Advancing Curvature-Based Methods in Machine Learning

Dissertation

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

vorgelegt von
M. Sc. Lukas Nicola Tatzel
aus Stuttgart

Tübingen
2025

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

Tag der mündlichen Qualifikation:	17.09.2025
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Philipp Hennig
2. Berichterstatter:	Prof. Dr. Matthias Hein

Disclaimer. This thesis uses Felix Dangel's PhD thesis template from https://github.com/f-dangel/phd-thesis-template. It is originally based on Federico Marotta's kaobook.

I acknowledge the use of ChatGPT 4 Turbo and Claude 3.7 Sonnet as assistive tools to improve the clarity and readability of selected paragraphs in this thesis. All AI-assisted content has been carefully reviewed and verified. I take full responsibility for the accuracy, originality, and academic integrity of the final work.

Acknowledgments

To begin with, I would like to express my sincere gratitude to my supervisor, Philipp Hennig. Thank you for offering me the opportunity to join your group and work on an exciting project. Your consistently positive and encouraging guidance has been invaluable throughout my PhD. I greatly admire your scientific insight, as well as your ability to foster a friendly, inclusive, and supportive research environment.

I thank Matthias Hein, my second examiner, for his time and effort in reviewing this thesis. I am also grateful to Hendrik Lensch and Peter Gehler for kindly agreeing to join the examination committee. I would also like to thank Philipp Hennig, Bob Williamson, and Michael Mühlebach, who served on my Thesis Advisory Committee and provided valuable guidance throughout my PhD.

My thanks extend to the IMPRS-IS community for providing a stimulating academic environment and a well-organized graduate program. In particular, I would like to thank Leila Masri and Sara Sorce for their tireless support.

I am very grateful to Franziska Weiler for her continuous support with administrative matters and for always being approachable and supportive, even with personal concerns.

To all members of the Methods of Machine Learning group: Thank you for making this time both productive and genuinely enjoyable. It has been a real privilege to work alongside such talented, kind, and generous people. I especially cherish the many moments we shared outside of academic life—music sessions, skiing trips, swimming, art and movie nights, and more.

I would also like to thank all colleagues and former group members who took the time to provide feedback on parts of this thesis.

I feel fortunate to have collaborated with many outstanding scientists during my PhD. In particular, I am grateful to Felix Dangel, Frank Schneider, Jonathan Wenger, and Bálint Mucsányi. Thank you for your patience with me, for the insightful and productive discussions, and for your support and encouragement when I needed it most.

Finally, I would like to express my heartfelt gratitude to my parents—Bianca and Georg Tatzel—my sister, Leonie Tatzel, and my friends for their unwavering support over the years. I am especially grateful to my partner, Julia Steberl. Your patience, understanding, and emotional support have meant a great deal to me. Thank you for being by my side throughout this journey.

Thank you!

Lukas Tatzel
Tübingen, May 27, 2025

Abstract

The second-order Taylor polynomial provides a local, "bowl-shaped" approximation of a given function. This approximation serves as the foundation for *curvature-based* methods. These include in particular (i) optimization techniques based on the Newton step—the step into the minimum of the local approximation—as well as (ii) the Laplace approximation. The latter allows arbitrary probability distributions to be approximated by simpler Gaussian distributions.

Such curvature-based approaches offer opportunities to improve existing machine learning methods. Training modern models on large data sets is often computationally expensive. Newton-based approaches could make a valuable contribution to improving training efficiency. Curvature-based approaches are also useful in the context of approximate inference: The Laplace approximation, for instance, is used for inference in non-conjugate Gaussian processes and for quantifying uncertainty in neural network predictions.

Despite their potential, curvature-based methods are still only used to a limited extent in machine learning. This is primarily due to the fact that these algorithms require second-order derivatives, whose evaluation incurs high computational costs. In addition, the Taylor polynomial is in some cases not computable accurately, but only accessible in the form of a stochastic estimate. The overarching goal of this work is to address these limitations in order to further unlock the benefits of curvature-based methods in machine learning. Specifically, the thesis presents three contributions.

We first introduce the iterative inference method IterNCGP for non-conjugate Gaussian processes, which is based on the Laplace approximation. Our method uses "incomplete" computations—enabling inference on large data sets—but captures the resulting approximation error as an additional source of uncertainty. This allows our approach to trade off reduced computational costs for increased uncertainty. Furthermore, by reusing computationally expensive intermediate results, the inference process is accelerated significantly.

Second, we present the open-source library ViViT, which provides efficient access to curvature information in the context of neural networks. Our approach exploits the low-rank structure of the generalized Gauss-Newton matrix. This enables efficient computation of eigenvalues and eigenvectors, as well as slopes and curvatures along those directions. Unlike existing methods, our approach explicitly exposes the noise in the slope and curvature estimates and implements principled approximations that allow for a trade-off between computational cost and accuracy.

Computing the exact Taylor polynomial is prohibitively expensive for neural networks. It is therefore typically approximated on a small subset of the training data. In our third contribution, we show that this leads to systematic biases in the Taylor approximation. We provide a theoretical explanation for this phenomenon and explain its negative effects on curvature-based methods. Finally, we develop effective strategies for correcting these biases. These strategies incur negligible computational overhead, but improve the quality of the approximation significantly.

This thesis advances curvature-based methods for machine learning: Methodological innovations improve their efficiency, software implementations make them more accessible, and acknowledging and correcting inherent approximation errors increases their reliability and effectiveness. Our contributions support the broader application of curvature-based methods in machine learning research and practice.

Zusammenfassung

Das Taylorpolynom zweiter Ordnung liefert eine lokale, „schüsselförmige“ Näherung einer gegebenen Funktion. Diese Approximation ist Ausgangspunkt für *krümmungsbasierte Methoden*. Dazu zählen insbesondere (i) Optimierungsverfahren, die auf dem Newton-Schritt—dem Schritt ins Minimum der lokalen Approximation—aufbauen, sowie (ii) die Laplace-Approximation. Diese erlaubt es, beliebige Wahrscheinlichkeitsverteilungen durch einfachere Gauß-Verteilungen anzunähern.

Derartige krümmungsbasierte Ansätze eröffnen Möglichkeiten zur Verbesserung bestehender Methoden des maschinellen Lernens. Das Trainieren moderner Modelle auf großen Datensätzen ist oft mit erheblichem Rechenaufwand verbunden. Newton-Schritt basierte Ansätze könnten hier einen wertvollen Beitrag zur Effizienzsteigerung leisten. Auch im Bereich der approximativen Inferenz sind krümmungsbasierte Ansätze hilfreich: Die Laplace-Approximation wird etwa zur Inferenz in nicht-konjugierten Gaußschen Prozessen sowie zur Quantifizierung von Unsicherheit in den Vorhersagen neuronaler Netze eingesetzt.

Trotz ihres Potenzials finden krümmungsbasierte Verfahren im maschinellen Lernen bislang nur begrenzt Anwendung. Das liegt primär daran, dass diese Algorithmen zweite Ableitungen berechnen, was mit erheblichen Rechenanforderungen verbunden ist. Des Weiteren ist das Taylorpolynom teilweise nicht exakt berechenbar, sondern nur in Form einer stochastischen Näherung zugänglich. Das übergeordnete Ziel dieser Arbeit ist es, derartige Hindernisse zu adressieren, um so die Vorzüge krümmungsbasierter Methoden im maschinellen Lernen weiter zu erschließen. Konkret werden in dieser Dissertation drei Beiträge vorgestellt.

Zunächst stellen wir das iterative Inferenzverfahren IterNCGP für nicht-konjugierte Gaußsche Prozesse vor, das auf der Laplace-Approximation basiert. Unser Ansatz verwendet „unvollständige“ Berechnungen—dies ermöglicht Inferenz auf großen Datensätzen—erfasst aber den entstehenden Approximationsfehler als zusätzliche Unsicherheit. So ermöglicht es unsere Methode, reduzierten Rechenaufwand gegen Unsicherheitszuwachs abzuwägen. Durch die Wiederverwendung rechenintensiver Zwischenergebnisse wird der Inferenzprozess erheblich beschleunigt.

Zweitens präsentieren wir die Open-Source-Bibliothek ViViT, die effizienten Zugang zu Krümmungsinformation im Kontext neuronaler Netze bietet. Unser Ansatz nutzt die Niedrigrangstruktur der generalisierten Gauß-Newton Matrix aus und ermöglicht so eine effiziente Berechnung von Eigenwerten und Eigenvektoren sowie Steigungen und Krümmungen entlang dieser Richtungen. Im Gegensatz zu bestehenden Verfahren macht unser Ansatz die Streuung in den Steigungs- und Krümmungsschätzern explizit sichtbar und implementiert fundierte Approximationen, die einen Kompromiss zwischen Rechenaufwand und Genauigkeit ermöglichen.

Im Kontext von neuronalen Netzen ist es zu aufwendig, das Taylorpolynom exakt zu berechnen. Stattdessen wird es typischerweise auf einem kleinen Teildatensatz der Trainingsdaten approximiert. In unserem dritten Beitrag zeigen wir, dass dies zu einer systematischen Verzerrung der Taylor-Approximation führt. Wir liefern eine theoretische Erklärung für dieses Phänomen und erklären dessen nachteilige Auswirkungen auf krümmungsbasierte Methoden. Schließlich entwickeln wir effektive Strategien zur Behebung der Verzerrungen. Diese erzeugen vernachlässigbaren Rechenmehraufwand, verbessern die Approximationsgüte aber erheblich.

In dieser Arbeit werden krümmungsbasierte Methoden für das maschinelle Lernen weiterentwickelt: Methodische Innovationen steigern ihre Effizienz, Softwareimplementierungen machen sie breiter zugänglich, und die Anerkennung und Behebung inhärenter Approximationsfehler steigern ihre Effektivität und Zuverlässigkeit. Unsere Beiträge fördern die breitere Anwendung krümmungsbasierter Methoden in der Forschung und Praxis des maschinellen Lernens.

Table of Contents

Notation

Scalars, Vectors & Matrices

The notation is influenced by Goodfellow et al. [45].

$a \in \mathbb{R}$	A scalar
$\boldsymbol{a} \in \mathbb{R}^N$	A column vector with N entries
$\boldsymbol{A} \in \mathbb{R}^{M \times N}$	A matrix with M rows and N columns
$(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \in \mathbb{R}^{M \times N}$	A matrix whose columns are the vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n \in \mathbb{R}^M$
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)
$\mathrm{diag}(\boldsymbol{a})$	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 square matrix $\boldsymbol{A}$
$\mathrm{blockdiag}(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_L)$	A block-diagonal matrix with L square matrices $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_L$ on the diagonal
$\boldsymbol{A}^{-1}, \boldsymbol{A}^\dagger$	Inverse and Moore-Penrose pseudo-inverse of a matrix $\boldsymbol{A}$
$\mathrm{Tr}(\boldsymbol{A}), \det(\boldsymbol{A})$	Trace and determinant of a matrix $\boldsymbol{A}$
$\mathrm{rank}(\boldsymbol{A})$	Rank of a matrix $\boldsymbol{A}$, *i.e.* the number of non-zero singular values
$\boldsymbol{a} = \mathrm{vec}(\boldsymbol{A})$	Matrix $\boldsymbol{A} \in \mathbb{R}^{M \times N}$ flattened into a vector $\boldsymbol{a} \in \mathbb{R}^{MN}$ by stacking $\boldsymbol{A}$'s columns
$\mathrm{span}(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n)$	Span of the vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, *i.e.* the set of all linear combinations that can be formed from these vectors
$\|\boldsymbol{a}\|_2$ or $\|\boldsymbol{a}\|$	ℓ^2 norm/Euclidean norm of a vector $\boldsymbol{a}$, *i.e.* $\|\boldsymbol{a}\|_2^2 = \boldsymbol{a}^\top \boldsymbol{a}$
$\|\boldsymbol{a}\|_A$	$\boldsymbol{A}$-norm/Mahalanobis norm of a vector $\boldsymbol{a}$, *i.e.* $\|\boldsymbol{a}\|_2 = \sqrt{\boldsymbol{a}^\top \boldsymbol{A} \boldsymbol{a}}$, with $\boldsymbol{A}$ spd
$(\lambda_i, \boldsymbol{u}_i)$	The ith eigenpair (eigenvalue λ_i, eigenvector $\boldsymbol{u}_i$) of a matrix $\boldsymbol{A}$, *i.e.* $\boldsymbol{A}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i$
$\boldsymbol{e}_i$	The ith unit vector whose ith entry is 1 and all other entries are zero
$\boldsymbol{1}_N$	An N-dimensional vector containing ones everywhere
$\boldsymbol{I}$ or $\boldsymbol{I}_N$	Identity matrix of size $N \times N$, omitting the index $_N$ when clear from context
$\boldsymbol{A} \otimes \boldsymbol{B}$	Kronecker product of two matrices

Data

$\mathbb{X} \subseteq \mathbb{R}^D, \mathbb{Y}$	Input and output spaces, $\mathbb{Y}$ depends on the task and the encoding of the outputs
C	Number of classes/outputs in classification/multi-output regression tasks
$(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{X} \times \mathbb{Y}$	Input-output pair, typically indexed by n
$\mathbb{D}$	Set of training examples $\mathbb{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X} \times \mathbb{Y}\}_{n=1}^N$
N	Number of data in a data set or mini-batch, depending on the context
$\mathcal{D} = \{1, \ldots, N\}$	The set of training data indices
$\mathcal{B} \subset \mathcal{D}$	A set of mini-batch data indices
$\|\mathcal{B}\|$ and $\|\mathcal{D}\|$	$\|\cdot\|$ denotes the cardinality of a set, *i.e.* the number of elements
$(\boldsymbol{x}_\diamond, \boldsymbol{y}_\diamond) \in \mathbb{X} \times \mathbb{Y}$	Input-output pair from the test set

Neural Networks

The layer number is indicated by parenthesized superscripts, *e.g.* $\boldsymbol{\theta}^{(l)} \in \mathbb{R}^{P^{(l)}}$ is the parameter vector of layer l.

$\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^P$	Neural network parameter vector
P	The total number of parameters in a model
$f_{\boldsymbol{\theta}} \colon \mathbb{X} \to \mathbb{F}$	A neural network, parameterized by $\boldsymbol{\theta} \in \Theta$, typically $\mathbb{F} \subseteq \mathbb{R}^C$
L	Number of layers in the neural network

Empirical Risk & Derivatives

For the quantities below, we omit the explicit dependence on $\boldsymbol{\theta}$ and/or data index set $\mathcal{D}$ when clear from context.

$\ell \colon \mathbb{F} \times \mathbb{Y} \to \mathbb{R}$	Loss function to compare the model's prediction $f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \mathbb{F}$ to the target $\boldsymbol{y} \in \mathbb{Y}$
$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D})$	Empirical risk $\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = 1/\lvert\mathcal{D}\rvert \sum_{n \in \mathcal{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)$ on index set $\mathcal{D}$ at $\boldsymbol{\theta}$
$\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D})$	Regularized empirical risk $\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) = \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) + r(\boldsymbol{\theta})$ with $r \colon \Theta \to \mathbb{R}$
$\nabla_{\boldsymbol{x}} h(\boldsymbol{x}_0)$ or $\nabla h(\boldsymbol{x}_0)$	Gradient vector of scalar-valued function h with respect to $\boldsymbol{x}$, evaluated at $\boldsymbol{x}_0$
$\boldsymbol{J}_{\boldsymbol{x}} h(\boldsymbol{x}_0) \in \mathbb{R}^{M \times N}$	Jacobian matrix of a vector-to-vector function $h \colon \mathbb{R}^N \to \mathbb{R}^M$. Its entries are given by the partial derivatives $[\boldsymbol{J}_{\boldsymbol{x}} h(\boldsymbol{x}_0)]_{i,j} = \partial h_i(\boldsymbol{x}) / \partial x_j \rvert_{\boldsymbol{x}=\boldsymbol{x}_0}$, evaluated at $\boldsymbol{x}_0$.
$\nabla^2_{\boldsymbol{x}} h(\boldsymbol{x}_0)$ or $\nabla^2 h(\boldsymbol{x}_0)$	Hessian matrix of scalar-valued function h with respect to $\boldsymbol{x}$, evaluated at $\boldsymbol{x}_0$
$\boldsymbol{g}_{\mathcal{D}}(\boldsymbol{\theta})$	The gradient of the (regularized) empirical risk on index set $\mathcal{D}$ at $\boldsymbol{\theta}$
$\boldsymbol{H}_{\mathcal{D}}(\boldsymbol{\theta})$	The Hessian of the (regularized) empirical risk on index set $\mathcal{D}$ at $\boldsymbol{\theta}$
$\boldsymbol{G}_{\mathcal{D}}(\boldsymbol{\theta})$	Generalized Gauss-Newton matrix of the empirical risk on index set $\mathcal{D}$ at $\boldsymbol{\theta}$
$\boldsymbol{F}_{\mathcal{D}}(\boldsymbol{\theta})$	Fisher information matrix on index set $\mathcal{D}$ at $\boldsymbol{\theta}$
$\boldsymbol{K}_{\mathcal{D}}(\boldsymbol{\theta})$	K-FAC curvature matrix on index set $\mathcal{D}$ at $\boldsymbol{\theta}$
$q_{\mathcal{D}}(\boldsymbol{\theta})$	Quadratic approximation of the (regularized) empirical risk on index set $\mathcal{D}$ at $\boldsymbol{\theta}$, around an anchor point $\boldsymbol{\theta}_0 \in \Theta$
$\boldsymbol{\theta}_\star$	The maximum a posteriori (MAP) estimate of the parameters

Distributions & Gaussian Processes

$\triangle^{C-1} \subset \mathbb{R}^C$	The probability simplex $\triangle^{C-1} \coloneqq \{\boldsymbol{p} \in \mathbb{R}^C \mid p_c \geq 0, \sum_{c=1}^C p_c = 1\}$
$\mathbb{E}[\mathbf{x}]$ or $\mathbb{E}_p[\mathbf{x}]$	Expectation of $\mathbf{x}$ under the probability distribution p
$\text{Var}[\mathbf{x}]$ or $\text{Var}_p[\mathbf{x}]$	Variance of $\mathbf{x}$ under the probability distribution p
$\text{Cov}[\mathbf{x}, \mathbf{y}]$ or $\text{Cov}_p[\mathbf{x}, \mathbf{y}]$	Covariance of $\mathbf{x}$ and $\mathbf{y}$ under the joint probability distribution p
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	Multi-variate Gaussian distribution with mean vector $\boldsymbol{\mu} \in \mathbb{R}^N$, spd covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{N \times N}$, and density $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-N/2} \det(\boldsymbol{\Sigma})^{-1/2} \exp(-1/2 (\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}))$
$\text{Cat}(y, \boldsymbol{p}) = p_y$	The PMF of the categorical distribution, where $\boldsymbol{p} \in \triangle^{C-1}$ and $y \in \{1, \dots, C\}$
$\mathcal{GP}(m, K)$	(Multi-output) Gaussian process with mean function $m \colon \mathbb{X} \to \mathbb{R}^C$ and kernel function $K \colon \mathbb{X} \times \mathbb{X} \to \mathbb{R}^{C \times C}$

Miscellaneous

$\overset{c}{=}$	Equality up to an additive constant
$\mathcal{O}(\cdot)$	Big-O notation, informally $f(x) \in \mathcal{O}(g(x))$ if $f(x)$ is bounded by a constant multiple of $g(x)$ for all sufficiently large values of x

Acronyms & Abbreviations

2D, 3D	Two-dimensional, three-dimensional
AD	Automatic differentiation
CG	Conjugate gradients
CNN	Convolutional neural network
CPU	Central processing unit
ECE	Expected calibration error
E.g. or *e.g.*	For example (*exempli gratia*)
ELBO	Evidence lower bound
FIM	Fisher information matrix
GGN	Generalized Gauss-Newton (matrix)
GP	Gaussian process
GPU	Graphics processing unit
I.e. or *i.e.*	That is (*id est*)
I.i.d. or *i.i.d.*	Independent and identically distributed
K-FAC	Kronecker-factored approximate curvature
KL divergence	Kullback-Leibler divergence
LA	Laplace approximation
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno (optimization algorithm)
MAP	Maximum a posteriori
MC	Monte Carlo
ML	Machine learning
MSE	Mean squared error
NCGP	Non-conjugate Gaussian process
NLL	Negative log-likelihood
NTK	Neural tangent kernel
PDF	Probability density function
PLS	Probabilistic linear solver
ResNet	Residual neural network
SNR	Signal-to-noise ratio
SVD	Singular value decomposition
spd	Symmetric positive definite
W.r.t. or *w.r.t.*	With respect to

1 Introduction

Machine learning models have achieved astonishing performance on a diverse set of tasks like image classification [54, 79], playing games [127], natural language processing [13], and protein structure prediction [68]. In this manuscript, we focus on two prominent classes of supervised machine learning models: Gaussian processes and neural networks. Rather than following explicit, hand-crafted rules, these models *learn* latent functional relationships directly from the data.

Second-Order Optimizers. The learning process of these machine learning models can be framed as an optimization problem—optimization thus lies at the very heart of machine learning. As models become more complex and data sets expand in size, this optimization process becomes more and more computationally intensive, requiring a considerable amount of time, energy and financial resources. The availability of *efficient* optimization algorithms is thus of crucial importance. For training neural networks, first-order methods—which rely primarily on the objective function's gradient—are the workhorse of model training due to their relative simplicity and scalability. To further enhance these methods, richer statistical or geometric information can be incorporated. For instance, curvature information has recently become more readily available in deep learning frameworks [26, 28, 106]. Curvature crucially provides access to a quadratic Taylor approximation of the objective function, which motivates a class of *second-order* optimizers that are *conceptionally* different from and potentially more powerful than customary first-order methods.

Uncertainty Quantification via the Laplace Approximation. As machine learning models are increasingly deployed in high-stakes domains such as autonomous driving, medical diagnostics, and judicial decision support, it is crucial that these models express not just their predictions, but also their confidence—or lack thereof—in those predictions. A simple and elegant way to quantify this uncertainty is the Laplace approximation [83, 90]. It is based on the idea of approximating the log-posterior distribution of a Bayesian model with a second-order Taylor expansion around the mode, which effectively turns the posterior into a Gaussian. The Laplace approximation is used, for instance, for approximate inference in non-conjugate Gaussian processes [114] and for equipping neural networks with predictive uncertainty [30, 77, 115].

Thesis Contributions. Second-order optimizers and the Laplace

approximation *both* fundamentally rely on a quadratic Taylor expansion and thus require access to *curvature* information. However, processing this curvature information presents significant challenges: The size of modern machine learning models combined with massive data sets typically incurs substantial computational costs in terms of memory and run time. In deep learning, additional complications arise, such as the stochasticity in the curvature estimate introduced by mini-batching. The overarching goal of this thesis is to address these challenges. Specifically, we aim to make curvature-based methods more *efficient* through methodological innovations, more *accessible* via software implementations, and more *effective* and *reliable* by acknowledging and addressing inherent approximation errors—ultimately advancing their wider application in machine learning research and practice.

Thesis Outline. This manuscript contains four parts:

- Part I establishes the foundational concepts necessary for understanding our research contributions. In Chapter 2, we examine the two primary machine learning models considered in this thesis: Gaussian processes and deep neural networks. Chapter 3 introduces curvature-based methods, including second-order optimization and the Laplace approximation. Finally, Chapter 4 discusses the specific challenges that arise when applying those methods to machine learning problems and outlines how our research contributions in Part II address these limitations.
- Part II constitutes the core of the thesis, presenting the research contributions across Chapters 5 to 7, each corresponding to a peer-reviewed scientific publication.
- Part III concludes with a summary of our findings and explores promising directions for future research.
- Part IV provides supplementary material for Part II: It includes mathematical derivations, implementation and experimental details as well as additional empirical results.

Part I

Background & Thesis Contributions

2 Machine Learning Models for Supervised Learning

Machine learning encompasses various learning paradigms, including supervised, unsupervised, and reinforcement learning [9, Sec. 1]. This thesis is concerned with the supervised setting.

Supervised Learning. In supervised learning, the goal is to learn a functional relationship between input data $\boldsymbol{x} \in \mathbb{X} \subseteq \mathbb{R}^D$ and corresponding outputs $\boldsymbol{y} \in \mathbb{Y}$ using a set of training examples $\mathbb{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X} \times \mathbb{Y}\}_{n=1}^N$. Two common supervised learning tasks are *regression* and *classification*, which differ in the output space $\mathbb{Y}$: For regression, the outputs are continuous (*i.e.* $\mathbb{Y} \subseteq \mathbb{R}$ or, more generally, $\mathbb{Y} \subseteq \mathbb{R}^C$ in multi-output regression), while for classification tasks, the output space is discrete. For instance, the output space is typically represented as $\mathbb{Y} = \{-1, 1\}$ in binary classification and as $\mathbb{Y} = \{1, \dots, C\}$ in multi-class classification.

Figure 2.1 shows a regression and a classification problem. The training data for these toy examples was generated using a process that *we* specified. In practice, however, this process is unknown. To make meaningful predictions for new inputs, we must therefore *learn* the underlying relationship between inputs and outputs from the training data. Below, we introduce Gaussian processes (an instance of a non-parametric model, see Section 2.1) and neural networks (an instance of a parametric model, see Section 2.2). Both models define a class of functions for the dependency between inputs and outputs, and need to be fitted to the training data.

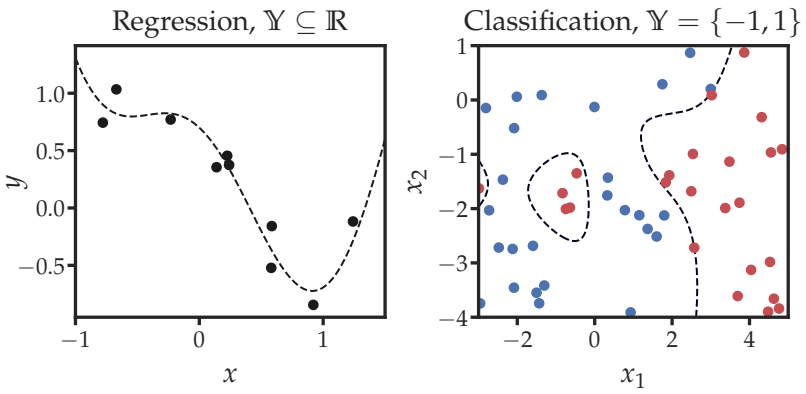


Figure 2.1: Supervised Learning Tasks. For one-dimensional regression *(Left)*, the goal is to predict a continuous output $y \in \mathbb{Y} \subseteq \mathbb{R}$ at $x \in \mathbb{R}$. The training data $\{(x_n, y_n)\}$ (●) were generated by the function $x^2 - x + 0.7\cos(3x)$ (shown as - -) with some additive noise. The right subplot shows inputs $\boldsymbol{x}_n \in \mathbb{X} \subseteq \mathbb{R}^2$ that belong to one of two classes $y_n \in \mathbb{Y} = \{-1, 1\}$ (●/●) and the decision boundary (- -) of the underlying data-generating process.

Overfitting & Generalization. A model that simply memorizes (or *overfits*) the training data without capturing the true underlying relationship between inputs and outputs is typically not useful in practice. Instead, we aim to learn a model that can make meaningful predictions on *unseen* test data (*i.e.* data that is *not* included in

the training set $\mathbb{D}$). This ability is referred to as *generalization*. Generalization relies on an *inductive bias*—a model assumption that constrains the space of possible outcomes in favor of those that are expected to generalize well.

2.1 Gaussian Processes & Bayesian Inference

Overview. Gaussian processes (GPs) provide a flexible framework for supervised learning tasks like regression and classification. They define a *distribution over functions*, *i.e.* they inherently provide an uncertainty estimate for the model's predictions. Such a *probabilistic* approach is essential when critical decisions must be made based on limited information, *e.g.* in medical diagnosis, robotics, or public policy. Below, we define what a GP is, explain the role of the kernel function, and introduce GP regression. This will serve as a basis for the more general setting of non-conjugate GPs that we will introduce in Chapter 5. For a thorough introduction to Gaussian processes, we refer the reader to Rasmussen and Williams [114].

Definition 2.1 (Gaussian Process) A Gaussian process is a collection of random variables, any finite subset of which follows a multivariate Gaussian distribution [114, Def. 2.1].

Notation. A Gaussian process $\mathrm{f} \sim \mathcal{GP}(m, K)$ is specified by its *mean* function $m\colon \mathbb{X} \to \mathbb{R}$ and *covariance* or *kernel* function $K\colon \mathbb{X} \times \mathbb{X} \to \mathbb{R}$. They encode our belief about the unknown underlying functional relationship f between inputs and outputs: The mean function $m(\boldsymbol{x})$ specifies the expected value of $\mathrm{f}(\boldsymbol{x})$ at input $\boldsymbol{x}$, while the kernel function $K(\boldsymbol{x}, \boldsymbol{x}')$ quantifies the covariance between the function values at inputs $\boldsymbol{x}$ and $\boldsymbol{x}'$, *i.e.*

$$m(\boldsymbol{x}) = \mathbb{E}[\mathrm{f}(\boldsymbol{x})] \quad \text{and} \quad K(\boldsymbol{x}, \boldsymbol{x}') = \mathrm{Cov}[\mathrm{f}(\boldsymbol{x}), \mathrm{f}(\boldsymbol{x}')].$$

By Definition 2.1, evaluating the Gaussian process $\mathrm{f} \sim \mathcal{GP}(m, K)$ at a set of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{X} \subseteq \mathbb{R}^D$ yields a multivariate Gaussian distribution

$$\mathbf{f} \coloneqq \begin{pmatrix} \mathrm{f}(\boldsymbol{x}_1) \\ \vdots \\ \mathrm{f}(\boldsymbol{x}_N) \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} m(\boldsymbol{x}_1) \\ \vdots \\ m(\boldsymbol{x}_N) \end{pmatrix}, \begin{pmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \ddots & \vdots \\ K(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & K(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix} \right),$$

which we write in short as $\mathbf{f} \sim \mathcal{N}(\boldsymbol{m}, \boldsymbol{K})$, where $\boldsymbol{m} = m(\boldsymbol{X}) \in \mathbb{R}^N$ is the mean vector and $\boldsymbol{K} = K(\boldsymbol{X}, \boldsymbol{X}) \in \mathbb{R}^{N \times N}$ is the covariance or kernel matrix.[1]

Kernel Functions. The kernel function is a crucial component of the GP model as it encodes assumptions about the properties of the

1: We adopt the notation from [114] and use $m(\boldsymbol{X}) \in \mathbb{R}^N$ and $\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X}') \in \mathbb{R}^{N \times M}$ to denote the evaluation of m and K on the *stacked* inputs $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)^\top \in \mathbb{R}^{N \times D}$ and $\boldsymbol{X}' = (\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_M)^\top \in \mathbb{R}^{M \times D}$, *i.e.*

$$[m(\boldsymbol{X})]_i = m(\boldsymbol{x}_i) \quad \text{and}$$
$$[\boldsymbol{K}(\boldsymbol{X}, \boldsymbol{X}')]_{i,j} = K(\boldsymbol{x}_i, \boldsymbol{x}'_j).$$

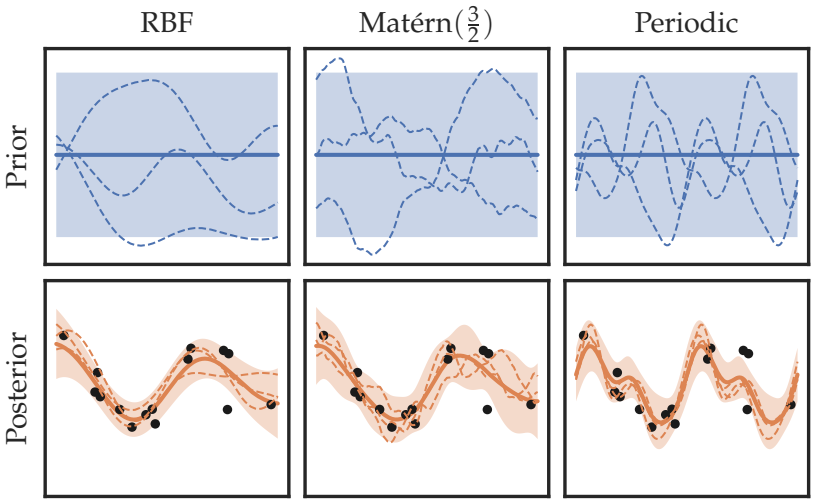


Figure 2.2: GP Regression with Different Kernels. *(Top)* The prior mean $m(\boldsymbol{x}) \equiv 0$ (shown as —), prior covariance $K(\boldsymbol{x}, \boldsymbol{x})$ (the shaded area ■ covers two standard deviations around the mean), and three samples from the GP prior (shown as - -) for three different kernels from *(Left)* to *(Right)*. *(Bottom)* The bottom panel shows the GP posterior mean m_{post} as —, posterior covariance $K_{\text{post}}(\boldsymbol{x}, \boldsymbol{x})$ as ■ (see Equations (2.3) and (2.4)), three samples as - - and the training data as ●.

function being modeled, such as smoothness or periodicity. These functions often have *hyperparameters* (*e.g.* a characteristic length scale) that can be adjusted to match the modeling assumptions. To be a valid covariance function, the kernel function must be symmetric and positive semidefinite, *i.e.* it must give rise to a positive semidefinite kernel matrix for any collection of inputs. Popular kernel functions include the *radial basis function* (RBF) kernel (for modelling smooth functions), the *Matérn* kernel (for controllable smoothness), and the *periodic* kernel (for periodic functions).[2] The top panel in Figure 2.2 shows samples from $\mathcal{GP}(0, K)$ for these three kernel functions. Kernel functions can also be combined via addition or multiplication to implement more complex dependencies between function values [114, Sec. 4.2.4].

2: For a comprehensive and more precise overview of kernel functions and their properties, we refer the reader to [114, Sec. 4.2].

Gaussian Process Regression. In GP regression, our goal is to *infer* an unknown function $f : \mathbb{X} \to \mathbb{R}$, $\mathbb{X} \subseteq \mathbb{R}^D$ from a set of input-output pairs. We start with the *prior* belief—the belief over f *before* observing any data. In GP regression, this prior belief is modelled with a GP $\mathrm{f} \sim \mathcal{GP}(m, K)$. Assume that we are given a set of training locations $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)^\top \in \mathbb{R}^{N \times D}$ for which we have observed the corresponding real-valued outputs $\boldsymbol{y} = (y_1, \ldots, y_N)^\top \in \mathbb{R}^N$ and a test location $\boldsymbol{x}_\diamond$ with unknown output. By Definition 2.1, the prior GP implies a prior Gaussian belief over the latent function values $\mathbf{f} = (\mathrm{f}(\boldsymbol{x}_1), \ldots, \mathrm{f}(\boldsymbol{x}_N))^\top \in \mathbb{R}^N$ and $\mathrm{f}_\diamond := \mathrm{f}(\boldsymbol{x}_\diamond) \in \mathbb{R}$:

$$\begin{pmatrix} \mathbf{f} \\ \mathrm{f}_\diamond \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} m(\boldsymbol{X}) \\ m(\boldsymbol{x}_\diamond) \end{pmatrix}, \begin{pmatrix} \boldsymbol{K} & K(\boldsymbol{X}, \boldsymbol{x}_\diamond) \\ K(\boldsymbol{x}_\diamond, \boldsymbol{X}) & K(\boldsymbol{x}_\diamond, \boldsymbol{x}_\diamond) \end{pmatrix} \right). \tag{2.1}$$

Next, we assume that the observed outputs $\boldsymbol{y}$ are generated by evaluating the latent function f at the training locations and adding *i.i.d.* Gaussian noise, *i.e.* $y_n = f(\boldsymbol{x}_n) + \epsilon_n$ with $\epsilon_n \sim \mathcal{N}(0, \sigma^2)$. We

can write the noise-term $\epsilon = (\epsilon_1, \dots, \epsilon_N) \in \mathbb{R}^N$ as

$$\begin{pmatrix} \epsilon \\ 0 \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mathbf{0} \\ 0 \end{pmatrix}, \begin{pmatrix} \sigma^2 \boldsymbol{I} & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix} \right). \tag{2.2}$$

As the Gaussian random variables in Equations (2.1) and (2.2) are independent, their sum is also Gaussian, with mean and covariance given by the sum of the individual means and covariances. The first component of the resulting random vector $\mathbf{f}+\epsilon$ is a random variable $\mathbf{y}$ that represents our prior belief over the regression targets. We thus arrive at the following joint prior Gaussian distribution

$$\begin{pmatrix} \mathbf{y} \\ \mathrm{f}_\diamond \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} m(\boldsymbol{X}) \\ m(\boldsymbol{x}_\diamond) \end{pmatrix}, \begin{pmatrix} \boldsymbol{K} + \sigma^2 \boldsymbol{I} & K(\boldsymbol{X}, \boldsymbol{x}_\diamond) \\ K(\boldsymbol{x}_\diamond, \boldsymbol{X}) & K(\boldsymbol{x}_\diamond, \boldsymbol{x}_\diamond) \end{pmatrix} \right).$$

We obtain the *posterior* distribution—the distribution *after* observing the training data—by conditioning on the observed training outputs $\mathbf{y} = \boldsymbol{y}$. For Gaussians, this can be done in closed form (see *e.g.* [9, Sec. 2.3.1]). We obtain $p(\mathrm{f}_\diamond \mid \mathbf{y} = \boldsymbol{y}) = \mathcal{N}(m_\diamond, K_\diamond)$, where

$$\begin{aligned} m_\diamond &:= m(\boldsymbol{x}_\diamond) + K(\boldsymbol{x}_\diamond, \boldsymbol{X})(\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1}(\boldsymbol{y} - m(\boldsymbol{X})), \\ K_\diamond &:= K(\boldsymbol{x}_\diamond, \boldsymbol{x}_\diamond) - K(\boldsymbol{x}_\diamond, \boldsymbol{X})(\boldsymbol{K} + \sigma^2 \boldsymbol{I})^{-1} K(\boldsymbol{X}, \boldsymbol{x}_\diamond). \end{aligned}$$

As the derivation above is valid for any finite set of test inputs, there is an associated posterior GP $\mathcal{GP}(m_{\text{post}}, K_{\text{post}})$ with mean and kernel function

$$m_{\text{post}}(\boldsymbol{x}) = m(\boldsymbol{x}) + K(\boldsymbol{x}, \boldsymbol{X})\hat{\boldsymbol{K}}^{-1}(\boldsymbol{y} - m(\boldsymbol{X})), \tag{2.3}$$

$$K_{\text{post}}(\boldsymbol{x}, \boldsymbol{x}') = K(\boldsymbol{x}, \boldsymbol{x}') - K(\boldsymbol{x}, \boldsymbol{X})\hat{\boldsymbol{K}}^{-1} K(\boldsymbol{X}, \boldsymbol{x}'), \tag{2.4}$$

where $\hat{\boldsymbol{K}} := \boldsymbol{K} + \sigma^2 \boldsymbol{I}$. The bottom panel of Figure 2.2 shows training data (as ●) and the resulting posterior GPs for the three different kernel functions (from *(Left)* to *(Right)*). The posteriors attempt to fit the training data under the constraints imposed by the kernel function. The reason why the uncertainty does not contract to zero at the observed data points is that our observation model contains noise (denoted by ϵ above).

Scalability of Gaussian Process Regression. Equations (2.3) and (2.4) each require solving a system of linear equations with symmetric positive definite (spd) coefficient matrix $\hat{\boldsymbol{K}} \in \mathbb{R}^{N \times N}$. The textbook implementation (see *e.g.* [114, Alg. 2.1]) uses a Cholesky decomposition $\hat{\boldsymbol{K}} = \boldsymbol{L}\boldsymbol{L}^\top$, where $\boldsymbol{L}$ is a lower triangular matrix. The linear systems can then be solved efficiently for any right-hand side by forward and backward substitution. However, the quadratic memory requirements and cubic run time complexity of the Cholesky decomposition are prohibitive for large N. Thus, many methods have been developed to scale GP regression to large data sets, including variational approaches like [59] (this

approach is based on a set of inducing points that serves as a compressed representation of the training data) or matrix-free low-rank approaches (*e.g.* [143]).

Bayesian Inference. GP regression is an instance of Bayesian inference: We start with a *prior* belief over the latent function values, define a *likelihood* that models how the observed data is generated from these latent values, and obtain the *posterior* distribution by conditioning on the observed data—this process is formalized in Bayes' theorem. GP regression assumes a GP prior and a Gaussian likelihood for which the posterior GP is available in closed form (and reduces to linear algebra, specifically the solution of a linear system, see Equations (2.3) and (2.4)). In Chapter 5, we derive an efficient inference method for the non-conjugate case, where the likelihood is *not* Gaussian and the posterior is therefore no longer available in closed form.

2.2 Neural Networks & Deep Learning

Neural networks form a powerful class of models capable of learning complex patterns from data. Their structure is loosely inspired by the information processing mechanisms in the human brain, *i.e.* a set of interconnected neurons—hence the term *neural networks*. These models are widely used in practice and have achieved state-of-the-art performance in various domains like computer vision [54, 79] and natural language processing [13].

Overview. We introduced supervised learning as the task of learning the underlying functional relationship between inputs $\boldsymbol{x} \in \mathbb{X}$ and outputs $\boldsymbol{y} \in \mathbb{Y}$. A neural network (see Section 2.2.1) is a *parameterized function* $f_{\boldsymbol{\theta}}$ that maps between those spaces.[3] The parameters $\boldsymbol{\theta} \in \mathbb{\Theta} \subseteq \mathbb{R}^P$ determine the behavior of the network and need to be adjusted for accurate predictions. The problem of finding "good" parameters can be formulated as an optimization problem (see Section 2.2.2): We want to choose the parameters such that the network's predictions $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ are as "close" as possible to the true training labels $\boldsymbol{y}_n$ for all training examples $(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{D}$. This optimization problem has an interpretation as maximum a posteriori (MAP) inference in a Bayesian treatment of the parameters (see Section 2.2.3). The prerequisite for developing optimizers for this problem is the ability to compute gradients of the objective function with respect to the network's parameters. We thus explain the core concepts of automatic differentiation (AD) in Section 2.2.4 and introduce common gradient-based optimizers in Section 2.2.5.

3: More precisely, the output space of the neural network is an "intermediate" space $\mathbb{F}$ that can differ from $\mathbb{Y}$. One example is multi-class classification, where the output space of the model is typically $\mathbb{F} \subseteq \mathbb{R}^C$, whereas the labels are discrete, *i.e.* $\mathbb{Y} = \{1, \dots, C\}$.

2.2.1 Neural Networks

Feature Hierarchies Through Layer Composition. With a neural network, we can model complicated input-output relationships[4] by using a composition (see Equation (2.5)) of simple functions called *layers*. Each layer serves a distinct purpose in transforming the incoming data. By stacking more and more of these layers, we obtain a *deep* neural network. The idea behind deep learning is that the network can learn a *hierarchy* of features: Lower layers capture simple patterns (*e.g.* edges in an image), while higher layers learn more abstract features (*e.g.* object shapes). Software libraries like TensorFlow [1], PyTorch [111], or JAX [12] enable users to easily combine these building blocks and build complex neural network architectures tailored to their specific needs.

4: There is a vast body of literature on the expressiveness and approximation properties of neural network architectures, *e.g.* [22, 62, 63, 136].

