Modular Block-diagonal Curvature Approximations for Feedforward Architectures

Felix Dangel
University of Tübingen
`fdangel@tue.mpg.de`

Stefan Harmeling
Heinrich Heine University Düsseldorf
`harmeling@hhu.de`

Philipp Hennig
University of Tübingen and
MPI for Intelligent Systems, Tübingen
`ph@tue.mpg.de`

Abstract

We propose a modular extension of backpropagation for the computation of block-diagonal approximations to various curvature matrices of the training objective (in particular, the Hessian, generalized Gauss-Newton, and positive-curvature Hessian). The approach reduces the otherwise tedious manual derivation of these matrices into local modules, and is easy to integrate into existing machine learning libraries. Moreover, we develop a compact notation derived from matrix differential calculus. We outline different strategies applicable to our method. They subsume recently-proposed block-diagonal approximations as special cases, and are extended to convolutional neural networks in this work.

1 Introduction

Gradient backpropagation is the central computational operation of contemporary deep learning. Its modular structure allows easy extension across network architectures, and thus automatic computation of gradients given the computational graph of the forward pass (for a review, see Baydin et al., 2018). But optimization using only the first-order information of the objective's gradient can be unstable and slow, due to "vanishing" or "exploding" behaviour of the gradient. Incorporating curvature, second-order methods can avoid such scaling issues and converge in fewer iterations. Such methods locally approximate the objective function E by a quadratic $E(x)+\delta x^\top(x_*-x)+\frac{1}{2}(x_*-x)^\top C(x_*-x)$ around the current location x, using the gradient

$\delta x = \partial E/\partial x$ and a positive semi-definite (PSD) curvature matrix C — the Hessian of E or approximations thereof. The quadratic is minimized by

$$x_* = x + \Delta x \quad \text{with} \quad \Delta x = -C^{-1}\delta x. \qquad (1)$$

Computing the update step requires that the $C\Delta x = -\delta x$ linear system be solved. To accomplish this task, providing a matrix-vector multiplication with the curvature matrix C is sufficient.

Approaches to second-order optimization: For some curvature matrices, exact multiplication can be performed at the cost of one backward pass by automatic differentiation (Pearlmutter, 1994; Schraudolph, 2002). This *matrix-free* formulation can then be leveraged to solve (1) using iterative solvers such as the method of conjugate gradients (CG) (Martens, 2010). However, since this linear solver can still require multiple iterations, the increased per-iteration progress of the resulting optimizer might be compensated by increased computational cost. Recently, a parallel version of Hessian-free optimization was proposed in (Zhang et al., 2017), which only considers the content of Hessian sub-blocks along the diagonal. Reducing the Hessian to a block diagonal allows for parallelization, tends to lower the required number of CG iterations, and seems to improve the optimizer's performance.

There have also been attempts to compute parts of the Hessian in an iterative fashion (Mizutani and Dreyfus, 2008). Storing these constituents efficiently often requires an involved manual analysis of the Hessian's structure, leveraging its outer-product form in many scenarios (Naumov, 2017; Bakker et al., 2018). Recent works developed different block-diagonal approximations (BDA) of curvature matrices that provide fast multiplication (Martens and Grosse, 2015; Grosse and Martens, 2016; Botev et al., 2017; Chen et al., 2018).

These works have repeatedly shown that, empirically, second-order information can improve the training of

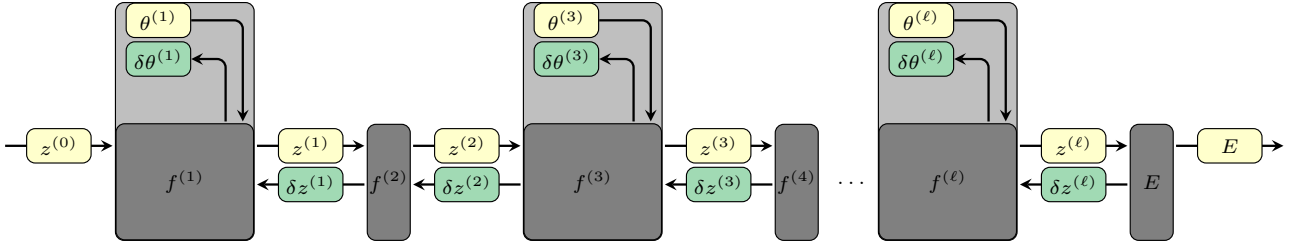Code available at `github.com/f-dangel/hbp`.

Figure 1: Standard feedforward network architecture, i.e. the repetition of affine transformations parameterized by $\theta^{(i)} = (W^{(i)}, b^{(i)})$ followed by elementwise activations. Arrows from left to right and vice versa indicate the data flow during forward pass and gradient backpropagation, respectively.

deep learning problems. Perhaps the most important practical hurdle to the adoption of second-order optimizers is that they tend to be tedious to integrate in existing machine learning frameworks because they require manual implementations. As efficient automated implementations have arguably been more important for the wide-spread use of deep learning than many conceptual advances, we aim to develop a framework that makes computation of Hessian approximations about as easy and automated as gradient backpropagation.

Contribution: This paper introduces a modular formalism for the computation of block-diagonal approximations of Hessian and curvature matrices, to various block resolutions, for feedforward neural networks. The framework unifies previous approaches in a form that, similar to gradient backpropagation, reduces implementation and analysis to local modules. Following the design pattern of gradient backprop also has the advantage that this formalism can readily be integrated into existing machine learning libraries, and flexibly modified for different block groupings and approximations.

