

---

# Modular Block-diagonal Curvature Approximations for Feedforward Architectures (Supplementary Material)

---

**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

This document provides additional information and derivations to clarify the relations of Hessian backpropagation (HBP). Section A contains a clean definition of Jacobian and Hessian matrices for multi-variate functions that are typically involved in the construction of neural networks. We introduce conventions for notation and show how the HBP Equation (7) results naturally from the chain rule for matrix derivatives (Magnus and Neudecker, 1999).

With matrix derivatives, the HBP equation for a variety of module functions can be derived elegantly. Sections B, C, and D contain the HBP derivations for all operations in Table 1. We split the considered operations into different categories to achieve a cleaner structure. Section B contains details on operations used for the construction of fully-connected neural networks (FCNNs) and skip-connections. Subsection B.4 illustrates the analytic composition of multiple modules by combining the backward passes of a nonlinear elementwise activation function and an affine transformation. This yields the recursive schemes of Botev et al. (2017) and Chen et al. (2018), the latter of which has been used in the experiment of Section 4. The analysis of the Hessian for common loss functions is provided in Section C. Operations occurring in convolutional neural networks (CNNs) are subject of Section D.

Section E provides details on model architectures, training procedures used in the experiments of Section 4, and an additional experiment on a modified test problem of the DeepOBS benchmark library (Schneider et al., 2019).

## A Matrix derivatives

Index notation for higher-order derivatives of multi-variate matrix functions can become heavy (Mizutani and Dreyfus, 2008; Naumov, 2017; Chen et al., 2018; Bakker et al., 2018). We tackle this issue by embedding the presented approach in the notation of matrix differential calculus, which

1. yields notation consistent with established literature on matrix derivatives (Magnus and Neudecker, 1999) and clarifies the origin of the symbols  $D$  and  $H$  that are used extensively in the main text.
2. allows for using a multi-dimensional generalization of the chain rule.
3. lets us extract first- and second-order derivatives from differentials using the identification rules of Magnus and Neudecker (1999) without bothering to deal with index notation.

With these techniques it is easy to see how structure, like Kronecker products, appears in the derivatives.

**Preliminaries & notation:** The following definitions and theorems represent a collection of results from the book of Magnus and Neudecker (1999). They generalize the concept of first- and second-order derivatives to multi-variate matrix functions in terms of the Jacobian and Hessian matrix. While there exist multiple ways to arrange the partial derivatives, the presented definitions allows for a multivariate generalization of the chain rule.

We denote matrix, vector, and scalar functions by  $F$ ,  $f$ , and  $\phi$ , respectively. Matrix (vector) arguments are written as  $X(x)$ . Vectorization ( $\text{vec}$ ) applies column-stacking, such that for matrices  $A, B, C$  of appropriate size

$$\text{vec}(ABC) = (C^\top \otimes A) \text{vec}(B). \quad (\text{S.1})$$

Whenever possible, we assign vectors to lower-case (for instance  $x, \theta$ ), matrices to upper-case ( $W, X, \dots$ ), and tensors to upper-case sans serif symbols ( $W, X, \dots$ ).  $\odot$  denotes elementwise multiplication (Hadamard product).

**Remark on vectorization:** The generalization of Jacobians and Hessians provided by Magnus and Neudecker (1999) relies on vectorization of matrices. Convolutional neural networks usually act on tensors and we incorporate these by assuming them to be flattened such that the first index varies fastest. For a matrix (tensor of order two), this is consistent with column-stacking. For instance, the vectorized version of the tensor  $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$  with  $n_1, n_2, n_3 \in \mathbb{N}$  is  $\text{vec } \mathbf{A} = (\mathbf{A}_{1,1,1}, \mathbf{A}_{2,1,1}, \dots, \mathbf{A}_{n_1,1,1}, \mathbf{A}_{1,2,1}, \dots, \mathbf{A}_{n_1,n_2,n_3})^\top$ . To formulate the generalized Jacobian or Hessian for tensor operations, its action on a vector or matrix view of the original tensor is considered. Consequently, all operations can be reduced to vector-valued functions, which we consider in the following.

The vectorization scheme is not unique. Most of the linear algebra literature assumes column-stacking. However, when it comes to implementations, a lot of programming languages store tensors in row-major order, corresponding to row-stacking vectorization (last index varies fastest). Thus, special attention has to be paid in implementations.

### A.1 Definitions

**Definition A.1** (Jacobian matrix). Let  $F : \mathbb{R}^{n \times q} \rightarrow \mathbb{R}^{m \times p}$ ,  $X \mapsto F(X)$  be a differentiable function mapping between two matrix-sized quantities. The *Jacobian*  $DF(X)$  of  $F$  with respect to  $X$  is an  $(mp \times nq)$  matrix

$$DF(X) = \frac{\partial \text{vec } F(X)}{\partial (\text{vec } X)^\top}, \quad \text{such that} \quad [DF(X)]_{i,j} = \frac{\partial [\text{vec } F(X)]_i}{\partial [(\text{vec } X)^\top]_j} \quad (\text{S.2})$$

(Magnus and Neudecker, 1999, Chapter 9.4).

In the context of FCNNs, the most common occurrences of Definition A.1 involve vector-to-vector functions  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ ,  $x \mapsto f(x)$  with

$$Df(x) = \frac{\partial f(x)}{\partial x^\top}.$$

For instance,  $x$  can be considered the input or bias vector of a layer applying an affine transformation. Other cases involve matrix-to-vector mappings  $f : \mathbb{R}^{n \times q} \rightarrow \mathbb{R}^m$ ,  $X \mapsto f(X)$  with

$$Df(X) = \frac{\partial f(X)}{\partial (\text{vec } X)^\top},$$

where  $X$  might correspond to the  $\mathbb{R}^{m \times q}$  weight matrix of an affine transformation.

Proper arrangement of the partial derivatives leads to a generalized formulation of the chain rule.

**Theorem A.1** (Chain rule for Jacobians). Let  $F : \mathbb{R}^{n \times q} \rightarrow \mathbb{R}^{m \times p}$  and  $G : \mathbb{R}^{m \times p} \rightarrow \mathbb{R}^{r \times s}$  be differentiable matrix-to-matrix mappings and their composition  $H$  be given by  $H = G \circ F : \mathbb{R}^{n \times q} \rightarrow \mathbb{R}^{r \times s}$ , then

$$DH(X) = [DG(F)] DF(X) \quad (\text{S.3})$$

(restricted from Magnus and Neudecker, 1999, Chapter 5.15).

Theorem A.1 is used to unroll the Jacobians in the composite structure of the feedforward neural network's loss function  $E \circ f^{(\ell)} \circ f^{(\ell-1)} \circ \dots \circ f^{(1)}$ , compare Subsection 3.1.