Sequential Feedforward Neural Networks. In this manuscript, we focus on sequential feedforward neural networks, which encompass many widely used architectures like the multilayer perceptron (MLP), convolutional networks (CNN) [84] and residual networks (ResNet) [54]. For a comprehensive introduction to neural networks, we refer the reader to Goodfellow et al. [45].

Definition 2.2 (Sequential Feedforward Neural Network) A sequential feedforward neural network $f_{\boldsymbol{\theta}} : \mathbb{X} \to \mathbb{F}$ transforms an input datum $\boldsymbol{x} \in \mathbb{X} \subseteq \mathbb{R}^D$ into an output $\boldsymbol{f} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \mathbb{F} \subseteq \mathbb{R}^C$ using a sequence of parameterized functions $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$ that are called *modules* or *layers*, *i.e.*

$$f_{\boldsymbol{\theta}} = f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ \ldots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}}, \tag{2.5}$$

where $\boldsymbol{\theta}^{(l)} \in \mathbb{R}^{P^{(l)}}$ is the parameter vector of the l-th layer and

$$\boldsymbol{\theta} = \begin{pmatrix} \boldsymbol{\theta}^{(1)} \\ \vdots \\ \boldsymbol{\theta}^{(L)} \end{pmatrix} \in \Theta \subseteq \mathbb{R}^P$$

is the layer-wise concatenation of these vectors. L is the number of layers and is also referred to as the network's *depth*. $P^{(l)}$ is the number of parameters in the l-th layer and is also called the layer's *width*; and $P = \sum_{l=1}^{L} P^{(l)}$ is the total number of parameters in the network.

For computing the network's output $\boldsymbol{f} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ (this is called a *forward pass*), we propagate the input $\boldsymbol{x}$ through the network layer by layer, *i.e.*

$$\begin{aligned} \boldsymbol{z}^{(0)} &:= \boldsymbol{x} \\ \boldsymbol{z}^{(l)} &:= f^{(l)}_{\boldsymbol{\theta}^{(l)}}(\boldsymbol{z}^{(l-1)}) \quad \text{for } l = 1, \ldots, L \end{aligned} \tag{2.6}$$

$$z^{(L)} = f.$$

This type of architecture is called a *feedforward* neural network because the data flows in one direction from the input layer through the intermediate *hidden* layers to the output layer without any backward connections.

Common Building Blocks. Here, we introduce some commonly-used neural network layer types, *i.e.* instances of $f^{(l)}_{\theta^{(l)}}$ in Definition 2.2.[5] For brevity, we omit the layer index for the parameters (*i.e.* we use θ instead of $\theta^{(l)}$) and use $z_{\text{in}} \in \mathbb{R}^I$ and $z_{\text{out}} \in \mathbb{R}^O$ to denote the input and output of a layer, respectively.

5: The notion of a layer is quite vague. Sometimes a single item from the list of building blocks below is considered a layer, whereas for larger architectures, it is more convenient to group multiple of these building blocks into one layer.

- **Dense (Fully-Connected) Linear Layer.** This module implements an affine linear transformation to the input, *i.e.*

  $$z_{\text{out}} = W z_{\text{in}} + b,$$

  where $W \in \mathbb{R}^{O \times I}$ is the weight matrix and $b \in \mathbb{R}^O$ is the bias vector. The trainable parameters of that layer are given by the concatenation of the flattened weight matrix and bias vector $\theta = (\text{vec}(W)^\top, b^\top)^\top$.
- **Convolutional Layer.** Convolutional layers leverage the spatial structure of the input data, making them particularly effective for image processing. These layers extract features by applying a set of convolution kernels, each of which is represented by a small square matrix. When a kernel is applied to an image, it generates a new image (called a *feature map*) where each pixel value is given by a weighted sum (the weights are defined by the kernel) of *neighboring* pixels in the input image. This operation enables the extraction of meaningful spatial features such as edges, textures, and other patterns from the input. Importantly, the entries of these kernel matrices are treated as trainable parameters that are optimized during the training process. By using multiple kernels within a single layer, the network learns to detect a diverse set of features simultaneously, creating rich representations of the input data that can be further processed by subsequent layers.
- **Pooling Layer.** Pooling is a downsampling operation: It splits the input image into windows and aggregates the values within each window into a single number. Common variants include *max pooling* [150], which extracts the maximum value from each window, and *average pooling* [85], which takes the average. This operation serves multiple purposes: It reduces computational complexity by decreasing feature map dimensions and helps with robustness to small translations in the input data (by focusing on feature *presence* rather than

exact spatial location). Unlike convolutional layers, pooling operations contain no trainable parameters.

- **Activation Layer.** This module is another example of a layer without trainable parameters (with few exceptions, *e.g.* PReLU [53]). An activation layer applies a transformation element-wise to all inputs, *i.e.*

$$[\boldsymbol{z}_{\text{out}}]_i = \sigma([\boldsymbol{z}_{\text{in}}]_i) \quad \text{for } i = 1, \dots, I.$$

The activation function $\sigma\colon \mathbb{R} \to \mathbb{R}$ is non-linear, which allows the network to model more complex relationships in the data. The most common activation function is the rectified linear unit (ReLU) $\sigma(z) = \max(0, z)$.

The list of modules is not exhaustive. Other common building blocks of neural networks are dropout layers, recurrent layers, normalization layers, skip connections, transformer layers, *etc.*[6]

6: A comprehensive list of such modules can be found in PyTorch's `torch.nn` documentation at https://pytorch.org/docs/stable/nn.html.

2.2.2 Empirical Risk Minimization

In the previous section, we introduced the feedforward neural network as a parameterized function $f_{\boldsymbol{\theta}}\colon \mathbb{X} \to \mathbb{F}$ with trainable parameters $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^P$. In order to tune the network's parameters, we must first define an objective function that quantifies the model's performance. This leads to the empirical risk, which we introduce in the following. The subsequent Sections 2.2.4 and 2.2.5 discuss how this minimization problem is solved in practice using automatic differentiation and specialized numerical optimization methods.

Loss Functions. Given an input $\boldsymbol{x}$, we want the network's prediction to be as "close" as possible to the ground truth $\boldsymbol{y}$. To measure "closeness", we need to specify a *loss function* $\ell\colon \mathbb{F} \times \mathbb{Y} \to \mathbb{R}$ that quantifies the dissimilarity between the predicted output $\boldsymbol{f} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \mathbb{F}$ and the true label $\boldsymbol{y} \in \mathbb{Y}$. In the multi-output regression context with $\mathbb{F} = \mathbb{Y} \subseteq \mathbb{R}^C$, a common choice is the mean squared error (MSE) loss

$$\ell(\boldsymbol{f}, \boldsymbol{y}) = \frac{1}{C} \sum_{c=1}^{C} (f_c - y_c)^2 = \frac{1}{C} \|\boldsymbol{f} - \boldsymbol{y}\|^2. \tag{2.7}$$

Multi-class classification tasks with $\mathbb{F} \subseteq \mathbb{R}^C$, $\mathbb{Y} = \{1, \dots, C\}$ often use the softmax cross-entropy loss

$$\ell(\boldsymbol{f}, y) = -\log([\text{softmax}(\boldsymbol{f})]_y) = -\log\left(\frac{\exp(f_y)}{\sum_{c=1}^{C} \exp(f_c)}\right), \tag{2.8}$$

where softmax: $\mathbb{R}^C \to \mathbb{\Delta}^{C-1}$ is defined as

$$\text{softmax}(\boldsymbol{f}) \coloneqq \left(\frac{\exp(f_1)}{\sum_{c=1}^{C} \exp(f_c)}, \dots, \frac{\exp(f_C)}{\sum_{c=1}^{C} \exp(f_c)} \right).$$

The softmax function maps the network's output $\boldsymbol{f}$ to the probability simplex $\mathbb{\Delta}^{C-1} \subset \mathbb{R}^C$, *i.e.* a vector of non-negative real numbers that sum to one. The output can thus be interpreted as the probabilities the network assigns to each class. The cross-entropy loss extracts the probability assigned to the true class y and penalizes if this probability is low.

Expected & Empirical Risk. Assume that the training data is sampled *i.i.d.* from a joint distribution $p(\boldsymbol{x}, \boldsymbol{y})$ over the input-output pairs $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{X} \times \mathbb{Y}$. The *expected risk* is defined as the expected loss under this data-generating distribution, *i.e.*

$$\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p(\boldsymbol{x},\boldsymbol{y})}[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})]. \tag{2.9}$$

Since the loss is a measure of *dissimilarity* between the network's predictions and the true labels, the model's parameters should be chosen such that they *minimize* this performance metric, *i.e.* $\boldsymbol{\theta}_\star = \arg\min_{\boldsymbol{\theta}\in\mathbb{\Theta}} \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p(\boldsymbol{x},\boldsymbol{y})}[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})]$. However, since the data-generating distribution is inaccessible in practice, we have to rely on an approximation thereof. By using the Monte Carlo (MC) estimator of the expectation based on the set of *i.i.d.* training examples $\mathbb{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X} \times \mathbb{Y}\}_{n=1}^{N}$, we obtain the *empirical risk*

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) \coloneqq \frac{1}{|\mathcal{D}|} \sum_{n\in\mathcal{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n). \tag{2.10}$$

Here, $\mathcal{D} = \{1, \dots, N\}$ denotes the indices of the training data.[7]

7: We choose this notation via the index set since it is a bit more compact compared to

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

Regularization. In practice, it is common to augment the empirical risk with a regularizer $r\colon \mathbb{\Theta} \to \mathbb{R}$ to prevent overfitting.[8] An example is the ℓ^2-regularizer $r(\boldsymbol{\theta}) \coloneqq \beta/2 \, \|\boldsymbol{\theta}\|^2$ with $\beta \in \mathbb{R}_{\geq 0}$. The underlying assumption is that overfitting is caused by overly large parameter values. By penalizing the parameter magnitudes, we thus expect a more robust model that generalizes better to unseen data [80].

8: There are many other strategies that address the issue of overfitting such as dropout layers, or data augmentation techniques.

Objective for Training Neural Networks. We want the model to (i) make accurate predictions on the training data and (ii) generalize well to unseen data. We thus use the *regularized* empirical risk (or *regularized loss*) as objective for training a neural network, *i.e.* we aim to find

$$\boldsymbol{\theta}_\star \coloneqq \arg\min_{\boldsymbol{\theta}\in\mathbb{\Theta}} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}, \mathcal{D}) \quad \text{where} \quad \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}, \mathcal{D}) \coloneqq \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) + r(\boldsymbol{\theta}). \tag{2.11}$$

2.2.3 Bayesian Neural Networks

Probabilistic Interpretation of the Regularized Empirical Risk. In many cases, the regularized empirical risk $\mathcal{L}_{\text{reg}}$ from Equation (2.11) can be interpreted as the log-posterior of a specific probabilistic model. One example is the softmax cross-entropy loss (see Equation (2.8)) with an ℓ^2-regularizer, as we will see in the following.[9] Let $\text{Cat}(y, \boldsymbol{p}) = p_y$ denote the probability mass function of the categorical distribution, where $\boldsymbol{p} \in \mathbb{\Delta}^{C-1}$ is a vector of probabilities (*i.e.* $p_c \geq 0$ and $\sum_{c=1}^{C} p_c = 1$), and $y \in \{1, \dots, C\}$. The regularized loss can be rewritten as

$$
\begin{aligned}
N \cdot \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) &\overset{(2.8)}{=} -\sum_{n \in \mathcal{D}} \log([\text{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))]_{y_n}) + \frac{N\beta}{2}\|\boldsymbol{\theta}\|_2^2 \\
&= -\sum_{n \in \mathcal{D}} \log\big(\text{Cat}\big(y_n, \text{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\big)\big) \\
&\quad - \left(-\frac{1}{2}\boldsymbol{\theta}^\top (N\beta \cdot \boldsymbol{I})\, \boldsymbol{\theta}\right) \\
&= -\log \Bigg(\underbrace{\prod_{n \in \mathcal{D}} \text{Cat}\big(y_n, \text{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\big)}_{\text{likelihood } p(\mathbb{D} \mid \boldsymbol{\theta})} \Bigg) \\
&\quad - \log \Bigg(\underbrace{\mathcal{N}\left(\boldsymbol{\theta}; \boldsymbol{0}, \frac{1}{N\beta}\boldsymbol{I}\right)}_{\text{prior } p(\boldsymbol{\theta})} \Bigg) - Z,
\end{aligned}
$$

where $Z := {}^{P}/_{2} \log({}^{2\pi}/_{N\beta})$ absorbs the normalization constant of the Gaussian prior.

Neural Network Training as MAP Estimation. As the derivation above shows, the empirical risk $\mathcal{L}$ is connected to the negative log categorical likelihood, and the regularizer r takes the role of a negative log Gaussian prior over the parameters (note that it does not depend on the data). The (rescaled) regularized loss function $\mathcal{L}_{\text{reg}}$ can thus be interpreted as the negative unnormalized log-posterior of a Bayesian model[10]

$$
N \cdot \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) \overset{\text{c}}{=} -\log p(\mathbb{D} \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) \overset{\text{c}}{=} -\log p(\boldsymbol{\theta} \mid \mathbb{D}). \quad (2.12)
$$

With $\overset{\text{c}}{=}$, we denote equality up to an additive constant that does not depend on the parameters $\boldsymbol{\theta}$. Equation (2.12) implies that minimizing the regularized empirical risk is equivalent to finding the mode of the posterior distribution (also called maximum a posteriori (MAP) estimate) $\boldsymbol{\theta}_\star = \arg\min_{\boldsymbol{\theta} \in \Theta} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}, \mathcal{D}) = \arg\max_{\boldsymbol{\theta} \in \Theta} \log p(\boldsymbol{\theta} \mid \mathbb{D})$.

Bayesian Neural Networks. The interpretation of the regularized empirical risk as a log-posterior suggests a Bayesian perspective on

9: Analogous derivations are possible for other loss functions, *e.g.* the MSE loss from Equation (2.7) can be interpreted as the negative logarithm of a Gaussian likelihood.

10: From Bayes' theorem

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

follows

$$
\begin{aligned}
\log p(\boldsymbol{\theta} \mid \mathbb{D}) = &\log p(\mathbb{D} \mid \boldsymbol{\theta}) \\
&+ \log p(\boldsymbol{\theta}) \\
&- \log p(\mathbb{D}).
\end{aligned}
$$

The log-evidence term $\log p(\mathbb{D})$ is constant *w.r.t.* the parameters $\boldsymbol{\theta}$ and can thus be omitted for the sake of our argument.

neural networks: Instead of finding a single MAP point estimate $\boldsymbol{\theta}_\star$, we could consider the full posterior distribution $p(\boldsymbol{\theta} \mid \mathbb{D})$ over the parameters. A neural network that is equipped with such a posterior distribution is called a *Bayesian neural network*.

Approximate Bayesian Inference with Laplace. Computing the *exact* Bayesian posterior $p(\boldsymbol{\theta} \mid \mathbb{D}) = {}^{p(\mathbb{D}\mid\boldsymbol{\theta})\,p(\boldsymbol{\theta})}/_{p(\mathbb{D})}$ is infeasible in practice as the evidence term $p(\mathbb{D}) = \int p(\mathbb{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\theta}$ in the denominator requires the computation of an integral over the entire (high-dimensional) parameter space and the integrand does generally not have structure we can leverage to obtain a closed-form solution. Consequently, we need to resort to approximate methods. A widely used approach is the *Laplace approximation* (LA) which approximates the posterior with a Gaussian distribution $p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}_\star, \boldsymbol{\Sigma})$ centered around the MAP estimate. We will discuss this method in more detail in Section 3.2.

Posterior Predictive Distribution. The posterior is particularly useful for quantifying uncertainty in the model's predictions by propagating the uncertainty over the parameters to the network's outputs: Given a test input $\boldsymbol{x}_\diamond \in \mathbb{X}$, the posterior predictive distribution over the output $y_\diamond$ is obtained by taking the expectation of the model likelihood under the posterior, *i.e.*

$$\begin{aligned} p(y_\diamond \mid \boldsymbol{x}_\diamond, \mathbb{D}) &= \mathbb{E}_{p(\boldsymbol{\theta}\mid\mathbb{D})}[p(y_\diamond \mid \boldsymbol{x}_\diamond, \boldsymbol{\theta})] \\ &= \int p(y_\diamond \mid \boldsymbol{x}_\diamond, \boldsymbol{\theta})\, p(\boldsymbol{\theta} \mid \mathbb{D})\, \mathrm{d}\boldsymbol{\theta}. \end{aligned} \tag{2.13}$$

A straightforward approximation to this integral is given by the Monte Carlo (MC) estimator

$$p(y_\diamond \mid \boldsymbol{x}_\diamond, \mathbb{D}) \approx \frac{1}{S} \sum_{s=1}^{S} p(y_\diamond \mid \boldsymbol{x}_\diamond, \boldsymbol{\theta}^{(s)}), \quad \text{with} \quad \boldsymbol{\theta}^{(s)} \sim p(\boldsymbol{\theta} \mid \mathbb{D}). \tag{2.14}$$

For instance, for the softmax cross-entropy loss (see Equation (2.8)), we have $p(y_\diamond \mid \boldsymbol{x}_\diamond, \boldsymbol{\theta}^{(s)}) = \mathrm{Cat}\big(y_\diamond, \mathrm{softmax}(f_{\boldsymbol{\theta}^{(s)}}(\boldsymbol{x}_\diamond))\big)$.[11] Being able to capture the uncertainty in the model's predictions is particularly important for decision-making in safety-critical applications like autonomous driving or medical diagnosis.

11: Immer et al. [66] suggests replacing $f_{\boldsymbol{\theta}}$ with a local *linearization* (in $\boldsymbol{\theta}$) to alleviate the common underfitting problem with the LA. More recent work by Roy et al. [117] supports this approach.

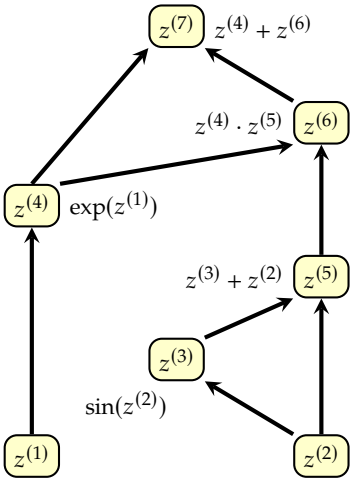
2.2.4 Automatic Differentiation

In the previous section, we established the equivalence between *minimizing* the regularized empirical risk and *maximizing* the posterior. No matter which perspective we take, we must solve a high-dimensional optimization problem to determine the optimal parameters $\boldsymbol{\theta}_\star \in \Theta \subseteq \mathbb{R}^P$. Most modern optimizers designed for this task are based on the gradient of the loss function with respect

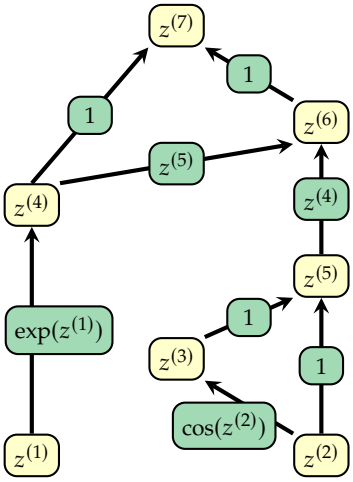
(a) Function as Python Program.

```python
from math import exp, sin

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

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

    # output variable
    z7 = z4 + z6

    return z7
```

(b) Computation Graph.



(c) Local Derivatives.



(d) Bauer Paths.

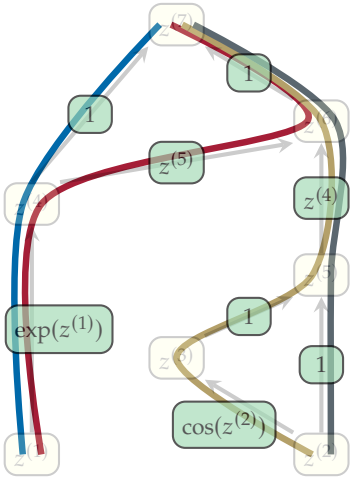


Figure 2.3: Basic Automated Differentiation Principles [105]. This figure is reproduced with the author's permission from [25, Fig. 2.5]. (a) The Python program produces the result

$$z^{(7)} = \exp(z^{(1)}) + \exp(z^{(1)})\big(\sin(z^{(2)}) + z^{(2)}\big)$$

from the inputs $z^{(1)}, z^{(2)}$ through a sequence of intermediate results $z^{(3)}, z^{(4)}, z^{(5)}, z^{(6)}$. (b) Representation as computation graph to track dependencies between the intermediate variables on the level of elementary 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.* another node in the graph requires summing the path product of local derivatives for all paths that connect them (see Equation (2.15)).

to the parameters $\nabla \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D})$. Popular libraries for building neural networks such as TensorFlow [1], PyTorch [111], and JAX [12] also provide *automatic differentiation* (AD) tools that efficiently compute these gradients using the *backpropagation* algorithm [119].

Automatic Differentiation (AD). AD produces a program that computes the derivative of a function with respect to its input. The underlying principle is that all programs can be decomposed into a sequence of *elementary operations* with known derivatives. By applying the chain rule of differential calculus, AD computes the derivative of the composite function by combining the derivatives of the elementary operations.

Automatic Differentiation Toy Example. Figure 2.3 illustrates this process. (a) shows a Python program that produces an output $z^{(7)} = \exp(z^{(1)}) + \exp(z^{(1)})(\sin(z^{(2)}) + z^{(2)})$ from inputs $z^{(1)}$ and $z^{(2)}$ through a sequence of intermediate results $z^{(3)}, z^{(4)}, z^{(5)}$, and $z^{(6)}$. (b) visualizes the relationship between those variables as computation graph. It consists of vertices $z^{(1)}, \ldots, z^{(7)}$, connected

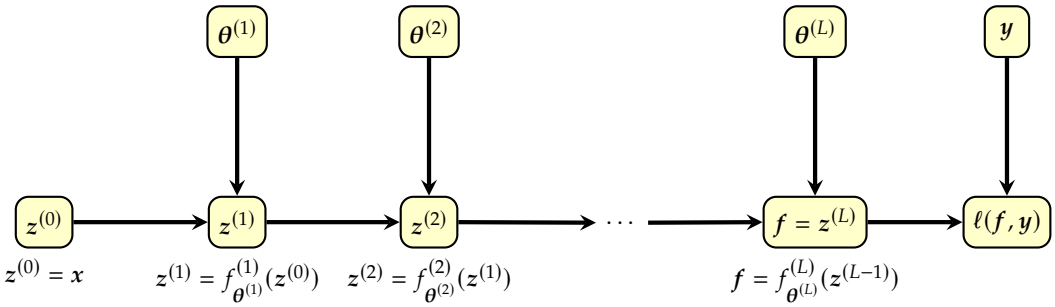


Figure 2.4: Computation Graph for a Sequential Feedforward Network. The computation graph (similar to Figure 2.3 (b)) for the loss of a single datum (x, y) in a feedforward neural network with L layers (see Definition 2.2). The loss $\ell(f, y)$ is computed by comparing the network's output $f = f_\theta(x)$ to the target y.

by directed edges that indicate a direct dependency. In (c), the graph is extended with the *local* derivatives of each node with respect to its inputs. Finally, in (d), we combine these local derivatives to obtain the derivatives of the output $z^{(7)}$ with respect to the inputs $z^{(1)}$ and $z^{(2)}$ using the path formulation of the chain rule from Bauer [6]

$$\frac{\partial z^{(j)}}{\partial z^{(i)}} = \sum_{p \in [z^{(i)} \to z^{(j)}]} \prod_{(z^{(k)}, z^{(l)}) \in p} \frac{\partial z^{(l)}}{\partial z^{(k)}}, \tag{2.15}$$

where $[z^{(i)} \to z^{(j)}]$ denotes the set of paths (a path is a sequence of directed edges) that connect $z^{(i)}$ to $z^{(j)}$. The local derivatives along the edges $(z^{(k)}, z^{(l)})$ of the path are then multiplied. We obtain (the colors used below correspond to the path color in Figure 2.3 (d))

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

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

From Scalar to Vector-Valued Nodes. The prerequisite for computing gradients of $\mathcal{L}_{\text{reg}}$ (see Equation (2.11)) is that we can compute gradients of the loss $\ell(f, y)$ (see Equations (2.7) and (2.8)) with respect to the network's parameters θ, *i.e.*

$$\nabla_\theta \ell(f, y) = \begin{pmatrix} \nabla_{\theta^{(1)}} \ell(f, y) \\ \vdots \\ \nabla_{\theta^{(L)}} \ell(f, y) \end{pmatrix}.$$

In principle, the AD procedure above can be used to compute this gradient. However, the nodes in the computation graph (see Fig-

ure 2.4) primarily represent *vector*-valued quantities. Fortunately, Equation (2.15) can be generalized to this setting. The overall structure of the formula remains the same; the only difference is that the scalar partial derivatives are replaced by Jacobian matrices (see Definition 2.3).

Definition 2.3 (Jacobian) The Jacobian matrix of a differentiable vector-to-vector function

$$h\colon \mathbb{R}^N \to \mathbb{R}^M,\ \boldsymbol{x} \mapsto h(\boldsymbol{x})$$

w.r.t. its inputs $\boldsymbol{x}$, evaluated at $\boldsymbol{x}_0$ is denoted by $\boldsymbol{J}_{\boldsymbol{x}} h(\boldsymbol{x}_0) \in \mathbb{R}^{M \times N}$. It contains the partial derivatives

$$[\boldsymbol{J}_{\boldsymbol{x}} h(\boldsymbol{x}_0)]_{i,j} = \left.\frac{\partial h_i(\boldsymbol{x})}{\partial x_j}\right|_{\boldsymbol{x}=\boldsymbol{x}_0}$$

for $i = 1, \dots, M$ and $j = 1, \dots, N$.

If h is scalar-valued ($M = 1$), the Jacobian reduces to the gradient $\left(\boldsymbol{J}_{\boldsymbol{x}} h(\boldsymbol{x}_0)\right)^\top = \nabla_{\boldsymbol{x}} h(\boldsymbol{x}_0) \in \mathbb{R}^N$.

Gradient Backpropagation. The assumed feedforward architecture of the network implies a chain structure in the computation graph (see Figure 2.4). Therefore, when computing $\nabla_{\boldsymbol{\theta}^{(l)}} \ell(\boldsymbol{f}, \boldsymbol{y})$, there is only a *single* path in $[\boldsymbol{\theta}^{(l)}, \ell(\boldsymbol{f}, \boldsymbol{y})]$. As a result, the sum in Equation (2.15) collapses to the Jacobian matrix chain

$$\begin{aligned}&\nabla_{\boldsymbol{\theta}^{(l)}} \ell(\boldsymbol{f}, \boldsymbol{y}) \\ &= \underbrace{\left[\boldsymbol{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}^{(l)}\right]^\top}_{\in \mathbb{R}^{P^{(l)} \times O^{(l)}}} \underbrace{\left[\boldsymbol{J}_{\boldsymbol{z}^{(l)}} \boldsymbol{z}^{(l+1)}\right]^\top}_{\in \mathbb{R}^{O^{(l)} \times O^{(l+1)}}} \cdots \underbrace{\left[\boldsymbol{J}_{\boldsymbol{z}^{(L-1)}} \boldsymbol{z}^{(L)}\right]^\top}_{\in \mathbb{R}^{O^{(L-1)} \times C}} \underbrace{\left[\boldsymbol{J}_{\boldsymbol{z}^{(L)}} \ell(\boldsymbol{f}, \boldsymbol{y})\right]^\top}_{= \nabla_{\boldsymbol{f}} \ell(\boldsymbol{f}, \boldsymbol{y}) \in \mathbb{R}^C},\end{aligned} \tag{2.16}$$

where $\boldsymbol{z}^{(l)} \in \mathbb{R}^{O^{(l)}}$ denotes the output of layer l and $\boldsymbol{\theta}^{(l)} \in P^{(l)}$ the number of parameters. The gradient from Equation (2.16) is usually computed from right to left (this is known as *reverse* mode AD or *backward pass*). There are two main reasons for this:

1. Multiplying from right to left results in a sequence of matrix-*vector* products (rather than a sequence of matrix-*matrix* products in *forward* mode AD) which is more efficient in terms of memory consumption and run time.
2. The paths from *e.g.* $\boldsymbol{\theta}^{(1)}$ and $\boldsymbol{\theta}^{(2)}$ to $\ell(\boldsymbol{f}, \boldsymbol{y})$ share most of their edges (see Figure 2.4). Consequently, most of the computations required for evaluating $\nabla_{\boldsymbol{\theta}^{(1)}} \ell(\boldsymbol{f}, \boldsymbol{y})$ and $\nabla_{\boldsymbol{\theta}^{(2)}} \ell(\boldsymbol{f}, \boldsymbol{y})$ via Equation (2.16) are identical and can be *shared*. This is not possible in forward mode AD because the intermediate results are "tainted" by the application of the left-most Jacobian in Equation (2.16) and thus cannot be reused.

The resulting algorithm is known as *backpropagation* [119] and is the cornerstone of neural network training. Deep learning frameworks such as PyTorch [111] implement backpropagation in a highly optimized way with support for GPU (graphics processing unit) acceleration.

Higher-Order Quantities. The standard backpropagation algorithm efficiently computes the gradient of the loss function *w.r.t.* the network's parameters. However, it can be extended to compute additional information, such as higher moments (e.g. the gradient variance) or higher-order derivatives like the Hessian matrix (or approximations thereof, see Section 3.4). Software libraries like BackPACK [28], ASDL [106], and CurvLinOps [26] implement these extensions, making them more accessible to researchers and practitioners.

2.2.5 Gradient-Based Optimizers for Training Neural Networks

Training a neural network is the process of finding the minimizer $\boldsymbol{\theta}_\star = \arg\min_{\boldsymbol{\theta}\in\mathbb{\Theta}} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}, \mathcal{D})$ of the regularized empirical risk.[12] Specialized numerical optimizers can be used to solve this high-dimensional optimization problem (approximately). Given an initial parameter vector $\boldsymbol{\theta}_0 \in \mathbb{\Theta} \subseteq \mathbb{R}^P$, these methods create a trajectory of iterates $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots$ to approach $\boldsymbol{\theta}_\star$ (see Figure 2.5). Different optimization methods differ in their *update rule*, *i.e.* how they compute the next iterate $\boldsymbol{\theta}_{t+1}$ given $\boldsymbol{\theta}_t$. The simplest one is *gradient descent*.

12: Practitioners are often not interested in the empirical risk on the training data itself. In classification tasks, for example, the ultimate goal is typically high *accuracy* (not loss) on previously unseen *test* (not training) data. However, test accuracy cannot be used directly as training objective: Test data cannot be part of the training pipeline and accuracy does not provide a meaningful gradient signal for optimization. We thus have to make do with a surrogate objective (*e.g.* cross-entropy loss on the training data) that does not necessarily reflect all the criteria that are relevant for the model's practical usefulness.

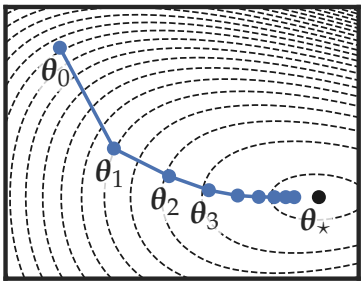


Figure 2.5: Gradient Descent in 2D. Trajectory of iterates created by gradient descent updates applied to a 2D objective function (contour lines - -).

Gradient Descent. Let $\boldsymbol{g}_{\mathcal{D}} := \nabla\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_t, \mathcal{D})$ denote the gradient of the regularized empirical risk *w.r.t.* the model's parameters $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_t$. The gradient descent update rule is defined as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \boldsymbol{g}_{\mathcal{D}}, \tag{2.17}$$

where $\alpha_t > 0$ is the scalar *learning rate* or *step size*. The following paragraph provides an intuitive explanation for why the gradient is a reasonable choice for the update direction.

Gradient Descent as Steepest Descent. Ultimately, our goal is to decrease $\mathcal{L}_{\text{reg}}$ as far as possible. Locally, the direction that reduces the loss function the fastest—this is the direction of *steepest descent*—is optimal. It holds [95, Sec. 6]

$$\lim_{\epsilon\to 0} \frac{1}{\epsilon} \arg\min_{\Delta\boldsymbol{\theta}\in\mathbb{\Theta}, \|\Delta\boldsymbol{\theta}\|_2 \leq \epsilon} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}, \mathcal{D}) = -\frac{\boldsymbol{g}_{\mathcal{D}}}{\|\boldsymbol{g}_{\mathcal{D}}\|_2}, \tag{2.18}$$

Equation (2.18) shows that, locally, the direction of steepest descent is given by the normalized negative gradient. This justifies the gradient descent update rule. By moving into the direction of steepest descent, the next iterate $\boldsymbol{\theta}_{t+1}$ will have a lower loss value (assuming a suitable learning rate $\alpha_t > 0$).

Stochastic Gradient Descent (SGD). The run time of gradient descent is dominated by the computation of the gradient $\boldsymbol{g}_{\mathcal{D}}$ which has to be re-evaluated via AD (see Section 2.2.4) in every iteration. This is infeasible for large-scale problems. *Stochastic gradient descent* (SGD) addresses this by replacing the gradient $\boldsymbol{g}_{\mathcal{D}}$ in Equation (2.17) with a *stochastic* estimate $\boldsymbol{g}_{\mathcal{B}_t} = \nabla\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_t, \mathcal{B}_t)$ evaluated on a *mini-batch* (*i.e.* a subset $\mathcal{B}_t \subset \mathcal{D}$ sampled uniformly at random from the training data in step t). The update rule is given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \boldsymbol{g}_{\mathcal{B}_t}. \tag{2.19}$$

In Section 2.2.2, we introduced the empirical risk $\mathcal{L}(\boldsymbol{\theta}, \mathcal{D})$ as a Monte Carlo estimate of the expected risk $\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p(\boldsymbol{x},\boldsymbol{y})}[\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})]$

based on a set of *i.i.d.* training examples drawn from the data distribution $p(\boldsymbol{x}, \boldsymbol{y})$ (see Equations (2.9) and (2.10)). Analogously, $\mathcal{L}(\boldsymbol{\theta}, \mathcal{B})$ yields an estimate of the expected risk—just based on a smaller set of samples from p. The respective gradients $\boldsymbol{g}_{\mathcal{D}}$ and $\boldsymbol{g}_{\mathcal{B}}$ thus approximate the same object, namely the gradient of the expected risk (plus the regularizer's gradient). The main difference between them is their variance: $\boldsymbol{g}_{\mathcal{D}}$ has lower variance than $\boldsymbol{g}_{\mathcal{B}}$ because the variance of the Monte Carlo estimate drops linearly with the size of the sampled data set.

Other Popular Gradient-Based Methods. While there is a *vast* number of different optimization methods [122, Tab. 2], they often share common fundamental techniques to improve convergence, stability, and efficiency. Here are two examples:

- **Running Averages.** Due to the noise in the observations caused by mini-batching, many optimizers use running averages to estimate the relevant quantities more reliably. An example is momentum [113] that introduces a velocity term, which is the exponentially weighted sum of past gradients. Another example is Adam [74] which uses two exponential moving averages to estimate the first and second moment of the gradient.
- **Adaptive Learning Rates.** Many optimizers adjust the effective learning rate dynamically based on the history of their observations. Examples are AdaGrad [36], RMSProp [137], and Adam [74]. All three methods use an estimate of the gradient's second moment to adapt the learning rate for each parameter individually. The second moment estimate appears in the "denominator" of the update rule, the intuition being that parameters with larger gradient variance should be updated more carefully than those with smaller variance.

Hyperparameters. All the methods mentioned above have *hyperparameters* that control the behavior of the optimizer and need to be set by the user or tuned. The hyperparameters include the learning rate α_t, the mini-batch size $|\mathcal{B}_t|$, the total number of iterations, and optimizer-specific parameters like momentum coefficients and exponential decay rates. As these parameters can drastically affect the optimizer's performance, they are typically tuned. Common approaches are to use a grid search over a pre-defined set of values, a random search or more advanced techniques like Bayesian optimization. There are also self-tuning methods that adapt their hyperparameters automatically during training. *Line search* methods, for instance, do not require a learning rate. Instead, they probe different values along the given search direction and choose a suitable step size using some selection criterion.

Gradient-Based Optimizers Perform Similarly. The most popu-

lar optimizers for training neural networks are gradient-based methods with structurally similar update rules. Schmidt et al. [122] empirically compare the performance of a variety of such methods on a range of tasks and data sets and report that "different optimizers exhibit a surprisingly similar performance distribution compared to a single method that is re-tuned or simply re-run with different random seeds". The authors thus question how much value there is in the development of novel optimizers as long as they are "conceptually and functionally close to the existing population" [122, Sec. 5]. In Section 3.1, we therefore discuss a *conceptually* different and potentially more powerful class of optimization methods: Second-order optimizers.

Curvature-Based Methods in Machine Learning

3

Curvature, *i.e.* the Hessian matrix or an approximation thereof, is a valuable quantity. Crucially, it provides access to a local quadratic Taylor expansion of a function, which forms the foundation of *curvature-based* or *second-order* methods. Minimizing this quadratic yields the Newton step which serves as the core building block for second-order optimization methods (see Section 3.1). A quadratic approximation is also useful for approximate inference as it allows to replace an arbitrary probability density function by a Gaussian—this is known as the Laplace approximation (LA) (see Section 3.2). We briefly discuss other use cases for curvature information in Section 3.3. While the Hessian is the most natural choice for the curvature matrix, it is often impractical due to undesirable properties and the associated computational costs. We therefore describe commonly-used alternative curvature matrices in Section 3.4.

3.1 Second-Order Optimization

Curvature information enables us to derive a powerful class of optimization methods: Second-order optimizers. These methods can converge much faster than first-order methods and have a long history of success in many scientific disciplines.

Derivation of Newton's Method. Assume that we want to solve the minimization problem $\min_{\boldsymbol{z}\in\mathbb{R}^I} h(\boldsymbol{z})$ where $h\colon \mathbb{R}^I \to \mathbb{R}$ is the objective function.[1] At a given location $\boldsymbol{z}_0 \in \mathbb{R}^I$, we can approximate h locally with a Taylor expansion. Using terms up to the second order, we obtain

$$h(\boldsymbol{z}) \approx q(\boldsymbol{z}) \coloneqq \frac{1}{2}(\boldsymbol{z}-\boldsymbol{z}_0)^\top \boldsymbol{H}\,(\boldsymbol{z}-\boldsymbol{z}_0) + (\boldsymbol{z}-\boldsymbol{z}_0)^\top \boldsymbol{g} + c, \qquad (3.1)$$

where $\boldsymbol{H} \coloneqq \nabla^2 h(\boldsymbol{z}_0) \in \mathbb{R}^{I\times I}$ is the objective function's Hessian matrix evaluated at $\boldsymbol{z}_0$, $\boldsymbol{g} \coloneqq \nabla h(\boldsymbol{z}_0) \in \mathbb{R}^I$ is the gradient and $c \coloneqq h(\boldsymbol{z}_0) \in \mathbb{R}$ is a scalar shift.

As q has a simple polynomial form (in contrast to h), we can derive its minimum in closed form and use it as next iterate. To find that minimizer, we set the gradient of the quadratic approximation from Equation (3.1) to zero:

$$\nabla q(\boldsymbol{z}) = \boldsymbol{H}\,(\boldsymbol{z}-\boldsymbol{z}_0) + \boldsymbol{g} \overset{!}{=} \boldsymbol{0} \quad \Rightarrow \quad \boldsymbol{z} = \boldsymbol{z}_0 - \boldsymbol{H}^{-1}\boldsymbol{g}. \qquad (3.2)$$

1: In deep learning, the objective function is the (regularized) empirical risk (see Equation (2.11)) and the input space is the parameter space $\Theta \subseteq \mathbb{R}^P$.

Algorithm 3.1: Newton's Method.

Input: Objective function $h\colon \mathbb{R}^I \to \mathbb{R}$, initial iterate z_0
Output: Approximate minimizer of h

for $i = 0, 1, 2, \ldots$ **while not** StoppingCriterion() **do**
  $g \leftarrow \nabla h(z_i)$ and $H \leftarrow \nabla^2 h(z_i)$ — Compute the gradient and spd Hessian at z_i
  $\Delta z \leftarrow -H^{-1} g$ — Compute the Newton step
  $z_{i+1} \leftarrow z_i + \Delta z$ — Use the minimizer of the local quadratic as next iterate

The stopping criterion StoppingCriterion() typically includes a maximum number of iterations and/or a threshold on the norm of the current gradient.

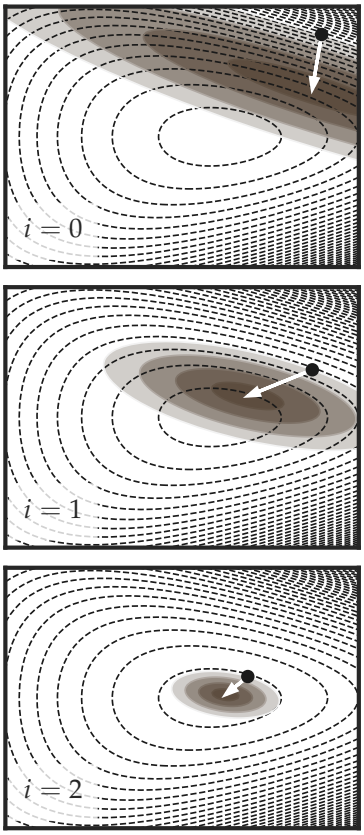


Figure 3.1: Newton's Method in 2D. Three iterations (from *(Top)* to *(Bottom)*) of Newton's method applied to a 2D objective function (contour lines - -). The local quadratic approximation around z_i (●) is shown as ▭ and the Newton step Δz as white arrow.

Newton's method (see Algorithm 3.1) applies this update rule iteratively. An illustration of the algorithm is given in Figure 3.1.

The Newton Step Requires a Positive Definite Hessian. The Hessian must be positive definite; otherwise the quadratic's minimum does not exist. Obviously, eigenvalues cannot be zero as this would imply that the Hessian is not invertible. But what if the Hessian has negative eigenvalues? Consider a cut through the quadratic (Equation (3.1)) from the anchor point z_0 along some normalized eigenvector u of the Hessian H, *i.e.* $Hu = \lambda u$, $\|u\| = 1$. It holds

$$r(\tau) := q(z_0 + \tau u) = \frac{1}{2}\tau^2 u^\top H\, u + \tau u^\top g + c = \frac{\lambda}{2}\tau^2 + (u^\top g)\tau + c.$$

This is a 1D parabola $r\colon \mathbb{R} \to \mathbb{R}$ with derivatives

$$r'(\tau) = \lambda\tau + u^\top g \quad \text{and} \quad r''(\tau) \equiv \lambda. \tag{3.3}$$

Equation (3.3) provides a nice geometric interpretation of the Hessian's eigenvalues: For a quadratic approximation, the *directional* curvature $r''(\tau)$ along the eigenvector u coincides with the respective eigenvalue λ. If $\lambda > 0$, the parabola r has a minimum at $\tau_\star := -u^\top g/\lambda$ since $r'(\tau_\star) = 0$ and $r''(\tau_\star) = \lambda > 0$. However, if $\lambda < 0$, r is *unbounded* from below and the concept of the Newton step as the step into q's *minimum* is thus no longer valid.

