using the default Pytorch implementation. On CIFAR10, we tested GoogLeNet [99], VGG19 [93], and MobileNet [36], as implemented in [47]. We also tested wider architectures, with a width multiplication factor $\gamma = 3$. On the largest model, Resnet50 on ImageNet with triple width, systematic SGD learning rate grid search was not performed due to the excessive computational burden, hence the omitted value in Tab. 2.1.

2.5.2 Other Tasks: Text Prediction, Reinforcement Learning

Text prediction on Penn TreeBank. To test Alrao on other kinds of tasks, we first used a recurrent neural network for text prediction on the Penn Treebank (PTB) [76] dataset. The Alrao experimental procedure is the same as above.

(a) Resnet50 trained on ImageNet.



(b) AlexNet trained on ImageNet.

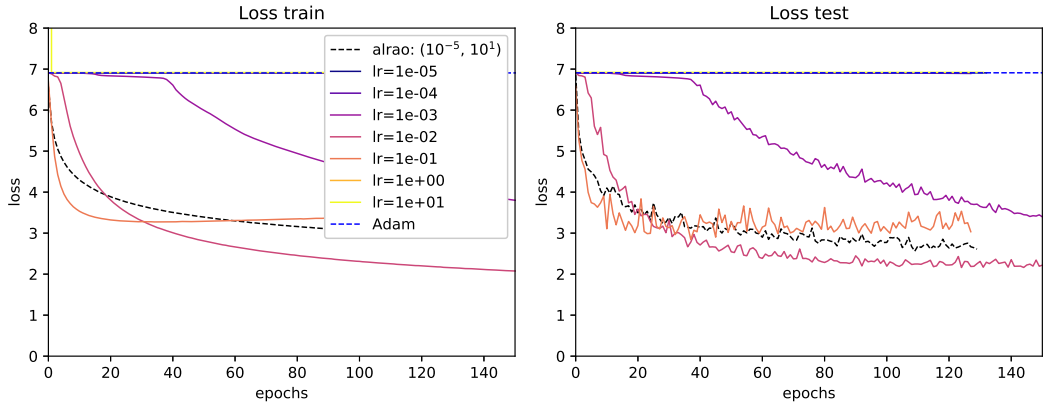


Figure 2.2 – Learning curves for Alrao, SGD with various learning rates, and Adam with its default setting, on ImageNet. Left: training loss; right: test loss. Curves are interrupted by the early stopping criterion. Alrao's performance is comparable to the optimal SGD learning rate.

The loss in Table 2.1 is given in bits per character and the accuracy is the proportion of correct character predictions. The model is a two-layer LSTM [35] with an embedding size of 100, and 100 hidden units. A dropout layer with rate 0.2 is included before the decoder. The training set is divided into 20 minibatchs. Gradients are computed via truncated backprop through time [108] with truncation every 70 characters.

The model was trained for *character* prediction rather than word prediction. This is technically easier for Alrao implementation: since Alrao uses copies of the output layer, memory issues arise for models with most parameters on the output layer. Word prediction (10,000 classes on PTB) requires many more output parameters than character prediction; see Section 2.7.

Reinforcement learning tasks. Next, we tested Alrao on two standard reinforcement learning problems: the Pendulum and Lunar Lander environments from OpenAI Gym [10]. We use standard deep Q-learning [79]. The Q-network is a standard MLP with 2 hidden layers. The experimental setting is the same as above, with regressors instead of classifiers on the output layer. For each environment, we select the best epoch on validation runs, and then report the return of the selected model on new test runs in that environment.

2.6 Performance and Robustness of Alrao

2.6.1 Alrao Compared to SGD with Optimal Learning Rate

First, Alrao does manage to learn; this was not obvious a priori.

Second, SGD with an optimally tuned learning rate usually performs better than Alrao. This can be expected when comparing a tuning-free method with a method that tunes the hyperparameter in hindsight.

Still, the difference between Alrao and optimally-tuned SGD is reasonably small across every setup, even with wide intervals $[\eta_{\min}; \eta_{\max}]$, with a somewhat larger gap in one case (AlexNet on ImageNet). Notably, this occurs even though SGD achieves good performance only for a few learning rates within the interval $[\eta_{\min}; \eta_{\max}]$. With $\eta_{\min} = 10^{-5}$ and $\eta_{\max} = 10$, among the 7 SGD learning rates tested ($10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$, and 10), only three are able to learn with AlexNet, and only one is better than Alrao (Fig. 2.2b); with ResNet50, only three are able to learn well, and only two of them achieve performance similar to Alrao (Fig. 2.2a); on the Pendulum environment, only two are able to learn well, only one of which converges as fast as Alrao.

Thus, surprisingly, Alrao manages to learn at a nearly optimal rate, even though most units in the network have learning rates unsuited for SGD.

2.6.2 Robustness of Alrao, and Comparison to Default Adam

Overall, Alrao learns reliably in every setup in Table 2.1. Moreover, this is quite stable over the course of learning: Alrao curves shadow optimal SGD curves over time (Fig. 2.2).

Often, Adam with its default parameters almost matches optimal SGD, but this is not always the case. Over the 13 setups in Table 2.1, default Adam gives a significantly poor performance in three cases. One of those is a pure optimization issue: with AlexNet on ImageNet, optimization does not start with the default parameters (Fig. 2.2b). The other two cases are due to strong overfit despite good train performance: MobileNet on CIFAR and ResNet with increased width on ImageNet.

In two further cases, Adam achieves good validation performance in Table 2.1, but actually overfits shortly after its peak score: ResNet (Fig. 2.2a) and DenseNet, [109, 46].

Overall, default Adam tends to give slightly better results than Alrao when it works, but does not learn reliably with its default hyperparameters. It can exhibit two kinds of lack of robustness: optimization failure, and overfit or non-robustness over the course of learning. On the other hand, every single run of Alrao reached reasonably close-to-optimal performance. Alrao also performs steadily over the course of learning (Fig. 2.2).

2.6.3 Sensitivity Study to $[\eta_{\min}; \eta_{\max}]$

We claim to remove a hyperparameter, the learning rate, but replace it with two hyperparameters $\eta_{\min}$ and $\eta_{\max}$. Formally, this is true. But a systematic study of the impact of these two hyperparameters (Fig. 2.3) shows that the sensitivity to $\eta_{\min}$ and $\eta_{\max}$ is much lower than the original sensitivity to the learning rate.

To assess this, we tested every combination of $\eta_{\min}$ and $\eta_{\max}$ in a grid from 10^{-9} to 10^{7} on GoogLeNet for CIFAR10 (left plot in Fig. 2.3, with SGD on the diagonal). The largest satisfactory learning rate for SGD is 1 (diagonal on Fig. 2.3). Unsurprisingly, if all the learning rates in Alrao are too large, or all too small, then Alrao fails (rightmost and leftmost zones in Fig. 2.3). Extremely large learning rates diverge numerically, both for SGD and Alrao.

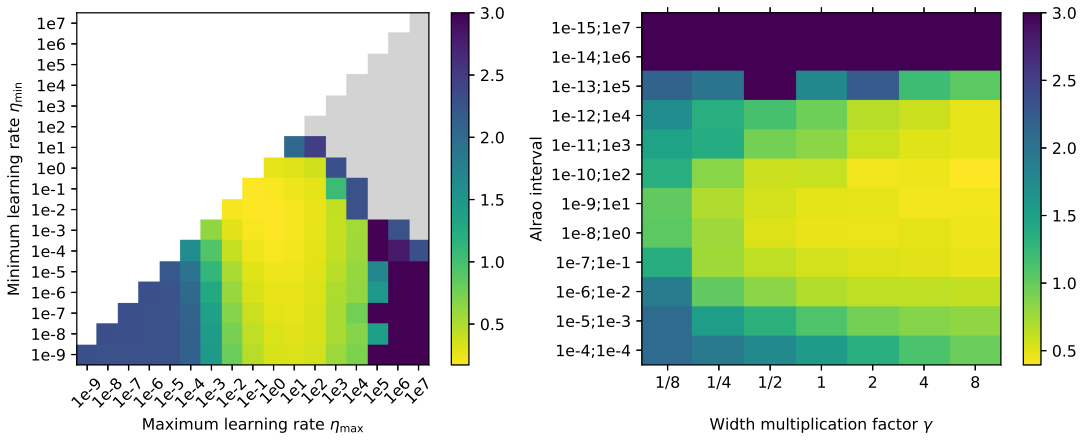


Figure 2.3 – Influence of $[\eta_{\min}; \eta_{\max}]$ and of network width on Alrao performance, with GoogLeNet on CIFAR10. Results are reported after 15 epochs, and averaged on three runs. Left plot: each point with coordinates $[\eta_{\min}; \eta_{\max}]$ below the diagonal represents the loss for Alrao with this interval. Points (η, η) on the diagonal represent standard SGD with learning rate η. Grey squares represent numerical divergence (NaN). Alrao works as soon as $[\eta_{\min}; \eta_{\max}]$ contains at least one suitable learning rate. Right plot: varying network width.

On the other hand, Alrao converges as soon as $[\eta_{\min}; \eta_{\max}]$ contains a reasonable learning rate (central zone Fig. 2.3), even with values of $\eta_{\max}$ for which SGD fails. A wide range of choices for $[\eta_{\min}; \eta_{\max}]$ will contain one good learning rate and achieve close-to-optimal performance. Thus, as a general rule, we recommend to just use an interval containing all the learning rates that would have been tested in a grid search, e.g., 10^{-5} to 10.

For a fixed network size, one might expect Alrao to perform worse with large intervals $[\eta_{\min}; \eta_{\max}]$, as most units would become useless. On the other hand, in a larger network, many units would have extreme learning rates, which might disturb learning. We tested how increasing or decreasing network width changes Alrao's sensitivity to $[\eta_{\min}; \eta_{\max}]$ (right plot of Fig. 2.3 for Alrao). The sensitivity of Alrao to $[\eta_{\min}; \eta_{\max}]$ decreases markedly with network width. For instance, a wide interval $[\eta_{\min}; \eta_{\max}] = [10^{-12}; 10^{4}]$ works reasonably well with an 8-fold network, even though most units receive unsuitable learning rates.

So, even if the choice of $\eta_{\min}$ and $\eta_{\max}$ is important, the results are much more stable to varying these two hyperparameters than to the original learning rate, especially with large networks.

2.6.4 Pruning Layers after Training

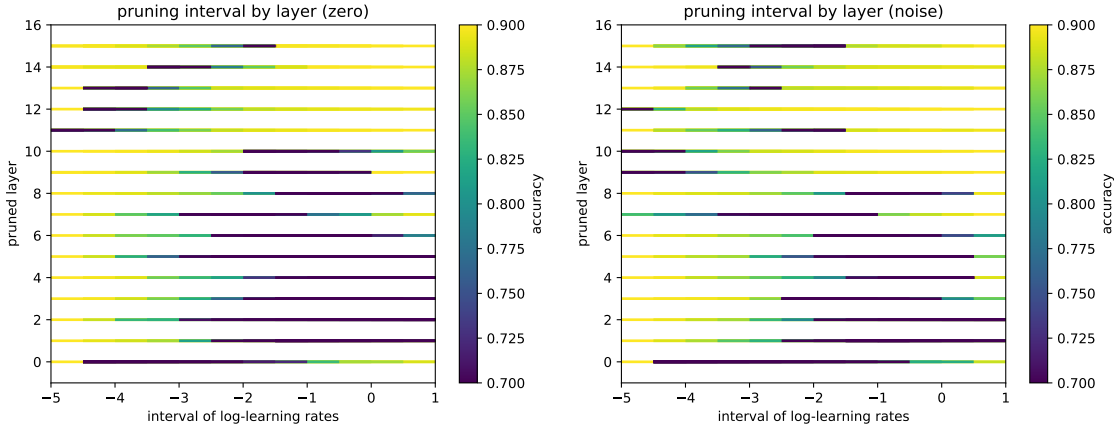
With Alrao, neurons are trained with learning rates lying in a wide interval. Our initial intuition is that neurons trained with a too large or too small learning rate will be progressively ignored by neurons on the next layer.

In this section, we check experimentally this intuition by pruning neurons at the end of training. For each layer, we try to prune its neurons while the other layers remain unchanged (see Algorithm 2). The considered layer is sequentially pruned by progressively removing neurons with the highest or lowest learning rate, and which affect the performance the least. This way, we delimit the interval of learning rates with which the most useful neurons have been trained.

In order to understand the behavior of the neurons, we have tested independently three pruning techniques on VGG16: simply set the weights of the pruned neurons to zero; replace pruned neurons by Gaussian noise; reinitialize pruned neurons to their value at initialization.

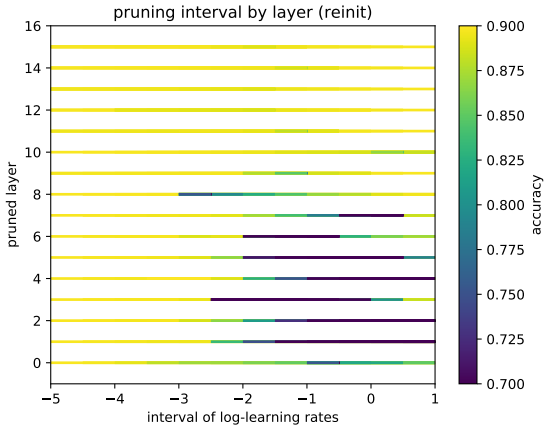
First, Figures 2.4a and 2.4b differ significantly from Figure 2.4c: reinitializing neurons leads to better results than simply replacing them with zero or noise. This observation corroborates the observations made in [112], where it has been shown than some layers of VGG (in particular the last ones) can be reinitialized with few accuracy drop. Besides, it is not surprising that reinitializing neurons with small learning rates does not degrade accuracy much.

Second, according to Figure 2.4a, it is very difficult to simply remove neurons in most layers, even neurons with a "too large" or "too small" learning rate. At the very least, we could remove neurons trained with a learning rate greater than 1 or 0.1, but this is feasible only in the five last layers. Results given in Figure 2.4b are



(a)

(b)



(c)

Figure 2.4 – These figures show how much each layer can be individually pruned, and how pruning affects the accuracy, which is initially 87.81 %. Each horizontal bar at ordinate l represents how much the l-th layer can be pruned: the larger a purple area is, the least the corresponding layer can be pruned without accuracy loss.

Algorithm 2 Pruning the neurons of one layer l in the final neural network.
We denote by $\mathcal{M}$ the final neural network, by $\mathcal{M}_l[a_{\min}, a_{\max}]$ a copy of $\mathcal{M}$ where all neurons in layer l trained with a learning rate $\eta \notin [a_{\min}, a_{\max}]$ have been pruned.

$a_{\min}, a_{\max} \leftarrow \eta_{\min}, \eta_{\max}$
$\mathrm{d}a \leftarrow 2$ ▷ rate of growth of $\eta_{\min}$ and decay of $\eta_{\max}$
while $a_{\min} < a_{\max}$ **do**
    $\mathcal{M}_{\min} \leftarrow \mathcal{M}_l[a_{\min} \cdot \mathrm{d}a, a_{\max}]$
    $\mathcal{M}_{\max} \leftarrow \mathcal{M}_l[a_{\min}, a_{\max}/\mathrm{d}a]$ ▷ test narrower intervals of lr
    if $\mathcal{M}_{\min}$ performs better than $\mathcal{M}_{\max}$ **then**
        $a_{\min} \leftarrow a_{\min} \cdot \mathrm{d}a$
    else
        $a_{\max} \leftarrow a_{\max}/\mathrm{d}a$
    end if
    report the performance of $\mathcal{M}_l[a_{\min}, a_{\max}]$ (see plots)

even worse: there is no clear intuition about the global usefulness of the learning rates, even the extreme ones.

In conclusion, our initial intuition is not clearly satisfied experimentally. In VGG16, neurons trained with an extreme learning rate cannot be pruned in general. Even if we can replace some of them by their value at initialization, we cannot simply remove them. This result tends to confirm the importance of diversity in neural networks: even roughly trained or untrained neurons are useful parts of the final neural network.

2.7 Discussion, Limitations, and Perspectives

Alrao specifically exploits redundancy between units in deep learning models, relying on the overall network approach of combining a large number of units built for diversity of behavior. Alrao would not make sense in a classical convex optimization setting. That Alrao works at all is already informative about some phenomena at play in deep neural networks.

Alrao can make lengthy SGD learning rate sweeps unnecessary on large models, such as the triple-width ResNet50 for ImageNet above. Incidentally, in our experiments, wider networks provided increased performance both for SGD and Alrao (Table 2.1 and Fig. 2.3): network size is still a limiting factor for the models used, independently of the algorithm.

Increased number of parameters for the classification layer. Since Alrao modifies the output layer of the optimized model, the number of parameters in the classification layer is multiplied by the number of classifier copies. (The number of parameters in the internal layers is unchanged.) This is a limitation for models with most parameters in the classifier layer.

On CIFAR10 (10 classes), the number of parameters increases by less than 5% for the models used. On ImageNet (1000 classes), it increases by 50–100% depending on the architecture. On Penn Treebank, the number of parameters increased by 26% in our setup (at character level); working at word level it would have increased fivefold. We provide a comparison in Section 2.9.5.

This can be mitigated by handling the copies of the classifiers on distinct computing units: in Alrao these copies work in parallel given the internal layers. Moreover, the additional output layer copies may be thrown away early in training. Finally, models with a large number of output classes usually rely on other parameterizations than a direct softmax, such as a hierarchical softmax (see references in [44]); Alrao can be used in conjunction with such methods.

Multiple output layer copies and expressiveness. Using several copies of the output layer in Alrao formally provides more expressiveness to the model, as it creates a larger architecture with more parameters. We performed two control experiments to check that Alrao's performance does not just stem from this. First, we performed ablation of the output layer copies in Alrao after one epoch, only keeping the copy with the highest model averaging weight a_i: the learning curves are identical. Second, we trained default Adam using copies of the output layer (all with the same Adam default learning rate): the learning curves are identical to Adam on the unmodified architecture. Thus, the copies of the output layer do not bring any useful added expressiveness.

Learning rate schedules, other optimizers, other hyperparameters... Learning rate schedules are often effective [6]. We did not use them here: this may partially explain why the results in Table 2.1 are worse than the state-of-the-art. One might have hoped that the diversity of learning rates in Alrao would effortlessly bring it to par with step size schedules, but the results above do not support this. Still, nothing prevents using a scheduler together with Alrao, e.g., by dividing all Alrao learning rates by a time-dependent constant.

The Alrao idea can also be used with other optimizers than SGD, such as Adam. We tested combining Alrao and Adam, and found the combination less reliable than standard Alrao: curves on the training set mostly look good, but the method quickly overfits (see Section 2.9.4).

The Alrao idea could be used on other hyperparameters as well, such as momentum. However, with more hyperparameters initialized randomly for each unit, the fraction of units having suitable values for all their hyperparameters simultaneously will quickly decrease.

Finally, we have tested our initial intuition that, with Alrao, some neurons are correctly trained and are useful, while the other tend to be ignored. This intuition is weakly confirmed by the results: in general, even the neurons that have been trained with an extreme learning rate are used by the rest of the network. Still, we show that many of them can be reinitialized without degrading the performance.

2.8 Conclusion

Applying stochastic gradient descent with multiple learning rates for different units is surprisingly resilient in our experiments, and provides performance close to SGD with an optimal learning rate, as soon as the range of random learning rates is not excessive. Alrao could save time when testing deep learning models, opening the door to more out-of-the-box uses of deep learning.

2.9 Appendix

2.9.1 Model Averaging with the Switch

As explained is Section 2.4, we use a model averaging method on the classifiers of the output layer. We could have used the Bayesian Model Averaging method [107]. But one of its main weaknesses is the *catch-up phenomenon* [104]: plain Bayesian posteriors are slow to react when the relative performance of models changes over time. Typically, for instance, some larger-dimensional models need more training data to reach good performance: at the time they become better than lower-dimensional models for predicting current data, their Bayesian posterior is so bad that they are not used right away (their posterior needs to "catch up" on their bad initial performance). This leads to very conservative model averaging methods.

The solution from [104] against the catch-up phenomenon is to *switch* between models. It is based on previous methods for prediction with expert advice (see for instance [32, 106] and the references in [50, 104]), and is well rooted in information theory. The switch method maintains a Bayesian posterior distribution, not over the set of models, but over the set of *switching strategies* between models. Intuitively, the model selected can be adapted online to the number of samples seen.

We now give a quick overview of the switch method from [104]: this is how the model averaging weights a_j are chosen in Alrao.

Assume that we have a set of prediction strategies $\mathcal{M} = \{p^j, j \in \mathcal{I}\}$. We define the set of *switch sequences*, $\mathbb{S} = \{((t_1, j_1), ..., (t_L, j_L)), 1 = t_1 < t_2 < ... < t_L \ , \ j \in \mathcal{I}\}$. Let $s = ((t_1, j_1), ..., (t_L, j_L))$ be a switch sequence. The associated prediction strategy $p_s(y_{1:n}|x_{1:n})$ uses model p^{j_i} on the time interval $[t_i; t_{i+1})$, namely

$$p_s(y_{1:i+1}|x_{1:i+1}, y_{1:i}) = p^{K_i}(y_{i+1}|x_{1:i+1}, y_{1:i}) \tag{2.4}$$

where K_i is such that $K_i = j_l$ for $t_l \leq i < t_{l+1}$. We fix a prior distribution π over switching sequences. In this work, $\mathcal{I} = \{1, ..., N_C\}$ the prior is, for a switch sequence $s = ((t_1, j_1), ..., (t_L, j_L))$:

$$\pi(s) = \pi_L(L)\pi_K(j_1) \prod_{i=2}^{L} \pi_T(t_i|t_i > t_{i-1})\pi_K(j_i) \tag{2.5}$$

with $\pi_L(L) = \frac{\theta^L}{1-\theta}$ a geometric distribution over the switch sequences lengths, $\pi_K(j) = \frac{1}{N_C}$ the uniform distribution over the models (here the classifiers) and $\pi_T(t) = \frac{1}{t(t+1)}$.