**Definition A.2** (Hessian). Let  $F : \mathbb{R}^{n \times q} \rightarrow \mathbb{R}^{m \times p}$  be a twice differentiable matrix function. The *Hessian*  $HF(X)$  is an  $(mpq \times nq)$  matrix defined by

$$HF(X) = D [DF(X)]^\top = \frac{\partial}{\partial (\text{vec } X)^\top} \text{vec} \left\{ \left[ \frac{\partial \text{vec } F(X)}{\partial (\text{vec } X)^\top} \right]^\top \right\} \quad (\text{S.4})$$

(Magnus and Neudecker, 1999, Chapter 10.2).

Forms of Equation (S.4) that are most likely to emerge in neural networks include

$$\mathbf{H}\phi(x) = \frac{\partial^2 \phi(x)}{\partial x^\top \partial x} \quad \text{and} \quad \mathbf{H}\phi(X) = \frac{\partial}{\partial (\text{vec } X)^\top} \frac{\partial \phi(X)}{\partial \text{vec } X}.$$

The scalar function  $\phi$  can be considered as the loss function  $E$ . Analogously, the Hessian matrix of a vector-in-vector-out function  $f$  reads

$$\mathbf{H}f(x) = \begin{pmatrix} \mathbf{H}f_1(x) \\ \vdots \\ \mathbf{H}f_m(x) \end{pmatrix}. \quad (\text{S.5})$$

Arranging the partial second derivatives in the fashion of Definition A.2 yields a direct generalization of the chain rule for second derivatives. We now provide this rule for a composition of vector-to-vector functions.

**Theorem A.2** (Chain rule for Hessian matrices). Let  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  and  $g : \mathbb{R}^m \rightarrow \mathbb{R}^p$  be twice differentiable and  $h = g \circ f : \mathbb{R}^n \rightarrow \mathbb{R}^p$ . The relation between the Hessian of  $h$  and the Jacobians and Hessians of the constituents  $f$  and  $g$  is given by

$$\mathbf{H}h(x) = [I_p \otimes \mathbf{D}f(x)]^\top [\mathbf{H}g(f)] \mathbf{D}f(x) + [\mathbf{D}g(f) \otimes I_n] \mathbf{H}f(x) \quad (\text{S.6})$$

(restricted from Magnus and Neudecker, 1999, Chapter 6.10).

## A.2 Relation to the modular approach

Theorem A.2 can directly be applied to the graph  $E \circ f^{(\ell)} \circ f^{(\ell-1)} \circ \dots \circ f^{(1)}$  of the feedforward net under investigation. Note that, for any module function  $f^{(i)}$ , the loss can be expressed as a composition of two functions by squashing preceding modules in the graph into a single function  $f^{(i-1)} \circ \dots \circ f^{(1)}$ , and likewise composing the module itself and all subsequent functions, i.e.  $E \circ f^{(\ell)} \circ \dots \circ f^{(i)}$ .

The analysis can therefore be reduced to the module shown in Figure 2 receiving an input  $x \in \mathbb{R}^n$  that is used to compute the output  $z \in \mathbb{R}^m$ . The scalar loss is then expressed as a mapping  $E(z(x), y) : \mathbb{R}^n \rightarrow \mathbb{R}^p$  with  $p = 1$ . Suppressing the label  $y$ , Equation (S.6) implies

$$\begin{aligned} \mathbf{H}E(x) &= [I_p \otimes \mathbf{D}z(x)]^\top [\mathbf{H}E(z)] \mathbf{D}z(x) + [\mathbf{D}E(z) \otimes I_n] \mathbf{H}z(x) \\ &= [\mathbf{D}z(x)]^\top [\mathbf{H}E(z)] \mathbf{D}z(x) + [\mathbf{D}E(z) \otimes I_n] \mathbf{H}z(x). \end{aligned} \quad (\text{S.7})$$

The HBP Equation (7) is obtained by substituting (S.5) into (S.7).

## B HBP for fully-connected neural networks

### B.1 Linear layer (matrix-vector multiplication, matrix-matrix multiplication, addition)

Consider the function  $f$  of a module applying an affine transformation to a vector. Apart from the input  $x$ , additional parameters of the module are given by the weight matrix  $W$  and the bias term  $b$ ,

$$\begin{aligned} f : \quad \mathbb{R}^n \times \mathbb{R}^{m \times n} \times \mathbb{R}^m &\rightarrow \mathbb{R}^m \\ (x, W, b) &\mapsto z = Wx + b. \end{aligned}$$

To compute the Jacobians with respect to each variable, we use the differentials

$$\begin{aligned} dz(x) &= W dx, \\ dz(b) &= db, \\ dz(W) &= (dW)x \implies d \text{vec } z(W) = (x^\top \otimes I_m) \text{vec}(dW), \end{aligned}$$

using Property (S.1) to establish the implication in the last line. With the *first identification tables* provided in Magnus and Neudecker (1999, Chapter 9.6), the Jacobians can be read off from the differentials as  $\mathbf{D}z(x) =$

$W, Dz(b) = I_m, Dz(W) = x^\top \otimes I_m$ . All second module derivatives  $\mathcal{H}z(x), \mathcal{H}z(W)$ , and  $\mathcal{H}z(b)$  vanish since  $f$  is linear in all inputs. Inserting the Jacobians into Equation (7) results in

$$\mathcal{H}x = W^\top \mathcal{H}zW, \quad (\text{S.8a})$$

$$\mathcal{H}b = \mathcal{H}z, \quad (\text{S.8b})$$

$$\mathcal{H}W = (x^\top \otimes I_m)^\top \mathcal{H}z (x^\top \otimes I_m) = xx^\top \otimes \mathcal{H}z = x \otimes x^\top \otimes \mathcal{H}z. \quad (\text{S.8c})$$

The HBP relations for matrix-vector multiplication and addition listed in Table 1 are special cases of Equation (S.8). HBP for matrix-matrix multiplication is derived in a completely analogous fashion.

## B.2 Elementwise activation

Next, consider the elementwise application of a nonlinear function,

$$\begin{aligned} \phi: \mathbb{R}^m &\rightarrow \mathbb{R}^m \\ x \mapsto z = \phi(x) &\text{ such that } \phi_k(x) = \phi(x_k), \end{aligned}$$

For the matrix differential with respect to  $x$ , this implies

$$d\phi(x) = \phi'(x) \odot dx = \text{diag}[\phi'(x)] dx,$$

and consequently, the Jacobian is given by  $D\phi(x) = \text{diag}[\phi'(x)]$ . For the Hessian, note that the function value  $\phi_k(x)$  only depends on  $x_k$  and thus  $\mathcal{H}\phi_k(x) = \phi''(x_k)e_k e_k^\top$ , with the one-hot unit vector  $e_k \in \mathbb{R}^m$  in coordinate direction  $k$ . Inserting all quantities into the Relation (7) results in

$$\begin{aligned} \mathcal{H}x &= \text{diag}[\phi'(x)] \mathcal{H}z \text{diag}[\phi'(x)] + \sum_k \phi''(x_k) e_k e_k^\top \delta z_k \\ &= \text{diag}[\phi'(x)] \mathcal{H}z \text{diag}[\phi'(x)] + \text{diag}[\phi''(x) \odot \delta z]. \end{aligned} \quad (\text{S.9})$$

