Backpropagation Beyond the Gradient

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Felix Julius Dangel, M. Sc.
aus Stuttgart

Tübingen
2022

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	31. Januar 2023
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Philipp Hennig
2. Berichterstatter:	Prof. Dr. Hendrik Lensch
3. Berichterstatterin:	Prof. Dr. Asja Fischer

Disclaimer: this thesis uses Federico Marotta's `kaobook` template based on Ken Arroyo Ohori's doctoral thesis. I am grateful to Frank Schneider who added many stylistic features and would like to thank him, Katharina Ott, Frederik Kunstner, Julia Grosse, Jonathan Wenger, Matthias Werner, Marvin Pförtner, Runa Eschenhagen, and Nathanel Bosch for their help to improve the text.

Acknowledgments

I want to express my deepest gratitude to Philipp Hennig for his guidance that allowed me to seamlessly change fields for the Ph.D. I am inspired by your passion for research, and grateful for your advice and everything I have learned from you. Thank you for allowing me to travel the world, for spontaneously stepping in on poster presentations, and for taking care of rebellious coffee machines.

Also, I am grateful to Asja Fischer and Hendrik Lensch who invested their time and effort into reviewing this thesis, as well as Zeynep Akata and Georg Martius for serving on my thesis advisory committee and forming my examination committee.

I want to thank Franziska Weiler (Methods of Machine Learning (MoML) group), Leila Masri & Sara Sorce (International Max Planck Research School for Intelligent Systems) for their effective administrative support.

This work was made possible by great collaborations. I want to thank Stefan Harmeling, Frederik Kunstner, Frank Schneider, and Lukas Tatzel for these experiences and their patience with me. I very much appreciated our regular coffee breaks, coding sessions, "grammar police" discussions, and "random questions of today". This also applies more broadly to the entire MoML group that provided a productive and friendly work environment. I consider myself lucky to have met all of you (apologies for the regular mess in the office kitchen). I also enjoyed supervising Christian Meier, Paul Fischer, Shrisha Bharadwaj, Tim Schäfer, Jessica Bader, and Elisabeth Knigge. Thank you for contributing to this work; hopefully you learned as much as I did.

During my time in Tübingen, I was fortunate enough to be surrounded by amazing people. I want to express my gratitude to Jack Brady, Nicolò Zottino, Ori Press, Emilia Magnani, Laura Iacovissi, Georg Hafner, Massimiliano Mancini, Ilya Petrov, Faezeh Zakeri, Wafaa Mohammed, Martina Contisciani, Alessandro Lonardi, Nicolò Ruggeri, Diego Theuerkauf, Ahmad Gazar, and everyone I forgot to list here. You made this a really great time.

Thank you to Jan Koschke, Marcel Bauer, and Jens Hollmann for being great friends, and your support during—and long before—my studies. Please do not ever call me Doctor D. (you know what I mean. . .).

I am grateful to my family, especially my parents, my sister, and my grandparents (happy birthday, Oma!). They sparked my curiosity about science, trusted in me, and provided me with everything I could hope for throughout the years. Eu também mando um abraço forte à minha segunda família brasileira. Dona Cícera, Senhor João Bosco, Gabriela, fico muito feliz por ter vocês na minha vida.

Finally, I cannot thank Daniela Leite enough for her support during this journey, and for being such a wonderful person. Thank you for showing me the world from perspectives I would never have known otherwise. I hope the future holds many more adventures and moments we share.

Thank you!

Felix Dangel
Tübingen, August 31, 2022

Abstract

Automatic differentiation is a key enabler of deep learning: previously, practitioners were limited to models for which they could manually compute derivatives. Now, they can create sophisticated models with almost no restrictions and train them using first-order, *i.e.* gradient, information. Popular libraries like PyTorch [126] and TensorFlow [1] compute this gradient efficiently, automatically, and conveniently with a single line of code. Under the hood, reverse-mode automatic differentiation, or gradient backpropagation, powers the gradient computation in these libraries. Their entire design centers around gradient backpropagation.

These frameworks are specialized around one specific task—computing the average gradient in a mini-batch. This specialization often complicates the extraction of other information like higher-order statistical moments of the gradient, or higher-order derivatives like the Hessian. It limits practitioners and researchers to methods that rely on the gradient. Arguably, this hampers the field from exploring the potential of higher-order information and there is evidence that focusing solely on the gradient has not lead to significant recent advances in deep learning optimization [145].

To advance algorithmic research and inspire novel ideas, information beyond the batch-averaged gradient must be made available at the same level of computational efficiency, automation, and convenience.

This thesis presents approaches to simplify experimentation with rich information beyond the gradient by making it more readily accessible. We present an implementation of these ideas as an extension to the backpropagation procedure in PyTorch. Using this newly accessible information, we demonstrate possible use cases by (i) showing how it can inform our understanding of neural network training by building a diagnostic tool, and (ii) enabling novel methods to efficiently compute and approximate curvature information.

First, we extend gradient backpropagation for sequential feedforward models to Hessian backpropagation which enables computing approximate per-layer curvature. This perspective unifies recently proposed block-diagonal curvature approximations. Like gradient backpropagation, the computation of these second-order derivatives is modular, and therefore simple to automate and extend to new operations.

Based on the insight that rich information beyond the gradient can be computed efficiently and at the same time, we extend the backpropagation in PyTorch with the BackPACK library. It provides efficient and convenient access to statistical moments of the gradient and approximate curvature information, often at a small overhead compared to computing just the gradient.

Next, we showcase the utility of such information to better understand neural network training. We build the Cockpit library that visualizes what is happening inside the model during training through various instruments that rely on BackPACK's statistics. We show how Cockpit provides a meaningful statistical summary report to the deep learning engineer to identify bugs in their machine learning pipeline, guide hyperparameter tuning, and study deep learning phenomena.

Finally, we use BackPACK's extended automatic differentiation functionality to develop ViViT, an approach to efficiently compute curvature information, in particular curvature noise. It uses the low-rank structure of the generalized Gauss-Newton approximation to the Hessian and addresses shortcomings in existing curvature approximations. Through monitoring curvature noise, we demonstrate how ViViT's information helps in understanding challenges to make second-order optimization methods work in practice.

This work develops new tools to experiment more easily with higher-order information in complex deep learning models. These tools have impacted works on Bayesian applications with Laplace approximations [40], out-of-distribution generalization [64, 130], differential privacy [179], and the design of automatic differentiation systems. They constitute one important step towards developing and establishing more efficient deep learning algorithms.

Zusammenfassung

Automatisches Differenzieren stellt eine wesentliche Komponente für Deep Learning dar: Zuvor waren Anwender auf Modelle beschränkt, deren Ableitungen sich manual berechnen ließen. Jetzt können sie komplexe Modelle mit fast beliebiger Struktur entwerfen und diese mit Gradienteninformation trainieren. Software-Bibliotheken wie PyTorch [126] und TensorFlow [1] berechnen den Gradienten effizient und automatisch in einer Codezeile. Im Hintergrund geschieht dies per automatischer Differenzierung im Rückwärtsmodus, genannt Backpropagation. Das Design dieser Bibliotheken basiert auf Backpropagation.

Solche Bibliotheken sind besonders auf eine spezielle Funktion – das Berechnen des gemittelten Gradienten über einen Mini-Batch – ausgerichtet. Diese Spezialisierung erschwert oft die Berechnung anderer Größen, wie höhere statistische Momente des Gradienten oder Ableitungen höherer Ordnung, etwa der Hesse-Matrix. Sie beschränkt Anwender und Forscher auf gradientenbasierte Methoden und hindert die Erforschung des Potenzials höherer Ableitungen und statistischer Momente. Es gibt Anzeichen, dass dieser Fokus auf den Gradienten keine signifikanten Fortschritte in der Optimierung neuronaler Netze erlaubt hat [145].

Um die algorithmische Forschung voranzutreiben und neue Ideen zu inspirieren müssen andere Größen, die über den gemittelten Gradienten eines Mini-Batches hinausgehen, genauso einfach – das heißt automatisiert und effizient – zugänglich gemacht werden.

Diese Arbeit stellt Ansätze vor, die das Experimentieren mit diversen Größen jenseits des Gradienten erleichtern, indem sie diese leichter zugänglich machen. Wir implementieren diese Ideen durch Erweiterung der bestehenden Backpropagation in PyTorch. Mit diesen neu zugänglichen Größen demonstrieren wir Anwendungsszenarien indem wir zeigen wie diese (i) anhand eines Diagnosetools zum besseren Verständnis des Trainings neuronaler Netze führen und (ii) neue Methoden für effiziente und approximative Berechnung von Krümmungsinformation ermöglichen.

Zunächst erweitern wir Gradient Backpropagation für sequenzielle neuronale Netze auf Backpropagation von Hesse-Matrizen, welche die Berechnung von Krümmung innerhalb einer Schicht ermöglicht. Diese Formulierung vereinheitlicht kürzlich vorgeschlagene Näherungsverfahren für Krümmung durch block-diagonale Matrizen. Wie Gradient Backpropagation ist dieses Verfahren für Ableitungen zweiter Ordnung zwischen Schichten entkoppelt und daher einfach zu automatisieren und zu erweitern.

Basierend auf der Erkenntnis, dass reichhaltige Information jenseits des Gradienten gleichzeitig mit diesem berechnet werden kann, erweitern wir die Standard-Backpropagation von PyTorch mit der BackPACK-Bibliothek. Diese bietet effizienten Zugriff auf statistische Gradientenmomente und approximative Krümmungsinformation. Im Vergleich zur Gradientenberechnung ist der Mehraufwand oft gering.

Danach demonstrieren wir den Nutzen solcher Information zum besseren Verständnis des Trainings neuronaler Netze. Wir entwickeln die Cockpit-Bibliothek, die während des Trainings anhand verschiedener Instrumente – basierend auf BackPACKs Größen – visualisiert, was innerhalb des Modells geschieht. Wir zeigen, wie Cockpit Ingenieuren hilft, Fehler in deren Pipeline zu identifizieren, Hyperparameter zu wählen und Deep Learning Phänomene zu untersuchen.

Zuletzt verwenden wir BackPACKs Funktionalität zur Entwicklung von ViViT, einem Verfahren zur effizienten Berechnung von Krümmungsinformation, insbesondere deren Rauschen. ViViT basiert auf dem äußeren Produkt in der verallgemeinerten Gauß-Newton-Matrix und behebt Probleme bestehender Methoden. Durch Beobachtung von Krümmungsrauschen zeigen wir, wie ViViTs Größen dabei helfen, Herausforderungen für die erfolgreiche Realisierung krümmungsbasierter Optimierer zu verstehen.

Diese Arbeit entwickelt neue Tools zum vereinfachten Experimentieren mit Information höherer Ordnung in komplizierten tiefen Netzen. Diese Tools haben Arbeiten zu bayesschen Anwendungen mit Laplace-Approximationen [40], Out-of-Distribution Generalisierung [64, 130], Differential Privacy [179], sowie das Design von Deep Learning Bibliotheken beeinflusst. Sie stellen einen wichtigen Schritt zur Entwicklung und Etablierung effizienterer Algorithmen für Deep Learning dar.

Table of Contents

Notation

The notation is influenced by Goodfellow et al. [60].

Tensors, Matrices, Vectors, Numbers

a	A scalar
$\boldsymbol{a}$	A column vector
$\boldsymbol{A}$	A matrix
A	A tensor
a_i or $[\boldsymbol{a}]_i$	The ith entry of the vector $\boldsymbol{a}$
$A_{i,j}$ or $[\boldsymbol{A}]_{i,j}$	The (i,j)th entry of the matrix $\boldsymbol{A}$ (row i, column j)
$[\boldsymbol{A}]_{i,:}$ (or $[\boldsymbol{A}]_{:,j}$)	The ith row (or jth column) of the matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$ or $[\mathsf{A}]_{i,j,k}$	The (i,j,k)th entry of the tensor A
$\mathrm{vec}(\boldsymbol{A}), \mathrm{vec}(\mathsf{A})$	Matrix/tensor flattened into a vector; convention implies $\mathrm{vec}(\boldsymbol{ABC}) = (\boldsymbol{C}^\top \otimes \boldsymbol{A})\,\mathrm{vec}(\boldsymbol{B})$
$\mathrm{diag}(\boldsymbol{a})$	The square matrix with vector $\boldsymbol{a}$ on the diagonal and zeros elsewhere
$\mathrm{diag}(\boldsymbol{A})$	The vector containing the diagonal elements of the matrix $\boldsymbol{A}$
$\mathrm{diag}(\boldsymbol{A}_1, \dots, \boldsymbol{A}_L)$	A block-diagonal matrix with diagonal blocks given by square matrices $\boldsymbol{A}_1, \dots, \boldsymbol{A}_L$
$\mathrm{Tr}(\boldsymbol{A}), \det(\boldsymbol{A})$	Trace and determinant of a matrix $\boldsymbol{A}$
$\|\boldsymbol{a}\|_2$	L_2 norm of vector $\boldsymbol{a}$, *i.e.* $\|\boldsymbol{a}\|_2^2 = \boldsymbol{a}^\top \boldsymbol{a}$
$\mathrm{eig}(\boldsymbol{A}) := \{(\lambda_k, \boldsymbol{e}_k)\}_k$	Eigendecomposition of the matrix $\boldsymbol{A}$, eigenpairs $(\lambda_k, \boldsymbol{e}_k)$ satisfy $\boldsymbol{A}\boldsymbol{e}_k = \lambda_k \boldsymbol{e}_k$
$(\lambda_k(\boldsymbol{A}), \boldsymbol{e}_k(\boldsymbol{A}))$	kth eigenpair (eigenvalue, eigenvector) of matrix $\boldsymbol{A}$
$\boldsymbol{A} \otimes \boldsymbol{B}$	Kronecker product of two matrices, For two vectors $\boldsymbol{a}, \boldsymbol{b}$, one has $\boldsymbol{a} \otimes \boldsymbol{b}^\top = \boldsymbol{a}\boldsymbol{b}^\top$
$\mathsf{A} \odot \mathsf{B}, \boldsymbol{A} \odot \boldsymbol{B}, \boldsymbol{a} \odot \boldsymbol{b}$	Elementwise multiplication (Hadamard product) of two tensors, matrices, vectors
$\mathsf{A} \oslash \mathsf{B}, \boldsymbol{A} \oslash \boldsymbol{B}, \boldsymbol{a} \oslash \boldsymbol{b}$	Elementwise division (Hadamard division) of two tensors, matrices, vectors
$\mathsf{A}^{\odot 2}, \boldsymbol{A}^{\odot 2}, \boldsymbol{a}^{\odot 2}$	Elementwise square of a tensor, matrix, vector
$\mathsf{A}^{\odot 1/2}, \boldsymbol{A}^{\odot 1/2}, \boldsymbol{a}^{\odot 1/2}$	Elementwise square root of a tensor, matrix, vector

Empirical Risk Minimization

A datum is usually indicated by a subscript $_n$.

$(\boldsymbol{x}, \boldsymbol{y})$	Labeled datum with input features $\boldsymbol{x}$ and target $\boldsymbol{y}$
$(\boldsymbol{x}_n, \boldsymbol{y}_n)$	Datum n from a dataset
D	Total number of parameters in a model
$\boldsymbol{\theta} \in \Theta := \mathbb{R}^D$	Parameter vector of a model
$\boldsymbol{f} := f_{\boldsymbol{\theta}}(\boldsymbol{x})$	Prediction of a model $f_{\boldsymbol{\theta}}$ for input features $\boldsymbol{x}$
ℓ or $\ell(\boldsymbol{f}, \boldsymbol{y})$	Loss function to compare prediction and target; convex in $\boldsymbol{f}$
$\mathbb{D} := \{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_{n=1}^{\lvert\mathbb{D}\rvert}$	A dataset containing instances of labeled data $(\boldsymbol{x}_n, \boldsymbol{y}_n)$ indexed by n
$\mathbb{B}$	A mini-batch $\mathbb{B} \subseteq \mathbb{D}$
N	Number of data in a mini-batch or a dataset, depending on the context
$\boldsymbol{f}_n := f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$	Model prediction for datum n
ℓ_n or $\ell(\boldsymbol{f}_n, \boldsymbol{y}_n)$	Loss of datum n
$p_{\mathbb{D}}(\boldsymbol{x}, \boldsymbol{y})$	Empirical distribution of a dataset $\mathbb{D}$
$\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$	Empirical risk implied by the empirical distribution of a dataset $\mathbb{D}$
$\mathcal{L}_{\mathbb{D}_{\text{train}}}(\boldsymbol{\theta}), \mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$, *etc.*	Training loss, mini-batch loss, *etc.*

Neural Networks

The layer number is indicated by parenthesized superscripts $^{(l)}$.

L	Total number of layers
$d^{(l)}$	Number of parameters in layer l; total number of parameters is $D = \sum_{l=1}^{L} d^{(l)}$
$\boldsymbol{\theta}^{(l)} \in \mathbb{R}^{d^{(l)}}$	Parameter vector of layer l, potentially empty for parameter-free layers like activations
$h^{(l-1)}$	Number of (hidden) inputs fed into layer l
$M := h^{(0)}, C := h^{(L)}$	Input feature dimension, output dimension (number of classes for classification)
$\boldsymbol{z}^{(l-1)} \in \mathbb{R}^{h^{(l-1)}}$	(Hidden) features fed into layer l (output of layer $l-1$)
$\boldsymbol{x} := \boldsymbol{z}^{(0)}, \boldsymbol{f} := \boldsymbol{z}^{(L)}$	Input to the neural network, and its prediction for input $\boldsymbol{x}$
$f^{(l)}_{\boldsymbol{\theta}^{(l)}}$	Layer l parameterized by $\boldsymbol{\theta}^{(l)}$, mapping input $\boldsymbol{z}^{(l-1)}$ to output $\boldsymbol{z}^{(l)}$
$f_{\boldsymbol{\theta}} := f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ \ldots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}}$	Sequential feedforward neural network parameterized by $\boldsymbol{\theta}$, maps input $\boldsymbol{x}$ to output $\boldsymbol{f}$
$\boldsymbol{\theta}$	Parameter vector, concatenation of parameters over layers $\boldsymbol{\theta} := (\boldsymbol{\theta}^{(1)\top}, \ldots, \boldsymbol{\theta}^{(L)\top})^\top$

Derivatives

∇, J and ∇^2 denote the gradient, Jacobian, and Hessian, respectively.

$\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}$	Jacobian matrix of a vector $\boldsymbol{b}$ *w.r.t.* a vector $\boldsymbol{a}$, $[\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}]_{i,j} = \partial[\boldsymbol{b}]_i / \partial[\boldsymbol{a}]_j$
$\mathrm{J}_{\mathbf{A}}\mathbf{B}$	Generalized Jacobian matrix for tensor variables, $[\mathrm{J}_{\mathbf{A}}\mathbf{B}]_{i,j} = \partial[\operatorname{vec}\mathbf{B}]_i / \partial[\operatorname{vec}\mathbf{A}]_j$
$\nabla_{\boldsymbol{a}} b := (\mathrm{J}_{\boldsymbol{a}} b)^\top$	Gradient vector of a scalar b *w.r.t.* a vector $\boldsymbol{a}$, $[\nabla_{\boldsymbol{a}} b]_i = \partial b / \partial a_i$
$\nabla^2_{\boldsymbol{a}} b$	Hessian matrix of a scalar b *w.r.t.* a vector $\boldsymbol{a}$, $[\nabla^2_{\boldsymbol{a}} b]_{i,j} = \partial^2 b / \partial[\boldsymbol{a}]_i \partial[\boldsymbol{a}]_j$ (symmetric)
$\nabla^2_{\boldsymbol{a}} \boldsymbol{b}$, $\nabla^2_{\mathbf{A}} \mathbf{B}$	Generalized Hessian matrix (in general not quadratic, hence not symmetric) of a vector $\boldsymbol{b}$ *w.r.t.* a vector $\boldsymbol{a}$, or more general tensor variables
$\boldsymbol{g}_n(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})$	Gradient of the loss implied by sample n
$\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$	Gradient of the empirical risk implied by a dataset $\mathbb{D}$
$\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$	Mini-batch gradient
$\boldsymbol{H}_n(\boldsymbol{\theta}) := \nabla^2_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})$	Hessian of the loss implied by sample n
$\boldsymbol{H}_{\mathbb{D}}(\boldsymbol{\theta}) := \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$	Hessian of the empirical risk implied by a dataset $\mathbb{D}$
$\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}) := \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$	Mini-batch Hessian
$\boldsymbol{H}^{(l)}(\boldsymbol{\theta}^{(l)})$ or $\boldsymbol{H}(\boldsymbol{\theta}^{(l)})$	The block in the Hessian corresponding to layer l
$\boldsymbol{G}_{\mathbb{D}}(\boldsymbol{\theta})$	Generalized Gauss-Newton matrix on a dataset $\mathbb{D}$
$\boldsymbol{G}^{(l)}(\boldsymbol{\theta}^{(l)})$ or $\boldsymbol{G}(\boldsymbol{\theta}^{(l)})$	The block in the generalized Gauss-Newton matrix corresponding to layer l

Statistics

$\mathcal{U}(\{1, \ldots, N\})$	Uniform distribution over $\{1, \ldots, N\}$
$\mathcal{N}(x \mid \mu, \sigma^2)$	Uni-variate normal/Gaussian distribution of random variable x, with mean μ, positive variance σ^2, and density $\mathcal{N}(x \mid \mu, \sigma^2) = 1/\sigma\sqrt{2\pi} \exp[-1/2((x-\mu)/\sigma)^2]$
$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Multi-variate normal/Gaussian distribution of random vector $\boldsymbol{x}$ with mean vector $\boldsymbol{\mu}$, PSD covariance matrix $\boldsymbol{\Sigma}$, and density $\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = 1/(\sqrt{2\pi \det \boldsymbol{\Sigma}}) \exp[-1/2(\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})]$
$\mathrm{Cat}(c \mid \boldsymbol{p})$	Multinomial/Categorical distribution with probabilities $\boldsymbol{p}$ for categories c

Miscellaneous

log	The natural logarithm (base e, *i.e.* $\log(\mathrm{e}) = 1$)

$\mathrm{onehot}(c)$	One-hot vector of class c with $\mathrm{onehot}(c) = \delta_{i,c}$
$\mathrm{softmax}(\boldsymbol{a})$	Softmax probabilities of the logits $\boldsymbol{a}$, $[\mathrm{softmax}(\boldsymbol{a})]_c = {}^{\exp(a_c)}/_{\sum_{i=1}\exp(a_i)}$.
$\delta_{i,j}$, $\delta(\boldsymbol{x} - \boldsymbol{a})$	Kronecker delta ($\delta_{i,i} = 1$ and $\delta_{i,j \neq i} = 0$), Dirac delta distribution
$(\mathbb{X} \to \mathbb{Y})$	Signature of a function that maps between $\mathbb{X}$ and $\mathbb{Y}$
$\{\boldsymbol{x}_n\}$ or $\{\boldsymbol{x}_n\}_n$	A set/collection of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots$ over the index set implied by n
$\hat{\boldsymbol{e}}_i$	Unit vector in direction i, *i.e.* $\hat{\boldsymbol{e}}_i = \mathrm{onehot}(i)$
$\mathbf{1}_m$	An m-dimensional vector containing ones everywhere
$\log(\boldsymbol{a}), \exp(\boldsymbol{a})$	Elementwise natural logarithm and exponential function of a vector
$m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta})$	Local approximation of the loss in around $\boldsymbol{\theta}_t$

Acronyms & Abbreviations

E.g. or *e.g.*	For example (*exempli gratia*)
Etc. or *etc.*	And so on (*et cetera*)
I.e. or *i.e.*	That is (*id est*)
I.i.d. or *i.i.d.*	Independent and identically distributed
W.r.t. or *w.r.t.*	With respect to
AD	Automatic differentiation
API	Application Programming Interface
BDA	Block diagonal approximation
CG	Conjugate gradients
CNN	Convolutional neural network
CPU	Central processing unit
DNN	Deep neural network
DP	Differential privacy
FCNN	Fully-connected neural network
GGN	Generalized Gauss-Newton (matrix)
GN	Gauss-Newton (matrix)
GPU	Graphics processing unit
HBP	Hessian backpropagation
JMP	Jacobian-matrix product
JVP	Jacobian-vector product
KFAC	Kronecker-factored curvature
KFC	Kronecker factors for convolution
KFLR	Kronecker-factored low rank
KFRA	Kronecker-factored recursive approximation
MAP	Maximum a posteriori (estimation)
MC	Monte Carlo
MJP	Matrix-Jacobian product
ML	Machine learning
MLE	Maximum likelihood estimation
MLP	Multi-layer perceptron
NGD	Natural gradient descent
PCH	Positive-curvature Hessian
PD	Positive definite
PSD	Positive semi-definite
ResNet	Residual (neural) network
SNR	Signal-to-noise ratio
TPU	Tensor processing unit
VJP	Vector-Jacobian product

Overview 1.

1.1 Introduction

Deep learning has achieved significant performance gains over traditional methods on various tasks like image classification [41, 68, 91, 153, 159, 181], image generation [35, 58, 84, 131], machine translation [7, 102, 172], and game play [111, 151, 152]. These applications are powered by machine learning (ML) frameworks [1, 32, 126, 165] that tremendously reduce the added complexity for practitioners of using highly-efficient implementations on hardware accelerators like GPUs [88] and TPUs [83]. The convenience introduced by these libraries makes deep learning more accessible and is one main factor for its popularity and success.

It is surprisingly easy to build, train, and deploy a model, despite deep learning being computationally extremely demanding. Typically, neural networks have millions [68, 91, 153, 159, 181], billions [24, 129], even trillions [49] of parameters, and are trained on large data sets [41, 98] that can only be processed in smaller batches. The parameters are adjusted during training, which is phrased as optimization of a performance measure called "the loss".

Training, *i.e.* optimization, proceeds in an iterative fashion: to update the model, a mini-batch of data is fed through the model (forward pass) to evaluate its current performance—the mini-batch loss. This loss is then differentiated *w.r.t.* the model's parameters to obtain the mini-batch gradient (backward pass). Finally, an optimizer incorporates this gradient to update the model parameters. Popular frameworks like PyTorch [126] and TensorFlow [1] realize this procedure in code that looks similar to the following pseudocode:

```
1   dataset = ...    # Learning task examples
2
3   model = ...      # Practitioner's choice
4   loss_func = ...  # Practitioner's choice
5
6   optimizer = ...  # First-order method
7
8   while not_converged: # Standard training loop
9       features, targets = dataset.next_minibatch()
10
11      # Forward pass: Compute the loss
12      predictions = model(features)
13      loss = loss_func(predictions, targets)
14
15      # Backward pass: Compute the gradient
16      loss.backward()
17
18      # Update model parameters using the gradient
19      optimizer.step()
20      optimizer.zero_grad()
```

Procedure 1.1: **Canonical deep learning training loop.** After setting up the data, model, loss function, and optimizer, iterate over batches: in each iteration, compute the mini-batch loss in a forward pass, and its gradient with a backward pass. Then use the gradient as learning signal to update the model parameters.

This framework frees practitioners from implementation details and lets them focus on, *e.g.*, specifying the neural network (Line 3) and performance criterion (Line 4). No matter how complicated a model

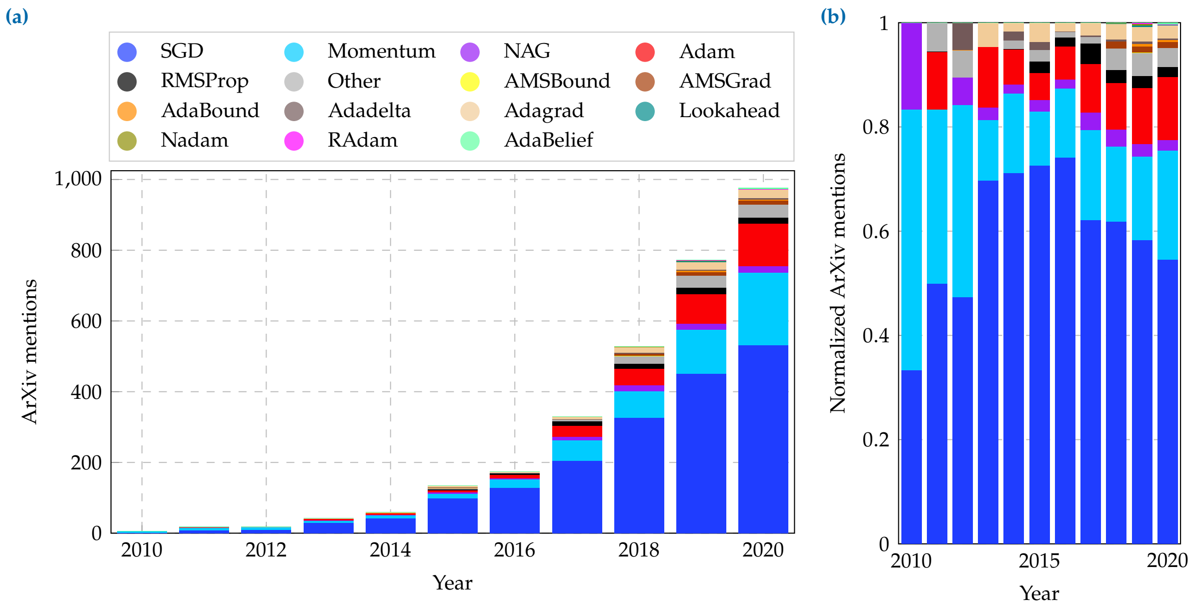


Figure 1.1: All popular deep learning optimizers are first-order methods that rely on the gradient. Reproduced from Schmidt et al. [145] with permission from the authors. **(a)** Number of ArXiv mentions in titles and abstracts of deep learning optimization methods, with other methods grouped into a single category. **(b)** The same plot with mentions normalized to the unit interval.

a practitioner may come up with: as long as it is differentiable, it will be compatible with the above training loop, and can be automatically differentiated and trained. This allows practitioners to focus on modeling aspects of their problem. Thanks to automatic differentiation built into ML libraries, they do not have to worry about low-level details of how to compute the learning signal. They obtain the gradient with a single function call that performs the backward pass (Line 16).

However, by abstracting these details, Procedure 1.1 limits practitioners and researchers to gradient access only. This is not restrictive for the most popular optimizers to train DNNs as they only rely on the gradient (Figure 1.1). But the development of novel methods focuses mainly on inventing alternative update rules involving the gradient. Currently, there exist more than one hundred such methods; see *e.g.* [145] for an overview. This leads to the question whether one of these update rules performs significantly better than others. A broad comparison of the methods in Figure 1.1, however, raises concerns that those methods do not seem to be specialized to problems or leverage different problem properties:

> *Despite efforts by the community, there is currently no method that clearly dominates the competition. [. . .] tuning helps about as much as trying other optimizers.* *Schmidt et al. [145]*

One reason for this may be that despite the diversity of update rules, they all rely on the same gradient information. Research and the development of novel methods should therefore use more than just the gradient.

Such information can be of statistical or geometrical nature (Figure 1.2). The loss/gradient is the empirical mean of the distribution of per-sample losses/gradients. Higher moments of the gradients, such as their variance, could enable more robust estimation of the mean. Higher-order derivatives, *e.g.* curvature in form of the Hessian, encode the loss

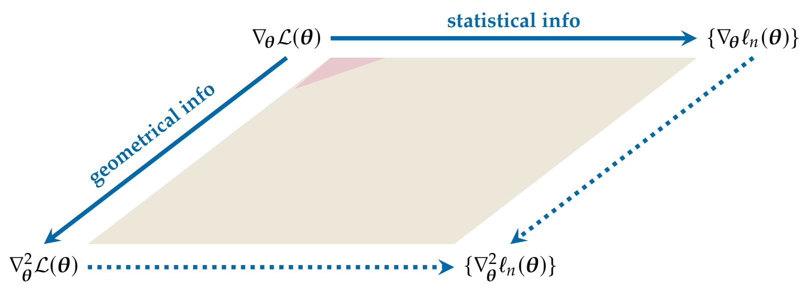

$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$
statistical info
$\{\nabla_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta})\}$
geometrical info
$\nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$
$\{\nabla^2_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta})\}$

Figure 1.2: **Beyond the average gradient.** Higher-order statistical (noise) and geometrical (curvature) information provide a richer local approximation of the loss in deep learning, which relies mostly on the gradient highlighted in ●. The goal of this thesis is making higher-order information ● more readily available (figure inspired by [22]). $\nabla_{\boldsymbol{\theta}}\mathcal{L}$, $\nabla^2_{\boldsymbol{\theta}}\mathcal{L}$ denote average gradient and Hessian; $\{\nabla_{\boldsymbol{\theta}}\ell_n\}$, $\{\nabla^2_{\boldsymbol{\theta}}\ell_n\}$ are the distributions of per-sample gradients and Hessians.

landscape's geometry. These quantities are useful to build approximations of the loss that take its stochastic nature and curvature into account.

Optimization methods that use more than the gradient are the default in other applications (*e.g.* convex optimization, generalized linear models) and have also been considered in deep learning [5, 14, 53, 69, 106, 109, 178, 183]. However, they are often so complicated to implement that practitioners barely try them out: efficiently computing with higher-order information is challenging because its explicit representation is often too large to be stored in memory. Therefore, practical methods rely on implicit schemes, such as matrix-vector products, or light-weight structured approximations. Understanding and efficiently implementing them requires expert knowledge, which complicates experimentation.

The adoption of, and research on, algorithms depend on ease-of-use within frameworks. Existing software has closely evolved with the popularity of first-order methods that only demand efficient access to the gradient. This leads to the question on how to make higher-order information more readily available within such frameworks to expand the available toolkit. As backpropagation [137] forms the computational backbone of deep learning, this thesis describes approaches to extend the backpropagation algorithm, and answers the research questions:

(Q1) *Which information beyond the gradient is efficiently accessible?*
(Q2) *How to compute this information—conveniently, automatically, and efficiently—re-using the existing backpropagation implementation of ML frameworks?*
(Q3) *How to use this information to advance gradient-based deep learning?*

1.2 Outline

This thesis contains three main parts: (I) an introduction to the setting, related concepts, and relevant notation along with motivation; (II) the achieved scientific contributions; and (III) a discussion of their impact and future directions. Throughout the text, additional details (examples, auxiliary calculations, *etc.*) are provided in margins. Readers that are familiar with the subject should feel encouraged to read the main text without interruptions of this additional material. An extensive appendix (IV) complements the main parts. It collects lengthier mathematical discussions and details about experiments and implementations, along with additional empirical results.

Part I provides background material and motivates the goal of this work. Chapter 2 introduces the components of supervised deep learning through gradient-based empirical risk minimization (Procedure 1.1). The goal is to highlight how each component introduces structure in the empirical risk that can be leveraged for convenient and efficient computation. This structure is helpful to identify new classes of interesting quantities and understand the computational pipeline that will be extended in this work. Specifically, we inspect the following components:

- **The loss:** Section 2.1 presents supervised learning through empirical risk minimization and outlines connections of loss-based learning to approximating the unknown data distribution through samples via maximum likelihood estimation (MLE) or maximum a posteriori (MAP) estimation. Further, it highlights the finite-sum structure in the loss. This allows for massive speed-ups on parallel hardware accelerators like GPUs [88] when evaluating the loss, and stochastic approximation via sub-sampling with mini-batches, which introduces noise into the computed quantities.
- **The model:** Section 2.2 outlines structure in the optimization space given by the parameter space of a deep neural network. DNNs are usually highly over-parameterized, containing millions [68, 91, 153, 159, 181], billions [24, 129], even trillions [49] of parameters. Yet, they are built from relatively simple functional units—layers, or modules—glued together through function composition. We give a selected overview of layers, and present sequential feedforward neural networks, which are the model class this work focuses on.
- **The gradient:** Section 2.3 outlines the importance of automatic differentiation (AD) for ML. It starts with fundamentals on how to automate the chain rule for functions represented by a computation graph that tracks the dependencies between input, output, and intermediate variables. One important property is the modularity of AD, which allows for an elegant and extensible implementation: because the chain rule relates the derivative of a function composition to the derivatives of its composites, only these composites need to implement derivatives. New operations that cannot be composed from primitives can be added by specifying their derivatives. The discussion concludes by presenting gradient backpropagation [137], the most prominent evaluation scheme to compute gradients of the loss in deep learning which will be extended in this work.

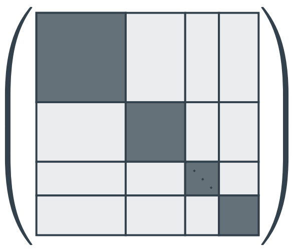
Chapter 3 motivates why focusing only on the gradient might be problematic, and motivates the importance of higher-order information, *e.g.* in form of noise (stochasticity) and curvature, to further enrich deep learning algorithms. First, Section 3.1 exemplifies the gradient's dominance in optimization for deep learning where state-of-the-art methods are first-order methods. Next, Section 3.2 introduces relevant higher-order information for second-order optimization methods that rely on curvature. It discusses popular curvature matrices, such as the Hessian, generalized Gauss-Newton (GGN), Fisher information, and gradient covariance matrix, as well as their connections. Section 3.3 further motivates the utility of such information through concrete examples.

This concludes the background material. We identified and appreciated the strengths of deep learning libraries, their potential shortcomings that might complicate advances in the field, and found higher-order information as a promising direction for their extension.

Part II tackles this manuscript's goal to *extend the functionality of machine learning libraries to efficiently, automatically, and conveniently provide access to higher-order information*. It describes doing so by extending the gradient backpropagation algorithm of ML libraries to go beyond the gradient.

Chapter 4 presents Hessian backpropagation (HBP), an extension of gradient backpropagation to compute layer-wise curvature in sequential feedforward neural networks. Just like gradient backpropagation recovers the gradient vectors in blocks that correspond to layers, local curvature, *i.e.* second-order partial derivatives of the loss *w.r.t.* parameters in a layer can be evaluated by backpropagating Hessians. Its computation is disentangled to the modular level, which allows for an elegant and extensible implementation in analogy to gradient backpropagation. We describe the backpropagation operations at the per-layer level, resulting in an algorithm that computes local curvature in an automated fashion, and at the same time as the gradient. Adaptations of the exact procedure recover positive semi-definite block diagonal approximations (BDAs)—*e.g.* of the GGN—and recently proposed Kronecker-factored curvature approximations [21, 31, 109] of the Hessian, unifying their computation.

Chapter 4: HBP: block-diagonal curvature via Hessian backpropagation.



github.com/f-dangel/hbp

Disclaimer 1.1 Chapter 4 is based on the peer-reviewed conference paper with the following co-author contributions:

F. Dangel, S. Harmeling, and P. Hennig. "Modular Block-diagonal Curvature Approximations for Feedforward Architectures". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2020 [37]

	Ideas	Experiments	Analysis	Writing
F. Dangel	70 %	80 %	70 %	65 %
S. Harmeling	10 %	10 %	10 %	10 %
P. Hennig	20 %	10 %	20 %	25 %

Chapter 5 further generalizes this idea and presents BackPACK, an efficient framework that extends the gradient backpropagation of PyTorch. The library provides access to higher-order statistical information about the gradient distribution, like individual gradients or an estimate of their variance, and structured curvature information, like the Hessian/GGN diagonal and block-diagonal Kronecker-factored curvature approximations. This is achieved with slightly more flexible implementations of AD functionality and by backpropagating additional information through the graph. Importantly, most quantities add small overhead to the gradient, making their exploration for research more attractive.

Chapter 5: BackPACK: an efficient framework built on top of PyTorch that extends the backpropagation algorithm.

github.com/f-dangel/backpack

Disclaimer 1.2 Chapter 5 is based on the peer-reviewed conference publication with the following co-author contributions:

F. Dangel, F. Kunstner, and P. Hennig. "BackPACK: Packing more into Backprop". *International Conference on Learning Representations (ICLR)*. 2020 [38]

	Ideas	Experiments	Analysis	Writing
F. Dangel	33 %	55 %	45 %	35 %
F. Kunstner	33 %	45 %	45 %	45 %
P. Hennig	33 %	0 %	10 %	20 %

Chapter 6: Cockpit: a debugging tool for the training of deep neural networks.

github.com/f-dangel/cockpit

Chapter 6 introduces Cockpit, a live-monitoring debugging tool that consists of various instruments which leverage higher-order information, efficiently provided by BackPACK. By providing a deeper look into the inner workings of neural networks through the lens of this information, Cockpit can help identify bugs in the ML pipeline, while keeping the computational overhead acceptable. This demonstrates the utility of higher-order information to assist deep learning practitioners and researchers to better understand their problems and conduct research.

Disclaimer 1.3 Chapter 6 is based on the peer-reviewed conference paper with the following co-author contributions:

F. Schneider, F. Dangel, and P. Hennig. "Cockpit: A Practical Debugging Tool for Training Deep Neural Networks". *Advances in Neural Information Processing Systems (NeurIPS)*. 2021 [147]

	Ideas	**Experiments**	**Analysis**	**Writing**
F. Schneider	45 %	40 %	40 %	45 %
F. Dangel	40 %	50 %	40 %	40 %
P. Hennig	15 %	10 %	20 %	15 %

Chapter 7: ViViT: efficient computation with the GGN's low-rank structure.

github.com/f-dangel/vivit

Chapter 7 presents ViViT, a method that leverages the low-rank structure in the GGN to efficiently extract eigenvalues, eigenvectors, per-sample first- and second-order directional derivatives, and Newton steps. In contrast to other popular curvature approximations, ViViT is capable of tracking off-diagonal curvature blocks, offers principled approximations to trade off cost and accuracy, and allows studying noise in the curvature. Under the hood, ViViT's quantities are efficiently computed during backpropagation, building on BackPACK's advanced AD functionality. This demonstrates how such functionality enables investigations of unexplored structure in higher-order information which may be used to develop novel algorithms and better understand challenges to make them work in practice, *e.g.* noise.

Disclaimer 1.4 Chapter 7 is based on the peer-reviewed journal paper with the following co-author contributions:

F. Dangel, L. Tatzel, and P. Hennig. "ViViT: Curvature Access Through The Generalized Gauss-Newton's Low-Rank Structure". *Transactions on Machine Learning Research (TMLR)* (2022) [39]

	Ideas	**Experiments**	**Analysis**	**Writing**
F. Dangel	50 %	40 %	40 %	40 %
L. Tatzel	35 %	50 %	40 %	45 %
P. Hennig	15 %	10 %	20 %	15 %

Part III summarizes the findings of this manuscript *w.r.t.* the inital questions from Page 3, as well as their impact and relation to the latest developments in the field. Finally, the manuscript identifies future research directions for the presented extended AD eco-system around PyTorch, and more broadly for the development of future ML libraries.

Part I.

Background & Motivation

Deep Learning Components 2.

Broadly speaking, ML seeks to find a well-performing algorithm for a given task from "experience". This thesis considers supervised deep learning, where "experience" is given through annotated examples in form of a dataset. A datum $(\boldsymbol{x}, \boldsymbol{y})$ consists of *input features* $\boldsymbol{x}$ and *targets*, or *labels*, $\boldsymbol{y}$. The algorithm, or *model*, is a deep neural network (Section 2.2) that tries to predict $\boldsymbol{y}$ from $\boldsymbol{x}$ and is selected by minimizing a performance criterion on the available data (the empirical risk, Section 2.1) using optimization methods which rely on automatically computed derivatives (Section 2.3). This chapter reviews these components, highlighting their structure *w.r.t.* implementation in ML libraries. For a broader introduction to deep learning, see *e.g.* [60].

2.1 Empirical Risk Minimization

"Learning" is connected to optimization by the risk minimization paradigm. The idea is to define a performance metric that assesses the quality of the model's prediction. Then, learning happens by maximizing that performance metric. Conversely, one can specify a metric for the prediction's error (also referred to as *risk*), and minimize the latter.

For example, an intuitive way to assess performance for a classification task is *accuracy*, the ratio of correct and total predictions (the error would be the ratio of incorrect and total predictions). However, such direct performance measures are hard to optimize with derivative-based methods[1] and must be substituted by a surrogate function that approximates the original performance measure, but is easier to optimize. In the deep learning terminology, these surrogates are commonly referred to as *loss functions*. This section expands on risk minimization, its characteristic properties in deep learning, and its probabilistic interpretation.

1: *e.g.* the accuracy on one datum is either 0 or 1, and hence its derivative vanishes everywhere it is defined.

2.1.1 Notation & Mathematical Details

Risk & Empirical Risk

Consider supervised learning with the goal to learn the functional relation between inputs $\boldsymbol{x} \in \mathbb{X}$ and targets $\boldsymbol{y} \in \mathbb{Y}$ (Figure 2.1). The mapping is described by a model $f_{\boldsymbol{\theta}} : \mathbb{X} \to \mathbb{F}$ with adjustable parameters $\boldsymbol{\theta} \in \mathbb{\Theta}$ that produces predictions $\boldsymbol{f} := f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \mathbb{F}$ for an input $\boldsymbol{x}$. A prediction's error is assessed through a convex loss function $\ell : \mathbb{F} \times \mathbb{Y} \to \mathbb{R}$.

Regression (Example 2.1) and *classification* (Example 2.2) are two common tasks that will be used frequently in later chapters. The remainder of this text sets $\mathbb{X} = \mathbb{R}^M$ and $\mathbb{F} = \mathbb{R}^C$, with input and prediction space dimensions M, C, respectively (C corresponds to the number of classes for classification). Because the discussion focuses on neural networks as model, it sets $\mathbb{\Theta} = \mathbb{R}^D$ in what follows.

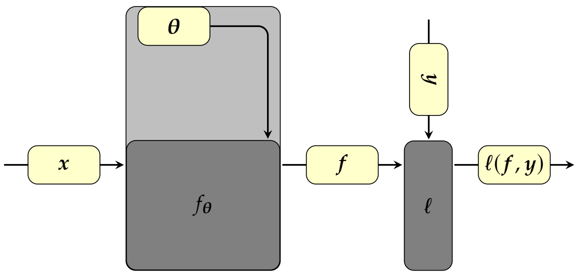


Figure 2.1: Components of supervised learning: The goal is to infer parameters $\boldsymbol{\theta}$ of a model $f_{\boldsymbol{\theta}}$, that relates inputs $\boldsymbol{x}$ to predictions $\boldsymbol{f}$, by minimizing a loss function ℓ between the prediction $\boldsymbol{f}$ and label $\boldsymbol{y}$.

Assume that the population of input-target pairs, *i.e.* the *data-generating process*, follows a distribution $p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$. A model's expected risk under this distribution is defined as

$$\begin{aligned}\mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta}) &:= \mathbb{E}_{p_{\text{data}}(\boldsymbol{x},\boldsymbol{y})}\left[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})\right] \\ &= \iint_{\mathbb{X},\mathbb{Y}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y}) \,\mathrm{d}\boldsymbol{x}\,\mathrm{d}\boldsymbol{y}\,.\end{aligned} \tag{2.3a}$$

The incentive for a model to perform well is to achieve a small expected risk. Therefore, training a model is minimizing Equation (2.3a),

$$\operatorname*{minimize}_{\boldsymbol{\theta}} \mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta})\,. \tag{2.3b}$$

But in practice, Equation (2.3b) is inaccessible because the data-generating process $p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$ is unknown. Instead, the problem is approximated through an *i.i.d.* dataset $\mathbb{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X}\times\mathbb{Y}\}_n$ of labeled data collected from $p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$. $p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$ is approximated by the empirical distribution

$$p_{\mathbb{D}}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n)\in\mathbb{D}} \delta(\boldsymbol{x} - \boldsymbol{x}_n)\delta(\boldsymbol{y} - \boldsymbol{y}_n) \tag{2.4a}$$

implied by $\mathbb{D}$. The model's empirical risk on $\mathbb{D}$ follows from substituting p_{data} with $p_{\mathbb{D}}$ in Equation (2.3a), which yields

$$\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) := \mathcal{L}_{p_{\mathbb{D}}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n)\in\mathbb{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)\,. \tag{2.4b}$$

Example 2.1 (Least squares regression & square loss) Regression associates features in $\mathbb{X} = \mathbb{R}^M$ with targets in $\mathbb{Y} = \mathbb{R}^C$. A prediction in $\mathbb{F} = \mathbb{R}^C$ compares to its ground truth via the mean squared error[2]

$$\ell(\boldsymbol{f}, \boldsymbol{y}) = \frac{1}{C}\sum_{c=1}^{C}(y_c - f_c)^2 = \frac{1}{C}\|\boldsymbol{y} - \boldsymbol{f}\|_2^2 \tag{2.1}$$

2: There exist different conventions for the normalization factor. This text adapts the implementation of `MSELoss` (with `reduction="mean"` mode) in PyTorch for consistency with the code presented in later chapters. Normalizing by $1/C$ is also close to what the name, mean squared error, suggests.

Example 2.2 (C-class classification & softmax cross-entropy loss) Classification assigns features in $\mathbb{X} = \mathbb{R}^M$ to classes in $\mathbb{Y} = \{1, \dots, C\}$ using a model to $\mathbb{F} = \mathbb{R}^C$. The softmax cross-entropy loss[3] maps the model's prediction to a probability distribution over classes, then uses cross-entropy to compare it with the ground truth,

$$\ell(\boldsymbol{f}, y) = -\log([p(\boldsymbol{f})]_y) = -\log p(\boldsymbol{f})^\top \operatorname{onehot}(y) \tag{2.2}$$

where $p(\boldsymbol{f}) = \operatorname{softmax}(\boldsymbol{f})$ and $[\operatorname{onehot}(y)]_c = \delta_{c,y}$.

3: Sometimes, the softmax is considered part of the model rather than the loss function. This text assigns it to the loss function, in line with the PyTorch implementation of `CrossEntropyLoss`, that combines softmax and cross-entropy, which is numerically more stable [60, Chapter 4].

In practice, learning happens by minimizing an empirical risk,

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\, \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})\,. \tag{2.4c}$$

Challenges for Optimization

Training a model relies on optimization algorithms which are initialized at some $\boldsymbol{\theta}_0 \in \Theta$ and seek to iteratively improve the solution to Equation (2.4c). At an iteration t, the optimizer is given access to information about the objective at $\boldsymbol{\theta}_t$, like derivatives (Section 2.3), to deduct a step $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t$, potentially using additional information from past observations. In general, the more local information is available, the more powerful a single update step can be. But richer information is often more costly to compute. The large-scale nature of deep learning poses challenges on the accessible information:

- **Big data:** deep learning datasets are often large because more data gives better approximations of the true distribution p_{data} via $p_{\mathbb{D}}$, and thus better task performance. The simultaneous computation of all per-datum losses for $\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$ does not fit into memory (for many contemporary tasks, it is even infeasible to hold $\mathbb{D}$ in memory[4]). Still, to obtain $\mathcal{L}_{\mathbb{D}}$, one could sequentially compute and reduce its summands on data chunks of manageable size. In practice, it is more common though to approximate $\mathcal{L}_{\mathbb{D}}$ on a randomly drawn small subset, a mini-batch, $\mathbb{B} \subseteq \mathbb{D}$ with $|\mathbb{B}| \ll |\mathbb{D}|$. While this avoids a computationally expensive full sweep over the data, subsampling introduces *noise* in the loss (Section 2.1.2). This noise is inherited by the computed information supplied to the optimizer. Algorithms must therefore take into account the stochastic nature of their observations.
- **Very large models:** the parameter space dimension D of DNNs usually exceeds the (already large) amount of data $|\mathbb{D}|$, *i.e.* $D \gg |\mathbb{D}|$. It affects the complexity to store and compute information, such as derivatives of the loss *w.r.t.* $\boldsymbol{\theta}$, and makes it more challenging to efficiently work with higher-order information[5]. Therefore, working with higher-order information relies on implicit schemes (*e.g.* matrix-free Hessian-vector products [127]) or light-weight structured approximations (*e.g.* through Kronecker products [109]). Such approaches are usually technical and challenging to implement. They also add significant computational work that must be compensated for by an improved update step quality in optimizers.

4: *E.g.* ImageNet [41] consists of 1,281,167 training images. The Kaggle download requires roughly 166 GB of memory.

5: For example, interesting quantities for an optimizer are the loss landscape's local slope (gradient) and curvature (Hessian, Definition 3.1). While the gradient is cheap to compute and store (D elements), holding the $D \times D$ Hessian in memory is infeasible.

In addition to these computational aspects, there exist other challenges:

- **Non-convexity:** although the loss function ℓ is convex *w.r.t.* the model's output f, convexity does not carry through to the model's parameters[6]. This is because DNNs $f_{\boldsymbol{\theta}}$ are highly non-linear, and therefore generally non-convex, in $\boldsymbol{\theta}$. Local minima of convex functions are global minima, meaning that local improvement gets us closer to a global solution. But non-convex problems like Equation (2.4c) have multiple local minima that need not be global. Through local improvements, an optimizer can arrive at one of these local minima, but with no path of improvement to a global

6: Convexity of ℓ in f is still useful to construct PSD approximations to the Hessian, see Section 3.2.3.

solution. This depends on various aspects, such as the update rule, hyperparameters, initialization, *etc.*.

- **Generalization:** learning is not pure optimization. While minimizing the empirical risk Equation (2.4c) improves the model's performance on the collected data, the actual goal is to obtain good performance on new unseen data, *i.e.* generalize well. To achieve generalization, it is crucial to prevent optimization to overfit specifics in the data. It is common practice to split the data into three disjoint sets $\mathbb{D}_{\text{train}}$, $\mathbb{D}_{\text{valid}}$, and $\mathbb{D}_{\text{test}}$. The train set's empirical risk $\mathcal{L}_{\mathbb{D}_{\text{train}}}(\boldsymbol{\theta})$ is minimized, and the validation loss $\mathcal{L}_{\mathbb{D}_{\text{valid}}}(\boldsymbol{\theta})$ serves to identify hyperparameters that lead to generalization on $\mathbb{D}_{\text{valid}}$. The held-out examples in the test set $\mathbb{D}_{\text{test}}$ are used to assess generalization to new data. Another way to improve generalization is to use more data during training. Data augmentation [150] allows for cheap generation of new examples without collecting new data. Sometimes, it may be desirable to penalize model properties by adding a regularization term to the objective in Equation (2.4c).

2.1.2 Batching & Noise

Due to the large-scale nature of $\mathbb{D}$ and $f_{\boldsymbol{\theta}}$ in the empirical risk, Equation (2.4b) is usually stochastically approximated through a mini-batch $\mathbb{B} \subseteq \mathbb{D}$, and assessed through a mini-batch loss

$$\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{B}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n) \,. \tag{2.5}$$

In the following, per-sample predictions and losses will often be abbreviated as $\boldsymbol{f}_n := f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ and $\ell_n := \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)$.

Batched Computations

Evaluation of the mini-batch loss in Equation (2.5) can be parallelized (Figure 2.2a). All mini-batch features $\{\boldsymbol{x}_n\}$ are mapped to predictions $\{\boldsymbol{f}_n\}$ by the *same instructions* $f_{\boldsymbol{\theta}}$, and compared in parallel with the labels $\{\boldsymbol{y}_n\}$, resulting in the per-sample losses $\{\ell_n\}$. Hardware accelerators can use this structure to achieve significant speed-up of evaluating the loss, or its derivatives (Section 2.3).

This single-instruction-multiple-data structure of the loss is often baked into ML libraries. Many of their operations natively support batched behavior, *i.e.* accept stacked inputs and assume one, usually the first, axis to correspond to a batch axis. The operation is then applied to all slices along the batch axis (Figure 2.2b).

The concept of `map` in functional programming [73] formalizes applying a function to a collection of inputs, which can be seen as a function transformation. Batched operations can be understood as transformations of the original operation. Recently developed ML libraries [23, 72] make batching explicit by providing a `vmap` interface to automatically vectorize, *i.e.* parallelize, the application of `map`.

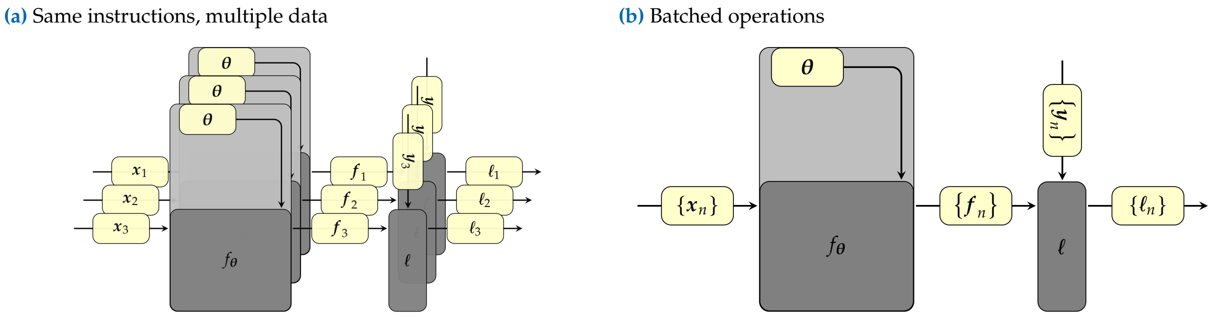


Figure 2.2: Different visualizations of empirical risk computation graphs. (a), The empirical risk is a weighted sum of per-sample losses that are computed independently and with the same instructions, allowing for efficient parallelization. **(b)** Many ML libraries natively support batched operations for efficiency. In code, this batching is often assumed, but not explicitly expressed. The vmap concept allows to make batching explicit: f_θ and ℓ in **(b)** correspond to vectorized versions $\mathrm{vmap}(f)$ and $\mathrm{vmap}(\ell)$ from **(a)**.

Definition 2.1 (vmap) Let $f : \mathbb{X} \to \mathbb{Y}, x \mapsto f(x)$ denote a function. The vectorized `map`, $\mathrm{vmap}(f)$, of f *w.r.t.* its argument x is a function that accepts a collection of inputs and maps each item by f, resulting in a collection of outputs,

$$\mathrm{vmap}: \quad (\mathbb{X} \to \mathbb{Y}) \to (\mathbb{X}^N \to \mathbb{Y}^N) \quad \forall N \in \mathbb{N}$$

such that

$$\mathrm{vmap}(f)(\{x_1, x_2, \dots, x_N\}) = \{f(x_1), f(x_2), \dots, f(x_N)\}\,.$$

Collections are often abbreviated as $\{x_n\} := \{x_1, x_2, \dots, x_N\}$ and $\{f(x_n)\} := \{f(x_1), f(x_2), \dots, f(x_N)\}$. Multi-variate functions can be mapped *w.r.t.* to a subset of arguments.

Making batching explicit with `map`, the mini-batch loss computation uses a vectorized model $\mathrm{vmap}(f_{\boldsymbol{\theta}})$ *w.r.t.* the input features $\boldsymbol{x}$, such that

$$\mathrm{vmap}(f_{\boldsymbol{\theta}})(\{\boldsymbol{x}_n\}) = \{\boldsymbol{f}_n(\boldsymbol{\theta})\}\,. \tag{2.6a}$$

Per-sample losses, which are reduced into the mini-batch loss, are

$$\{\ell_n(\boldsymbol{\theta})\} = \mathrm{vmap}(\ell)(\{\boldsymbol{f}_n(\boldsymbol{\theta})\}, \{\boldsymbol{y}_n\})\,. \tag{2.6b}$$

Numerically, collections of vectors like $\{\boldsymbol{x}_n\}$ *etc.* are represented as matrices, and more generally, collections of r-dimensional arrays are stacked into a $(r+1)$-dimensional arrays with an additional batch axis (*e.g.*, $\{\boldsymbol{x}_n\}$ is a $|\mathbb{B}| \times M$ matrix). These arrays can then be efficiently processed in hardware accelerators.

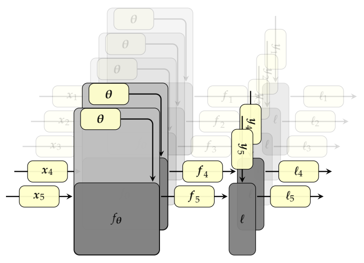


Figure 2.3: Illustration of stochastic sub-sampling: The computation graph of the empirical risk (with five data in this example) is only evaluated on a subset of data (two in this example) to save computations. While this preserves parallelization, the transparent parts are not evaluated, which introduces noise.

Noise

While the sum structure in the loss Equation (2.4b) can be efficiently parallelized, it can also be used for stochastic approximation via sub-sampling (Figure 2.3). The mini-batch loss $\mathcal{L}_{\mathbb{B}}$ in Equation (2.5) is an estimator of the empirical risk $\mathcal{L}_{\mathbb{D}}$ implied by the stochastic sampling procedure of $\mathbb{B}$. Increasing the batch size makes this estimator more precise but more costly to compute.

To see this cost-accuracy trade-off, consider the loss of a single datum ℓ_n where n is uniformly drawn from $\{1, \dots, |\mathbb{D}|\}$. Then, ℓ_n is a random variable, implied by the sampling distribution $n \sim p(n) = \mathcal{U}(\{1, \dots, |\mathbb{D}|\})$, and an unbiased estimator of the empirical risk,

$$\mathbb{E}_{p(n)}[\ell_n(\boldsymbol{\theta})] = \sum_{n=1}^{|\mathbb{D}|} p(n)\ell_n(\boldsymbol{\theta}) = \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})\,, \tag{2.7a}$$

with variance

$$\begin{aligned}\sigma^2 := \mathrm{Var}_{p(n)}[\ell_n(\boldsymbol{\theta})] &= \mathbb{E}_{p(n)}\left[(\ell_n(\boldsymbol{\theta}) - \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}))^2\right] \\ &= \mathbb{E}_{p(n)}\left[\ell_n(\boldsymbol{\theta})^2\right] - \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})^2\,.\end{aligned} \tag{2.7b}$$

Next, consider a batch with $\mathbb{B}|$ randomly drawn samples $\boldsymbol{n} = (n_1, \dots, n_{|\mathbb{B}|})$ from a joint distribution $p(\boldsymbol{n})$, *i.e.* the mean of $\ell_{n_1}, \dots, \ell_{n_{|\mathbb{B}|}}$,

$$\mathcal{L}_{\boldsymbol{n}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|}\sum_{i=1}^{|\mathbb{B}|} \ell_{n_i}(\boldsymbol{\theta})\,. \tag{2.8a}$$

If samples are drawn uniformly *i.i.d.* ($p(\boldsymbol{n}) = \prod_{i=1}^{|\mathbb{B}|} p(n_i)$ with $p(n_i) = \mathcal{U}(\{1, \dots, |\mathbb{D}|\})$), this estimator is also unbiased,

$$\mathbb{E}_{p(\boldsymbol{n})}[\mathcal{L}_{\boldsymbol{n}}(\boldsymbol{\theta})] = \frac{1}{|\mathbb{B}|}\sum_{i=1}^{|\mathbb{B}|} \mathbb{E}_{p(n_i)}[\ell_{n_i}(\boldsymbol{\theta})] = \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})\,, \tag{2.8b}$$

but has smaller variance than the single-sample estimator:

$$\mathrm{Var}_{p(\boldsymbol{n})}[\mathcal{L}_{\boldsymbol{n}}(\boldsymbol{\theta})] = \mathbb{E}_{p(\boldsymbol{n})}[\mathcal{L}_{\boldsymbol{n}}(\boldsymbol{\theta})^2] - \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})^2 = \frac{\sigma^2}{|\mathbb{B}|}\,. \tag{2.8c}$$

Using more samples, *i.e.* a larger $|\mathbb{B}|$, decreases the variance, and thereby reduces noise in the mini-batch loss. The central limit theorem [50] connects the mini-batch estimator to a normal distribution. As the mini-batch size $|\mathbb{B}|$ approaches infinity, $\mathcal{L}_{\boldsymbol{n}}$ converges to a normal distribution with mean and variance from Equations (2.8b) and (2.8c)

$$\lim_{|\mathbb{B}|\to\infty} : \quad \mathcal{L}_{\boldsymbol{n}} \sim \mathcal{N}(\mathcal{L}_{\boldsymbol{n}} \mid \mathcal{L}_{\mathbb{D}}, {}^{\sigma^2}\!/\!_{|\mathbb{B}|})\,. \tag{2.9}$$

While applications rely on finite batch sizes, it is sometimes useful to assume that this Gaussian distribution holds approximately.

Noise in the mini-batch loss propagates to other quantities, like the mini-batch gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$, that are used for applications like training. Therefore, it represents a fundamental challenge for deep learning methods. It can be assessed through higher-order statistical moments such as the variance (the centered second moment, see Section 3.2.5 and Appendix C.3 for examples). Motivated by the central limit theorem Equation (2.9)—if the Gaussian approximation holds sufficiently well—only second-order statistical moments are required.

Like higher-order derivatives of multi-variate functions, higher-order statistical moments of multi-variate random variables such as the mini-batch gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta}) \in \mathbb{R}^D$ (first moment) scale exponentially with dimension D (*e.g.* the $D \times D$ gradient covariance matrix, Section 3.2.5). Therefore, they are challenging to work with using naive approaches.

2.1.3 Probabilistic Interpretation

Risk-based supervised learning is often connected to learning the unknown true distribution $p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$ through a model distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})$ and data $\mathbb{D}$, using estimation techniques to finding a good set of parameters that minimizes a measure of dissimilarity between p_{data} and $p_{\boldsymbol{\theta}}$. The following section links the loss function ℓ and the model $f_{\boldsymbol{\theta}}$ to probabilistic objects. This will be helpful to identify additional structure in the risk minimization problem and use it for structural approximation of higher-order information (*e.g.* the Fisher information matrix, Section 3.2.4), and to motivate probabilistic applications (*e.g.* Laplace approximations, Section 3.3.1).

Connections to Maximum Likelihood Estimation (MLE)

The KL-divergence between the true and the model distribution,

$$\begin{aligned} &D_{\text{KL}}(p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})) \\ &\quad = \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})} \left[\log p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y}) - \log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})\right] , \end{aligned} \tag{2.10a}$$

can be used to measure their dissimilarity. Minimizing the above expression over $\boldsymbol{\theta}$, and dropping parameter-independent terms leads to

$$\begin{aligned} &\underset{\boldsymbol{\theta}}{\text{minimize}}\, D_{\text{KL}}(p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})) \\ \Leftrightarrow\ &\underset{\boldsymbol{\theta}}{\text{minimize}}\, \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})} \left[- \log p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})\right] . \end{aligned} \tag{2.10b}$$

$p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$ is inaccessible and therefore empirically approximated through data $\mathbb{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_n$. Under the *i.i.d.* assumption in the data, p_{data} is replaced with the empirical distribution $p_{\mathbb{D}}$ from Equation (2.4a) and yields the accessible optimization task

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\, \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} - \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_n, \boldsymbol{y}_n) \tag{2.10c}$$

This expression resembles the empirical risk minimization Equation (2.4c) with a specific loss function ℓ that produces the negative log-probability of a datum $(\boldsymbol{x}_n, \boldsymbol{y}_n)$.

Supervised learning only processes features $\boldsymbol{x}$ to predict labels $\boldsymbol{y}$. The probabilistic model $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})$ thus has a more special form, in that it only parameterizes the likelihood $\boldsymbol{y} \mid \boldsymbol{x}$,

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}) . \tag{2.11a}$$

Since only the likelihood contains parameters, Equation (2.10b) simplifies to minimizing the expected negative log-likelihood

$$\begin{aligned} &\underset{\boldsymbol{\theta}}{\text{minimize}}\, \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})} \left[- \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) - \log p(\boldsymbol{x})\right] \\ \Leftrightarrow\ &\underset{\boldsymbol{\theta}}{\text{minimize}}\, \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})} \left[- \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x})\right] \end{aligned}$$

and with empirical approximation through *i.i.d.* data as Equation (2.10c),

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \; \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} -\log p_{\boldsymbol{\theta}}(\boldsymbol{y}_n \mid \boldsymbol{x}_n) \,. \tag{2.11b}$$

This minimization problem corresponds to MLE[7] with a statistical model $p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x})$. This is a specific form of empirical risk minimization where the model's prediction $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ parameterizes a likelihood q for $\boldsymbol{y} \mid \boldsymbol{f}$, and a negative log-likelihood loss function ℓ, *i.e.*

$$p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) = q(\boldsymbol{y} \mid f_{\boldsymbol{\theta}}(\boldsymbol{x}))\,, \tag{2.12a}$$

$$\ell(\boldsymbol{f}, \boldsymbol{y}) = -\log q(\boldsymbol{y} \mid \boldsymbol{f})\,. \tag{2.12b}$$

Both the square loss and softmax cross-entropy loss (Examples 2.1 and 2.2) have such a probabilistic interpretation, see Examples 2.3 and 2.4.

7: Finding the negative log-likelihood's minimum, Equation (2.11b), is equivalent to finding the maximum of the *i.i.d.* data's likelihood

$$p(\mathbb{D} \mid \boldsymbol{\theta}) = \prod_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} p_{\boldsymbol{\theta}}(\boldsymbol{y}_n \mid \boldsymbol{x}_n) p(\boldsymbol{x}_n)\,.$$

The MLE satisfies

$$\begin{aligned} \boldsymbol{\theta}_{\text{MLE}} &= \arg\max p(\mathbb{D} \mid \boldsymbol{\theta}) \\ &= \arg\max \log p(\mathbb{D} \mid \boldsymbol{\theta}) \\ &= \arg\min -\log p(\mathbb{D} \mid \boldsymbol{\theta})\,. \end{aligned}$$

Inserting $p(\boldsymbol{\theta} \mid \mathbb{D})$ and dropping parameter-independent terms recovers the problem Equation (2.11b).

Example 2.3 (Probabilistic interpretation of square loss) The square loss Equation (2.1) is the negative log-likelihood of a Gaussian centered around the model's prediction with diagonal constant covariance,

$$\begin{aligned} &\ell(\boldsymbol{f}, \boldsymbol{y}) = -\log q(\boldsymbol{y} \mid \boldsymbol{f}) \\ \text{with} \quad & q(\boldsymbol{y} \mid f_{\boldsymbol{\theta}}(\boldsymbol{x})) = \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) \end{aligned}$$

where[8] $\boldsymbol{\mu} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ and $\boldsymbol{\Sigma} = {}^{C}\!/_{2}\boldsymbol{I}$.

8: Inserting mean and covariance into the negative log-probability yields,

$$\begin{aligned} &-\log \mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ &= {}^{1}\!/_{2}(\boldsymbol{y} - \boldsymbol{\mu})^{\top} \boldsymbol{\Sigma}^{-1} (\boldsymbol{y} - \boldsymbol{\mu}) \\ &\quad + {}^{1}\!/_{2}\left[\log\det \boldsymbol{\Sigma} + C \log 2\pi\right]\,, \end{aligned}$$

i.e. the square loss Equation (2.1) up to a $\boldsymbol{\theta}$-independent term which does not affect optimization.

Example 2.4 (Probabilistic interpretation of softmax cross-entropy loss) The softmax cross-entropy loss Equation (2.2) is the negative log-likelihood of a multinomial distribution parameterized by the softmax probabilities,

$$\begin{aligned} &\ell(\boldsymbol{f}, y) = -\log q(y \mid \boldsymbol{f}) \\ \text{with} \quad & q(y \mid f_{\boldsymbol{\theta}}(\boldsymbol{x})) = \text{Cat}(y; \boldsymbol{p}) \end{aligned}$$

where[9] $\boldsymbol{p} = \text{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}))$.

9: With p_c denoting the probability to observe class $y = c$,

$$-\log \text{Cat}(y; \boldsymbol{p}) = -\log p_y\,,$$

which is the softmax cross-entropy loss Equation (2.2).

Following the maximum likelihood principle results in the MLE parameter $\boldsymbol{\theta}_{\text{MLE}}$ which satisfies Equation (2.11b) and gives rise to the distribution $p_{\boldsymbol{\theta}_{\text{MLE}}}(\boldsymbol{y} \mid \boldsymbol{x}) = q(\boldsymbol{y} \mid f_{\boldsymbol{\theta}_{\text{MLE}}}(\boldsymbol{x}))$ as an approximation to $p_{\text{data}}(\boldsymbol{y} \mid \boldsymbol{x})$.

Connections to Maximum A Posteriori (MAP) Estimation

MLE maximizes the likelihood $p(\mathbb{D} \mid \boldsymbol{\theta})$. In a probabilistic formulation with a prior $p(\boldsymbol{\theta})$ over the parameters and evidence $p(\mathbb{D}) = \int_{\Theta} p(\mathbb{D} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\theta}$ for the data, one can instead consider the posterior

$$p(\boldsymbol{\theta} \mid \mathbb{D}) = \frac{p(\mathbb{D} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathbb{D})} \tag{2.13}$$

which corrects the prior with data observations. Such a posterior is useful to form probabilistic beliefs over predictions $\boldsymbol{y}_\star$ for new inputs $\boldsymbol{x}_\star$,

$$\begin{aligned} p(\boldsymbol{y}_\star \mid \boldsymbol{x}_\star, \mathbb{D}) &= \int_\Theta p(\boldsymbol{y}_\star \mid \boldsymbol{x}_\star, \mathbb{D}, \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \boldsymbol{x}_\star, \mathbb{D}) \,\mathrm{d}\boldsymbol{\theta} \\ &= \int_\Theta p(\boldsymbol{y}_\star | \boldsymbol{x}_\star, \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \mathbb{D}) \,\mathrm{d}\boldsymbol{\theta}\,. \end{aligned} \tag{2.14}$$

However, this requires integration over the posterior, which itself is almost always intractable and thus needs to be approximated.

The MAP principle approximates the posterior with a delta distribution around the posterior mode, *i.e.* the point of maximum posterior density,

$$p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \delta(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \quad \text{where} \quad \boldsymbol{\theta}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathbb{D})\,. \tag{2.15}$$

It is connected to empirical risk minimization with a regularization term that results from the prior $p(\boldsymbol{\theta})$. To see this, reformulate Equation (2.15) to minimize the negative log-posterior, expand Baye's rule (Equation (2.13)) and neglect the parameter-independent evidence term. Then, apply the same assumptions as for MLE[10], *i.e. i.i.d.* data and $\boldsymbol{\theta}$ only parameterizing the likelihood $\boldsymbol{y} \mid \boldsymbol{x}$. This yields

$$\begin{aligned} \boldsymbol{\theta}_{\text{MAP}} &= \arg\min_{\boldsymbol{\theta}} -\log p(\boldsymbol{\theta} \mid \mathbb{D}) \\ &= \arg\min_{\boldsymbol{\theta}} -\log p(\mathbb{D} \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) \\ &= \arg\min_{\boldsymbol{\theta}} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} -\log p(\boldsymbol{y}_n \mid \boldsymbol{x}_n, \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) \\ &= \arg\min_{\boldsymbol{\theta}} \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} -\log p(\boldsymbol{y}_n \mid \boldsymbol{x}_n, \boldsymbol{\theta}) - \frac{\log p(\boldsymbol{\theta})}{|\mathbb{D}|}\,. \end{aligned} \tag{2.16}$$

10: Mathematically, they translate into

$$\begin{aligned} &p(\mathbb{D} \mid \boldsymbol{\theta}) \\ &\quad= \prod_n p(\boldsymbol{x}_n, \boldsymbol{y}_n \mid \boldsymbol{\theta}) \\ &\quad= \prod_n p(\boldsymbol{y}_n \mid \boldsymbol{x}_n, \boldsymbol{\theta}) p(\boldsymbol{x}_n \mid \boldsymbol{\theta}) \\ &\quad= \prod_n p(\boldsymbol{y}_n \mid \boldsymbol{x}_n, \boldsymbol{\theta}) p(\boldsymbol{x}_n) \end{aligned}$$

with slightly different notation for $\boldsymbol{\theta}$ in comparison to the MLE discussion, as it is now treated probabilistically.

In analogy to Equation (2.12), the first term is an empirical risk (Equation (2.11b)) with negative log-likelihood loss of a distribution q for targets given model predictions,

$$p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{\theta}) = q(\boldsymbol{y} \mid f_{\boldsymbol{\theta}}(\boldsymbol{x})) \quad \text{and} \quad \ell(\boldsymbol{f}, \boldsymbol{y}) = -\log q(\boldsymbol{y} \mid \boldsymbol{f})\,. \tag{2.17}$$

However, this risk is extended by a regularization term from the prior,

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\min_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) + r(\boldsymbol{\theta}) \quad \text{where} \quad r(\boldsymbol{\theta}) = -\frac{\log p(\boldsymbol{\theta})}{|\mathbb{D}|}\,. \tag{2.18}$$

Equations (2.13), (2.16) and (2.18) connect the posterior with the loss via $\log p(\mathbb{D} \mid \boldsymbol{\theta}) = -|\mathbb{D}|\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$ and $\log p(\boldsymbol{\theta}) = -|\mathbb{D}| r(\boldsymbol{\theta})$,

$$\begin{aligned} p(\boldsymbol{\theta} \mid \mathbb{D}) &= \frac{\exp\left[\log p(\mathbb{D} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})\right]}{p(\mathbb{D})} \\ &= \frac{\exp\left\{-|\mathbb{D}|\left[\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) + r(\boldsymbol{\theta})\right]\right\}}{p(\mathbb{D})}\,. \end{aligned} \tag{2.19}$$

It underlines the aforementioned challenges to track the posterior. The exponent is non-linear in $\boldsymbol{\theta}$, and $p(\mathbb{D})$ requires computing an integral.

Equation (2.19) gives rise to posterior approximations that go beyond a delta distribution. The Laplace approximation [93] (Section 3.3.1) also

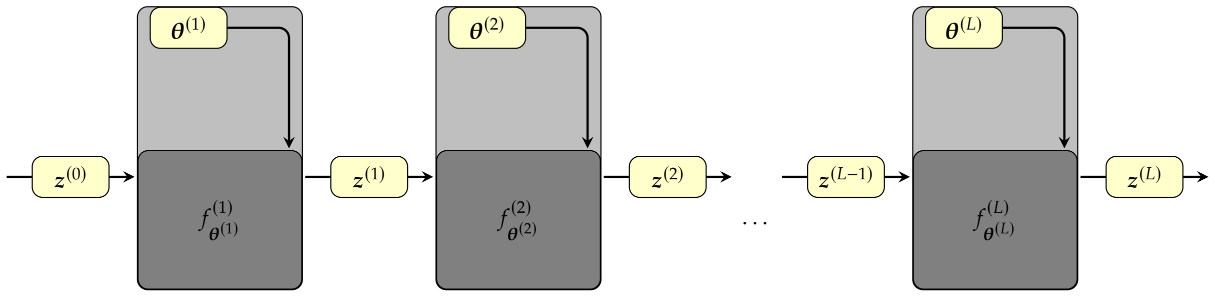


Figure 2.4: **Forward pass of a sequential feedforward neural network (Equation (2.20a)).** The computational graph indicates the data flow and dependencies of intermediate variables.

starts with the MAP estimate, but uses a quadratic Taylor expansion of the log-posterior around $\boldsymbol{\theta}_{\text{MAP}}$ to approximate the posterior by a Gaussian. This quadratic expansion requires higher-order information in form of second-order derivatives, presented in Chapter 3.

2.2 Neural Networks

The previous section focused on structure in the empirical risk—like its sum structure, and interpretations of risk-based learning for specific loss functions—without assumptions about the model $f_{\boldsymbol{\theta}}$. This section introduces structure in the model. While DNNs are generally highly over-parameterized, they usually rely on relatively simple components and construction principles.

This manuscript considers sequential feedforward neural networks that consist of layers. They comprise "classic" architectures like multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), and established architectures like VGG [153], and ResNets [68]. Most of the discussion in this text also applies to other architectures, but would require heavier notation.

2.2.1 Layer-wise Notation

Sequential feedforward neural networks of depth L consist of *modules*, or *layers*, $f^{(l)}_{\boldsymbol{\theta}^{(l)}}, l = 1, \dots, L$, stacked on top of each other such that

$$f_{\boldsymbol{\theta}} = f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ f^{(L-1)}_{\boldsymbol{\theta}^{(L-1)}} \circ \dots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}} \tag{2.20a}$$

They map input features $\boldsymbol{x} =: \boldsymbol{z}^{(0)}$ to predictions $f_{\boldsymbol{\theta}}(\boldsymbol{x}) =: \boldsymbol{z}^{(L)}$ via a sequence of intermediate hidden features $\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(L-1)}$. In a forward pass, a module $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ receives the parental output $\boldsymbol{z}^{(l-1)} \in \mathbb{R}^{h^{(l-1)}}$ and applies an operation with (potentially empty) parameters $\boldsymbol{\theta}^{(l)} \in \mathbb{R}^{d^{(l)}}$,

$$\boldsymbol{z}^{(l)} = f^{(l)}_{\boldsymbol{\theta}^{(l)}}(\boldsymbol{z}^{(l-1)})\,. \tag{2.20b}$$

The output features $\boldsymbol{z}^{(l)} \in \mathbb{R}^{h^{(l)}}$ serve as input to the next layer $l+1$. This builds up dependencies in form of the computational graph shown in Figure 2.4 that maps the leaf nodes $\boldsymbol{z}^{(0)}$ and $\boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(L)}$ to the prediction

$z^{(L)}$. The neural network parameters are often treated as a single vector $\boldsymbol{\theta} \in \mathbb{R}^D$ which results from layer-wise concatenation,

$$\boldsymbol{\theta} = \begin{pmatrix} \boldsymbol{\theta}^{(1)} \\ \boldsymbol{\theta}^{(2)} \\ \vdots \\ \boldsymbol{\theta}^{(L)} \end{pmatrix} . \tag{2.22}$$

To simplify the presentation, Equation (2.20b) assumes vector-shaped quantities. However, many neural networks process higher-dimensional data like images, represented by tensors. Sometimes, the tensor structure is convenient to work with. One can convert between the tensor and vector view without loss of generality by introducing conventions for tensor flattening (Definition 2.2) and reshaping (Definition 2.3); after all, multi-dimensional arrays are represented in a vector format in memory.

However, there exist different flattening conventions. Implementations often favor row-major ordering. This manuscript uses (the more common in literature) column-major order as it allows for elegant generalizations of derivative concepts for multi-variate functions, like the Jacobian (Definition 2.5), the Hessian (Definition 3.2), and their chain rules (Theorems 2.2 and 3.1). To translate analytical results into implementations, it is crucial to be aware of these differing conventions.

Definition 2.2 (Tensor flattening) Let $\mathsf{A} \in \mathbb{R}^{n_1 \times n_2 \times \ldots, \times n_m}$ denote a tensor of rank m with dimensions $n_1, n_2, \ldots, n_m$. The flattened tensor $\operatorname{vec}(\mathsf{A}) \in \mathbb{R}^{n_1 n_2 \cdots n_m}$ is a vector that concatenates A's elements in a first-index-varies-fastest fashion,

$$\operatorname{vec}(\mathsf{A}) = \begin{pmatrix} \mathsf{A}_{1,1,1,\ldots,1} \\ \mathsf{A}_{2,1,1,\ldots,1} \\ \vdots \\ \mathsf{A}_{n_1,1,1,\ldots,1} \\ \mathsf{A}_{1,2,1,\ldots,1} \\ \vdots \\ \mathsf{A}_{n_1,2,1,\ldots,1} \\ \vdots \\ \mathsf{A}_{n_1,n_2,n_3,\ldots,n_m} \end{pmatrix} . \tag{2.21}$$

The matrix case $m = 2$ corresponds to column-stacking. Flattening a vector $\boldsymbol{a}$ leaves it unaffected, *i.e.* $\operatorname{vec}(\boldsymbol{a}) = \boldsymbol{a}$.

Definition 2.3 (Vector reshaping) Let $\boldsymbol{a} \in \mathbb{R}^{n_1 n_2 \cdots n_m}$ be a vector. Reshaping that vector into a rank-m tensor of shape $S = (n_1, n_2, \ldots, n_m)$ happens by filling $\boldsymbol{a}$'s elements into the tensor in a first-index-varies-fastest fashion,

$$\mathsf{A} = \operatorname{reshape}_S(\boldsymbol{a}) \tag{2.23a}$$

with elements

$$\begin{aligned} \mathsf{A}_{1,1,1,\ldots,1} &= a_1 , \\ \mathsf{A}_{2,1,1,\ldots,1} &= a_2 , \\ &\vdots \\ \mathsf{A}_{n_1,1,1,\ldots,1} &= a_{n_1} , \\ \mathsf{A}_{1,2,1,\ldots,1} &= a_{n_1+1} , \\ &\vdots \\ \mathsf{A}_{n_1,2,1,\ldots,1} &= a_{2n_1} , \\ &\vdots \\ \mathsf{A}_{n_1,n_2,n_3,\ldots,n_m} &= a_{n_1 n_2 n_3 \cdots n_m} . \end{aligned} \tag{2.23b}$$

The matrix case $m = 2$ corresponds to column-filling. With tensor flattening (Definition 2.2) this allows to define tensor reshaping: A tensor B_1 of shape S_1 is rearranged into any tensor B_2 of compatible shape S_2 by first flattening, then reshaping it, *i.e.* $\mathsf{B}_2 := \operatorname{reshape}_{S_2}(\mathsf{B}_1) := \operatorname{reshape}_{S_2}(\operatorname{vec} \mathsf{B}_1)$.

2.2.2 Modularity & Common Operations

An important strength of deep learning is its *modularity*. ML libraries provide a large number of operations, or modules, that can be combined in almost arbitrary ways through function composition, like in Equation (2.20a). Training the resulting models with first-order methods remains simple because their gradient can be automatically computed via AD (Section 2.3). New operations can easily be added because its implementation is decoupled to the modular level.

Modules are vaguely defined. Often, multiple operations that form a logical processing unit in a neural network are grouped into a single module, *e.g.* an MLP layer combines affine transformation and elementwise activation (see below). In extreme cases, even an entire neural network can be considered a single module that can be used in other neural networks; *e.g.* the neural network in Figure 2.1 resembles a single layer in Figure 2.4 and could act as one layer in a larger network.

For theoretical analyses, it is preferable to consider units with a small number of operations as modules. This, however, is inconvenient for constructing large architectures, where many operations are grouped into higher-level units. This text adapts a rather fine-grained view on modules that is close to their implementation in ML libraries like PyTorch.

A common categorization for modules distinguishes trainable functions with parameters, and parameter-free operations. Table 2.1 lists the forward passes of common operations that will be illustrated in the following presentation of network architectures. To distringuish more clearly between input and output of an operation, the notation uses the symbols

Table 2.1: Forward pass for common modules used in feedforward networks. Input and output are denoted $\boldsymbol{x}, \boldsymbol{z}$ rather than $\boldsymbol{z}^{(l)}, \boldsymbol{z}^{(l+1)}$ to avoid clutter. $\boldsymbol{I}$ is the identity matrix. Bold upper-case symbols $(\boldsymbol{W}, \boldsymbol{X}, \boldsymbol{Z}, \dots)$ denote matrices and bold upper-case sans serif symbols $(\mathsf{W}, \mathsf{X}, \mathsf{Z}, \dots)$ denote tensors. See Appendices A.2 to A.4 for details, and Tables 2.2 and 4.1 for extended versions of this table for the backward, and Hessian backward, pass.

OPERATION	FORWARD
Matrix-vector multiplication	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}$
Matrix-matrix multiplication	$\boldsymbol{Z}(\boldsymbol{X}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{X}$
Addition	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{b}) = \boldsymbol{x} + \boldsymbol{b}$
Elementwise activation	$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})$, s.t. $z_i(\boldsymbol{x}) = \phi(x_i)$
Skip-connection	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{x} + \boldsymbol{s}(\boldsymbol{x}, \boldsymbol{\theta})$
Reshape/view	$\mathsf{Z}(\mathsf{X}) = \operatorname{reshape}(\mathsf{X})$
Index select/map π	$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{\Pi}\boldsymbol{x}\,, \Pi_{j,\pi(j)} = 1\,,$
Convolution	$\mathsf{Z}(\mathsf{X}, \mathsf{W}) = \mathsf{X} \star \mathsf{W}\,,$ $\boldsymbol{Z}(\boldsymbol{W}, [\![\mathsf{X}]\!]) = \boldsymbol{W}[\![\mathsf{X}]\!]\,,$
Square loss	$\ell(\boldsymbol{f}, \boldsymbol{y}) = {}^1\!/\!_C(\boldsymbol{y} - \boldsymbol{f})^\top(\boldsymbol{y} - \boldsymbol{f})$
Softmax cross-entropy	$\ell(\boldsymbol{f}, y) = -\operatorname{onehot}(y)^\top \log[\boldsymbol{p}(\boldsymbol{f})]$

$\boldsymbol{x}, \boldsymbol{z}$ for module input and output instead of $\boldsymbol{z}^{(l-1)}, \boldsymbol{z}^{(l)}$, and neglects the layer superscript for the parameters, writing $\boldsymbol{\theta}$ instead of $\boldsymbol{\theta}^{(l)}$.

Deep Linear Networks & Multi-layer Perceptrons (MLPs)

Linear layers process inputs $\boldsymbol{x}$ by affine transformation, *i.e.* multiplication with a weight matrix $\boldsymbol{W}$, followed by addition of a bias vector $\boldsymbol{b}$,

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \qquad \text{where} \qquad \boldsymbol{\theta} = \left((\operatorname{vec}\boldsymbol{W})^\top \quad \boldsymbol{b}^\top\right)^\top\,. \tag{2.24}$$

They are also referred to as *fully-connected* layers, because each output z_i depends on all inputs $\boldsymbol{x}$ through $\boldsymbol{W}_{i,:}$ and b_i.

Deep linear networks (Example 2.5) consist of only linear layers and are of interest for analytical studies [*e.g.* 16, 112, 143] as they are somewhat tractable. They describe a linear feature map, *i.e.* a linear function *w.r.t.* the input $\boldsymbol{z}^{(0)}$, that is non-linear in the parameters. Therefore, such networks are as expressive as a single linear layer, but highly overparameterized.

Example 2.5 (Deep linear network) In notation of Equation (2.20), a deep linear network of depth L reads

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{z}^{(\ell-1)} + \boldsymbol{b}^{(l)}$$

where

$$\boldsymbol{\theta}^{(l)} = \left(\left(\operatorname{vec}\boldsymbol{W}^{(l)}\right)^\top \quad \boldsymbol{b}^{(l)\top}\right)^\top, \quad l = 1, \dots, L\,.$$

A common technique to turn a deep linear network into a non-linear feature map is to interlace affine transformations with non-linear activations [136]. An *activation layer* $\boldsymbol{\phi}$ acts elementwise on its input, *i.e.* applies the same function ϕ to each input element,

$$\boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x}) \qquad \text{such that} \qquad z_i = \phi(x_i)\,.$$

There exist many activations (ReLU, sigmoid, tanh, *etc.* [60, Chapter 6]), and recent works proposing new choices (*e.g.* squared ReLU [156]).

Multi-layer perceptrons (MLPs, Example 2.6) combine affine transformation and activation in a layer. Activation functions $\boldsymbol{\phi}^{(l)}$ may vary between layers, but are often chosen identically, with no activation in the last layer. One way to interpret this design is that the mapping $\boldsymbol{z}^{(0)} \mapsto \boldsymbol{z}^{(L-1)}$ acts as non-linear feature transformation, and the last layer $\boldsymbol{z}^{(L-1)} \mapsto \boldsymbol{z}^{(L)}$ as linear classifier for the learned features.

Example 2.6 (Multi-layer perceptron (MLP)) In terms of Equation (2.20), an MLP of depth L reads

$$\boldsymbol{z}^{(l)} = \boldsymbol{\phi}^{(l)}\left(\boldsymbol{W}^{(l)}\boldsymbol{z}^{(\ell-1)} + \boldsymbol{b}^{(l)}\right),$$

where

$$\boldsymbol{\theta}^{(l)} = \left(\left(\operatorname{vec}\boldsymbol{W}^{(l)}\right)^\top \quad \boldsymbol{b}^{(l)\top}\right)^\top, \quad l = 1, \dots, L\,, \quad \boldsymbol{\phi}^{(L)} = \operatorname{id}$$

and id denotes the identity.

Convolutional Neural Networks (CNNs)

CNNs represent an important neural network architecture revolution and were the first class of deep neural network to beat "classical" methods on the ImageNet competition [41, 91, 138]. Broadly speaking, such networks contain convolutional layers with trainable parameters.

Convolutional layers: Convolutions process multi-channel input features such as images and are parameterized by a kernel that can be imagined as a filter for patterns like edges, corners, *etc.* During the convolution operation, the kernel slides over the input features and produces an output element by contraction with the overlapping elements of the image. In most cases, each output channel is also shifted by a bias parameter which will be neglected in this presentation for simplicity (detailed discussion in Appendix A.4, example in Equation (A.8)). Because the kernel moves over the image, it can detect patterns irrespective of their position. The process can be adjusted with various hyperparameters, such as stride, padding, groups, dilation (see [45] for a visual guide).

In contrast to the linear layer (Equation (2.24)) where each output is connected to all inputs via independent rows of the weight matrix, the parameters in the kernel are shared across all outputs. Therefore, convolutions usually require less parameters than fully-connected layers.

Nevertheless, both layers are related because convolution is a linear operation and can therefore be regarded as matrix multiplication. Due to the weight sharing, this matrix is structured by the kernel elements[11]. Alternatively, one can stack patches—input elements that overlap with the kernel at each stage—into columns of a matrix, which yields the unfolded input, denoted by $[\![\mathbf{X}]\!]$ in Table 2.1. Then, convolution is a matrix-matrix product between a matrix reshape of the kernel and the unfolded input [29] (see Figure A.3c for an illustration).

11: *E.g.*, in the one-dimensional case, the convolution of two vectors can be computed by expanding one into a Toeplitz matrix, and multiplying that onto the second vector [171].

Padding & pooling layers: Convolutions are often combined with other modules. *Padding layers* add pixels around the outer dimensions of an image, which helps to reduce information loss at the image boundaries during convolution. *Pooling layers* down-sample images by summarizing patches of pixels and reduce the number of hidden features. Similar to convolution, pooling considers patches of an input image. Two common summary operations are per-channel averaging and taking the per-channel maximum. They give rise to maximum and average pooling.

One can interpret padding and pooling as scatter operations, realized by multiplication with a sparse, binary matrix $\mathbf{\Pi}$ (compare Table 2.1:

- For padding, $\mathbf{\Pi}$ does not dependent on the input, but only its shape and the hyperparameters. A row is empty if its index corresponds to the padded area, and otherwise contains a one at the element's index to be copied from the input.
- For maximum pooling, $\mathbf{\Pi}$ depends on the input. Each row contains a one at the index of the element with maximum value in the patch.
- For average pooling, $\mathbf{\Pi}$ does not dependent on the input, but only its shape and the hyperparameters. Each row contains the inverse patch size at indices of the elements in the current patch.

Residual Networks (ResNets)

The inclusion of skip (or residual) connections [68] represents another revolution in the design of CNNs, and enabled training of deeper architectures, with 100 or even 1000 layers. This lead to improved performance of such CNNs on tasks like ImageNet [41, 138]. Skip connections branch off a hidden feature and feed it back after the residual block $\boldsymbol{s}(\boldsymbol{x}, \boldsymbol{\theta})$,

$$\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{s}(\boldsymbol{x}, \boldsymbol{\theta})\,.$$

This can be seen as learning a small modification $\boldsymbol{s}(\boldsymbol{x}, \boldsymbol{\theta})$—a residual function—for $\boldsymbol{x}$; hence the name *residual connection*.