Local Quadratic Convergence of Newton's Method. A key quantity in the theoretical analysis of an optimization method is its convergence rate. It describes, under certain conditions, how fast the sequence of iterates converges to the optimum. For instance, gradient descent algorithms can be shown to converge quite slowly (at a linear rate [102, Theorem 3.4]) and are greatly affected by the condition number of the Hessian [11, Sec. 9.3]: If the problem is ill-conditioned (*i.e.* there are directions of small and large curvature present at the same time), this greatly restricts the applicable step sizes and leads to slow convergence. In Newton's method, these local "deformations" of the target function's geometry are compensated for by multiplying the gradient by the inverse Hessian matrix. The resulting robustness against ill-conditioned problems

is one of the key advantages of second-order methods and leads to faster local convergence of quadratic order.

Theorem 3.1 (Quadratic Convergence of Newton's Method) Suppose that h is twice differentiable and that the Hessian $\nabla^2 h(\boldsymbol{z})$ is Lipschitz continuous in a neighborhood of a solution $\boldsymbol{z}_\star$[2] at which the sufficient conditions[3] are satisfied. For the sequence of iterates created by Newton's method (see Algorithm 3.1) holds: If the starting point $\boldsymbol{z}_0$ is sufficiently close to $\boldsymbol{z}_\star$, the sequence of iterates converges to $\boldsymbol{z}_\star$ and the rate of convergence is quadratic.[4]

Proof. See [102, Theorem 3.5].

2: This condition ensures that the Hessian cannot change arbitrarily fast within the neighborhood around $\boldsymbol{z}_\star$ (see [102, Eq. (A.42)]). The smaller the bound on these changes, the more accurate the quadratic approximation.

3: These conditions ensure that $\boldsymbol{z}_\star$ is a strict local minimizer of h (see [102, Theorem 2.4]).

4: The sequence of iterates $\{\boldsymbol{z}_i\}$ converges quadratically if

$$\frac{\|\boldsymbol{z}_{i+1} - \boldsymbol{z}_\star\|}{\|\boldsymbol{z}_i - \boldsymbol{z}_\star\|^2} \leq M,$$

for all i sufficiently large and some positive constant M [102, Sec. A.2].

Failure Analysis (1D Example). Theorem 3.1 only ensures convergence to a solution in a *neighborhood* of the optimum. That means, if the initial iterate lies outside the convergence radius, the vanilla Newton approach may not converge. To illustrate this, consider the function $h\colon \mathbb{R} \to \mathbb{R}, z \mapsto z^2 - \cos(z)$ (see Figure 3.2) with derivatives $h'(z) = 2z + \sin(z)$ and $h''(z) = 2 + \cos(z)$. The objective h fulfills the assumptions of Theorem 3.1: It is smooth (implying that the Hessian is Lipschitz continuous) with a strict global minimum at $z_\star = 0$. Furthermore, h is strictly convex (since $h''(z) \geq 1$ for all z). Despite those favorable properties, Newton's method fails to converge for $z = \pi$. The subsequent iterate is given by $z_1 = z_0 - {h'(z_0)}/{h''(z_0)} = \pi - {2\pi + \sin(\pi)}/{2 + \cos(\pi)} = -\pi$. Similarly, $z_2 = \pi = z_0$, *i.e.* Newton's method oscillates between π and $-\pi$.

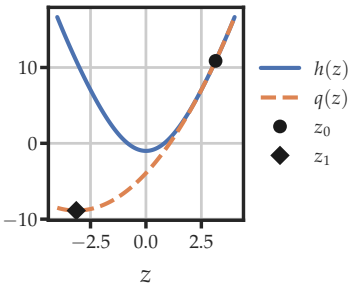


Figure 3.2: Example. The function $h(z) = z^2 - \cos(z)$, its quadratic approximation q around $z_0 = \pi$, and the subsequent Newton iterate $z_1 = -\pi$. Newton's method fails to converge despite the favorable properties of the objective function.

Practical Second-Order Optimizers. The example above demonstrates that Newton's method in its original form can be "brittle". To improve its applicability and robustness, practical implementations typically incorporate modifications such as line searches [102, Sec. 3] or trust-region approaches [102, Sec. 4].

For large-scale problems, solving the linear system $\Delta \boldsymbol{z} = -\boldsymbol{H}^{-1}\boldsymbol{g}$ is often computationally infeasible. To mitigate this, we must exploit or impose structure in the curvature matrix. Section 3.4 discusses commonly used curvature proxies and approximations for the empirical risk. In the context of deep learning, additional challenges arise, *e.g.* the non-convexity of the loss landscape and the stochasticity of the gradient and curvature estimates—these are discussed in Section 4.2.

3.2 The Laplace Approximation

The Laplace approximation (LA) goes back to the mathematician Pierre-Simon Laplace [83]. The idea is to approximate a difficult-to-handle probability density function (PDF) with a Gaussian. This

technique is useful, in particular, when dealing with posterior distributions in Bayesian inference.

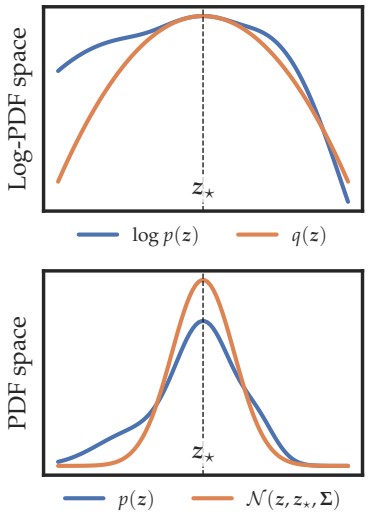


Figure 3.3: Laplace Approximation in 1D. *(Top)* The top panel shows the log-density and the local quadratic Taylor expansion q around the mode $z_\star$ (dashed vertical line). *(Bottom)* The bottom panel shows the original (non-Gaussian) density function and the Laplace approximation.

Derivation of the Laplace Approximation. Let p denote a PDF with mode $z_\star = \arg\max_{z \in \mathbb{R}^I} p(z)$. Our goal is to find a Gaussian approximation to p around its mode $z_\star$. As the log-density of a Gaussian is a quadratic function, the first step towards that goal is to approximate the log-density $\log(p(z))$ locally around its mode $z_\star$ with a quadratic Taylor expansion (see top panel in Figure 3.3)

$$\log(p(z)) \approx q(z) := \log(p(z_\star)) + \frac{1}{2}(z - z_\star)^\top \nabla^2 \log(p(z_\star))\,(z - z_\star).$$

Note that the linear term $\nabla \log(p(z_\star))^\top (z - z_\star)$ is zero because $z_\star$ is a local maximizer and thus the gradient $\nabla \log(p(z_\star))$ vanishes. Exponentiating both sides yields

$$p(z) \approx \exp(q(z)) = p(z_\star)\, \exp\left(\frac{1}{2}(z - z_\star)^\top \nabla^2 \log(p(z_\star))(z - z_\star)\right).$$

We want the approximation on the right-hand side to be a valid PDF, *i.e.* we scale it such that it integrates to one. The right-hand side has the form of a Gaussian PDF with mean $z_\star$ and covariance matrix $\boldsymbol{\Sigma} := -\left(\nabla^2 \log(p(z_\star))\right)^{-1}$, *i.e.* to make it integrate to one, we simply have to replace $p(z_\star)$ by the Gaussian normalization constant $(2\pi)^{-I/2} \det(\boldsymbol{\Sigma})^{-1/2}$. By substituting this scaling factor, we finally arrive at the LA $p(z) \approx \mathcal{N}(z; z_\star, \boldsymbol{\Sigma}) = (2\pi)^{-I/2} \det(\boldsymbol{\Sigma})^{-1/2} \exp(-1/2 (z - z_\star)^\top \boldsymbol{\Sigma}^{-1} (z - z_\star))$ (see bottom panel in Figure 3.3).

The LA can still be applied even if p is not normalized. This is because the normalization constant appears as an *additive* constant in the log-density and thus neither affects the mode $z_\star$ (the arg max of the log-density) nor the curvature $\nabla^2 \log(p(z_\star))$. This is useful in the context of approximate inference where the normalization constant is often intractable.

Potential Issues with the Laplace Approximation. The LA is a *local* approximation, *i.e.* it only captures the shape of $p(z)$ close to the mode $z_\star$. The approximation can thus be poor if p is highly "non-Gaussian", *e.g.* if it is very skewed or multimodal. Another limitation is that the support of the Gaussian approximation is always $\mathbb{R}^I$. This can be problematic if p has a different support, *e.g.* the positive real numbers. Finally, the LA requires access to the Hessian matrix $\nabla^2 \log(p(z_\star))$, which is computationally expensive to store and invert for high-dimensional problems. We must also ensure that the covariance matrix $\boldsymbol{\Sigma} := -\left(\nabla^2 \log(p(z_\star))\right)^{-1}$ is positive definite at $z_\star$, *i.e.* the stationary point $z_\star$ has to be a local maximizer of p (and not a local minimum or saddle point). This can require using an alternative curvature proxy (see Section 3.4).

The Laplace Approximation for Neural Networks. The LA can be used to turn a neural network into a *Bayesian* neural network (see Section 2.2.3) in a post-hoc manner [30, 66, 77, 90, 115–117]. Recall from Equation (2.12) that the softmax cross-entropy loss with an ℓ_2 regularizer can be interpreted as the negative log-posterior over the network's parameters $N \cdot \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) \overset{c}{=} -\log p(\boldsymbol{\theta} \mid \mathbb{D})$. The LA to the posterior is thus given by $p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}_\star, \boldsymbol{\Sigma})$, where $\boldsymbol{\theta}_\star$ is the posterior mode and the covariance matrix is given by

$$\boldsymbol{\Sigma} = -\left(\nabla^2 \log p(\boldsymbol{\theta}_\star \mid \mathbb{D})\right)^{-1} \overset{(2.12)}{=} \frac{1}{N}\left(\nabla^2 \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_\star; \mathcal{D})\right)^{-1}. \tag{3.4}$$

A great advantage of the LA is that it can be applied in a post-hoc manner to any pre-trained neural network as it only depends on local quantities at $\boldsymbol{\theta}_\star$.

Sampling from the Laplace Approximation. The primary goal of Bayesian deep learning is to quantify uncertainty in the model's predictions. Using a LA, we can approximate the posterior predictive via MC samples (see Equation (2.14)), which requires samples $\boldsymbol{\theta}^{(s)} \sim \mathcal{N}(\boldsymbol{\theta}_\star, \boldsymbol{\Sigma})$. Assume $\boldsymbol{s} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, then $\boldsymbol{Q}\boldsymbol{s} + \boldsymbol{q} \sim \mathcal{N}\left(\boldsymbol{q}, \boldsymbol{Q}\boldsymbol{Q}^\top\right)$. Consequently, we obtain samples from the LA by setting $\boldsymbol{q} \leftarrow \boldsymbol{\theta}_\star$ and $\boldsymbol{Q}$ such that $\boldsymbol{Q}\boldsymbol{Q}^\top = \boldsymbol{\Sigma}$. Such a matrix can be obtained via Cholesky decomposition $\boldsymbol{L}\boldsymbol{L}^\top = \boldsymbol{\Sigma}$, where $\boldsymbol{L}$ is a lower triangular matrix. Another option is to construct $\boldsymbol{Q}$ from an eigendecomposition $\boldsymbol{\Sigma} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$, where columns of $\boldsymbol{U}$ are eigenvectors of $\boldsymbol{\Sigma}$ and $\boldsymbol{\Lambda}$ contains the eigenvalues. We obtain $\boldsymbol{\Sigma} = \boldsymbol{Q}\boldsymbol{Q}^\top$ by setting $\boldsymbol{Q} \leftarrow \boldsymbol{U}\boldsymbol{\Lambda}^{1/2}$.

However, a Cholesky decomposition or eigendecomposition of the covariance matrix (Equation (3.4)) is computationally infeasible in deep learning applications, as both methods exhibit cubic run time complexity in the number of network parameters P. Besides, the Hessian $\nabla^2 \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_\star; \mathcal{D})$ can be indefinite which would lead to an invalid covariance matrix (that must be positive definite). We thus explore alternative curvature proxies and approximations with more favorable properties in Section 3.4.

3.3 Other Use Cases for Curvature in Deep Learning

In this manuscript, we focus on two main applications of curvature information: Second-order optimization and uncertainty quantification via the LA (see Sections 3.1 and 3.2). In this section, we briefly outline additional use cases for curvature information in the deep learning context.

Empirical Analyses of the Loss Landscape. Second-order approximations to the loss landscape have been used to analyze the geometry of the loss function and its local minima [29, 44, 50, 124, 134]. These analyses have proven valuable for understanding fundamental challenges in training deep neural networks. Furthermore, studying the spectral properties of the Hessian (or approximations thereof) provides insights into the stability of the training process [4, 20] and the network's generalization properties [71, 121].

Influence Functions. Influence functions [76] leverage curvature information to determine which training data points most significantly impact a model's prediction. This is useful for understanding the behavior of complex models like deep neural networks, detecting mislabeled training data, and studying model robustness against adversarial attacks or input perturbations. Koh and Liang [76] derive a closed-form approximation for how model parameters would change if a training point were upweighted or removed—*without retraining* the network. This expression (see [76, Eq. (1)]) includes an inverse curvature-vector product.

Use Cases Based on the Laplace Approximation. Beyond uncertainty quantification in Bayesian neural networks (see Section 2.2.3), the LA has several other important applications in deep learning, some of which we highlight below.

- **Model Selection.** In a Bayesian model, the *marginal likelihood* or *model evidence* is the normalization constant of the posterior distribution and quantifies the "plausibility" of observing the training data under a given model. Via the LA, one can obtain a closed-form estimate of this quantity [65, Eq. (3)], providing a principled approach to compare different model architectures and hyperparameter configurations.
- **Model Pruning.** Pruning techniques aim to reduce the number of parameters[5] in a neural network while maintaining accuracy, resulting in faster inference and lower memory requirements. Curvature-based approaches [34, 52, 86, 128] leverage a quadratic approximation of the loss around the MAP estimate $\boldsymbol{\theta}_\star$. Under this approximation, we obtain a closed-form expression that quantifies the expected increase in training loss when removing a specific parameter. Parameters are then ranked according to this statistic, and those with minimal predicted impact are pruned from the model.
- **Model Merging.** The goal of model merging is to combine multiple trained models into a single one while preserving the capabilities of the individual models. When each model is equipped with a posterior distribution over its parameters, we ideally want the merged model's parameters to have high probability under *all* individual posteriors. Matena and Raffel

5: Removing a parameter means fixing it at zero.

[98] propose *Fisher merging*, which approximates each posterior with a LA—this helps "weigh" the relative importance of parameters across different models and exhibits superior performance compared to a naive averaging approach.

- **Continual Learning.** In continual learning, a model is trained on a *sequence* of tasks. The goal is to learn each task sequentially without compromising previously acquired knowledge. Models must thus adapt to new tasks while preserving performance on earlier objectives, balancing between catastrophic forgetting and loss of plasticity. Bayesian approaches leveraging the LA [75, 116, 129] offer a principled framework for this. The LA effectively identifies directions in parameter space that are crucial for the performance on the *current* task. Penalizing changes in this subspace during training on a *new* task via regularization thus helps to mitigate catastrophic forgetting.

3.4 Curvature Matrices for the Empirical Risk

The Hessian Matrix. We already encountered the Hessian matrix in Equation (3.1), which introduced the second-order Taylor expansion as a meaningful local approximation to a given function. The Hessian of the empirical risk $\nabla^2 \mathcal{L}(\boldsymbol{\theta}_\star; \mathcal{D}) \in \mathbb{R}^{P \times P}$ contains the second-order partial derivatives with respect to the model parameters $\boldsymbol{\theta}$, *i.e.* $[\nabla^2 \mathcal{L}(\boldsymbol{\theta}; \mathcal{D})]_{i,j} = {}^{\partial^2 \mathcal{L}(\boldsymbol{\theta}; \mathcal{D})}/_{\partial \theta_i \partial \theta_j}$. It will often be denoted by $\boldsymbol{H}$ or $\boldsymbol{H}_{\mathcal{D}}$ in the following.

While the Hessian matrix may be the most natural choice for the curvature matrix, it is often impractical for applications in deep learning for two main reasons: (i) Second-order optimizers and the LA (see Sections 3.1 and 3.2) both require a positive definite curvature proxy. This assumption does typically *not* hold for deep learning models due to the non-convexity of the loss landscape. (ii) The Hessian $\nabla^2 \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_\star; \mathcal{D}) \in \mathbb{R}^{P \times P}$ is quadratic in the number of parameters and thus often infeasible to store ($\mathcal{O}(P^2)$), let alone invert ($\mathcal{O}(P^3)$) even for moderately sized models. In Section 3.4.1, we describe positive semidefinite curvature proxies for the empirical risk—this addresses (i). In the subsequent Section 3.4.2, we discuss approximations and tricks to mitigate the computational challenges—this addresses (ii).

3.4.1 Alternative Curvature Matrices

How to Obtain Positive Definite Curvature? Below, we introduce positive *semi*-definite curvature matrices for the empirical risk. However, Newton steps and the LA require a positive definite

curvature matrix. One way to ensure this is to include a regularizer/prior. For instance, the Hessian of the ℓ_2 regularizer $r(\boldsymbol{\theta}) = \beta/2\|\boldsymbol{\theta}\|^2$ is given by $\nabla^2 r(\boldsymbol{\theta}) = \beta \boldsymbol{I}$. This effectively shifts the eigenvalues of a positive semidefinite curvature matrix by $\beta > 0$ and thus yields a positive definite matrix. The damping term $\delta \boldsymbol{I}$, which is commonly used in stochastic second-order optimizers [94, 95, 97], has the same effect. Curvature damping results in more conservative parameter updates and is used to compensate for (i) the stochasticity in the gradient and curvature estimates due to mini-batching and (ii) the approximation error caused by the quadratic Taylor approximation (or further approximations). It is thus closely connected to trust-region methods in classic optimization literature [102, Sec. 4].

The Generalized Gauss-Newton Matrix. The generalized Gauss-Newton (GGN) matrix arises naturally when we consider the split between network and loss function $\boldsymbol{\theta} \mapsto f_{\boldsymbol{\theta}}(\boldsymbol{x}) \mapsto \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})$.[6] Using results from matrix differential calculus [93], we obtain

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

where $\boldsymbol{f} \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x})$. The Hessian of the loss for a single datum thus decomposes into two terms: The first term is the positive semidefinite GGN and captures the curvature of the *loss function* with respect to its inputs. The second term is a residual that contains the curvature of the *network* with respect to its parameters. The GGN approximation neglects this residual term. For models that are linear in $\boldsymbol{\theta}$, the GGN and the Hessian coincide since $\nabla^2_{\boldsymbol{\theta}} [\boldsymbol{f}]_c = \boldsymbol{0}$ and the residual vanishes (see *e.g.* [95, Sec. 8] for details).

6: The definition of the GGN is sensitive to the "position" of the split, *i.e.* where exactly the network ends, and the loss function begins. For instance, the softmax transformation in the cross-entropy loss (see Equation (2.8)) could alternatively be considered part of the network, that is, included in $\boldsymbol{f}$ rather than ℓ. For a discussion on this, see [95, Sec. 9.2].

Applying the decomposition above to each of the individual loss contributions in the empirical risk, the Hessian $\boldsymbol{H}_{\mathcal{D}}$ decomposes into the GGN $\boldsymbol{G}_{\mathcal{D}}$ and a residual term $\boldsymbol{R}_{\mathcal{D}}$:

$$\begin{aligned} \boldsymbol{H}_{\mathcal{D}} &= \nabla^2 \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} \nabla^2 \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) = \boldsymbol{G}_{\mathcal{D}} + \boldsymbol{R}_{\mathcal{D}} \quad \text{with} \\ \boldsymbol{G}_{\mathcal{D}} &\coloneqq \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} (\boldsymbol{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top \left(\nabla^2_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y})\right) (\boldsymbol{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n) \qquad (3.5) \\ \boldsymbol{R}_{\mathcal{D}} &\coloneqq \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} \sum_{c=1}^{C} \nabla^2_{\boldsymbol{\theta}} [\boldsymbol{f}_n]_c [\nabla_{\boldsymbol{f}_n} \ell(\boldsymbol{f}_n, \boldsymbol{y})]_c , \end{aligned}$$

where $\boldsymbol{f}_n \coloneqq f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$. The GGN $\boldsymbol{G}_{\mathcal{D}}$ is symmetric and positive semidefinite (if $\ell(\boldsymbol{f}, \boldsymbol{y})$ is convex in $\boldsymbol{f}$, as is customary). For the MSE loss $\ell(\boldsymbol{f}, \boldsymbol{y}) = 1/C \|\boldsymbol{f} - \boldsymbol{y}\|^2$ (see Equation (2.7)) holds $\nabla^2_{\boldsymbol{f}} \ell(\boldsymbol{f}, \boldsymbol{y}) = 2/C \boldsymbol{I}$ and the GGN from Equation (3.5) simplifies to

$$\boldsymbol{G}_{\mathcal{D}} = \frac{2}{|\mathcal{D}| C} \sum_{n \in \mathcal{D}} (\boldsymbol{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n)^\top (\boldsymbol{J}_{\boldsymbol{\theta}} \boldsymbol{f}_n).$$

This matrix is often referred to as the *Gauss-Newton* matrix. The GGN is a generalization of the Gauss-Newton matrix for arbitrary (convex) loss functions.

The Fisher Information Matrix & Natural Gradient Descent. We motivated gradient descent as the method based on the direction of *steepest descent* in Section 2.2.5. However, the notion of "steepness" is dependent on the metric we use to measure distances in the parameter space. So far, we have considered only the Euclidean metric $\|\Delta\boldsymbol{\theta}\|_2$ to measure the distance between two parameter vectors $\boldsymbol{\theta}$ and $\boldsymbol{\theta} + \Delta\boldsymbol{\theta}$ (see Equation (2.18)). In Section 2.2.3, we established a Bayesian perspective on neural network training. In particular, we observed that $\boldsymbol{\theta}$ often parametrizes a likelihood, *i.e.* a conditional distribution $p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x})$. For the softmax cross-entropy loss (see Equation (2.8)), for instance, we have $p_{\boldsymbol{\theta}}(y \mid \boldsymbol{x}) = \mathrm{Cat}\big(y, \mathrm{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}))\big)$ and $\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) = -\log p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n)$. This motivates a different notion of distance: Instead of measuring the distance between two parameter *vectors*, we measure the distance between the associated *distributions* using the Kullback-Leibler (KL) divergence. A local second-order Taylor expansion of the KL divergence gives rise to the *Fisher information matrix* (FIM) or, in short, the *Fisher* [3]

$$\boldsymbol{F}_{\mathcal{D}} \coloneqq \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} \mathbb{E}_{\boldsymbol{y} \sim p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n)} \left[\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}) \left(\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}) \right)^\top \right], \tag{3.6}$$

where $\boldsymbol{f}_n = f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$. The FIM is symmetric and positive semidefinite (since it takes an expectation over the vector outer-product which is trivially positive semidefinite). Based on the implied notion of distance between distributions, it can be shown that the steepest descent direction is given by the *natural gradient* [3, 95]

$$\lim_{\epsilon \to 0} \frac{1}{\epsilon} \operatorname*{arg\,min}_{\Delta\boldsymbol{\theta} \in \mathbb{D}, \|\Delta\boldsymbol{\theta}\|_{\boldsymbol{F}_{\mathcal{D}}} \leq \epsilon} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}, \mathcal{D}) = -\frac{\boldsymbol{F}_{\mathcal{D}}^{-1} \boldsymbol{g}_{\mathcal{D}}}{\|\boldsymbol{g}_{\mathcal{D}}\|_{\boldsymbol{F}_{\mathcal{D}}^{-1}}}. \tag{3.7}$$

Note that the un-normalized natural gradient $-\boldsymbol{F}_{\mathcal{D}}^{-1} \boldsymbol{g}_{\mathcal{D}}$ has the same form as the Newton step but uses the inverse FIM $\boldsymbol{F}_{\mathcal{D}}^{-1}$ instead of the inverse Hessian $\boldsymbol{H}_{\mathcal{D}}^{-1}$.

Equivalence of GGN and FIM. The GGN and the FIM are closely related. In particular, they are identical for many commonly-used loss functions, including MSE and softmax cross-entropy loss [95, Sec. 9.2]. In the following, we will thus refer to the GGN and FIM interchangeably.

Monte Carlo Fisher & Empirical Fisher. The expectation in Equation (3.6) can be approximated with a Monte Carlo estimate by drawing S samples $\{\boldsymbol{y}^{(n,s)} \sim p_{\boldsymbol{\theta}}(\boldsymbol{y} \mid \boldsymbol{x}_n)\}_{s=1}^{S}$ for each training

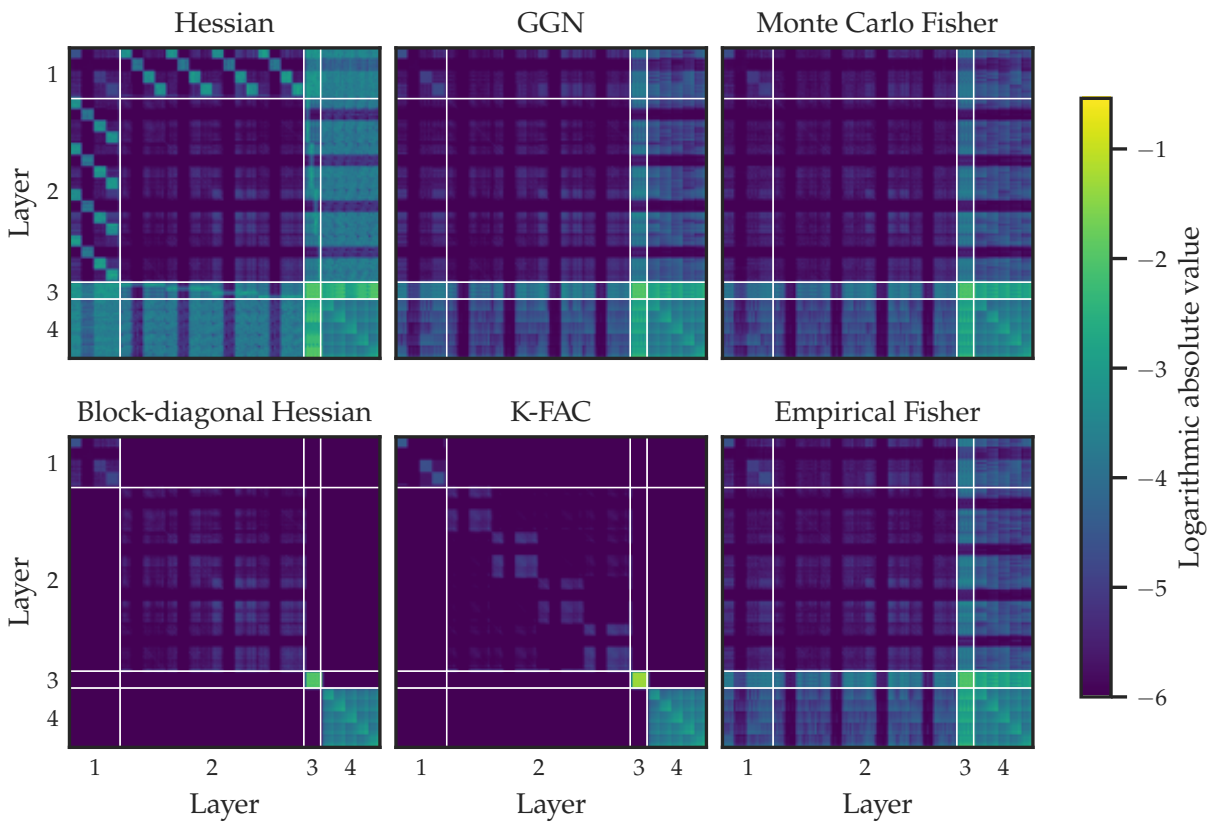


Figure 3.4: Visual Tour of Curvature Matrices. Curvature matrices for a toy neural network ($P = 683$) with four layers (three convolutional layers with ReLU or sigmoid activation and one dense linear layer), equipped with a softmax cross-entropy loss. We use a dummy data set of 50 randomly generated samples from $C = 5$ classes. Parameters from different layers are visually separated by white lines. This figure is based on the *Visual tour of curvature matrices* tutorial from the CurvLinOps documentation [26].

datum $\boldsymbol{x}_n$ and replacing the expectation with an average

$$\boldsymbol{F}_{\mathcal{D}} \approx \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} \frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}^{(n,s)}) \left(\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}^{(n,s)})\right)^\top.$$

A related curvature matrix is the *empirical Fisher*, which is defined as the un-centered gradient covariance matrix

$$\hat{\boldsymbol{F}}_{\mathcal{D}} \coloneqq \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n) \left(\nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{f}_n, \boldsymbol{y}_n)\right)^\top. \tag{3.8}$$

Contrary to what its name suggests, the empirical Fisher is *not* a Monte Carlo estimate of the FIM from Equation (3.6) because $\boldsymbol{y}_n$ is the training label and not a sample from the model distribution. For a discussion on the relationship between Fisher and empirical Fisher and the limitations of the latter in the context of optimization, the reader is referred to Kunstner et al. [82].

Figure 3.4 provides a visual tour of the curvature matrices discussed above.

3.4.2 Curvature Approximations

Below, we present approaches that address the computational challenges associated with storing and inverting curvature matrices for large-scale problems. Before diving into the computational techniques, recall the specific requirements for second-order optimization and the LA (we assume a spd curvature proxy $\boldsymbol{C}_\mathcal{D}$, *e.g.* $\boldsymbol{C}_\mathcal{D} \coloneqq \boldsymbol{G}_\mathcal{D} + \beta \boldsymbol{I}$):

- **Second-Order Optimization.** Second-order optimizers (see Section 3.1) rely on the Newton step, *i.e.* on the solution to the linear system $\boldsymbol{C}_\mathcal{D} \cdot \Delta\boldsymbol{\theta} = -\boldsymbol{g}_\mathcal{D}$.
- **Laplace Approximation.** The posterior predictive (see Equation (2.13)) can be approximated via MC samples from the LA $\mathcal{N}(\boldsymbol{\theta}_\star, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma} = {}^1\!/_N \boldsymbol{C}_\mathcal{D}^{-1}$. This requires access to matrix-vector products with $\boldsymbol{Q}$, where $\boldsymbol{Q}$ fulfills $\boldsymbol{Q}\boldsymbol{Q}^\top = \boldsymbol{\Sigma}$ (or at least $\boldsymbol{Q}\boldsymbol{Q}^\top \approx \boldsymbol{\Sigma}$). Such a matrix can be obtained via Cholesky decomposition or (truncated) SVD (see Section 3.2).

Most approaches to scalable curvature approximations fall into one of two categories: (i) matrix-free methods or (ii) structured light-weight approximations.

Matrix-Free Methods. Matrix-vector products with the Hessian $\boldsymbol{v} \mapsto \boldsymbol{H}_\mathcal{D} \cdot \boldsymbol{v}$ and GGN $\boldsymbol{v} \mapsto \boldsymbol{G}_\mathcal{D} \cdot \boldsymbol{v}$ can be implemented efficiently *without* materializing the dense matrices in memory [112, 125] (thus the name *matrix-free*). This enables the application of iterative routines from linear algebra that rely on matrix-vector products. For instance, Martens [94] trains neural networks with approximate Newton steps computed via conjugate gradient (CG) iterations [61];[7] and a partial eigendecompositon computed matrix-free (*e.g.* via power iteration, Lanczos methods or sketching methods from randomized linear algebra [38, 51]) can be used to construct a low-rank (plus diagonal) approximation of the covariance matrix in the LA [30, Ap. B] which enables efficient sampling from the approximate posterior.

A *single* matrix-vector product with the Hessian or GGN entails computational costs equivalent to *multiple* gradient evaluations [23]. Depending on the number of iterations, this might be prohibitive. An alternative is to impose structure to obtain an *explicit* but "light-weight" approximation of the curvature matrix.

Structured Light-Weight Approximations. The simplest approach is to use only the diagonal of $\boldsymbol{C}_\mathcal{D}$ which is trivial to invert or factorize. For instance, LeCun et al. [86] approximate the diagonal of the Hessian to compute a pruning statistic (see Section 3.3), the diagonal of the FIM has been used extensively as a curvature proxy for the LA [30, 75, 115], and Adam [74] is loosely inspired

7: A somewhat related approach is L-BFGS [89]. This is a quasi-Newton method that implicitly maintains a positive definite diagonal plus low-rank approximation of the inverse Hessian [102, Sec. 7.2]. However, it does *not* use the Hessian, GGN or FIM but rather approximates the curvature based on a limited number of past *gradient* evaluations.

by natural gradient descent with a diagonal approximation of the empirical Fisher as preconditioner.[8]

The diagonal curvature approximation neglects curvature interactions between parameters and thus yields a rather crude curvature proxy. A more nuanced alternative is the *block-diagonal* approximation, which considers curvature within layers but neglects curvature between layers [10, 27, 148].

Kronecker-Factored Approximate Curvature (K-FAC). A widely-used example is the *Kronecker-Factored Approximate Curvature* (K-FAC) approach [37, 43, 96, 97] (see Figure 3.4). For the lth layer in the network, K-FAC approximates the corresponding $P^{(l)} \times P^{(l)}$ block from the FIM with a Kronecker product of two relatively small[9] square matrices $\boldsymbol{F}_{\mathcal{D}}^{(l)} \approx \boldsymbol{A} \otimes \boldsymbol{B}$. The approximation is based on the assumption that the expectation over the Kronecker product can be estimated by the Kronecker product of the expectations—a simplification unlikely to be valid in practice [97]. Due to the imposed block-diagonal structure, inverting the curvature matrix reduces to inverting the blocks which can be done efficiently using the identity $(\boldsymbol{F}_{\mathcal{D}}^{(l)})^{-1} = \boldsymbol{A}^{-1} \otimes \boldsymbol{B}^{-1}$ (and similarly for eigendecompositions [43]). K-FAC has been used successfully in practice not only for second-order optimization [97, 107], but also for approximating the LA [30, 115, 116].

For the LA, another established structural approximation is the *last-layer* approach [31, 77], which only treats the parameters of the last layer in the network probabilistically, and thus greatly reduces the size of the covariance matrix $\boldsymbol{\Sigma}$.

Mini-Batch Approximation. The computational costs for the curvature approximations presented above scale linearly with the number of training samples. While it might still be possible to use the entire training set $\mathcal{D}$ in some cases (*e.g.* when a LA has to be computed only once), this is often impractical. A straightforward way to reduce costs is to use a mini-batch $\mathcal{B} \subset \mathcal{D}$ for the curvature estimates $\boldsymbol{C}_{\mathcal{D}} \approx \boldsymbol{C}_{\mathcal{B}}$. This is analogous to the mini-batch approximation of the gradient we discussed in Section 2.2.5 and adds another layer to the hierarchy of approximations.

8: This connection is highly questionable as (i) the empirical Fisher is not an MC estimate of the FIM [82], (ii) Adam squares the *mini-batch* gradient (*i.e.* it squares a sum of gradients), whereas the diagonal of the empirical Fisher (see Equation (3.8)) is a sum over squared gradients, and (iii) Adam's update rule takes the *root* of the second moment estimate to scale the gradient. For latest insights towards partially resolving these points, see [88].

9: The size of $\boldsymbol{A}$ and $\boldsymbol{B}$ is given by the input and output dimensions of the layer, respectively.

4 Thesis Contributions

The works presented in Part II make several contributions to curvature-based methods in machine learning.

4.1 Contributions to Inference in Non-Conjugate Gaussian Processes

Section 2.1 introduced GPs as a powerful probabilistic framework for regression tasks. Other important applications like Poisson regression or classification can also be modeled using GPs but require a non-Gaussian likelihood—this model class is known as *non-conjugate Gaussian processes* (NCGPs). Inference in these models is more challenging than in the GP regression case as the posterior is no longer available in closed form. One approach is to find the maximum a posteriori estimate via Newton steps and then apply a Laplace approximation to the posterior [114, Sec. 3.4]. However, computing *exact* Newton steps (*e.g.* via Cholesky decomposition) is computationally expensive and infeasible for large data sets.

Chapter 5 introduces IterNCGP, a family of efficient inference algorithms for NCGPs based on the Laplace approximation. This method is an extension of IterGP [143] (which assumes a conjugate Gaussian likelihood) to the non-conjugate case. Like IterGP, IterNCGP is a matrix-free approach (see Section 3.4.2) which allows it to scale to large data sets. It computes approximate Newton steps using a probabilistic linear solver, *i.e.* the solver keeps track of the uncertainty over the solution of the Newton step linear system. This uncertainty can be propagated to the posterior over the latent function, which allows us to trade off reduced computation for increased uncertainty. The specific form of the log-posterior in NCGPs results in Newton step linear systems that share a lot of structure. IterNCGP exploits this and recycles information between steps for faster convergence. This requires storing some previous results in buffers. A compression mechanism is introduced to keep the required memory manageable while preserving the most relevant information.

Disclaimer 4.1 Chapter 5 is based on the peer-reviewed journal paper [135] with the following co-author contributions:

L. Tatzel, J. Wenger, F. Schneider, and P. Hennig. "Accelerating Non-Conjugate Gaussian Processes By Trading Off Computa-

tion For Uncertainty". *Transactions on Machine Learning Research (TMLR)* (2025)

	Ideas	Experiments	Analysis	Writing
L. Tatzel	35 %	75 %	60 %	50 %
J. Wenger	45 %	15 %	20 %	30 %
F. Schneider	10 %	10 %	10 %	10 %
P. Hennig	10 %	0 %	10 %	10 %

4.2 Contributions to Second-Order Methods in Deep Learning

Section 2.2.5 motivated second-order optimizers (see Section 3.1) as a *conceptually* different and potentially more powerful alternative to gradient-based first-order methods; and the LA is a powerful tool for uncertainty quantification in Bayesian deep learning. However, the application of those curvature-based methods to deep learning is challenging for several reasons:

1. **Non-Convex Loss Landscapes.** Newton steps and the LA require a spd curvature proxy. However, the loss landscape of deep learning models is typically non-convex. Section 3.4.1 therefore introduced principled positive semidefinite alternatives to the Hessian, *e.g.* the GGN and FIM.
2. **Computational Costs.** Curvature matrices are quadratic in the number of parameters and thus often infeasible to store ($\mathcal{O}(P^2)$) and invert/decompose ($\mathcal{O}(P^3)$). To mitigate this, matrix-free approaches or structural approximations to the GGN or FIM can be used, *e.g.* K-FAC (see Section 3.4.2).
3. **Accessibility of Curvature Information.** Curvature proxies like the GGN are often not readily available in standard AD frameworks like PyTorch. This hinders the practical application and further development of curvature-based methods. Several open-source libraries address this limitation, *e.g.* BackPACK [28], ASDL [106], and CurvLinOps [26].
4. **Stochasticity.** In Deep Learning, we have to rely on Monte Carlo estimates of the gradient and curvature. These are typically computed on a small mini-batch of data, introducing a considerable amount of noise into the estimates. Practical second-order optimizers like K-FAC [97] try to mitigate this by using exponential averages to emulate larger mini-batches, and curvature damping.

Chapter 6 targets Items 2 to 4. It introduces ViViT, a method for computing eigendecompositions of the GGN, per-sample directional derivatives of the implied quadratic model, and Newton steps. ViViT achieves computational efficiency by leveraging the

GGN's inherent low-rank structure (Item 2). While K-FAC uses a block-diagonal approximation that does not become exact in any limit, ViViT uses the full GGN, and offers principled mechanisms to balance computational costs against approximation accuracy. The method is implemented on top of BackPACK [28] within the PyTorch [111] ecosystem, and publicly available (Item 3). Through per-sample directional derivatives, ViViT also provides a notion of curvature *noise* (Item 4). This is important to better understand the challenges of stochastic second-order methods and a prerequisite for developing novel "uncertainty-aware" methods.

Disclaimer 4.2 Chapter 6 is based on the peer-reviewed journal paper [29] 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)* (2023)

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

*Equal contribution

Chapter 7 focuses on Item 4. While practical optimizers like K-FAC already use mechanisms to counteract the stochasticity of the curvature and gradient estimates, the underlying *principles* remain poorly understood. We therefore investigate how the fundamental building block of second-order methods—the quadratic Taylor approximation of the loss landscape—is affected by the stochasticity due to mini-batching and find a general principle: Quadratic approximations based on mini-batches of the training data provide a deformed representation of the true underlying loss landscape. In particular, the mini-batch quadratic tends to strongly overestimate the curvature of the true loss along the directions of large curvature. This bias introduces a systematic error that can be traced back to the misalignment of the curvature matrices' eigenspaces. These insights are highly relevant for applications: The bias can lead to detrimental updates in second-order optimizers and unreliable uncertainty estimates with the LA. Consequently, we develop debiasing strategies and demonstrate their effectiveness.

Disclaimer 4.3 Chapter 7 is based on the peer-reviewed conference paper [134] with the following co-author contributions:

L. Tatzel, B. Mucsányi, O. Hackel, and P. Hennig. "Debiasing Mini-Batch Quadratics for Applications in Deep Learning".

International Conference on Learning Representations (ICLR). 2025

	Ideas	Experiments	Analysis	Writing
L. Tatzel	70 %	70 %	65 %	70 %
B. Mucsányi	15 %	25 %	20 %	15 %
O. Hackel	5 %	5 %	5 %	5 %
P. Hennig	10 %	0 %	10 %	10 %

4.3 List of Publications

A complete list of publications that was (co-)authored during the PhD is given below.

[133] L. Tatzel, P. Hennig, and F. Schneider. "Late-Phase Second-Order Training". *Has it Trained Yet? NeurIPS 2022 Workshop*. 2022

[29] 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)* (2023)

[117] H. Roy, M. Miani, C. H. Ek, P. Hennig, M. Pförtner, L. Tatzel, and S. Hauberg. "Reparameterization invariance in approximate Bayesian inference". *Advances in Neural Information Processing Systems (NeurIPS)*. 2024

[135] L. Tatzel, J. Wenger, F. Schneider, and P. Hennig. "Accelerating Non-Conjugate Gaussian Processes By Trading Off Computation For Uncertainty". *Transactions on Machine Learning Research (TMLR)* (2025)

[134] L. Tatzel, B. Mucsányi, O. Hackel, and P. Hennig. "Debiasing Mini-Batch Quadratics for Applications in Deep Learning". *International Conference on Learning Representations (ICLR)*. 2025

[26] F. Dangel, R. Eschenhagen, W. Ormaniec, A. Fernandez, L. Tatzel, and A. Kristiadi. "Position: Curvature Matrices Should Be Democratized via Linear Operators". *arXiv* (2025)

Part II

Advancing Curvature-Based Methods in Machine Learning

5 Accelerating Non-Conjugate Gaussian Processes by Trading Off Computation for Uncertainty

Abstract

Non-conjugate Gaussian processes (NCGPs) define a flexible probabilistic framework to model categorical, ordinal and continuous data, and are widely used in practice. However, exact inference in NCGPs is prohibitively expensive for large data sets, thus requiring approximations in practice. The approximation error adversely impacts the reliability of the model and is not accounted for in the uncertainty of the prediction. We introduce a family of iterative methods that explicitly model this error. They are uniquely suited to parallel modern computing hardware, efficiently recycle computations, and compress information to reduce both the time and memory requirements for NCGPs. As we demonstrate on large-scale classification problems, our method significantly accelerates posterior inference compared to competitive baselines by trading off reduced computation for increased uncertainty.