## B.3 Skip-connection

Residual learning He et al. (2016) uses skip-connection units to facilitate the training of deep neural networks. In its simplest form, the mapping  $f: \mathbb{R}^m \rightarrow \mathbb{R}^m$  reads

$$z(x, \theta) = x + y(x, \theta),$$

with a potentially nonlinear operation  $(x, \theta) \mapsto y$ . The Jacobians with respect to the input and parameters are given by  $Dz(x) = I_m + Dy(x)$  and  $Dz(\theta) = Dy(\theta)$ . Using (7), one finds

$$\begin{aligned} \mathcal{H}x &= [I_m + Dy(x)]^\top \mathcal{H}z [I_m + Dy(x)] + \sum_k [Hy_k(x)] \delta z_k, \\ \mathcal{H}\theta &= [Dy(\theta)]^\top \mathcal{H}z [Dy(\theta)] + \sum_k [Hy_k(\theta)] \delta z_k. \end{aligned}$$

## B.4 Relation to recursive schemes in previous work

The modular decomposition of curvature backpropagation facilitates the analysis of modules composed of multiple operations. Now, we analyze the composition of two modules. This yields the recursive schemes presented by Botev et al. (2017) and Chen et al. (2018), referred to as KFRA and BDA-PCH, respectively.

### B.4.1 Analytic composition of multiple modules

Consider the module  $g = f \circ \phi, x \mapsto y = \phi(x), y \mapsto z = f(y(x))$ . We assume  $\phi$  to act elementwise on the input, followed by a linear layer  $f: z(y) = Wy + b$  (Figure S.1a). Analytic elimination of the intermediate backward pass yields a single module composed of two operations (Figure S.1b).

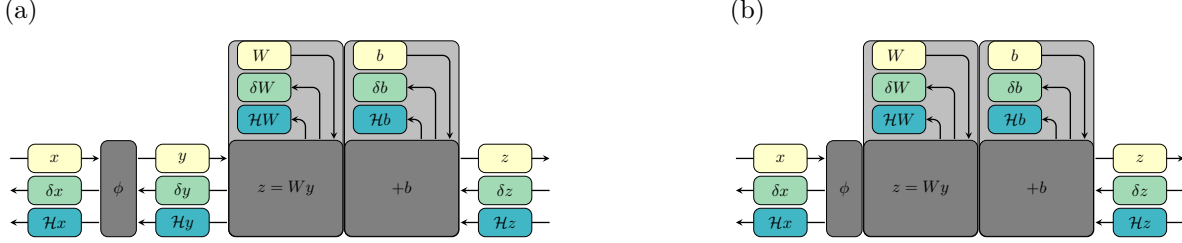


Figure S.1: Composition of elementwise activation  $\phi$  and linear layer. (a) Both operations are analyzed separately to derive the HBP. (b) Backpropagation of  $\mathcal{H}z$  is expressed in terms of  $\mathcal{H}x$  without intermediate message.

The first Hessian backward pass through the linear module  $f$  (Equations (S.8)) implies

$$\mathcal{H}y = W^\top \mathcal{H}z W, \quad (\text{S.10a})$$

$$\mathcal{H}W = y \otimes y^\top \otimes \mathcal{H}z = \phi(x) \otimes \phi(x)^\top \otimes \mathcal{H}z, \quad (\text{S.10a})$$

$$\mathcal{H}b = \mathcal{H}z. \quad (\text{S.10b})$$

Further backpropagation through  $\phi$  by means of Equation (S.9) results in

$$\begin{aligned} \mathcal{H}x &= \text{diag}[\phi'(x)] \mathcal{H}y \text{diag}[\phi'(x)] + \text{diag}[\phi''(x) \odot \delta y] \\ &= \text{diag}[\phi'(x)] [W^\top \mathcal{H}z W] \text{diag}[\phi'(x)] + \text{diag}[\phi''(x) \odot W^\top \delta z] \\ &= \{W \text{diag}[\phi'(x)]\}^\top \mathcal{H}z \{W \text{diag}[\phi'(x)]\} + \text{diag}[\phi''(x) \odot W^\top \delta z]. \end{aligned} \quad (\text{S.10c})$$

We use the invariance of a diagonal matrix under transposition and  $\delta y = W^\top \delta z$  for the backpropagated gradient to establish the last equality. The Jacobian  $Dg(x)$  of the module shown in Figure S.1b is  $Dg(x) = W \text{diag}[\phi(x)] = [W^\top \odot \phi'(x)]^\top$ . In summary, the HBP relation for the composite layer  $z(x) = W\phi(x) + b$  is given by (S.10).

#### B.4.2 Obtaining the relations of KFRA and BDA-PCH

The derivations for the composite module given above are closely related to the recursive schemes of Botev et al. (2017); Chen et al. (2018). Their relations are obtained from a straightforward conversion of the HBP rules (S.10). Consider a sequence of a linear layer  $f^{(1)}$  and multiple composite modules  $f^{(2)}, \dots, f^{(\ell)}$  as shown in Figure S.2.

According to Equation (S.10b) both the linear layer and the composite  $f^{(i)}$  identify the gradient (Hessian) with respect to their outputs,  $\delta z^{(i)}$  ( $\mathcal{H}z^{(i)}$ ), as the gradient (Hessian) with respect to their bias term,  $\delta b^{(i)}$  ( $\mathcal{H}b^{(i)}$ ). Introducing layer indices for all quantities, one finds the recursion

$$\mathcal{H}b^{(i)} = \mathcal{H}z^{(i)}, \quad (\text{S.11a})$$

$$\mathcal{H}W^{(i)} = \phi(z^{(i-1)}) \otimes \phi(z^{(i-1)})^\top \otimes \mathcal{H}b^{(i)}, \quad (\text{S.11b})$$

for  $i = \ell - 1, \dots, 1$ , and

$$\begin{aligned} \mathcal{H}z^{(i-1)} &= \left\{ W^{(i)} \text{diag}[\phi'(z^{(i-1)})] \right\}^\top \mathcal{H}b^{(i)} \left\{ W^{(i)} \text{diag}[\phi'(z^{(i-1)})] \right\} + \text{diag}[\phi''(z^{(i-1)}) \odot W^{(i)\top} \delta b^{(i)}] \\ &= \left\{ W^{(i)\top} \odot \phi'(z^{(i-1)}) \right\} \mathcal{H}b^{(i)} \left\{ W^{(i)\top} \odot \phi'(z^{(i-1)}) \right\}^\top + \text{diag}[\phi''(z^{(i-1)}) \odot W^{(i)\top} \delta b^{(i)}] \end{aligned} \quad (\text{S.11c})$$

for  $i = \ell - 1, \dots, 2$ . It is initialized with the gradient  $\delta z^{(\ell)}$  and Hessian  $\mathcal{H}z^{(\ell)}$  of the loss function.