The framework consists of three principal parts:

1. a modular formulation for *exact* computation of Hessian block diagonals of feedforward neural nets. We achieve a clear presentation by leveraging the notation of matrix differential calculus (Magnus and Neudecker, 1999).

2. projections onto the positive semi-definite cone by eliminating sources of concavity.

3. backpropagation strategies to obtain (i) exact curvature matrix-vector products (with previously inaccessible BDAs of the Hessian) and (ii) further approximated multiplication routines that save computations by evaluating the matrix representations of intermediate quantities once, at the cost of additional memory consumption.

The first two contributions can be understood as an explicit formulation of well-known tricks for fast mul-

tiplication by curvature matrices using automatic differentiation (Pearlmutter, 1994; Schraudolph, 2002). However, we also address a new class of curvature matrices, the positive-curvature Hessian (PCH) introduced in Chen et al. (2018). Our solutions to the latter two points are generalizations of previous works (Botev et al., 2017; Chen et al., 2018) to the fully modular case, which become accessible due to the first contribution. They represent additional modifications to make the scheme computationally tractable and obtain curvature approximations with desirable properties for optimization.
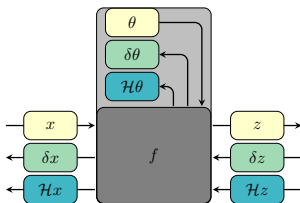
2 Notation

We consider feedforward neural networks composed of ℓ modules $f^{(i)}, i = 1, \ldots, \ell$, which can be represented as a computational graph mapping the input $z^{(0)} = x$ to the output $z^{(\ell)}$ (Figure 1). A module $f^{(i)}$ receives the parental output $z^{(i-1)}$, applies an operation involving the network parameters $\theta^{(i)}$, and sends the output $z^{(i)}$ to its child. Thus, $f^{(i)}$ is of the form $z^{(i)} = f^{(i)}(z^{(i-1)}, \theta^{(i)})$. Typical choices include elementwise nonlinear activation without any parameters and affine transformations $z^{(i)} = W^{(i)} z^{(i-1)} + b^{(i)}$ with parameters given by the weights $W^{(i)}$ and the bias $b^{(i)}$. Affine and activation modules are usually considered as a single conceptual unit, one *layer* of the network. However, for backpropagation of derivatives it is simpler to consider them separately as two *modules*.

Given the network output $z^{(\ell)}(x, \theta^{(1,\ldots,\ell)})$ of a datum x with label y, the goal is to minimize the expected risk of the loss function $E(z^{(\ell)}, y)$. Under the framework of empirical risk minimization, the parameters are tuned to optimize the loss on the training set $Q = \{(x, y)_{i=1}^N\}$,

$$\min_{\theta^{(1,\ldots,\ell)}} \frac{1}{|Q|} \sum_{(x,y) \in Q} E(z^{(\ell)}(x), y). \qquad (2)$$

In practice, the objective is typically further approximated stochastically by drawing a mini-batch $B \subset Q$ from the training set. We will treat both scenarios

Felix Dangel, Stefan Harmeling, Philipp Hennig

Figure 2: Forward pass, gradient backpropagation, and Hessian backpropagation for a single module. Arrows from left to right indicate the data flow in the forward pass $z = f(x, \theta)$, while the opposite orientation indicates the gradient backpropagation by Equation (4). We suggest to extend this by the backpropagation of the Hessian as indicated by Equation (7).

without further distinction, since the structure relevant to our purposes is that Equation (2) is an average of terms depending on individual data points. Quantities for optimization, be it gradients or second derivatives of the loss with respect to the network parameters, can be processed in parallel, then averaged.

3 Main contribution

First-order auto-differentiation for a custom module requires the definition of only two local operations, *forward* and *backward*, whose outputs are propagated along the computation graph. This modularity facilitates the extension of gradient backpropagation by new operations, which can then be used to build networks by composition. To illustrate the principle, we consider a single module from the network of Figure 1, depicted in Figure 2, in this section. The forward pass $f(x, \theta)$ maps the input x to the output z by means of the module parameters θ (to simplify notation, we drop layer indices). All quantities are assumed to be vector-shaped (tensor-valued quantities can be vectorized, see Section A of the Supplements). Optimization requires the gradient of the loss function with respect to the parameters, $\partial E(\theta)/\partial \theta = \delta\theta$. We will use the shorthand

$$\delta\cdot = \frac{\partial E(\cdot)}{\partial \operatorname{vec}(\cdot)}. \tag{3}$$

During gradient backpropagation the module receives the loss gradient with respect to its output, δz, from its child. The backward operation computes gradients with respect to the module parameters and input, $\delta\theta$ and δx from δz. Backpropagation continues by sending the gradient with respect to the module's input to its parent, which proceeds in the same way (see Figure 1). By the chain rule, gradients with respect to an element of the module's input can be computed as $\delta x_i = \sum_j (\partial z_j/\partial x_i)\delta z_j$. The vectorized version is compactly written in terms of the Jacobian matrix $\mathrm{D}z(x) = \partial z(x)/\partial x^\top$, which contains all partial derivatives of z with respect to x. The arrangement of partial derivatives is such that $[\mathrm{D}z(x)]_{j,i} = \partial z_j(x)/\partial x_i$, i.e.

$$\delta x = [\mathrm{D}z(x)]^\top \delta z. \tag{4}$$

Analogously, the parameter gradients are given by $\delta\theta_i = \sum_j \frac{\partial z_j}{\partial \theta_i} \delta z_j$, i.e. $\delta\theta = [\mathrm{D}z(\theta)]^\top \delta z$, which reflects the

symmetry of both x and θ acting as input to the module. Implementing gradient backpropagation thus requires multiplications by (transposed) Jacobians.