Closing Remarks & Sources of Non-linearity

This short overview of common neural network layers and architectures is, of course, incomplete. Other famous layers include dropout [158], recursive [33, 47, 71], normalization [79, 174], attention layers [167], *etc.*

An interesting observation about the operations in Table 2.1 is that most of them are linear (linear, convolution, padding, and average pooling layers) or piece-wise linear (maximum pooling and ReLU activation layer) *w.r.t.* both their input and parameters. This implies that their second- and higher-order derivatives vanish. Non-linearity is often only introduced by activation layers (elementwise) and loss functions (after the model's forward pass). While the properties of these components are not inherited by the entire neural network, they give rise to structure in a network's higher-order derivatives, see *e.g.* Chapter 4.

2.3 Automatic Differentiation

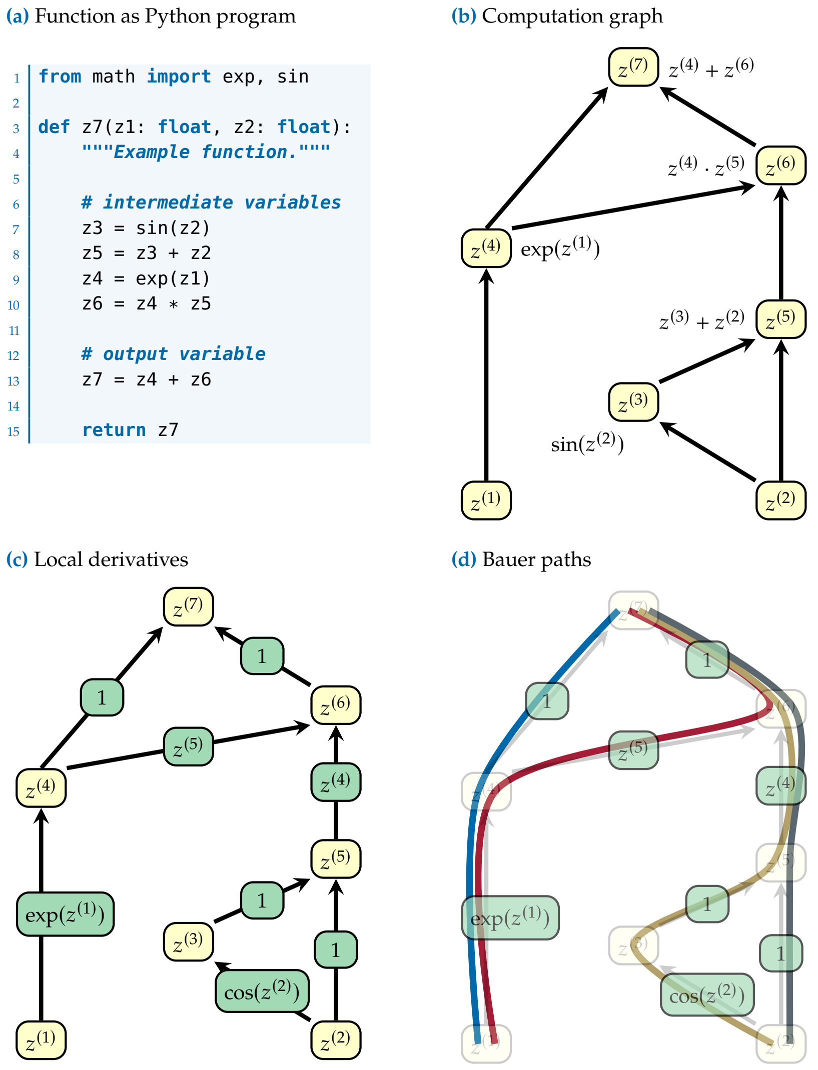
Together with empirical risk minimization and neural networks, the last ingredient in the ML pipeline, Procedure 1.1, is computing the gradient. Contemporary methods to train neural networks (Section 3.1) rely on this quantity. ML libraries compute it via their built-in automatic differentiation (AD), built around the famous backpropagation algorithm [137], described in more detail here.

2.3.1 Foundations

An Example & Path Interpretation

Given a program that evaluates a function, AD produces a program to evaluate its derivative. The general idea is to consider the function as composition of atomic operations (*e.g.* addition, multiplication, . . .) with known derivatives. To automatically compute the derivatives, one needs to track the dependencies between intermediate variables and combine their derivatives using the chain rule.

Figure 2.5 illustrates the basic principles of AD for an example from [119]. The starting point is a function defined by code (Figure 2.5a). Such a function transforms input to output variables through atomic operations and builds up intermediate variables along its execution. The relation between these operations are described by a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a set of nodes $\mathcal{V} = \{z^{(1)}, z^{(2)}, \dots\}$ and a set of edges $\mathcal{E} = \{(z^{(i_1)}, z^{(j_1)}), (z^{(i_2)}, z^{(j_2)}), \dots\}$ where $(z^{(i)}, z^{(j)})$ denotes a directed edge from node $z^{(i)}$ to $z^{(j)}$ (Figure 2.5b).


(a) Function as Python program
(b) Computation graph
$z^{(7)}$ $z^{(4)} + z^{(6)}$
$z^{(4)} \cdot z^{(5)}$ $z^{(6)}$
$z^{(4)}$ $\exp(z^{(1)})$
$z^{(3)} + z^{(2)}$ $z^{(5)}$
$z^{(3)}$
$\sin(z^{(2)})$
$z^{(1)}$ $z^{(2)}$
(c) Local derivatives
1
$z^{(5)}$
$\exp(z^{(1)})$
$\cos(z^{(2)})$
(d) Bauer paths

```python
from math import exp, sin

def z7(z1: float, z2: float):
    """Example function."""

    # intermediate variables
    z3 = sin(z2)
    z5 = z3 + z2
    z4 = exp(z1)
    z6 = z4 * z5

    # output variable
    z7 = z4 + z6

    return z7
```

Figure 2.5: Basic AD principles [119]. **(a)** Example input program represented by Python code. The atomic operations combined via composition are addition, multiplication, exponential function, and sine. The result $z^{(7)}$ is computed from the inputs $z^{(1)}, z^{(2)}$ through intermediates $z^{(3)}, z^{(4)}, z^{(5)}, z^{(6)}$,

$$z^{(7)} = \exp(z^{(1)}) + \exp(z^{(1)}) \left(\sin(z^{(2)}) + z^{(2)}\right) .$$

(b) Representation as computation graph to track dependencies between the intermediate variables on the level of atomic operations. **(c)** Computing derivatives relies on local derivatives $\partial z^{(j)} / \partial z^{(i)}$ on edges $(z^{(i)}, z^{(j)})$, which need to be accumulated according to the chain rule. **(d)** Interpretation of the chain rule as sum over path products. Computing the derivatives of a node *w.r.t.* to another node in the graph requires summing the path product of local derivatives for all paths that connect them. In detail:

$$\begin{aligned}
\frac{\partial z^{(7)}}{\partial z^{(1)}} &= \frac{\partial z^{(7)}}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial z^{(1)}} + \frac{\partial z^{(7)}}{\partial z^{(6)}} \frac{\partial z^{(6)}}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial z^{(1)}} \\
&= \exp(z^{(1)}) + z^{(5)} \exp(z^{(1)}) \\
&= \exp(z^{(1)}) \\
&\quad + \left(\sin(z^{(2)}) + z^{(2)}\right) \exp(z^{(1)}) , \\
\frac{\partial z^{(7)}}{\partial z^{(2)}} &= \frac{\partial z^{(7)}}{\partial z^{(6)}} \frac{\partial z^{(6)}}{\partial z^{(5)}} \frac{\partial z^{(5)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial z^{(2)}} \\
&\quad + \frac{\partial z^{(7)}}{\partial z^{(6)}} \frac{\partial z^{(6)}}{\partial z^{(5)}} \frac{\partial z^{(5)}}{\partial z^{(2)}} \\
&= z^{(4)} \cos(z^{(2)}) + z^{(4)} \\
&= \exp(z^{(1)}) \cos(z^{(2)}) + \exp(z^{(1)}) .
\end{aligned}$$

To compute derivatives, the local derivatives on edges (Figure 2.5c) are combined according to the chain rule. For $\partial z^{(j)} / \partial z^{(i)}$ between two variables in the graph, all paths connecting them need to be considered. A path between two nodes $z^{(i)}$ and $z^{(j)}$ is a sequence of edges that connect them: starting from $z^{(i)}$, following the edges in a path leads to $z^{(j)}$. Let $[z^{(i)} \longrightarrow z^{(j)}]$ denote the set of paths connecting $z^{(i)}$ to $z^{(j)}$. Then the derivative is the sum of path products of local derivatives (Figure 2.5d),

$$\frac{\partial z^{(j)}}{\partial z^{(i)}} = \sum_{p \in [z^{(i)} \longrightarrow z^{(j)}]} \prod_{(z^{(k)}, z^{(l)}) \in p} \frac{\partial z^{(l)}}{\partial z^{(k)}} . \tag{2.25}$$

The path formulation goes back to Bauer [12].

The Jacobian Matrix & Its Chain Rule

For the computation graph of a neural network's empirical risk, tracking dependencies between variables at a scalar level would result in a considerable book-keeping overhead due to the large number of connections. This can be circumvented by using vector-valued (or tensor-valued) nodes $z^{(1)}, z^{(2)}, \ldots$ and tracking edges between vectors (or tensors), see

Figure 2.6. However, this requires a generalization of Equation (2.25) to multi-variate nodes. The accumulation is efficiently expressed as matrix multiplication by arranging partial derivatives into *Jacobians*.

Definition 2.4 (Jacobian) Let $\boldsymbol{b} : \mathbb{R}^n \to \mathbb{R}^m, \boldsymbol{a} \mapsto \boldsymbol{b}(\boldsymbol{a})$ be a differentiable vector-to-vector function. The Jacobian $\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}(\boldsymbol{a})$ of $\boldsymbol{b}$ *w.r.t.* $\boldsymbol{a}$ is an $m \times n$ matrix with partial derivatives,

$$\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}(\boldsymbol{a}) = \frac{\partial \boldsymbol{b}(\boldsymbol{a})}{\partial \boldsymbol{a}^\top}, \quad \text{with} \quad [\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}(\boldsymbol{a})]_{i,j} = \frac{\partial b_i(\boldsymbol{a})}{\partial a_j}. \tag{2.26}$$

Matrix- and tensor-valued functions require flattening their arguments into vectors[12]. For a vector-to-scalar function $b(\boldsymbol{a})$, *i.e.* $m = 1$, the Jacobian has one row that contains the gradient, $[\mathrm{J}_{\boldsymbol{a}}b(\boldsymbol{a})]^\top = \nabla_{\boldsymbol{a}} b$. The gradient will often be denoted by $\boldsymbol{g}$. *E.g.* $\boldsymbol{g}_{p_\text{data}}(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}}\mathcal{L}_{p_\text{data}}(\boldsymbol{\theta})$ for the gradient of the population risk Equation (2.3a), and $\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$ for the gradient of the empirical risk Equation (2.4b) on a dataset $\mathbb{D}$ (with $\mathbb{D} = \mathbb{D}_\text{train}, \mathbb{B}$ for the train loss and the mini-batch gradient).

12: Definition 2.4 assumes vector-valued functions. With the flattening convention Definition 2.2, it generalizes to tensor-valued functions as follows:

Definition 2.5 (Generalized Jacobian) Let $\boldsymbol{B} : \mathbb{R}^{n\times q} \to \mathbb{R}^{m\times p}, \boldsymbol{A} \mapsto \boldsymbol{B}(\boldsymbol{A})$ be a differentiable matrix-to-matrix function. The Jacobian $\mathrm{J}_{\boldsymbol{A}}\boldsymbol{B}(\boldsymbol{A})$ of $\boldsymbol{B}$ *w.r.t.* $\boldsymbol{A}$ is an $mp \times nq$ matrix

$$\mathrm{J}_{\boldsymbol{A}}\boldsymbol{B}(\boldsymbol{A}) = \frac{\partial \operatorname{vec} \boldsymbol{B}(\boldsymbol{A})}{\partial (\operatorname{vec} \boldsymbol{A})^\top} \tag{2.27a}$$

with entries

$$[\mathrm{J}_{\boldsymbol{A}}\boldsymbol{B}(\boldsymbol{A})]_{i,j} = \frac{\partial\, [\operatorname{vec} \boldsymbol{B}(\boldsymbol{A})]_i}{\partial\, [\operatorname{vec} \boldsymbol{A}]_j} \tag{2.27b}$$

and the flattening operation vec from Definition 2.2 [103, Chapter 9.4]. The analogous tensor case $(\boldsymbol{A}, \boldsymbol{B}) \to (\mathsf{A}, \mathsf{B})$ requires lengthy notation and is therefore omitted.

In the context of neural networks, the most common occurrences of Definition 2.5 involve vector-to-vector functions $f : \mathbb{R}^n \to \mathbb{R}^m, \boldsymbol{x} \mapsto f(\boldsymbol{x})$ with

$$\mathrm{J}_{\boldsymbol{x}} f(\boldsymbol{x}) = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}^\top}.$$

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

$$\mathrm{J}_{\boldsymbol{X}} f(\boldsymbol{X}) = \frac{\partial f(\boldsymbol{X})}{\partial (\operatorname{vec} \boldsymbol{X})^\top},$$

where $\boldsymbol{X}$ might correspond to the $\mathbb{R}^{m\times q}$ weight matrix of a linear layer. See Table 2.2 for an overview.

In the vector-valued case, one must accumulate Jacobians through matrix multiplies instead of scalar multiplications to compute derivatives,

$$\mathrm{J}_{\boldsymbol{z}^{(j)}}\boldsymbol{z}^{(i)} = \sum_{p \in [\boldsymbol{z}^{(i)} \to \boldsymbol{z}^{(j)}]} \prod_{(\boldsymbol{z}^{(k)}, \boldsymbol{z}^{(l)}) \in p} \mathrm{J}_{\boldsymbol{z}^{(k)}}\boldsymbol{z}^{(l)}(\boldsymbol{z}^{(k)}). \tag{2.28}$$

The product term generalizes the chain rule to vector-valued functions.

Theorem 2.1 (Jacobian chain rule) Let $\boldsymbol{b} : \mathbb{R}^n \to \mathbb{R}^m, \boldsymbol{a} \mapsto \boldsymbol{b}(\boldsymbol{a})$ and $\boldsymbol{c} : \mathbb{R}^m \to \mathbb{R}^r, \boldsymbol{b} \mapsto \boldsymbol{c}(\boldsymbol{b})$ be differentiable vector-to-vector functions. Consider their composition $\boldsymbol{d} = \boldsymbol{c} \circ \boldsymbol{b} : \mathbb{R}^n \to \mathbb{R}^r, \boldsymbol{a} \mapsto \boldsymbol{d}(\boldsymbol{a}) = \boldsymbol{c}(\boldsymbol{b}(\boldsymbol{a}))$. The composition's Jacobian $\mathrm{J}_{\boldsymbol{a}}\boldsymbol{d}(\boldsymbol{a}) \in \mathbb{R}^{r\times n}$ is related to the composite Jacobians via

$$\mathrm{J}_{\boldsymbol{a}}\boldsymbol{d}(\boldsymbol{a}) = [\mathrm{J}_{\boldsymbol{b}}\boldsymbol{c}(\boldsymbol{b})]\,\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}(\boldsymbol{a}). \tag{2.29}$$

This can be generalized to tensor-valued functions[13].

13: Proper arrangement of partial derivatives leads to a generalized Jacobian chain rule for matrices/tensors:

Theorem 2.2 (Generalized Jacobian chain rule) Let $\boldsymbol{B} : \mathbb{R}^{n\times q} \to \mathbb{R}^{m\times p}$ and $\boldsymbol{C} : \mathbb{R}^{m\times p} \to \mathbb{R}^{r\times s}$ be differentiable matrix-to-matrix functions. Let $\boldsymbol{D} = \boldsymbol{C} \circ \boldsymbol{B} : \mathbb{R}^{n\times q} \to \mathbb{R}^{r\times s}, \boldsymbol{A} \to \boldsymbol{D}(\boldsymbol{A}) = \boldsymbol{C}(\boldsymbol{B}(\boldsymbol{A}))$ be their composition. Then,

$$\mathrm{J}_{\boldsymbol{A}}\boldsymbol{D}(\boldsymbol{A}) = [\mathrm{J}_{\boldsymbol{B}}\boldsymbol{C}(\boldsymbol{B})]\,\mathrm{J}_{\boldsymbol{A}}\boldsymbol{B}(\boldsymbol{A}) \tag{2.30}$$

with the generalized Jacobian Definition 2.5 [103, Chapter 5.15]. The tensor case $(\boldsymbol{D}, \boldsymbol{C}, \boldsymbol{B}, \boldsymbol{A}) \to (\mathsf{D}, \mathsf{C}, \mathsf{B}, \mathsf{A})$ is analogous.

Jacobian Accumulation (Automatic Differentiation Modes)

Given the computation graph $\mathcal{G}$ of a function to be differentiated, Equation (2.28) describes the operations that need to be performed. But there are different schedules for carrying out these computations, with differing performance: *e.g.*, it is possible to share accumulated derivative products between paths that share subpaths (like paths ● and ● in Figure 2.5d). And for a single path in the vector case, the optimal contraction order of the Jacobian matrix chain (one summand of Equation (2.28)) depends on the dimension of the nodes (Example 2.7). The following Jacobian accumulation schedules are of specific interest for AD:

- **Forward accumulation, or forward mode AD,** starts at the leafs, *i.e.* the nodes *w.r.t.* which the function is differentiated. Jacobians are accumulated in the same order as the function evaluation.

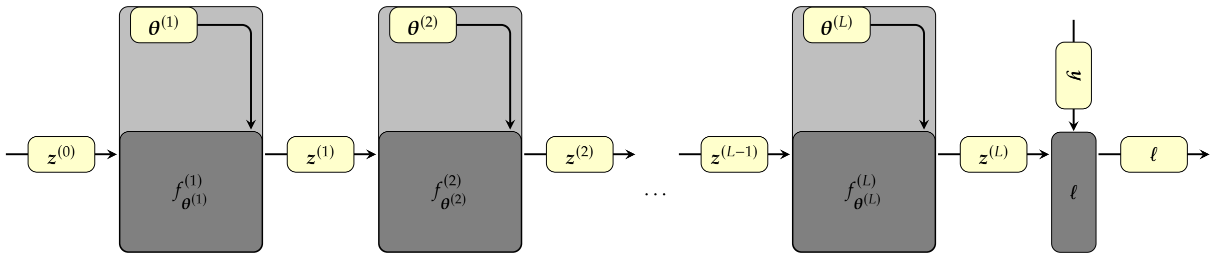


Figure 2.6: **Computation graph of a sequential feedforward neural network's loss for a single datum from Equation (2.31a).**

- **Reverse accumulation, or reverse mode AD [62, 99],** starts at the root, *i.e.* the variable that is differentiated. Jacobians are accumulated from root to leaf nodes, traversing the graph backwards. This is often called a *backward pass*.
- **Optimal Jacobian accumulation** computes derivatives according to the optimal schedule which usually traverses the computation graph in a nontrivial fashion. For arbitrary computation graphs, finding this schedule is NP-hard [114].

Due to the specific structure of computation graphs in deep learning, reverse mode AD is often more practical than forward accumulation. This is outlined in in the following section, that illustrates reverse mode for differentiation of a neural network's loss *w.r.t.* its parameters, leading to the famous backpropagation algorithm [137].

2.3.2 Gradient Backpropagation

Gradient backpropagation [137] enables efficient differentiation of the training objective in deep learning. It is the central algorithm of popular ML libraries with built-in AD. This section presents backpropagation for chain-structured computation graphs (see [60, Chapter 6] for the general case) like the loss of a sequential feedforward neural network. Starting from the loss of a single datum, the goal is to show that ML libraries combine AD and batching to maximize efficiency. But this limits their functionality to computing the gradient, ignoring *e.g.* the per-sample structure in the loss. Alleviating this limitation to compute richer information (Chapter 3) using the existing implementation of gradient backpropagation is a main goal of Part II in this thesis.

Loss of a Single Datum

Consider the loss implied by a single datum $(\boldsymbol{x}, \boldsymbol{y})$, a loss function ℓ, and a neural network $f_{\boldsymbol{\theta}}$ depicted in Figure 2.6,

$$\ell(\boldsymbol{\theta}) = \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) \qquad \text{with} \qquad f_{\boldsymbol{\theta}} = f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ f^{(L-1)}_{\boldsymbol{\theta}^{(L-1)}} \circ \ldots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}} \,. \tag{2.31a}$$

Its computation graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ has nodes

$$\mathcal{V} = \left\{\boldsymbol{\theta}^{(l)}\right\}_{l=1}^{L} \cup \left\{\boldsymbol{z}^{(l)}\right\}_{l=0}^{L} \cup \{\boldsymbol{y}, \ell\} \tag{2.31b}$$

and edges

$$\mathcal{E} = \left\{(z^{(l-1)}, z^{(l)})\right\}_{l=1}^{L} \cup \left\{(\boldsymbol{\theta}^{(l)}, z^{(l)})\right\}_{l=1}^{L} \cup \left\{(z^{(L)}, \ell), (\boldsymbol{y}, \ell)\right\} . \tag{2.31c}$$

Each edge implies a Jacobian, categorized as one of the following:

- The *input-output* Jacobian $\mathrm{J}_{z^{(l-1)}} z^{(l)}(z^{(l-1)})$ of a module l.
- The *parameter-output* Jacobian $\mathrm{J}_{\boldsymbol{\theta}^{(l)}} z^{(l)}(\boldsymbol{\theta}^{(l)})$ of a module l.
- The *prediction-loss* Jacobian $\mathrm{J}_{z^{(L)}} \ell(z^{(L)})$ has one column and will be written as gradient of the loss *w.r.t.* the model prediction, $\mathrm{J}_{z^{(L)}} \ell(z^{(L)}) = [\nabla_{z^{(L)}} \ell(z^{(L)})]^\top$.

The goal is to compute *parameter-loss* Jacobians, *i.e.* gradients of the loss *w.r.t.* parameters $\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \ell(\boldsymbol{\theta}^{(l)}) = [\nabla_{\boldsymbol{\theta}^{(l)}} \ell(\boldsymbol{\theta}^{(l)})]^\top$ for all layers $l = 1, \dots, L$.

First, consider only the gradient $\nabla_{\boldsymbol{\theta}^{(l)}} \ell$ of one parameter $\boldsymbol{\theta}^{(l)}$. Equation (2.28) requires identifying all paths connecting $\boldsymbol{\theta}^{(l)}$ to ℓ. Due to the computation graph's chain structure, this is only a single path,

$$[\boldsymbol{\theta}^{(l)} \to \ell] = \{p\} \tag{2.32a}$$

with

$$p = \left(\left(\boldsymbol{\theta}^{(l)}, z^{(l)}\right), \left(z^{(l)}, z^{(l+1)}\right), \dots \left(z^{(L-1)}, z^{(L)}\right), \left(z^{(L)}, \ell\right)\right) . \tag{2.32b}$$

Plugging this into Equation (2.28) simplifies to

$$\underbrace{\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \ell(\boldsymbol{\theta})}_{1\times d^{(l)}} = \underbrace{[\mathrm{J}_{z^{(L)}} \ell]}_{1\times h^{(L)}} \underbrace{\left[\mathrm{J}_{z^{(L-1)}} z^{(L)}\right]}_{h^{(L)}\times h^{(L-1)}} \cdots \underbrace{\left[\mathrm{J}_{z^{(l)}} z^{(l+1)}\right]}_{h^{(l+1)}\times h^{(l)}} \underbrace{\left[\mathrm{J}_{\boldsymbol{\theta}^{(l)}} z^{(l)}\right]}_{h^{(l)}\times d^{(l)}} , \tag{2.32c}$$

and in gradient notation

$$\underbrace{\nabla_{\boldsymbol{\theta}^{(l)}} \ell(\boldsymbol{\theta})}_{d^{(l)}} = \underbrace{\left[\mathrm{J}_{\boldsymbol{\theta}^{(l)}} z^{(l)}\right]^\top}_{d^{(l)}\times h^{(l)}} \underbrace{\left[\mathrm{J}_{z^{(l)}} z^{(l+1)}\right]^\top}_{h^{(l)}\times h^{(l+1)}} \cdots \underbrace{\left[\mathrm{J}_{z^{(L-1)}} z^{(L)}\right]^\top}_{h^{(L-1)}\times h^{(L)}} \underbrace{\nabla_{z^{(L)}} \ell}_{h^{(L)}} . \tag{2.32d}$$

14: Assuming no cost to compute a Jacobian, the optimal Jacobian contraction of Equations (2.32c) and (2.32d) are matrix chain problems that can be solved with dynamic programming: given n matrices $A_1, A_2, \dots, A_n$, the task is to find the optimal contraction schedule of $A_1 A_2 \cdots A_n$. This is crucial for performance, as this example from Cormen et al. [36, Chapter 15.2] illustrates:

Example 2.7 (Matrix chain contraction) Let $A_1 \in \mathbb{R}^{10\times100}, A_2 \in \mathbb{R}^{100\times5}, A_3 \in \mathbb{R}^{5\times50}$. There are two schedules to evaluate the chain $A_1A_2A_3$ (cost for addition neglected for simplicity):

- $(A_1A_2)A_3$: $\boldsymbol{B}_1 = A_1A_2 \in \mathbb{R}^{10\times5}$ costs 100 multiplications per element (5,000 in total). $\boldsymbol{B}_1 A_3 \in \mathbb{R}^{10\times50}$ costs 5 multiplications per element (2,500 in total).
- $A_1(A_2A_3)$: $\boldsymbol{B}_2 = A_2A_3 \in \mathbb{R}^{100\times50}$ costs 5 multiplications per element (25,000 in total). $A_1\boldsymbol{B}_2 \in \mathbb{R}^{10\times50}$ costs 100 multiplications per element (50,000 in total).

The order $(A_1A_2)A_3$ uses 10x fewer operations (7,500 versus 75,000).

In comparison to the general formulation Equation (2.28), the rather simple graphs of a neural network's loss yield much simpler expressions (Equations (2.32c) and (2.32d)) that are a result of the Jacobian chain rule (Theorem 2.1) applied to the loss Equation (2.31a). They also illustrate the impact of contraction order on performance due to the connection to matrix chains[14], mentioned in Section 2.3.1.

In forward mode, the matrix chain Equation (2.32d) would be evaluated from left to right, starting with a $d^{(l)} \times h^{(l)}$ Jacobian that is transformed into $d^{(l)} \times h^{(l')}$ matrices where $l' > l$. Since the parameter count $d^{(l)}$ in a layer is large in DNNs, these intermediate matrices are costly to store.

In reverse mode, Equation (2.32d) is evaluated from right to left, starting with a $h^{(L)}$-dimensional vector that is transformed into vectors of dimension $h^{(l')}$ with $L > l' \geq l$. These intermediate accumulations require less memory than forward mode.

Each approach has drawbacks, however. Reverse mode uses a more efficient matrix multiplication order, but the entire graph must have been evaluated and stored, or re-computed, to construct the Jacobians. While this is not needed for forward mode, forward accumulation starts with the Jacobian of an edge $(\boldsymbol{\theta}^{(l)}, z^{(l)})$ that is not shared with paths for other parameters. Therefore, intermediate accumulations can not be reused for other gradients. This is another crucial property of reverse mode, which does allow reuse of intermediate results: consider the accumulations to obtain the gradient $\nabla_{\boldsymbol{\theta}}\ell$ of all layers,

$$\begin{array}{rcllll}
\nabla_{\boldsymbol{\theta}^{(1)}}\ell &=& \left[\mathrm{J}_{\boldsymbol{\theta}^{(1)}}\boldsymbol{z}^{(1)}\right]^\top & \left[\mathrm{J}_{\boldsymbol{z}^{(1)}}\boldsymbol{z}^{(2)}\right]^\top & \left[\mathrm{J}_{\boldsymbol{z}^{(2)}}\boldsymbol{z}^{(3)}\right]^\top \cdots \left[\mathrm{J}_{\boldsymbol{z}^{(L-1)}}\boldsymbol{z}^{(L)}\right]^\top & \nabla_{\boldsymbol{z}^{(L)}}\ell \\
\nabla_{\boldsymbol{\theta}^{(2)}}\ell &=& \left[\mathrm{J}_{\boldsymbol{\theta}^{(2)}}\boldsymbol{z}^{(2)}\right]^\top & & \left[\mathrm{J}_{\boldsymbol{z}^{(2)}}\boldsymbol{z}^{(3)}\right]^\top \cdots \left[\mathrm{J}_{\boldsymbol{z}^{(L-1)}}\boldsymbol{z}^{(L)}\right]^\top & \nabla_{\boldsymbol{z}^{(L)}}\ell \\
\vdots &=& \vdots & & \ddots & \vdots \\
\nabla_{\boldsymbol{\theta}^{(L-1)}}\ell &=& \left[\mathrm{J}_{\boldsymbol{\theta}^{(L-1)}}\boldsymbol{z}^{(L-1)}\right]^\top & & \left[\mathrm{J}_{\boldsymbol{z}^{(L-1)}}\boldsymbol{z}^{(L)}\right]^\top & \nabla_{\boldsymbol{z}^{(L)}}\ell \\
\nabla_{\boldsymbol{\theta}^{(L)}}\ell &=& \left[\mathrm{J}_{\boldsymbol{\theta}^{(L)}}\boldsymbol{z}^{(L)}\right]^\top & & & \nabla_{\boldsymbol{z}^{(L)}}\ell
\end{array}$$

The paths for any two parameters $\boldsymbol{\theta}^{(l_1)}, \boldsymbol{\theta}^{(l_2)}$ with $l_2 > l_1$ share edges

$$\left(\boldsymbol{z}^{(l_2)}, \boldsymbol{z}^{(l_2+1)}\right), \dots, \left(\boldsymbol{z}^{(L-1)}, \boldsymbol{z}^{(L)}\right), \left(\boldsymbol{z}^{(L)}, \ell\right).$$

Therefore, their matrix chains share the accumulated gradient

$$\nabla_{\boldsymbol{z}^{(l_2)}}\ell = \left[\mathrm{J}_{\boldsymbol{z}^{(l_2)}}\boldsymbol{z}^{(l_2+1)}\right]^\top \left[\mathrm{J}_{\boldsymbol{z}^{(l_2+1)}}\boldsymbol{z}^{(l_2+2)}\right]^\top \cdots \left[\mathrm{J}_{v\boldsymbol{z}^{(L-1)}}\boldsymbol{z}^{(L)}\right]^\top \nabla_{\boldsymbol{z}^{(L)}}\ell\,.$$

This gradient *w.r.t.* hidden features is passed backwards through the graph and used by a layer, before updating it and passing it to the next. The interpretation of the described accumulation scheme is therefore known as gradient backpropagation algorithm [137]:

Definition 2.6 (Gradient backpropagation for sequential feedforward neural networks) Given the computation graph of a loss $\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})$ implied by a datum $(\boldsymbol{x}, \boldsymbol{y})$, a loss function ℓ, and a sequential feedforward neural network $f_{\boldsymbol{\theta}}$ from Equation (2.31a), gradient backpropagation recovers the gradient vector $\nabla_{\boldsymbol{\theta}}\ell = \left((\nabla_{\boldsymbol{\theta}^{(1)}}\ell)^\top, (\nabla_{\boldsymbol{\theta}^{(2)}}\ell)^\top, \dots, (\nabla_{\boldsymbol{\theta}^{(L)}}\ell)^\top\right)^\top$ in stages by passing gradients backward through the graph (Figure 2.7a):

- Initialize the backpropagated vector with $\nabla_{\boldsymbol{z}^{(L)}}\ell$ at L.
- For layer $l = L, \dots, 1$
  1. Receive the output gradient $\nabla_{\boldsymbol{z}^{(l)}}\ell$.
  2. Recover the parameter gradient $\nabla_{\boldsymbol{\theta}^{(l)}}\ell = \left[\mathrm{J}_{\boldsymbol{\theta}^{(l)}}\boldsymbol{z}^{(l)}\right]^\top \nabla_{\boldsymbol{z}^{(l)}}\ell\,.$
  3. Compute the input gradient $\nabla_{\boldsymbol{z}^{(l-1)}}\ell = \left[\mathrm{J}_{\boldsymbol{\theta}^{(l)}}\boldsymbol{z}^{(l)}\right]^\top \nabla_{\boldsymbol{z}^{(l)}}\ell\,.$
  4. Free $\nabla_{\boldsymbol{z}^{(l)}}\ell$ and send $\nabla_{\boldsymbol{z}^{(l-1)}}\ell$ to layer $l-1$.

Definition 2.6 is *modular*, as mentioned earlier in Section 2.2.2: it only relies on local derivatives. Supporting a new operation only requires specifying its forward pass and the vector-Jacobian products (VJPs[15]) with its input-output and parameter-output Jacobians. Given functionality to create and traverse computation graphs, implementations of backpropagation are very extensible due to its abstraction to the modular level.

15: During backpropagation, the transposed Jacobian is right-multiplied onto the backpropagated gradient vector (Equation (2.32d) from right to left). One can see this as left-multiplying the Jacobian to a column vector (Equation (2.32c) from left to right), *i.e.* computing a vector-Jacobian product. Forward accumulation instead left-multiplies the transposed Jacobian onto a matrix (Equation (2.32d) from left to right). One can view this as right-multiplying the Jacobian onto the transposed matrix (Equation (2.32c) from right to left). This requires multiple Jacobian-vector products (JVPs), or a Jacobian-matrix product (JMP).

Backpropagation itself performs a VJP with the network's parameter-output Jacobian $\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z}^{(L)}$. Choosing the vector to be $\nabla_{\boldsymbol{z}^{(L)}}\ell$ yields the gradient $\nabla_{\boldsymbol{\theta}}\ell = [\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z}^{(L)}]^\top \nabla_{\boldsymbol{z}^{(L)}}\ell$. But one can also compute VJPs $[\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z}^{(L)}]^\top \boldsymbol{v}$ with arbitrary vectors $\boldsymbol{v}$. The model's parameter-output Jacobian is crucial for computing higher-order information (Chapter 3 and Part II).

Although backpropagation only requires VJPs, the Jacobian matrix is

Table 2.2: Jacobians (Definition 2.5) for common modules in feedforward networks. Input and output are denoted $\boldsymbol{x}, \boldsymbol{z}$ rather than $\boldsymbol{z}^{(l)}, \boldsymbol{z}^{(l+1)}$ to avoid clutter. $\boldsymbol{I}$ is the identity matrix. Matrices use bold upper-case symbols ($\boldsymbol{W}, \boldsymbol{X}, \boldsymbol{Z}, \dots$), tensors use bold upper-case sans serif symbols ($\mathbf{\mathsf{W}}, \mathbf{\mathsf{X}}, \mathbf{\mathsf{Z}}, \dots$). Most Jacobians can be elegantly derived with matrix differential calculus, see Appendix A.1 for details.

OPERATION	FORWARD	JACOBIAN (Definition 2.5)	DETAILS
Matrix-vector multiplication	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}$	$\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z} = \boldsymbol{W}$, $\mathrm{J}_{\boldsymbol{W}}\boldsymbol{z} = \boldsymbol{x}^\top \otimes \boldsymbol{I}$	Appendix A.2.1
Matrix-matrix multiplication	$\boldsymbol{Z}(\boldsymbol{X}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{X}$	$\mathrm{J}_{\boldsymbol{X}}\boldsymbol{Z} = \boldsymbol{I} \otimes \boldsymbol{W}$, $\mathrm{J}_{\boldsymbol{W}}\boldsymbol{Z} = \boldsymbol{X}^\top \otimes \boldsymbol{I}$	Appendix A.2.1
Addition	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{b}) = \boldsymbol{x} + \boldsymbol{b}$	$\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z} = \mathrm{J}_{\boldsymbol{b}}\boldsymbol{z} = \boldsymbol{I}$	Appendix A.2.1
Elementwise activation	$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})$, s.t. $z_i(\boldsymbol{x}) = \phi(x_i)$	$\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z} = \mathrm{diag}[\boldsymbol{\phi}'(\boldsymbol{x})]$	Appendix A.2.2
Skip-connection	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{x} + \boldsymbol{s}(\boldsymbol{x}, \boldsymbol{\theta})$	$\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z} = \boldsymbol{I} + \mathrm{J}_{\boldsymbol{x}}\boldsymbol{s}$, $\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z} = \mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{s}$	Appendix A.2.3
Reshape/view	$\mathbf{\mathsf{Z}}(\mathbf{\mathsf{X}}) = \mathrm{reshape}(\mathbf{\mathsf{X}})$	$\mathrm{J}_{\mathbf{\mathsf{X}}}\mathbf{\mathsf{Z}} = \boldsymbol{I}$	Appendix A.4.1
Index select/map π	$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{\Pi}\boldsymbol{x}$, $\Pi_{j,\pi(j)} = 1$	$\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z} = \boldsymbol{\Pi}$	Appendix A.4.2
Convolution	$\mathbf{\mathsf{Z}}(\mathbf{\mathsf{X}}, \mathbf{\mathsf{W}}) = \mathbf{\mathsf{X}} \star \mathbf{\mathsf{W}}$, $\boldsymbol{Z}(\boldsymbol{W}, [\![\mathbf{\mathsf{X}}]\!]) = \boldsymbol{W}[\![\mathbf{\mathsf{X}}]\!]$	$\mathrm{J}_{[\![\mathbf{\mathsf{X}}]\!]}\mathbf{\mathsf{Z}} = \boldsymbol{I} \otimes \boldsymbol{W}$, $\mathrm{J}_{\boldsymbol{W}}\mathbf{\mathsf{Z}} = [\![\mathbf{\mathsf{X}}]\!]^\top \otimes \boldsymbol{I}$	Appendix A.4.3
Square loss	$\ell(\boldsymbol{f}, \boldsymbol{y}) = {}^1\!/_C(\boldsymbol{y} - \boldsymbol{f})^\top(\boldsymbol{y} - \boldsymbol{f})$	$\mathrm{J}_{\boldsymbol{f}}\ell = 2(\boldsymbol{f} - \boldsymbol{y})^\top$	Appendix A.3.1
Softmax cross-entropy	$\ell(\boldsymbol{f}, y) = -\mathrm{onehot}(y)^\top \log[\boldsymbol{p}(\boldsymbol{f})]$	$\mathrm{J}_{\boldsymbol{f}}\ell = (\boldsymbol{y} - \boldsymbol{p}(\boldsymbol{f}))^\top$	Appendix A.3.2

an interesting object for analytical studies into its structure, and for efficient implementation of functionality that goes beyond computing gradients (*e.g.* matrix-Jacobian products (MJPs)). Table 2.2 contains the Jacobians of the common operations in neural networks from Section 2.2.2. These Jacobians are conveniently obtained using matrix differential calculus [103], presented in Appendix A.1.

Backpropagation & Batching

To see the interplay between AD and batching (Section 2.1.2), consider differentiation of the mini-batch loss (Equation (2.5)). Popular ML libraries like PyTorch construct the computation graph on the level of tensor-valued variables with an additional batch axis.

General case: Starting from the previous differentiation of a single datum loss, the mini-batch scenario follows by the substitutions

$$
\begin{aligned}
\boldsymbol{x} =: \boldsymbol{z}^{(0)} \quad &\leftrightarrow \quad \boldsymbol{X} =: \boldsymbol{Z}^{(0)} \in \mathbb{R}^{|\mathbb{B}| \times h^{(0)}} \,, \\
\boldsymbol{y} \quad &\leftrightarrow \quad \boldsymbol{Y} \in \mathbb{R}^{|\mathbb{B}| \times C} \,, \\
\boldsymbol{z}^{(\ell)} \quad &\leftrightarrow \quad \boldsymbol{Z}^{(l)} \in \mathbb{R}^{|\mathbb{B}| \times h^{(l)}} \,, \\
\ell \quad &\leftrightarrow \quad \boldsymbol{\ell} \in \mathbb{R}^{|\mathbb{B}|} \,,
\end{aligned}
\tag{2.33a}
$$

with stacked data $\boldsymbol{X} = (\boldsymbol{x}_1 \, \boldsymbol{x}_2 \, \dots \, \boldsymbol{x}_{|\mathbb{B}|})$ and $\boldsymbol{Y} = (\boldsymbol{y}_1 \, \boldsymbol{y}_2 \, \dots \, \boldsymbol{y}_{|\mathbb{B}|})$, and assuming matrix-to-matrix layer functions. To account for the reduction of per-sample losses $\boldsymbol{\ell}$, the graph is extended by

$$
\mathcal{L}(\boldsymbol{\ell}) = \mathrm{mean}(\boldsymbol{\ell}) = \frac{1}{|\mathbb{B}|} \sum_{n=1}^{|\mathbb{B}|} [\boldsymbol{\ell}]_n \,. \tag{2.33b}
$$

The computation graph, shown in Figure 2.7b, is $\mathcal{G} = (\mathcal{E}, \mathcal{V})$ with nodes

$$
\mathcal{V} = \left\{ \boldsymbol{\theta}^{(l)} \right\}_{l=1}^{L} \cup \left\{ \boldsymbol{Z}^{(l)} \right\}_{l=0}^{L} \cup \{ \boldsymbol{Y}, \boldsymbol{\ell}, \mathcal{L} \} \tag{2.33c}
$$

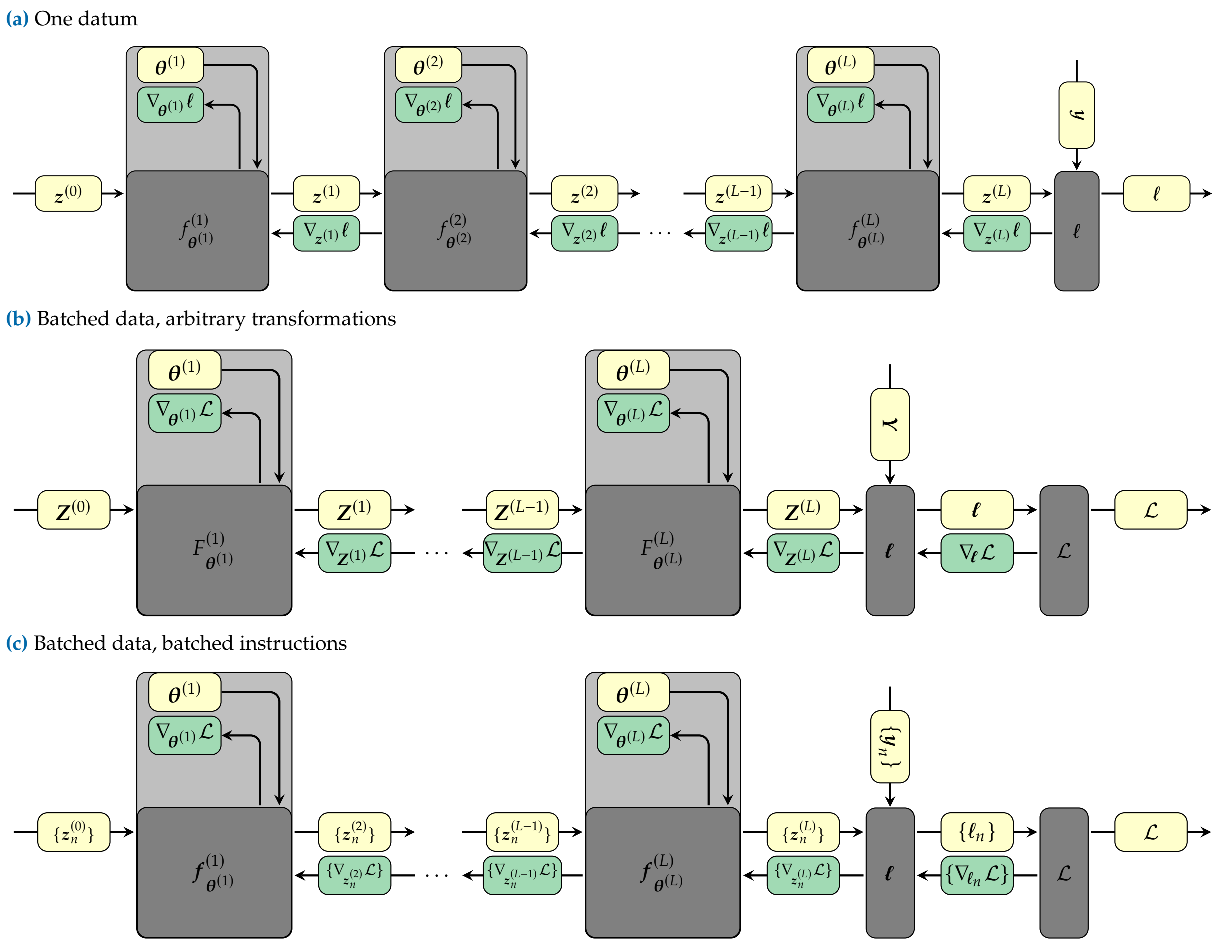

(a) One datum
(b) Batched data, arbitrary transformations
(c) Batched data, batched instructions

Figure 2.7: Gradient backpropagation and (un)awareness of per-sample structure in many ML libraries. (a) Computation graph of of a neural network's loss on a single datum $(\boldsymbol{x}, \boldsymbol{y})$. Gradients are backpropagated through the graph as described by Definition 2.6 to obtain $\nabla_{\boldsymbol{\theta}} \ell$. **(b)** To exploit parallelism in the computations, multiple data are stacked into matrices $(\boldsymbol{X}, \boldsymbol{Y})$ which are then processed by a sequence of matrix-to-matrix functions $F^{(l)}_{\boldsymbol{\theta}^{(l)}}$ into a batch of losses $\boldsymbol{\ell}$, and reduced into a scalar $\mathcal{L}$ via mean reduction. AD in popular ML libraries like PyTorch tracks variables on the level of batched tensors. Therefore, operations are allowed to build up dependencies between data—such that $\mathcal{L} = {}^{1}/_{|\mathbb{B}|} \sum_n [\boldsymbol{\ell}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{\theta})]_n$ where each component of $\boldsymbol{\ell}$ may depend on *all* data (batch normalization [79] is such a case)—without breaking gradient backpropagation. ML libraries implement VJPS for the matrix-to-matrix functions $F^{(l)}_{\boldsymbol{\theta}^{(l)}}$. This loses structure for operations that treat inputs independently along the batch axis. **(c)** The empirical risk on a mini-batch (Equation (2.5) is such a case: all operations in the graph process inputs independently and with the same instructions along the batch axis. The following connections to the single datum case **(a)** hold: $F^{(l)}_{\boldsymbol{\theta}^{(l)}} \leftrightarrow f^{(l)}_{\boldsymbol{\theta}^{(l)}} = \mathrm{vmap}(f^{(l)}_{\boldsymbol{\theta}^{(l)}}), \boldsymbol{\ell} = \mathrm{vmap}(\ell)$ with vmap from Definition 2.1. Due to the more general support of AD in ML libraries for graphs of the form **(b)**, their VJPs cannot be accessed per-sample.

and edges

$$
\begin{aligned}
\mathcal{E} = &\left\{ (\boldsymbol{Z}^{(l-1)}, \boldsymbol{Z}^{(l)}), \right\}_{l=1}^{L} \cup \left\{ (\boldsymbol{\theta}^{(l)}, \boldsymbol{Z}^{(l)}) \right\}_{l=1}^{L} \\
&\cup \left\{ (\boldsymbol{Z}^{(L)}, \boldsymbol{\ell}), (\boldsymbol{Y}, \boldsymbol{\ell}), (\boldsymbol{\ell}, \mathcal{L}) \right\} .
\end{aligned}
\tag{2.33d}
$$

Gradient backpropagation (Definition 2.6) carries over the mini-batch loss graph Equation (2.33) and efficiently recovers the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}$.

Popular libraries like PyTorch implement the required functionality, VJPs, for the matrix-to-matrix functions that process inputs with a batch axis (see Definition 2.5 for the Jacobian's generaliation to matrix functions). Hence, operations $\boldsymbol{Z}^{(l)} \mapsto \boldsymbol{Z}^{(l+1)}$ are allowed to create dependencies across the batch axis without breaking the gradient computation[16].

16: *E.g.* batch normalization [79] introduces dependencies along the batch dimension by centering and re-scaling the input with statistics computed across the batch axis.

Per-sample structure: This work focuses on empirical risks that are averages over *per-sample* losses (recall Equation (2.4b)),

$$\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{B}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)\,.$$

Hence, all operations act independently, and using the same instructions, along the batch dimension. All variables in the graph inherit this independence along their batch axis,

$$\boldsymbol{Z}^{(l)} = \begin{pmatrix} \boldsymbol{z}_1^{(l)} & \boldsymbol{z}_2^{(l)} & \dots & \boldsymbol{z}_{|\mathbb{B}|}^{(l)} \end{pmatrix}\,, \qquad l = 0, \dots, L\,, \tag{2.34a}$$

$$\boldsymbol{\ell} = \begin{pmatrix} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_1), \boldsymbol{y}_1) \\ \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_2), \boldsymbol{y}_2) \\ \vdots \\ \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_{|\mathbb{B}|}), \boldsymbol{y}_{|\mathbb{B}|}) \end{pmatrix}\,. \tag{2.34b}$$

For all layers $l = 1, \dots, L$, this independence across samples implies block-diagonal input-output Jacobians,

$$\mathrm{J}_{\boldsymbol{Z}^{(l-1)}} \boldsymbol{Z}^{(l)} = \begin{pmatrix} \mathrm{J}_{\boldsymbol{z}_1^{(l-1)}} \boldsymbol{z}_1^{(l)} & 0 & \dots & 0 \\ 0 & \mathrm{J}_{\boldsymbol{z}_2^{(l-1)}} \boldsymbol{z}_2^{(l)} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \mathrm{J}_{\boldsymbol{z}_{|\mathbb{B}|}^{(l-1)}} \boldsymbol{z}_{|\mathbb{B}|}^{(l)}, \end{pmatrix} \tag{2.34c}$$

and per-sample block structure in the output-parameter Jacobian

$$\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{Z}^{(l)} = \begin{pmatrix} \mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_1^{(l)} & \mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_2^{(l)} & \dots & \mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_{|\mathbb{B}|}^{(l)}, \end{pmatrix}\,. \tag{2.34d}$$

Many implementations of backpropagation make it difficult to access this per-sample structure. They only expose VJPs $\boldsymbol{v} \mapsto [\mathrm{J}_{\boldsymbol{Z}^{(l)}} \boldsymbol{Z}^{(l+1)}]^\top \boldsymbol{v}$ and $\boldsymbol{v} \mapsto [\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{Z}^{(l+1)}]^\top \boldsymbol{v}$ for Equations (2.34c) and (2.34d). While this allows supporting AD of more general graphs than those of an empirical risk (Equation (2.4b), it limits access to only the *average* gradient

$$\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{B}} \nabla_{\boldsymbol{\theta}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n) \tag{2.35}$$

when differentiating an empirical risk. Computing per-sample gradients is not more demanding than computing the average gradient—the only difference is taking the average—but their computation is not supported. More flexible access to the Jacobians Equations (2.34c) and (2.34d) through per-sample VJPs (or MJPs), enables the computation of various higher-order quantities. Their efficient realization will be the main focus of Chapters 4, 5 and 7.

To highlight independence across the batch axis in a computation graph, a set notation will be preferred over the matrix notation (Equations (2.33a) and (2.34)) in the following. *E.g.* the text uses $\{\boldsymbol{z}_n^{(l)}\}_n$, or just $\{\boldsymbol{z}_n^{(l)}\}$, instead of $\boldsymbol{Z}^{(l)}$. Figure 2.7c illustrates this set notation[17].

17: This notation is closer to recent developments in AD for ML libraries [23, 72] that separate batching and AD more clearly through vectorization via a vmap interface (Definition 2.1). A different way to arrive at the batched computation graph in Figure 2.7c is to start from the computation graph of a single datum's loss $\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})$ and vectorize it to obtain the set of graphs $\{\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)\}$, which can be stacked into a single graph that produces $\boldsymbol{\ell}$. Appending the reduction node to this graph, one obtains the computation graph for the average loss.

Higher-order Information 3.

The previous chapter highlighted the important structure of deep neural networks and the empirical risk used to train them:

- **Sum structure & noise:** Given a model f_{θ}, a convex loss function ℓ, and a dataset $\mathbb{D}$, the empirical risk is an average over per-datum losses. Each of these losses is identically computed by feeding a datum through the model and the loss function. Batching these computations allows to efficiently evaluate them in parallel. Doing so on only a random subset of data—a mini-batch—reduces computational cost in exchange for noise.
- **Probabilistic interpretation:** For specific loss functions, which include regression and softmax classification, empirical risk minimization can be interpreted as maximum likelihood—or maximum a posteriori—estimation where the model parameterizes a likelihood distribution $q(\boldsymbol{y} \mid f_{\theta}(\boldsymbol{x}))$.
- **Layer-structure & modularity:** Neural networks consist of layers, or modules, which are glued together by function composition. Backpropagation abstract the differentiation by the composition of the module-level derivatives, which simplifies supporting new operations and hides the complexity of AD from practitioners.

Popular ML libraries allow for efficient and automated gradient computation, but combine AD and batching in a way that allows to support more general computation graphs than the empirical risk. However, the added optimization complicates efficiently assessing the empirical risk's per-sample structure, and higher-order information which requires slightly more flexible AD operations, like matrix-Jacobian products.

This chapter motivates why relying solely on the gradient has limitations, and—based on structure in the empirical risk—proposes quantities in the form of higher-order information that go beyond the gradient. It starts by presenting currently popular deep learning optimization methods (Section 3.1) which rely heavily on the average mini-batch gradient provided through AD in ML libraries. While training algorithms that use more information than the gradient have been historically mostly unexplored—because the quantities required are not as optimized and automated as gradient computations—they have exciting promises.

Second-order methods (Section 3.2) use more information than just the gradient. They are known to improve over first-order methods in "classic" optimization problems (convex optimization, general linear models). They incorporate curvature information in form of the Hessian (Section 3.2.1), or PSD approximations thereof, such as the generalized Gauss-Newton (Section 3.2.3) and Fisher information matrix (Section 3.2.4).

While optimization methods for neural networks are one main application in deep learning, there are other applications to quantities beyond the gradient. Section 3.3 highlights additional use cases which underline the relevance of such information beyond optimization. Chapter 6 presents another use case focused on improving the training of neural networks.

3.1 Popular Deep Learning Optimizers

Section 2.1 formulates learning as minimizing the empirical risk, for which an optimization algorithm seeks to find a solution. Deep learning optimizers are iterative: they start from an initial parameter $\boldsymbol{\theta}_0$ and aim to improve the parameters in iterations, leading to a trajectory $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \dots$ in parameter space. An iteration is described by an update rule $\mathcal{M} : \boldsymbol{\theta}_t \mapsto \boldsymbol{\theta}_{t+1}$ that relies on internal states of the optimizer from the observation history H_t, local observations at the current iterate $\boldsymbol{\theta}_t$, and hyperparameters ϕ_t [34]. The next iterate results from a simplified optimization problem where the actual objective is replaced by a computationally cheap approximation. Often, this direction is further adapted by outer-loop mechanisms, like line searches, or hyperparameters that require manual tuning to achieve good performance.

The update quality is influenced by the quality of the local approximation. Computing exact information about the objective $\mathcal{L}_{\mathbb{D}}$ is expensive, but yields a more accurate local proxy. Due to the large-scale nature of deep learning, algorithms only rely on stochastic information from a mini-batch $\mathbb{B}_t$ in form of the mini-batch loss $\mathcal{L}_{\mathbb{B}_t}$ (Section 2.1.2). In addition to locality, noise further complicates deriving a precise local approximation.

Stochastic Gradient Descent

One of the most popular methods, according to Figure 1.1, is stochastic gradient descent (SGD, [135]). Its update derives from

$$\mathcal{M}: \qquad \boldsymbol{\theta}_{t+1} = \arg\min_{\boldsymbol{\theta}} m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta}) = \boldsymbol{\theta}_t - \eta \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t) \tag{3.1a}$$

where $m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta})$ is a first-order Taylor around the current iterate $\boldsymbol{\theta}_t$ based on the mini-batch gradient, regularized by a quadratic L_2 penalty to discourage large steps with a learning rate η,

$$m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta}) = \mathcal{L}_{\mathbb{B}_t}(\boldsymbol{\theta}_t) + \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2\eta}\|\boldsymbol{\theta} - \boldsymbol{\theta}_t\|_2^2 \tag{3.1b}$$

This leads to Update Rule 3.1, which updates the parameters with the scaled negative mini-batch gradient. The negative gradient also corresponds to the direction of steepest descent, *i.e.* along which the loss decreases most rapidly around $\boldsymbol{\theta}_t$ (see Section 3.2.4 and Equation (3.18)).

Momentum Methods (Incorporating Past Knowledge)

The descent direction in SGD becomes poorer as the mini-batch gradient is subject to more noise. Incorporating previous gradient observations can help reduce noise and find a better descent direction [60, Chapter 8.3]. From a physical interpretation, the optimizer builds up a velocity (referred to as *momentum*) that is adapted with new gradient observation. Two variations are the heavy ball [128] and Nesterov [117] momentum methods (Update Rules 3.2 and 3.3). Another intuition for momentum is that it suppresses SGD's oscillating behavior in the presence of two directions with differing curvature by averaging out gradients that point in opposite directions (see *e.g.* central panel of Figure 6.1).

Update Rule 3.1 (SGD [135])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)$$

with

learning rate $\eta \in \mathbb{R}^+$

Update Rule 3.2 (Momentum [128])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{v}_t$$

where

$$\boldsymbol{v}_t = \rho \boldsymbol{v}_{t-1} + \eta \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)$$

with

learning rate $\eta \in \mathbb{R}^+$

momentum factor $\rho \in [0;1)$

initial momentum $\boldsymbol{v}_0 \in \mathbb{R}^D$

Update Rule 3.3 (NAG [117])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{v}_t$$

where

$$\boldsymbol{v}_t = \rho \boldsymbol{v}_{t-1} + \eta \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t - \rho \boldsymbol{v}_{t-1})$$

with

learning rate $\eta \in \mathbb{R}^+$

momentum factor $\rho \in [0;1)$

initial momentum $\boldsymbol{v}_0 \in \mathbb{R}^D$

Update Rule 3.4 (AdaGrad [44])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t) \oslash (\boldsymbol{s}_t^{\circ 1/2} + \epsilon \mathbf{1})$$

where

$$\boldsymbol{s}_t = \boldsymbol{s}_{t-1} + (\boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t))^{\circ 2}$$

with

learning rate $\eta \in \mathbb{R}^+$

divide-by-zero safe guard $\epsilon \in \mathbb{R}^+$

initial $\boldsymbol{s}_0 \in \mathbb{R}^D$

Update Rule 3.5 (RMSProp [164])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t) \oslash (\boldsymbol{s}_t^{\circ 1/2} + \epsilon \mathbf{1})$$

where

$$\boldsymbol{s}_t = \rho \boldsymbol{s}_{t-1} + (1-\rho)(\boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t))^{\circ 2}$$

with

learning rate $\eta \in \mathbb{R}^+$

decay rate $\rho \in [0;1)$

divide-by-zero safe guard $\epsilon \in \mathbb{R}^+$

initial $\boldsymbol{s}_0 \in \mathbb{R}^D$

Update Rule 3.6 (Adadelta [180])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{\Delta}_t$$

where

$$\boldsymbol{\Delta}_t = (\boldsymbol{d}_t + \epsilon \mathbf{1})^{\circ 1/2} \oslash (\boldsymbol{s}_t + \epsilon \mathbf{1})^{\circ 1/2} \odot \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)$$

where

$$\boldsymbol{s}_t = \rho \boldsymbol{s}_{t-1} + (1-\rho)(\boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t))^{\circ 2}$$

$$\boldsymbol{d}_t = \rho \boldsymbol{d}_{t-1} + (1-\rho)\boldsymbol{\Delta}_{t-1}^{\circ 2}$$

with

learning rate $\eta \in \mathbb{R}^+$

decay rate $\rho \in [0;1)$

divide-by-zero safe guard $\epsilon \in \mathbb{R}^+$

initial $\boldsymbol{s}_0, \boldsymbol{d}_0, \boldsymbol{\Delta}_0 \in \mathbb{R}^D$

Update Rule 3.7 (Adam [87])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \hat{\boldsymbol{m}}_t \oslash (\hat{\boldsymbol{v}}_t^{\circ 1/2} + \epsilon \mathbf{1})$$

where

$$\hat{\boldsymbol{m}}_t = {}^{\boldsymbol{m}_t}/(1-\beta_1^t)$$

where

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)$$

$$\hat{\boldsymbol{v}}_t = {}^{\boldsymbol{v}_t}/(1-\beta_2^t)$$

where

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)(\boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t))^{\circ 2}$$

with

learning rate $\eta \in \mathbb{R}^+$

decay rates $\beta_1, \beta_2 \in [0;1)$

divide-by-zero safe guard $\epsilon \in \mathbb{R}^+$

initial $\boldsymbol{m}_0, \boldsymbol{v}_0, \in \mathbb{R}^D$

Update Rule 3.8 (AMSGrad [132])

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \hat{\boldsymbol{m}}_t \oslash (\hat{\boldsymbol{v}}_t^{\circ 1/2} + \epsilon \mathbf{1})$$

where

$$\hat{\boldsymbol{m}}_t = {}^{\boldsymbol{m}_t}/(1-\beta_1^t)$$

where

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)$$

$$\hat{\boldsymbol{v}}_t = {}^{\boldsymbol{v}_t}/(1-\beta_2^t)$$

where

$$\boldsymbol{v}_t = \max(\beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)(\boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t))^{\circ 2}, \boldsymbol{v}_{t-1})$$

with

learning rate $\eta \in \mathbb{R}^+$

decay rates $\beta_1, \beta_2 \in [0;1)$

divide-by-zero safe guard $\epsilon \in \mathbb{R}^+$

initial $\boldsymbol{m}_0, \boldsymbol{v}_0, \in \mathbb{R}^D$

Figure 3.1: **Popular deep learning optimizers rely on the average mini-batch gradient.** At iteration t, they incorporate information in form of the mini-batch average gradient $\boldsymbol{g}_{\mathbb{B}_t}$. This is a representative subset of popular methods; see [145] for a more complete overview.

Adaptive Methods (Per-parameter Learning Rate)

Adaptive methods like AdaGrad [44], RMSProp [164], Adadelta [180], Adam [87] & AMSGrad [132] keep a learning rate for each parameter that is adapted over time. This rescaling is driven by the elementwise square of the mini-batch gradient, $g_{\mathbb{B}_t}(\boldsymbol{\theta}_t)^{\odot 2}$ (Update Rules 3.4 to 3.8). These algorithms use this quantities in different ways, motivated by shortcomings of predecessors (*e.g.* too aggressive learning rate shrinking [164, 180] and convergence problems [132]).

No Clear Winner & Mini-batch Gradient as Central Object

Figure 3.1 summarizes the update rules of the above optimization methods. Their update rules to derive $\boldsymbol{\theta}_{t+1}$ rely on three ingredients:

- The history of iterates $\{\boldsymbol{\theta}_{t'}\}_{t'=0}^{t}$
- The history of gradients[1] $\{g_{\mathbb{B}_{t'}}(\boldsymbol{\theta}_{t'})\}_{t'=0}^{t}$
- The history of elementwise gradient squares $\{g_{\mathbb{B}_{t'}}(\boldsymbol{\theta}_{t'})^{\odot 2}\}_{t'=0}^{t}$

1: One mild exception is Nesterov momentum (Update Rule 3.3). It relies on the "lookahead" gradient history rather than the gradient history of iterates.

In summary, the main ingredient in all of these methods is the average mini-batch gradient (and its elementwise square). Their updates are cheap and only require computation of average mini-batch gradients, as efficiently provided by AD in popular ML libraries (Section 2.3):

$$\begin{aligned} &\mathcal{M} = \mathcal{M}(H_t, \phi_t) \\ \text{with} \quad &H_t = \{\boldsymbol{\theta}_{t'}\}_{t'=0}^{t}, \cup \left\{g_{\mathbb{B}_{t'}}(\boldsymbol{\theta}_{t'})\right\}_{t'=0}^{t}, \cup \left\{g_{\mathbb{B}_{t'}}(\boldsymbol{\theta}_{t'})^{\odot 2}\right\}_{t'=0}^{t}, \end{aligned} \tag{3.2}$$

While the mentioned methods are a representative subset of popular algorithms, there are more than one hundred algorithms with structurally similar update rules (see Table 2 in [145] and references therein). Some adaptive methods like Adam and RMSProp contain simpler methods like SGD and Momentum as special cases [34]. Therefore, they should, in principle, be able to perform better. However, recent work that compares deep learning optimizers through benchmarks to identify the best-performing method finds that the currently existing methods perform quite similarly [145].

One reason why newly developed optimizers of the structure Equation (3.2) seem to not be able to clearly improve over existing methods could be that their update rules are constrained to using the average gradient, as it is readily available in software. To overcome these limitations, it might be helpful to study the potential of methods that leverage information beyond the gradient. Second-order optimization incorporate curvature information and represent the state-of-the art in "classical" optimization problems (convex optimization, generalized linear models). Noise-reduction methods focus on improving the gradient estimator $g_{\mathbb{B}_t}$ by reducing its variance through per-sample information. The rest of this chapter provides definitions for the information used by such methods, and provides additional motivation for them by showcasing applications outside optimization.

3.2 Second-order Optimization

Second-order methods iteratively optimize an objective $\mathcal{L}(\boldsymbol{\theta}) : \mathbb{R}^D \to \mathbb{R}$ using a local quadratic approximation $\mathcal{L}(\boldsymbol{\theta}) \approx m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta})$ around the current point $\boldsymbol{\theta}_t$ after t iterations,

$$m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta}) = a(\boldsymbol{\theta}_t) + \boldsymbol{b}(\boldsymbol{\theta}_t)^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^\top \boldsymbol{C}(\boldsymbol{\theta}_t)(\boldsymbol{\theta} - \boldsymbol{\theta}_t)\,, \tag{3.3a}$$

with an offset $a(\boldsymbol{\theta}_t) \in \mathbb{R}$, a slope vector $\boldsymbol{b}(\boldsymbol{\theta}_t) \in \mathbb{R}^D$, and a curvature matrix $\boldsymbol{C}(\boldsymbol{\theta}_t) \in \mathbb{R}^{D\times D}$. As described in Section 3.1, the next iterate $\boldsymbol{\theta}_{t+1}$ is obtained by minimizing the local approximation. For this proxy to possess a minimum, the curvature matrix $\boldsymbol{C}(\boldsymbol{\theta}_t)$ must be PD. Then, Equation (3.3a) is minimized by

$$\boldsymbol{\theta}_{t+1} = \arg\min_{\boldsymbol{\theta}} m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta}) = \boldsymbol{\theta}_t - \boldsymbol{C}(\boldsymbol{\theta}_t)^{-1}\boldsymbol{b}(\boldsymbol{\theta}_t)\,. \tag{3.3b}$$

This update is computationally challenging, because the size of the curvature matrix is quadratic in D and generally infeasible to store. Additionally, the computational complexity of matrix inversion scales cubically in D. These problems can somewhat be addressed by approximately solving the linear system

$$\boldsymbol{C}(\boldsymbol{\theta}_t)\boldsymbol{\theta}_{t+1} = -\boldsymbol{b}(\boldsymbol{\theta}_t) \tag{3.3c}$$

for $\boldsymbol{\theta}_{t+1}$. This can be done with iterative solvers, such as CG, which only require matrix-vector products with $\boldsymbol{C}(\boldsymbol{\theta}_t)$, which can often be implemented without expanding the matrix representation in memory.

The first-order methods from Section 3.1 circumvent these issues by using a diagonal—and often quite crude—curvature approximation that is cheap to store and invert (see [34] for an overview). *E.g.*, SGD's local approximation, Equation (3.1b), uses $1/2\eta\boldsymbol{I}$ as curvature matrix. The following sections introduce common curvature matrices for local approximations of empirical risks.

3.2.1 Newton's Method & the Hessian Matrix

The Taylor series provides a meaningful local approximation of an analytic function. Its expansion up to second-order is

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) = &\underbrace{\mathcal{L}(\boldsymbol{\theta}_t)}_{a(\boldsymbol{\theta}_t)} + \underbrace{\nabla_{\boldsymbol{\theta}_t}\mathcal{L}(\boldsymbol{\theta}_t)}_{\boldsymbol{b}_{\boldsymbol{\theta}_t}}{}^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_t) \\ &+ \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^\top \underbrace{\nabla^2_{\boldsymbol{\theta}_t}\mathcal{L}(\boldsymbol{\theta}_t)}_{\boldsymbol{C}(\boldsymbol{\theta}_t)}(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \mathcal{O}\left((\boldsymbol{\theta} - \boldsymbol{\theta}_t)^3\right)\end{aligned} \tag{3.4}$$

where $\mathcal{O}((\boldsymbol{\theta} - \boldsymbol{\theta}_t)^3)$ denotes polynomial terms in the components of $\boldsymbol{\theta} - \boldsymbol{\theta}_t$ of cubic order and above. This reveals the objective's Hessian $\nabla^2_{\boldsymbol{\theta}_t}\mathcal{L}(\boldsymbol{\theta}_t)$, which collects its second-order partial derivatives in a matrix $[\nabla^2_{\boldsymbol{\theta}_t}\mathcal{L}(\boldsymbol{\theta}_t)]_{i,j} = \partial^2\mathcal{L}(\boldsymbol{\theta}_t)/\partial[\boldsymbol{\theta}_t]_i\partial[\boldsymbol{\theta}_t]_j$, as curvature matrix.

Definition 3.1 (Hessian) Let $b : \mathbb{R}^D \to \mathbb{R}; \boldsymbol{a} \mapsto b(\boldsymbol{a})$ be a differentiable vector-to-scalar function. The Hessian $\nabla^2_{\boldsymbol{a}} b \in \mathbb{R}^{D \times D}$ of b *w.r.t.* $\boldsymbol{a}$ is a symmetric matrix containing the second-order partial derivatives

$$\nabla^2_{\boldsymbol{a}} b = \frac{\partial^2 b}{\partial \boldsymbol{a} \partial \boldsymbol{a}^\top} \qquad \text{with} \qquad [\nabla^2_{\boldsymbol{a}} b]_{i,j} = \frac{\partial^2 b}{\partial a_i \partial a_j} \tag{3.5}$$

Definition 3.2 generalizes this to the matrix/tensor case. The Hessian will often be denoted by $\boldsymbol{H}$. *E.g.* $\boldsymbol{H}_{p_{\text{data}}}(\boldsymbol{\theta}) := \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta})$ for the Hessian of the population risk Equation (2.3a), and $\boldsymbol{H}_{\mathbb{D}}(\boldsymbol{\theta}) := \nabla^2_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$ for the Hessian of the empirical risk Equation (2.4b) on a dataset $\mathbb{D}$ (with $\mathbb{D} = \mathbb{D}_{\text{train}}, \mathbb{B}$ for the train loss and mini-batch Hessian).

Newton's method uses the Hessian as curvature matrix.

Update Rule 3.9 (Newton's method (simplified)) A Newton step is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{H}_{\mathbb{D}}(\boldsymbol{\theta}_t)^{-1} \boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}_t) \tag{3.8}$$

with the gradient and Hessian from Definitions 2.4 and 3.1 (practical implementations vary and often introduce additional hyperparameters such as a learning rate, damping term, mini-batch size, *etc.*).

Matrix-free multiplication with the Hessian [127] can been combined with CG to compute Newton steps via solving the linear system Equation (3.3c). This idea is known as Hessian-free optimization [106].

While the Taylor expansion motivates using the Hessian as curvature matrix, it leads to problems for non-convex functions like the empirical risk, as it is in general indefinite. Therefore Equation (3.3b) does not have a solution. In practice, PSD curvature matrices that are approximations of the Hessian are popular substitutes for the Hessian to avoid this issue (Sections 3.2.2 to 3.2.4).

Definition 3.2 (Generalized Hessian) Let $\boldsymbol{B} : \mathbb{R}^{n \times q} \to \mathbb{R}^{m \times p}$ be a twice differentiable matrix function. The *Hessian* $\nabla^2_{\boldsymbol{A}} \boldsymbol{B}(\boldsymbol{A})$ is an $(mnpq \times nq)$ matrix defined by

$$\begin{aligned} &\nabla^2_{\boldsymbol{A}} \boldsymbol{B}(\boldsymbol{A}) \\ &= \mathrm{J}_{\boldsymbol{A}} \left[\mathrm{J}_{\boldsymbol{A}} \boldsymbol{B}(\boldsymbol{A})\right]^\top \\ &= \frac{\partial}{\partial (\operatorname{vec} \boldsymbol{A})^\top} \operatorname{vec} \left\{ \left[\frac{\partial \operatorname{vec} \boldsymbol{B}(\boldsymbol{A})}{\partial (\operatorname{vec} \boldsymbol{A})^\top} \right]^\top \right\} \end{aligned} \tag{3.6}$$

[103, Chapter 10.2] with flattening defined by Definition 2.2. In element-wise notation, this is the matrix-stack of all output component Hessians

$$\nabla^2_{\boldsymbol{A}} \boldsymbol{B}(\boldsymbol{A}) = \begin{pmatrix} \nabla^2_{\operatorname{vec} \boldsymbol{A}} [\operatorname{vec} \boldsymbol{B}]_1 \\ \nabla^2_{\operatorname{vec} \boldsymbol{A}} [\operatorname{vec} \boldsymbol{B}]_2 \\ \vdots \\ \nabla^2_{\operatorname{vec} \boldsymbol{A}} [\operatorname{vec} \boldsymbol{B}]_{np} \end{pmatrix},$$

with the Hessian from Definition 3.1.

The tensor case is analogous but requires cluttered notation and is therefore omitted. Common forms for neural networks include vector-to-vector functions $f : \mathbb{R}^m \to \mathbb{R}^n, \boldsymbol{x} \mapsto f(\boldsymbol{x})$ with

$$\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x}) = \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}^\top \partial \boldsymbol{x}},$$

where $\boldsymbol{x}$ can be considered the input or bias vector if a linear layer. In details,

$$\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x}) = \begin{pmatrix} \nabla^2_{\boldsymbol{x}} f_1(\boldsymbol{x}) \\ \vdots \\ \nabla^2_{\boldsymbol{x}} f_m(\boldsymbol{x}) \end{pmatrix}. \tag{3.7}$$

Others are matrix-to-vector mappings $f : \mathbb{R}^{n \times q} \to \mathbb{R}^m, \boldsymbol{X} \to f(\boldsymbol{X})$ with

$$\nabla^2_{\boldsymbol{X}} f(\boldsymbol{X}) = \frac{\partial}{\partial (\operatorname{vec} \boldsymbol{X})^\top} \frac{\partial f(\boldsymbol{X})}{\partial \operatorname{vec} \boldsymbol{X}},$$

e.g. with $\boldsymbol{X}$ the weight of a linear layer.

3.2.2 The Gauss-Newton Method & Matrix

The Gauss-Newton (GN) method [22, chapter 6.3] tackles the nonlinear least squares regression task (Example 2.1) of fitting a function $f_{\boldsymbol{\theta}}$ to data in the form of vector-valued inputs $\boldsymbol{x}_n \in \mathbb{R}^M$ and scalar-valued outputs $y_n \in \mathbb{R}$ by minimizing the mean squared error,

$$\underset{\boldsymbol{\theta}}{\text{minimize}}\, \mathcal{L}(\boldsymbol{\theta}), \quad \text{where} \quad \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, y_n) \in \mathbb{D}} (\underbrace{f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)}_{:= f_n} - y_n)^2, \tag{3.9}$$

The objective's gradient is

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{2}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, y_n) \in \mathbb{D}} (\mathrm{J}_{\boldsymbol{\theta}} f_n)^\top (f_n - y_n), \tag{3.10}$$

with the Jacobian $\mathrm{J}_{\boldsymbol{\theta}} f_n \in \mathbb{R}^{1\times D}$ (Definition 2.4). The Hessian is

$$\nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{2}{|\mathbb{D}|} \sum_{(x_n, y_n)\in\mathbb{D}} (\mathrm{J}_{\boldsymbol{\theta}} f_n)^\top \mathrm{J}_{\boldsymbol{\theta}} f_n + \frac{2}{|\mathbb{D}|} \sum_{(x_n, y_n)\in\mathbb{D}} \nabla^2_{\boldsymbol{\theta}} f_n \, (f_n - y_n) \,. \tag{3.11a}$$

The first term is the PSD *Gauss-Newton matrix* (up to scaling)

$$\boldsymbol{G}_{\mathbb{D}}(\boldsymbol{\theta}) = \frac{2}{|\mathbb{D}|} \sum_{(x_n, y_n)\in\mathbb{D}} (\mathrm{J}_{\boldsymbol{\theta}} f_n)^\top \mathrm{J}_{\boldsymbol{\theta}} f_n \,, \tag{3.11b}$$

and the second term is the residual matrix

$$\boldsymbol{R}_{\mathbb{D}}(\boldsymbol{\theta}) = \frac{2}{|\mathbb{D}|} \sum_{(x_n, y_n)\in\mathbb{D}} \nabla^2_{\boldsymbol{\theta}} f_n \, (f_n - y_n) \,. \tag{3.11c}$$

The *Gauss-Newton matrix* approximates the Hessian through first-order information of the model, $(\mathrm{J}_{\boldsymbol{\theta}} f_n)^\top = \nabla_{\boldsymbol{\theta}} f_n$, which is cheap to compute. For vanishing residual terms $\lim_{(f_n - y_n)\to 0}$ (as the model predictions match the labels), or linear models ($\nabla^2_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}} = \mathbf{0}$) it corresponds to the Hessian.

Update Rule 3.10 (Gauss-Newton method (simplified)) For the nonlinear least squares task Equation (3.9), a Gauss-Newton step is[2]

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{G}_{\mathbb{D}}(\boldsymbol{\theta}_t)^{-1} \nabla_{\boldsymbol{\theta}_t} \mathcal{L}(\boldsymbol{\theta}_t) \,, \tag{3.12}$$

with the gradient and Gauss-Newton matrix from Definition 2.4 and Equation (3.11b) (practical implementations vary and often introduce additional hyperparameters such as a learning rate, damping term, mini-batch size, *etc.*).

2: Stacking Jacobians and residuals,

$$\boldsymbol{J}(\boldsymbol{\theta}) = \begin{pmatrix} \mathrm{J}_{\boldsymbol{\theta}} f_1 \\ \mathrm{J}_{\boldsymbol{\theta}} f_2 \\ \vdots \\ \mathrm{J}_{\boldsymbol{\theta}} f_{|\mathbb{D}|} \end{pmatrix} \in \mathbb{R}^{|\mathbb{D}|\times D} \,,$$

$$\boldsymbol{r}(\boldsymbol{\theta}) = \begin{pmatrix} f_1 - y_1 \\ f_2 - y_2 \\ \vdots \\ f_{|\mathbb{D}|} - y_{|\mathbb{D}|} \end{pmatrix} \in \mathbb{R}^{|\mathbb{D}|} \,,$$

absorbs the sums into matrix multiplies

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \left(\boldsymbol{J}(\boldsymbol{\theta}_t)^\top \boldsymbol{J}(\boldsymbol{\theta}_t)\right)^{-1} \boldsymbol{J}(\boldsymbol{\theta}_t)^\top \boldsymbol{r}(\boldsymbol{\theta}_t)$$

which can be solved through

$$\boldsymbol{J}(\boldsymbol{\theta}_t)^\top \boldsymbol{J}(\boldsymbol{\theta}_t) \boldsymbol{x} = -\boldsymbol{J}(\boldsymbol{\theta}_t)^\top \boldsymbol{r}(\boldsymbol{\theta}_t)$$

via JVPs & VJPs in combination with CG.

3.2.3 The Generalized Gauss-Newton Matrix

The *generalized Gauss-Newton (GGN) matrix* is a PSD approximation to the Hessian that generalizes the GN through abstraction via empirical risk minimization (see Section 2.1.1). The GGN can be understood through different perspectives, and, using the probabilistic interpretation of the model, is related to the natural gradient method through its connections to the Fisher (Section 3.2.4) .

From Gauss-Newton to Generalized Gauss-Newton

The GN matrix from Section 3.2.3 stems from a nonlinear least squares problem that can be viewed as supervised regression (Example 2.1), *i.e.* mean squared error loss function, with scalar-valued labels ($C = 1$). For the general case of empirical risk minimization (Equation (2.4b)), the Hessian decomposes due to the split between model and loss function,

$$\ell(\boldsymbol{\theta}) = \ell(\cdot, \boldsymbol{y}) \circ f_{\boldsymbol{\theta}}(\boldsymbol{x}) \tag{3.13a}$$

as a result of the Hessian chain rule Theorem 3.1. For a single datum $(\boldsymbol{x}, \boldsymbol{y})$ with prediction $\boldsymbol{f} := f_{\boldsymbol{\theta}}(\boldsymbol{x})$, this yields

$$\nabla^2_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) = (\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f})^\top \left[\nabla^2_{\boldsymbol{f}} \ell(\boldsymbol{f}, \boldsymbol{y})\right] \mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f} + \sum_{c=1}^{C} \left(\nabla^2_{\boldsymbol{\theta}} [\boldsymbol{f}]_c\right) \nabla_{\boldsymbol{f}} \ell(\boldsymbol{f}, \boldsymbol{y}) \tag{3.13b}$$

The arrangement of partial derivatives in the generalizations of Jacobian Definition 2.5 and Hessian Definition 3.2 implies the following chain rule generalization for second-order derivatives:

Theorem 3.1 (Chain rule for the generalized Hessian) Let $\boldsymbol{b} : \mathbb{R}^n \to \mathbb{R}^m$ and $\boldsymbol{c} : \mathbb{R}^m \to \mathbb{R}^p$ be twice differentiable and $\boldsymbol{d} = \boldsymbol{c} \circ \boldsymbol{b} : \mathbb{R}^n \to \mathbb{R}^p, \boldsymbol{a} \mapsto \boldsymbol{d}(\boldsymbol{a}) = \boldsymbol{c}(\boldsymbol{b}(\boldsymbol{a}))$. The relation between the Hessian of $\boldsymbol{d}$ and the Jacobians and Hessians of the constituents $\boldsymbol{c}$ and $\boldsymbol{b}$ is given by

$$\begin{aligned}&\nabla^2_{\boldsymbol{a}} \boldsymbol{d}(\boldsymbol{a}) \\ &\quad = \left[\boldsymbol{I}_p \otimes \mathrm{J}_{\boldsymbol{a}} \boldsymbol{b}(\boldsymbol{a})\right]^\top \left[\nabla^2_{\boldsymbol{b}} \boldsymbol{c}(\boldsymbol{b})\right] \mathrm{J}_{\boldsymbol{a}} \boldsymbol{b}(\boldsymbol{a}) \\ &\qquad + \left[\mathrm{J}_{\boldsymbol{b}} \boldsymbol{c}(\boldsymbol{b}) \otimes \boldsymbol{I}_n\right] \nabla^2_{\boldsymbol{a}} \boldsymbol{b}(\boldsymbol{a})\end{aligned} \tag{3.14}$$

[restricted from 103, Chapter 6.10].

The matrix/vector case is analogous. Matrix differential calculus [103] is a useful tool to easily read off the Hessian from specific expressions, see Appendix A.1. For applications of the Hessian chain rule, see Chapter 4.

The first term carries curvature information of the loss function, while the second term contains curvature information of the model. Because $\ell(\boldsymbol{f}, \boldsymbol{y})$ is convex in $\boldsymbol{f}$, the first term is PSD, whereas the second term is indefinite in general.

The GGN is the first term and neglects curvature information of the model. For the empirical risk Equation (2.4b), and using the shorthand $\boldsymbol{f}_n := f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$, the Hessian is

$$\boldsymbol{H}_{\mathbb{D}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} \nabla^2_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) = \boldsymbol{G}_{\mathbb{D}}(\boldsymbol{\theta}) + \mathbb{R}_{\mathbb{D}}(\boldsymbol{\theta}) \tag{3.15a}$$

with the *generalized Gauss-Newton matrix*

$$\boldsymbol{G}_{\mathbb{D}}(\boldsymbol{\theta}) := \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} (\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top \left[\nabla^2_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n)\right] \mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n \tag{3.15b}$$

and the *residual matrix*

$$\boldsymbol{R}_{\mathbb{D}}(\boldsymbol{\theta}) := \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} \sum_{c=1}^{C} \left(\nabla^2_{\boldsymbol{\theta}} [\boldsymbol{f}_n]_c\right) \nabla_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) \,. \tag{3.15c}$$

This decomposition reduces to Equation (3.11) for nonlinear least squares, where $\nabla^2_{\boldsymbol{f}}(\boldsymbol{x}) \ell(\boldsymbol{f}, \boldsymbol{y}) = {}^2/_C \boldsymbol{I}$ (Table 4.1), $\nabla_{\boldsymbol{f}} \ell(\boldsymbol{f}, \boldsymbol{y}) = {}^2/_C(\boldsymbol{f} - \boldsymbol{y})$ (Table 2.2), and $C = 1$. Therefore, the GGN is a generalization of the GN via the chain rule applied to the model-loss function split.

From Linearization to Generalized Gauss-Newton

Alternatively, one can replace the model in $\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})$ with a linear Taylor expansion around $\boldsymbol{\theta}$,

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) \leftrightarrow \hat{f}_{\boldsymbol{\theta}'}(\boldsymbol{x}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}) + \left[\mathrm{J}_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x})\right] (\boldsymbol{\theta}' - \boldsymbol{\theta}) \,.$$

This eliminates second-order terms in the model, *i.e.* $\nabla^2_{\boldsymbol{\theta}'} \hat{f}_{\boldsymbol{\theta}'}(\boldsymbol{x}) = \boldsymbol{0}$. Application of the Hessian chain rule to the loss with a linearized model,

$$\hat{\ell}(\boldsymbol{\theta}') = \ell(\cdot, \boldsymbol{y}) \circ \hat{f}_{\boldsymbol{\theta}'}(\boldsymbol{x}) \,, \tag{3.16a}$$

and using the shorthand $\hat{f}_{\boldsymbol{\theta}'}(\boldsymbol{x}) := \hat{\boldsymbol{f}}$, yields

$$\nabla^2_{\boldsymbol{\theta}'} \hat{\ell}(\boldsymbol{\theta}') = \left(\mathrm{J}_{\boldsymbol{\theta}'} \hat{\boldsymbol{f}}\right)^\top \left[\nabla^2_{\hat{\boldsymbol{f}}} \ell(\hat{\boldsymbol{f}}, \boldsymbol{y})\right] \mathrm{J}_{\boldsymbol{\theta}'} \hat{\boldsymbol{f}} \,. \tag{3.16b}$$

At the expansion point where model predictions and Jacobians match, this yields the first term of the decomposition Equation (3.13b)

$$\left(\nabla^2_{\boldsymbol{\theta}'} \ell(\hat{f}_{\boldsymbol{\theta}'}(\boldsymbol{x}), \boldsymbol{y})\right)\Big|_{\boldsymbol{\theta}'=\boldsymbol{\theta}} = (\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f})^\top \left[\nabla^2_{\boldsymbol{f}} \ell(\boldsymbol{f}, \boldsymbol{y})\right] \mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f} \,. \tag{3.16c}$$

This carries over the empirical risk $\hat{\mathcal{L}}(\boldsymbol{\theta}') = {}^1/_{|\mathbb{D}|} \sum_{(\mathbf{x}_n,\mathbf{y}_n)\in\mathbb{D}} \hat{\ell}(\boldsymbol{\theta}')$ under a linearized model, whose Hessian is the GGN from Equation (3.15b),

$$\left(\nabla^2_{\boldsymbol{\theta}'} \hat{\mathcal{L}}(\boldsymbol{\theta}')\right)\Big|_{\boldsymbol{\theta}'=\boldsymbol{\theta}} = \boldsymbol{G}_{\mathbb{D}}(\boldsymbol{\theta}) \tag{3.16d}$$

Hence, the GGN is a Hessian approximation that neglects curvature from the model, and becomes equivalent to the Hessian for linear models.

3.2.4 Natural Gradient Descent & the Fisher

Natural gradient descent [5, NGD] uses the Fisher information matrix as curvature matrix. The natural gradient provides the direction of steepest in the space of probability distributions described by a model (Figure 3.2, recall the probabilistic interpretation of certain empirical risks from Section 2.1.3). The Fisher is connected to the GGN from Section 3.2.3 and for most models used in modern ML, is equivalent but provides a probabilistic interpretation.

Steepest Descent Direction & Notion of Distance

Concept (steepest descent direction): Consider an arbitrary objective function $\mathcal{L} : \Theta \to \mathbb{R}$ with $\Theta = \mathbb{R}^D$. At a location $\boldsymbol{\theta} \in \Theta$, the steepest descent direction is the direction in which the objective increases at the fastest rate, *i.e.* per infinitesimally small distance moved. Formally,

$$\Delta\boldsymbol{\theta}^{(\text{steepest})} = \lim_{\epsilon\to 0} \frac{1}{\epsilon} \underset{\substack{\Delta\boldsymbol{\theta} \\ d(\boldsymbol{\theta},\boldsymbol{\theta}+\Delta\boldsymbol{\theta})\leq\epsilon}}{\arg\min} \mathcal{L}(\boldsymbol{\theta}+\Delta\boldsymbol{\theta}) \tag{3.17}$$

and depends on the distance measure $d(\cdot,\cdot)$ between elements in a small neighborhood around $\boldsymbol{\theta}$.

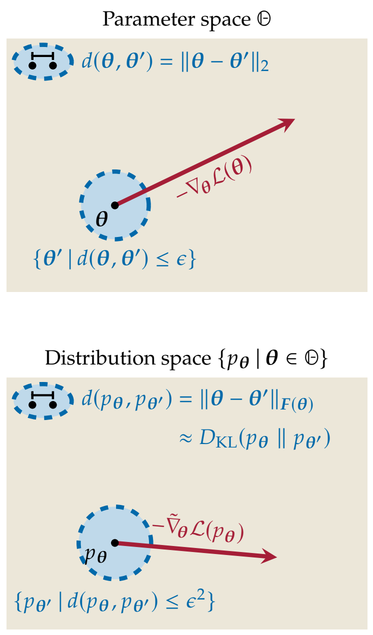

Parameter space Θ
$d(\boldsymbol{\theta},\boldsymbol{\theta}') = \|\boldsymbol{\theta}-\boldsymbol{\theta}'\|_2$
$-\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$
$\boldsymbol{\theta}$
$\{\boldsymbol{\theta}' \mid d(\boldsymbol{\theta},\boldsymbol{\theta}') \leq \epsilon\}$
Distribution space $\{p_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \Theta\}$
$d(p_{\boldsymbol{\theta}},p_{\boldsymbol{\theta}'}) = \|\boldsymbol{\theta}-\boldsymbol{\theta}'\|_{\boldsymbol{F}(\boldsymbol{\theta})}$
$\approx D_{\text{KL}}(p_{\boldsymbol{\theta}} \parallel p_{\boldsymbol{\theta}'})$
$-\tilde{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(p_{\boldsymbol{\theta}})$
$p_{\boldsymbol{\theta}}$
$\{p_{\boldsymbol{\theta}'} \mid d(p_{\boldsymbol{\theta}},p_{\boldsymbol{\theta}'}) \leq \epsilon^2\}$

Figure 3.2: Gradient descent and natural gradient descent via steepest descent. Gradient descent (*top*) follows the direction of steepest descent in Euclidean parameter space. NGD (*bottom*) considers the space of distributions, where local distances are measured via a quadratic expansion of the KL divergence, which gives rise to the Fisher. Details in the text. Figure inspired by [107].

Gradient descent as steepest descent: Using the Euclidean 2-norm to measure distances in Θ via $d(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2) = \|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\|_2$, the steepest descent direction points along the negative gradient $-\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ [107, Chapter 6] ,

$$\lim_{\epsilon\to 0} \frac{1}{\epsilon} \underset{\Delta\boldsymbol{\theta}:\|\Delta\boldsymbol{\theta}\|_2\leq\epsilon}{\arg\min} \mathcal{L}(\boldsymbol{\theta}+\Delta\boldsymbol{\theta}) = \frac{-\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})}{\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})\|_2}\,. \tag{3.18}$$

Natural gradient as steepest descent: Section 2.1.3 showed that in many tasks, such as regression (Example 2.1) and softmax cross-entropy classification (Example 2.2), the parameters $\boldsymbol{\theta}$ model a probability distribution $p_{\boldsymbol{\theta}}(\mathbf{z})$ over a random variable $\mathbf{z} \in \Omega$. One could therefore establish a different notion of distance by comparing probability distributions. The KL divergence is a similarity measure between densities, but it is not a proper metric; *e.g.* it is not symmetric in its arguments. The steepest descent direction only requires measuring distances within an infinitesimal ball though. For this purpose, a metric—described by the Fisher—can be established through Taylor expansion of the KL divergence.

Starting from the KL divergence between two infinitesimally close distributions $p_{\boldsymbol{\theta}}(z), p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(z)$,

$$D_{\mathrm{KL}}(p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}} \parallel p_{\boldsymbol{\theta}}) = \int_{\Omega} p_{\boldsymbol{\theta}}(z)\left[\log p_{\boldsymbol{\theta}}(z) - \log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(z)\right] \mathrm{d}z \tag{3.19}$$

the first step is to Taylor-expand the logarithm around $\boldsymbol{\theta}$,

$$\begin{aligned}\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(z) &= \log p_{\boldsymbol{\theta}}(z) + (\Delta\boldsymbol{\theta})^{\top}\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z) \\ &\quad + \frac{1}{2}(\Delta\boldsymbol{\theta})^{\top}\nabla^2_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z)(\Delta\boldsymbol{\theta}) + \mathcal{O}\left((\Delta\boldsymbol{\theta})^3\right) .\end{aligned}$$

Inserting this into Equation (3.19) results in

$$\begin{aligned}D_{\mathrm{KL}}(p_{\boldsymbol{\theta}}(z) \parallel p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(z)) = -\Delta\boldsymbol{\theta}^{\top} &\underbrace{\int_{\Omega} p_{\boldsymbol{\theta}}(z)\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z)\, \mathrm{d}z}_{=\,0,\ \text{see Remark 3.1}} \\ &- \frac{1}{2}\Delta\boldsymbol{\theta}^{\top}\left(\int_{\Omega} p_{\boldsymbol{\theta}}(z)\nabla^2_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z)\, \mathrm{d}z\right)\Delta\boldsymbol{\theta} \\ &+ \mathcal{O}\left((\Delta\boldsymbol{\theta})^3\right)\end{aligned}$$

The Hessian in the second term is expressed as (Remark 3.2)

$$\nabla^2_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z) = -\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z) \left(\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z)\right)^{\top} + \frac{1}{p_{\boldsymbol{\theta}}(z)}\nabla^2_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(z)\,, \tag{3.20}$$

whose second term again vanishes in expectation (Remark 3.3). Hence,

$$D_{\mathrm{KL}}(p_{\boldsymbol{\theta}}(z) \parallel p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(z)) = \frac{1}{2}\Delta\boldsymbol{\theta}^{\top}\boldsymbol{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} + \mathcal{O}\left((\Delta\boldsymbol{\theta})^3\right) \tag{3.21a}$$

with the two equivalent forms of the Fisher

$$\begin{aligned}\boldsymbol{F}(\boldsymbol{\theta}) &= -\int_{\Omega} p_{\boldsymbol{\theta}}(z)\nabla^2_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z)\mathrm{d}z &\text{(3.21b)}\\ &= \int_{\Omega} p_{\boldsymbol{\theta}}(z)(\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z))(\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z))^{\top}\mathrm{d}z &\text{(3.21c)}\end{aligned}$$

Equation (3.21b) will be helpful to draw connections to the Hessian, and Equation (3.21c) provides links to the GGN.