5.1 Introduction

Non-conjugate Gaussian processes[1] (NCGPs) form a fundamental interpretable model class widely used throughout the natural and social sciences. For example, NCGPs are applied to count data in biomedicine, categorical data in object classification tasks, and continuous data in time series regression. An NCGP assumes the data is generated from an exponential family likelihood with a Gaussian process (GP) prior over the latent function. Such a *probabilistic* approach is essential in domains where critical decisions must be made based on limited information, such as in public policy, medicine or robotics.

Unfortunately, even the conjugate Gaussian case, where fitting NCGPs reduces to GP regression, naively has cubic time complexity $\mathcal{O}(N^3)$ in the number of training data N and requires $\mathcal{O}(N^2)$ memory, which is prohibitive for modern large-scale data sets. For non-Gaussian likelihoods, inference has to be done approximately, which generally exacerbates this problem. For example, inference via the Laplace approximation (LA) boils down to finding the mode of the log posterior via Newton's method, which is equivalent to solving a *sequence* of regression problems [9, 91, 131].

1: Such a model is also called Generalized Gaussian Process Model [15] or Generalized Linear Model [100]. The latter name is sometimes used only for latent Gaussian models, which can lead to confusion. The models studied in this work are generally of *nonparametric* nature. The resulting large linear problems are the main reason why the algorithms we propose are relevant in the first place.

Section 2.1 provides a more detailed introduction to Gaussian processes and how they can be applied to regression tasks.

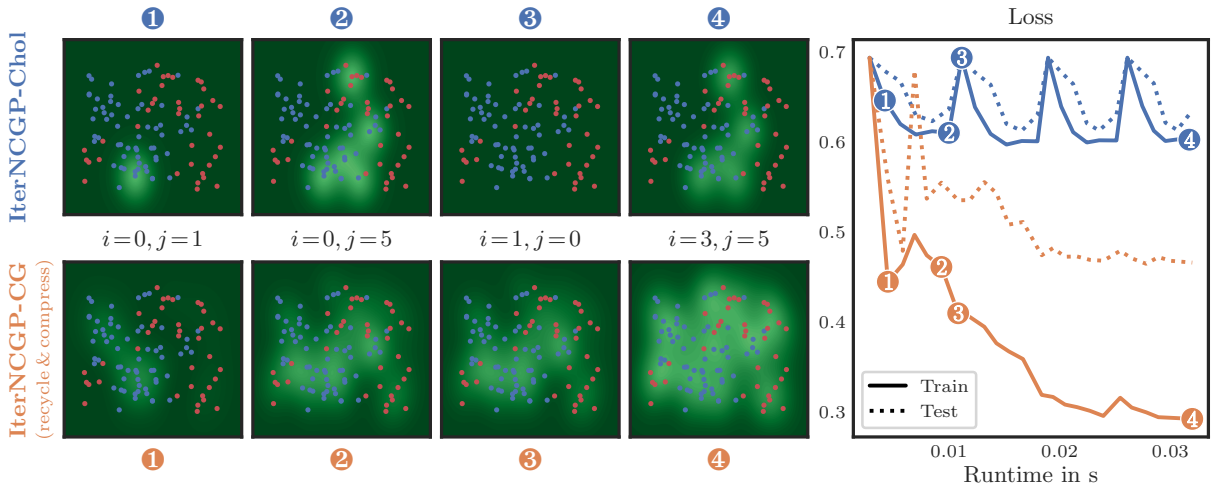


Figure 5.1: Binary Classification with IterNCGP. Comparison of two IterNCGP variants: *(Top)* IterNCGP variant corresponding to data subsampling and solving each regression problem exactly in each Newton step i. *(Bottom)* IterNCGP variant with a more informative policy (details in Section 5.3.2), recycling of computations between Newton steps (details in Section 5.3.3) and compression to reduce memory (details in Section 5.3.4). The panels show the marginal uncertainty ( ) over the latent function at Newton step i and solver iteration j. Using recycling, the current belief is efficiently propagated between mode-finding steps i (❷→❸) without performance drops *(Right)*. Details in Appendix A.3.1.

Due to limited computational resources, large-scale problems often require approximations. The resulting error affects a model's predictive accuracy but also its uncertainty quantification. Hence, the question arises: **Can NCGPs be efficiently trained on extensive data without compromising reliability?**

Recently, iterative methods have emerged which in the conjugate Gaussian case allow an explicit, tunable trade-off between reduced computation and increased uncertainty [139, 143]. This computational uncertainty quantifies the inevitable approximation error in the sense of probabilistic numerics [19, 57, 58, 104].

Contributions. In this work, we take a similar approach and extend Wenger et al. [143]'s IterGP (that assumes a *conjugate* Gaussian likelihood) to *non-conjugate* exponential family likelihoods. This is a non-trivial extension, as the posterior is no longer Gaussian and *multiple* related regression problems have to be solved. Specifically, we propose **(i)** IterNCGP: a family of efficient inference algorithms for NCGPs with a tunable trade-off between computational savings and added uncertainty (Section 5.3.1) with **(ii)** mechanisms to tailor the inference algorithm to a specific downstream application (Section 5.3.2). In response to the specific computational challenges in the non-conjugate setting, we develop **(iii)** novel strategies to optimally recycle costly computations (Section 5.3.3) and **(iv)** to restrict the memory usage, with minimal impact on inference (Section 5.3.4).

Our algorithm IterNCGP consists of two nested loops: An outer loop (indexed by i) which iterates over Newton steps/GP regression

problems, each of which is solved approximately via an inner loop (indexed by j) that implements a probabilistic linear solver. Figure 5.1 shows the marginal uncertainty over the latent function at different stages of that process and illustrates the effectiveness of (ii), (iii) and (iv). Specifically, by recycling computations between Newton steps, we are able to traverse the two-loop structure "diagonally" which leads to steady progress without performance drops between Newton steps.

5.2 Background

Let $(\boldsymbol{X}, \boldsymbol{y})$ be a data set of N input vectors $\{\boldsymbol{x}_n \in \mathbb{X}\}_{n=1}^N$ stacked into $\boldsymbol{X} = (\boldsymbol{x}_1, \dots \boldsymbol{x}_N)^\intercal \in \mathbb{R}^{N \times D}$ and corresponding outputs $\boldsymbol{y} = (y_1, \dots, y_N)^\intercal \in \mathbb{Y}^N$, where $\mathbb{X} \subseteq \mathbb{R}^D$ and $\mathbb{Y} \subseteq \mathbb{R}$ or $\mathbb{Y} \subseteq \mathbb{N}_0$ (regression) or $\mathbb{Y} = \{1, \dots, C\}$ (classification).

5.2.1 Non-Conjugate Gaussian Processes (NCGPs)

We consider the probabilistic model $p(\boldsymbol{y}, \boldsymbol{f} \mid \boldsymbol{X}) = p(\boldsymbol{y} \mid \boldsymbol{f})\, p(\boldsymbol{f} \mid \boldsymbol{X})$, where the vector $\boldsymbol{f} \coloneqq f(\boldsymbol{X}) \in \mathbb{R}^{NC}$ is given by a latent function $f : \mathbb{X} \to \mathbb{R}^C$ evaluated at the training data.

Prior. Assume a multi-output Gaussian process prior $\mathcal{GP}(m, K)$ over the latent function with mean function $m : \mathbb{X} \to \mathbb{R}^C$ and kernel function $K : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^{C \times C}$. Therefore, the latent vector has density $p(\boldsymbol{f} \mid \boldsymbol{X}) = \mathcal{N}(\boldsymbol{f}; \boldsymbol{m}, \boldsymbol{K})$ with mean $\boldsymbol{m} \coloneqq m(\boldsymbol{X}) \in \mathbb{R}^{NC}$ and covariance $\boldsymbol{K} = K(\boldsymbol{X}, \boldsymbol{X}) \in \mathbb{R}^{NC \times NC}$ defined by N^2 blocks $K(\boldsymbol{x}_i, \boldsymbol{x}_j) \in \mathbb{R}^{C \times C}$. Each such block represents the covariance between the C latent functions evaluated at inputs $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$.

Likelihood. Assume *i.i.d.* data, which depends on the latent function via an inverse link function $\lambda : \mathbb{R}^C \to \mathbb{R}^C$, such that $p(\boldsymbol{y} \mid \boldsymbol{f}) = \prod_{n=1}^N p(y_n \mid \lambda(\boldsymbol{f}_n))$, where $p(y_n \mid \lambda(\boldsymbol{f}_n))$ is a log-concave likelihood, *e.g.* any exponential family distribution.[2] For example, for Poisson regression the inverse link function is given by $\lambda(f_n) = \exp(f_n)$ and for multi-class classification by $\lambda(\boldsymbol{f}_n) = \mathrm{softmax}(\boldsymbol{f}_n)$.

2: The Hessian of an exponential family likelihood is the negative Hessian of its log-partition function, which equals the *positive definite* covariance matrix of its sufficient statistics.

For nonlinear inverse link functions, the posterior $p(f \mid \boldsymbol{X}, \boldsymbol{y})$ and predictive distribution $p(y_\diamond \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x}_\diamond) = \int p(y_\diamond \mid f_\diamond) p(f_\diamond \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x}_\diamond)\, \mathrm{d}f_\diamond$ are computationally intractable, requiring approximations.

5.2.2 Approximate Inference via Laplace

A popular way to perform approximate inference in an NCGP is to use a Laplace approximation (LA) [9, 91, 114, 131]. The idea is to approximate the posterior

$$p(\boldsymbol{f} \mid \boldsymbol{X}, \boldsymbol{y}) \approx q(\boldsymbol{f} \mid \boldsymbol{X}, \boldsymbol{y}) \coloneqq \mathcal{N}\big(\boldsymbol{f}; \boldsymbol{f}_\star, \boldsymbol{\Sigma}\big), \tag{5.1}$$

with a Gaussian with mean given by the mode $\boldsymbol{f}_\star$ of the log-posterior and covariance $\boldsymbol{\Sigma} \coloneqq -(\nabla^2 \log p(\boldsymbol{f}_\star \mid \boldsymbol{X}, \boldsymbol{y}))^{-1}$ given by the negative inverse Hessian (with respect to $\boldsymbol{f}$) at the mode. Due to the assumed GP prior over the latent function, the log-posterior is given by

Section 3.2 provides a detailed introduction to the Laplace approximation.

$$\begin{aligned}\Psi(\boldsymbol{f}) &\coloneqq \log p(\boldsymbol{f} \mid \boldsymbol{X}, \boldsymbol{y}) \\ &\overset{c}{=} \log p(\boldsymbol{y} \mid \boldsymbol{f}) + \log p(\boldsymbol{f} \mid \boldsymbol{X}) \\ &\overset{c}{=} \log p(\boldsymbol{y} \mid \boldsymbol{f}) - \frac{1}{2}(\boldsymbol{f} - \boldsymbol{m})^\top \boldsymbol{K}^{-1}(\boldsymbol{f} - \boldsymbol{m})\end{aligned} \tag{5.2}$$

We use $\overset{c}{=}$ to denote equality up to an additive constant.

Mode-Finding via Newton's Method. To find the mode $\boldsymbol{f}_\star$, one typically uses Newton steps [114, Sec. 3.4], *i.e.*

Section 3.1 provides a detailed introduction to second-order optimization methods.

$$\begin{aligned}\boldsymbol{f}_\star \approx \boldsymbol{f}_{i+1} &= \boldsymbol{f}_i - \nabla^2\Psi(\boldsymbol{f}_i)^{-1} \cdot \nabla\Psi(\boldsymbol{f}_i), \\ \text{where} \quad \nabla\Psi(\boldsymbol{f}_i) &= \nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i) - \boldsymbol{K}^{-1}(\boldsymbol{f}_i - \boldsymbol{m}) \\ \text{and} \quad \nabla^2\Psi(\boldsymbol{f}_i) &= -\boldsymbol{W}(\boldsymbol{f}_i) - \boldsymbol{K}^{-1}.\end{aligned} \tag{5.3}$$

The negative Hessian $\boldsymbol{W}(\boldsymbol{f}_i) \coloneqq -\nabla^2 \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)$ of the log-likelihood at $\boldsymbol{f}_i$ is positive definite for all $\boldsymbol{f}$, since we assumed a log-concave likelihood. Therefore Ψ is concave and the Newton updates are well-defined.

5.2.3 Predictions

Using a local quadratic Taylor approximation of the log-posterior Ψ around the current iterate $\boldsymbol{f}_i$, we obtain the LA $\mathcal{N}(\boldsymbol{f}; \boldsymbol{f}_{i+1}, -\nabla^2\Psi(\boldsymbol{f}_i)^{-1})$ whose mean is given by the maximizer of the local quadratic, *i.e.* the subsequent Newton iterate $\boldsymbol{f}_{i+1}$. Substituting this in place of the posterior, the predictive distribution for the latent function $p(f(\cdot) \mid \boldsymbol{X}, \boldsymbol{y}) = \int p(f(\cdot) \mid \boldsymbol{f})\, \mathcal{N}(\boldsymbol{f}; \boldsymbol{f}_{i+1}, -\nabla^2\Psi(\boldsymbol{f}_i)^{-1})\, \mathrm{d}\boldsymbol{f}$ is a Gaussian process $\mathcal{GP}(m_{i,*}, K_{i,*})$, with mean and covariance functions

$$m_{i,*}(\cdot) \coloneqq m(\cdot) + K(\cdot, \boldsymbol{X})\boldsymbol{K}^{-1}(\boldsymbol{f}_{i+1} - \boldsymbol{m}), \tag{5.4}$$

$$K_{i,*}(\cdot, \cdot) \coloneqq K(\cdot, \cdot) - K(\cdot, \boldsymbol{X})\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1}K(\boldsymbol{X}, \cdot), \tag{5.5}$$

where $\hat{\boldsymbol{K}}(\boldsymbol{f}_i) := \boldsymbol{K} + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}$ [114, Eq. (3.24)]. We obtain the predictive distribution for $y_\diamond$ at test input $\boldsymbol{x}_\diamond$ by integrating this approximative posterior against the likelihood, *i.e.* $p(y_\diamond \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x}_\diamond) = \int p(y_\diamond \mid \boldsymbol{f}_\diamond)\, p(\boldsymbol{f}_\diamond \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x}_\diamond)\, \mathrm{d}\boldsymbol{f}_\diamond$. This C-dimensional integral can be approximated via quadrature, MC-sampling or specialized approaches (like the probit method [91] for a categorical likelihood and softmax inverse link function).

5.3 Computation-Aware Inference in NCGPs

While Newton's method typically converges in a few steps for a log-concave likelihood, each step in (5.3) requires linear system solves with symmetric positive (semi-)definite matrices of size $NC \times NC$. Naively computing these solves via Cholesky decomposition is problematic even for moderately sized data sets due to its cubic time $\mathcal{O}(N^3C^3)$ and quadratic memory complexity $\mathcal{O}(N^2C^2)$. We will demonstrate in the following how to circumvent this issue by reducing the computations in exchange for increased uncertainty about the latent function.

5.3.1 Derivation of the IterNCGP Framework

Overview. As a first step, we reinterpret the posterior predictive mean (Equation (5.4)) as the GP posterior for a specific regression problem (Equation (5.6)). The sequence of regression problems is connected to the sequence of Newton steps and forms the outer loop of our algorithm IterNCGP, indexed by i (Algorithm 5.1). The formulation as GP regression problem allows us to apply IterGP [143] as an inner loop, indexed by j (Algorithm 5.2). By using IterGP, we can solve each regression problem approximately and quantify the resulting error in the form of additional uncertainty (Equation (5.9)).

Outer Loop: Newton's Method as Sequential GP Regression. Through the LA, we obtain a posterior predictive $f \sim \mathcal{GP}(m_{i,*}, K_{i,*})$ over the latent function for each Newton step. As we show in Appendix A.1.1, the posterior predictive (Equations (5.4) and (5.5)) in step i can be written as

$$m_{i,*}(\cdot) = m(\cdot) + K(\cdot, \boldsymbol{X})\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1}(\hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m}) \tag{5.6}$$

$$K_{i,*}(\cdot,\cdot) = K(\cdot,\cdot) - K(\cdot, \boldsymbol{X})\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1}K(\boldsymbol{X}, \cdot), \tag{5.7}$$

where $\hat{\boldsymbol{y}}(\boldsymbol{f}_i) := \boldsymbol{f}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)$. Equations (5.6) and (5.7) have the exact form of the posterior for a GP regression problem with fictitious *pseudo-targets* $\hat{\boldsymbol{y}}(\boldsymbol{f}_i)$ observed with Gaussian

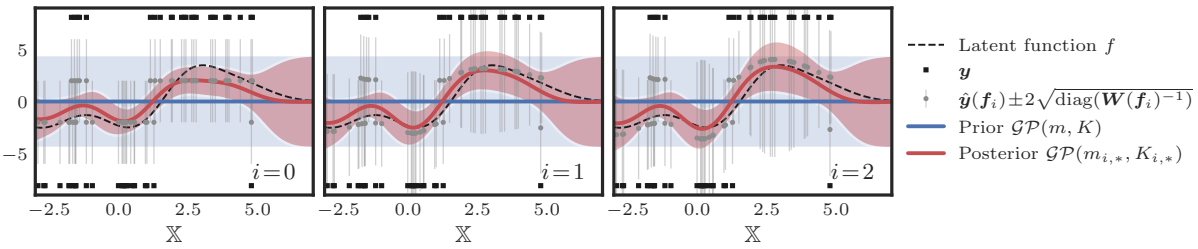


Figure 5.2: Approximate Inference in NCGPs as Sequential GP Regression. Performing a LA at a Newton iterate f_i results in a posterior GP that coincides with the posterior to a GP regression problem with pseudo-targets $\hat{y}(f_i)$ observed with Gaussian noise $\mathcal{N}(0, W(f_i)^{-1})$. The plot shows an illustration of this connection for binary classification on a toy problem with the latent function drawn from a GP. Notice how similar the posteriors are between Newton steps. This motivates our proposed strategy for recycling computations between steps in Section 5.3.3. Details in Appendix A.3.1.

Algorithm 5.1: IterNCGP Outer Loop.

Input: GP prior $\mathcal{GP}(m, K)$, training data (X, y), $\nabla p(y|f, X)$ and access to products with K and $W(f)^{-1}$

Output: GP posterior $\mathcal{GP}(m_{i,j}, K_{i,j})$

	Procedure: IterNCGP$(m, K, X, y, f_0 = m)$		**Time**	**Memory**
1	$m \leftarrow m(X)$	m : Prior mean vector	$\mathcal{O}(t_m)$	$\mathcal{O}(NC)$
2	Provide access to $w \mapsto Kw$	K : Prior covariance/kernel matrix		$\mathcal{O}(\mu_K)$
3	Initialize buffers $S, T \in \mathbb{R}^{NC\times 0}$	S: actions, T: products with K		
4	**for** $i = 0, 1, 2, \ldots$ **while not** OuterStoppingCriterion() **do**			
5	Provide access to $w \mapsto W(f_i)^{-1}w$	$W(f_i)^{-1}$: Observation noise		$\mathcal{O}(\mu_{W^{-1}})$
6	$\hat{y}(f_i) \leftarrow f_i + W(f_i)^{-1}\nabla \log p(y|f_i)$	$\hat{y}(f_i)$: Pseudo-targets	$\mathcal{O}(t_{W^{-1}}+NC)$	$\mathcal{O}(NC)$
7	$\mathcal{GP}(m_{i,j}, K_{i,j}), v_j \leftarrow$ IterGP$(m, K, X, y, m, K, W(f_i)^{-1}, \hat{y}(f_i), S, T)$			
8	$f_{i+1} \leftarrow Kv_j + m$	Approximate Newton update	$\mathcal{O}(t_K+NC)$	$\mathcal{O}(NC)$
9	**return** $\mathcal{GP}(m_{i,j}, K_{i,j})$			

The IterGP algorithm is given in Algorithm 5.2. Instructions in blue are needed for recycling (see Section 5.3.3). The matrices K and $W^{-1}(f_i)$ are evaluated lazily. We thus report the run time costs when the matrix-vector products are actually computed. For an in-depth discussion of the computational costs, see Appendix A.2.3.

noise $\mathcal{N}(0, W(f_i)^{-1})$.[3] Figure 5.2 shows an illustration of this interpretation.

Equation (5.6) requires solving a linear system $\hat{K}(f_i)\, v = \hat{y}(f_i) - m$ of size $NC \times NC$. The posterior mean is then simply given by $m_{i,*}(\cdot) = m(\cdot) + K(\cdot, X)v$. However, also the Newton update from Equation (5.3) follows directly from v since $f_{i+1} = Kv + m$. In that sense, computing the posterior predictive mean and performing Newton updates are equivalent. The sequence of Newton steps/GP regression problems forms the outer loop of our algorithm IterNCGP. The pseudocode is given in Algorithm 5.1.

Inner Loop: Computation-Aware GP Regression via IterGP. Reframing the Newton iteration as sequential GP regression does not yet solve the need for linear solves with a matrix of size $NC \times NC$. However, it allows us to leverage recent advances for GP regression, specifically the IterGP algorithm introduced by Wenger et al. [143]. IterGP is matrix-free, *i.e.* only relies on matrix-vector products $s \mapsto \hat{K}(f_i)\, s$, reducing the required memory from quadratic to

3: If $W(f_i)^{-1}$ does not exist, *e.g.* in multi-class classification, we substitute its pseudo-inverse $W(f_i)^{\dagger}$, which for multi-class classification can be evaluated efficiently (see Appendix A.1.6). Alternatively, one can place a prior on the sum of the C latent functions [92, Eq. (10)].

linear, and efficiently exploits modern parallel GPU hardware [16].

Internally, IterGP uses a probabilistic linear solver (PLS) [18, 56, 141] to iteratively compute a Gaussian belief over the so-called *representer weights*, *i.e.* over the solution of the linear system $\hat{\boldsymbol{K}}(\boldsymbol{f}_i)\, v = \hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m}$. In each solver iteration, indexed by j, the belief $\mathcal{N}(v; v_j, \boldsymbol{\Omega}_j)$ is updated by conditioning the Gaussian on a one-dimensional projection $\alpha_j = \boldsymbol{s}_j^\top \boldsymbol{r}_{j-1}$ of the preceding residual $\boldsymbol{r}_{j-1} = \hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m} - \hat{\boldsymbol{K}}(\boldsymbol{f}_i)\boldsymbol{v}_{j-1}$. The vector $\boldsymbol{s}_j \leftarrow \textsc{Policy}()$ is called an *action* and is generated by a user-specified *policy*. The policy determines the behavior of the solver by "weighting" the residual $\boldsymbol{r}_{j-1}$ for specific data points (we discuss the role of the policy in the context of NCGPs in Section 5.3.2.). Wenger et al. [143, Tab. 1] lists policies that have classic counterparts, *e.g.* unit vectors $\boldsymbol{s}_j \leftarrow \boldsymbol{e}_j$ correspond to partial Cholesky and residual actions $\boldsymbol{s}_j \leftarrow \boldsymbol{r}_{j-1}$ to conjugate gradients [61].

The belief over the representer weights translates into an approximate GP posterior over the latent function,

$$m_{i,j}(\cdot) \coloneqq m(\cdot) + K(\cdot, \boldsymbol{X})\boldsymbol{v}_j \tag{5.8}$$

$$K_{i,j}(\cdot,\cdot) \coloneqq K(\cdot,\cdot) - K(\cdot, \boldsymbol{X})\, \boldsymbol{C}_j K(\boldsymbol{X}, \cdot), \tag{5.9}$$

where $\boldsymbol{v}_j = \boldsymbol{C}_j(\hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m})$. Crucially, by Wenger et al. [143, Thm. 2], the posterior covariance in Equation (5.9) *exactly* quantifies the error in each approximate Newton step introduced by only using *limited computational resources*, *i.e.* running the linear solver for $j \ll NC$ iterations. This reduces the time complexity to $\mathcal{O}(jN^2C^2)$. The approximate precision matrix in Equations (5.8) and (5.9),

$$\boldsymbol{C}_j = \boldsymbol{S}_j(\boldsymbol{S}_j^\top \hat{\boldsymbol{K}}(\boldsymbol{f}_i)\boldsymbol{S}_j)^{-1}\boldsymbol{S}_j^\top \tag{5.10}$$

with $\boldsymbol{S}_j = (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_j) \in \mathbb{R}^{NC\times j}$, has rank j and approaches $\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1}$ as $j \to NC$. Intuitively, it projects $\hat{\boldsymbol{K}}(\boldsymbol{f}_i)$ onto the subspace spanned by the actions $\boldsymbol{S}_j$, then inverts and projects the result back into the original space. The IterGP algorithm is given in Algorithm 5.2.

The Marginal Uncertainty Decreases in the Inner Loop. As we perform more solver iterations, the marginal uncertainty captured by the posterior covariance (Equation (5.9)) contracts, *i.e.* for each i it holds (element-wise) that $\operatorname{diag}(K_{i,j}(\boldsymbol{x},\boldsymbol{x})) \geq \operatorname{diag}(K_{i,k}(\boldsymbol{x},\boldsymbol{x}))$ for any $k \geq j$ and arbitrary $\boldsymbol{x}$. This is because the approximate precision matrix $\boldsymbol{C}_j$ grows in rank with each solver iteration. For a detailed derivation, see Appendix A.1.3.

Summary. Finding the posterior mode $\boldsymbol{f}_\star$ and the corresponding predictive distributions (Equations (5.4) and (5.5)) can be viewed

Algorithm 5.2: IterNCGP Inner Loop: IterGP with a Virtual Solver Run.

Input: GP prior $\mathcal{GP}(m, K)$, training data $(\boldsymbol{X}, \boldsymbol{y})$, $\boldsymbol{m}$, access to products with $\boldsymbol{K}$ and $\boldsymbol{W}^{-1}$, pseudo-targets $\hat{\boldsymbol{y}}$, buffers $\boldsymbol{S}, \boldsymbol{T}$
Output: GP posterior $\mathcal{GP}(m_{i,j}, K_{i,j})$

	Procedure: IterGP$(m, K, \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{m}, \boldsymbol{K}, \boldsymbol{W}^{-1}, \hat{\boldsymbol{y}}, \boldsymbol{S}, \boldsymbol{T})$		**Time**	**Memory**
1	$\boldsymbol{C}_0, \boldsymbol{S}, \boldsymbol{T} \leftarrow$ VirtualSolverRun$(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{W}^{-1})$	See Algorithm 5.3		
2	$\boldsymbol{v}_0 \leftarrow \boldsymbol{C}_0(\hat{\boldsymbol{y}} - \boldsymbol{m})$	$\boldsymbol{v}_0$: Consistent initial iterate	$\mathcal{O}(RNC)$	$\mathcal{O}(NC)$
3	**for** $j = 0, 1, 2, \ldots$ **while not** InnerStoppingCriterion() **do**			
4	$\quad \boldsymbol{r}_{j-1} \leftarrow (\hat{\boldsymbol{y}} - \boldsymbol{m}) - \boldsymbol{K}\boldsymbol{v}_{j-1} - \boldsymbol{W}^{-1}\boldsymbol{v}_{j-1}$	$\boldsymbol{r}_{j-1}$: Residual vector	$\mathcal{O}(t_{\boldsymbol{K}} + t_{\boldsymbol{W}^{-1}} + NC)$	$\mathcal{O}(NC)$
5	$\quad \boldsymbol{s}_j \leftarrow$ Policy()	Select action $\boldsymbol{s}_j$ via policy	$\mathcal{O}(t_{\text{POLICY}})$	$\mathcal{O}(NC)$
6	$\quad \boldsymbol{S} \leftarrow (\boldsymbol{S}, \boldsymbol{s}_j) \in \mathbb{R}^{NC \times B}$	Append $\boldsymbol{s}_j$ to buffer		$\mathcal{O}(BNC)$
7	$\quad \alpha_j \leftarrow \boldsymbol{s}_j^\top \boldsymbol{r}_{j-1}$	α_j : Observation is projection of residual onto action	$\mathcal{O}(NC)$	$\mathcal{O}(1)$
8	$\quad \boldsymbol{t}_j \leftarrow \boldsymbol{K}\boldsymbol{s}_j$	First term in $\hat{\boldsymbol{K}}\boldsymbol{s}_j = \boldsymbol{K}\boldsymbol{s}_j + \boldsymbol{W}^{-1}\boldsymbol{s}_j$	$\mathcal{O}(t_{\boldsymbol{K}})$	$\mathcal{O}(NC)$
9	$\quad \boldsymbol{T} \leftarrow (\boldsymbol{T}, \boldsymbol{t}_j) \in \mathbb{R}^{NC \times B}$	Append $\boldsymbol{t}_j$ to buffer		$\mathcal{O}(BNC)$
10	$\quad \boldsymbol{z}_j \leftarrow \boldsymbol{t}_j + \boldsymbol{W}^{-1}\boldsymbol{s}_j$	Second term in $\hat{\boldsymbol{K}}\boldsymbol{s}_j = \boldsymbol{K}\boldsymbol{s}_j + \boldsymbol{W}^{-1}\boldsymbol{s}_j$	$\mathcal{O}(t_{\boldsymbol{W}^{-1}} + NC)$	$\mathcal{O}(NC)$
11	$\quad \boldsymbol{d}_j \leftarrow \boldsymbol{s}_j - \boldsymbol{C}_{j-1}\boldsymbol{z}_j$	$\boldsymbol{d}_j$: Search direction	$\mathcal{O}(BNC)$	$\mathcal{O}(NC)$
12	$\quad \eta_j \leftarrow \boldsymbol{z}_j^\top \boldsymbol{d}_j$	η_j : Normalization constant	$\mathcal{O}(NC)$	$\mathcal{O}(1)$
13	$\quad \boldsymbol{Q}_j \leftarrow (\boldsymbol{Q}_{j-1}, 1/\sqrt{\eta_j}\,\boldsymbol{d}_j) \in \mathbb{R}^{NC \times B}$	Append column	$\mathcal{O}(NC)$	$\mathcal{O}(BNC)$
14	$\quad \boldsymbol{C}_j \leftarrow \boldsymbol{Q}_j \boldsymbol{Q}_j^\top$	Rank B approximation $\boldsymbol{C}_j \approx \hat{\boldsymbol{K}}^{-1}$		
15	$\quad \boldsymbol{v}_j \leftarrow \boldsymbol{v}_{j-1} + \frac{\alpha_j}{\eta_j}\boldsymbol{d}_j$	$\boldsymbol{v}_j$: Updated representer weights estimate	$\mathcal{O}(NC)$	$\mathcal{O}(NC)$
16	$\quad m_{i,j}(\cdot) \leftarrow m(\cdot) + K(\cdot, \boldsymbol{X})\boldsymbol{v}_j$	Equation (5.8)	$\mathcal{O}(NN_\diamond C^2)$	$\mathcal{O}(N_\diamond C)$
17	$\quad K_{i,j}(\cdot, \cdot) \leftarrow K(\cdot, \cdot) - K(\cdot, \boldsymbol{X})\boldsymbol{C}_j K(\boldsymbol{X}, \cdot)$	Equation (5.9)	$\mathcal{O}(B(N+N_\diamond)N_\diamond C^2)$	$\mathcal{O}(N_\diamond^2 C^2)$
18	**return** $\mathcal{GP}(m_{i,j}, K_{i,j})$ and $\boldsymbol{v}_j$			

Instructions in blue are needed for recycling (see Section 5.3.3). $\boldsymbol{C}_j$ is represented via its root $\boldsymbol{Q}_j$ and evaluated lazily. We thus report the run time costs when the matrix-vector products are actually computed. The costs for evaluating the posterior GP $\mathcal{GP}(m_{i,j}, K_{i,j})$ are based on $N_\diamond$ test data points $\boldsymbol{X}_\diamond \in \mathbb{R}^{N_\diamond \times D}$. For an in-depth discussion of the computational costs, see Appendix A.2.3.

from different angles. Through an optimization lens, we use Newton updates, each maximizing a local quadratic approximation of the log-posterior. From a probabilistic perspective, we solve a sequence of *related* GP regression problems and IterGP enables us to propagate a probabilistic estimate of the latent function throughout the *entire* optimization process.

For Gaussian likelihoods, the LA (Equation (5.1)) is exact and a single Newton step suffices. Consequently, our framework generalizes IterGP to arbitrary log-concave likelihoods (Theorem A.2). We now explore the role of the policy and its potential for actions tailored to specific problems (Section 5.3.2). We also leverage the relatedness of GP regression problems in the outer loop for further speedups (Section 5.3.3) and introduce a mechanism to control IterNCGP's memory usage (Section 5.3.4).

5.3.2 Policy Choice: Targeted Computations

Algorithm 5.2 defines a *family* of inference algorithms. Its instances, defined by a concrete action policy, generally behave quite differ-

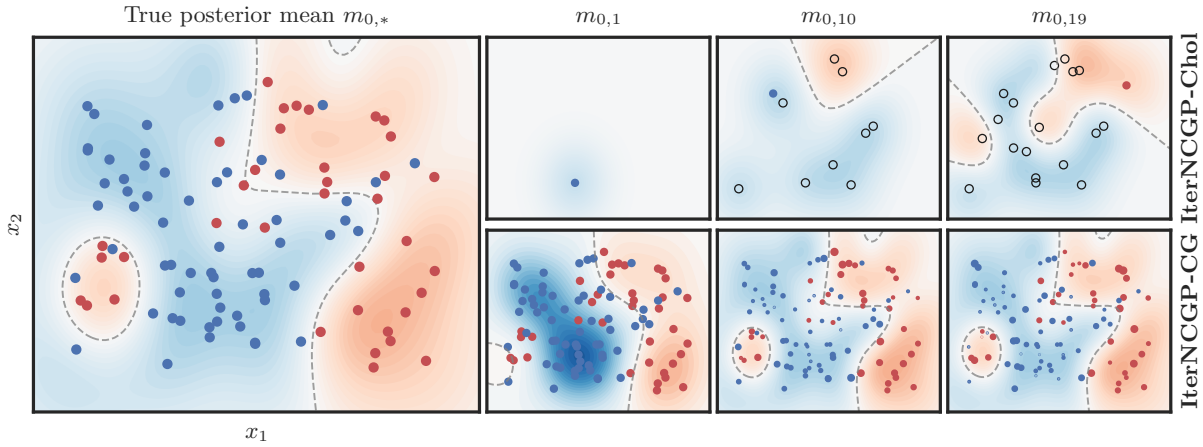


Figure 5.3: Different IterNCGP Policies Applied to GP Classification. *(Left)* The true posterior mean $m_{0,*}$ ( ) for binary classification (●/●) and its decision boundary (- -). *(Right)* Current posterior mean estimate after 1, 10, and 19 iterations with the unit vector policy *(Top)* and the CG policy *(Bottom)*. Shown are the data points selected by the policy in this iteration with the dot size indicating their relative weight. For IterNCGP-Chol, data points are targeted one by one and previously used data points are marked with (○). Details in Appendix A.3.1

ently. To better understand what effect the sequence of actions $\boldsymbol{S}_j = (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_j) \in \mathbb{R}^{NC \times j}$ has on IterNCGP, we consider the following examples.

Unit Vector Policy = Subset of Data (SoD). Choosing the actions $\boldsymbol{s}_j = \boldsymbol{e}_j$ to be unit vectors with all zeros except for a one at entry j, corresponds to (sequentially) conditioning on the first j data points in the training data in each GP regression subproblem, since $K(\cdot, \boldsymbol{X})\boldsymbol{S}_j = K(\cdot, \boldsymbol{X}_{1:j})$. Therefore this policy is equivalent to simply using a subset $\boldsymbol{X}_{1:j} \in \mathbb{R}^{j \times D}$ of the data and performing *exact* GP regression (*e.g.* via a Cholesky decomposition) in each Newton iteration in Algorithm 5.1. This basic policy shows how actions *target computation* as illustrated in the top row of Figure 5.3.

(Conjugate) Gradient Policy. Instead of targeting individual data points, we can also specify *weighted* linear combinations of them to target the data more globally. *E.g.*, using the current residual $\boldsymbol{s}_j = \boldsymbol{r}_{j-1} = \hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m} - \hat{\boldsymbol{K}}(\boldsymbol{f}_i)\boldsymbol{v}_{j-1}$, approximately targets those data most, where the posterior mean prediction is far off.[4] As Wenger et al. [143] show, this corresponds to using conjugate gradients (CG) [61] to estimate the posterior mean. This policy is illustrated in the bottom row of Figure 5.3.

4: Since $\boldsymbol{r}_{j-1} \approx \hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m} - \boldsymbol{K}\boldsymbol{v}_{j-1} = \hat{\boldsymbol{y}}(\boldsymbol{f}_i) - m_{i,j-1}(\boldsymbol{X})$.

5.3.3 Recycling: Reusing Computations

Using IterGP with a suitable policy for GP inference allows us to solve each GP regression problem more efficiently. However, for NCGP inference, we must solve *multiple* regression problems—one per mode-finding step. Figure 5.2 suggests that GP posteriors across steps are highly similar. Leveraging this observation, we

Algorithm 5.3: Recycling: Virtual Solver Run with Optional Compression.

Input: Buffers $\boldsymbol{S}, \boldsymbol{T} \in \mathbb{R}^{NC \times B}$, access to products with $\boldsymbol{W}^{-1}$, compression parameter $R \leq B$ (optional)
Output: $\boldsymbol{C}_0$, updated buffers $\boldsymbol{S}, \boldsymbol{T}$

	Procedure: VirtualSolverRun$(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{W}^{-1})$		**Time**	**Memory**
1	$\boldsymbol{M} \leftarrow \boldsymbol{S}^\top(\boldsymbol{T} + \boldsymbol{W}^{-1}\boldsymbol{S})$	$\boldsymbol{M} = \boldsymbol{S}^\top(\boldsymbol{K} + \boldsymbol{W}^{-1})\boldsymbol{S} \in \mathbb{R}^{B \times B}$	$\mathcal{O}(Bt_{\boldsymbol{W}^{-1}} + B^2NC)$	$\mathcal{O}(B^2)$
2	$\boldsymbol{U}, \boldsymbol{\Lambda} \leftarrow \mathrm{ED}(\boldsymbol{M})$, $\boldsymbol{U} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_B), \boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \ldots, \lambda_B) \in \mathbb{R}^{B \times B}, \lambda_1 \geq \ldots \geq \lambda_B$	Eigendecomposition $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$	$\mathcal{O}(B^3)$	$\mathcal{O}(B^2)$
	Subroutine: Compression$(\boldsymbol{U}, \boldsymbol{\Lambda}, R)$			
3	$\boldsymbol{U} \leftarrow (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_R), \boldsymbol{\Lambda} \leftarrow \mathrm{diag}(\lambda_1, \ldots, \lambda_R)$	Truncation, $R \leq B$		$\mathcal{O}(BR)$
4	$\boldsymbol{S} \leftarrow \boldsymbol{S}\boldsymbol{U}, \boldsymbol{T} \leftarrow \boldsymbol{T}\boldsymbol{U}$	Update buffers	$\mathcal{O}(BRNC)$	$\mathcal{O}(RNC)$
5	$\boldsymbol{Q}_0 \leftarrow \boldsymbol{S}\boldsymbol{\Lambda}^{-1/2}$	Construct root $\boldsymbol{C}_0 = \boldsymbol{Q}_0\boldsymbol{Q}_0^\top = \boldsymbol{S}\boldsymbol{\Lambda}^{-1}\boldsymbol{S}^\top$	$\mathcal{O}(R^2NC)$	$\mathcal{O}(RNC)$
6	**return** $\boldsymbol{C}_0 \leftarrow \boldsymbol{Q}_0\boldsymbol{Q}_0^\top$ and $\boldsymbol{S}, \boldsymbol{T}$	$\boldsymbol{C}_0$ has rank R		

$\boldsymbol{C}_0$ is *never* formed explicitly in memory but evaluated lazily via its root $\boldsymbol{Q}_0$, *i.e.* $\boldsymbol{w} \mapsto \boldsymbol{C}_0\boldsymbol{w} = \boldsymbol{Q}_0(\boldsymbol{Q}_0^\top \boldsymbol{w})$.

develop a novel approach, designed specifically for the NCGP setting, that *efficiently recycles costly computations* between outer loop steps (pseudocode in Algorithm 5.3).

The cost of IterNCGP is dominated by repeated matrix-vector products with $\boldsymbol{K}$ (see Section 5.3.5 for details). However, these costly operations can be recycled and used over multiple Newton steps: Consider the matrix-vector products with an action vector $\boldsymbol{s}$ in the first and second mode-finding step as an example:

$$\begin{aligned} &\text{Step } i = 0: \quad \boldsymbol{s} \mapsto \hat{\boldsymbol{K}}(\boldsymbol{f}_0)\,\boldsymbol{s} = \boldsymbol{K}\boldsymbol{s} + \boldsymbol{W}(\boldsymbol{f}_0)^{-1}\boldsymbol{s} \\ &\text{Step } i = 1: \quad \boldsymbol{s} \mapsto \hat{\boldsymbol{K}}(\boldsymbol{f}_1)\,\boldsymbol{s} = \boldsymbol{K}\boldsymbol{s} + \boldsymbol{W}(\boldsymbol{f}_1)^{-1}\boldsymbol{s}. \end{aligned}$$

Since $\boldsymbol{K}$ is independent of $\boldsymbol{f}_i$, the product $\boldsymbol{K}\boldsymbol{s}$ is *shared* among both operations.

Virtual Solver Run. Assume we have used B action vectors $(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_B) =: \boldsymbol{S} \in \mathbb{R}^{NC \times B}$ in step $i = 0$, and buffered the matrix-vector products $(\boldsymbol{K}\boldsymbol{s}_1, \ldots, \boldsymbol{K}\boldsymbol{s}_B) = \boldsymbol{K}\boldsymbol{S} =: \boldsymbol{T}$. In the next Newton step $i = 1$ we apply the *same* actions to a *new* linear system of equations. From Equation (5.10) we obtain

$$\begin{aligned} \boldsymbol{C} = \boldsymbol{S}\boldsymbol{M}^{-1}\boldsymbol{S}^\top \quad \text{with} \quad \boldsymbol{M} &:= \boldsymbol{S}^\top(\boldsymbol{K}\boldsymbol{S} + \boldsymbol{W}(\boldsymbol{f}_1)^{-1}\boldsymbol{S}) \\ &= \boldsymbol{S}^\top(\boldsymbol{T} + \boldsymbol{W}(\boldsymbol{f}_1)^{-1}\boldsymbol{S}). \end{aligned} \tag{5.11}$$

So, we can *imitate* a solver run with the previous actions $\boldsymbol{S}$ and construct $\boldsymbol{C}$ without ever having to multiply with $\boldsymbol{K}$. The associated computational costs comprise memory for the two buffers $\boldsymbol{S}, \boldsymbol{T}$ as well as the run time costs for matrix-matrix products in Equation (5.11) and inverting $\boldsymbol{M} \in \mathbb{R}^{B \times B}$. This virtual solver run is generally orders of magnitude cheaper than running the solver from scratch with new actions (details in Appendix A.2.3). Within IterGP (Algorithm 5.2), we can use this matrix as an initial estimate $\boldsymbol{C}_0 \leftarrow \boldsymbol{C}$ of the precision matrix. Subsequently, the algorithm can

proceed as usual with new actions.