We can apply the chain rule a second time to obtain expressions for second-order partial derivatives of the loss function E with respect to elements of x or θ,

$$\begin{aligned}
\frac{\partial^2 E(x)}{\partial x_i \partial x_j} &= \frac{\partial}{\partial x_j}\left(\sum_k \frac{\partial z_k}{\partial x_i}\delta z_k\right) \\
&= \sum_{k,l} \frac{\partial z_k}{\partial x_i}\frac{\partial^2 E(z)}{\partial z_k \partial z_l}\frac{\partial z_l}{\partial x_j} + \sum_k \frac{\partial^2 z_k}{\partial x_i \partial x_j}\delta z_k,
\end{aligned} \tag{5}$$

by means of $\partial/\partial x_j = \sum_l (\partial z_l/\partial x_j)\partial/\partial z_l$ and the product rule. The first term of Equation (5) propagates curvature information of the output further back, while the second term introduces second-order effects of the module itself. Using the Hessian matrix $\mathrm{H}E(x) = \partial^2 E(x)/(\partial x^\top \partial x)$ of a scalar function with respect to a vector-shaped quantity x, the Hessian of the loss function will be abbreviated by

$$\mathrm{H}E(\cdot) = \mathcal{H}\cdot = \frac{\partial^2 E(\cdot)}{\partial \operatorname{vec}(\cdot)^\top \partial \operatorname{vec}(\cdot)}, \tag{6}$$

which results in the matrix version of Equation (5),

$$\mathcal{H}x = [\mathrm{D}z(x)]^\top \mathcal{H}z [\mathrm{D}z(x)] + \sum_k [\mathrm{H}z_k(x)]\delta z_k. \tag{7}$$

Note that the second-order effect introduced by the module itself via $\mathrm{H}z_k(x)$ vanishes if $f_k(x, \theta)$ is linear in x. Because the layer parameters θ can be regarded as inputs to the layer, they are treated in exactly the same way, replacing x by θ in the above expression.

Equation (7) is the central functional expression herein, and will be referred to as the *Hessian backpropagation (HBP) equation*. Our suggested extension of gradient backpropagation is to also send the Hessian $\mathcal{H}z$ back through the graph. To do so, existing modules have to be extended by the HBP equation: *Given the Hessian $\mathcal{H}z$ of the loss with respect to all module outputs, an extended module has to extract the Hessians $\mathcal{H}\theta, \mathcal{H}x$ by means of Equation (7), and forward the Hessian with respect to its input $\mathcal{H}x$ to the parent module which proceeds likewise.* In this way, backprop of gradients
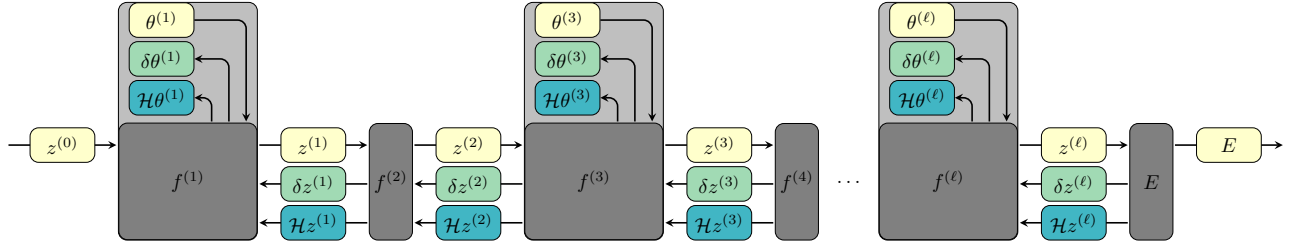
Figure 3: Extension of backprop to Hessians. It yields diagonal blocks of the full parameter Hessian.

can be extended to compute curvature information in modules. This corresponds to BDAs of the Hessian that ignore second-order partial derivatives of parameters in different modules. Figure 3 shows the data flow. The computations required in Equation (7) depend only on *local quantities* that are, mostly, already being computed during gradient backpropagation.[1]

Before we proceed, we highlight the following aspects:

- The BDA of the Hessian need not be PSD. But our scheme can be modified to provide PSD curvature matrices by projection onto the positive semi-definite cone (see Subsection 3.1).

- Instead of evaluating all matrices during backpropagation, we can define matrix-vector products recursively. This yields exact curvature matrix products with the block diagonals of the Hessian, the generalized Gauss-Newton (GGN) matrix and the PCH. Products with the first two matrices can also be obtained by use of automatic differentiation (Pearlmutter, 1994; Schraudolph, 2002). We also get access to the PCH which, in contrast to the GGN, considers curvature information introduced by the network (see Subsection 3.1).[2] For standard neural networks, only second derivatives of nonlinear activations have to be stored compared to gradient backpropagation.