This defines a Bayesian mixture distribution:

$$p_{sw}(y_{1:T}|x_{1:T}) = \sum_{s\in\mathbb{S}} \pi(s) p_s(y_{1:T}|x_{1:T}). \tag{2.6}$$

Then, the model averaging weight a_j for the classifier j after seeing T samples is the posterior of the switch distribution: $\pi(K_{T+1} = j|y_{1:T}, x_{1:T})$.

$$a_j = p_{sw}(K_{T+1} = j|y_{1:T}, x_{1:T}) \tag{2.7}$$

$$= \frac{p_{sw}(y_{1:T}, K_{T+1} = j|x_{1:T})}{p_{sw}(y_{1:T}|x_{1:T})}. \tag{2.8}$$

These weights can be computed online exactly in a quick and simple way [104], thanks to dynamic programming methods from hidden Markov models.

The implementation of the switch used in Alrao exactly follows the pseudo-code from [105], with hyperparameter $\theta = 0.999$ (allowing for many switches a priori). It can be found in the accompanying online code.

2.9.2 Influence of Model Averaging in Alrao

We investigated the importance of the model averaging method in Alrao.

Evolution of the model averaging weights. In Figure 2.7a, we represent the evolution of the model averaging weights a_j during training of GoogLeNet with Alrao on CIFAR10. We can make several observations. First, after only a few gradient descent steps, the model averaging weights corresponding to the three classifiers with the largest learning rates go to practically zero. This means that their parameters are moving too fast, and their loss is getting very large. Next, for a short time, a classifier with a moderately large learning rate gets the largest posterior weight, presumably because it is the first to learn a useful model. Finally, after the model has seen approximately 4,000 samples, a classifier with a slightly smaller learning rate is assigned a posterior weight a_j close to 1, while all the others go to 0. Thus, after a number of gradient steps, the model averaging method acts like a model selection method.

Model selection instead of model averaging. This evolution of the model averaging weights suggests that the averaging in the last layer is acting as a model selection procedure. Once the weight a_j of some classifier j is close to 1, the output of the full Alrao architecture with model averaging is close to the output of the original architecture with a single classifier with weights θ_j^{out} from this classifier. We compared Alrao with a modified version of Alrao in which after 1 epoch, the classifier with largest model averaging weight is selected, and the other classifiers are dropped (Fig. 2.7b). The behaviors of these two variants are exactly the same.

Adam with the Alrao output layer. In order to control the effect of the increased expressiveness induced by the expanded output layer in the Alrao architecture, we ran Adam (with its default parameters) on GoogLeNet modified as in Alrao (namely, with model averaging over 10 classifiers) (Fig. 2.7c). The learning behavior is exactly the same as Adam on the original architecture. Thus, changing the architecture by replacing the single classifier layer with an average of classifiers does not by itself improve training performance.

2.9.3 Additional Experimental Details and Results

In the case of CIFAR-10 and ImageNet, we normalize each input channel x_i ($1 \leq i \leq 3$), using its mean and its standard deviation over the training set. Let μ_i and σ_i be respectively the mean and the standard deviation of the i-th channel. Then each input (x_1, x_2, x_3) is transformed into $(\frac{x_1-\mu_1}{\sigma_1}, \frac{x_2-\mu_2}{\sigma_2}, \frac{x_3-\mu_3}{\sigma_3})$. This operation is done over all the data (training, validation and test).

Moreover, we use data augmentation: every time an image of the training set is sent as input of the NN, this image is randomly cropped and randomly flipped horizontally. Cropping consists in filling with black a band at the top, bottom, left and right of the image. The size of this band is randomly chosen between 0 and 4 in our experiments.

The batch size is: 32 on CIFAR10 for every architecture, 20 on PTB, and 256 on ImageNet for Alexnet and ResNet50, and 128 for Densenet121.

On Reinforcement Learning environments, we use vanilla Q-learning [79] with a soft target update as in [62] $\tau = 0.9$, and a memory buffer of size 1,000,000. The architecture for the Q network is a MLP with 2 hidden layers. The learning curves are in Fig. 2.8d. For the optimization, the switch is used with 10 output layers. An output layer is a linear layer. Since the switch is a probability model averaging method, we consider each output layer as a probabilistic model, defined as a Normal distribution with variance 1 and mean the predicted value by the output layer. The loss for the Alrao model is the negative log-likelihood of the model mixture.

2.9.4 Alrao with Adam

In Figure 2.9, we report our experiments with Alrao-Adam on CIFAR10. As explained in Section 2.7, Alrao is much less reliable with Adam than with SGD.

This is especially true for the test performance, which can even diverge while training performance remains either good or acceptable (Fig. 2.9). Thus Alrao-Adam seems to send the model into atypical regions of the search space.

We have no definitive explanation for this at present. It might be that changing Adam's learning rate requires changing its momentum parameters accordingly. It might be that Alrao does not work on Adam because Adam is more sensitive to its hyperparameters.

2.9.5 Number of Parameters

As explained in Section 2.7, Alrao increases the number of parameters of a model, due to output layer copies. The additional number of parameters is approximately equal to $(N_{\text{out}} - 1) \times K \times d$ where N_{out} is the number of classifier copies used in Alrao, d is the dimension of the input to the output layer, and K is the number of classes in the classification task (assuming a standard softmax output; classification with many classes often uses other kinds of output parameterization instead).

Table 2.2 – Comparison between the number of parameters in models used without and with Alrao. LSTM (C) is a simple LSTM cell used for character prediction while LSTM (W) is the same cell used for word prediction.

Model	Number of parameters	
	Without Alrao	With Alrao
GoogLeNet	6.166M	6.258M
VGG	20.041M	20.087M
MobileNet	2.297M	2.412M
LSTM (C)	0.172M	0.217M
LSTM (W)	2.171M	11.261M

The number of parameters for the models used, with and without Alrao, are in Table 2.2. Using Alrao for classification tasks with many classes, such as word prediction (10,000 classes on PTB), increases the number of parameters noticeably.

For those model with significant parameter increase, the various classifier copies may be done on parallel GPUs.

2.9.6 Frozen Features Do Not Hurt Training

As explained in the introduction, several works support the idea that not all units are useful when learning a deep learning model. Additional results supporting this hypothesis are presented in Figure 2.5. We trained a GoogLeNet architecture on CIFAR10 with standard SGD with learning rate η_0, but learned only a random fraction p of the features (chosen at startup), and kept the others at their initial value. This is equivalent to sampling each learning rate η from the probability distribution $P(\eta = \eta_0) = p$ and $P(\eta = 0) = 1 - p$.

We observe that even with a fraction of the weights not being learned, the model's performance is close to its performance when fully trained.

When training a model with Alrao, many features might not learn at all, due to too small learning rates. But Alrao is still able to reach good results. This could be explained by the resilience of neural networks to partial training.

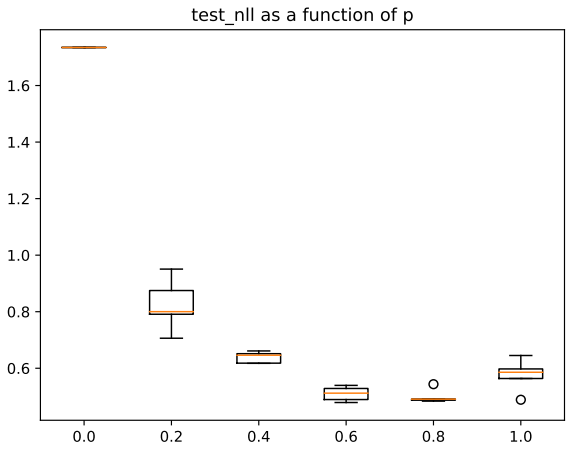


Figure 2.5 – Loss of a model where only a random fraction p of the features are trained, and the others left at their initial value, as a function of p. The architecture is GoogLeNet, trained on CIFAR10.

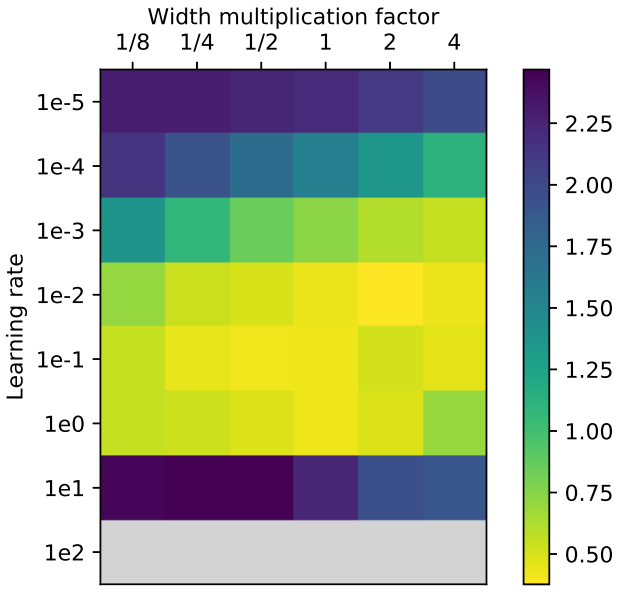
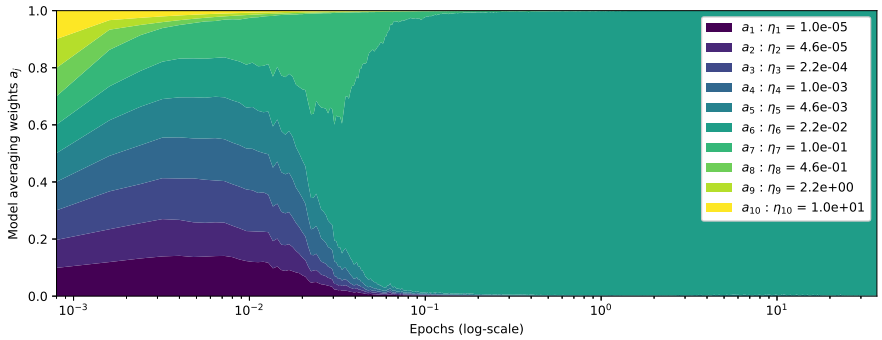
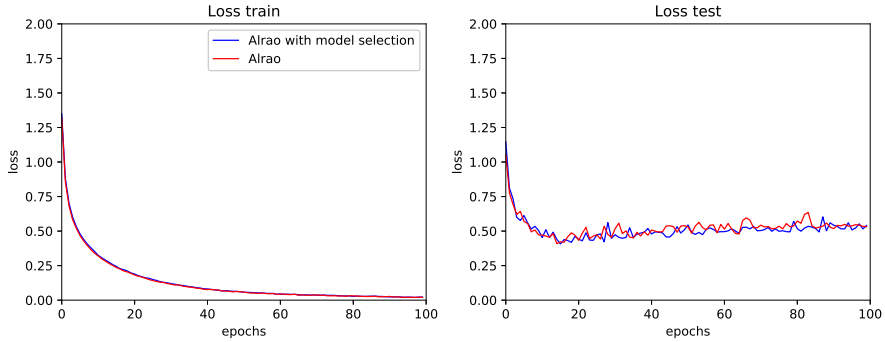


Figure 2.6 – Effect of width with SGD on GoogLeNet trained on CIFAR10, after 15 epochs (average over three runs). Grey means numerical divergence (NaN).

(a) Model averaging weights during training of GoogLeNet with Alrao on CIFAR10 with 10 classifiers. We represent the weights a_j, depending on the corresponding classifier's learning rate.



(b) Training with standard Alrao and with Alrao with model selection. With model selection, after one epoch the best classifier is selected according to the model averaging weights, so that the architecture reverts to the original architecture without model averaging. The results are averaged on three runs.



(c) Training with Adam on GoogLeNet, and on the same architecture modified as in Alrao (with 10 output layer copies). Every classifier copy uses the same default Adam learning rate. The overall output of the model is, as in Alrao, a switch model averaging over the classifier. The results are averaged on three runs.

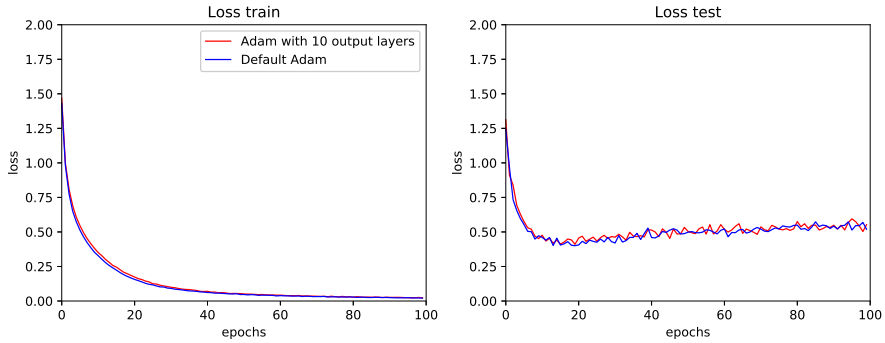
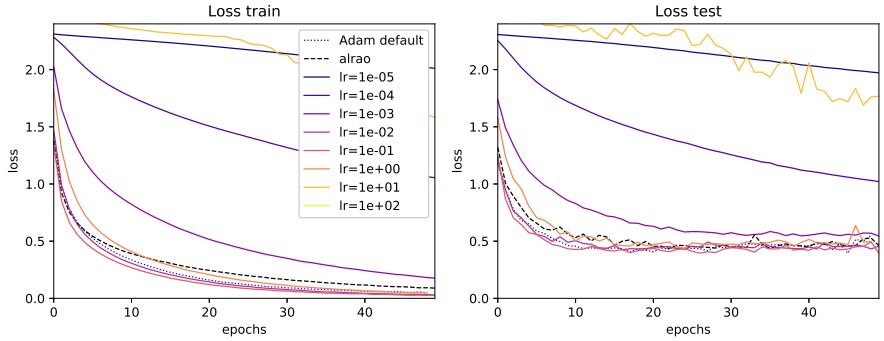
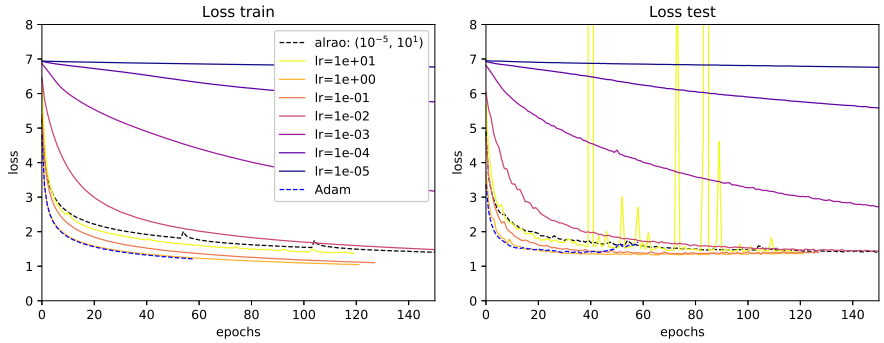


Figure 2.7 – Experiments on the effect of the model averaging layer in Alrao. In all experiments, GoogLeNet is trained on CIFAR10.

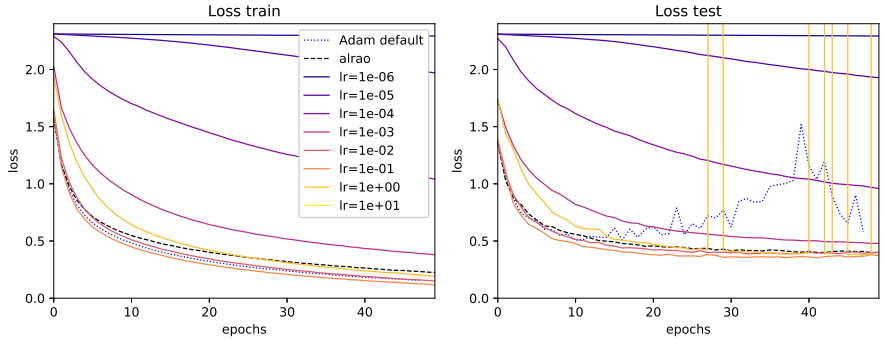
(a) GoogLeNet on CIFAR10 (Average on three runs)



(b) Densenet121 trained on ImageNet



(c) MobileNetV2 on Cifar10 (average over 3 runs)



(d) Reinforcement Learning in the pendulum environment

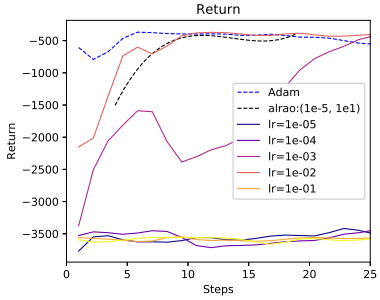
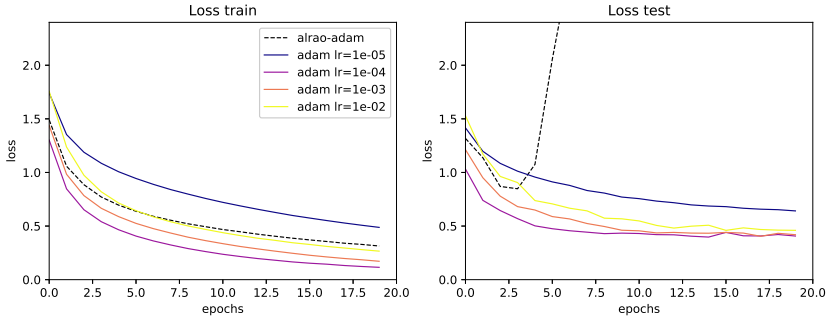
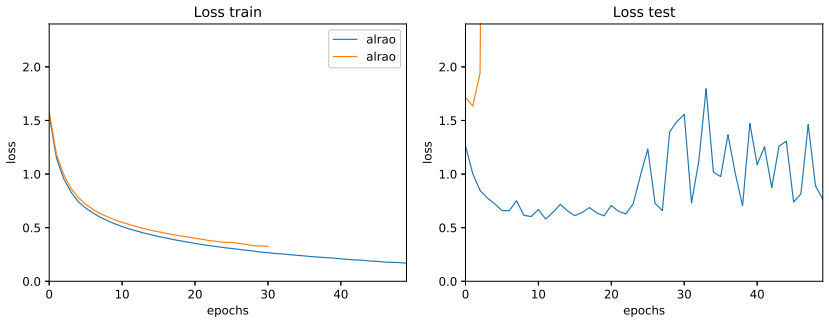


Figure 2.8 – Additional learning curves for SGD with various learning rates, Alrao, and Adam with its default setting. Left: training loss; right: test loss.

(a) Alrao-Adam with GoogLeNet on CIFAR10: Alrao-Adam compared with standard Adam with various learning rates. Alrao uses 10 classifiers and learning rates in the interval $(10^{-6}; 1)$. Each plot is averaged on 10 experiments. We observe that optimization with Alrao-Adam is efficient, since train loss is comparable to the usual Adam methods. But the model starkly overfits, as the test loss diverges.



(b) Alrao-Adam with MobileNet on CIFAR10: Alrao-Adam with two different learning rate intervals, $(10^{-6}; 10^{-2})$ for the first one, $(10^{-6}; 10^{-1})$ for the second one, with 10 classifiers each. The first one is with $\eta_{\min} = 10^{-6}$. Each plot is averaged on 10 experiments. Exactly as with GoogLeNet model, optimization itself is efficient (for both intervals). For the interval with the smallest $\eta_{\max}$, the test loss does not converge and is very unstable. For the interval with the largest $\eta_{\max}$, the test loss diverges.



(c) Alrao-Adam with VGG19 on CIFAR10: Alrao-Adam on the interval $(10^{-6}, 1)$, with 10 classifiers. The 10 plots are 10 runs of the same experiments. While 9 of them do converge and generalize, the last one exhibits wide oscillations, both in train and test.

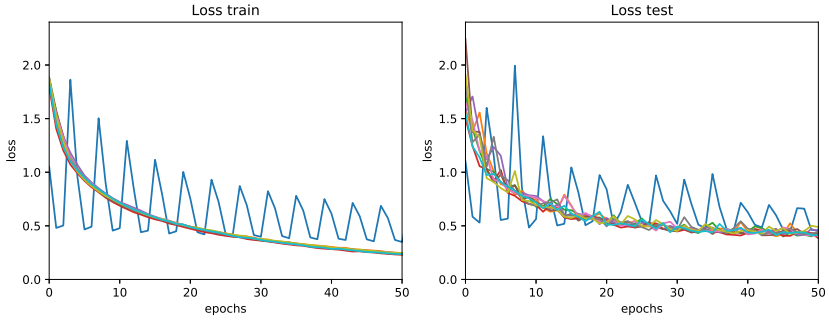


Figure 2.9 – Alrao-Adam: Experiments with the VGG19, GoogLeNet and MobileNet networks on CIFAR10.

Chapter 3

Asymmetrical Scaling Layers for Stable Network Pruning

3.1 Introduction

Neural network architectures are becoming bigger and bigger. From hand-tuned (VGG [94], GoogLeNet [99], ResNets [31]) to automatically generated architectures (NAS [113], hypernetworks [9]), 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 $S = \mathrm{Diag}(\sigma_1, \cdots, \sigma_{n_{\text{in}}})$. 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 [16]. 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 3.3) 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 [22], 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 3.4 we define ScaLP, a pruning algorithm based on scaling layers.

ScaLa and ScaLP are tested in Section 3.5. 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.

