Gathering and Exploiting Higher-Order Information when Training Large Structured Models

Pierre Wolinski[0000−0003−1007−0144] (✉)

LAMSADE, Paris-Dauphine University, PSL University, CNRS, 75016 Paris, France
pierre.wolinski@dauphine.psl.eu

Abstract. When training large models, such as neural networks, the full derivatives of order 2 and beyond are usually inaccessible, due to their computational cost. Therefore, among the second-order optimization methods, it is common to bypass the computation of the Hessian by using first-order information, such as the gradient of the parameters (e.g., quasi-Newton methods) or the activations (e.g., K-FAC).
In this paper, we focus on the exact and explicit computation of projections of the Hessian and higher-order derivatives on well-chosen subspaces relevant for optimization. Namely, for a given partition of the set of parameters, we compute tensors that can be seen as "higher-order derivatives according to the partition", at a reasonable cost as long as the number of subsets of the partition remains small.
Then, we give some examples of how these tensors can be used. First, we show how to compute a learning rate per subset of parameters, which can be used for hyperparameter tuning. Second, we show how to use these tensors at order 2 to construct an optimization method that uses information contained in the Hessian. Third, we show how to use these tensors at order 3 (information contained in the third derivative of the loss) to regularize this optimization method. The resulting training step has several interesting properties, including: it takes into account long-range interactions between the layers of the trained neural network, which is usually not the case in similar methods (e.g., K-FAC); the trajectory of the optimization is invariant under affine layer-wise reparameterization. Our code is available on GitHub: `https://github.com/p-wol/GroupedNewton`.

Keywords: Optimization · Newton's method · Deep Learning.

1 Introduction

In machine learning, computing the derivatives of the loss at various orders is challenging when using large models, such as neural networks. While the first-order derivative is relatively cheap to compute and easy to use to train neural networks, things get difficult when it comes to higher-order derivatives. In particular, Hessian-based training algorithms such as Newton's method are

very expensive to use on large models. Therefore, the study of the Hessian of a loss according to many parameters has become a research area in its own right. For derivatives of order 3 and higher, the situation is even worse: their exact computation is far more expensive than the Hessian, and only a few optimization algorithms use them.

Main contribution: extracting higher-order information. Formally, we study a loss $\mathcal{L}$ to be minimized according to a vector of parameters $\boldsymbol{\theta} \in \mathbb{R}^P$. Thus, the order-$d$ derivative of $\mathcal{L}$ at a given point is a tensor of order d with P^d coefficients. Usually, such tensors cannot be computed exactly and explicitly with $d \geq 2$ (which includes the Hessian) for medium-sized models ($P \gtrsim 10^6$).

Instead of trying to approximate these tensors, we propose to compute their projections along well-chosen directions, that are relevant for optimization. This computation can be done efficiently by taking advantage of the practical implementation of the vector of parameters $\boldsymbol{\theta}$ as a tuple of tensors $(\mathbf{T}^1, \cdots, \mathbf{T}^S)$. Such a projection of the order-d derivative yields a tensor of order d with S^d coefficients, instead of P^d. Thus, the Hessian of a model of size $P = 10^6$ represented by a tuple of $S = 20$ tensors can be reduced to a matrix of size $S^2 = 400$, instead of $P^2 = 10^{12}$. More generally, whenever $S \ll P$, the projected order-d derivative of $\mathcal{L}$ is significantly smaller and easier to compute than the full order-d derivative.

Application: computing per-layer learning rates. Then, we show that such projections of the order-1 and order-2 derivatives can be used to compute the optimal learning rates to choose for each one of the S tensors (or subsets of parameters). The procedure we propose to compute per-layer learning rates is both theoretically well-grounded and usable in practice (as long as the number of layers is not too large). In particular, our computation does not neglect long-range interactions between layers.

Application: second-order optimization method. Finally, we show that the information contained in the 1st, 2nd, and 3rd order derivatives is not only accessible at reasonable cost, but can also be used for optimization. In particular, we propose an optimization method that exploits higher-order information about the loss obtained by using the main contribution. For simplicity, our optimization method and Newton's method look similar: in both cases, a linear system $\mathbf{H}_0\mathbf{x} = \mathbf{g}_0$ has to be solved (w.r.t. $\mathbf{x}$), where $\mathbf{g}_0$ and $\mathbf{H}_0$ contain respectively first-order and second-order information about $\mathcal{L}$. Despite this formal resemblance, the difference is enormous: with Newton's method, $\mathbf{H}_0$ is equal to the Hessian $\mathbf{H}$ of $\mathcal{L}$ of size $P \times P$, while with ours, $\mathbf{H}_0$ is equal to a matrix $\bar{\mathbf{H}}$ of size $S \times S$. Thus, $\bar{\mathbf{H}}$ is undoubtedly smaller and easier to compute than $\mathbf{H}$ when $S \ll P$. Nevertheless, since $\bar{\mathbf{H}}$ is a dense matrix, it still contains information about the interactions between the tensors $\mathbf{T}^s$ when they are used in $\mathcal{L}$. This point is crucial because most second-order optimization methods applied to neural networks use a simplified version of the Hessian (or its inverse), usually a diagonal or block-diagonal approximation, ignoring interactions between layers. Additionally,

we propose an anisotropic version of Nesterov's cubic regularization [24], which uses order-3 information to regularize $\bar{\mathbf{H}}$ and avoid instabilities when computing $\bar{\mathbf{H}}^{-1}\bar{\mathbf{g}}$. In particular, the resulting training trajectory is invariant by layer-wise affine reparameterizations, so our method preserves some interesting properties of Newton's method.

Structure of the paper. First, we show the context and motivation of our work in Section 2. Then, we provide in Section 3 our core method, and in Sections 4 and 5 its applications. In Section 6, we present experimental results showing that the developed methods are usable in practice. Finally, we discuss the results in Section 7.

2 Context and motivation

2.1 Higher-order information

It is not a novel idea to extract higher-order information about a loss at a minimal computational cost to improve optimization. This is typically what is done by [6], although it does not go beyond the second-order derivative. In this line of research, the *Hessian-vector product* [28] is a decisive tool, that allows to compute the projection of higher-order derivatives in given directions at low cost (see Appendix A). For derivatives of order 3, Nesterov's cubic regularization of Newton's method [24] uses information of order 3 to avoid too large training steps. Incidentally, we develop an anisotropic variant of this in Section 5. In the same spirit, the use of derivatives of any order for optimization has been proposed [3]

2.2 Using and estimating the Hessian in optimization

The Hessian $\mathbf{H}$ of the loss $\mathcal{L}$ according to the vector of parameters $\boldsymbol{\theta}$ is known to contain useful information about $\mathcal{L}$. Above all, the Hessian is used to develop second-order optimization algorithms. Let us denote by $\boldsymbol{\theta}_t$ the value of $\boldsymbol{\theta}$ at time step t, $\mathbf{g}_t \in \mathbb{R}^P$ the gradient of $\mathcal{L}$ at step t and $\mathbf{H}_t$ its Hessian at step t. One of the most widely known second-order optimization method is Newton's method, whose step is [25, Chap. 3.3]:

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \mathbf{H}_t^{-1}\mathbf{g}_t. \tag{1}$$

Under certain conditions, including strong convexity of $\mathcal{L}$, the convergence rate of Newton's method is quadratic [25, Th. 3.7], which makes it very appealing. Besides, other methods use second-order information without requiring the full computation of the Hessian. For instance, Cauchy's steepest descent [4] is a variation of the usual gradient descent, where the step size is tuned by extracting very little information from the Hessian:

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \eta_t^* \mathbf{g}_t, \quad \text{where} \quad \eta_t^* := \frac{\mathbf{g}_t^T \mathbf{g}_t}{\mathbf{g}_t^T \mathbf{H}_t \mathbf{g}_t}, \tag{2}$$

where the value of $\mathbf{g}_t^T \mathbf{H}_t \mathbf{g}_t$ can be obtained with little computational cost (see Appendix A). However, when optimizing a quadratic function f with Cauchy's steepest descent, $f(\theta_t)$ is known to decrease at a rate $(\frac{\lambda_{\max}-\lambda_{\min}}{\lambda_{\max}+\lambda_{\min}})^2$, where $\lambda_{\max}$ and $\lambda_{\min}$ are respectively the largest and the smallest eigenvalues of the Hessian of f [19, Chap. 8.2, Th. 2]. If the Hessian of f is strongly anisotropic, then this rate is close to one and optimization is slow. For a comparison of the two methods, see [9,19,25].

Finally, there should be some space between Newton's method, which requires the full Hessian $\mathbf{H}$, and Cauchy's steepest descent, which requires minimal and computationally cheap information about $\mathbf{H}$. The optimization method presented in Section 5 explores this in-between space.

Quasi-Newton methods. When the parameter space is high-dimensional, computation of the Hessian $\mathbf{H}_t$ and inversion of the linear system $\mathbf{g}_t = \mathbf{H}_t \mathbf{x}$ are computationally intensive. Quasi-Newton methods are designed to avoid any direct computation of the Hessian, and make extensive use of gradients and finite difference methods to approximate the direction of $\mathbf{H}_t^{-1}\mathbf{g}_t$. For a list of quasi-Newton methods, see [25, Chap. 8]. However, [25] argue that, since it is easy to compute the Hessian by using Automatic Differentiation (AutoDiff), quasi-Newton methods tend to lose their interest.

Applications to deep learning. Many methods overcome the curse of the number of parameters by exploiting the structure of the neural networks. It is then common to neglect interactions between layers, leading to a (block)-diagonal approximation of the Hessian. A first attempt has been made by [32]: they divide the Hessian into blocks, following the division of the network into layers, and its off-diagonal blocks are removed. From another perspective, [27] keeps this block-diagonal structure, but performs an additional approximation on the remaining blocks.

More recently, K-BFGS has been proposed [10], which is a variation of the quasi-Newton method BFGS with block-diagonal approximation and an approximate representation of these blocks. In a similar spirit, the Natural Gradient method TNT [29] also exploits the structure of neural networks by performing a block-diagonal approximation. Finally, AdaHessian [34] efficiently implements a second-order method by approximating the Hessian by its diagonal.

Kronecker-Factored Approximate Curvature (K-FAC) is a method for approximating of the Hessian proposed in [20] in the context of neural network training. K-FAC exploits the specific architecture of neural networks to output a cheap approximation of the true Hessian. Despite its scalability, K-FAC suffers from several problems. First, the main approximation is quite rough, since "[it assumes] statistical independence between products [...] of unit activities and products [...] of unit input derivatives" [20, Sec. 3.1]. Second, even with an approximation of the Hessian, one has to invert it, which is computationally intensive even for small networks. To overcome this difficulty, a block-(tri)diagonal approximation of the inverse of the Hessian is made, which eliminates many of the interactions between the layers.

Summarizing the Hessian. In Section 5, we propose to summarize the Hessian to avoid the expensive computation of the full Hessian. This idea is not new. For instance, [18] proposes to approximate the Hessian with a matrix composed of blocks in which all coefficients are identical. A more broadly used technique to compress the Hessian is to perform *sketching* on it, that is, project it on randomly chosen directions. This idea is used for solving linear systems [35], as well as for minimizing functions [11], and can be further adapted to Newton's method with cubic regularization [13]. Finally, it is also possible to choose the directions of the projection by using available information [26]. This is the strategy that we have adopted in Section 5.

Invariance by affine reparameterization. Several optimization methods, such as Newton's, have an optimization step invariant by affine reparameterization of $\boldsymbol{\theta}$ [1] [23, Chap. 4.1.2]. Specifically, when using Newton's method, it is equivalent to optimize $\mathcal{L}$ according to $\boldsymbol{\theta}$ and according to $\tilde{\boldsymbol{\theta}} = \mathbf{A}\boldsymbol{\theta} + \mathbf{B}$ ($\mathbf{A} \in \mathbb{R}^{P \times P}$ invertible, $\mathbf{B} \in \mathbb{R}^P$). This affine-invariance property holds even if the function $\mathcal{L}$ to minimize is a negative log-likelihood, and one chooses to minimize $\boldsymbol{\theta}$ by the *natural gradient* method [1]. This method is still being studied for its invariance property [37] (which is also a feature of K-FAC), but it requires computing the Hessian of $\mathcal{L}$ at some point.

Methods based on the moments of the gradients. Finally, many methods acquire geometric information on the loss by using only the gradients. For instance, Shampoo [12] uses second-moment information of the accumulated gradients.

2.3 Motivation

What are we really looking for? The methods that aim to estimate the Hessian matrix $\mathbf{H}$ or its inverse $\mathbf{H}^{-1}$ in order to imitate Newton's method implicitly assume that Newton's method is adapted to the current problem. This assumption is certainly correct when the loss to optimize is strongly convex. But, when the loss is not convex and very complicated, e.g. when training a neural network, this assumption is not justified. Worse, it has been shown empirically that, at the end of the training of a neural network, the eigenvalues of the Hessian are concentrated around zero [30], with only a few large positive eigenvalues. Therefore, Newton's method itself does not seem to be recommended for neural network training, so we may not need to compute the full Hessian at all, which would relieve us of a tedious, if not impossible, task.

To avoid such problems, it is very common to regularize the Hessian by adding a small, constant term $\lambda\mathbf{I}$ to it [25, Chap. 6.3]. Also, trust-region Newton methods are designed to handle non-positive-definite Hessian matrices [25, Chap. 6.4] [22].

Importance of the interactions between layers. Also, some empirical works have shown that the role and the behavior of each layer must be considered along its interactions with the other layers, which emphasize the importance of off-diagonal blocks in the Hessian or its inverse. We give two examples. First, [36]

has shown that, at the end of their training, many networks exhibit a strange feature: some (but not all) layers can be reinitialized to their initial value with little loss of the performance. Second, [15] has compared the similarity between the representations of the data after each layer: changing the number of layers can qualitatively change the similarity matrix of the layers [15, Fig. 3]. Among all, these results motivate our search for mathematical objects that show how layers interact.