Equations (S.11) are equivalent to the expressions provided in Botev et al. (2017); Chen et al. (2018). Their emergence from compositions of HBP equations of simple operations represents one key insight of this paper. Both works use the batch average strategy presented in Subsection 3.2 to obtain curvature estimates.

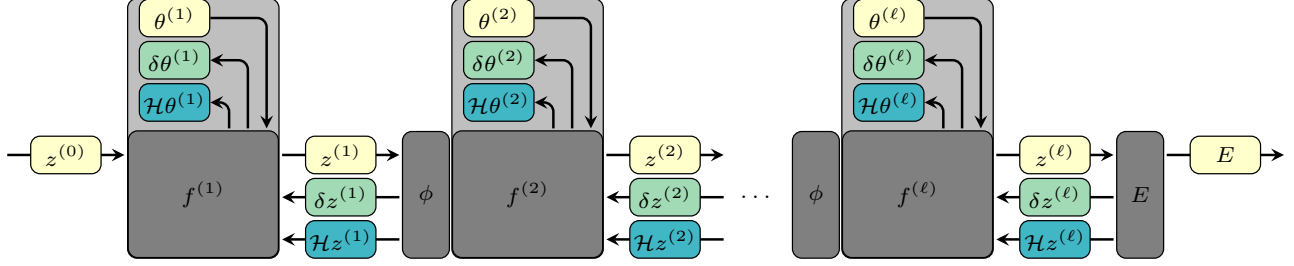


Figure S.2: Grouping scheme for the recursive Hessian computation proposed by KFRA and BDA-PCH. The backward messages between the linear layer and the preceding nonlinear activation are analytically absorbed.

## C HBP for loss functions

### C.1 Square loss

Square loss of the model prediction  $x \in \mathbb{R}^m$  and the true label  $y \in \mathbb{R}^m$  is computed by

$$E(x, y) = (y - x)^\top (y - x).$$

Differentiating  $dE(x) = -(dx)^\top (y - x) - (y - x)^\top dx = -2(y - x)^\top dx$  once more yields

$$d^2E(x) = 2(dx)^\top dx = 2(dx)^\top I_m dx.$$

The Hessian is extracted with the *second identification tables* from Magnus and Neudecker (1999, Chapter 10.4) and reproduces the expected result

$$HE(x) = \mathcal{H}x = 2I_m. \quad (\text{S.12})$$

### C.2 Softmax cross-entropy loss

The computation of cross-entropy from logits is composed of two operations. First, the neural network outputs are transformed into log-probabilities by the softmax function. Then, the cross-entropy with the label is computed.

**Log-softmax:** The output's elements  $x \in \mathbb{R}^m$  of a neural network are assigned to log-probabilities  $z(x) = \log[p(x)] \in \mathbb{R}^m$  by means of the softmax function  $p(x) = \exp(x)/\sum_i \exp(x_i)$ . Consequently,

$$z(x) = x - \log \left[ \sum_i \exp(x_i) \right],$$

and the Jacobian reads  $Dz(x) = I_m - p(x)\mathbf{1}_m^\top$  with  $\mathbf{1}_m^\top \in \mathbb{R}^m$  denoting a vector of ones. Moreover, the log-probability Hessians with respect to  $x$  are given by  $\mathcal{H}z_k(x) = -\text{diag}[p(x)] + p(x)p(x)^\top$ .

**Cross-entropy:** The negative log-probabilities are used to compute the cross-entropy with the probability distribution of the label  $y \in \mathbb{R}^m$  with  $\sum_k y_k = 1$  (usually a one-hot vector),

$$E(z, y) = -y^\top z.$$

Since  $E$  is linear in the log-probabilities, that is  $HE(z) = 0$ , the HBP is given by

$$\begin{aligned} \mathcal{H}x &= [Dz(x)]^\top HE(z) [Dz(x)] + \sum_k \mathcal{H}z_k(x) \frac{\partial E(z)}{\partial z_k} \\ &= \{-\text{diag}[p(x)] + p(x)p(x)^\top\} \sum_k (-y_k) \\ &= \text{diag}[p(x)] - p(x)p(x)^\top. \end{aligned}$$

## D HBP for convolutional neural networks

The recursive approaches presented in Botev et al. (2017); Chen et al. (2018) tackle the computation of curvature blocks for FCNNs. To the best of our knowledge, an extension to CNNs has not been achieved so far. One reason this might be is that convolutions come with heavy notation that is difficult to deal with in index notation.

Martens and Grosse (2015) provide a procedure for computing a Kronecker-factored approximation of the Fisher for convolutions (KFC). This scheme relies on the property of the Fisher to describe the covariance of the log-likelihood’s gradients under the model’s distribution. Curvature information is thus encoded in the expectation value, and not by backpropagation of second-order derivatives.

To derive the HBP for convolutions, we use the fact that an efficient implementation of the forward pass is decomposed into multiple operations (see Figure S.4), which can be considered independently by means of our modular approach (see Figure S.5 for details). Our analysis starts by considering the backpropagation of curvature through operations that occur frequently in CNNs. This includes the reshape (Subsection D.1) and extraction operation (Subsection D.2). In Subsection D.3 we outline the modular decomposition and curvature backpropagation of convolution for the two-dimensional case. The approach carries over to other dimensions.

All operations under consideration in this Section are linear. Hence the second terms in Equation (7) vanish. Again, we use the framework of matrix differential calculus (Magnus and Neudecker, 1999) to avoid index notation.

### D.1 Reshape/view

The reshape operation reinterprets a tensor  $\mathbf{X} \in \mathbb{R}^{n_1 \times \dots \times n_x}$  as another tensor  $\mathbf{Z} \in \mathbb{Z}^{m_1 \times \dots \times m_z}$

$$\mathbf{Z}(\mathbf{X}) = \text{reshape}(\mathbf{X}),$$

which possesses the same number of elements, that is  $\prod_i n_i = \prod_i m_i$ . One example is given by the `vec` operation. It corresponds to a reshape into a tensor of order one. As the arrangement of elements remains unaffected,  $\text{vec } \mathbf{Z} = \text{vec } \mathbf{X}$ , and reshaping corresponds to the identity map on the vectorized input. Consequently, one finds (remember that  $\mathcal{H}\mathbf{X}$  is a shorthand notation for  $\mathcal{H} \text{vec } \mathbf{X}$ )

$$\mathcal{H}\mathbf{X} = \mathcal{H}\mathbf{Z}.$$

### D.2 Index select/map

Whenever elements of a tensor are selected by an operation, it can be described as a matrix-vector multiplication of a binary matrix  $\mathbf{\Pi}$  and the vectorized tensor. The mapping is described by an index map  $\pi$ . Element  $j$  of the output  $z \in \mathbb{R}^m$  is selected as element  $\pi(j)$  from the input  $x \in \mathbb{R}^n$ . Only elements  $\Pi_{j,\pi(j)}$  in the selection matrix  $\mathbf{\Pi} \in \mathbb{R}^{m \times n}$  are one, while all other entries vanish. Consequently, index selection can be expressed as

$$z_j = x_{\pi(j)} \Leftrightarrow z(x) = \mathbf{\Pi}x \quad \text{with} \quad \Pi_{j,\pi(j)} = 1.$$

The HBP is equivalent to the linear layer discussed in Subsection B.1,

$$\mathcal{H}z = \mathbf{\Pi}^\top (\mathcal{H}x) \mathbf{\Pi}.$$

Applications include max-pooling and the `im2col/unfold` operation (see Subsection D.3). Average-pooling represents a weighted sum of index selections and can be treated analogously.