3.2 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 [3], decrease the rank of large matrices in linear layers [14], share or quantize the weights inside matrices [11, 23]

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 [57, 29]. 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 [27], 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 [89]: group Lasso itself [89], sparse-group Lasso [1] or $\ell_{\infty,1}$-norm [81]. 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 [60].

Another type of approach recently proposed [65] 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 [38]. 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 3.3). Interest for the latter has recently increased, continuing the seminal work of Neal [82], who pointed out a connection with Gaussian processes. In [78], infinitely wide deep neural networks are proven to behave like Gaussian processes, while [58] and [41] 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 3.3, 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.

3.3 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 3.4).

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?

3.3.1 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 [22], 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 1 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.

3.3.2 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:

$$\mathbf{w} = S\tilde{\mathbf{w}}, \tag{3.1}$$

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 [22], 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:

$$\begin{cases} \lim_{N\to\infty} \mathrm{Var}(y_{(N)}) & < \infty \\ \lim_{N\to\infty} \mathrm{Var}(y'_{(N)} - y_{(N)}) & < \infty \end{cases}$$
$$\Leftrightarrow \begin{cases} \lim_{N\to\infty} \sum_{k=1}^N \sigma^2_{(N)k} & < \infty \\ \lim_{N\to\infty} \sum_{k=1}^N \sigma^4_{(N)k} & < \infty \\ \lim_{N\to\infty} \tau_{(N)} & < \infty \end{cases}. \tag{3.2}$$

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

Remark 1 (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 (3.2):

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

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. 3.3a, 3.3b).

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 3.4). This can be expressed as follows.

Corollary 2. *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:*

$$\sum_{k=1}^{\infty} \sigma_k^2 < \infty \Leftrightarrow \begin{cases} \lim_{N\to\infty} \mathrm{Var}(y_{(N)}) & < \infty \\ \lim_{N\to\infty} \mathrm{Var}(y'_{(N)} - y_{(N)}) & < \infty \end{cases} . \tag{3.3}$$

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.

3.3.3 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 (3.10)):

$$\sum_{k=1}^{N} \sigma_k^2 + \tau_b^2 = 2. \tag{3.4}$$

This model is represented in Figure 3.1, where S is the scaling layer.

Remark 2. *If we substitute* $\mathbf{w}$ *for* $\tilde{\mathbf{w}}$*, as proposed in equation* (3.1)*, then one can write:*

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

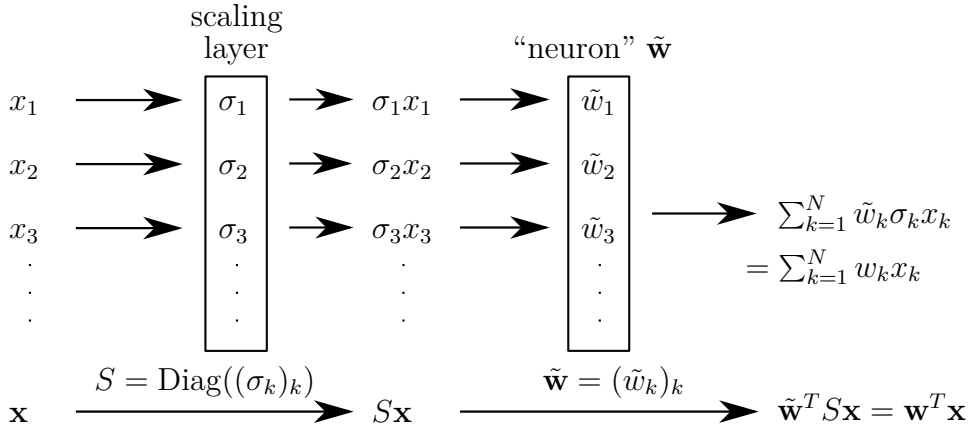


Figure 3.1 – The new model for the simple neuron.

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:

$$c^2 \cdot \sum_{k=1}^{N} \sigma_k^2 + \tau_b^2 = 2. \tag{3.5}$$

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

3.3.4 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.

3.4 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.

3.4.1 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 3. *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 3:

$$\text{pen}(\mathbf{w}) = \sum_{l=1}^{L} \sum_{k=1}^{n_l} \text{pen}(\mathbf{w}_k^l) = \sum_{l=0}^{L-1} \sum_{k=1}^{n_l} \frac{m(\mathbf{w}_k^l)^2}{\sigma_{l+1,k}^2}. \tag{3.6}$$

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 4. *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:*

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

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 3. *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 (3.6)). The following proposition indicates how to permute the neurons to minimize the penalty.

Proposition 5. *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 [89], 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 (3.6). 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 3.4.2.

3.4.2 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 [60, 89, 65], 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 $\mathbf{w}_k^l$:

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

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

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

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 3.2:

1. perform a reordering in layer l, as described in section 3.4.1;
2. establish the list of neurons to prune in layer l. We prune every neuron verifying:

$$\bar{m}(\mathbf{w}_k^l) < \epsilon \; ; \tag{3.7}$$

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 4, 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).

3.4.3 Choice of $(\sigma_k)_k$

As shown in equation (3.6), 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 (3.4) in the case of a fully connected layer, or equation (3.5) in the case of a convolutional layer. In both cases, the condition can be written:

$$\sum_{k=1}^{N} \sigma_k^2 = s^2, \tag{3.8}$$

where s is a constant depending only on the activation function [30].

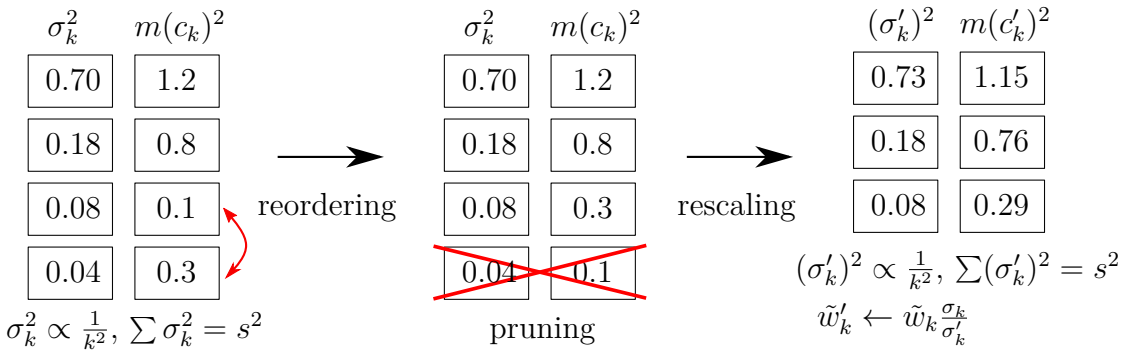


Figure 3.2 – Steps of the pruning phase for one layer: 1) the neurons are reordered by descending order of norm; 2) the neurons with a norm lower than a fixed threshold (e.g. $m(\mathbf{w})^2 \leq \epsilon = 0.2$) are pruned; 3) in the next layer, the scaling layer is recomputed to fit equation (3.8), and its weights are modified so as not to alter its behavior.

Algorithm 3 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 $+ 1$

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 (3.8). Note that such a choice of $(\sigma_k)_k$ satisfies Corollary 2 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:

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

3.5 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 3.3, 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.

3.5.1 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 $S = \mathrm{Diag}(\sigma_1, \sigma_2, \cdots, \sigma_N)$, defined by:

$$\begin{array}{rll} \text{Uniform:} & \sigma_k = \frac{1}{\sqrt{N}} & \\ \frac{1}{\sqrt{k}\log(k)}\text{:} & \sigma_k = C\frac{1}{\sqrt{k+1}\log(k+1)} & \text{with } C \text{ s.t. } \sum_{k=1}^{N} \sigma_k^2 = 1 \\ \frac{1}{k}\text{:} & \sigma_k^2 = C'\frac{1}{k} & \text{with } C' \text{ s.t. } \sum_{k=1}^{N} \sigma_k^2 = 1 \end{array} \tag{3.9}$$

- 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 3.3 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.

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. However, this stability is not always costless (see Figure 3.3a): in the

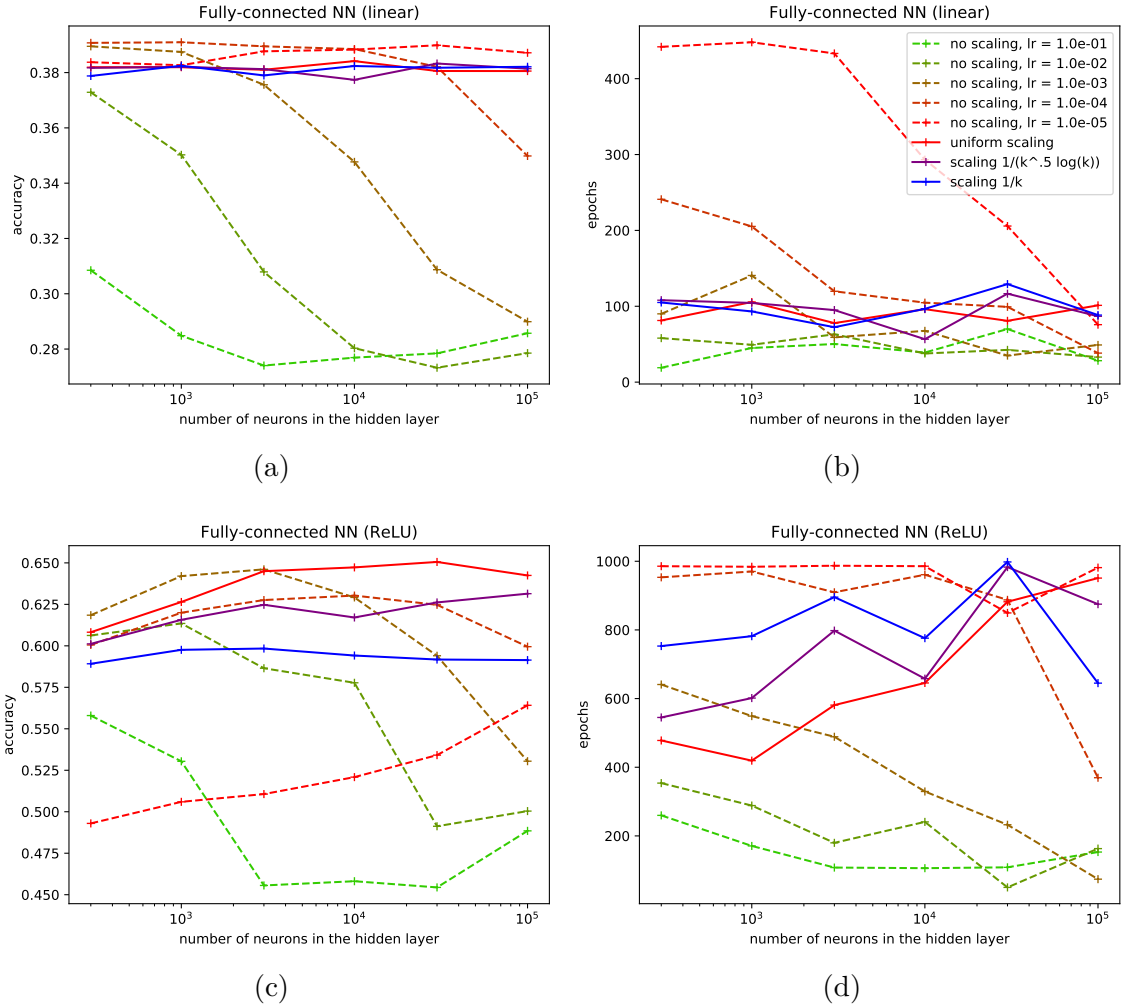


Figure 3.3 – Results of the training of neural networks with one hidden layer of size $N \in [3 \cdot 10^2, 10^5]$ on CIFAR-10, with two setups of activations functions: identity (top) and ReLU (bottom). Figures (3.3a) and (**??**) show their best accuracies on the validation set. Figures (3.3b) and (**??**) 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 any improvement has been made during 50 epochs.

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 very simple setup (training a linear 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.

3.5.2 Pruning: Results and Comparison

In this section, we compare our algorithm with other pruning techniques. As pointed out in [66], pruning techniques can be split into two categories. Some, as in [60], need a predefined pruning rate (and consequently, predefined resulting architectures), while the other ones discover automatically the final architecture, as in [65]. 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.

3.5.3 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 [101] 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 [101].

Group-Lasso penalty. The group-Lasso penalty [110] 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 [21] 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 [89, 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, [65] proposes to introduce learned scaling parameters γ, penalized by their 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.

3.5.4 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 3.4.2) with different scalings (3.9) and different penalties. The tested penalties are: $\mathcal{L}^2$, Lasso and group-Lasso (denoted by *GLasso*). We tested the penalty proposed in [65], 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. 3.4), 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. 3.5).

In Figure 3.4, the 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.) Thus, ScaLP with group-Lasso and uniform scaling provides the top performance in Figure 3.4, 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.

Pruning without scaling layers, either with Lasso or group-Lasso penalties, leads to poor performance compared to ScaLP and BN-Lasso (Fig. 3.4 and Fig. 3.5). 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

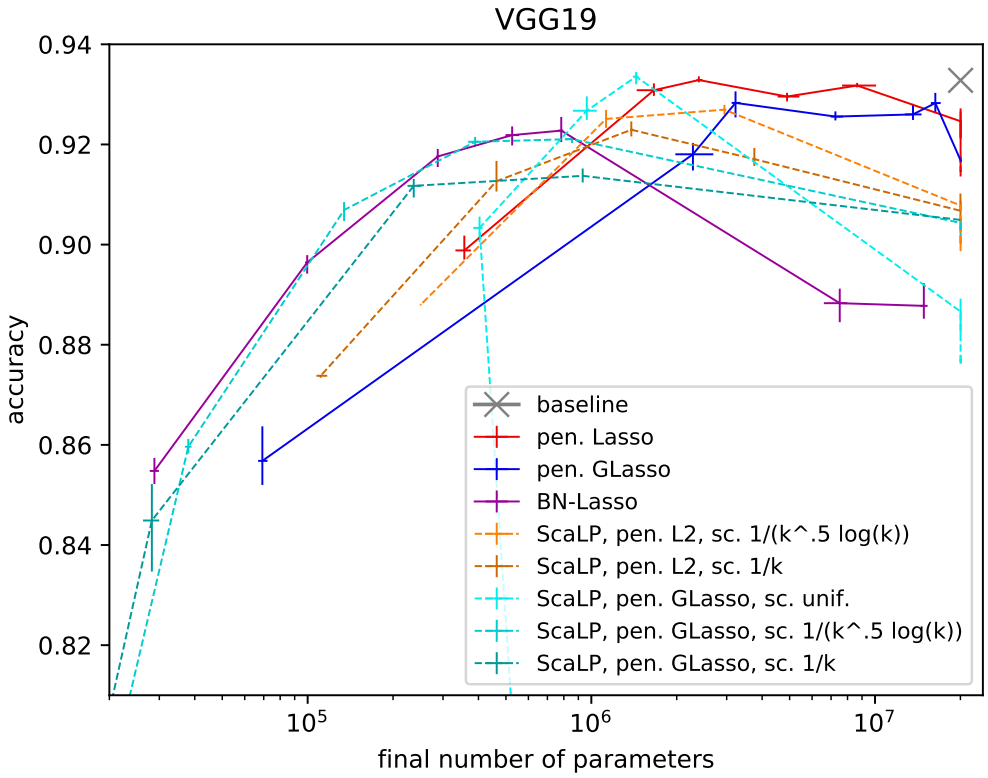


Figure 3.4 – Comparison between ScaLP, BN-Lasso, and standard pruning setups without scaling: Lasso penalty (red) and group-Lasso penalty (blue). Performance of a VGG19 network trained on CIFAR-10. Each setup (pruning method + penalty + scaling) is represented by a line, and each point of a line corresponds 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.

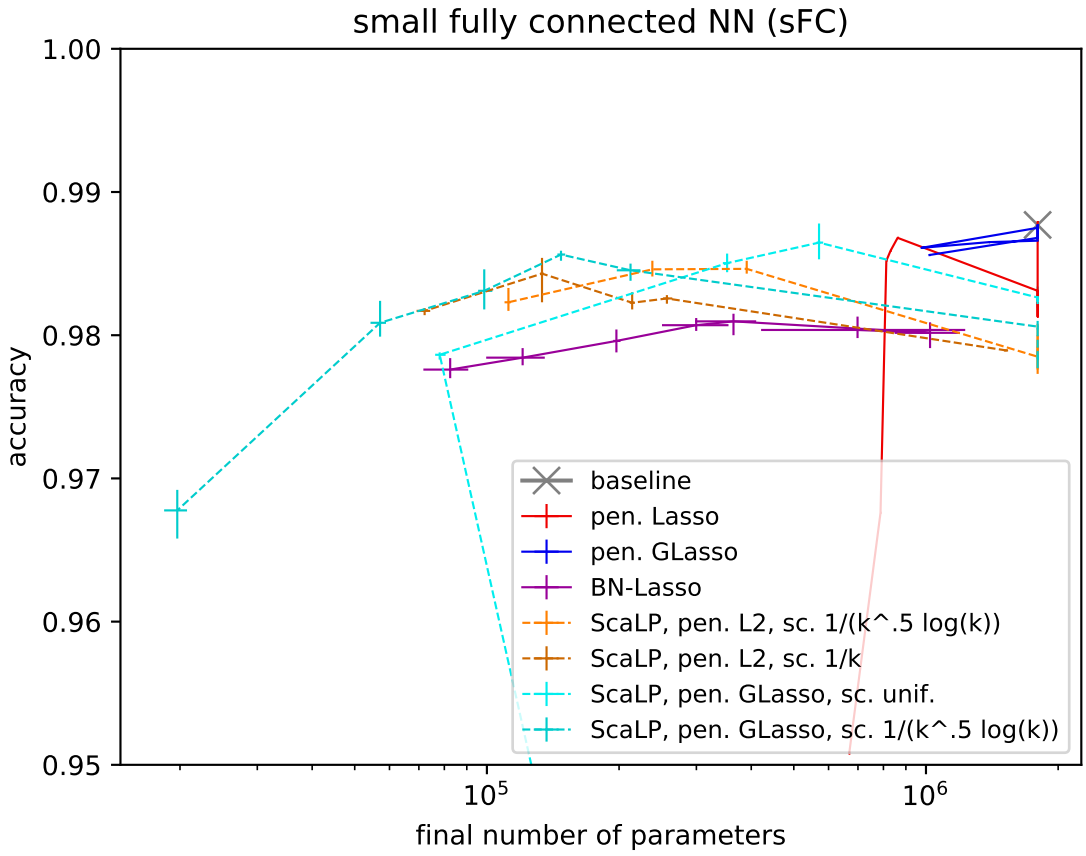


Figure 3.5 – Final accuracy according to final number of parameters for different setups, for a fully connected network trained on MNIST.

always below the Pareto front.

Table 3.1 – 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	$932K \pm 4$	-	-

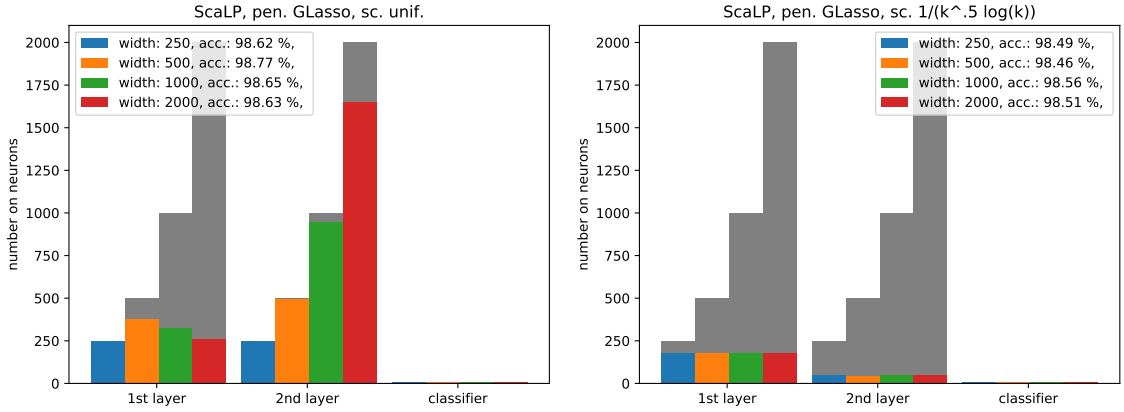
3.5.5 Stability of the Final Architecture with Respect to Initial Width

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 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 3.6: each colored bar is the average final number of neurons, computed over 3 runs.

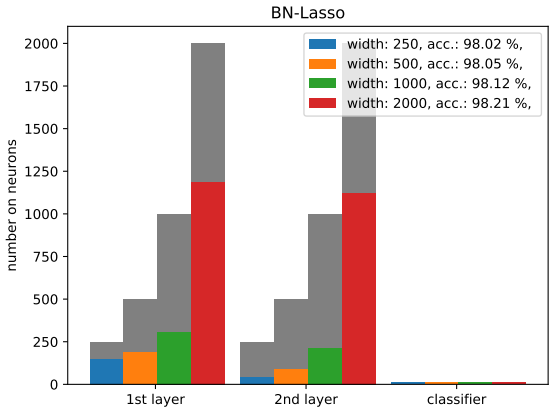
ScaLP with $1/(k^{1/2} \log k)$ scaling leads to the same network architecture, no matter the initial width. On the contrary, the other two methods return different architectures, depending on the initial width N.