Remark 3.1 (The log-probability's θ-gradient vanishes in expectation)

$$\begin{aligned}&-\int_{\Omega} p_{\boldsymbol{\theta}}(z)\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(z)\,\mathrm{d}z \\ &\quad = -\int_{\Omega} p_{\boldsymbol{\theta}}(z)\frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(z)}{p_{\boldsymbol{\theta}}(z)}\,\mathrm{d}z \\ &\quad = -\nabla_{\boldsymbol{\theta}}\left(\int_{\Omega} p_{\boldsymbol{\theta}}(z)\,\mathrm{d}z\right) \\ &\quad = -\nabla_{\boldsymbol{\theta}} 1 = 0\end{aligned}$$

Remark 3.2 (Decomposition of the log-probability's θ-Hessian) Consider element (i, j) of the Hessian,

$$\begin{aligned}&\frac{\partial^2 \log p_{\boldsymbol{\theta}}(z)}{\partial\theta_i\partial\theta_j} \\ &\quad = \frac{\partial}{\partial\theta_i}\left(\frac{1}{p_{\boldsymbol{\theta}}(z)}\frac{\partial p_{\boldsymbol{\theta}}(z)}{\partial\theta_j}\right) \\ &\quad = -\frac{1}{p_{\boldsymbol{\theta}}(z)^2}\frac{\partial p_{\boldsymbol{\theta}}(z)}{\partial\theta_j}\frac{\partial p_{\boldsymbol{\theta}}(z)}{\partial\theta_i} \\ &\qquad + \frac{1}{p_{\boldsymbol{\theta}}(z)}\frac{\partial^2 p_{\boldsymbol{\theta}}(z)}{\partial\theta_i\partial\theta_j} \\ &\quad = -\frac{\partial \log p_{\boldsymbol{\theta}}(z)}{\partial\theta_j}\frac{\partial \log p_{\boldsymbol{\theta}}(z)}{\partial\theta_i} \\ &\qquad + \frac{1}{p_{\boldsymbol{\theta}}(z)}\frac{\partial^2 p_{\boldsymbol{\theta}}(z)}{\partial\theta_i\partial\theta_j}\end{aligned}$$

which, in vector notation, translates into Equation (3.20).

Remark 3.3 (Hessian of the model distribution vanishes in expectation)

$$\begin{aligned}&\int_{\Omega} p_{\boldsymbol{\theta}}(z)\frac{1}{p_{\boldsymbol{\theta}}(z)}\nabla^2_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(z)\,\mathrm{d}z \\ &\quad = \nabla^2_{\boldsymbol{\theta}}\left(\int_{\Omega} p_{\boldsymbol{\theta}}(z)\,\mathrm{d}z\right) \\ &\quad = \nabla^2_{\boldsymbol{\theta}} 1 = 0\end{aligned}$$

Locally, the KL divergence Equation (3.21a) gives rise to a metric induced by the Fisher-norm $\|\cdot\|_{\boldsymbol{F}(\boldsymbol{\theta})}$,

$$d(p_{\boldsymbol{\theta}}, p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}) = \|\Delta\boldsymbol{\theta}\|_{\boldsymbol{F}(\boldsymbol{\theta})} := \sqrt{(\Delta\boldsymbol{\theta})^{\top}\boldsymbol{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta}} \tag{3.22}$$

Ollivier et al. [120] show that the steepest descent for a function $\mathcal{L}(p_{\boldsymbol{\theta}})$ points along the negative natural gradient $-\tilde{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(p_{\boldsymbol{\theta}}) := -\boldsymbol{F}(\boldsymbol{\theta})^{-1}\nabla_{\boldsymbol{\theta}}\mathcal{L}(p_{\boldsymbol{\theta}})$,

$$\lim_{\epsilon\to 0}\frac{1}{\epsilon}\ \underset{\|\Delta\boldsymbol{\theta}\|_{\boldsymbol{F}(\boldsymbol{\theta})}\le\epsilon^2/2}{\arg\min_{\Delta\boldsymbol{\theta}}}\ \mathcal{L}(p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}) = -\frac{\tilde{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(p_{\boldsymbol{\theta}})}{\left\|\tilde{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(p_{\boldsymbol{\theta}})\right\|_{\boldsymbol{F}(\boldsymbol{\theta})^{-1}}}\,. \tag{3.23}$$

Natural Gradient Descent & Fisher in Empirical Risk Minimization

Section 2.1.3 presented connections of empirical risk minimization to learning $p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$ via $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) p(\boldsymbol{x}) = q(\boldsymbol{y} \mid \boldsymbol{f}) p(\boldsymbol{x})$ with a negative log-likelihood loss $-\log q(\boldsymbol{y} \mid \boldsymbol{f}) = \ell(\boldsymbol{f}, \boldsymbol{y})$ and $f_{\boldsymbol{\theta}}(\boldsymbol{x}) := \boldsymbol{f}$. In these cases, the empirical risk Equation (2.4b) depends on a probability distribution, and the distribution space's geometry gives rise to NGD with $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{y})$ in the Fisher Equation (3.21). After simplifying the $\boldsymbol{\theta}$-derivatives and grouping dependencies of $\boldsymbol{x}$ and $\boldsymbol{y}$, the Fisher reads

$$\begin{aligned}
&\boldsymbol{F}(\boldsymbol{\theta}) \\
&= \int_{\mathbb{X}} p(\boldsymbol{x}) \left(- \int_{\mathbb{Y}} p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) \nabla^2_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) \, \mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x} \\
&= \int_{\mathbb{X}} p(\boldsymbol{x}) \left(\int_{\mathbb{Y}} p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) (\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x})) (\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}))^\top \, \mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x} \,,
\end{aligned}$$

or in short form

$$\begin{aligned}
\boldsymbol{F}(\boldsymbol{\theta}) &= \mathbb{E}_{\boldsymbol{x} \sim p(\boldsymbol{x})} \mathbb{E}_{\boldsymbol{y} \sim p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x})} \left[-\nabla^2_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) \right] \\
&= \mathbb{E}_{\boldsymbol{x} \sim p(\boldsymbol{x})} \mathbb{E}_{\boldsymbol{y} \sim p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x})} \left[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}) (\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}))^\top \right]
\end{aligned} \tag{3.24}$$

Next, replace $p(\boldsymbol{x}) \leftrightarrow p_{\mathbb{D}}(\boldsymbol{x})$ by empirical approximation through data,

$$\begin{aligned}
&\boldsymbol{F}_{\mathbb{D}}(\boldsymbol{\theta}) \\
&= \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \_) \in \mathbb{D}} \mathbb{E}_{\boldsymbol{y} \sim p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n)} \left[-\nabla^2_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n) \right] \\
&= \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \_) \in \mathbb{D}} \mathbb{E}_{\boldsymbol{y} \sim p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n)} \left[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n) (\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n))^\top \right]
\end{aligned} \tag{3.25}$$

(note that the Fisher is independent of the labels $\{\boldsymbol{y}_n\}$!). Using the relation between log-likelihood and loss function leads to (with $\boldsymbol{f}_n := f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$)

$$\begin{aligned}
\boldsymbol{F}_{\mathbb{D}}(\boldsymbol{\theta}) &= \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \_) \in \mathbb{D}} \mathbb{E}_{\boldsymbol{y} \sim q(\boldsymbol{y} \mid \boldsymbol{f}_n)} \left[\nabla^2_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}) \right] \\
&= \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \_) \in \mathbb{D}} \mathbb{E}_{\boldsymbol{y} \sim q(\boldsymbol{y} \mid \boldsymbol{f}_n)} \left[\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}) (\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}))^\top \right] .
\end{aligned}$$

The second equality views the Fisher as expected gradient outer product. With the Jacobian chain rule Theorem 2.2 applied to $\ell \circ f_{\boldsymbol{\theta}}$, one obtains

$$\begin{aligned}
&\boldsymbol{F}_{\mathbb{D}}(\boldsymbol{\theta}) \\
&= \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \_) \in \mathbb{D}} \mathbb{E}_{\boldsymbol{y} \sim q(\boldsymbol{y} \mid \boldsymbol{f}_n)} \left[(\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top \nabla_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}) ((\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top \nabla_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}))^\top \right] \\
&= \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \_) \in \mathbb{D}} (\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top \mathbb{E}_{\boldsymbol{y} \sim q(\boldsymbol{y} \mid \boldsymbol{f}_n)} \left[\nabla_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}) (\nabla_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}))^\top \right] \mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n
\end{aligned} \tag{3.26}$$

Equation (3.26) offers an interesting perspective to approximate the Fisher via MC sampling through computing gradients of the loss on targets drawn from q and will be used later, *e.g.* Chapters 5 and 7 and Appendix B.1.

The first equality in Equation (3.25) views the Fisher as an expected Hessian under the model's likelihood[3]. However, in general it does not coincide with the Hessian, $\boldsymbol{F}_{\mathbb{D}}(\boldsymbol{\theta}) \neq \boldsymbol{H}_{\mathbb{D}}(\boldsymbol{\theta})$, as $\mathbb{E}_{\boldsymbol{y}\sim q(\boldsymbol{y}\mid \boldsymbol{f}_n)}[\nabla^2_{\boldsymbol{\theta}}\ell(\boldsymbol{f}_n, \boldsymbol{y})] \neq \nabla^2_{\boldsymbol{\theta}}\ell(\boldsymbol{f}_n, \boldsymbol{y}_n)$. After applying the Hessian chain rule Theorem 3.1 onto $\ell \circ f_{\boldsymbol{\theta}}$,

3: It is important to stress the expectation over the *model's likelihood* $\boldsymbol{y} \mid \boldsymbol{x}$, and *not* over the empirical data. It may be tempting to replace the expectation over $\boldsymbol{y}$ with the empirical distribution $p_{\mathbb{D}}(\boldsymbol{y})$. This leads to the empirical Fisher (Section 3.2.5), which is often used instead of the Fisher, but has limitations for applications like optimization [92].

$$
\begin{aligned}
\boldsymbol{F}_{\mathbb{D}}(\boldsymbol{\theta}) &= \frac{1}{|\mathbb{D}|}\sum_{(\boldsymbol{x}_n,\_)\in\mathbb{D}} (\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{f}_n)^\top \mathbb{E}_{\boldsymbol{y}\sim q(\boldsymbol{y}\mid \boldsymbol{f}_n)}\left[\nabla^2_{\boldsymbol{f}_n}\ell(\boldsymbol{f}_n,\boldsymbol{y})\right]\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{f}_n \\
&\quad + \frac{1}{|\mathbb{D}|}\sum_{(\boldsymbol{x}_n,\_)\in\mathbb{D}}\sum_{c=1}^{C}\nabla^2_{\boldsymbol{\theta}}([\boldsymbol{f}_n]_c)\underbrace{\mathbb{E}_{\boldsymbol{y}\sim q(\boldsymbol{y}\mid \boldsymbol{f}_n)}\left[\nabla_{f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)}\ell(\boldsymbol{f}_n,\boldsymbol{y})\right]}_{=\boldsymbol{0}\text{, same argument as Remark 3.1}} \\
&= \frac{1}{|\mathbb{D}|}\sum_{(\boldsymbol{x}_n,\_)\in\mathbb{D}} (\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{f}_n)^\top \mathbb{E}_{\boldsymbol{y}\sim q(\boldsymbol{y}\mid \boldsymbol{f}_n)}\left[\nabla^2_{\boldsymbol{f}_n}\ell(\boldsymbol{f}_n,\boldsymbol{y})\right]\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{f}_n\,,
\end{aligned}
\tag{3.27}
$$

one can see connections to the GGN (Equation (3.15b)).

Connections Between Fisher & GGN

Update Rule 3.11 (Natural gradient descent (simplified)) For empirical risks with loss functions that have a probabilistic interpretation, an NGD step is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{F}_{\mathbb{D}}(\boldsymbol{\theta}_t)^{-1}\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}_t)$$

with the gradient from Definition 2.4 and the Fisher from Equation (3.26) or Equation (3.27) (practical implementations vary and often introduce additional hyperparameters such as a learning rate, damping term, mini-batch size, *etc.*).

The Fisher (Equation (3.27)) and the GGN (Equation (3.15b)) are structurally similar. Both are identical if the expected Hessian of the loss *w.r.t.* the model's prediction under the model is identical to the empirical Hessian,

$$\nabla^2_{\boldsymbol{f}_n}\ell(\boldsymbol{f}_n,\boldsymbol{y}_n) = \mathbb{E}_{\boldsymbol{y}\sim q(\boldsymbol{y}\mid \boldsymbol{f}_n)}\left[\nabla^2_{\boldsymbol{f}_n}\ell(\boldsymbol{f}_n,\boldsymbol{y})\right] \quad\Longrightarrow\quad \boldsymbol{G}_{\mathbb{D}} = \boldsymbol{F}_{\mathbb{D}}\,. \tag{3.28}$$

For both softmax cross-entropy and square loss, this $\boldsymbol{f}$-Hessian is independent of $\boldsymbol{y}$, see Table 4.1. Therefore, the expectation has no effect, and the above equality is satisfied: for least squares regression (Example 2.1) and softmax cross-entropy classification (Example 2.1), the Fisher equals the GGN[4]. NGD (Update Rule 3.11) for those models is thus equivalent to the GGN and can be seen as an approximation of Newton's method (Update Rule 3.9) with the GGN instead of the Hessian.

4: There are more scenarios in which Fisher and GGN coincide, for instance if $q(\boldsymbol{y} \mid \boldsymbol{f})$ is an exponential family distribution with natural parameters $\boldsymbol{f}$, see [107, Chapter 9] for a detailed presentation.

3.2.5 The Gradient Covariance Matrix

The Fisher's form in Equation (3.26) reminds of an uncentered second moment of "would-be" gradients sampled from the likelihood implied by the model [124]. The uncentered gradient covariance matrix

$$
\begin{aligned}
\boldsymbol{K}_{\mathbb{D}}(\boldsymbol{\theta}) &= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p_{\mathbb{D}}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y})(\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y}))^\top\right] \\
&= \frac{1}{|\mathbb{D}|}\sum_{(\boldsymbol{x}_n,\boldsymbol{y}_n)\in\mathbb{D}}\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{f}_n,\boldsymbol{y}_n)(\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{f}_n,\boldsymbol{y}_n))^\top
\end{aligned}
\tag{3.29}
$$

if often referred to as empirical Fisher in some of the ML literature, because it follows from replacing the expectation over the model's distribution $q(\boldsymbol{y} \mid \boldsymbol{f}_n)$ by the expectation over the empirical data distribution $p_{\mathbb{D}}(\boldsymbol{y})$ in the Fisher. However, empirical Fisher and Fisher are, almost always, *not* identical. The importance to distinguish them has been expressed in works like [92] and [162], and it is questionable whether the uncentered gradient covariance Equation (3.29) approximates curvature.

However, an alternative perspective to use $\boldsymbol{K}_{\mathbb{D}}$ for optimization is to make stochastic gradient-based optimization aware of the noise [9] (sketch in Figure 3.4). With the centered gradient covariance on data $\mathbb{D}$

$$\begin{aligned}\boldsymbol{\Sigma}_{\mathbb{D}}(\boldsymbol{\theta}) &= \mathrm{Var}_{(\boldsymbol{x},\boldsymbol{y})\sim p_{\mathbb{D}}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y})\right] \\ &= \frac{1}{|\mathbb{D}|}\sum_{(\boldsymbol{x}_n,\boldsymbol{y}_n)\in\mathbb{D}}\left(\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{f}_n,\boldsymbol{y}_n)-\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})\right)\left(\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{f}_n,\boldsymbol{y}_n)-\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})\right)^\top \\ &= \boldsymbol{K}_{\mathbb{D}}(\boldsymbol{\theta})-\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})^\top\,,\end{aligned} \tag{3.30}$$

an update step of the form (usually with $\mathbb{D}=\mathbb{B}_t$)

$$\boldsymbol{\theta}_{t+1}=\boldsymbol{\theta}_t-\boldsymbol{\Sigma}_{\mathbb{D}}(\boldsymbol{\theta}_t)^{-1}\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}_t)$$

can be regarded as re-scaling the gradient according to fluctuations: directions that are subject to stronger noise will be shortened more strongly than directions of small noise.

The gradient covariance has also been proposed to adapt the batch size during training [6, 11, 19, 26] and to stop training before overfitting sets in [104]. Similar ideas of variance-adaptation can be found in currently popular deep learning optimizers such as Adam (Update Rule 3.7), that keep an exponential average of the squared mini-batch gradient. This shows up in the diagonal of the gradient covariance (second term)

$$\mathrm{diag}(\boldsymbol{\Sigma}_{\mathbb{D}}(\boldsymbol{\theta}))=\frac{1}{|\mathbb{D}|}\left(\sum_{(\boldsymbol{x}_n,\boldsymbol{y}_n)\in\mathbb{D}}\left(\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{f}_n,\boldsymbol{y}_n)\right)^{\odot 2}\right)-\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})^{\odot 2}\,. \tag{3.31}$$

See [10] for a precise analysis of the connections.

The empirical Fisher is formed by the gradients that make up the mini-batch average gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$, and therefore basically free to compute on top of the gradient. But the required per-sample gradients are difficult to access in popular ML libraries because they are not agnostic to this per-sample structure (see Section 2.3.2). This complicates efficient usage of the gradient covariance matrix for applications like optimization, where performance is key.

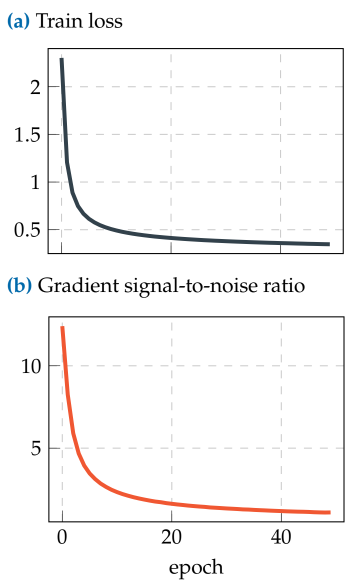


Figure 3.4: Illustration of gradient noise during training. Evolution of **(a)** training loss and **(b)** gradient signal-to-noise ratio (computed with BackPACK, Chapter 5) during training (logistic regression on MNIST). As the loss decreases, the gradient noise increases.

3.3 Other Applications

So far, this text showed use cases of higher-order information, in the form of curvature and gradient covariance, for optimization. While optimization is one main component of deep learning, there exist many other applications that require access to richer information. The following briefly describes some applications (mostly) outside the optimization field. Another argument for the utility of higher-order information to better understand the training of neural networks is made by Chapter 6.

3.3.1 Bayesian Deep Learning & Laplace Approximations

Section 2.1.3 described the relation between regularized empirical risk minimization and maximum a posteriori (MAP) estimation. The MAP estimate $\boldsymbol{\theta}_{\text{MAP}}$, can be used to approximate the Bayesian posterior for the deep model's parameters via $p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \delta(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})$. The Laplace approximation [93] extends this posterior approximation to a multivariate Gaussian around $\boldsymbol{\theta}_{\text{MAP}}$ (see Figure 3.5 for an illustration).

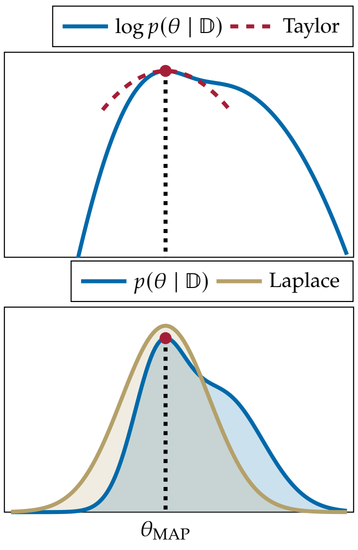


Figure 3.5: Conceptual sketch of the Laplace approximation. The goal is to approximate $p(\theta \mid \mathbb{D})$ with a Gaussian around a mode. (*Top*) First, one identifies a mode θ_{MAP} of the log-posterior $\log p(\theta \mid \mathbb{D})$, then Taylor-expands the latter around it up to second order. (*Bottom*) The exponentiated Taylor expansion is proportional to a Gaussian. Normalizing it yields the desired Gaussian approximation.

Let $\mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}) := \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) + r(\boldsymbol{\theta})$ denote the regularized empirical risk from Equation (2.18) that gives rise to the MAP estimate $\boldsymbol{\theta}_{\text{MAP}}$. Its quadratic Taylor expansion around $\boldsymbol{\theta}_{\text{MAP}}$ (assumed to be a stationary point) is

$$
\begin{aligned}
\mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}) = \mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}_{\text{MAP}}) &+ \underbrace{\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}_{\text{MAP}})^{\top}}_{=\mathbf{0},\text{stationary point}} (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \\
&+ \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})^{\top} \nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}_{\text{MAP}})(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \\
&+ \mathcal{O}\left((\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})^3\right) .
\end{aligned}
$$

Using up to quadratic term in the posterior Equation (2.19) yields

$$
\begin{aligned}
p(\boldsymbol{\theta} \mid \mathbb{D}) \approx\, &\text{const}(\mathbb{D}, \boldsymbol{\theta}_{\text{MAP}}) \\
&\times \exp\left[-\frac{|\mathbb{D}|}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}})^{\top} \nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}_{\text{MAP}})(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{MAP}}) \right] ,
\end{aligned}
$$

which corresponds to a Gaussian

$$
p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \mathcal{N}(\boldsymbol{\theta} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{Z} \exp\left[-\frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\mu})^{\top} \boldsymbol{\Sigma}^{-1} (\boldsymbol{\theta} - \boldsymbol{\mu}) \right] \tag{3.32a}
$$

with mean, covariance, and normalization constant

$$
\boldsymbol{\mu} = \boldsymbol{\theta}_{\text{MAP}} , \tag{3.32b}
$$

$$
\boldsymbol{\Sigma} = \left(|\mathbb{D}| \nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}_{\text{MAP}}) \right)^{-1} , \tag{3.32c}
$$

$$
Z = (2\pi)^{D/2} \det\left[\left(|\mathbb{D}| \nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}}^{\text{MAP}}(\boldsymbol{\theta}_{\text{MAP}}) \right)^{-1} \right]^{D/2} . \tag{3.32d}
$$

The Gaussian posterior Equation (3.32a) is useful for Bayesian predictions with neural networks (Equation (2.14)) and various other tasks (*e.g.* model selection [75] and continual learning, see [40] for an overview).

The covariance matrix $\boldsymbol{\Sigma}$ in the Laplace approximation requires the Hessian $\nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}}^{\text{MAP}} = \nabla_{\boldsymbol{\theta}}^2 \mathcal{L}_{\mathbb{D}} + \nabla_{\boldsymbol{\theta}}^2 r$ (specifically, its inverse). While the regularization term often has a simple Hessian, computing the empirical risk's Hessian is expensive and suffers from the same issues of the Laplace approximation being undefined if the Hessian is not PSD. Applications often replace this Hessian with one of the PSD approximations presented in Section 3.2 because a covariance matrix must be PSD.

3.3.2 Model Compression

Neural networks are extremely overparameterized. Reducing their parameter count, also referred to as *pruning*, is important to deploy them on low-resource devices such as mobile phones and often decreases computational cost of inference. In general, the goal is to reduce parameters, while keeping the model's performance (see [18] for a review).

One classic approach for post-training compression of models is the *Optimal Brain Damage/Surgeon (OBD/OBS)* framework of Hassibi and Stork [67] and LeCun et al. [96] that leverages approximate second-order information and is still used nowadays[43, 154, 161, 182]. The idea is to identify unimportant weights through a local quadratic Taylor approximation of the loss. Assuming the model is trained to a stationary point $\boldsymbol{\theta}_\star$, the Taylor expansion's first-order term vanishes. A perturbation $\boldsymbol{\delta} := \boldsymbol{\theta} - \boldsymbol{\theta}_\star$ induces the change $\Delta\mathcal{L}_\mathbb{D}(\boldsymbol{\delta}) := \mathcal{L}_\mathbb{D}(\boldsymbol{\delta}) - \mathcal{L}_\mathbb{D}(\boldsymbol{\theta}_\star)$ in the loss,

$$\Delta\mathcal{L}_\mathbb{D}(\boldsymbol{\delta}) = \underbrace{\left(\nabla_{\boldsymbol{\theta}_\star}\mathcal{L}_\mathbb{D}(\boldsymbol{\theta}_\star)\right)}_{=\mathbf{0},\text{stationary point}}{}^\top\boldsymbol{\delta} + \frac{1}{2}\boldsymbol{\delta}^\top\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)\boldsymbol{\delta} + \mathcal{O}\left(\boldsymbol{\delta}^3\right). \tag{3.33}$$

To simplify the discussion, consider eliminating only a single parameter. For a fixed direction i, the "best" perturbation $\boldsymbol{\delta}_\star(i)$ that eliminates parameter $[\boldsymbol{\theta}_\star]_i$ through satisfying the constraint $\hat{\boldsymbol{e}}_i^\top\boldsymbol{\delta}_\star(i) = [\boldsymbol{\theta}_\star]_i$ (with $\hat{\boldsymbol{e}}_i$ the unit vector in direction i) introduces the smallest increase in the loss. This leads to the constrained optimization problem

$$\boldsymbol{\delta}_\star(i) = \operatorname*{arg\,min}_{\substack{\boldsymbol{\delta}\\ \hat{\boldsymbol{e}}_i^\top\boldsymbol{\delta}-[\boldsymbol{\theta}_\star]_i=0}} \frac{1}{2}\boldsymbol{\delta}^\top\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)\boldsymbol{\delta}. \tag{3.34a}$$

As described in Remark 3.4, the solution to Equation (3.34a) is

$$\boldsymbol{\delta}_\star(i) = -\frac{[\boldsymbol{\theta}_\star]_i\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)^{-1}\hat{\boldsymbol{e}}_i}{[\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)^{-1}]_{i,i}} \tag{3.34b}$$

which induces the change

$$\Delta\mathcal{L}_\mathbb{D}(\boldsymbol{\delta}_\star(i)) = \frac{[\boldsymbol{\theta}_\star]_i^2}{2[\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)^{-1}]_{i,i}} \tag{3.34c}$$

Among these perturbations, $\{\boldsymbol{\delta}_\star(i)\}_{i=1}^D$, the one introducing the smallest change in the loss identifies the parameter to eliminate[5]. This requires computing the pruning statistics Equation (3.34c) for all directions, and therefore evaluation of the inverse Hessian's diagonal. The perturbation Equation (3.34b) requires multiplication by the inverse Hessian.

To simplify these computations, the OBD framework [96, Equation (3)] approximates the Hessian by its diagonal for the induced change Equation (3.33), *i.e.* $\Delta\mathcal{L}_\mathbb{D}^{\text{OBD}}(\boldsymbol{\delta}) := {}^1\!/_2\boldsymbol{\delta}^\top \operatorname{diag}(\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star))\boldsymbol{\delta}$. By substituting $\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star) \leftrightarrow \operatorname{diag}(\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star))$ into the above discussion, one obtains the approximate perturbations $\boldsymbol{\delta}_\star^{\text{OBD}}(i) = [\boldsymbol{\theta}_\star]_i\hat{\boldsymbol{e}}_i$ and pruning statistics $\Delta\mathcal{L}_\mathbb{D}^{\text{OBD}}(\boldsymbol{\delta}_\star^{\text{OBD}}(i)) = {}^{[\boldsymbol{\theta}_\star]_i^2[\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)]_{i,i}}\!/_2$. The diagonal approximation removes the need to access elements of the inverse Hessian.

Remark 3.4 (Optimal perturbation to prune $[\theta]_i$) The constraint in Equation (3.33) is incorporated via a Lagrange multiplier λ,

$$\Delta\mathcal{L}_\mathbb{D}(\boldsymbol{\delta},\lambda) = \frac{1}{2}\boldsymbol{\delta}^\top\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)\boldsymbol{\delta} - \lambda\left(\hat{\boldsymbol{e}}_i^\top\boldsymbol{\delta} - [\boldsymbol{\theta}_\star]_i\right).$$

The optimal perturbation follows as

$$\nabla_{\boldsymbol{\delta}}\Delta\mathcal{L}_\mathbb{D}(\boldsymbol{\delta},\lambda) = \boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)\boldsymbol{\delta} + \lambda\hat{\boldsymbol{e}}_i \overset{!}{=} \mathbf{0}$$
$$\implies \boldsymbol{\delta}(\lambda) = -\lambda\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)^{-1}\hat{\boldsymbol{e}}_i.$$

Substituting this into the induced change yields

$$\Delta\mathcal{L}_\mathbb{D}(\lambda) = -\frac{1}{2}\lambda^2[\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)^{-1}]_{i,i} + \lambda[\boldsymbol{\theta}_\star]_i$$

and

$$\nabla_\lambda\Delta\mathcal{L}_\mathbb{D}(\lambda) \overset{!}{=} 0$$
$$\implies \lambda = \frac{[\boldsymbol{\theta}_\star]_i}{[\boldsymbol{H}_\mathbb{D}(\boldsymbol{\theta}_\star)^{-1}]_{i,i}}.$$

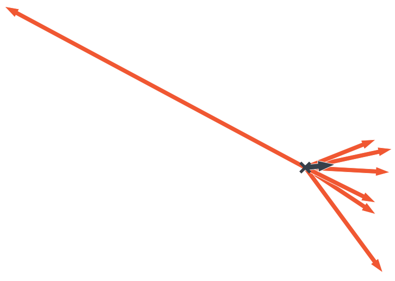
Equations (3.34b) and (3.34c) follow by substitution of $\delta_\star(i) := \boldsymbol{\delta}(\lambda)$.

5: Considering the removal of multiple parameters becomes prohibitive as the search space over indices increases exponentially (see [154] for details). It is therefore common to consider pruning candidates independently by sorting them according to their induced change.

3.3.3 Differential Privacy

The data used for training deep models sometimes contains sensitive information about individuals, like medical data, personal images *etc.*, which is incorporated into the model during training. Adversaries might be able to reconstruct such personal data given only black box access to the model [52]. Often though, such adversaries even have access to details about the training procedure, or explicit access to model parameters[6]. Such knowledge can be leveraged to make attacks more efficient [149].

6: *E.g.* distributed training on mobile devices that avoids sending data to a central server, but transfers the model to devices, which communicates updates to a central model.


(a) Individual gradients

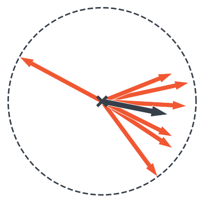

(b) Clipped individual gradients

Figure 3.6: (Sketch) DP-SGD requires access to per-sample gradients. (a) The average gradient (black arrow) might be dominated by the per-sample gradients (orange arrows) of a small number of data. Adversaries might use this to extract potentially sensitive information about that data. **(b)** Clipping per-sample gradients before averaging them limits the influence of individual data. The dashed circle's radius corresponds to the privacy threshold C.

Differential privacy [46, DP] aims at preserving privacy of an algorithm by injecting noise to limit the influence of individual data.

One prominent example for deep learning is differentially-private SGD (DP-SGD) [2, Algorithm 1], which uses a negative average of processed individual gradients over a mini-batch. To bound the influence of individual data, per-sample gradients whose norm exceeds a specific threshold C are clipped back to norm C (Figure 3.6),

$$\tilde{\boldsymbol{g}}_n(\boldsymbol{\theta}_t) = \frac{\boldsymbol{g}_n(\boldsymbol{\theta}_t)}{\max\left(1, \frac{\|\boldsymbol{g}_n(\boldsymbol{\theta}_t)\|_2}{C}\right)}\,.$$

Gaussian noise of scale σ is then added to each gradient,

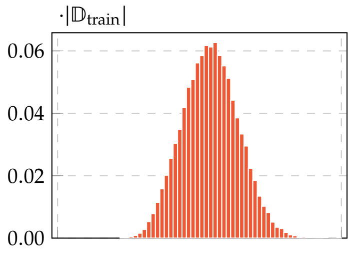
$$\hat{\boldsymbol{g}}_n(\boldsymbol{\theta}_t) = \tilde{\boldsymbol{g}}_n(\boldsymbol{\theta}_t) + \boldsymbol{\epsilon}_n\,, \qquad \boldsymbol{\epsilon}_n \sim \mathcal{N}(\boldsymbol{0}, (\sigma C)^2 \boldsymbol{I})\,.$$

DP-SGD performs the update rule of SGD, with $\boldsymbol{g}_n$ replaced by $\hat{\boldsymbol{g}}_n$,

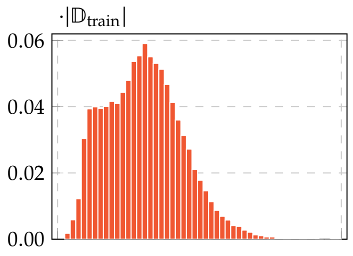
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{B}} \hat{\boldsymbol{g}}_n(\boldsymbol{\theta}_t)\,.$$

This requires access to per-sample gradient norms $\{\|\boldsymbol{g}_n(\boldsymbol{\theta}_t)\|_2\}_n$.

(a) Epoch 0, train accuracy 9.83%



(b) Epoch 2, train accuracy 83.4%



(c) Epoch 49, train accuracy 90.5%

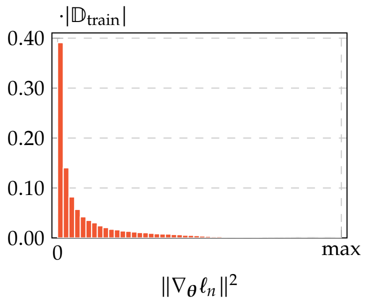


Figure 3.8: Importance distribution of samples during training. Importance is measured by individual gradient L_2 norms, computed with BackPACK (Chapter 5). As training proceeds and more examples are correctly classified and become "unimportant". Details: logistic regression on MNIST, trained with SGD, $|\mathbb{B}| = 128$, and learning rate 0.005.

3.3.4 Importance Sampling

A hypothesis about learning in ML is that the model first learns to correctly predict "easy" examples. Only in later phases are the "difficult" examples learned. Using these harder examples more frequently for training might help speed up the learning procedure. Put differently, some data matter more at certain stages of training than other, which is quantified through a measure of importance. Importance sampling realizes this idea of selecting important data more frequently. It does so by adapting the sampling procedure for mini-batches to reduce the gradient variance, which beneficially influences the convergence of stochastic first-order methods like SGD, and therefore speeds up training. A common strategy is to weight the importance of samples by the per-sample L_2 norm (Equation (3.38) and Figure 3.8).

As a starting point to see how the sampling procedure affects optimization, consider the generalization of uniform sampling from Section 2.1.2 for $|\mathbb{B}| = 1$. At training iteration t, a sample $n_t \sim p_t(n_t)$ is drawn from a current sampling distribution $p_t(n_t)$ over $\{1, \dots, |\mathbb{D}|\}$. SGD with a learning rate η uses the unbiased gradient estimator[7]

$$\hat{\boldsymbol{g}}_{n_t}(\boldsymbol{\theta}_t) := \frac{1}{|\mathbb{D}| p_t(n_t)} \nabla_{\boldsymbol{\theta}_t} \ell_{n_t}(\boldsymbol{\theta}_t) \tag{3.35a}$$

with

$$\mathbb{E}_{n_t \sim p_t(n_t)} \left[\hat{\boldsymbol{g}}_{n_t}(\boldsymbol{\theta}_t) \right] = \frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}} \nabla_{\boldsymbol{\theta}_t} \ell_{n_t}(\boldsymbol{\theta}_t) = \boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}_t) \tag{3.35b}$$

to update the parameters

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \hat{\boldsymbol{g}}_{n_t}(\boldsymbol{\theta}_t)\,. \tag{3.35c}$$

7: For uniform sampling $p_t(n_t) = 1/|\mathbb{D}|$, which is the most common case in practice, the scaling factor cancels and yields the commonly used gradient estimator that would be used by "normal" SGD with $|\mathbb{B}| = 1$.

Using a non-uniform distribution over stochastic gradients would introduce bias in the update, so in importance sampling the samples are weighted by their probability of being selected to undo the bias. The distribution can then be tuned to minimize the variance of the estimator and improve performance: one can assess convergence of Equation (3.35c) through the progress towards the minimizer $\boldsymbol{\theta}_\star$ in terms of squared L_2 distance [85, 169],

$$\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star\|_2^2 - \|\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_\star\|_2^2\,. \tag{3.36a}$$

In expectation, this measure of convergence depends on the gradient estimator's variance (more specifically, its trace),

$$\begin{aligned}
&\mathbb{E}_{n_t}\left[\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star\|_2^2 - \|\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_\star\|_2^2\right] \\
&\quad= \mathbb{E}_{n_t}\left[\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star\|_2^2 - \|\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star - \eta \hat{\boldsymbol{g}}_{n_t}\|_2^2\right] \\
&\quad= \mathbb{E}_{n_t}\left[2\eta \left(\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star\right)^\top \hat{\boldsymbol{g}}_{n_t} - \eta^2 \hat{\boldsymbol{g}}_{n_t}^\top \hat{\boldsymbol{g}}_{n_t}\right] \\
&\quad= 2\eta \left(\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star\right)^\top \boldsymbol{g}_{\mathbb{D}} - \eta^2 \mathbb{E}_{n_t}\left[\|\hat{\boldsymbol{g}}_{n_t}\|_2^2\right] \\
&\quad= 2\eta \left(\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star\right)^\top \boldsymbol{g}_{\mathbb{D}} - \eta^2 \|\boldsymbol{g}_{\mathbb{D}}\|_2^2 - \eta^2 \mathbb{E}_{n_t}\left[\|\hat{\boldsymbol{g}}_{n_t} - \boldsymbol{g}_{\mathbb{D}}\|_2^2\right] \\
&\quad= 2\eta \left(\boldsymbol{\theta}_t - \boldsymbol{\theta}_\star\right)^\top \boldsymbol{g}_{\mathbb{D}} - \eta^2 \|\boldsymbol{g}_{\mathbb{D}}\|_2^2 - \eta^2 \operatorname{Tr}\left\{\operatorname{Var}_{n_t}\left[\hat{\boldsymbol{g}}_{n_t}\right]\right\}
\end{aligned} \tag{3.36b}$$

For what follows, the intuition is that sampling affects convergence through the variance term, and minimizing this term improves convergence. Hence, the goal is to identify the optimal sampling via

$$\underset{\substack{p_t(n_t) \\ \sum_{n=1}^{|\mathbb{D}|} p_t(n) = 1}}{\text{minimize}} \operatorname{Tr}\left\{\operatorname{Var}_{n_t}\left[\hat{\boldsymbol{g}}_{n_t}\right]\right\} \quad \Leftrightarrow \quad \underset{\substack{p_t(n_t) \\ \sum_{n=1}^{|\mathbb{D}|} p_t(n) = 1}}{\text{minimize}} \mathbb{E}_{n_t}\left[\|\hat{\boldsymbol{g}}_{n_t}\|_2^2\right] \tag{3.37}$$

The solution, outlined in Remark 3.5, is

$$p_t(n_t) = \frac{\|\nabla_{\boldsymbol{\theta}_t} \ell_{n_t}(\boldsymbol{\theta}_t)\|_2}{\sum_{n=1}^{|\mathbb{D}|} \|\ell_n(\boldsymbol{\theta}_t)\|_2}\,. \tag{3.38}$$

and depends on individual gradient L_2 norms in the entire dataset. Samples with higher importance, *i.e.* gradient norm, are drawn more frequently. Because computing the sampling distribution requires a sweep over all data, practical versions further approximate Equation (3.38), *e.g.* by relaxing the optimization through bounds that are easier to compute, or updating the distribution only every few iterations [85].

Remark 3.5 (Optimal sampling distribution [116, 184]) (The iteration index t is suppressed for brevity) To derive Equation (3.38), one incorporates the constraint in Equation (3.37) via a Lagrange multiplier $\mu \in \mathbb{R}$, writing $p(n)$ as $|\mathbb{D}|$-dimensional vector $\boldsymbol{p}$. Expanding $\hat{\boldsymbol{g}}$ with Equation (3.35a) leads to,

$$\begin{aligned}
A(\boldsymbol{p}, \mu) := &\frac{1}{|\mathbb{D}|^2} \sum_{n=1}^{|\mathbb{D}|} \frac{\|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2^2}{p_n} \\
&+ \mu \left(\sum_{n=1}^{|\mathbb{D}|} p_n - 1\right).
\end{aligned}$$

Setting the p_n-derivative of that to zero yields

$$\begin{aligned}
\nabla_{p_n} A(\boldsymbol{p}, \mu) &= -\frac{\|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2^2}{|\mathbb{D}|^2 p_n^2} + \mu \overset{!}{=} 0 \\
\implies \quad p_n &= \frac{\|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2}{\sqrt{\mu}}\,,
\end{aligned}$$

using that all elements $p_n \in (0;1)$ of $\boldsymbol{p}$. Inserting into A gives

$$A(\mu) = \frac{2}{|\mathbb{D}|} \sqrt{\mu} \sum_{n=1}^{|\mathbb{D}|} \|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2 - \mu$$

Setting the μ-derivative to zero yields

$$\begin{aligned}
\nabla_\mu A(\mu) &= \frac{1}{|\mathbb{D}| \sqrt{\mu}} \sum_{n=1}^{|\mathbb{D}|} \|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2 - 1 \overset{!}{=} 0 \\
\implies \quad \sqrt{\mu} &= \frac{1}{|\mathbb{D}|} \sum_{n=1}^{|\mathbb{D}|} \|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2
\end{aligned}$$

This then leads to

$$p_n = \frac{\|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2}{\sum_{n=1}^{|\mathbb{D}|} \|\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})\|_2}$$

which equals Equation (3.38) when switching back notation and introducing the iteration count.

Part II.

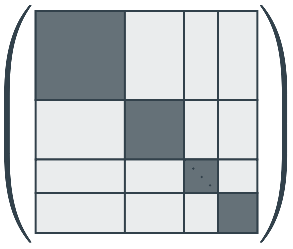
Backpropagation Beyond the Gradient

Modular Block-diagonal Curvature Approximations for Feedforward Architectures 4.

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.

Code and experiments available at the Github repository `f-dangel/hbp`



4.1 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 13]. 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 $\mathcal{L}$ by a quadratic $\mathcal{L}(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})^\top(\boldsymbol{\theta}_\star - \boldsymbol{\theta}) + {}^1\!/_2(\boldsymbol{\theta}_\star - \boldsymbol{\theta})^\top \boldsymbol{C}(\boldsymbol{\theta})(\boldsymbol{\theta}_\star - \boldsymbol{\theta})$ around the current location $\boldsymbol{\theta}$, using the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) = {}^{\partial\mathcal{L}(\boldsymbol{\theta})}\!/_{\partial\boldsymbol{\theta}}$ and a positive semi-definite (PSD) curvature matrix $\boldsymbol{C}(\boldsymbol{\theta})$—the Hessian of $\mathcal{L}$ or approximations thereof. The quadratic is minimized by

$$\boldsymbol{\theta}_\star = \boldsymbol{\theta} + \Delta\boldsymbol{\theta} \quad \text{with} \quad \Delta\boldsymbol{\theta} = -\boldsymbol{C}(\boldsymbol{\theta})^{-1}\nabla_{\boldsymbol{\theta}}\mathcal{L}\,. \tag{4.1}$$

Computing the update step requires solving the linear system $\boldsymbol{C}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} = -\nabla_{\boldsymbol{\theta}}\mathcal{L}$. To accomplish this task, providing a matrix-vector multiplication with the curvature matrix $\boldsymbol{C}(\boldsymbol{\theta})$ 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 [127, 148]. This *matrix-free* formulation can then be leveraged to solve Equation (4.1) using iterative solvers such as conjugate gradients (CG) [106]. 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 [183], which only considers the content of Hessian sub-blocks along the diagonal. Reducing the Hessian to a block

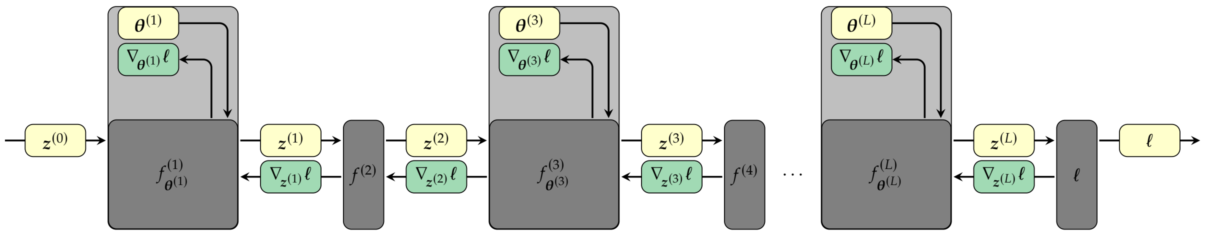


Figure 4.1: Standard sequential feedforward network architecture. *I.e.* the repetition of affine transformations parameterized by $\boldsymbol{\theta}^{(l)} = ((\operatorname{vec} \boldsymbol{W}^{(l)})^\top, \boldsymbol{b}^{(l)\top})^\top$ followed by element-wise activations. Arrows from left to right and vice versa indicate the data flow during forward pass and gradient backpropagation, respectively.

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 [110]. Storing these constituents efficiently often requires an involved manual analysis of the Hessian's structure, leveraging its outer-product form in many scenarios [8, 115]. Recent works developed different block-diagonal approximations (BDA) of curvature matrices that provide fast multiplication [21, 31, 63, 109].

These works have repeatedly shown that, empirically, second-order information can improve the training of 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 ML 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 chapter 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 ML 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 [103].
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 multiplication by curvature matrices using automatic differentiation [127, 148]. However, we also address a new class of curvature matrices, the positive-curvature Hessian (PCH) introduced in [31]. Our solutions to the latter two points are generalizations of previous works [21, 31] 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.

4.2 Notation

We consider feedforward neural networks $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ composed of L modules $f^{(l)}_{\boldsymbol{\theta}^{(l)}}, l = 1, \dots, L$, which can be represented as a computational graph mapping the input $\boldsymbol{z}^{(0)} = \boldsymbol{x}$ to the output $\boldsymbol{z}^{(L)}$ (Figure 4.1). A module $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ receives the parental output $\boldsymbol{z}^{(l-1)}$, applies an operation involving the module parameters $\boldsymbol{\theta}^{(l)}$, and sends the output $\boldsymbol{z}^{(l)}$ to its child. Thus, $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ is of the form $\boldsymbol{z}^{(l)} = f^{(l)}_{\boldsymbol{\theta}^{(l)}}(\boldsymbol{z}^{(l-1)})$. Typical choices include element-wise nonlinear activation without any parameters and affine transformations $\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{z}^{(l-1)} + \boldsymbol{b}^{(l)}$ with parameters given by the weights $\boldsymbol{W}^{(l)}$ and the bias $\boldsymbol{b}^{(l)}$. 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 $\boldsymbol{z}^{(L)}(\boldsymbol{x}, \boldsymbol{\theta}^{(1,\dots,L)}) = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ of a datum $\boldsymbol{x}$ with label $\boldsymbol{y}$, the goal is to minimize the expected risk of the loss function $\ell(\boldsymbol{z}^{(L)}, \boldsymbol{y})$. Under the framework of empirical risk minimization, the parameters are tuned to optimize the loss on the training set $\mathbb{D}_{\text{train}} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_{n=1}^N$,

$$\min_{\boldsymbol{\theta}^{(1,\dots,L)}} \mathcal{L}_{\mathbb{D}_{\text{train}}}(\boldsymbol{\theta}^{(1,\dots,L)}) = \min_{\boldsymbol{\theta}^{(1,\dots,L)}} \frac{1}{|\mathbb{D}_{\text{train}}|} \sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)\,. \tag{4.2}$$

In practice, the objective is typically further approximated stochastically by drawing a mini-batch $\mathbb{B} \subset \mathbb{D}_{\text{train}}$ from the training set. We will treat both scenarios without further distinction, since the structure relevant to our purposes is that Equation (4.2) is an average of terms depending on individual data points. Quantities for optimization, be it gradients or second derivatives of the loss *w.r.t.* the network parameters, can be processed in parallel, then averaged.

4.3 Modular Hessian Backpropagation

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 net of Figure 4.1, depicted in Figure 4.2, in this section[1]. The forward pass $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ maps the input $\boldsymbol{x}$ to the output $\boldsymbol{z}$ by means of the module parameters $\boldsymbol{\theta}$. All quantities are

1: To simplify the presentation, we drop layer indices, choose distinct variable names for input and output ($\boldsymbol{x} \leftarrow \boldsymbol{z}^{(l-1)}$, $\boldsymbol{z} \leftarrow \boldsymbol{z}^{(l)}$, $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^{(l)}$, $f_{\boldsymbol{\theta}} \leftarrow f^{(l)}_{\boldsymbol{\theta}^{(l)}}$) and focus on a single datum.

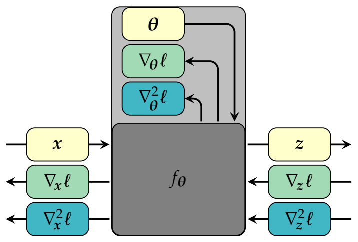


Figure 4.2: Forward pass, gradient backpropagation, and Hessian backpropagation for a single module. Arrows from left to right indicate the data flow in the forward pass $z = f_\theta(x)$, while the opposite orientation indicates the gradient backpropagation by Equation (4.4). We suggest to extend this by the backpropagation of the Hessian as indicated by Equation (4.7).

assumed to be vector-shaped (tensor-valued quantities can be flattened, see Definition 2.2). Optimization requires the gradient of the loss function *w.r.t.* the parameters, $\partial\ell(\boldsymbol{\theta})/\partial\boldsymbol{\theta} = \nabla_{\boldsymbol{\theta}}\ell$. We will use the shorthand

$$\nabla_{\cdot}\ell = \frac{\partial \ell(\cdot)}{\partial \operatorname{vec}(\cdot)}\,. \tag{4.3}$$

During gradient backpropagation the module receives the loss gradient with respect to its output, $\nabla_z\ell$, from its child. The backward operation computes gradients *w.r.t.* the module parameters and input, $\nabla_{\boldsymbol{\theta}}\ell$ and $\nabla_{\boldsymbol{x}}\ell$ from $\nabla_z\ell$. Backpropagation continues by sending the gradient *w.r.t.* the module's input to its parent, which proceeds in the same way (see Figure 4.1). By the chain rule, gradients *w.r.t.* an element x_i of the module's input can be computed as $\nabla_{x_i}\ell = \sum_j (\partial z_j/\partial x_i)\nabla_{z_j}\ell$. The vectorized version is compactly written in terms of the Jacobian matrix $\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z} = \partial z/\partial x^\top$ (Definition 2.4), which contains all partial derivatives of $\boldsymbol{z}$ *w.r.t.* $\boldsymbol{x}$. The arrangement of partial derivatives is $[\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z}]_{j,i} = \partial z_j/\partial x_i$, such that

$$\nabla_{\boldsymbol{x}}\ell = (\mathrm{J}_{\boldsymbol{x}}\boldsymbol{z})^\top \nabla_z \ell\,. \tag{4.4}$$

Analogously, the parameter gradients are given by $\nabla_{\theta_i}\ell = \sum_j (\partial z_j/\partial\theta_i)\nabla_{z_j}\ell$, or in vectorized form $\nabla_{\boldsymbol{\theta}}\ell = (\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z})^\top \nabla_z\ell$. This reflects the symmetry of both $\boldsymbol{x}$ and $\boldsymbol{\theta}$ 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 ℓ *w.r.t.* elements of $\boldsymbol{x}$ or $\boldsymbol{\theta}$,

$$\begin{aligned}\frac{\partial^2\ell}{\partial x_i \partial x_j} &= \frac{\partial}{\partial x_j}\left(\sum_k \frac{\partial z_k}{\partial x_i}\nabla_{z_k}\ell\right)\\ &= \sum_{k,l}\frac{\partial z_k}{\partial x_i}\frac{\partial^2\ell}{\partial z_k \partial z_l}\frac{\partial z_l}{\partial x_j} + \sum_k \frac{\partial^2 z_k}{\partial x_i \partial x_j}\nabla_{z_k}\ell\,,\end{aligned} \tag{4.5}$$

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 (4.5) propagates curvature information of the output further back, while the second term introduces second-order effects of the module itself. Using the Hessian matrix $\nabla^2_{\boldsymbol{x}}\ell = \partial^2\ell/(\partial x^\top \partial x)$ of a scalar function *w.r.t.* a vector-shaped quantity $\boldsymbol{x}$ (Definition 3.1),

$$\nabla^2_{\cdot}\ell(\cdot) = \frac{\partial^2\ell(\cdot)}{\partial \operatorname{vec}(\cdot)^\top \partial \operatorname{vec}(\cdot)}\,, \tag{4.6}$$

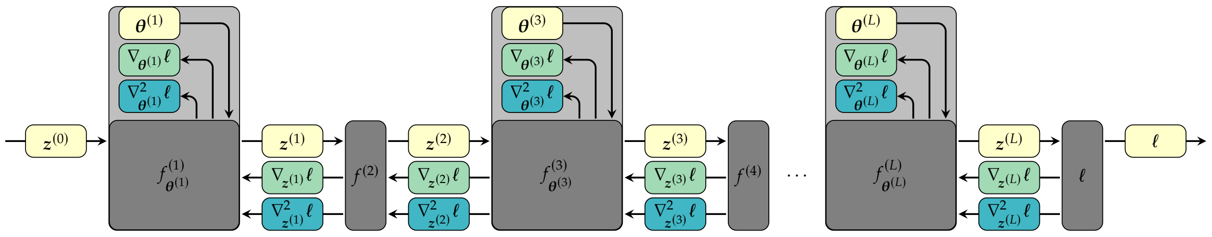
results in the matrix version of Equation (4.5),

$$\nabla^2_{\boldsymbol{x}}\ell = (\mathrm{J}_{\boldsymbol{x}} z)^\top \nabla^2_z \ell\ (\mathrm{J}_{\boldsymbol{x}} z) + \sum_k \left(\nabla^2_{\boldsymbol{x}} z_k\right) \nabla_{z_k}\ell\,. \tag{4.7a}$$

Note that the second-order effect introduced by the module itself via $\nabla^2_{\boldsymbol{x}} z_k$ vanishes if all components $[f_{\boldsymbol{\theta}}(\boldsymbol{x})]_k$ are linear in $\boldsymbol{x}$. Because the layer parameters $\boldsymbol{\theta}$ can be regarded as inputs to the layer, they are treated in exactly the same way, replacing $\boldsymbol{x}$ by $\boldsymbol{\theta}$ in the above expression,

$$\nabla^2_{\boldsymbol{\theta}}\ell = (\mathrm{J}_{\boldsymbol{\theta}} z)^\top \nabla^2_z \ell\ (\mathrm{J}_{\boldsymbol{\theta}} z) + \sum_k \left(\nabla^2_{\boldsymbol{\theta}} z_k\right) \nabla_{z_k}\ell\,. \tag{4.7b}$$

Equation (4.7) is the central functional expression herein, and will be referred to as the *Hessian backpropagation (HBP) equation*. Our suggested



$\boldsymbol{\theta}^{(1)}$
$\nabla_{\boldsymbol{\theta}^{(1)}}\ell$
$\nabla^2_{\boldsymbol{\theta}^{(1)}}\ell$
$z^{(0)}$
$f^{(1)}_{\boldsymbol{\theta}^{(1)}}$
$z^{(1)}$
$\nabla_{z^{(1)}}\ell$
$\nabla^2_{z^{(1)}}\ell$
$f^{(2)}$
$z^{(2)}$
$\nabla_{z^{(2)}}\ell$
$\nabla^2_{z^{(2)}}\ell$
$\boldsymbol{\theta}^{(3)}$
$\nabla_{\boldsymbol{\theta}^{(3)}}\ell$
$\nabla^2_{\boldsymbol{\theta}^{(3)}}\ell$
$f^{(3)}_{\boldsymbol{\theta}^{(3)}}$
$z^{(3)}$
$\nabla_{z^{(3)}}\ell$
$\nabla^2_{z^{(3)}}\ell$
$f^{(4)}$
...
$\boldsymbol{\theta}^{(L)}$
$\nabla_{\boldsymbol{\theta}^{(L)}}\ell$
$\nabla^2_{\boldsymbol{\theta}^{(L)}}\ell$
$f^{(L)}_{\boldsymbol{\theta}^{(L)}}$
$z^{(L)}$
$\nabla_{z^{(L)}}\ell$
$\nabla^2_{z^{(L)}}\ell$
ℓ
ℓ

Figure 4.3: Extension of backpropagation to Hessians. It yields diagonal blocks of the full parameter Hessian.

extension of gradient backpropagation is to also send the Hessian $\nabla^2_{z}\ell$ back through the graph. To do so, existing modules have to be extended by the HBP equation: *Given the Hessian $\nabla^2_{z}\ell$ of the loss w.r.t. all module outputs, an extended module has to extract the Hessians $\nabla^2_{\boldsymbol{\theta}}\ell, \nabla^2_{x}\ell$ by means of Equation (4.7), and forward the Hessian w.r.t. its input $\nabla^2_{x}\ell$ to the parent module which proceeds likewise.* In this way, backprop of gradients can be extended to compute curvature information in modules. This corresponds to BDAs of the Hessian which ignore second-order derivatives *w.r.t.* parameters in different modules. Figure 4.3 shows the data flow. The computations required in Equation (4.7) depend only on *local quantities* that are, mostly, already being computed during gradient backpropagation[2].

2: By Faà di Bruno's formula [82] 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.

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 Section 4.3.1).
- Instead of evaluating all matrices during backpropagation, we can define matrix-vector products recursively. This yields exact curvature matrix multiplications by the block diagonals of the Hessian, the generalized Gauss-Newton (GGN) matrix and the PCH. Multiplications by the first two matrices can also be obtained by use of automatic differentiation [127, 148]. We also get access to the PCH which, in contrast to the GGN, considers curvature information introduced by the network (see Section 4.3.1).[3] For standard neural networks, only second derivatives of nonlinear activations have to be stored compared to gradient backpropagation.
- There are approaches [21, 31] that propagate matrix representations back through the graph to save repeated computations in the curvature matrix-vector product. The size of the matrices $\nabla^2_{z^{(l)}}\ell$ passed between layer $l+1$ and l scales quadratically in the number of layer l's output features. For convolutional layers, the dimension of these quantities exceeds computational budgets. And for batched computations, such a matrix has to be backpropagated for every datum in the mini-batch. Like previous schemes [21, 31], we introduce additional approximations for batch learning in Section 4.3.2. A connection to existing schemes is drawn in Appendix A.2.4.

3: Implementations of HBP for exact matrix-vector products can reuse multiplication by the (transposed) Jacobian provided by many ML libraries. The second term Equation (4.7) needs special treatment though.

HBP is easy to integrate into current ML libraries, so that curvature BDAs can be provided automatically for novel or existing second-order optimization methods[4]. Such methods have repeatedly been shown to be competitive with first-order methods [21, 31, 63, 109, 183].

4: *E.g.* the matrix-vector products with block-diagonal curvature matrices described in this work have been integrated into the BackPACK library that operates on top of PyTorch and will be presented in Chapter 5 of this thesis.

Table 4.1: Hessian backpropagation for common modules used in feedforward networks. $\boldsymbol{I}$ denotes the identity matrix. We assign matrices to upper-case ($\boldsymbol{W}, \boldsymbol{X}, \dots$) and tensors to upper-case sans serif symbols ($\mathsf{W}, \mathsf{X}, \dots$).

OPERATION	FORWARD	HBP (Equation (4.7))	DETAILS
Matrix-vector multiplication	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{x}$	$\nabla^2_{\boldsymbol{x}}\ell = \boldsymbol{W}^\top(\nabla^2_{\boldsymbol{z}}\ell)\boldsymbol{W}$, $\nabla^2_{\boldsymbol{W}}\ell = \boldsymbol{x} \otimes \boldsymbol{x}^\top \otimes \nabla^2_{\boldsymbol{z}}\ell$	Appendix A.2.1
Matrix-matrix multiplication	$\boldsymbol{Z}(\boldsymbol{X}, \boldsymbol{W}) = \boldsymbol{W}\boldsymbol{X}$	$\nabla^2_{\boldsymbol{X}}\ell = (\boldsymbol{I} \otimes \boldsymbol{W})^\top \nabla^2_{\boldsymbol{Z}}\ell(\boldsymbol{I} \otimes \boldsymbol{W})$, $\nabla^2_{\boldsymbol{W}}\ell = (\boldsymbol{X}^\top \otimes \boldsymbol{I})^\top \nabla^2_{\boldsymbol{Z}}\ell(\boldsymbol{X}^\top \otimes \boldsymbol{I})$	Appendix A.2.1
Addition	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{b}) = \boldsymbol{x} + \boldsymbol{b}$	$\nabla^2_{\boldsymbol{x}}\ell = \nabla^2_{\boldsymbol{b}}\ell = \nabla^2_{\boldsymbol{z}}\ell$	Appendix A.2.1
Elementwise activation	$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})$, s.t., $z_i(\boldsymbol{x}) = \phi(x_i)$	$\nabla^2_{\boldsymbol{x}}\ell = \operatorname{diag}[\boldsymbol{\phi}'(\boldsymbol{x})](\nabla^2_{\boldsymbol{z}}\ell)\operatorname{diag}[\boldsymbol{\phi}'(\boldsymbol{x})] + \operatorname{diag}[\phi''(\boldsymbol{x}) \odot \nabla_{\boldsymbol{z}}\ell]$	Appendix A.2.2
Skip-connection	$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{x} + \boldsymbol{s}(\boldsymbol{x}, \boldsymbol{\theta})$	$\nabla^2_{\boldsymbol{x}}\ell = (\boldsymbol{I} + \mathrm{J}_{\boldsymbol{x}}\boldsymbol{s})^\top \nabla^2_{\boldsymbol{z}}\ell(\boldsymbol{I} + \mathrm{J}_{\boldsymbol{x}}\boldsymbol{s}) + \sum_k(\nabla^2_{\boldsymbol{x}}s_k)\nabla_{z_k}\ell$, $\nabla^2_{\boldsymbol{\theta}}\ell = (\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{s})^\top \nabla^2_{\boldsymbol{z}}\ell(\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{s}) + \sum_k(\nabla^2_{\boldsymbol{\theta}}s_k)\nabla_{z_k}\ell$	Appendix A.2.3
Reshape/view	$\mathsf{Z}(\mathsf{X}) = \operatorname{reshape}(\mathsf{X})$	$\nabla^2_{\mathsf{Z}}\ell = \nabla^2_{\mathsf{X}}\ell$	Appendix A.4.1
Index select/map π	$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{\Pi}\boldsymbol{x}$, $\Pi_{j,\pi(j)} = 1$,	$\nabla^2_{\boldsymbol{x}}\ell = \boldsymbol{\Pi}^\top(\nabla^2_{\boldsymbol{z}}\ell)\boldsymbol{\Pi}$	Appendix A.4.2
Convolution	$\mathsf{Z}(\mathsf{X}, \mathsf{W}) = \mathsf{X} \star \mathsf{W}$, $\boldsymbol{Z}(\boldsymbol{W}, [\![\mathsf{X}]\!]) = \boldsymbol{W}[\![\mathsf{X}]\!]$,	$\nabla^2_{[\![\mathsf{X}]\!]}\ell = (\boldsymbol{I} \otimes \boldsymbol{W})^\top \nabla^2_{\boldsymbol{Z}}\ell(\boldsymbol{I} \otimes \boldsymbol{W})$, $\nabla^2_{\boldsymbol{W}}\ell = ([\![\mathsf{X}]\!]^\top \otimes \boldsymbol{I})^\top \nabla^2_{\boldsymbol{Z}}\ell([\![\mathsf{X}]\!]^\top \otimes \boldsymbol{I})$	Appendix A.4.3
Square loss	$\ell(\boldsymbol{f}, \boldsymbol{y}) = {}^1\!/\!_C(\boldsymbol{y} - \boldsymbol{f})^\top(\boldsymbol{y} - \boldsymbol{f})$	$\nabla^2_{\boldsymbol{f}}\ell = {}^2\!/\!_C\boldsymbol{I}$	Appendix A.3.1
Softmax cross-entropy	$\ell(\boldsymbol{f}, y) = -\operatorname{onehot}(y)^\top \log[\boldsymbol{p}(\boldsymbol{f})]$	$\nabla^2_{\boldsymbol{f}}\ell = \operatorname{diag}[\boldsymbol{p}(\boldsymbol{f})] - \boldsymbol{p}(\boldsymbol{f})\boldsymbol{p}(\boldsymbol{f})^\top$	Appendix A.3.2

Relationship to Matrix Differential Calculus

To some extent, this paper is a re-formulation of earlier results [21, 31, 109] in the framework of matrix differential calculus [103], 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 (4.7) is a special case of the matrix chain rule of that framework (Theorem 2.1 and Equation (3.14)). A more detailed discussion of this connection can be found in Appendix A.1 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 4.2. Like for gradient backprop, the operations required for HBP can be tabulated. Table 4.1 provides a selection of common modules. The derivations, which again leverage the matrix differential calculus framework, can be found in Appendices A.2 to A.4.

4.3.1 Obtaining Different Curvature Matrices

The HBP equation yields *exact* diagonal blocks $\nabla^2_{\boldsymbol{\theta}^{(1)}}\ell, \dots, \nabla^2_{\boldsymbol{\theta}^{(L)}}\ell$ of the full parameter Hessian $\nabla^2_{\boldsymbol{\theta}}\ell$. 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 equation 4.1, as they need neither be PSD nor invertible. For application in optimization, HBP can be modified to yield semi-definite BDAs of the Hessian. Equation (4.7) again provides the foundation for this adaptation, which is closely related to the concepts of KFRA [21], BDA-PCH [31], and, under certain conditions, KFAC [109]. 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 $\ell(\boldsymbol{z}^{(L)}, \boldsymbol{y})$ in the network output $\boldsymbol{z}^{(L)}$ (Section 3.2.3). It is also obtained by linearizing the network output $\boldsymbol{z}^{(L)}(\boldsymbol{\theta}, \boldsymbol{x})$ in $\boldsymbol{\theta}$ before computing the loss Hessian [107], and reads (Equation (3.15b))

$$\boldsymbol{G}(\boldsymbol{\theta}) = \left(\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{z}^{(L)}\right)^{\top} \nabla^2_{\boldsymbol{z}^{(L)}} \ell \left(\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{z}^{(L)}\right) .$$

For diagonal blocks $\boldsymbol{G}(\boldsymbol{\theta}^{(l)})$, the Jacobian is unrolled using its chain rule (Theorem 2.2) $\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(L)} = (\mathrm{J}_{\boldsymbol{z}^{(L-1)}} \boldsymbol{z}^{(L)})(\mathrm{J}_{\boldsymbol{z}^{(L-2)}} \boldsymbol{z}^{(L-1)}) \cdots (\mathrm{J}_{\boldsymbol{z}^{(l)}} \boldsymbol{z}^{(l+1)})(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)})$. This shows that the Hessian $\nabla^2_{\boldsymbol{z}^{(L)}} \ell$ of the loss function *w.r.t.* 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 (4.7)[5]. 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 $\nabla^2_{\boldsymbol{z}^{(L)}} \ell$ 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 4.1 are PSD. The resulting recursive scheme has been used by Botev et al. [21] under the acronym KFRA to optimize convex loss functions of fully-connected neural networks with piecewise linear activation functions.

5: Recall from Section 2.2 that many common neural network operations, such as affine transformations, convolutions, padding & pooling, have vanishing second-order derivatives *w.r.t.* both their input and parameters. Hence, the second term in Equation (4.7) vanishes exactly for these operations.

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. [31] and named the PCH. It is obtained by modification of terms in the second summand of Equation (4.7) that can introduce concavity during HBP. This ensures positive semi-definiteness since the first summand is semi-definite, assuming the loss Hessian $\nabla^2_{\boldsymbol{z}^{(L)}} \ell$ with respect to the network output to be positive semi-definite. Chen et al. [31] 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. [31], the PCH can empirically outperform optimization using the GGN. In usual feedforward neural networks, the concavity is introduced by nonlinear element-wise activations, and corresponds to a diagonal matrix (Table 4.1). 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 output models a conditional probability density on the labels, $q(\mathbf{y} \mid \boldsymbol{z}^{(L)})$, maximum likelihood learning for the parameterized density $p_{\boldsymbol{\theta}}(\mathbf{y} \mid \boldsymbol{x}) = q(\mathbf{y} \mid \boldsymbol{z}^{(L)}(\boldsymbol{x}, \boldsymbol{\theta}))$ will correspond to choosing a negative log-likelihood loss function, *i.e.* $\ell(\boldsymbol{z}^{(L)}, \boldsymbol{y}) =$

$-\log q(\boldsymbol{y} \mid \boldsymbol{z}^{(L)})$ (Section 2.1.3). Common loss functions like square and cross-entropy loss can be interpreted in this way (Examples 2.3 and 2.4), . Natural gradient descent [5] uses the Fisher information matrix $\boldsymbol{F}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{y}\sim p_{\boldsymbol{\theta}}(\mathrm{y} \mid \boldsymbol{x})}\left[(\partial \log p_{\theta}(\boldsymbol{y} \mid \boldsymbol{x})/\partial\boldsymbol{\theta})\,(\partial \log p_{\theta}(\boldsymbol{y} \mid \boldsymbol{x})/\partial\boldsymbol{\theta})^{\top}\right]$ as a PSD curvature matrix approximating the Hessian. It can be expressed as the log-predictive density's expected Hessian under $p_{\boldsymbol{\theta}}$ itself: $\boldsymbol{F}(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{y}\sim p_{\boldsymbol{\theta}}(\mathrm{y} \mid \boldsymbol{z}^{(L)})}\left[\nabla^2_{\boldsymbol{\theta}} \log q(\boldsymbol{y} \mid \boldsymbol{z}^{(L)})\right]$, see Equation (3.24). Assuming truly *i.i.d.* samples $\boldsymbol{x}$, the log-likelihood of multiple data decomposes and, after using the chain rule, results in the approximation

$$\boldsymbol{F}_{\mathbb{D}_{\text{train}}}(\boldsymbol{\theta}) \approx \frac{1}{|\mathbb{D}_{\text{train}}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathbb{D}_{\text{train}}} \left(\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z}^{(L)}\right)^{\top} \boldsymbol{F}_q(\boldsymbol{z}^{(L)}) \left(\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z}^{(L)}\right)$$

with $\boldsymbol{F}_q(\boldsymbol{z}^{(L)}) = -\mathbb{E}_{\boldsymbol{y}\sim q(\mathrm{y} \mid \boldsymbol{z}^{(L)})}[\nabla^2_{\boldsymbol{z}^{(L)}} \log q(\boldsymbol{y} \mid \boldsymbol{z}^{(L)})]$, see Equation (3.27). In this form, the computational scheme for Fisher BDAs resembles the HBP of the GGN. However, instead of propagating back the loss Hessian *w.r.t.* the network, it uses the negative log-likelihood's expected Hessian the model's predictive distribution. Martens and Grosse [109] use Monte Carlo sampling to estimate this matrix in their KFAC optimizer. Relations between the Fisher and GGN are discussed in [107, 125]; for square and cross-entropy loss, they are equivalent (Section 3.2.4).

4.3.2 Batch Learning Approximations

In our HBP framework, exact multiplication by the curvature block of a module's parameter $\boldsymbol{\theta}$ requires one backpropagation to this layer. The multiplication is recursively defined in terms of multiplication by the layer output Hessian $\nabla^2_{\boldsymbol{z}}\ell$. If it were possible to have an explicit representation of this matrix in memory, the recursive computations hidden in $\nabla^2_{\boldsymbol{z}}\ell$ could be saved during the solution of the linear system implied by Equation (4.1). Unfortunately, the size of the backpropagated exact matrices scales quadratically in both the batch size[6] 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. [21]). In combination with structural information about the parameter Hessian, this strategy is used in Botev et al. [21] and Chen et al. [31] to further approximate curvature multiplications, using quantities computed in a 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 $\mathbb{B}$ by a bar, for instance $1/|\mathbb{B}| \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathbb{B}} \nabla^2_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y}) = \overline{\nabla^2_{\boldsymbol{\theta}}\ell}$. The modified backward pass of curvature information during HBP for a module receives a batch average of the Hessian *w.r.t.* the output, $\overline{\nabla^2_{\boldsymbol{z}}\ell}$, which is used to formulate the matrix-vector product with the batch-averaged parameter Hessian $\overline{\nabla^2_{\boldsymbol{\theta}}\ell}$. An average of the Hessian *w.r.t.* the module input, $\overline{\nabla^2_{\boldsymbol{x}}\ell}$, is passed back. Existing work [21, 31] differs primarily in the specifics of how this batch average is computed. In HBP, these approximations can be formulated compactly within Equation (4.7). Relations to the cited works are discussed in more detail in Appendix A.2.4. The approximations

6: 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.

amounting to relations used by Botev et al. [21] read

$$\overline{\nabla_{\boldsymbol{x}}^2 \ell} \approx \overline{(\mathrm{J}_{\boldsymbol{x}} \boldsymbol{z})^\top \, \overline{\nabla_{\boldsymbol{z}}^2 \ell} \, (\mathrm{J}_{\boldsymbol{x}} \boldsymbol{z})} + \sum_k \overline{\left(\nabla_{\boldsymbol{x}}^2 z_k\right) \nabla_{z_k} \ell} \,, \tag{4.8}$$

and likewise for $\boldsymbol{\theta}$. In case of a linear layer $\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, this approximation implies the relations $\overline{\nabla_{\boldsymbol{W}}^2 \ell} = \overline{\boldsymbol{x} \otimes \boldsymbol{x}^\top} \otimes \overline{\nabla_{\boldsymbol{z}}^2 \ell}$, $\overline{\nabla_{\boldsymbol{b}}^2 \ell} = \overline{\nabla_{\boldsymbol{z}}^2 \ell}$, and $\overline{\nabla_{\boldsymbol{x}}^2 \ell} = \boldsymbol{W}^\top (\overline{\nabla_{\boldsymbol{z}}^2 \ell}) \boldsymbol{W}$. Multiplication by this weight Hessian approximation with a vector $\boldsymbol{v}$ is achieved by storing $\overline{\boldsymbol{x} \otimes \boldsymbol{x}^\top}$, $\overline{\nabla_{\boldsymbol{z}}^2 \ell}$ and performing the required contractions $\boldsymbol{v} \mapsto (\overline{\boldsymbol{x} \otimes \boldsymbol{x}^\top} \otimes \overline{\nabla_{\boldsymbol{z}}^2 \ell})\boldsymbol{v}$. 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. [31],

$$\overline{\nabla_{\boldsymbol{x}}^2 \ell} \approx \overline{(\mathrm{J}_{\boldsymbol{x}} \boldsymbol{z})}^\top \, \overline{\nabla_{\boldsymbol{z}}^2 \ell} \; \overline{(\mathrm{J}_{\boldsymbol{x}} \boldsymbol{z})} + \sum_k \overline{\left(\nabla_{\boldsymbol{x}}^2 z_k\right) \nabla_{z_k} \ell} \,, \tag{4.9}$$

leads to the modified relation $\overline{\nabla_{\boldsymbol{W}}^2 \ell} = \overline{\boldsymbol{x}} \otimes \overline{\boldsymbol{x}}^\top \otimes \overline{\nabla_{\boldsymbol{z}}^2 \ell}$ for a linear layer. As this approximation is of the same rank as $\overline{\nabla_{\boldsymbol{z}}^2 \ell}$, 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 $\overline{\boldsymbol{x}}$ be stored instead of $\overline{\boldsymbol{x} \otimes \boldsymbol{x}^\top}$.

Transformations that are linear in the module parameters (*e.g.* linear and convolutional layers), possess constant Jacobians *w.r.t.* the module input for each sample (see Table 4.1). Hence, in a network consisting of only these layers, both Equations (4.8) and (4.9) yield the same backpropagated Hessians $\overline{\nabla_{\boldsymbol{x}}^2 \ell}$. This still leaves the degree of freedom for choosing the approximation scheme in the analogous equations for $\boldsymbol{\theta}$.

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 and implementation[7].

7: The BackPACK library described in Chapter 5 uses the insights of this section to implement block-diagonal curvature approximations as extensions of gradient backpropagation on the modular level.

4.4 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 [90]. Following the guidelines of Schneider et al. [146], 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.

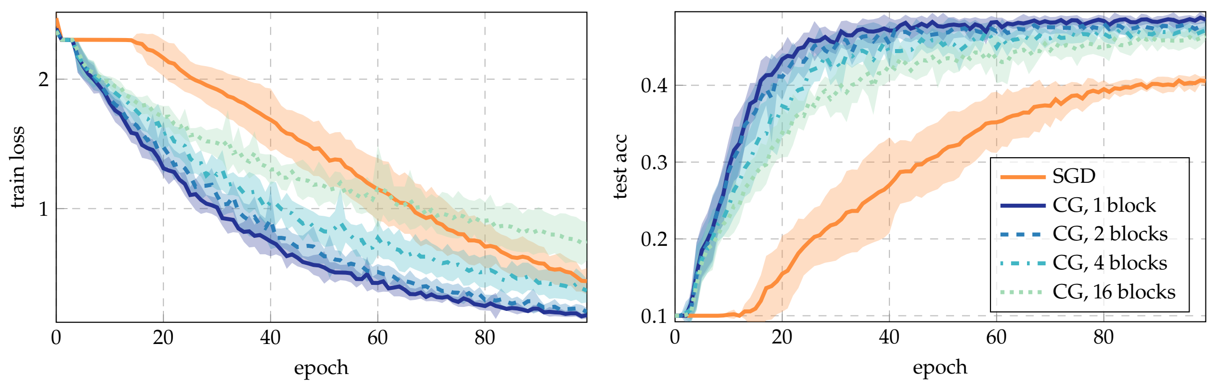


Figure 4.4: SGD and different Newton-style optimizers based on the PCH-abs with batch approximations. The same fully-connected neural network of [31] was used to generate the solid baseline results. Our modular approach allows further splitting the parameter blocks into sub-blocks that can independently be optimized in parallel (dashed lines).

For the loss function we use softmax cross-entropy. Details on the model architectures and hyperparameters are given in Appendix A.5.

Training Procedure & 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 $\boldsymbol{C}^{(l)}$ for each parameter block $\boldsymbol{\theta}^{(l)}$ of the network. Parameter updates $\Delta\boldsymbol{\theta}^{(l)}$ are obtained by applying CG to solve the linear system[8]

8: We use the same update rule as Chen et al. [31] since we extend some of the results shown within this work.

$$\left[\alpha \boldsymbol{I} + (1-\alpha)\boldsymbol{C}^{(l)}\right] \Delta\boldsymbol{\theta}^{(l)} = -\nabla_{\boldsymbol{\theta}^{(l)}}\mathcal{L}\,, \tag{4.10}$$

where α acts as a step size limitation to improve robustness against noise. The CG routine terminates if the ratio of residual and 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 η, and parameters are updated by the relation $\boldsymbol{\theta}^{(l)} \leftarrow \boldsymbol{\theta}^{(l)} + \eta\Delta\boldsymbol{\theta}^{(l)}$.

FCNN, Batch Approximations & Sub-Blocking

The flexibility of HBP is illustrated by extending the results in Chen et al. [31]. Investigations are performed on a fully-connected network with sigmoid activations. Solid lines in Figure 4.4 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.

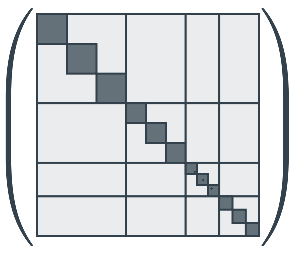


Figure 4.6: Illustration of sub-blocking. Here, each diagonal block is split into three blocks of equal size.

The modularity of HBP allows for additional parallelism by splitting the linear system Equation (4.10) into smaller sub-blocks (Figure 4.6, which then also need fewer iterations of CG. Doing so only requires a minor modification of the parameter Hessian computation by Equation (4.7). Consequently, we split weights and bias terms row-wise into a specified number of sub-blocks. Figure 4.4 shows performance curves. 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

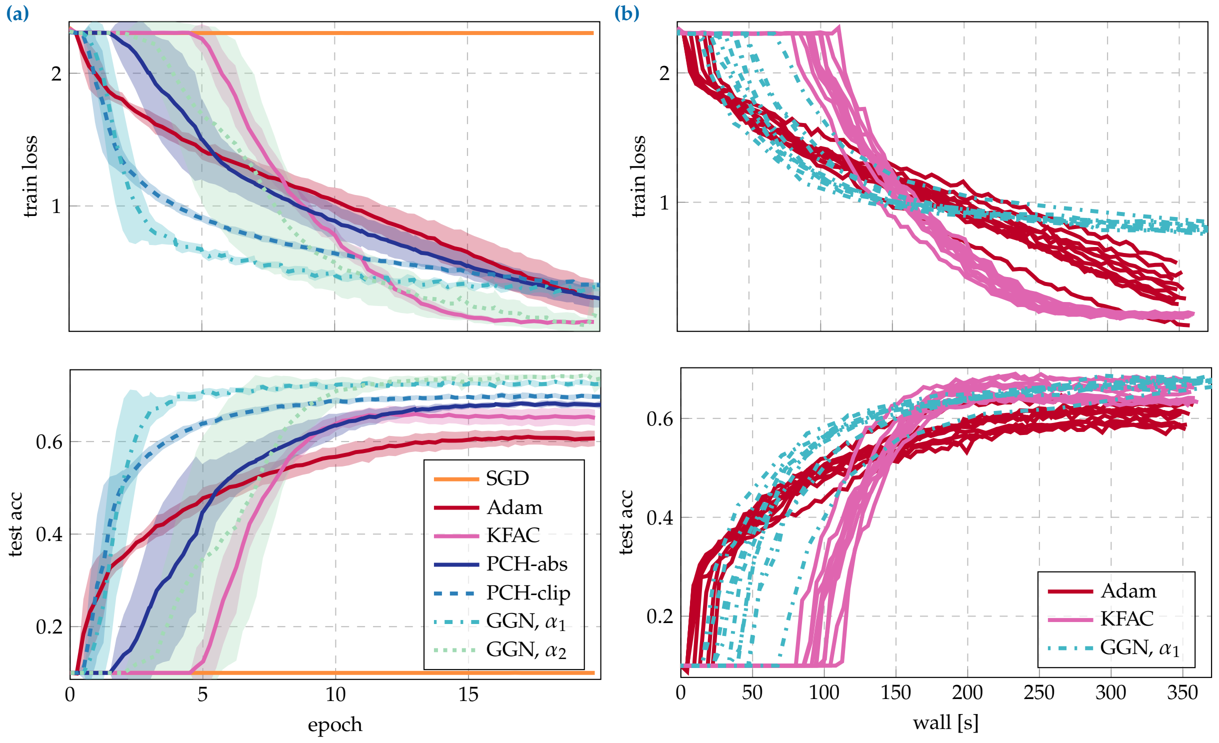


Figure 4.5: SGD, Adam, KFAC, and Newton-style methods with different exact curvature matrix-vector products (HBP). The architecture is a CNN with sigmoid activations (see Appendix A.5). **(a)** Comparison of train and test loss over iterations. SGD cannot train the net. **(b)** Wall-clock time comparison (on an RTX 2080 Ti GPU; same colors realize different random seeds).

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 HBP's flexibility 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.

CNN, Matrix-free Exact Curvature Multiplication

For convolutions, the large number of hidden features prohibits back-propagating 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 (Figure 4.5a). For comparison with another second-order method, we experiment with a public KFAC implementation [63, 109, see Appendix A.5 for details].

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 equation 4.10: 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 4.5b 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 [21, 106, 109] that aim to develop fully fletched second-order optimizers. Such adaptation, however, is beyond the scope of this text.

4.5 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 [21, 31].

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

The authors would like to thank Frederik Kunstner, Matthias Werner, Frank Schneider, and Agustinus Kristiadi for their constructive feedback on the manuscript, and gratefully acknowledge financial support by the European Research Council through ERC StG Action 757275 / PANAMA; the DFG Cluster of Excellence “Machine Learning - New Perspectives for Science”, EXC 2064/1, project number 390727645; the German Federal Ministry of Education and Research (BMBF) through the Tübingen AI Center (FKZ: 01IS18039A); and funds from the Ministry of Science, Research and Arts of the State of Baden-Württemberg. F. D. is grateful to the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for support.

BackPACK: Packing More into Backprop 5.

Abstract

Automatic differentiation frameworks are optimized for exactly one thing: computing the average mini-batch gradient. Yet, other quantities such as the variance of the mini-batch gradients or many approximations to the Hessian can, *in theory*, be computed efficiently, and at the same time as the gradient. While these quantities are of great interest to researchers and practitioners, current deep-learning software does not support their automatic calculation. Manually implementing them is burdensome, inefficient if done naïvely, and the resulting code is rarely shared. This hampers progress in deep learning, and unnecessarily narrows research to focus on gradient descent and its variants; it also complicates replication studies and comparisons between newly developed methods that require those quantities, to the point of impossibility. To address this problem, we introduce BackPACK, an efficient framework built on top of PyTorch, that extends the backpropagation algorithm to extract additional information from first- and second-order derivatives. Its capabilities are illustrated by benchmark reports for computing additional quantities on deep neural networks, and an example application by testing several recent curvature approximations for optimization.

Code and experiments available at the Github repositories `f-dangel/backpack`, `f-dangel/backpack-experiments`

5.1 Introduction

The success of deep learning and the applications it fuels can be traced to the popularization of automatic differentiation frameworks. Packages like TensorFlow [1], Chainer [165], MXNet [32], and PyTorch [126] provide efficient implementations of parallel, GPU-based gradient computations to a wide range of users, with elegant syntactic sugar.

However, this specialization also has its shortcomings: it assumes the user only wants to compute gradients or, more precisely, the average of gradients across a mini-batch of examples. Other quantities can also be computed with AD at a comparable cost or minimal overhead to the gradient backpropagation pass; for example, approximate second-order information or the variance of gradients within the batch. These quantities are valuable to understand the geometry of deep neural networks, for the identification of free parameters, and to push the development of more efficient optimization algorithms. But researchers who want to investigate their use face a chicken-and-egg problem: AD tools required to go beyond standard gradient methods are not available, but there is no incentive for their implementation in existing deep-learning software as long as no large portion of the users need it.

Second-order methods for deep learning have been continuously investigated for decades [*e.g.* 5, 14, 20, 109]. But still, the standard optimizers used in deep learning remain some variant of stochastic gradient descent (SGD); more complex methods have not found wide-spread, practical use. This is in stark contrast to domains like convex optimization and generalized linear models, where second-order methods are the default. There may of course be good scientific reasons for this difference; maybe second-order methods do not work well in the (non-convex, stochastic) setting of deep learning. And the computational cost associated with the high dimensionality of deep models may offset their benefits. Whether these are the case remains somewhat unclear though, because a much more direct road-block is that these methods are so complex to implement that few practitioners ever try them out.

Recent approximate second-order methods such as KFAC [109] show promising results, even on hard deep learning problems [166]. Their approach, based on the earlier work of Schraudolph [148], uses the structure of the network to compute approximate second-order information in a way that is similar to gradient backpropagation. This work sparked a new line of research to improve the second-order approximation [21, 54, 63, 108]. However, all of these methods require low-level applications of automatic differentiation to compute quantities other than the averaged gradient. It is a daunting task to implement them from scratch. Unless users spend significant time familiarizing themselves with the internals of their software tools, the resulting implementation is often inefficient, which also puts the original usability advantage of those packages into question. Even motivated researchers trying to develop new methods, who need not be expert software developers, face this problem. They often end up with methods that cannot compete in run time, not necessarily because the method is inherently bad, but because the implementation is not efficient. New methods are also frequently not compared to their predecessors and competitors because they are so hard to reproduce. Authors do not want to represent the competition in an unfair light caused by a bad implementation.

Another example is offered by a recent string of research to adapt to the *stochasticity* induced by mini-batch sampling. An empirical estimate of the (marginal) variance of the gradients within the batch has been found to be theoretically and practically useful for adapting hyperparameters like learning rates [105] and batch sizes [11], or regularize first-order optimization [10, 85, 94]. To get such a variance estimate, one simply has to square, then sum, the individual gradients after the backpropagation, but before they are aggregated to form the average gradient. Doing so should have negligible cost *in principle*, but is programmatically challenging in the standard packages.

Members of the community have repeatedly asked for such features[1] but the established AD frameworks have yet to address such requests, as their focus has been—rightly—on improving their technical backbone. Features like those outlined above are not generally defined for arbitrary functions, but rather emerge from the specific structure of machine learning applications. General AD frameworks can not be expected to serve such specialist needs. This does not mean, however, that it is impossible to efficiently realize such features within these frameworks: in essence, backpropagation is a technique to compute multiplications with Jacobians. Methods to extract second-order information [110] or individual gradients from a mini-batch [59] have been known to a small group of specialists; they are just rarely discussed or implemented.

1: See *e.g.* the Github issues `github.com/pytorch/pytorch/issues/1407`, `7786`, `8897` and forum discussions `discuss.pytorch.org/t/1433`, `8405`, `15270`, `17204`, `19350`, `24955`.

Computing the gradient with PyTorch . . .

```
X, y     = load_mnist_data()
model    = Linear(784, 10)
lossfunc = CrossEntropyLoss()

loss     = lossfunc(model(X), y)

loss.backward()

for param in model.parameters():
    print(param.grad)
```

. . . and the variance with BackPACK

```
X, y     = load_mnist_data()
model    = extend(Linear(784, 10))
lossfunc = extend(CrossEntropyLoss())

loss     = lossfunc(model(X), y)
with backpack(Variance()):
    loss.backward()

for param in model.parameters():
    print(param.grad)
    print(param.var)
```

Procedure 5.1: **BackPACK integrates with PyTorch to seamlessly extract more information from the backward pass.** Instead of the variance (or alongside it, in the same pass), BackPACK can compute individual gradients in the mini-batch, their L_2 norm and 2nd moment. It can also compute curvature approximations like diagonal or Kronecker factorizations of the GGN such as KFAC, KFLR & KFRA.

5.1.1 Our Contribution

To address this need for a specialized framework focused on machine learning, we propose a framework for the implementation of generalized backpropagation to compute additional quantities. The structure is based on the conceptual work of Dangel et al. [37] for modular backpropagation. This framework can be built on top of existing graph-based backpropagation modules; we provide an implementation on top of PyTorch, coined BackPACK, available at

`https://f-dangel.github.io/backpack/`.

The initial release supports efficient computation of individual gradients from a mini-batch, their L_2 norm, an estimate of the variance, as well as diagonal and Kronecker factorizations of the generalized Gauss-Newton (GGN) matrix (see Table 5.1 for a feature overview). The library was designed to be minimally verbose to the user, easy to use (see Procedure 5.1), and to have low overhead (see Section 5.3). While other researchers are aiming to improve the flexibility of AD systems [23, 77, 78], our goal with this package is to provide access to quantities that are only byproducts of the backpropagation pass, rather than gradients themselves.

To illustrate the capabilities of BackPACK, we use it to implement preconditioned gradient descent optimizers with diagonal approximations of the GGN and recent Kronecker factorizations KFAC [109], KFLR, and KFRA [21]. Our results show that the curvature approximations based on Monte Carlo (MC) estimates of the GGN, the approach used by KFAC, give similar progress per iteration to their more accurate counterparts, but being much cheaper to compute. While the naïve update rule we implement does not surpass first-order baselines such as SGD with momentum and Adam [87], its implementation with various curvature approximations is made straightforward.

5.2 Theory & Implementation

We will distinguish between quantities that can be computed from information already present during a traditional backward pass (which we suggestively call *first-order extensions*), and quantities that need additional information (termed *second-order extensions*). The former group contains additional statistics such as the variance of the gradients within the mini-batch or the L_2 norm of the gradient for each sample. Those can be computed with minimal overhead during the backprop pass. The latter class contains approximations of second-order information, like the diagonal or Kronecker factorization of the generalized Gauss-Newton (GGN) matrix, which require the propagation of additional information through the graph. We will present those two classes separately:

First-order extensions	**Second-order extensions**
Extract more from the standard backward pass.	Propagate new information along the graph.
▸ Individual gradients from a mini-batch	▸ Diagonal of the GGN and the Hessian
▸ L_2 norm of the individual gradients	▸ KFAC [109]
▸ Diagonal covariance and 2nd moment	▸ KFRA and KFLR [21]

These quantities are only defined, or reasonable to compute, for a subset of models: the concept of individual gradients for each sample in a mini-batch or the estimate of the variance requires the loss for each sample to be independent. While such functions are common in machine learning, not all neural networks fit into this category. *E.g.* if the network uses batch normalization [79], the individual gradients in a mini-batch are correlated. Then, the variance is not meaningful anymore, and computing the individual contribution of a sample to the mini-batch gradient or the GGN becomes prohibitive. For those reasons, and to limit the scope of the project for version 1.0, BackPACK currently restricts the type of models it accepts. The supported models are traditional feedforward networks that can be expressed as a *sequence of modules*, for example a sequence of convolutional, pooling, linear and activation layers. Recurrent networks like LSTMs [71] or residual networks [68] are not yet supported, but the framework can be extended to cover them[2].

2: BackPACK has been continuously developed since the initial release. Noteworthy added features include:

- ▸ New extensions: per-sample Hessian/GGN diagonal (version 1.3), matrix-free multiplication with block-diagonal curvature matrices from Chapter 4 (version 1.2), and GGN low-rank factors (versions 1.4, 1.5), see Chapter 7.
- ▸ Broader support of modules and hyperparameters, especially basic support for residual and recurrent networks (version 1.4).