### D.3 Convolution

The convolution operation acts on local patches of a multi-channel input of sequences, images, or volumes. In the following, we restrict the discussion to two-dimensional convolution. Figure S.3a illustrates the setting. A collection of filter maps, the *kernel*  $\mathbf{W}$ , is slid over the spatial coordinates of the input tensor  $\mathbf{X}$ . In each step, the kernel is contracted with the current area of overlap (the *patch*).

Both the sliding process as well as the structure of the patch area can be controlled by hyperparameters of the operation (kernel size, stride, dilation). Moreover, it is common practice to extend the input tensor, for instance by zero-padding (for an introduction to the arithmetics of convolutions, see Dumoulin and Visin, 2016). The approach presented here is not limited to a certain choice of convolution hyperparameters.

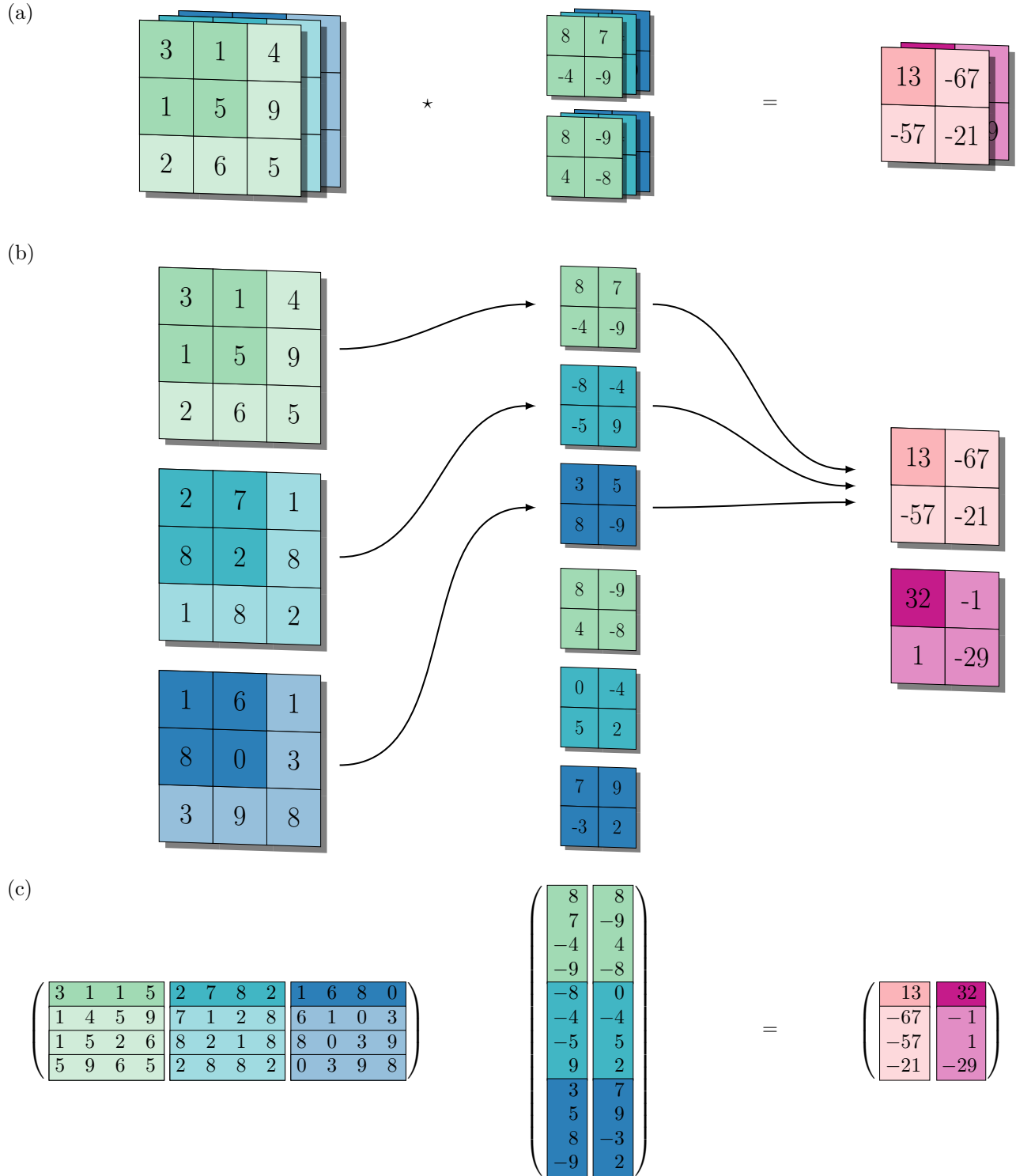


Figure S.3: Two-dimensional convolution  $Y = X \star W$  without bias term. (a) The input  $X$  consists of  $C_{in} = 3$  channels (different colors) of  $(3 \times 3)$  images. Filter maps of size  $(2 \times 2)$  are provided by the kernel  $W$  for the generation of  $C_{out} = 2$  output channels. Patch and kernel volumes that are contracted in the first step of the convolution are highlighted. Assuming no padding and a stride of one results in four patches. New features  $Y$  consist of  $C_{out} = 2$  channels of  $(2 \times 2)$  images. (b) Detailed view. All tensors are unrolled along the first axis. (c) Convolution as matrix multiplication. From left to right, the matrices  $[[X]]^T, W^T$ , and  $Y^T$  are shown.



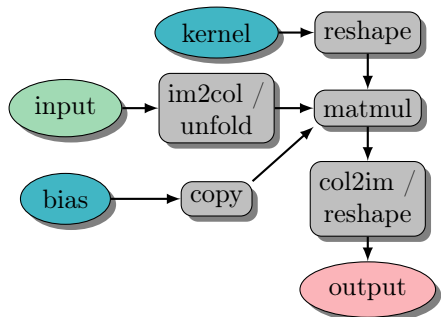


Figure S.4: Decomposition of the convolution operation’s forward pass.

Table S.1: Important quantities for the convolution operation. The number of patches equals the product of the outputs’ spatial dimensions, i.e.  $P = Y_1 Y_2$ .