Per-layer scaling of the learning rates. A whole line of research is concerned with building a well-founded method for finding a good scaling for the initialization distribution of the parameters, and for the learning rates, which can be chosen layer-wise. For instance, a layer-wise scaling for the weights was proposed and theoretically justified in the paper introducing the Neural Tangent Kernels [14]. Also, in the "feature learning" line of work, [33] proposes a relationship between different scalings related to weight initialization and training. Therefore, there is an interest in finding a scalable and theoretically grounded method to build per-layer learning rates.

Unleashing the power of AutoDiff. Nowadays, several libraries provide easy-to-use automatic differentiation packages that allow the user to compute numerically the gradient of a function, and even higher-order derivatives.[1] Ignoring the computational cost, the full Hessian could theoretically be computed numerically without any approximation. To make this computation feasible, one should aim for an simpler goal: instead of computing the Hessian, one can consider a smaller matrix, consisting of projections of the Hessian.

Moreover, one might hope that such projections would "squeeze" the close-to-zero eigenvalues of the Hessian, so that the eigenvalues of the projected matrix would not be too close to zero.

3 Summarizing higher-order information

Let us consider the minimization of a loss function $\mathcal{L} : \mathbb{R}^P \to \mathbb{R}$ according to a variable $\boldsymbol{\theta} \in \mathbb{R}^P$.

Notation. Let us consider a tensor $\mathbf{A} \in \mathbb{R}^{P^d}$. $\mathbf{A}$ contains P^d coefficients denoted by $A_{i_1,\cdots,i_d}$, indexed by a multi-index $(i_1, \cdots, i_d) \in \{1, \cdots, P\}^d$. The tensor $\mathbf{A}$ can be regarded as a multi-linear form on $\mathbf{R}^{P^d}$: for a tuple of vectors $(\mathbf{u}^1, \cdots, \mathbf{u}^d) \in \mathbb{R}^P \times \cdots \times \mathbb{R}^P$, the application of $\mathbf{A}$ to $(\mathbf{u}^1, \cdots, \mathbf{u}^d)$ is defined as follows:

$$\mathbf{A}[\mathbf{u}^1, \cdots, \mathbf{u}^d] := \sum_{i_1=1}^{P} \cdots \sum_{i_d=1}^{P} A_{i_1,\cdots,i_d} u^1_{i_1} \cdots u^d_{i_d} \in \mathbb{R}, \tag{3}$$

where $u^k_{i_k}$ is the i_k-th coordinate of $\mathbf{u}^k$. This operation is also called *tensor contraction.*

[1] With PyTorch: torch.autograd.grad.

Full computation of the derivatives. The order-d derivative of $\mathcal{L}$ at a point $\boldsymbol{\theta}$, that we denote by $\frac{\mathrm{d}^d\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^d}(\boldsymbol{\theta})$, can be viewed as either a d-linear form (see [7] and Appendix L) or as an order-d tensor belonging to $\mathbb{R}^{P^d}$. For convenience, we will use the latter: the coefficients of the tensor $\mathbf{A} = \frac{\mathrm{d}^d\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^d}(\boldsymbol{\theta}) \in \mathbb{R}^{P^d}$ are $A_{i_1,\cdots,i_d} = \frac{\partial^d\mathcal{L}}{\partial\theta_{i_1}\cdots\partial\theta_{i_d}}(\boldsymbol{\theta})$, where $(i_1,\cdots,i_d) \in \{1,\cdots,P\}^d$ is a multi-index. The order-d derivative $\frac{\mathrm{d}^d\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^d}(\boldsymbol{\theta}) \in \mathbb{R}^{P^d}$ contains P^d scalars. But, even when considering its symmetries[2], it is computationally too expensive to compute it exactly for $d \geq 2$ in most cases. For instance, it is not even possible to compute numerically the full Hessian of $\mathcal{L}$ according to the parameters of a small neural network, i.e., with $P = 10^5$ and $d = 2$, the Hessian contains $P^d = 10^{10}$ scalars.

Terms of the Taylor expansion. At the opposite, one can obtain cheap higher-order information about $\mathcal{L}$ at $\boldsymbol{\theta}$ by considering a specific direction $\mathbf{u} \in \mathbb{R}^P$. The Taylor expansion of $\mathcal{L}(\boldsymbol{\theta}+\mathbf{u})$ gives:

$$\mathcal{L}(\boldsymbol{\theta}+\mathbf{u}) = \mathcal{L}(\boldsymbol{\theta}) + \sum_{d=1}^{D} \frac{1}{d!}\frac{\mathrm{d}^d\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^d}(\boldsymbol{\theta})[\mathbf{u},\cdots,\mathbf{u}] + o(\|\mathbf{u}\|^D). \tag{4}$$

The terms of the Taylor expansion contain higher-order information about $\mathcal{L}$ in the direction $\mathbf{u}$. In particular, they can be used to predict how $\mathcal{L}(\boldsymbol{\theta})$ would change if $\boldsymbol{\theta}$ was translated in the direction of $\mathbf{u}$. Additionally, computing the first D terms has a complexity of order $D \times P$, which is manageable even for large models. The trick that allows for such a low complexity, the *Hessian-vector product*, was proposed by [28] and is recalled in Appendix A.

An intermediate solution. First, we define the partial tensor contractions of the order-d derivative of $\mathcal{L}$ at $\boldsymbol{\theta}$ applied to one vector $\mathbf{u}$. We express $\boldsymbol{\theta} \in \mathbb{R}^P$ and $\mathbf{u} \in \mathbb{R}^P$ as tuples of S tensors: $(\mathbf{T}^1,\cdots,\mathbf{T}^S)$ for $\boldsymbol{\theta}$ and $(\mathbf{U}^1,\cdots,\mathbf{U}^S)$ for $\mathbf{u}$. In other words, for any $i \in \{1,\cdots,P\}$, there exist $s \in \{1,\cdots,S\}$ and an index j such that the parameter θ_i is located at T^s_j, and, similarly, u_i is located at U^s_j. We can now define the partial tensor contraction $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u}) \in \mathbb{R}^{S^d}$, which is a tensor with coefficients:

$$(\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u}))_{s_1,\cdots,s_d} = \frac{\partial^d\mathcal{L}}{\partial\mathbf{T}^{s_1}\cdots\partial\mathbf{T}^{s_d}}(\boldsymbol{\theta})[\mathbf{U}^{s_1},\cdots,\mathbf{U}^{s_d}] \tag{5}$$

$$= \sum_{i_1=1}^{P_{s_1}} \cdots \sum_{i_d=1}^{P_{s_d}} \frac{\partial^d\mathcal{L}}{\partial T^{s_1}_{i_1}\cdots\partial T^{s_d}_{i_d}}(\boldsymbol{\theta}) U^{s_1}_{i_1}\cdots U^{s_d}_{i_d}, \tag{6}$$

where P_s is the number of coefficients of the tensor $\mathbf{T}^s$. Thus, $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$ is a tensor of order d and size S in every dimension resulting from a partial contraction of the full derivative $\frac{\mathrm{d}^d\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^d}(\boldsymbol{\theta})$.