- There are approaches (Botev et al., 2017; Chen et al., 2018) that propagate matrix representations back through the graph in order to save repeated computations in the curvature matrix-vector product. The size of the matrices $\mathcal{H}z^{(i)}$ passed between layer $i+1$ and i scales quadratically in the number of output features of layer i. For convolutional

layers and in case of batched input data, the dimension of these quantities exceeds computational budgets. In line with previous schemes (Botev et al., 2017; Chen et al., 2018), we introduce additional approximations for batch learning in Subsection 3.2. A connection to existing schemes is drawn in the Supplements B.4.

HBP can easily be integrated into current machine learning libraries, so that BDAs of curvature information can be provided automatically for novel or existing second-order optimization methods. Such methods have repeatedly been shown to be competitive with first-order methods (Martens and Grosse, 2015; Grosse and Martens, 2016; Botev et al., 2017; Zhang et al., 2017; Chen et al., 2018).

Relationship to matrix differential calculus: To some extent, this paper is a re-formulation of earlier results (Martens and Grosse, 2015; Botev et al., 2017; Chen et al., 2018) in the framework of matrix differential calculus (Magnus and Neudecker, 1999), leveraged to achieve a new level of modularity. Matrix differential calculus is a set of notational rules that allow a concise construction of derivatives without the heavy use of indices. Equation (7) is a special case of the matrix chain rule of that framework. A more detailed discussion of this connection can be found in Section A of the Supplements, which also reviews definitions generalizing the concepts of Jacobian and Hessian in a way that preserves the chain rule. The elementary building block of our procedure is a *module* as shown in Figure 2. Like for gradient backprop, the operations required for HBP can be tabulated. Table 1 provides a selection of common modules. The derivations, which again leverage the matrix differential calculus framework, can be found in Supplements B, C, and D.

3.1 Obtaining different curvature matrices

The HBP equation yields *exact* diagonal blocks $\mathcal{H}\theta^{(1)}, \ldots, \mathcal{H}\theta^{(\ell)}$ of the full parameter Hessian. They can be of interest in their own right for analysis of the loss function, but are not generally suitable for second-order optimization in the sense of (1), as they need

---

[1] By Faà di Bruno's formula (Johnson, 2002) higher-order derivatives of function compositions are expressed recursively in terms of the composites' lower-order derivatives. Recycling these quantities can give significant speedup compared to repeatedly applying first-order auto-differentiation, which represents one key aspect of our work.

[2] Implementations of HBP for exact matrix-vector products can reuse multiplication by the (transposed) Jacobian provided by many machine learning libraries. The second term of (7) needs special treatment though.

Table 1: Hessian backpropagation for common modules used in feedforward networks. I denotes the identity matrix. We assign matrices to upper-case $(W, X, \dots)$ and tensors to upper-case sans serif symbols $(\mathsf{W}, \mathsf{X}, \dots)$.

OPERATION	FORWARD	HBP (Equation (7))
Matrix-vector multiplication	$z(x, W) = Wx$	$\mathcal{H}x = W^\top (\mathcal{H}z) W$, $\mathcal{H}W = x \otimes x^\top \otimes \mathcal{H}z$
Matrix-matrix multiplication	$Z(X, W) = WX$	$\mathcal{H}X = (I \otimes W)^\top \mathcal{H}Z (I \otimes W)$, $\mathcal{H}W = (X^\top \otimes I)^\top \mathcal{H}Z (X^\top \otimes I)$
Addition	$z(x, b) = x + b$	$\mathcal{H}x = \mathcal{H}b = \mathcal{H}z$
Elementwise activation	$z(x) = \phi(x)$, $z_i(x) = \phi(x_i)$	$\mathcal{H}x = \mathrm{diag}[\phi'(x)] \mathcal{H}z \, \mathrm{diag}[\phi'(x)] + \mathrm{diag}[\phi''(x) \odot \delta z]$
Skip-connection	$z(x, \theta) = x + y(x, \theta)$	$\mathcal{H}x = [I + \mathrm{D}y(x)]^\top \mathcal{H}z [I + \mathrm{D}y(x)] + \sum_k [\mathrm{H}y_k(x)] \delta z_k$, $\mathcal{H}\theta = [\mathrm{D}y(\theta)]^\top \mathcal{H}z [\mathrm{D}y(\theta)] + \sum_k [\mathrm{H}y_k(\theta)] \delta z_k$
Reshape/view	$\mathsf{Z}(\mathsf{X}) = \mathrm{reshape}(\mathsf{X})$	$\mathcal{H}\mathsf{Z} = \mathcal{H}\mathsf{X}$
Index select/map π	$z(x) = \Pi x$, $\Pi_{j, \pi(j)} = 1$,	$\mathcal{H}x = \Pi^\top (\mathcal{H}z) \Pi$
Convolution	$\mathsf{Z}(\mathsf{X}, \mathsf{W}) = \mathsf{X} \star \mathsf{W}$, $Z(W, [\![\mathsf{X}]\!]) = W[\![\mathsf{X}]\!]$,	$\mathcal{H}[\![\mathsf{X}]\!] = (I \otimes W) \mathcal{H}Z (I \otimes W)$ $\mathcal{H}W = ([\![\mathsf{X}]\!]^\top \otimes I)^\top \mathcal{H}Z ([\![\mathsf{X}]\!]^\top \otimes I)$
Square loss	$E(x, y) = (y - x)^\top (y - x)$	$\mathcal{H}x = 2I$
Softmax cross-entropy	$E(x, y) = -y^\top \log[p(x)]$	$\mathcal{H}x = \mathrm{diag}[p(x)] - p(x)p(x)^\top$

neither be PSD nor invertible. For application in optimization, HBP can be modified to yield semi-definite BDAs of the Hessian. Equation (7) again provides the foundation for this adaptation, which is closely related to the concepts of KFRA (Botev et al., 2017), BDA-PCH (Chen et al., 2018), and, under certain conditions, KFAC (Martens and Grosse, 2015). We draw their connections by briefly reviewing them here.