While Figure 3.6 reports the average over three runs of the final architecture width, Figure 3.7 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 of 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)

(b)



(c)

Figure 3.6 – 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)

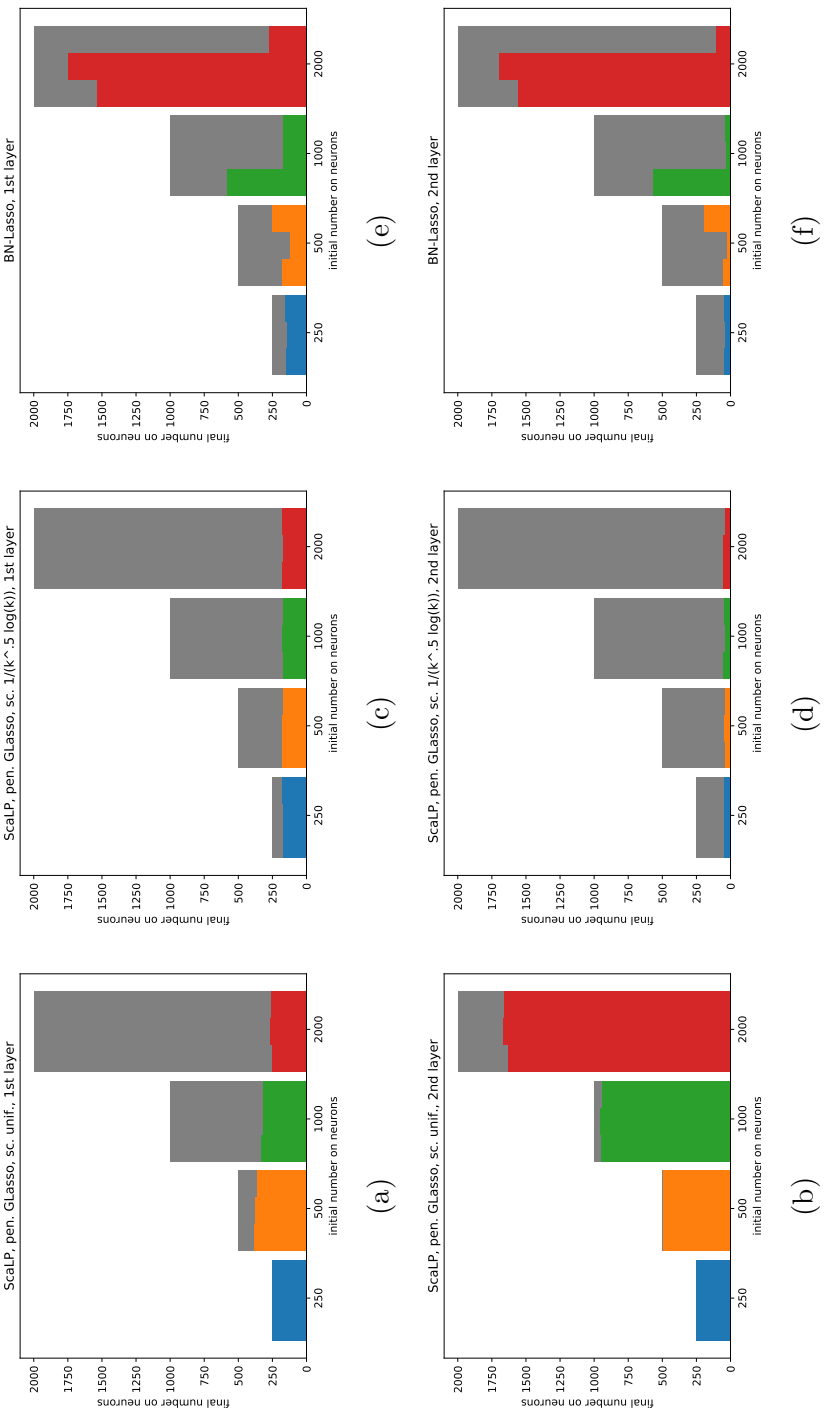


Figure 3.7 – Variability of the final number of neurons. Each column corresponds to a pruning setup. For each setup, four initial layer widths were tested (one per color), and three experiments per initial layer width were run. In one plot, each colored bar shows the number of neurons at the end of one run. Figures 3.7a and 3.7b: ScaLP with uniform scaling and group-Lasso penalty. Figures 3.7c and 3.7d: ScaLP with scaling $1/(\sqrt{k}\log k)$ and group-Lasso penalty. Figures 3.7e and 3.7f: BN-Lasso.

3.5.6 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 3.2, 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 (3.3), and we would expect it to work in general situations.

Table 3.2 – 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	$932K \pm 4$

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 3.4.1, Proposition 3), 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.

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 but all neurons in a layer 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 3.5.1), as well as for training and pruning more complex networks (Section 3.5.2 and 3.5.5). 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. 3.3a), 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.

3.6 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 is still heavily network-width-dependent, 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 both provides theoretical understanding and could reduce reliance on grid search for learning rates.

Using ScaLa together with non-uniform scalings and penalties leads to the pruning method ScaLP. ScaLP 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 this is still adjustable via a regularization constant), and also more regular between runs. Neither is the case with other pruning methods.

These 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. One difference is that our non-uniform weights retain the individuality of neurons in the infinite-width limit. These theoretical considerations deserve further exploration.

3.7 Appendix

3.7.1 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 $\tau^2_{(N)}$ ($\tau^2_{(N)} = 1$ 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 $\left(\sum_{k=1}^N \sigma^2_{(N)k}\right)_N$ is weakly monotonic and does not admit a subse-

quence that converges to 0, the following equivalence holds:

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

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:

$$\mathrm{Var}(y_{(N)}) = \tau_{(N)}^2 \sum_{k=1}^{N} \sigma_{(N),k}^2 + \tau_b^2. \tag{3.10}$$

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:

$$\begin{aligned} &\mathrm{Var}\left(y'_{(N)} - y_{(N)}\right) \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[\sum_{k=1}^{N} \sigma^4_{(N),k} \mathbb{E}(x_k^4) + \sum_{k \neq l} \sigma^2_{(N),k} \sigma^2_l \mathrm{Var}(x_k)\mathrm{Var}(x_l) + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} \mathrm{Var}(x_k) + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[3 \sum_{k=1}^{N} \sigma^4_{(N),k} + \sum_{k=1}^{N} \sigma^2_{(N),k} \sum_{l=1, l\neq k}^{N} \sigma^2_l + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[3 \sum_{k=1}^{N} \sigma^4_{(N),k} + \sum_{k=1}^{N} \sigma^2_{(N),k} \left(\sum_{l=1}^{N} \sigma^2_l - \sigma^2_{(N),k} \right) + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[3 \sum_{k=1}^{N} \sigma^4_{(N),k} + \left(\sum_{k=1}^{N} \sigma^2_{(N),k} \right)^2 - \sum_{k=1}^{N} \sigma^4_{(N),k} + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right] \\ &= \eta^2 \mathrm{Var}\left(\frac{\partial L}{\partial y}\right) \left[2 \sum_{k=1}^{N} \sigma^4_{(N),k} + \left(\sum_{k=1}^{N} \sigma^2_{(N),k} \right)^2 + 2 \sum_{k=1}^{N} \sigma^2_{(N),k} + 1 \right]. \end{aligned} \tag{3.11}$$

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 $\left(\sum_{k=1}^{N} \sigma^2_{(N)k}\right)_N$ and $\left(\sum_{k=1}^{N} \sigma^4_{(N)k}\right)_N$. We recall equation (3.11):

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

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

Convergence of $(\tau^2_{(N)})_N$. We recall equation (3.10):

$$\mathrm{Var}\left(y_{(N)}\right) = \tau^2_{(N)} \sum_{k=1}^{N} \sigma^2_{(N)k} + \tau^2_b.$$

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

□

Chapter 4

Interpreting the Penalty as the Influence of a Bayesian Prior

4.1 Introduction

Adding a penalty term to a loss, in order to make the trained model fit some user-defined property, is very common in machine learning. For instance, penalties are used to improve generalization, prune neurons or reduce the rank of tensors of weights. Therefore, usual penalties are mostly empirical and user-defined, and integrated to the loss as follows:

$$L(\mathbf{w}) = \ell(\mathbf{w}) + r(\mathbf{w}),$$

with $\mathbf{w}$ the vector of all parameters in the network, $\ell(\mathbf{w})$ the error term and $r(\mathbf{w})$ the penalty term.

From a Bayesian point of view, optimizing such a loss L is equivalent to finding the Maximum A Posteriori (MAP) of the parameters $\mathbf{w}$ given the training data and a prior $\alpha \propto \exp(-r)$. Indeed, assuming that the loss ℓ is a log-likelihood loss, namely, $\ell(\mathbf{w}) = -\ln p_{\mathbf{w}}(\mathcal{D})$ with dataset $\mathcal{D}$, then minimizing L is equivalent to minimizing $L_{\text{MAP}}(\mathbf{w}) = -\ln p_{\mathbf{w}}(\mathcal{D}) - \ln(\alpha(\mathbf{w}))$. Thus, within the MAP framework, we can interpret the penalty term r as the influence of a prior α [70].

However, the MAP approximates the Bayesian posterior very roughly, by taking its maximum. Variational Inference (VI) provides a *variational posterior* distribution rather than a single value, hopefully representing the Bayesian posterior much better. VI looks for the best posterior approximation within a family $\beta_{\mathbf{u}}(\mathbf{w})$ of approximate posteriors over $\mathbf{w}$, parameterized by a vector $\mathbf{u}$. For instance, the weights $\mathbf{w}$ may be drawn from a Gaussian distribution with mean $\mathbf{u}$ and fixed variance. The loss to be minimized over $\mathbf{u}$ is then:

$$L_{\text{VI}}(\beta_{\mathbf{u}}) \;=\; -\mathbb{E}_{\mathbf{w}\sim\beta_{\mathbf{u}}} \ln p_{\mathbf{w}}(\mathcal{D}) \;+\; \text{KL}(\beta_{\mathbf{u}}\|\alpha) \;=\; \underbrace{\mathbb{E}_{\mathbf{w}\sim\beta_{\mathbf{u}}}\ell(\mathbf{w})}_{\text{data fit term}} + \underbrace{\text{KL}(\beta_{\mathbf{u}}\|\alpha)}_{\text{penalty term}}, \quad (4.1)$$

and is also an upper bound on the Bayesian negative log-likelihood of the data [43]. Here $p_{\mathbf{w}}(\mathcal{D})$ is the likelihood of the full dataset $\mathcal{D}$ given $\mathbf{w}$, $\ell(\mathbf{w}) = -\ln p_{\mathbf{w}}(\mathcal{D})$ is the log-likelihood loss, and α is the Bayesian prior. The posteriors $\beta_{\mathbf{u}}$ that minimize this loss will be concentrated around values of $\mathbf{w}$ that assign high probability to the data, while not diverging too much from the prior α. Thus, the KL divergence term can be seen as a penalty $r(\cdot)$ over the vector $\mathbf{u}$.

Contributions and outline. We start from the following question: given some arbitrary penalty $r(\cdot)$, does it admit such an interpretation? Does there exist a prior α such that for all $\mathbf{u}$, $r(\mathbf{u}) = \mathrm{KL}(\beta_{\mathbf{u}} \| \alpha)$ (up to an additive constant)? If so, is there a systematic way to compute such α?

First, we provide a necessary and sufficient condition (Theorem 1) over the penalty r, that ensures the existence of a prior α such that VI with prior α reproduces the penalty r. The theorem comes with an explicit formula for α. We recover the MAP case as a degenerate case (Section 4.4.3).

Thus, we are able to determine whether a penalty r makes sense in a Bayesian framework and can be interpreted as the influence of a prior. We find this to be a strong constraint on r (Section 4.4.2). Here the regularizer r operates on the variational posterior $\beta_{\mathbf{u}}$; for deterministic β this reduces to r directly acting on $\mathbf{w}$, and we recover the traditional MAP correspondence in this case (Section 4.4.3).

Second (Section 4.5), we propose a heuristic to predict a priori useful values of the penalty factor λ to be put in front of a penalty r for neural networks, potentially bypassing the usual hyperparameter search on λ. Namely, we posit that the Bayesian prior $\alpha(\mathbf{w})$ corresponding to λr should reasonably match what is known for good a priori values of weights $\mathbf{w}$ in neural networks, namely, that the variance of weights under the prior α should match the Glorot initialization procedure [22]. This usually provides a specific value of λ. Moreover, the penalty size gets automatically adjusted depending on the width of the various layers in the network.

We test this prediction for various penalties (Section 4.6), including group-Lasso [110], for which per-layer adjustment of the penalty is needed [1]. Experimentally, the predicted value of the regularization factors leads to reasonably good results without extensive hyperparameter search. Still, the optimal penalization factor is found to be systematically about 0.01 to 0.1 times our predicted value, showing that our heuristic provides a usable order of magnitude but not a perfect value, and suggesting that the Bayesian VI viewpoint may over-regularize.

4.2 Related Work

Interpretations of existing empirical deep learning methods in a Bayesian variational inference framework include, for instance, *variational drop-out* [49], a version of drop-out which fits the Bayesian framework. Further developments of variational drop-out have been made by [80] for weight pruning and by [67] for neuron pruning.

Closer to our present work, the links between a penalized loss and the Bayesian point of view have previously been mentioned by [83] with a few approximations. [70] noted the equivalence of the penalized loss $L(\mathbf{w}) = \ell(\mathbf{w}) + r(\mathbf{w})$ and the MAP loss $L_{\text{MAP}}(\mathbf{w}) = \ell(\mathbf{w}) - \ln(\alpha(\mathbf{w}))$ when $\ell(\mathbf{w})$ is the negative log-likelihood of the training dataset given the weights $\mathbf{w}$, and with a prior $\alpha(\mathbf{w}) \propto \exp(-r(\mathbf{w}))$. That is, finding the vector $\hat{\mathbf{w}}$ minimizing a loss L can be equivalent to finding the MAP estimator $\hat{\mathbf{w}}_{\text{MAP}}$ by minimizing the loss L_{MAP} with a well-chosen prior distribution α.

However, the MAP framework is not completely satisfying from a Bayesian point of view: instead of returning a distribution over the weights, which contains information about their uncertainty, it returns the most reasonable value. In order to evaluate this uncertainty, [69] proposed a second-order approximation of the Bayesian posterior. In the process, [71] also proposed a complete Bayesian framework and interpretation of neural networks. Still, this approximation of the Bayesian posterior is quite limited.

In the same period, [34] applied the Minimum Description Length (MDL) principle to neural networks. Then, [72] made the link between the MDL principle and variational inference, and [24] applied it to neural networks, allowing for variational approximations of the Bayesian posterior in a tractable way.

4.3 Variational Inference

We include here a reminder on variational inference for neural networks, following [24].

From a Bayesian viewpoint, we describe the vector of weights $\mathbf{w} \in \mathbb{R}^N$ of a neural network as a random variable. Given a dataset $\mathcal{D}$, we denote by $p_{\mathbf{w}}(\mathcal{D})$ the probability given to $\mathcal{D}$ by the network with parameter $\mathbf{w}$. For instance, with a dataset $\mathcal{D} = \{(x_1, y_1), \cdots, (x_n, y_n)\}$ of n input-output pairs and a model that outputs (log-)probabilities $p_{\mathbf{w}}(y_i|x_i)$ for the outputs, then $\ln p_{\mathbf{w}}(\mathcal{D}) = \sum_{i=1}^{n} \ln p_{\mathbf{w}}(y_i|x_i)$ is the total log-likelihood of the data given the model.

Given the dataset $\mathcal{D}$, the posterior distribution over weights $\mathbf{w}$ is:

$$\pi_{\mathcal{D}}(\mathbf{w}) = \frac{p_{\mathbf{w}}(\mathcal{D})\alpha(\mathbf{w})}{\mathbb{P}(\mathcal{D})}, \qquad \mathbb{P}(\mathcal{D}) = \int_{\mathbf{w}} \alpha(\mathbf{w}) p_{\mathbf{w}}(\mathcal{D})$$

which is analytically intractable for multi-layer nonlinear neural networks. However, the posterior π_D can be approximated by looking for probability distributions β that minimize the loss

$$L_{\text{VI}}(\beta) = -\mathbb{E}_{\mathbf{w}\sim\beta} \ln p_{\mathbf{w}}(\mathcal{D}) + \text{KL}(\beta\|\alpha), \tag{4.2}$$

where $\text{KL}(\beta\|\alpha) = \int_{\mathbb{R}^N} \ln\left(\frac{\beta(\mathbf{w})}{\alpha(\mathbf{w})}\right) \beta(\mathbf{w})\, \mathrm{d}\mathbf{w}$ is the Kullback–Leibler divergence. Indeed, one has $L_{\text{VI}}(\beta) = -\ln \mathbb{P}(\mathcal{D}) + \text{KL}(\beta\|\pi_{\mathcal{D}})$, which is minimal when $\beta = \pi_{\mathcal{D}}$.

The first term in the loss (4.2) represents the error made over the dataset $\mathcal{D}$: it is small if β is concentrated around good parameters $\mathbf{w}$. The second term can be seen as a user-defined penalty over β that keeps it from diverging too much from the prior α. Moreover, for any distribution β, the quantity $L_{\mathrm{VI}}(\beta)$ is a bound on the Bayesian log-likelihood of the data: $L_{\mathrm{VI}}(\beta) \geq -\ln \int_{\mathbf{w}} \alpha(\mathbf{w}) p_{\mathbf{w}}(\mathcal{D})$ [43].

In variational inference, a parametric family $\mathcal{B}$ of probability distributions β is fixed, and one looks for the best approximation β^* of the Bayesian posterior in $\mathcal{B}$ by minimizing $L_{\mathrm{VI}}(\beta)$ in this family: the *variational posterior*. Importantly, for some families such as Gaussians with fixed variance, the gradient of $L_{\mathrm{VI}}(\beta)$ can be computed if the gradients of $\ln p_{\mathbf{w}}(\mathcal{D})$ can be computed [24], so that $L_{\mathrm{VI}}(\beta)$ can be optimized by stochastic gradient descent. Thus, this is well-suited for models such as neural networks.

Thus, we consider a parametric family $\mathcal{B} = \{\beta_{\mathbf{u}} : \mathbf{u} \in \mathbb{R}^P\}$ with parameter $\mathbf{u}$, where each $\beta_{\mathbf{u}}$ is a probability distribution over $\mathbf{w} \in \mathbb{R}^N$. Then we learn the parameters $\mathbf{u}$ instead of the weights $\mathbf{w}$. For instance, we can choose one of the following families of variational posteriors.

Example 4.3.1. *The family of products of Gaussian distributions over* $\mathbf{w} = (w_1, \cdots, w_N)$*:*

$$\beta_{\mathbf{u}} = \beta_{(\mu_1, \sigma_1^2, \cdots, \mu_N, \sigma_N^2)} = \mathcal{N}(\mu_1, \sigma_1^2) \otimes \cdots \otimes \mathcal{N}(\mu_N, \sigma_N^2).$$

In this case, the weights of the neural network are random and independently sampled from different Gaussian distributions $\mathcal{N}(\mu_k, \sigma_k^2)$*. Instead of learning them directly, the vector of parameters* $\mathbf{u} = (\mu_1, \sigma_1^2, \cdots, \mu_N, \sigma_N^2)$ *is learned to minimize* $L_{\mathrm{VI}}(\beta_{\mathbf{u}})$*.*

Example 4.3.2. *The family of products of Dirac distributions over* $\mathbf{w} = (w_1, \cdots, w_N)$*:*

$$\beta_{\mathbf{u}} = \beta_{(\mu_1, \cdots, \mu_N)} = \delta_{\mu_1} \otimes \cdots \otimes \delta_{\mu_N}.$$

In this case, the weights are deterministic: w_k *and* μ_k *are identical for all* k*.*

4.4 Bayesian Interpretation of Penalties

4.4.1 When Can a Penalty Be Interpreted as a Prior?

In this section, we provide a necessary and sufficient condition ensuring that a penalty $r(\mathbf{u})$ over the parameters of a variational posterior can be interpreted as a Kullback–Leibler divergence with respect to a prior α; namely, that

$$\exists\, K \in \mathbb{R},\ \forall \mathbf{u}, \qquad r(\mathbf{u}) = \mathrm{KL}(\beta_{\mathbf{u}} \| \alpha) + K$$

(the constant K does not affect optimization). In the process, we give a formula expressing α as a function of $r(\cdot)$.

In a nutshell, we place ourselves in the framework of variational inference: we assume the vector of weights $\mathbf{w}$ of the probabilistic model to be a random variable, drawn from a learned distribution $\beta_{\mathbf{u}}$ parameterized by a vector $\mathbf{u}$. For instance, the vector of weights $\mathbf{w}$ can be drawn from a multivariate normal distribution $\beta_{\mu,\mathbf{\Sigma}} \sim \mathcal{N}(\mu, \mathbf{\Sigma})$.

We use some notions of distribution theory. We provide a reminder in Appendix 4.8.3.

Notation. To approximate the posterior distribution of a vector $\mathbf{w} \in \mathbb{R}^N$, we denote by $(\beta_{\mu,\boldsymbol{\nu}})_{\mu,\boldsymbol{\nu}}$ the family of variational posteriors over $\mathbf{w}$, parameterized by its mean μ and a vector of additional parameters $\boldsymbol{\nu}$. The basic example is a multivariate Gaussian distribution $\beta_{\mu,\boldsymbol{\nu}}$ parameterized by its mean $\mu \in \mathbb{R}^N$ and its covariance matrix $\boldsymbol{\nu} = \mathbf{\Sigma} \in \mathcal{M}_{N,N}(\mathbb{R})$.

We say that the family $(\beta_{\mu,\boldsymbol{\nu}})_{\mu,\boldsymbol{\nu}}$ of variational posteriors is *translation-invariant* if $\beta_{\mu,\boldsymbol{\nu}}(\theta) = \beta_{0,\boldsymbol{\nu}}(\theta - \mu)$ for all $\mu, \boldsymbol{\nu}, \theta$.

We denote by $r(\mu, \boldsymbol{\nu}) = r_{\boldsymbol{\nu}}(\mu)$ some penalty, to be applied to the distribution $\beta_{\mu,\boldsymbol{\nu}}$.