[2] If $\mathcal{L}$ is smooth, then, for any permutation σ of $\{1,\cdots,d\}$, $\frac{\partial^d\mathcal{L}}{\partial\theta_{i_1}\cdots\partial\theta_{i_d}} = \frac{\partial^d\mathcal{L}}{\partial\theta_{\sigma(i_1)}\cdots\partial\theta_{\sigma(i_d)}}$.

Table 1: Comparison between three techniques extracting higher-order information about $\mathcal{L}$: size of the result and complexity of the computation.

Technique	Size	Complexity
Full derivative $\frac{\mathrm{d}^d\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^d}(\boldsymbol{\theta})$	P^d	P^d
Taylor term $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})[\mathbb{1}_S, \cdots, \mathbb{1}_S]$	1	$d \times P$
Tensor $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$	S^d	$S^{d-1} \times P$

Now, let us assume that, in the practical implementation of a gradient-based method of optimization of $\mathcal{L}(\boldsymbol{\theta})$, $\boldsymbol{\theta}$ is represented by a tuple of tensors $(\mathbf{T}^1, \cdots, \mathbf{T}^S)$. So, each Taylor term can be expressed as:

$$\begin{aligned}\frac{\mathrm{d}^d\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^d}(\boldsymbol{\theta})[\mathbf{u}, \cdot, \mathbf{u}] &= \sum_{s_1=1}^{S} \cdots \sum_{s_d=1}^{S} \frac{\partial^d\mathcal{L}}{\partial\mathbf{T}^{s_1}\cdots\partial\mathbf{T}^{s_d}}(\boldsymbol{\theta})[\mathbf{U}^{s_1}, \cdots, \mathbf{U}^{s_d}] \\ &= \mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})[\mathbb{1}_S, \cdots, \mathbb{1}_S], \end{aligned} \tag{7}$$

where $\mathbb{1}_S \in \mathbb{R}^S$ is a vector full of ones, the tuple of tensors $(\mathbf{U}^1, \cdots, \mathbf{U}^S)$ represents $\mathbf{u}$.[3] In this case, the trick of [28] applies to the computation of $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$, which is then much less expensive to compute than the full derivative (see Appendix A).

Properties of $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$. We show a comparison between the three techniques in Table 1. If S is small enough, computing $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$ becomes feasible for $d \geq 2$. For usual multilayer perceptrons with L layers, there is one tensor of weights and one vector of biases per layer, so $S = 2L$. This allows to compute $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$ in practice for $d = 2$ even when $L \approx 20$.

According to Eqn. (7), the Taylor term can be obtained by full contraction of $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$. However, $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$, is a tensor of size S^d, and cannot be obtained from the Taylor term, which is only a scalar. Thus, the tensors $\mathbf{D}^d_{\boldsymbol{\theta}}(\mathbf{u})$ extract more information than the Taylor terms, while keeping a reasonable computational cost. Moreover, their off-diagonal elements give access to information about one-to-one interactions between tensors $(\mathbf{T}^1, \cdots, \mathbf{T}^S)$ when they are processed in the function $\mathcal{L}$.

4 Application: computing per-layer learning rates

To build per-layer (or per-subset-of-parameters) learning rates, we partition the set of indices of parameters $\{1, \cdots, P\}$ into S subsets $(\mathcal{I}_s)_{1\leq s\leq S}$, we assign for all $1 \leq s \leq S$ the same learning rate η_s to the parameters $(\theta_p)_{p\in\mathcal{I}_s}$, and we find the vector of learning rates $\boldsymbol{\eta} = (\eta_1, \cdots, \eta_S)$ optimizing the decrease of the loss $\mathcal{L}$ for

[3] $(\mathbf{U}^1, \cdots, \mathbf{U}^S)$ is to $\mathbf{u}$ as $(\mathbf{T}^1, \cdots, \mathbf{T}^S)$ is to $\boldsymbol{\theta}$.

the current training step t, by using its order-2 Taylor approximation.[4] Formally, given a direction $\mathbf{u}_t \in \mathbb{R}^P$ in the parameter space (typically, $\mathbf{u}_t = \mathbf{g}_t$, the gradient) and $\mathbf{U}_t := \mathrm{Diag}(\mathbf{u}_t) \in \mathbb{R}^{P\times P}$, we consider the training step: $\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \mathbf{U}_t\mathbf{I}_{P:S}\boldsymbol{\eta}_t$, that is a training step in a direction based on $\mathbf{u}_t$, distorted by a subset-wise step size $\boldsymbol{\eta}_t$. Then, we minimize the order-2 Taylor approximation of $\mathcal{L}(\boldsymbol{\theta}_{t+1}) - \mathcal{L}(\boldsymbol{\theta}_t)$: $\boldsymbol{\Delta}_2(\boldsymbol{\eta}_t) := -\mathbf{g}_t^T\mathbf{U}_t\mathbf{I}_{P:S}\boldsymbol{\eta}_t + \frac{1}{2}\boldsymbol{\eta}_t^T\mathbf{I}_{S:P}\mathbf{U}_t\mathbf{H}_t\mathbf{U}_t\mathbf{I}_{P:S}\boldsymbol{\eta}_t$, which gives:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{U}_t\mathbf{I}_{P:S}\boldsymbol{\eta}_t^*, \qquad \boldsymbol{\eta}_t^* := (\mathbf{I}_{S:P}\mathbf{U}_t\mathbf{H}_t\mathbf{U}_t\mathbf{I}_{P:S})^{-1}\mathbf{I}_{S:P}\mathbf{U}_t\mathbf{g}_t, \tag{8}$$

where $\mathbf{I}_{S:P} \in \mathbb{R}^{S\times P}$ is the *partition matrix*, verifying $(\mathbf{I}_{S:P})_{sp} = 1$ if $p \in \mathcal{I}_s$ and 0 otherwise, and $\mathbf{I}_{P:S} := \mathbf{I}_{S:P}^T$. Alternatively, $\boldsymbol{\eta}_t^*$ can be written (details are provided in Appendix B):

$$\boldsymbol{\eta}_t^* = \bar{\mathbf{H}}_t^{-1}\bar{\mathbf{g}}_t, \quad \text{where:} \quad \bar{\mathbf{H}}_t := \mathbf{I}_{S:P}\mathbf{U}_t\mathbf{H}_t\mathbf{U}_t\mathbf{I}_{P:S} \in \mathbb{R}^{S\times S}, \quad \bar{\mathbf{g}}_t := \mathbf{I}_{S:P}\mathbf{U}_t\mathbf{g}_t \in \mathbb{R}^S. \tag{9}$$

With the notation of Section 3, $\bar{\mathbf{H}}_t = \mathbf{D}^{(2)}_{\boldsymbol{\theta}_t}(\mathbf{u}_t)$ and $\bar{\mathbf{g}}_t = \mathbf{D}^{(1)}_{\boldsymbol{\theta}_t}(\mathbf{u}_t)$. Incidentally, computing $\bar{\mathbf{H}}$ is of complexity SP, and solving $\bar{\mathbf{H}}\mathbf{x} = \bar{\mathbf{g}}$ is of complexity S^2.