Generalized Gauss-Newton matrix: The GGN emerges as the curvature matrix in the quadratic expansion of the loss function $E(z^{(\ell)})$ in terms of the network output $z^{(\ell)}$. It is also obtained by linearizing the network output $z^{(\ell)}(\theta, x)$ in θ before computing the loss Hessian (Martens, 2014), and reads

$$\mathrm{G}(\theta) = \frac{1}{|Q|} \sum_{(x,y) \in Q} \left[\mathrm{D}z^{(\ell)}(\theta) \right]^\top \mathrm{H}E(z^{(\ell)}) \left[\mathrm{D}z^{(\ell)}(\theta) \right].$$

To obtain diagonal blocks $\mathrm{G}(\theta^{(i)})$, the Jacobian can be unrolled by means of the chain rule for Jacobians (Supplements, Theorem A.1) as $\mathrm{D}z^{(\ell)}(\theta^{(i)}) = \left[\mathrm{D}z^{(\ell)}(z^{(\ell-1)}) \right] \left[\mathrm{D}z^{(\ell-1)}(\theta^{(i)}) \right] \dots$ Continued expansion shows that the Hessian $\mathrm{H}E(z^{(\ell)})$ of the loss function with respect to the network output is propagated back through a layer by multiplication from left and right with its Jacobian. This is accomplished in HBP by *ignoring second-order effects introduced by modules*, that is by setting the Hessian of the module function to zero, therefore neglecting the second term in Equation (7). In fact, if all activations in the network are piecewise linear (e.g. ReLUs), the GGN and Hessian blocks are equivalent. Moreover, diagonal blocks of the

GGN are PSD if the loss function is convex (and thus $\mathrm{H}E(z^{(\ell)})$ is PSD). This is because blocks are recursively left- and right-multiplied with Jacobians, which does not alter the definiteness. Hessians of the loss functions listed in Table 1 are PSD. The resulting recursive scheme has been used by Botev et al. (2017) under the acronym KFRA to optimize convex loss functions of fully-connected neural networks with piecewise linear activation functions.

Positive-curvature Hessian: Another concept of positive semi-definite BDAs of the Hessian (that additionally considers second-order module effects) was studied in Chen et al. (2018) and named the PCH. It is obtained by modification of terms in the second summand of Equation (7) that can introduce concavity during HBP. This ensures positive semi-definiteness since the first summand is semi-definite, assuming the loss Hessian $\mathrm{H}E(z^{(\ell)})$ with respect to the network output to be positive semi-definite. Chen et al. (2018) suggest to eliminate negative curvature of a matrix by computing the eigenvalue decomposition and either discard negative eigenvalues or cast them to their absolute value. This allows the construction of PSD curvature matrices even for non-convex loss functions. In the setting of Chen et al. (2018), the PCH can empirically outperform optimization using the GGN. In usual feedforward neural networks, the concavity is introduced by nonlinear elementwise activations, and corresponds to a diagonal matrix (Table 1). Thus, convexity can be maintained during HBP by either clipping negative values to zero (PCH-clip), or taking their magnitude in the diagonal concave term (PCH-abs).

Fisher information matrix: If the network defines a conditional probability density $r(y|z^{(\ell)})$ on the labels, maximum likelihood learning for the parameterized density $p_\theta(y|x)$ will correspond to choosing a negative log-likelihood loss function, i.e. $E(z^{(\ell)}, y) = -\log r(y|z^{(\ell)})$. Common loss functions like square and cross-entropy loss can be interpreted in this way. Natural gradient descent (Amari, 1998) uses the Fisher information matrix $\mathrm{F}(\theta) = \mathbb{E}_{p_\theta(y|x)}\left[\left(\mathrm{d}\log p_\theta(y|x)/\mathrm{d}\theta\right)\left(\mathrm{d}\log p_\theta(y|x)/\mathrm{d}\theta^\top\right)\right]$ as a PSD curvature matrix approximating the Hessian. It can be expressed as the log-predictive density's expected Hessian under r itself: $F_r(z^{(\ell)}) = -\mathbb{E}_{r(y|z^{(\ell)})}\left[\mathrm{H}\log r(y|z^{(\ell)})\right]$. Assuming truly i.i.d. samples x, the log-likelihood of multiple data decomposes and results in the approximation

$$\mathrm{F}(\theta) \approx \frac{1}{|Q|} \sum_{(x,y)\in Q} \left[\mathrm{D}z^{(\ell)}(\theta)\right]^\top F_r(z^{(\ell)}) \left[\mathrm{D}z^{(\ell)}(\theta)\right].$$

In this form, the computational scheme for BDAs of the Fisher resembles the HBP of the GGN. However, instead of propagating back the loss Hessian with respect to the network, the expected Hessian of the negative log-likelihood under the model's predictive distribution is used. Martens and Grosse (2015) use Monte-Carlo sampling to estimate this matrix in their KFAC optimizer. Relations between the Fisher and GGN are discussed in (Pascanu and Bengio, 2013; Martens, 2014); for square and cross-entropy loss, they are equivalent.