We assume a sequential model $f_{\boldsymbol{\theta}} : \times\mathbb{X} \to \mathbb{F}$ and a dataset of N samples $(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X} \times \mathbb{Y}$ with $n = 1, \dots, N$. The model maps each sample $\boldsymbol{x}_n$ to a prediction $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ using some parameters $\boldsymbol{\theta} \in \Theta$. The predictions are evaluated with a loss function $\ell : \mathbb{F} \times \mathbb{Y} \to \mathbb{R}$, for example the softmax cross-entropy (Equation (2.4)), which compares them to the ground truth $\boldsymbol{y}_n$. This leads to the objective function $\mathcal{L} : \Theta \to \mathbb{R}$,

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)\,. \tag{5.1}$$

As a shorthand, we will use $\ell_n(\boldsymbol{\theta}) = \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)$ for the loss and $\boldsymbol{f}_n(\boldsymbol{\theta}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ for the model output of individual samples. Our goal is to provide more information about the derivatives of $\{\ell_n(\boldsymbol{\theta})\}_{n=1}^{N}$ *w.r.t.* the parameters $\boldsymbol{\theta}$ of the model $f_{\boldsymbol{\theta}}$.

5.2.1 Primer on Backpropagation

ML libraries with integrated automatic differentiation use the modular structure of $\boldsymbol{f}_n(\boldsymbol{\theta})$ to compute derivatives (see [13] for an overview). If $f_{\boldsymbol{\theta}}$ is a sequence of L transformations, it can be expressed as

$$\boldsymbol{f}_n(\boldsymbol{\theta}) = \left(f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ \ldots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}}\right)(\boldsymbol{x}_n)\,, \tag{5.2}$$

where $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ is the lth transformation with parameters $\boldsymbol{\theta}^{(l)}$, such that $\boldsymbol{\theta} = (\boldsymbol{\theta}^{(1)\top}, \dots, \boldsymbol{\theta}^{(L)\top})^\top$. The loss function can also be seen as another

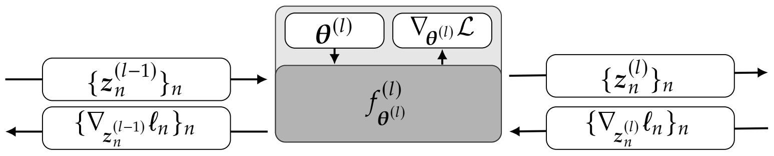

$\theta^{(l)}$
$\nabla_{\theta^{(l)}}\mathcal{L}$
$\{z_n^{(l-1)}\}_n$
$\{\nabla_{z_n^{(l-1)}}\ell_n\}_n$
$f_{\theta^{(l)}}^{(l)}$
$\{z_n^{(l)}\}_n$
$\{\nabla_{z_n^{(l)}}\ell_n\}_n$

Figure 5.1: Schematic representation of the standard backpropagation pass for module l with N samples.

transformation, appended to the network. Let $z_n^{(l-1)}, z_n^{(l)}$ denote the input and output of the operation $f_{\theta^{(l)}}^{(l)}$ for sample n, such that $z_n^{(0)}$ is the original data and $z_n^{(1)}, \dots, z_n^{(L)}$ represent the transformed output of each layer, leading to the computation graph

$$z_n^{(0)} \xrightarrow{f_{\theta^{(1)}}^{(1)}(z_n^{(0)})} z_n^{(1)} \xrightarrow{f_{\theta^{(2)}}^{(2)}(z_n^{(1)})} \dots \xrightarrow{f_{\theta^{(L)}}^{(L)}(z_n^{(L-1)})} z_n^{(L)} \xrightarrow{\ell(z_n^{(L)}, y_n)} \ell_n(\theta)\,.$$

To compute the gradient of ℓ_n *w.r.t.* $\theta^{(l)}$, one unrolls the chain rule,

$$\begin{aligned}\nabla_{\theta^{(l)}}\ell_n(\theta) &= \left(J_{\theta^{(l)}} z_n^{(l)}\right)^\top \left(J_{z_n^{(l)}} z_n^{(l+1)}\right)^\top \cdots \left(J_{z_n^{(L-1)}} z_n^{(L)}\right)^\top \nabla_{z_n^{(L)}}\ell_n(\theta) \\ &= \left(J_{\theta^{(l)}} z_n^{(l)}\right)^\top \nabla_{z^{(l)}}\ell_n(\theta)\,,\end{aligned} \tag{5.3}$$

where $J_a b$ is the Jacobian of b *w.r.t.* a, $[J_a b]_{i,j} = \partial[b]_i/\partial[a]_j$ (see Definition 2.4). A similar expression exists for the module inputs $z_n^{(l-1)}$: $\nabla_{z_n^{(l-1)}}\ell_n(\theta) = (J_{z_n^{(l-1)}} z_n^{(l)})^\top \nabla_{z_n^{(l)}}\ell_n(\theta)$. This recursive structure makes it possible to extract the gradient by propagating the gradient of the loss. In the backpropagation algorithm, a module l receives the loss gradient *w.r.t.* its output, $\nabla_{z_n^{(l)}}\ell_n(\theta)$. It then extracts the gradient with respect to its parameters and inputs, $\nabla_{\theta^{(l)}}\ell_n(\theta)$ and $\nabla_{z_n^{(l-1)}}\ell_n(\theta)$, according to Equation (5.3). The gradient *w.r.t.* its input is sent further down the graph. This process, illustrated in Figure 5.1, is repeated for each transformation until all gradients are computed. To implement backpropagation, each module only needs to know how to multiply with its Jacobians.

For second-order quantities, we rely on the work of Mizutani and Dreyfus [110] and Dangel et al. [37], who showed that a scheme similar to Equation (5.3) exists for the *block diagonal* of the Hessian. A block *w.r.t.* the parameters of a module, $\nabla^2_{\theta^{(l)}}\ell_n(\theta)$, can be obtained by the recursion

$$\begin{aligned}\nabla^2_{\theta^{(l)}}\ell_n(\theta) = &\left(J_{\theta^{(l)}} z_n^{(l)}\right)^\top \nabla^2_{z_n^{(l)}}\ell_n(\theta) \left(J_{\theta^{(l)}} z_n^{(l)}\right) \\ &+ \sum_j \left(\nabla^2_{\theta^{(l)}}\left[z_n^{(l)}\right]_j\right)\left[\nabla_{z_n^{(l)}}\ell_n(\theta)\right]_j\,.\end{aligned} \tag{5.4}$$

A similar relation holds for the module's output Hessian $\nabla^2_{z_n^{(l)}}\ell_n(\theta)$.

Both backpropagations of Equations (5.3) and (5.4) hinge on the multiplication by Jacobians to both vectors and matrices. However, the design of AD limits the application of Jacobians to vectors only. This prohibits the exploitation of vectorization in the matrix case, which is needed for second-order information. The lacking flexibility of Jacobians is one motivation for our work. Since all quantities needed to compute statistics of the derivatives are already computed during the backward pass, another motivation is to provide access to them at minor overhead.

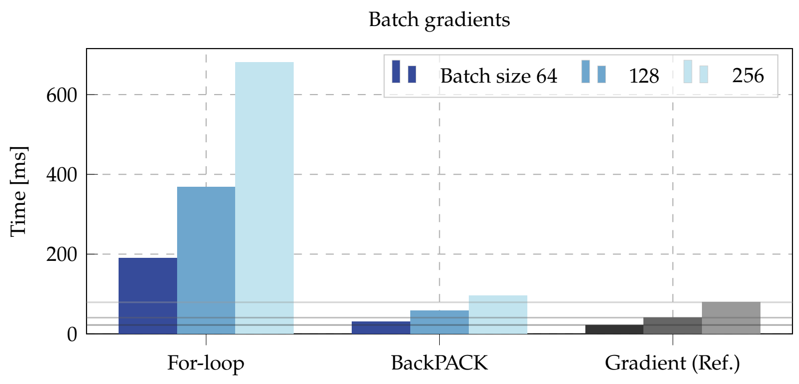


Figure 5.2: Computing individual gradients in a batch using a for-loop (*i.e.* one individual forward and backward pass per sample) or using vectorized operations with BackPACK. The plot shows computation time, comparing to a traditional gradient computation, on the 3c3d network (see Section 5.4) for the CIFAR-10 dataset [146].

5.2.2 First-order Extensions

As the principal first-order extension, consider computing the *per-sample* gradients in a batch of size N. These individual gradients are implicitly computed during a traditional backward pass because the batch gradient is their sum, but they are not directly accessible. The naïve way to compute N individual gradients is to do N separate forward and backward passes, This (inefficiently) replaces every matrix-matrix multiplication by N matrix-vector multiplications. BackPACK batches computations to obtain large efficiency gains, as illustrated by Figure 5.2[3].

3: The latest developments in ML libraries have lead to more efficient alternatives than the for-loop. PyTorch 1.11.0 (released on March 10, 2022) introduced an `is_grads_batched` argument in the API of the `grad` function of its `autograd` library, which allows to compute multiple VJPs in parallel. This reflects the importance of the feature.

As the quantities necessary to compute the individual gradients are already propagated through the computation graph, we can reuse them by inserting code in the standard backward pass. With access to this information, before it is cleared for memory efficiency, BackPACK computes the Jacobian-multiplications for each sample

$$\left\{\nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta})\right\}_{n=1}^N = \left\{\left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right)^\top \nabla_{\boldsymbol{z}_n^{(l)}} \ell_n(\boldsymbol{\theta})\right\}_{n=1}^N , \tag{5.5}$$

without summing the result—see Figure 5.3 for a schematic representation. This duplicates some of the computation performed by the backpropagation, as the Jacobian is applied twice (once by PyTorch and BackPACK with and without summation over the samples, respectively). However, the associated overhead is small compared to the for-loop approach: the major computational cost arises from the propagation of information required for each layer, rather than the formation of the gradient *within* each layer.

This scheme for individual gradient computation is the basis for all first-order extensions. In this direct form, however, it is expensive in memory: if the model is D-dimensional, storing $\mathcal{O}(ND)$ elements is prohibitive for large batches. For the variance, 2nd moment and L_2 norm, BackPACK takes advantage of the Jacobian's structure to directly compute them without forming the individual gradient, reducing memory overhead. See Appendix B.1.1 for details.

Example 5.1 (Symmetric decomposition of the softmax cross-entropy loss Hessian [122]) Consider the softmax cross-entropy loss (Equation (2.2)) Hessian from Table Table 4.1. Its symmetric decomposition $\boldsymbol{S} \in \mathbb{R}^{C\times C}$ is

$$\begin{aligned}\boldsymbol{S} &= \left(\boldsymbol{I} - \boldsymbol{p}\boldsymbol{1}^\top\right)\operatorname{diag}\left(\sqrt{\boldsymbol{p}}\right)\\ &= \operatorname{diag}\left(\sqrt{\boldsymbol{p}}\right) - \boldsymbol{p}\sqrt{\boldsymbol{p}}^\top ,\end{aligned} \tag{5.6}$$

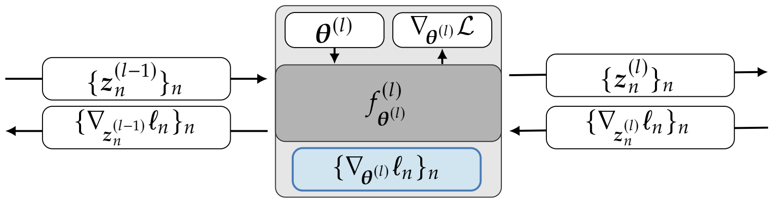
where $\sqrt{\boldsymbol{p}(\boldsymbol{f})} = \boldsymbol{p}(\boldsymbol{f})^{\odot 1/2}$. It satisfies $\nabla_{\boldsymbol{f}}^2 \ell(\boldsymbol{f}, \boldsymbol{y}) = \boldsymbol{S}\boldsymbol{S}^\top$,

$$\begin{aligned}\boldsymbol{S}\boldsymbol{S}^\top &= \left[\operatorname{diag}\left(\sqrt{\boldsymbol{p}}\right) - \boldsymbol{p}\sqrt{\boldsymbol{p}}^\top\right]\\ &\quad\left[\operatorname{diag}\left(\sqrt{\boldsymbol{p}}\right) - \sqrt{\boldsymbol{p}}\boldsymbol{p}^\top\right]\\ &= \operatorname{diag}(\boldsymbol{p}) - 2\boldsymbol{p}\boldsymbol{p}^\top + \boldsymbol{p}\sqrt{\boldsymbol{p}}^\top\sqrt{\boldsymbol{p}}\boldsymbol{p}^\top\\ &= \operatorname{diag}(\boldsymbol{p}) - \boldsymbol{p}\boldsymbol{p}^\top ,\end{aligned}$$

using that $\boldsymbol{p}$'s elements sum to one, *i.e.* $\sqrt{\boldsymbol{p}}^\top\sqrt{\boldsymbol{p}} = 1$. This is the expression from Table 4.1.

5.2.3 Second-order Extensions

Second-order extensions require propagation of more information through the graph. As an example, we will focus on the GGN matrix



$\theta^{(l)}$
$\nabla_{\theta^{(l)}}\mathcal{L}$
$\{z_n^{(l-1)}\}_n$
$\{\nabla_{z_n^{(l-1)}}\ell_n\}_n$
$f_{\theta^{(l)}}^{(l)}$
$\{\nabla_{\theta^{(l)}}\ell_n\}_n$
$\{z_n^{(l)}\}_n$
$\{\nabla_{z_n^{(l)}}\ell_n\}_n$

Figure 5.3: Schematic representation of the individual gradients' extraction in addition to the standard backward pass at the lth module for N samples.

[148]. It is guaranteed to be PSD and is a reasonable approximation of the Hessian near the minimum, which motivates its use in approximate second-order methods. For popular loss functions, it coincides with the Fisher information matrix used in natural gradient methods [5]; for a more in depth discussion of the equivalence, see Section 3.2.4 and the reviews of Martens [107] and Kunstner et al. [92]. For an objective function that can be written as the composition of a loss function ℓ and a model f_θ, such as Equation (5.1), the GGN of $1/N \sum_n \ell(f_\theta(x_n), y_n)$ is

$$G(\theta) = \frac{1}{N}\sum_n \left(\mathrm{J}_\theta f_n\right)^\top \nabla^2_{f_n}\ell(f_n, y_n)\left(\mathrm{J}_\theta f_n\right) . \tag{5.7}$$

The full matrix is too large to compute and store. Current approaches focus on its diagonal blocks, where each block corresponds to a layer in the network. Every block itself is further approximated, for example using a Kronecker factorization. The approach used by BackPACK for their computation is a refinement of the *Hessian Backpropagation equations* of Dangel et al. [37]. It relies on two insights: firstly, the computational bottleneck in the GGN's computation is the multiplication with the Jacobian of the network, $\mathrm{J}_\theta f_n$, while the network output Hessian is easy to compute for most popular loss functions. Secondly, it is not necessary to compute and store each of the N $D \times D$ matrices for a network with D parameters, as Equation (5.7) is a quadratic expression. Given a symmetric factorization S_n of the Hessian, $S_n S_n^\top = \nabla^2_{f_n}\ell(f_n, y_n)$ (*e.g.* Example 5.1), it is sufficient to compute $(\mathrm{J}_\theta f_n)^\top S_n$ and square the result. A network output is typically small compared to its inner layers; networks on CIFAR-100 need $C = 100$ class outputs but could use convolutional layers with more than 100,000 parameters.

The factorization leads to a $D \times C$ matrix, which makes it possible to efficiently compute GGN block diagonals. Also, the computation is very similar to that of a gradient, which computes $(\mathrm{J}_\theta f_n)^\top \nabla_{f_n}\ell_n$. A module $f_{\theta^{(l)}}^{(l)}$ receives the symmetric factorization of the GGN *w.r.t.* its output, $z_n^{(l)}$, and multiplies it with the Jacobians *w.r.t.* the parameters $\theta^{(l)}$ and inputs $z_n^{(l-1)}$ to produce a symmetric factorization of the GGN *w.r.t.* the parameters and inputs, as shown in Figure 5.4.

This propagation serves as the basis of the second-order extensions. If the full symmetric factorization is not wanted, for memory reasons, it is possible to extract more specific information such as the diagonal. If B is the symmetric factorization for a GGN block, the diagonal can be computed as $[\mathrm{diag}(BB^\top)]_i = [BB^\top]_{i,i} = \sum_j [B]^2_{i,j}$.

This framework can be used to extract the main Kronecker factorizations of the GGN, KFAC and KFLR, which we extend to convolution using

Example 5.2 (MC approximation of the softmax-cross entropy loss Hessian) Consider the softmax cross-entropy loss (Equation (2.2)) Hessian from Table 4.1. An MC approximation of the symmetric decomposition (Equation (5.6)) is constructed by the vectors

$$\tilde{s} = \tilde{y}(c) - p(f) \tag{5.8}$$

with $\tilde{y}(c) = \mathrm{onehot}(c)$ and c drawn from a categorical distribution implied by the softmax-probabilities, $c \sim \mathrm{Cat}(c; p(f))$. The random vector $\tilde{y}$ satisfies $\mathbb{E}_c[\tilde{y}] = p(f)$ and $\mathbb{E}_c\left[\tilde{y}\tilde{y}^\top\right] = \mathrm{diag}[p(f)]$. With these properties, we can show $\mathbb{E}_{\tilde{s}}\left[\tilde{s}\tilde{s}^\top\right] = \nabla^2_f \ell$, *i.e.* the expected outer product of Equation (5.8) is the Hessian,

$$\begin{aligned}\mathbb{E}_{\tilde{s}}\left[\tilde{s}\tilde{s}^\top\right] &= \mathbb{E}_c[\tilde{y}\tilde{y}^\top] - \mathbb{E}_c[\tilde{y}]p^\top \\ &\quad - p\mathbb{E}_c[\tilde{y}^\top] + pp^\top \\ &= \mathrm{diag}(p) - pp^\top .\end{aligned}$$

Instead of using the $C \times C$ matrix square root S, we can draw $M < C$ samples $c_1, \dots, c_M$ and stack their $\tilde{s}$-vectors into a smaller $C \times M$ matrix

$$\tilde{S} = \frac{1}{\sqrt{M}}\left(\tilde{s}(c_1)\ \dots\ \tilde{s}(c_M)\right) \tag{5.9}$$

that approximates Equation (5.6),

$$\begin{aligned}\tilde{S}\tilde{S}^\top &= \frac{1}{M}\sum_{i=1}^{M}\tilde{s}(c_i)\tilde{s}(c_i)^\top \\ &\approx \mathbb{E}_{\tilde{s}}\left[\tilde{s}\tilde{s}^\top\right] = SS^\top ,\end{aligned}$$

using less memory. From the connection between Fisher and Hessian, Equation (5.8) are 'would-be gradients' under targets sampled from the model's likelihood, *i.e.* $\tilde{s}^\top = \mathrm{J}_f \ell(f, \tilde{y})$ with $\tilde{y} \sim q(\cdot \mid f)$ and the Jacobian from Table 2.2.

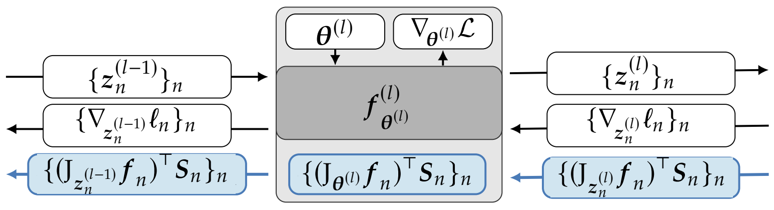

$\theta^{(l)}$
$\nabla_{\theta^{(l)}}\mathcal{L}$
$\{z_n^{(l-1)}\}_n$
$\{\nabla_{z_n^{(l-1)}}\ell_n\}_n$
$\{(\mathrm{J}_{z_n^{(l-1)}} f_n)^\top S_n\}_n$
$f^{(l)}_{\theta^{(l)}}$
$\{(\mathrm{J}_{\theta^{(l)}} f_n)^\top S_n\}_n$
$\{z_n^{(l)}\}_n$
$\{\nabla_{z_n^{(l)}}\ell_n\}_n$
$\{(\mathrm{J}_{z_n^{(l)}} f_n)^\top S_n\}_n$

Figure 5.4: Schematic of the additional backward pass to compute a symmetric factorization of the GGN,

$$G(\theta) = \frac{1}{N}\sum_n (\mathrm{J}_\theta f_n)^\top S_n S_n^\top (\mathrm{J}_\theta f_n)$$

alongside the gradient at the lth module, for N samples.

the approach of Grosse and Martens [63]. The important difference between the two methods is the initial matrix factorization S_n. Using a full symmetric factorization of the initial Hessian, $S_n S_n^\top = \nabla^2_{f_n}\ell_n$, yields the KFLR approximation. KFAC uses an MC approximation by sampling a vector s_n such that $\mathbb{E}_{s_n}[s_n s_n^\top] = \nabla^2_{f_n}\ell_n$ (see Example 5.2). KFLR is therefore more precise but more expensive than KFAC, especially for networks with high-dimensional outputs, which is reflected in our benchmark on CIFAR-100 in Section 5.3. The technical details on how Kronecker factors are extracted and information is propagated for second-order BackPACK extensions are documented in Appendix B.1.2.

5.3 Evaluation & Benchmarks

We benchmark the overhead of BackPACK on CIFAR-10 and CIFAR-100, using the 3c3d network and the All-CNN-C network of Springenberg et al. [157] provided by DeepOBS [146][4]. Figure 5.5 shows the results.

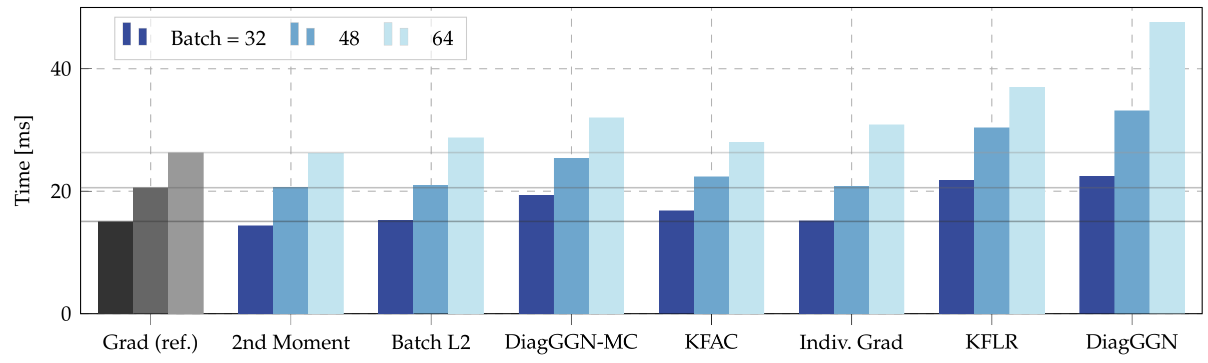
4: 3c3d is a sequence of three convolutions and three dense linear layers with 895,210 parameters. All-CNN-C is a sequence of nine convolutions with 1,387,108 parameters.

For first-order extensions, the computation of individual gradients from a mini-batch adds noticeable overhead due to the additional memory requirements to store them. But more specific quantities such as the L_2 norm, 2nd moment and variance can be extracted efficiently. Regarding second-order extensions, the GGN computation can be expensive for nets with large outputs like CIFAR-100, regardless of the approximation being diagonal of Kronecker-factored. Thankfully, the MC approximation used by KFAC, which we also implement for a diagonal approximation, can be computed at minimal overhead—much less than two backward passes. This last point is encouraging, as our optimization experiment in Section 5.4 suggest that this approximation is reasonably accurate.

5.4 Experiments

To illustrate the utility of BackPACK, we implement preconditioned gradient descent optimizers using diagonal and Kronecker approximations of the GGN. To our knowledge, and despite their apparent simplicity, results using diagonal approximations or the naïve damping update rule we chose have not been reported in publications so far. However, this section is not meant to introduce a bona-fide new optimizer. Our goal is to show that BackPACK can enable research of this kind. The update rule we implement uses a curvature matrix $C(\theta_t^{(l)})$, which could be a

(a) 3c3d, CIFAR-10



(b) All-CNN-C, CIFAR-100

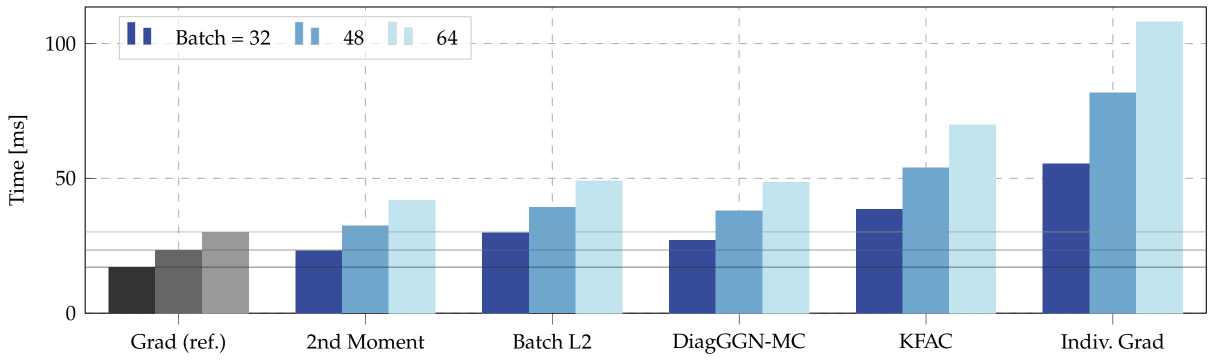


Figure 5.5: Overhead benchmark for computing the gradient *and* first- or second-order extensions on real networks, compared to just the gradient. Most quantities add little overhead. KFLR and DiagGGN propagate 100× more information than KFAC and DiagGGN-MC on CIFAR-100 and are two orders of magnitude slower. We report benchmarks on those, and the Hessian's diagonal, in Appendix B.2.

diagonal or Kronecker factorization of the GGN blocks, and a damping parameter λ to precondition the gradient:

$$\boldsymbol{\theta}^{(l)}_{t+1} = \boldsymbol{\theta}^{(l)}_{t} - \eta \left(\boldsymbol{C}(\boldsymbol{\theta}^{(l)}_{t}) + \lambda \boldsymbol{I} \right)^{-1} \nabla_{\boldsymbol{\theta}^{(l)}_{t}} \mathcal{L}(\boldsymbol{\theta}^{(l)}_{t}) \,, \qquad l = 1, \dots, L \,. \tag{5.10}$$

We run the update rule with the following approximations of the generalized Gauss-Newton: the exact diagonal (DiagGGN) and an MC estimate (DiagGGN-MC), and the Kronecker factorizations KFAC [109], KFLR and KFRA [5] [21].The inversion required by the update rule is straightforward for the diagonal curvature. For the Kronecker-factored quantities, we use the approximation introduced by [109] (see Appendix B.3.3).

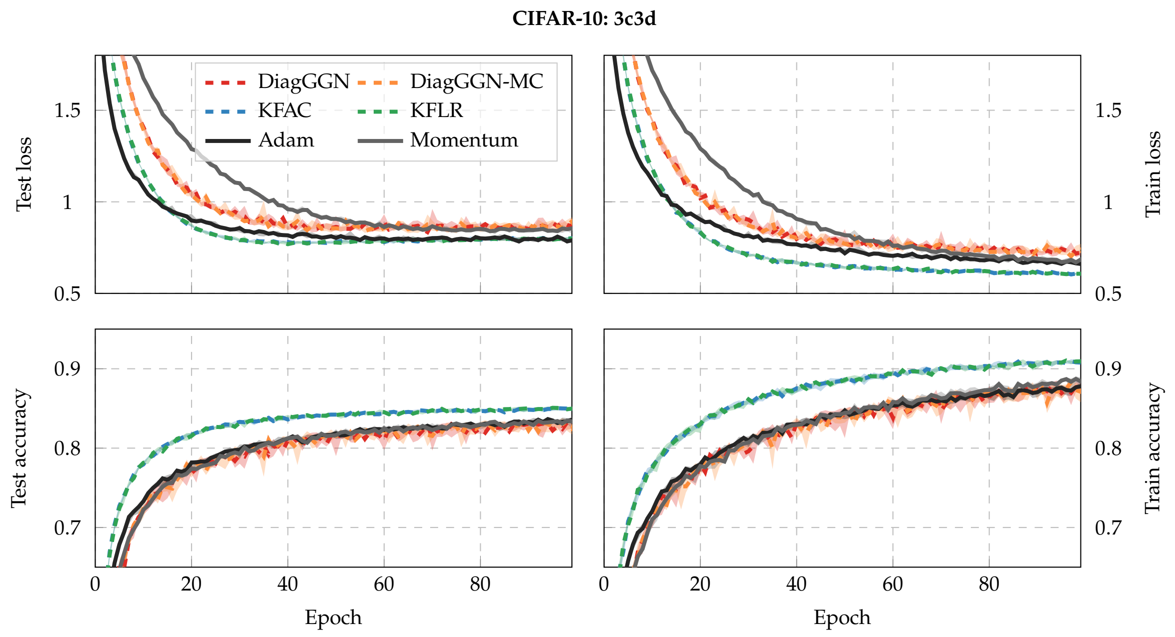
5: KFRA was not originally designed for convolutions; we extend it using the Kronecker factorization of Grosse and Martens [63]. While it can be computed for small networks on MNIST, which we report in Appendix B.3.4, the approximate backward pass of KFRA does not seem to scale to large convolution layers.

These curvature estimates are tested for the training of deep neural networks by running the corresponding optimizers on the main test problems of the benchmarking suite DeepOBS [6] [146]. We use the setup (batch size, number of training epochs) of DeepOBS' baselines, and tune the learning rate η and damping parameter λ with a grid search for each optimizer (details in Appendix B.3.2). The best hyperparameter settings is chosen according to the final accuracy on a validation set. We report the median and quartiles of the performance for ten random seeds.

6: `deepobs.github.io`. We cannot run BackPACK on all test problems in this benchmark due to the limitations outlined in Section 5.2. Despite this limitation, we still run on models spanning a representative range of image classification problems.

Figure 5.6a shows the results for the 3c3d network trained on CIFAR-10. The optimizers that leverage Kronecker-factored curvature approximations beat the baseline performance in terms of per-iteration progress on the training loss, training and test accuracy. Using the same hyperparam-

(a)



(b)

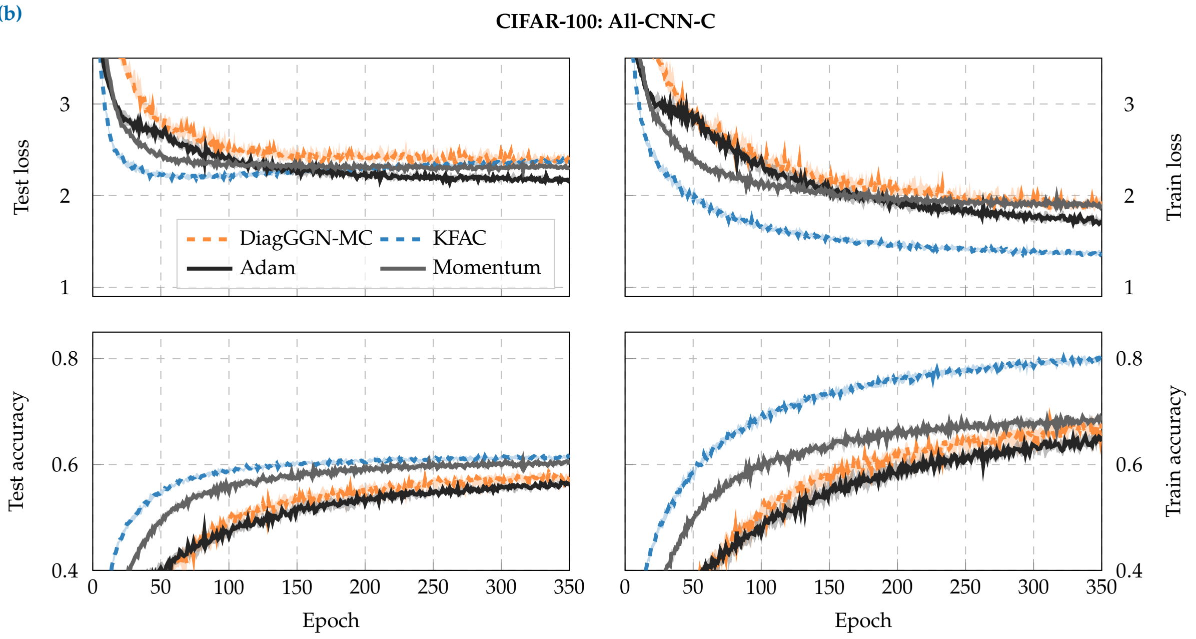


Figure 5.6: Median performance with shaded quartiles of the DeepOBS benchmark for **(a)** 3c3d (895,210 parameters) on CIFAR-10 and **(a)** All-CNN-C (1,387,108 parameters) on CIFAR-100. Solid lines show DeepOBS' baselines of momentum SGD and Adam.

eters, there is little difference between KFAC and KFLR, or DiagGGN and DiagGGN-MC. Given that the quantities based on MC-sampling are considerably cheaper, this experiment suggests it being an important technique to reduce the computational burden of curvature proxies.

Figure 5.6b shows benchmarks for the All-CNN-C network trained on CIFAR-100. Due to the high-dimensional output, the curvatures using a full matrix propagation rather than an MC sample cannot be run on this problem due to memory issues. Both DiagGGN-MC and KFAC can compete with the baselines in terms of progress per iteration. As the update rule we implemented is simplistic on purpose, this is promising for future applications of second-order methods that can more efficiently use the additional information given by curvature approximations.

Feature	Details
Individual gradients	$\frac{1}{N}\nabla_{\boldsymbol{\theta}^{(l)}}\ell_n(\boldsymbol{\theta}), \quad n = 1, \dots, N$
Batch variance	$\frac{1}{N}\sum_{n=1}^{N}\left[\nabla_{\boldsymbol{\theta}^{(l)}}\ell_n(\boldsymbol{\theta})\right]_j^2 - \left[\nabla_{\boldsymbol{\theta}^{(l)}}\mathcal{L}(\boldsymbol{\theta})\right]_j^2$
2nd moment	$\frac{1}{N}\sum_{n=1}^{N}\left[\nabla_{\boldsymbol{\theta}^{(l)}}\ell_n(\boldsymbol{\theta})\right]_j^2, \quad j = 1, \dots, d^{(l)}.$
Indiv. gradient L_2 norm	$\left\lVert\frac{1}{N}\nabla_{\boldsymbol{\theta}^{(l)}}\ell_n(\boldsymbol{\theta})\right\rVert_2^2, \quad n = 1, \dots, N$
DiagGGN	$\mathrm{diag}\left(\boldsymbol{G}(\boldsymbol{\theta}^{(l)})\right)$
DiagGGN-MC	$\mathrm{diag}\left(\tilde{\boldsymbol{G}}(\boldsymbol{\theta}^{(l)})\right)$
Hessian diagonal	$\mathrm{diag}\left(\nabla_{\boldsymbol{\theta}}^2\mathcal{L}(\boldsymbol{\theta})\right)$
KFAC	$\tilde{\boldsymbol{G}}^{(l)}(\boldsymbol{\theta}^{(l)}) \approx \boldsymbol{A}^{(l)} \otimes \boldsymbol{B}_{\text{KFAC}}^{(l)}$
KFLR	$\boldsymbol{G}^{(l)}(\boldsymbol{\theta}^{(l)}) \approx \boldsymbol{A}^{(l)} \otimes \boldsymbol{B}_{\text{KFLR}}^{(l)}$
KFRA	$\boldsymbol{G}^{(l)}(\boldsymbol{\theta}^{(l)}) \approx \boldsymbol{A}^{(l)} \otimes \boldsymbol{B}_{\text{KFRA}}^{(l)}$

Table 5.1: Overview of the features supported in the first release of BackPACK.

5.5 Conclusion

Machine learning's coming-of-age has been accompanied, and in part driven, by a maturing of the software ecosystem. This has drastically simplified the lives of developers and researchers alike, but has also crystallized parts of the algorithmic landscape. This has dampened research in cutting-edge areas that are far from mature, like second-order optimization for deep neural networks. To ensure that good ideas can bear fruit, researchers must be able to compute new quantities without an overwhelming software development burden. To support research and development in optimization for deep learning, we have introduced BackPACK, an efficient implementation in PyTorch of recent conceptual advances and extensions to backpropagation (Table 5.1 lists all features). BackPACK enriches the syntax of AD packages to offer additional observables to optimizers beyond the batch-averaged gradient. Our experiments demonstrate that BackPACK's implementation offers drastic efficiency gains over the kind of naïve implementation within reach of the typical researcher. As a demonstrative example, we "invented" a few optimization routines that, without BackPACK, would require demanding implementation work and can now be tested with ease. We hope that studies like this allow BackPACK to help mature the ML software ecosystem further.

Acknowledgments

We would like to thank Aaron Bahde, Ludwig Bald, and Frank Schneider for their help with DeepOBS and Lukas Balles, Simon Bartels, Filip de Roos, Tim Fischer, Nicolas Krämer, Agustinus Kristiadi, Frank Schneider, Jonathan Wenger, and Matthias Werner for constructive feedback.

The authors gratefully acknowledge financial support by the European Research Council through ERC StG Action 757275 / PANAMA; the DFG Cluster of Excellence "Machine Learning - New Perspectives for Science", EXC 2064/1, project number 390727645; the German Federal Ministry of Education and Research (BMBF) through the Tübingen AI Center (FKZ: 01IS18039A); and funds from the Ministry of Science, Research and Arts of the State of Baden-Württemberg. F. D. is grateful to the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for support.

Cockpit: A Practical Debugging Tool for the Training of Deep Neural Networks 6.

Abstract

When engineers train deep learning models, they are very much "flying blind". Commonly used methods for real-time training diagnostics, such as monitoring the train/test loss, are limited. Assessing a net's training process solely through these performance indicators is akin to debugging software without access to internal states through a debugger. To address this, we present Cockpit, a collection of instruments that enable a closer look into the inner workings of a learning machine, and a more informative and meaningful status report for practitioners. It facilitates the identification of learning phases and failure modes, like ill-chosen hyperparameters. The instruments leverage novel higher-order information about the gradient distribution and curvature, which has only recently become efficiently accessible. We believe that such a debugging tool, which we open-source for PyTorch, is valuable in troubleshooting the training process. By revealing new insights, it also more generally contributes to explainability and interpretability of deep nets.

Code and experiments available at the Github repositories `f-dangel/cockpit`, `f-dangel/cockpit-experiments`

6.1 Introduction & Motivation

Deep learning represents a new programming paradigm: instead of deterministic programs, users design models and "simply" train them with data. In this metaphor, deep learning is a meta-programming form, where *coding* is replaced by *training*. Here, we ponder the question how we can provide more insight into this process by building a *debugger* specifically designed for deep learning.

Debuggers are crucial for traditional software development. When things fail, they provide access to the internal workings of the code, allowing a look "into the box". This is much more efficient than re-running the program with different inputs. And yet, deep learning is arguably closer to the latter. If the attempt to train a deep net fails, a machine learning engineer faces various options: should they change the training hyperparameters (how?); the optimizer (to which one?); the model (how?); or just re-run with a different seed? Machine learning toolboxes provide scant help to guide these decisions.

Of course, traditional debuggers can be applied to deep learning. They will give access to every single weight of a neural net, or the individual pixels of its training data. But this rarely yields insights towards successful training. Extracting meaningful information requires a statistical approach and distillation of the bewildering complexity into a manageable summary. Tools like TensorBoard [1] or Weights & Biases [17] were built in part to streamline this visualization. Yet, the quantities that are widely monitored (mainly train/test loss & accuracy), provide only scant explanation for relative differences between multiple training runs, because *they do not show the network's internal state*. Figure 6.1 illustrates

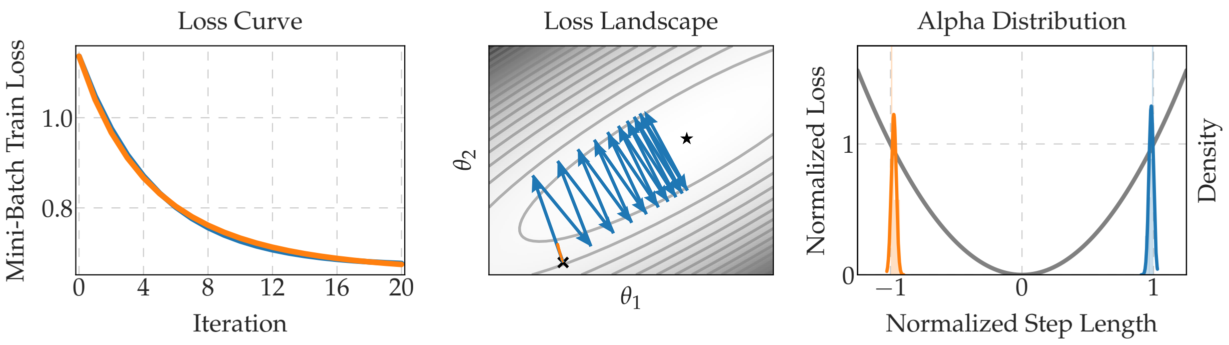


Figure 6.1: Illustrative example: learning curves do not tell the whole story. Two different optimization runs (—/—) can lead to virtually the same loss curve (*left*). However, the actual optimization trajectories (*middle*), exhibit vastly different behaviors. In practice, the trajectories are intractably large and cannot be visualized directly. Recommendable actions for both scenarios (increase/decrease the learning rate) cannot be inferred from the loss curve. The α-distribution, one Cockpit instrument (*right*), not only clearly distinguishes the two scenarios, but also allows for taking decisions how the learning rate should be adapted. See Section 6.3.3 for further details.

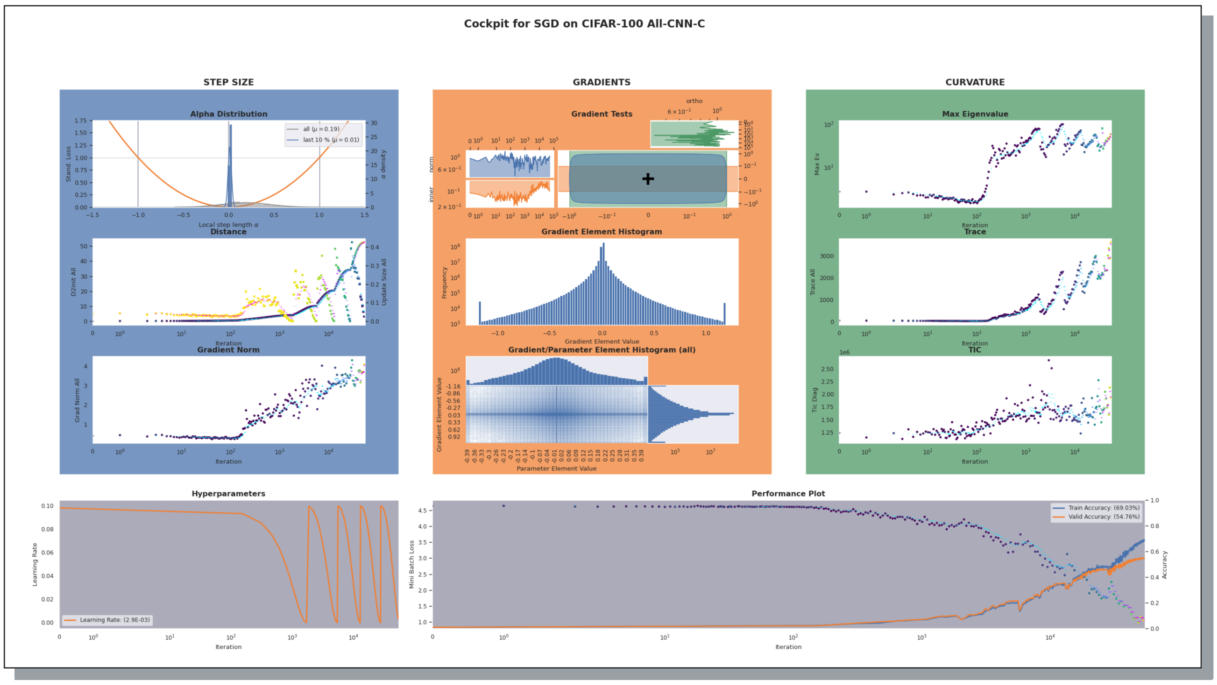


Figure 6.2: Screenshot of Cockpit's full view while training the All-CNN-C [157] on CIFAR-100 with SGD using a cyclical learning rate schedule. This figure and its labels are not meant to be legible, but rather give an impression of Cockpit's user experience. Gray panels (bottom row) show the information currently tracked by most practitioners. The individual instruments are discussed in Section 6.2, and observations are described in Section 6.4. An animated version can be found in the accompanying Github repository.

how such established learning curves can describe the *current* state of the model – whether it is performing well or not – while failing to inform about training state and dynamics. They tell the user *that* things are going well or badly, but not *why*. The situation is similar to flying a plane by sight, without instruments to provide feedback. It is not surprising, then, that achieving state-of-the-art performance in deep learning requires expert intuition, or plain trial & error.

We aim to enrich the deep learning pipeline with a visual and statistical debugging tool that uses newly proposed observables as well as several established ones (Section 6.2). We leverage and augment recent extensions to AD (*i.e.* BackPACK [38] for PyTorch [126]) to efficiently access second-order statistical (*e.g.* gradient variances) and geometric (*e.g.* Hessian) information. We show how these quantities can aid the deep learning engineer in tasks, like learning rate selection, as well as detecting common bugs with data processing or model architectures (Section 6.3).

Concretely, we introduce Cockpit, a flexible and efficient framework for online-monitoring these observables during training in carefully designed plots we call "instruments" (Figure 6.2). To practical, such visualization must have a manageable computational overhead. We show that Cockpit scales well to real-world deep learning problems (see Figure 6.2 and Section 6.4). We also provide three different configurations of varying computational complexity and demonstrate that their instruments keep the computational cost *well below* a factor of 2 in run time (Section 6.5). It is available as open-source code, extendable, and seamlessly integrates into existing PyTorch training loops (Appendix C.1).

6.2 Cockpit's Instruments

Setting

We consider supervised regression/classification with labeled data $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{X} \times \mathbb{Y}$ generated by a distribution $p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$. The training set $\mathbb{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_{n=1}^N$ consists of N *i.i.d.* samples from p_{data} and the deep model $f_{\boldsymbol{\theta}} : \mathbb{X} \to \mathbb{F}$ maps inputs $\boldsymbol{x}_n$ to predictions $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ by parameters $\boldsymbol{\theta} \in \mathbb{\Theta} := \mathbb{R}^D$. This prediction is evaluated by a loss function $\ell : \mathbb{F} \times \mathbb{Y} \to \mathbb{R}$ which compares to the label $\boldsymbol{y}_n$. The goal is minimizing an inaccessible expected risk $\mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta}) = \int \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) \, \mathrm{d}p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y})$ by empirical approximation through $\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) = {}^1\!/_N \sum_{n=1}^N \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n) := {}^1\!/_N \sum_{n=1}^N \ell_n(\boldsymbol{\theta})$, which in practice is stochastically sub-sampled on mini-batches $\mathbb{B} \subset \mathbb{D}$,

$$\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{B}} \ell_n(\boldsymbol{\theta}) \,. \tag{6.1}$$

As is standard practice, we use first- and second-order information of the mini-batch loss, described by its gradient $\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})$ and Hessian $\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})$,

$$\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{B}} \nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta}) \,, \quad \boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{B}|} \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{B}} \nabla_{\boldsymbol{\theta}}^2 \ell_n(\boldsymbol{\theta}) \,. \tag{6.2}$$

Design Choices

To minimize computational and design overhead, we restrict the metrics to quantities that require no additional model evaluations. This means that, at training step $t \to t+1$ with mini-batches $\mathbb{B}_t, \mathbb{B}_{t+1}$ and parameters $\boldsymbol{\theta}_t, \boldsymbol{\theta}_{t+1}$, we access information about the mini-batch losses $\mathcal{L}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)$ and $\mathcal{L}_{\mathbb{B}_{t+1}}(\boldsymbol{\theta}_{t+1})$, but no cross-terms that require additional forward passes.

Key Point

$\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$, $\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})$, and $\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})$ are just expected values of a *distribution* over the batch. Only recently, this distribution has begun to attract attention [48] as its computation has become more accessible [23, 38]. Contemporary optimizers leverage only the *mean* gradient and neglect higher moments.

Table 6.1: Overview of Cockpit quantities. They range from cheap byproducts, to nonlinear transformations of first-order information and Hessian-based measures. Some quantities have already been proposed, others are first to be considered in this work. They are categorized into configurations *economy* ⊆ *business* ⊆ *full* based on their run time overhead (see Section 6.5 for a detailed evaluation).

Name	Short Description	Config	Pos. in Fig. 6.2
`Alpha`	Normalized step size on a noisy quadratic interpolation between $\boldsymbol{\theta}_t, \boldsymbol{\theta}_{t+1}$	*economy*	top left
`Distance`	Distance from initialization $\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_0\|_2$	*economy*	middle left
`UpdateSize`	Update size $\|\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t\|_2$	*economy*	middle left
`GradNorm`	Mini-batch gradient norm $\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|_2$	*economy*	bottom left
`NormTest`	Normalized fluctuations of residual norms $\|\boldsymbol{g}_{\mathbb{B}} - \boldsymbol{g}_n\|_2$, proposed in [26]	*economy*	top center
`InnerTest`	Normalized fluctuations of $\boldsymbol{g}_n$'s parallel to $\boldsymbol{g}_{\mathbb{B}}$, proposed in [19]	*economy*	top center
`OrthoTest`	Like `InnerTest` but using the orthogonal components, proposed in [19]	*economy*	top center
`GradHist1d`	Histogram of individual gradient elements, $\{[\boldsymbol{g}_n]_j\}_{(\boldsymbol{x}_n,\boldsymbol{y}_n)\in\mathbb{B}}^{j=1,\dots,D}$	*economy*	middle center
`TICDiag`	Relation of (diagonal) curvature and gradient noise, inspired by [162]	*business*	bottom right
`HessTrace`	Exact or approximate Hessian trace, $\mathrm{Tr}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, inspired by [177]	*business*	middle right
`HessMaxEV`	Maximum Hessian eigenvalue, $\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, inspired by [177]	*full*	top right
`GradHist2d`	Histogram of weights & per-sample gradients, $\{(\theta_j, [\boldsymbol{g}_n]_j)\}_{(\boldsymbol{x}_n,\boldsymbol{y}_n)\in\mathbb{B}}^{j=1,\dots,D}$	*full*	bottom center

One core point of our work is making extensive use of these distribution properties, trying to visualize them in various ways. This distinguishes Cockpit from being "just a collection of plots" that could be built in tools like TensorBoard. Leveraging these distributional quantities, we create instruments and show how they can help adapt hyperparameters (Section 6.2.1), analyze the loss landscape (Section 6.2.2), and track network dynamics (Section 6.2.3). Instruments can sometimes be built from already-computed information or are efficient variants of previously proposed observables. To keep the presentation concise, we highlight the instruments shown in Figure 6.2 and listed in Table 6.1. Appendix C.3 defines them formally and contains more extensions, such as the mean GSNR [100], the early stopping [104] and CABS [11] criterion, which can all be used in Cockpit.

6.2.1 Adapting Hyperparameters

One big challenge in deep learning is setting the hyperparameters correctly, which is currently mostly done by trial & error through parameter searches. We aim to augment this process with instruments that inform the user about the effect that the chosen parameters have on the current training process.

`Alpha`: Are We Crossing the Valley?

Using individual loss and gradient observations at the start and end point of each iteration, we build a noise-informed uni-variate quadratic approximation along the step direction (*i.e.* the loss as a function of the step size), and assess to which point on this parabola our optimizer moves. We standardize this value α such that stepping to the valley-floor is assigned $\alpha = 0$, the starting point is at $\alpha = -1$ and updates to the point exactly opposite of the starting point have $\alpha = 1$ (see Appendix C.3.2 for a more detailed visual and mathematical description of α). Figure 6.1 illustrates the scenarios $\alpha = \pm 1$ and how monitoring the

α-distribution (right panel) can help distinguish between two training runs with similar performance but distinct failure sources. By default, this Cockpit instrument shows the α-distribution for the last 10 % of training and the entire training process (top left plot in Figure 6.2). In Section 6.3.3 we show empirically that, *counter-intuitively*, it is generally *not* a good idea to choose the step size such that α is close to zero.

Distances: Are We Making Progress?

Another way to discern the trajectories in Figure 6.1 is by measuring the L_2 *distance from initialization* [113] and the *update size* [4, 51] in parameter space. Both are shown together in one Cockpit instrument (see also center-left plot in Figure 6.2) and are far larger for the blue line in Figure 6.1. These distance metrics are also able to disentangle phases for the blue path. Using the same step size, it will continue to "jump back and forth" between the loss valley's walls but at some point cease to make progress. During this "surfing of the walls", the *distance from initialization* increases, ultimately though, it will stagnate, with the *update size* remaining non-zero, indicating diffusion. While the initial "surfing the wall"-phase benefits training (see Section 6.3.3), achieving stationarity may require adaptation once the optimizer reaches that diffusion.

Gradient Norm: How Steep Is the Wall?

The *update size* will show that the orange trajectory is stuck. But why? Such slow-down can result from both a bad learning rate and from loss landscape plateaus. The *gradient norm* (bottom left panel in Figure 6.2) distinguishes these two causes.

Gradient Tests: How Noisy Is the Batch?

The batch size trades off gradient accuracy versus computational cost. Recently, adaptive sampling strategies based on testing geometric constraints between mean and individual gradients have been proposed [19, 26]. The *norm*, *inner product*, and *orthogonality tests* use a standardized radius and two band widths (parallel and orthogonal to the gradient mean) that indicate how strongly individual gradients scatter around the mean. The original works use these values to adapt batch sizes. Instead, Cockpit combines all three tests into a single gauge (top middle plot of Figure 6.2) using the standardized noise radius and band widths for visualization. These noise signals can be used to guide batch size adaptation on- and offline, or to probe the influence of gradient alignment on training speed [142] and generalization [27, 28, 100].

6.2.2 Hessian Properties for Local Loss Geometry

An intuition for the local loss landscape helps in many ways. It can help diagnose whether training is stuck, to adapt the step size, and explain stability or regularization [56, 81]. The key challenge is the large number of weights: low-dimensional projections of surfaces can behave unintuitively [112], but tracking the extreme or average behaviors may help in debugging, especially if first-order metrics fail.

Hessian Eigenvalues: A Gorge or a Lake?

In convex optimization, the maximum Hessian eigenvalue crucially determines the appropriate step size [144]. Many works have studied the Hessian spectrum in machine learning [*e.g.* 55, 56, 112, 139, 140, 177]. In short: curvature matters. Established [127] and recent autodiff frameworks [38] can compute Hessian properties without requiring the full matrix. Cockpit leverages this to provide the *Hessian's largest eigenvalue* and *trace* (right top and middle plots in Figure 6.2). The former resembles the loss surface's sharpest valley and can thus hint at training instabilities [81]. The *trace* describes a notion of "average curvature", since the eigenvalues λ_i relate to it by $\sum_i \lambda_i = \operatorname{Tr}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, which might correlate with generalization [80].

TIC: How Do Curvature & Gradient Noise Interact?

There is an ongoing debate about curvature's link to generalization [*e.g.* 42, 70, 86]. The *Takeuchi Information Criterion (TIC)* [160, 162] estimates the generalization gap by a ratio between Hessian and non-central second gradient moment. It also provides intuition for changes in the objective function implied by gradient noise. Inspired by [162], Cockpit provides mini-batch TIC estimates (bottom right plot of Figure 6.2).

6.2.3 Visualizing Internal Network Dynamics

Histograms are a natural visual compression of the high-dimensional $|\mathbb{B}| \times D$ individual gradient values. They give insights into the gradient *distribution* and hence offer a more detailed view of the learning signal. Together with the parameter associated to each individual gradient, the entire model status and dynamics can be visualized in a single plot and be monitored during training. This provides a more fine-grained view of training compared to tracking parameters and gradient norms [51].

Gradient & Parameter Histograms: What Is Happening in Our Net?

Cockpit offers a uni-variate *histogram of the gradient elements*, *i.e.* the numbers $\{[\boldsymbol{g}_n(\boldsymbol{\theta})]_j\}_{(\boldsymbol{x}_n,\boldsymbol{y}_n)\in\mathbb{B}}^{j=1,\dots,D}$. Additionally, a combined *histogram of parameter-gradient pairs* $\{([\boldsymbol{\theta}]_j, [\boldsymbol{g}_n(\boldsymbol{\theta})]_j\}_{(\boldsymbol{x}_n,\boldsymbol{y}_n)\in\mathbb{B}}^{j=1,\dots,D}$ provides a two-dimensional look into the network's gradient and parameter values in a mini-batch. Section 6.3.1 shows an example use-case of the gradient histogram; Section 6.3.2 makes the case for the layer-wise variants of the instruments.

6.3 Experiments

The diverse information provided by Cockpit can help users and researchers in many ways, some of which, just like for a traditional debugger, only become apparent in practical use. In this section, we present a few motivating examples, selecting specific instruments and scenarios in which they are practically useful. Specifically, we show that Cockpit can help the user discern between, and thus fix, common training bugs (Sections 6.3.1 and 6.3.2) that are otherwise hard to distinguish as they lead to the same failure: bad training. We demonstrate that Cockpit can guide practitioners to choose efficient hyperparameters *within a single training run* (Sections 6.3.2 and 6.3.3). Finally, we highlight that Cockpit's instruments can provide research insights about the optimization process (Section 6.3.3). Our empirical findings are demonstrated on problems from the DeepOBS [146] benchmark collection.

6.3.1 Incorrectly Scaled Data

One prominent source of bugs is the data pipeline. To pick a relatively simple example: for standard optimizers to work at their usual learning rates, network inputs must be standardized (*i.e.* between zero and one, or have zero mean and unit variance [*e.g.* 15]). If the user forgets to do this, optimizer performance is likely to degrade. It can be difficult to identify the source of this problem as it does not cause obvious failures, `NaN` or `Inf` gradients, *etc.*. We now construct a semi-realistic example, to show how using Cockpit can help diagnose this problem upon observing slow training performance.

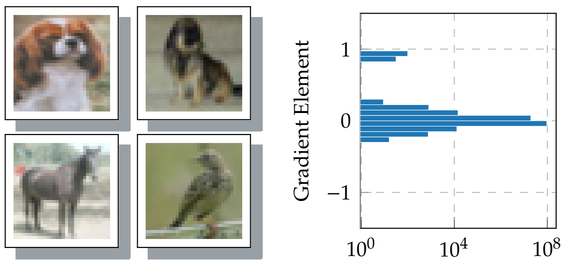
By default[1], the popular image datasets CIFAR-10/100 [90] are provided as NumPy [66] arrays that consist of integers in the interval $[0, 255]$. This *raw* data, instead of the widely used version with floats in $[0, 1]$, changes the data scale and thus the gradients by a factor of 255. Therefore, the optimizer's optimal learning rate is scaled as well. In other words, the default parameters of popular optimization methods may not work well anymore, or good hyperparameters may take extreme values. Even if the user directly inspects the training images, this may not be apparent (Figure 6.3). But the gradient histogram instrument of Cockpit, which has a deliberate default plotting range around $[-1, 1]$ to highlight such problems, immediately and prominently shows that there is an issue.

1: See the documentation, available at `cs.toronto.edu/~kriz/cifar.html`

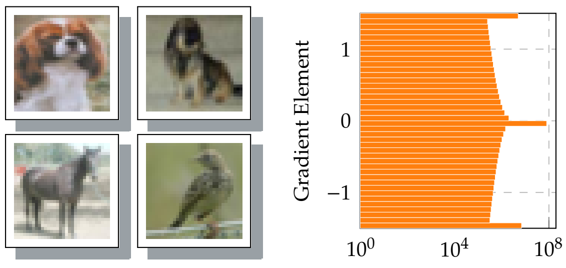
Of course, this particular data is only a placeholder for real practical data sets. While this problem may not frequently arise in the highly pre-processed, packaged CIFAR-10, it is not a rare problem for practitioners who work with their personal datasets. This is particularly likely in domains outside standard computer vision, *e.g.* when working with mixed-type data without obvious natural scales.

6.3.2 Vanishing Gradients

The model itself can be a source of training bugs. As before, such problems mostly arise with novel datasets, where well-working architectures are

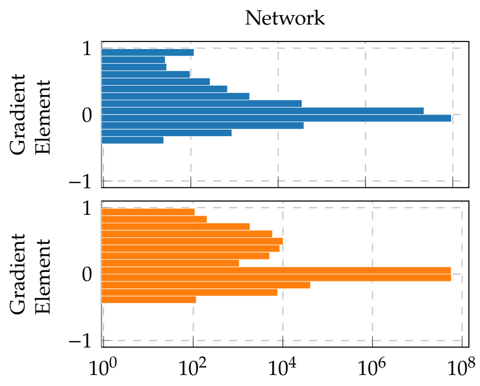


(a) Normalized Data

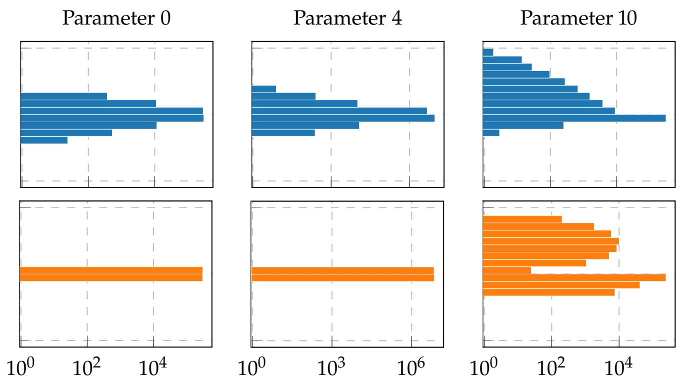


(b) Raw Data

Figure 6.3: Same inputs, different gradients; Catching data bugs with Cockpit. (a) *normalized* ($[0, 1]$) and (b) *raw* ($[0, 255]$) images look identical in auto-scaled front-ends like matplotlib's `imshow`. The gradients on 3c3d, however, are crucially affected by this scaling.



(a) Network Histogram



(b) Layer-wise Histograms

Figure 6.4: Gradient distributions of two similar architectures on the same problem. (a) Distribution of individual gradient elements summarized over the entire network. Both seem similar. (b) Layer-wise histograms for a subset of layers. Parameter 0 is the layer closest to the network's input, parameter 10 closest to its output. Only the layer-wise view reveals that there are several degenerated gradient distributions for the orange network making training unnecessary hard.

unknown. The following example shows how even small (in terms of code) model modifications may severely harm the training.

Figure 6.4a shows the distribution of gradient values of two different network architectures in blue and orange. Although the blue model trains considerably better than the orange one, their gradient distributions look quite similar. The difference becomes evident when inspecting the histogram *layer-wise*. We can see that multiple layers have a degenerated gradient distribution with many elements being practically zero (see Figure 6.4b, bottom row). Since the fully connected layers close to the output have far more parameters (a typical pattern of convolutional networks), they dominate the network-wide histogram. This obscures that a major part of the model is effectively unable to train.

Both the blue and orange networks follow DeepOBS's 3c3d architecture. The only difference is the non-linearity: the blue net uses standard ReLU activations, while the orange one has sigmoid activations. Here, the layer-wise histogram instrument of Cockpit highlights which part of the architecture makes training unnecessarily hard. Accessing information layer-wise is also essential due to the strong overparameterization in deep models where training can happen in small subspaces [65]. Again, this is hard to do with common monitoring tools, such as the loss curve.

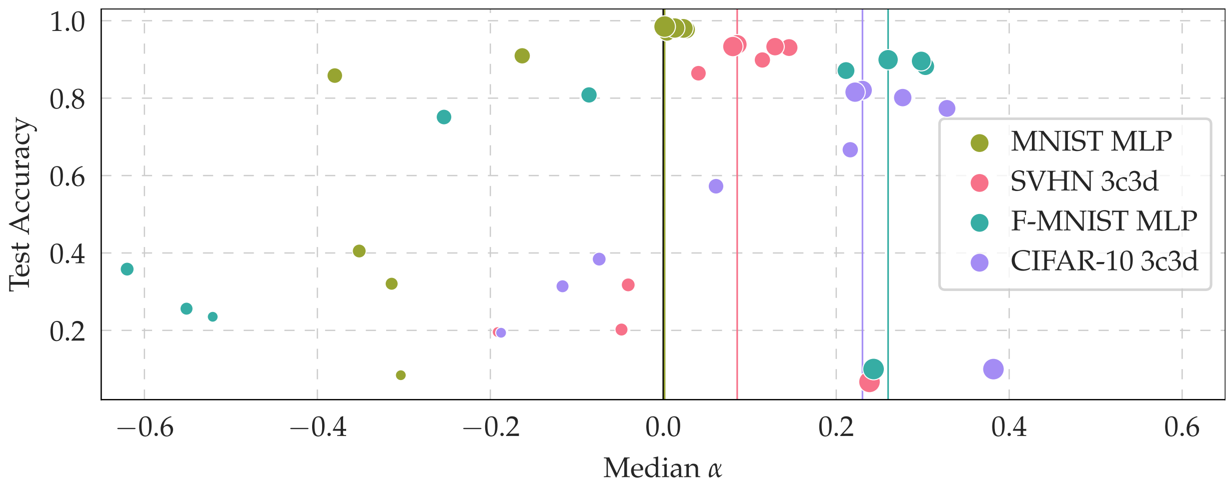


Figure 6.5: Test accuracy as a function of standardized step size α. For four DeepOBS problems (see Appendix C.5), final test accuracy is shown versus the median α-value over the entire training. Marker size indicates the magnitude of the raw learning rate, marker color identifies tasks (see legend). For each problem, the best-performing setting is highlighted by a vertical colored line.

6.3.3 Tuning Learning Rates

Once the architecture is defined, the optimizer's learning rate is the most important hyperparameter to tune. Getting it right requires extensive hyperparameter searches at high resource costs. Cockpit's instruments can provide intuition and information to streamline this process: in contrast to the raw learning rate, the curvature-standardized step size α-quantity (see Section 6.2.1) has a natural scale.

Across multiple optimization problems, we observe, perhaps surprisingly, that the best runs and indeed all good runs have a median $\alpha > 0$ (Figure 6.5). This illustrates a fundamental difference between stochastic optimization, as is typical for machine learning, and classic deterministic optimization. Instead of locally stepping "to the valley floor" (optimal in the deterministic case), stochastic optimizers should *overshoot* the valley somewhat. This need to "surf the walls" has been hypothesized before [e.g. 173, 176] as a property of neural network training. Frequently, learning rates are adapted during training, which fits with our observation about positive α-values: "overshooting" allows fast early progression towards areas of lower loss, but it does not yield convergence in the end. Real-time visualizations of the training state, as offered by Cockpit, can augment these fine-tuning processes.

Figure 6.5 also indicates a major challenge preventing simple automated tuning solutions: the optimal α-value is problem-dependent, and simpler problems, such as a multi-layer perceptron (MLP) on MNIST [95], behave much more similar to classic optimization problems. Algorithmic research on small problems can thus produce misleading conclusions. The figure also shows that the α-gauge is not sufficient by itself: extreme overshooting with a too-large learning rate leads to poor performance, which however can be prevented by taking additional instruments into account. This makes the case for the cockpit metaphor of increasing interpretability from several instruments in conjunction. By combining the α-instrument with other gauges that capture the local geometry or network dynamics, the user can better identify good choices of the learning rate and other hyperparameters.

6.4 Showcase

Having introduced the tool, we can now return to Figure 6.2 for a closer look. The figure shows a snapshot from training the All-CNN-C [157] on CIFAR-100 using SGD with a cyclic learning rate schedule (bottom left panel). Diagonal curvature instruments are configured to use an MC approximation to save run time (here, $C = 100$, compare Section 6.5).

A glance at all panels shows that the learning rate schedule is reflected in the metrics. However, the instruments also provide insights into the early phase of training (first ~ 100 iterations), where the learning rate is still unaffected by the schedule: there, the loss plateaus and the optimizer takes relatively small steps (compared to later, as can be seen in the small gradient norms, and small distance from initialization). Based on these low-cost instruments, one may thus at first suspect that training was poorly initialized; but training indeed succeeds after iteration 100! Viewing Cockpit entirely though, it becomes clear that optimization in these first steps is not stuck at all: while loss, gradient norms, and distance in parameter space remain almost constant, curvature changes, which expresses itself in a clear downward trend of the maximum Hessian eigenvalue (top right panel).

The importance of early training phases has recently been hypothesized [51], suggesting a logarithmic timeline. Not only does our showcase support this hypothesis, but it also provides an explanation from the curvature-based metrics, which in this particular case are the only meaningful feedback in the first few training steps. It also suggests monitoring training at log-spaced intervals. Cockpit provides the flexibility to do so, indeed, Figure 6.2 has been created with log-scheduled tracking events.

As a final note, we recognize that the approach taken here promotes an amount of *manual* work (monitoring metrics, deliberately intervening, *etc.*) that may seem ironic and at odds with the paradigm of automation that is at the heart of machine learning. However, we argue that this might be what is needed at this point in the evolution of the field. Deep learning has been driven notably by scaling compute resources [163], and fully automated one-shot training may still be some way out. To develop better training methods, researchers, not just users, need *algorithmic* interpretability and explainability: direct insights and intuition about the processes taking place "inside" neural nets. To highlight how Cockpit might provide this, we contrast in Appendix C.6 the view of two convex DeepOBS problems (noisy quadratic & logistic regression on MNIST). In both cases, the instruments behave differently compared to the deep learning problem in Figure 6.2. In particular, the gradient norm increases (left column, bottom panel) during training, and individual gradients become less scattered (center column, top panel). This is diametrically opposed to the convex problems and shows that deep learning differs even qualitatively from well-understood optimization problems.

6.5 Benchmark

Section 6.3 made a case for Cockpit as an effective debugging and tuning tool. To make the library useful in practice, it must also have limited computational cost. We now show that it is possible to compute all quantities at reasonable overhead. The user can control the absolute cost along two dimensions, by reducing the number of instruments, or by reducing their update frequency.

All benchmark results show SGD without momentum. Cockpit's quantities, however, work for generic optimizers and can mostly be used identically without increased costs. One current exception is `Alpha` which can be computed more efficiently given the update rule.[2]

2: This is currently implemented for vanilla SGD. Otherwise, Cockpit falls back to a less efficient scheme.

Complexity Analysis

Computing more information adds computational overhead, of course. However, recent work [38] has shown that first-order information, like distributional statistics on the batch gradients, can be computed on top of the mean gradient at little extra cost. Similar savings apply for most quantities in Table 6.1, as they are (non-)linear transformations of individual gradients. A subset of Cockpit's quantities also uses second-order information from the Hessian diagonal. For ReLU networks on a classification task with C classes, the additional work is proportional to C gradient backpropagations (*i.e.* $C = 10$ for CIFAR-10, $C = 100$ for CIFAR-100). Parallel processing can, to some extent, process these extra backpropagations in parallel without significant overhead. If this is no longer possible, we can fall back to a Monte Carlo (MC) sampling approximation, which reduces the number of extra backprop passes to the number of samples (1 by default).[3]

3: An MC-sampled approximation of the Hessian/generalized Gauss-Newton has been used in Figure 6.2 to reduce the prohibitively large number of extra backprops on CIFAR-100 ($C = 100$).

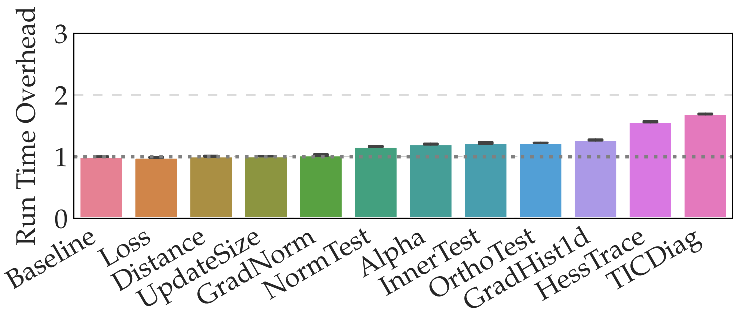
While parallelization is possible for the gradient instruments, computing the maximum Hessian eigenvalue is inherently sequential. Similar to Yao et al. [177], we use matrix-free Hessian-vector products by automatic differentiation [127], where each product's costs are proportional to one gradient computation. Regardless of the underlying iterative eigensolver, multiple such products must be queried to compute the spectral norm (the number depends on the spectral gap to the second-largest eigenvalue).

Run Time Benchmark

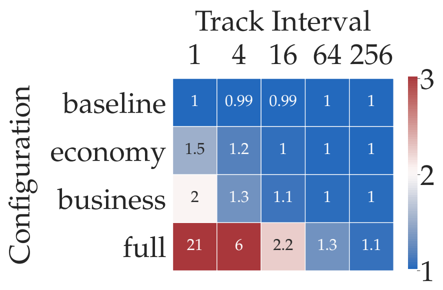
Figure 6.6a shows the wall-clock computational overhead for individual instruments (details in Appendix C.5).[4] As expected, byproducts are virtually free, and quantities that rely solely on first-order information add little overhead (at most roughly 25 % on this problem). Thanks to parallelization, the ten extra backward passes required for Hessian quantities reduce to less than 100 % overhead. Individual overheads also do not simply add up when multiple quantities are tracked, because quantities relying on the same information share computations.

4: To improve readability, we exclude `HessMaxEV` here, because its overhead is large compared to other quantities. Surprisingly, we also observed significant cost for the 2D histogram on GPU. It is caused by an implementation bottleneck for histogram shapes observed in deep models. We thus also omit `GradHist2d` here, as we expect it to be eliminated with future implementations (see Appendix C.5.2 for a detailed analysis and further benchmarks). Both quantities, however, are part of the benchmark shown in Figure 6.6b.

To allow a rough cost control, Cockpit currently offers three configurations, called `"economy"`, `"business"`, and `"full"`, in increasing order of cost (Table 6.1). As a basic guideline, we consider a factor of two to be an acceptable limit for the increase in training time and benchmark the configurations' run times for different tracking intervals. Figure 6.6b shows a run time matrix for the CIFAR-10 3c3d problem, where settings that meet this limit are set in blue (more problems including ImageNet



(a) Overhead Cockpit instruments



(b) Overhead Cockpit configurations

Figure 6.6: Run time overhead for individual Cockpit instruments and configurations as shown on CIFAR-10 3c3d on a GPU. **(a)** The run time overheads for individual instruments are shown as multiples of the *baseline* (no tracking). Most instruments add little overhead. This plot shows the overhead in one iteration, determined by averaging over multiple iterations and random seeds. **(b)** Overhead for different Cockpit configurations. Adjusting the tracking interval and re-using the computation shared by multiple instruments can make the overhead orders of magnitude smaller. Blue fields mark settings that allow tracking without doubling the training time.

are shown in Appendix C.5). Speedups due to shared computations are easy to read off: summing all the individual overheads shown in Figure 6.6a would result in a total overhead larger than 200 %, while the joint overhead (*business*) reduces to 140 %. The *economy* configuration can easily be tracked at every step of this problem and stay well below our threshold of doubling the execution time. Cockpit's full view, shown in Figure 6.2, can be updated every 64-th iteration without a major increase in training time (this corresponds to about five updates per epoch). Finally, tracking any configuration about once per epoch—which is common in practice—adds overhead close to zero (rightmost column).

This good performance is largely due to the efficiency of the BackPACK package [38], which we leverage with custom and optimized modification, that compacts information layer-wise and then discards unneeded buffers. Using layer-wise information (Section 6.3.2) scales better to large networks, where storing the entire model's individual gradients all at once becomes increasingly expensive (see Appendix C.5). To the best of our knowledge, many of the quantities in Table 6.1, especially those relying on individual gradients, have only been explored on rather small problems. With Cockpit they can now be accessed at a reasonable rate for deep learning models outside the toy problem category.

6.6 Conclusion

Contemporary machine learning, in particular deep learning, remains a craft and an art. High dimensionality, stochasticity, and non-convexity require constant tracking and tuning, often resulting in a painful process of trial and error. When things fail, popular performance measures, like the training loss, do not provide enough information by themselves. These metrics only tell *whether* the model is learning, but not *why*. Alternatively, traditional debugging tools can provide access to individual weights and data. However, in models whose power only arises from possessing myriad weights, this approach is hopeless, like looking for the proverbial needle in a haystack.

To mitigate this, we proposed Cockpit, a practical visual debugging tool for deep learning. It offers instruments to monitor the network's internal dynamics during training, in real-time. In its presentation, we focused on two crucial factors affecting user experience: Firstly, such a debugger must provide meaningful insights. To demonstrate Cockpit's utility, we showed how it can identify bugs where traditional tools fail. Secondly, it must come at a feasible computational cost. Although Cockpit uses rich second-order information, efficient computation keeps the necessary run time overhead cheap. The open-source PyTorch package can be added to many existing training loops.

Obviously, such a tool is never complete. Just like there is no perfect universal debugger, the list of current instruments is naturally incomplete. Further practical experience with the tool, for example in the form of a future larger user study, could provide additional evidence for its utility. However, our analysis shows that Cockpit provides useful tools and extracts valuable information presently not accessible to the user. We believe that this improves algorithmic interpretability – helping practitioners understand how to make their models work – but may also inspire new research. The code is designed flexibly, deliberately separating the computation and visualization. New instruments can be added easily and also be shown by the user's preferred visualization tool, *e.g.* TensorBoard. Of course, instead of just showing the data, the same information can be used by novel algorithms directly, side-stepping the human in the loop.

Acknowledgments

The authors gratefully acknowledge financial support by the European Research Council through ERC StG Action 757275 / PANAMA; the DFG Cluster of Excellence "Machine Learning - New Perspectives for Science", EXC 2064/1, project number 390727645; the German Federal Ministry of Education and Research (BMBF) through the Tübingen AI Center (FKZ: 01IS18039A); and funds from the Cyber Valley Initiative of the Ministry for Science, Research and Arts of the State of Baden-Württemberg. Moreover, the authors thank the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for supporting Felix Dangel and Frank Schneider. Further, we are grateful to Agustinus Kristiadi, Alexandra Gessner, Christian Fröhlich, Filip de Roos, Jonathan Wenger, Julia Grosse, Lukas Tatzel, Marius Hobbhahn, and Nicholas Krämer for providing feedback to the manuscript.

ViViT: Curvature Access Through the Generalized Gauss-Newton's Low-rank Structure 7.

Abstract

Curvature in form of the Hessian or its generalized Gauss-Newton (GGN) approximation is valuable for algorithms that rely on a local model for the loss to train, compress, or explain deep networks. Existing methods based on implicit multiplication via automatic differentiation or Kronecker-factored block diagonal approximations do not consider noise in the mini-batch. We present ViViT, a curvature model that leverages the GGN's low-rank structure without further approximations. It allows for efficient computation of eigenvalues, eigenvectors, as well as per-sample first- and second-order directional derivatives. The representation is computed in parallel with gradients in one backward pass and offers a fine-grained cost-accuracy trade-off, which allows it to scale. We demonstrate this by conducting performance benchmarks and substantiate ViViT's usefulness by studying the impact of noise on the GGN's structural properties during neural network training.

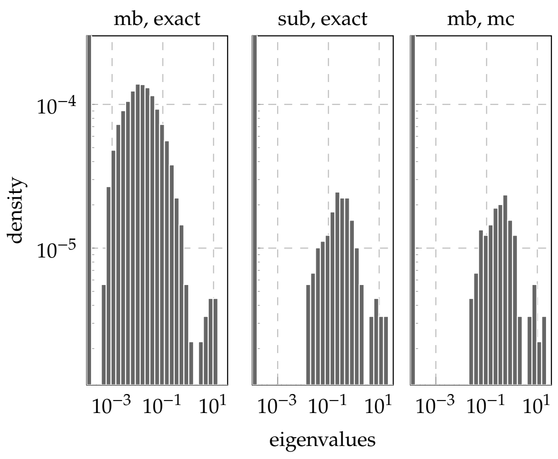
Code and experiments available at the Github repositories `f-dangel/vivit`, `f-dangel/vivit-experiments`

7.1 Introduction & Motivation

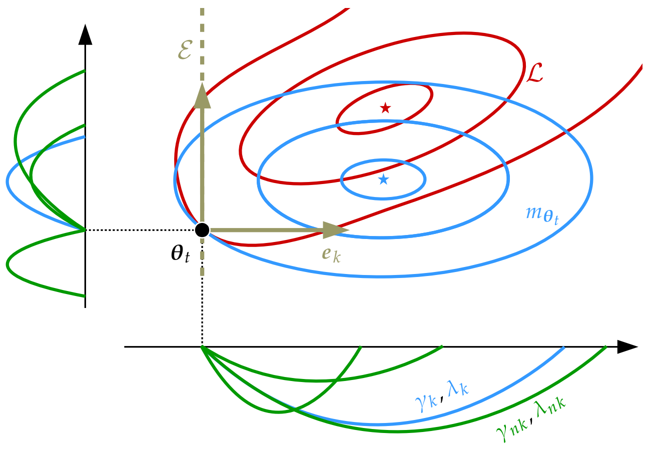
The large number of trainable parameters in deep neural networks imposes computational constraints on the information that can be made available to optimization algorithms. Standard machine learning libraries [1, 126] mainly provide access to first-order information in the form of *average* mini-batch gradients. This is a limitation that complicates the development of novel methods that may outperform the state-of-the-art: They must use the same objects to remain easy to implement and use, and to rely on the highly optimized code of those libraries. There is evidence that this has led to stagnation in the performance of first-order optimizers [145]. Here, we thus study how to provide efficient access to richer information, namely higher-order derivatives and their distribution across the mini-batch.

Recent advances in AD [23, 38] have made such information more readily accessible through leveraging algebraic structure in the differentiated loss. We use and extend this functionality to efficiently access curvature in form of the Hessian's generalized Gauss-Newton (GGN) approximation. It offers practical advantages over the Hessian and is established for training [106, 109], compressing [154], or adding uncertainty to [89, 133, 134] neural networks. It is also linked theoretically to the natural gradient method [5] via the Fisher information matrix [107, Section 9.2].

Traditional ways to access curvature fall into two categories. Firstly, repeated automatic differentiation allows for matrix-free exact multiplication with the Hessian [127] and GGN [148]. Iterative linear and eigensolvers can leverage such functionality to compute Newton steps [53, 106, 183] and spectral properties [3, 55, 61, 123, 139, 140, 177] on



(a) Spectral densities



(b) Per-sample directional derivatives

Figure 7.1: Overview of ViViT's quantities: (a) GGN eigenvalue distribution of DeepOBS' 3c3d architecture on CIFAR-10 [146] for settings with different costs on a mini-batch of size $N = 128$. From left to right: exact GGN, exact GGN on a mini-batch fraction, MC approximation of the GGN. **(b)** Pictorial illustration: loss function $\mathcal{L}$ from Equation (7.1), quadratic model m_{θ_t} around $\boldsymbol{\theta}_t \in \mathbb{R}^2$ from Equation (7.6) (both represented by their contour lines). The low-rank structure provides efficient access to the GGN's eigenvectors $\{e_k\}$, along which m_{θ_t} decouples into one-dimensional parabolas characterized by the directional derivatives γ_k, λ_k and per-sample contributions $\gamma_{n,k}, \lambda_{n,k}$ (Equation (7.8)). $\mathcal{E}$ is the GGN's top-1 eigenspace.

arbitrary architectures thanks to the generality of AD. However, repeated matrix-vector products are potentially detrimental to performance.

Secondly, K-FAC (Kronecker-factored approximate curvature) [21, 63, 108, 109] constructs an explicit light-weight representation of the GGN based on its algebraic Kronecker structure. The computations are streamlined via gradient backpropagation and the resulting matrices are cheap to store and invert. This allows K-FAC to scale: It has been used successfully with large mini-batches [121]. One reason for this efficiency is that K-FAC only approximates the GGN's block diagonal, neglecting interactions across layers. Such terms could be useful, however, for applications like uncertainty quantification with Laplace approximations [40, 89, 133, 134] that currently rely on K-FAC. Moreover, due to its specific design for optimization, the Kronecker representation does not become more accurate with more data. It remains a simplification, exact only under assumptions unlikely to be met in practice [109]. This might be a downside for applications that depend on a precise curvature proxy.

Here, we propose ViViT (inspired by $\boldsymbol{V}\boldsymbol{V}^\top$ in Equation (7.3)), a curvature model that leverages the GGN's low-rank structure. Like K-FAC, its representation is computed in parallel with gradients. But it allows a cost-accuracy trade-off, ranging from the *exact* GGN to an approximation that costs a single gradient computation. Our contributions are:

- We highlight the GGN's low-rank structure, and the structural limit for the inherent curvature information contained in a mini-batch.
- We present how to compute various GGN properties efficiently by exploiting this structure (Figure 7.1): The exact eigenvalues, eigenvectors, and per-sample directional derivatives. In contrast to other methods, these quantities allow modeling curvature noise.
- We introduce approximations that allow a flexible trade-off between computational cost and accuracy. We also provide a fully-featured efficient implementation in PyTorch [126] on top of BackPACK [38].

- We empirically demonstrate scalability and efficiency of leveraging the GGN's low-rank structure through benchmarks on different deep neural network architectures. Finally, we use ViViT's quantities to study the GGN, and how it is affected by noise, during training.

The main focus is demonstrating that many interesting curvature properties, including uncertainty, can be computed efficiently. Practical applications of this curvature uncertainty are discussed in Section 7.5.

7.2 Notation & Method

Consider a model $f_{\boldsymbol{\theta}} : \mathbb{X} \to \mathbb{Y}$ and a dataset $\{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X} \times \mathbb{Y}\}_{n=1}^N$. For simplicity we use N for both the mini-batch and training set size. The network, parameterized by $\boldsymbol{\theta} \in \Theta$, maps a sample $\boldsymbol{x}_n$ to a prediction $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n) \in \mathbb{F}$. Predictions are scored by a convex loss function $\ell : \mathbb{F} \times \mathbb{Y} \to \mathbb{R}$ (*e.g.* cross-entropy or square loss), which compares to the ground truth $\boldsymbol{y}_n$. The training objective $\mathcal{L} : \Theta \to \mathbb{R}$ is the empirical risk

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n) \,. \tag{7.1}$$

We use $\ell_n(\boldsymbol{\theta}) = \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)$ and $\boldsymbol{f}_n(\boldsymbol{\theta}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ for per-sample losses and predictions. For gradients, we write $\boldsymbol{g}_n(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})$ and $\boldsymbol{g}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$, suppressing $\boldsymbol{\theta}$ if unambiguous. We also set $\Theta = \mathbb{R}^D$ and $\mathbb{F} = \mathbb{R}^C$ with D, C the model parameter and prediction space dimension, respectively. For classification, C is the number of classes.

Hessian & GGN

Two-fold chain rule application to the split $\ell \circ f$ decomposes the Hessian of Equation (7.1) into two parts $\nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \boldsymbol{G}(\boldsymbol{\theta}) + \boldsymbol{R}(\boldsymbol{\theta}) \in \mathbb{R}^{D \times D}$; the positive semi-definite GGN

$$\boldsymbol{G}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \left(\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n\right)^\top \nabla^2_{\boldsymbol{f}_n} \ell_n \left(\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n\right) = \frac{1}{N} \sum_{n=1}^N \boldsymbol{G}_n(\boldsymbol{\theta}) \tag{7.2}$$

and a residual $\boldsymbol{R} = {}^1\!/_N \sum_{n=1}^N \sum_{c=1}^C (\nabla^2_{\boldsymbol{\theta}} [\boldsymbol{f}_n]_c) [\nabla_{\boldsymbol{f}_n} \ell_n]_c$. Here, we use the Jacobian $\mathrm{J}_{\boldsymbol{a}} \boldsymbol{b}$ that contains partial derivatives of $\boldsymbol{b}$ *w.r.t.* $\boldsymbol{a}$, $[\mathrm{J}_{\boldsymbol{a}} \boldsymbol{b}]_{i,j} = \partial b_i / \partial a_j$ (Definition 2.4. As the residual may alter the Hessian's definiteness—undesirable in many applications—we focus on the GGN. Section 7.3.2 provides empirical evidence that the curvature's top eigenspace is largely unaffected by this simplification.

Low-rank Structure

By basic inequalities, Equation (7.2) has $\operatorname{rank}(\boldsymbol{G}) \le NC$.[1] To make this explicit, we factorize the positive semi-definite Hessian $\nabla^2_{\boldsymbol{f}_n} \ell_n = \sum_{c=1}^C \boldsymbol{s}_{n,c} \boldsymbol{s}_{n,c}^\top$, where $\boldsymbol{s}_{n,c} \in \mathbb{R}^C$ and denote its backpropagated version by $\boldsymbol{v}_{n,c} = (\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top \boldsymbol{s}_{n,c} \in \mathbb{R}^D$. Absorbing sums into matrix multiplications,

1: We assume the overparameterized deep learning setting ($NC < D$) and suppress the trivial rank bound D.

we arrive at the GGN's outer product representation that lies at the heart of the ViViT concept,

$$\boldsymbol{G} = \frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} \boldsymbol{v}_{n,c}\boldsymbol{v}_{n,c}^\top = \boldsymbol{V}\boldsymbol{V}^\top \tag{7.3}$$

with $\boldsymbol{V} = {}^1\!/\!\sqrt{N}\begin{pmatrix}\boldsymbol{v}_{1,1} & \boldsymbol{v}_{1,2} & \dots & \boldsymbol{v}_{N,C}\end{pmatrix} \in \mathbb{R}^{D\times NC}$. $\boldsymbol{V}$ allows for *exact* computations with the explicit GGN matrix, at linear rather than quadratic memory cost in D. We first formulate the extraction of relevant GGN properties from this factorization, before addressing how to further approximate $\boldsymbol{V}$ to reduce memory and computation costs.

7.2.1 Computing the Full GGN Eigenspectrum

Each GGN eigenvalue $\lambda \in \mathbb{R}$ is a root of the characteristic polynomial $\det(\boldsymbol{G} - \lambda\boldsymbol{I}_D)$ with identity matrix $\boldsymbol{I}_D \in \mathbb{R}^{D\times D}$. Leveraging the factorization of Equation (7.3) and the matrix determinant lemma, the D-dimensional eigenproblem reduces to that of the much smaller Gram matrix $\tilde{\boldsymbol{G}} = \boldsymbol{V}^\top\boldsymbol{V} \in \mathbb{R}^{NC\times NC}$ which contains pairwise scalar products of $\boldsymbol{v}_{n,c}$ (see Appendix D.1.1),

$$\det(\boldsymbol{G} - \lambda\boldsymbol{I}_D) = 0 \quad \Leftrightarrow \quad \det(\tilde{\boldsymbol{G}} - \lambda\boldsymbol{I}_{NC}) = 0\,. \tag{7.4}$$

With at least $D - NC$ trivial solutions, the GGN curvature is zero along most directions in parameter space. Nontrivial solutions that give rise to curved directions are fully-contained in the Gram matrix, and hence *much* cheaper to compute.

Despite various Hessian spectral studies which rely on iterative eigensolvers and implicit matrix multiplication [3, 55, 61, 123, 139, 140, 177], we are not aware of works that efficiently extract the *exact* GGN spectrum from its Gram matrix. In contrast to those techniques, this matrix can be computed in parallel with gradients in a single backward pass, which results in less sequential overhead. We demonstrate in Section 7.3.1 that exploiting the low-rank structure for computing the leading eigenpairs is superior to a power iteration based on matrix-free multiplication in terms of run time.

Eigenvalues themselves can help identify reasonable hyperparameters, like learning rates [97]. But we can also reconstruct the associated eigenvectors. These are directions along which curvature information is contained in the mini-batch. Let $\tilde{\mathbb{S}}_+ = \{(\lambda_k, \tilde{\boldsymbol{e}}_k) \mid \lambda_k \neq 0, \tilde{\boldsymbol{G}}\tilde{\boldsymbol{e}}_k = \lambda_k\tilde{\boldsymbol{e}}_k\}_{k=1}^K$ denote the nontrivial Gram spectrum[2] with orthonormal eigenvectors $\tilde{\boldsymbol{e}}_j^\top\tilde{\boldsymbol{e}}_k = \delta_{j,k}$ (δ represents the Kronecker delta and $K = \operatorname{rank}(\boldsymbol{G})$). Then, the transformed vectors $\boldsymbol{e}_k = {}^1\!/\!\sqrt{\lambda_k}\boldsymbol{V}\tilde{\boldsymbol{e}}_k$ ($k = 1, \dots, K$) are orthonormal eigenvectors of $\boldsymbol{G}$ associated to eigenvalues λ_k (see Appendix D.1.2), *i.e.* for all $(\lambda_k, \tilde{\boldsymbol{e}}_k) \in \tilde{\mathbb{S}}_+$

2: In the following, we assume ordered eigenvalues, *i.e.* $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_K$, for convenience.

$$\tilde{\boldsymbol{G}}\tilde{\boldsymbol{e}}_k = \lambda_k\tilde{\boldsymbol{e}}_k \quad \Longrightarrow \quad \boldsymbol{G}\boldsymbol{e}_k = \lambda_k\boldsymbol{e}_k\,. \tag{7.5}$$

The eigenspectrum also provides access to the GGN's pseudo-inverse based on $\boldsymbol{V}$ and $\tilde{\mathbb{S}}_+$, required by *e.g.* second-order methods.[3]

3: Appendix D.3.2 describes implicit multiplication with $\boldsymbol{G}^{-1}$.

7.2.2 Computing Directional Derivatives

Various algorithms rely on a local quadratic approximation of the loss landscape. For instance, optimization methods adapt their parameters by stepping into the minimum of the local proxy. Curvature, in the form of the Hessian or GGN, allows to build a quadratic model given by the Taylor expansion. Let $m_{\boldsymbol{\theta}_t}$ denote the quadratic model for the loss around position $\boldsymbol{\theta}_t \in \Theta$ that uses curvature represented by the GGN,

$$m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta}) = \text{const} + (\boldsymbol{\theta} - \boldsymbol{\theta}_t)^\top \boldsymbol{g}(\boldsymbol{\theta}_t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^\top \boldsymbol{G}(\boldsymbol{\theta}_t)(\boldsymbol{\theta} - \boldsymbol{\theta}_t)\,. \tag{7.6}$$

At its base point $\boldsymbol{\theta}_t$, the shape of $m_{\boldsymbol{\theta}_t}$ along an arbitrary normalized direction $\boldsymbol{e} \in \Theta$ (*i.e.* $\|\boldsymbol{e}\|_2 = 1$) is determined by the local gradient and curvature. Specifically, the projection of Equation (7.6) onto $\boldsymbol{e}$ gives rise to the (scalar) first-and second-order directional derivatives

$$\gamma_{\boldsymbol{e}} = \boldsymbol{e}^\top \nabla_{\boldsymbol{\theta}} m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta}_t) \quad = \boldsymbol{e}^\top \boldsymbol{g}(\boldsymbol{\theta}_t) \quad \in \mathbb{R}\,, \tag{7.7a}$$

$$\lambda_{\boldsymbol{e}} = \boldsymbol{e}^\top \nabla^2_{\boldsymbol{\theta}} m_{\boldsymbol{\theta}_t}(\boldsymbol{\theta}_t)\, \boldsymbol{e} = \boldsymbol{e}^\top \boldsymbol{G}(\boldsymbol{\theta}_t)\, \boldsymbol{e} \in \mathbb{R}\,. \tag{7.7b}$$

As $\boldsymbol{G}$'s characteristic directions are its eigenvectors, they form a natural basis for the quadratic model. Denoting $\gamma_k = \gamma_{\boldsymbol{e}_k}$ and $\lambda_k = \lambda_{\boldsymbol{e}_k}$ the directional gradient and curvature along eigenvector $\boldsymbol{e}_k$, we see from Equation (7.7b) that the directional curvature indeed coincides with the GGN's eigenvalue.