Tensor	Dimensionality	Description
$X$	$(C_{\text{in}}, X_1, X_2)$	Input
$W$	$(C_{\text{out}}, C_{\text{in}}, K_1, K_2)$	Kernel
$Y$	$(C_{\text{out}}, Y_1, Y_2)$	Output
$\llbracket X \rrbracket$	$(C_{\text{in}} K_1 K_2, P)$	Expanded input
$W$	$(C_{\text{out}}, C_{\text{in}} K_1 K_2)$	Matricized kernel
$Y$	$(C_{\text{out}}, P)$	Matricized output
$b$	$C_{\text{out}}$	Bias vector
$B$	$(C_{\text{out}}, P)$	Bias matrix

### D.3.1 Forward pass and notation

We now introduce the quantities involved in the process along with their dimensions. For a summary, see Table S.1. A forward pass of convolution proceeds as follows (cf. Figure S.3b for an example):

- The input  $X$ , a tensor of order three, stores a collection of two-dimensional data. Its components  $X_{c_{\text{in}}, x_1, x_2}$  are referenced by indices for the channel  $c_{\text{in}}$  and the spatial location  $(x_1, x_2)$ .  $C_{\text{in}}, X_1, X_2$  denote the input channels, width, and height of the image, respectively.
- The kernel  $W$  is a tensor of order four with dimensions  $(C_{\text{out}}, C_{\text{in}}, K_1, K_2)$ . Kernel width  $K_1$  and height  $K_2$  determine the patch size  $P = K_1 K_2$  for each channel. New features are obtained by contracting the patch and kernel. This is repeated for a collection of  $C_{\text{out}}$  output channels stored along the first axis of  $W$ .
- Each output channel  $c_{\text{out}}$  is shifted by a bias  $b_{c_{\text{out}}}$ , stored in the  $C_{\text{out}}$ -dimensional vector  $b$ .
- The output  $Y = X \star W$  with components  $Y_{c_{\text{out}}, y_1, y_2}$  is of the same structure as the input. We denote the spatial dimensions of  $Y$  by  $Y_1, Y_2$ , respectively. Hence  $Y$  is of dimension  $(C_{\text{out}}, Y_1, Y_2)$ .

**Example (index notation):** A special case where input and output have the same spatial dimensions (Grosse and Martens, 2016) uses a stride of one, kernel widths  $K_1 = K_2 = 2K + 1$ , ( $K \in \mathbb{N}$ ), and padding  $K$ . Elements of the filter  $W_{c_{\text{out}}, c_{\text{in}}, :, :}$  are addressed with the index set  $\{-K, \dots, 0, \dots, K\} \times \{-K, \dots, 0, \dots, K\}$ . In this setting,

$$Y_{c_{\text{out}}, y_1, y_2} = \sum_{k_1=-K}^K \sum_{k_2=-K}^K X_{c_{\text{in}}, x_1+k_1, x_2+k_2} W_{c_{\text{out}}, c_{\text{in}}, k_1, k_2} + b_{c_{\text{out}}}. \quad (\text{S.13})$$

Elements of  $X$  addressed out of bounds evaluate to zero. Arbitrary convolutions come with even heavier notation.

**Convolution by matrix multiplication:** Evaluating convolutions by sums of the form (S.13) leads to poor memory locality (Grosse and Martens, 2016). For improved performance, the computation is mapped to a matrix multiplication (Chellapilla et al., 2006). To do so, patches of the input  $X$  are extracted and flattened into columns of a matrix. The patch extraction is indicated by the operator  $\llbracket \cdot \rrbracket$  and the resulting matrix  $\llbracket X \rrbracket$  is of dimension  $(C_{\text{in}} K_1 K_2 \times P)$  (cf. left part of Figure S.3c showing  $\llbracket X \rrbracket^\top$ ). In other words, elements contracted with the kernel are stored along the first axis of  $\llbracket X \rrbracket$ .  $\llbracket \cdot \rrbracket$  is also referred to as im2col or unfold operation<sup>1</sup>, and accounts for padding.

The kernel tensor  $W$  is reshaped into a  $(C_{\text{out}} \times C_{\text{in}} K_1 K_2)$  matrix  $W$ , and elements of the bias vector  $b$  are copied columnwise into a  $(C_{\text{out}} \times P)$  matrix  $B = b \mathbf{1}_P^\top$ , where  $\mathbf{1}_P$  is a  $P$ -dimensional vector of ones. Patchwise contractions are carried out by matrix multiplication and yield a matrix  $Y$  of shape  $(C_{\text{out}}, P)$  with  $P = Y_1 Y_2$ ,

$$Y = W \llbracket X \rrbracket + B \quad (\text{S.14})$$

(Figure S.3c shows the quantities  $W^\top$ ,  $\llbracket X \rrbracket^\top$ , and  $Y$  from left to right). Reshaping  $Y$  into a  $(C_{\text{out}}, Y_1, Y_2)$  tensor, referred to as col2im, yields  $Y$ . Figure S.4 summarizes the outlined decomposition of the forward pass.

<sup>1</sup>Our definition of the unfold operator slightly differs from Grosse and Martens (2016), where flattened patches are stacked rowwise. This lets us achieve an analogous form to a linear layer. Conversion is achieved by transposition.

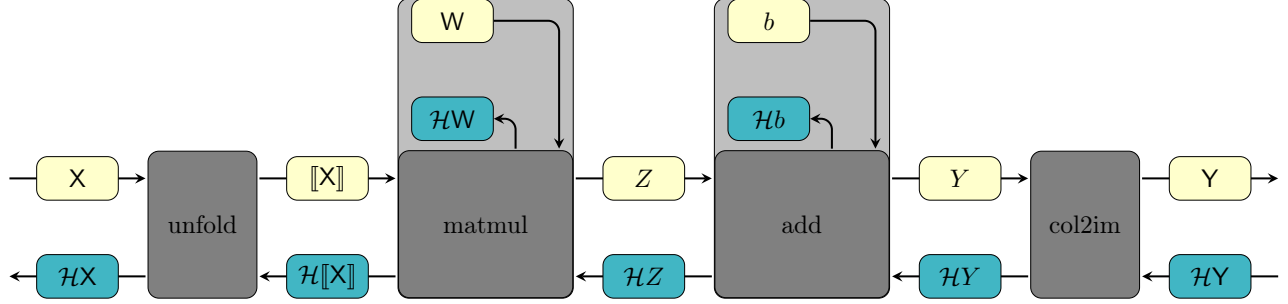


Figure S.5: Decomposition of convolution with notation for the study of curvature backpropagation.

#### D.4 HBP for convolution

We now compose the HBP for convolution, proceeding from right to left with the operations depicted in Figure S.5, by analyzing the backpropagation of curvature for each module, adopting the figure’s notation.

**Col2im/reshape:** The col2im operation takes a matrix  $Y \in \mathbb{R}^{C_{\text{out}} \times Y_1 Y_2}$  and reshapes it into the tensor  $Y \in \mathbb{R}^{C_{\text{out}} \times Y_1 \times Y_2}$ . According to Subsection D.1,  $\mathcal{H}Y = \mathcal{H}Y$ .