3.2 Batch learning approximations

In our HBP framework, exact multiplication by the block of the curvature matrix of parameter θ in a module comes at the cost of one gradient backpropagation to this layer. The multiplication is recursively defined in terms of multiplication by the layer output Hessian $\mathcal{H}z$. If it were possible to have an explicit representation of this matrix in memory, the recursive computations hidden in $\mathcal{H}z$ could be saved during the solution of the linear system implied by Equation (1). Unfortunately, the size of the backpropagated exact matrices scales quadratically in both the batch size[3] and the number of layer's output features. However, instead of propagating back the exact Hessian, a batch-averaged version can be used instead to circumvent the batch size scaling (originating from Botev et al. (2017)). In combination with structural information about the parameter Hessian, this strategy is used in Botev et al. (2017); Chen et al. (2018) to further approximate curvature multiplications, using quantities computed in a

---

[3] If samples are processed independently in every module, these matrices have block structure and scale linearly in batch size. Quadratic scaling is caused by transformations across different samples, like batch normalization.

single backward pass and then kept in memory for application of the matrix-vector product. We can embed these explicit schemes into our modular approach. To do so, we denote averages over a batch B by a bar, for instance $1/|B| \sum_{(x,y)\in B} \mathrm{H}E(\theta) = \overline{\mathrm{H}E(\theta)}$. The modified backward pass of curvature information during HBP for a module receives a batch average of the Hessian with respect to the output, $\overline{\mathcal{H}z}$, which is used to formulate the matrix-vector product with the batch-averaged parameter Hessian $\overline{\mathcal{H}\theta}$. An average of the Hessian with respect to the module input, $\overline{\mathcal{H}x}$, is passed back. Existing work (Botev et al., 2017; Chen et al., 2018) differs primarily in the specifics of how this batch average is computed. In HBP, these approximations can be formulated compactly within Equation (7). Relations to the cited works are discussed in more detail in the Supplements B.4. The approximations amounting to relations used by Botev et al. (2017) read

$$\overline{\mathcal{H}x} \approx \overline{[\mathrm{D}z(x)]^\top \overline{\mathcal{H}z} [\mathrm{D}z(x)]} + \sum_k \overline{[\mathrm{H}z_k(x)] \delta z_k}, \quad (8)$$

and likewise for θ. In case of a linear layer $z(x) = Wx+b$, this approximation implies the relations $\overline{\mathcal{H}W} = \overline{x \otimes x^\top} \otimes \overline{\mathcal{H}z}$, $\overline{\mathcal{H}b} = \overline{\mathcal{H}z}$, and $\overline{\mathcal{H}x} = W^\top(\overline{\mathcal{H}z})W$. Multiplication by this weight Hessian approximation with a vector v is achieved by storing $\overline{x \otimes x^\top}$, $\overline{\mathcal{H}z}$ and performing the required contractions $v \mapsto (\overline{x \otimes x^\top} \otimes \overline{\mathcal{H}z})v$. Note that this approach is not restricted to curvature matrix-vector multiplication routines only. Kronecker structure in the approximation gives rise to optimization methods relying on direct inversion.

A cheaper approximation, used in Chen et al. (2018),

$$\overline{\mathcal{H}x} \approx \overline{[\mathrm{D}z(x)]}^\top \overline{\mathcal{H}z} \ \overline{[\mathrm{D}z(x)]} + \sum_k \overline{[\mathrm{H}z_k(x)] \delta z_k}, \quad (9)$$

leads to the modified relation $\overline{\mathcal{H}W} = \overline{x} \otimes \overline{x}^\top \otimes \overline{\mathcal{H}z}$ for a linear layer. As this approximation is of the same rank as $\overline{\mathcal{H}z}$, which is typically small, CG requires only a few iterations during optimization. It avoids large memory requirements for layers with numerous inputs, since it requires $\overline{x}$ be stored instead of $\overline{x \otimes x^\top}$.

Transformations that are linear in the module parameters (e.g. linear and convolutional layers), possess constant Jacobians with respect to the module input for each sample (see Table 1). Hence, in a network consisting of only these layers, both Equation (8) and (9) yield the same backpropagated Hessians $\overline{\mathcal{H}x}$. This still leaves the degree of freedom for choosing the approximation scheme in the analogous equations for θ.

Remark: Both strategies for obtaining curvature matrix BDAs (implicit exact matrix-vector multiplications and explicit propagation of approximated curvature)

are compatible. Regarding the connection to cited works, we note that the maximally modular structure of our framework changes the nature of these approximations and allows a more flexible formulation.

4 Experiments & Implementation

We illustrate the usefulness of incorporating curvature information with the two outlined strategies by experiments with a fully-connected and a convolutional neural network (CNN) on the CIFAR-10 dataset (Krizhevsky, 2009). Following the guidelines of Schneider et al. (2019), the training loss is estimated on a random subset of the training set of equal size as the test set. Each experiment is performed for 10 different random seeds and we show the mean values with shaded intervals of one standard deviation. For the loss function we use cross-entropy. Details on the model architectures and hyperparameters are given in Supplements E.