We denote by $\mathcal{F}$ the Fourier transform, given by $(\mathcal{F}\varphi)(\xi) := \int_{\mathbb{R}^N} \varphi(x) e^{-i\xi\cdot x}\,\mathrm{d}x$ for $\varphi \in \mathcal{L}^1(\mathbb{R}^N)$. This definition extends to the class of tempered distributions $\mathcal{S}'(\mathbb{R}^N)$ (see Appendix 4.8.3).

In the sequel, we always restrict ourselves to priors $\alpha \in \mathcal{T}(\mathbb{R}^N) = \{\alpha \text{ s.t. } \ln(\alpha) \in \mathcal{S}'(\mathbb{R}^N)\}$, i.e. log-tempered probability distributions, hence the condition $\ln(\alpha) \in \mathcal{S}'(\mathbb{R}^N)$ in the results below. This provides a reasonable behavior of α at infinity. Common probability distributions belong to this set. As a simple counter-example, one may take $\alpha(\theta) \propto \exp(-\exp|\theta|)$, for which $\ln(\alpha) \notin \mathcal{S}'(\mathbb{R})$.

Definition 1. *We define the following distribution over* $\mathbb{R}^N$*:*

$$A_{\boldsymbol{\nu}} \;:=\; -\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\,\mathbb{1} \;-\; \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right], \tag{4.3}$$

where $\check{\beta}_{0,\boldsymbol{\nu}}(\theta) := \beta_{0,\boldsymbol{\nu}}(-\theta)$*,* $\mathbb{1}$ *is the constant function equal to* 1*, and* $\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}}) := -\mathbb{E}_{\theta\sim\beta_{0,\boldsymbol{\nu}}}[\ln \beta_{0,\boldsymbol{\nu}}(\theta)]$ *is the entropy of* $\beta_{0,\boldsymbol{\nu}}$*. We say that*

$$r \text{ fulfills } (\star) \Leftrightarrow \begin{cases} A_{\boldsymbol{\nu}} \text{ does not depend on } \boldsymbol{\nu}\text{, i.e. } A_{\boldsymbol{\nu}} = A; \\ A \text{ is a function such that } \exp(A) \text{ integrates to } \kappa > 0. \end{cases}$$

Theorem 1. *Let* $(\beta_{\mu,\boldsymbol{\nu}})_{\mu,\boldsymbol{\nu}}$ *be a translation-invariant family of variational posteriors. Let* $r(\mu, \boldsymbol{\nu}) = r_{\boldsymbol{\nu}}(\mu)$ *be a penalty over* $\beta_{\mu,\boldsymbol{\nu}}$*.*

We assume that $\forall \boldsymbol{\nu}$*, the probability distribution* $\beta_{0,\boldsymbol{\nu}}$ *has finite entropy and lies in the Schwartz class* $\mathcal{S}(\mathbb{R}^N)$*; that* $\mathcal{F}\beta_{0,\boldsymbol{\nu}}$ *is nonzero everywhere; and that* $\mathcal{F}r_{\boldsymbol{\nu}} \in \mathcal{E}'(\mathbb{R}^N)$*, the class of distributions with compact support.*

We are looking for probability distributions α *such that:*

$$\exists\, K \in \mathbb{R} : \forall(\mu, \boldsymbol{\nu}), \qquad r_{\boldsymbol{\nu}}(\mu) = \mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}} \| \alpha) + K. \tag{4.4}$$

We have the following equivalence:

$$\alpha \text{ is a solution to (4.4), with } \alpha \in \mathcal{T}(\mathbb{R}^N) \quad \Leftrightarrow \quad r \text{ fulfills } (\star) \text{ and } \alpha = \frac{1}{\kappa}\exp(A),$$

where A is defined in Equation (4.3).

The proof is given in Appendix 4.8.4. It is based on the resolution of a classical integral equation ([85], Section 10.3-1) adapted to the wider framework of distribution theory. This extension is necessary, since the Fourier transform of the widely-used $\mathcal{L}^2$ penalty ($r(x) = x^2$) cannot be expressed as a function.

The preceding result holds under some technical assumptions. Even if some of the technical assumptions fail, the formula is still useful to compute a candidate prior α from a penalty $r(\cdot)$: apply Equation (4.3) on a penalty r to compute $A_{\boldsymbol{\nu}}$, then check that Condition $(\star)$ holds, define $\alpha = \frac{1}{\kappa}\exp(A)$, and finally compute $\mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}}\|\alpha)$ analytically and compare it to r.

Remark 4. *The assumption $\mathcal{F}r_{\boldsymbol{\nu}} \in \mathcal{E}'(\mathbb{R}^N)$ includes the $\mathcal{L}^2$ penalty and all $\mathcal{L}^{2p}$ penalties (for any positive integer p), but not the $\mathcal{L}^1$ penalty. Indeed, For $r_2(x) = x^2$ one has $\mathcal{F}r_2 = -2\pi\delta'' \in \mathcal{E}'(\mathbb{R})$. For $r_{2p}(x) = x^{2p}$, one has $\mathcal{F}r_{2p} = (-1)^p 2\pi\delta^{(2p)} \in \mathcal{E}'(\mathbb{R})$, but for $r_1(x) = |x|$, $(\mathcal{F}r_1)(t) = 2t^{-2} \notin \mathcal{E}'(\mathbb{R})$. Still, for any penalty $r \in \mathcal{S}'(\mathbb{R}^N)$, it is always possible to find a sequence $(r_n)_n \in \mathcal{S}'(\mathbb{R}^N)$ converging to r in $\mathcal{S}'(\mathbb{R}^N)$ such that $\mathcal{F}r_n \in \mathcal{E}'(\mathbb{R}^N)$ for all n (see Appendix 4.8.5).*

Particular case: variational posteriors parameterized by their mean ($\boldsymbol{\nu} = \emptyset$). Theorem 1 provides a method to compute a prior from a penalty and a family of variational posteriors $(\beta_{\mu,\boldsymbol{\nu}})_{\mu,\boldsymbol{\nu}}$. Still, Eq. (4.3) returns a distribution $A_{\boldsymbol{\nu}}$ which may or may not satisfy the condition. Here we present a corollary which guarantees that the condition is satisfied; this holds under stricter conditions over the variational posteriors.

Corollary 6. *Assume that the family of posterior distributions $(\beta_{\mu,\boldsymbol{\nu}})_{\mu,\boldsymbol{\nu}}$ is only parameterized by their means, that is $\beta_{\mu,\boldsymbol{\nu}} = \beta_\mu$ and $r_{\boldsymbol{\nu}}(\mu) = r(\mu)$. Assume that $\beta_0 \in \mathcal{S}(\mathbb{R}^N)$, $\mathcal{F}r \in \mathcal{E}'(\mathbb{R}^N)$, $\mathcal{F}\beta_0$ is nonzero everywhere and β_0 has a finite entropy. Assume that $\mathcal{F}^{-1}[\frac{\mathcal{F}r}{\mathcal{F}\check{\beta}_0}]$ is a function satisfying*

$$\exists\,(a,b,k) \in \mathbb{R}_*^+ \times \mathbb{R}_*^+ \times \mathbb{R} : \forall \theta \in \mathbb{R}^N, \qquad \mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathcal{F}\check{\beta}_0}\right](\theta) \geq a\|\theta\|^b + k, \tag{4.5}$$

where $\|\cdot\|$ is the $\mathcal{L}^2$ norm on $\mathbb{R}^N$.

Then there exists $\kappa > 0$ such that:

$$\alpha(\theta) = \frac{1}{\kappa}\exp\left(-\mathrm{Ent}(\beta_0) - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathcal{F}\check{\beta}_0}\right](\theta)\right) = \frac{1}{\kappa}e^{A(\theta)}$$

is a probability density satisfying $r(\mu) = \mathrm{KL}(\beta_\mu\|\alpha)$ up to a constant, and is the unique such probability density in $\mathcal{T}(\mathbb{R}^N)$.

Remark 5. *In many cases, condition* (4.5) *is not a limitation. For instance, with* $\beta_\mu \sim \mathcal{N}(\mu, \sigma^2)$*, where* σ^2 *is a constant, and with the penalty* r *the* $\mathcal{L}^2$ *penalty, Condition* (4.5) *reads:* $\exists\,(a, b, k) \in \mathbb{R}_*^+ \times \mathbb{R}_*^+ \times \mathbb{R}$ *such that* $\theta^2 - \sigma^2 \geq a\|\theta\|^b + k$ *for all* $\theta \in \mathbb{R}^N$*, which is satisfied.*

4.4.2 Example 4.3.1: Gaussian Distributions with $\mathcal{L}^2$ Penalty

Let us study the variational posteriors from Example 4.3.1: each vector of weights $\mathbf{w} \in \mathbb{R}^N$ is drawn from a Gaussian distribution $\beta_{\mathbf{u}} = \mathcal{N}(\mu_1, \sigma_1^2) \otimes \cdots \otimes \mathcal{N}(\mu_N, \sigma_N^2)$ with parameter $\mathbf{u} = (\mu_1, \sigma_1^2, \cdots, \mu_N, \sigma_N^2)$. We assume that each pair of variational parameters (μ_k, σ_k) is penalized independently by some penalty $r_{a,b}$ depending on two real-valued functions a and b:

$$r_{a,b}(\mu_k, \sigma_k^2) = a(\sigma_k^2) + b(\sigma_k^2)\mu_k^2. \tag{4.6}$$

The penalty over $\mathbf{u}$ is assumed to be the sum of the penalties $r_{a,b}(\mu_k, \sigma_k)$. Therefore, we study each pair (μ_k, σ_k) independently and we omit the index k.

Corollary 7. *If the penalty* (4.6) *above corresponds to a prior* α*, then* α *is Gaussian.*

More precisely, let α *be a probability distribution in* $\mathcal{T}(\mathbb{R})$*, and assume that* $r_{a,b}(\mu, \sigma^2) = \mathrm{KL}(\beta_{\mu,\sigma^2}\|\alpha)$*, up to a constant. Then there exists* $\sigma_0^2 > 0$ *such that* $\alpha = \mathcal{N}(0, \sigma_0^2)$*. Moreover, in that case, the penalty is, up to a constant:*

$$r_{a,b}(\mu, \sigma^2) = \frac{1}{2}\left[\frac{\sigma^2 + \mu^2}{\sigma_0^2} + \ln\left(\frac{\sigma_0^2}{\sigma^2}\right) - 1\right].$$

This corollary is an application of Theorem 1. We give the proof in Appendix 4.8.7.

Thus, the assumption that a penalty r arises from a variational interpretation is a strong constraint over r. Here the penalty r was initially parameterized by a pair of real functions (a, b), and is finally parameterized by a single number σ_0^2.

4.4.3 Example 4.3.2: Deterministic Posteriors and the MAP

Another basic example is to use Dirac functions as variational posteriors (Example 4.3.2): $\beta_{\mu,\nu} = \delta_\mu$. Since $\delta_0 \notin \mathcal{S}(\mathbb{R})$, the technical conditions of Theorem 1 are not satisfied. However, it is possible to apply Formula (4.3) and check that the resulting prior α is consistent with a chosen penalty r.

Applying Formula (4.3) yields

$$A = -\mathrm{Ent}(\delta_0)\mathbb{1} - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathcal{F}\delta_0}\right] = -\mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathbb{1}}\right] = -r.$$

Thus, if $\exp(-r)$ integrates to $0 < \kappa < \infty$, then we can define $\alpha = \frac{1}{\kappa}\exp(-r)$. Then, we can check that indeed $\mathrm{KL}(\delta_\mu \| \alpha) = r(\mu)$ up to a constant:

$$\mathrm{KL}(\delta_\mu \| \alpha) = -\operatorname{Ent}(\delta_0) - \langle \delta_\mu, \ln(\alpha) \rangle = -\ln \alpha(\mu) - \operatorname{Ent}(\delta_0) = r(\mu) + \ln \kappa - \operatorname{Ent}(\delta_0),$$

which confirms that the proposed prior α is consistent with the penalty $r(\cdot)$.

Thus, this formula recovers via variational inference the well-known penalty–prior equivalence in the MAP approximation, $\alpha_{\mathrm{VI}}(\theta) \propto \exp(-r(\theta)) \propto \alpha_{\mathrm{MAP}}(\theta)$ [70].

However, this is somewhat formal: the entropy $\operatorname{Ent}(\delta_0)$ of a Dirac function is technically undefined and is an "infinite constant". In practice, though, with a finite machine precision ϵ, a Dirac mass can be defined as a uniform distribution over an interval of size ϵ, and $\operatorname{Ent}(\delta_0)$ becomes the finite constant $\ln \epsilon$.

4.5 Application to Neural Networks: Choosing the Penalty Factor

In this section, we compare the prior α arising from a penalty via Theorem 1, to reasonable weight priors for neural networks. In particular, we study how the prior varies when scaling the penalty by a factor λ, namely, $r_\lambda(\cdot) = \lambda \tilde{r}(\cdot)$ for a reference penalty $\tilde{r}$. Requiring that α is comparable to standard priors for neural network weights provides a specific value of the regularization constant λ. Specifically, we compare the prior to the standard initialization of neuron weights [22]: the variance of weights sampled from the prior α should be approximately equal to the inverse of the number of incoming weights to a neuron. This constraint can be used to determine λ. In particular, this suggests different values of λ for different layers, depending on their size.

Possible advantages of being able to predict a good value for λ include avoiding a hyperparameter search, and better adjustment of the relative penalties of different layers or groups of neurons. For instance, one application of penalties in neural networks is to push neurons or convolutional filters towards zero, allowing for network pruning and reduced computational overhead. Penalties have been developed to remove entire neurons or filters, often based on the Lasso penalty: for instance, group-Lasso [89] and sparse group-Lasso [1]. In the latter work, different penalties are used for different layers, with values of λ determined empirically. We will compare our predicted values of λ to the values used in these works.

Additive penalties and independent priors. Below, we focus on penalties that are expressed as sums over neurons or groups of neurons, such as $\mathcal{L}^2$ or group-Lasso penalties. In the Bayesian setup, this corresponds to the additivity property of Kullback–Leibler divergence over products of distributions, $\mathrm{KL}(\beta_1 \otimes \beta_2 \| \alpha_1 \otimes \alpha_2) = \mathrm{KL}(\beta_1 \| \alpha_1) + \mathrm{KL}(\beta_2 \| \alpha_2)$. Thus, sums of penalties over different neurons or groups of neurons correspond to priors α and variational posteriors β that decompose as independent distributions over these neurons or groups of neurons.

4.5.1 A Reasonable Condition over the Prior α

Let us consider the variational inference framework applied to one weight w_{lij}, the j-th weight of the i-th neuron in the l-th layer of a neural network. As a default distribution, one could expect α to be usable to initialize the weight w_{lij}.

Therefore, we require α to satisfy the condition given by [22] over the initialization procedure, which we call ($\sharp$):

$$\alpha \text{ fulfills } (\sharp) \quad \Leftrightarrow \quad \mathbb{E}_{w_{lij}\sim\alpha}[w_{lij}] = 0 \text{ and } \mathbb{E}_{w_{lij}\sim\alpha}[w_{lij}^2] = 1/P_l$$

where P_l is the number of incoming weights in one neuron of the layer l. More generally, this condition can be written for the whole set of incoming weights to each neuron: denoting by $\mathbf{w}_{li}$ the vector of all incoming weights of neuron i in layer l, one can define Condition ($\sharp'$):

$$\alpha \text{ fulfills } (\sharp') \quad \Leftrightarrow \quad \mathbb{E}_{\mathbf{w}_{li}\sim\alpha}[\mathbf{w}_{li}] = 0 \text{ and } \mathbb{E}_{\mathbf{w}_{li}\sim\alpha}[\|\mathbf{w}_{li}\|^2] = 1$$

which slightly extends ($\sharp$).

Thus, if the prior α depends on some variable b, then Condition ($\sharp$) is reflected on b. For instance, in Example 4.3.1 (Section 4.4.2), ($\sharp$) is satisfied if and only if $\alpha_{\sigma_0^2}$ is $\mathcal{N}(0, 1/P_l)$. In the end, our suggested recipe for finding reasonable values for parameters b of a penalty r_b is the following:

1. Follow the formulas in Thm. 1 to compute a prior α_b such that $r_b(\mu, \boldsymbol{\nu}) = \mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}}\|\alpha_b) + K$.
2. write Condition ($\sharp$) or ($\sharp'$) (depending on the case) for α_b: this provides a constraint on b.

4.5.2 Examples: $\mathcal{L}^2$, $\mathcal{L}^1$, and Group-Lasso Penalties

We now review some standard penalties, and apply this criterion to compute the penalty factor λ in front of each penalty.

Here we work with Dirac posterior distributions $\beta_\mu = \delta_\mu$ (Example 4.3.2 and Section 4.4.3). In that case, since each weight w_{lij} is deterministically set to μ_{lij}, we use w_{lij} and μ_{lij} indifferently. Thus, the penalty with penalty factor λ is

$$r_\lambda(w) = r_\lambda(\mu) = \lambda\tilde{r}(\mu) = \mathrm{KL}(\delta_\mu\|\alpha_\lambda) + K$$

for some reference penalty $\tilde{r}$. For each λ, the corresponding prior is $\alpha_\lambda(\theta) \propto e^{-\lambda\tilde{r}(\theta)}$.

We will apply Condition ($\sharp$) to find a value for λ, for various penalties. In Table 4.1 we compare these values to usual ones.

We recall that P_l in the number of incoming weights of a neuron in layer l.

Remark 6. *We recall that $r(\mathbf{w})$ is a full-dataset penalty (see Eq. 4.1), so that the actual per-minibatch penalty for stochastic gradient is $\frac{1}{n}r(\mathbf{w})$ for individual samples or $\frac{B}{n}r(\mathbf{w})$ for minibatches of size B.*

The results below apply equally to fully connected and to convolutional networks; in the latter case, "neuron" should read as "individual filter".

$\mathcal{L}^2$ penalty. We have $r_\lambda(w) = \lambda w^2$, thus $\alpha_\lambda(\theta) \sim \mathcal{N}(0, 1/(2\lambda))$. Then Condition ($\sharp$) is equivalent to $\lambda = \frac{P_l}{2}$. Thus, from a Bayesian viewpoint, when using a $\mathcal{L}^2$-penalty, each weight w of a neuron with P_l incoming weights should be penalized by

$$r_\lambda(w) = \frac{P_l}{2} w^2. \tag{4.7}$$

$\mathcal{L}^1$ penalty. We have $r_\lambda(w) = \lambda|w|$, thus $\alpha_\lambda(\theta) = \frac{\lambda}{2} e^{-\lambda|\theta|}$. Therefore, Condition ($\sharp$) is equivalent to $\lambda = \sqrt{2P_l}$. As a consequence, when penalizing weight w in the l-th layer of a neural network with a $\mathcal{L}^1$-penalty, this weight w should be penalized with the term:

$$r_\lambda(w) = \sqrt{2P_l}|w|. \tag{4.8}$$

Standard Group-Lasso penalty. The group-Lasso jointly penalizes all the incoming weights of each neuron $\mathbf{w}_{l,i} \in \mathbb{R}^{P_l}$. Thus, we consider a prior and a posterior that are probability distributions on $\mathbb{R}^{P_l}$, and use Condition ($\sharp'$). Denoting by $\mathbf{w} \in \mathbb{R}^{P_l}$ the incoming weights of a neuron, we have

$$r_\lambda(w) = \lambda \|\mathbf{w}\|_2. \tag{4.9}$$

Then $\alpha_\lambda(\theta) = \frac{\lambda^{P_l}}{S_{P_l-1}\Gamma(P_l)} e^{-\lambda\|\theta\|_2}$, where Γ is Euler's Gamma function and S_{n-1} is the surface of the $(n-1)$-sphere. After computation of the variance, Condition ($\sharp'$) is equivalent to $\lambda = \sqrt{P_l(P_l+1)}$. As a consequence, when penalizing neuron $\mathbf{w}$ in the l-th layer of a neural network with a group-Lasso penalty, this neuron $\mathbf{w}$ should be penalized with the term:

$$r_\lambda(\mathbf{w}) = \sqrt{P_l(P_l+1)}\|\mathbf{w}\|_2. \tag{4.10}$$

This choice differs from [1], who use $\lambda \propto \sqrt{P_l}$, after an intuition proposed in [110]. Their whole-network penalty is

$$L(\mathbf{w}) = \ell(\mathbf{w}) + \sum_{l=1}^{L} \tilde{\lambda}_l \left((1-\gamma)\sqrt{P_l} \sum_{i=1}^{n_l} \|\mathbf{w}_{l,i}\|_2 + \gamma \sum_{i=1}^{n_l} \|\mathbf{w}_{l,i}\|_1 \right), \tag{4.11}$$

where $\gamma \in [0,1]$ is a fixed constant, L is the number of layers, n_l is the number of neurons in the l-th layer, P_l is the number of parameters in each neuron in the l-th layer, and $\mathbf{w}_{l,i}$ is the set of weights of the i-th neuron of the l-th layer. The per-layer regularization constants $\tilde{\lambda}_l$ are chosen empirically in [1]. On the other hand, our Bayesian reasoning yields a scaling $\sqrt{P_l(P_l+1)}$ instead of $\sqrt{P_l}$ for the penalties $\|\mathbf{w}_{l,i}\|_2$ in (4.11).

Table 4.1 – How the regularization constant λ depends on the number of neurons n_l and the number of parameters per neuron P_l in layer l, both for our heuristics ($\lambda_{\text{Bayesian}}$) and for standard settings (λ_{usual}). For details on group-Lasso and reversed group-Lasso penalties, see Equations (4.9) and (4.12).