**Bias Hessian:** Forward pass  $Y = Z + B$  and Equation (S.8b) imply  $\mathcal{H}Y = \mathcal{H}B = \mathcal{H}Z$ . To obtain the Hessian with respect to  $b$  from  $\mathcal{H}B$ , consider the columnwise copy operation  $B(b) = b \mathbf{1}_P^\top$ , whose matrix differential is  $dB(b) = (db) \mathbf{1}_P^\top$ . Vectorization yields  $d \text{vec } B(b) = (\mathbf{1}_P \otimes I_{C_{\text{out}}}) db$ . Hence, the Jacobian is  $DB(b) = \mathbf{1}_P \otimes I_{C_{\text{out}}}$ , and insertion into Equation (7) results in

$$\mathcal{H}b = (\mathbf{1}_P \otimes I_{C_{\text{out}}})^\top \mathcal{H}B (\mathbf{1}_P \otimes I_{C_{\text{out}}}) .$$

This performs a linewise and columnwise summation over  $\mathcal{H}B$ , summing entities that correspond to copies of the same entry of  $b$  in the matrix  $B$ . It can also be regarded as a reshape of  $\mathcal{H}B$  into a  $(C_{\text{out}}, P, C_{\text{out}}, P)$  tensor, which is then contracted over the second and fourth axis.

**Weight Hessian:** For the matrix-matrix multiplication  $Z(W, \llbracket X \rrbracket) = W \llbracket X \rrbracket$ , the HBP was discussed in Subsection B.1. The Jacobians are given by  $DZ(\llbracket X \rrbracket) = I_P \otimes W$  and  $DZ(W) = \llbracket X \rrbracket^\top \otimes I_S$  with the patch size  $S = C_{\text{in}} K_1 K_2$ . Hence, the HBP for the unfolded input and the weight matrix are

$$\begin{aligned} \mathcal{H}\llbracket X \rrbracket &= (I_P \otimes W)^\top \mathcal{H}Z (I_P \otimes W) , \\ \mathcal{H}W &= (\llbracket X \rrbracket^\top \otimes I_S)^\top \mathcal{H}Z (\llbracket X \rrbracket^\top \otimes I_S) . \end{aligned}$$

From what has been said about the reshape operation in Subsection D.1, it follows that  $\mathcal{H}W = \mathcal{H}W$ .

**Im2col/unfold:** The patch extraction operation  $\llbracket \cdot \rrbracket$  copies all elements in a patch into the columns of a matrix and thus represents a selection of elements by an index map, which is hard to express in notation. Numerically, it can be obtained by calling im2col on a  $(C_{\text{in}}, X_1, X_2)$  index tensor whose entries correspond to the indices. The resulting tensor contains all information about the index map. HBP follows the relation of Subsection D.2.

**Discussion:** Although convolution can be understood as a matrix multiplication, the parameter Hessian is not identical to that of the linear layer discussed in Subsection B.1. The difference is due to the parameter sharing of the convolution. In the case of a linear layer  $z = Wx + b$ , the Hessian of the weight matrix for a single sample possesses Kronecker structure (Botev et al., 2017; Chen et al., 2018; Bakker et al., 2018), i.e.  $\mathcal{H}W = x \otimes x^\top \otimes \mathcal{H}z$ . For convolution layers, however, it has been argued by Bakker et al. (2018) that block diagonals of the Hessian do not inherit the same structure. Rephrasing the forward pass (S.14) in terms of vectorized quantities, we find

$$\text{vec } Y = (I_P \otimes W) \text{vec} \llbracket X \rrbracket + \text{vec } B .$$

In this perspective, convolution corresponds to a fully-connected linear layer, with the additional constraints that the weight and bias matrix be circular. Defining  $\hat{W} = I_P \otimes W$ , one then finds the Hessian  $\mathcal{H}\hat{W}$  to possess Kronecker structure. Parameterization with a kernel tensor encodes the circularity constraint in weight sharing.

Table S.2: Model architectures under consideration. We use the patterns Linear(in\_features, out\_features), Conv2d(in\_channels, out\_channels, kernel\_size, padding), MaxPool2d(kernel\_size, stride), and ZeroPad2d(padding\_left, padding\_right, padding\_top, padding\_bottom) to describe module hyperparameters. Convolution strides are always chosen to be one. (a) FCNN used to extend the experiment in Chen et al. (2018) (3 846 810 parameters). (b) CNN architecture (1 099 226 parameters). (c) DeepOBS c3d3 test problem with three convolutional and three dense layers (895 210 parameters). ReLU activation functions are replaced by sigmoids.

(a)	FCNN (Figure 4)	(b)	CNN (Figure 5)	(c)	DeepOBS c3d3 (Figure S.6)
#	Module	#	Module	#	Module
1	Flatten	1	Conv2d(3, 16, 3, 1)	1	Conv2d(3, 64, 5, 0)
2	Linear(3072, 1024)	2	Sigmoid	2	Sigmoid
3	Sigmoid	3	Conv2d(16, 16, 3, 1)	3	ZeroPad2d(0, 1, 0, 1)
4	Linear(1024, 512)	4	Sigmoid	4	MaxPool2d(3, 2)
5	Sigmoid	5	MaxPool2d(2, 2)	5	Conv2d(64, 96, 3, 0)
6	Linear(512, 256)	6	Conv2d(16, 32, 3, 1)	6	Sigmoid
7	Sigmoid	7	Sigmoid	7	ZeroPad2d(0, 1, 0, 1)
8	Linear(256, 128)	8	Conv2d(32, 32, 3, 1)	8	MaxPool2d(3, 2)
9	Sigmoid	9	Sigmoid	9	ZeroPad2d(1, 1, 1, 1)
10	Linear(128, 64)	10	MaxPool2d(2, 2)	10	Conv2d(96, 128, 3, 0)
11	Sigmoid	11	Flatten	11	Sigmoid
12	Linear(64, 32)	12	Linear(2048, 512)	12	ZeroPad2d(0, 1, 0, 1)
13	Sigmoid	13	Sigmoid	13	MaxPool2d(3, 2)
14	Linear(32, 16)	14	Linear(512, 64)	14	Flatten
15	Sigmoid	15	Sigmoid	15	Linear(1152, 512)
16	Linear(16, 10)	16	Linear(64, 10)	16	Sigmoid
				17	Linear(512, 256)
				18	Sigmoid
				19	Linear(256, 10)

For the Hessian  $\mathcal{H}W$  of the kernel to possess Kronecker structure, the output Hessian  $\mathcal{H}Z$  has to be assumed to factorize into a Kronecker product of  $S \times S$  and  $C_{\text{out}} \times C_{\text{out}}$  matrices. These assumptions are somewhat in parallel with the additional approximations introduced in Grosse and Martens (2016) to obtain KFC.