5 Application: optimization method

5.1 Presentation

Now that we can compute per-layer learning rates, we decide to incorporate them into an optimization method. However, computing them requires to compute $\bar{\mathbf{H}}^{-1}\bar{\mathbf{g}}$. Usually, inverting such a linear system at every step is considered as hazardous and unstable. Therefore, when using Newton's method, instead of computing descent direction $\mathbf{u} := \mathbf{H}^{-1}\mathbf{g}$, it is very common to add a regularization term: $\mathbf{u}_\lambda := (\mathbf{H} + \lambda\mathbf{I})^{-1}\,\mathbf{g}$ [25, Chap. 6.3].

However, the theoretical ground of such a regularization technique is not fully satisfactory. Basically, the main problem is not having a matrix $\bar{\mathbf{H}}$ with close-to-zero eigenvalues: after all, if the loss landscape is very flat in a specific direction, it is better to make a large training step. The problem lies in the order-2 approximation of the loss made in the training step (8), as well as in Newton's method: instead of optimizing the true decrease of the loss, we optimize the decrease of its order-2 approximation. Thus, the practical question is: does this approximation faithfully model the loss at the current point $\boldsymbol{\theta}_t$, in a region that also includes the next point $\boldsymbol{\theta}_{t+1}$?

To answer this question, one has to take into account order-3 information, and regularize $\bar{\mathbf{H}}$ so that the resulting update remains in a region around $\boldsymbol{\theta}_t$ where the cubic term of the Taylor approximation is negligible. In practice, we propose an anisotropic version of Nesterov's cubic regularization [24].

[4] With the notation of Section 3, $\mathcal{I}_s$ is the set of indices p of the parameters θ_p belonging to the tensor $\mathbf{T}^s$, so the scalars $(\theta_p)_{p\in\mathcal{I}_s}$ correspond to the scalars belonging to $\mathbf{T}^s$. So, everything is as if a specific learning rate η_s is assigned to each $\mathbf{T}^s$.

Anisotropic Nesterov cubic regularization. By using the technique presented in Section 3, the diagonal coefficients $(D_1, \cdots, D_S)$ of $\mathbf{D}^{(3)}_{\boldsymbol{\theta}}(\mathbf{u}) \in \mathbb{R}^{S\times S\times S}$ are available with little computational cost. Let $\mathbf{D} := \mathrm{Diag}(|D_1|^{1/3}, \cdots, |D_S|^{1/3}) \in \mathbb{R}^S$.

We modify the method of [24] by integrating an anisotropic factor $\mathbf{D}$ into the cubic term. Thus, our goal is to minimize according to $\boldsymbol{\eta}$ the function T: $T(\boldsymbol{\eta}) := -\boldsymbol{\eta}^T\bar{\mathbf{g}} + \frac{1}{2}\boldsymbol{\eta}\bar{\mathbf{H}}\boldsymbol{\eta} + \frac{\lambda_{\text{int}}}{6}\|\mathbf{D}\boldsymbol{\eta}\|^3$, where λ_{int} is the *internal damping* coefficient, which can be used to tune the strength of the cubic regularization. Under conditions detailed in Appendix D, this minimization problem is equivalent to finding a solution $\boldsymbol{\eta}_*$ such that:

$$\boldsymbol{\eta}_* = \left(\bar{\mathbf{H}} + \frac{\lambda_{\text{int}}}{2}\|\mathbf{D}\boldsymbol{\eta}_*\|\mathbf{D}^2\right)^{-1}\bar{\mathbf{g}}, \tag{10}$$

which is a regularized version of (8). Finally, this multi-dimensional minimization problem boils down to a scalar root finding problem (see Appendix D).

5.2 Properties

The final method is a combination of the learning rate computed in Eqn. (8) with regularization (10):

Method 1 *Training step* $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{U}_t\mathbf{I}_{P:S}\boldsymbol{\eta}^*_t$*, where* $\boldsymbol{\eta}^*_t$ *is the solution with the largest norm* $\|\mathbf{D}_t\boldsymbol{\eta}\|$ *of the equation:* $\boldsymbol{\eta} = \left(\bar{\mathbf{H}}_t + \frac{\lambda_{\text{int}}}{2}\|\mathbf{D}_t\boldsymbol{\eta}\|\mathbf{D}_t^2\right)^{-1}\bar{\mathbf{g}}_t$.

Encompassing Newton's method and Cauchy's steepest descent. Without the cubic regularization ($\lambda_{\text{int}} = 0$), Newton's method is recovered when using the *discrete partition*, that is, $S = P$ with $\mathcal{I}_s = \{s\}$ for all s, and Cauchy's steepest descent is recovered when using the *trivial partition*, that is, $S = 1$ with $\mathcal{I}_1 = \{1, \cdots, P\}$. See Appendix C for more details.

No need to compute or approximate the full Hessian. The full computation of the Hessian $\mathbf{H}_t \in \mathbb{R}^{P\times P}$ is not required. Instead, one only needs to compute the $S \times S$ matrix $\bar{\mathbf{H}}_t := \mathbf{I}_{S:P}\mathbf{U}_t\mathbf{H}_t\mathbf{U}_t\mathbf{I}_{P:S}$, which can be done efficiently by computing $\mathbf{u}^T\mathbf{H}_t\mathbf{v}$ for a number $S \times S$ of pairs of well-chosen directions $(\mathbf{u}, \mathbf{v}) \in \mathbb{R}^P \times \mathbb{R}^P$. This property is especially useful when $S \ll P$. When optimizing a neural network with $L = 10$ layers and $P = 10^6$ parameters, one can naturally partition the set of parameters into $S = 2L$ subsets, each one containing either all the weights or all the biases of each of the L layers. In this situation, one has to solve a linear system of size $2L = 20$ at each step, which is much more reasonable than solving a linear system of $P = 10^6$ equations. We call this natural partition of the parameters of a neural network the *canonical partition*.

No need to solve a large linear system. Using Equations (8) or (10) requires solving only a linear system of S equations, instead of P in Newton's method. With the cubic regularization, only a constant term is added to the complexity, since it is a matter of scalar root finding.

The interactions between different tensors are not neglected. The matrix $\bar{\mathbf{H}}_t$, which simulates the Hessian $\mathbf{H}_t$, is basically dense: it does not exhibit a (block-)diagonal structure. So, the interactions between subsets of parameters are taken into account when performing optimization steps. In the context of neural networks with the canonical partition, this means that interactions between layers are taken into account during optimization, even if the layers are far from each other. This is a major advantage over many existing approximations of the Hessian or its inverse, which are diagonal or block-diagonal.