Analogous to the gradient and GGN, the directional derivatives γ_k and λ_k inherit the sum structure of the loss function from Equation (7.1), *i.e.* they decompose into contributions from individual samples. Let $\gamma_{n,k}$ and $\lambda_{n,k}$ denote these first- and second-order derivatives contributions of sample $\boldsymbol{x}_n$ in direction k, *i.e.*

$$\gamma_{n,k} = \boldsymbol{e}_k^\top \boldsymbol{g}_n = \frac{\tilde{\boldsymbol{e}}_k^\top \boldsymbol{V}^\top \boldsymbol{g}_n}{\sqrt{\lambda_k}}\,, \tag{7.8a}$$

$$\lambda_{n,k} = \boldsymbol{e}_k^\top \boldsymbol{G}_n \boldsymbol{e}_k = \frac{\|\boldsymbol{V}_n^\top \boldsymbol{V} \tilde{\boldsymbol{e}}_k\|_2^2}{\lambda_k}\,, \tag{7.8b}$$

where $\boldsymbol{V}_n \in \mathbb{R}^{D \times C}$ is a scaled sub-matrix of $\boldsymbol{V}$ with fixed sample index. Note that directional derivatives can be evaluated efficiently with the Gram matrix eigenvectors without explicit access to the associated directions in parameter space.

In Equation (7.7), gradient $\boldsymbol{g}$ and curvature $\boldsymbol{G}$ are sums over $\boldsymbol{g}_n$ and $\boldsymbol{G}_n$, respectively, from which follows the relationship between directional derivatives and per-sample contributions $\gamma_k = {}^1\!/_N \sum_{n=1}^N \gamma_{n,k}$ and $\lambda_k = {}^1\!/_N \sum_{n=1}^N \lambda_{n,k}$. Figure 7.1b shows a pictorial view of the quantities provided by ViViT.

Access to per-sample directional gradients $\gamma_{n,k}$ and curvatures $\lambda_{n,k}$ along $\boldsymbol{G}$'s natural directions is a distinct feature of ViViT. They provide geometric information about the local loss landscape *as well as* about the model's directional curvature stochasticity over the mini-batch.

7.2.3 Computational Complexity

So far, we have formulated the computation of the GGN's eigenvalues (Equation (7.4)), eigenvectors (Equation (7.5)), and per-sample directional derivatives (Equation (7.8)). Now, we analyze their computational complexity in more detail to identify critical performance factors. Those limitations can effectively be addressed with approximations that allow the costs to be decreased in a fine-grained fashion. We substantiate our theoretical analysis with empirical performance measurements in Section 7.3.1.

Relation to Gradient Computation

Machine learning libraries are optimized to backpropagate signals $1/N \nabla_{f_n} \ell_n$ and accumulate the result into the mini-batch gradient $g = 1/N \sum_{n=1}^{N} [\mathrm{J}_{\theta} f_n]^\top \nabla_{f_n} \ell_n$. Each column $v_{n,c}$ of V also involves applying the Jacobian, but to a different vector $s_{n,c}$ from the loss Hessian's symmetric factorization. For popular loss functions, like square and cross-entropy loss, this factorization is analytically known and available at negligible overhead. Hence, computing V basically costs C gradient computations as it involves NC backpropagations, while the gradient requires N. However, the practical overhead is expected to be smaller: computations can re-use information from BackPACK's vectorized Jacobians and enjoy additional speedup on parallel processors like GPUs.

Stage-wise Discarding V

V's columns correspond to backpropagated vectors. During backpropagation, sub-matrices of V, associated to parameters in the current layer, become available once at a time and can be discarded immediately after their use. This allows for memory savings without any approximations.

One example is the Gram matrix $\tilde{G}$ formed by pairwise scalar products of $\{v_{n,c}\}_{n=1,c=1}^{N,C}$ in $\mathcal{O}((NC)^2 D)$ operations. The spectral decomposition $\tilde{\mathbb{S}}_+$ has additional cost of $\mathcal{O}((NC)^3)$. Similarly, the terms for the directional derivatives in Equation (7.8) can be built up stage-wise: first-order derivatives $\{\gamma_{n,k}\}_{n=1,k=1}^{N,K}$ require the vectors $\{V^\top g_n \in \mathbb{R}^{NC}\}_{n=1}^{N}$ that cost $\mathcal{O}(N^2 CD)$ operations. Second-order derivatives are basically for free, as $\{V_n^\top V \in \mathbb{R}^{C \times NC}\}_{n=1}^{N}$ is available from $\tilde{G}$.

GGN Eigenvectors

Transforming an eigenvector $\tilde{e}_k$ of the Gram matrix to the GGN eigenvector e_k through application of V (Equation (7.5)) costs $\mathcal{O}(NCD)$ operations. However, repeated application of V can be avoided for sums of the form $\sum_k (c_k/\sqrt{\lambda_k}) e_k$ with arbitrary weights $c_k \in \mathbb{R}$. The summation can be performed in the Gram space at negligible overhead, and only the resulting vector $\sum_k c_k \tilde{e}_k$ needs to be transformed. For a practical example – computing damped Newton steps – see Appendix D.2.1.

7.2.4 Approximations & Implementation

Although the GGN's representation by $\boldsymbol{V}$ has linear memory cost in D, it requires memory equivalent to NC model copies.[4] Of course, this is infeasible for many networks and datasets, *e.g.* ImageNet ($C = 1000$). So far, our formulation was concerned with *exact* computations. We now present approximations that allow N, C and D in the above cost analysis to be replaced by smaller numbers, enabling ViViT to trade-off accuracy and performance.

4: Our implementation uses a more memory-efficient approach that avoids expanding $\boldsymbol{V}$ for linear layers by leveraging structure in their Jacobian (see Appendix D.3.1).

MC approximation & Curvature Sub-sampling

To reduce the scaling in C, we can approximate the factorization $\nabla^2_{f_n} \ell_n(\boldsymbol{\theta}) = \sum_{c=1}^{C} \boldsymbol{s}_{n,c} \boldsymbol{s}_{n,c}^\top$ by a smaller set of vectors. One principled approach is to draw MC samples $\{\tilde{\boldsymbol{s}}_{n,m}\}$ with $\mathbb{E}_m[\tilde{\boldsymbol{s}}_{n,m} \tilde{\boldsymbol{s}}_{n,m}^\top] = \nabla^2_{f_n} \ell_n(\boldsymbol{\theta})$ as in [38] or Example 5.2. This reduces the scaling of backpropagated vectors from C to the number of MC samples M ($= 1$ in the following if not specified). A common independent approximation to reduce the scaling in N is computing curvature on a mini-batch subset [25, 183].

Parameter Groups (Block-diagonal Approximation)

Some applications, *e.g.* computing Newton steps, require $\boldsymbol{V}$ to be kept in memory for performing the transformation from Gram space into the parameter space. Still, we can reduce costs by using the GGN's diagonal blocks $\{\boldsymbol{G}^{(l)}\}_{l=1}^{L}$ of each layer, rather than the full matrix $\boldsymbol{G}$. Such blocks are available during backpropagation and can thus be used and discarded step by step. In addition to the previously described approximations for reducing the costs in N and C, this technique tackles scaling in D.

Implementation Details

BackPACK's functionality allows us to efficiently compute individual gradients and $\boldsymbol{V}$ in a single backward pass, using either an exact or MC-factorization of the loss Hessian. To reduce memory consumption, we extend its implementation with a protocol to support mini-batch sub-sampling and parameter groups. By hooks into the package's extensions, we can discard buffers as soon as possible during backpropagation, effectively implementing all discussed approximations and optimizations.

In Section 7.3, we specifically address how the above approximations affect run time and memory requirements, and study their impact on structural properties of the GGN.

7.3 Experiments

For the practical use of the ViViT concept, it is essential that (i) the computations are efficient and (ii) that we gain an understanding of how sub-sampling noise and the approximations introduced in Section 7.2.4 alter the structural properties of the GGN. In the following, we therefore empirically investigate ViViT's scalability and approximation properties in the context of deep learning. The insights from this analysis substantiate ViViT's value as a monitoring tool for deep learning optimization.

Experimental Setting

Architectures include three deep CNNs from DeepOBS [146] (2c2d on Fashion-MNIST, 3c3d on CIFAR-10 and All-CNN-C on CIFAR-100), as well as ResNets from He et al. [68] on CIFAR-10 based on Idelbayev [74]—all architectures use cross-entropy loss. Based on the approximations presented in Section 7.2.4, we distinguish the following cases:

- **mb, exact:** Exact GGN with all mini-batch samples. Backpropagates NC vectors.
- **mb, mc:** MC-approximated GGN with all mini-batch samples. Backpropagates NM vectors with M the number of MC-samples.
- **sub, exact:** Exact GGN on a subset of mini-batch samples ($\lfloor N/8 \rfloor$ as in [183]). Backpropagates $\lfloor N/8 \rfloor C$ vectors.
- **sub, mc:** MC-approximated GGN on a subset of mini-batch samples. Backpropagates $\lfloor N/8 \rfloor M$ vectors with M the number of MC-samples.

7.3.1 Scalability

We now complement the theoretical computational complexity analysis from Section 7.2.3 with empirical studies. Results were generated on a workstation with an Intel Core i7-8700K CPU (32 GB) and one NVIDIA GeForce RTX 2080 Ti GPU (11 GB). We use $M = 1$ in the following.

Memory Performance

We consider two tasks:

1. **Computing eigenvalues:** The nontrivial eigenvalues $\{\lambda_k \mid (\lambda_k, \tilde{e}_k) \in \tilde{\mathbb{S}}_+\}$ are obtained by forming and eigen-decomposing the Gram matrix $\tilde{G}$, allowing stage-wise discarding of V (see Sections 7.2.1 and 7.2.3).
2. **Computing the top eigenpair:** For (λ_1, e_1), we compute the Gram matrix spectrum $\tilde{\mathbb{S}}_+$, extract its top eigenpair $(\lambda_1, \tilde{e}_1)$, and transform it into parameter space by Equation (7.5), *i.e.* $(\lambda_1, e_1 = 1/\sqrt{\lambda_1} V \tilde{e}_1)$. This requires more memory than task 1 as V must be stored.

As a comprehensive memory performance measure, we use the largest batch size before our system runs out of memory—we call this the *critical batch size* N_{crit}.

Figure 7.2a tabularizes the critical batch sizes on GPU for the 3c3d architecture on CIFAR-10. As expected, computing eigenpairs requires more memory and leads to consistently smaller critical batch sizes in comparison to computing only eigenvalues. Yet, they all exceed the traditional batch size used for training ($N = 128$, see [146]), even when using the exact GGN. With ViViT's approximations, the memory overhead can be reduced to significantly increase the applicable batch size.

(a) Memory performance

GGN \ Data	N_{crit} (eigenvalues) mb	N_{crit} (eigenvalues) sub
exact	909	4375
mc	3840	6626

GGN \ Data	N_{crit} (top eigenpair) mb	N_{crit} (top eigenpair) sub
exact	677	3184
mc	3060	6029

(b) Run time performance

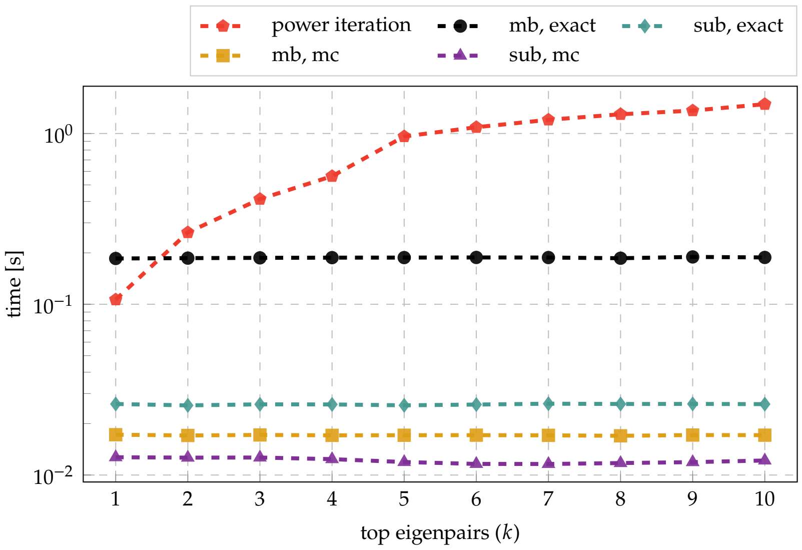


Figure 7.2: GPU memory and run time performance: Performance measurements for the 3c3d architecture ($D = 895{,}210$) on CIFAR-10 ($C = 10$). **(a)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(b)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a batch of size $N = 128$.

We report similar results for more architectures, a block-diagonal approximation (as in Zhang et al. [183]), and on CPU in Appendix D.2.1, where we also benchmark a third task—computing damped Newton steps.

Run Time Performance

Next, we consider computing the k leading eigenvectors and eigenvalues of a matrix. A power iteration that computes eigenpairs iteratively via matrix-vector products serves as a reference. For a fixed value of k, we repeat both approaches 20 times and report the shortest time.

For the power iteration, we adapt the implementation from the PyHessian library [177] and replace its Hessian-vector product by a matrix-free GGN-vector product [148] through PyTorch's AD. We use the same default hyperparameters for the termination criterion. Similar to task 1, our method obtains the top-k eigenpairs[5] by computing $\tilde{\mathbb{S}}_+$, extracting its leading eigenpairs and transforming the eigenvectors $\tilde{\boldsymbol{e}}_1, \tilde{\boldsymbol{e}}_2, \dots, \tilde{\boldsymbol{e}}_k$ into parameter space by application of $\boldsymbol{V}$ (see Equation (7.5)).

5: In contrast to the power iteration that is restricted to dominating eigenpairs, our approach allows choosing arbitrary eigenpairs.

Figure 7.2b shows the GPU run time for the 3c3d architecture on CIFAR-10, using a mini-batch of size $N = 128$. Without any approximations to the GGN, our method already outperforms the power iteration for $k > 1$ and increases *much* slower in run time as more leading eigenpairs are requested. This means that, relative to the transformation of each eigenvector from the Gram space into the parameter space through $\boldsymbol{V}$, the run time mainly results from computing $\boldsymbol{V}, \tilde{\boldsymbol{G}}$, and eigendecomposing the latter. This is consistent with the computational complexity of those operations in NC (compare Section 7.2.3) and allows for efficient extraction of a large number of eigenpairs. The run time curves of the

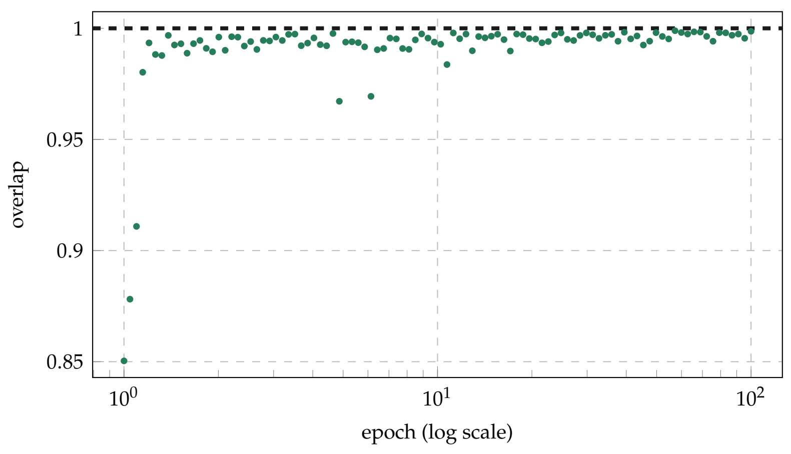


Figure 7.3: Full-batch GGN versus full-batch Hessian: Overlap between the top-C eigenspaces of the full-batch GGN and full-batch Hessian during training of the 3c3d network on CIFAR-10 with SGD.

approximations confirm this behavior by featuring the same flat profile. Additionally, they require significantly less time than the exact mini-batch computation. Results for more network architectures, a block-diagonal approximation and on CPU are reported in Appendix D.2.1.

7.3.2 Approximation Quality

ViViT is based on the Hessian's generalized Gauss-Newton approximation (see Equation (7.2)). In practice, the GGN is only computed on a mini-batch which yields a statistical estimator for the *full-batch* GGN (*i.e.* the GGN evaluated on the entire training set). Additionally, we introduce curvature sub-sampling and an MC approximation (see Section 7.2.4), *i.e.* further approximations that alter the curvature's structural properties. In this section, we compare quantities at different stages within this hierarchy of approximations. We use the test problems from above and train the networks with both SGD and Adam (details in Appendix D.2.2).

GGN Versus Hessian

First, we empirically study the relationship between the GGN and the Hessian in the deep learning context. To capture *solely* the effect of neglecting the residual $\boldsymbol{R}$ (see Equation (7.2)), we consider the noise-free case and compute $\boldsymbol{H}$ and $\boldsymbol{G}$ on the entire training set.

We characterize both curvature matrices by their top-C eigenspace: the space spanned by the eigenvectors to the C largest eigenvalues. This is a C-dimensional subspace of the parameter space Θ, on which the loss function is subject to particularly strong curvature. The *overlap* between these spaces serves as the comparison metric. Let $\{\boldsymbol{e}_c^{\boldsymbol{U}}\}_{c=1}^{C}$ the set of orthonormal eigenvectors to the C largest eigenvalues of some symmetric matrix $\boldsymbol{U}$ and $\mathcal{E}^{\boldsymbol{U}} = \operatorname{span}(\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})$. The projection onto this subspace $\mathcal{E}^{\boldsymbol{U}}$ is given by the projection matrix $\boldsymbol{P}^{\boldsymbol{U}} = (\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})(\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})^\top$. As in Gur-Ari et al. [65], we define the overlap between two top-C eigenspaces $\mathcal{E}^{\boldsymbol{U}}$ and $\mathcal{E}^{\boldsymbol{V}}$ of the matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ by

$$\operatorname{overlap}(\mathcal{E}^{\boldsymbol{U}}, \mathcal{E}^{\boldsymbol{V}}) = \frac{\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{U}} \boldsymbol{P}^{\boldsymbol{V}})}{\sqrt{\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{U}}) \operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{V}})}} \in [0, 1] \,. \tag{7.9}$$

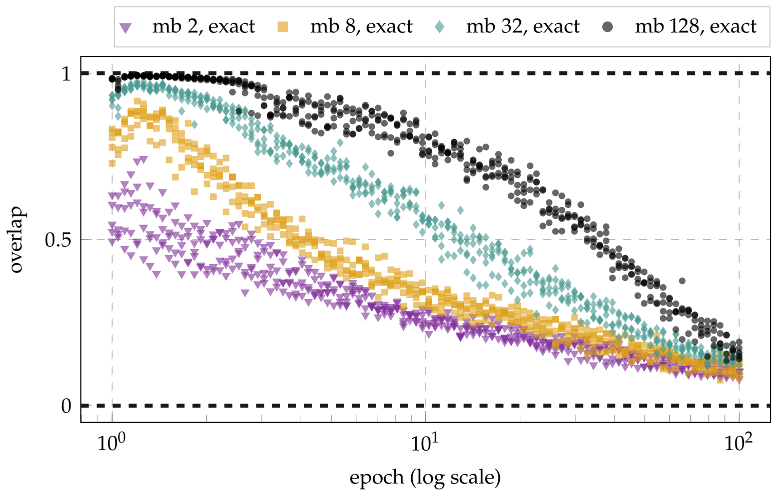


Figure 7.4: Mini-batch GGN versus full-batch GGN: Overlap between the top-C eigenspaces of the mini-batch GGN and full-batch GGN during training of the 3c3d network on CIFAR-10 with SGD. For each mini-batch size, 5 different mini-batches are drawn.

If overlap$(\mathcal{E}^U, \mathcal{E}^V) = 0$, then $\mathcal{E}^U$ and $\mathcal{E}^V$ are orthogonal to each other; if the overlap is 1, the subspaces are identical.

Figure 7.3 shows the overlap between the full-batch GGN and Hessian during training of the 3c3d network on CIFAR-10 with SGD. Except for a short phase at the beginning of the optimization procedure (note the log scale for the epoch-axis), a strong agreement (overlap ≥ 0.85) between the top-C eigenspaces is observed. We make similar observations with the other test problems (see Appendix D.2.3), yet to a slightly lesser extent for CIFAR-100. Consequently, we identify the GGN as an interesting object, since it consistently shares relevant structure with the Hessian matrix.

Eigenspace Under Noise & Approximations

ViViT uses mini-batching to compute a statistical estimator of the full-batch GGN. This approximation alters the top-C eigenspace, as shown in Figure 7.4: with decreasing mini-batch size, the approximation carries less and less structure of its full-batch counterpart, as indicated by dropping overlaps. In addition, at constant batch size, a decrease in approximation quality can be observed over the course of training. This might be a valuable insight for the design of second-order optimization methods, where this structural decay could lead to performance degradation over the course of the optimization, which has to be compensated for by a growing batch-size (*e.g.* Martens [106] reports that the optimal batch size grows during training).

To allow for a fine-grained cost-accuracy trade-off, ViViT introduces *further* approximations to the mini-batch GGN (see Section 7.2.4). Figure 7.5 shows the overlap between these GGN approximations and the full-batch GGN[6]. The order of the approximations is as expected: with increasing computational effort, the approximations improve and, despite the greatly reduced computational effort compared to the exact mini-batch GGN, significant structure of the top-C eigenspace is preserved. Details and results for the other test problems are reported in Appendix D.2.4.

6: A comparison with the mini-batch GGN as ground truth can be found in Appendix D.2.4

So far, our analysis is based on the top-C eigenspace of the curvature matrices. We extend it by studying the effect of noise and approximations on the curvature *magnitude* along the top-C directions in Appendix D.2.5.

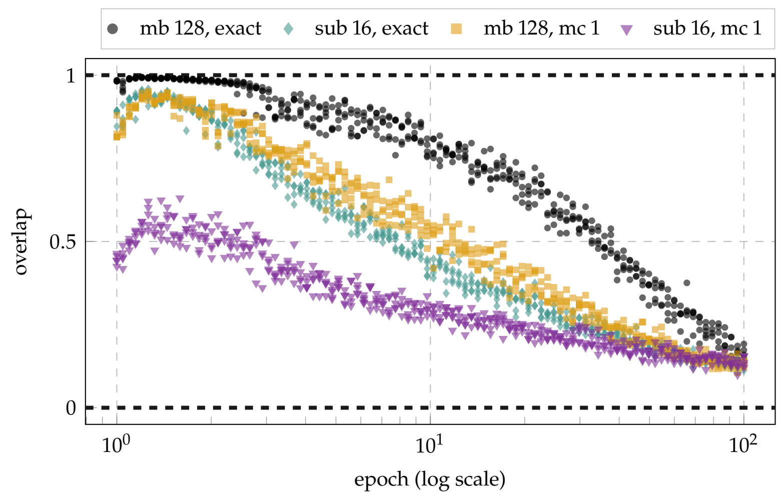


Figure 7.5: Approximations versus full-batch GGN: Overlap between the top-C eigenspaces of the mini-batch GGN, ViViT's approximations and the full-batch GGN during training of the 3c3d network on CIFAR-10 with SGD. Each approximation is evaluated on 5 mini-batches.

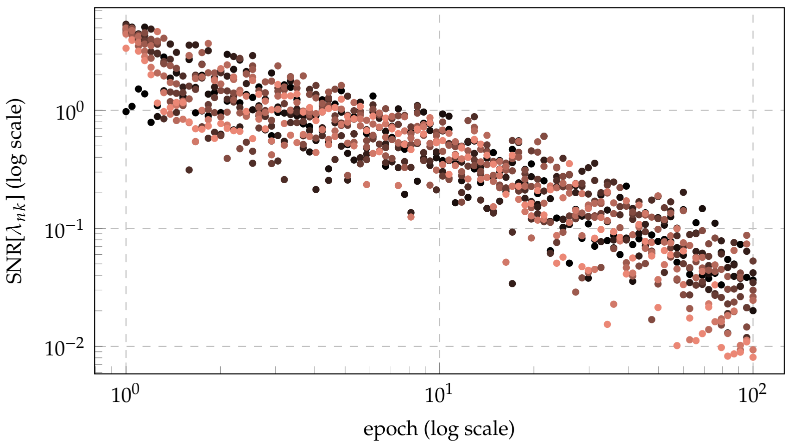


Figure 7.6: Directional curvature SNRs: Curvature SNRs along each of the mini-batch GGN's top-C eigenvectors during training of the 3c3d network on CIFAR-10 with SGD. At fixed epoch, the SNR for the most curved direction is shown in ● and the SNR for the direction with the smallest curvature is shown in ●.

7.3.3 Per-sample Directional Derivatives

A unique feature of ViViT's quantities is that they provide a notion of *curvature uncertainty* through *per-sample* first- and second-order directional derivatives (Equation (7.8)). To quantify noise in these derivatives, we compute their signal-to-noise ratios (SNRs). For each direction $\boldsymbol{e}_k$, the SNR is given by the squared empirical mean divided by the empirical variance of the N mini-batch samples $\{\gamma_{n,k}\}_{n=1}^N$ and $\{\lambda_{n,k}\}_{n=1}^N$, respectively.

Figure 7.6 shows curvature SNRs during training the 3c3d network on CIFAR-10 with SGD. The curvature signal along the top-C eigenvectors decreases from SNR > 1 by two orders of magnitude. In comparison, the directional gradients do not exhibit such a pattern (see Appendix D.2.6). Results for the other test cases can be found in Appendix D.2.6.

In this section, we have given a glimpse of the *very rich* quantities that can be efficiently computed under ViViT's concept. In Section 7.5, we discuss their practical use—curvature uncertainty in particular.

7.4 Related Work

GGN Spectrum & Low-rank Structure

Other works point out the GGN's low-rank structure. Botev et al. [21] present the rank bound (NC) and propose an alternative to K-FAC based on backpropagating a decomposition of the loss Hessian. Papyan [122] presents the factorization in Equation (7.3) and studies the eigenvalue spectrum's hierarchy for cross-entropy loss. In this setting, the GGN further decomposes into summands, some of which are then analyzed through similar Gram matrices. These can be obtained as contractions of $\tilde{\boldsymbol{G}}$, but our approach goes beyond them as it does not neglect terms. We are not aware of works that obtain the exact spectrum *and* leverage a highly-efficient fully-parallel implementation. This may be because, until recently [23, 38], vectorized Jacobians required to perform those operations efficiently were not available.

Efficient Operations with Low-rank Matrices in Deep Learning

Chen et al. [30] use Equation (7.3) for element-wise evaluation of the GGN in FCNNs. They also present a variant based on MC sampling. This element-wise evaluation is then used to construct hierarchical matrix approximations of the GGN. ViViT instead leverages the global low-rank structure that also enables efficient eigendecomposition.

Another prominent low-rank matrix in deep learning is the un-centered gradient covariance (sometimes called empirical Fisher). Singh and Alistarh [154] describe implicit multiplication with its inverse and apply it for neural network compression, assuming the empirical Fisher as a Hessian proxy. However, this assumption has limitations, specifically for optimization [92]. In principle though, the low-rank structure also permits the application of our methods from Section 7.2.

7.5 Use Cases

Aiming to provide a well-founded, theoretical and empirical evaluation, we have consciously focused on studying the approximation quality of ViViT's quantities, as well as on demonstrating the efficiency of their computation. We believe it is interesting in itself that the low-rank structure provides access to quantities that would otherwise be costly. Still, we want to briefly address possible use cases—their full development and assessment, however, will amount to separate paper(s):

- **Monitoring tool:** Our computationally efficient curvature model provides geometric *and* stochastic information about the local loss landscape and can be used by tools like Cockpit [147] to debug optimizers or to gain insights into the optimization problem itself (as in Sections 7.3.2 and 7.3.3).
- **Second-order optimization:** The quantities provided by ViViT, in particular the first- and second-order directional derivatives, can be used to build a stochastic quadratic model of the loss function and perform Newton-like parameter updates. In contrast to existing

second-order methods, *per-sample* quantities contain information about the reliability of that quadratic model. This offers a new dimension for improving second-order methods through statistics on the mini-batch *distribution* of the directional derivatives (*e.g.* for variance-adapted step sizes), potentially increasing the method's performance and stability.

7.6 Conclusion

We have presented ViViT, a curvature model based on the low-rank structure of the Hessian's generalized Gauss-Newton (GGN) approximation. This structure allows for efficient extraction of *exact* curvature properties, such as the GGN's full eigenvalue spectrum and directional gradients and curvatures along the associated eigenvectors. ViViT's quantities scale by approximations that allow for a fine-grained cost-accuracy trade-off. In contrast to alternatives, these quantities offer a notion of curvature uncertainty across the mini-batch in the form of directional derivatives.

We empirically demonstrated the efficiency of leveraging the GGN's low-rank structure and substantiated its usefulness by studying characteristics of curvature noise on various deep learning architectures.

The low-rank representation is efficiently computed in parallel with gradients during a single backward pass. As it mainly relies on vectorized Jacobians, it is general enough to be integrated into existing machine learning libraries in the future. For now, we provide an efficient open-source implementation in PyTorch [126] by extending the existing BackPACK [38] library.

Part III.

Conclusion & Future Directions

Conclusion & Future Directions 8.

Contemporary deep learning is powered by methods that solely rely on the gradient. This is reflected in popular machine learning libraries which prioritize its computation. However, it narrows research to focus on gradient-based algorithms that are not agnostic to the empirical risk's stochasticity and geometry beyond first order. To advance the field, we need to explore the potential of higher-order information beyond the gradient. One main hindrance to further explore its utility has been that it is complicated to implement, which makes it difficult for practitioners to try it out. Therefore, one major goal of this work was to ease experimentation with higher-order information by making it as conveniently accessible as the gradient. This thesis demonstrates that rich information beyond the gradient is *affordable*, can be made *readily available* in existing machine learning libraries, and is *useful* to enable novel approaches for advancing deep learning:

(Q1) *Which information beyond the gradient is efficiently accessible?*

Figure 8.1 provides an overview of the higher-order information made accessible in this work: per-sample gradients—whose empirical mean is the mini-batch gradient—can be explicitly computed, or reduced into higher-order statistics like the gradient variance. Light-weight structural approximations through diagonal and Kronecker matrices enable the computation of per-layer second-order derivatives in the Hessian or generalized Gauss-Newton. The generalized Gauss-Newton's outer product structure allows to go beyond per-layer terms, even to compute with the full matrix, and to access curvature noise.

(Q2) *How to compute this information—conveniently, automatically, and efficiently—re-using the existing backpropagation implementation of ML frameworks?*

All quantities presented in this thesis are phrased as extensions of a standard backward pass.

Therefore, they can be computed *efficiently* and at the same time as the gradient (Figure 8.2). Gradient statistics share most computations with the gradient, and can recycle information from the standard backward

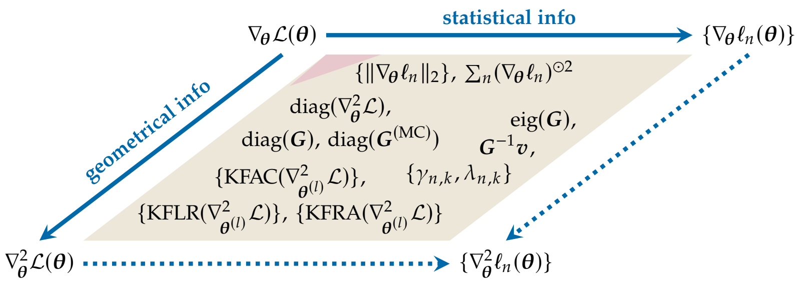

statistical info
geometrical info
$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$
$\{\nabla_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta})\}$
$\nabla^2_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$
$\{\nabla^2_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta})\}$
$\{\|\nabla_{\boldsymbol{\theta}}\ell_n\|_2\}$, $\sum_n(\nabla_{\boldsymbol{\theta}}\ell_n)^{\odot 2}$
$\mathrm{diag}(\nabla^2_{\boldsymbol{\theta}}\mathcal{L})$,
$\mathrm{diag}(\boldsymbol{G})$, $\mathrm{diag}(\boldsymbol{G}^{(\mathrm{MC})})$
$\mathrm{eig}(\boldsymbol{G})$,
$\boldsymbol{G}^{-1}\boldsymbol{v}$,
$\{\mathrm{KFAC}(\nabla^2_{\boldsymbol{\theta}^{(l)}}\mathcal{L})\}$,
$\{\gamma_{n,k}, \lambda_{n,k}\}$
$\{\mathrm{KFLR}(\nabla^2_{\boldsymbol{\theta}^{(l)}}\mathcal{L})\}$, $\{\mathrm{KFRA}(\nabla^2_{\boldsymbol{\theta}^{(l)}}\mathcal{L})\}$

Figure 8.1: Higher-order information made available by this work (extends Figure 1.2). Quantities are roughly mapped onto the landscape spanned by the stochasticity and geometry of the loss. They can be categorized as (i) gradient statistics, (ii) diagonal curvature approximations, (iii) Kronecker-factored curvature approximations, and (iv) noise-aware curvature from the generalized Gauss-Newton's low-rank structure.

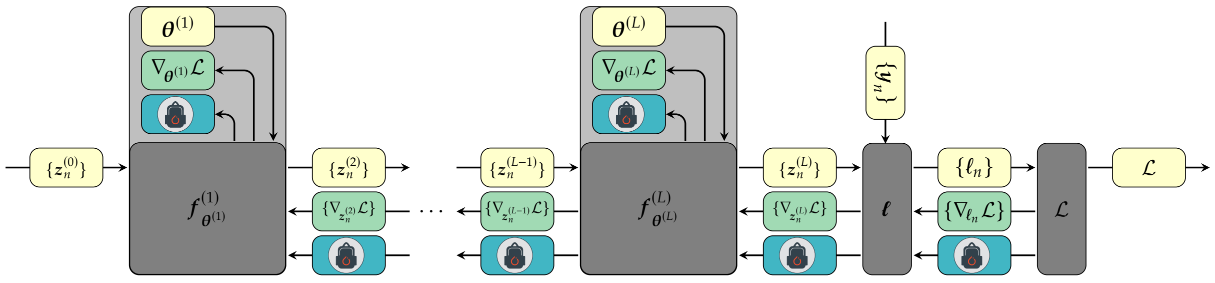

$\theta^{(1)}$
$\nabla_{\theta^{(1)}}\mathcal{L}$
$\{z_n^{(0)}\}$
$f_{\theta^{(1)}}^{(1)}$
$\{z_n^{(2)}\}$
$\{\nabla_{z_n^{(2)}}\mathcal{L}\}$
$\{z_n^{(L-1)}\}$
$\{\nabla_{z_n^{(L-1)}}\mathcal{L}\}$
$\theta^{(L)}$
$\nabla_{\theta^{(L)}}\mathcal{L}$
$f_{\theta^{(L)}}^{(L)}$
$\{z_n^{(L)}\}$
$\{\nabla_{z_n^{(L)}}\mathcal{L}\}$
$\{y_n\}$
ℓ
$\{\ell_n\}$
$\{\nabla_{\ell_n}\mathcal{L}\}$
$\mathcal{L}$
$\mathcal{L}$

Figure 8.2: All information in Figure 8.1 is an extension of the standard backward pass for the gradient. Forward (○) and backward (●) pass are implemented by machine learning libraries. Minimal invasion by adding an extended backward pass (●) allows to compute various other quantities. The specifics (●) on what and how to backpropagate depend on the quantity of interest.

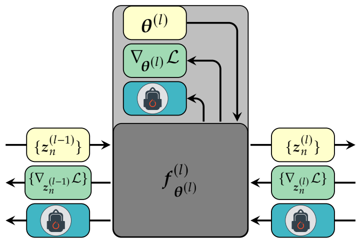

$\theta^{(l)}$
$\nabla_{\theta^{(l)}}\mathcal{L}$
$\{z_n^{(l-1)}\}$
$\{\nabla_{z_n^{(l-1)}}\mathcal{L}\}$
$f_{\theta^{(l)}}^{(l)}$
$\{z_n^{(l)}\}$
$\{\nabla_{z_n^{(l)}}\mathcal{L}\}$

Figure 8.3: Modularity is crucial for automation and extensibility. Given a functioning backpropagation procedure, the extended backward pass in Figure 8.2 reduces to specifying a set of rules for each module and each quantity. This simplifies adding new operations, while preserving automatic computation.

pass. Approximate second-order derivatives require sending additional information through the computation graph. In runtime, the additional work is often significantly reduced through hardware parallelism.

Their formulations vary in what, and how, objects are being backpropagated through layers, and how the target quantities are extracted for each parameter. As these rules are defined on a per-module basis (Figure 8.3), the approach is extensible and guarantees *automatic* computation.

From an implementation perspective, this is achieved through lightweight extension of an existing gradient backpropagation implementation without implementing a new framework. This enables easy integration into existing machine learning libraries, and makes it *convenient* for practitioners to extend their code at minimal overhead.

(Q3) *How to use this information to advance gradient-based deep learning?*

In addition to second-order optimization, various deep learning methods require higher-order information: Laplace approximations [*e.g.* 40], model compression (pruning) [*e.g.* 154], differential privacy [*e.g.* 2], importance sampling [*e.g.* 85], variance adaptation [*e.g.* 9], parameter initialization [*e.g.* 155], batch size adaptation [*e.g.* 11], *etc.* Some of them were briefly outlined in this thesis to motivate the demand that this information be more readily available. Easier access to this information enables more efficient and creative research in these areas and helps establish the resulting methods through user-friendly implementations.

As a use case, this thesis focused on enabling a deeper look into the inner workings of neural networks during training through the lens of higher-order information. This allows to identify common failure modes, which makes training less painful, and thereby highlights the utility of higher-order information for deep learning.

Going further, this work also underlines the unexplored nature of higher-order information: the developed extended automatic differentiation functionality enables novel efficient computational schemes that address shortcomings in existing approaches. This allows to build more powerful local approximations of the loss landscape which are agnostic to noise, curvature, and even noise in the curvature. Such works help to identify important techniques to reduce run time, make algorithms that rely on such information more competitive with gradient-based methods, and shape the development of future machine learning frameworks.

8.1 Summary & Impact

Extending Backpropagation to the Hessian

Gradient backpropagation is efficient, automated, and extensible. Chapter 4 presented how—for sequential feedforward architectures—this carries over to second-order derivatives: just like backpropagation recovers the layer-wise gradients, Hessian backpropagation recovers the per-layer Hessians. It fully aligns the computation of local Hessians with gradients and unifies the view on block-diagonal curvature approximations like the block-diagonal generalized Gauss-Newton [148], Fisher [5], and its Kronecker-factorized approximations [21, 31, 63, 109].

Packing More into Backprop

The BackPACK library, presented in Chapter 5, provides efficient access to various deep learning quantities through implementing the insights on extended backpropagation for the Hessian on top of PyTorch. During a standard backward pass that computes the average gradient, it extracts (i) per-sample gradients and gradient statistics, and (ii) approximate second-order derivatives in the form of diagonal and Kronecker-factorized curvature. This often adds only little overhead. BackPACK easily integrates into existing code and simplifies experimentation with the above quantities: it powers other libraries for Bayesian applications with Laplace approximations [40, 76], out-of-distribution generalization [64, 130], and differential privacy [179], as well as the follow-up works in this thesis (Cockpit [147], ViViT [39]). More than two years after its release, the library is still actively used, with multiple hundred downloads per week at the time of writing (July 2022).

Enabling a Closer Look Into Neural Nets

Higher-order information as provided by BackPACK is valuable to guide neural network training. Common methods for real-time training diagnostics, such as monitoring the loss, are limited because they only indicate whether a model is training, but not why. The Cockpit library, presented in Chapter 6, enables a closer look into neural networks during training. The live-monitoring tool visualizes established, recently proposed [11, 19, 26, 100, 104, 162, 177], and novel summary statistics that are efficiently computed by BackPACK. It allows to identify common bugs in the machine learning pipeline, such as improper data pre-processing or vanishing gradients, but also to guide learning rate selection, and to study implicit regularization [56, 112]. This showcases the potential of higher-order information to assist practitioners.

Enabling Novel Ways to Compute with Curvature

BackPACK's extended automatic differentiation functionality enables algorithmic advances to tackle limitations of existing curvature proxies: diagonal or Kronecker-factorized curvatures are (i) not agnostic to noise in the mini-batch, (ii) strict approximations that do not become exact in any limit, and (iii) restricted to the block diagonal. ViViT's quantities, presented in Chapter 7, address this through the generalized Gauss-Newton's low-rank structure, which allows for exact computation with the full—rather than block-diagonal—matrix, and principled approximations to reduce cost in exchange for less accuracy. ViViT enables efficient computation of spectral properties, as well as directional gradients and curvatures on a per-sample basis that quantify noise. Monitoring this noise through signal-to-noise ratios helps understand its characteristics in deep learning [48] and to identify challenges for optimization and generalization from the interplay between noise and curvature [162].

8.2 Future Work

Deep learning needs more than just the gradient. To leverage the full potential of higher-order information, we need to (i) build more—and refine existing—tools to (ii) study and better understand algorithmic challenges in deep learning, and (iii) amplify the practicality of such next-generation algorithms through user-friendly and highly-efficient implementations like gradient-based methods.

Extending Cockpit

Chapter 6 demonstrated Cockpit's utility for identifying common bugs in the machine learning pipeline. These failures were deliberately designed on well-known, standardized machine learning problems, to illustrate Cockpit's purpose. Since they were implicitly tuned over many years to work well with currently popular methods, they rarely exhibit failure modes. Cockpit will be even more useful for debugging unknown, non-standardized problems that have *not* undergone such tuning, and are therefore extremely likely to exhibit failures. We want to establish Cockpit as a tool for practitioners facing such problems and are looking forward to its first success stories "in the wild".

Cockpit also provides functionality for scientific analyses of neural networks. Training such models is often wasteful in computation, *e.g.* using large grid searches whose computations are effectively discarded after identifying one well-performing set of hyperparameters. Cockpit's summary statistics are condensed and could be stored for a large number of training trajectories corresponding to different hyperparameter settings. These trajectories could be collected into a dataset that could serve for the analysis of optimization algorithms to understand their implicit bias, to study properties of neural networks that generalize well, or for meta-learning optimization strategies.

To further improve the meaningfulness and interpretability of Cockpit's instruments, its control over parts of the network could be made more customizable: currently, most quantities can be computed either on all parameters, or per parameter. For very large networks, it will be more practical to group parameters, and to compute and visualize Cockpit's instruments per parameter group.

Noise-aware Second-order Methods

Newton steps are powerful, but their stability is strongly affected by noise: one corrupted step might undo all previous progress. Improving their stability is thus one key challenge to make them work in the mini-batch setting. To do that, we need to quantify noise in the mini-batch. But popular curvature proxies used in second-order methods are not noise-agnostic. Therefore, we need curvature approximations that give access to noise, *e.g.* through per-sample information as provided through ViViT. Such information could be used to develop noise-aware stabilizing mechanisms for Newton steps, like damping.

Optimizing Run Time & Advancing Automatic Differentiation

In contrast to gradient-based methods, the run time performance of higher-order methods can still be significantly improved: *e.g.* BackPACK outperforms naive implementations (like for-loops) and achieves practical overheads. But it relies on PyTorch's Python API, which is sometimes not flexible enough, and therefore realizes some functionality through less efficient workarounds. Recent advances in automatic differentiation, like JAX [23] and functorch [72], rely on function transformation to achieve a clean separation of automatic differentiation and batching, and allow for more efficient implementations through just-in-time compilation.

Often, performance improvements are achieved through leveraging linear algebra, like properties of the Kronecker product [101] and matrix decompositions [*e.g.* 38, 39]. Recent work suggests that there is potential for further improvements, as a number of these optimizations are not yet realized [141]. Therefore, one direction would be to improve the automated optimization of operations in second-order methods in these libraries. This would further reduce the overhead of second-order methods stemming from poor implementation, and make these performance gains widely available to the machine learning community.

Part IV.

Appendix

Additional Material for Chapter 4 A.

Here, we provide additional details and derivations for Hessian backpropagation (HBP).

Appendix A.1 relates the HBP Equation (4.7) to the chain rule for matrix derivatives [103]. It relies on the clean definitions of generalized Jacobian and Hessian matrices for multi-variate functions (Equations (2.27) and (3.6)), and their chain rules (Theorems 2.2 and 3.1).

With matrix derivatives, the HBP equation for a variety of module functions can be derived elegantly. Appendices A.2 to A.4 contain the HBP derivations for all operations in Table 4.1. We split the considered operations into different categories to achieve a cleaner structure. Appendix A.2 contains details on operations used for the construction of fully-connected neural networks (FCNNs) and skip-connections. Appendix A.2.4 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. [21] and Chen et al. [31], the latter of which has been used in the experiment of Section 4.4. The analysis of the Hessian for common loss functions is provided in Appendix A.3. Operations occurring in convolutional neural networks (CNNs) are subject of Appendix A.4.

Appendix A.5 provides details on model architectures, training procedures used in the experiments of Section 4.4, and an additional experiment on a modified test problem of the DeepOBS benchmark library [146].

A.1 Matrix Derivatives

Index notation for higher-order derivatives of multi-variate matrix functions can become heavy [8, 31, 110, 115]. We tackle this by embedding our approach in the notation of matrix differential calculus, which

1. yields notation consistent with established literature on matrix derivatives [103] and clarifies the origin of the symbols J and ∇^2, used extensively in the main text (Equations (2.27) and (3.6))[1].
2. allows for using a multi-dimensional generalization of the chain rule (Theorems 2.2 and 3.1).
3. lets us extract first- and second-order derivatives from differentials using the identification rules of Magnus and Neudecker [103] without bothering to deal with index notation.

1: Magnus and Neudecker [103] use $\mathrm{D}\boldsymbol{B}(\boldsymbol{A}), \mathrm{H}\boldsymbol{B}(\boldsymbol{A})$ for the Jacobian $\mathrm{J}_{\boldsymbol{A}}\boldsymbol{B}$ and Hessian $\nabla^2_{\boldsymbol{A}}\boldsymbol{B}$ of matrix variables $\boldsymbol{A}, \boldsymbol{B}$.

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

Preliminaries & Notation

Equations (2.27) and (3.6) and theorems 2.2 and 3.1 represent a collection of results from the book of Magnus and Neudecker [103]. 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 multi-variate generalization of the chain rule.

We denote matrix, vector, and scalar functions by $\boldsymbol{F}, \boldsymbol{f}$, and ϕ, respectively. Matrix (vector) arguments are written as $\boldsymbol{X}$ ($\boldsymbol{x}$). Vectorization (vec, Definition 2.2) applies column-stacking, such that for matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$,

$$\operatorname{vec}(\boldsymbol{A}\boldsymbol{B}\boldsymbol{C}) = \left(\boldsymbol{C}^\top \otimes \boldsymbol{A}\right) \operatorname{vec}(\boldsymbol{B})\,. \tag{A.1}$$

We assign vectors to bold lower-case ($\boldsymbol{x}, \boldsymbol{\theta}, \dots$), matrices to bold upper-case ($\boldsymbol{W}, \boldsymbol{X}, \dots$), and tensors to bold upper-case sans serif symbols ($\mathsf{\mathbf{W}}, \mathsf{\mathbf{X}}, \dots$). $\odot$ means elementwise multiplication (Hadamard product).

Remark on Vectorization

The generalized Jacobian and Hessian from [103] rely 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. *E.g.*, the vectorized version of the tensor $\mathsf{\mathbf{A}} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ with $n_1, n_2, n_3 \in \mathbb{N}$ is $\operatorname{vec} \mathsf{\mathbf{A}} = (\mathsf{A}_{1,1,1}, \mathsf{A}_{2,1,1}, \dots, \mathsf{A}_{n_1,1,1}, \mathsf{A}_{1,2,1}, \dots, \mathsf{A}_{n_1,n_2,n_3})^\top$. 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.1.1 Relation to the Modular Approach

Theorem 3.1 can directly be applied to the graph $\ell \circ f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ f^{(L-1)}_{\boldsymbol{\theta}^{(L-1)}} \circ \dots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}}$ of the sequential feedforward net under investigation. For any module function $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$, the loss can be expressed as a composition of two functions by squashing preceding modules in the graph into a single function $f^{(l-1)}_{\boldsymbol{\theta}^{(l-1)}} \circ \dots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}}$, and likewise composing the module itself and all subsequent functions, *i.e.* $\ell \circ f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ \dots \circ f^{(l)}_{\boldsymbol{\theta}^{(l)}}$.

The analysis can therefore be reduced to the module shown in Figure 4.2 receiving an input $\boldsymbol{x} \in \mathbb{R}^n$ that is used to compute the output $\boldsymbol{z} \in \mathbb{R}^m$.

The scalar loss is then expressed as a mapping $\ell(\boldsymbol{z}(\boldsymbol{x}), \boldsymbol{y}) : \mathbb{R}^n \to \mathbb{R}^p$ with $p = 1$. Suppressing the label $\boldsymbol{y}$, Equation (3.14) implies

$$\begin{aligned}\nabla_{\boldsymbol{x}}^2 \ell &= \left(\boldsymbol{I}_p \otimes \mathrm{J}_{\boldsymbol{x}} \boldsymbol{z}\right)^\top \left(\nabla_{\boldsymbol{z}}^2 \ell\right) \mathrm{J}_{\boldsymbol{x}} \boldsymbol{z} + \left(\mathrm{J}_{\boldsymbol{z}} \ell \otimes \boldsymbol{I}_n\right) \nabla_{\boldsymbol{x}}^2 \boldsymbol{z} \\ &= \left(\mathrm{J}_{\boldsymbol{x}} \boldsymbol{z}\right)^\top \left(\nabla_{\boldsymbol{z}}^2 \ell\right) \mathrm{J}_{\boldsymbol{x}} \boldsymbol{z} + \left(\mathrm{J}_{\boldsymbol{z}} \ell \otimes \boldsymbol{I}_n\right) \nabla_{\boldsymbol{x}}^2 \boldsymbol{z} \,.\end{aligned} \tag{A.2}$$

The HBP Equation (4.7), which contains Hessians of elements of $\boldsymbol{z}$, is obtained by substituting the form Equation (3.7) into Equation (A.2).

A.2 HBP for FCNNs

A.2.1 Linear Layer (Matrix-vector Multiplication, Matrix-Matrix Multiplication, Addition)

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

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

To compute the Jacobians *w.r.t.* each variable, we use the differentials

$$\begin{aligned}\mathrm{d}\boldsymbol{z}(\boldsymbol{x}) &= \boldsymbol{W} \mathrm{d}\boldsymbol{x} \,, \\ \mathrm{d}\boldsymbol{z}(\boldsymbol{b}) &= \mathrm{d}\boldsymbol{b} \,, \\ \mathrm{d}\boldsymbol{z}(\boldsymbol{W}) &= (\mathrm{d}\boldsymbol{W})\boldsymbol{x} \quad \Longrightarrow \quad \mathrm{d}\,\mathrm{vec}\,\boldsymbol{z}(\boldsymbol{W}) = \left(\boldsymbol{x}^\top \otimes \boldsymbol{I}_m\right) \mathrm{vec}(\mathrm{d}\boldsymbol{W}) \,,\end{aligned}$$

using Equation (A.1) to establish the implication in the last line. With the *first identification tables* provided in Magnus and Neudecker [103, Chapter 9.6], the Jacobians can be read off from the differentials as $\mathrm{J}_{\boldsymbol{x}} \boldsymbol{z} = \boldsymbol{W}$, $\mathrm{J}_{\boldsymbol{b}} \boldsymbol{z} = \boldsymbol{I}_m$, $\mathrm{J}_{\boldsymbol{W}} \boldsymbol{z} = \boldsymbol{x}^\top \otimes \boldsymbol{I}_m$. All second module derivatives $\nabla_{\boldsymbol{x}}^2 \boldsymbol{z}$, $\nabla_{\boldsymbol{W}}^2 \boldsymbol{z}$, and $\nabla_{\boldsymbol{b}}^2 \boldsymbol{z}$ vanish since $\boldsymbol{f}$ is linear in all inputs. Inserting the Jacobians into Equation (4.7) results in

$$\nabla_{\boldsymbol{x}}^2 \ell = \boldsymbol{W}^\top \left(\nabla_{\boldsymbol{z}}^2 \ell\right) \boldsymbol{W} \,, \tag{A.3a}$$

$$\nabla_{\boldsymbol{b}}^2 \ell = \nabla_{\boldsymbol{z}}^2 \ell \,, \tag{A.3b}$$

$$\begin{aligned}\nabla_{\boldsymbol{W}}^2 \ell &= \left(\boldsymbol{x}^\top \otimes \boldsymbol{I}_m\right)^\top \left(\nabla_{\boldsymbol{z}}^2 \ell\right) \left(\boldsymbol{x}^\top \otimes \boldsymbol{I}_m\right) \\ &= \boldsymbol{x}\boldsymbol{x}^\top \otimes \nabla_{\boldsymbol{z}}^2 \ell = \boldsymbol{x} \otimes \boldsymbol{x}^\top \otimes \nabla_{\boldsymbol{z}}^2 \ell \,.\end{aligned} \tag{A.3c}$$

The HBP relations for matrix-vector multiplication and addition listed in Table 4.1 are special cases of Equation (A.3). HBP for matrix-matrix multiplication is derived in a completely analogous fashion.

A.2.2 Elementwise Activation

Next, consider the elementwise application of a nonlinear function ϕ,

$$\begin{aligned}\boldsymbol{\phi} : \quad & \mathbb{R}^m \to \mathbb{R}^m \\ & \boldsymbol{x} \mapsto \boldsymbol{z} = \boldsymbol{\phi}(\boldsymbol{x}) \quad \text{such that} \quad [\boldsymbol{\phi}(\boldsymbol{x})]_k = \phi([\boldsymbol{x}]_k) \,,\end{aligned}$$

For the matrix differential *w.r.t.* $\boldsymbol{x}$, this implies

$$\mathrm{d}\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{\phi}'(\boldsymbol{x}) \odot \mathrm{d}\boldsymbol{x} = \operatorname{diag}\left[\boldsymbol{\phi}'(\boldsymbol{x})\right] \mathrm{d}\boldsymbol{x}\,,$$

where $\boldsymbol{\phi}'$ means elementwise application of ϕ', and consequently, the Jacobian is a diagonal matrix $\mathrm{J}_{\boldsymbol{x}}\boldsymbol{\phi}(\boldsymbol{x}) = \operatorname{diag}\left[\boldsymbol{\phi}'(\boldsymbol{x})\right]$. For the Hessian, note that the function value $\phi_k(\boldsymbol{x}) := [\boldsymbol{\phi}(\boldsymbol{x})]_k$ only depends on $x_k := [\boldsymbol{x}]_k$ and thus $\nabla^2_{\boldsymbol{x}}\phi_k = \phi''(x_k)\hat{\boldsymbol{e}}_k\hat{\boldsymbol{e}}_k^\top$, with the one-hot unit vector $\hat{\boldsymbol{e}}_k \in \mathbb{R}^m$ in coordinate direction k. Inserting all quantities into Equation (4.7) yields

$$\begin{aligned} \nabla^2_{\boldsymbol{x}}\ell &= \operatorname{diag}\left[\boldsymbol{\phi}'(\boldsymbol{x})\right]\left(\nabla^2_{\boldsymbol{z}}\ell\right)\operatorname{diag}\left[\boldsymbol{\phi}'(\boldsymbol{x})\right] + \sum_k \phi''(x_k)\hat{\boldsymbol{e}}_k\hat{\boldsymbol{e}}_k^\top\left(\nabla_{z_k}\ell\right) \\ &= \operatorname{diag}\left[\boldsymbol{\phi}'(\boldsymbol{x})\right]\left(\nabla^2_{\boldsymbol{z}}\ell\right)\operatorname{diag}\left[\boldsymbol{\phi}'(\boldsymbol{x})\right] + \operatorname{diag}\left[\boldsymbol{\phi}''(\boldsymbol{x}) \odot \nabla_{\boldsymbol{z}}\ell\right]\,, \end{aligned} \tag{A.4}$$

where $\boldsymbol{\phi}''$ means elementwise application of ϕ''.

A.2.3 Skip-connection

Residual learning [68] uses skip-connection units to facilitate the training of DNNs. In its simplest form, the mapping $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^m$ reads

$$\boldsymbol{z}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{x} + \boldsymbol{s}(\boldsymbol{x}, \boldsymbol{\theta})\,,$$

with a potentially nonlinear operation $(\boldsymbol{x}, \boldsymbol{\theta}) \mapsto \boldsymbol{s}$. The input and parameter Jacobians are given by $\mathrm{J}_{\boldsymbol{z}}\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{I}_m + \mathrm{J}_{\boldsymbol{x}}\boldsymbol{s}(\boldsymbol{x})$ and $\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{z}(\boldsymbol{\theta}) = \mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{s}(\boldsymbol{\theta})$. Using Equation (4.7), one finds

$$\begin{aligned} \nabla^2_{\boldsymbol{x}}\ell &= \left[\boldsymbol{I}_m + \mathrm{J}_{\boldsymbol{x}}\boldsymbol{s}(\boldsymbol{x})\right]^\top\left(\nabla^2_{\boldsymbol{z}}\ell\right)\left[\boldsymbol{I}_m + \mathrm{J}_{\boldsymbol{x}}\boldsymbol{s}(\boldsymbol{x})\right] + \sum_k\left[\nabla^2_{\boldsymbol{x}}s_k(\boldsymbol{x})\right]\left(\nabla_{z_k}\ell\right)\,, \\ \nabla^2_{\boldsymbol{\theta}}\ell &= \left[\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{s}(\boldsymbol{\theta})\right]^\top\left(\nabla^2_{\boldsymbol{z}}\ell\right)\left[\mathrm{J}_{\boldsymbol{\theta}}\boldsymbol{s}(\boldsymbol{\theta})\right] + \sum_k\left[\nabla^2_{\boldsymbol{\theta}}s_k(\boldsymbol{\theta})\right]\left(\nabla_{z_k}\ell\right)\,. \end{aligned}$$

A.2.4 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. [21] (KFRA) and Chen et al. [31] (BDA-PCH).

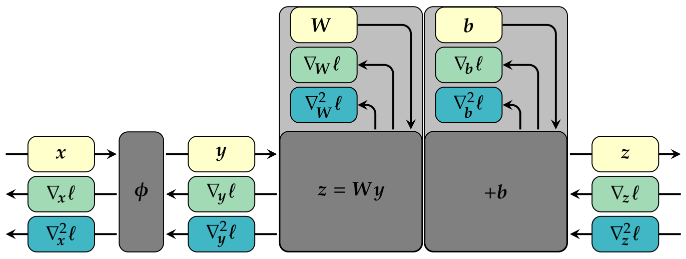
Analytic Composition of Multiple Modules

Consider the module $\boldsymbol{g} = \boldsymbol{f} \circ \boldsymbol{\phi}, \boldsymbol{x} \mapsto \boldsymbol{y} := \boldsymbol{\phi}(\boldsymbol{x}) \mapsto \boldsymbol{z} = \boldsymbol{f}(\boldsymbol{y}(\boldsymbol{x}))$. Assume $\boldsymbol{\phi}$ to act elementwise on the input, followed by a linear layer $\boldsymbol{f} : \boldsymbol{z}(\boldsymbol{y}) = \boldsymbol{W}\boldsymbol{y} + \boldsymbol{b}$ (Figure A.1a). Analytic elimination of the intermediate backward pass recovers the backward pass of the fused module that consists of both operations (Figure A.1b). The first Hessian backward pass through the linear module $\boldsymbol{f}$ (Equation (A.3)) implies

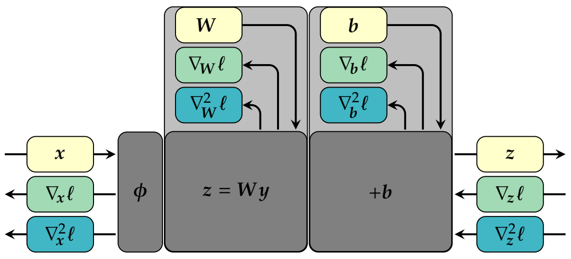
$$\nabla^2_{\boldsymbol{y}}\ell = \boldsymbol{W}^\top\left(\nabla^2_{\boldsymbol{z}}\ell\right)\boldsymbol{W}\,,$$

$$\nabla^2_{\boldsymbol{W}}\ell = \boldsymbol{y} \otimes \boldsymbol{y}^\top \otimes \nabla^2_{\boldsymbol{z}}\ell = \boldsymbol{\phi}(\boldsymbol{x}) \otimes \boldsymbol{\phi}(\boldsymbol{x})^\top \otimes \nabla^2_{\boldsymbol{z}}\ell\,, \tag{A.5a}$$

$$\nabla^2_{\boldsymbol{b}}\ell = \nabla^2_{\boldsymbol{z}}\ell\,. \tag{A.5b}$$



(a) Activation and linear layer as two separate modules



(b) Activation and linear layer fused into a single module

Figure A.1: The sequence of elementwise activation ϕ and linear layer can be interpreted as two modules, or a single one. (a) Both operations are analyzed separately to derive the HBP. **(b)** Backpropagation of $\nabla_z^2 \ell$ is expressed in terms of $\nabla_x^2 \ell$ without intermediate message.

Further backpropagation through ϕ with Equation (A.4) results in

$$\begin{aligned}\nabla_x^2 \ell &= \operatorname{diag}[\phi'(x)] \left(\nabla_y^2 \ell\right) \operatorname{diag}[\phi'(x)] + \operatorname{diag}\left[\phi''(x) \odot (\nabla_y \ell)\right] \\ &= \operatorname{diag}[\phi'(x)] \left[W^\top (\nabla_z^2 \ell) W\right] \operatorname{diag}[\phi'(x)] \\ &\quad + \operatorname{diag}\left[\phi''(x) \odot W^\top (\nabla_z \ell)\right] \\ &= \{W \operatorname{diag}[\phi'(x)]\}^\top (\nabla_z^2 \ell) \{W \operatorname{diag}[\phi'(x)]\} \\ &\quad + \operatorname{diag}\left[\phi''(x) \odot W^\top (\nabla_z \ell)\right] .\end{aligned} \tag{A.5c}$$

We use invariance of a diagonal matrix under transposition and $\nabla_y \ell = W^\top (\nabla_z \ell)$ for the backpropagated gradient for the last equality. The Jacobian $\mathrm{J}_x g(x)$ of the module shown in Figure A.1b is $\mathrm{J}_x g(x) = W \operatorname{diag}[\phi(x)] = [W^\top \odot \phi'(x)]^\top$ (broadcasting $\phi'(x)$). In summary, HBP for the composite layer $z(x) = W\phi(x) + b$ is given by Equation (A.5).

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. [21] and Chen et al. [31]. Their relations are obtained from a straightforward conversion of the HBP rules equation A.5. Consider a sequence of a linear layer $f^{(1)}_{\theta^{(1)}}$ and multiple composite modules $f^{(2)}_{\theta^{(2)}}, \dots, f^{(L)}_{\theta^{(L)}}$ as shown in Figure A.2.

According to Equation (A.5b) both the linear layer and the composite $f^{(l)}_{\theta^{(l)}}$ identify the gradient (Hessian) *w.r.t.* their outputs, $\nabla_{z^{(l)}} \ell$ ($\nabla^2_{z^{(l)}} \ell$), as the gradient (Hessian) *w.r.t.* their bias term, $\nabla_{b^{(l)}} \ell$ ($\nabla^2_{b^{(l)}} \ell$). Introducing layer indices for all quantities, one finds the recursion

$$\nabla^2_{b^{(l)}} \ell = \nabla^2_{z^{(l)}} \ell \,, \tag{A.6a}$$

$$\nabla^2_{W^{(l)}} \ell = \phi(z^{(l-1)}) \otimes \phi(z^{(l-1)})^\top \otimes \nabla^2_{b^{(l)}} \ell \,, \tag{A.6b}$$

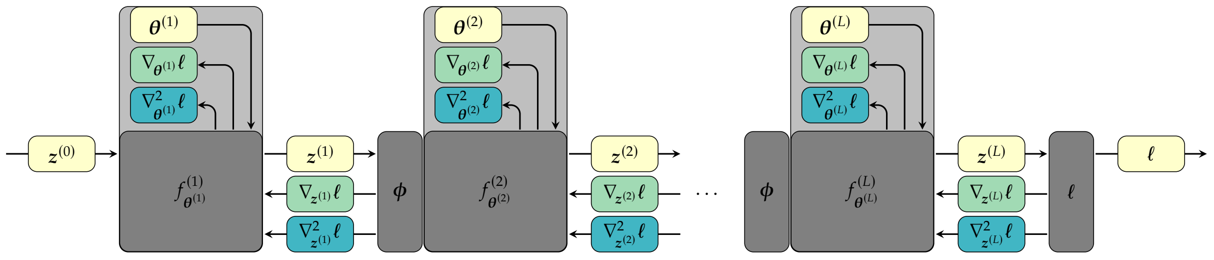


Figure A.2: Grouping scheme for the recursive Hessian computation proposed by KFRA and BDA-PCH. The backward messages between the linear layer and the preceding nonlinear activation are analytically fused.

for $l = L-1, \dots, 1$, and

$$\begin{aligned}\nabla^2_{z^{(l-1)}}\ell &= \left\{W^{(l)} \operatorname{diag}\left[\phi'(z^{(l-1)})\right]\right\}^\top \nabla^2_{b^{(l)}}\ell \left\{W^{(l)} \operatorname{diag}\left[\phi'(z^{(l-1)})\right]\right\} \\ &\quad + \operatorname{diag}\left[\phi''(z^{(l-1)}) \odot W^{(l)\top}\nabla_{b^{(l)}}\ell\right] \\ &= \left\{W^{(l)\top} \odot \phi'(z^{(l-1)})\right\} \nabla^2_{b^{(l)}}\ell \left\{W^{(l)\top} \odot \phi'(z^{(l-1)})\right\}^\top \\ &\quad + \operatorname{diag}\left[\phi''(z^{(l-1)}) \odot W^{(l)\top}\nabla_{b^{(l)}}\ell\right]\end{aligned} \tag{A.6c}$$

for $l = L-1, \dots, 2$. It is initialized with the loss function's gradient (Hessian) $\nabla_{z^{(L)}}\ell$ ($\nabla^2_{z^{(L)}}\ell$).

Equation (A.6) are equivalent to the expressions provided in [21, 31]. Their emergence from compositions of HBP equations of simple operations represents one key insight of Chapter 4. Both works use the batch average strategy presented in Section 4.3.2 to obtain curvature estimates.

A.3 HBP for Loss Functions

A.3.1 Square Loss

Square loss of the model prediction $f \in \mathbb{R}^C$ and the true label $y \in \mathbb{R}^C$ is computed by (Equation (2.1))

$$\ell(f, y) = \frac{1}{C}(y-f)^\top(y-f)\,.$$

Differentiating

$$d\ell(f) = \frac{1}{C}\left[-(df)^\top(y-f) - (y-f)^\top df\right] = -\frac{2}{C}(y-f)^\top df$$

once more yields

$$d^2\ell(f) = \frac{2}{C}(df)^\top df = \frac{2}{C}(df)^\top I_C (df)\,.$$

The Hessian is extracted with the *second identification tables* from Magnus and Neudecker [103, Chapter 10.4] and reproduces the expected result

$$\nabla^2_f \ell(f) = \frac{2}{C} I_C\,. \tag{A.7}$$

A.3.2 Softmax Cross-entropy Loss

The computation of cross-entropy from logits (Equation (2.2)) 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 distribution implied by the label is computed.

Log-softmax

The output's elements $\boldsymbol{f} \in \mathbb{R}^C$ of a neural network are assigned to log-probabilities $\boldsymbol{z}(\boldsymbol{f}) = \log p(\boldsymbol{f}) \in \mathbb{R}^C$ by means of the softmax function $p(\boldsymbol{f}) = {}^{\exp(\boldsymbol{f})}/_{\sum_i \exp(f_i)}$. Consequently,

$$\boldsymbol{z}(\boldsymbol{f}) = \boldsymbol{f} - \log\left[\sum_i \exp(f_i)\right] ,$$

and the Jacobian reads $\mathrm{J}_{\boldsymbol{f}}\boldsymbol{z}(\boldsymbol{f}) = \boldsymbol{I}_C - \boldsymbol{1}_C p(\boldsymbol{f})^\top$ with $\boldsymbol{1}_C \in \mathbb{R}^C$ denoting a vector of ones. The log-probability Hessians *w.r.t.* $\boldsymbol{f}$ are given by

$$\nabla^2_{\boldsymbol{f}}[\boldsymbol{z}(\boldsymbol{f})]_k = -\operatorname{diag}\left[p(\boldsymbol{f})\right] + p(\boldsymbol{f})p(\boldsymbol{f})^\top .$$

Cross-entropy

The negative log-probabilities are used for the cross-entropy with the probability distribution of the label $y \in \{1, \dots, c\}$,

$$\ell(\boldsymbol{z}, y) = -\operatorname{onehot}(y)^\top \boldsymbol{z} .$$

Since ℓ is linear in the log-probabilities $\boldsymbol{z}$, *i.e.* $\nabla^2_{\boldsymbol{z}}\ell(\boldsymbol{z}) = \boldsymbol{0}$, the HBP is

$$\begin{aligned}\nabla^2_{\boldsymbol{f}}\ell(\boldsymbol{f}) &= \left[\mathrm{J}_{\boldsymbol{f}}\boldsymbol{z}(\boldsymbol{f})\right]^\top \nabla^2_{\boldsymbol{z}}\ell(\boldsymbol{z}) \left[\mathrm{J}_{\boldsymbol{f}}\boldsymbol{z}(\boldsymbol{f})\right] + \sum_k \nabla^2_{\boldsymbol{f}}[\boldsymbol{z}(\boldsymbol{f})]_k \frac{\partial \ell(\boldsymbol{z})}{\partial [\boldsymbol{z}]_k} \\ &= \left\{-\operatorname{diag}\left[p(\boldsymbol{f})\right] + p(\boldsymbol{f})p(\boldsymbol{f})^\top\right\} \sum_k [-\operatorname{onehot}(y)]_k \\ &= \operatorname{diag}\left[p(\boldsymbol{f})\right] - p(\boldsymbol{f})p(\boldsymbol{f})^\top .\end{aligned}$$

A.4 HBP for CNNs

The recursive approaches in [21, 31] 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 might be that convolutions come with heavy notation that is difficult to deal with in index notation.

Martens and Grosse [109] 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 back-propagation of second-order derivatives.

To derive the HBP for convolutions, we use that an efficient implementation of the forward pass is decomposed into multiple operations (see

Figure A.4), which can be considered independently by means of our modular approach (see Figure A.5 for details). Our analysis starts by considering the backpropagation of curvature through operations that occur frequently in CNNs. This includes the reshape (Appendix A.4.1) and extraction operation (Appendix A.4.2). In Appendix A.4.3 we outline the modular decomposition and curvature backpropagation of convolution in two dimensions. The approach carries over to other dimensions.

All operations under consideration in this section are linear. Hence the second terms in Equation (4.7) vanish. Again, we use the framework of matrix differential calculus [103] to avoid index notation.

A.4.1 Reshape/View

The reshape operation reinterprets a tensor $\mathsf{X} \in \mathbb{R}^{n_1 \times \cdots \times n_x}$ as another tensor $\mathsf{Z} \in \mathbb{Z}^{m_1 \times \cdots \times m_z}$ (Definition 2.3)

$$\mathsf{Z}(\mathsf{X}) = \mathrm{reshape}(\mathsf{X})\,,$$

which possesses the same number of elements, *i.e.* $\prod_i n_i = \prod_i m_i$. One example is given by the vec operation from Definition 2.2. It corresponds to a reshape into a tensor of order one. As the arrangement of elements remains unaffected, $\mathrm{vec}\,\mathsf{Z} = \mathrm{vec}\,\mathsf{X}$, and reshaping corresponds to the identity map on the vectorized input. Consequently, one finds (remember that $\nabla^2_{\mathsf{X}}\ell$ is a shorthand notation for $\nabla^2_{\mathrm{vec}\,\mathsf{X}}\ell$)

$$\nabla^2_{\mathsf{X}}\ell = \nabla^2_{\mathsf{Z}}\ell\,.$$

A.4.2 Index Select/Map

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

$$[\boldsymbol{z}]_j = [\boldsymbol{x}]_{\pi(j)} \quad\Leftrightarrow\quad \boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{\Pi}\boldsymbol{x} \quad \text{with} \quad [\boldsymbol{\Pi}]_{j,\pi(j)} = 1\,.$$

The HBP is equivalent to the linear layer discussed in Appendix A.2.1,

$$\nabla^2_{\boldsymbol{z}}\ell = \boldsymbol{\Pi}^\top(\nabla^2_{\boldsymbol{x}}\ell)\boldsymbol{\Pi}\,.$$

Applications include max-pooling and the `im2col`/`unfold` operation (see Appendix A.4.3). Average-pooling represents a weighted sum of index selections and can be treated analogously.

A.4.3 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 A.3a illustrates the

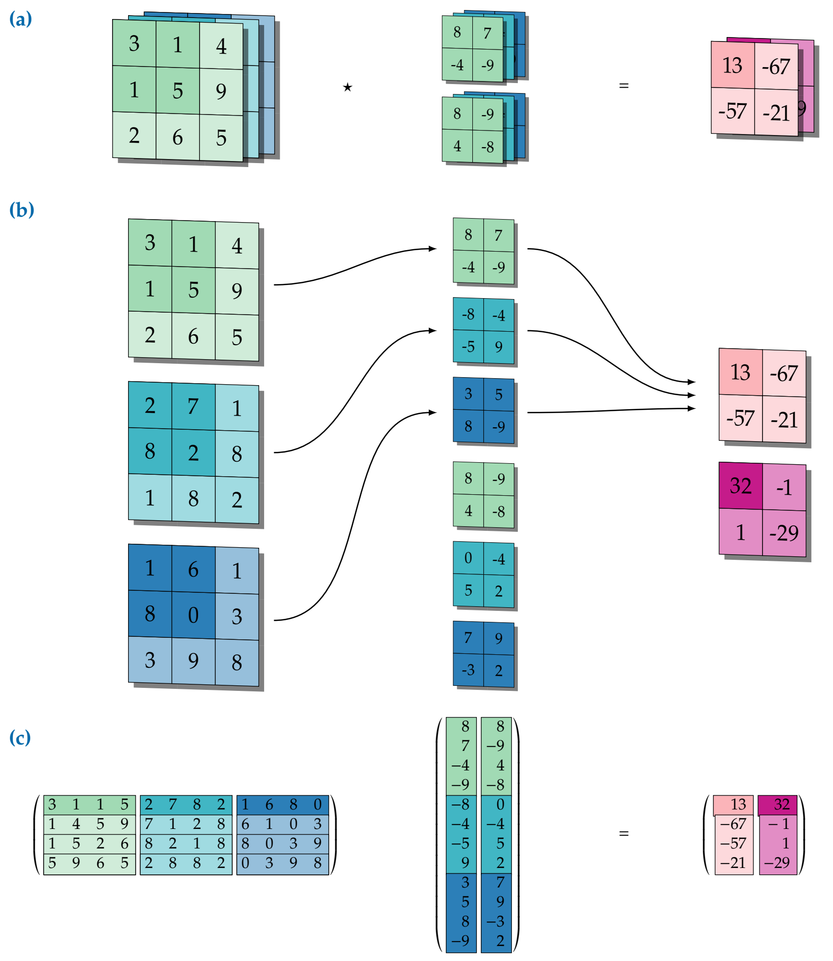


Figure A.3: Two-dimensional convolution $\mathsf{Y} = \mathsf{X} \star \mathsf{W}$ without bias term. (a) The input X consists of $C_{\text{in}} = 3$ channels (different colors) of (3×3) images. Filter maps of size (2×2) are provided by the kernel W for the generation of $C_{\text{out}} = 2$ output channels. 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 $C_{\text{out}} = 2$ channels of (2×2) images. **(b)** Detailed view. All tensors are unrolled along the first axis. **(c)** Convolution as matrix multiplication. From left to right, the matrices $[\![\mathsf{X}]\!]^\top$, $W^\top$, and $Y^\top$ are shown.

setting. A collection of filter maps, the *kernel* W, is slid over the spatial coordinates of the input tensor 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 45]. The approach presented here is not limited to a certain choice of convolution hyperparameters.

Forward Pass & Notation

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

- The input X, a tensor of order three, stores a collection of two-dimensional data. Its components $\mathsf{X}_{c_{\text{in}},x_1,x_2}$ are referenced by indices for the channel c_{in} and the spatial location (x_1, x_2). C_{in}, X_1, X_2 denote input channels, width & height of the image, respectively.
- The kernel W is a tensor of order four with shape $(C_{\text{out}}, C_{\text{in}}, K_1, K_2)$. Kernel width K_1 and height K_2 determine the patch size $P = K_1 K_2$

Table A.1: Important quantities for the convolution operation. The number of patches equals the product of the outputs' spatial dimensions, *i.e.* $P = Y_1Y_2$.

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
$[\![\mathsf{X}]\!]$	$(C_{\text{in}}K_1K_2, P)$	Expanded input
$\boldsymbol{W}$	$(C_{\text{out}}, C_{\text{in}}K_1K_2)$	Matricized kernel
$\boldsymbol{Y}$	(C_{out}, P)	Matricized output
$\boldsymbol{b}$	C_{out}	Bias vector
$\boldsymbol{B}$	(C_{out}, P)	Bias matrix

for each channel. New features are obtained by contracting the patch and kernel. This is repeated for a collection of C_{out} output channels stored along the first axis of W.

- Each output channel c_{out} is shifted by a bias $b_{c_{\text{out}}}$, stored in the C_{out}-dimensional vector $\boldsymbol{b}$.
- The output $\mathsf{Y} = \mathsf{X} \star \mathsf{W}$ with components $[\mathsf{Y}]_{c_{\text{out}},y_1,y_2}$ 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 $(C_{\text{out}}, Y_1, Y_2)$.

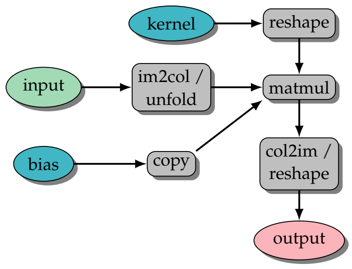


Figure A.4: Decomposition of the convolution operation's forward pass.

Example (index notation): A special case where input and output have the same spatial dimensions [63] uses a stride of one, kernel widths $K_1 = K_2 = 2K+1, (K \in \mathbb{N})$, and padding K. Elements of the filter $[\mathsf{W}]_{c_{\text{out}},c_{\text{in}},:,:}$ are addressed with the index set $\{-K, \dots, 0, \dots, K\} \times \{-K, \dots, 0, \dots, K\}$:

$$[\mathsf{Y}]_{c_{\text{out}},y_1,y_2} = \sum_{k_1=-K}^{K} \sum_{k_2=-K}^{K} [\mathsf{X}]_{c_{\text{in}},x_1+k_1,x_2+k_2} [\mathsf{W}]_{c_{\text{out}},c_{\text{in}},k_1,k_2} + [b]_{c_{\text{out}}} \,. \quad \text{(A.8)}$$

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

Convolution as Matrix Multiplication

Evaluating convolutions by sums of the form Equation (A.8) leads to poor memory locality [63]. For improved performance, the computation is mapped to a matrix multiplication [29]. 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 $[\![\cdot]\!]$ and the resulting matrix $[\![\mathsf{X}]\!]$ is of dimension $(C_{\text{in}}K_1K_2 \times P)$ (see left part of Figure A.3c showing $[\![\mathsf{X}]\!]^\top$). In other words, elements contracted with the kernel are stored along the first axis of $[\![\mathsf{X}]\!]$. $[\![\cdot]\!]$ is also referred to as `im2col` or `unfold` operation[2], and accounts for padding.

2: Our definition of the `unfold` operator slightly differs from [63], where flattened patches are stacked rowwise. This lets us achieve an analogous form to a linear layer. Conversion is achieved by transposition.

The kernel tensor W is reshaped into a $(C_{\text{out}} \times C_{\text{in}}K_1K_2)$ matrix $\boldsymbol{W}$, and elements of the bias vector vb are copied columnwise into a $(C_{\text{out}} \times P)$ matrix $\boldsymbol{B} = \boldsymbol{b}\boldsymbol{1}_P^\top$, where $\boldsymbol{1}_P$ is a P-dimensional vector of ones. Patchwise contractions are carried out by matrix multiplication and yield a matrix $\boldsymbol{Y}$ of shape (C_{out}, P) with $P = Y_1Y_2$,

$$\boldsymbol{Y} = \boldsymbol{W}[\![\mathsf{X}]\!] + \boldsymbol{B} \quad \text{(A.9)}$$

(Figure A.3c shows $\boldsymbol{W}^\top$, $[\![\mathsf{X}]\!]^\top$, and $\boldsymbol{Y}$ from left to right). Reshaping $\boldsymbol{Y}$ into a $(C_{\text{out}}, Y_1, Y_2)$ tensor, referred to as `col2im`, yields Y. Figure A.4 summarizes the outlined decomposition of the forward pass.

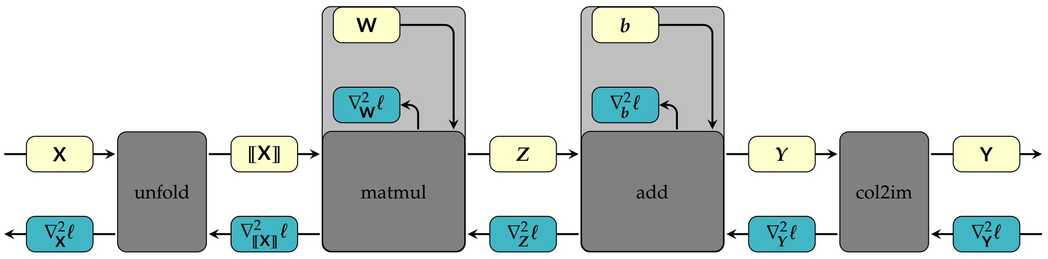


Figure A.5: Decomposition of convolution with notation for the study of curvature backpropagation.

A.4.4 HBP for Convolution

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

Reshape/`col2im`

The `col2im` operation takes a matrix $\boldsymbol{Y} \in \mathbb{R}^{C_{\text{out}} \times Y_1 Y_2}$ and reshapes it into the tensor $\mathbf{Y} \in \mathbb{R}^{C_{\text{out}} \times Y_1 \times Y_2}$. According to Appendix A.4.1, $\nabla^2_{\boldsymbol{Y}} \ell = \nabla^2_{\mathbf{Y}} \ell$.

Bias Hessian

Forward pass $\boldsymbol{Y} = \boldsymbol{Z} + \boldsymbol{B}$ and Equation (A.3b) imply $\nabla^2_{\boldsymbol{Y}} \ell = \nabla^2_{\boldsymbol{B}} \ell = \nabla^2_{\boldsymbol{Z}} \ell$. To obtain the Hessian *w.r.t.* $\boldsymbol{b}$ from $\nabla^2_{\boldsymbol{B}} \ell$, consider the columnwise copy operation $\boldsymbol{B}(\boldsymbol{b}) = \boldsymbol{b}\,\mathbf{1}_P^\top$, whose matrix differential is $\mathrm{d}\boldsymbol{B}(\boldsymbol{b}) = (\mathrm{d}\boldsymbol{b})\,\mathbf{1}_P^\top$. Vectorization yields $\mathrm{d}\operatorname{vec}\boldsymbol{B}(\boldsymbol{b}) = (\mathbf{1}_P \otimes \boldsymbol{I}_{C_{\text{out}}})\mathrm{d}\boldsymbol{b}$. Hence, the Jacobian is $\mathrm{J}_{\boldsymbol{b}}\boldsymbol{B}(\boldsymbol{b}) = \mathbf{1}_P \otimes \boldsymbol{I}_{C_{\text{out}}}$, and Equation (4.7) yields

$$\nabla^2_{\boldsymbol{b}} \ell = \left(\mathbf{1}_P \otimes \boldsymbol{I}_{C_{\text{out}}}\right)^\top \nabla^2_{\boldsymbol{B}} \ell \left(\mathbf{1}_P \otimes \boldsymbol{I}_{C_{\text{out}}}\right) .$$

This performs a linewise and columnwise summation over $\nabla^2_{\boldsymbol{B}} \ell$, summing entities that correspond to copies of the same entry of $\boldsymbol{b}$ in the matrix $\boldsymbol{B}$. It can also be regarded as a reshape of $\nabla^2_{\boldsymbol{B}} \ell$ into a $(C_{\text{out}}, P, C_{\text{out}}, P)$ tensor, which is then contracted over the second and fourth axis.

Weight Hessian

HBP for the matrix-matrix multiplication $\boldsymbol{Z}(\boldsymbol{W}, [\![\mathbf{X}]\!]) = \boldsymbol{W}[\![\mathbf{X}]\!]$ was discussed in Appendix A.2.1. The Jacobians are given by $\mathrm{J}_{[\![\mathbf{X}]\!]}\boldsymbol{Z}([\![\mathbf{X}]\!]) = \boldsymbol{I}_P \otimes \boldsymbol{W}$ and $\mathrm{J}_{\boldsymbol{W}}\boldsymbol{Z}(\boldsymbol{W}) = [\![\mathbf{X}]\!]^\top \otimes \boldsymbol{I}_S$ with the patch size $S = C_{\text{in}} K_1 K_2$. Hence, HBP for the weight matrix & unfolded input are

$$\begin{aligned}\nabla^2_{[\![\mathbf{X}]\!]} \ell &= \left(\boldsymbol{I}_P \otimes \boldsymbol{W}\right)^\top \nabla^2_{\boldsymbol{Z}} \ell \left(\boldsymbol{I}_P \otimes \boldsymbol{W}\right) ,\\ \nabla^2_{\boldsymbol{W}} \ell &= \left([\![\mathbf{X}]\!]^\top \otimes \boldsymbol{I}_S\right)^\top \nabla^2_{\boldsymbol{Z}} \ell \left([\![\mathbf{X}]\!]^\top \otimes \boldsymbol{I}_S\right) .\end{aligned}$$

From what has been said about the reshape operation in Appendix A.4.1, it follows that $\nabla^2_{\mathbf{W}} \ell = \nabla^2_{\boldsymbol{W}} \ell$.

Im2col/Unfold

The patch extraction operation $[\![\cdot]\!]$ copies all patch elements into the column of a matrix. It thus represents a selection of elements by an index map which is hard to express in notation. Numerically, it is 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 Appendix A.4.2.

Discussion

Although convolution can be understood as a matrix multiplication, the parameter Hessian is not identical to that of the linear layer discussed in Appendix A.2.1. The difference is due to the parameter sharing of the convolution. In the case of a linear layer $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, the Hessian of the weight matrix for a single sample possesses Kronecker structure [8, 21, 31], *i.e.* $\nabla^2_{\boldsymbol{W}}\ell = \boldsymbol{x} \otimes \boldsymbol{x}^\top \otimes \nabla^2_{\boldsymbol{z}}\ell$. For convolutional layers, however, it has been argued by [8] that block diagonals of the Hessian do not inherit the same structure. Rephrasing the forward pass equation A.9 in terms of vectorized quantities, we find

$$\operatorname{vec}\boldsymbol{Y} = (I_P \otimes \boldsymbol{W}) \operatorname{vec}[\![\mathbf{X}]\!] + \operatorname{vec}\boldsymbol{B}\,.$$

In this perspective, convolution corresponds to a fully-connected linear layer, with the additional constraints that the weight and bias matrix be circular. Defining $\hat{\boldsymbol{W}} := \boldsymbol{I}_P \otimes \boldsymbol{W}$, one then finds the Hessian $\nabla^2_{\hat{\boldsymbol{W}}}\ell$ to possess Kronecker structure. Parameterization with a kernel tensor encodes the circularity constraint in weight sharing.

For the kernel Hessian $\nabla^2_{\boldsymbol{W}}\ell$ to possess Kronecker structure, the output Hessian $\nabla^2_{\boldsymbol{Z}}\ell$ must 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 approximations introduced in [63] to obtain KFC.

A.5 Experimental Details

Fully-connected Neural Network

The same model as in [31] (see Table A.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 [57]. Bias terms are intialized to zero. Backpropagation of the Hessian uses approximation Equation (4.9) of Equation (A.6) to compute the curvature blocks $\overline{\nabla^2_{\boldsymbol{W}^{(l)}}\ell}$ and $\overline{\nabla^2_{\boldsymbol{b}^{(l)}}\ell}$.

Hyperparameters are chosen as follows to obtain consistent results with the original work: all runs shown in Figure 4.4 use a batch size of $|\mathbb{B}| = 500$. For SGD, the learning rate is assigned to $\eta = 0.1$ with momentum $\rho = 0.9$. Block-splitting experiments with the second-order method use the PCH-abs. All runs were performed with a learning rate $\eta = 0.1$ 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 $\epsilon_{\text{CG}} = 0.1$.

Table A.2: Model architectures under consideration. We use `Conv2d(in_channels, out_channels, kernel_size, padding)`, `ZeroPad2d(padding_left, padding_right, padding_top, padding_bottom)`, `Linear(in_features, out_features)`, and `MaxPool2d(kernel_size, stride)` as patterns to describe module hyperparameters. Convolution strides are always one. **(a)** FCNN used to extend the experiment in Chen et al. [31] (3 846 810 parameters). **(b)** CNN architecture (1 099 226 parameters). **(c)** DeepOBS 3c3d test problem with three convolutional and three dense layers (895 210 parameters). ReLU activation functions are replaced by sigmoids.

(a) FCNN (Figure 4.4)

#	Module
1	`Flatten()`
2	`Linear(3072, 1024)`
3	`Sigmoid()`
4	`Linear(1024, 512)`
5	`Sigmoid()`
6	`Linear(512, 256)`
7	`Sigmoid()`
8	`Linear(256, 128)`
9	`Sigmoid()`
10	`Linear(128, 64)`
11	`Sigmoid()`
12	`Linear(64, 32)`
13	`Sigmoid()`
14	`Linear(32, 16)`
15	`Sigmoid()`
16	`Linear(16, 10)`

(b) CNN (Figure 4.5)

#	Module
1	`Conv2d(3, 16, 3, 1)`
2	`Sigmoid()`
3	`Conv2d(16, 16, 3, 1)`
4	`Sigmoid()`
5	`MaxPool2d(2, 2)`
6	`Conv2d(16, 32, 3, 1)`
7	`Sigmoid()`
8	`Conv2d(32, 32, 3, 1)`
9	`Sigmoid()`
10	`MaxPool2d(2, 2)`
11	`Flatten()`
12	`Linear(2048, 512)`
13	`Sigmoid()`
14	`Linear(512, 64)`
15	`Sigmoid()`
16	`Linear(64, 10)`

(c) DeepOBS 3c3d (Figure A.6)

#	Module
1	`Conv2d(3, 64, 5, 0)`
2	`Sigmoid()`
3	`ZeroPad2d(0, 1, 0, 1)`
4	`MaxPool2d(3, 2)`
5	`Conv2d(64, 96, 3, 0)`
6	`Sigmoid()`
7	`ZeroPad2d(0, 1, 0, 1)`
8	`MaxPool2d(3, 2)`
9	`ZeroPad2d(1, 1, 1, 1)`
10	`Conv2d(96, 128, 3, 0)`
11	`Sigmoid()`
12	`ZeroPad2d(0, 1, 0, 1)`
13	`MaxPool2d(3, 2)`
14	`Flatten()`
15	`Linear(1152, 512)`
16	`Sigmoid()`
17	`Linear(512, 256)`
18	`Sigmoid()`
19	`Linear(256, 10)`

Convolutional Neural Net

The CNN architecture shown in Table A.2b is trained on a hyperparameter grid. Runs with smallest final training loss are selected to rerun on different random seeds. The curves in Figure 4.5b represent mean values and standard deviations for ten different realizations over the random seed. All layer parameters were initialized with PyTorch's default.

For the first-order methods (SGD, Adam), we considered batch sizes $|\mathbb{B}| \in \{100, 200, 500\}$. For SGD, momentum ρ was tuned over the set $\{0, 0.45, 0.9\}$. Although we varied the learning rate over a large range of values $\eta \in \left\{10^{-3}, 10^{-2}, 0.1, 1, 10\right\}$, losses kept plateauing and did not decrease. In particular, the loss even increased for the large learning rates. For Adam, we only vary the learning rate $\eta \in \left\{10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10\right\}$.

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

To compare with another second-order method, we experimented with a public PyTorch implementation of the KFAC optimizer [63, 109] (`github.com/alecwangcq/KFAC-Pytorch`). All hyperparameters were kept at their default setting. The learning rate was varied over $\eta \in \left\{10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10\right\}$.

The hyperparameters of results shown in Figure 4.5 read as follows:

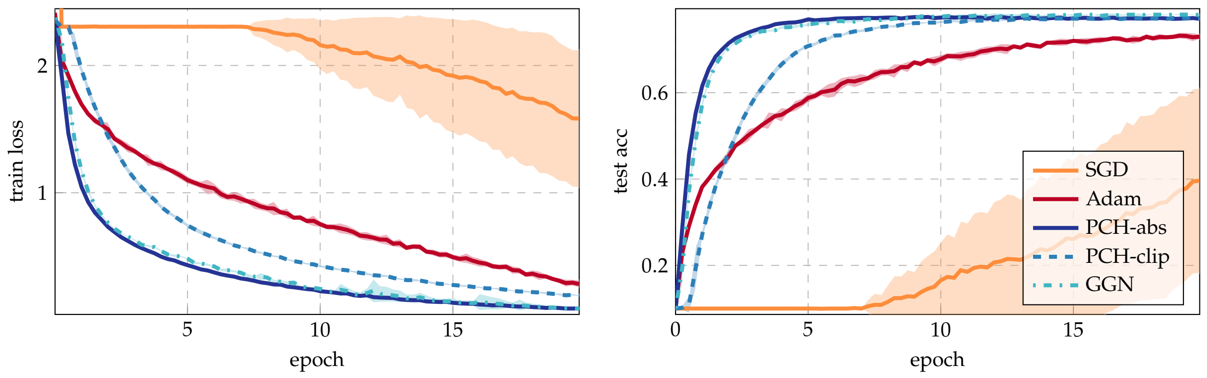


Figure A.6: Comparison of SGD, Adam, and Newton-style methods with different exact curvature matrix-vector products (HBP). The architecture is the DeepOBS 3c3d network [146] with sigmoid activations (Table A.2c).