Penalty	$\mathcal{L}^2$	$\mathcal{L}^1$	group-Lasso	rev. gr.-Lasso
$\lambda_{\text{Bayesian}}$	$P_l/2$	$\sqrt{2P_l}$	$\sqrt{P_l(P_l+1)}$	$\sqrt{P_l(n_l+1)}$
λ_{usual}	1	1	$\sqrt{P_l}$	$\sqrt{n_l}$

Reversed Group-Lasso penalty. We recall that the standard group-Lasso penalty groups the weights of a layer by neurons, that is output features. It is also possible to group them by input features, namely, to group together the outgoing weights of each neuron. For a fully-connected neural network, the loss penalized by a "reversed" group-Lasso can be written:

$$L(\mathbf{w}) = \ell(\mathbf{w}) + \sum_{l=1}^{L} \lambda_l \sum_{j=1}^{P_l} \|\mathbf{w}_{l,\cdot,j}\|_2, \tag{4.12}$$

where $\mathbf{w}_{l,\cdot,j} \in \mathbb{R}^{n_l}$ is the vector of weights of the l-th layer linked to the j-th input feature and n_l is the number of neurons in the l-th layer. By computations similar to standard group-Lasso, ($\sharp'$) is equivalent to $\lambda = \sqrt{P_l(n_l+1)}$ where n_l is the number of neurons in layer l, namely,

$$r_\lambda(\mathbf{w}_{l,\cdot,j}) = \sqrt{P_l(n_l+1)}\|\mathbf{w}_{l,\cdot,j}\|_2.$$

4.6 Experiments

Penalty terms are often used in the literature [89, 1, 27] in order to prune neural networks, that is, to make them sparse. Indeed, penalties such as $\mathcal{L}^1$ or group-Lasso tend to push weights or entire groups of weights towards 0 [102]. The efficiency of such methods can be measured in terms of final accuracy and number of remaining parameters.

We have chosen to compare within this context our heuristic to usual setups for the penalty factor, for different penalties. Our main criterion remains the final accuracy of the pruned network. According to the results given in Table 4.1, the global penalty should be decomposed into a sum of penalties over each layer:

$$r(\mathbf{w}) = \lambda \sum_{l=1}^{L} \lambda_l r(\mathbf{w}_l),$$

where λ is a global penalty factor, L is the number of layers, $r(\mathbf{w}_l)$ is the penalty term due to the l-th layer and λ_l is its corresponding penalty factor. According to our heuristic, each λ_l should be set to $\lambda_{\text{Bayesian}}$ given in Table 4.1 and $\lambda = \lambda_{\text{Th}} = 1/n$, where n is the size of the training dataset (by Remark 6). In the usual setup, each λ_l is set to λ_{usual} given in Table 4.1, while several values for λ are tested.

We test the quality of our heuristic in two steps. First, we only test the computed partial penalty factors λ_l: they are fixed to $\lambda_{\text{Bayesian}}$, while several values for λ are tested. Second, we test the full heuristic: each λ_l is fixed to its corresponding $\lambda_{\text{Bayesian}}$ and $\lambda = \lambda_{\text{Th}} = 1/n$.

Experimental setup. We consider two neural networks: a version of VGG19 [94] with one fully connected layer of size 512, and CVNN, which is a simple network with two convolutional layers (of respective sizes 100 and 200 with 5×5 patches), each followed by a ReLU and a 2×2 max-pool layer, and two fully-connected layers (of sizes 1000 and 200).

The training and pruning procedure is detailed in Appendix 4.8.1 and the full experimental procedure is detailed in Appendix 4.8.2. The CIFAR-10 dataset is decomposed into three parts: a training set (42000 images), a validation set (8000 images), and a test set (10000 images). All reported accuracies are computed over the test set, which is never used during each run.

Results. We give the results in Figure 4.1. Each point gives the final accuracy and number of parameters of a neural network for a given setup, averaged over 3 runs. The red line and the blue line correspond respectively to the usual setup and our setup for per-layer penalty λ_l, for various global factors λ. To check the quality of the heuristics over λ_l, we should compare the maxima of the two lines in each graph. The theoretical value $\lambda = \lambda_{\text{Th}}$ is marked by the grey vertical line, so the blue point on the grey vertical line illustrates the performance of the joint theoretical values on both λ_l and λ.

We report in Table 4.2 the values of relevant quantities: $\text{acc}^*_{\text{usual}}$, the best (over λ) accuracy of the usual per-layer scaling λ_l; $\text{acc}^*_{\text{Bayesian}}$, the best (over λ) accuracy of the Bayesian per-layer scaling λ_l (we denote by λ^* the optimal λ); and $\text{acc}_{\text{Bayesian}}$, the accuracy of the Bayesian per-layer λ_l together with the Bayesian predicted $\lambda = \lambda_{\text{Th}}$. This would allow us to conclude:

1. if $\text{acc}^*_{\text{Bayesian}} > \text{acc}^*_{\text{usual}}$, then we can conclude that our theoretical estimation for λ_l is better that the usual ones;

2. moreover, we check whether $\text{acc}_{\text{Bayesian}} > \text{acc}^*_{\text{usual}}$. If it is, then we can conclude that our theoretical estimation for λ_l *and* the choice $\lambda = \lambda_{\text{Th}}$ are better than the usual ones;

3. finally, the closer the ratio $\lambda_{\text{Th}}/\lambda^*$ is to 1, the closer we are to the Bayesian theoretical framework.

Regarding Point 1, our theoretical estimation for λ_l leads to accuracies which remain close to accuracies obtained in the usual setup. Our setup leads to slightly better accuracies when training CVNN, and slightly worse for VGG19.

Regarding Points 2 and 3, the usual setup with optimized λ and our setup with Bayesian estimation for $\lambda = \lambda_{\text{Th}}$ perform similarly, though the second one usually performs slightly worse, due to a systematic mismatch between λ_{Th} and λ^*. Indeed, λ_{Th} is always roughly 10 times greater than λ^*.

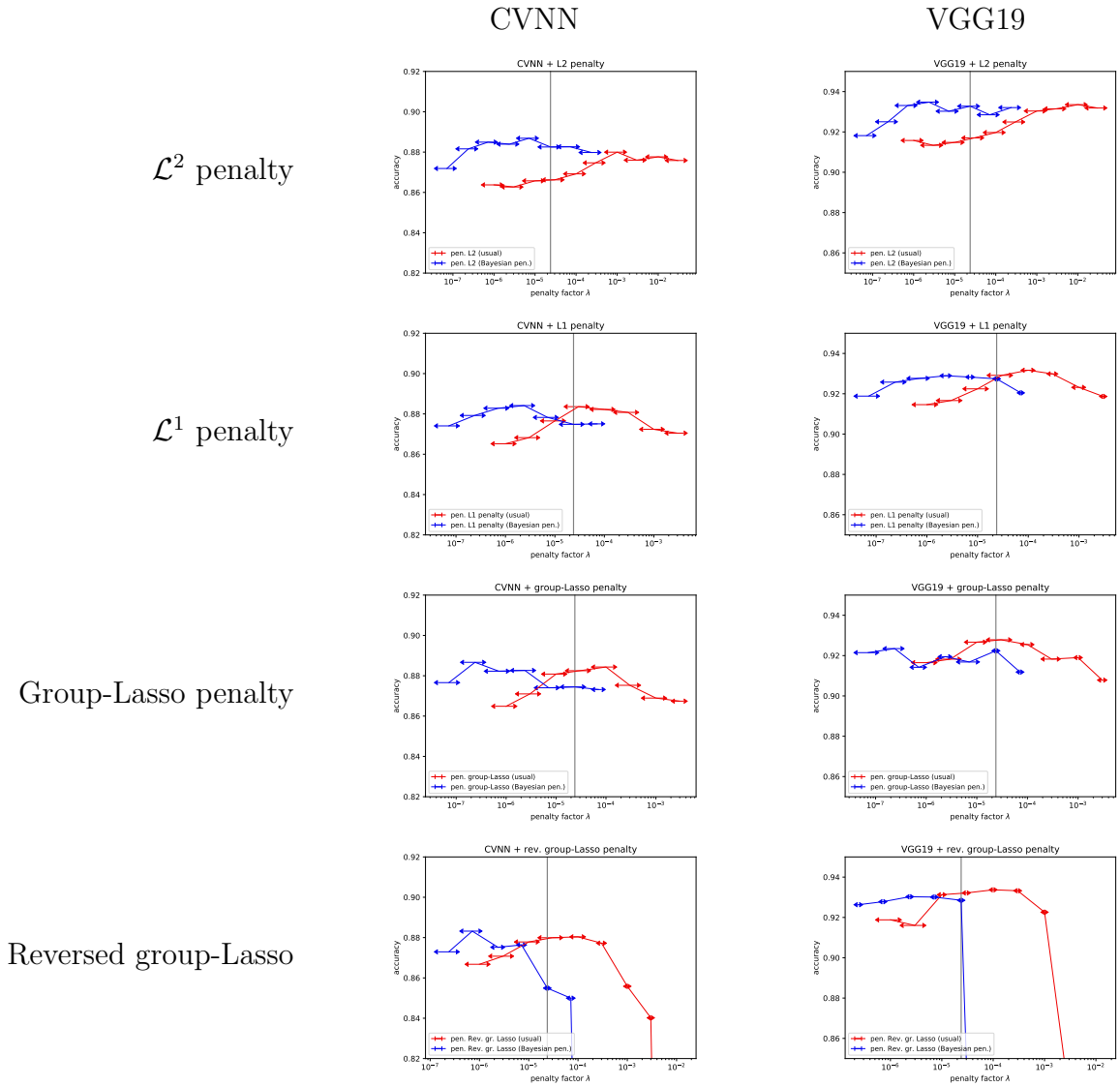


Figure 4.1 – Final performance and number of parameters for various penalties in function of the penalty factor λ. Red line: standard setup for the penalty; blue line: setup provided by the heuristic ($\sharp$) or ($\sharp'$). Each bar corresponds to a final neural network: its abscissa is the penalty factor λ used for training it, its ordinate is its final accuracy, and its width corresponds to its final number of parameters. The vertical grey line is the value of the heuristic for λ.

Table 4.2 – Comparison of the results. We show on the first row the best accuracy $\text{acc}^*_{\text{usual}}$ obtained with the usual setup. Then we show the best accuracy $\text{acc}^*_{\text{Bayesian}}$ obtained with our setup, the accuracy $\text{acc}_{\text{Bayesian}}$ obtained with our setup with the Bayesian theoretical value for λ, that is $\lambda_{\text{Th}} = n^{-1}$, and the ratio between λ_{Th} and the factor $\lambda = \lambda^*$, that is the optimal value for λ. Among the results obtained with our methods, those highlighted in blue are better, and those highlighted in red are worse than in the usual setup.

	$\mathcal{L}^2$ pen.		$\mathcal{L}^1$ pen.		group-Lasso pen.		rev. gr.-Lasso pen.	
	CVNN	VGG	CVNN	VGG	CVNN	VGG	CVNN	VGG
$\text{acc}^*_{\text{usual}}$ (%)	$88.00 \pm .4$	$93.35 \pm .15$	$88.36 \pm .3$	$93.17 \pm .3$	$88.43 \pm .14$	$92.78 \pm .19$	$88.04 \pm .4$	$93.37 \pm .09$
$\text{acc}^*_{\text{Bayesian}}$	$88.69 \pm .12$	$93.48 \pm .09$	$88.41 \pm .3$	$92.89 \pm .2$	$88.67 \pm .09$	$92.35 \pm .18$	$88.32 \pm .16$	$93.03 \pm .15$
$\text{acc}_{\text{Bayesian}}$	$88.25 \pm .3$	$93.28 \pm .17$	$87.48 \pm .08$	$92.74 \pm .19$	$87.45 \pm .17$	$92.24 \pm .14$	$85.49 \pm .3$	$92.85 \pm .06$
$\lambda_{\text{Th}}/\lambda^*$	$10^{0.5}$	10^1	10^1	10^1	10^2	10^2	$10^{1.5}$	10^1

Discussion. This systematic overestimation of the penalty factor indicates that some phenomenon is not yet understood. We have two mutually compatible explanations.

First, the choice $\lambda = 1/n$ could be overestimated, as a strict Bayesian setup might be overcautious from the very beginning. Indeed, in other Bayesian approaches to neural networks, such observations have been made. For instance, when using stochastic Langevin dynamics to approximate the posterior, performance is better if the weight of the posterior is arbitrarily decreased by an additional factor n (see footnote 5 in [75], and [59]).

Second, our choice for λ_l, based on Glorot's initialization, could also be overestimated. Indeed, λ_l only depends on P_l (the number of parameters in each neuron in layer l), and not on the location of layer l in the network for instance, while the following observations suggest that the heuristic used to set λ_l should take into account the architecture. We noticed that in most of the pruning experiments we made, the reached level of sparsity is much higher in the second half of the network (layers closer to the output). This is corroborated by [112] which shows that the layers may behave differently from a training point of view, depending on the architecture of the network and their position in it. Notably, they have proven that, in a trained VGG, some of the last convolutional layers can be reset to their initial value without changing much the final accuracy, while this cannot be done for the first layers.

4.7 Conclusion

We have provided a theorem that bridges the gap between empirical penalties and Bayesian priors when learning the distribution of the parameters of a model. This way, various regularization techniques can be studied in the same Bayesian framework, and be seen as probability distributions. This unified point of view allowed us to take into account well known heuristics (as Glorot's initialization) in order to find reasonable values for hyperparameters of the penalty.

We have checked experimentally that our theoretical framework leads to reasonable results. However, we noticed a constant mismatch (about a factor 10) between our predicted penalty factor λ_{Th} and the best one λ^*. This fact raises interesting questions about a possible overcautiousness of the Bayesian framework and the possible impact of the architecture when penalizing layers, which we plan to investigate further.

4.8 Appendix

4.8.1 Training and Pruning: Details

Since pruning neurons causes an accuracy drop, we divided the learning into two phases: learning and pruning phase; fine-tuning phase. This trick is widely used in pruning literature [60, 89, 65], in order to achieve better performance.

We define a pruning criterion based on the "norm" of each neuron. The tested penalties, i.e. $\mathcal{L}^2$ penalty, $\mathcal{L}^1$ penalty, group-Lasso penalty, and reversed group-Lasso penalty, can be separated into two categories: penalization of the output features of each layer ($\mathcal{L}^2$, $\mathcal{L}^1$, group-Lasso) and penalization of input features of each layer (reversed group-Lasso).

Pruning with $\mathcal{L}^2$, $\mathcal{L}^1$ and group-Lasso penalties. For each neuron $\mathbf{w}_{l,i}$, we check whether:

$$\|\mathbf{w}_{l,i}\|_2 \leq 0.001.$$

If so, then the neuron is pruned.

Pruning with reversed group-Lasso penalty. For each vector $\mathbf{w}_{l,\cdot,j}$ of the j-th input weights of the neurons in layer l, we check whether:

$$\|\mathbf{w}_{l,\cdot,j}\|_2 \leq 0.001.$$

If so, then the j-th neuron in layer $l-1$ is pruned, because its output is almost not used in layer l.

Pruning and training phase. The penalty is applied and, after each training epoch, pruning is performed over all layers. This phase ends when the number of neurons and the best validation accuracy have not improved for 50 epochs.

Fine-tuning phase. The penalty is removed and 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. No pruning is performed during this phase.

4.8.2 Experimental Procedure

For each combination of neural network (VGG19 or CVNN) and penalty ($\mathcal{L}^2$, $\mathcal{L}^1$, group-Lasso, or reversed group Lasso), we have tested two setups: our Bayesian heuristic for the penalty factor λ_l of each layer l, and the usual setup for them (as described in Table 4.1). For each setup, we have planned to plot the final accuracy and final number of parameters in function of the global penalty factor λ. We have proceeded as follows:

1. we define a set Λ of λ we want to use into our experiments. Typically, we have chosen $\Lambda = (1/n) \cdot \{10^{-2.5}, 10^{-2}, 10^{-1.5}, 10^{-1}, 10^{-0.5}, 10^{0}, 10^{0.5}\}$ in our setup (where n is the size of the training set), and $\Lambda = \{10^{-6}, 10^{-5.5}, 10^{-5}, 10^{-4.5}, 10^{-4}, 10^{-3.5}, 10^{-3}, 10^{-2.5}\}$ in the usual setup;

2. for each $\lambda \in \Lambda$, we test several learning rates $\eta \in \{10^{-2}, 10^{-3}, 10^{-4}\}$, and we select the learning rate η_λ which led to the best accuracy;

3. for each $\lambda \in \Lambda$, we run 2 more experiments with the selected learning rate η_λ. Thus, we are able to average our results over 3 runs.

4.8.3 Reminder of Distribution Theory

In order to explain the main result, we recall some basic concepts of distribution theory. We use three functional spaces: the Schwartz class $\mathcal{S}(\mathbb{R}^N)$, the space of tempered distributions $\mathcal{S}'(\mathbb{R}^N)$, and the space of distributions with compact support $\mathcal{E}'(\mathbb{R}^N)$.

Above all, we recall the definition of the space of distributions $\mathcal{D}'(\mathbb{R}^N)$. We denote by $C^\infty(\mathbb{R}^N)$ the space of infinitely derivable functions mapping $\mathbb{R}^N$ to $\mathbb{R}$, and by $C_c^\infty(\mathbb{R}^N) \subset C^\infty(\mathbb{R}^N)$ the subspace of functions with compact support, that is $\varphi \in C_c^\infty(\mathbb{R}^N)$ if, and only if:

$$\varphi \in C^\infty(\mathbb{R}^N) \text{ and } \exists K \subset \mathbb{R}^N \text{ compact s.t.: } \{x \in \mathbb{R}^N : \varphi(x) \neq 0\} \subseteq K.$$

The set $\{x \in \mathbb{R}^N : \varphi(x) \neq 0\}$ is also denoted by $\text{supp}(\varphi)$.

Space of distributions $\mathcal{D}'(\mathbb{R}^N)$. The space of distributions $\mathcal{D}'(\mathbb{R}^N)$ is defined as the space of continuous linear forms over $C_c^\infty(\mathbb{R}^N)$. For any *distribution* $T \in \mathcal{D}'(\mathbb{R}^N)$, we denote by $\langle T, \phi \rangle$ the value of T at a given *test function* $\varphi \in C_c^\infty(\mathbb{R}^N)$.

More formally, $T \in \mathcal{D}'(\mathbb{R}^N)$ if, and only if, for all compact set K of $\mathbb{R}^N$, there exists $p \in \mathbb{N}$ and $C > 0$ such that:

$$\forall \varphi \in C_c^\infty(\mathbb{R}^N) \text{ with } \text{supp}(\varphi) \subseteq K, \qquad |\langle T, \varphi \rangle| \leq C \sup_{|\alpha| \leq p} \|\partial^\alpha \varphi\|_\infty.$$

Distributions are easier to visualize in specific cases. For instance, if a function $f : \mathbb{R}^N \to \mathbb{R}$ is integrable on every compact set $K \in \mathbb{R}^N$, then we can define a distribution T_f by:

$$\forall \varphi \in C_c^\infty(\mathbb{R}^N), \qquad \langle T_f, \varphi \rangle = \int f\varphi.$$

Therefore, distributions are often called "generalized functions", since most functions can be seen as distributions. In fact, by abuse of notation, $\langle f, \varphi \rangle$ stands for $\langle T_f, \varphi \rangle$.

Another classic example of distribution is the Dirac at zero δ, defined as follows:

$$\forall \varphi \in C_c^\infty(\mathbb{R}^N), \qquad \langle \delta, \varphi \rangle = \varphi(0).$$

Schwartz class $\mathcal{S}(\mathbb{R}^N)$. A function φ belongs to $\mathcal{S}(\mathbb{R}^N)$ if, and only if $\varphi \in C^\infty(\mathbb{R}^N)$ and:

$$\forall p \in \mathbb{N}, \exists C_p > 0 : \sup_{|\alpha| \leq p, |\beta| \leq p} \|x^\alpha \partial^\beta \varphi(x)\|_\infty \leq C_p.$$

In few words, $\mathcal{S}(\mathbb{R}^N)$ contains smooth and rapidly decreasing functions. For instance, any Gaussian density function belongs to $\mathcal{S}(\mathbb{R}^N)$.

We can easily define the Fourier transform on $\mathcal{S}(\mathbb{R}^N)$. Let $\varphi \in \mathcal{S}(\mathbb{R}^N)$:

$$(\mathcal{F}\varphi)(\xi) = \int_{\mathbb{R}^N} \varphi(x) e^{-i\xi x} \, \mathrm{d}x.$$

Thus, $\mathcal{F}^{-1} = (2\pi)^{-N} \bar{\mathcal{F}}$, where $\bar{\mathcal{F}}\varphi = \mathcal{F}\check{\varphi}$ and $\check{\varphi}(x) = \varphi(-x)$.

Space of tempered distributions $\mathcal{S}'(\mathbb{R}^N)$. Let $T \in \mathcal{D}'(\mathbb{R}^N)$ be a distribution. T belongs to $\mathcal{S}'(\mathbb{R}^N)$ if, and only if, there exists $p \in \mathbb{N}$ and $C > 0$ such that:

$$\forall \varphi \in C_c^\infty(\mathbb{R}^N), \qquad |\langle T, \varphi \rangle| \leq \sup_{|\alpha| \leq p, |\beta| \leq p} \|x^\alpha \partial^\beta \varphi\|_\infty.$$

The Fourier transform is defined on $\mathcal{S}'(\mathbb{R}^N)$ by duality. For any $T \in \mathcal{S}'(\mathbb{R}^N)$, $\mathcal{F}T \in \mathcal{S}'(\mathbb{R}^N)$ and is defined by:

$$\forall \varphi \in \mathcal{S}(\mathbb{R}^N), \qquad \langle \mathcal{F}T, \varphi \rangle = \langle T, \mathcal{F}\varphi \rangle.$$

Notably, this definition allows us to compute the Fourier transform of functions that do not lie in $\mathcal{L}^2$. This is very useful in the applications of Theorem 1, where we need the Fourier transform of $f : x \mapsto x^2$, which is $\mathcal{F}f = -2\pi\delta'' \in \mathcal{S}'(\mathbb{R})$ (where δ'' is the second derivative of the Dirac, defined below).