Invariance by subset-wise affine reparameterization. As showed in Appendix E, under a condition on the directions $\mathbf{u}_t$,[5] the trajectory of optimization of a model trained by Method 1 is invariant by affine reparameterization of the sub-vectors of parameters $\boldsymbol{\theta}_{\mathcal{I}_s} := \mathrm{vec}(\{\theta_p : p \in \mathcal{I}_s\})$. Let $(\alpha_s)_{1\leq s\leq S}$ and $(\beta_s)_{1\leq s\leq S}$ be a sequence of nonzero scalings and a sequence of offsets, and $\tilde{\boldsymbol{\theta}}$ such that, for all $1 \leq s \leq S$, $\tilde{\boldsymbol{\theta}}_{\mathcal{I}_s} = \alpha_s \boldsymbol{\theta}_{\mathcal{I}_s} + \beta_s$. Then, the training trajectory of the model is the same with both parameterizations $\boldsymbol{\theta}$ and $\tilde{\boldsymbol{\theta}}$. This property is desirable in the case of neural networks, where one can use either the usual or the NTK parameterization, which consists of a layer-wise scaling of the parameters. The relevance of this property is discussed in Appendix E.1.

Compared to the standard regularization $\bar{\mathbf{H}} + \lambda \mathbf{I}$ and Nesterov's cubic regularization, the anisotropic Nesterov regularization does not break the property of invariance by subset-wise scaling of the parameters of (8). This is mainly due to our choice to keep only the diagonal coefficients of $\mathbf{D}^{(3)}_{\boldsymbol{\theta}}(\mathbf{u})$ while discarding the others. In particular, the off-diagonal coefficients contain cross-derivatives that would be difficult to include in an invariant training step.

6 Experiments

6.1 Empirical computation of $\bar{\mathbf{H}}$ and $\boldsymbol{\eta}_*$

As recalled in Section 2, many works perform a diagonal, block-diagonal or block-tridiagional [20] approximation of the Hessian or its inverse. Since a summary $\bar{\mathbf{H}}$ of the Hessian and its inverse $\bar{\mathbf{H}}^{-1}$ are available and all their off-diagonal coefficients have been computed and kept, one can to check if these coefficients are indeed negligible.

Setup. We have trained LeNet-5 and VGG-11'[6] on CIFAR-10 using SGD with momentum. Before each epoch, we compute the full-batch gradient, denoted by $\mathbf{u}$, which we use as a direction to compute $\bar{\mathbf{H}}$, again in full-batch. We report submatrices of $\bar{\mathbf{H}}$ and $\bar{\mathbf{H}}^{-1}$ at initialization and at the epoch where the validation loss is the best in Figure 1a (LeNet) and Figure 1b (VGG-11').

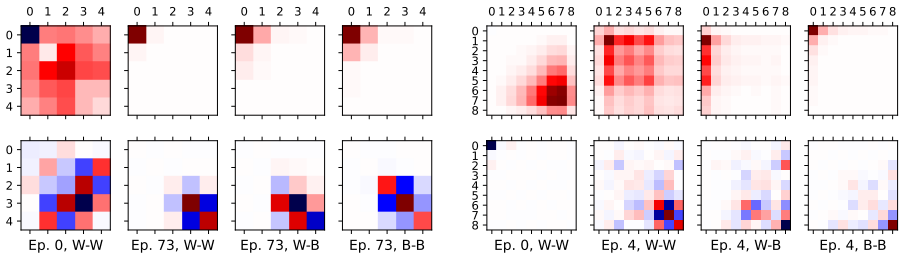
For the sake of readability, $\bar{\mathbf{H}}$ has been divided into blocks: a weight-weight block $\bar{\mathbf{H}}_{\mathrm{WW}}$, a bias-bias block $\bar{\mathbf{H}}_{\mathrm{BB}}$, and a weight-bias block $\bar{\mathbf{H}}_{\mathrm{WB}}$. They represent the interactions between the layers: for instance, $(\bar{\mathbf{H}}_{\mathrm{WB}})_{l_1 l_2}$ represents the

[5] It holds if $\mathbf{u}_t$ is the gradient or a moving average of the gradients (momentum).
[6] VGG-11' is a variant of VGG-11 with 1 final fully-connected layer instead of 3.

interaction between the tensor of weights of layer l_1 and the vector of biases of layer l_2.

Results on $\bar{\mathbf{H}}$. First, the block-diagonal approximation of the Hessian is indeed very rough, while the block-diagonal approximation of the inverse Hessian seems to be more reasonable (at least in these setups), which has already been shown by [20]. Second, there seem to be long-range interactions between layers, both at initialization and after several epochs. For LeNet, all the layers (except the first one) seem to interact together at initialization (Fig. 1a). In the matrix $\bar{\mathbf{H}}^{-1}$ computed on VGG, the last 3 layers interact strongly and the last 6 layers also interact, but a bit less.

According to these observations, a neural network should also be considered as a whole, in which layers can hardly be studied independently from each other. To our knowledge, this result is the first scalable representation of interactions between distant layers, based on second-order information.



(a) LeNet + CIFAR10. (b) VGG-11' + CIFAR10.

Fig. 1: Setup: models trained by SGD on CIFAR-10. Submatrices of $\bar{\mathbf{H}}$ (1st row) and $\bar{\mathbf{H}}^{-1}$ (2nd row), where focus is on interactions: weight-weight, weight-bias, bias-bias of the different layers, at initialization and before best validation loss.

Results on $\boldsymbol{\eta}_*$. The evolution of the learning rates η_* computed according to (10) in LeNet and VGG is shown in Figure 2. First, the learning rates computed for the biases are larger than those computed for the weights. Second, even if only the weights are considered, the computed η_* can differ by several orders of magnitude. Finally, the first two layers of LeNet (which are convolutional) have smaller η_* than the last three layers (which are fully-connected). Conversely, in VGG, the weights of the last (convolutional) layers have a smaller η_* than those of the first layers.

6.2 Training experiments

To show that the projections of the 2nd and 3rd order derivatives of the loss defined in Section 3 can be practically used to train neural networks, we test our

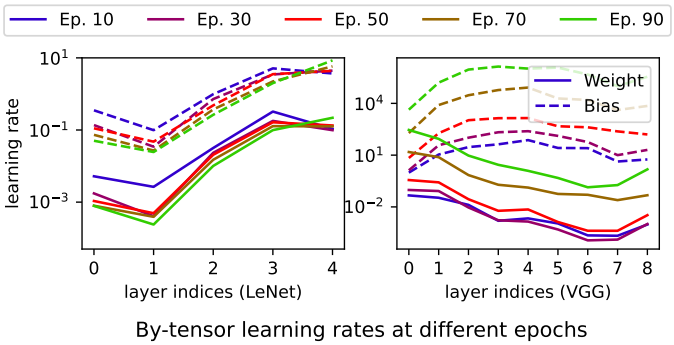


Fig. 2: Setup: LeNet, VGG-11' trained by SGD on CIFAR-10. Learning rates η_* computed according to (10), specific to each tensor of weights and tensor of biases of each layer. For each epoch $k \in \{10, 30, 50, 70, 90\}$, the reported value has been averaged over the epochs $[k-10, k+9]$ to remove the noise.

optimization method 1 (summarized in Algorithm 1) on simple vision tasks. All the implementation details are available in Appendix G. In particular, we have introduced a step size λ_1 that leads to the following modification of the training step (8): $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda_1 \mathbf{U}_t \mathbf{I}_{P:S} \boldsymbol{\eta}_t^*$.