Training procedure and update rule: In comparison to a first-order optimization procedure, the training loop with HBP has to be extended by a single backward pass to backpropagate the batch-averaged or exact loss Hessian. This yields matrix-vector products with a curvature estimate $C^{(i)}$ for each parameter block $\theta^{(i)}$ of the network. Parameter updates $\Delta\theta^{(i)}$ are obtained by applying CG to solve the linear system[4]

$$\left[\alpha I + (1-\alpha)C^{(i)}\right]\Delta\theta^{(i)} = -\delta\theta^{(i)}, \qquad (10)$$

where α acts as a step size limitation to improve robustness against noise. The CG routine terminates if the ratio of the residual norm and the gradient norm falls below a certain threshold or the maximum number of iterations has been reached. The solution returned by CG is scaled by a learning rate γ, and parameters are updated by the relation $\theta^{(i)} \leftarrow \theta^{(i)} + \gamma\Delta\theta^{(i)}$.

Fully-connected network, batch approximations, and sub-blocking: The flexibility of HBP is illustrated by extending the results in Chen et al. (2018). Investigations are performed on a fully-connected network with sigmoid activations. Solid lines in Figure 4 show the performance of the Newton-style optimizer and momentum SGD in terms of the training loss and test accuracy. The second-order method is capable to escape the initial plateau in fewer iterations.

The modularity of HBP allows for additional parallelism by splitting the linear system (10) into smaller sub-blocks, which then also need fewer iterations of CG. Doing so only requires a minor modification of the
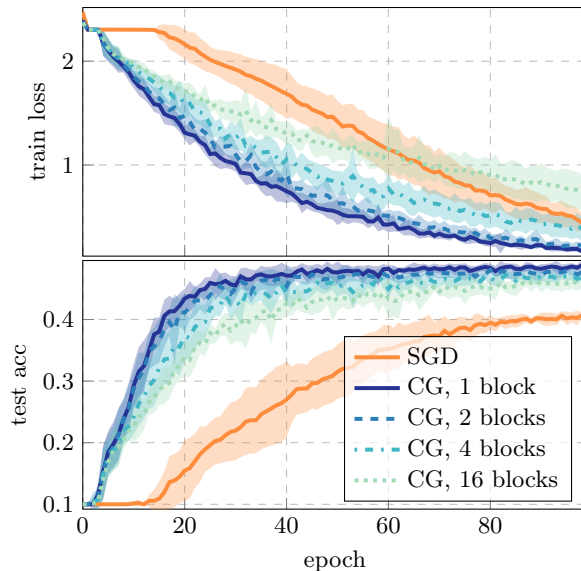

Figure 4: SGD and different Newton-style optimizers based on the PCH-abs with batch approximations. The same fully-connected neural network of (Chen et al., 2018) was used to generate the solid baseline results. Our modular approach allows further splitting the parameter blocks into sub-blocks that can independently be optimized in parallel (dashed lines).

parameter Hessian computation by (7). Consequently, we split weights and bias terms row-wise into a specified number of sub-blocks. Performance curves are shown in Figure 4. In the initial phase, the BDA can be split into a larger number of sub-blocks without suffering from a loss in performance. The reduced curvature information is still sufficient to escape the initial plateau. However, larger blocks have to be considered in later stages to further reduce the loss efficiently.

The fact that this switch in modularity is necessary is an argument in favor of the flexible form of HBP, which allows to efficiently realize such switches: For this experiment, the splitting for each block was artificially chosen to illustrate this flexibility. In principle, the splitting could be decided individually for each parameter block, and even changed at run time.

Convolutional neural network, matrix-free exact curvature multiplication: For convolutions, the large number of hidden features prohibits backpropagating a curvature matrix batch average. Instead, we use exact curvature matrix-vector products provided within HBP. The CNN possesses sigmoid activations and cannot be trained by SGD (cf. Figure 5a). For comparison with another second-order method, we experiment with a public KFAC implementation (Martens and Grosse, 2015; Grosse and Martens, 2016, see Supplements E for details).

---

[4]We use the same update rule as Chen et al. (2018) since we extend some of the results shown within this work.
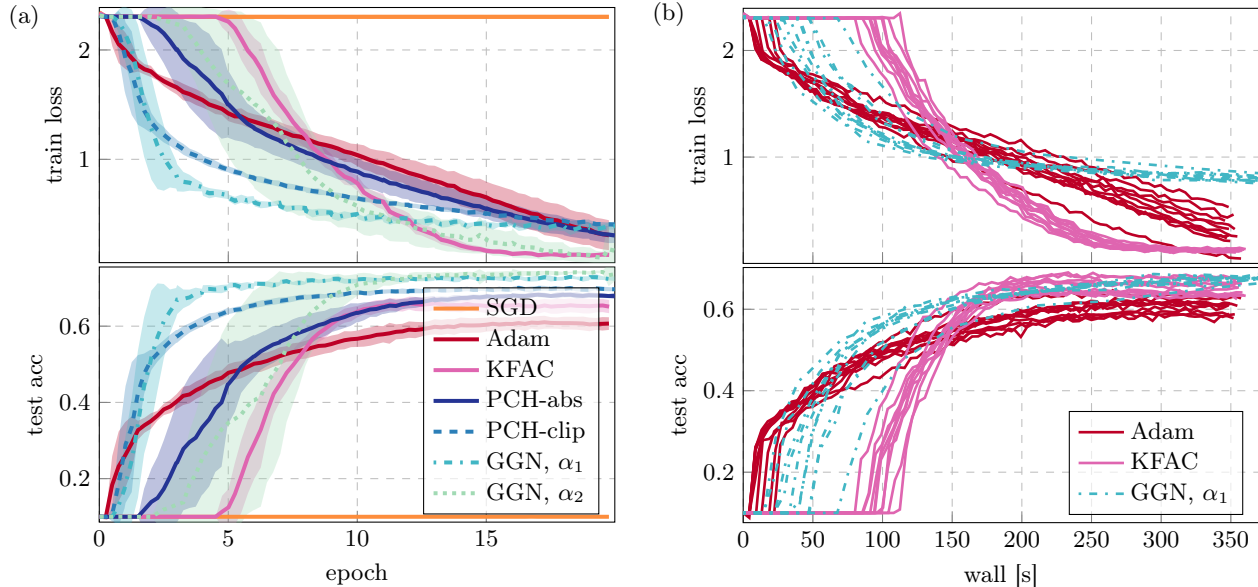
Figure 5: (a) Comparison of SGD, Adam, KFAC, and Newton-style methods with different exact curvature matrix-vector products (HBP) on a CNN with sigmoid activations (see Supplements E). SGD cannot train the net. (b) Wall-clock time comparison (on an RTX 2080 Ti GPU; same colors realize different random seeds).