- SGD ($|\mathbb{B}| = 100, \eta = 10^{-3}, \rho = 0.9$). The particular choice of these hyperparameters is artificial. This run is representative for SGD, which does not achieve any progress at all.
- Adam ($|\mathbb{B}| = 100, \eta = 10^{-3}$)
- KFAC ($|\mathbb{B}| = 500, \eta = 0.1$)
- PCH-abs ($|\mathbb{B}| = 1000, \eta = 0.2, \alpha = 10^{-3}$),
  PCH-clip ($|\mathbb{B}| = 1000, \eta = 0.1, \alpha = 10^{-4}$)
- GGN, α_1 ($|\mathbb{B}| = 1000, \eta = 0.1, \alpha = 10^{-4}$). This run does not yield the minimum training loss on the grid; it is shown to illustrate that the second-order method can escape the flat regions in early stages.
- GGN, α_2 ($|\mathbb{B}| = 1000, \eta = 0.1, \alpha = 10^{-3}$). Compared to α_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 training stages.

Additional experiment: Another experiment conducted with HBP considers the 3c3d architecture (Table A.2c) of DeepOBS [146] 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 A.6. In particular, the hyperparameter settings for each competitor are:

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

Additional Material for Chapter 5 B.

B.1 BackPACK Extensions

This section provides technical details on BackPACK's quantities.

Notation

Consider an arbitrary module $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ of a network $l = 1, \dots, L$, parameterized by $\boldsymbol{\theta}^{(l)}$. It transforms the output of its parent layer for sample n, $\boldsymbol{z}^{(l-1)}_n$, to its output $\boldsymbol{z}^{(l)}_n$, *i.e.*

$$\boldsymbol{z}^{(l)}_n = f^{(l)}_{\boldsymbol{\theta}^{(l)}}(\boldsymbol{z}^{(l-1)}_n) , \qquad n = 1, \dots, N , \tag{B.1}$$

where N is the number of samples. In particular, $\boldsymbol{z}^{(0)}_n = \boldsymbol{x}_n$ and $\boldsymbol{z}^{(L)}_n(\boldsymbol{\theta}) = \boldsymbol{f}_n$, where $f_{\boldsymbol{\theta}}$ is the transformation of the whole network. The dimension of the hidden layer l's output $\boldsymbol{z}^{(l)}_n$ is written $h^{(l)}$ and $\boldsymbol{\theta}^{(l)}$ is of dimension $d^{(l)}$. The dimension of the network output, the prediction $\boldsymbol{z}^{(L)}$, is $h^{(L)} = C$. For classification, C corresponds to the number of classes.

All quantities are assumed to be vector-shaped. For image-processing transformations that usually act on tensor-shaped inputs, we can reduce to the vector scenario by vectorizing all quantities (Definition 2.2); this discussion does not rely on a specific flattening scheme. However, for an efficient implementation, vectorization should match the layout of the memory of the underlying arrays.

Jacobian

The Jacobian (Definition 2.4) $\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}$ of a vector $\boldsymbol{b} \in \mathbb{R}^B$ *w.r.t.* another vector $\boldsymbol{a} \in \mathbb{R}^A$ is a $[B \times A]$ matrix of partial derivatives, $[\mathrm{J}_{\boldsymbol{a}}\boldsymbol{b}]_{i,j} = \partial [\boldsymbol{b}]_i / \partial [\boldsymbol{a}]_j$.

B.1.1 First-order Quantities

The basis for the extraction of additional information about first-order derivatives is Equation (5.3), which we state again for multiple samples,

$$\nabla_{\boldsymbol{\theta}^{(l)}} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}_n \right)^\top \left(\nabla_{\boldsymbol{z}^{(l)}_n} \ell_n(\boldsymbol{\theta}) \right) .$$

During the backpropagation step of module l, we have access to $\nabla_{\boldsymbol{z}^{(l)}_n} \ell(\boldsymbol{\theta})$, $n = 1, \dots, N$. To extract more quantities involving the gradient, we use additional information about the transformation $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ in our custom implementation of Jacobian $\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}_n$ and transposed Jacobian $(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}_n)^\top$.

Individual Gradients

The contribution of each sample to the overall gradient, $1/N \nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta})$, is computed by application of the transposed Jacobian,

$$\frac{1}{N} \nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta}) = \frac{1}{N} \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)} \right)^\top \left(\nabla_{\boldsymbol{z}_n^{(l)}} \ell_n(\boldsymbol{\theta}) \right) , \qquad n = 1, \dots, N . \tag{B.2}$$

For each parameter $\boldsymbol{\theta}^{(l)}$ the individual gradients are of size $[N \times d^{(l)}]$.

Individual Gradient L_2 Norm

The quantity $\|1/N \nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta})\|_2^2$, for $n = 1, ..., N$, could be extracted from the individual gradients (Equation (B.2)) as

$$\begin{aligned} &\left\| \frac{1}{N} \nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta}) \right\|_2^2 \\ &\quad = \left[\frac{1}{N} \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)} \right)^\top \left(\nabla_{\boldsymbol{z}_n^{(l)}} \ell_n(\boldsymbol{\theta}) \right) \right]^\top \left[\frac{1}{N} \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)} \right)^\top \left(\nabla_{\boldsymbol{z}_n^{(l)}} \ell_n(\boldsymbol{\theta}) \right) \right] , \end{aligned}$$

which is an N-dimensional object per parameter $\boldsymbol{\theta}^{(l)}$. However, this is memory inefficient as the individual gradients are $[N \times d^{(l)}]$. BackPACK circumvents this by using the Jacobian's structure whenever possible.

For a specific example, take a linear layer with parameters $\boldsymbol{\theta}$ as an $[A \times B]$ matrix. The layer transforms the inputs $\{\boldsymbol{z}_n^{(l-1)}\}_{n=1}^N$, an $[N \times A]$ matrix which we will now refer to as $\boldsymbol{A}$. During the backward pass, it receives the gradient of the individual losses *w.r.t.* its output, $\{1/N \nabla_{\boldsymbol{z}_n^{(l)}} \ell_n\}_{n=1}^N$, as an $[N \times B]$ matrix which we will refer to as $\boldsymbol{B}$. The overall gradient, an $[A \times B]$ matrix, can be computed as $\boldsymbol{A}^\top \boldsymbol{B}$, and the individual gradients are a set of N $[A \times B]$ matrices, $\{[\boldsymbol{A}]_{n,:} [\boldsymbol{B}]_{n,:}^\top\}_{n=1}^N$. We want to avoid storing that information. To reduce the memory requirement, note that the individual gradient norm can be written as

$$\left\| \frac{1}{N} \nabla_{\boldsymbol{\theta}} \ell_n \right\|^2 = \sum_i \sum_j ([\boldsymbol{A}]_{n,i} [\boldsymbol{B}]_{n,j})^2 ,$$

and that the summation can be done independently for each matrix, as $\sum_i \sum_j ([\boldsymbol{A}]_{n,i} [\boldsymbol{B}]_{n,j})^2 = (\sum_i [\boldsymbol{A}]_{n,i})^2 (\sum_j [\boldsymbol{B}]_{n,j}^2)$. Therefore, we can square each matrix (element-wise) and sum over non-batch dimensions. This yields vectors $\boldsymbol{a}, \boldsymbol{b}$ of N elements, where $[\boldsymbol{a}]_n = \sum_i [\boldsymbol{A}]_{n,i}^2$. The individual gradients' L_2 norm is then given by $\boldsymbol{a} \odot \boldsymbol{b}$.

Second moment

The gradient second moment (or more specifically, the diagonal of the second moment) is the sum of the squared elements of the individual gradients in a mini-batch, *i.e.*

$$\frac{1}{N} \sum_{n=1}^N \left[\nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta}) \right]_j^2 , \qquad j = 1, \dots, d^{(l)} . \tag{B.3}$$

It can be used to evaluate the variance of individual elements of the gradient (see below). The second moment is of dimension $d^{(l)}$, like the layer parameter $\boldsymbol{\theta}^{(l)}$. Similarly to the L_2 norm, it can be computed from individual gradients, but is more efficiently computed implicitly.

Revisiting the linear layer example from the individual L_2 norm computation, the second moment of the parameters $\boldsymbol{\theta}[i, j]$ is given by $\sum_n ([\boldsymbol{A}]_{n,i}[\boldsymbol{B}]_{n,j})^2$, which can be directly computed by taking the element-wise square of $\boldsymbol{A}$ and $\boldsymbol{B}$, $\boldsymbol{A}^{\odot 2}$, $\boldsymbol{B}^{\odot 2}$, and computing $(\boldsymbol{A}^{\odot 2})^\top \boldsymbol{B}^{\odot 2}$.

Variance

Gradient variances over a mini-batch (or more precisely, the covariance diagonal) are computed from the second moment and the gradient,

$$\frac{1}{N} \sum_{n=1}^{N} \left[\nabla_{\boldsymbol{\theta}^{(l)}} \ell_n(\boldsymbol{\theta})\right]_j^2 - \left[\nabla_{\boldsymbol{\theta}^{(l)}} \mathcal{L}(\boldsymbol{\theta})\right]_j^2 , \qquad j = 1, \dots, d^{(l)} . \tag{B.4}$$

The element-wise gradient variance is of same dimension as the layer parameter $\boldsymbol{\theta}^{(l)}$, *i.e.* $d^{(l)}$.

B.1.2 Second-order Quantities Based on the GGN

Backpropagation for the GGN's Block Diagonal

The computation of quantities that originate from approximations of the Hessian require an additional backward pass (see [37]). Most curvature approximations supported by BackPACK rely on the generalized Gauss-Newton (GGN) matrix [148] from Equation (3.15b)

$$\boldsymbol{G}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top \nabla^2_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) (\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n) . \tag{B.5}$$

One interpretation of the GGN is that it corresponds to the empirical risk Hessian when the model is replaced with its first-order Taylor expansion, *i.e.* by linearizing the network and ignoring second-order effects. Hence, the effect of module curvature in the recursive scheme of Equation (5.4) can be ignored to obtain the simpler expression

$$\begin{aligned} \boldsymbol{G}(\boldsymbol{\theta}^{(l)}) &= \frac{1}{N} \sum_{n=1}^{N} (\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{f}_n)^\top \nabla^2_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) (\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{f}_n) \\ &= \frac{1}{N} \sum_{n=1}^{N} \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right)^\top \boldsymbol{G}(\boldsymbol{z}_n^{(l)}) \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right) \end{aligned} \tag{B.6}$$

for the exact block diagonal of the full GGN. In analogy to $\boldsymbol{G}(\boldsymbol{\theta}^{(l)})$ we have introduced the $[d^{(l)} \times d^{(l)}]$-dimensional quantity

$$\boldsymbol{G}(\boldsymbol{z}_n^{(l)}) = \left(\mathrm{J}_{\boldsymbol{z}_n^{(l)}} \boldsymbol{f}_n\right)^\top \nabla^2_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) \left(\mathrm{J}_{\boldsymbol{z}_n^{(l)}} \boldsymbol{f}_n\right)$$

that needs to be backpropagated following Equation (5.4) as

$$G(z_n^{(l-1)}) = \left(\mathrm{J}_{z_n^{(l-1)}} z_n^{(l)}\right)^\top G(z_n^{(l)}) \left(\mathrm{J}_{z_n^{(l-1)}} z_n^{(l)}\right), \qquad l = 1, \dots, L, \tag{B.7a}$$

initialized with the loss Hessian *w.r.t.* to the network prediction, *i.e.*

$$G(z_n^{(L)}) = \nabla^2_{f_n} \ell(f_n, y_n). \tag{B.7b}$$

Although this scheme is exact, it is computationally infeasible as it requires backpropagation of N $[h^{(l)} \times h^{(l)}]$ matrices between layer $l+1$ and l. Even for small N, this is impossible for nets with large convolutions.

As an example, the first layer of the All-CNN-C network outputs 29×29 images with 96 channels, which already gives $h^{(l)} = 80{,}736$, which leads to half a Gigabyte per sample. Moreover, storing all the $[d^{(l)} \times d^{(l)}]$-dimensional blocks $G(\theta^{(l)})$ is not possible. BackPACK implements different approximation strategies, developed by Martens and Grosse [109] and Botev et al. [21] that address both of these complexity issues from different perspectives.

Symmetric Factorization Scheme

One way to improve the memory footprint of the backpropagated matrices in the case where the model prediction's dimension C (the number of classes in an image classification task) is small compared to all hidden features $h^{(l)}$ is to propagate a symmetric factorization of the GGN instead. It relies on the observation that if the loss function itself is convex, even though its composition with the network might not be, its Hessian *w.r.t.* the network output can be decomposed as (*e.g.* Example 5.1)

$$\nabla^2_{f_n} \ell(f_n, y_n) = S(z_n^{(L)}) S(z_n^{(L)})^\top \tag{B.8}$$

with the $[C \times C]$-dimensional matrix factorization of the loss Hessian, $S(z_n^{(L)})$, for sample n. Consequently, the GGN in Equation (B.5) reduces to an outer product,

$$G(\theta) = \frac{1}{N} \sum_{n=1}^N \left[(\mathrm{J}_\theta f_n)^\top S(z_n^{(L)}) \right] \left[(\mathrm{J}_\theta f_n)^\top S(z_n^{(L)}) \right]^\top. \tag{B.9}$$

The analogue for diagonal blocks follows from Equation (B.6) and reads

$$G(\theta^{(l)}) = \frac{1}{N} \sum_{n=1}^N \left[\left(\mathrm{J}_{\theta^{(l)}} z_n^{(l)}\right)^\top S(z_n^{(l)}) \right] \left[\left(\mathrm{J}_{\theta^{(l)}} z_n^{(l)}\right)^\top S(z_n^{(l)}) \right]^\top, \tag{B.10}$$

where we defined the $[h^{(l)} \times C]$-dimensional matrix square root $S(z_n^{(l)}) := (\mathrm{J}_{z_n^{(l)}} f_n)^\top S(z_n^{(L)})$. Instead of having layer l backpropagate N objects of shape $[h^{(l)} \times h^{(l)}]$ according to Equation (B.7), we instead backpropagate the matrix square root via

$$S(z_n^{(l-1)}) = \left(\mathrm{J}_{z_n^{(l-1)}} z_n^{(l)}\right)^\top S(z_n^{(l)}), \qquad l = 1, \dots, L, \tag{B.11}$$

starting with Equation (B.8). This reduces the backpropagated matrix of layer l to $[h^{(l)} \times C]$ for each sample.

Diagonal Curvature Approximations

Diagonal of the GGN (DiagGGN): The factorization trick for the loss Hessian reduces the size of the backpropagated quantities, but does not address the intractable size of the GGN diagonal blocks $\boldsymbol{G}(\boldsymbol{\theta}^{(l)})$. In BackPACK, we can extract $\operatorname{diag}(\boldsymbol{G}(\boldsymbol{\theta}^{(l)}))$ given the backpropagated quantities $\boldsymbol{S}(\boldsymbol{z}_n^{(l)}),\ l = 1, \dots, N$, without building up the matrix representation of Equation (B.10). In particular, we compute

$$\begin{aligned}
&\operatorname{diag}\left[\boldsymbol{G}(\boldsymbol{\theta}^{(l)})\right] \\
&= \frac{1}{N}\sum_{n=1}^{N} \operatorname{diag}\left\{\left[\left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}}\boldsymbol{z}_n^{(l)}\right)^\top \boldsymbol{S}(\boldsymbol{z}_n^{(l)})\right]\left[\left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}}\boldsymbol{z}_n^{(l)}\right)^\top \boldsymbol{S}(\boldsymbol{z}_n^{(l)})\right]^\top\right\}.
\end{aligned} \tag{B.12}$$

Diagonal of the GGN with MC-sampled loss Hessian (DiagGGN-MC): We use the same backpropagation strategy of Equation (B.11), replacing the symmetric factorization of Equation (B.8) with an approximation by a smaller matrix $\tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(L)})$ of size $[C \times \tilde{C}]$ and $\tilde{C} < C$ (*e.g.* Example 5.2),

$$\nabla^2_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) \approx \tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(L)}) \left(\tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(L)})\right)^\top. \tag{B.13}$$

This further reduces the size of backpropagated curvature quantities. Martens and Grosse [109] introduced such a sampling scheme with KFAC based on the connection between the GGN and the Fisher. Most loss functions used in machine learning have a probabilistic interpretation as negative log-likelihood of a probabilistic model (see Section 2.1.3). The squared error of regression is equivalent to a Gaussian noise assumption and the cross-entropy is linked to the categorical distribution. In this case, the loss Hessian *w.r.t.* the network output is equal, in expectation, to the outer products of gradients *if the target is sampled according to a particular distribution*, $q(\boldsymbol{y} \mid \boldsymbol{f})$, defined by the network output. Sampling targets $\hat{\boldsymbol{y}} \sim q(\boldsymbol{y} \mid \boldsymbol{f})$ for a datum $(\boldsymbol{x}, \boldsymbol{y})$ with $\boldsymbol{f} := f_{\boldsymbol{\theta}}(\boldsymbol{x})$, we have

$$\mathbb{E}_{\hat{\boldsymbol{y}} \sim q(\cdot \mid \boldsymbol{f})}\left[\left(\nabla_{\boldsymbol{f}} \ell(\boldsymbol{f}, \hat{\boldsymbol{y}})\right)\left(\nabla_{\boldsymbol{f}} \ell(\boldsymbol{f}, \hat{\boldsymbol{y}})\right)^\top\right] = \nabla^2_{\boldsymbol{f}} \ell(\boldsymbol{f}, \boldsymbol{y}). \tag{B.14}$$

Sampling one such gradient leads to a rank-1 MC approximation of the loss Hessian. With the substitution $\boldsymbol{S} \leftrightarrow \tilde{\boldsymbol{S}}$, we compute an MC approximation of the GGN diagonal in BackPACK as

$$\begin{aligned}
&\operatorname{diag}\left[\boldsymbol{G}(\boldsymbol{\theta}^{(l)})\right] \\
&\approx \frac{1}{N}\sum_{n=1}^{N} \operatorname{diag}\left\{\left[\left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}}\boldsymbol{z}_n^{(l)}\right)^\top \tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(l)})\right]\left[\left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}}\boldsymbol{z}_n^{(l)}\right)^\top \tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(l)})\right]^\top\right\}.
\end{aligned} \tag{B.15}$$

Kronecker-factored Curvature Approximations

A different approach to reduce memory complexity of the GGN blocks $\boldsymbol{G}(\boldsymbol{\theta}^{(l)})$, apart from diagonal curvature approximations, is representing them as Kronecker products (KFAC for linear [109] and convolutional layers [63] KFLR and KFRA for linear layers by [21]),

$$\boldsymbol{G}(\boldsymbol{\theta}^{(l)}) \approx \boldsymbol{A}^{(l)} \otimes \boldsymbol{B}^{(l)}. \tag{B.16}$$

For both linear and convolutional layers, the first Kronecker factor $\boldsymbol{A}^{(l)}$ is obtained from the inputs $\boldsymbol{z}_n^{(l-1)}$ to layer l. Instead of repeating the technical details of the aforementioned references, we will focus on how they differ in (i) the backpropagated quantities and (ii) the backpropagation strategy. As a result, we will be able to extend KFLR and KFRA to CNNs[1].

1: We keep the PyTorch convention that weights and bias are treated as separate parameters. For the bias terms, we can store the full matrix representation of the GGN. This factor reappears in the Kronecker factorization of the GGN *w.r.t.* the weights.

KFAC & KFLR: KFAC uses an MC-sampled estimate of the loss Hessian with a square root factorization $\tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(L)})$ like in Equation (B.13). The backpropagation is equivalent to the computation of the GGN diagonal. For the weights of a linear layer l, the second Kronecker term is

$$\boldsymbol{B}_{\text{KFAC}}^{(l)} = \frac{1}{N}\sum_{n=1}^{N} \tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(l)}) \left(\tilde{\boldsymbol{S}}(\boldsymbol{z}_n^{(l)})\right)^\top ,$$

which at the same time corresponds to the GGN of the layer's bias[2].

2: In the case of convolutions, one has to sum over the spatial indices of a single channel of $z_n^{(l)}$ as the bias is added to an entire channel, see [63] for details.

In contrast to KFAC, KFLR backpropagates the exact square root factorization $\boldsymbol{S}(\boldsymbol{z}_n^{(L)})$, *i.e.* for the weights of a linear layer[2] (details in [21])

$$\boldsymbol{B}_{\text{KFLR}}^{(l)} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{S}(\boldsymbol{z}_n^{(l)}) \left(\boldsymbol{S}(\boldsymbol{z}_n^{(l)})\right)^\top .$$

KFRA: The backpropagation strategy for KFRA eliminates the scaling of the backpropagated curvature quantities with the batch size N in Equation (B.7). Instead of layer l receiving the N exact $[h^{(l)} \times h^{(l)}]$ matrices $\boldsymbol{G}(\boldsymbol{z}_n^{(l)}),\ n = 1, \dots, N$, only a single averaged object, denoted $\overline{\boldsymbol{G}}^{(l)}$, is used as an approximation. In particular, the recursion changes to

$$\overline{\boldsymbol{G}}^{(l-1)} = \frac{1}{N}\sum_{n=1}^{N} \left(\mathrm{J}_{\boldsymbol{z}_n^{(l-1)}} \boldsymbol{z}_n^{(l)}\right)^\top \overline{\boldsymbol{G}}^{(l)} \left(\mathrm{J}_{\boldsymbol{z}_n^{(l-1)}} \boldsymbol{z}_n^{(l)}\right) , \qquad l = 1, \dots, L , \tag{B.17a}$$

and is initialized with the batch-averaged loss Hessian

$$\overline{\boldsymbol{G}}^{(L)} = \frac{1}{N}\sum_{n=1}^{N} \nabla^2_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) . \tag{B.17b}$$

For a linear layer, KFRA uses[2] (see [21] for more details)

$$\boldsymbol{B}_{\text{KFRA}}^{(l)} = \overline{\boldsymbol{G}}^{(l)} .$$

B.1.3 The Exact Hessian Diagonal

For neural networks consisting only of piecewise linear activation functions, computing the diagonal of the Hessian is equivalent to computing the GGN diagonal. This is because for these activations the second term in the Hessian backpropagation recursion (Equation (5.4)) vanishes.

However, for activation functions with non-vanishing second derivative, these residual terms have to be accounted for in the backpropagation.

The Hessian backpropagation for module l reads

$$\nabla^2_{\boldsymbol{\theta}^{(l)}} \ell(\boldsymbol{\theta}) = \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right)^\top \nabla^2_{\boldsymbol{z}_n^{(l)}} \ell(\boldsymbol{\theta}) \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right) + \boldsymbol{R}_n^{(l)}(\boldsymbol{\theta}^{(l)}) \,, \tag{B.18a}$$

$$\nabla^2_{\boldsymbol{z}_n^{(l-1)}} \ell(\boldsymbol{\theta}) = \left(\mathrm{J}_{\boldsymbol{z}_n^{(l-1)}} \boldsymbol{z}_n^{(l)}\right)^\top \nabla^2_{\boldsymbol{z}_n^{(l)}} \ell(\boldsymbol{\theta}) \left(\mathrm{J}_{\boldsymbol{z}_n^{(l-1)}} \boldsymbol{z}_n^{(l)}\right) + \boldsymbol{R}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}) \,, \tag{B.18b}$$

for $n = 1, \dots, N$. Those $[h^{(l)} \times h^{(l)}]$ residuals are

$$\boldsymbol{R}_n^{(l)}(\boldsymbol{\theta}^{(l)}) = \sum_j \left(\nabla^2_{\boldsymbol{\theta}^{(l)}} [\boldsymbol{z}_n^{(l)}]_j\right) \left[\nabla_{\boldsymbol{z}_n^{(l)}} \ell(\boldsymbol{\theta})\right]_j \,,$$

$$\boldsymbol{R}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}) = \sum_j \left(\nabla^2_{\boldsymbol{z}_n^{(l-1)}} [\boldsymbol{z}_n^{(l)}]_j\right) \left[\nabla_{\boldsymbol{z}_n^{(l)}} \ell(\boldsymbol{\theta})\right]_j \,.$$

Common parameterized layers (linear and convolution) have $\boldsymbol{R}_n^{(l)}(\boldsymbol{\theta}^{(l)}) = \boldsymbol{0}$. For elementwise activations, $\boldsymbol{R}_n^{(l)}(\boldsymbol{z}_n^{(l-1)})$ are diagonal matrices.

Storing these quantities becomes very memory-intensive for high-dimensional nonlinear activation layers. In BackPACK, this complexity is reduced by application of the aforementioned matrix square root factorization trick. To do so, we divide the symmetric factorization of $\boldsymbol{R}_n^{(l)}(\boldsymbol{z}_n^{(l-1)})$ into the positive- and negative-definite terms

$$\begin{aligned} &\boldsymbol{R}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}) \\ &\quad = \boldsymbol{P}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}) \left(\boldsymbol{P}_n^{(l)}(\boldsymbol{z}_n^{(l-1)})\right)^\top - \boldsymbol{N}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}) \left(\boldsymbol{N}_n^{(l)}(\boldsymbol{z}_n^{(l-1)})\right)^\top \,. \end{aligned} \tag{B.19}$$

$\boldsymbol{P}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}), \boldsymbol{N}_n^{(l)}(\boldsymbol{z}_n^{(l-1)})$ represent the matrix square root of $\boldsymbol{R}_n^{(l)}(\boldsymbol{z}_n^{(l-1)})$ projected on its positive and negative eigenspace, respectively.

This composition allows for the extension of the GGN backpropagation: in addition to $\boldsymbol{S}(\boldsymbol{z}_n^{(l)})$, the residual decompositions $\boldsymbol{P}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}), \boldsymbol{N}_n^{(l)}(\boldsymbol{z}_n^{(l-1)})$ also have to be backpropagated according to Equation (B.11). All diagonals are extracted from the backpropagated matrix square roots (see Equation (B.12)). Diagonals from decompositions in the negative residual eigenspace have to be weighted by a factor of -1 before summation.

In terms of complexity, one backpropagation for $\boldsymbol{R}_n^{(l)}(\boldsymbol{z}^{(l-1)})$ changes the dimensionality as follows

$$\boldsymbol{R}_n^{(l)}(\boldsymbol{z}^{(l-1)}) : \quad [h^{(l)} \times h^{(l)}] \to [h^{(l-1)} \times h^{(l-1)}] \to [h^{(l-2)} \times h^{(l-2)}] \to \dots \,.$$

With the square root factorization, one instead obtains

$$\begin{aligned} \boldsymbol{P}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}) : &\quad [h^{(l)} \times h^{(l)}] \to [h^{(l-1)} \times h^{(l)}] \to [h^{(l-2)} \times h^{(l)}] \to \dots \,, \\ \boldsymbol{N}_n^{(l)}(\boldsymbol{z}_n^{(l-1)}) : &\quad [h^{(l)} \times h^{(l)}] \to [h^{(l-1)} \times h^{(l)}] \to [h^{(l-2)} \times h^{(l)}] \to \dots \,. \end{aligned}$$

Roughly speaking, this is more efficient if the hidden dimension of a nonlinear activation layer deceeds the net's largest hidden dimension.

Algorithm

Consider one backpropagation step of module l. Assume $\boldsymbol{R}_n^{(l)}(\boldsymbol{\theta}^{(l)}) = \boldsymbol{0}$, *i.e.* a linear, convolution, or non-parameterized layer. Then the following computations are performed in the protocol for the diagonal Hessian:

- Receive the following from the child module $l+1$ (for $n = 1, \dots, N$):

$$
\begin{aligned}
\Phi = \Big\{ & \boldsymbol{S}(\boldsymbol{z}_n^{(l)}) \,, \\
& \boldsymbol{P}_n^{(l+1)}(\boldsymbol{z}_n^{(l)}) \,, \\
& \boldsymbol{N}_n^{(l+1)}(\boldsymbol{z}_n^{(l)}) \,, \\
& (\mathrm{J}_{\boldsymbol{z}_n^{(l)}} \boldsymbol{z}_n^{(l+1)})^\top \boldsymbol{P}_n^{(l+2)}(\boldsymbol{z}_n^{(l+1)}) \,, \\
& (\mathrm{J}_{\boldsymbol{z}_n^{(l)}} \boldsymbol{z}_n^{(l+1)})^\top \boldsymbol{N}_n^{(l+2)}(\boldsymbol{z}_n^{(l+1)}) \,, \\
& \dots \\
& (\mathrm{J}_{\boldsymbol{z}_n^{(l)}} \boldsymbol{z}_n^{(l+1)})^\top (\mathrm{J}_{\boldsymbol{z}_n^{(l+1)}} \boldsymbol{z}_n^{(l+2)})^\top \dots (\mathrm{J}_{\boldsymbol{z}_n^{(L-3)}} \boldsymbol{z}_n^{(L-2)})^\top \boldsymbol{P}_n^{(L-1)}(\boldsymbol{z}_n^{(L-2)}) \,, \\
& (\mathrm{J}_{\boldsymbol{z}_n^{(l)}} \boldsymbol{z}_n^{(l+1)})^\top (\mathrm{J}_{\boldsymbol{z}_n^{(l+1)}} \boldsymbol{z}_n^{(l+2)})^\top \dots (\mathrm{J}_{\boldsymbol{z}_n^{(L-3)}} \boldsymbol{z}_n^{(L-2)})^\top \boldsymbol{N}_n^{(L-1)}(\boldsymbol{z}_n^{(L-2)}) \Big\}
\end{aligned}
$$

- Extract the parameter Hessian diagonal $\mathrm{diag}(\nabla^2_{\boldsymbol{\theta}^{(l)}} \mathcal{L}(\boldsymbol{\theta}))$
  - For each quantity $\boldsymbol{A} \in \Phi$ extract the diagonal from the square root factorization and sum over the samples, *i.e.* compute

$$
\frac{1}{N} \sum_{n=1}^{N} \mathrm{diag} \left\{ \left[\left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)} \right)^\top \boldsymbol{A}_n \right] \left[\left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)} \right)^\top \boldsymbol{A}_n \right]^\top \right\} .
$$

    Multiply the expression by -1 if $\boldsymbol{A}$ stems from backpropagation of a residual's negative eigenspace's factorization.
  - Sum all expressions to obtain the block Hessian's diagonal $\mathrm{diag}(\nabla^2_{\boldsymbol{\theta}^{(l)}} \mathcal{L}(\boldsymbol{\theta}))$
- Backpropagate the received quantities to the parent module $l-1$
  - For each quantity $\boldsymbol{A}_n \in \Phi$, apply $(\mathrm{J}_{\boldsymbol{z}_n^{(l-1)}} \boldsymbol{z}_n^{(l)})^\top \boldsymbol{A}_n$
  - Append $\boldsymbol{P}_n^{(l+1)}(\boldsymbol{z}_n^{(l)})$ and $\boldsymbol{N}_n^{(l+1)}(\boldsymbol{z}_n^{(l)})$ to Φ

B.2 Benchmark Details

KFAC versus KFLR: As the KFLR of Botev et al. [21] is orders of magnitude more expensive to compute than the KFAC of Martens and Grosse [109] on CIFAR-100, it was not included in the main plot. This is not an implementation error; it follows from the definition of those methods. To approximate the GGN, $\boldsymbol{G}(\boldsymbol{\theta}) = {}^1\!/\!_N \sum_n [\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n]^\top \nabla^2_{\boldsymbol{f}_n} \ell_n \, [\mathrm{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n]$, KFAC uses a rank-1 approximation for each of the inner Hessian $\nabla^2_{\boldsymbol{f}_n} \ell_n \approx \boldsymbol{s}_n \boldsymbol{s}_n^\top$, and needs to propagate a *vector* through the computation graph for each sample. KFLR uses the complete inner Hessian instead. For CIFAR-100, the network has 100 output nodes—one for each class—and the inner Hessians are $[100 \times 100]$ matrices. KFLR needs to propagate a *matrix* through the computation graph for each sample, which is 100× more expensive as shown in Figure B.1.

GGN diagonal versus Hessian diagonal: Most nets used in deep learning use ReLU activations. ReLU functions have no *curvature* as they are piecewise linear. Because of this, the diagonal of the GGN is

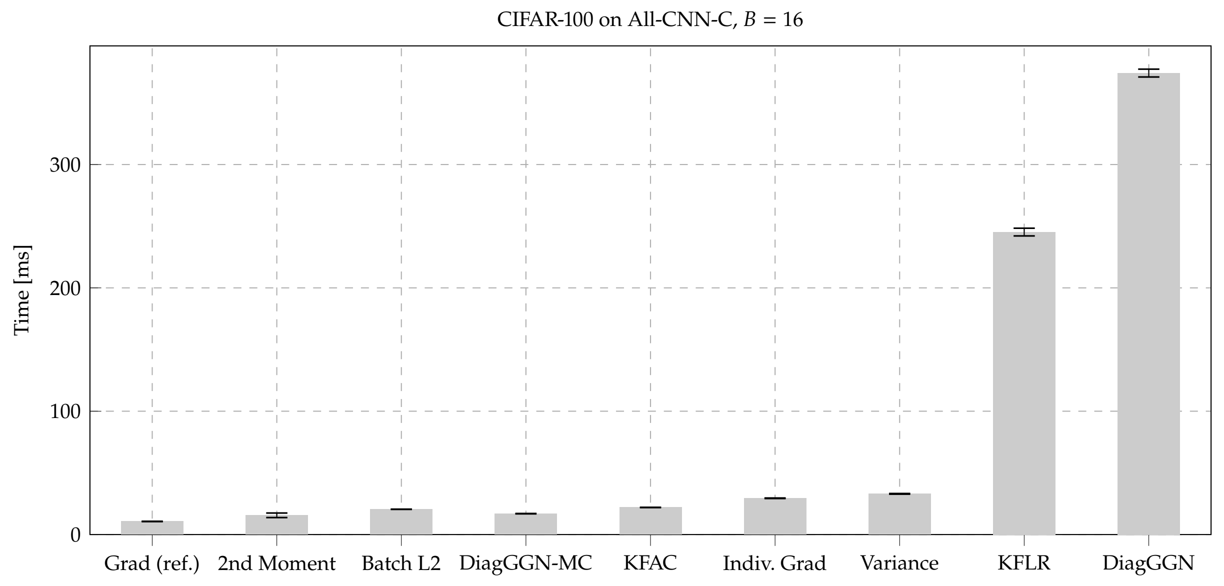


Figure B.1: KFLR and DiagGGN are more expensive to run on large networks. The gradient takes less than 20 ms to compute, but KFLR and DiagGGN are approximately 100× more expensive.

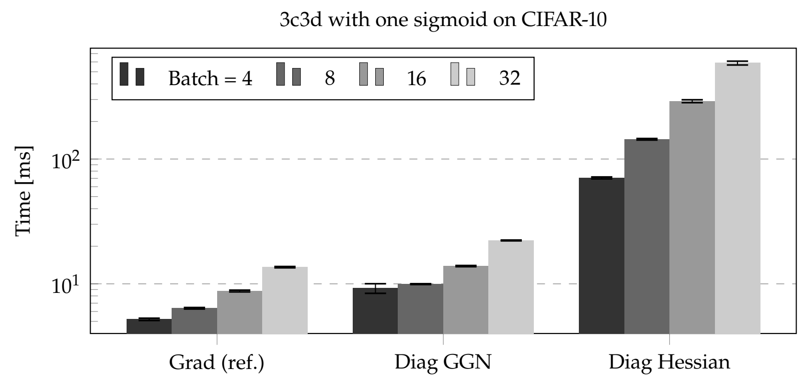


Figure B.2: Diagonal of the Hessian versus the GGN. If the network contains a single sigmoid activation function, the diagonal of the Hessian is an order of magnitude more computationally intensive than the diagonal of the GGN.

equivalent to the diagonal of the Hessian [107]. However, for networks that use non piecewise linear activation functions like sigmoids or tanh, computing the Hessian diagonal can be much more expensive than the GGN diagonal. To illustrate this, we modify the smaller net used in our benchmarks to include a single sigmoid activation function before the last classification layer. The results in Figure B.2 show that the Hessian diagonal computation is already an order of magnitude more expensive than for the GGN.

B.3 Experimental Details

B.3.1 Protocol

The optimizer experiments are performed according to the protocol suggested by DeepOBS:

- Train the neural network with the investigated optimizer and vary its hyperparameters on a specified grid. This training is performed for a single random seed only.

- DeepOBS evaluates metrics during the training procedure. From all runs of the grid search, it selects the best run automatically. The results shown in this work were obtained with the default strategy, favoring highest final accuracy on the validation set.
- For a better understanding of the optimizer performance with respect to randomized routines in the training process, DeepOBS reruns the best hyperparameter setting for ten different random seeds. The results show mean values over these repeated runs, with standard deviations as uncertainty indicators.
- Along with the benchmarked optimizers, we show the DeepOBS base line performances for Adam and momentum SGD (Momentum). They are provided by DeepOBS.

The optimizers built upon BackPACK's curvature estimates were benchmarked on DeepOBS's image classification problems from Table B.1.

Table B.1: Image classification test problems considered from the DeepOBS library [146].

Codename	Description	Dataset	# Parameters
LogReg	Linear model	MNIST	7,850
2c2d	2 convolutional and 2 dense linear layers	Fashion-MNIST	3,274,634
3c3d	3 convolutional and 3 dense linear layers	CIFAR-10	895,210
All-CNN-C	9 convolutional layers [157]	CIFAR-100	1,387,108

B.3.2 Grid Search & Best Hyperparameter Setting

Both the learning rate η and damping λ are tuned over the grid

$$\eta \in \left\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\right\}\,, \quad \lambda \in \left\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10\right\}\,.$$

We use the same batch size ($N = 128$ for all problems, except $N = 256$ for All-CNN-C on CIFAR-100) as the base lines and the optimizers run for the identical number of epochs.

The best hyperparameter settings are summarized in Table B.2.

B.3.3 Update rule

We use a simple update rule with a constant damping parameter λ. Consider the parameters $\boldsymbol{\theta}$ of a single module in a neural network with

Table B.2: Best hyperparameter settings for optimizers and baselines shown in this work. In the Momentum baselines, the momentum was fixed to 0.9. Parameters for computation of the running averages in Adam use the default values $(\beta_1, \beta_2) = (0.9, 0.999)$. The symbols ✓ and ✗ denote whether the hyperparameter setting is an interior point of the grid or not, respectively.

Curvature	`mnist_logreg`			`fmnist_2c2d`			`cifar10_3c3d`			`cifar100_allcnnc`		
	η	λ	int	η	λ	int	η	λ	int	η	λ	int
DiagGGN	10^{-3}	10^{-3}	✓	10^{-4}	10^{-4}	✗	10^{-3}	10^{-2}	✓	-	-	-
DiagGGN-MC	10^{-3}	10^{-3}	✓	10^{-4}	10^{-4}	✗	10^{-3}	10^{-2}	✓	10^{-3}	10^{-3}	✓
KFAC	10^{-2}	10^{-2}	✓	10^{-3}	10^{-3}	✓	1	10	✗	1	1	✓
KFLR	10^{-2}	10^{-2}	✓	10^{-2}	10^{-3}	✓	1	10	✗	-	-	-
KFRA	10^{-2}	10^{-2}	✓	-	-	-	-	-	-	-	-	-
Baseline	η			η			η			η		
Momentum	$\approx 2.07 \cdot 10^{-2}$			$\approx 2.07 \cdot 10^{-2}$			$\approx 3.79 \cdot 10^{-3}$			$\approx 4.83 \cdot 10^{-1}$		
Adam	$\approx 2.98 \cdot 10^{-4}$			$\approx 1.27 \cdot 10^{-4}$			$\approx 2.98 \cdot 10^{-4}$			$\approx 6.95 \cdot 10^{-4}$		

L_2-regularization of strength δ. Let $\boldsymbol{C}(\boldsymbol{\theta}_t)$ denote the curvature matrix and $\nabla_{\boldsymbol{\theta}_t}\mathcal{L}(\boldsymbol{\theta}_t)$ the gradient at step t. One iteration applies

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \left[\boldsymbol{C}(\boldsymbol{\theta}_t) + (\lambda + \delta)\boldsymbol{I}\right]^{-1} \left[\nabla_{\theta_t}\mathcal{L}(\boldsymbol{\theta}_t) + \delta\boldsymbol{\theta}_t\right] . \tag{B.20}$$

The inverse cannot be computed exactly (in reasonable time) for the Kronecker-factored curvatures KFAC, KFLR, and KFRA. We use the scheme of [109] to approximately invert $\boldsymbol{C}(\boldsymbol{\theta}_t) + (\lambda + \delta)\boldsymbol{I}$ if $\boldsymbol{C}(\boldsymbol{\theta}_t)$ is Kronecker-factored; $\boldsymbol{C}(\boldsymbol{\theta}_t) = \boldsymbol{A}(\boldsymbol{\theta}_t) \otimes \boldsymbol{B}(\boldsymbol{\theta}_t)$. It replaces the expression $(\lambda + \delta)\boldsymbol{I}$ by diagonal terms added to each Kronecker factor. In summary, this replaces

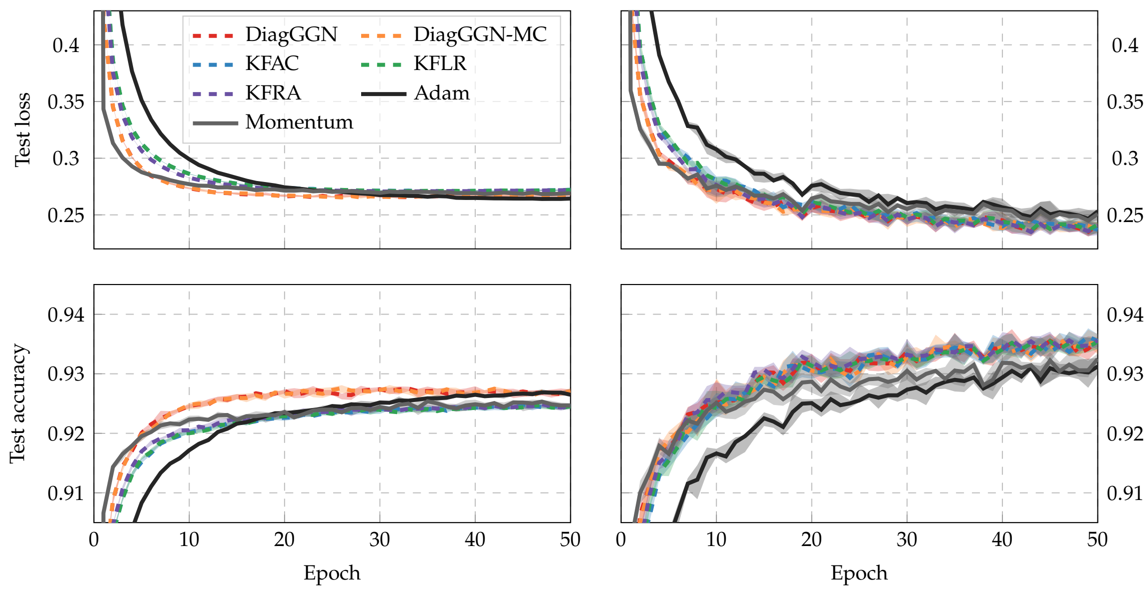
$$\begin{aligned} &\left[\boldsymbol{A}(\boldsymbol{\theta}_t) \otimes \boldsymbol{B}(\boldsymbol{\theta}_t) + (\lambda + \delta)\boldsymbol{I}\right]^{-1} \\ \text{by} \quad &\left[\boldsymbol{A}(\boldsymbol{\theta}_t) + \pi_t\sqrt{\lambda + \delta}\boldsymbol{I}\right]^{-1} \otimes \left[\boldsymbol{B}(\boldsymbol{\theta}_t) + \frac{1}{\pi_t}\sqrt{\lambda + \delta}\boldsymbol{I}\right]^{-1} . \end{aligned} \tag{B.21}$$

A principled choice for the parameter π_t is $\pi_t = \sqrt{\|\boldsymbol{A}(\boldsymbol{\theta}_t)\otimes \boldsymbol{I}_B\| / \|\boldsymbol{I}_A \otimes \boldsymbol{B}(\boldsymbol{\theta}_t)\|}$ for any matrix norm $\|\cdot\|$. We follow [109] and choose the trace norm,

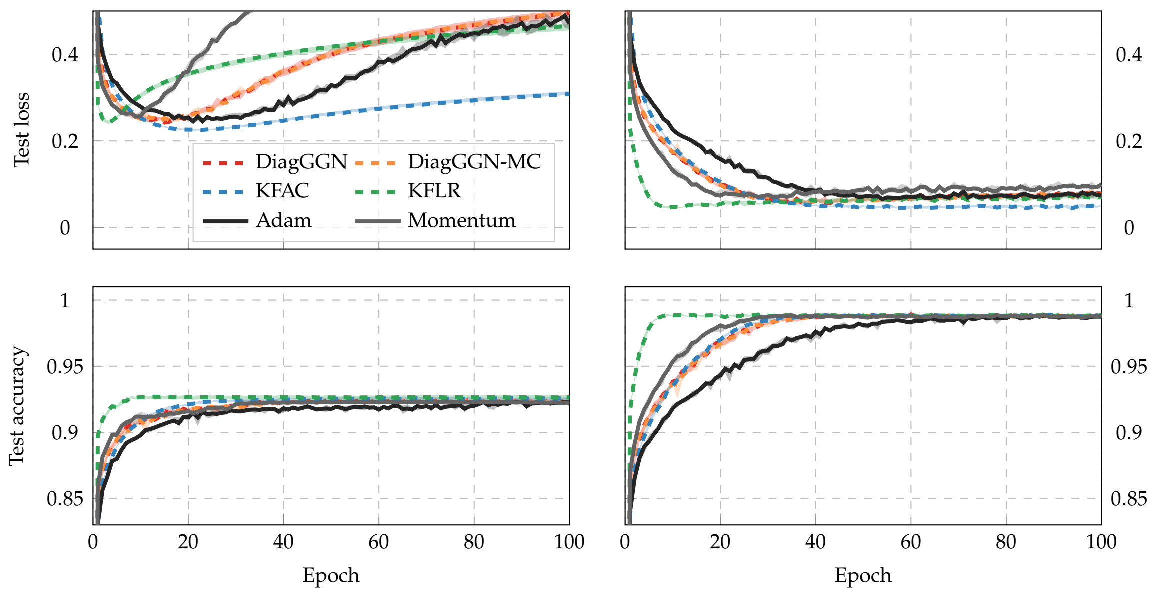
$$\pi_t = \sqrt{\frac{\operatorname{Tr}(\boldsymbol{A}(\boldsymbol{\theta}_t)) \dim(\boldsymbol{B})}{\dim(\boldsymbol{A}) \otimes \operatorname{Tr}(\boldsymbol{B}(\boldsymbol{\theta}_t))}} . \tag{B.22}$$

B.3.4 Additional results

This section presents the results for MNIST using a logistic regression in Figure B.3a and Fashion-MNIST using the 2c2d network, composed of two convolutional and two linear layers, in Figure B.3b.



(a) LogReg (7,850 parameters) on MNIST



(b) 2c2d (3,274,634 parameters) on Fashion-MNIST

Figure B.3: Additional results. Median performance with shaded quartiles of the best hyperparameter settings chosen by DeepOBS. Solid lines show well-tuned baselines of momentum SGD and Adam that are provided by DeepOBS.

B.4 BackPACK Cheat Sheet

- Assumptions
  - Sequential feedforward network

$$\begin{tikzcd} z_n^{(0)} \arrow[r, "f^{(1)}_{\theta^{(1)}}(z_n^{(0)})"] & z_n^{(1)} \arrow[r, "f^{(2)}_{\theta^{(2)}}(z_n^{(1)})"] & \dots \arrow[r, "f^{(L)}_{\theta^{(L)}}(z_n^{(L-1)})"] & z^{(L)} \arrow[r, "\ell(z_n^{(L)}, y)"] & \ell(\theta) \end{tikzcd}$$

  - $d^{(l)}$: Dimension of parameter $\theta^{(l)}$
  - Empirical risk

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{n=1}^{N} \ell(f_\theta(x_n), y_n)$$

- Shorthands

$$\ell_n(\theta) := \ell(f_\theta(x_n), y_n)\,, \qquad n = 1, \dots, N\,,$$
$$f_n := f_\theta(x_n) = z_n^{(L)}(\theta)\,, \qquad n = 1, \dots, N$$

- Generalized Gauss-Newton matrix

$$G(\theta) = \frac{1}{N}\sum_{n=1}^{N} \left(\mathrm{J}_\theta f_n\right)^\top \nabla^2_{f_n} \ell_n(\theta) \left(\mathrm{J}_\theta f_n\right)$$

- Approximative GGN via MC sampling

$$\tilde{G}(\theta) = \frac{1}{N}\sum_{n=1}^{N} \left(\mathrm{J}_\theta f_n\right)^\top \left[\left(\nabla_{f_n} \ell(f_n, \hat{y})\right) \left(\nabla_{f_n} \ell(f_n, \hat{y}_n)\right)^\top\right]_{\hat{y}_n \sim q(\cdot \mid f_n)} \left(\mathrm{J}_\theta f_n\right)$$

Table B.3: **Overview of the features supported in the first release of BackPACK.** The quantities are computed separately for all module parameters, *i.e.* $l = 1, \dots, L$.

Feature	**Details**
Individual gradients	$\frac{1}{N}\nabla_{\theta^{(l)}}\ell_n(\theta), \quad n = 1, \dots, N$
Batch variance	$\frac{1}{N}\sum_{n=1}^{N}\left[\nabla_{\theta^{(l)}}\ell_n(\theta)\right]_j^2 - \left[\nabla_{\theta^{(l)}}\mathcal{L}(\theta)\right]_j^2, \quad j = 1, \dots, d^{(l)}$
2nd moment	$\frac{1}{N}\sum_{n=1}^{N}\left[\nabla_{\theta^{(l)}}\ell_n(\theta)\right]_j^2, \quad j = 1, \dots, d^{(l)}.$
Individual gradient L_2 norms	$\left\Vert\frac{1}{N}\nabla_{\theta^{(l)}}\ell_n(\theta)\right\Vert_2^2, \quad n = 1, \dots, N$
DiagGGN	$\mathrm{diag}\left(G(\theta^{(l)})\right)$
DiagGGN-MC	$\mathrm{diag}\left(\tilde{G}(\theta^{(l)})\right)$
Hessian diagonal	$\mathrm{diag}\left(\nabla^2_{\theta^{(l)}}\mathcal{L}(\theta)\right)$
KFAC	$\tilde{G}(\theta^{(l)}) \approx A^{(l)} \otimes B^{(l)}_{\text{KFAC}}$
KFLR	$G(\theta^{(l)}) \approx A^{(l)} \otimes B^{(l)}_{\text{KFLR}}$
KFRA	$G(\theta^{(l)}) \approx A^{(l)} \otimes B^{(l)}_{\text{KFRA}}$

Additional Material for Chapter 6 C.

C.1 Code Example

One design principle of Cockpit is its easy integration with conventional PyTorch training loops. Procedure C.1 shows a working example of a standard training loop with Cockpit. More examples and tutorials are described in the documentation. Cockpit's syntax is inspired by BackPACK: it can be used interchangeably with the library responsible for most back-end computations. Changes to the code are straightforward:

```
 1  """Example: Training Loop using Cockpit."""
 2
 3  import torch
 4  from _utils_examples import cnn, fmnist_data, get_logpath
 5  from backpack import extend
 6  from cockpit import Cockpit, CockpitPlotter
 7  from cockpit.utils.configuration import configuration as config
 8
 9  fmnist_data = fmnist_data()
10  model = extend(cnn())
11  loss_fn = extend(torch.nn.CrossEntropyLoss(reduction="mean"))
12  losses_fn = torch.nn.CrossEntropyLoss(reduction="none")
13  opt = torch.optim.SGD(model.parameters(), lr=1e-2)
14
15  cockpit = Cockpit(model.parameters(), quantities=config("full"))
16  plotter = CockpitPlotter()
17
18  max_steps, global_step = 50, 0
19  for inputs, labels in iter(fmnist_data):
20      opt.zero_grad()
21
22      outputs = model(inputs)
23      loss = loss_fn(outputs, labels)
24      losses = losses_fn(outputs, labels)
25
26      with cockpit(
27          global_step,
28          info={
29              "batch_size": inputs.shape[0],
30              "individual_losses": losses,
31              "loss": loss,
32              "optimizer": opt,
33          },
34      ):
35          loss.backward(
36              create_graph=cockpit.create_graph(global_step),
37          )
38
39      opt.step()
40      global_step += 1
41
42      if global_step >= max_steps:
43          break
44
45  cockpit.write(get_logpath())
46  plotter.plot(get_logpath())
```

Procedure C.1: Complete training loop with Cockpit in PyTorch. Line changes are highlighted in blue (■).

- **Importing** (*Lines 5, 6* and *7*): besides importing Cockpit we also need to import BackPACK which is required for extending (parts of) the model (see next step).

- **Extending** (*Lines 10* and *11*): when defining the model and the loss function, we need to *extend* both of them using BackPACK. This is as trivial as wrapping them in the `extend()` function provided by BackPACK and lets BackPACK know that additional quantities (such as the individual gradients) should be computed for them. Note, that while applying BackPACK is easy, it currently does not support all possible model architectures and layer types. Specifically, *batch norm* layers are not supported since using them results in ill-defined individual gradients.
- **Individual losses** (*Line 12*): for the `Alpha` quantity, Cockpit also requires the individual loss values (to estimate the variance of the loss estimate). This can be computed cheaply but is not usually part of a conventional training loop. Creating this loss is done analogously to creating any other loss, with the only exception of setting `reduction="none"`. Since we don't differentiate this loss, we don't need to extend it.
- **Cockpit configuration** (*Line 15* and *16*): Initializing the Cockpit requires passing them (extended) model parameters as well as a list of quantities that should be tracked. Table 6.1 provides an overview of all possible quantities. In this example, we use one of the pre-defined configurations offered by Cockpit. Separately, we initialize the plotting part of Cockpit. We deliberately detached the visualization from the tracking to allow greater flexibility.
- **Quantity computation** (*Line 26* to *37*): Performing the training is very similar to a regular training loop, with the only difference being that the backward pass should be surrounded by the Cockpit context (`with cockpit():`). Additionally to the `global_step` we also pass a few additional information to the Cockpit that are computed anyway and can be re-used, such as the batch size, the individual losses, or the optimizer itself. After the backward pass (when the context is left) all Cockpit quantities are automatically computed.
- **Logging & visualizing** (*Line 45* and *46*): at any point during training—here at the end—we can write all quantities to a log file. We can use this log file, or alternatively the Cockpit directly, to visualize all quantities in a status screen similar to Figure 6.2.

C.2 Instrument Overview

Table C.1 lists all quantities available in the first public release of Cockpit. If necessary, we provide references to their mathematical definition. This table contains additional quantities, compared to Table 6.1 in the main text. To improve the presentation of this work, we decided to not describe every quantity available in Cockpit in the main part and instead focus on the investigated metrics. Custom quantities can be added easily without having to understand the inner-workings.

Table C.1: **Overview of all Cockpit quantities** with a short description and, if necessary, a reference to mathematical definition.

Name	Description	Math
`Loss`	Mini-batch training loss at current iteration, $\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$	(6.1)
`Parameters`	Parameter values $\boldsymbol{\theta}_t$ at the current iteration	-
`Distance`	L_2 distance from initialization $\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_0\|_2$	-
`UpdateSize`	Update size of the current iteration $\|\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t\|_2$	
`GradNorm`	Mini-batch gradient norm $\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|_2$	-
`Time`	Time of the current iteration (*e.g.* used in benchmark of Appendix C.5)	-
`Alpha`	Normalized step on a noisy quadratic interpolation between two iterates $\boldsymbol{\theta}_t, \boldsymbol{\theta}_{t+1}$	(C.7)
`CABS`	Adaptive batch size for SGD, optimizes expected objective gain per cost, from [11]	(C.9)
`EarlyStopping`	Evidence-based early stopping criterion for SGD, proposed in [104]	(C.11d)
`GradHist1d`	Histogram of individual gradient elements, $\{[\boldsymbol{g}_n(\boldsymbol{\theta})]_j\}_{n\in\mathbb{B}}^{j=1,\dots,D}$	(C.12)
`GradHist2d`	Histogram of weights and individual gradient elements, $\{([\boldsymbol{\theta}]_j, [\boldsymbol{g}_n(\boldsymbol{\theta})]_j)\}_{n\in\mathbb{B}}^{j=1,\dots,D}$	(C.13)
`NormTest`	Normalized fluctuations of the residual norms $\|\boldsymbol{g}_{\mathbb{B}} - \boldsymbol{g}_n\|$, proposed in [26]	(C.16c)
`InnerTest`	Normalized fluctuations of $\boldsymbol{g}_n$'s parallel components along $\boldsymbol{g}_{\mathbb{B}}$, proposed in [19]	(C.19c)
`OrthoTest`	Normalized fluctuations of $\boldsymbol{g}_n$'s orthogonal components along $\boldsymbol{g}_{\mathbb{B}}$, proposed in [19]	(C.22b)
`HessMaxEV`	Maximum Hessian eigenvalue, $\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, inspired by [177]	(C.23)
`HessTrace`	Exact or approximate Hessian trace, $\operatorname{Tr}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$, inspired by [177]	-
`TICDiag`	Relation between (diagonal) curvature and gradient noise, inspired by [162]	(C.26)
`TICTrace`	Relation between curvature and gradient noise trace, inspired by [162]	(C.25)
`MeanGSNR`	Average gradient signal-to-noise ratio (GSNR), inspired by [100]	(C.28b)

C.3 Mathematical Details

Here, we provide the mathematical background for each instrument in Table C.1. This complements the more informal description presented in Section 6.2, which focused more on the expressiveness of the individual quantities. We start by setting up the necessary notation in addition to the one introduced in Section 6.2. See Sections 2.1 and 3.2 for more details.

C.3.1 Additional Notation

Population Properties

The population risk $\mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta}) \in \mathbb{R}$ and its variance $\Lambda(\boldsymbol{\theta}) \in \mathbb{R}$ are given by

$$\begin{aligned}\mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta}) &= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p_{\text{data}}}\left[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})\right] \\ &= \int \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{y}\,,\end{aligned} \tag{C.1a}$$

$$\begin{aligned}\Lambda_{p_{\text{data}}}(\boldsymbol{\theta}) &= \operatorname{Var}_{(\boldsymbol{x},\boldsymbol{y})\sim p_{\text{data}}}\left[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})\right] \\ &= \int \left(\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) - \mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta})\right)^2 p_{\text{data}}(\boldsymbol{x}, \boldsymbol{y}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{y}\,,\end{aligned} \tag{C.1b}$$

and its gradient $\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta}) \in \mathbb{R}^D$ and variance $\boldsymbol{\Sigma}_{p_{\text{data}}}(\boldsymbol{\theta}) \in \mathbb{R}^{D\times D}$ are

$$\begin{aligned}\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta}) &= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p_{\text{data}}}\left[\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y})\right] \\ &= \int \nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y})p_{\text{data}}(\boldsymbol{x},\boldsymbol{y})\mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{y}\,,\end{aligned} \tag{C.2a}$$

$$\begin{aligned}\boldsymbol{\Sigma}_{p_{\text{data}}}(\boldsymbol{\theta}) &= \mathrm{Var}_{(\boldsymbol{x},\boldsymbol{y})\sim p_{\text{data}}}\left[\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y})\right] \\ &= \int \left(\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y}) - \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right) \\ &\qquad \left(\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y}) - \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right)^\top p_{\text{data}}(\boldsymbol{x},\boldsymbol{y})\mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{y}\,.\end{aligned} \tag{C.2b}$$

Empirical Approximations

Let $\mathbb{D}$ denote a set of samples drawn *i.i.d.* from p_{data}, *i.e.* $\mathbb{D} = \{(\boldsymbol{x}_n,\boldsymbol{y}_n)\}_{n=1}^{|\mathbb{D}|}$. With a slight abuse of notation ($n \in \mathbb{D}$ instead of $(\boldsymbol{x}_n,\boldsymbol{y}_n) \in \mathbb{D}$) the empirical risk approximated with $\mathbb{D}$ is

$$\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{D}|}\sum_{n\in\mathbb{D}} \ell_n(\boldsymbol{\theta}) \tag{C.3a}$$

(later, $\mathbb{D}$ will represent either a mini-batch $\mathcal{B}$, or the train set $\mathbb{D}_{\text{train}}$). The empirical risk gradient $\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}) \in \mathbb{R}^D$ on $\mathbb{D}$ is

$$\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) = \frac{1}{|\mathbb{D}|}\sum_{n\in\mathbb{D}}\nabla_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta}) = \frac{1}{|\mathbb{D}|}\sum_{n\in\mathbb{D}}\boldsymbol{g}_n(\boldsymbol{\theta})\,, \tag{C.3b}$$

with individual gradients $\boldsymbol{g}_n(\boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta}) \in \mathbb{R}^D$ from a sample n. Population risk and gradient variances $\Lambda_{p_{\text{data}}}(\boldsymbol{\theta}), \boldsymbol{\Sigma}_{p_{\text{data}}}(\boldsymbol{\theta})$ can be empirically estimated on $\mathbb{D}$ with the sample variances $\hat{\Lambda}_{\mathbb{D}}(\boldsymbol{\theta}) \in \mathbb{R}, \hat{\boldsymbol{\Sigma}}_{\mathbb{D}}(\boldsymbol{\theta}) \in \mathbb{R}^{D\times D}$,

$$\Lambda_{p_{\text{data}}}(\boldsymbol{\theta}) \approx \frac{1}{|S|-1}\sum_{n\in\mathbb{D}}\left(\ell_n(\boldsymbol{\theta}) - \mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})\right)^2 := \hat{\Lambda}_{\mathbb{D}}(\boldsymbol{\theta})\,, \tag{C.4a}$$

$$\begin{aligned}\boldsymbol{\Sigma}_{p_{\text{data}}}(\boldsymbol{\theta}) &\approx \frac{1}{|\mathbb{D}|-1}\sum_{n\in\mathbb{D}}\left(\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})\right)\left(\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})\right)^\top := \hat{\boldsymbol{\Sigma}}_{\mathbb{D}}(\boldsymbol{\theta}) \\ &\approx \frac{1}{|\mathbb{D}|-1}\left[\left(\sum_{n\in\mathbb{D}}\boldsymbol{g}_n(\boldsymbol{\theta})\boldsymbol{g}_n(\boldsymbol{\theta})^\top\right) - |\mathbb{D}|\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})\boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})^\top\right]\,.\end{aligned} \tag{C.4b}$$

Often, gradient elements are assumed independent and hence their variance is diagonal ($^{\odot 2}$ denotes element-wise square),

$$\mathrm{diag}\left[\boldsymbol{\Sigma}_P(\boldsymbol{\theta})\right] \approx \frac{1}{|S|-1}\sum_{n\in\mathbb{D}}\left(\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathbb{D}}(\boldsymbol{\theta})\right)^{\odot 2} = \mathrm{diag}\left[\hat{\boldsymbol{\Sigma}}_{\mathbb{D}}(\boldsymbol{\theta})\right] \in \mathbb{R}^D\,. \tag{C.5}$$

Slicing

To avoid confusion between $\boldsymbol{\theta}_t$ (parameter at iteration t) and θ_j (j-th parameter entry), we denote the latter as $[\boldsymbol{\theta}]_j$.

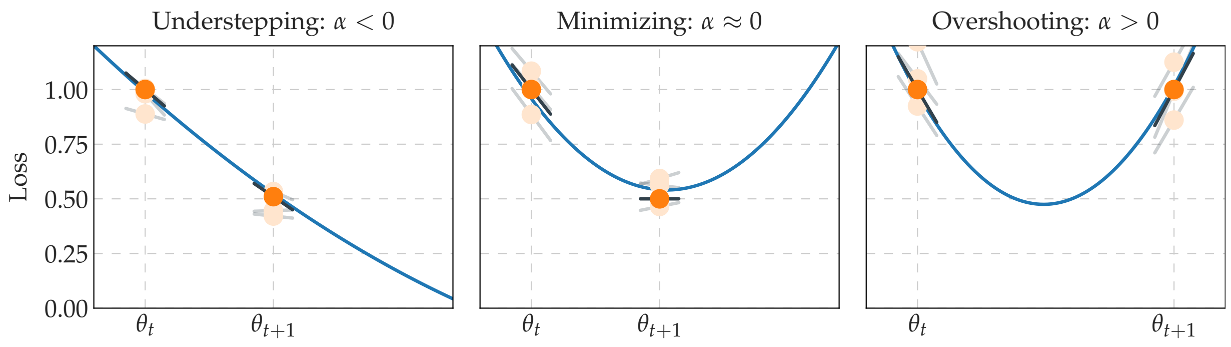


Figure C.1: Motivational sketch for the α quantity. In each iteration of the optimizer we observe the loss function at two positions $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_{t+1}$ (shown in ●). The black lines (—) show the observed slope at this position, which we can get from projecting the gradients onto the current step direction $\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$. Note, that all four observations (two loss and two slope values) are noisy, due to being computed on a mini-batch. With access to the individual losses and gradients (some samples shown in ●/—), we can estimate their noise level and build a noise-informed quadratic fit (—). Using this fit, we determine whether the optimizer minimizes the local uni-variate loss (*middle plot*), or whether we understep (*left plot*) or overshoot (*right plot*) the minimum.

C.3.2 Normalized Step Length (`Alpha`)

Motivation

The goal of the α-quantity is to estimate and quantify the effect that a selected learning rate has on the optimizer's steps. Consider the optimizer's step at training iteration t. This parameter update from $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_{t+1}$ happens in a one-dimensional space, defined by the update direction $\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = \boldsymbol{s}_t$. The update direction depends on the update rule of the optimizer, *e.g.* for SGD with learning rate η it is simply $\boldsymbol{s}_t = -\eta \boldsymbol{g}_{\mathbb{B}_t}(\boldsymbol{\theta}_t)$.

We build a noise-informed uni-variate quadratic approximation along this update step ($\boldsymbol{\theta}_t \rightarrow \boldsymbol{\theta}_{t+1}$) based on the two noisy loss function observations at $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_{t+1}$ and the two noisy slope observation at these two points. Examining this quadratic fit, we are able to determine where on this parabola our optimizer steps. Standardizing this, we express a step to the minimum of the loss in the update direction as $\alpha = 0$. Analogously, steps that end short of this minimum result in $\alpha < 0$, and a step over the minimum in $\alpha > 0$. These three different scenarios are illustrated in Figure C.1 also showing the underlying observations that would lead to them. Figure 6.1 shows the distribution of α-values for two very different optimization trajectories.

Noisy Observations

In order to build an approximation for the loss function in the update direction, we leverage the four observations of the function (and its derivative) that are available in each iteration. Due to the stochasticity of deep learning optimization, we also take into account the noise-level of all observations by estimating them. The first two observations are the mini-batch training losses $\mathcal{L}_{\mathbb{B}_t}(\boldsymbol{\theta}_t), \mathcal{L}_{\mathbb{B}_{t+1}}(\boldsymbol{\theta}_{t+1})$ at point $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_{t+1}$, which are computed in every standard training loop. The mini-batch losses are averages over individual losses,

$$\mathcal{L}_{\mathbb{B}_t}(\boldsymbol{\theta}_t) = \mathbb{E}_{\mathbb{B}_t}\left[\ell(\boldsymbol{\theta}_t)\right] = \frac{1}{|\mathbb{B}_t|}\sum_{n\in\mathbb{B}_t}\ell_n(\boldsymbol{\theta}_t)\,,$$
$$\mathcal{L}_{\mathbb{B}_{t+1}}(\boldsymbol{\theta}_{t+1}) = \mathbb{E}_{\mathbb{B}_{t+1}}\left[\ell(\boldsymbol{\theta}_{t+1})\right] = \frac{1}{|\mathbb{B}_{t+1}|}\sum_{n\in\mathbb{B}_{t+1}}\ell_n(\boldsymbol{\theta}_{t+1})\,,$$

and using these individual losses, we can also compute the variances to estimate the noise-level of our loss observation,

$$\mathrm{Var}_{\mathbb{B}_t}\left[\ell(\boldsymbol{\theta}_t)\right] = \left(\frac{1}{|\mathbb{B}_t|}\sum_{n\in\mathbb{B}_t}\ell_n(\boldsymbol{\theta}_t)^2\right) - \left(\frac{1}{|\mathbb{B}_t|}\sum_{n\in\mathbb{B}_t}\ell_n(\boldsymbol{\theta}_t)\right)^2\,,$$
$$\mathrm{Var}_{\mathbb{B}_{t+1}}\left[\ell(\boldsymbol{\theta}_{t+1})\right] = \left(\frac{1}{|\mathbb{B}_{t+1}|}\sum_{n\in\mathbb{B}_{t+1}}\ell_n(\boldsymbol{\theta}_{t+1})^2\right) - \left(\frac{1}{|\mathbb{B}_{t+1}|}\sum_{n\in\mathbb{B}_{t+1}}\ell_n(\boldsymbol{\theta}_{t+1})\right)^2\,.$$

Similarly, we proceed with the slope in the update direction. To compute the slope of the loss function in the direction of the optimizer's update $\boldsymbol{s}_t$, we project the current gradient along this update direction

$$\mathbb{E}_{\mathbb{B}_t}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right] = \frac{1}{|\mathbb{B}_t|}\sum_{n\in\mathbb{B}_t}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\,,$$
$$\mathbb{E}_{\mathbb{B}_{t+1}}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right] = \frac{1}{|\mathbb{B}_{t+1}|}\sum_{n\in\mathbb{B}_{t+1}}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\,.$$

Just like before, we can also compute the variance of this slope, by leveraging individual gradients,

$$\begin{aligned}
&\mathrm{Var}_{\mathbb{B}_t}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right] \\
&= \frac{1}{|\mathbb{B}_t|}\sum_{n\in B_t}\left(\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right)^2 - \left(\frac{1}{|\mathbb{B}_t|}\sum_{n\in\mathbb{B}_t}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right)^2\,,\\
&\mathrm{Var}_{\mathbb{B}_{t+1}}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right] \\
&= \frac{1}{|\mathbb{B}_{t+1}|}\sum_{n\in\mathbb{B}_{t+1}}\left(\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right)^2 - \left(\frac{1}{|\mathbb{B}_{t+1}|}\sum_{n\in\mathbb{B}_{t+1}}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right)^2\,.
\end{aligned}$$

Quadratic Fit & Normalization

Using our (noisy) observations, we are now ready to build an approximation for the loss as a function of the step size, which we will denote as $f(\tau)$. We assume a quadratic function for f, which follows recent reports for the loss landscape of neural networks [176], *i.e.* a function $f(\tau) = w_0 + w_1\tau + w_2\tau^2$ parameterized by $\boldsymbol{w} \in \mathbb{R}^3$. We further assume a Gaussian likelihood of the form

$$p\left(\tilde{f} \mid \boldsymbol{w}, \boldsymbol{\Phi}\right) = \mathcal{N}\left(\tilde{f} \mid \boldsymbol{\Phi}^\top \boldsymbol{w}, \boldsymbol{\Lambda}\right) \qquad \text{(C.6)}$$

for observations $\tilde{f}$ of the loss and its slope. The observation matrix $\boldsymbol{\Phi}$ and the noise matrix of the observations $\boldsymbol{\Lambda}$ are

$$\boldsymbol{\Phi} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ \tau_1 & \tau_2 & 1 & 1 \\ \tau_1^2 & \tau_2^2 & 2\tau_1 & 2\tau_2 \end{pmatrix}, \qquad \boldsymbol{\Lambda} = \begin{pmatrix} \sigma_{\tilde{f}_1} & 0 & 0 & 0 \\ 0 & \sigma_{\tilde{f}_2} & 0 & 0 \\ 0 & 0 & \sigma_{\tilde{f}'_1} & 0 \\ 0 & 0 & 0 & \sigma_{\tilde{f}'_2} \end{pmatrix},$$

where τ denotes the position and σ denotes the noise-level estimate of the observation. The maximum likelihood solution of Equation (C.6) for the parameters of our quadratic fit is given by

$$w = \left(\boldsymbol{\Phi}\boldsymbol{\Lambda}^{-1}\boldsymbol{\Phi}^{\top}\right)^{-1} \boldsymbol{\Phi}\boldsymbol{\Lambda}^{-1}\tilde{f}\,. \tag{C.7}$$

Once we have the quadratic fit of the uni-variate loss along the update direction, we normalize the scales such that the resulting α expresses the effective step taken by the optimizer sketched in Figure C.1.

Usage

The α-quantity is related to recent line search approaches [105, 168]. However, instead of searching for an acceptable step by repeated attempts, we instead report the effect of the current step size selection. This could, for example, be used to disentangle the two optimization runs in Figure 6.1. Additionally, this information could also be used to automatically adapt the learning rate during the training process. But, as discussed in Section 6.3.3, it isn't trivial what the "correct" decision is, as it might depend on the optimization problem, the training phase, and other factors. Having this α-quantity can, however, provide more insight into what kind of steps are used in well-tuned runs with traditional optimizers such as SGD.

C.3.3 CABS Criterion: Coupling Adaptive Batch Sizes with Learning Rates (CABS)

The CABS criterion, proposed by Balles et al. [11], can be used to adapt the mini-batch size during training with SGD. It relies on the gradient noise and approximately optimizes the objective's expected gain per cost. The adaptation rule is (with learning rate η)

$$|\mathbb{B}| \leftarrow \eta \frac{\operatorname{Tr}(\boldsymbol{\Sigma}_{p_{\text{data}}}(\boldsymbol{\theta}))}{\mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta})}\,, \tag{C.8}$$

and the practical implementation approximates $\mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta}) \approx \mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$and $\operatorname{Tr}(\boldsymbol{\Sigma}_{p_{\text{data}}}(\boldsymbol{\theta})) \approx {}^{(|\mathbb{B}|-1)}/_{|\mathbb{B}|} \operatorname{Tr}(\hat{\boldsymbol{\Sigma}}_{\mathbb{B}}(\boldsymbol{\theta}))$ (compare equations (10, 22) and first paragraph of Section 4 in [11]). This yields the quantity computed in Cockpit's CABS instrument,

$$|\mathbb{B}| \leftarrow \eta \frac{\frac{1}{|\mathbb{B}|}\sum_{j=1}^{D}\sum_{n\in\mathbb{B}}\left[g_n(\boldsymbol{\theta}) - g_{\mathbb{B}}(\boldsymbol{\theta})\right]_j^2}{\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})}\,. \tag{C.9}$$

Usage

The CABS criterion suggests a batch size which is optimal under certain assumptions. This suggestion can support practitioners in the batch size selection for their deep learning task.

C.3.4 Early-stopping Criterion for SGD (`EarlyStopping`)

The empirical risk $\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta})$, and the mini-batch loss $\mathcal{L}_{\mathbb{B}}(\boldsymbol{\theta})$ are only estimators of the target objective $\mathcal{L}_{p_{\text{data}}}(\boldsymbol{\theta})$. Mahsereci et al. [104] motivate $p(\boldsymbol{g}_{\mathbb{B},\mathbb{D}}(\boldsymbol{\theta}) \mid \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta}) = \boldsymbol{0})$ as a measure for detecting noise in the finite datasets $\mathbb{B}, \mathbb{D}$ due to sampling from p_{data}. They propose an evidence-based (EB) criterion for early stopping the training procedure based on mini-batch statistics, and model $p(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}))$ with a sampled diagonal variance approximation (compare Equation (C.5)),

$$p(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})) \approx \prod_{j=1}^{D} \mathcal{N}\left(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \mid \left[\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta}) \right]_j, \frac{\left[\hat{\boldsymbol{\Sigma}}_{\mathbb{B}}(\boldsymbol{\theta}) \right]_{j,j}}{|\mathbb{B}|} \right). \tag{C.10}$$

Their SGD stopping criterion is

$$\frac{2}{D} \left[\log p(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})) - \mathbb{E}_{\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \sim p(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}))} \left[\log p(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})) \right] \right] > 0\,, \tag{C.11a}$$

and translates into

$$1 - \frac{|\mathbb{B}|}{D} \sum_{d=1}^{D} \frac{\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \right]_d^2}{\left[\hat{\boldsymbol{\Sigma}}_{\mathbb{B}}(\boldsymbol{\theta}) \right]_{d,d}} > 0\,, \tag{C.11b}$$

$$1 - \frac{|\mathbb{B}|}{D} \sum_{d=1}^{D} \frac{\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \right]_d^2}{\frac{1}{|\mathbb{B}|-1} \sum_{n\in\mathbb{B}} \left[\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \right]_d^2} > 0\,, \tag{C.11c}$$

$$1 - \frac{|\mathbb{B}|(|\mathbb{B}|-1)}{D} \sum_{d=1}^{D} \frac{\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \right]_d^2}{\left(\sum_{n\in\mathbb{B}} \left[\boldsymbol{g}_n(\boldsymbol{\theta}) \right]_d^2 \right) - |\mathbb{B}| \left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) \right]_d^2} > 0\,. \tag{C.11d}$$

Cockpit's `EarlyStopping` quantity computes the left side of Equation (C.11d).

Usage

Cockpit's `EarlyStopping` quantity can inform practitioners that training is about to be completed and the model might be at risk of overfitting.

C.3.5 Individual Gradient Element Histograms (`GradHist1d`, `GradHist2d`)

For the $|\mathbb{B}| \times D$ individual gradient elements, Cockpit's `GradHist1d` instrument displays a histogram of

$$\left\{ [\boldsymbol{g}_n(\boldsymbol{\theta})]_d \right\}_{n\in\mathbb{B}, d=1,\dots,D} \,. \tag{C.12}$$

Cockpit's `GradHist2d` instrument displays a two-dimensional histogram of the $|\mathbb{B}| \times D$ tuples

$$\left\{\left([\boldsymbol{\theta}]_d, [\boldsymbol{g}_n(\boldsymbol{\theta})]_d\right)\right\}_{n\in\mathbb{B}, d=1,\dots,D} \tag{C.13}$$

and the marginalized one-dimensional histograms over the parameter and gradient axes.

Usage

Sections 6.3.1 and 6.3.2 provide use cases (identifying data pre-processing issues and vanishing gradients) for both the gradient histogram as well as its layer-wise extension.

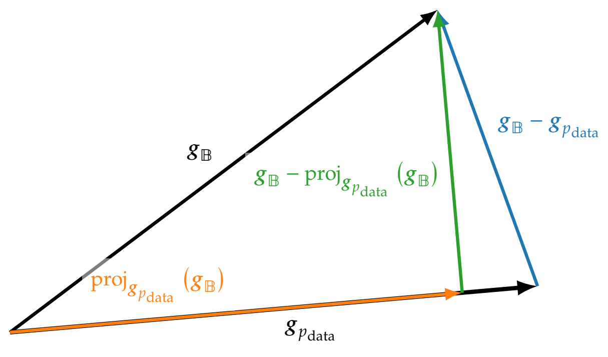
C.3.6 Gradient Tests (`NormTest`, `InnerTest`, `OrthoTest`)

Bollapragada et al. [19] and Byrd et al. [26] propose batch size adaptation schemes based on the gradient noise. They formulate geometric constraints between population and mini-batch gradient and accessible approximations that can be probed to decide whether the mini-batch size should be increased. Because mini-batches are *i.i.d.* from p_{data}, it holds that

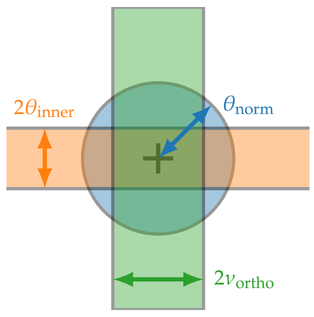
$$\mathbb{E}\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right] = \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta}), \tag{C.14a}$$

$$\mathbb{E}\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right] = \|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|^2. \tag{C.14b}$$

The above works propose enforcing other weaker similarity in expectation during optimization. These geometric constraints reduce to basic vector geometry (see Figure C.2a for an overview of the relevant vectors). We recall their formulation here for consistency and derive the practical versions, which can be computed from training observables and are used in Cockpit (consult Figure C.2b for the visualization).


$\boldsymbol{g}_{\mathbb{B}}$
$\boldsymbol{g}_{\mathbb{B}} - \boldsymbol{g}_{p_{\text{data}}}$
$\boldsymbol{g}_{\mathbb{B}} - \text{proj}_{\boldsymbol{g}_{p_{\text{data}}}}(\boldsymbol{g}_{\mathbb{B}})$
$\text{proj}_{\boldsymbol{g}_{p_{\text{data}}}}(\boldsymbol{g}_{\mathbb{B}})$
$\boldsymbol{g}_{p_{\text{data}}}$

(a) Relevant vectors



(b) Cockpit's gradient test visualization.

Figure C.2: Conceptual sketch for gradient tests. (a) Relevant vectors to formulate the geometric constraints between population and mini-batch gradient probed by the gradient tests. **(b)** Gradient test visualization in Cockpit.

Usage

All three gradient tests describe the noise level of the gradients. Bollapragada et al. [19] and Byrd et al. [26] adapt the batch size so that the proposed geometric constraints are fulfilled. Practitioners can use the combined gradient test plot, *i.e.* top center plot in Figure 6.2, to monitor gradient noise during training and adjust hyperparameters such as the batch size.

Norm Test (`NormTest`)

The norm test [26] constrains the residual norm $\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) - \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|$, rescaled by $\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|$. This gives rise to a standardized ball of radius $\theta_{\text{norm}} \in (0, \infty)$ around the population gradient, where the mini-batch gradient should reside. Byrd et al. [26] set $\theta_{\text{norm}} = 0.9$ in their experiments and increase the batch size if (in the practical version, see below) the following constraint is not fulfilled

$$\mathbb{E}\left[\frac{\left\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) - \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2}{\left\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2}\right] \leq \theta_{\text{norm}}^2 \,. \tag{C.15a}$$

Instead of taking the expectation over mini-batches, Byrd et al. [26] note that the above will be satisfied if

$$\frac{1}{|\mathbb{B}|}\mathbb{E}\left[\frac{\left\|\boldsymbol{g}_{n}(\boldsymbol{\theta}) - \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2}{\left\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2}\right] \leq \theta_{\text{norm}}^2 \,. \tag{C.15b}$$

They propose a practical form of this test,

$$\frac{1}{|\mathbb{B}|(|\mathbb{B}|-1)} \frac{\sum_{n\in\mathbb{B}} \left\|\boldsymbol{g}_{n}(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2}{\left\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2} \leq \theta_{\text{norm}}^2 \,, \tag{C.16a}$$

which can be computed from mini-batch statistics. Rearranging

$$\sum_{n\in\mathbb{B}} \left\|\boldsymbol{g}_{n}(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2 = \left(\sum_{n\in\mathbb{B}} \left\|\boldsymbol{g}_{n}(\boldsymbol{\theta})\right\|^2\right) - |\mathbb{B}| \left\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2 \,, \tag{C.16b}$$

we arrive at

$$\frac{1}{|\mathbb{B}|(|\mathbb{B}|-1)} \left[\frac{\sum_{n\in\mathbb{B}} \left\|\boldsymbol{g}_{n}(\boldsymbol{\theta})\right\|^2}{\left\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2} - |\mathbb{B}|\right] \leq \theta_{\text{norm}}^2 \tag{C.16c}$$

that leverages the norm of both the mini-batch and the individual gradients, which can be aggregated over parameters during a backward pass. Cockpit's `NormTest` corresponds to the maximum radius θ_{norm} for which the above inequality holds.

Inner Product Test (`InnerTest`)

The inner product test [19] constrains the projection of $g_{\mathbb{B}}(\boldsymbol{\theta})$ onto $g_{p_{\text{data}}}(\boldsymbol{\theta})$ (compare Figure C.2a),

$$\operatorname{proj}_{g_{p_{\text{data}}}(\boldsymbol{\theta})}\left(g_{\mathbb{B}}(\boldsymbol{\theta})\right) := \frac{g_{\mathbb{B}}(\boldsymbol{\theta})^\top g_{p_{\text{data}}}(\boldsymbol{\theta})}{\left\|g_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2} g_{p_{\text{data}}}(\boldsymbol{\theta}), \tag{C.17}$$

rescaled by $\|g_{p_{\text{data}}}(\boldsymbol{\theta})\|$. This restricts the mini-batch gradient to reside in a standardized band of relative width $\theta_{\text{inner}} \in (0, \infty)$ around the population risk gradient. Bollapragada et al. [19] use $\theta_{\text{inner}} = 0.9$ (in the practical version, see below) to adapt the batch size if the parallel component's variance does not satisfy the condition

$$\operatorname{Var}\left(\frac{g_{\mathbb{B}}(\boldsymbol{\theta})^\top g_{p_{\text{data}}}(\boldsymbol{\theta})}{\left\|g_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2}\right) = \mathbb{E}\left[\left(\frac{g_{\mathbb{B}}(\boldsymbol{\theta})^\top g_{p_{\text{data}}}(\boldsymbol{\theta})}{\left\|g_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2} - 1\right)^2\right] \leq \theta_{\text{inner}}^2 \tag{C.18a}$$

(note that by Equation (C.14) we have $\mathbb{E}[g_{\mathbb{B}}(\boldsymbol{\theta})^\top g_{p_{\text{data}}}(\boldsymbol{\theta}) / \left\|g_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2] = 1$). Bollapragada et al. [19] bound Equation (C.18a) by the individual gradient variance,

$$\begin{aligned}&\frac{1}{|\mathbb{B}|}\operatorname{Var}\left(\frac{g_n(\boldsymbol{\theta})^\top g_{p_{\text{data}}}(\boldsymbol{\theta})}{\left\|g_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2}\right)\\ &\quad = \frac{1}{|\mathbb{B}|}\mathbb{E}\left[\left(\frac{g_n(\boldsymbol{\theta})^\top g_{p_{\text{data}}}(\boldsymbol{\theta})}{\left\|g_{p_{\text{data}}}(\boldsymbol{\theta})\right\|^2} - 1\right)^2\right] \leq \theta_{\text{inner}}^2 .\end{aligned} \tag{C.18b}$$

They then propose a practical form of Equation (C.18b), which uses the mini-batch sample variance,

$$\begin{aligned}&\frac{1}{|\mathbb{B}|}\operatorname{Var}\left(\frac{g_n(\boldsymbol{\theta})^\top g_{\mathbb{B}}(\boldsymbol{\theta})}{\left\|g_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2}\right)\\ &\quad = \frac{1}{|\mathbb{B}|(|\mathbb{B}|-1)}\left[\sum_{n\in\mathbb{B}}\left(\frac{g_n(\boldsymbol{\theta})^\top g_{\mathbb{B}}(\boldsymbol{\theta})}{\left\|g_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2} - 1\right)^2\right] \leq \theta_{\text{inner}}^2 .\end{aligned} \tag{C.19a}$$

Expanding

$$\sum_{n\in\mathbb{B}}\left(\frac{g_n(\boldsymbol{\theta})^\top g_{\mathbb{B}}(\boldsymbol{\theta})}{\left\|g_{\mathbb{B}}(\boldsymbol{\theta})\right\|^2} - 1\right)^2 = \frac{\sum_{n\in\mathbb{B}}\left(g_n(\boldsymbol{\theta})^\top g_{\mathbb{B}}(\boldsymbol{\theta})\right)^2}{\left\|g_{\mathbb{B}}(\boldsymbol{\theta})\right\|^4} - |\mathbb{B}| \tag{C.19b}$$

and inserting Equation (C.19b) into Equation (C.19a) yields

$$\frac{1}{|\mathbb{B}|(|\mathbb{B}|-1)}\left[\frac{\sum_{n\in\mathbb{B}}\left(g_n(\boldsymbol{\theta})^\top g_{\mathbb{B}}(\boldsymbol{\theta})\right)^2}{\left\|g_{\mathbb{B}}(\boldsymbol{\theta})\right\|^4} - |\mathbb{B}|\right] \leq \theta_{\text{inner}}^2 . \tag{C.19c}$$

It relies on pairwise scalar products between individual gradients, which can be aggregated over layers during backpropagation. Cockpit's `InnerTest` quantity computes the maximum band width θ_{inner} that satisfies Equation (C.19c).

Orthogonality Test (`OrthoTest`)

In contrast to the inner product test (Appendix C.3.6) which constrains the projection (Equation (C.17)), the orthogonality test [19] constrains the orthogonal part (see Figure C.2 (a))

$$\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) - \operatorname{proj}_{\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})}\left(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right) , \tag{C.20}$$

rescaled by $\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|$. This restricts the mini-batch gradient to a standardized band of relative width $\nu_{\text{ortho}} \in (0, \infty)$ parallel to the population gradient. Bollapragada et al. [19] use $\nu = \tan(80°) \approx 5.84$ (in the practical version, see below) to adapt the batch size if the following condition is violated,

$$\mathbb{E}\left[\left\|\frac{\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta}) - \operatorname{proj}_{\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})}\left(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right)}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|}\right\|^2\right] \leq \nu_{\text{ortho}}^2 . \tag{C.21a}$$

Expanding the norm, and inserting Equation (C.17), this simplifies to

$$\begin{aligned}\mathbb{E}\left[\left\|\frac{\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|} - \frac{\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|^2}\frac{\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|}\right\|^2\right] &\leq \nu_{\text{ortho}}^2 , \\ \mathbb{E}\left[\frac{\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|^2}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|^2} - \frac{\left(\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right)^2}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|^4}\right] &\leq \nu_{\text{ortho}}^2 .\end{aligned} \tag{C.21b}$$

Bollapragada et al. [19] bound this inequality with individual gradients,

$$\frac{1}{|\mathbb{B}|}\mathbb{E}\left[\left\|\frac{\boldsymbol{g}_n(\boldsymbol{\theta})}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|^2} - \frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\|^2}\frac{\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})\right\|}\right\|^2\right] \leq \nu_{\text{ortho}}^2 . \tag{C.21c}$$

They propose the practical form

$$\frac{1}{|\mathbb{B}|(|\mathbb{B}|-1)}\mathbb{E}\left[\left\|\frac{\boldsymbol{g}_n(\boldsymbol{\theta})}{\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|} - \frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|^2}\frac{\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right\|}\right\|^2\right] \leq \nu_{\text{ortho}}^2 , \tag{C.22a}$$

which simplifies to

$$\frac{1}{|\mathbb{B}|(|\mathbb{B}|-1)}\sum_{n\in\mathbb{B}}\left(\frac{\|\boldsymbol{g}_n(\boldsymbol{\theta})\|^2}{\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|^2} - \frac{\left(\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right)^2}{\|\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\|^4}\right) \leq \nu_{\text{ortho}}^2 . \tag{C.22b}$$

It relies on pairwise scalar products between individual gradients which can be aggregated over layers during a backward pass. Cockpit's `OrthoTest` quantity computes the maximum band width ν_{ortho} which satisfies Equation (C.22b).

Relation to Acute Angle Test

Recently, a novel "acute angle test" was proposed by Bahamou and Goldfarb [6]. While the theoretical constraint between $\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})$ and $\boldsymbol{g}_{p_{\text{data}}}(\boldsymbol{\theta})$ differs from the orthogonality test, the practical versions coincide. Hence, we do not incorporate the acute angle here.

C.3.7 Hessian Maximum Eigenvalue (`HessMaxEV`)

The Hessian's maximum eigenvalue $\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$ is computed with an iterative eigensolver from Hessian-vector products through PyTorch's automatic differentiation [127]. Like Yao et al. [177], we employ power iterations with similar default stopping parameters (stop after at most 100 iterations, or if the iterate does converged with a relative and absolute tolerance of $10^{-3}, 10^{-6}$, respectively) to compute $\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$ through the `HessMaxEV` quantity in Cockpit.

In principle, more sophisticated eigensolvers (for example Arnoldi's method) could be applied to converge in fewer iterations or compute eigenvalues other than the leading ones. Warsa et al. [170] empirically demonstrate that the FLOP ratio between power iteration and implicitly restarted Arnoldi method can reach values larger than 100. While we can use such a beneficial method on a CPU through `scipy.sparse.linalg.eigsh` we are restricted to the GPU-compatible power iteration for GPU training. We expect that extending the support of popular machine learning libraries like PyTorch for such iterative eigensolvers on GPUs can help to save computation time.

$$\lambda_{\max}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})) = \max_{\|\boldsymbol{v}\|_2=1} \|\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\boldsymbol{v}\|_2 = \max_{\boldsymbol{v}\in\mathbb{R}^D} \frac{\boldsymbol{v}^\top \boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\boldsymbol{v}}{\boldsymbol{v}^\top \boldsymbol{v}}. \tag{C.23}$$

Usage

The Hessian's maximum eigenvalue describes the loss surface's sharpest direction and thus provides an understanding of the current loss landscape. Additionally, in convex optimization, the largest Hessian eigenvalue crucially determines the appropriate step size [144]. In Section 6.4, we can observe that although training seems stuck in the very first few iterations progress is visible when looking at the maximum Hessian eigenvalue.

C.3.8 Hessian Trace (`HessTrace`)

In comparison to Yao et al. [177], who leverage Hessian-vector products [127] to estimate the Hessian trace, we compute the exact value $\operatorname{Tr}(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta}))$ with the `HessTrace` quantity in Cockpit by aggregating the output of BackPACK's `DiagHessian` extension, which computes the diagonal entries of $\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})$. Alternatively, the trace can also be estimated with the GGN matrix, or an MC-sampled approximation thereof.

Usage

The Hessian trace equals the sum of the eigenvalues and thus provides a notion of "average curvature" of the current loss landscape. It has long been theorized and discussed that curvature and generalization performance may be linked [70, *e.g.*].