Algorithm 1 Informal description of the 2nd-order method described in Sec. 5. Let $u_t(\cdot)$ be a function computing a direction of descent $\mathbf{u}_t$ from a gradient $\mathbf{g}_t$ and $\mathbf{U}_t = \mathrm{Diag}(\mathbf{u}_t)$.

Hyperparameters: $\lambda, \lambda_{\text{int}}$
$\mathcal{D}_{\text{g}}, \mathcal{D}_{\text{newt}}$: independent samplers of minibatches
for $t \in [1, T]$ **do**
  $Z_t \sim \mathcal{D}_{\text{g}}, \tilde{Z}_t \sim \mathcal{D}_{\text{newt}}$ (sample minibatches)
  $\mathbf{g}_t \leftarrow \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}}(\boldsymbol{\theta}_t, Z_t)$ (backward pass)
  $\mathbf{u}_t \leftarrow u_t(\mathbf{g}_t)$ (custom direction of descent)
  $\bar{\mathbf{g}}_t \leftarrow \mathbf{D}^{(1)}_{\boldsymbol{\theta}_t}(\mathbf{u}_t) = \mathbf{I}_{S:P} \mathbf{U}_t \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}}(\boldsymbol{\theta}_t, \tilde{Z}_t)$
  $\bar{\mathbf{H}}_t \leftarrow \mathbf{D}^{(2)}_{\boldsymbol{\theta}_t}(\mathbf{u}_t) = \mathbf{I}_{S:P} \mathbf{U}_t \frac{\mathrm{d}^2\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}^2}(\boldsymbol{\theta}_t, \tilde{Z}_t) \mathbf{U}_t \mathbf{I}_{P:S}$
  $\mathbf{D}_t \leftarrow \mathrm{Diag}(|\mathbf{D}^{(3)}_{\boldsymbol{\theta}_t}(\mathbf{u}_t)|^{1/3}_{iii} : i \in \{1, \cdots, S\}) \in \mathbb{R}^{S^2}$
  $\boldsymbol{\eta}_t \leftarrow$ sol. of $\boldsymbol{\eta} = \left(\bar{\mathbf{H}}_t + \frac{\lambda_{\text{int}}}{2} \|\mathbf{D}_t \boldsymbol{\eta}\| \mathbf{D}_t^2\right)^{-1} \bar{\mathbf{g}}_t$ with max. norm $\|\mathbf{D}_t \boldsymbol{\eta}\|$ (Method 1)
  $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \lambda \mathbf{U}_t \mathbf{I}_{P:S} \boldsymbol{\eta}_t$ (training step)
end for

Setup. We consider 4 image classification setups:

– **MLP**: multilayer perceptron trained on MNIST with layers of sizes 1024, 200, 100, 10, and tanh activation;

- **LeNet**: LeNet-5 [16] model trained on CIFAR-10 with 2 convolutional layers of sizes 6, 16, and 3 fully connected layers of sizes 120, 84, 10;
- **VGG**: VGG-11' trained on CIFAR-10. VGG-11' is a variant of VGG-11 [31] with only one fully-connected layer at the end, instead of 3, with ELU activation function [5], without batch-norm;
- **BigMLP**: multilayer perceptron trained on CIFAR-10, with 20 layers of size 1024 and one classification layer of size 10, with ELU activation function.

And we have tested 3 optimization methods:

- **Adam**: learning rate selected by grid-search;
- **K-FAC**: learning rate and damping selected by grid-search;
- **NewtonSummary** (ours): λ_1 and λ_{int} selected by grid search.

Results. The evolution of the training loss is plotted in Figure 3 for each of the 3 optimization methods, for 5 different seeds. In each set of experiments, the training is successful, but slow or unstable at some points (e.g., see BigMLP + CIFAR-10). Anyway, the minimum training loss achieved by Method 1 (NewtonSummary) is comparable to the minimum training loss achieved by K-KAC or Adam in all the series except for MLP, whose training is slower. We provide the results on the test set in Appendix I and a comparison of the training times in Appendix M.

Some runs have encountered instabilities due to very large step sizes $\boldsymbol{\eta}_*$. In fact, we did not use any safeguards, such as a regularization term $\lambda\mathbf{I}$ added to $\bar{\mathbf{H}}$, or clipping the learning rates to avoid increasing the number of hyperparameters. So, the training process is vulnerable to the rare situations where $\bar{\mathbf{H}}^{-1}$ has very small or negative eigenvalues and $\mathbf{D}$ has very small coefficients on the diagonal.

Extension to very large models. Since the matrix $\bar{\mathbf{H}}$ can be computed numerically as long as S remains relatively small, this method may become unpractical for very large models. However, Method 1 is flexible enough to be adapted to such models: one can regroup tensors "of the same kind" to build a coarser partition of the parameters, and thus obtain a small S, which is exactly what is needed to compute $\bar{\mathbf{H}}$ and invert it. The difficulty would then be to find a good partition of the parameters, by grouping all the tensors that "look alike". We provide an example in Appendix H with a very deep multilayer perceptron.

Choice of the partition. We propose in Appendix J an empirical study and a discussion about the choice of the partition of the parameters. We show how it affects the training time and the final loss.

Importance of the interactions between layers. We show in Appendix K that the interactions between layers cannot be neglected when using our method: Method 1 outperforms its *diagonal approximation* on LeNet and VGG11', showing the importance of off-diagonal coefficients of $\bar{\mathbf{H}}$.

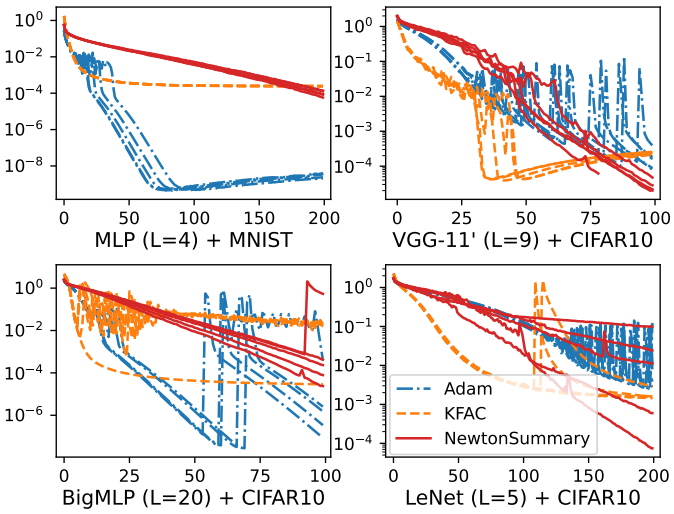


Fig. 3: Training curves in different setups. The reported loss is the negative log-likelihood computed on the training set.

7 Discussion

We have shown that it is possible to obtain 2nd and 3rd order information about the loss, and that this information can be used to construct per-layer learning rates and an optimization method with interesting properties. However, this optimization method can only be seen as a proof of concept, showing that higher-order derivatives are accessible and can be used to train neural networks, and not as a generic optimizer with excellent results on a wide range of tasks and models. Therefore, we propose future research directions.