## E Experimental details

**Fully-connected neural network:** The same model as in Chen et al. (2018) (see Table S.2a) is used to extend the experiment performed therein. The weights of each linear layer are initialized with the Xavier method of Glorot and Bengio (2010). Bias terms are initialized to zero. Backpropagation of the Hessian uses approximation (9) of (S.11) to compute the curvature blocks of the weights and bias,  $\overline{\mathcal{H}W^{(i)}}$  and  $\overline{\mathcal{H}b^{(i)}}$ .

Hyperparameters are chosen as follows to obtain consistent results with the original work: All runs shown in Figure 4 use a batch size of  $|B| = 500$ . For SGD, the learning rate is assigned to  $\gamma = 0.1$  with momentum  $v = 0.9$ . Block-splitting experiments with the second-order method use the PCH-abs. All runs were performed with a learning rate  $\gamma = 0.1$  and a regularization strength of  $\alpha = 0.02$ . For the convergence criterion of CG, the maximum number of iterations is restricted to  $n_{\text{CG}} = 50$ ; convergence is reached at a relative tolerance  $\epsilon_{\text{CG}} = 0.1$ .

**Convolutional neural net:** The training procedure of the CNN architecture shown in Table S.2b is evaluated on a hyperparameter grid. Runs with smallest final training loss are selected to rerun on different random seeds. The curves shown in Figure 5b represent mean values and standard deviations for ten different realizations over the random seed. All layer parameters were initialized with the default method in PyTorch.

For the first-order optimizers (SGD, Adam), we considered batch sizes  $B \in \{100, 200, 500\}$ . In the case of SGD, momentum  $v$  was tuned over the set  $\{0, 0.45, 0.9\}$ . Although we varied the learning rate over a large range of values  $\gamma \in \{10^{-3}, 10^{-2}, 0.1, 1, 10\}$ , losses kept plateauing and did not decrease. In particular, the loss function even

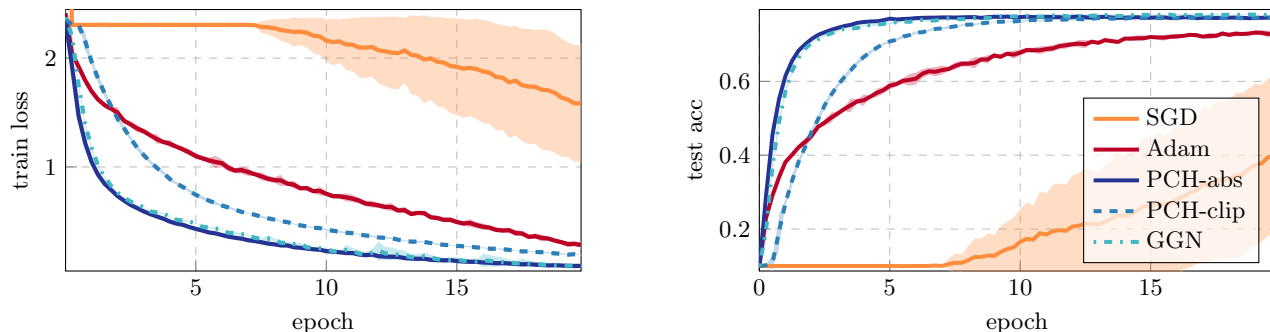


Figure S.6: Comparison of SGD, Adam, and Newton-style methods with different exact curvature matrix-vector products (HBP) on the DeepOBS c3d3 network (Schneider et al., 2019) with sigmoid activations (Table S.2c).

increased for the large learning rates. For Adam, we only vary the learning rate  $\gamma \in \{10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10\}$ .

As second-order methods scale better to large batch sizes, we considered  $B \in \{200, 500, 1000\}$  for them. The convergence parameters for CG were fixed to  $n_{CG} = 200$  and  $\epsilon_{CG} = 0.1$ . For all curvature matrices, we varied the learning rates over the grid  $\gamma \in \{0.05, 0.1, 0.2\}$  and  $\alpha \in \{10^{-4}, 10^{-3}, 10^{-2}\}$ .

In order to compare with another second-order method, we experimented with a public PyTorch implementation<sup>2</sup> of the KFAC optimizer (Martens and Grosse, 2015; Grosse and Martens, 2016). All hyperparameters were kept at their default setting. The learning rate was varied over  $\gamma \in \{10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10\}$ .

The hyperparameters of results shown in Figure 5 read as follows:

- SGD ( $|B| = 100, \gamma = 10^{-3}, v = 0.9$ ). The particular choice of these hyperparameters is artificial. This run is representative for SGD, which does not achieve any progress at all.
- Adam ( $|B| = 100, \gamma = 10^{-3}$ )
- KFAC ( $|B| = 500, \gamma = 0.1$ )
- PCH-abs ( $|B| = 1000, \gamma = 0.2, \alpha = 10^{-3}$ ), PCH-clip ( $|B| = 1000, \gamma = 0.1, \alpha = 10^{-4}$ )
- GGN,  $\alpha_1$  ( $|B| = 1000, \gamma = 0.1, \alpha = 10^{-4}$ ). This run does not yield the minimum training loss on the grid, and is shown to illustrate that the second-order method is capable to escape the flat regions in early stages.
- GGN,  $\alpha_2$  ( $|B| = 1000, \gamma = 0.1, \alpha = 10^{-3}$ ). In comparison with  $\alpha_1$ , the second-order method requires more iterations to escape the initial plateau, caused by the larger regularization strength. However, this leads to improved robustness against noise in later stages of the training procedure.

**Additional experiment:** Another experiment conducted with HBP considers the c3d3 architecture (Table S.2c) of DeepOBS (Schneider et al., 2019) on CIFAR-10. ReLU activations are replaced by sigmoids to make the problem more challenging. The hyperparameter grid is chosen identically to the CNN experiment above, and results are summarized in Figure S.6. In particular, the hyperparameter settings for each competitor are:

- SGD ( $|B| = 100, \gamma = 1, v = 0$ )
- Adam ( $|B| = 100, \gamma = 10^{-3}$ )
- PCH-abs ( $|B| = 500, \gamma = 0.1, \alpha = 10^{-3}$ ), PCH-clip ( $|B| = 500, \gamma = 0.1, \alpha = 10^{-2}$ )
- GGN ( $|B| = 500, \gamma = 0.05, \alpha = 10^{-3}$ )

<sup>2</sup><https://github.com/alecwangcq/KFAC-Pytorch>

**References**

- Bakker, C., Henry, M. J., and Hodas, N. O. (2018). The outer product structure of neural network derivatives. *CoRR*, abs/1810.03798.
- 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.
- Chellapilla, K., Puri, S., and Simard, P. (2006). High Performance Convolutional Neural Networks for Document Processing. In Lorette, G., editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France). Université de Rennes 1, Suvisoft.
- 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.
- Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.
- Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pages 249–256.
- 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.
- He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778.
- Magnus, J. R. and Neudecker, H. (1999). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Probabilistics and Statistics. Wiley.
- 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.
- Schneider, F., Balles, L., and Hennig, P. (2019). DeepOBS: A deep learning optimizer benchmark suite. In *International Conference on Learning Representations*.