C.3.9 Takeuchi Information Criterion (`TICDiag`, `TICTrace`)

Recent work by Thomas et al. [162] suggests that optimizer convergence speed and generalization is mainly influenced by curvature and gradient noise; and hence their interaction is crucial to understand the generalization and optimization behavior of deep neural networks. They reinvestigate the Takeuchi Information criterion [160], an estimator for the generalization gap in overparameterized maximum likelihood estimation. At a local minimum $\boldsymbol{\theta}_\star$, the generalization gap is estimated by the TIC

$$\frac{1}{|\mathbb{D}|}\operatorname{Tr}\left(\boldsymbol{H}_{p_{\text{data}}}(\boldsymbol{\theta}_\star)^{-1}\boldsymbol{K}_{p_{\text{data}}}(\boldsymbol{\theta}_\star)\right)\,, \tag{C.24}$$

where $\boldsymbol{H}_{p_{\text{data}}}(\boldsymbol{\theta}_\star)$ is the population Hessian and $\boldsymbol{K}_{p_{\text{data}}}(\boldsymbol{\theta}_\star)$ is the gradient's uncentered second moment,

$$\boldsymbol{K}_{p_{\text{data}}}(\boldsymbol{\theta}_\star) = \int \nabla_{\boldsymbol{\theta}_\star}\ell(f_{\boldsymbol{\theta}_\star}(\boldsymbol{x}),\boldsymbol{y})\left(\nabla_{\boldsymbol{\theta}_\star}\ell(f_{\boldsymbol{\theta}_\star}(\boldsymbol{x}),\boldsymbol{y})\right)^\top p_{\text{data}}(\boldsymbol{x},\boldsymbol{y})\mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{y}.$$

Both matrices are inaccessible in practice. In their experiments, Thomas et al. [162] propose the approximation $\operatorname{Tr}(\boldsymbol{K})/\operatorname{Tr}(\boldsymbol{H})$ for $\operatorname{Tr}(\boldsymbol{H}^{-1}\boldsymbol{K})$. They also replace the Hessian by the Fisher as it is easier to compute. With these practical simplifications, they investigate the TIC of trained neural networks where the curvature and noise matrix are evaluated on a large dataset.

The TIC provided in Cockpit differs from this setting, since by design we want to observe quantities during training, while avoiding additional model predictions. Also, BackPACK provides access to the Hessian; hence we don't need to use the Fisher. We propose the following two approximations of the TIC from a mini-batch:

- `TICTrace`: uses the approximation of Thomas et al. [162] which replaces the matrix-product trace by the product of traces,

$$\frac{\operatorname{Tr}\left(\boldsymbol{K}_{\mathbb{B}}(\boldsymbol{\theta})\right)}{\operatorname{Tr}\left(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right)} = \frac{\frac{1}{|\mathbb{B}|}\sum_{n\in\mathbb{B}}\|\boldsymbol{g}_n(\boldsymbol{\theta})\|^2}{\operatorname{Tr}\left(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right)}\,. \tag{C.25}$$

- `TICDiag`: uses a diagonal approximation of the Hessian, which is cheap to invert,

$$\begin{aligned}&\operatorname{Tr}\left(\operatorname{diag}\left(\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right)^{-1}\boldsymbol{K}_{\mathbb{B}}(\boldsymbol{\theta})\right)\\ &\quad= \frac{1}{|\mathbb{B}|}\sum_{d=1}^{D}\left[\boldsymbol{H}_{\mathbb{B}}(\boldsymbol{\theta})\right]^{-1}_{d,d}\left[\sum_{n\in\mathbb{B}}\boldsymbol{g}_n(\boldsymbol{\theta})^{\odot 2}\right]_d\,.\end{aligned} \tag{C.26}$$

Usage

The TIC is a proxy for the generalization gap, see Thomas et al. [162].

C.3.10 Gradient Signal-to-noise Ratio (`MeanGSNR`)

The gradient signal-to-noise ratio $\operatorname{GSNR}([\boldsymbol{\theta}]_d) \in \mathbb{R}$ for a single parameter $[\boldsymbol{\theta}]_d$ is defined as

$$\operatorname{GSNR}([\boldsymbol{\theta}]_d) = \frac{\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[[\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y})]_d\right]^2}{\operatorname{Var}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[[\nabla_{\boldsymbol{\theta}}\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}),\boldsymbol{y})]_d\right]} = \frac{\left[\boldsymbol{g}_P(\boldsymbol{\theta})\right]_d^2}{[\boldsymbol{\Sigma}_P(\boldsymbol{\theta})]_{d,d}}\,. \tag{C.27}$$

Liu et al. [100] use it to explain generalization properties of models in the early training phase. We apply their estimation to mini-batches,

$$\operatorname{GSNR}([\boldsymbol{\theta}]_d) \approx \frac{\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right]_d^2}{\frac{|\mathbb{B}|-1}{|\mathbb{B}|}\left[\hat{\boldsymbol{\Sigma}}_{\mathbb{B}}(\boldsymbol{\theta})\right]_{d,d}} = \frac{\left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right]_d^2}{\frac{1}{|\mathbb{B}|}\left(\sum_{n\in\mathbb{B}}\left[\boldsymbol{g}_n(\boldsymbol{\theta})\right]_d^2\right) - \left[\boldsymbol{g}_{\mathbb{B}}(\boldsymbol{\theta})\right]_d^2}\,. \tag{C.28a}$$

Inspired by Liu et al. [100], Cockpit's `MeanGSNR` computes the average GSNR over all parameters,

$$\frac{1}{D}\sum_{j=1}^{D}\operatorname{GSNR}([\boldsymbol{\theta}]_j)\,. \tag{C.28b}$$

Usage

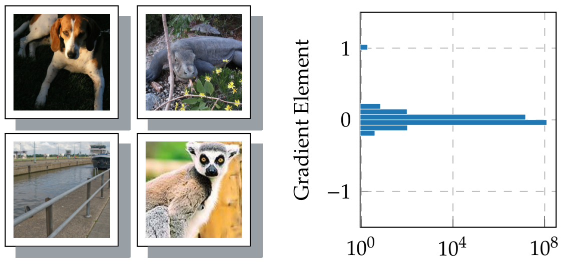
The GSNR describes the gradient noise level which is influenced, among other things, by the batch size. Using the GSNR, perhaps in combination with the gradient tests or the CABS criterion could provide practitioners a clearer picture of suitable batch sizes for their particular problem. As shown by Liu et al. [100], the GSNR is also linked to generalization of neural networks.

C.4 Additional Experiments

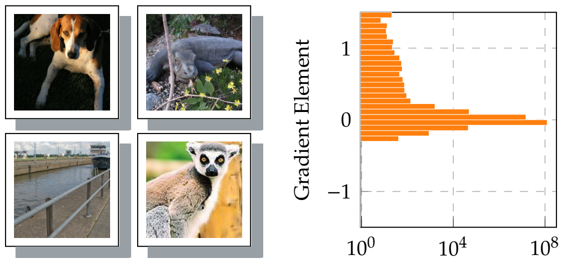
In this section, we present additional experiments and use cases that showcase Cockpit's utility. Appendix C.4.1 shows that Cockpit is able to scale to larger datasets by running the experiment with incorrectly scaled data (see Section 6.3.1) on ImageNet instead of CIFAR-10. Appendix C.4.2 provides another concrete use case similar to Figure 6.1: detecting regularization during training.

C.4.1 Incorrectly Scaled Data for ImageNet

We repeat the experiment of Section 6.3.1 on the ImageNet [41] dataset instead of CIFAR-10. We also use a larger neural network model, switching from 3c3d to VGG16 [153]. This demonstrates that Cockpit is able to scale



(a) Normalized Data



(b) Raw Data

Figure C.3: Same inputs, different gradients on ImageNet. This is structurally the same plot as Figure 6.3, but using ImageNet and VGG16. **(a)** *normalized* ($[0, 1]$) and **(b)** *raw* ($[0, 255]$) images look identical in auto-scaled front-ends like matplotlib's `imshow`. The gradient distribution on the VGG16 model, however, is affected by this scaling.

to both larger models and datasets. The input size of the images is almost fifty times larger (224×224 instead of 32×32). The model size increased by roughly a factor of 150 (VGG16 for ImageNet has roughly 138 million parameters, 3c3d has less than a million).

Similar to the example in the main text, the gradients are affected by the scaling introduced via the input images, albeit less drastically (see Figure C.3). Due to the gradient scaling, default optimization hyper-parameters might not work well anymore for the model using the raw data.

C.4.2 Detecting Implicit Regularization of The Optimizer

In non-convex optimization, optimizers can converge to local minima with different properties. Here, we illustrate this by investigating the effect of sub-sampling noise on a simple task from [56, 112].

We generate synthetic data $\mathbb{D} = \{(x_n, y_n) \in \mathbb{R} \times \mathbb{R}\}_{n=1}^{N=100}$ for a regression task with $x \sim \mathcal{N}(x \mid 0, 1)$ with noisy observations $y = 1.4x + \epsilon$ where $\epsilon \sim \mathcal{N}(\epsilon \mid 0, 1)$. The model is a scalar net with parameters $\boldsymbol{\theta} = \begin{pmatrix} w_1 & w_2 \end{pmatrix}^\top \in \mathbb{R}^2$, initialized at $\boldsymbol{\theta}_0 = \begin{pmatrix} 0.1 & 1.7 \end{pmatrix}^\top$, that produces predictions $f_{\boldsymbol{\theta}}(x) = w_2 w_1 x$. We seek to minimize the mean squared error

$$\mathcal{L}_{\mathbb{D}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (f_{\boldsymbol{\theta}}(x_n) - y_n)^2$$

and compare SGD ($|\mathbb{B}| = 95$) with GD ($|\mathbb{B}| = N = 100$) at a learning rate of 0.1 (see Figure C.4).

We observe that the loss of both SGD and GD is almost identical. Using a noisy gradient regularizes the Hessian's maximum eigenvalue though. It decreases in later stages where the loss curve suggests that training has converged. This regularization effect constitutes an important phenomenon that cannot be observed by monitoring only the loss.

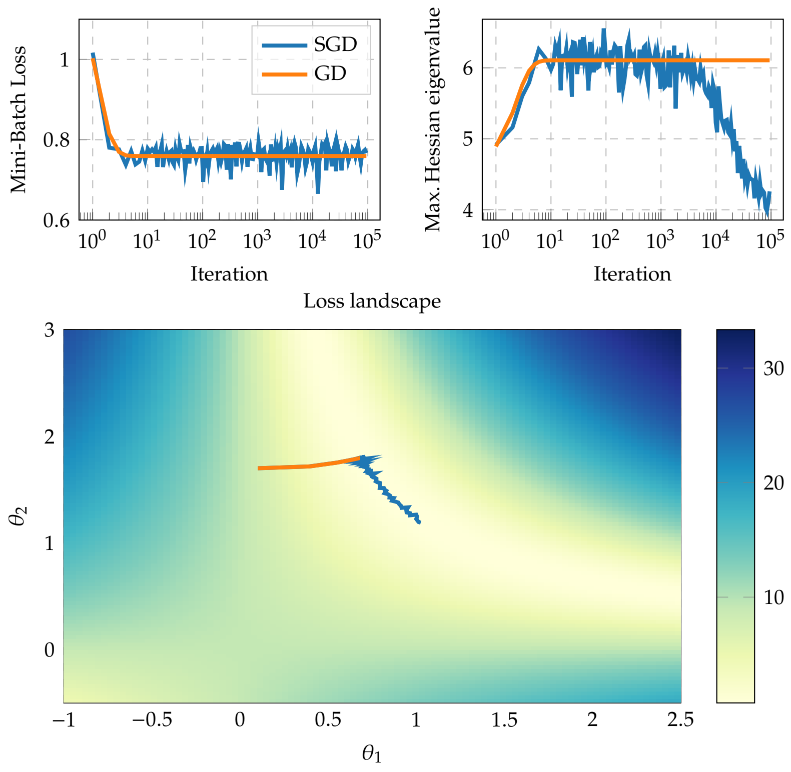


Figure C.4: **Observing implicit regularization of the optimizer with Cockpit** through a comparison of SGD and GD on a synthetic problem inspired by [56, 112] (details in the text). *Top left:* The mini-batch loss of both optimizers looks similar. *Top right:* Noise due to mini-batching regularizes the Hessian's maximum eigenvalue in stages where the loss suggests that training has converged. *Bottom:* Optimization trajectories in parameter space. SGD is attracted to the flattest minimum.

C.5 Implementation Details & Additional Benchmarks

In this section, we provide more details about our implementation (Appendix C.5.1) to access the desired quantities with as little overhead as possible. Additionally, we present more benchmarks for individual instruments (Appendix C.5.2) and Cockpit configurations (Appendix C.5.2). These are similar but extended versions of the ones presented in Figures 6.6a and 6.6b in the main text. Lastly, we benchmark different implementations of computing the two-dimensional gradient histogram (Appendix C.5.3), identifying a computational bottleneck for its current GPU implementation.

Hardware Details

We conducted benchmarks on the following setup:

- **CPU:** Intel Core i7-8700K CPU @ 3.70 GHz × 12 (32 GB)
- **GPU:** NVIDIA GeForce RTX 2080 Ti (11 GB)

Test Problem Details

The experiments in this paper rely mostly on optimization problems provided by the DeepOBS benchmark suite [146]. If not stated otherwise, we use the default training details suggested by DeepOBS, that are summarized below. For more details see the original paper.

- **Quadratic Deep:** A stochastic quadratic problem with an eigenspectrum similar to what has been reported for neural nets. Default batch size 128, default number of epochs 100.
- **MNIST Log. Reg.:** Multinomial logistic regression on MNIST [95]. Default batch size 128, default number of epochs 50.
- **MNIST MLP:** Multi-layer perceptron on MNIST. Default batch size 128, default number of epochs 100.
- **Fashion-MNIST MLP:** Multi-layer perceptron on Fashion-MNIST [175]. Default batch size 128, default number of epochs 100.
- **Fashion-MNIST 2c2d:** A two convolutional and two dense layered neural network on Fashion-MNIST. Default batch size 128, default number of epochs 100.
- **CIFAR-10 3c3d:** A three convolutional and three dense layered neural network on CIFAR-10 [90]. Default batch size 128, default number of epochs 100.
- **CIFAR-100 All-CNN-C:** All Convolutional Neural Network C (All-CNN-C [157]) on CIFAR-100 [90]. Default batch size 256, default number of epochs 350.
- **SVHN 3c3d:** A three convolutional and three dense layered neural network on SVHN [118]. Default batch size 128, default number of epochs 100.

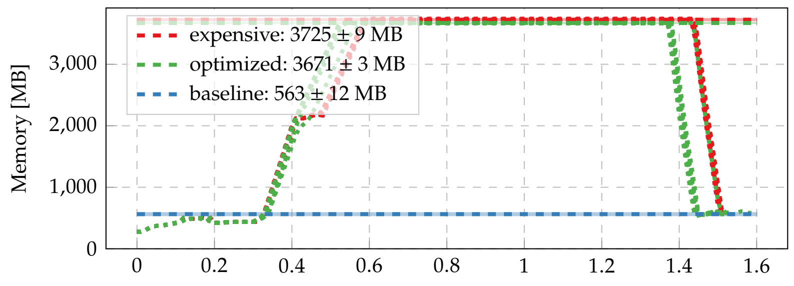
C.5.1 Hooks & Memory Benchmarks

To improve memory consumption, we compact information during the backward pass by adding hooks to the neural network's layers. These are executed after BackPACK extensions and have access to the quantities computed therein. They compress information to what is requested by a quantity and free the memory occupied by BackPACK buffers. Such savings primarily depend on the parameter distribution over layers, and are bigger for more balanced architectures (compare Figure C.5).

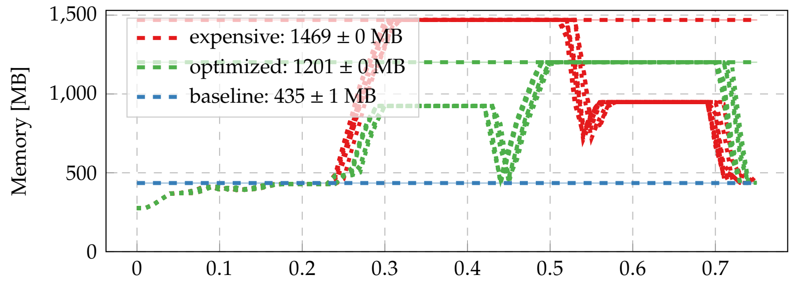
Example

Say, we want to compute a histogram over the $|\mathbb{B}| \times D$ individual gradient elements of a network. Suppose that $|\mathbb{B}| = 128$ and the model is DeepOBS's CIFAR-10 3c3d test problem with $895,210$ parameters. Given that every parameter is stored in single precision, the model requires $895,210 \times 4\,\text{Bytes} \approx 3.41\,\text{MB}$. Storing the individual gradients will require $128 \times 895,210 \times 4\,\text{Bytes} \approx 437\,\text{MB}$ (for larger networks this quickly exceeds the available memory as the individual gradients occupy $|\mathbb{B}|$ times the model size). If instead, the layer-wise individual gradients are condensed into histograms of negligible size and immediately freed afterwards during backpropagation, the maximum memory overhead reduces to storing the individual gradients of the largest layer. For our example, the largest layer has $589,824$ parameters, and the associated individual gradients will require $128 \times 589,824 \times 4\,\text{Bytes} \approx 288\,\text{MB}$, saving roughly 149 MB of RAM. In practice, we observe these expected savings, see Figure C.5c.

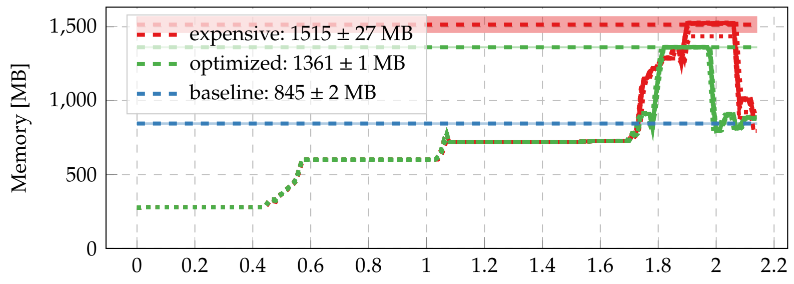
(a) Fashion-MNIST 2c2d



(b) MNIST MLP



(c) CIFAR-10 3c3d



(d) CIFAR-100 All-CNN-C

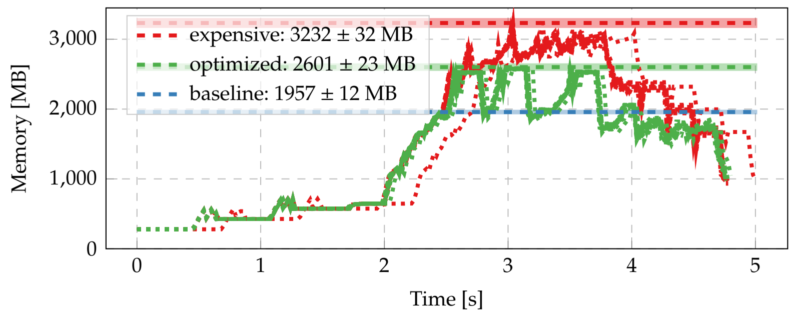


Figure C.5: Memory consumption and savings with hooks during one forward-backward step on a CPU for different DeepOBS problems. We compare three settings; i) without Cockpit (baseline); ii) Cockpit with `GradHist1d` with BackPACK (expensive); iii) Cockpit with `GradHist1d` with BackPACK and additional hooks (optimized). Peak memory consumptions are highlighted by horizontal dashed bars and shown in the legend. Shaded areas, if visible, fill two standard deviations above and below the mean value, all of them result from ten independent runs. Dotted lines indicate individual runs. Our optimized approach allows to free obsolete tensors during backpropagation and thereby reduces memory consumption. From top to bottom: the effect is less pronounced for architectures that concentrate the majority of parameters in a single layer (**(a)** $3,274,634$ total, $3,211,264$ largest layer) and increases for more balanced networks **(b)** $1,336,610$ total, $784,000$ largest layer, **(c)**: $895,210$ total, $589,824$ largest layer).

C.5.2 Additional Run Time Benchmarks

Individual Instrument Overhead

To estimate the computational overhead for individual instruments, we run Cockpit with that instrument for 32 iterations, tracking at every step. Training proceeds with the default batch size specified by the DeepOBS problem and uses SGD with learning rate 10^{-3}. We measure the time between iterations 1 and 32, and average for the overhead per step. Every such estimate is repeated over 10 random seeds to obtain mean and error bars as reported in Figure 6.6a.

Note that this protocol does *not* include initial overhead for setting up data loading and also does *not* include the time for evaluating train/test loss on a larger dataset, which is usually done by practitioners. Hence, we even expect the shown overheads to be smaller in a conventional training loop which includes the above steps.

Individual Overhead on GPU Versus CPU

Figure C.6 and Figure C.7 show the individual overhead for four different DeepOBS problems on GPU and CPU, respectively. The left part of Figure C.6 (c) corresponds to Figure 6.6a. Right panels show the expensive quantities, which we omitted in the main text as they were expected to be expensive due to their computational work (`HessMaxEV`) or bottlenecks in the implementation (`GradHist2d`, see Appendix C.5.3 for details). We see that they are in many cases equally or more expensive than computing all other instruments. Another expected feature of the GPU-to-CPU comparison is that parallelism on the CPU is significantly less pronounced. Hence, we observe an increased overhead for all quantities that contain non-linear transformations and contractions of the high-dimensional individual gradients, or require additional backpropagations (curvature).

Configuration Overhead

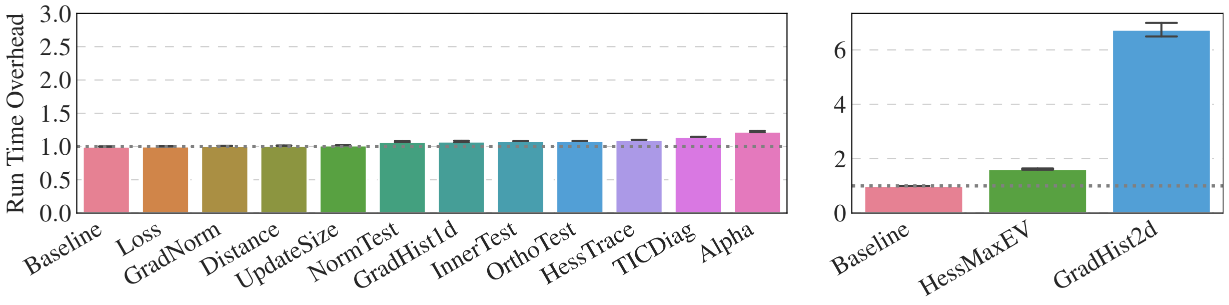
For the estimation of different Cockpit configuration overheads, we use almost the same setting as described above, training for 512 iterations and tracking only every specified interval.

Configuration Overhead on GPU Versus CPU

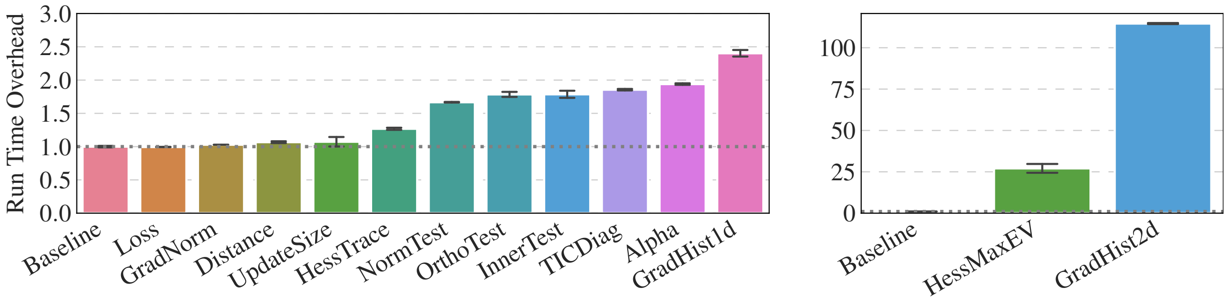
Figure C.8 and Figure C.9 show the configuration overhead for four different DeepOBS problems. The bottom left part of Figure C.8 corresponds to Figure 6.6b. In general, increased parallelism can be exploited on a GPU, leading to smaller overheads in comparison to a CPU.

Cockpit can even scale to significantly larger problems, such as a ResNet-50 on ImageNet-like data. Figure C.10 shows the computational overhead for different tracking intervals on such a large-scale problem. Using the *economy* configuration, we can achieve our self-imposed goal of at most doubling the run time even when tracking every fourth step. More extensive configurations (such as the *full* set) would indeed have almost prohibitively large costs associated. However, these costs could be dramatically reduced when one decides to only inspect a part of the network using Cockpit. Note, individual gradients are not properly defined when using batch norm, therefore, we replaced these batch norm layers with identity layers when using the ResNet-50.

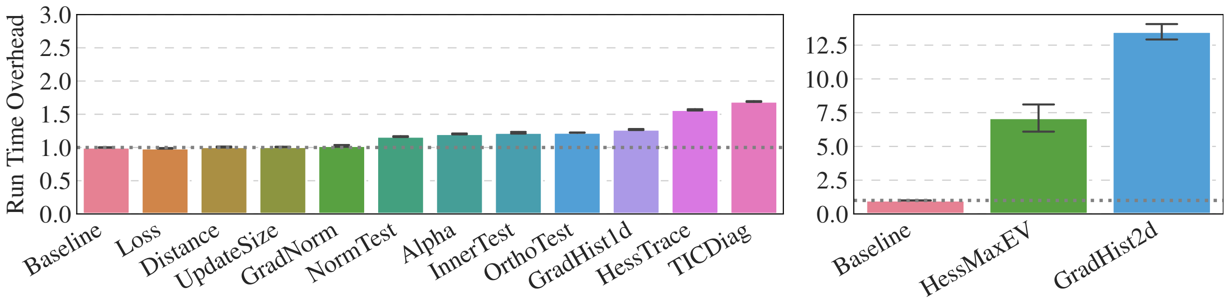
(a) Computational overhead for MNIST LogReg (GPU)



(b) Computational overhead for MNIST MLP (GPU)



(c) Computational overhead for CIFAR-10 3c3d (GPU)



(d) Computational overhead for Fashion-MNIST 2c2d (GPU)

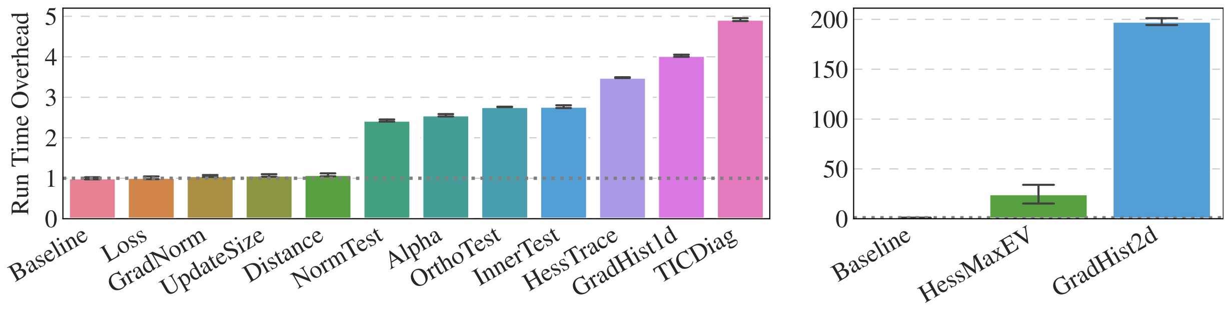
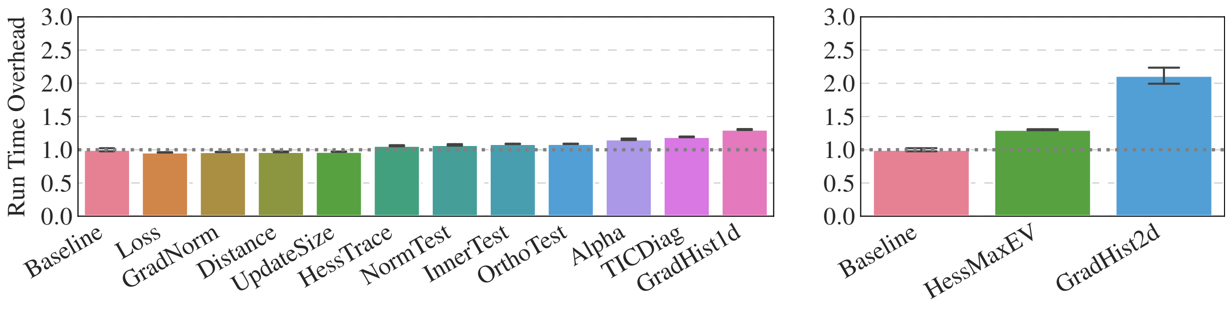
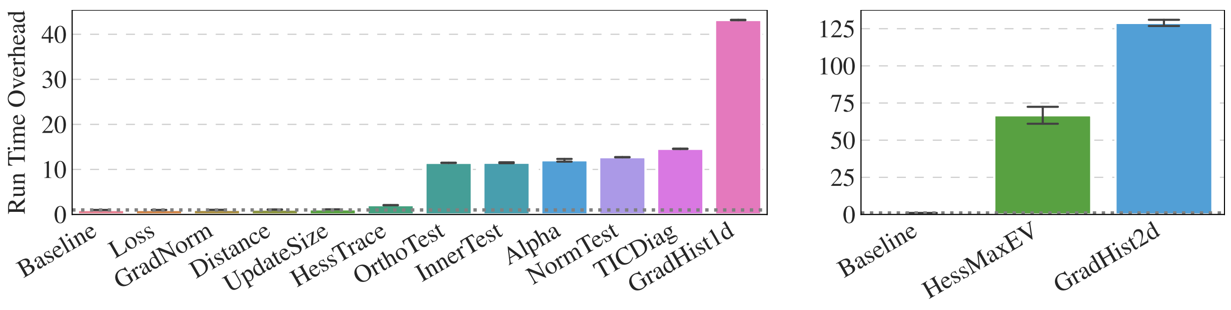


Figure C.6: Individual overhead of Cockpit's instruments on GPU for four different problems. All run times are shown as multiples of the *baseline* without tracking. Expensive quantities are displayed in separate panels on the right. Experimental details in the text.

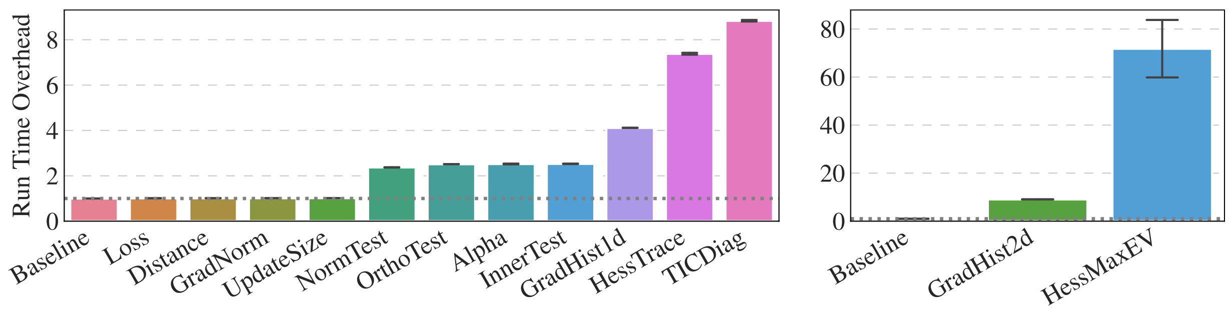
(a) Computational overhead for MNIST LogReg (CPU)



(b) Computational overhead for MNIST MLP (CPU)



(c) Computational overhead for CIFAR-10 3c3d (CPU)



(d) Computational overhead for Fashion-MNIST 2c2d (CPU)

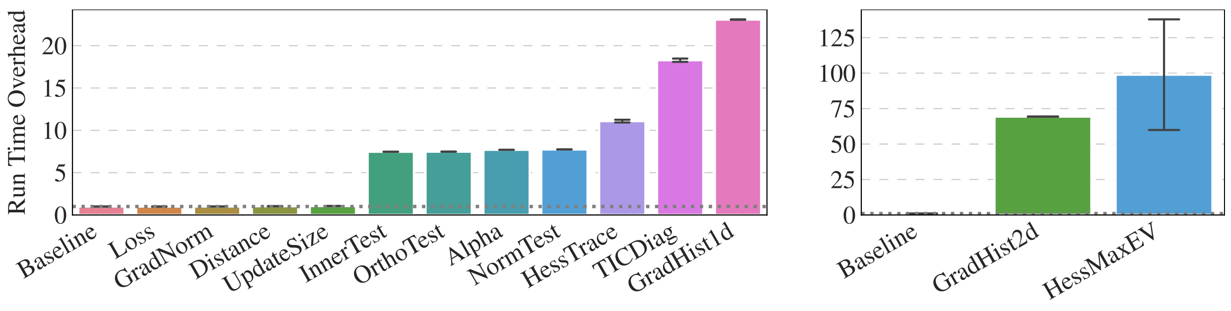
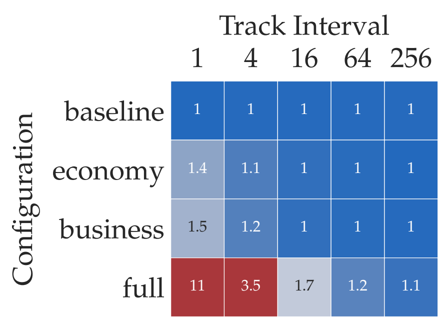
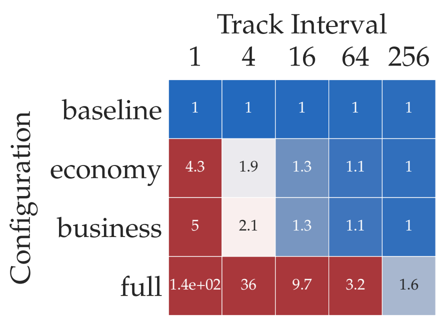


Figure C.7: Individual overhead of Cockpit's instruments on CPU for four different problems. All run times are shown as multiples of the *baseline* without tracking. Expensive quantities are displayed in separate panels on the right. Experimental details in the text.

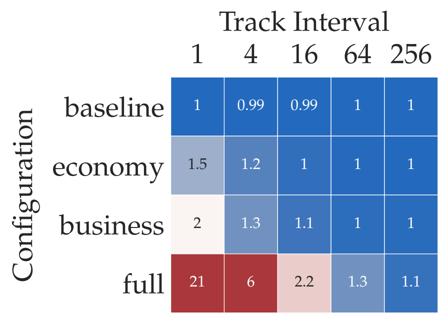
(a) MNIST LogReg (GPU)



(b) MNIST MLP (GPU)



(c) CIFAR-10 3c3d (GPU)



(d) Fashion-MNIST 2c2d (GPU)

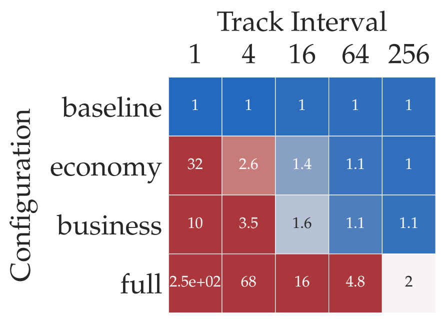
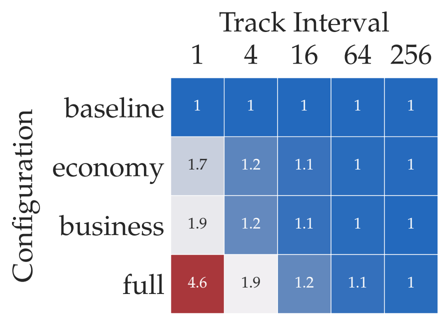
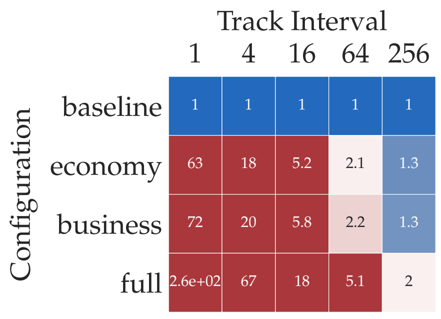


Figure C.8: Overhead of Cockpit configurations on GPU for four different problems with varying tracking interval. Color bar is the same as in Figure 6.6.

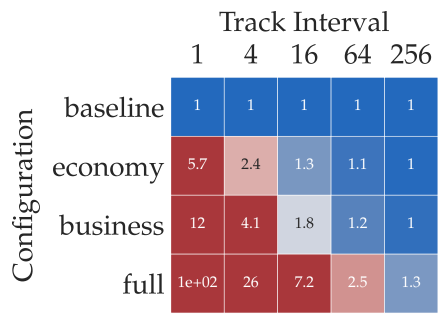
(a) MNIST LogReg (CPU)



(b) MNIST MLP (CPU)



(c) CIFAR-10 3c3d (CPU)



(d) Fashion-MNIST 2c2d (CPU)

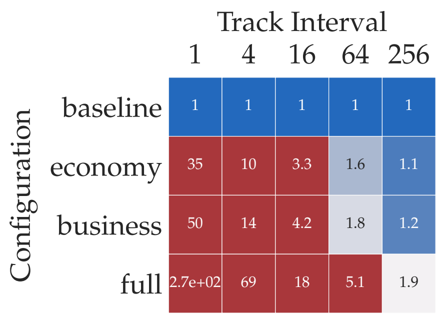


Figure C.9: Overhead of Cockpit configurations on CPU for four different problems with varying tracking interval. Color bar is the same as in Figure 6.6.

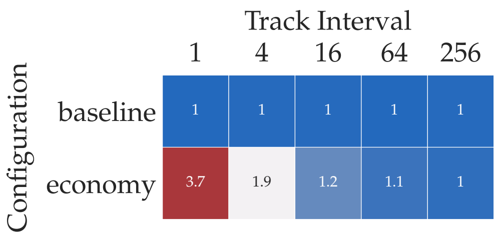


Figure C.10: Overhead of Cockpit configurations on GPU for ResNet-50 on ImageNet. Cockpit's instruments scale efficiently even to very large problems (here: 1000 classes, $(3, 224, 224)$-sized inputs, and a batch size of 64. For individual gradients to be defined, we replaced the batch norm layers of the ResNet-50 model with identities.) Color bar is the same as in Figure 6.6.

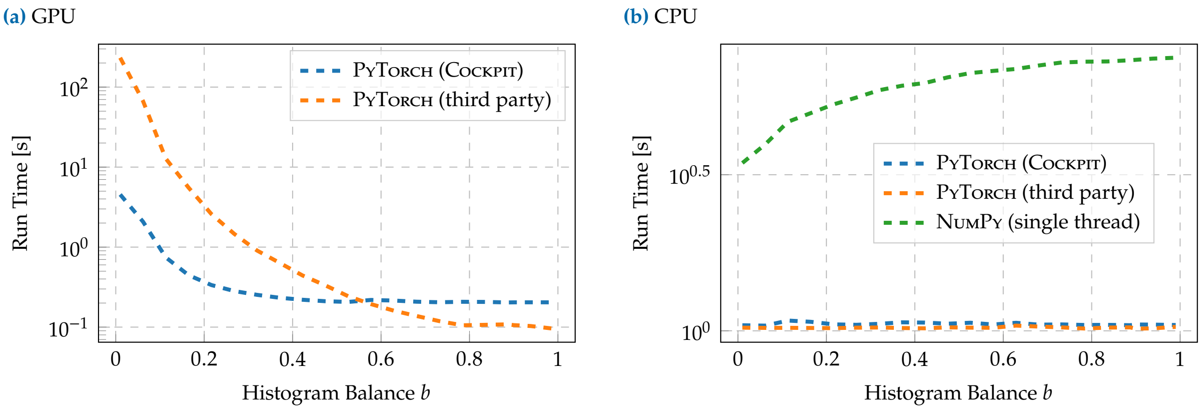


Figure C.11: Performance of two-dimensional histogram GPU implementations depends on the data. (a) Run time for two different GPU implementations with histograms of different imbalance. Cockpit's implementation outperforms the third party solution by more than one order of magnitude in the deep learning regime ($b \ll 1$). (b) On CPU, performance is robust to histogram balance. The run time difference between NumPy and PyTorch is due to multi-threading. Data has the same size as DeepOBS's CIFAR-10 3c3d problem ($D = 895, 210, |\mathbb{B}| = 128$). Curves represent averages over 10 independent runs. Error bars are omitted to improve legibility.

C.5.3 Performance of Two-dimensional Histograms

Both one- and two-dimensional histograms require $|\mathbb{B}| \times D$ elements be accessed, and hence perform similarly. However, we observed different behavior on GPU and decided to omit the two-dimensional histogram's run time in the main text. As explained here, this performance lack is not fundamental, but a shortcoming of the GPU implementation. PyTorch provides built-in functionality for computing one-dimensional histograms at the time of writing, but is not yet featuring multi-dimensional histograms. We experimented with three implementations:

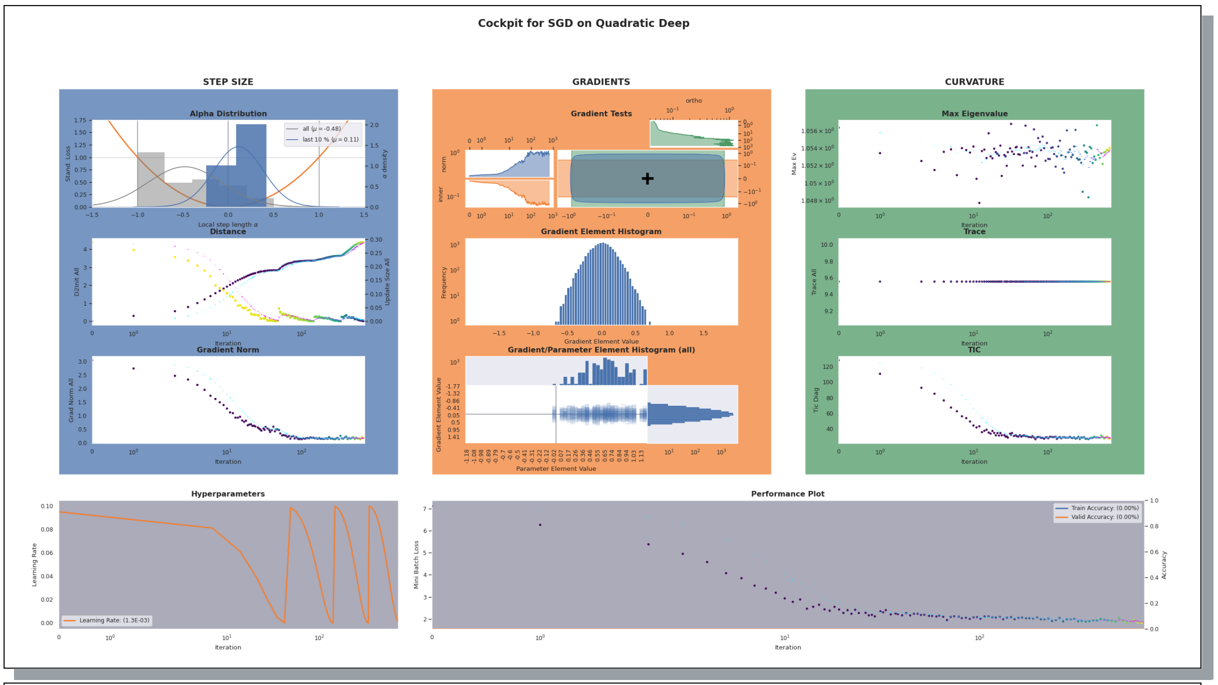
- **PyTorch (third party):** A third party implementation[*] under review for being integrated into PyTorch[†]. It relies on `torch.bincount`, which uses `atomicAdd`s that represent a bottleneck for histograms where most counts are contained in one bin.[‡] This occurs often for over-parameterized deep models, as most of the gradient elements are zero.
- **PyTorch (Cockpit):** Our implementation uses a workaround, computes bin indices and scatters the counts into their associated bins with `torch.Tensor.put_`. This circumvents `atomicAdd`s, but has poor memory locality.
- **NumPy:** The single-threaded `numpy.histogram2d` serves as baseline, but does not run on GPUs.

[*] Permission granted by the authors of github.com/miranov25/.../histogramdd_-pytorch.py.
[†] See https://github.com/pytorch/pytorch/pull/44485.
[‡] See https://discuss.pytorch.org/t/torch-bincount-1000x-slower-on-cuda/42654

To demonstrate the strong performance dependence on the data, we generate data from a uniform distribution over $[0, b] \times [0, b]$, where $b \in (0, 1)$ parametrizes the histogram's balance, and compute two-dimensional histograms on $[0, 1] \times [0, 1]$. Figure C.11a shows a clear increase in run time of both GPU implementations for more imbalanced histograms. Note that even though our implementation outperforms the third party by more than one order of magnitude in the deep neural network regime ($b \ll 1$), it is still considerably slower than a one-dimensional histogram (see Figure C.6 (c)), and even slower on GPU than on CPU (Figure C.11 (b)). As expected, the CPU implementations do not significantly depend on the data (Figure C.11b). The performance difference between PyTorch and NumPy is likely due to multi-threading versus single-threading.

Although a carefully engineered histogram GPU implementation is currently not available, we think it will reduce the computational overhead to that of a one-dimensional histogram in future releases.

C.6 Cockpit View of Convex Stochastic Problems



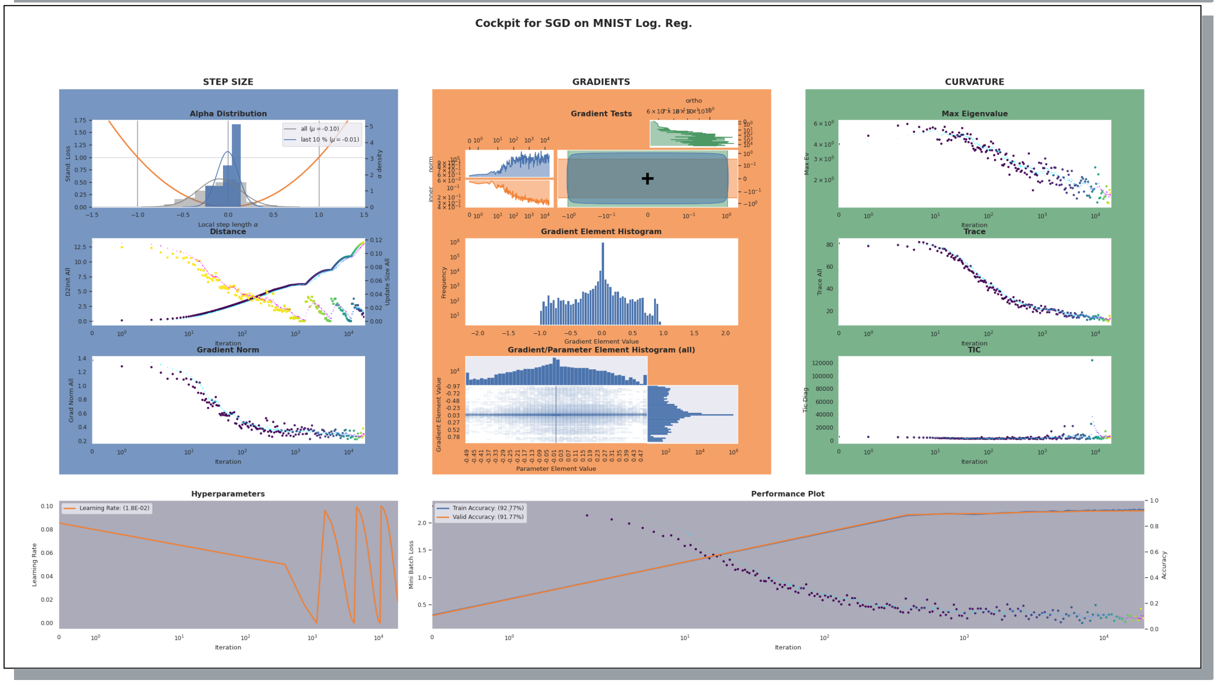


Figure C.12: Screenshot of Cockpit's full view for convex DeepOBS problems. Top Cockpit shows training on a noisy quadratic loss function. Bottom shows training on logistic regression on MNIST. Figure and labels are not meant to be legible. It is evident, that there is a fundamental difference in the optimization process, compared to training deep networks, *i.e.* Figure 6.2. This is, for example, visible when comparing the gradient norms, which converge to zero for convex problems but not for deep learning.

Additional Material for Chapter 7 D.

D.1 Mathematical Details

D.1.1 The GNN's Eigenvalues & the Gram Matrix

For Equation (7.4), consider the left hand side of the GGN's characteristic polynomial $\det(\boldsymbol{G} - \lambda \boldsymbol{I}_D) = 0$. Inserting the ViViT factorization (Equation (7.3)) and using the matrix determinant lemma yields

$$
\begin{aligned}
&\det\left(-\lambda \boldsymbol{I}_D + \boldsymbol{G}\right) \\
&\quad = \det\left(-\lambda \boldsymbol{I}_D + \boldsymbol{V}\boldsymbol{V}^\top\right) && \text{(Low-rank structure (7.3))} \\
&\quad = \det\left(\boldsymbol{I}_{NC} + \boldsymbol{V}^\top(-\lambda \boldsymbol{I}_D)^{-1}\boldsymbol{V}\right)\det(-\lambda \boldsymbol{I}_D) && \text{(Matrix determinant lemma)} \\
&\quad = \det\left(\boldsymbol{I}_{NC} - \frac{1}{\lambda}\boldsymbol{V}^\top\boldsymbol{V}\right)(-\lambda)^D \\
&\quad = \left(-\frac{1}{\lambda}\right)^{NC}\det\left(\boldsymbol{V}^\top\boldsymbol{V} - \lambda \boldsymbol{I}_{NC}\right)(-\lambda)^D \\
&\quad = (-\lambda)^{D-NC}\det\left(\tilde{\boldsymbol{G}} - \lambda \boldsymbol{I}_{NC}\right)\,. && \text{(Gram matrix)}
\end{aligned}
$$

Setting the above expression to zero reveals that the GGN's spectrum decomposes into $D - NC$ zero eigenvalues and the Gram matrix spectrum obtained from $\det(\tilde{\boldsymbol{G}} - \lambda \boldsymbol{I}_{NC}) = 0$.

D.1.2 Relation Between GGN & Gram Matrix Eigenvectors

Assume the nontrivial Gram matrix spectrum $\tilde{\mathbb{S}}_+ := \{(\lambda_k, \tilde{\boldsymbol{e}}_k) \mid \lambda_k \neq 0, \tilde{\boldsymbol{G}}\tilde{\boldsymbol{e}}_k = \lambda_k \tilde{\boldsymbol{e}}_k\}_{k=1}^K$ with orthonormal eigenvectors $\tilde{\boldsymbol{e}}_j^\top \tilde{\boldsymbol{e}}_k = \delta_{j,k}$ (δ is the Kronecker delta) and $K = \operatorname{rank}(\boldsymbol{G})$. We now show that $\boldsymbol{e}_k = {}^1\!/\!\sqrt{\lambda_k}\,\boldsymbol{V}\tilde{\boldsymbol{e}}_k$ are normalized eigenvectors of $\boldsymbol{G}$ and inherit orthogonality from $\tilde{\boldsymbol{e}}_k$.

To see the first, consider right-multiplication of the GGN with $\boldsymbol{e}_k$, then expand the low-rank structure,

$$
\begin{aligned}
\boldsymbol{G}\boldsymbol{e}_k &= \frac{1}{\sqrt{\lambda_k}}\boldsymbol{V}\boldsymbol{V}^\top\boldsymbol{V}\tilde{\boldsymbol{e}}_k && \text{(Equation (7.3) and definition of } \boldsymbol{e}_k\text{)} \\
&= \frac{1}{\sqrt{\lambda_k}}\boldsymbol{V}\tilde{\boldsymbol{G}}\tilde{\boldsymbol{e}}_k && \text{(Gram matrix)} \\
&= \lambda_k\frac{1}{\sqrt{\lambda_k}}\boldsymbol{V}\tilde{\boldsymbol{e}}_k && \text{(Eigenvector property of } \tilde{\boldsymbol{e}}_k\text{)} \\
&= \lambda_k \boldsymbol{e}_k\,.
\end{aligned}
$$

Orthonormality of the $\boldsymbol{e}_k$ results from the Gram matrix eigenvector orthonormality,

$$
\begin{aligned}
\boldsymbol{e}_j^\top \boldsymbol{e}_k &= \left(\frac{1}{\sqrt{\lambda_j}}\tilde{\boldsymbol{e}}_j^\top \boldsymbol{V}^\top\right)\left(\frac{1}{\sqrt{\lambda_k}}\boldsymbol{V}\tilde{\boldsymbol{e}}_k\right) && \text{(Definition of } \boldsymbol{e}_j, \boldsymbol{e}_k\text{)}\\
&= \frac{1}{\sqrt{\lambda_j\lambda_k}}\tilde{\boldsymbol{e}}_j^\top \tilde{\boldsymbol{G}}\tilde{\boldsymbol{e}}_k && \text{(Gram matrix)}\\
&= \frac{\lambda_k}{\sqrt{\lambda_j\lambda_k}}\tilde{\boldsymbol{e}}_j^\top \tilde{\boldsymbol{e}}_k && \text{(Eigenvector property of } \tilde{\boldsymbol{e}}_k\text{)}\\
&= \delta_{j,k}\,. && \text{(Orthonormality)}
\end{aligned}
$$

D.2 Experimental Details

This section uses the notation from Section 7.3 (see Table D.1).

Table D.1: Notation for curvature approximations. The notation is introduced in Section 7.3. This table recapitulates the abbreviations (referring to the approximations introduced in Section 7.2.4) and provides corresponding explanations.

Abbreviation	Explanation
mb, exact	Exact GGN with all mini-batch samples. Backpropagates NC vectors.
mb, mc	MC-approximated GGN with all mini-batch samples. Backpropagates NM vectors with M the number of MC-samples.
sub, exact	Exact GGN on a subset of mini-batch samples ($\lfloor N/8 \rfloor$ as in [183]). Backpropagates $\lfloor N/8 \rfloor C$ vectors.
sub, mc	MC-approximated GGN on a subset of mini-batch samples. Backpropagates $\lfloor N/8 \rfloor M$ vectors with M the number of MC-samples.

GGN Spectra (Figure 7.1a)

To obtain the spectra of Figure 7.1a we initialize the respective architecture, then draw a mini-batch and evaluate the GGN eigenvalues under the described approximations, clipping the Gram matrix eigenvalues at 10^{-4}. Figure D.1 provides the spectra for all used architectures with both the full GGN and a per-layer block-diagonal approximation.

D.2.1 Performance Evaluation

Hardware Details

Results were generated on a workstation with an Intel Core i7-8700K CPU (32 GB) and one NVIDIA GeForce RTX 2080 Ti GPU (11 GB).

Note

ViViT's quantities are implemented through BackPACK, which is triggered by PyTorch's gradient computation. Consequently, they can only be computed together with PyTorch's mini-batch gradient.

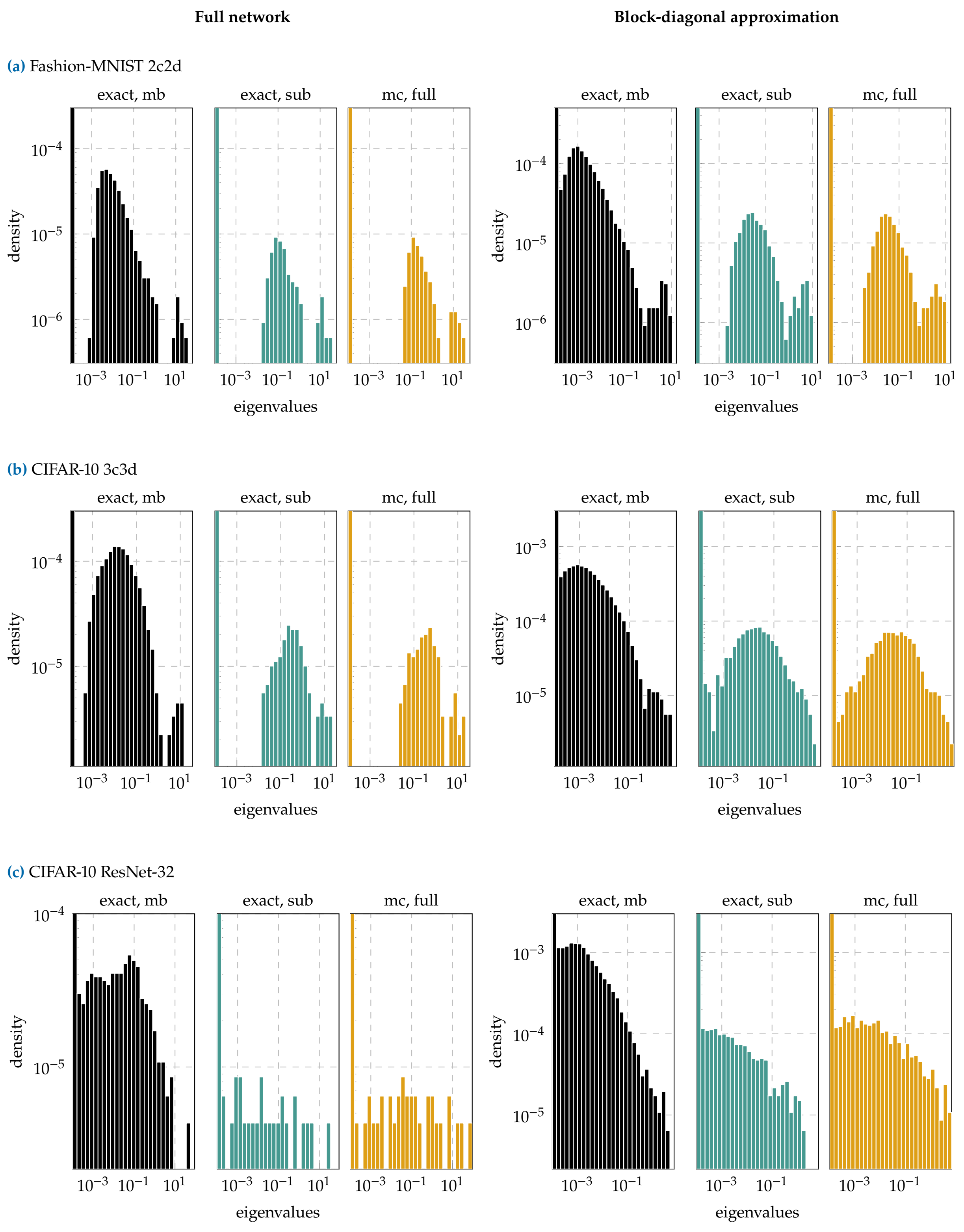


Figure D.1: GGN spectra of different architectures under ViViT's approximations: Left and right columns contain results with the full network's GGN and a per-layer block-diagonal approximation, respectively.

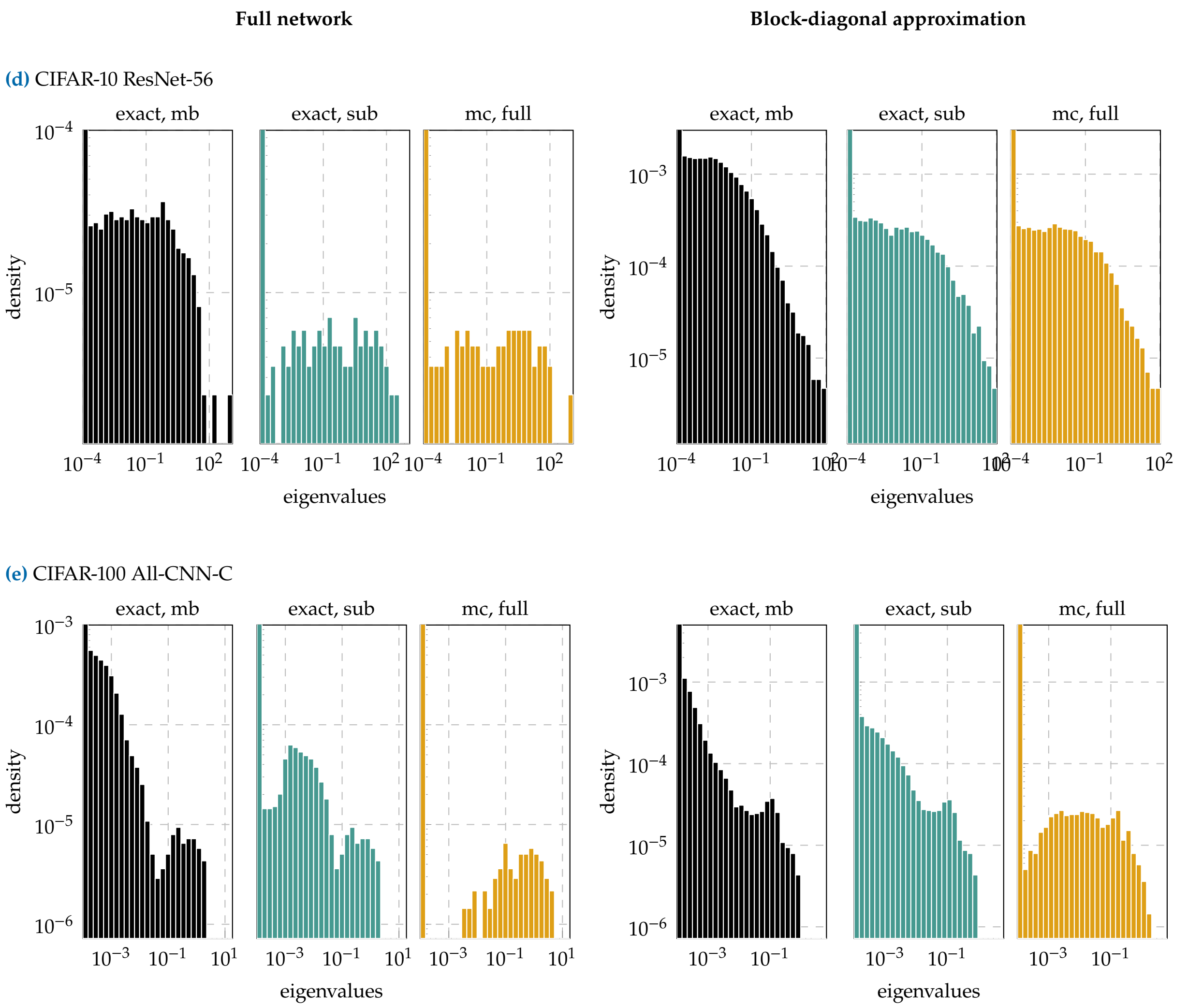


Figure D.1: GGN spectra of different architectures under ViViT's approximations: Left and right columns contain results with the full network's GGN and a per-layer block-diagonal approximation, respectively.

Architectures

We use untrained deep convolutional and residual networks from DeepOBS [146] and [74]. If a net has batch normalization layers, we set them to evaluation mode. Otherwise, the loss would not obey the sum structure of Equation (7.1). The batch normalization layers' internal moving averages, required for evaluation mode, are initialized by performing five forward passes with the current mini-batch in training mode before.

In experiments with fixed mini-batches the batch sizes correspond to DeepOBS' default value for training where possible (CIFAR-10: $N = 128$, Fashion-MNIST: $N = 128$). The ResNets use a batch size of $N = 128$. On CIFAR-100 (trained with $N = 256$), we reduce the batch size to $N = 64$ to fit the exact computation on the full mini-batch, used as baseline, into memory. If the GGN approximation is evaluated on a subset of the mini-batch (**sub**), $\lfloor N/8 \rfloor$ of the samples are used (as in [183]). The MC approximation is always evaluated with a single sample ($M = 1$).

Memory Performance (Critical Batch Sizes)

Two tasks are considered (see Section 7.3.1):

1. **Computing eigenvalues:** Compute the nontrivial eigenvalues $\{\lambda_k \,|\, (\lambda_k, \tilde{e}_k) \in \tilde{\mathbb{S}}_+\}$.
2. **Computing the top eigenpair:** Compute the top eigenpair $(\lambda_1, \boldsymbol{e}_1)$.

We repeat the tasks above and vary the mini-batch size until the device runs out of memory. The largest mini-batch size that can be handled by our system is denoted as N_{crit}, the critical batch size. We determine this number by bisection on the interval $[1; 32768]$.

Subfigures (a) and (b) of Figures D.2 to D.11 present the results. As described in Section 7.2.3, computing eigenvalues is more memory-efficient than computing eigenvectors and exhibits larger critical batch sizes. In line with the description in Section 7.2.4, a block-diagonal approximation is usually more memory-efficient and results in a larger critical batch size. Curvature sub-sampling and MC approximation further increase the applicable batch sizes.

In summary, there always exists a combination of approximations which allows for critical batch sizes larger than the traditional size used for training (some architectures even permit exact computation). Different accuracy-cost trade-offs may be preferred, depending on the application and the computational budget. By the presented approximations, ViViT's representation is capable to adapt over a wide range.

Run Time Performance

Here, we consider the task of computing the k leading eigenvectors and eigenvalues of a matrix. ViViT's eigenpair computation is compared with a power iteration that computes eigenpairs iteratively via matrix-vector products. The power iteration baseline is based on the PyHessian library [177] and uses the same termination criterion (at most 100 matrix-vector products per eigenvalue; stop if the eigenvalue estimate's relative change is less than 10^{-3}). In contrast to PyHessian, we use a different data format and stack the computed eigenvectors. This reduces the number of `for`-loops in the orthonormalization step. We repeat each run time measurement 20 times and report the shortest execution time as result.

Subfigures (c) and (d) of Figures D.2 to D.11 show the results. For most architectures, our exact method outperforms the power iteration for $k > 1$ and increases only marginally in run time as the number of requested eigenvectors grows. The proposed approximations share this property, and further reduce run time.

Note On CIFAR-100 (Large C)

For datasets with a large number of classes, like CIFAR-100 ($C = 100$), computations with the exact GGN are costly. In particular, constructing the Gram matrix $\tilde{G}$ has quadratic memory cost in C, and its eigendecomposition has cubic cost in time with C (see Section 7.2.3).

Full network **Block-diagonal approximation**

(a) Memory performance

	N_{crit} (eigenvalues)	
GGN^{Data}	mb	sub
exact	1753	8235
mc	7585	12434

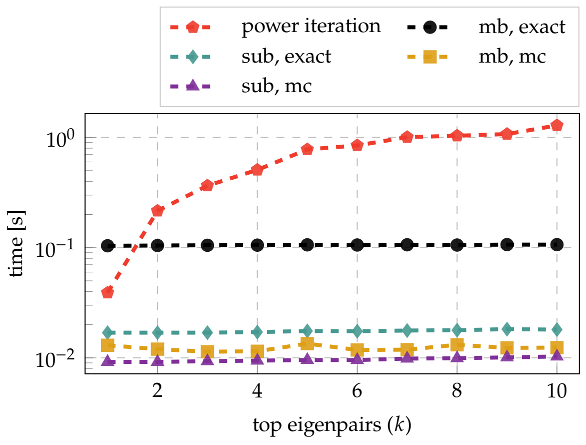
	N_{crit} (top eigenpair)	
GGN^{Data}	mb	sub
exact	872	6439
mc	6708	12162

(b) Memory performance

	N_{crit} (eigenvalues)	
GGN^{Data}	mb	sub
exact	1012	8317
mc	8448	12455

	N_{crit} (top eigenpair)	
GGN^{Data}	mb	sub
exact	993	7479
mc	8336	12413

(c) Run time performance



(d) Run time performance

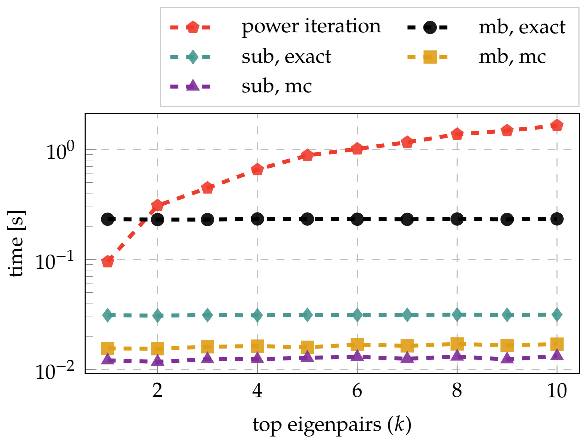


Figure D.2: GPU memory and run time performance for the 2c2d architecture on Fashion-MNIST. Left and right columns show results with the full network's GGN ($D = 3{,}274{,}634$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(a)**,**(b)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

As a result, the exact computation only works with batch sizes smaller than DeepOBS' default ($N = 256$ for CIFAR-100, see subfigures (a) and (b) of Figures D.10 and D.11). For the GGN block-diagonal approximation, which fits into CPU memory for $N = 64$, the exact computation of top eigenpairs is slower than a power iteration and only becomes comparable if a large number of eigenpairs is requested, see Figure D.11d.

For such datasets, the approximations proposed in Section 7.2.4 are essential to reduce costs. The most effective approximation to eliminate the scaling with C is using an MC approximation. Figures D.10 and D.11 confirm that the approximate computations scale to batch sizes used for training and that computing eigenpairs takes less time than a power iteration.

Computing Damped Newton Steps

A Newton step $-(\boldsymbol{G} + \delta \boldsymbol{I})^{-1}\boldsymbol{g}$ with damping $\delta > 0$ can be decomposed into updates along the eigenvectors of the GGN $\boldsymbol{G}$,

$$-(\boldsymbol{G} + \delta \boldsymbol{I})^{-1}\boldsymbol{g} = \sum_{k=1}^{K} \frac{-\gamma_k}{\lambda_k + \delta} \boldsymbol{e}_k + \sum_{k=K+1}^{D} \frac{-\gamma_k}{\delta} \boldsymbol{e}_k \,. \tag{D.1}$$

It corresponds to a Newton update along nontrivial eigendirections that uses the first- and second-order directional derivatives described in Section 7.2.2 and a gradient descent step with learning rate $1/\delta$ along trivial directions (with $\lambda_k = 0$). In the following, we refer to the first summand of Equation (D.1) as Newton step. As described in Section 7.2.3,

Full network

(a) Memory performance

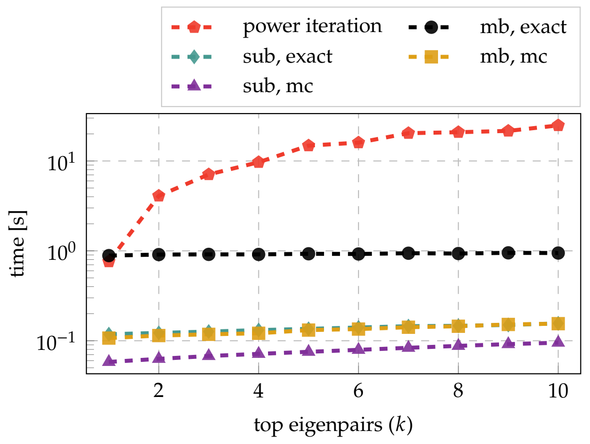
	N_{crit} (eigenvalues)			N_{crit} (top eigenpair)	
GGN \ Data	mb	sub	GGN \ Data	mb	sub
exact	4224	21164	exact	3276	21295
mc	24064	> 32768	mc	24064	> 32768

Block-diagonal approximation

(b) Memory performance

	N_{crit} (eigenvalues)			N_{crit} (top eigenpair)	
GGN \ Data	mb	sub	GGN \ Data	mb	sub
exact	4416	21453	exact	3276	20378
mc	23723	> 32768	mc	23457	> 32768

(c) Run time performance



(d) Run time performance

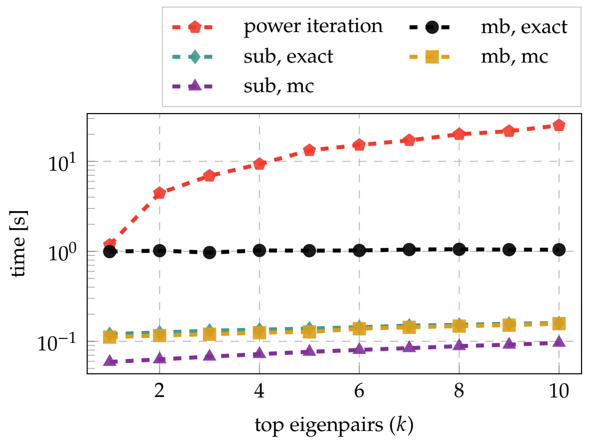


Figure D.3: CPU memory and run time performance for the 2c2d architecture on Fashion-MNIST. Left and right columns show results with the full network's GGN ($D = 3{,}274{,}634$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(a)**,**(b)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

we can perform the weighted sum in the Gram matrix space, rather than the parameter space, by computing

$$\sum_{k=1}^{K} \frac{-\gamma_k}{\lambda_k + \delta} \boldsymbol{e}_k = \sum_{k=1}^{K} \frac{-\gamma_k}{\lambda_k + \delta} \frac{1}{\sqrt{\lambda_k}} \boldsymbol{V} \tilde{\boldsymbol{e}}_k = \boldsymbol{V} \left(\sum_{k=1}^{K} \frac{-\gamma_k}{(\lambda_k + \delta)\sqrt{\lambda_k}} \tilde{\boldsymbol{e}}_k \right) .$$

This way, only a single vector needs to be transformed from Gram space into parameter space.

Table D.2 shows critical batch sizes for the Newton step computation (first term on the right side of Equation (D.1)), using Gram matrix eigenvalues larger than 10^{-4} and constant damping $\delta = 1$. Second-order directional derivatives λ_k are evaluated on the same samples as the GGN eigenvectors, but we *always* use all mini-batch samples to compute the directional gradients γ_n. Using our approximations, the Newton step computation scales to batch sizes beyond the traditional sizes used for training.

Full network

Block-diagonal approximation

(a) Memory performance

N_{crit} (eigenvalues)		
GGN \ Data	mb	sub
exact	909	4375
mc	3840	6626

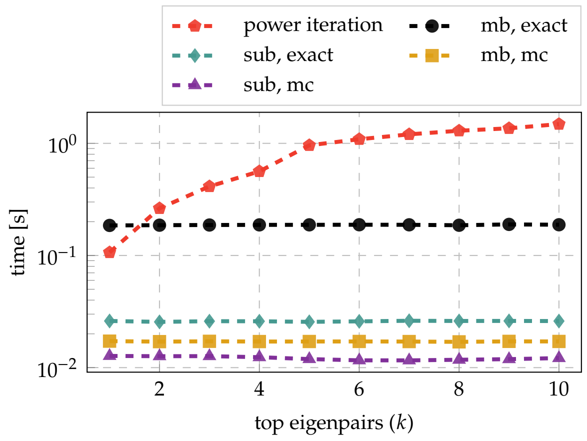
N_{crit} (top eigenpair)		
GGN \ Data	mb	sub
exact	677	3184
mc	3060	6029

(b) Memory performance

N_{crit} (eigenvalues)		
GGN \ Data	mb	sub
exact	1379	4464
mc	3854	6626

N_{crit} (top eigenpair)		
GGN \ Data	mb	sub
exact	1085	4415
mc	3854	6624

(c) Run time performance



(d) Run time performance

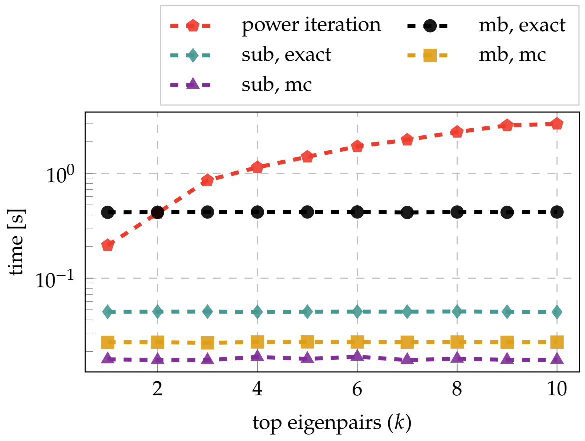


Figure D.4: GPU memory and run time performance for the 3c3d architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($D = 895{,}210$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(a)**,**(b)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full network

Block-diagonal approximation

(a) Memory performance

N_{crit} (eigenvalues)		
GGN \ Data	mb	sub
exact	2688	12768
mc	14330	20828

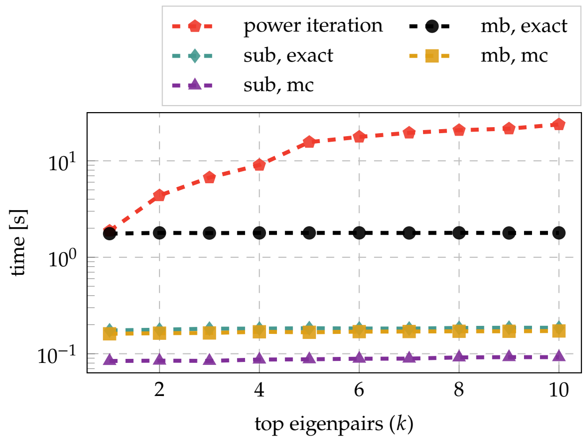
N_{crit} (top eigenpair)		
GGN \ Data	mb	sub
exact	2479	11138
mc	11499	19703

(b) Memory performance

N_{crit} (eigenvalues)		
GGN \ Data	mb	sub
exact	2978	13312
mc	14848	20732

N_{crit} (top eigenpair)		
GGN \ Data	mb	sub
exact	2911	13186
mc	14656	20757

(c) Run time performance



(d) Run time performance

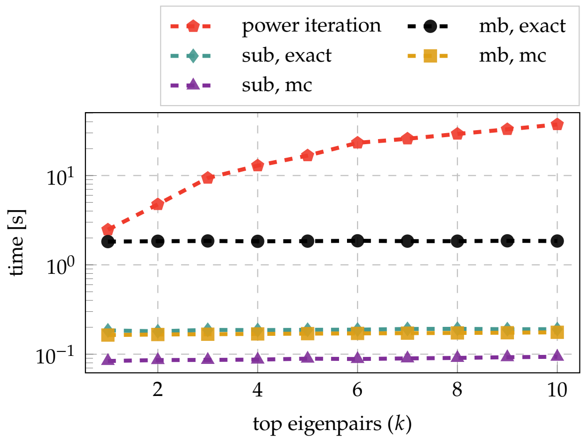


Figure D.5: CPU memory and run time performance for the 3c3d architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($D = 895{,}210$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(a)**,**(b)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full network

(a) Memory performance

GGN^{Data}	N_{crit} (eigenvalues) mb	N_{crit} (eigenvalues) sub
exact	1054	2052
mc	2064	2271

GGN^{Data}	N_{crit} (top eigenpair) mb	N_{crit} (top eigenpair) sub
exact	364	1360
mc	1536	2176

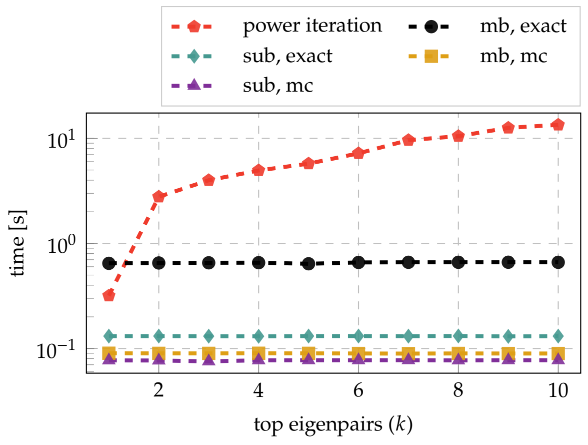
Block-diagonal approximation

(b) Memory performance

GGN^{Data}	N_{crit} (eigenvalues) mb	N_{crit} (eigenvalues) sub
exact	1061	2048
mc	2064	2271

GGN^{Data}	N_{crit} (top eigenpair) mb	N_{crit} (top eigenpair) sub
exact	1040	2048
mc	2064	2271

(c) Run time performance



(d) Run time performance

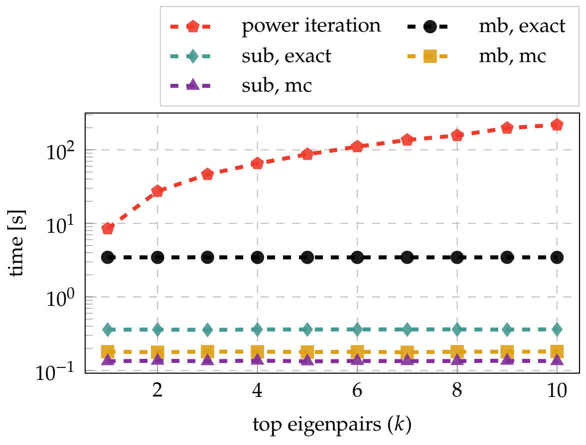
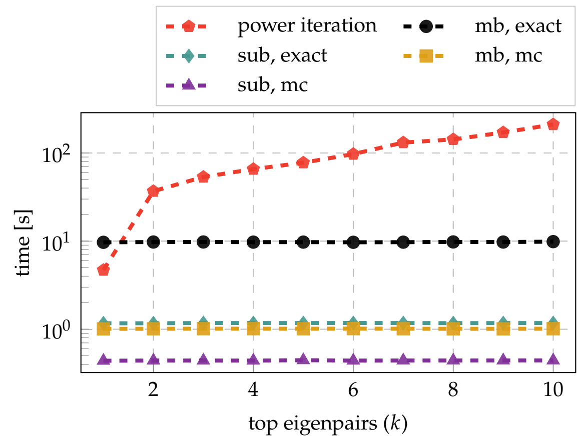


Figure D.6: GPU memory and run time performance for the ResNet-32 architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($D = 464{,}154$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(a)**,**(b)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full network

(c) Run time performance



Block-diagonal approximation

(d) Run time performance

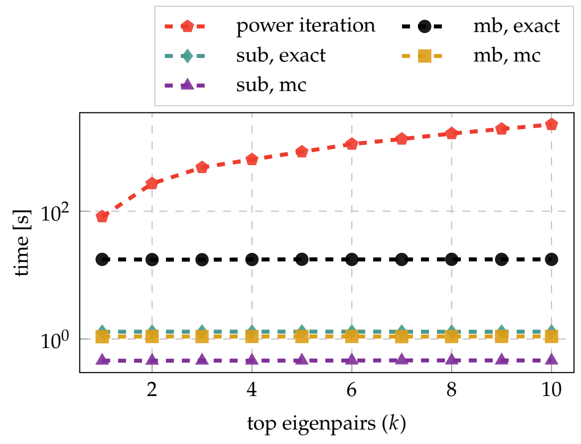


Figure D.7: CPU memory and run time performance for the ResNet-32 architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($D = 464{,}154$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full network

(a) Memory performance

Data / GGN	mb	sub
exact	837	1247
mc	1262	1312

N_{crit} (eigenvalues)

Data / GGN	mb	sub
exact	217	765
mc	896	1240

N_{crit} (top eigenpair)

Block-diagonal approximation

(b) Memory performance

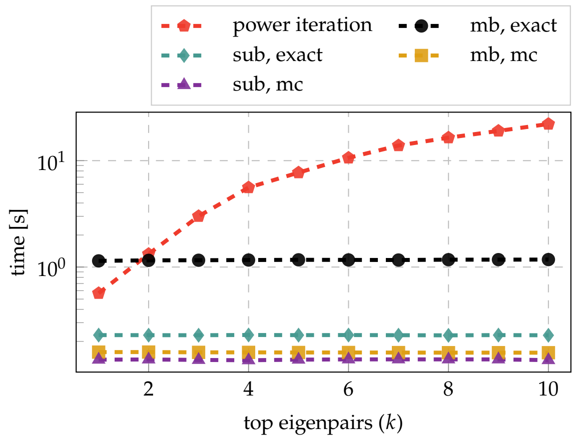
Data / GGN	mb	sub
exact	688	1247
mc	1232	1259

N_{crit} (eigenvalues)

Data / GGN	mb	sub
exact	383	1247
mc	1232	1255

N_{crit} (top eigenpair)

(c) Run time performance



(d) Run time performance

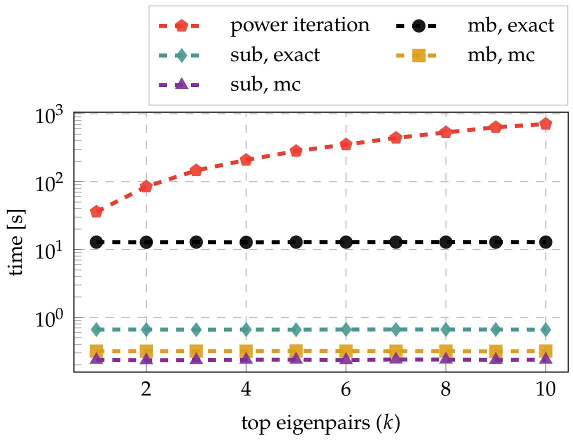
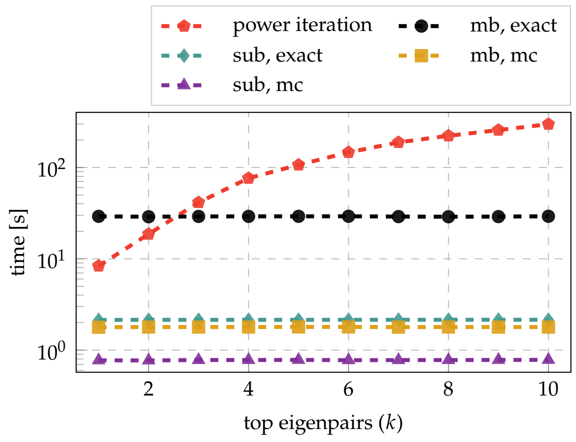


Figure D.8: GPU memory and run time performance for the ResNet-56 architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($D = 853{,}018$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full network

(c) Run time performance



Block-diagonal approximation

(d) Run time performance

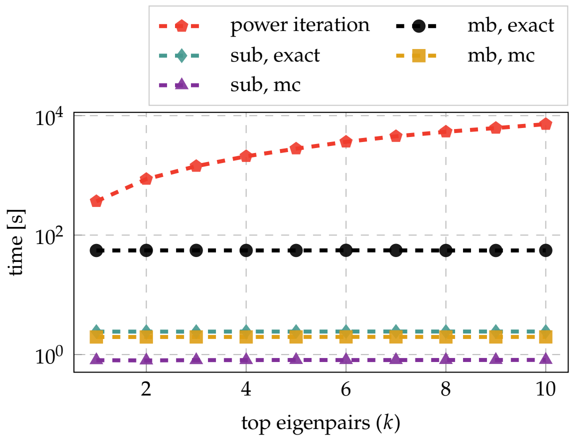


Figure D.9: CPU memory and run time performance for the ResNet-56 architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($D = 853{,}018$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (a, b) **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full network

(a) Memory performance

N_{crit} (eigenvalues)		
GGN^{Data}	mb	sub
exact	35	255
mc	1119	1536

N_{crit} (top eigenpair)		
GGN^{Data}	mb	sub
exact	14	111
mc	745	1402

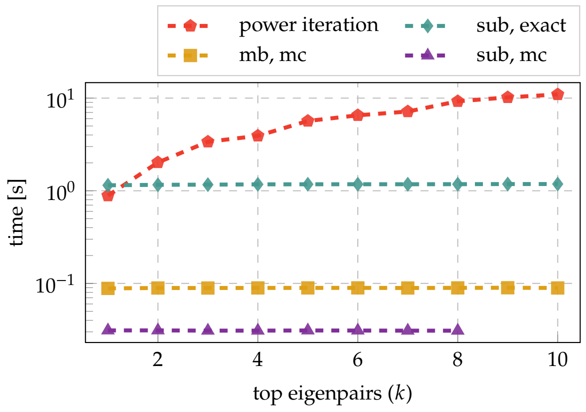
Block-diagonal approximation

(b) Memory performance

N_{crit} (eigenvalues)		
GGN^{Data}	mb	sub
exact	36	256
mc	1119	1536

N_{crit} (top eigenpair)		
GGN^{Data}	mb	sub
exact	36	255
mc	1119	1536

(c) Run time performance



(d) Run time performance

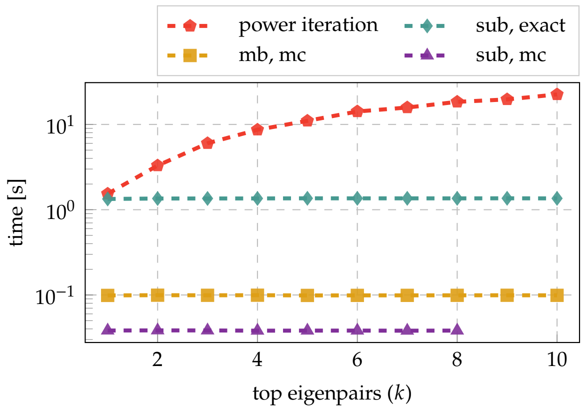


Figure D.10: GPU memory and run time performance for the All-CNN-C architecture on CIFAR-100. Left and right columns show results with the full network's GGN ($D = 1{,}387{,}108, C = 100$) and a per-layer block-diagonal approximation, respectively. **(a)**,**(b)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 64$.

Full network

(a) Memory performance

N_{crit} (eigenvalues)		
GGN^{Data}	mb	sub
exact	96	695
mc	3771	4941

N_{crit} (top eigenpair)		
GGN^{Data}	mb	sub
exact	45	351
mc	2558	4454

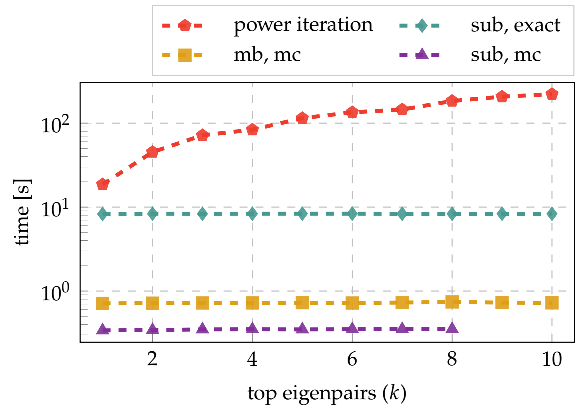
Block-diagonal approximation

(b) Memory performance

N_{crit} (eigenvalues)		
GGN^{Data}	mb	sub
exact	97	703
mc	3783	4929

N_{crit} (top eigenpair)		
GGN^{Data}	mb	sub
exact	97	703
mc	3759	4918

(c) Run time performance



(d) Run time performance

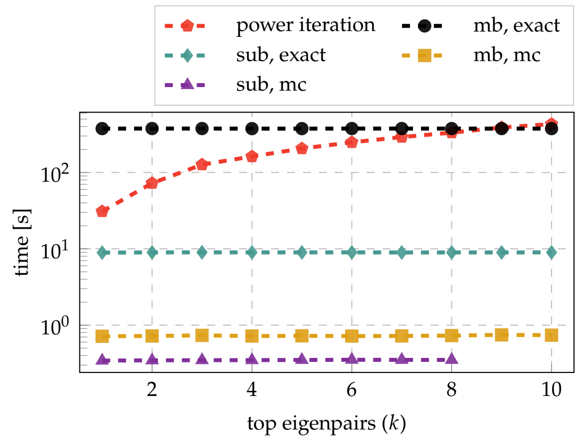


Figure D.11: CPU memory and run time performance for the All-CNN-C architecture on CIFAR-100. Left and right columns show results with the full network's GGN ($D = 1{,}387{,}108, C = 100$) and a per-layer block-diagonal approximation, respectively. **(a)**,**(b)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(c)**,**(d)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 64$.

Table D.2: Memory performance for computing damped Newton steps: Left and right columns show the critical batch sizes with the full network's GGN and a per-layer block-diagonal approximation, respectively.

Fashion-MNIST 2c2d

Full network

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	66	159
mc	362	528

N_{crit} (CPU)		
GGN^{Data}	mb	sub
exact	202	487
mc	1107	1639

Block-diagonal approximation

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	68	159
mc	368	528

N_{crit} (CPU)		
GGN^{Data}	mb	sub
exact	210	487
mc	1137	1643

CIFAR-10 3c3d

Full network

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	208	727
mc	1055	1816

N_{crit} (CPU)		
GGN^{Data}	mb	sub
exact	667	2215
mc	3473	5632

Block-diagonal approximation

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	349	795
mc	1659	2112

N_{crit} (CPU)		
GGN^{Data}	mb	sub
exact	1046	2423
mc	4997	6838

CIFAR-10 ResNet-32

Full network

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	344	1119
mc	1205	1535

Block-diagonal approximation

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	1051	1851
mc	2048	2208

CIFAR-10 ResNet-56

Full network

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	209	640
mc	687	890

Block-diagonal approximation

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	767	1165
mc	1232	1255

CIFAR-100 All-CNN-C

Full network

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	13	87
mc	640	959

N_{crit} (CPU)		
GGN^{Data}	mb	sub
exact	43	309
mc	2015	2865

Block-diagonal approximation

N_{crit} (GPU)		
GGN^{Data}	mb	sub
exact	35	135
mc	1079	1536

N_{crit} (CPU)		
GGN^{Data}	mb	sub
exact	95	504
mc	3360	3920

D.2.2 Training of Neural Networks

Procedure

We train the following DeepOBS [146] architectures with SGD and Adam: 3c3d on CIFAR-10, 2c2d on Fashion-MNIST and All-CNN-C on CIFAR-100; all are equipped with cross-entropy loss. To ensure successful training, we use the hyperparameters from [38] (see Table D.3).

We also train a residual network ResNet-32 [68] with cross-entropy loss on CIFAR-10 with both SGD and Adam. For this, we use a batch size of 128 and train for 180 epochs. Momentum for SGD was fixed to 0.9, and Adam uses the default parameters ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$). For both optimizers, the learning rate was determined via grid search. Following [146], we use a log-equidistant grid from 10^{-5} to 10^{2} and 36 grid points. As performance metric, the best test accuracy during training (evaluated once every epoch) is used.

Results

The results for the hyperparameter grid search are reported in Table D.3. The training metrics training/test loss/accuracy for all eight test problems are shown in Figures D.12 and D.13.

Problem	SGD	Adam	Batch size	Epochs
Fashion-MNIST 2c2d	$\eta \approx 2.07 \cdot 10^{-2}$	$\eta \approx 1.27 \cdot 10^{-4}$	$N = 128$	100
CIFAR-10 3c3d	$\eta \approx 3.79 \cdot 10^{-3}$	$\eta \approx 2.98 \cdot 10^{-4}$	$N = 128$	100
CIFAR-10 ResNet-32	$\eta \approx 6.31 \cdot 10^{-2}$	$\eta \approx 2.51 \cdot 10^{-3}$	$N = 128$	180
CIFAR-100 All-CNN-C	$\eta \approx 4.83 \cdot 10^{-1}$	$\eta \approx 6.95 \cdot 10^{-4}$	$N = 256$	350

Table D.3: **Hyperparameter settings for training runs.** For both SGD and Adam, we report their learning rates η (taken from the baselines in [38] or, for ResNet-32, determined via grid search). Momentum for SGD is fixed to 0.9. Adam uses the default parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. We also report the batch size used for training and the number of training epochs.