Convergence rate. Method 1 does not come with a precise convergence rate. The rate proposed in Appendix F (Theorem 1) gives only a heuristic. Given the convergence rates of Newton's method and Cauchy's steepest descent, we can expect to find some in-between convergence rates. Since Cauchy's steepest method is vulnerable to a highly anisotropic Hessian, it would be valuable to know how much this weakness is overcome by our method.

Accounting for the noise during training. Our optimization method remains subject to instabilities during training, which is expected for a second-order method, but not acceptable for the end user. In fact, it is very likely that our algorithm would achieve better performance if it were designed from the beginning to work in a stochastic context. Currently, it is designed as if training was done in full batch.

Acknowledgments. The project leading to this work has received funding from the French National Research Agency (ANR-21-JSTM-0001 and ANR-19-CHIA-0021). This work was granted access to the HPC resources of IDRIS under the allocation 2024-AD011013762R2 made by GENCI. We thank Julyan Arbel, Michael N. Arbel, Gilles Blanchard and Christophe Giraud for their support.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Amari, S.i.: Natural gradient works efficiently in learning. Neural Computation **10**(2), 251–276 (1998)
2. Arora, S., Du, S., Hu, W., Li, Z., Wang, R.: Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In: International Conference on Machine Learning. pp. 322–332 (2019)
3. Birgin, E.G., Gardenghi, J., Martínez, J.M., Santos, S.A., Toint, P.L.: Worst-case evaluation complexity for unconstrained nonlinear optimization using high-order regularized models. Mathematical Programming **163**, 359–368 (2017)
4. Cauchy, A.L.: Méthode générale pour la résolution des systèmes d'équations simultanées. Comptes rendus hebdomadaires des séances de l'Académie des sciences, Paris **25**, 536–538 (1847)
5. Clevert, D.A., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289 (2015)
6. Dangel, F.J.: Backpropagation beyond the gradient. Ph.D. thesis, Universität Tübingen (2023)
7. Dieudonné, J.: Foundations of Modern Analysis. No. 10 in Pure and Applied Mathematics, Academic press (1960)
8. Du, S., Lee, J., Li, H., Wang, L., Zhai, X.: Gradient descent finds global minima of deep neural networks. In: International Conference on Machine Learning. pp. 1675–1685 (2019)
9. Gill, P.E., Murray, W., Wright, M.H.: Practical optimization. Academic Press, San Diego (1981)
10. Goldfarb, D., Ren, Y., Bahamou, A.: Practical quasi-Newton methods for training deep neural networks. In: Advances in Neural Information Processing Systems. vol. 33, pp. 2386–2396 (2020)
11. Gower, R., Kovalev, D., Lieder, F., Richtárik, P.: RSN: randomized subspace Newton. Advances in Neural Information Processing Systems **32** (2019)
12. Gupta, V., Koren, T., Singer, Y.: Shampoo: Preconditioned stochastic tensor optimization. In: International Conference on Machine Learning. pp. 1842–1850 (2018)
13. Hanzely, F., Doikov, N., Nesterov, Y., Richtarik, P.: Stochastic subspace cubic Newton method. In: International Conference on Machine Learning. pp. 4027–4038 (2020)
14. Jacot, A., Gabriel, F., Hongler, C.: Neural tangent kernel: Convergence and generalization in neural networks. In: Advances in Neural Information Processing Systems. vol. 31 (2018)
15. Kornblith, S., Norouzi, M., Lee, H., Hinton, G.: Similarity of neural network representations revisited. In: International Conference on Machine Learning. pp. 3519–3529 (2019)

16. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
17. Lee, J., Xiao, L., Schoenholz, S., Bahri, Y., Novak, R., Sohl-Dickstein, J., Pennington, J.: Wide neural networks of any depth evolve as linear models under gradient descent. In: Advances in Neural Information Processing Systems. vol. 32 (2019)
18. Lu, Y., Harandi, M., Hartley, R., Pascanu, R.: Block mean approximation for efficient second order optimization. arXiv preprint arXiv:1804.05484 (2018)
19. Luenberger, D.G., Ye, Y.: Linear and Nonlinear Programming. Springer, fourth edn. (2008)
20. Martens, J., Grosse, R.: Optimizing neural networks with Kronecker-factored approximate curvature. In: International Conference on Machine Learning. pp. 2408–2417 (2015)
21. Mei, S., Montanari, A.: The generalization error of random features regression: Precise asymptotics and the double descent curve. Communications on Pure and Applied Mathematics **75**(4), 667–766 (2022)
22. Nash, S.G.: Newton-type minimization via the Lanczos method. SIAM Journal on Numerical Analysis **21**(4), 770–788 (1984)
23. Nesterov, Y.: Introductory lectures on convex optimization: A basic course, vol. 87. Springer Science & Business Media (2003)
24. Nesterov, Y., Polyak, B.T.: Cubic regularization of Newton method and its global performance. Mathematical Programming **108**(1), 177–205 (2006)
25. Nocedal, J., Wright, S.J.: Numerical optimization. Springer (1999)
26. Nonomura, T., Ono, S., Nakai, K., Saito, Y.: Randomized subspace Newton convex method applied to data-driven sensor selection problem. IEEE Signal Processing Letters **28**, 284–288 (2021)
27. Ollivier, Y.: Riemannian metrics for neural networks i: feedforward networks. Information and Inference: A Journal of the IMA **4**(2), 108–153 (2015)
28. Pearlmutter, B.A.: Fast exact multiplication by the Hessian. Neural computation **6**(1), 147–160 (1994)
29. Ren, Y., Goldfarb, D.: Tensor normal training for deep learning models. In: Advances in Neural Information Processing Systems. vol. 34, pp. 26040–26052 (2021)
30. Sagun, L., Evci, U., Guney, V.U., Dauphin, Y., Bottou, L.: Empirical analysis of the Hessian of over-parametrized neural networks. In: International Conference on Learning Representations (2018)
31. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
32. Wang, Y.J., Lin, C.T.: A second-order learning algorithm for multilayer networks based on block Hessian matrix. Neural Networks **11**(9), 1607–1622 (1998)
33. Yang, G., Hu, E.J.: Tensor programs iv: Feature learning in infinite-width neural networks. In: International Conference on Machine Learning. pp. 11727–11737 (2021)
34. Yao, Z., Gholami, A., Shen, S., Mustafa, M., Keutzer, K., Mahoney, M.: ADAHESSIAN: An adaptive second order optimizer for machine learning. Proceedings of the AAAI Conference on Artificial Intelligence **35**(12), 10665–10673 (May 2021)
35. Yuan, R., Lazaric, A., Gower, R.M.: Sketched Newton–Raphson. SIAM Journal on Optimization **32**(3), 1555–1583 (2022)
36. Zhang, C., Bengio, S., Singer, Y.: Are all layers created equal? Journal of Machine Learning Research **23**(67), 1–28 (2022)
37. Zhang, G., Martens, J., Grosse, R.B.: Fast convergence of natural gradient descent for over-parameterized neural networks. Advances in Neural Information Processing Systems **32** (2019)