The presented recycling approach can easily be extended to all Newton steps. Whenever $\boldsymbol{K}$ is multiplied with an action vector, the vector itself and the resulting vector are appended to the respective buffers $\boldsymbol{S}$ and $\boldsymbol{T}$. For each Newton step, an initial $\boldsymbol{C}_0$ can be constructed via Algorithm 5.3.

Numerical Perspective. Crucially, the above strategy does not affect the solver's convergence properties: From a numerical linear algebra viewpoint, the strategy above is a form of *subspace recycling* [110]. Specifically, $\boldsymbol{C}_0$, as described above, defines a *deflation preconditioner* [39]: The projection of the initial residual $\boldsymbol{r}_0 = (\hat{\boldsymbol{y}} - \boldsymbol{m}) - \hat{\boldsymbol{K}}\boldsymbol{v}_0$ for the first iterate $\boldsymbol{v}_0 = \boldsymbol{C}_0(\hat{\boldsymbol{y}} - \boldsymbol{m})$ onto the subspace $\operatorname{span}\{\boldsymbol{S}\}$ spanned by the actions is zero (see Appendix A.1.4 for details). That means, the solution within the subspace $\operatorname{span}\{\boldsymbol{S}\}$ is already perfectly identified at initialization.

Probabilistic Perspective. Via Equation (5.9), we can quantify the effect of $\boldsymbol{C}_0$ on the total marginal uncertainty of predictions at the training data $\operatorname{Tr}(K_{i,0}(\boldsymbol{X}, \boldsymbol{X})) = \operatorname{Tr}(\boldsymbol{K}) - \operatorname{Tr}(\boldsymbol{K}\boldsymbol{C}_0\boldsymbol{K})$. Assuming observation noise $\boldsymbol{W}^{-1} = \boldsymbol{0}$ and all actions in $\boldsymbol{S}$ eigenvectors of $\hat{\boldsymbol{K}} = \boldsymbol{K}$, it simplifies to

$$\operatorname{Tr}(K_{i,0}(\boldsymbol{X}, \boldsymbol{X})) = \operatorname{Tr}(\boldsymbol{K}) - \operatorname{Tr}(\boldsymbol{M}), \tag{5.12}$$

see Appendix A.1.4. The second term $\operatorname{Tr}(\boldsymbol{M})$ describes the *reduction* of the prior uncertainty due to $\boldsymbol{C}_0$. It can be maximized (which is our goal) when $\boldsymbol{S}$ contains those eigenvectors of $\hat{\boldsymbol{K}}$ with the largest eigenvalues. We take this insight as motivation for a buffer compression approach that we describe next.

5.3.4 Compression: Memory-Efficient Beliefs

Whenever $\boldsymbol{K}$ is applied to an action vector, the buffers $\boldsymbol{S}, \boldsymbol{T} \in \mathbb{R}^{NC \times B}$ grow by NC entries. To limit memory requirements for large-scale data, we propose a compression strategy (see Algorithm 5.3).

Compression via Truncation. In Algorithm 5.3, $\boldsymbol{M}^{-1} \in \mathbb{R}^{B \times B}$ is computed via an eigendecomposition $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}$, such that $\boldsymbol{C}_0 = \boldsymbol{Q}_0\boldsymbol{Q}_0^\top$ can be represented via its matrix root $\boldsymbol{Q}_0 := \boldsymbol{S}\boldsymbol{U}\boldsymbol{\Lambda}^{-1/2}$ for efficient storage and matrix-vector multiplies. To limit memory usage, we can use a *truncated* eigendecomposition of $\boldsymbol{M}$. Based on the intuition we gained from Equation (5.12), it makes sense to keep the *largest* eigenvalues (to maximize the trace) and corresponding eigenvectors. Keeping the R largest eigenvalues/-vectors yields a rank R approximation $\tilde{\boldsymbol{M}} = \tilde{\boldsymbol{U}}\tilde{\boldsymbol{\Lambda}}\tilde{\boldsymbol{U}}^\top$ of $\boldsymbol{M}$.

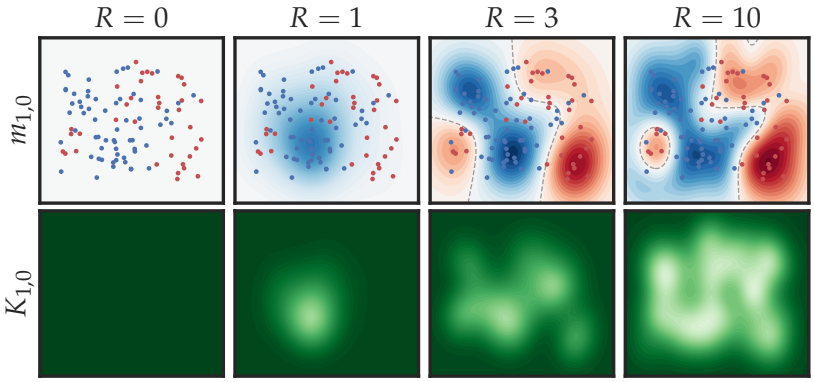


Figure 5.4: Compressed Beliefs. Recycled initial beliefs in the *second* Newton step ($i = 1$) with means $m_{1,0}$ *(Top)* and (co-)variance functions $K_{1,0}$ *(Bottom)* using compression with different buffer sizes $R \in \{0, 1, 3, 10\}$. Buffer size $R = 0$ is equivalent to not using recycling. The larger the buffer size/rank of $\boldsymbol{C}_0$, the more expressive the belief. Details in Appendix A.3.1.

Compression as Re-Weighting Actions. Forming $\boldsymbol{C}_0 = \boldsymbol{S}\tilde{\boldsymbol{M}}^{-1}\boldsymbol{S}^\top$ from the above approximation is equivalent to a virtual solver run with the modified buffers $\tilde{\boldsymbol{S}} = \boldsymbol{S}\boldsymbol{U} \in \mathbb{R}^{NC \times R}$, $\tilde{\boldsymbol{T}} = \boldsymbol{K}(\boldsymbol{S}\boldsymbol{U}) = \boldsymbol{T}\boldsymbol{U} \in \mathbb{R}^{NC \times R}$ in Equation (5.11). This shows that the truncated eigendecomposition effectively re-weights the previous B actions to form R new ones—and the weights are the eigenvectors from $\boldsymbol{M}$ that maximize the uncertainty reduction. The limit R on the buffer size controls the *memory usage* as well as the rank of $\boldsymbol{C}_0$ and thereby the *expressiveness* of the associated belief (see Figure 5.4).

5.3.5 Cost Analysis of IterNCGP

IterNCGP's total run time is dominated by the repeated application of $\boldsymbol{K}$ in Algorithm 5.2, *i.e.* $\mathcal{O}(J t_{\boldsymbol{K}})$, with J describing the *total* number of solver iterations over *all* Newton steps. $t_{\boldsymbol{K}}$ denotes the cost of a single matrix-vector product with $\boldsymbol{K}$. Typically, $t_{\boldsymbol{K}}$ is quadratic in the number of training data points. In terms of memory, the buffers $\boldsymbol{S}$, $\boldsymbol{T}$ and the matrix root $\boldsymbol{Q}$ are the decisive factors with $\mathcal{O}(BNC)$. Without compression, their final size is $B = J$. Otherwise, their maximum size is given by the sum of the rank bound R and the maximum solver iterations in Algorithm 5.2 (Appendix A.2.3 provides an in-depth discussion of run time and memory costs).

5.4 Related Work

The Laplace approximation [9, 91, 118, 131] is commonly used for approximate inference in (Bayesian) Generalized Linear Models. Here, we consider the function-space generalization of Bayesian Generalized Linear Models, namely non-conjugate GPs, for which a multitude of approximate methods have been proposed, arguably the most popular being variational approaches (*e.g.* [72]), such as SVGP [60, 138]. In contrast, to address the computational shortcomings of NCGPs on large data sets, we leverage iterative methods

to obtain and efficiently update low-rank approximations. Similar approaches were used previously to accelerate the conjugate Gaussian special case [21, 41, 48, 99, 140, 142], binary classification [149] and general Bayesian linear inverse problems [130]. Trippe et al. [139] is closest in spirit to our approach if viewed from a weight-space perspective. Their choice of low-rank projection corresponds to a specific policy in our framework. Our approach not only enables the use of policies that are more suited to the given link function, but also saves additional computation, as well as memory, via recycling and compression. In each Newton iteration, the posterior for the current regression problem is approximated via IterGP [143], which internally uses a probabilistic linear solver [18, 56, 141]. Therefore, IterNCGP is a probabilistic numerical method [19, 57, 58, 104]: It quantifies uncertainty arising from limited computation.

5.5 Experiments

We apply IterNCGP to a Poisson regression problem to explore the trade-off between the number of (outer loop) mode-finding steps and (inner loop) solver iterations (Section 5.5.1). In Section 5.5.2, we demonstrate our algorithm's scalability and the impact of compression on performance.

5.5.1 Poisson Regression

Consider count data $\boldsymbol{y} \in \mathbb{N}_0^N$ generated from a Poisson likelihood with unknown rate $\lambda : \mathbb{X} \to \mathbb{R}_{>0}$. The log-rate f is modeled by a GP. See Appendix A.3.2 for details.

Data & Model. We generate a synthetic data set by (i) sampling the log-rate from a GP with an RBF kernel (ii) transforming it into the latent rate λ by exponentiation, and (iii) sampling counts $y_n \in \mathbb{N}_0$ from the Poisson distribution with rate $\lambda(\boldsymbol{x}_n)$. The functions f, λ, and the resulting count data are shown in Figure 5.5 *(Right)*. Our model uses the same RBF prior GP to avoid model mismatch.

Newton Steps vs. Solver Iterations. From a practical standpoint, the performance achievable within a given budget of solver iterations is highly relevant: How many linear solver iterations should be performed for each regression problem before updating the problem to maximize performance? To investigate this, we use IterNCGP-CG and distribute 100 iterations uniformly over $\{5, 10, 20, 100\}$ outer loop steps. Each run uses recycling without compression and is repeated 10 times.

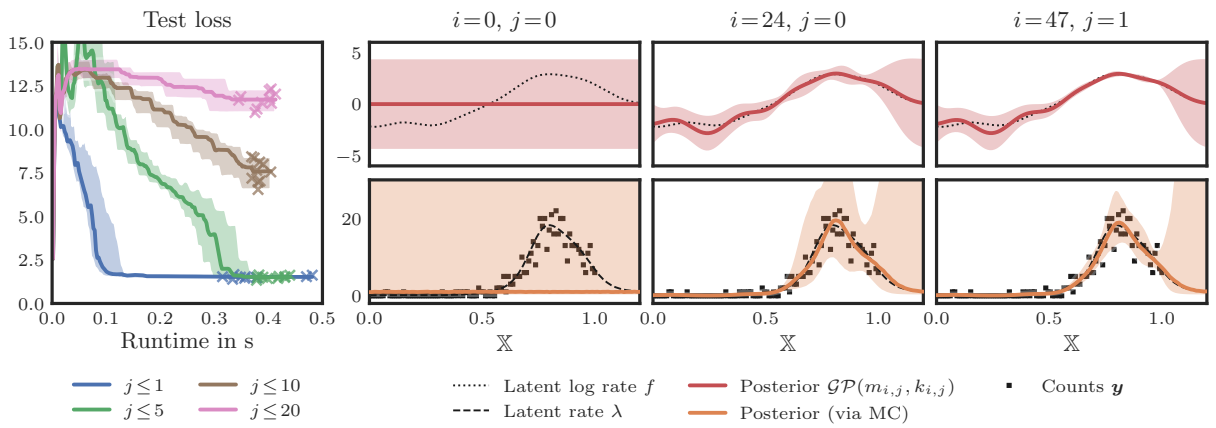


Figure 5.5: Poisson Regression with IterNCGP. *(Left)* Test loss performance for IterNCGP-CG with recycling and four schedules ($j \leq 1, 5, 10$ or 20) over $100, 20, 10$ or 5 steps (always using the same total budget of 100 iterations). For each schedule, the median (solid line) and min/max (shaded area) over 10 runs are reported. The crosses indicate the end of each run. *(Right)* Posterior $\mathcal{GP}(m_{i,j}, k_{i,j})$ for the latent log rate f *(Top)* and the corresponding belief about the rate λ *(Bottom)* computed via MC at three time points during a run of IterNCGP. The shaded 95% credible intervals show how stopping early trades less computation for increased uncertainty. Details in Appendix A.3.2.

Results. Figure 5.5 *(Left)* indicates that the strategy with a single iteration per step is the most efficient. An explanation might be that there is no reason to spend compute on an "outdated" regression problem that could be updated instead. Of course, this only applies if recycling is used, such that the *effective* number of actions accumulates. As long as $B \ll N$, the cost due to repeated recycling ($\mathcal{O}(N)$) is dwarfed by the cost of products with $\boldsymbol{K}$ ($\mathcal{O}(N^2)$). Figure 5.5 *(Right)* shows an IterNCGP-CG run with one iteration per step. As we spend more computational resources, our estimates approach the underlying latent function and, where data is available, the uncertainty contracts.

5.5.2 Large-Scale GP Multi-Class Classification

Here, we showcase IterNCGP's scalability. See Appendix A.3.3 for details.

Data & Model. We generate $N = 10^5$ data points from a Gaussian mixture model with $C = 10$ classes. We use the softmax likelihood and assume independent GPs (each equipped with a Matérn($\frac{3}{2}$) kernel) for the C outputs of the latent function. While this experiment uses synthetic data, the latent function is *not* drawn from the assumed GP model and thus the kernel is *not* perfectly identified. Also note that, if we formed $\hat{\boldsymbol{K}}$ in (working) memory explicitly, this would require $(NC)^2 \cdot 8$ byte $= 8000$ GB (in double precision). Solving the linear systems *precisely*, *e.g.* via Cholesky decomposition, is therefore infeasible, whereas our family of methods is matrix-free and can still be applied.

Methods. We compare the subset of data (SoD) approach from Section 5.3.2, the popular variational approximation SVGP [60, 138],

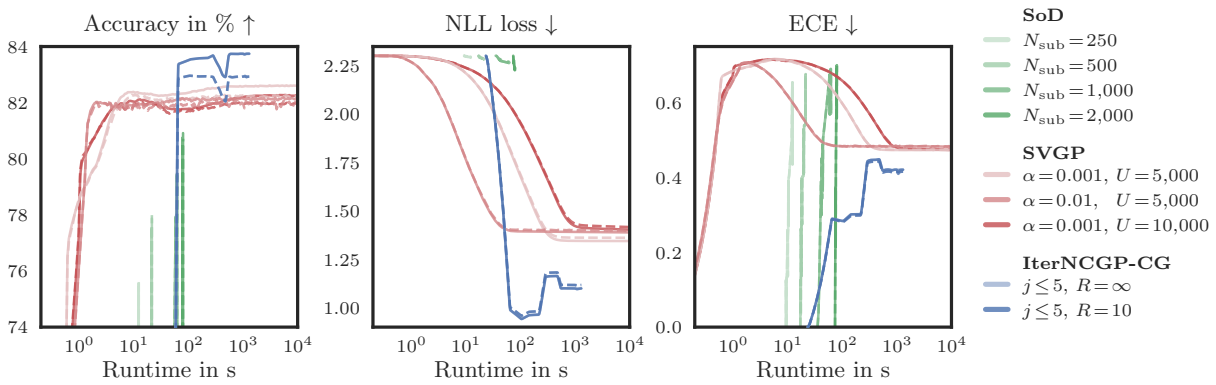


Figure 5.6: Large-Scale GP Classification. Comparison of SoD, SVGP with learning rate $\alpha \in \{0.001, 0.01, 0.05\}$ and $U \in \{1000, 2500, 5000, 10000\}$ inducing points (showing only the best three runs) and IterNCGP-CG with $R \in \{\infty, 10\}$ on a classification problem with $N = 10^5$ training points and $C = 10$ classes in terms of accuracy *(Left)*, NLL loss *(Center)* and ECE *(Right)*. Performance metrics are averaged over five runs and are shown as solid (training set) or dashed (test set) lines. IterNCGP-CG performs best in *all* three performance metrics, with minimal memory requirements. The two variants with $R \in \{\infty, 10\}$ are visually indistinguishable, *i.e.* the performance is not affected by compression. Details in Appendix A.3.3.

and our IterNCGP-CG. For SoD, we materialize $\hat{\boldsymbol{K}}$ in memory and compute its Cholesky decomposition. Four different subset sizes are used—the largest one $N_{\text{sub}} = 2000$ requires 3.2 GB of memory for $\hat{\boldsymbol{K}}$. For SVGP, we use the implementation provided by GPyTorch [41]. We optimize the ELBO for 10^4 seconds using Adam with batch size 1024 and determine suitable hyperparameters via grid search over the learning rate α and the number of inducing points U. Only the best three settings are included in our final benchmark. IterNCGP-CG is applied to the *full* training set with recycling and $R \in \{\infty, 10\}$. The number of solver iterations is limited by $j \leq 5$. We use KeOps [16] and GPyTorch for fast kernel-matrix multiplies. For this work, we consider kernel hyperparameter optimization out of scope—all methods therefore use the same fixed hyperparameters. The benchmark is run on an NVIDIA A100 GPU.

Results. Figure 5.6 shows the average performance of each method over five runs that use different random seeds. Once the matrix is formed in memory, the SoD approaches are very fast—even with $N_{\text{sub}} = 2000$, they converge within 100 s (all SoD runs require only two Newton steps). With increasing N_{sub}, the runs reach higher accuracy at the cost of increased memory requirements. To achieve top performance, SVGP requires a large number of inducing points (at the cost of slower training). Increasing the learning rate to compensate results in instabilities—these runs do not exhibit competitive performance. Both SoD and SVGP fall short of IterNCGP-CG in all three performance metrics. Using recycling, IterNCGP-CG maintains low loss/high accuracy throughout training, even when compression is used. It reaches the lowest final negative log-likelihood (NLL) and expected calibration error (ECE) demonstrating better uncertainty quantification. It is more memory-efficient than SoD (especially with compression), and, in

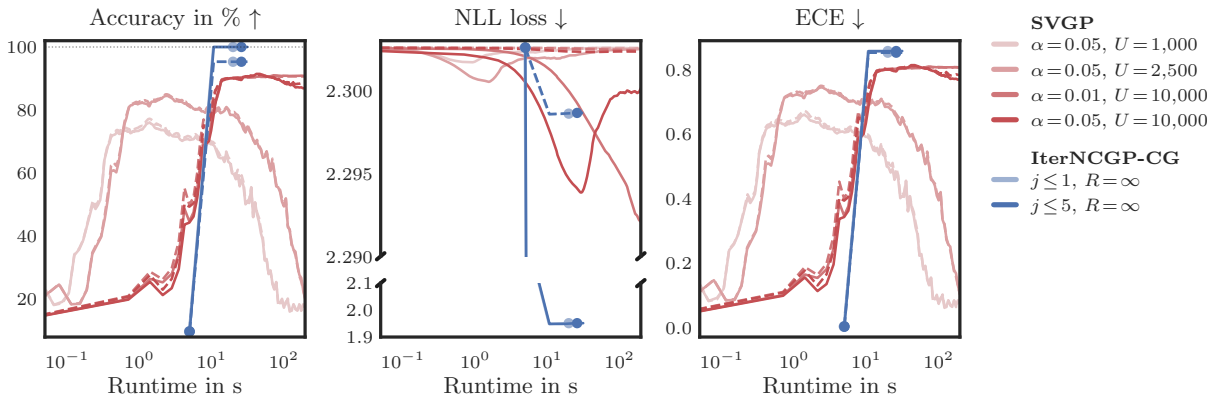


Figure 5.7: GP Classification on MNIST. Comparison of SVGP with learning rate $\alpha \in \{0.001, 0.01, 0.05\}$ and $U \in \{1000, 2500, 5000, 10000\}$ inducing points (showing only the best four runs) and IterNCGP-CG on a classification problem with $N = 20{,}000$ training points and $C = 10$ classes in terms of accuracy *(Left)*, NLL loss *(Center)* and ECE *(Right)*. Performance metrics are shown as solid (training set) or dashed (test set) lines. For IterNCGP, the dots mark the start of a new outer-loop iteration. IterNCGP-CG performs best in terms of accuracy and NLL loss but slightly worse in terms of ECE. Details in Appendix A.3.4.

contrast to SVGP, does not require extensive tuning.

Extension to MNIST. To demonstrate IterNCGP's applicability to real-world data, we perform a similar experiment on MNIST [85]. See Appendix A.3.4 for details. As KeOps scales poorly with the data dimension ($D = 28^2 = 784$ for MNIST), we revert to GPyTorch's standard kernel implementation, which requires more memory. We thus limit the training data to $N_{\text{sub}} = 20{,}000$ images. The results (Figure 5.7) are mostly aligned with Figure 5.6: IterNCGP-CG outperforms the well-tuned SVGP baselines in terms of accuracy and NLL loss. Only the ECE of IterNCGP is slightly worse than for SVGP. This is easily explained, by the fact that one can achieve smaller ECE by accepting lower accuracy—the canonical example being a random baseline, which is perfectly calibrated.

5.6 Conclusion

Non-conjugate Gaussian processes (NCGPs) provide a flexible probabilistic framework encompassing, among others, GP classification and Poisson regression. Training NCGPs on large data sets, however, necessitates approximations. Our method IterNCGP quantifies and continuously propagates the errors caused by these approximations, in the form of uncertainty. The information collected during training is efficiently recycled and compressed, reducing run time and memory requirements.

Limitations. A limitation of our method is directly inherited from the Laplace approximation: The inherent error in approximating the posterior with a Gaussian is not captured and generally depends on the choice of likelihood. However, under mild regularity

conditions, the (relative) error of the Laplace approximation scales inversely proportional to the number of data [8, 70]. Since we are considering the large data regime in this work one might reasonably expect the error contribution of the Laplace approximation itself to be small, relative to the error contribution from approximating the MAP via Newton's method, equivalently the error contribution of approximate GP regression.

Another limitation is the lack of a ready-to-use approach for kernel hyperparameter estimation. However, our method is entirely composed of basic linear algebra operations (see Algorithms 5.1 to 5.3). Therefore one can in principle simply differentiate with respect to the kernel hyperparameters through the entire solve. Recent work by Wenger et al. [144] proposes an ELBO objective to perform model selection for IterGP, which one could therefore readily apply in our setting. We leave the open question on how to optimally choose the number of Newton and IterGP steps per hyperparameter optimization step for future work.

Future Work. So far, we have only explored the policy design space in a limited fashion. The policy controls which areas of the data space are targeted and accounted for in the posterior. Tailoring the actions to the *specific* problem could further increase our method's efficiency. For classification problems, a good strategy might be not to spend compute on data points where the prediction is already definitive.

Finally, a promising application for IterNCGP may be Bayesian deep learning. A popular approach to equip a neural net with uncertainty is via a Laplace approximation [73, 90, 115], which is equivalent to a GP classification problem with a neural tangent kernel prior [66, 67]. There, the SoD approach is regularly used [66, Sec. A2.2], for which our approach might offer significant improvements.

Acknowledgments

The authors gratefully acknowledge co-funding by the European Union (ERC, ANUBIS, 101123955). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. Philipp Hennig is a member of the Machine Learning Cluster of Excellence, funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC number 2064/1 - Project number 390727645; The authors further gratefully acknowledge

financial support by the DFG through Project HE 7114/5-1 in SPP2298/1; 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. Frank Schneider is supported by funds from the Cyber Valley Research Fund. Lukas Tatzel is grateful to the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for support. Jonathan Wenger was supported by the Gatsby Charitable Foundation (GAT3708), the Simons Foundation (542963), the NSF AI Institute for Artificial and Natural Intelligence (ARNI: NSF DBI 2229929) and the Kavli Foundation.

6 ViViT: Curvature Access Through The Generalized Gauss-Newton's Low-Rank Structure

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.

6.1 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, 111] 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 [122]. 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 automatic differentiation [12, 28] have made such information more readily accessible through vectorization of algebraic structure in the differentiated loss. We leverage 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 [94, 97], compressing [128], or adding uncertainty to [77, 115, 116] neural networks. It is also linked theoretically to the natural gradient method [3] via the Fisher information matrix [95, 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 [112] and GGN [125]. Iterative linear and eigensolvers can leverage such functionality to compute Newton steps [42, 94, 148] and spectral properties [2, 44, 46, 109, 120, 121, 145] on arbitrary architectures thanks to the generality of automatic differentiation. However, repeated matrix-vector products are potentially detrimental to performance.

Secondly, K-FAC (Kronecker-factored approximate curvature) [10, 47, 96, 97] 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 [107]. 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 [30, 77, 115, 116] 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 [97]. This might be a downside for applications that depend on a precise curvature proxy.

Contributions. Here, we propose ViViT (inspired by $\boldsymbol{V}\boldsymbol{V}^\top$ in Equation (6.3)), 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 has the cost of a single gradient computation. Our contributions are as follows:

- **(i)** We present how to compute various GGN properties efficiently by exploiting its low-rank structure: The exact eigenvalues, eigenvectors, and per-sample directional derivatives (Figure 6.1). In contrast to other methods, these quantities allow modeling curvature noise.
- **(ii)** We introduce approximations that allow a flexible trade-off between computational cost and accuracy, and provide a fully-featured efficient implementation in PyTorch [111] on top of the BackPACK [28] package at `https://github.com/f-dangel/vivit`.

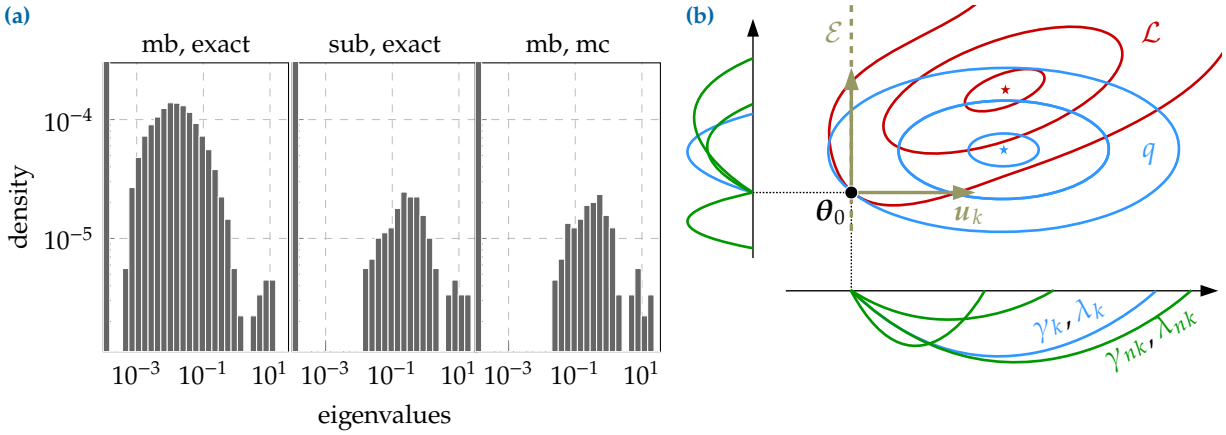


Figure 6.1: Overview of ViViT's Quantities. (a) GGN eigenvalue distribution of DeepOBS' 3c3d architecture on CIFAR-10 [123] for settings with different costs on a mini-batch of size $N = 128$. From left to right: Exact GGN, exact GGN on a mini-batch fraction, MC approximation of the GGN. **(b)** Pictorial illustration of ViViT's quantities in the loss landscape: The contour lines visualize the loss function $\mathcal{L}$ (Equation (6.1)) and its quadratic model q around $\boldsymbol{\theta}_0 \in \mathbb{R}^2$ (Equation (6.6)). The low-rank structure provides efficient access to the GGN's eigenvectors $\{\boldsymbol{u}_k\}$. The quadratic model's one-dimensional projections along the eigenvectors (black axes) are parabolas characterized by the directional derivatives γ_k, λ_k and their per-sample contributions $\gamma_{nk}, \lambda_{nk}$ (Equation (6.8)). $\mathcal{E}$ is the GGN's top-1 eigenspace.

(iii) We empirically demonstrate scalability and efficiency of leveraging the GGN's low-rank structure through benchmarks on different deep architectures. Finally, we use the previously inaccessible properties of the GGN to study how it is affected by noise during training.

The main focus of this work is demonstrating that many interesting curvature properties, including uncertainty, can be computed efficiently. Practical applications are discussed in Section 6.5.

6.2 Notation & Method

Setting. Consider a model $f_{\boldsymbol{\theta}} : \mathbb{X} \to \mathbb{F}$ and a data set $\mathbb{D}$ of tuples $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{X} \times \mathbb{Y}$. The network, parameterized by $\boldsymbol{\theta} \in \Theta$, maps a sample $\boldsymbol{x}$ to a prediction $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$. 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 $\boldsymbol{y}$. The training objective is the empirical risk $\frac{1}{|\mathbb{D}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathbb{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y})$. In the following, we consider the loss $\mathcal{L} : \Theta \to \mathbb{R}$

Section 2.2.2 provides a detailed introduction to the empirical risk minimization problem.

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n) \,, \tag{6.1}$$

evaluated on a mini-batch $\{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X} \times \mathbb{Y}\}_{n=1}^{N} \subset \mathbb{D}$ with N samples. We use $\ell_n(\boldsymbol{\theta}) = \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n)$ and $f_n(\boldsymbol{\theta}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ 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 $\Theta \subseteq \mathbb{R}^P$ and $\mathbb{F} \subseteq \mathbb{R}^C$ with P, 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 (6.1) into two parts $\nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \boldsymbol{G}(\boldsymbol{\theta}) + \boldsymbol{R}(\boldsymbol{\theta}) \in \mathbb{R}^{P \times P}$; the positive semidefinite GGN

$$\boldsymbol{G} = \frac{1}{N} \sum_{n=1}^{N} \left(\mathrm{J}_{\boldsymbol{\theta}} f_n\right)^\top \left(\nabla^2_{f_n} \ell_n\right) \left(\mathrm{J}_{\boldsymbol{\theta}} f_n\right) = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{G}_n \tag{6.2}$$

and a residual $\boldsymbol{R} = {}^1\!/_N \sum_{n=1}^{N} \sum_{c=1}^{C} (\nabla^2_{\boldsymbol{\theta}} [f_n]_c) \left[\nabla_{f_n} \ell_n\right]_c$. Here, we use the Jacobian $\mathrm{J}_{\boldsymbol{a}} \boldsymbol{b}$ that contains partial derivatives of $\boldsymbol{b}$ with respect to $\boldsymbol{a}$, $[\mathrm{J}_{\boldsymbol{a}} \boldsymbol{b}]_{ij} = \partial [\boldsymbol{b}]_i / \partial [\boldsymbol{a}]_j$. As the residual may alter the Hessian's definiteness—undesirable in many applications—we focus on the GGN.

Low-Rank Structure. By basic inequalities, Equation (6.2) has $\operatorname{rank}(\boldsymbol{G}) \leq NC$.[1] To make this explicit, we factorize the positive semidefinite Hessian $\nabla^2_{f_n} \ell_n = \sum_{c=1}^{C} \boldsymbol{s}_{nc} \boldsymbol{s}_{nc}^\top$, where $\boldsymbol{s}_{nc} \in \mathbb{R}^C$ and denote its backpropagated version by $\boldsymbol{v}_{nc} = (\mathrm{J}_{\boldsymbol{\theta}} f_n)^\top \boldsymbol{s}_{nc} \in \mathbb{R}^P$. Absorbing sums into matrix multiplications, we arrive at the GGN's outer product form that lies at the heart of the ViViT concept,

$$\boldsymbol{G} = \frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} \boldsymbol{v}_{nc} \boldsymbol{v}_{nc}^\top = \boldsymbol{V} \boldsymbol{V}^\top \tag{6.3}$$

with $\boldsymbol{V} = {}^1\!/_{\sqrt{N}} (\boldsymbol{v}_{11}, \boldsymbol{v}_{12}, \dots, \boldsymbol{v}_{NC}) \in \mathbb{R}^{P \times NC}$. $\boldsymbol{V}$ allows for *exact* computations with the explicit GGN matrix, at linear rather than quadratic memory cost in P. 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.

1: We assume the overparameterized deep learning setting ($NC < P$) and suppress the trivial rank bound P.

6.2.1 Computing the Full GGN Eigenspectrum

Each GGN eigenvalue $\lambda \in \mathbb{R}_{\geq 0}$ is a root of the characteristic polynomial $\det(\boldsymbol{G} - \lambda \boldsymbol{I}_P)$ with identity matrix $\boldsymbol{I}_P \in \mathbb{R}^{P \times P}$. Leveraging the factorization of Equation (6.3) and the matrix determinant lemma, the P-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 $\boldsymbol{v}_{nc}$ (see Appendix B.1.1),

$$\det(\boldsymbol{G} - \lambda \boldsymbol{I}_P) = 0 \quad \Leftrightarrow \quad \det(\tilde{\boldsymbol{G}} - \lambda \boldsymbol{I}_{NC}) = 0\,. \tag{6.4}$$

With at least $P - NC$ 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 [2, 44, 46, 109, 120, 121, 145], 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 6.3.1 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 [87]. But we can also reconstruct the associated eigenvectors. These are directions along which curvature information is contained in the mini-batch. Let $\tilde{\mathbb{S}}_+ = \{(\lambda_k, \tilde{\boldsymbol{u}}_k) \mid \lambda_k \neq 0, \tilde{\boldsymbol{G}}\tilde{\boldsymbol{u}}_k = \lambda_k \tilde{\boldsymbol{u}}_k\}_{k=1}^{K}$ denote the nontrivial Gram spectrum[2] with orthonormal eigenvectors $\tilde{\boldsymbol{u}}_j^\top \tilde{\boldsymbol{u}}_k = \delta_{jk}$ (δ represents the Kronecker delta and $K = \operatorname{rank}(\boldsymbol{G})$). Then, the transformed vectors $\boldsymbol{u}_k = {}^1/\sqrt{\lambda_k}\boldsymbol{V}\tilde{\boldsymbol{u}}_k$ ($k = 1, \ldots, K$) are orthonormal eigenvectors of $\boldsymbol{G}$ associated to eigenvalues λ_k (see Appendix B.1.2), i.e.

$$\forall (\lambda_k, \tilde{\boldsymbol{u}}_k) \in \tilde{\mathbb{S}}_+ : \quad \tilde{\boldsymbol{G}}\tilde{\boldsymbol{u}}_k = \lambda_k \tilde{\boldsymbol{u}}_k \implies \boldsymbol{G}\boldsymbol{u}_k = \lambda_k \boldsymbol{u}_k \,. \tag{6.5}$$

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]

2: In the following, we assume ordered eigenvalues, *i.e.* $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_K$, for convenience.

3: Appendix B.3.2 describes implicit multiplication with $\boldsymbol{G}^{-1}$.

6.2.2 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 q denote the quadratic model for the loss around position $\boldsymbol{\theta}_0 \in \Theta$ that uses curvature represented by the GGN,

$$q(\boldsymbol{\theta}) = \text{const} + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{g}(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{G}(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \,. \tag{6.6}$$

At its base point $\boldsymbol{\theta}_0$, the shape of q along an arbitrary normalized direction $\boldsymbol{d} \in \Theta$ (*i.e.* $\|\boldsymbol{d}\| = 1$) is determined by the local gradient and curvature.[4] Specifically, the projection of Equation (6.6) onto $\boldsymbol{d}$ gives rise to the (scalar) first-and second-order directional derivatives

$$\begin{aligned} \gamma_{\boldsymbol{d}} &= \boldsymbol{d}^\top \nabla_{\boldsymbol{\theta}} q(\boldsymbol{\theta}_0) = \boldsymbol{d}^\top \boldsymbol{g}(\boldsymbol{\theta}_0) \in \mathbb{R} \qquad \text{and} \\ \lambda_{\boldsymbol{d}} &= \boldsymbol{d}^\top \nabla^2_{\boldsymbol{\theta}} q(\boldsymbol{\theta}_0)\, \boldsymbol{d} = \boldsymbol{d}^\top \boldsymbol{G}(\boldsymbol{\theta}_0)\, \boldsymbol{d} \in \mathbb{R} \,. \end{aligned} \tag{6.7}$$

4: Consider a cut through the quadratic (Equation (6.6)) from $\boldsymbol{\theta}_0$ in direction $\boldsymbol{d}$, *i.e.* $r(\tau) \coloneqq q(\boldsymbol{\theta}_0 + \tau\boldsymbol{d})$. The *directional* slope and curvature are thus given by

$$\begin{aligned} r'(\tau) &= \boldsymbol{d}^\top \boldsymbol{g}(\boldsymbol{\theta}_0) + \tau \boldsymbol{d}^\top \boldsymbol{G}(\boldsymbol{\theta}_0)\, \boldsymbol{d} \\ r''(\tau) &= \boldsymbol{d}^\top \boldsymbol{G}(\boldsymbol{\theta}_0)\, \boldsymbol{d}. \end{aligned}$$

Equation (6.7) introduces $\gamma_{\boldsymbol{d}} = r'(0)$ and $\lambda_{\boldsymbol{d}} = r''(0)$ as the directional slope and curvature at $\boldsymbol{\theta}_0$ in direction $\boldsymbol{d}$.

As $\boldsymbol{G}$'s characteristic directions are its eigenvectors, they form a natural basis for the quadratic model. Denoting $\gamma_k = \gamma_{\boldsymbol{u}_k}$ and $\lambda_k = \lambda_{\boldsymbol{u}_k}$ the directional gradient and curvature along eigenvector $\boldsymbol{u}_k$, we see from Equation (6.7) 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 loss function's sum structure (Equation (6.1)), *i.e.* they decompose into contributions from individual samples. Let γ_{nk} and λ_{nk} denote these first- and second-order derivatives contributions of sample $\boldsymbol{x}_n$ in direction $\boldsymbol{u}_k$,

$$\begin{aligned} \gamma_{nk} &= \boldsymbol{u}_k^\top \boldsymbol{g}_n = \frac{\tilde{\boldsymbol{u}}_k^\top \boldsymbol{V}^\top \boldsymbol{g}_n}{\sqrt{\lambda_k}} \qquad \text{and} \\ \lambda_{nk} &= \boldsymbol{u}_k^\top \boldsymbol{G}_n \boldsymbol{u}_k = \frac{\|\boldsymbol{V}_n^\top \boldsymbol{V} \tilde{\boldsymbol{u}}_k\|^2}{\lambda_k} , \end{aligned} \tag{6.8}$$

where $\boldsymbol{V}_n \in \mathbb{R}^{P \times C}$ 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 (6.7), gradient $\boldsymbol{g}$ and curvature $\boldsymbol{G}$ are sums over $\boldsymbol{g}_n$ and $\boldsymbol{G}_n$, respectively. This implies the relationships between directional derivatives and per-sample contributions $\gamma_k = {}^1\!/_N \sum_{n=1}^N \gamma_{nk}$ and $\lambda_k = {}^1\!/_N \sum_{n=1}^N \lambda_{nk}$. Figure 6.1b shows a pictorial view of the quantities provided by ViViT. Access to per-sample directional gradients γ_{nk} and curvatures λ_{nk} along $\boldsymbol{G}$'s natural directions is one distinct feature of ViViT. These quantities provide geometric information about the local loss landscape *as well as* about the model's directional curvature stochasticity over the mini-batch.

6.2.3 Computational Complexity

So far, we have formulated the computation of the GGN's eigenvalues (Equation (6.4)), including eigenvectors (Equation (6.5)), and per-sample directional derivatives (Equation (6.8)). We now analyze their computational complexity 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 benchmarks in Section 6.3.1.

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 $\boldsymbol{g} = {}^1\!/_N \sum_{n=1}^N (\mathrm{J}_{\boldsymbol{\theta}} f_n)^\top \nabla_{f_n} \ell_n$. Each column $\boldsymbol{v}_{nc}$ of $\boldsymbol{V}$ also involves applying the Jacobian, but to a

Section 2.2.4 provides a detailed derivation on how loss gradients can be computed via automatic differentiation.

different vector $\boldsymbol{s}_{nc}$ 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 $\boldsymbol{V}$ 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 GPUs.

Stage-Wise Discarding $\boldsymbol{V}$. The columns of $\boldsymbol{V}$ correspond to backpropagated vectors. During backpropagation, sub-matrices of $\boldsymbol{V}$, 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 $\tilde{\boldsymbol{G}}$ formed by pairwise scalar products of $\{\boldsymbol{v}_{nc}\}_{n=1,c=1}^{N,C}$ in $\mathcal{O}((NC)^2P)$ operations. The spectral decomposition $\tilde{\mathbb{S}}_+$ has additional cost of $\mathcal{O}((NC)^3)$. Similarly, the terms for the directional derivatives in Equation (6.8) can be built up stage-wise: First-order derivatives $\{\gamma_{nk}\}_{n=1,k=1}^{N,K}$ require the vectors $\{\boldsymbol{V}^\top \boldsymbol{g}_n \in \mathbb{R}^{NC}\}_{n=1}^{N}$ that cost $\mathcal{O}(N^2CP)$ operations. Second-order derivatives are basically for free, as $\{\boldsymbol{V}_n^\top \boldsymbol{V} \in \mathbb{R}^{C\times NC}\}_{n=1}^{N}$ is available from $\tilde{\boldsymbol{G}}$.

GGN Eigenvectors. Transforming a Gram matrix eigenvector $\tilde{\boldsymbol{u}}_k$ to the GGN eigenvector $\boldsymbol{u}_k$ by application of $\boldsymbol{V}$ (Equation (6.5)) costs $\mathcal{O}(NCP)$ operations. However, repeated application of $\boldsymbol{V}$ can be avoided for sums of the form $\sum_k (c_k/\sqrt{\lambda_k})\boldsymbol{u}_k$ 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 $\sum_k c_k \tilde{\boldsymbol{u}}_k$ needs to be transformed. For a practical example—computing damped Newton steps—see Appendix B.2.1.

6.2.4 Approximations & Implementation

Although the GGN's representation by $\boldsymbol{V}$ has linear memory cost in P, it requires memory equivalent to NC model copies.[5] Of course, this is infeasible for many networks and data sets, *e.g.* ImageNet ($C = 1000$). So far, our formulation was concerned with *exact* computations. We now present approximations that allow N, C and P in the above cost analysis to be replaced by smaller numbers, enabling ViViT to trade-off accuracy and performance.

5: Our implementation uses a more memory-efficient approach that avoids expanding $\boldsymbol{V}$ for linear layers by leveraging structure in their Jacobian. We describe additional optimizations in Appendix B.3.1, and demonstrate a 50x speed-up for computing the Gram matrix over naive computation via vectorized Jacobians. The additional backpropagations are carried out in 50 % of the expected time due to parallelism on the GPU.

MC Approximation & Curvature Sub-Sampling. To reduce the scaling in C, we can approximate the factorization $\nabla^2_{f_n} \ell_n(\boldsymbol{\theta}) = \sum_{c=1}^{C} \boldsymbol{s}_{nc}\boldsymbol{s}_{nc}^\top$ by a smaller set of vectors. One principled approach is to draw MC samples $\{\tilde{\boldsymbol{s}}_{nm}\}$ such that $\mathbb{E}_m[\tilde{\boldsymbol{s}}_{nm}\tilde{\boldsymbol{s}}_{nm}^\top] = \nabla^2_{f_n} \ell_n(\boldsymbol{\theta})$ as in [28]. This reduces the scaling of backpropagated vectors from

C to the number of MC samples M ($M = 1$ in the following if not specified otherwise). A common independent approximation to reduce the scaling in N is computing curvature on a mini-batch subset [14, 148].