D.2.3 GGN versus Hessian

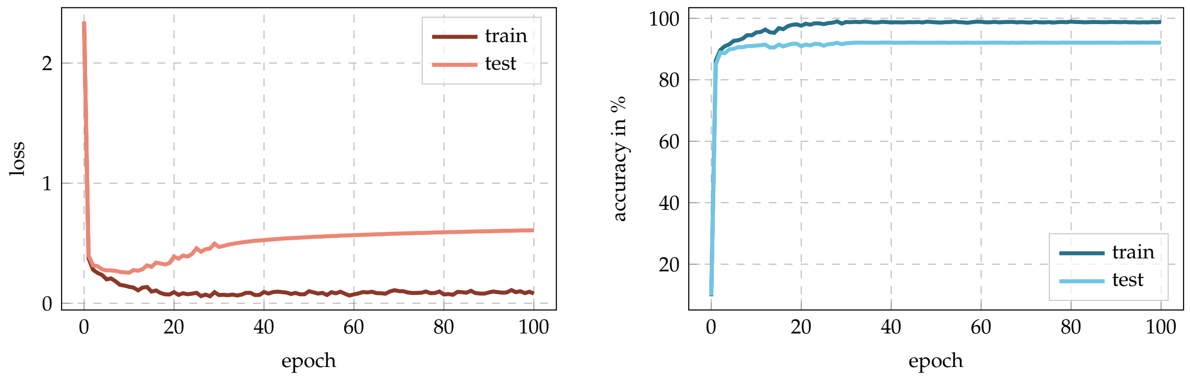
Checkpoints

During training of the neural networks (see Appendix D.2.2), we store a copy of the model (*i.e.* the network's current parameters) at specific checkpoints. This grid defines the temporal resolution for all subsequent computations. Since training progresses much faster in the early training stages, we use a log-grid with 100 grid points between 1 and the number of training epochs and shift this grid by -1.

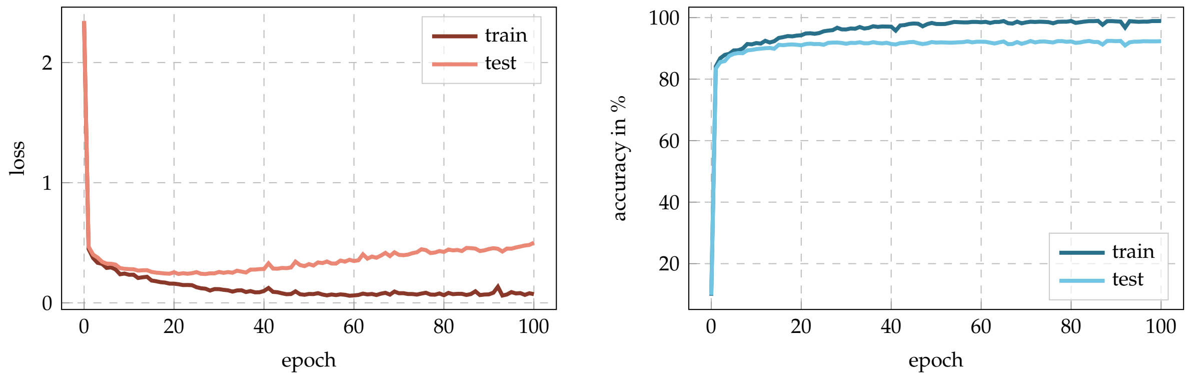
Overlap

Recall from Section 7.3.2: for the set of orthonormal eigenvectors $\{e_c^{\boldsymbol{U}}\}_{c=1}^{C}$ to the C largest eigenvalues of some symmetric matrix $\boldsymbol{U}$, let $\boldsymbol{P}^{\boldsymbol{U}} = (e_1^{\boldsymbol{U}}, \dots, e_C^{\boldsymbol{U}})(e_1^{\boldsymbol{U}}, \dots, e_C^{\boldsymbol{U}})^{\top}$. As in [65], the overlap between two subspaces

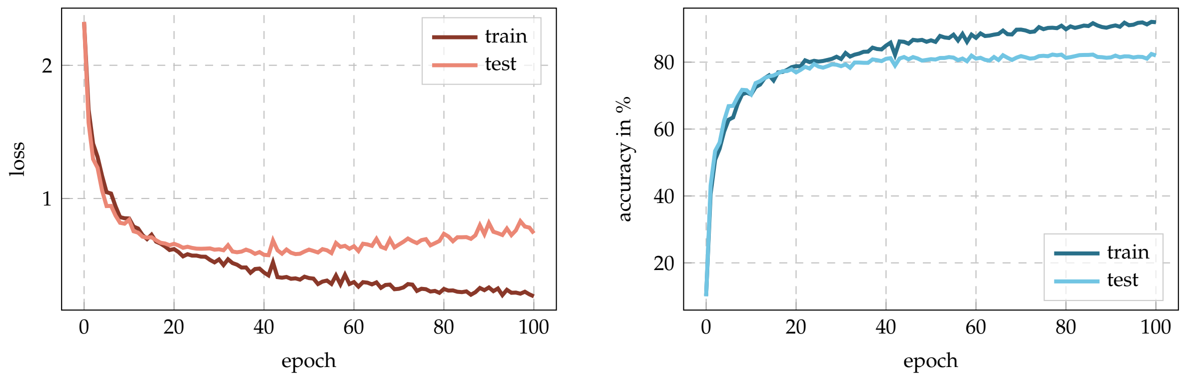
(a) Fashion-MNIST 2c2d SGD



(b) Fashion-MNIST 2c2d Adam



(c) CIFAR-10 3c3d SGD



(d) CIFAR-10 3c3d Adam

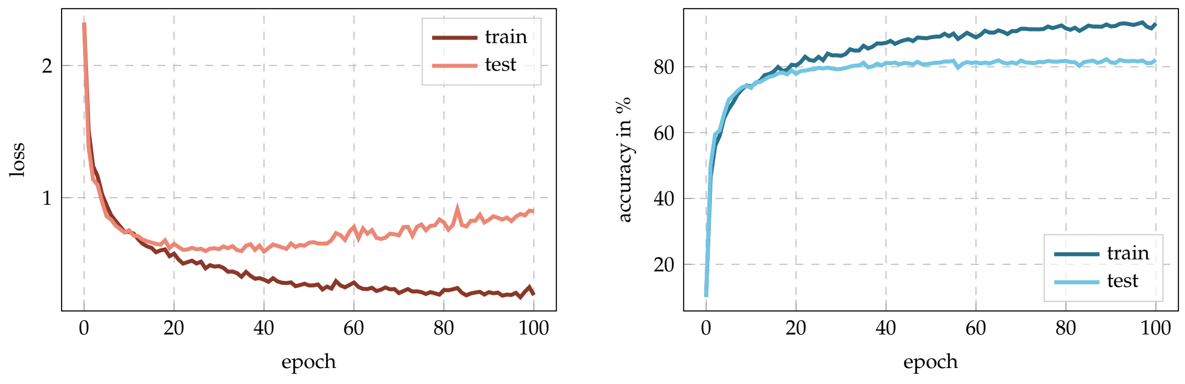
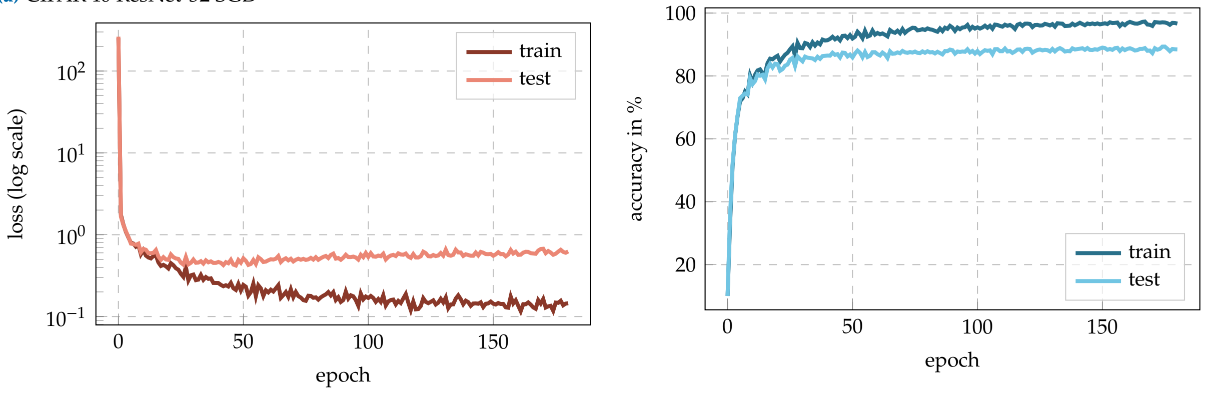
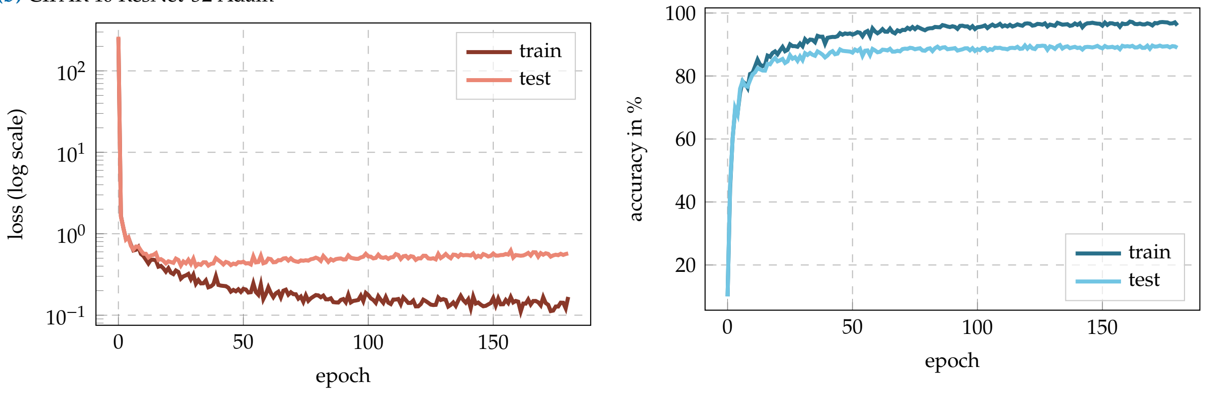


Figure D.12: Training metrics (1). Training/test loss/accuracy for all test problems.

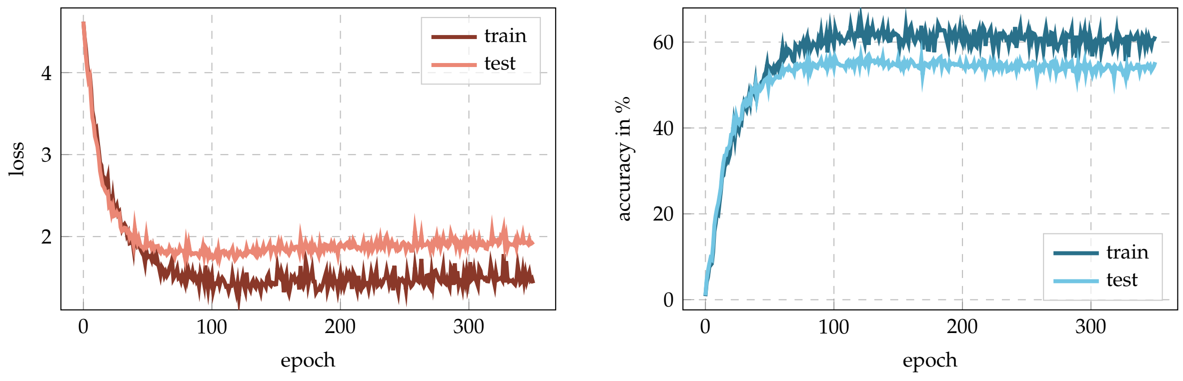
(a) CIFAR-10 ResNet-32 SGD



(b) CIFAR-10 ResNet-32 Adam



(c) CIFAR-100 All-CNN-C SGD



(d) CIFAR-100 All-CNN-C Adam

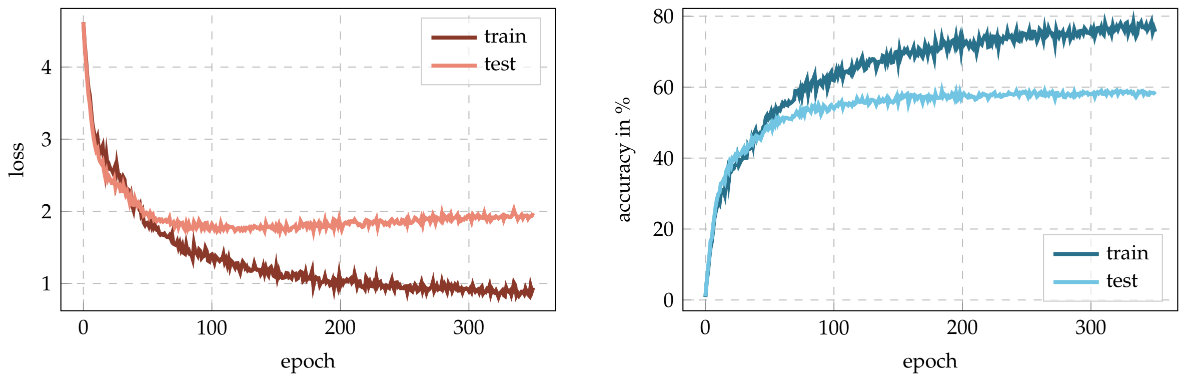


Figure D.13: **Training metrics (2).** Training/test loss/accuracy for all test problems.

$\mathcal{E}^{\boldsymbol{U}} = \operatorname{span}(\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})$ and $\mathcal{E}^{\boldsymbol{V}} = \operatorname{span}(\boldsymbol{e}_1^{\boldsymbol{V}}, \dots, \boldsymbol{e}_C^{\boldsymbol{V}})$ of the matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ is defined by

$$\operatorname{overlap}(\mathcal{E}^{\boldsymbol{U}}, \mathcal{E}^{\boldsymbol{V}}) = \frac{\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{U}}\boldsymbol{P}^{\boldsymbol{V}})}{\sqrt{\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{U}})\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{V}})}} \in [0, 1] \,.$$

The overlap can be computed efficiently by using the trace's cyclic property: it holds $\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{U}}\boldsymbol{P}^{\boldsymbol{V}}) = \operatorname{Tr}(\boldsymbol{W}^\top\boldsymbol{W})$ with $\boldsymbol{W} = (\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})^\top(\boldsymbol{e}_1^{\boldsymbol{V}}, \dots, \boldsymbol{e}_C^{\boldsymbol{V}}) \in \mathbb{R}^{C\times C}$. Note that this is a small $C \times C$ matrix, whereas $\boldsymbol{P}^{\boldsymbol{U}}, \boldsymbol{P}^{\boldsymbol{V}} \in \mathbb{R}^{D\times D}$. Since

$$\begin{aligned}
\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{U}}) &= \operatorname{Tr}((\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})(\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})^\top) \\
&= \operatorname{Tr}((\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})^\top(\boldsymbol{e}_1^{\boldsymbol{U}}, \dots, \boldsymbol{e}_C^{\boldsymbol{U}})) && \text{(Cyclic property of trace)} \\
&= \operatorname{Tr}(\boldsymbol{I}_C) && \text{(Orthonormality of the eigenvectors)} \\
&= C
\end{aligned}$$

(and analogous $\operatorname{Tr}(\boldsymbol{P}^{\boldsymbol{V}}) = C$), the denominator simplifies to C.

Procedure

For each checkpoint, we compute the top-C eigenvalues and associated eigenvectors of the full-batch GGN and Hessian (*i.e.* GGN and Hessian are both evaluated on the entire training set) using an iterative matrix-free approach. We then compute the overlap between the top-C eigenspaces as described above. The eigspaces (*i.e.* the top-C eigenvalues and associated eigenvectors) are stored on disk such that they can be used as a reference by subsequent experiments.

Results

The results for all test problems are presented in Figure D.14. Except for a short phase at the beginning of the optimization procedure (note the log scale for the epoch-axis), a strong agreement (note the different limits for the overlap-axis) between the top-C eigenspaces is observed. We make similar observations for all test problems, yet to a slightly lesser extent for CIFAR-100. A possible explanation for this would be that the 100-dimensional eigenspaces differ in the eigenvectors associated with relatively small curvature. The corresponding eigenvalues already transition into the bulk of the spectrum, where the "sharpness of separation" decreases. However, since all directions are equally weighted in the overlap, overall slightly lower values are obtained.

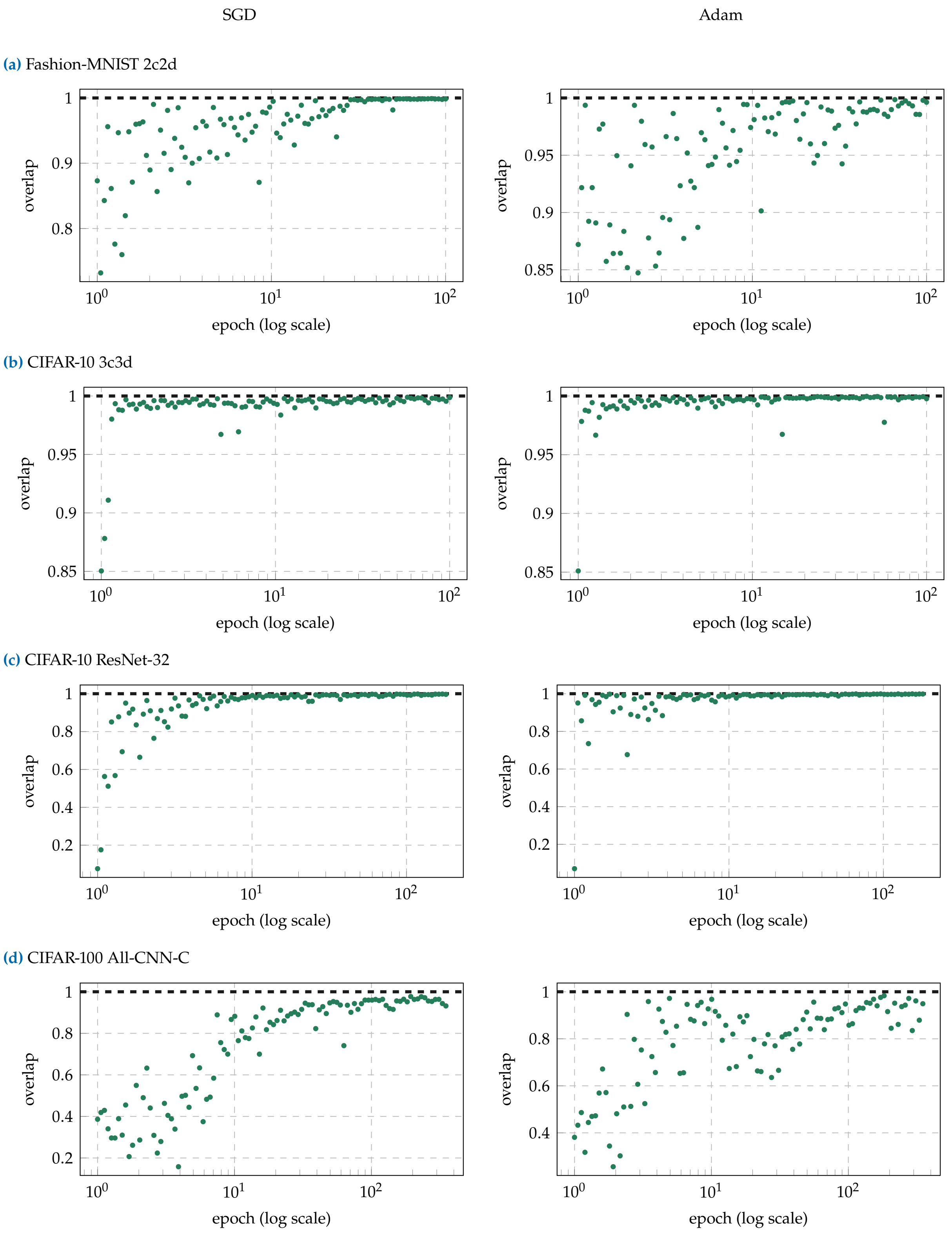


Figure D.14: Full-batch GGN versus full-batch Hessian. Overlap between the top-C eigenspaces of the full-batch GGN and full-batch Hessian during training for all test problems.

D.2.4 Eigenspace Under Noise & Approximations

Procedure (1)

We use the checkpoints and the definition of overlaps between eigenspaces from Appendix D.2.3. For the approximation of the GGN, we consider the cases listed in Table D.4.

Table D.4: Considered cases for approximation of the eigenspace: We use a different set of cases for the approximation of the GGN's full-batch eigenspace depending on the test problem. For the test problems with $C = 10$, we use $M = 1$ MC-sample, for the CIFAR-100 All-CNN-C test problem ($C = 100$), we use $M = 10$ MC-samples in order to reduce the computational costs by the same factor.

Problem	**Cases**
Fashion-MNIST 2c2d CIFAR-10 3c3d and CIFAR-10 ResNet-32	**mb, exact** with mini-batch sizes $N \in \{2, 8, 32, 128\}$ **mb, mc** with $N = 128$ and $M = 1$ MC-sample **sub, exact** using 16 samples from the mini-batch **sub, mc** using 16 samples from the mini-batch and $M = 1$ MC-sample
CIFAR-100 All-CNN-C	**mb, exact** with mini-batch sizes $N \in \{2, 8, 32, 128\}$ **mb, mc** with $N = 128$ and $M = 10$ MC-samples **sub, exact** using 16 samples from the mini-batch **sub, mc** using 16 samples from the mini-batch and $M = 10$ MC-samples

For every checkpoint and case, we compute the top-C eigenvectors of the respective approximation to the GGN. The eigenvectors are either computed directly using ViViT (by transforming eigenvectors of the Gram matrix into parameter space, see Section 7.2.1) or, if not applicable (because memory requirements exceed N_{crit}, see Section 7.3.1), using an iterative matrix-free approach. The overlap is computed in reference to the GGN's full-batch top-C eigenspace (see Appendix D.2.3). We extract 5 mini-batches from the training data and repeat the above procedure for each mini-batch (*i.e.* we obtain 5 overlap measurements for every checkpoint and case). The same 5 mini-batches are used over all checkpoints and cases.

Results (1)

The results can be found in Figure D.15 and D.16. All test problems show the same characteristics: with decreasing computational effort, the approximation carries less and less structure of its full-batch counterpart, as indicated by dropping overlaps. In addition, for a fixed approximation method, a decrease in approximation quality can be observed over the course of training.

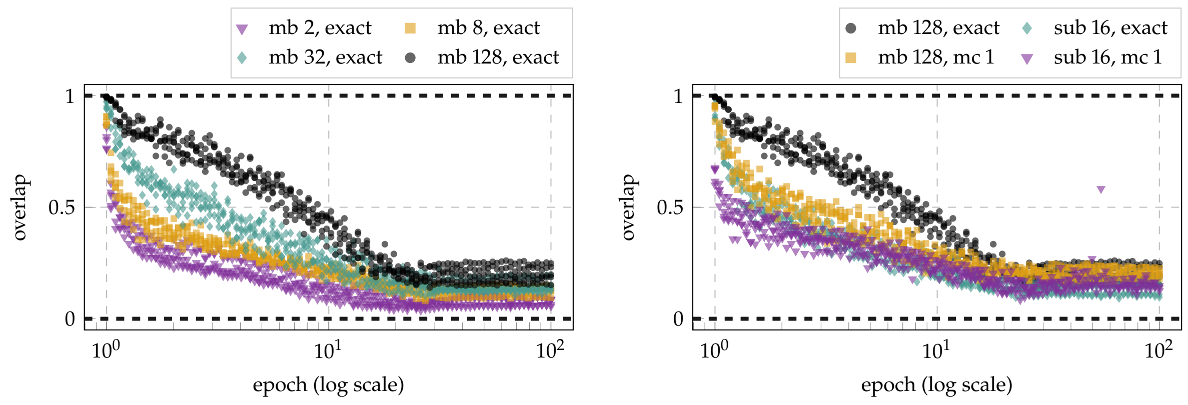
Procedure (2)

Since ViViT's GGN approximations using curvature sub-sampling and/or the MC approximation (the cases **mb, mc** as well as **sub, exact** and **sub, mc** in Table D.4) are based on the *mini*-batch GGN, we cannot expect them to perform better than this baseline. We thus repeat the analysis from above but use the mini-batch GGN with batch-size $N = 128$ as ground truth instead of the full-batch GGN. Of course, the mini-batch reference top-C eigenspace is always evaluated on the same mini-batch as the approximation.

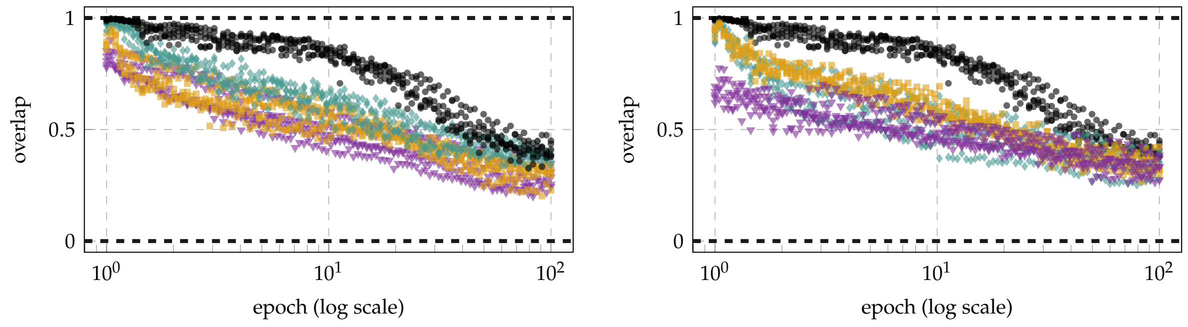
Impact of batch size

Impact of batch size & approximations

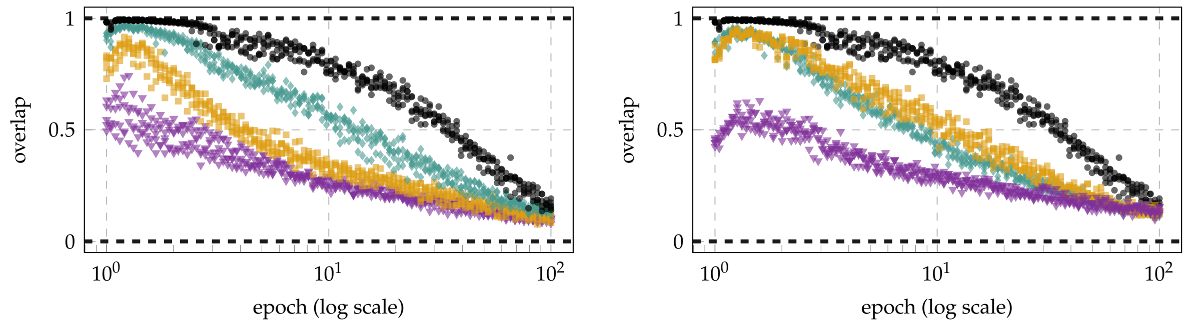
(a) Fashion-MNIST 2c2d SGD



(b) Fashion-MNIST 2c2d Adam



(c) CIFAR-10 3c3d SGD



(d) CIFAR-10 3c3d Adam

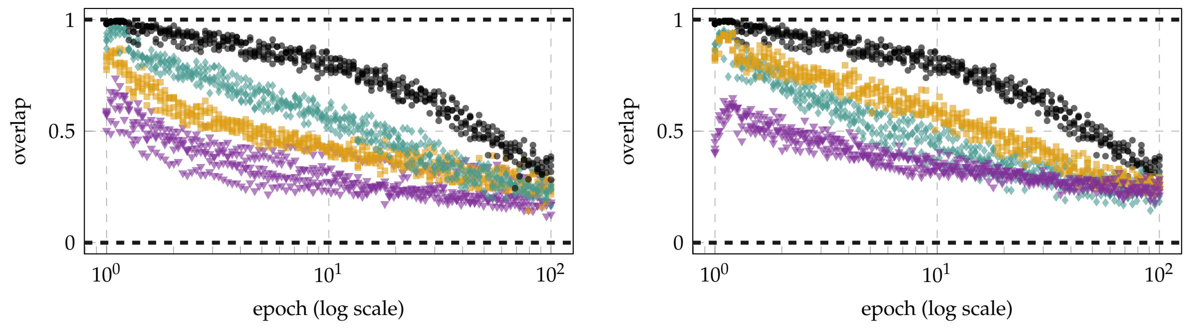


Figure D.15: ViViT versus full-batch GGN (1). Overlap between the top-C eigenspaces of different GGN approximations and the full-batch GGN during training for all test problems. Each approximation is evaluated on 5 different mini-batches.

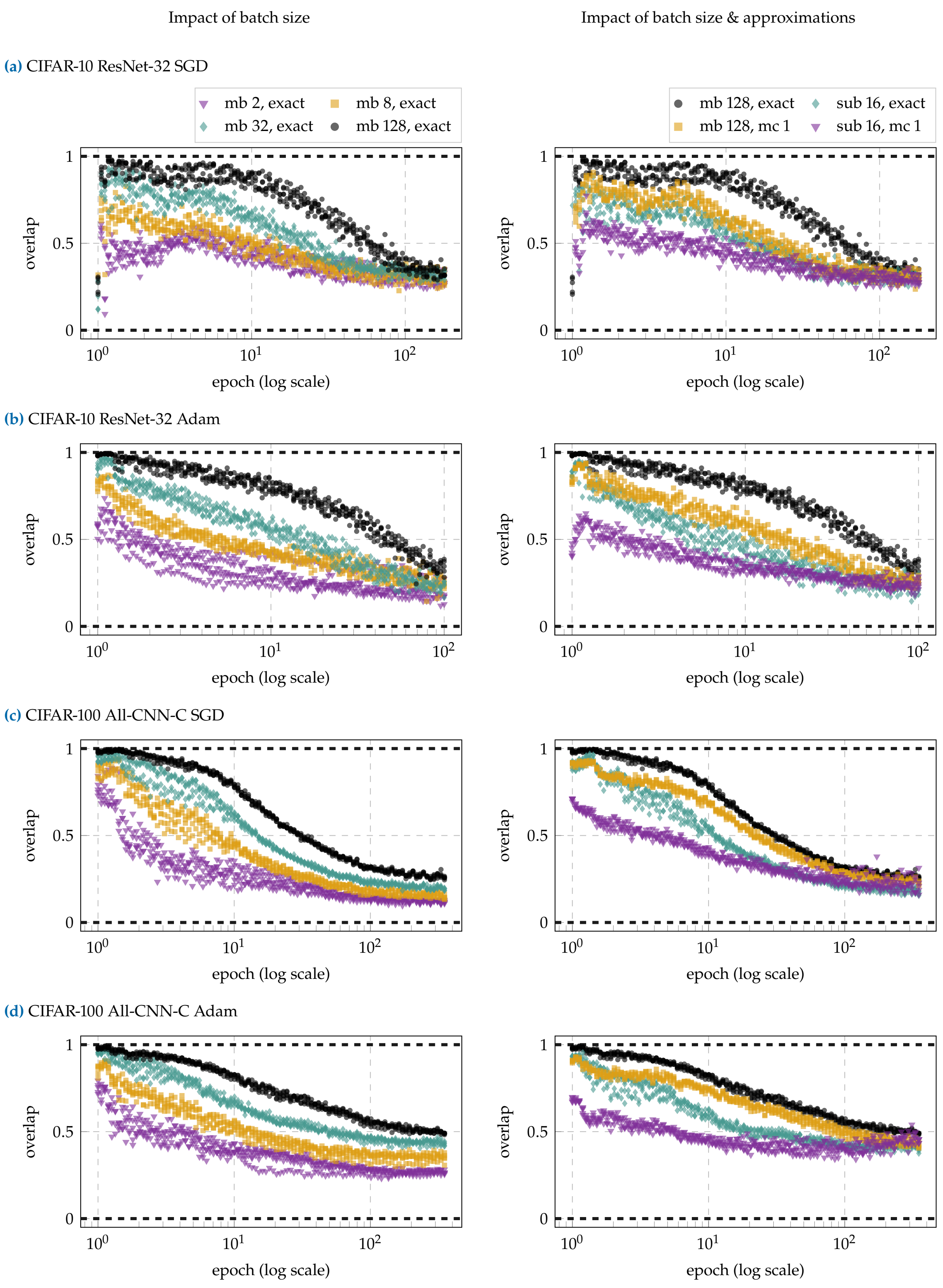


Figure D.16: ViViT versus full-batch GGN (2). Overlap between the top-C eigenspaces of different GGN approximations and the full-batch GGN during training for all test problems. Each approximation is evaluated on 5 different mini-batches.

Results (2)

Figure D.17 shows the results. Over large parts of the optimization (note the log scale for the epoch-axis), the MC approximation seems to be better suited than curvature sub-sampling (which has comparable computational cost). For the CIFAR-100 All-CNN-C problem, the MC approximation stands out particularly early from the other approximations and consistently yields higher overlaps with the mini-batch GGN.

D.2.5 Curvature Under Noise & Approximations

GGN and Hessian are predominantly used to locally approximate the loss by a quadratic model q (see Equation (7.6)). Even if the curvature's eigenspace is completely preserved in spite of the approximations, they can still alter the curvature *magnitude* along the eigenvectors.

Procedure

Table D.5 shows the cases considered in this experiment.

Table D.5: Considered cases for approximation of curvature: We use a different set of cases for the approximation of the GGN depending on the test problem. For the test problems with $C = 10$, we use $M = 1$ MC-sample, for the CIFAR-100 All-CNN-C test problem ($C = 100$), we use $M = 10$ MC-samples in order to reduce the computational costs by the same factor.

Problem	Cases
Fashion-MNIST 2c2d CIFAR-10 3c3d and CIFAR-10 ResNet-32	**mb, exact** with mini-batch size $N = 128$ **mb, mc** with $N = 128$ and $M = 1$ MC-sample **sub, exact** using 16 samples from the mini-batch **sub, mc** using 16 samples from the mini-batch and $M = 1$ MC-sample
CIFAR-100 All-CNN-C	**mb, exact** with mini-batch size $N = 128$ **mb, mc** with $N = 128$ and $M = 10$ MC-samples **sub, exact** using 16 samples from the mini-batch **sub, mc** using 16 samples from the mini-batch and $M = 10$ MC-samples

Due to the large computational effort for evaluating the full-batch directional derivatives, a subset of the checkpoints from Appendix D.2.3 is used for two problems: we use every second checkpoint for CIFAR-10 ResNet-32 and every forth checkpoint for CIFAR-100 All-CNN-C.

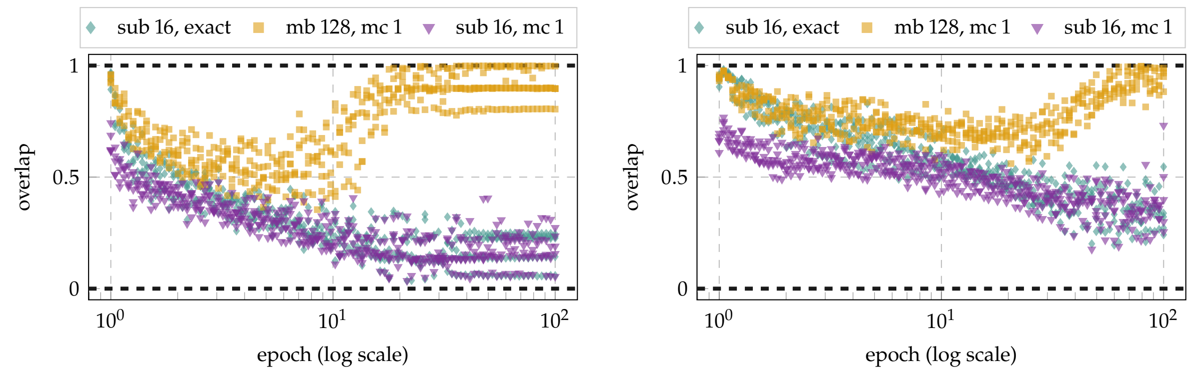
For each checkpoint and case, we compute the top-C eigenvectors $\{\boldsymbol{e}_k\}_{k=1}^C$ of the GGN approximation $\boldsymbol{G}^{(\text{ap})}$ either with ViViT or using an iterative matrix-free approach (as in Appendix D.2.4). The second-order directional derivative of the corresponding quadratic model along direction $\boldsymbol{e}_k$ is then given by $\lambda_k^{(\text{ap})} = \boldsymbol{e}_k^\top \boldsymbol{G}^{(\text{ap})} \boldsymbol{e}_k$ (see Equation (7.7)). As a reference, we compute the full-batch GGN $\boldsymbol{G}^{(\text{fb})}$ and the resulting directional derivatives along the same eigenvectors $\{\boldsymbol{e}_k\}_{k=1}^C$, *i.e.* $\lambda_k^{(\text{fb})} = \boldsymbol{e}_k^\top \boldsymbol{G}^{(\text{fb})} \boldsymbol{e}_k$. The average (over all C directions) relative error is given by

$$\epsilon = \frac{1}{C} \sum_{k=1}^{C} \frac{|\lambda_k^{(\text{ap})} - \lambda_k^{(\text{fb})}|}{\lambda_k^{(\text{fb})}} .$$

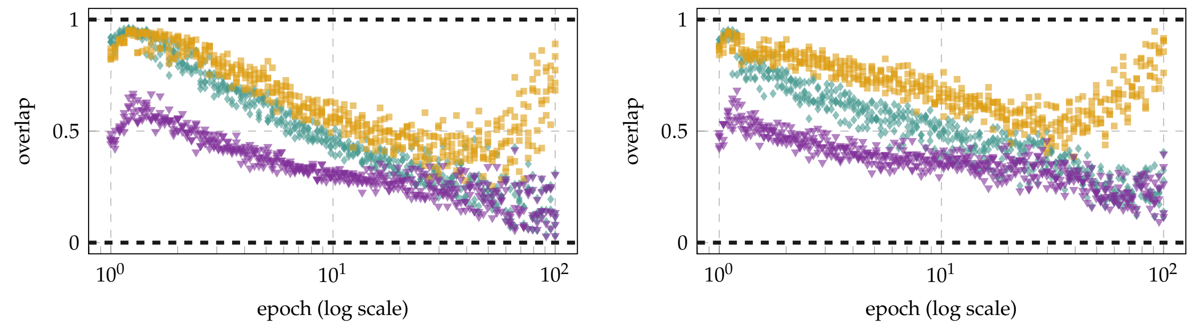
The procedure above is repeated on 3 mini-batches from the training data (*i.e.* we obtain 3 average relative errors for every checkpoint and case) –

SGD Adam

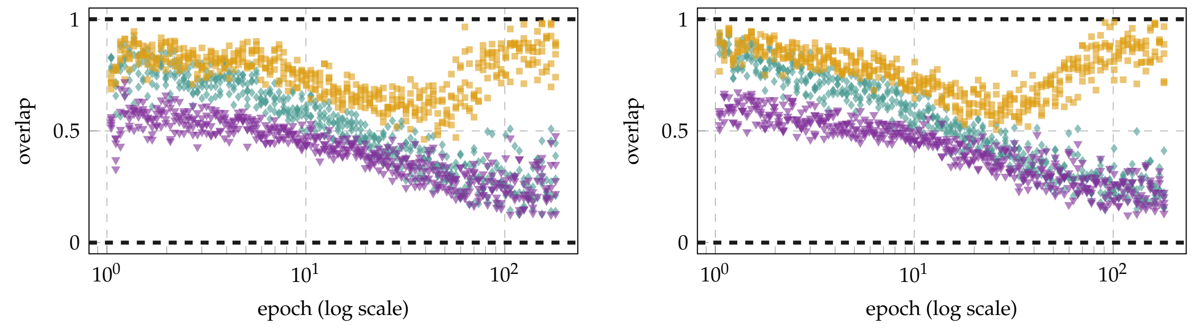
(a) Fashion-MNIST 2c2d



(b) CIFAR-10 3c3d



(c) CIFAR-10 ResNet-32



(d) CIFAR-100 All-CNN-C

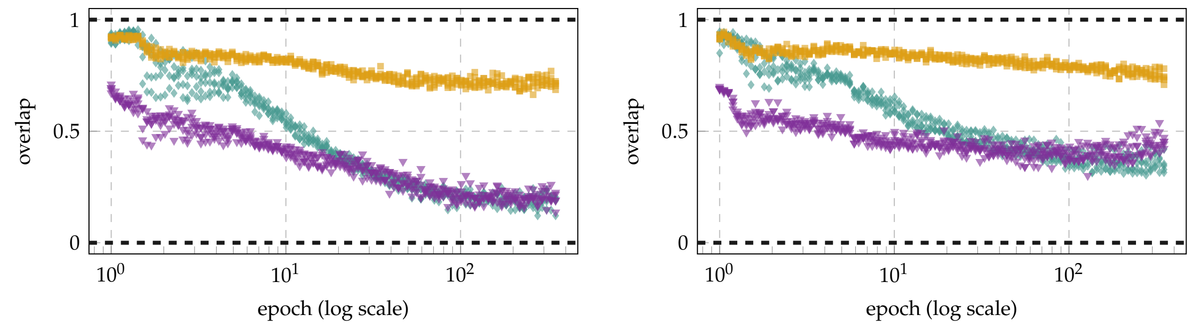


Figure D.17: ViViT versus mini-batch GGN. Overlap between the top-C eigenspaces of different GGN approximations and the mini-batch GGN during training for all test problems. Each approximation is evaluated on 5 different mini-batches.

except for the CIFAR-100 All-CNN-C test problem, where we perform only a single run to keep the computational effort manageable.

Results

Figure D.18 shows the results. We observe similar results as in Appendix D.2.4: with increasing computational effort, the approximated directional derivatives become more precise and the overall approximation quality decreases over the course of the optimization. For the ResNet-32 architecture, the average errors are particularly large.

D.2.6 Directional Derivatives

Procedure

We use the checkpoints from Appendix D.2.3. For every checkpoint, we compute the top-C eigenvectors of the mini-batch GGN ($N = 128$) using an iterative matrix-free method. We also compute the mini-batch gradient. The first- and second-order directional derivatives of the resulting quadratic model (Equation (7.6)) are given by Equation (7.8).

We use these directional derivatives $\{\gamma_{n,k}\}_{n=1,k=1}^{N,C}$, $\{\lambda_{n,k}\}_{n=1,k=1}^{N,C}$ to compute signal-to-noise ratios (SNRs) along the top-C eigenvectors. The curvature SNR along direction $\boldsymbol{e}_k$ is given by the squared sample mean divided by the empirical variance of the samples $\{\lambda_{n,k}\}_{n=1}^{N}$, *i.e.*

$$\mathrm{SNR} = \frac{\lambda_k^2}{{}^1\!/_{N-1}\sum_{n=1}^{N}(\lambda_{n,k} - \lambda_k)^2} \quad \text{where} \quad \lambda_k = \frac{1}{N}\sum_{n=1}^{N}\lambda_{n,k}\,.$$

(and similarly for $\{\gamma_{n,k}\}_{n=1}^{N}$).

Results

Figures D.19 and D.20 show the results. These plots show the SNRs in C distinct colors that generated from linear interpolation in the RGB color space from black (●) to light red (●). At each checkpoint, the colors are assigned based on the *order* of the respective directional curvature λ_k: the SNR that belongs to the direction with the smallest curvature is shown in black and the SNR that belongs to the direction with the largest curvature is shown in light red. The color thus encodes only the order of the top-C directional curvatures – *not* their magnitude. We use this color encoding to reveal potential correlations between SNR and curvature.

We find that the gradient SNR along the top-C eigenvectors is consistently small (in comparison to the curvature SNR) and remains roughly on the same level during the optimization. The curvature signal decreases as training proceeds. The SNRs along the top-C eigendirections do not appear to show any significant correlation with the corresponding curvatures. Only for the CIFAR-100 test problems we can suspect a correlation between strong curvature and small curvature SNR.

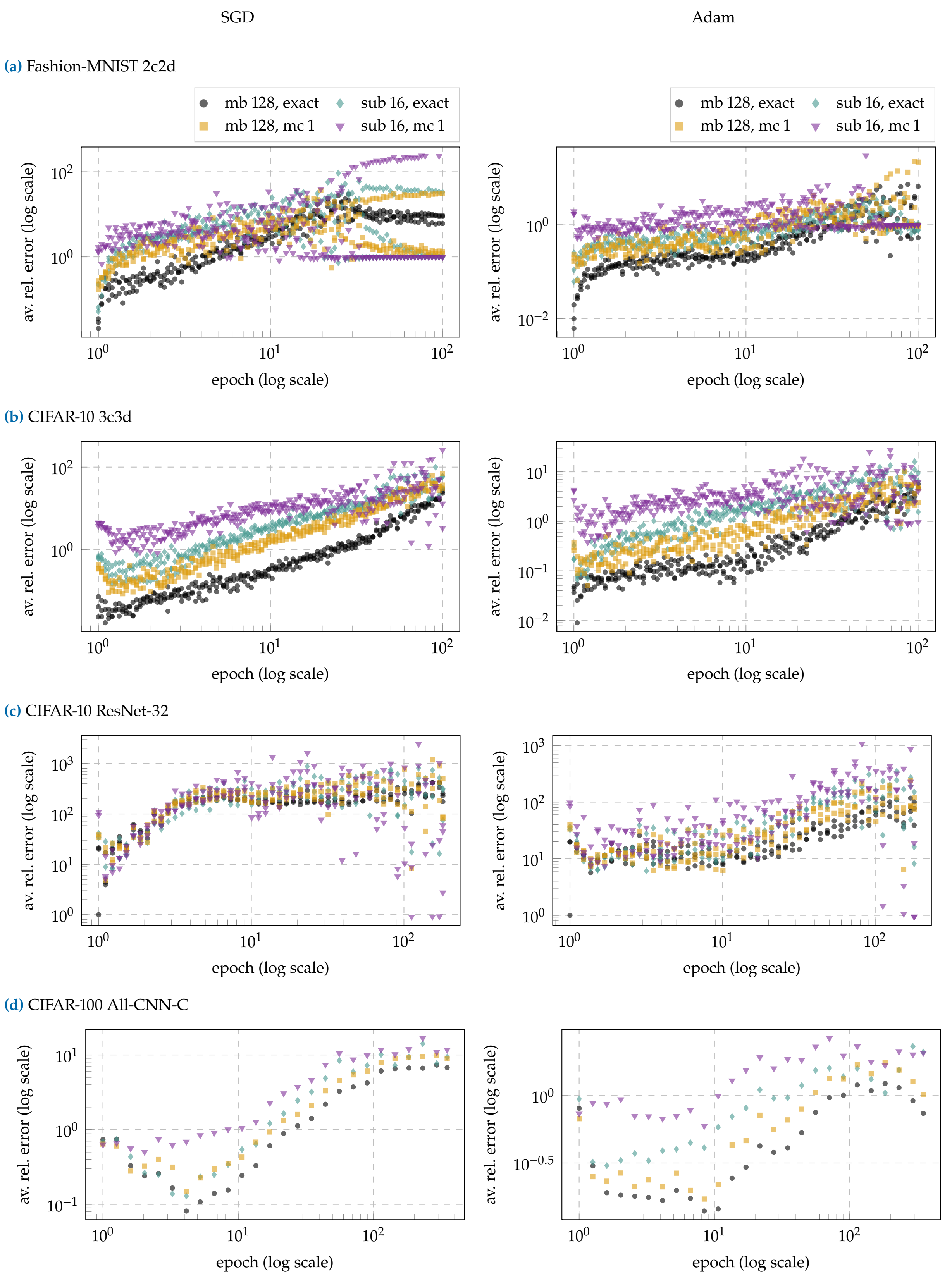
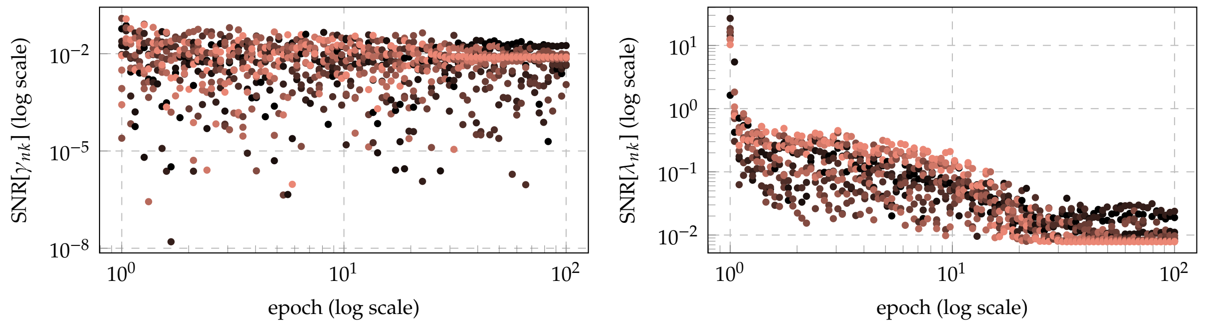


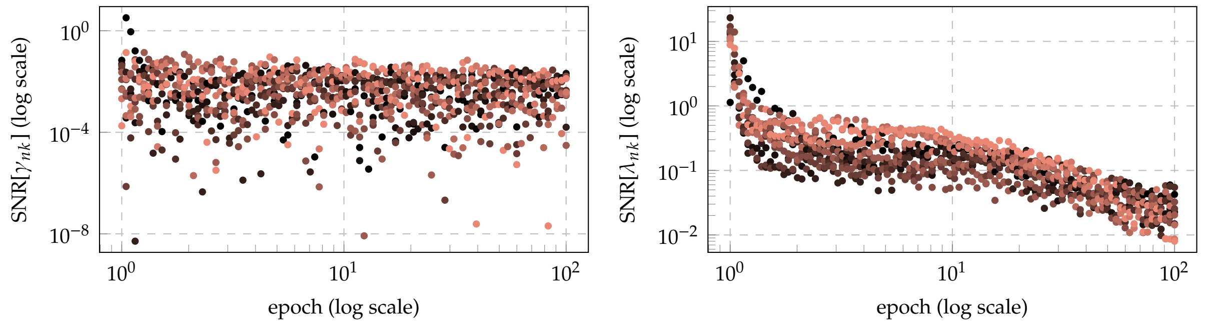
Figure D.18: ViViT's versus full-batch quadratic model. Comparison between approximations to the quadratic model and the full-batch model in terms of the average relative error for the directional curvature during training for all test problems.

First-order derivatives Second-order derivatives

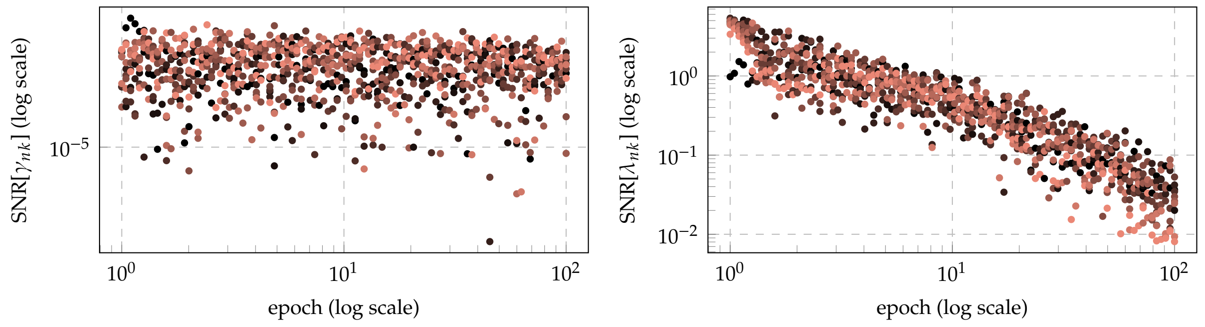
(a) Fashion-MNIST 2c2d SGD



(b) Fashion-MNIST 2c2d Adam



(c) CIFAR-10 3c3d SGD



(d) CIFAR-10 3c3d Adam

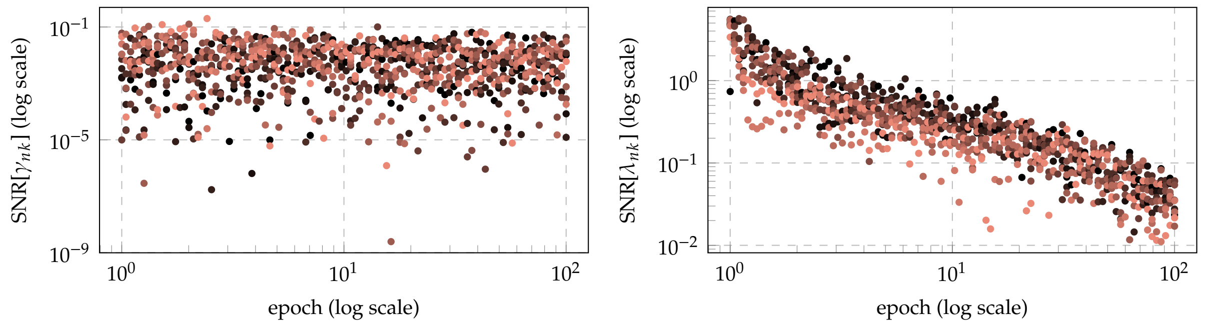
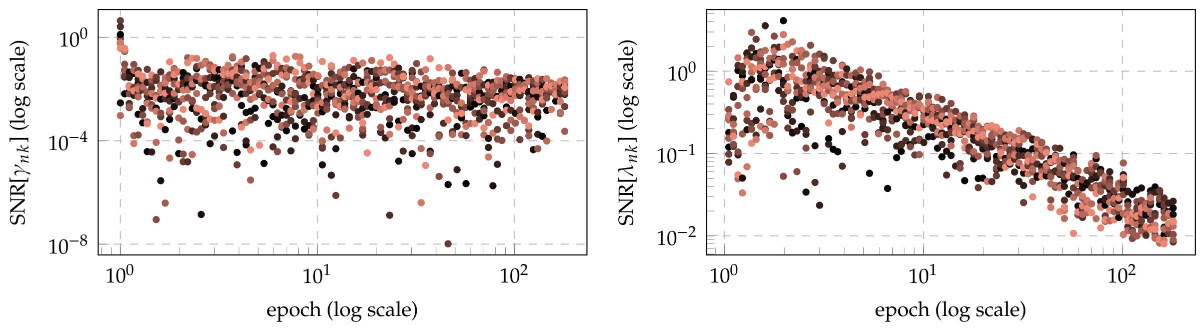


Figure D.19: Directional SNRs (1). SNR along each of the mini-batch GGN's top-C eigenvectors during training for all test problems. At fixed epoch, the SNR for the most curved direction is shown in ● and for the least curved direction in ●.

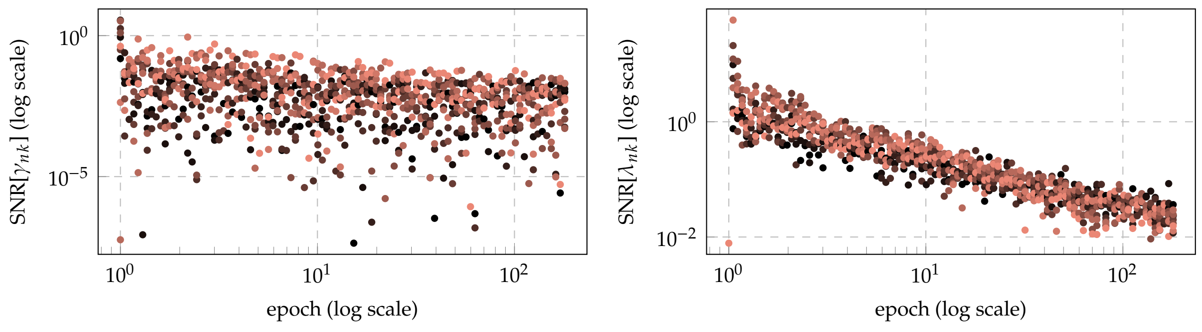
First-order derivatives

Second-order derivatives

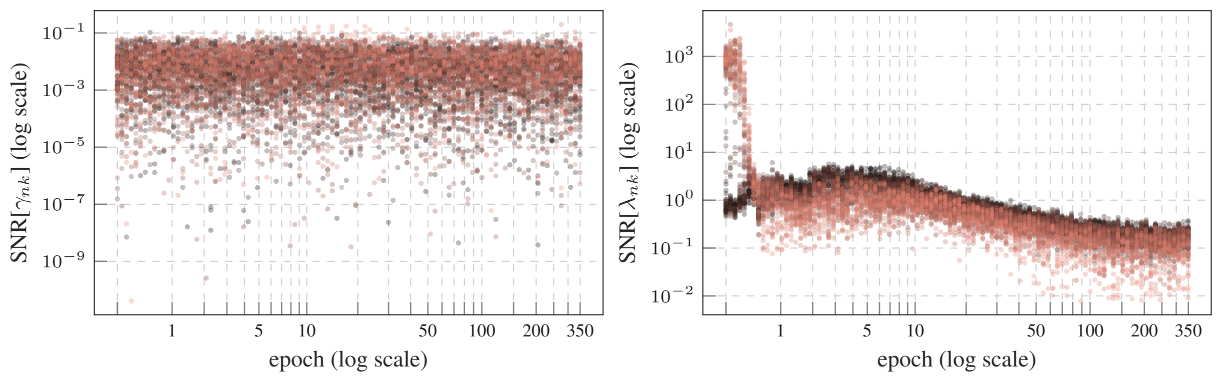
(a) CIFAR-10 ResNet-32 SGD



(b) CIFAR-10 ResNet-32 Adam



(c) CIFAR-100 All-CNN-C SGD



(d) CIFAR-100 All-CNN-C Adam

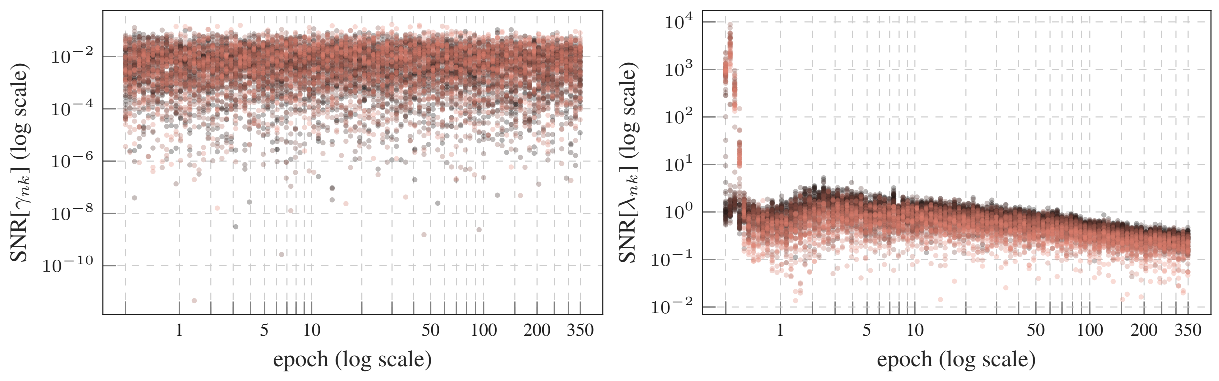


Figure D.20: Directional SNRs (2). SNR along each of the mini-batch GGN's top-C eigenvectors during training for all test problems. At fixed epoch, the SNR for the most curved direction is shown in ● and for the least curved direction in ●.

D.3 Implementation Details

Layer View of Backpropagation

Consider a single layer $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ that transforms inputs $\boldsymbol{z}^{(l-1)}_n \in \mathbb{R}^{h^{(l-1)}}$ into outputs $\boldsymbol{z}^{(l)}_n \in \mathbb{R}^{h^{(l)}}$ by means of a parameter $\boldsymbol{\theta}^{(l)} \in \mathbb{R}^{d^{(l)}}$. During back-propagation for $\boldsymbol{V}$, the layer receives vectors $\boldsymbol{s}^{(l)}_{n,c} = (\mathrm{J}_{\boldsymbol{z}^{(l)}_n} \boldsymbol{f}_n)^\top \boldsymbol{s}_{n,c}$ from the previous stage (recall $\nabla^2_{\boldsymbol{f}_n} \ell_n = \sum_{c=1}^C \boldsymbol{s}_{n,c} \boldsymbol{s}_{n,c}^\top$). Parameter contributions $\boldsymbol{v}^{(l)}_{nc}$ to $\boldsymbol{V}$ are obtained by application of its Jacobian,

$$\begin{aligned}
\boldsymbol{v}^{(l)}_{nc} &= \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{f}_n\right)^\top \boldsymbol{s}_{n,c} \\
&= \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}_n\right)^\top \left(\mathrm{J}_{\boldsymbol{z}^{(l)}_n} \boldsymbol{f}_n\right)^\top \boldsymbol{s}_{n,c} && \text{(Chain rule)} \\
&= \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}_n\right)^\top \boldsymbol{s}^{(l)}_{n,c} \,. && \text{(Definition of } \boldsymbol{s}^{(l)}_{n,c}\text{)} \qquad \text{(D.2)}
\end{aligned}$$

Consequently, $\boldsymbol{\theta}^{(l)}$'s contribution to $\boldsymbol{V}$, denoted by $\boldsymbol{V}^{(l)} \in \mathbb{R}^{d^{(l)} \times NC}$, is

$$\begin{aligned}
&\boldsymbol{V}^{(l)} = \frac{1}{\sqrt{N}} \begin{pmatrix} \boldsymbol{v}^{(l)}_{1,1} & \boldsymbol{v}^{(l)}_{1,2} & \dots & \boldsymbol{v}^{(l)}_{N,C} \end{pmatrix} \\
\text{with} \quad &\boldsymbol{v}^{(l)}_{nc} = \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{f}_n\right)^\top \boldsymbol{s}_{n,c} \,.
\end{aligned} \tag{D.3}$$

D.3.1 Optimized Gram Matrix for Linear Layers

Our goal is to efficiently extract $\boldsymbol{\theta}^{(l)}$'s contribution to the Gram matrix,

$$\tilde{\boldsymbol{G}}^{(l)} = \boldsymbol{V}^{(l)\top} \boldsymbol{V}^{(l)} \in \mathbb{R}^{NC \times NC} \,. \tag{D.4}$$

Gram Matrix via Expanding $\boldsymbol{V}^{(l)}$

One way to construct $\boldsymbol{G}^{(l)}$ is to first expand $\boldsymbol{V}^{(l)}$ (Equation (D.3)) via the Jacobian $\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}_n$, then contract it (Equation (D.4)). This can be a memory bottleneck for large linear layers which are common in many architectures close to the network output. However if only the Gram matrix rather than $\boldsymbol{V}$ is required, structure in the Jacobian can be used to construct $\tilde{\boldsymbol{G}}^{(l)}$ without expanding $\boldsymbol{V}^{(l)}$ and thus reduce this overhead.

Optimization for Linear Layers

Now, let $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ be a linear layer with weights $\boldsymbol{W}^{(l)} \in \mathbb{R}^{h^{(l)} \times h^{(l-1)}}$, *i.e.* $\boldsymbol{\theta}^{(l)} = \mathrm{vec}(\boldsymbol{W}^{(l)}) \in \mathbb{R}^{d^{(l)} = h^{(l)} h^{(l-1)}}$ with column stacking as flattening,

$$f^{(l)}_{\boldsymbol{\theta}^{(l)}} : \quad \boldsymbol{z}^{(l)}_n = \boldsymbol{W}^{(l)} \boldsymbol{z}^{(l-1)}_n \,.$$

The Jacobian is

$$\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}_n = \boldsymbol{z}^{(l-1)\top}_n \otimes \boldsymbol{I}_{h^{(l)}} \,, \tag{D.5}$$

and can be used to compute Gram matrix entries without expanding $\boldsymbol{V}^{(l)}$,

$$
\begin{aligned}
&\left[\tilde{\boldsymbol{G}}^{(l)}\right]_{(n,c),(n',c')} \\
&\quad= \boldsymbol{v}_{n,c}^{(l)\top} \boldsymbol{v}_{n',c'}^{(l)} && \text{(Equation (D.4))} \\
&\quad= \boldsymbol{s}_{n,c}^{(l)\top} \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right) \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_{n'}^{(l)}\right)^\top \boldsymbol{s}_{n',c'}^{(l)} \\
&\quad= \boldsymbol{s}_{n,c}^{(l)\top} \left(\boldsymbol{z}_n^{(l-1)\top} \otimes \boldsymbol{I}_{h^{(l)}}\right) \left(\boldsymbol{z}_{n'}^{(l-1)\top} \otimes \boldsymbol{I}_{h^{(l)}}\right)^\top \boldsymbol{s}_{n',c'}^{(l)} && \text{(Equation (D.5))} \\
&\quad= \boldsymbol{s}_{n,c}^{(l)\top} \left(\boldsymbol{z}_n^{(l-1)\top} \boldsymbol{z}_{n'}^{(l-1)} \otimes \boldsymbol{I}_{h^{(l)}}\right) \boldsymbol{s}_{n',c'}^{(l)} && \text{(Equation (D.2))} \\
&\quad= \boldsymbol{z}_n^{(l-1)\top} \boldsymbol{z}_{n'}^{(l-1)} \boldsymbol{s}_{n,c}^{(l)\top} \boldsymbol{I}_{h^{(l)}} \boldsymbol{s}_{n',c'}^{(l)} && (\boldsymbol{z}_n^{(l-1)\top} \boldsymbol{z}_{n'}^{(l-1)} \in \mathbb{R}) \\
&\quad= \left(\boldsymbol{z}_n^{(l-1)\top} \boldsymbol{z}_{n'}^{(l-1)}\right) \left(\boldsymbol{s}_{n,c}^{(l)\top} \boldsymbol{s}_{n',c'}^{(l)}\right) .
\end{aligned}
$$

We see that the Gram matrix is built from two Gram matrices based on $\{\boldsymbol{z}_n^{(l-1)}\}_{n=1}^N$ and $\{\boldsymbol{s}_{n,c}^{(l)}\}_{n=1,c=1}^{N,C}$, that require $\mathcal{O}(N^2)$ and $\mathcal{O}((NC)^2)$ memory, respectively. In comparison, the naïve approach via $\boldsymbol{V}^{(l)} \in \mathbb{R}^{d^{(l)} \times NC}$ scales with the number of weights, which is often comparable to D. For instance, the 3c3d architecture on CIFAR-10 has $D = 895{,}210$ and the largest weight matrix has $d^{(l)} = 589{,}824$, whereas $NC = 1{,}280$ during training [146].

D.3.2 Implicit Multiplication by the Inverse (Block-Diagonal) GGN

Inverse GGN-Vector Products

A damped Newton step[1] requires multiplication by $(\boldsymbol{G} + \delta \boldsymbol{I}_D)^{-1}$. By means of Equation (7.3) and the matrix inversion lemma,

1: Instead of $\delta \boldsymbol{I}_D$, any other easy-to-invert matrix can be used.

$$
\begin{aligned}
&(\delta \boldsymbol{I}_D + \boldsymbol{G})^{-1} \\
&\quad= \left(\delta \boldsymbol{I}_D + \boldsymbol{V}\boldsymbol{V}^\top\right)^{-1} && \text{(Equation (7.3))} \qquad \text{(D.6)} \\
&\quad= \frac{1}{\delta}\left(\boldsymbol{I}_D + \frac{1}{\delta}\boldsymbol{V}\boldsymbol{V}^\top\right)^{-1} \\
&\quad= \frac{1}{\delta}\left[\boldsymbol{I}_D - \frac{1}{\delta}\boldsymbol{V}\left(\boldsymbol{I}_{NC} + \boldsymbol{V}^\top \frac{1}{\delta}\boldsymbol{V}\right)^{-1} \boldsymbol{V}^\top\right] && \text{(Matrix inversion lemma)} \\
&\quad= \frac{1}{\delta}\left[\boldsymbol{I}_D - \boldsymbol{V}\left(\delta \boldsymbol{I}_{NC} + \boldsymbol{V}^\top \boldsymbol{V}\right)^{-1} \boldsymbol{V}^\top\right] && \text{(Gram matrix)} \\
&\quad= \frac{1}{\delta}\left[\boldsymbol{I}_D - \boldsymbol{V}\left(\delta \boldsymbol{I}_{NC} + \tilde{\boldsymbol{G}}\right)^{-1} \boldsymbol{V}^\top\right] . && \text{(D.7)}
\end{aligned}
$$

Inverse GGN-vector products require inversion of the damped Gram matrix as well as applications of $\boldsymbol{V}, \boldsymbol{V}^\top$ for the transformations between Gram and parameter space.

Inverse Block-Diagonal GGN-Vector Products

Next, we replace the full GGN by its block diagonal approximation $\boldsymbol{G} \approx \boldsymbol{G}_{\text{BDA}} = \text{diag}(\boldsymbol{G}^{(1)}, \boldsymbol{G}^{(2)}, \dots)$ with

$$\boldsymbol{G}^{(l)} = \boldsymbol{V}^{(l)}\boldsymbol{V}^{(l)\top} \in \mathbb{R}^{d^{(l)} \times d^{(l)}}$$

and $\boldsymbol{V}^{(l)}$ as in Equation (D.3). Then, inverse multiplication reduces to each block,

$$\boldsymbol{G}_{\text{BDA}}^{-1} = \text{diag}\left(\boldsymbol{G}^{(1)^{-1}}, \boldsymbol{G}^{(2)^{-1}}, \dots\right) .$$

If again a damped Newton step is considered, we can reuse Equation (D.7) with the substitutions

$$(\boldsymbol{G}, D, \boldsymbol{V}, \boldsymbol{V}^\top, \tilde{\boldsymbol{G}}) \leftrightarrow \left(\boldsymbol{G}^{(l)}, d^{(l)}, \boldsymbol{V}^{(l)}, \boldsymbol{V}^{(l)^\top}, \tilde{\boldsymbol{G}}^{(l)}\right)$$

to apply the inverse and immediately discard the ViViT factors: At backpropagation of layer $T^{(l)}_{\boldsymbol{\theta}^{(l)}}$

1. Compute $\boldsymbol{V}^{(l)}$ using Equation (D.3).
2. Compute $\tilde{\boldsymbol{G}}^{(l)}$ using Equation (D.4).
3. Compute $\left(\delta \boldsymbol{I}_{NC} + \tilde{\boldsymbol{G}}^{(l)}\right)^{-1}$.
4. Apply the inverse in Equation (D.7) with the above substitutions to the target vector.
5. Discard $\boldsymbol{V}^{(l)}, \boldsymbol{V}^{(l)^\top}, \tilde{\boldsymbol{G}}^{(l)}$, and $\left(\delta \boldsymbol{I}_{NC} + \tilde{\boldsymbol{G}}^{(l)}\right)^{-1}$. Proceed to layer $i-1$.

Note that the above scheme should only be used for parameters that satisfy $d^{(l)} > NC$, *i.e.* $\dim(\boldsymbol{G}^{(l)}) > \dim(\tilde{\boldsymbol{G}}^{(l)})$. Low-dimensional parameters can be grouped with others to increase their joint dimension, and to control the block structure of $\boldsymbol{G}_{\text{BDA}}$.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A System for Large-Scale Machine Learning". *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[2] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. "Deep Learning with Differential Privacy". *ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[3] R. P. Adams, J. Pennington, M. J. Johnson, J. Smith, Y. Ovadia, B. Patton, and J. Saunderson. "Estimating the Spectral Density of Large Implicit Matrices". 2018.

[4] A. M. Agrawal, A. Tendle, H. Sikka, S. Singh, and A. Kayid. "Investigating Learning in Deep Neural Networks using Layer-Wise Weight Change". 2020.

[5] S.-I. Amari. "Natural Gradient works Efficiently in Learning". *Neural computation* (1998).

[6] A. Bahamou and D. Goldfarb. "A Dynamic Sampling Adaptive-SGD Method for Machine Learning". 2019.

[7] D. Bahdanau, K. Cho, and Y. Bengio. "Neural machine translation by jointly learning to align and translate" (2014).

[8] C. Bakker, M. J. Henry, and N. O. Hodas. "The Outer Product Structure of Neural Network Derivatives" (2018).

[9] L. Balles. "Noise-Aware Stochastic Optimization" (2022). PhD thesis.

[10] L. Balles and P. Hennig. "Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients". *International Conference on Machine Learning (ICML)*. 2018.

[11] L. Balles, J. Romero, and P. Hennig. "Coupling Adaptive Batch Sizes with Learning Rates". *Conference on Uncertainty in Artificial Intelligence (UAI)*. 2017.

[12] F. L. Bauer. "Computational Graphs and Rounding Error". *SIAM Journal on Numerical Analysis* (1974).

[13] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. "Automatic Differentiation in Machine Learning: A Survey". *Journal of Machine Learning Research (JMLR)* (2018).

[14] S. Becker and Y. Le Cun. "Improving the Convergence of Back-Propagation Learning with Second Order Methods". *Connectionist Models Summer School*. 1989.

[15] Y. Bengio. "Practical recommendations for gradient-based training of deep architectures". *Neural Networks* (2012).

[16] A. Bernacchia, M. Lengyel, and G. Hennequin. "Exact natural gradient in deep linear networks and its application to the nonlinear case". *Advances in Neural Information Processing Systems (NeurIPS*. 2018.

[17] L. Biewald. "Experiment Tracking with Weights and Biases". 2020.

[18] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag. "What is the state of neural network pruning?" *Proceedings of machine learning and systems* (2020).

[19] R. Bollapragada, R. H. Byrd, and J. Nocedal. "Adaptive Sampling Strategies for Stochastic Optimization". *SIAM Journal on Optimization* (2017).

[20] A. Bordes, L. Bottou, and P. Gallinari. "SGD-QN: Careful Quasi-Newton Stochastic Gradient Descent". *Journal of Machine Learning Research (JMLR)* (2009).

[21] A. Botev, H. Ritter, and D. Barber. "Practical Gauss-Newton Optimisation for Deep Learning". *International Conference on Machine Learning (ICML)*. 2017.

[22] L. Bottou, F. E. Curtis, and J. Nocedal. "Optimization Methods for Large-Scale Machine Learning" (2016).

[23] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. "JAX: composable transformations of Python + NumPy programs" (2018).

[24] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. "Language Models are Few-Shot Learners" (2020).

[25] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal. "On the Use of Stochastic Hessian Information in Optimization Methods for Machine Learning". *SIAM Journal on Optimization* (2011).

[26] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. "Sample Size Selection in Optimization Methods for Machine Learning". *Mathematicl Programming* (2012).

[27] S. Chatterjee. "Coherent Gradients: An Approach to Understanding Generalization in Gradient Descent-based Optimization". *International Conference on Learning Representations (ICLR)*. 2020.

[28] S. Chatterjee and P. Zielinski. "Making Coherence Out of Nothing At All: Measuring the Evolution of Gradient Alignment". 2020.

[29] K. Chellapilla, S. Puri, and P. Simard. "High Performance Convolutional Neural Networks for Document Processing". *International Workshop on Frontiers in Handwriting Recognition*. 2006.

[30] C. Chen, S. Reiz, C. D. Yu, H.-J. Bungartz, and G. Biros. "Fast Approximation of the Gauss–Newton Hessian Matrix for the Multilayer Perceptron". *SIAM Journal on Matrix Analysis and Applications* (2021).

[31] S.-W. Chen, C.-N. Chou, and E. Chang. "BDA-PCH: Block-Diagonal Approximation of Positive-Curvature Hessian for Training Neural Networks" (2018).

[32] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". *Neural Information Processing Systems (NIPS), Workshop on Machine Learning Systems*. 2015.

[33] K. Cho, B. Merrienboer, D. Bahdanau, and Y. Bengio. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches" (2014).

[34] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl. "On Empirical Comparisons of Optimizers for Deep Learning". 2020.

[35] Y. Choi, Y. Uh, J. Yoo, and J.-W. Ha. "StarGAN v2: Diverse Image Synthesis for Multiple Domains". *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "Introduction to Algorithms". 2001.

[37] F. Dangel, S. Harmeling, and P. Hennig. "Modular Block-diagonal Curvature Approximations for Feedforward Architectures". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2020.

[38] F. Dangel, F. Kunstner, and P. Hennig. "BackPACK: Packing more into Backprop". *International Conference on Learning Representations (ICLR)*. 2020.

[39] F. Dangel, L. Tatzel, and P. Hennig. "ViViT: Curvature Access Through The Generalized Gauss-Newton's Low-Rank Structure". *Transactions on Machine Learning Research (TMLR)* (2022).

[40] E. Daxberger, A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig. "Laplace Redux–Effortless Bayesian Deep Learning". *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". *IEEE conference on computer vision and pattern recognition (CVPR)*. 2009.

[42] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio. "Sharp Minima Can Generalize For Deep Nets". *International Conference on Machine Learning (ICML)*. 2017.

[43] X. Dong, S. Chen, and S. Pan. "Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon". *Advances in Neural Information Processing Systems (NIPS)*. 2017.

[44] J. Duchi, E. Hazan, and Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *Journal of Machine Learning Research (JMLR)* (2011).

[45] V. Dumoulin and F. Visin. "A guide to convolution arithmetic for deep learning" (2016).

[46] C. Dwork, A. Roth, et al. "The algorithmic foundations of differential privacy". *Foundations and Trends in Theoretical Computer Science* (2014).

[47] J. L. Elman. "Finding structure in time". *Cognitive Science* (1990).

[48] F. Faghri, D. Duvenaud, D. J. Fleet, and J. Ba. "A Study of Gradient Variance in Deep Learning". 2020.

[49] W. Fedus, B. Zoph, and N. Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". *Journal of Machine Learning Research (JMLR)* (2022).

[50] H. Fischer. "A History of the Central Limit Theorem: From Classical to Modern Probability Theory". 2010.

[51] J. Frankle, D. J. Schwab, and A. S. Morcos. "The Early Phase of Neural Network Training". *International Conference on Learning Representations (ICLR)*. 2020.

[52] M. Fredrikson, S. Jha, and T. Ristenpart. "Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures". *ACM Conference on Computer and Communications Security (CCS)*. 2015.

[53] M. Gargiani, A. Zanelli, M. Diehl, and F. Hutter. "On the Promise of the Stochastic Generalized Gauss-Newton Method for Training DNNs". 2020.

[54] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent. "Fast Approximate Natural Gradient Descent in a Kronecker-factored Eigenbasis" (2018).

[55] B. Ghorbani, S. Krishnan, and Y. Xiao. "An Investigation into Neural Net Optimization via Hessian Eigenvalue Density". *International Conference on Machine Learning (ICML)*. 2019.

[56] B. Ginsburg. "On regularization of gradient descent, layer imbalance and flat minima". 2020.

[57] X. Glorot and Y. Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2010.

[58] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative Adversarial Nets". *Neural Information Processing Systems (NIPS)*. 2014.

[59] I. J. Goodfellow. "Efficient Per-Example Gradient Computations" (2015).

[60] I. J. Goodfellow, Y. Bengio, and A. Courville. "Deep Learning". 2016.

[61] D. Granziol, X. Wan, and T. Garipov. "Deep Curvature Suite". 2021.

[62] A. Griewank. "Who invented the reverse mode of differentiation?" *Documenta Mathematica* (2012).

[63] R. Grosse and J. Martens. "A Kronecker-factored approximate Fisher matrix for convolution layers". *International Conference on Machine Learning (ICML)*. 2016.

[64] I. Gulrajani and D. Lopez-Paz. "In Search of Lost Domain Generalization". *International Conference on Learning Representations (ICLR)*. 2021.

[65] G. Gur-Ari, D. A. Roberts, and E. Dyer. "Gradient Descent Happens in a Tiny Subspace". 2018.

[66] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. "Array programming with NumPy". *Nature* (2020).

[67] B. Hassibi and D. Stork. "Second order derivatives for network pruning: Optimal Brain Surgeon". *Advances in Neural Information Processing Systems (NIPS)*. 1992.

[68] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[69] J. F. Henriques, S. Ehrhardt, S. Albanie, and A. Vedaldi. "Small Steps and Giant Leaps: Minimal Newton Solvers for Deep Learning". *International Conference on Computer Vision (ICCV)*. 2019.

[70] S. Hochreiter and J. Schmidhuber. "Flat Minima". *Neural Computation* (1997).

[71] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory”. *Neural Computation* (1997).

[72] R. Z. Horace He. “functorch: JAX-like composable function transforms for PyTorch”. 2021.

[73] J. Hughes. “Why Functional Programming Matters”. *Computer Journal* (1989).

[74] Y. Idelbayev. “Proper ResNet Implementation for CIFAR10/CIFAR100 in PyTorch”. 2018.

[75] A. Immer, M. Bauer, V. Fortuin, G. Rätsch, and K. M. Emtiyaz. “Scalable Marginal Likelihood Estimation for Model Selection in Deep Learning”. *International Conference on Machine Learning (ICML)*. 2021.

[76] A. Immer, M. Korzepa, and M. Bauer. “Improving predictions of Bayesian neural nets via local linearization”. *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2021.

[77] M. Innes. “Don’t Unroll Adjoint: Differentiating SSA-Form Programs” (2018).

[78] M. Innes. “Flux: Elegant machine learning with Julia”. *Journal of Open Source Software* (2018).

[79] S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. *International Conference on Machine Learning (ICML)*. 2015.

[80] S. Jastrzebski, D. Arpit, O. Astrand, G. B. Kerg, H. Wang, C. Xiong, R. Socher, K. Cho, and K. J. Geras. “Catastrophic Fisher Explosion: Early Phase Fisher Matrix Impacts Generalization”. *International Conference on Machine Learning (ICML)*. 2021.

[81] S. Jastrzebski, M. Szymczak, S. Fort, D. Arpit, J. Tabor, K. Cho, and K. Geras. “The Break-Even Point on Optimization Trajectories of Deep Neural Networks”. *International Conference on Learning Representations (ICLR)*. 2020.

[82] W. P. Johnson. “The curious history of Faà di Bruno’s formula”. *The American mathematical monthly* (2002).

[83] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. *Annual International Symposium on Computer Architecture*.

[84] T. Karras, T. Aila, S. Laine, and J. Lehtinen. “Progressive Growing of GANs for Improved Quality, Stability, and Variation”. *International Conference on Learning Representations (ICLR)*. 2018.

[85] A. Katharopoulos and F. Fleuret. “Not All Samples Are Created Equal: Deep Learning with Importance Sampling”. *International Conference on Machine Learning (ICML)*. 2018.

[86] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. *International Conference on Learning Representations (ICLR)*. 2017.

[87] D. P. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. *International Conference on Learning Representations (ICLR)*. 2015.

[88] D. Kirk. “NVIDIA CUDA software and GPU parallel computing architecture”. 2007.

[89] A. Kristiadi, M. Hein, and P. Hennig. “Being Bayesian, even just a bit, fixes overconfidence in ReLU networks”. *International Conference on Machine Learning (ICML)*. 2020.

[90] A. Krizhevsky. *Learning multiple layers of features from tiny images*. Technical report. 2009.

[91] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. *Neural Information Processing Systems (NIPS)*. 2012.

[92] F. Kunstner, L. Balles, and P. Hennig. “Limitations of the Empirical Fisher Approximatiom”. *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[93] P. S. Laplace. "Mémoire sur la probabilité de causes par les évenements". *Mémoire de l'académie royale des sciences* (1774).

[94] N. Le Roux, P. Manzagol, and Y. Bengio. "Topmoumoute Online Natural Gradient Algorithm". *Advances in Neural Information Processing Systems 20*. 2007.

[95] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". *Proceedings of the IEEE*. 1998.

[96] Y. LeCun, J. Denker, and S. Solla. "Optimal Brain Damage". *Neural Information Processing Systems (NIPS)*. 1989.

[97] Y. LeCun, P. Simard, and B. Pearlmutter. "Automatic Learning Rate Maximization by On-Line Estimation of the Hessian's Eigenvectors". *Neural Information Processing Systems (NIPS)*. 1993.

[98] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. "Microsoft COCO: Common Objects in Context". *European Conference on Computer Vision (ECCV)*. 2014.

[99] S. Linnainmaa. "Taylor expansion of the accumulated rounding error". *BIT Numerical Mathematics* (1976).

[100] J. Liu, Y. Bai, G. Jiang, T. Chen, and H. Wang. "Understanding Why Neural Networks Generalize Well Through GSNR of Parameters". *International Conference on Learning Representations (ICLR)*. 2020.

[101] C. F. Loan. "The ubiquitous Kronecker product". *Journal of Computational and Applied Mathematics* (2000).

[102] M.-T. Luong, H. Pham, and C. D. Manning. "Effective approaches to attention-based neural machine translation" (2015).

[103] J. R. Magnus and H. Neudecker. "Matrix Differential Calculus with Applications in Statistics and Econometrics". 1999.

[104] M. Mahsereci, L. Balles, C. Lassner, and P. Hennig. "Early Stopping without a Validation Set". 2017.

[105] M. Mahsereci and P. Hennig. "Probabilistic Line Searches for Stochastic Optimization". *Journal of Machine Learning Research (JMLR)* (2017).

[106] J. Martens. "Deep Learning via Hessian-Free Optimization." *International Conference on Machine Learning (ICML)*. 2010.

[107] J. Martens. "New perspectives on the natural gradient method" (2014).

[108] J. Martens, J. Ba, and M. Johnson. "Kronecker-factored Curvature Approximations for Recurrent Neural Networks". *International Conference on Learning Representations (ICLR)*. 2018.

[109] J. Martens and R. Grosse. "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature". *International Conference on Machine Learning (ICML)*. 2015.

[110] E. Mizutani and S. E. Dreyfus. "Second-Order Stagewise Backpropagation for Hessian-Matrix Analyses and Investigation of Negative Curvature". *Neural Networks* (2008).

[111] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing Atari with Deep Reinforcement Learning" (2013).

[112] R. Mulayoff and T. Michaeli. "Unique Properties of Flat Minima in Deep Networks". *International Conference on Machine Learning (ICML)*. 2020.

[113] V. Nagarajan and J. Z. Kolter. "Generalization in Deep Networks: The Role of Distance from Initialization". 2019.

[114] U. Naumann. "Optimal Jacobian Accumulation is NP-Complete". *Mathematical Programming* (2008).

[115] M. Naumov. "Feedforward and Recurrent Neural Networks Backward Propagation and Hessian in Matrix Form" (2017).

[116] D. Needell, R. Ward, and N. Srebro. "Stochastic Gradient Descent, Weighted Sampling, and the Randomized Kaczmarz algorithm". *Advances in Neural Information Processing Systems (NIPS)*. 2014.

[117] Y. Nesterov. "A method for solving the convex programming problem with convergence rate $O(1/k^2)$". *Proceedings of the USSR Academy of Sciences* (1983).

[118] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. "Reading Digits in Natural Images with Unsupervised Feature Learning". 2011.

[119] D. Oktay, N. McGreivy, J. Aduol, A. Beatson, and R. P. Adams. "Randomized Automatic Differentiation". *International Conference on Learning Representations (ICLR)*. 2021.

[120] Y. Ollivier, L. Arnold, A. Auger, and N. Hansen. "Information-Geometric Optimization Algorithms: A Unifying Picture via Invariance Principles" (2011).

[121] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka. "Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks". *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[122] V. Papyan. "Measurements of Three-Level Hierarchical Structure in the Outliers in the Spectrum of Deepnet Hessians". *International Conference on Machine Learning (ICML)*. 2019.

[123] V. Papyan. "The Full Spectrum of Deepnet Hessians at Scale: Dynamics with SGD Training and Sample Size". 2019.

[124] V. Papyan. "Traces of Class/Cross-Class Structure Pervade Deep Learning Spectra". *Journal of Machine Learning Research (JMLR)* (2020).

[125] R. Pascanu and Y. Bengio. "Revisiting natural gradient for deep networks" (2013).

[126] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[127] B. A. Pearlmutter. "Fast Exact Multiplication by the Hessian". *Neural Computation* (1994).

[128] B. Polyak. "Some methods of speeding up the convergence of iteration methods". *USSR Computational Mathematics and Mathematical Physics* (1964).

[129] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. "Language Models are Unsupervised Multitask Learners" (2019).

[130] A. Rame, C. Dancette, and M. Cord. "Fishr: Invariant Gradient Variances for Out-of-distribution Generalization". *International Conference on Machine Learning (ICML)*. 2022.

[131] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. "Zero-Shot Text-to-Image Generation". *International Conference on Machine Learning (ICML)*. 2021.

[132] S. J. Reddi, S. Kale, and S. Kumar. "On the Convergence of Adam and Beyond". *International Conference on Learning Representations (ICLR)*. 2018.

[133] H. Ritter, A. Botev, and D. Barber. "A Scalable Laplace Approximation for Neural Networks". *International Conference on Learning Representations (ICLR)*. 2018.

[134] H. Ritter, A. Botev, and D. Barber. "Online Structured Laplace Approximations for Overcoming Catastrophic Forgetting". *Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

[135] H. Robbins and S. Monro. "A Stochastic Approximation Method". *The Annals of Mathematical Statistics* (1951).

[136] F. Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological Review* (1958).

[137] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. 1986.

[138] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. "Imagenet large scale visual recognition challenge". *International journal of computer vision* (2015).

[139] L. Sagun, L. Bottou, and Y. LeCun. "Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond". 2017.

[140] L. Sagun, U. Evci, V. U. Guney, Y. Dauphin, and L. Bottou. “Empirical Analysis of the Hessian of Over-Parametrized Neural Networks”. 2018.

[141] A. Sankaran, N. A. Alashti, C. Psarras, and P. Bientinesi. “Benchmarking the Linear Algebra Awareness of TensorFlow and PyTorch”. 2022.

[142] K. A. Sankararaman, S. De, Z. Xu, W. R. Huang, and T. Goldstein. “The Impact of Neural Network Overparameterization on Gradient Confusion and Stochastic Gradient Descent”. *International Conference on Machine Learning (ICML)*. 2020.

[143] A. M. Saxe, J. L. McClelland, and S. Ganguli. “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks”. *International Conference on Learning Representations (ICLR)*. 2014.

[144] M. Schmidt. “Convergence rate of stochastic gradient with constant step size” (2014).

[145] R. M. Schmidt, F. Schneider, and P. Hennig. “Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers”. *International Conference on Machine Learning (ICML)*. 2021.

[146] F. Schneider, L. Balles, and P. Hennig. “DeepOBS: A Deep Learning Optimizer Benchmark Suite”. *International Conference on Learning Representations (ICLR)*. 2019.

[147] F. Schneider, F. Dangel, and P. Hennig. “Cockpit: A Practical Debugging Tool for Training Deep Neural Networks”. *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[148] N. N. Schraudolph. “Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent”. *Neural Computation* (2002).

[149] R. Shokri and V. Shmatikov. “Privacy-Preserving Deep Learning”. *ACM SIGSAC Conference on Computer and Communications Security*. 2015.

[150] C. Shorten and T. M. Khoshgoftaar. “A survey on image data augmentation for deep learning”. *Journal of big data* (2019).

[151] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. *Nature* (2016).

[152] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. *Science* (2018).

[153] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. *International Conference on Learning Representations (ICLR)*. 2015.

[154] S. P. Singh and D. Alistarh. “WoodFisher: Efficient Second-Order Approximation for Neural Network Compression”. *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

[155] M. Skorski, A. Temperoni, and M. Theobald. “Revisiting Weight Initialization of Deep Neural Networks”. *Asian Conference on Machine Learning (ACML)*. 2021.

[156] D. So, W. Mańke, H. Liu, Z. Dai, N. Shazeer, and Q. V. Le. “Searching for Efficient Transformers for Language Modeling”. *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[157] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. “Striving for Simplicity: The All Convolutional Net”. *International Conference on Learning Representations (ICLR)*. 2015.

[158] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. *Journal of Machine Learning Research (JMLR)* (2014).

[159] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. “Going deeper with convolutions”. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.

[160] K. Takeuchi. “The distribution of information statistics and the criterion of goodness of fit of models”. *Mathematical Science* (1976).

[161] L. Theis, I. Korshunova, A. Tejani, and F. Huszár. “Faster gaze prediction with dense networks and Fisher pruning” (2018).

[162] V. Thomas, F. Pedregosa, B. van Merriënboer, P.-A. Manzagol, Y. Bengio, and N. L. Roux. "On the interplay between noise and curvature and its effect on optimization and generalization". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2020.

[163] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso. "The Computational Limits of Deep Learning". 2020.

[164] T. Tieleman, G. Hinton, et al. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". *COURSERA: Neural networks for machine learning* (2012).

[165] S. Tokui, K. Oono, S. Hido, and J. Clayton. "Chainer: A next-generation open source framework for deep learning". *Neural Information Processing Systems (NIPS), Workshop on Machine Learning Systems*. 2015.

[166] Y. Tsuji, K. Osawa, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka. "Performance Optimizations and Analysis of Distributed Deep Learning with Approximated Second-Order Optimization Method". *International Conference on Parallel Processing, Workshop Proceedings*. 2019.

[167] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. "Attention is All you Need". *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[168] S. Vaswani, A. Mishkin, I. Laradji, M. Schmidt, G. Gidel, and S. Lacoste-Julien. "Painless Stochastic Gradient: Interpolation, Line-Search, and Convergence Rates". *Neural Information Processing Systems (NeurIPS)*. 2019.

[169] L. Wang, Y. Yang, M. R. Min, and S. T. Chakradhar. "Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent". *Neural networks* (2017).

[170] J. Warsa, T. Wareing, J. Morel, J. Mcghee, and R. Lehoucq. "Krylov Subspace Iterations for Deterministic k-Eigenvalue Calculations". *Nuclear Science and Engineering* (2004).

[171] Wikipedia. "Toeplitz matrix". `https://en.wikipedia.org/w/index.php?title=Toeplitz_matrix&oldid=1094551484`. (online) accessed August 11. 2022.

[172] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. "Google's neural machine translation system: Bridging the gap between human and machine translation" (2016).

[173] Y. Wu, M. Ren, R. Liao, and R. B. Grosse. "Understanding short-horizon bias in stochastic meta-optimization". *International Conference on Learning Representations (ICLR)* (2018).

[174] Y. Wu and K. He. "Group Normalization". *International Journal of Computer Vision* (2019).

[175] H. Xiao, K. Rasul, and R. Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". 2017.

[176] C. Xing, D. Arpit, C. Tsirigotis, and Y. Bengio. "A Walk with SGD". 2018.

[177] Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney. "PyHessian: Neural Networks Through the Lens of the Hessian". *IEEE International Conference on Big Data*. 2020.

[178] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. W. Mahoney. "ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning". 2021.

[179] A. Yousefpour, I. Shilov, A. Sablayrolles, D. Testuggine, K. Prasad, M. Malek, J. Nguyen, S. Ghosh, A. Bharadwaj, J. Zhao, G. Cormode, and I. Mironov. "Opacus: User-Friendly Differential Privacy Library in PyTorch". *Advances in Neural Information Processing Systems (NeurIPS), Workshop Privacy in Machine Learning*. 2021.

[180] M. Zeiler. "ADADELTA: An adaptive learning rate method" (2012).

[181] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". *European Conference on Computer Vision (ECCV)*. 2014.

[182] W. Zeng and R. Urtasun. "MLPrune: Multi-Layer Pruning for Automated Neural Network Compression". 2018.

[183] H. Zhang, C. Xiong, J. Bradbury, and R. Socher. "Block-diagonal Hessian-free Optimization for Training Neural Networks" (2017).

[184] P. Zhao and T. Zhang. “Stochastic Optimization with Importance Sampling for Regularized Loss Minimization”. *International Conference on Machine Learning (ICML)*. 2015.