Space of distributions with compact support $\mathcal{E}'(\mathbb{R}^N)$. Let $T \in \mathcal{D}'(\mathbb{R}^N)$. The support of T is defined by:

$$\mathrm{supp}(T) = \mathbb{R}^N \backslash \{x \in \mathbb{R}^N : \exists \omega \text{ neighborhood of } x \text{ s.t. } T_{|\omega} = 0\}.$$

Thus, T is said to have a compact support if, and only if, $\mathrm{supp}(T)$ is contained into a compact subset of $\mathbb{R}^N$. As fundamental property of $\mathcal{E}'(\mathbb{R}^N)$, one should notice that $\mathcal{E}'(\mathbb{R}^N) \subset \mathcal{S}'(\mathbb{R}^N)$. That is, the Fourier transform is defined on $\mathcal{E}'(\mathbb{R}^N)$.

For instance, the Dirac at zero δ and its derivatives $\delta^{(k)}$ have support $\{0\}$, which is compact:

$$\begin{aligned} \langle \delta, \varphi \rangle &= \varphi(0) \\ \langle \delta^{(k)}, \varphi \rangle &= (-1)^k \varphi^{(k)}(0), \end{aligned}$$

for any test function $\varphi \in C_c^\infty(\mathbb{R}^N)$.

4.8.4 Proof of Theorem 1

Proof. The proof can be split into three parts:

1. according to Definition 1, we have:

$$A_{\boldsymbol{\nu}} := -\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\mathbb{1} - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right],$$

   and we prove that, for all $(\mu, \boldsymbol{\nu})$:

$$\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}}) - \langle A_{\boldsymbol{\nu}}, \beta_{\mu,\boldsymbol{\nu}}\rangle = r_{\boldsymbol{\nu}}(\mu). \tag{4.13}$$

2. we prove that, if $\exp(A_{\boldsymbol{\nu}})$ is a function integrating to $\kappa > 0$, then there exists $K \in \mathbb{R}$ such that:

$$\forall(\mu, \boldsymbol{\nu}), \qquad r(\mu, \boldsymbol{\nu}) = \mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}} \| \alpha_{\boldsymbol{\nu}}) + K, \tag{4.14}$$

   where $\alpha_{\boldsymbol{\nu}} = \frac{1}{\kappa}\exp(A_{\boldsymbol{\nu}})$.

   At that point, if Condition $(\star)$ is satisfied, then r can be written as a KL-divergence with respect to a prior. Thus, Condition $(\star)$ is a sufficient condition to ensure the existence of a solution to Equation (4.4).

3. we prove that, if α is a solution to Equation (4.4), then $\ln(\alpha)$ is equal to $A_{\boldsymbol{\nu}}$ (up to a constant) and $A_{\boldsymbol{\nu}}$ satisfies Condition $(\star)$.

In the proof, and notably in Appendix 4.8.4, we need the following proposition.

Proposition 8. *Let $T \in \mathcal{S}'(\mathbb{R}^N)$ and $\varphi \in \mathcal{S}(\mathbb{R}^N)$. Then $T * \varphi \in \mathcal{S}'(\mathbb{R}^N)$ and:*

$$\mathcal{F}[T * \varphi] = (\mathcal{F}T) \cdot (\mathcal{F}\varphi).$$

Proof. This statement is directly given in the proof of Theorem 3.18 [97]. □

Proof of Equation (4.13)

The distribution $A_{\boldsymbol{\nu}}$ is defined by:

$$A_{\boldsymbol{\nu}} = -\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\mathbb{1} - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right].$$

Since $\mathbb{1} \in \mathcal{S}'(\mathbb{R}^N)$ and $\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}} \in \mathcal{S}'(\mathbb{R}^N)$, then $A_{\boldsymbol{\nu}} \in \mathcal{S}'(\mathbb{R}^N)$ by stability of $\mathcal{S}'$ by Fourier transform ([39], Lemma 7.1.3).

We compute:

$$\begin{aligned}
&-\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}}) - \langle A_{\boldsymbol{\nu}}, \beta_{\mu,\boldsymbol{\nu}}\rangle \\
&= -\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}}) + \left\langle \mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\mathbb{1} + \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right], \beta_{\mu,\boldsymbol{\nu}}\right\rangle \\
&= \left\langle \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right], \beta_{\mu,\boldsymbol{\nu}}\right\rangle,
\end{aligned}$$

since $\int \beta_{\mu,\boldsymbol{\nu}} = 1$.

By definition of the Fourier transform over $\mathcal{S}'(\mathbb{R}^N)$ ([39], Definition 7.1.9), we have:

$$\begin{aligned} &- \operatorname{Ent}(\beta_{0,\boldsymbol{\nu}}) - \langle A_{\boldsymbol{\nu}}, \beta_{\mu,\boldsymbol{\nu}} \rangle \\ &= \left\langle \frac{\mathcal{F} r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}, \mathcal{F}^{-1}\beta_{\mu,\boldsymbol{\nu}} \right\rangle. \end{aligned} \tag{4.15}$$

We compute $\mathcal{F}^{-1}\beta_{\mu,\boldsymbol{\nu}}$:

$$\begin{aligned} (\mathcal{F}^{-1}\beta_{\mu,\boldsymbol{\nu}})(\xi) &= (2\pi)^{-N}\langle e^{i\xi\cdot}, \beta_{\mu,\boldsymbol{\nu}}(\cdot)\rangle \\ &= (2\pi)^{-N}\langle e^{i\xi\cdot}, \beta_{0,\boldsymbol{\nu}}(\cdot - \mu)\rangle \\ &= (2\pi)^{-N}\langle e^{i\xi\cdot}, \check{\beta}_{0,\boldsymbol{\nu}}(\mu - \cdot)\rangle, \end{aligned}$$

using the assumptions over the family $(\beta_{\mu,\boldsymbol{\nu}})_{\mu,\boldsymbol{\nu}}$.

Thus:

$$\begin{aligned} (\mathcal{F}^{-1}\beta_{\mu,\boldsymbol{\nu}})(\xi) &= (2\pi)^{-N}\langle e^{i\xi(\mu-\cdot)}, \check{\beta}_{0,\boldsymbol{\nu}}(\cdot)\rangle \\ &= (2\pi)^{-N} e^{i\xi\mu}\langle e^{-i\xi\cdot}, \check{\beta}_{0,\boldsymbol{\nu}}(\cdot)\rangle \\ &= (2\pi)^{-N} e^{i\xi\mu}(\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}})(\xi). \end{aligned}$$

By injecting the result into Equation (4.15), we have:

$$\begin{aligned} &- \operatorname{Ent}(\beta_{0,\boldsymbol{\nu}}) - \langle A_{\boldsymbol{\nu}}, \beta_{\mu,\boldsymbol{\nu}} \rangle \\ &= (2\pi)^{-N} \left\langle \frac{\mathcal{F} r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}, e^{i\mu\cdot}(\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}})(\cdot) \right\rangle \\ &= (2\pi)^{-N} \left\langle \mathcal{F} r_{\boldsymbol{\nu}}, e^{i\mu\cdot}\frac{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}} \right\rangle \\ &= (2\pi)^{-N} \left\langle \mathcal{F} r_{\boldsymbol{\nu}}, e^{i\mu\cdot} \right\rangle. \end{aligned}$$

Since $\mathcal{F} r_{\boldsymbol{\nu}} \in \mathcal{E}'(\mathbb{R}^N)$, we can apply Theorem 7.1.14 [39]:

$$\begin{aligned} (2\pi)^{-N} \left\langle \mathcal{F} r_{\boldsymbol{\nu}}, e^{i\mu\cdot} \right\rangle &= (\mathcal{F}^{-1}\mathcal{F} r_{\boldsymbol{\nu}})(\mu) \\ &= r_{\boldsymbol{\nu}}(\mu), \end{aligned}$$

which achieves the proof of Equation (4.13).

Proof of Equation (4.14)

Now, we assume that $\exp(A_{\boldsymbol{\nu}})$ is a function integrating to $\kappa > 0$. We define $\alpha_{\boldsymbol{\nu}}$:

$$\alpha_{\boldsymbol{\nu}} = \frac{1}{\kappa}\exp(A_{\boldsymbol{\nu}}),$$

which is non-negative and integrates to 1. Thus, $\alpha_{\boldsymbol{\nu}}$ is a density of probability.

We compute the KL-divergence between $\beta_{\mu,\boldsymbol{\nu}}$ and $\alpha_{\boldsymbol{\nu}}$:

$$\begin{aligned}
\mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}} \| \alpha_{\boldsymbol{\nu}}) &= \int_{\mathbb{R}^N} \ln\left(\frac{\beta_{\mu,\boldsymbol{\nu}}(\theta)}{\alpha_{\boldsymbol{\nu}}(\theta)}\right) \beta_{\mu,\boldsymbol{\nu}}(\theta)\,\mathrm{d}\theta \\
&= -\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) - \langle \ln(\alpha_{\boldsymbol{\nu}}), \beta_{\mu,\boldsymbol{\nu}} \rangle \\
&= -\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) - \langle -\ln(\kappa)\mathbb{1} + A_{\boldsymbol{\nu}}, \beta_{\mu,\boldsymbol{\nu}} \rangle \\
&= -\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) + \ln(\kappa)\,\langle \mathbb{1}, \beta_{\mu,\boldsymbol{\nu}} \rangle - \langle A_{\boldsymbol{\nu}}, \beta_{\mu,\boldsymbol{\nu}} \rangle \\
&= r_{\boldsymbol{\nu}}(\mu) + \ln \kappa,
\end{aligned}$$

since $\beta_{\mu,\boldsymbol{\nu}}$ integrates to 1 and $\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}})$ does not depend on μ. Therefore, using Equation (4.13):

$$\mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}} \| \alpha_{\boldsymbol{\nu}}) = r(\mu, \boldsymbol{\nu}) + \ln \kappa.$$

Moreover, if $A_{\boldsymbol{\nu}}$ does not depend on $\boldsymbol{\nu}$, then:

$$\mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}} \| \alpha) = r(\mu, \boldsymbol{\nu}) + \ln \kappa.$$

Therefore, Condition $(\star)$ is a sufficient condition to ensure the existence of a solution α to Equation (4.4).

Uniqueness of the Solution

We assume that α is a probability distribution in $\mathcal{T}(\mathbb{R}^N)$ and:

$$r_{\boldsymbol{\nu}}(\mu) = \mathrm{KL}(\beta_{\mu,\boldsymbol{\nu}} \| \alpha) + K,$$

where $K \in \mathbb{R}$ is a constant.

Thus:

$$\begin{aligned}
r_{\boldsymbol{\nu}}(\mu) &= \int_{\mathbb{R}^N} \ln\left(\frac{\beta_{\mu,\boldsymbol{\nu}}(\theta)}{\alpha(\theta)}\right) \beta_{\mu,\boldsymbol{\nu}}(\theta)\,\mathrm{d}\theta + K \\
&= -\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) - \int_{\mathbb{R}^N} [\ln(\alpha(\theta)) - K]\,\beta_{\mu,\boldsymbol{\nu}}(\theta)\,\mathrm{d}\theta \\
&= -\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) - \int_{\mathbb{R}^N} \hat{A}(\theta)\beta_{\mu,\boldsymbol{\nu}}(\theta)\,\mathrm{d}\theta \qquad (\text{where } \hat{A}(\theta) := \ln(\alpha(\theta)) - K) \\
&= -\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) - \int_{\mathbb{R}^N} \hat{A}(\theta)\beta_{0,\boldsymbol{\nu}}(\theta - \mu)\,\mathrm{d}\theta \\
&= -\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) - \int_{\mathbb{R}^N} \hat{A}(\theta)\check{\beta}_{0,\boldsymbol{\nu}}(\mu - \theta)\,\mathrm{d}\theta \\
&= -\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}}) - (\hat{A} * \check{\beta}_{0,\boldsymbol{\nu}})(\mu),
\end{aligned}$$

since the convolution between $\hat{A} \in \mathcal{S}'(\mathbb{R}^N)$ and $\check{\beta}_{0,\boldsymbol{\nu}} \in \mathcal{S}(\mathbb{R}^N)$ is well-defined ([97], Theorem 3.13), and $\mathrm{Ent}(\beta_{\mu,\boldsymbol{\nu}}) = \mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})$.

Then, we can apply the Fourier transform:

$$\begin{aligned}\mathcal{F}r_{\boldsymbol{\nu}} &= -2\pi\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\delta - \mathcal{F}(\hat{A} * \check{\beta}_{0,\boldsymbol{\nu}}) \\ &= -2\pi\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\delta - (\mathcal{F}\hat{A}) \cdot (\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}),\end{aligned}$$

by applying Proposition 8. Since $\mathcal{F}\beta_{0,\boldsymbol{\nu}}$ is supposed to be nonzero everywhere, then:

$$\mathcal{F}\hat{A} = \frac{-2\pi\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\delta - \mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}.$$

From now, we just have to compute the inverse Fourier transform of $\mathcal{F}\hat{A}$ to get $\hat{A}$:

$$\begin{aligned}\hat{A} &= \mathcal{F}^{-1}\left[\frac{-2\pi\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\delta - \mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right] \\ &= -\mathcal{F}^{-1}\left[\frac{2\pi\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\delta}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right] - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right] \\ &= -2\pi\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\mathcal{F}^{-1}\left[\frac{\delta}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right] - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right].\end{aligned}$$

We compute the first term, which is a tempered distribution. For all $\varphi \in \mathcal{S}(\mathbb{R}^N)$, we have:

$$\begin{aligned}\left\langle \frac{\delta}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}, \varphi \right\rangle &= \left\langle \delta, \frac{\varphi}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}} \right\rangle \\ &= \frac{\varphi(0)}{(\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}})(0)} \\ &= \varphi(0),\end{aligned}$$

since $(\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}})(0)$ is equal to $\int \check{\beta}_{0,\boldsymbol{\nu}} = \int \beta_{0,\boldsymbol{\nu}} = 1$. Thus, $\frac{\delta}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}} = \delta$.

Therefore:

$$\begin{aligned}\hat{A} &= -2\pi\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\mathcal{F}^{-1}\delta - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right] \\ &= -\mathrm{Ent}(\beta_{0,\boldsymbol{\nu}})\mathbb{1} - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{\boldsymbol{\nu}}}{\mathcal{F}\check{\beta}_{0,\boldsymbol{\nu}}}\right] \\ &= A_{\boldsymbol{\nu}}.\end{aligned}$$

Recalling that $\hat{A}(\theta) := \ln(\alpha(\theta)) - nK$, we have:

$$\ln(\alpha(\theta)) - K = A_{\boldsymbol{\nu}}(\theta),$$

thus $A_{\boldsymbol{\nu}}$ does not depend on $\boldsymbol{\nu}$, from which we deduce that r fulfills Condition $(\star)$ and $\alpha \propto \exp(A)$. □

4.8.5 Note on Remark 4

We show that, for all $r \in \mathcal{S}'(\mathbb{R}^N)$, there exists a sequence $(r_n)_n \in \mathcal{S}'(\mathbb{R}^N)$ converging to r in $\mathcal{S}'(\mathbb{R}^N)$ such that $\mathcal{F}r_n \in \mathcal{E}'(\mathbb{R}^N)$ for all n.

For all $r \in \mathcal{S}'(\mathbb{R}^N)$, $\mathcal{F}r \in \mathcal{S}'(\mathbb{R}^N)$. Let us build the sequence of distributions:

$$q_n = (\mathcal{F}r)\mathbb{1}_{[-n,n]^N}.$$

The sequence $(q_n)_n \in \mathcal{E}'(\mathbb{R}^N)$ converges to $\mathcal{F}r$ in $\mathcal{S}'(\mathbb{R}^N)$. Since $\mathcal{F}^{-1}$ is continuous, we have:

$$\mathcal{F}^{-1}q_n \to r \qquad \text{in } \mathcal{S}'(\mathbb{R}^N).$$

Therefore, we can pose $r_n = \mathcal{F}^{-1}q_n$, which is appropriate.

4.8.6 Proof of Corollary 6

Proof. We apply Formula (4.3):

$$A = -\mathrm{Ent}(\beta_0)\mathbb{1} - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathcal{F}\beta_0}\right].$$

Thus:

$$\exp(A(\theta)) = e^{-\mathrm{Ent}(\beta_0)} \exp\left(-\mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathcal{F}\beta_0}\right](\theta)\right).$$

Assuming there exists $a > 0, b > 0$ and $k \in \mathbb{R}$ such that:

$$\mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathcal{F}\beta_0}\right](\theta) \geq a\|\theta\|^b + k,$$

we can write:

$$0 \leq \exp(A(\theta))e^{\mathrm{Ent}(\beta_0)} = \exp\left(-\mathcal{F}^{-1}\left[\frac{\mathcal{F}r}{\mathcal{F}\beta_0}\right](\theta)\right) \leq \exp\left(-\left[a\|\theta\|^b + k\right]\right).$$

Since $\int \exp\left(-\left[a\|\theta\|^b + k\right]\right) \mathrm{d}\theta$ converges, then $\int \exp(A(\theta))\,\mathrm{d}\theta$ converges. Thus, according to Theorem 1, there exists $\kappa > 0$ such that $\alpha = \frac{1}{\kappa}\exp(A)$ verifies:

$$\mathrm{KL}(\beta_\mu \| \alpha) = r(\mu),$$

up to a constant. □

4.8.7 Proof of Corollary 7

We want to apply Theorem 1 in this case. We assume that α is a solution to Equation (4.4) with $r_{a,b}(\mu_k, \sigma_k^2) = a(\sigma_k^2) + b(\sigma_k^2)\mu_k^2$, such that $\alpha \in \mathcal{T}(\mathbb{R})$. That is, for some constant K:

$$r_{a,b}(\mu_k, \sigma_k^2) = \mathrm{KL}(\beta_{\mu_k,\sigma_k^2} \| \alpha) + K.$$

We check the hypotheses:

- for all $(\mu, \sigma^2) \in \mathbb{R} \times \mathbb{R}^+$, $\beta_{\mu,\sigma^2}(\cdot) = \beta_{0,\sigma^2}(\cdot - \mu)$ and $\beta_{0,\sigma^2} = \check{\beta}_{0,\sigma^2}$;
- $\beta_{\mu,\sigma^2} \in \mathcal{S}(\mathbb{R})$ and $\mathcal{F}\beta_{\mu,\sigma^2}$ is nonzero everywhere;
- $\mathcal{F}r_{a,b,\sigma^2} = 2\pi \left[a(\sigma^2)\delta - b(\sigma^2)\delta''\right] \in \mathcal{E}'(\mathbb{R})$.

Then, we apply Theorem 1, which ensures that r fulfills $(\star)$, that is A_{a,b,σ^2} does not depend on σ^2. First, we compute A_{a,b,σ^2} by using its definition, given in Equation (4.3). A first calculus (see Appendix 4.8.7) leads to:

$$\frac{\mathcal{F}r_{a,b,\sigma^2}}{\mathcal{F}\beta_{0,\sigma^2}} = 2\pi \left[\left(a(\sigma^2) - \sigma^2 b(\sigma^2)\right)\delta - b(\sigma^2)\delta''\right], \tag{4.16}$$

and, according to the calculus made in Appendix 4.8.7, we have:

$$A_{a,b,\sigma^2}(\theta) = -\ln(2\pi e\sigma^2) + \left[-a(\sigma^2) + \sigma^2 b(\sigma^2) - b(\sigma^2)\theta^2\right]. \tag{4.17}$$

Second, we prove in Appendix 4.8.7 that A_{a,b,σ^2} does not depend on σ^2 if, and only if:

$$a(\sigma^2) = \sigma^2 b_0 - a_0 - \frac{\ln(2\pi e\sigma^2)}{2} \tag{4.18}$$

$$b(\sigma^2) = b_0, \tag{4.19}$$

where a_0 and b_0 are real constants. Thus, the loss $r_{a,b}$ takes the following form:

$$r_{a,b}(\mu, \sigma^2) = \sigma^2 b_0 - a_0 - \frac{\ln(2\pi e\sigma^2)}{2} + b_0\mu^2 + K.$$

In that case, $A_{a,b,\sigma^2}(\theta) = A_{a,b}(\theta) = a_0 - b_0\theta^2$. Thus $\alpha_{a,b} = \frac{1}{\kappa}\exp(A_{a,b})$ is a probability distribution, where:

$$\kappa = e^{a_0}\sqrt{\frac{2\pi}{2b_0}}$$

$$\alpha_{a,b}(\theta) = \frac{1}{\sqrt{2\pi\frac{1}{2b_0}}}\exp\left(-\frac{\theta^2}{2\frac{1}{2b_0}}\right).$$

Thus, $\alpha_{a,b}$ can be seen as the density of the Gaussian distribution $\mathcal{N}(0, \frac{1}{2b_0})$.