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 P.

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.

Next, we specifically address how the above approximations affect run time and memory requirements, and study their impact on structural properties of the GGN.

6.3 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 6.2.4 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, where it can serve as a monitoring tool of novel quantities that have not been explored previously to analyze training and other phenomena. The code used for the experiments is available at `https://github.com/f-dangel/vivit-experiments`.

Experimental Setting. Architectures include three deep convolutional neural networks from DeepOBS [123] (2c2d on Fashion-MNIST, 3c3d on CIFAR-10 and All-CNN-C on CIFAR-100), as well as residual networks from He et al. [54] on CIFAR-10 based on Idelbayev [64]—all are equipped with cross-entropy loss. Based on the approximations presented in Section 6.2.4, 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 [148]). 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.

6.3.1 Scalability

We now complement the theoretical computational complexity analysis from Section 6.2.3 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 $\{\lambda_k \mid (\lambda_k, \tilde{\boldsymbol{u}}_k) \in \tilde{\mathbb{S}}_+\}$ are obtained by forming and eigendecomposing the Gram matrix $\tilde{\boldsymbol{G}}$, allowing stage-wise discarding of $\boldsymbol{V}$ (see Sections 6.2.1 and 6.2.3).
2. **Computing the Top Eigenpair.** For $(\lambda_1, \boldsymbol{u}_1)$, we compute the Gram matrix spectrum $\tilde{\mathbb{S}}_+$, extract its top eigenpair $(\lambda_1, \tilde{\boldsymbol{u}}_1)$, and transform it into parameter space by Equation (6.5), *i.e.* $(\lambda_1, \boldsymbol{u}_1 = {}^1/\sqrt{\lambda_1} \boldsymbol{V} \tilde{\boldsymbol{u}}_1)$. This requires more memory than task 1 as $\boldsymbol{V}$ must be stored.

As a comprehensive measure for memory performance, we use the largest batch size before our system runs out of memory—we call this the *critical* batch size N_{crit}.

Figure 6.2a 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 Schneider et al. [123]), even when using the exact GGN. With ViViT's approximations, the memory overhead is reduced to significantly increase the applicable batch size. We report similar results for more architectures, a block-diagonal approximation (as in Zhang et al. [148]), and on CPU in Appendix B.2.1, where we also benchmark a third task—computing damped Newton steps.

In the large-batch regime, computing the Gram matrix and its spectrum becomes a run time bottleneck, see Section 6.2.3. As

we show next, ViViT is highly efficient in the mini-batch setting. However, for spectral analyses on the entire data set, iterative eigensolvers, like power iterations, should be preferred.

Run Time Performance. Here, we consider the task of 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 [145] and replace its Hessian-vector product by a matrix-free GGN-vector product [125] through PyTorch's automatic differentiation. We use the same default hyperparameters for the termination criterion.[6] Similar to task 1, our method obtains the top-k eigenpairs[7] by computing $\tilde{\mathbb{S}}_+$, extracting its leading eigenpairs and transforming the eigenvectors $\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2, \ldots, \tilde{\boldsymbol{u}}_k$ into parameter space by application of $\boldsymbol{V}$.

6: We find similar results with relaxed convergence hyperparameters, see Appendix B.2.1.

7: The power iteration computes the leading eigenpairs. Our approach allows choosing arbitrary eigenpairs.

Figure 6.2b 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 6.2.3) and allows for efficient extraction of a large number of eigenpairs. The run time curves of the approximations confirm this behavior by featuring the same flat profile. Additionally, they require significantly less time than the exact mini-batch computation. Appendix B.2.1 reports additional results for more architectures, a block-diagonal approximation, and on CPU.

6.3.2 Approximation Quality

ViViT is based on the Hessian's generalized Gauss-Newton approximation (see Equation (6.2)). 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 6.2.4), 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 B.2.2).

(a)

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)

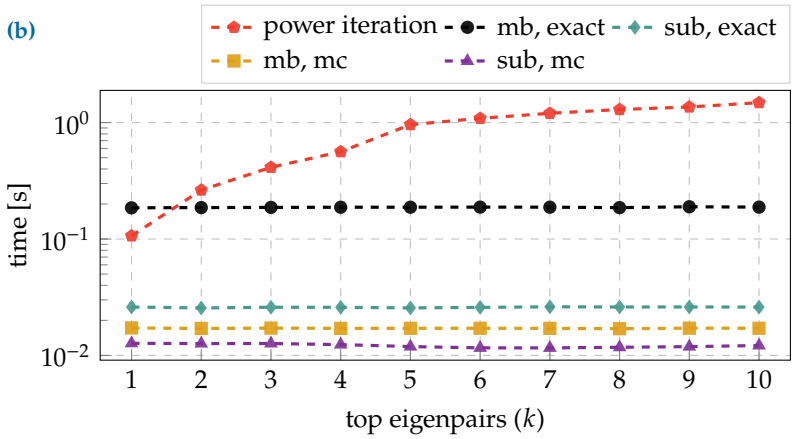


Figure 6.2: GPU Memory and Run Time Performance. Performance measurements for the 3c3d architecture ($P = 895\,210$) on CIFAR-10 ($C = 10$). **(a)** Critical batch sizes N_{crit} for computing eigenvalues and the top eigenpair. **(b)** Run time comparison with a power iteration for extracting the k leading eigenpairs using a batch of size $N = 128$.

GGN vs. 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 (6.2)), 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 C leading eigenvectors. This is a C-dimensional subspace of the parameter space $\mathbb{\Theta}$, on which the loss function is subject to particularly strong curvature. The *overlap* between these spaces serves as the comparison metric. Let $\{\boldsymbol{u}_c^A\}_{c=1}^C$ be the set of orthonormal eigenvectors to the C largest eigenvalues of some symmetric matrix $\boldsymbol{A}$ and $\mathcal{E}^A = \text{span}(\boldsymbol{u}_1^A, \dots, \boldsymbol{u}_C^A)$. The projection onto this subspace $\mathcal{E}^A$ is given by the projection matrix $\boldsymbol{P}^A = (\boldsymbol{u}_1^A, \dots, \boldsymbol{u}_C^A)(\boldsymbol{u}_1^A, \dots, \boldsymbol{u}_C^A)^\top$. As in Gur-Ari et al. [50], we define the overlap between two top-C eigenspaces $\mathcal{E}^A$ and $\mathcal{E}^B$ of the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ by

$$\text{overlap}(\mathcal{E}^A, \mathcal{E}^B) = \frac{\text{Tr}\left(\boldsymbol{P}^A \boldsymbol{P}^B\right)}{\sqrt{\text{Tr}\left(\boldsymbol{P}^A\right) \text{Tr}\left(\boldsymbol{P}^B\right)}} \in [0, 1]\,. \tag{6.9}$$

If $\text{overlap}(\mathcal{E}^A, \mathcal{E}^B) = 0$, then $\mathcal{E}^A$ and $\mathcal{E}^B$ are orthogonal; if the overlap is 1, they are identical.

Figure 6.3a 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), it shows a strong agreement (overlap ≥ 0.85) between the top-C eigenspaces. We make similar observations with the other test problems (see Appendix B.2.3), yet to a slightly lesser extent for CIFAR-100. Con-

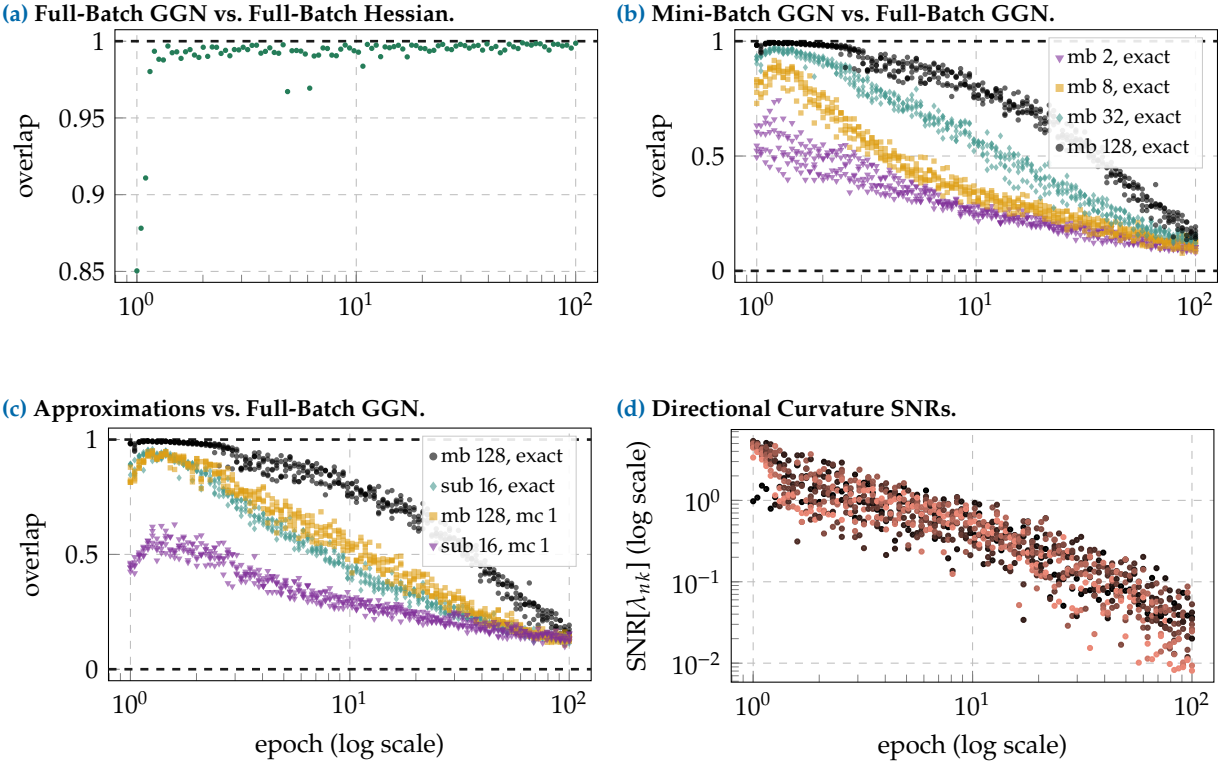


Figure 6.3: Curvature Monitoring During Training 3c3d on CIFAR-10 with SGD. **(a)** Overlap between the top-C eigenspaces of the full-batch GGN and full-batch Hessian during training. **(b)** Overlap between the top-C eigenspaces of the mini-batch GGN and full-batch GGN during training. For each mini-batch size, 5 different mini-batches are drawn. **(c)** Overlap between the top-C eigenspaces of the mini-batch GGN and ViViT's approximations with the full-batch GGN during training. Each approximation is evaluated on 5 mini-batches. **(d)** Curvature SNRs along each of the mini-batch GGN's top-C eigenvectors during training. 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 ●.

sequently, we identify the GGN as an interesting object, since it consistently shares relevant structure with the Hessian matrix.

Eigenspace Under Noise and 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 6.3b: 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 [94] reports that the optimal batch size grows during training).

In order to allow for a fine-grained cost-accuracy trade-off, ViViT introduces *further* approximations to the mini-batch GGN (see Section 6.2.4). Figure 6.3c shows the overlap between these GGN approximations and the full-batch GGN.[8] The order of the approx-

8: A comparison with the mini-batch GGN as ground truth can be found in Appendix B.2.4

imations 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 B.2.4.

So far, our analysis is based on the top-C eigenspace of the curvature matrices. We extend this analysis by studying the effect of noise and approximations on the curvature *magnitude* along the top-C directions in Appendix B.2.5.

6.3.3 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 (see Equation (6.8)). To quantify noise in the directional derivatives, we compute their signal-to-noise ratios (SNRs). For each direction $\boldsymbol{u}_k$, the SNR is given by the squared empirical mean divided by the empirical variance of the N mini-batch samples $\{\gamma_{nk}\}_{n=1}^{N}$ and $\{\lambda_{nk}\}_{n=1}^{N}$, respectively.

Figure 6.3d 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. This might be a challenging setting for second-order methods because the noise varies dramatically during different stages of training. In comparison, the directional gradients do not exhibit such a pattern (see Appendix B.2.6). Results for the other test cases can be found in Appendix B.2.6.

In this section, we gave a glimpse of the *very rich* quantities that are efficiently computed via ViViT. Section 6.5 discusses the practical use of this information, in particular curvature uncertainty.

6.4 Related Work

GGN Spectrum & Low-Rank Structure. Other works point out the GGN's low-rank structure. Botev et al. [10] present the rank bound and propose an alternative to K-FAC based on backpropagating a decomposition of the loss Hessian. Papyan [108] presents the factorization in Equation (6.3) 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 [12, 28], vectorized Jacobians required to perform those operations efficiently were not available.

Efficient Operations with Large-Scale Matrices in Deep Learning. Chen et al. [17] use Equation (6.3) for element-wise evaluation of the GGN in fully-connected feed-forward neural networks. 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 enjoys efficient eigen-decomposition.

A special case of ViViT's Gram matrix extraction is computing empirical neural tangent kernel (NTK) matrices, like [103]. While the NTK only requires a model (its Jacobian [67]), the GGN also incorporates the loss function via its Hessian. For mean squared error, this Hessian in Equation (6.2) is proportional to the identity, and the GGN Gram matrix coincides with the scaled empirical NTK.

Another prominent low-rank matrix in deep learning is the uncentered gradient covariance (sometimes called empirical Fisher). Singh and Alistarh [128] describe implicit multiplication with its inverse and apply it for neural network compression, assuming the empirical Fisher as Hessian proxy. However, this assumption has limitations, specifically for optimization [82]. In principle though, the low-rank structure also permits the application of our methods from Section 6.2.

6.5 Use Cases

Our efficient implementation enables the community to explore deep learning through richer information that would previously have been costly. Here, we want to briefly address possible use cases—their full development and assessment, however, will amount to separate paper(s). They include:

- **Second-Order Optimization.** Second-order methods use curvature to build a local quadratic model of the loss. Established curvature proxies neglect the sampling-induced noise and therefore the quadratic model's reliability. ViViT provides access to this information in the form of *per-sample* quantities. This offers a new dimension for improving second-order methods: Through statistics on the mini-batch *distribution* of directional derivatives, we might be able to adapt to the dynamics of noise (e.g. via variance-adapted step sizes).

- **Monitoring Tool.** However, to develop conceptually novel second-order optimizers, we believe that a crucial intermediate step is to better understand the setting they operate in. The techniques we present primarily tackle this intermediate step. Sections 6.3.2 and 6.3.3 are examples of ViViT's application as a monitoring tool. Due to its efficiency, ViViT could be integrated into live diagnostic tools like Cockpit [124] that aim at debugging optimizers or gaining insights into the inner workings of deep learning.

6.6 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 [111] by extending the existing BackPACK [28] library.

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 Ministry of 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 Lukas Tatzel. Further, we are grateful to Agustinus Kristiadi, Filip de Roos, Frank

Schneider, Jonathan Wenger, and Marius Hobbhahn for providing feedback to the manuscript.

7 Debiasing Mini-Batch Quadratics for Applications in Deep Learning

Abstract

Quadratic approximations form a fundamental building block of machine learning methods. E.g., second-order optimizers try to find the Newton step into the minimum of a local quadratic proxy to the objective function; and the second-order approximation of a network's loss function can be used to quantify the uncertainty of its outputs via the Laplace approximation. When computations on the entire training set are intractable—typical for deep learning—the relevant quantities are computed on mini-batches. This, however, distorts and biases the shape of the associated *stochastic* quadratic approximations in an intricate way with detrimental effects on applications. In this paper, we (i) show that this bias introduces a systematic error, (ii) provide a theoretical explanation for it, (iii) explain its relevance for second-order optimization and uncertainty quantification via the Laplace approximation in deep learning, and (iv) develop and evaluate debiasing strategies.

7.1 Introduction

Quadratic approximations of the loss landscape are increasingly used by algorithms in deep learning, from pruning methods [34, 147] and influence functions [76] to second-order optimizers [3, 10, 43, 47, 94, 96, 97, 107, 148] and uncertainty quantification via the Laplace approximation [30, 66, 77, 115, 116]. When the computations are intractable on the entire training set—typical for deep learning—the quantities of interest are computed on mini-batches subsampled from the training data. The goal of this work is to highlight that mini-batching systematically biases the shape of a quadratic approximation.

An extended overview of these and other curvature-based approaches is given in Chapter 3.

A Systematic Bias? Figure 7.1 illustrates the phenomenon. It shows five *mini-batch* quadratics in their top-curvature 2D subspace for the fully trained ALL-CNN-C model on CIFAR-100 data. For comparison, the full-batch quadratic, where all quantities are evaluated on the *entire* training set, is projected into the *same* 2D subspace. Within that subspace, the two quadratics are quite different: The mini-batch quadratic is much "narrower" (exhibits larger curvature) than the full-batch version. Given that the full-batch quadratic is the "right" object to serve as the basis for, e.g., a Newton step or

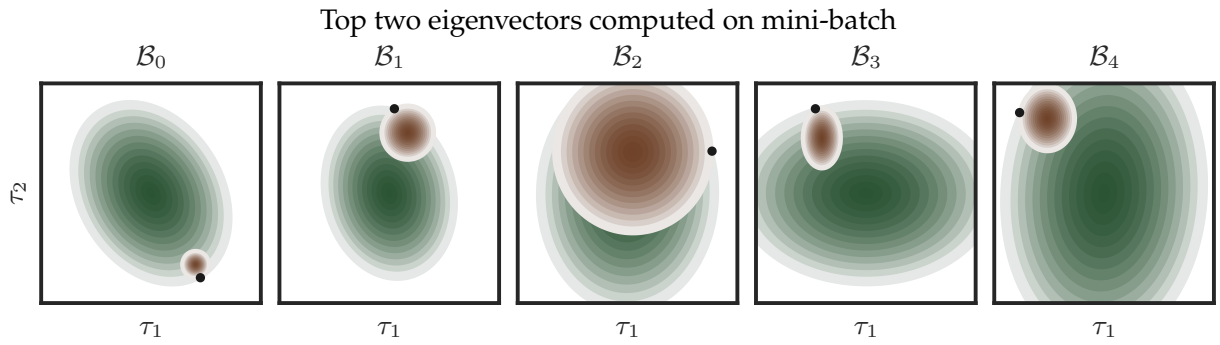


Figure 7.1: A Systematic Bias? We compute five mini-batch quadratics $q(\cdot\,; \mathcal{B}_m)$ with batch size $|\mathcal{B}_m| = 512$ for the loss landscape of the fully trained ALL-CNN-C model on CIFAR-100 data around $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}_\star$ (shown as ●). Each mini-batch quadratic defines a 2D subspace spanned by the top two eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2$ of $\boldsymbol{H}_{\mathcal{B}_m}$, in which we evaluate (i) the quadratic $q(\boldsymbol{\theta}_\star + \tau_1\boldsymbol{u}_1 + \tau_2\boldsymbol{u}_2; \mathcal{B}_m)$ itself (shown in ▭) and (ii) the full-batch quadratic $q(\boldsymbol{\theta}_\star + \tau_1\boldsymbol{u}_1 + \tau_2\boldsymbol{u}_2; \mathcal{D})$ (shown in ▭). In that subspace, the mini-batch quadratic is much "narrower" than the full-batch version which leads to overly small Newton steps and overconfident uncertainty estimates via the Laplace approximation.

a Laplace approximation,[1] the mini-batch version is *not* a meaningful surrogate: Its Newton step is overly small and a Laplace approximation yields an overconfident uncertainty estimate.

Contributions. To enable stable and efficient stochastic second-order optimizers, as well as reliable techniques for uncertainty quantification, we analyze this phenomenon and develop strategies to mitigate it. More specifically, our contributions are as follows: **(i)** We study mini-batch quadratics empirically and show that their geometry is systematically biased; **(ii)** we provide an explanation for this phenomenon, explaining the bias as an instance of the classic regression to the mean (directions of extreme steepness/curvature for one particular mini-batch are less extreme for other mini-batches), **(iii)** we explain the relevance of this bias for second-order optimization and uncertainty quantification via the Laplace approximation, and **(iv)** develop and evaluate debiasing strategies.

1: We are *not* concerned with the approximation error arising from the quadratic approximation of the non-quadratic function $\mathcal{L}_{\text{reg}}(\cdot\,; \mathcal{D}) \approx q(\cdot\,; \mathcal{D})$ (see Equation (7.2)), but only with the consequences of replacing the full-batch quantities by their mini-batch counterparts $q(\cdot\,; \mathcal{D}) \approx q(\cdot\,; \mathcal{B})$.

7.2 Notation & Background

The Regularized Loss. We consider a general supervised learning problem, where we try to find the optimal parameters $\boldsymbol{\theta}_\star = \arg\min_{\boldsymbol{\theta} \in \Theta} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}, \mathcal{D})$ for the parameterized function $f_{\boldsymbol{\theta}} : \mathbb{X} \to \mathbb{F}$ with $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^P$, $\mathbb{X} \subseteq \mathbb{R}^D$ and $\mathbb{F} \subseteq \mathbb{R}^C$ by minimizing the regularized loss $\mathcal{L}_{\text{reg}}$ on N training examples $\mathbb{D} \coloneqq \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in \mathbb{X} \times \mathbb{Y}\}_{n \in \mathcal{D}}$,

$$\begin{aligned} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) &\coloneqq \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) + r(\boldsymbol{\theta}) \quad \text{with} \\ \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) &\coloneqq \frac{1}{|\mathcal{D}|} \sum_{n \in \mathcal{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n), \quad \mathcal{D} = \{1, \dots, N\}. \end{aligned} \tag{7.1}$$

Section 2.2.2 provides a detailed introduction to the empirical risk minimization problem.

The empirical risk $\mathcal{L}$ measures the dissimilarity between the model's predictions $f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)$ and the true outputs $\boldsymbol{y}_n$ via a loss function $\ell : \mathbb{F} \times \mathbb{Y} \to \mathbb{R}$. The regularizer $r : \Theta \to \mathbb{R}$, $r(\boldsymbol{\theta}) := \beta/2 \, \|\boldsymbol{\theta}\|_2^2$ with $\beta \in \mathbb{R}_{\geq 0}$, penalizes the "complexity" of the model.

7.2.1 Local Full-Batch & Mini-Batch Quadratic Approximations

Full-Batch Quadratic. A local quadratic approximation of the regularized loss around $\boldsymbol{\theta}_0 \in \Theta \subseteq \mathbb{R}^P$ is given by the second-order Taylor expansion

$$\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) \approx q(\boldsymbol{\theta}; \mathcal{D}) := \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}_{\mathcal{D}}(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{g}_{\mathcal{D}} + c_{\mathcal{D}}, \tag{7.2}$$

where $c_{\mathcal{D}} := \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_0; \mathcal{D})$, $\boldsymbol{g}_{\mathcal{D}} := \nabla \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_0; \mathcal{D})$ and $\boldsymbol{H}_{\mathcal{D}} := \nabla^2 \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_0; \mathcal{D}) = \nabla^2 \mathcal{L}(\boldsymbol{\theta}_0; \mathcal{D}) + \beta \boldsymbol{I}$ is (some approximation of) the Hessian at $\boldsymbol{\theta}_0$ (all derivatives are with respect to the parameters $\boldsymbol{\theta}$ unless stated otherwise). As all quantities are evaluated on the *entire* training set $\mathcal{D}$, we refer to this as the *full-batch* quadratic. It holds that $\nabla q(\boldsymbol{\theta}; \mathcal{D}) = \boldsymbol{H}_{\mathcal{D}}(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \boldsymbol{g}_{\mathcal{D}}$ and $\nabla^2 q(\boldsymbol{\theta}; \mathcal{D}) \equiv \boldsymbol{H}_{\mathcal{D}}$.

Mini-Batch Quadratic. When the computations are intractable on the entire training set, the quantities in Equation (7.2) are typically computed on a *mini-batch*—a small randomly drawn subset—of the training data $\mathcal{B} \subset \mathcal{D}$, $|\mathcal{B}| \ll N$, resulting in a *stochastic* quadratic approximation $q(\,\cdot\,; \mathcal{B}) \approx q(\,\cdot\,; \mathcal{D})$. As $c_{\mathcal{B}}$, $\boldsymbol{g}_{\mathcal{B}}$ and $\boldsymbol{H}_{\mathcal{B}}$ are unbiased estimates of $c_{\mathcal{D}}$, $\boldsymbol{g}_{\mathcal{D}}$ and $\boldsymbol{H}_{\mathcal{D}}$, this substitution may seem innocent, but, as we will see in Section 7.3, it affects the geometry of the quadratic approximation *substantially*.

Directional Slope & Curvature. Consider a cut r through the quadratic $q(\,\cdot\,; \mathcal{B})$ from $\boldsymbol{\theta}_\bullet \in \Theta$ along the normalized direction $\boldsymbol{d}$, $\|\boldsymbol{d}\| = 1$. It holds that (derivation in Appendix C.1.1) $r(\tau) := q(\boldsymbol{\theta}_\bullet + \tau \boldsymbol{d}; \mathcal{B}) = 1/2 \tau^2 \boldsymbol{d}^\top \nabla^2 q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \, \boldsymbol{d} + \tau \boldsymbol{d}^\top \nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}) + \text{const}$. So, as a function of τ, $r : \mathbb{R} \to \mathbb{R}$ is a 1D parabola with derivatives $r'(\tau) = \tau \, \boldsymbol{d}^\top \nabla^2 q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \, \boldsymbol{d} + \boldsymbol{d}^\top \nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})$ and $r''(\tau) \equiv \boldsymbol{d}^\top \nabla^2 q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \, \boldsymbol{d}$. We denote the *directional* slope and curvature of the quadratic $q(\,\cdot\,, \mathcal{B})$ at $\boldsymbol{\theta}_\bullet$ in direction $\boldsymbol{d}$ by

$$\begin{aligned} \partial_{\boldsymbol{d}} \, q(\boldsymbol{\theta}_\bullet; \mathcal{B}) &:= r'(0) = \boldsymbol{d}^\top \nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \quad \text{and} \\ \partial_{\boldsymbol{d}}^2 \, q(\boldsymbol{\theta}_\bullet; \mathcal{B}) &:= r''(0) = \boldsymbol{d}^\top \nabla^2 q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \, \boldsymbol{d}. \end{aligned}$$

The directional slope and curvature at $\boldsymbol{\theta}_\bullet$ are thus simply the projections of the quadratic's gradient and Hessian at that location onto the direction.

Eigenvalues as Directional Curvatures. The directional curvature of the quadratic $q(\,\cdot\,; \mathcal{B})$ along one of $\boldsymbol{H}_{\mathcal{B}}$'s normalized eigen-

vectors $\boldsymbol{u}$ coincides with the corresponding eigenvalue λ since $\partial^2_{\boldsymbol{u}}\, q(\boldsymbol{\theta}_\bullet; \mathcal{B}) = \boldsymbol{u}^\top \nabla^2 q(\boldsymbol{\theta}_\bullet; \mathcal{B})\, \boldsymbol{u} = \boldsymbol{u}^\top \boldsymbol{H}_\mathcal{B}\, \boldsymbol{u} = \lambda \|\boldsymbol{u}\|^2 = \lambda$. Thus, in the context of a quadratic, an eigenvalue of the Hessian $\boldsymbol{H}_\mathcal{B}$ has a geometric interpretation as the directional curvature along the respective eigenvector.

GGN & FIM. Next, we discuss second-order optimization methods and the Laplace approximation (LA) for neural networks. Both techniques rely on local quadratic approximations of the regularized loss function and require a *positive definite* curvature matrix $\boldsymbol{H}_\mathcal{B}$. The empirical risk's Hessian $\nabla^2 \mathcal{L}(\boldsymbol{\theta}_0; \mathcal{B})$ can be *indefinite* and is therefore typically replaced by (an approximation of) the positive semidefinite Generalized Gauss-Newton matrix (GGN) $\boldsymbol{G}_\mathcal{B}$ or Fisher information matrix (FIM) $\boldsymbol{F}_\mathcal{B}$. In fact, GGN and FIM are often identical [95, Sec. 9.2]. The resulting $\boldsymbol{H}_\mathcal{B}$ is positive definite if $\beta > 0$, or when a damping term $\delta \boldsymbol{I}$, $\delta \in \mathbb{R}_{>0}$ is added (*e.g.*, in trust-region methods).

7.2.2 Second-Order Methods & Conjugate Gradients

The Newton Step. Due to the simple polynomial form of the quadratic $q(\,\cdot\,; \mathcal{B})$, its minimum can be derived in closed form and is given by the Newton step $\arg\min_{\boldsymbol{\theta}} q(\boldsymbol{\theta}; \mathcal{B}) = \boldsymbol{\theta}_0 - \boldsymbol{H}_\mathcal{B}^{-1} \boldsymbol{g}_\mathcal{B}$. This serves as the basis for second-order optimizers like L-BFGS [89, 101], the Hessian-free approach [94] or K-FAC (Kronecker-factored approximate curvature) [47, 96, 97].

Section 3.1 provides a detailed introduction to second-order optimization methods.

Conjugate Gradients. We focus on the method of conjugate gradients (CG) [61], as it is a powerful method specifically designed to minimize quadratics with positive definite Hessians effectively (details in Appendix C.1.2). It is particularly useful in the context of large-scale optimization because it only requires access to matrix-vector products with the curvature matrix $\boldsymbol{v} \mapsto \boldsymbol{H}_\mathcal{B}\, \boldsymbol{v}$ that can be computed in a matrix-free manner [112, 125], *i.e.* without ever materializing the full matrix in memory—and it has been used successfully for training neural networks [94]. Starting at $\boldsymbol{\theta}_0$, CG creates a sequence of iterates $(\boldsymbol{\theta}_0, \dots, \boldsymbol{\theta}_P)$. In each iteration p, two main steps are performed: (i) Given the current position $\boldsymbol{\theta}_p$ and a normalized search direction $\boldsymbol{d}_p$, the algorithm finds the minimum of the quadratic along $\boldsymbol{\theta}_p + \tau \boldsymbol{d}_p$, *i.e.*

$$\begin{aligned} \boldsymbol{\theta}_{p+1} &= \boldsymbol{\theta}_p + \tau_p \boldsymbol{d}_p \quad \text{with} \\ \tau_p &:= \arg\min_{\tau \in \mathbb{R}} q(\boldsymbol{\theta}_p + \tau \boldsymbol{d}_p; \mathcal{B}) = -\frac{\partial_{\boldsymbol{d}_p} q(\boldsymbol{\theta}_p; \mathcal{B})}{\partial^2_{\boldsymbol{d}_p} q(\boldsymbol{\theta}_p; \mathcal{B})}. \end{aligned} \tag{7.3}$$

In the second step (ii), CG constructs the next search direction $\boldsymbol{d}_{p+1}$ such that it is *conjugate* to all previous search directions, *i.e.*

$\boldsymbol{d}_{p+1}^\top \boldsymbol{H}_\mathcal{B}\, \boldsymbol{d}_i = 0$ for all $i \in \{1, \dots, p\}$.

7.2.3 Laplace Approximation for Neural Networks

Section 3.2 provides a detailed introduction to the Laplace approximation.

Laplace Approximation. The Laplace approximation (LA) turns a trained standard neural network into a Bayesian neural network in a post-hoc manner [30, 77, 90, 115] (details in Appendix C.1.3). The idea is to reinterpret the regularized loss $\mathcal{L}_\text{reg}$ as the negative unnormalized log-posterior $\log p(\boldsymbol{\theta} \mid \mathbb{D})$ of a specific Bayesian model. This interpretation identifies the optimal parameters $\boldsymbol{\theta}_\star = \arg\min_{\boldsymbol{\theta}} \mathcal{L}_\text{reg}(\boldsymbol{\theta}; \mathcal{D}) = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathbb{D})$ as the mode of the posterior, *i.e.* as the maximum a posteriori (MAP) estimate. A second-order approximation of the regularized loss around $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}_\star$ then translates into a Gaussian approximation of the posterior—the so-called Laplace approximation [91], *i.e.*

$$\mathcal{L}_\text{reg}(\boldsymbol{\theta}; \mathcal{D}) \approx \underbrace{q(\boldsymbol{\theta}; \mathcal{D})}_{= \frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}_\star)^\top \boldsymbol{H}_\mathcal{D}(\boldsymbol{\theta}-\boldsymbol{\theta}_\star) + \text{const.}} \quad \rightsquigarrow \quad p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{D}), \tag{7.4}$$

with $\boldsymbol{\Sigma}_\mathcal{D} \coloneqq N^{-1} \boldsymbol{H}_\mathcal{D}^{-1}$ (or an approximation thereof). We obtain the predictive uncertainty $p(\boldsymbol{y}_\diamond \mid \boldsymbol{x}_\diamond, \mathbb{D})$ for some test input $\boldsymbol{x}_\diamond \in \mathbb{X}$ by propagating the parameter uncertainty $\mathcal{N}(\boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{D})$ to the model's outputs via the linearized network [66, 117].

Full-Batch vs. Mini-Batch LA. It is common practice to compute the LA on the *entire* training set. Depending on the curvature approximation, this comes at considerable computational costs. For example, a full-batch LA is prohibitive even for moderately sized models/data sets when a low-rank approximation of the Hessian is computed via repeated Hessian-vector products (each of which requires a full pass over the training data); and it is essentially infeasible for model classes like LLMs, which are trained on massive data sets. However, what adds most to the costs is that it is standard to *tune* the prior precision [30]. This adds another outer loop that requires the same procedure to be performed multiple times. Being able to emulate the behavior of the full-batch quadratic on a mini-batch would thus be useful. We therefore study the mini-batch setting, *i.e.* we replace $q(\cdot\,; \mathcal{D})$ by $q(\cdot\,; \mathcal{B})$ in Equation (7.4). We will see in Section 7.6.2 that, when mitigating the associated biases, a mini-batch LA can be a good proxy for the full-batch LA.

7.3 The Shape of a Mini-Batch Quadratic

This section studies the "shape" of a mini-batch quadratic $q(\cdot\,; \mathcal{B})$ and how it differs from $q(\cdot\,; \mathcal{D})$.

7.3.1 Empirical Study of the Directional Slopes and Curvatures

The High-Curvature Subspace Is Relevant for All Common Use Cases. In our empirical study, we focus on the *top*-curvature subspace. This subspace is particularly relevant for several reasons:

1. By the Eckart-Young-Mirsky Theorem, a truncated singular value decomposition (SVD) is Frobenius norm-optimal, *i.e.* a low-rank approximation using the top eigenvectors is ideal from a theoretical perspective. As the spectrum of the Hessian typically decays quickly [44], the bulk of the curvature information is contained in the top-curvature subspace.
2. In the context of optimization, it has been observed that the gradient mainly lives in the high-curvature space [29, 50]. Thus, it makes sense for a second-order method to operate mainly in that space, as steps outside of it can not be expected to reduce the objective function significantly.
3. In the context of the LA, directions of large curvature correspond to directions in the weight space with *low* variance, *i.e.* these directions capture what we *know* about the model's parameters. Consequently, Daxberger et al. [30, p. 19] describe a low-rank approximation of the Hessian based on its top eigenvectors.

Experimental Procedure. We use the same setting as in Figure 7.1: The fully trained All-CNN-C model on the CIFAR-100 data set. To isolate the effect of data subsampling, we eliminate *all other* sources of noise. Thus, we remove the dropout layers from the model. We use the cross-entropy loss, an ℓ^2-regularizer, and train the model with SGD for 350 epochs, see Appendix C.2.1 for details.

We then pick a mini-batch $\mathcal{B}_m$ of size 512 and compute the 100 eigenvectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_{100}$ to the 100 *largest* eigenvalues of $\boldsymbol{H}_{\mathcal{B}_m} \leftarrow \boldsymbol{G}_{\mathcal{B}_m} + \beta \boldsymbol{I}$. That is the Hessian of the L^2-regularized mini-batch loss, where we replaced $\nabla^2 \mathcal{L}(\boldsymbol{\theta}_\star; \mathcal{B}_m)$ with the GGN approximation. Next, we compute the directional slopes and curvatures for *all* mini-batch quadratics $q(\cdot; \mathcal{B}_{m'})$, $m' \in \{1, \dots, M\}$ along those 100 eigenvectors. For a fixed eigenvector $\boldsymbol{u}_p$, the *average* of those directional slopes/curvatures over *all* mini-batches coincides with the directional slope/curvature of the full-batch quadratic, *i.e.*

$$\begin{aligned} \frac{1}{M} \sum_{m'=1}^{M} \underbrace{\partial_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})}_{\text{one ● and many ●}} &= \underbrace{\partial_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{D})}_{\text{✚}} \quad \text{and} \\ \frac{1}{M} \sum_{m'=1}^{M} \underbrace{\partial^2_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})}_{\text{one ● and many ●}} &= \underbrace{\partial^2_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{D})}_{\text{✚}}, \end{aligned} \tag{7.5}$$

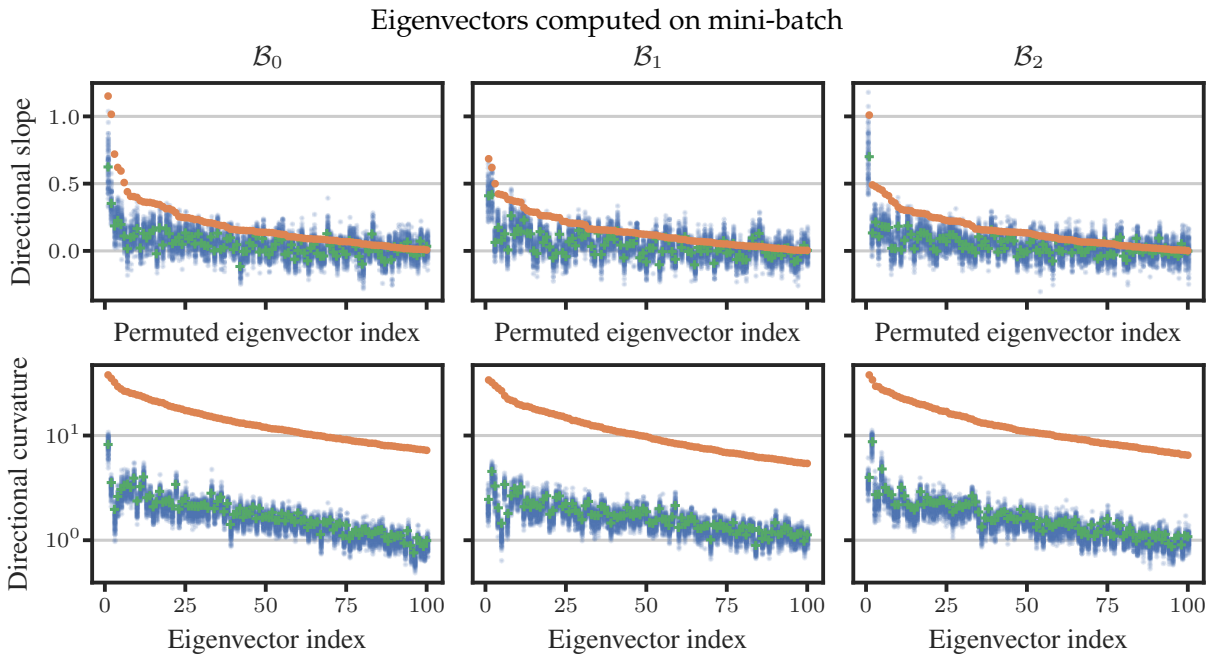


Figure 7.2: Directional Slopes and Curvatures Are Biased. We use the CIFAR-100 data set with the fully trained All-CNN-C model and draw three mini-batches $\mathcal{B}_m$ of size $|\mathcal{B}_m| = 512$ to compute the top 100 eigenvectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_{100}$. For each mini-batch/column, we show the directional slopes *(Top)* and curvatures *(Bottom)* evaluated on (i) $q(\boldsymbol{\theta}_\star; \mathcal{B}_m)$ (*i.e.* on the *same* mini-batch of data) as ●, (ii) $q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})$ for $m' \neq m$ (*i.e.* for all *other* mini-batches) as ● and (iii) the full-batch quadratic (the average of the orange and all blue dots, see Equation (7.5)) as ✚. For the top panel, we switch the order and sign of the eigenvectors such that the orange dots are all above zero and in descending order. There is a strong, systematic bias, particularly in the curvature: Computing the eigenvectors and directional curvatures on the same data results in over-estimation by roughly one order of magnitude.

derivation in Appendix C.1.1. The colored markers in Equation (7.5) refer to Figure 7.2. We repeat this procedure for three mini-batches $\mathcal{B}_m$, $m \in \{0, 1, 2\}$.

Directional Slopes and Curvatures Are Biased. Figure 7.2 reveals a systematic bias in the directional slopes and, more pronounced, in the directional curvatures: When eigenvectors and directional derivatives are computed on the *same* mini-batch, curvature is overestimated by roughly one order of magnitude compared to the curvature of the full-batch quadratic $q(\cdot\,; \mathcal{D})$. Within the space that actually carries curvature information—the top-eigenspace of the quadratic's Hessian—the curvature magnitude is thus not at all representative of the true underlying loss landscape. For *other* mini-batches, the directional slopes and curvatures are similar to the full-batch quadratic. This is because the averages in Equation (7.5) are dominated by these unbiased samples.

We present additional results for batch size 2048 (instead of 512), the Hessian $\nabla^2 \mathcal{L}(\boldsymbol{\theta}_\star; \mathcal{B}_m)$ (instead of its GGN approximation) and CG search directions (instead of eigenvectors) in Appendix C.2.4. The bias is present in *all* settings, but less pronounced for larger batch sizes. For the CG search directions, the bias in the directional slope is much more distinct than for the eigenvectors: The biased

slope is always negative, while the full-batch quadratic's slope is in fact positive for most CG directions!

7.3.2 Where Do the Biases Come From?

In the following, we provide an explanation for the biases in the directional slopes and curvatures (*i.e.* the gap between the orange and a blue dot in Figure 7.2) from a theoretical perspective.

Bias in the Directional Slope

Here, we focus on the CG search directions since the bias in the directional slope is even more pronounced for those directions than for the eigenvectors. This is not accidental! The CG directions are constructed from gradients—and gradients are directions that *maximize* the steepness ● for one particular mini-batch quadratic. For *other* mini-batch quadratics, the steepness along those directions (*i.e.* the directional slope ●) is therefore *less* extreme. We formalize this intuition in Appendix C.1.4.

Bias in the Directional Curvature

Next, we consider the bias in the directional curvature along the eigenvectors of the curvature matrix.