The matrix-free second-order methods progress fast in the initial stage of the optimization. However, progress in later phases stagnates. This may be caused by the limited sophistication of the update rule (10): If a small value for α is chosen, the optimizer will perform well in the beginning (GGN, α_1). As the gradients become smaller, and hence more noisy, the step size limitation is too optimistic, which leads to a slow-down in optimization progress. A more conservative step size limitation improves the overall performance at the cost of fewer initial progress (GGN, α_2). In the training phase where damping is "effective", our illustrative methods, and KFAC, exhibit better progress per iteration on the objective than the first-order competitor Adam, underlining the usefulness of curvature even if only computed block-wise.

For an impression on performance in terms of run time, Figure 5b compares the wall-clock time of one matrix-free method and the baselines. The HBP-based optimizer can compete with existing methods and offers potential for further improvements, like sub-blocking and parallelized CG. Despite the more adaptive nature of second-order methods, their full power seems to still require adaptive damping, to account for the quality of the local quadratic approximation and restrict the update if necessary. The importance of these techniques to properly adapt the Newton direction has been emphasized in previous works (Martens, 2010; Martens and Grosse, 2015; Botev et al., 2017) that aim to develop fully fletched second-order optimizers. Such adaptation, however, is beyond the scope of this text.

5 Conclusion

We have outlined a procedure to compute block-diagonal approximations of different curvature matrices for feedforward neural networks by a scheme that can be realized on top of gradient backpropagation. In contrast to other recently proposed methods, our implementation is aligned with the design of current machine learning frameworks and can flexibly compute Hessian sub-blocks to different levels of refinement. Its modular formulation facilitates closed-form analysis of Hessian diagonal blocks, and unifies previous approaches (Botev et al., 2017; Chen et al., 2018).

Within our framework we presented two strategies: (i) Obtaining exact curvature matrix-vector products that have not been accessible before by auto-differentiation (PCH), and (ii) backpropagation of further approximated matrix representations to save computations during training. As for gradient backpropagation, the Hessian backpropagation for different operations can be derived independently of the underlying graph. The extended modules can then be used as a drop-in replacement for existing modules to construct deep neural networks. Internally, backprop is extended by an additional Hessian backward pass through the graph to compute curvature information. It can be performed in parallel to, and reuse the quantities computed in, gradient backpropagation.

Acknowledgments

References

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276.

Bakker, C., Henry, M. J., and Hodas, N. O. (2018). The outer product structure of neural network derivatives. *CoRR*, abs/1810.03798.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *Journal of Marchine Learning Research*, 18:1–43.

Botev, A., Ritter, H., and Barber, D. (2017). Practical Gauss-Newton optimisation for deep learning. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pages 557–565. PMLR.

Chen, S.-W., Chou, C.-N., and Chang, E. (2018). BDA-PCH: Block-diagonal approximation of positive-curvature Hessian for training neural networks. *CoRR*, abs/1802.06502.

Grosse, R. and Martens, J. (2016). A Kronecker-factored approximate Fisher matrix for convolution layers. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 573–582. PMLR.

Johnson, W. P. (2002). The curious history of Faà di Bruno's formula. *The American mathematical monthly*, 109(3):217–234.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

Magnus, J. R. and Neudecker, H. (1999). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Probabilistics and Statistics. Wiley.

Martens, J. (2010). Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning*, volume 27, pages 735–742.

Martens, J. (2014). New insights and perspectives on the natural gradient method. *CoRR*, abs/1412.1193.

Martens, J. and Grosse, R. (2015). Optimizing neural networks with Kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pages 2408–2417. JMLR.

Mizutani, E. and Dreyfus, S. E. (2008). Second-order stagewise backpropagation for Hessian-matrix analyses and investigation of negative curvature. *Neural Networks*, 21(2-3):193–203.

Naumov, M. (2017). Feedforward and recurrent neural networks backward propagation and Hessian in matrix form. *CoRR*, abs/1709.06080.

Pascanu, R. and Bengio, Y. (2013). Revisiting natural gradient for deep networks. *arXiv:1301.3584*.

Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. *Neural Computation*, 6:147–160.

Schneider, F., Balles, L., and Hennig, P. (2019). DeepOBS: A deep learning optimizer benchmark suite. In *International Conference on Learning Representations*.

Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14:1723–1738.

Zhang, H., Xiong, C., Bradbury, J., and Socher, R. (2017). Block-diagonal Hessian-free optimization for training neural networks. *CoRR*, abs/1712.07296.