Therefore $\alpha_{a,b}$ is Gaussian and the penalty is necessarily the Kullback–Leibler divergence between two Gaussian distributions $\beta_{\mu,\sigma^2} \sim \mathcal{N}(\mu, \sigma^2)$ and $\alpha_{a,b} = \alpha_{\sigma_0^2} \sim \mathcal{N}(0, \sigma_0^2)$:

$$\begin{aligned} r_{\sigma_0^2}(\mu, \sigma^2) &= \frac{\sigma^2}{2\sigma_0^2} + \frac{\ln(2\pi\sigma_0^2)}{2} - \frac{\ln(2\pi e\sigma^2)}{2} + \frac{1}{2\sigma_0}\mu^2 + K \\ &= \frac{1}{2}\left[\frac{\sigma^2 + \mu^2}{\sigma_0^2} + \ln\left(\frac{\sigma_0^2}{\sigma^2}\right) - 1\right] + K. \end{aligned}$$

Proof of Equation (4.16)

We compute $\frac{\mathcal{F}r_{a,b,\sigma^2}}{\mathcal{F}\beta_{0,\sigma^2}}$. We recall that $\mathcal{F}r_{a,b,\sigma^2} = 2\pi\left[a(\sigma^2)\delta - b(\sigma^2)\delta''\right]$. Thus we have, for each $\varphi \in \mathcal{S}(\mathbb{R})$:

$$\begin{aligned} \left\langle \frac{\mathcal{F}r_{a,b,\sigma^2}}{\mathcal{F}\beta_{0,\sigma^2}}, \varphi \right\rangle &= \left\langle \frac{2\pi\left[a(\sigma^2)\delta - b(\sigma^2)\delta''\right]}{\mathcal{F}\beta_{0,\sigma^2}}, \varphi \right\rangle \\ &= 2\pi \left\langle a(\sigma^2)\delta - b(\sigma^2)\delta'', \frac{\varphi}{\mathcal{F}\beta_{0,\sigma^2}} \right\rangle \\ &= 2\pi a(\sigma^2)\varphi(0) - 2\pi b(\sigma^2) \left\langle \delta'', \varphi(x)\exp\left(\frac{\sigma^2 x^2}{2}\right) \right\rangle_x \\ &= 2\pi a(\sigma^2)\varphi(0) + 2\pi b(\sigma^2) \left\langle \delta', \left(\varphi'(x) + \varphi(x)\sigma^2 x\right)\exp\left(\frac{\sigma^2 x^2}{2}\right) \right\rangle_x \\ &= 2\pi a(\sigma^2)\varphi(0) \\ &\quad - 2\pi b(\sigma^2) \left\langle \delta, \left(\varphi''(x) + 2\varphi'(x)\sigma^2 x + \varphi(x)\sigma^2(1 + \sigma^2 x^2)\right)\exp\left(\frac{\sigma^2 x^2}{2}\right) \right\rangle_x \\ &= 2\pi\left[a(\sigma^2)\varphi(0) - b(\sigma^2)\left(\varphi''(0) + \sigma^2\varphi(0)\right)\right] \end{aligned}$$

Thus, $\frac{\mathcal{F}r_{a,b,\sigma^2}}{\mathcal{F}\beta_{0,\sigma^2}} = 2\pi\left[\left(a(\sigma^2) - \sigma^2 b(\sigma^2)\right)\delta - b(\sigma^2)\delta''\right]$.

Proof of Equation (4.17)

Theorem 1 defines the following distribution:

$$A_{a,b,\sigma^2} = -\mathrm{Ent}(\beta_{0,\sigma^2})\mathbb{1} - \mathcal{F}^{-1}\left[\frac{\mathcal{F}r_{a,b,\sigma^2}}{\mathcal{F}\beta_{0,\sigma^2}}\right]$$

We have:

$$\begin{cases} \mathrm{Ent}(\beta_{0,\sigma^2}) & = \frac{1}{2}\ln(2\pi e\sigma^2) \\ \frac{\mathcal{F}r_{a,b\sigma^2}}{\mathcal{F}\beta_{0,\sigma^2}} & = 2\pi\left[\left(a(\sigma^2) - \sigma^2 b(\sigma^2)\right)\delta - b(\sigma^2)\delta''\right] \end{cases}$$

Besides, $2\pi\mathcal{F}^{-1}\delta = \mathbb{1}$ and $2\pi(\mathcal{F}^{-1}\delta'')(\theta) = -\theta^2$.

Thus:

$$\begin{aligned} A_{a,b,\sigma^2}(\theta) &= -\frac{1}{2}\ln(2\pi e\sigma^2) - \left[a(\sigma^2) - \sigma^2 b(\sigma^2) + b(\sigma^2)\theta^2\right] \\ &= -\frac{1}{2}\ln(2\pi e\sigma^2) + \left[-a(\sigma^2) + \sigma^2 b(\sigma^2) - b(\sigma^2)\theta^2\right] \end{aligned}$$

Proof of Conditions (4.18) **and** (4.19)

We have:

$$A_{a,b,\sigma^2}(\theta) = -\frac{1}{2}\ln(2\pi e\sigma^2) + \left[-a(\sigma^2) + \sigma^2 b(\sigma^2) - b(\sigma^2)\theta^2\right].$$

The polynomial A_{a,b,σ^2} does not depend on σ^2 if, and only if, its coefficients do not depend on σ^2. That is:

$$\begin{cases} b(\sigma^2) &= b_0 \\ -\frac{1}{2}\ln(2\pi e\sigma^2) - a(\sigma^2) + \sigma^2 b(\sigma^2) &= a_0 \end{cases},$$

that is:

$$\begin{cases} b(\sigma^2) &= b_0 \\ a(\sigma^2) &= \sigma^2 b_0 - a_0 - \frac{\ln(2\pi e\sigma^2)}{2} \end{cases}.$$

Chapter 5

Conclusion

Among the present works, the main results are the tools Alrao, ScaLa and ScaLP, and the formula expressing the Bayesian prior corresponding to an empirical penalty. The training algorithm Alrao is based on the intuition that diversity in a neural network can counterbalance a poorly precise optimization. This intuition has been tested in Saxe's work [88] by drawing randomly the weights, and we have extended it by drawing randomly the learning rates. As a consequence, Alrao is sufficiently robust and efficient to be used in a very wide range of problems without learning rate tuning. This result is itself quite surprising, and validates the initial intuition: in neural networks, diversity appears to be an important parameter. Moreover, Alrao could be used to train and test automatically generated architecture, without hyperparameter tuning, which could be useful to improve the speed of architecture search algorithms.

The work which has resulted in the tools ScaLa and ScaLP is initially theoretical. It goes two steps beyond the well-known Glorot's heuristics for the initialization of a neural network: first, the activations should be reasonable not only at initialization, but also after the first gradient step; second, the heuristics should remain reasonable for infinitely wide neural networks. The first step leads to ScaLa, whose main benefit is its resilience to layer width change. The second step leads naturally to the pruning method ScaLP, which is more robust than other pruning methods, while being competitive with them. Practical use aside, these good properties tends to confirm the theoretical importance of the initial intuition about the variance of the activations. Therefore, this subject still needs to be explored.

The formula given in Chapter 4, which builds a link from a given penalty towards its corresponding prior distribution is a contribution to the theoretical analysis of empirical regularization methods. Still, the technical conditions of the main theorem seem to be too strict regarding the usual penalties and families of variational posteriors.

To summarize, these works point towards four research axes: evaluate the importance of diversity in neural networks, push further the intuition about the variance of the activations, understand the behavior of infinitely wide neural networks,

and translate empirical regularization methods into Bayesian terms. These research axes may seem purely theoretical, but, as we have shown in this thesis, they are also likely to lead naturally to useful new methods (removing hyperparameters, find heuristics, training and pruning methods, etc.).

Bibliography

[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] Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Comput.*, 10:251–276, February 1998.

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

[4] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[5] Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. In *International Conference on Learning Representations*, 2018.

[6] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[7] Yoshua Bengio, Nicolas L. Roux, Pascal Vincent, Olivier Delalleau, and Patrice Marcotte. Convex neural networks. In Y. Weiss, B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 123–130. MIT Press, 2006.

[8] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[9] 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.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[11] 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.

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

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[14] 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.

[15] Michael Denkowski and Graham Neubig. Stronger baselines for trustable results in neural machine translation. *arXiv preprint arXiv:1706.09733*, 2017.

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

[17] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159, 2011.

[18] Akram Erraqabi and Nicolas Le Roux. Combining adaptive algorithms and hypergradient method: a performance and robustness study. 2018.

[19] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Small, Trainable Neural Networks. *arXiv preprint arXiv:1704.04861*, mar 2018.

[20] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. The lottery ticket hypothesis at scale, 2019.

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

[22] 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.

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

[24] Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356, 2011.

[25] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, et al. A brief review of the ChaLearn AutoML challenge: any-time any-dataset learning without human intervention. In *Workshop on Automatic Machine Learning*, pages 21–30, 2016.

[26] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149*, 2015.

[27] 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.

[28] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both Weights and Connections for Efficient Neural Networks. In *NIPS*, 2015.

[29] 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.

[30] 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.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *ICCV*, pages 770–778, 2016.

[32] Mark Herbster and Manfred K Warmuth. Tracking the best expert. *Machine learning*, 32(2):151–178, 1998.

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

[34] Geoffrey E Hinton and Drew van Camp. Keeping neural networks simple. In *International Conference on Artificial Neural Networks*, pages 11–18. Springer, 1993.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[36] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[37] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

[38] 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.

[39] Lars Hörmander. *The Analysis of Linear Partial Differential Operators I*. Springer, 1998.

[40] Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

[41] 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.

[42] Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[43] Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.

[44] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[45] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pages 2342–2350, 2015.

[46] Nitish Shirish Keskar and Richard Socher. Improving generalization performance by switching from Adam to SGD. *arXiv preprint arXiv:1712.07628*, 2017.

[47] Kianglu. pytorch-cifar, 2018.

[48] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*, 2015.

[49] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015.

[50] Wouter Koolen and Steven De Rooij. Combining expert advice efficiently. *arXiv preprint arXiv:0802.2015*, 2008.

[51] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. 2009.

[52] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[53] Keita Kurita. Learning Rate Tuning in Deep Learning: A Practical Guide | Machine Learning Explained, 2018.

[54] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[55] Yann LeCun, Leon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998.

[56] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *NIPS 2*, pages 598–605. Morgan-Kaufmann, 1990.

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

[58] 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.

[59] Chunyuan Li, Changyou Chen, David E. Carlson, and Lawrence Carin. Preconditioned stochastic gradient Langevin dynamics for deep neural networks. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 1788–1794, 2016.

[60] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[61] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 18(1):6765–6816, 2017.

[62] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[63] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[64] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[65] 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.

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

[67] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017.

[68] David Mack. How to pick the best learning rate for your machine learning project, 2016.

[69] David JC MacKay. Bayesian model comparison and backprop nets. In *Advances in neural information processing systems*, pages 839–846, 1992.

[70] David JC MacKay. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

[71] David JC MacKay. Probable networks and plausible predictions—a review of practical bayesian methods for supervised neural networks. *Network: computation in neural systems*, 6(3):469–505, 1995.

[72] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[73] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[74] Ashique Rupam Mahmood, Richard S Sutton, Thomas Degris, and Patrick M Pilarski. Tuning-free step-size adaptation. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 2121–2124. IEEE, 2012.

[75] Gaétan Marceau-Caron and Yann Ollivier. Natural langevin dynamics for neural networks. In *International Conference on Geometric Science of Information*, pages 451–459. Springer, 2017.

[76] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993.

[77] Pierre-Yves Massé and Yann Ollivier. Speed learning on the fly. *arXiv preprint arXiv:1511.02540*, 2015.

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

[79] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[80] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org, 2017.

[81] Kenton Murray and David Chiang. Auto-sizing neural networks: With applications to n-gram language models. *arXiv preprint arXiv:1508.05051*, 2015.

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

[83] Bruno A Olshausen and David J Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325, 1997.

[84] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[85] Andrei D Polyanin and Alexander V Manzhirov. *Handbook of integral equations*. CRC press, 1998.

[86] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[87] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[88] Andrew M Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Y Ng. On random weights and unsupervised feature learning. In *ICML*, pages 1089–1096, 2011.

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

[90] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351, 2013.

[91] Nicol N Schraudolph. Local gain adaptation in stochastic gradient descent. 1999.

[92] Abigail See, Minh-Thang Luong, and Christopher D Manning. Compression of neural machine translation models via pruning. *CoNLL 2016*, page 291, 2016.

[93] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

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

[95] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[96] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[97] Elias M Stein and Guido Weiss. *Introduction to Fourier analysis on Euclidean spaces (PMS-32)*, volume 32. Princeton university press, 2016.

[98] Pavel Surmenok. Estimating an Optimal Learning Rate For a Deep Neural Network, 2017.

[99] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *ICCV*, pages 1–9, 2015.

[100] Sergios Theodoridis. *Machine learning: a Bayesian and optimization perspective*. Academic Press, 2015.

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

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

[103] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[104] Tim Van Erven, Peter Grünwald, and Steven De Rooij. Catching up faster by switching sooner: A predictive approach to adaptive estimation with an application to the AIC-BIC dilemma. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 74(3):361–417, 2012.

[105] Tim Van Erven, Steven D. Rooij, and Peter Grünwald. Catching up faster in Bayesian model selection and model averaging. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *NIPS 20*, pages 417–424. Curran Associates, Inc., 2008.

[106] Paul AJ Volf and Frans MJ Willems. Switching between two universal source coding algorithms. In *Data Compression Conference, 1998. DCC'98. Proceedings*, pages 491–500. IEEE, 1998.

[107] Larry Wasserman. Bayesian Model Selection and Model Averaging. *Journal of Mathematical Psychology*, 44, 2000.

[108] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[109] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In *NIPS*, pages 4148–4158, 2017.

[110] 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.

[111] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. 2017.

[112] Chiyuan Zhang, Samy Bengio, and Yoram Singer. Are all layers created equal? *arXiv preprint arXiv:1902.01996*, 2019.

[113] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendix A

Proofs

A.1 He's Initialization Rule: Details of Example 1.2.2

The content of this section is derived from the paper of He et al. [30].

Let us consider a neuron in the layer l performing the operation:

$$x_{l+1} = \phi(y_l) = \phi(\mathbf{w}_l \cdot \mathbf{x}_l),$$

where $\mathbf{x}_l$ is the input vector, x_{l+1} is the output (number) of the neuron, $\mathbf{w}_l$ is its vector of weights, and ϕ its activation function.

We assume that the components of $\mathbf{w}_l$ are i. i. d. from a symmetric distribution centered in 0. Therefore:

$$\begin{aligned}\mathrm{Var}(y_l) &= \mathrm{Var}\left(\mathbf{w}_l \cdot \mathbf{x}_l\right) \\ &= n_{l-1}\mathrm{Var}\left(w_l x_l\right),\end{aligned}$$

where w_l and x_l are random variables respectively with the same distribution as each component of $\mathbf{w}_l$ and $\mathbf{x}_l$, and n_{l-1} is their dimension. Thus:

$$\mathrm{Var}(y_l) = n_{l-1}\mathrm{Var}(w_l)\mathbb{E}x_l^2$$

Moreover, if ϕ is the Rectified Linear Unit (ReLU) activation function, i. e. $\phi(x) = \mathrm{ReLU}(x) = \max(x, 0)$, then:

$$\begin{aligned}\mathrm{Var}(y_l) &= n_{l-1}\mathrm{Var}(w_l)\mathbb{E}\left[x_l^2\right] \\ &= n_{l-1}\mathrm{Var}(w_l)\mathbb{E}\left[\phi(y_{l-1})^2\right] \\ &= n_{l-1}\mathrm{Var}(w_l)\mathbb{E}\left[y_{l-1}^2 \mathbb{1}_{y_{l-1}\geq 0}\right]\end{aligned}$$

Since all the weights of the neural network are supposed to be independently drawn from a symmetric distribution centered in 0, the output y_{l-1} of a neuron

in the $(l-1)$-th layer is a random variable whose distribution is symmetric and centered in 0. Therefore:

$$\mathbb{E}\left[y_{l-1}^2 \mathbb{1}_{y_{l-1}\geq 0}\right] = \frac{1}{2}\mathbb{E}\left[y_{l-1}^2\right] = \mathrm{Var}(y_{l-1}).$$

Finally, we have:

$$\mathrm{Var}(y_l) = \frac{1}{2} n_{l-1} \mathrm{Var}(w_l)\mathrm{Var}(y_{l-1}),$$

which means that the variance of the pre-activations y_l is preserved if:

$$\mathrm{Var}(w_l) = \frac{2}{n_{l-1}}.$$

A.2 Variational Inference: Proof of Equation (1.4)

The content of this section is derived from the paper of Graves [24].

We recall equation (1.4):

$$\mathrm{KL}(\pi_{\mathcal{D}} \| \beta) = -\mathbb{E}_{\mathbf{w}\sim\beta} \ln p_{\mathbf{w}}(\mathcal{D}) + \mathrm{KL}(\alpha\|\beta) + \ln \mathbb{P}(\mathcal{D}),$$

where $\mathcal{D}$ is the training dataset, $p_{\mathbf{w}}(\mathcal{D})$ is the likelihood of $\mathcal{D}$ according to the model $\mathcal{M}_{\mathbf{w}}$, α is the prior distribution over $\mathbf{w}$, β is a probability distribution over $\mathbf{w}$, and $\pi_{\mathcal{D}}$ is the posterior distribution of $\mathbf{w}$ given $\mathcal{D}$.

Proof. For any distribution β, the following equations hold:

$$\begin{aligned}
\pi_{\mathcal{D}}(\mathbf{w}) &= \frac{p_{\mathbf{w}}(\mathcal{D})\alpha(\mathbf{w})}{\mathbb{P}(\mathcal{D})} \\
\frac{\pi_{\mathcal{D}}(\mathbf{w})}{\beta(\mathbf{w})} &= p_{\mathbf{w}}(\mathcal{D}) \cdot \frac{\alpha(\mathbf{w})}{\beta(\mathbf{w})} \cdot \frac{1}{\mathbb{P}(\mathcal{D})} \\
-\ln \frac{\pi_{\mathcal{D}}(\mathbf{w})}{\beta(\mathbf{w})} &= -\ln p_{\mathbf{w}}(\mathcal{D}) - \ln \frac{\alpha(\mathbf{w})}{\beta(\mathbf{w})} - \ln \frac{1}{\mathbb{P}(\mathcal{D})} \\
\ln \frac{\beta(\mathbf{w})}{\pi_{\mathcal{D}}(\mathbf{w})} &= -\ln p_{\mathbf{w}}(\mathcal{D}) + \ln \frac{\beta(\mathbf{w})}{\alpha(\mathbf{w})} + \ln \mathbb{P}(\mathcal{D}).
\end{aligned}$$

By integrating over β, we have:

$$\begin{aligned}
\mathbb{E}_{\mathbf{w}\sim\beta} \ln \frac{\beta(\mathbf{w})}{\pi_{\mathcal{D}}(\mathbf{w})} &= -\mathbb{E}_{\mathbf{w}\sim\beta} \ln p_{\mathbf{w}}(\mathcal{D}) + \mathbb{E}_{\mathbf{w}\sim\beta} \ln \frac{\beta(\mathbf{w})}{\alpha(\mathbf{w})} + \ln \mathbb{P}(\mathcal{D}) \\
\mathrm{KL}(\pi_{\mathcal{D}}\|\beta) &= -\mathbb{E}_{\mathbf{w}\sim\beta} \ln p_{\mathbf{w}}(\mathcal{D}) + \mathrm{KL}(\alpha\|\beta) + \ln \mathbb{P}(\mathcal{D}).
\end{aligned}$$

□

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

Titre : Apprentissage de structure pour les réseaux de neurones

Mots clés : réseaux de neurones, apprentissage profond, hyperparamètres, élagage, Bayes

Résumé : La structure d'un réseau de neurones détermine dans une large mesure son coût d'entraînement et d'utilisation, ainsi que sa capacité à apprendre. Ces deux aspects sont habituellement en compétition : plus un réseau de neurones est grand, mieux il remplira la tâche qui lui a été assignée, mais plus son entraînement nécessitera des ressources en mémoire et en temps de calcul. L'automatisation de la recherche des structures de réseaux efficaces –de taille raisonnable, mais performantes dans l'accomplissement de la tâche– est donc une question très étudiée dans ce domaine.
Dans ce contexte, des réseaux de neurones aux structures variées doivent être entraînés, ce qui nécessite un nouveau jeu d'hyperparamètres d'entraînement à chaque nouvelle structure testée. L'objectif de la thèse est de traiter différents aspects de ce problème. La première contribution est une méthode d'entraînement de réseau qui fonctionne dans un vaste périmètre de structures de réseaux et de tâches à accomplir, sans nécessité de régler le taux d'apprentissage. La deuxième contribution est une technique d'entraînement et d'élagage de réseau, conçue pour être insensible à la largeur initiale de celui-ci. La dernière contribution est principalement un théorème qui permet de traduire une pénalité d'entraînement empirique en *a priori* bayésien, théoriquement bien fondé.
Ce travail résulte d'une recherche des propriétés que doivent théoriquement vérifier les algorithmes d'entraînement et d'élagage pour être valables sur un vaste ensemble de réseaux de neurones et d'objectifs.

Title : Structural Learning for Neural Networks

Keywords : neural networks, deep learning, hyperparameters, pruning, Bayes

Abstract : The structure of a neural network determines to a large extent its cost of training and use, as well as its ability to learn. These two aspects are usually in competition: the larger a neural network is, the better it will perform the task assigned to it, but the more it will require memory and computing time resources for training. Automating the search of efficient network structures –of reasonable size and performing well– is then a very studied question in this area.
Within this context, neural networks with various structures are trained, which requires a new set of training hyperparameters for each new structure tested. The aim of the thesis is to address different aspects of this problem. The first contribution is a training method that operates within a large perimeter of network structures and tasks, without needing to adjust the learning rate. The second contribution is a network training and pruning technique, designed to be insensitive to the initial width of the network. The last contribution is mainly a theorem that makes possible to translate an empirical training penalty into a Bayesian *prior*, theoretically well founded.
This work results from a search for properties that theoretically must be verified by training and pruning algorithms to be valid over a wide range of neural networks and objectives.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France