Directional Curvature Along $\boldsymbol{H}_{\mathcal{B}}$'s Eigenvectors. Let $\boldsymbol{u}_1, \dots, \boldsymbol{u}_P$ and $\tilde{\boldsymbol{u}}_1, \dots, \tilde{\boldsymbol{u}}_P$ denote the eigenvectors of two mini-batch Hessians $\boldsymbol{H}_{\mathcal{B}}$ and $\boldsymbol{H}_{\tilde{\mathcal{B}}}$, respectively. Assume that the corresponding eigenvalues are in descending order, *i.e.* $\lambda_1 \geq \ldots \geq \lambda_P$ and $\tilde{\lambda}_1 \geq \ldots \geq \tilde{\lambda}_P$. We can write the directional curvatures on $\mathcal{B}$ and $\tilde{\mathcal{B}}$ along an eigenvector $\boldsymbol{u}_i$ as (details in Appendix C.1.5)

$$\begin{aligned} \underbrace{\partial^2_{\boldsymbol{u}_i} q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})}_{\bullet} &= \boldsymbol{u}_i^\top \boldsymbol{H}_{\tilde{\mathcal{B}}} \boldsymbol{u}_i = \sum_{p=1}^{P} \tilde{\lambda}_p \, \Omega_{i,p}, \quad \text{and} \\ \underbrace{\partial^2_{\boldsymbol{u}_i} q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet} &= \boldsymbol{u}_i^\top \boldsymbol{H}_{\mathcal{B}} \boldsymbol{u}_i = \lambda_i. \end{aligned} \tag{7.6}$$

The weights $\{\Omega_{i,p} \coloneqq (\boldsymbol{u}_i^\top \tilde{\boldsymbol{u}}_p)^2\}_{p=1}^{P}$ are non-negative and sum to one, *i.e.* $\sum_{p=1}^{P} \Omega_{i,p} = 1$. The bias in the directional curvature thus originates from misalignment of the eigenspaces of the two curvature matrices—captured by the weights $\Omega_{i,p}$—and/or a systematic difference in their spectra.

Curvature Bias Is Not Due to Differing Spectra... Assume that the eigenspaces are perfectly aligned, *i.e.* $\Omega_{i,i} = 1$ and $\Omega_{i,p} =$

$0\ \forall p \neq i$. In this case, we obtain $\partial^2_{\boldsymbol{u}_i} q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}}) = \tilde{\lambda}_i$ from Equation (7.6), so the bias originates *exclusively* from the differences in the spectra. Figure 7.2 shows the eigenvalues (as directional curvatures) for three different CIFAR-100 mini-batches. As they are very similar, this cannot serve as the main explanation for the curvature bias.

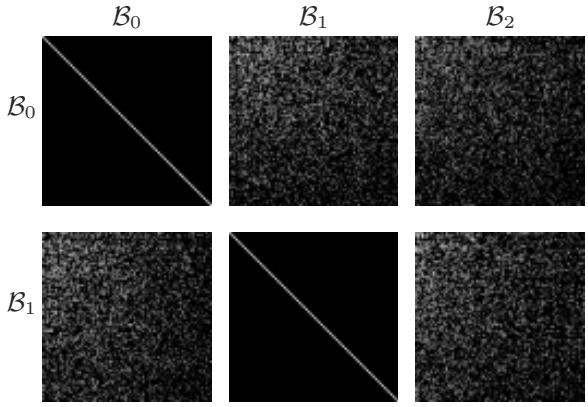
. . . but Due to Misaligned Eigenspaces. Now, assume that the spectra of $\boldsymbol{H}_{\mathcal{B}}$ and $\boldsymbol{H}_{\tilde{\mathcal{B}}}$ are identical, *i.e.* $\lambda_p = \tilde{\lambda}_p\ \forall p \in \{1, \dots, P\}$ which simplifies the unbiased estimate in Equation (7.6) to $\partial^2_{\boldsymbol{u}_i} q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}}) = \sum_{p=1}^{P} \lambda_p\, \Omega_{i,p}$. Consider the curvature along $\boldsymbol{u}_1$ as an example. If $\boldsymbol{u}_1 = \tilde{\boldsymbol{u}}_1$, there is only one non-zero weight $\Omega_{1,1} = 1$ and the two directional curvatures are identical. However, if there is significant overlap with any other eigenvectors (*i.e.* some weight is distributed also on the *lower*-curvature directions), the resulting curvature $\partial^2_{\boldsymbol{u}_1} q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})$ is *smaller* than $\partial^2_{\boldsymbol{u}_1} q(\boldsymbol{\theta}_\bullet; \mathcal{B})$. Analogously, for $\boldsymbol{u}_P$, the directional curvature on $\tilde{\mathcal{B}}$ is *larger* than on $\mathcal{B}$ if there is significant overlap between $\boldsymbol{u}_P$ and some of $\boldsymbol{H}_{\tilde{\mathcal{B}}}$'s *higher*-curvature eigenvectors. This can be formalized by the following inequalities (derivation in Appendix C.1.5):

$$\underbrace{\partial^2_{\boldsymbol{u}_1} q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})}_{\bullet} \leq \underbrace{\partial^2_{\boldsymbol{u}_1} q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet} \quad \text{and} \qquad \underbrace{\partial^2_{\boldsymbol{u}_P} q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})}_{\bullet} \geq \underbrace{\partial^2_{\boldsymbol{u}_P} q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet}. \tag{7.7}$$

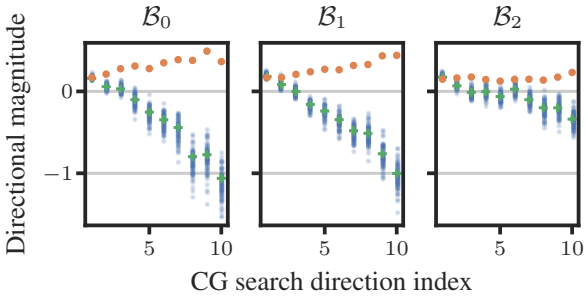
In general, if the eigenspaces are not perfectly aligned such that weight is distributed on several eigenvectors, this leads to *less extreme* directional curvatures on mini-batch $\tilde{\mathcal{B}}$ on *both* ends of the spectrum. Figure 7.3a shows the weights $\Omega_{i,p}$ as pixels. The overlap between the eigenspaces is far from perfect, *i.e.* one eigenvector from $\mathcal{B}$ overlaps with several eigenvectors from $\tilde{\mathcal{B}}$ to some extent. This explains the curvature overestimation in the top curvature subspace we observe in Figure 7.2 and identifies the misalignment of the eigenspaces as the dominating factor for the curvature bias.

Summary of the Theoretical Findings

The CG search directions and the eigenvectors of the curvature matrix are both *designed* to be "extreme" in some sense: The CG directions are based on gradients that maximize steepness of the quadratic $q(\cdot\,; \mathcal{B})$, while the top/bottom eigenvectors subsume directions of largest/smallest curvature. However, these directions are extreme only for one particular mini-batch $\mathcal{B}$. Another mini-batch $\tilde{\mathcal{B}}$ has its own extreme directions that typically differ from those of $\mathcal{B}$. Thus, **the extreme directions for $\mathcal{B}$ are less extreme**



(a) In Practice, Eigenspaces Are Misaligned. We reuse the setting of Figure 7.2 and compute the top 100 eigenvectors $\boldsymbol{U}_m \in \mathbb{R}^{P\times 100}$ for $\{\mathcal{B}_m\}_{m\in\{0,1,2\}}$. The weights $\Omega_{i,j}$ are shown as a 100×100 grayscale image (color ranges from black for $\Omega_{i,j} \leq 10^{-8}$ to white for $\Omega_{i,j} = 1$) for $m \in \{0, 1\}$, $m' \in \{0, 1, 2\}$. Clearly, the eigenspaces for different mini-batches are not perfectly aligned as eigenvectors from $\mathcal{B}_m$ overlap with several eigenvectors from $\mathcal{B}_{m'}$.



(b) CG Update Magnitudes Are Biased. Same setting as Figure C.3 *(Bottom)*. We run CG on $\{\mathcal{B}_m\}_{m\in\{0,1,2\}}$ and show the directional update magnitudes $\tau_1, \ldots, \tau_{10}$ for the first 10 CG steps using (i) the same mini-batch $\mathcal{B}_m$ (as ●), (ii) all other mini-batches (as ●) and (iii) the entire training set (as ✚). The magnitudes are given by the negative ratio of the directional slope and curvature (see Equation (7.3)) and thus inherit the attached biases. Note that most of the update magnitudes that are based on a single mini-batch of data (●) have the wrong sign resulting in detrimental updates in the wrong direction.

for $\tilde{\mathcal{B}}$. Projecting both quadratics onto $\mathcal{B}$'s directions consequently leads to extreme steepness and curvatures for $\mathcal{B}$, but less extreme values for $\tilde{\mathcal{B}}$. This result is an instance of the classic **regression to the mean** [40]: Using an algorithm to find the directions of most extreme steepness/curvature in one particular mini-batch, we must expect the steepness/curvature to be *less* extreme on most *other* batches (and thus also on the entire data set). We can expect this phenomenon to occur for other data sets, models, and curvature proxies as well, since the underlying mechanism is the stochasticity of the geometric information.

7.4 Debiasing Mini-Batch Quadratics for Applications

We here argue that the biases affect second-order applications, and develop debiasing strategies.

7.4.1 Implications for Second-Order Optimization and the Laplace Approximation

Detrimental Updates in Second-Order Optimizers. In the context of second-order optimization, both the biases in the directional slopes and curvatures are relevant. From Equation (7.3), the CG update magnitude τ_p to reach the minimum of $q(\boldsymbol{\theta}_p + \tau \boldsymbol{d}_p; \mathcal{B}_m)$ is given by the negative ratio of the directional slope and curvature at the current iterate $\boldsymbol{\theta}_p$ in the direction $\boldsymbol{d}_p$—that is the 1D Newton step along that cut.[2] As both these quantities are biased, so is τ_p

2: The exact same argument can also be made for the eigenvector directions as the Newton step can be decomposed into 1D Newton steps along the eigenvectors.

as shown in Figure 7.3b: While the update magnitudes for $\mathcal{B}_m$ are *always positive* (a property of CG), minimizing the other—*equally valid*—quadratics would require *negative* update magnitudes for most of the CG directions. In this sense, naive CG on a single mini-batch of data makes updates in the wrong direction. This can be attributed to the bias in the directional slope since the slope determines the sign of τ_p. Overestimation of the directional curvature is "accidentally beneficial" in this case as it leads to smaller steps.

Unreliable Uncertainty Quantification with the Laplace Approximation. For the LA, only the bias in the directional curvature is relevant. After Section 7.2.3, the approximate posterior covariance over the network's parameters is given by $\boldsymbol{\Sigma}_{\mathcal{B}} = N^{-1}\boldsymbol{H}_{\mathcal{B}}^{-1}$. Via the eigendecomposition $\boldsymbol{H}_{\mathcal{B}} = \sum_{p=1}^{P} \lambda_p \boldsymbol{u}_p \boldsymbol{u}_p^\top$, we obtain $\boldsymbol{\Sigma}_{\mathcal{B}} = N^{-1} \sum_{p=1}^{P} \lambda_p^{-1} \boldsymbol{u}_p \boldsymbol{u}_p^\top$. Due to the biases we describe in Section 7.3, the directional curvatures λ_p are not representative of the true underlying curvature, resulting in a *deformed* posterior covariance $\boldsymbol{\Sigma}_{\mathcal{B}}$. Specifically, the overestimation of the curvature in the top-curvature subspace translates into an *underestimation* of the uncertainty in the posterior covariance (due to the inversion) with potentially severe consequences in safety-critical applications.

7.4.2 Debiasing Strategies

We now turn to strategies that mitigate the biases in second-order optimization and the LA. An empirical evaluation of these methods follows in Sections 7.6.1 and 7.6.2.

Decoupling Directions from Magnitudes with a Two-Batch Strategy. The approaches we propose are simple yet effective. The idea is to commit to the imperfect directions from one mini-batch (as before) but use an *additional* mini-batch to estimate the directional derivatives. With this, we *decouple* the mechanism that determines the parameter subspace in which the method operates from the slope and curvature measurements within the space. Effectively, we use the blue dots from Figures 7.2 and 7.3b instead of the orange dots and thus obtain much more realistic estimates of the actual loss function's geometry (within the subspace defined by the first mini-batch). Next, we describe in more detail how this strategy can be applied to debias CG and the LA.

Debiased Conjugate Gradients

Debiased Approach. For the debiased CG method, we need to implement two processes: (i) The first process applies $K \leq P$ CG iterations to the mini-batch quadratic $q(\,\cdot\,; \mathcal{B})$ and collects the

search directions $(\boldsymbol{d}_1, \dots, \boldsymbol{d}_K)$. As before, this defines the subspace in which CG operates. (ii) The second process recomputes the trajectory $(\tilde{\boldsymbol{\theta}}_1, \dots, \tilde{\boldsymbol{\theta}}_K)$ within that subspace using debiased update magnitudes that are computed on a different mini-batch $\tilde{\mathcal{B}}$, *i.e.* we use $\tilde{\boldsymbol{\theta}}_0 \coloneqq \boldsymbol{\theta}_0$ and

$$\tilde{\boldsymbol{\theta}}_{p+1} \coloneqq \tilde{\boldsymbol{\theta}}_p + \tilde{\tau}_p \boldsymbol{d}_p \quad \text{with} \quad \underbrace{\tilde{\tau}_p}_{\bullet} \coloneqq -\frac{\partial_{\boldsymbol{d}_p} q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}})}{\partial^2_{\boldsymbol{d}_p} q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}})} \qquad \text{instead of} \quad \underbrace{\tau_p}_{\bullet} = -\frac{\partial_{\boldsymbol{d}_p} q(\tilde{\boldsymbol{\theta}}_p; \mathcal{B})}{\partial^2_{\boldsymbol{d}_p} q(\tilde{\boldsymbol{\theta}}_p; \mathcal{B})}. \tag{7.8}$$

For $\tilde{\mathcal{B}} = \mathcal{B}$, $(\tilde{\boldsymbol{\theta}}_1, \dots, \tilde{\boldsymbol{\theta}}_K)$ is congruent with the original trajectory $(\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_K)$ from the single-batch CG approach. If the two mini-batches are different, the debiased trajectory will use updates into the directional minima of $q(\cdot\,; \tilde{\mathcal{B}})$ instead of $q(\cdot\,; \mathcal{B})$. Processes (i) and (ii) can either be run side by side in an alternating fashion (offering more flexibility regarding *e.g.* the termination criterion at the cost of a small memory overhead, details in Appendix C.1.2) or one after the other.

Debiased Laplace Approximation

Laplace Approximation with K-FAC. We briefly explore another popular curvature proxy: Kronecker-Factored Approximate Curvature (K-FAC), which is commonly used both in optimization [37, 96, 97] and uncertainty quantification [115]. It is a block-diagonal approximation to the FIM $\boldsymbol{F}_{\mathcal{B}} \approx \boldsymbol{K}_{\mathcal{B}} \coloneqq \text{blockdiag}_{l=1,\dots,L}(\boldsymbol{K}^{(l)}_{\mathcal{B}})$, where each block $\boldsymbol{K}^{(l)}_{\mathcal{B}} \coloneqq \boldsymbol{A}^{(l)} \otimes \boldsymbol{B}^{(l)}$ is approximated by the Kronecker product of two smaller matrices $\boldsymbol{A}^{(l)}$ and $\boldsymbol{B}^{(l)}$. Sampling from the respective LA with covariance $\boldsymbol{\Sigma}_{\mathcal{B}} = N^{-1}(\boldsymbol{K}_{\mathcal{B}} + \beta \boldsymbol{I})^{-1}$ can be performed efficiently due to the specific structure of the K-FAC approximation. For details, see Appendix C.1.3.

Debiased Approach. For the debiased LA, we use two K-FAC approximations $\boldsymbol{K}_{\mathcal{B}}$ and $\boldsymbol{K}_{\tilde{\mathcal{B}}}$ computed on different mini-batches. Via the eigendecomposition $\boldsymbol{K}_{\mathcal{B}} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$ with $\boldsymbol{U} = (\boldsymbol{u}_1, \dots, \boldsymbol{u}_P)$ and $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \dots, \lambda_P)$, we can re-write the covariance matrix as $\boldsymbol{\Sigma}_{\mathcal{B}} = N^{-1}(\boldsymbol{K}_{\mathcal{B}} + \beta \boldsymbol{I})^{-1} = N^{-1}\boldsymbol{U}(\boldsymbol{\Lambda} + \beta \boldsymbol{I})^{-1}\boldsymbol{U}$. For the debiased approach, we keep the eigenspace defined by $\boldsymbol{K}_{\mathcal{B}}$ but recompute the directional curvatures based on $\boldsymbol{K}_{\tilde{\mathcal{B}}}$, *i.e.* we use

$$\tilde{\boldsymbol{\Sigma}}_{\mathcal{B}} = \frac{1}{N}\boldsymbol{U}\left(\text{diag}(\tilde{\lambda}_1, \dots, \tilde{\lambda}_P) + \beta \boldsymbol{I}\right)^{-1}\boldsymbol{U}^\top \quad \text{with} \quad \underbrace{\tilde{\lambda}_p}_{\bullet} = \boldsymbol{u}_p^\top \boldsymbol{K}_{\tilde{\mathcal{B}}} \boldsymbol{u}_p \qquad \text{instead of} \quad \underbrace{\lambda_p}_{\bullet} = \boldsymbol{u}_p^\top \boldsymbol{K}_{\mathcal{B}} \boldsymbol{u}_p. \tag{7.9}$$

Computational Cost of Debiasing

Both debiasing techniques roughly double run time compared to their single-batch counterparts: Debiased CG performs one extra matrix-vector product with $\boldsymbol{H}_{\tilde{\mathcal{B}}}$ per CG iteration to compute the debiased update magnitude (details in Appendix C.1.2). Debiased LA requires an additional K-FAC approximation $\boldsymbol{K}_{\tilde{\mathcal{B}}}$; all subsequent operations to compute the debiased K-FAC can be carried out efficiently at Kronecker factor level (details in Appendix C.1.3). In Section 7.6, we thus use the debiased versions at *half* the batch size, for a fair comparison. We will see that, at the resulting *similar* computational cost, the debiased versions clearly outperform the single-batch alternatives.

7.5 Related Work

We here briefly list other, related forms of bias correction that have been suggested elsewhere.

A Different Two-Batch Approach. Other works have proposed to use different (not necessarily disjoint) mini-batches for the gradient and the Hessian [14, 94]. Benzing [7, Sec. I.5] mentions the idea of using *independent* mini-batches for the gradient and the Hessian to obtain an, in some sense, unbiased estimate $-\boldsymbol{H}_{\mathcal{B}}^{-1}\boldsymbol{g}_{\tilde{\mathcal{B}}}$ of the exact Newton step. This does, however, not resolve the biases described in this paper. Via the eigendecomposition of the Hessian $\boldsymbol{H}_{\mathcal{B}} = \sum_{p=1}^{P} \lambda_p \boldsymbol{u}_p \boldsymbol{u}_p^\top$, we obtain $-\boldsymbol{H}_{\mathcal{B}}^{-1}\boldsymbol{g}_{\tilde{\mathcal{B}}} = -\sum_{p=1}^{P} \partial_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \tilde{\mathcal{B}}) (\partial^2_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{B}))^{-1} \boldsymbol{u}_p$. While the numerator yields an unbiased estimate of the directional slope (similar to a *blue* dot in the upper panel of Figure 7.2), the bias in the denominator remains since eigenvectors and directional curvatures are based on the same mini-batch (as for the *orange* dots in the bottom panel of Figure 7.2).

Another Debiasing Approach. EKFAC (Eigenvalue-corrected Kronecker Factorization) [43] corrects the eigenvalues of the K-FAC approximation (similar to Equation (7.9)) which, provably, yields a more accurate approximation of the FIM than K-FAC (in Frobenius norm). This correction, however, is designed to resolve a different kind of bias that is specific to the K-FAC approximation and does not address the biases described in this work.

Running Averages. Other popular deep learning optimizers *aggregate* curvature information over multiple steps via exponential moving averages (see *e.g.* [97]). This emulates larger mini-batch sizes and thus reduces the curvature biases. However, when curvature evolves rapidly, obsolete curvature estimates from past steps

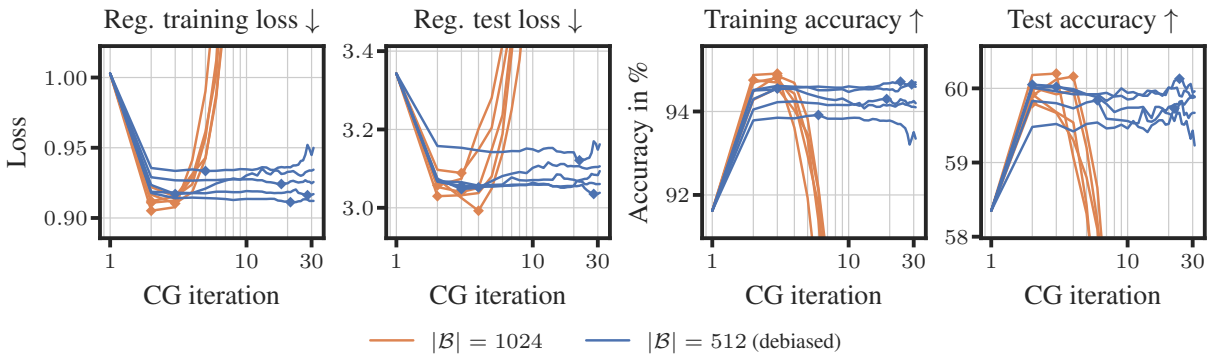


Figure 7.4: Debiased CG Is Much More Stable than the Single-Batch Approach. We compare CG runs without curvature damping ($\delta = 0$) with $K = 30$ iterations for the fully trained ALL-CNN-C model on the CIFAR-100 data set in terms of training/test loss/accuracy at similar computational cost: The single-batch approach (shown as —) uses one mini-batch of size 1024 while the debiased approach (shown as —) uses two mini-batches of size 512 each. Both approaches use the GGN curvature proxy and are run 5 times on different mini-batches. The markers ◆ and ◆ are placed at peak performance. While the single-batch runs diverge quickly, the debiased CG runs are stable.

might slow down training. Aggregating more robust *debiased* curvature estimates instead might allow for shorter moving average windows and accelerate the optimization progress. We leave it to future work to explore these interactions.

7.6 Experiments

In this section, we evaluate the effectiveness of the debiasing strategies from Section 7.4.2. In Appendix C.2, we provide the experimental details as well as additional empirical analyses. For instance, we show how the curvature biases with K-FAC depend on the mini-batch size (see Appendix C.2.8), how the biases evolve over the course of training (see Appendix C.2.9), and that the curvature biases become more pronounced for deeper/wider models (see Appendix C.2.10).

7.6.1 Debiased Conjugate Gradients

We compare the standard single-batch CG method to the debiased version (see Section 7.4.2) on the fully trained ALL-CNN-C model without curvature damping. For a fair comparison, the single-batch approach uses one mini-batch of size 1024 while the debiased approach uses two mini-batches of size 512, such that a similar amount of data and run time budget is used. Details in Appendix C.2.5.

Results & Discussion. All CG runs in Figure 7.4 achieve a significant improvement in all four performance metrics. The most striking difference between the two approaches is their stability: The single-batch runs quickly reach peak performance and then

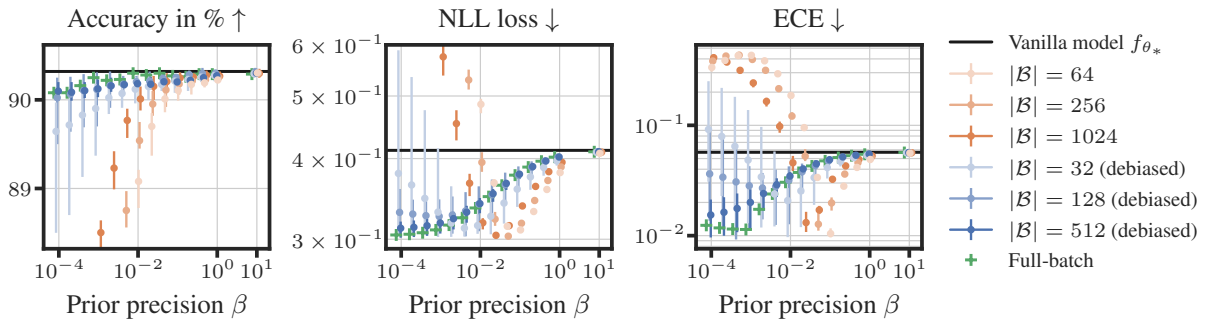


Figure 7.5: Debiased LA Mimics the Full-Batch LA Very Well. We compare LAs for the fully trained ALL-CNN-C model on CIFAR-10 in terms of accuracy, negative log likelihood loss (NLL) and expected calibration error (ECE) on the CIFAR-10 test set. For each mini-batch size (lighter color indicating smaller batch size), we draw 5 mini-batches and report the mean performance as dot and min/max as vertical line. We also report the performance of the vanilla model without LA (shown as —) and the full-batch approach based on $\boldsymbol{K}_{\mathcal{D}}$ (shown as ✚). In contrast to the single-batch approach, the debiased version mimics the behavior of the full-batch approach very well over the entire range of prior precisions.

diverge. In contrast, although the steps tend to be *larger* (see Figure 7.3b), the debiased CG runs are much more stable (without using any curvature damping). This suggests that the reason for the divergence in the single-batch approach is *not* the missing damping but the misinformed update magnitudes. The peak performance is slightly better for the single-batch runs which can be attributed to its more informative search directions (they were computed using *double* the data). The peak performance of the debiased runs could likely be improved (at the same computational cost) by using a larger batch size for the directions and a smaller one for the update magnitudes (as the former seem harder to estimate, see Section 7.3.2).

7.6.2 Debiased Laplace Approximation

Here, we use a fully trained ALL-CNN-C model on CIFAR-10 data and compare (i) the vanilla model without LA, (ii) the single-batch K-FAC LA approach, (iii) the debiased version (see Section 7.4.2), and (iv) the full-batch approach (where we compute K-FAC on the entire training set) in terms of accuracy, NLL and ECE. Again, we apply the debiased approach at half the batch size of the single-batch approach for a fair comparison. We use prior precisions between 10^{-4} and 10. Appendix C.2.6 contains the experimental details and additional results on the training and OOD data.

Results & Discussion. Figure 7.5 shows the results. If the prior precision is low (*i.e.* the LA relies mainly on the curvature information without the regularizing diagonal term), the single-batch version acts erratically due to the deformed curvature model—its performance drops drastically. The single-batch approach is also more sensitive to the choice of prior precision (in fact, it suggests a much larger prior precision than the full-batch approach). In

contrast, the debiased approach achieves good performance over a wider range of prior precisions and mimics the full-batch approach ($N = 40\,000$) very well despite using only a tiny fraction of the data.

In order to showcase our debiasing strategy's scalability, we provide additional results on ResNet-50 and ViT Little on the ImageNet data set in Appendix C.2.6.

Summary of the Experimental Results. The use of mini-batch quadratics is a simple way to keep the costs of second-order optimization and uncertainty quantification manageable. The resulting biases, however, severely degrade their value, requiring large mini-batches or algorithmic add-ons. Our experiments suggest that even simple debiasing strategies can largely mitigate this issue.

7.7 Conclusion

Our main takeaway is a general principle: **Quadratic approximations to the training loss computed on *mini-batches* of the training data provide a *distorted* representation of the true underlying loss landscape.** In particular, along the directions of large curvature, the mini-batch quadratics tend to strongly overestimate the curvature of the true loss. Our theoretical analysis shows that these biases can be traced back to the misalignment of the curvature matrices' eigenspaces. These insights are highly relevant for applications: As we demonstrated empirically, the biases in the directional slope and curvature lead to severely misinformed updates in stochastic second-order optimizers, and cause unreliable uncertainty estimates with Laplace approximations. We also proposed simple two-batch strategies to mitigate these biases. Our experiments demonstrate their superiority over the single-batch approaches in terms of stability and quality at similar computational costs. Our findings reveal a design prerequisite for building better stochastic curvature-based methods, which should be generally considered, and further developed, for all methods using such curvature evaluations.

Acknowledgments

The authors gratefully acknowledge co-funding by the European Union (ERC, ANUBIS, 101123955). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. Philipp Hennig is a member of the

Machine Learning Cluster of Excellence, funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC number 2064/1 - Project number 390727645; The authors further gratefully acknowledge financial support by the DFG through Project HE 7114/5-1 in SPP2298/1; 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. Lukas Tatzel and Bálint Mucsányi are grateful to the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for support. Further, the authors thank Felix Dangel and Runa Eschenhagen for providing feedback to the manuscript.

Part III

Conclusion & Outlook

8 Conclusion & Outlook

We conclude this thesis with a summary of our findings (see Section 8.1) and a discussion of potential directions for future research (see Section 8.2).

8.1 Summary

Chapter 1 introduced second-order optimizers and uncertainty quantification via the Laplace approximation as two key applications that rely on curvature information through a quadratic Taylor expansion. However, a naive use of curvature is computationally prohibitive, even for models and data sets of moderate size. Computational costs are thus typically the primary concern when applying curvature-based methods to large-scale machine learning problems. The deep learning regime introduces additional challenges, such as the accessibility of curvature information and the stochasticity of gradient and curvature estimates (for details, see Chapter 4). Our work, presented in Part II, addresses these challenges. We briefly summarize the main contributions below.

Approximate Inference Without Sacrificing Reliability. Chapter 5 introduced an iterative inference scheme for non-conjugate Gaussian processes (NCGPs)—a flexible and widely-used model class. As exact inference in NCGPs is computationally infeasible, practical methods rely on approximations. Our method, IterNCGP, captures the resulting approximation error as an additional source of uncertainty, which allows us to trade off reduced computation for increased uncertainty. IterNCGP is matrix-free and uses GPU accelerated matrix-vector products to compute approximate Newton steps, which enables our method to scale. Additional speedups are achieved by reusing intermediate results across steps. In summary, IterNCGP allows for large-scale probabilistic inference in non-conjugate Gaussian processes without sacrificing reliability.

Easy and Efficient Access to Rich Curvature Quantities. Chapter 6 presented ViViT, an open-source software library for accessing curvature information in the form of the generalized Gauss-Newton (GGN) matrix—a popular curvature proxy in deep learning. In contrast to Kronecker-factored approximations that do not become exact in any limit and neglect curvature between layers, ViViT is able to compute exact GGN quantities and implements principled approximations to reduce costs in exchange for decreased accuracy.

By exploiting the GGN's low-rank structure, ViViT makes a variety of curvature quantities (*e.g.* the GGN's spectum, eigenvectors, and per-sample directional slopes and curvatures) efficiently available to researchers and practitioners. Importantly, per-sample curvatures provide a notion of curvature noise offering valuable insights into the challenges of designing effective second-order methods.

More Effective Curvature-Based Methods Through Debiasing. Chapter 7 investigated the fundamental limitations of using quadratic approximations to the loss based on mini-batches of training data in deep learning. While sub-sampling is essential for scalability, it introduces a systematic bias, heavily affecting second-order optimization and Laplace-based uncertainty quantification. Our work provided a theoretical explanation for the bias, explained implications for curvature-based methods, and proposed debiasing strategies that provide more meaningful mini-batch approximations to the loss landscape at negligible computational overhead.

Conclusion. This thesis advances curvature-based methods by making them (i) more *efficient* through methodological innovations, (ii) more *accessible* through software implementations, and (iii) more *effective* and *reliable* by acknowledging and addressing inherent approximation errors—thereby supporting their broader adoption in machine learning research and practice.

Are Curvature-Based Optimizers Really Too Expensive? A common belief—particularly within the deep learning community—is that curvature-based methods are prohibitively expensive for practical use. While these methods do incur higher computational costs than first-order approaches, we think that their potential to significantly enhance optimization performance justifies continued research. As our work shows, it is indeed possible to scale curvature-based methods to practically relevant scales by (i) *condensing* curvature information into lower-cost yet expressive internal representations (*e.g.* the compressed low-rank belief for IterNCGP, the Gram matrix for ViViT), (ii) exploiting the *specific structure* of the problem (*e.g.* by recycling costly computations in IterNCGP or by exploiting the GGN's low-rank structure with ViViT), and (iii) implementing further *principled approximations* (*e.g.* ViViT's MC approximation and the mini-batch approximation discussed in Chapter 7). We hope that these advances encourage further research into curvature-based methods.

8.2 Future Research Directions

Several natural extensions of the work presented in this thesis are already discussed in the individual Chapters 5 to 7. Rather than reiterating these points, this section focuses on broader, longer-term research directions that we consider particularly interesting.

8.2.1 Beyond Curvature: Expanding the Scope of Debiasing

Exploring the Use of Debiasing in Practical Applications. Chapter 7 addressed a *fundamental* limitation of stochastic second-order methods: A quadratic approximation computed on a single mini-batch of data yields a deformed representation of the true underlying loss landscape. Through debiasing, we obtain a more reliable approximation. In principle, *all* curvature-based methods from Chapter 3 might benefit from these insights, provided that they rely on mini-batch curvature estimates. Exploring this potential in practical settings is a logical direction for future work.

Expanding Debiasing to Covariance Matrices. It seems that we can extend the use cases of our insights from Chapter 7 even further. Essentially, the origin of the curvature biases lies in the fact that the directions of very large/small curvature only apply to a *specific* mini-batch—on other mini-batches, we therefore obtain less extreme curvatures within the respective subspaces. It is plausible that this principle also applies to other matrix-valued stochastic estimates, *e.g.* covariance matrices evaluated on mini-batches. As a consequence, we would overestimate/underestimate the directional *variance* in the subspace spanned by the top/bottom eigenvectors, respectively. This would be particularly relevant for optimizers that rely on the empirical Fisher (*i.e.* the un-centered gradient covariance matrix, see Section 3.4.1) as a preconditioner, *e.g.* for the Shampoo optimizer [5, 49, 126] that recently gained popularity in the deep learning community.[1]

1: Shampoo recently won the AlgoPerf: Training Algorithms competition [24, 69] (in the external tuning ruleset), an extensive optimization efficiency benchmark that compared popular optimizers across multiple deep learning workloads.

8.2.2 From Static Update Rules Towards Resource-Efficient Autonomous Agents

A Long-Term Vision. Currently, training neural networks often requires a high degree of human intervention, *e.g.* with regard to the selection of hyperparameters or the monitoring of the training process. A long-term vision for optimization methods in deep learning is therefore that they evolve into resource-efficient, autonomous agents that actively collect and process relevant information and take intelligent decisions on that basis. In the following, we outline

two research directions that could support this development, with a particular focus on curvature-based methods.

Recognizing and Leveraging Stochasticity. We believe that a critical aspect in the design of new optimizers is the explicit acknowledgment that geometric information computed from a mini-batch is inherently *stochastic*. Chapter 7 provides strong evidence for this perspective: It showed that second-order methods are affected by noise in an intricate way and a deeper understanding of the underlying mechanisms is crucial to develop more effective algorithms. The success of variance-adapted first-order methods such as Adam [74] further highlights the importance of incorporating stochasticity into the updates. In contrast, most second-order methods remain largely agnostic to the noise in curvature estimates. While debiasing addresses systematic error, incorporating curvature noise could further enhance the robustness of second-order methods. The ViViT library (see Chapter 6) provides such observables in the form of per-sample directional derivatives. These *distributions* over slopes and curvatures are substantially more informative than point estimates, as they capture both the local geometry of the loss landscape *and* the associated uncertainty. Leveraging such quantities could enable the development of more robust and efficient noise-aware second-order methods (*e.g.* through variance-adapted updates or dynamic batch size adaptation), better suited for the inherently noisy regime of mini-batch training.

Revisiting Late-Phase Second-Order Training. Optimizers typically progress through different training phases—from an initial phase of randomness and instability to a final fine-tuning phase. One intermediate goal towards more autonomous training is to develop efficient strategies tailored to the specific demands of each phase. In the late stage, once the model has reached a region of low loss, precise steps toward the optimum become particularly valuable. Current approaches based on first-order methods usually rely on manually tuned, decaying learning rate schedules to enforce "convergence". An alternative is to replace this approach with a few computationally expensive, but accurate, Newton-based updates. This approach is particularly promising as second-order updates are robust against ill-conditioned curvature and exhibit fast convergence close to the local optimum. We previously explored this idea in preliminary work [133], but obtained mixed results—likely due to biased curvature estimates. Revisiting this approach using *debiased* Newton steps may yield a more effective fine-tuning strategy, potentially removing the need for manual tuning of the decay schedule.

Part IV

Appendix

A Additional Material for Chapter 5

The supplementary materials contain derivations for our theoretical framework and proofs for the mathematical statements in the main text. We also provide implementation specifics and describe our experimental setup in more detail.

A.1 Mathematical Details

A.1.1 Newton's Method as Sequential GP Regression

In Section 5.3.1, we reinterpret the Newton iteration as a sequence of GP regression problems. More specifically, we rewrite the posterior predictive mean (Equation (5.4)) as a GP posterior for a regression problem (Equation (5.6)). Here, we provide a proof for this connection.

Proposition A.1 (Reformulation of the Newton Step) Let $\boldsymbol{W}(\boldsymbol{f}_i)$ be invertible. Using the transform $\boldsymbol{g} := \boldsymbol{f} - \boldsymbol{m}$ and consequently $\boldsymbol{g}_i = \boldsymbol{f}_i - \boldsymbol{m}$, the Newton step (Equation (5.3)) can be written as

$$\boldsymbol{g}_{i+1} = \boldsymbol{K}(\boldsymbol{K} + \boldsymbol{W}(\boldsymbol{f}_i)^{-1})^{-1} \left(\boldsymbol{g}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1} \nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right).$$

Proof. Recall from Equation (5.3) that

$$\begin{aligned} \boldsymbol{f}_{i+1} &= \boldsymbol{f}_i - \nabla^2 \Psi(\boldsymbol{f}_i)^{-1} \cdot \nabla \Psi(\boldsymbol{f}_i), \quad \text{with} \\ \nabla \Psi(\boldsymbol{f}_i) &= \nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i) - \boldsymbol{K}^{-1}(\boldsymbol{f}_i - \boldsymbol{m}) \\ \nabla^2 \Psi(\boldsymbol{f}_i) &= -\boldsymbol{W}(\boldsymbol{f}_i) - \boldsymbol{K}^{-1}, \end{aligned}$$

where $\boldsymbol{W}(\boldsymbol{f}_i) = -\nabla^2 \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)$ denotes the negative Hessian (with respect to $\boldsymbol{f}$) of the log-likelihood evaluated at $\boldsymbol{f}_i$. It holds

$$\begin{aligned} &\boldsymbol{f}_{i+1} \\ &= \boldsymbol{f}_i - \nabla^2 \Psi(\boldsymbol{f}_i)^{-1} \cdot \nabla \Psi(\boldsymbol{f}_i) \\ &= \boldsymbol{f}_i + (\boldsymbol{W}(\boldsymbol{f}_i) + \boldsymbol{K}^{-1})^{-1} \cdot \left(\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i) - \boldsymbol{K}^{-1}(\boldsymbol{f}_i - \boldsymbol{m})\right) \end{aligned}$$

By subtracting $\boldsymbol{m}$ from both sides we obtain

$$\begin{aligned} &\boldsymbol{g}_{i+1} \\ &= \boldsymbol{g}_i + (\boldsymbol{W}(\boldsymbol{f}_i) + \boldsymbol{K}^{-1})^{-1} \cdot \left(\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i) - \boldsymbol{K}^{-1}\boldsymbol{g}_i\right) \\ &= (\boldsymbol{W}(\boldsymbol{f}_i) + \boldsymbol{K}^{-1})^{-1} \left((\boldsymbol{W}(\boldsymbol{f}_i) + \boldsymbol{K}^{-1})\boldsymbol{g}_i + \nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i) - \boldsymbol{K}^{-1}\boldsymbol{g}_i\right) \end{aligned}$$

$$
\begin{aligned}
&= (\boldsymbol{W}(\boldsymbol{f}_i) + \boldsymbol{K}^{-1})^{-1} \left(\boldsymbol{W}(\boldsymbol{f}_i)\boldsymbol{g}_i + \nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right) \\
&= (\boldsymbol{W}(\boldsymbol{f}_i) + \boldsymbol{K}^{-1})^{-1}\boldsymbol{W}(\boldsymbol{f}_i) \left(\boldsymbol{g}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right) \\
&= (\boldsymbol{I} + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\boldsymbol{K}^{-1})^{-1} \left(\boldsymbol{g}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right) \\
&= (\boldsymbol{K}\boldsymbol{K}^{-1} + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\boldsymbol{K}^{-1})^{-1} \left(\boldsymbol{g}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right) \\
&= \boldsymbol{K}(\boldsymbol{K} + \boldsymbol{W}(\boldsymbol{f}_i)^{-1})^{-1} \left(\boldsymbol{g}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right) \\
&= \boldsymbol{K}\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1} \left(\boldsymbol{g}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right),
\end{aligned}
$$

with $\hat{\boldsymbol{K}}(\boldsymbol{f}_i) \coloneqq \boldsymbol{K} + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}$. □

Newton's Method as Sequential GP Regression. Using the LA at $\boldsymbol{f}_i$, we obtain a GP posterior (see Equations (5.4) and (5.5) in Section 5.2). With Proposition A.1 (*i.e.* assuming that $\boldsymbol{W}(\boldsymbol{f}_i)^{-1}$ exists), we can rewrite Equation (5.4) as

$$
\begin{aligned}
&m_{i,*}(\cdot) \\
&\quad = m(\cdot) + K(\cdot, \boldsymbol{X})\boldsymbol{K}^{-1}(\boldsymbol{f}_{i+1} - \boldsymbol{m}) \\
&\quad = m(\cdot) + K(\cdot, \boldsymbol{X})\boldsymbol{K}^{-1}\boldsymbol{g}_{i+1} \\
&\quad = m(\cdot) + K(\cdot, \boldsymbol{X})\boldsymbol{K}^{-1}\boldsymbol{K}\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1} \left(\boldsymbol{g}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)\right) \\
&\quad = m(\cdot) + K(\cdot, \boldsymbol{X})\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1}\left(\boldsymbol{f}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i) - \boldsymbol{m}\right) \\
&\quad = m(\cdot) + K(\cdot, \boldsymbol{X})\hat{\boldsymbol{K}}(\boldsymbol{f}_i)^{-1}(\hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m}),
\end{aligned}
$$

where $\hat{\boldsymbol{y}}(\boldsymbol{f}_i) \coloneqq \boldsymbol{f}_i + \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_i)$. This proves Equation (5.6). Together with Equation (5.5), $m_{i,*}$ defines a GP posterior for a GP regression problem with pseudo-targets $\hat{\boldsymbol{y}}(\boldsymbol{f}_i)$ observed with Gaussian noise $\mathcal{N}\left(\boldsymbol{0}, \boldsymbol{W}(\boldsymbol{f}_i)^{-1}\right)$ [114, Eqs. (2.24) and (2.38)].

Equation (5.6) requires solving the linear system $\hat{\boldsymbol{K}}(\boldsymbol{f}_i) \cdot \boldsymbol{v} = \hat{\boldsymbol{y}}(\boldsymbol{f}_i) - \boldsymbol{m}$ of size NC. Then, $m_{i,*}(\cdot) = m(\cdot) + K(\cdot, \boldsymbol{X})\boldsymbol{v}$. In Proposition A.1, we can write $\boldsymbol{g}_{i+1}$ as $\boldsymbol{g}_{i+1} = \boldsymbol{K}\boldsymbol{v}$, *i.e.* $\boldsymbol{f}_{i+1} = \boldsymbol{K}\boldsymbol{v} + \boldsymbol{m}$. So, *both* the predictive mean $m_{i,*}$ *and* the Newton update $\boldsymbol{f}_{i+1}$ follow directly from the solution $\boldsymbol{v}$. In that sense, performing inference and computing Newton iterates are equivalent.

What If $\boldsymbol{W}(\boldsymbol{f}_i)$ Is *Not* Invertible? For multi-class classification, $\boldsymbol{W}$ has rank $N(C-1)$ and thus $\boldsymbol{W}^{-1}$ does not exist. Therefore, we use its pseudo-inverse $\boldsymbol{W}^{\dagger}$ instead. We derive an explicit expression for $\boldsymbol{W}^{\dagger}$ in Appendix A.1.6 which allows for efficient matrix-vector multiplies.

A.1.2 Our Algorithm is an Extension of IterGP

Our algorithm IterNCGP uses IterGP as a core building block. IterNCGP's outer loop (Algorithm 5.1) can be understood as a sequence of GP regression problems, and we use IterGP (that

implements the inner loop, see Algorithm 5.2) for finding approximate solutions to each of these problems. In the case of GP regression (*i.e.* with a Gaussian likelihood), the outer loop collapses to a *single* iteration and IterNCGP coincides exactly with IterGP, as we show in the following.

Theorem A.2 (Generalization of IterGP) For a Gaussian likelihood $p(\boldsymbol{y} \mid \boldsymbol{f}) = \mathcal{N}(\boldsymbol{y}; \boldsymbol{f}, \boldsymbol{\Lambda})$, IterNCGP converges in a single Newton step (*i.e.* $\boldsymbol{f}_1 = \boldsymbol{f}_\star$) and IterNCGP (Algorithm 5.1) coincides exactly with IterGP (Algorithm 5.2).

Proof. Since the likelihood is Gaussian $p(\boldsymbol{y} \mid \boldsymbol{f}) = \mathcal{N}(\boldsymbol{y}; \boldsymbol{f}, \boldsymbol{\Lambda})$, the log-likelihood is given by

$$\log p(\boldsymbol{y} \mid \boldsymbol{f}) \overset{\mathrm{c}}{=} -\frac{1}{2}(\boldsymbol{f} - \boldsymbol{y})^\top \boldsymbol{\Lambda}^{-1}(\boldsymbol{f} - \boldsymbol{y}).$$

This gives rise to a log-posterior (Equation (5.2))

$$\begin{aligned}\Psi(\boldsymbol{f}) &:= \log p(\boldsymbol{f} \mid \boldsymbol{X}, \boldsymbol{y}) \\ &\overset{\mathrm{c}}{=} \log p(\boldsymbol{y} \mid \boldsymbol{f}) - \frac{1}{2}(\boldsymbol{f} - \boldsymbol{m})^\top \boldsymbol{K}^{-1}(\boldsymbol{f} - \boldsymbol{m}) \\ &= -\frac{1}{2}(\boldsymbol{f} - \boldsymbol{y})^\top \boldsymbol{\Lambda}^{-1}(\boldsymbol{f} - \boldsymbol{y}) - \frac{1}{2}(\boldsymbol{f} - \boldsymbol{m})^\top \boldsymbol{K}^{-1}(\boldsymbol{f} - \boldsymbol{m})\end{aligned}$$

that is *quadratic* in $\boldsymbol{f}$. The first Newton iterate $\boldsymbol{f}_1$ therefore coincides with log-posterior's maximizer $\boldsymbol{f}_1 = \boldsymbol{f}_\star$. The outer loop of IterNCGP thus reduces to a single iteration.

How does this step look from the perspective of the IterNCGP algorithm? First note that $\boldsymbol{W}(\boldsymbol{f}) = -\nabla^2 \log p(\boldsymbol{y} \mid \boldsymbol{f}) \equiv \boldsymbol{\Lambda}^{-1}$. Given an initial $\boldsymbol{f}_0$, IterNCGP computes the observation noise $\boldsymbol{W}^{-1}(\boldsymbol{f}_0) = \boldsymbol{\Lambda}$ and pseudo regression targets $\hat{\boldsymbol{y}}(\boldsymbol{f}_0) = \boldsymbol{f}_0 + \boldsymbol{W}(\boldsymbol{f}_0)^{-1} \nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}_0) = \boldsymbol{f}_0 - \boldsymbol{\Lambda}\boldsymbol{\Lambda}^{-1}(\boldsymbol{f}_0 - \boldsymbol{y}) = \boldsymbol{y}$. Both these quantities are *independent* of the initialization $\boldsymbol{f}_0$. Thus, the first (and only) regression problem that IterNCGP forms in the outer loop is the *original* regression problem (defined by labels $\boldsymbol{y}$ and the observation noise Λ) and IterGP is applied to solve that regression problem. This shows that our framework recovers IterGP in the case of a Gaussian likelihood and our algorithm can thus be regarded as an extension thereof. □

A.1.3 The Marginal Uncertainty Decreases in the Inner Loop

We claim in Section 5.3.1 that the marginal uncertainty captured by $K_{i,j}(\boldsymbol{x}, \boldsymbol{x}) \in \mathbb{R}^{C \times C}$ (see Equation (5.9)) within a Newton step

decreases with each solver iteration j. Here, we provide the proof for this statement.

Proposition A.3 (The Uncertainty Decreases in the Inner Loop) For each i it holds (element-wise) that $\operatorname{diag}(K_{i,j}(\boldsymbol{x},\boldsymbol{x})) \geq \operatorname{diag}(K_{i,k}(\boldsymbol{x},\boldsymbol{x}))$ for any $k \geq j$ and arbitrary $\boldsymbol{x}$.

Proof. To see this, we rewrite $\boldsymbol{C}_j$ as a sum of j rank-1 matrices $\boldsymbol{C}_j = \sum_{\ell=1}^{j} \boldsymbol{d}_\ell \boldsymbol{d}_\ell^\top$ and substitute this into Equation (5.9). It holds that

$$
\begin{aligned}
&\operatorname{diag}(K_{i,j}(\boldsymbol{x},\boldsymbol{x})) \\
&= \operatorname{diag}(K(\boldsymbol{x},\boldsymbol{x})) - \sum_{\ell=1}^{j} \operatorname{diag}(\underbrace{K(\boldsymbol{x},\boldsymbol{X})\boldsymbol{d}_\ell}_{=:\hat{\boldsymbol{d}}_\ell} \underbrace{\boldsymbol{d}_\ell^\top K(\boldsymbol{X},\boldsymbol{x})}_{=\hat{\boldsymbol{d}}_\ell^\top}) \\
&= \operatorname{diag}(K(\boldsymbol{x},\boldsymbol{x})) - \sum_{\ell=1}^{j} \hat{\boldsymbol{d}}_\ell^2 \quad \text{(The square is applied element-wise)} \\
&\geq \operatorname{diag}(K(\boldsymbol{x},\boldsymbol{x})) - \sum_{\ell=1}^{k} \hat{\boldsymbol{d}}_\ell^2 \quad \text{for } k \geq j \\
&= \operatorname{diag}(K_{i,k}(\boldsymbol{x},\boldsymbol{x}))
\end{aligned}
$$

for any $\boldsymbol{x} \in \mathbb{X}$. □

A.1.4 Virtual Solver Run

In Section 5.3.3, we showed that it is possible to *imitate* a solver run using the *previous* actions on the *new* problem, without ever having to multiply by $\boldsymbol{K}$. The pseudocode is given in Algorithm 5.3. Here, we discuss the numerical and probabilistic perspective on that procedure in more detail and provide derivations for the statements in the main text.

Numerical Perspective. Let $\boldsymbol{S} = (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_B)$ the matrix of stacked linearly independent actions. We use $\boldsymbol{C}_0 = \boldsymbol{S}(\boldsymbol{S}^\top \hat{\boldsymbol{K}} \boldsymbol{S})^{-1} \boldsymbol{S}^\top$ (see Equation (5.10)) as an initial estimate of the precision matrix in Algorithm 5.2. The corresponding initial residual (see Algorithm 5.2) $\boldsymbol{r}_0 = (\hat{\boldsymbol{y}} - \boldsymbol{m}) - \hat{\boldsymbol{K}} \boldsymbol{v}_0$ for the first iterate $\boldsymbol{v}_0 = \boldsymbol{C}_0(\hat{\boldsymbol{y}} - \boldsymbol{m})$ can be decomposed into $\boldsymbol{P}_S \boldsymbol{r}_0$ and $(\boldsymbol{I} - \boldsymbol{P}_S)\boldsymbol{r}_0$. $\boldsymbol{P}_S = \boldsymbol{S}(\boldsymbol{S}^\top \boldsymbol{S})^{-1} \boldsymbol{S}^\top$ is the orthogonal projection onto the subspace $\operatorname{span}\{\boldsymbol{S}\}$ spanned by the actions.

Proposition A.4 (Residual in $\operatorname{span}\{\boldsymbol{S}\}$ Is Zero) The orthogonal projection $\boldsymbol{P}_S \boldsymbol{r}_0$ of the initial residual $\boldsymbol{r}_0$ onto $\operatorname{span}\{\boldsymbol{S}\}$ is zero.

Proof. It holds that

$$\begin{aligned}
\boldsymbol{P}_S \boldsymbol{r}_0 &= \boldsymbol{P}_S(\hat{\boldsymbol{y}} - \boldsymbol{m}) - \boldsymbol{P}_S \hat{\boldsymbol{K}} \boldsymbol{v}_0 \\
&= \boldsymbol{P}_S(\hat{\boldsymbol{y}} - \boldsymbol{m}) - \boldsymbol{P}_S \hat{\boldsymbol{K}} \boldsymbol{C}_0 (\hat{\boldsymbol{y}} - \boldsymbol{m}) \\
&= \boldsymbol{P}_S(\hat{\boldsymbol{y}} - \boldsymbol{m}) - \underbrace{\boldsymbol{S}(\boldsymbol{S}^\top \boldsymbol{S})^{-1}(\boldsymbol{S}^\top}_{=\boldsymbol{P}_S} \hat{\boldsymbol{K}} \underbrace{\boldsymbol{S})(\boldsymbol{S}^\top \hat{\boldsymbol{K}} \boldsymbol{S})^{-1} \boldsymbol{S}^\top}_{=\boldsymbol{C}_0} (\hat{\boldsymbol{y}} - \boldsymbol{m}) \\
&= \boldsymbol{P}_S(\hat{\boldsymbol{y}} - \boldsymbol{m}) - \underbrace{\boldsymbol{S}(\boldsymbol{S}^\top \boldsymbol{S})^{-1} \boldsymbol{S}^\top}_{=\boldsymbol{P}_S} (\hat{\boldsymbol{y}} - \boldsymbol{m}) \\
&= 0.
\end{aligned}$$

□

Proposition A.4 shows that the residual in $\text{span}\{\boldsymbol{S}\}$ is zero. In that sense, the solution within this subspace is already perfectly identified at initialization. The remaining residual thus lies in the orthogonal complement of $\text{span}\{\boldsymbol{S}\}$ which can be targeted through additional actions. If we measure the error in the representer weights $\boldsymbol{v} - \boldsymbol{v}_0$, a similar result holds, as we show in the following.

Proposition A.5 (Error in Representer Weights in $\text{span}\{\boldsymbol{S}\}$ Is Zero) The $\hat{\boldsymbol{K}}$-orthogonal projection of the representer weights approximation error $\hat{\boldsymbol{P}}_S(\boldsymbol{v} - \boldsymbol{v}_0)$ onto $\text{span}\{\boldsymbol{S}\}$ is zero.

Proof. The $\hat{\boldsymbol{K}}$-orthogonal (orthogonal with respect to the inner product $\langle \cdot, \cdot \rangle_{\hat{\boldsymbol{K}}}$) projection onto the subspace $\text{span}\{\boldsymbol{S}\}$ spanned by the actions is given by $\hat{\boldsymbol{P}}_S = \boldsymbol{C}_0 \hat{\boldsymbol{K}}$ [143, Section S2.1]. It holds that

$$\begin{aligned}
\hat{\boldsymbol{P}}_S(\boldsymbol{v} - \boldsymbol{v}_0) &= \boldsymbol{C}_0 \hat{\boldsymbol{K}}(\boldsymbol{v} - \boldsymbol{v}_0) \\
&= \boldsymbol{C}_0 \hat{\boldsymbol{K}} \hat{\boldsymbol{K}}^{-1}(\hat{\boldsymbol{y}} - \boldsymbol{m}) - \boldsymbol{C}_0 \hat{\boldsymbol{K}} \boldsymbol{C}_0 \underbrace{(\hat{\boldsymbol{y}} - \boldsymbol{m})}_{=\hat{\boldsymbol{K}}\boldsymbol{v}} \\
&= \boldsymbol{C}_0(\hat{\boldsymbol{y}} - \boldsymbol{m}) - \underbrace{\boldsymbol{C}_0 \hat{\boldsymbol{K}}}_{=\hat{\boldsymbol{P}}_S} \underbrace{\boldsymbol{C}_0 \hat{\boldsymbol{K}}}_{=\hat{\boldsymbol{P}}_S} \boldsymbol{v} \\
&= \boldsymbol{C}_0(\hat{\boldsymbol{y}} - \boldsymbol{m}) - \boldsymbol{C}_0 \underbrace{\hat{\boldsymbol{K}} \boldsymbol{v}}_{=\hat{\boldsymbol{y}} - \boldsymbol{m}} \\
&= 0,
\end{aligned}$$

where we used that $\boldsymbol{v} = \hat{\boldsymbol{K}}^{-1}(\hat{\boldsymbol{y}} - \boldsymbol{m})$ is the solution of the GP regression linear system, $\boldsymbol{v}_0 = \boldsymbol{C}_0(\hat{\boldsymbol{y}} - \boldsymbol{m})$ and the idempotence of the projection matrix $\hat{\boldsymbol{P}}_S = \hat{\boldsymbol{P}}_S \hat{\boldsymbol{P}}_S$. □

Probabilistic Perspective. Equation (5.9) describes the effect of $\boldsymbol{C}_0$ from a probabilistic perspective. Initializing $\boldsymbol{C}_0 = \boldsymbol{0}$ in step i results in $m_{i,0} = m(\cdot)$ (prior mean) and $K_{i,0} = K(\cdot, \cdot)$ (prior covariance) since the *reduction* of uncertainty $K(\cdot, \boldsymbol{X})\, \boldsymbol{C}_0\, K(\boldsymbol{X}, \cdot)$ is zero. This case, where no information from past steps is included, is illustrated in the first column $R = 0$ in Figure 5.4.

Special Case. We consider a special case, where the general intricate form of the total marginal variance from Section 5.3.4

$$\mathrm{Tr}(K_{i,0}(\boldsymbol{X},\boldsymbol{X})) = \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{K}\boldsymbol{C}_0\boldsymbol{K}) \tag{A.1}$$

collapses. Let $\lambda_1,\dots,\lambda_{NC} > 0$ denote the eigenvalues of $\hat{\boldsymbol{K}}$ and $\boldsymbol{b}_1,\dots,\boldsymbol{b}_{NC}$ the corresponding (pairwise orthogonal) eigenvectors. We make the following two assumptions: **(A1):** We assume $\boldsymbol{W}^{-1} = \boldsymbol{0}$, *i.e.* $\hat{\boldsymbol{K}} = \boldsymbol{K}$. **(A2):** We assume that the actions coincide with a subset $\mathbb{L} \subseteq \{1,\dots,NC\}$ of $\hat{\boldsymbol{K}}$'s eigenvectors $\boldsymbol{S} = (\boldsymbol{b}_l)_{l\in\mathbb{L}} \in \mathbb{R}^{NC\times|\mathbb{L}|}$.

Proposition A.6 (Total Marginal Uncertainty) Under assumptions **(A1)** and **(A2)** it holds that

$$\mathrm{Tr}(K_{i,0}(\boldsymbol{X},\boldsymbol{X})) = \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{M}).$$

Proof. Let $\boldsymbol{S} = (\boldsymbol{b}_l)_{l\in\mathbb{L}} \in \mathbb{R}^{NC\times|\mathbb{L}|}$ and $\boldsymbol{\Lambda} = \mathrm{diag}((\lambda_l)_{l\in\mathbb{L}}) \in \mathbb{R}^{|\mathbb{L}|\times|\mathbb{L}|}$ contain a subset $\mathbb{L} \subseteq \{1,\dots,NC\}$ of $\hat{\boldsymbol{K}}$'s eigenpairs. The remaining eigenvectors and eigenvalues are given by $\boldsymbol{S}_+ = (\boldsymbol{b}_l)_{l\notin\mathbb{L}} \in \mathbb{R}^{NC\times NC-|\mathbb{L}|}$ and $\boldsymbol{\Lambda}_+ = \mathrm{diag}((\lambda_l)_{l\notin\mathbb{L}}) \in \mathbb{R}^{NC-|\mathbb{L}|\times NC-|\mathbb{L}|}$. First note that we can write the eigendecomposition of $\hat{\boldsymbol{K}} = \boldsymbol{K}$ as a sum of two components $\hat{\boldsymbol{K}} = \boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{S}^\top + \boldsymbol{S}_+\boldsymbol{\Lambda}_+\boldsymbol{S}_+^\top$, each of which covers one part of the spectrum. It holds

$$\boldsymbol{S}^\top\boldsymbol{S} = \boldsymbol{I}, \quad \boldsymbol{S}_+^\top\boldsymbol{S}_+ = \boldsymbol{I}, \quad \boldsymbol{S}^\top\boldsymbol{S}_+ = \boldsymbol{0} \quad \text{and} \quad \boldsymbol{S}_+^\top\boldsymbol{S} = \boldsymbol{0}$$

since $\hat{\boldsymbol{K}}$ is symmetric and its eigenvectors are thus pairwise orthogonal. It follows

$$\begin{aligned}
\boldsymbol{K}\boldsymbol{S} &= (\boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{S}^\top + \boldsymbol{S}_+\boldsymbol{\Lambda}_+\boldsymbol{S}_+^\top)\boldsymbol{S} = \boldsymbol{S}\boldsymbol{\Lambda} \\
\boldsymbol{S}^\top\boldsymbol{K} &= \boldsymbol{S}^\top(\boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{S}^\top + \boldsymbol{S}_+\boldsymbol{\Lambda}_+\boldsymbol{S}_+^\top) = \boldsymbol{\Lambda}\boldsymbol{S}^\top \\
\boldsymbol{M} = \boldsymbol{S}^\top\boldsymbol{K}\boldsymbol{S} &= \boldsymbol{S}^\top(\boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{S}^\top + \boldsymbol{S}_+\boldsymbol{\Lambda}_+\boldsymbol{S}_+^\top)\boldsymbol{S} = \boldsymbol{\Lambda}.
\end{aligned}$$

Plugging those expressions into Equation (A.1) yields

$$\begin{aligned}
\mathrm{Tr}(K_{i,0}(\boldsymbol{X},\boldsymbol{X})) &= \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{K}\boldsymbol{C}_0\boldsymbol{K}) \\
&= \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{K}\boldsymbol{S}\underbrace{(\boldsymbol{S}^\top\hat{\boldsymbol{K}}\boldsymbol{S})^{-1}}_{=\boldsymbol{M}^{-1}}\boldsymbol{S}^\top\boldsymbol{K}) \\
&= \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{\Lambda}^{-1}\boldsymbol{\Lambda}\boldsymbol{S}^\top) \\
&= \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{S}^\top\boldsymbol{S}\boldsymbol{\Lambda}) \\
&= \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{\Lambda}) \\
&= \mathrm{Tr}(\boldsymbol{K}) - \mathrm{Tr}(\boldsymbol{M}) \\
&= \sum_{l\notin\mathbb{L}} \lambda_l.
\end{aligned}$$

The last equation is due to

$$\begin{aligned}\operatorname{Tr}(\boldsymbol{K}) &= \operatorname{Tr}(\boldsymbol{S}\boldsymbol{\Lambda}\boldsymbol{S}^\top) + \operatorname{Tr}(\boldsymbol{S}_+\boldsymbol{\Lambda}_+\boldsymbol{S}_+^\top)\\ &= \operatorname{Tr}(\boldsymbol{S}^\top\boldsymbol{S}\boldsymbol{\Lambda}) + \operatorname{Tr}(\boldsymbol{S}_+^\top\boldsymbol{S}_+\boldsymbol{\Lambda}_+)\\ &= \operatorname{Tr}(\boldsymbol{\Lambda}) + \operatorname{Tr}(\boldsymbol{\Lambda}_+).\end{aligned}$$ □

Proposition A.6 shows that the *reduction* of the marginal uncertainty is determined by the sum of $\boldsymbol{M}$'s eigenvalues $\sum_{l\in\mathbb{L}} \lambda_l$. If $\boldsymbol{S}$ contains the eigenvectors $\boldsymbol{b}_l$ to the largest eigenvalues (*i.e.* $\boldsymbol{S}$ is "aligned" with the high-variance subspace of $\hat{\boldsymbol{K}}$), the remaining uncertainty $\sum_{l\notin\mathbb{L}} \lambda_l$ is small. In contrast, if $\boldsymbol{S}$ covers the low-variance subspace of $\hat{\boldsymbol{K}}$, the uncertainty remains largely unaffected.

A.1.5 Derivatives of the Poisson Log-Likelihood

One of our main experiments in Section 5.5 is Poisson regression (see Appendix A.3.2 for details). In order to apply IterNCGP, we have to formulate the problem within the NCGP framework. In particular, we have to specify the derivatives of the log-likelihood.

The Poisson likelihood is given by

$$p(\boldsymbol{y} \mid \boldsymbol{f}) = \prod_{n=1}^{N} \frac{\lambda_n^{y_n} \exp(-\lambda_n)}{y_n!},$$

where $y_n \in \mathbb{N}_0$ and $\boldsymbol{\lambda} := \lambda(\boldsymbol{X}) = \exp(f(\boldsymbol{X})) = \exp(\boldsymbol{f})$. Taking the logarithm yields

$$\begin{aligned}\log p(\boldsymbol{y} \mid \boldsymbol{f}) &= \sum_{n=1}^{N} \log\left(\frac{\lambda_n^{y_n} \exp(-\lambda_n)}{y_n!}\right)\\ &= \sum_{n=1}^{N} \left(y_n \log(\lambda_n) - \lambda_n - \log(y_n!)\right).\end{aligned}$$

The log-likelihood's gradient and Hessian with respect to $\boldsymbol{f}$ are therefore given by

$$\begin{aligned}\nabla \log p(\boldsymbol{y} \mid \boldsymbol{f}) &= \boldsymbol{y} - \exp(\boldsymbol{f}) \quad \text{and}\\ \nabla^2 \log p(\boldsymbol{y} \mid \boldsymbol{f}) &= -\operatorname{diag}(\exp(\boldsymbol{f})),\end{aligned}$$

where the exponential function is applied element-wise. This implies that the log likelihood is concave which was one of the prerequisites of our algorithm (see Section 5.2.1). It follows that $\boldsymbol{W}(\boldsymbol{f})^{-1} = \operatorname{diag}(\exp(-\boldsymbol{f}))$.

A.1.6 Pseudo-Inverse of Negative Hessian of the Log-Likelihood for Multi-Class Classification

For multi-class classification (see Appendix A.3.3 for details), we need access to the pseudo-inverse $\boldsymbol{W}^\dagger$. For this to be efficient, we derive an explicit form of $\boldsymbol{W}^\dagger$ in the following and show that matrix-vector multiplies can be implemented efficiently in $\mathcal{O}(NC)$. Since the ordering (see Appendix A.2.1) of $\boldsymbol{W}$ plays an important role in the derivation, we use an explicit notation in this section.

Lemma A.7 (Explicit Pseudo-Inverse for Multi-Class Classification) Consider multi-class classification, such that the log-likelihood $\log p(\boldsymbol{y} \mid \boldsymbol{f})$ is given by a categorical likelihood with a softmax inverse link function, then the pseudo-inverse $[\boldsymbol{W}(\boldsymbol{f})]^\dagger_{CN} \in \mathbb{R}^{NC \times NC}$ of $\boldsymbol{W}(\boldsymbol{f})$ in CN-ordering is given by

$$[\boldsymbol{W}(\boldsymbol{f})]^\dagger_{CN} = \operatorname{blockdiag}_{c=1,\dots,C}\left((\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_c^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\right)$$

where $\boldsymbol{\pi}_n = (\pi_n^1, \dots, \pi_n^C)^\top \in \mathbb{R}^C$ denotes the output of the softmax for $\boldsymbol{x}_n$, *i.e.* $\pi_n^c \coloneqq \exp(f_n^c)/\sum_{c'} \exp(f_n^{c'})$. The cost of one matrix-vector multiplication $\boldsymbol{v} \mapsto [\boldsymbol{W}(\boldsymbol{f})]^\dagger_{CN}\boldsymbol{v}$ with the pseudo-inverse is $\mathcal{O}(NC)$.

Proof. By Eq. (3.38) in Rasmussen and Williams [114], the matrix $\boldsymbol{W}(\boldsymbol{f})$ in NC-ordering is given by

$$[\boldsymbol{W}(\boldsymbol{f})]_{NC} = [\operatorname{diag}(\boldsymbol{\pi})]_{NC} - \boldsymbol{\Pi}\boldsymbol{\Pi}^\top,$$

where $[\operatorname{diag}(\boldsymbol{\pi})]_{NC} = \operatorname{diag}(\pi_1^1, \dots, \pi_N^1, \dots, \pi_1^C, \dots \pi_N^C)$ and

$$\boldsymbol{\Pi} = \begin{pmatrix} \operatorname{diag}(\pi_1^1, \dots, \pi_N^1) \\ \vdots \\ \operatorname{diag}(\pi_1^C, \dots, \pi_N^C) \end{pmatrix} \in \mathbb{R}^{NC \times N}.$$

Rewriting $\boldsymbol{W}(\boldsymbol{f})$ in the CN-ordering, we obtain using $[\operatorname{diag}(\boldsymbol{\pi})]_{CN} = \operatorname{diag}(\pi_1^1, \dots, \pi_1^C, \dots, \pi_N^1, \dots, \pi_N^C)$ that

$$\begin{aligned}[\boldsymbol{W}(\boldsymbol{f})]_{CN} &= [\operatorname{diag}(\boldsymbol{\pi})]_{CN} - \begin{pmatrix} \boldsymbol{\pi}_1 & & \\ & \ddots & \\ & & \boldsymbol{\pi}_N \end{pmatrix}\begin{pmatrix} \boldsymbol{\pi}_1^\top & & \\ & \ddots & \\ & & \boldsymbol{\pi}_N^\top \end{pmatrix} \\ &= \operatorname{blockdiag}(\operatorname{diag}(\boldsymbol{\pi}_n) - \boldsymbol{\pi}_n\boldsymbol{\pi}_n^\top).\end{aligned}$$

Now the pseudo-inverse of a block-diagonal matrix is the block-diagonal of the block pseudo-inverses, *i.e.* $\operatorname{blockdiag}(\boldsymbol{A}_n)^\dagger = \operatorname{blockdiag}(\boldsymbol{A}_n^\dagger)$ which can be shown by simply checking the definition criteria of the pseudo-inverse and using basic properties

of block matrices. Therefore, it suffices to show that the block pseudo-inverses are given by

$$(\operatorname{diag}(\boldsymbol{\pi}_n) - \boldsymbol{\pi}_n\boldsymbol{\pi}_n^\top)^\dagger = (\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)$$

for $n \in \{1, \ldots, N\}$. We do so by checking the definition criteria of a pseudo-inverse. Let $\boldsymbol{A}_n = \operatorname{diag}(\boldsymbol{\pi}_n) - \boldsymbol{\pi}_n\boldsymbol{\pi}_n^\top$. We begin by showing the following intermediate result:

$$\begin{aligned}\boldsymbol{A}_n(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top) &= \boldsymbol{A}_n - \frac{1}{C}(\operatorname{diag}(\boldsymbol{\pi}_n) - \boldsymbol{\pi}_n\boldsymbol{\pi}_n^\top)\mathbf{1}\mathbf{1}^\top \\ &= \boldsymbol{A}_n - \frac{1}{C}(\boldsymbol{\pi}_n - \boldsymbol{\pi}_n(\boldsymbol{\pi}_n^\top\mathbf{1}))\mathbf{1}^\top \\ &= \boldsymbol{A}_n.\end{aligned} \tag{A.2}$$

Now let's verify the first criterion in the definition of the pseudo-inverse. We have

$$\begin{aligned}&\boldsymbol{A}_n(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\boldsymbol{A}_n \\ &\quad= \boldsymbol{A}_n\operatorname{diag}(\boldsymbol{\pi}_n^{-1})\boldsymbol{A}_n \\ &\quad= \boldsymbol{A}_n\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\operatorname{diag}(\boldsymbol{\pi}_n) - \boldsymbol{\pi}_n\boldsymbol{\pi}_n^\top) \\ &\quad= \boldsymbol{A}_n(\boldsymbol{I} - \mathbf{1}\boldsymbol{\pi}_n^\top) \\ &\quad= \boldsymbol{A}_n - (\operatorname{diag}(\boldsymbol{\pi}_n) - \boldsymbol{\pi}_n\boldsymbol{\pi}_n^\top)\mathbf{1}\boldsymbol{\pi}_n^\top \\ &\quad= \boldsymbol{A}_n,\end{aligned}$$

where we used (A.2). Next, we'll verify the second criterion.

$$\begin{aligned}&(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\boldsymbol{A}_n(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top) \\ &\quad= (\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})\boldsymbol{A}_n\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top) \\ &\quad= (\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\end{aligned}$$

where we used

$$\operatorname{diag}(\boldsymbol{\pi}_n^{-1})\boldsymbol{A}_n = \boldsymbol{I} = \boldsymbol{A}_n\operatorname{diag}(\boldsymbol{\pi}_n^{-1}) \tag{A.3}$$

as shown above. Finally, we verify the symmetry of the product of $\boldsymbol{A}_n$ and its pseudo-inverse. Observe that both $\boldsymbol{A}_n$ and $(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)$ are symmetric. Therefore, we have

$$\begin{aligned}&(\boldsymbol{A}_n(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top))^* \\ &\quad= (\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\boldsymbol{A}_n\end{aligned}$$

and

$$((\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\boldsymbol{A}_n)^*$$

$$= \boldsymbol{A}_n(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top).$$

Thus, if we can show that $\boldsymbol{A}_n$ and $(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)$ commute we have shown the remaining symmetry criteria of the pseudo-inverse. It holds that

$$\begin{aligned}\boldsymbol{A}_n(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top) &\overset{(A.2)}{=} \boldsymbol{A}_n \operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top) \\ &\overset{(A.3)}{=} (\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\end{aligned}$$

as well as

$$\begin{aligned}(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\boldsymbol{A}_n &\overset{(A.2)}{=} (\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})\boldsymbol{A}_n \\ &\overset{(A.3)}{=} (\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\end{aligned}$$

This completes the proof for the form of the pseudo-inverse. For the complexity of multiplication, note that multiplying with $(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)\operatorname{diag}(\boldsymbol{\pi}_n^{-1})(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)$ has cost $\mathcal{O}(C)$, since it decomposes into two multiplications with $(\boldsymbol{I} - \frac{1}{C}\mathbf{1}\mathbf{1}^\top)$ which is linear and one element-wise scaling. Therefore, the cost of multiplication with the pseudo-inverse consisting of N blocks has complexity $\mathcal{O}(NC)$. □

A.2 Implementation Details

A.2.1 Ordering Within Vectors & Matrices

Ordering Within Vectors. By default, we assume all vectors and matrices to be represented in CN-ordering. For example, the mean vector was introduced as the aggregated outputs of the mean function $m\colon \mathbb{X} \to \mathbb{R}^C$ for all data points $\boldsymbol{m} = m(\boldsymbol{X}) = (m(\boldsymbol{x}_1)^\top, \ldots, m(\boldsymbol{x}_N)^\top)^\top$. With $m(\boldsymbol{x}_n)^\top = (m_n^1, \ldots, m_n^C)$ denoting the C outputs for data point $\boldsymbol{x}_n$, we can write $\boldsymbol{m}$ as $\boldsymbol{m} = (m_1^1, m_1^2, \ldots, m_1^C, \ldots, m_N^1, m_N^2, \ldots, m_N^C)$. We call that representation CN-ordering because the superscript c moves *first* and the subscript n moves *second*. Consecutively, $(m_1^1, m_2^1, \ldots, m_N^1, \ldots, m_1^C, m_2^C, \ldots, m_N^C)$ corresponds to NC-ordering.

Ordering Within Matrices. The same terminology can be applied to matrices, where the rows and columns can be represented in CN or NC-ordering. Depending on the context, different representations are beneficial. For example, in CN-ordering, $\boldsymbol{W}$ is block-diagonal (due to our iid assumption, see Section 5.2.1) with N blocks of size $C \times C$ on the diagonal. In contrast, when the C outputs of the hidden function are assumed to be independent of each other, $\boldsymbol{K}$ is block diagonal only in NC-ordering. So, based on the chosen ordering, different structures arise that we can exploit

in subsequent computations (*e.g.* when we compute the inverse of $\boldsymbol{W}$, see Appendix A.2.3).

A.2.2 Stopping Criteria in Algorithms 5.1 and 5.2

Stopping Criterion in Algorithm 5.1. The OuterStoppingCriterion() we use for our experiments is based on the *relative change* of the vector $\boldsymbol{g}_i = \boldsymbol{f}_i - \boldsymbol{m}$. When $\|\boldsymbol{g}_i - \boldsymbol{g}_{i-1}\| \, \|\boldsymbol{g}_i\|^{-1} \leq \delta$ falls below the convergence tolerance δ (by default, $\delta = 1\%$), the loop over i terminates. Of course, other convergence criteria are also conceivable. Depending on the application one might want to customize the criterion and, for example, include the marginal uncertainty at the training data.

Stopping Criterion in Algorithm 5.2. We use the same InnerStoppingCriterion() as in [143, Section S3.2]: The loop over j terminates if the norm of the residual $\|\boldsymbol{r}_j\| < \max\{\delta_{\text{abs}}, \delta_{\text{rel}}\|\hat{\boldsymbol{y}} - \boldsymbol{m}\|\}$ is below an absolute tolerance δ_{abs} or below the scaled norm of the right-hand side $\hat{\boldsymbol{y}} - \boldsymbol{m}$ of the linear system. By default, both tolerances are set to 10^{-5}. Additionally, we typically specify a maximum number of iterations. The solver is also terminated when the normalization constant $\eta_j \leq 0$. This can happen due to numerical imprecision if the linear system is badly conditioned, *e.g.* if some eigenvalues of the linear system are close to zero.

A.2.3 Cost Analysis of IterNCGP

In this section, we investigate the computational costs of IterNCGP in more detail. We start with a discussion of the computational costs for matrix-vector products with $\boldsymbol{K}$, $\boldsymbol{W}^{-1}$ and $\boldsymbol{C}_j$ and then analyze the run time and memory costs of the individual algorithms (Algorithms 5.1 to 5.3).

Matrix-Vector Products

IterGP is an *iterative matrix-free* algorithm and our algorithm IterNCGP inherits that property: The matrices $\boldsymbol{K}$, $\boldsymbol{W}^{-1}$ and $\boldsymbol{C}_j$ are evaluated lazily, *i.e.* matrix-vector products are evaluated *without* forming the matrices in memory explicitly. This enables our algorithm to scale to problems where a naive approach causes memory overflows. In Algorithms 5.1 to 5.3, the memory and run time cost for matrix-vector products with $\boldsymbol{K}$ are denoted by $\mu_{\boldsymbol{K}}$ and $t_{\boldsymbol{K}}$ and by $\mu_{\boldsymbol{W}^{-1}}$ and $t_{\boldsymbol{W}^{-1}}$ for products with $\boldsymbol{W}^{-1}$.

Products with $\boldsymbol{K}$. Matrix-vector products with $\boldsymbol{K}$ can be decomposed into products with its sub-matrices. The associated memory

costs $\mathcal{O}(\mu_K)$ can thereby be reduced basically arbitrarily and the run time can be improved by using specialized software libraries such as KeOps [16] and parallel hardware (*i.e.* GPUs). Still, products with $\boldsymbol{K}$ are computationally relatively expensive, since this operation is typically *quadratic* in the number of training data points N.

Products with $\boldsymbol{W}^{-1}$ (General Case). Under the assumptions on the likelihood from Section 5.2.1, $\boldsymbol{W}$ is block-diagonal with N blocks of size $C \times C$ (in CN-ordering, see Appendix A.2.1). Here, we denote these blocks by $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_N$. It can be easily verified that $\boldsymbol{W}^{-1}$ is also a block-diagonal matrix and the blocks on its diagonal are the inverses of $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_N$.

Consider the matrix-vector product $\boldsymbol{v} \mapsto \boldsymbol{W}^{-1}\boldsymbol{v} =: \boldsymbol{w} \in \mathbb{R}^{NC}$. In the vectors $\boldsymbol{v}$ and $\boldsymbol{w}$, we repeatedly group C consecutive entries which results in segments $\boldsymbol{w}_n, \boldsymbol{v}_n \in \mathbb{R}^C$ for $n = 1, \ldots, N$, *i.e.*

$$\underbrace{\begin{pmatrix} \boldsymbol{W}_1^{-1} & & \\ & \ddots & \\ & & \boldsymbol{W}_N^{-1} \end{pmatrix}}_{=\boldsymbol{W}^{-1}} \cdot \underbrace{\begin{pmatrix} \boldsymbol{v}_1 \\ \vdots \\ \boldsymbol{v}_N \end{pmatrix}}_{=\boldsymbol{v}} = \underbrace{\begin{pmatrix} \boldsymbol{w}_1 \\ \vdots \\ \boldsymbol{w}_N \end{pmatrix}}_{=\boldsymbol{w}}.$$

It holds that $\boldsymbol{w}_n = \boldsymbol{W}_n^{-1}\boldsymbol{v}_n$, *i.e.* each segment in $\boldsymbol{w}$ is the product of a single $C \times C$ block from $\boldsymbol{W}^{-1}$ with one segment from $\boldsymbol{v}$. Computing $\boldsymbol{w}_n$ thus amounts to solving a linear system of size C with cost $\mathcal{O}(C^3)$. The total cost for all N segments is thus $\mathcal{O}(NC^3)$. However, the N linear systems are independent of each other and can thus be solved in parallel. So, if appropriate computational resources are available, the total run time complexity can be reduced to $\mathcal{O}(C^3)$.

In general, $\boldsymbol{W}^{-1}$ requires $\mathcal{O}(NC^2)$ in terms of memory consumption. If needed, these costs can be reduced further to $\mathcal{O}(C^2)$ because (as explained above), products with $\boldsymbol{W}^{-1}$ can be broken down into products with the individual blocks of $\boldsymbol{W}^{-1}$. We can perform those products sequentially such that only a single block is present in memory at a time.

Products with $\boldsymbol{W}^{-1}$ (Special Cases). In many cases, we can multiply with $\boldsymbol{W}^{-1}$ more efficiently. In the multi-class classification case, the run time and memory costs for multiplication with the pseudo-inverse $\boldsymbol{W}^{\dagger}$ can be reduced to $\mathcal{O}(NC)$ (see Appendix A.1.6). In the regression case ($C = 1$), $\boldsymbol{W}^{-1}$ is a diagonal matrix of size $N \times N$. The memory and run time costs are thus in $\mathcal{O}(N)$. An example is Poisson regression, for which we derive the explicit form of $\boldsymbol{W}^{-1}$ in Appendix A.1.5.

Products with $\boldsymbol{C}_j$. $\boldsymbol{C}_j = \boldsymbol{Q}_j\boldsymbol{Q}_j^\top$ is represented via its matrix root $\boldsymbol{Q}_j \in \mathbb{R}^{NC \times B}$. This allows for efficient storage and matrix-vector

multiplies $\boldsymbol{v} \mapsto \boldsymbol{C}_j \boldsymbol{v} = \boldsymbol{Q}_j(\boldsymbol{Q}_j^\top \boldsymbol{v})$ in $\mathcal{O}(BNC)$.

Cost Analysis Algorithms 5.1 to 5.3

The run time and memory complexity for the operations in Algorithms 5.1 to 5.3 is given directly in the pseudocode. Here, we provide some additional information for the costs that depend on the user's choices and put the costs of the individual algorithms into perspective.

Algorithm 5.2 (IterGP). The run time cost for selecting an action $\mathcal{O}(t_{\text{POLICY}})$ depends on the underlying policy. For Cholesky actions ($\boldsymbol{s}_j = \boldsymbol{e}_j$) or CG ($\boldsymbol{s}_j = \boldsymbol{r}_{j-1}$), the run time cost is insignificant since no additional computations are required at all.

One iteration's total computational cost (without prediction) is dominated by two matrix-vector products with $\boldsymbol{K}$ in terms of run time and $\mathcal{O}(BNC)$ in terms of storage requirements (for the buffers $\boldsymbol{S}$ and $\boldsymbol{T}$ as well as the matrix root $\boldsymbol{Q}_j$). The *initial* size (*i.e.* the number of columns) of $\boldsymbol{S}$, $\boldsymbol{T}$ and $\boldsymbol{Q}_j$ is given by the rank limit R used in Algorithm 5.3. Henceforth, one column is added to each of the buffers and matrix root in *each* solver iteration, increasing their size to $B = R + j$ in iteration j. It is thus reasonable to include an upper bound on the iteration number in the stopping criterion of Algorithm 5.2.

Algorithm 5.3 (Virtual Solver Run with Optional Compression). The total run time complexity of Algorithm 5.3 is $\mathcal{O}(Bt_{\boldsymbol{W}^{-1}} + B^2NC)$, *i.e.* dominated by matrix-matrix products involving the buffers and $\boldsymbol{W}^{-1}$. In terms of memory requirements, the buffers $\boldsymbol{S}$, $\boldsymbol{T}$, and $\boldsymbol{Q}_0$ are the decisive contributors with $\mathcal{O}(RNC)$. The truncation of the eigendecomposition provides a way to control that bound by resetting the current buffer size B to an arbitrary number $R \leq B$. In comparison to Algorithm 5.2, the computational costs are practically of minor importance since no multiplications with $\boldsymbol{K}$ are necessary.

Algorithm 5.1 (IterNCGP Outer Loop). The costs $\mathcal{O}(t_m)$ for evaluating m on the training data depends on the choice of mean function. For a constant mean function, no computations are necessary, so run time costs are negligible. This can be different *e.g.* for applications in Bayesian deep learning, where evaluating m requires forward passes through a neural network.

A.3 Experimental Details

Throughout the paper, we use binary classification as an illustrative and supporting example (Figures 5.1 to 5.4). The two main experiments follow in Section 5.5: Poisson regression (Section 5.5.1, Figure 5.5) and large-scale GP multi-class classification (Section 5.5.2, Figure 5.6). In the following, we provide additional details for all those experiments.

A.3.1 Binary Classification

Binary Classification with *One* Latent Function. Consider a binary classification task, *i.e.* $C = 2$. Being able to report the probability for *one* of the two classes is sufficient because they have to add up to one for every data point. Thus, while $C = 2$, N-dimensional vectors are typically used to describe this problem [114, Section 3.4]. Using only a single latent function is convenient for illustrative purposes, as *e.g.* the action vectors $\boldsymbol{s}$ in Algorithm 5.2 are N-dimensional (not $2N$-dimensional) and thus easier to visualize.

1D Data. We use a one-dimensional training set in Figure 5.2. $\boldsymbol{X}$ is created by sampling $N = 50$ data points between -3 and 5. The hidden function f is a draw from a GP with mean zero and a GPyTorch [41] RBF kernel with `lengthscale = 1.0` and `outputscale = 5.0`. For each datapoint $\boldsymbol{x}_n$, we sample the positive label with probability $\text{logistic}(f(\boldsymbol{x}_n))$.

2D Data. Two-dimensional data is used in Figures 5.1, 5.3 and 5.4. The data-generating process is analogous to the 1D data, only now, the $N = 100$ training inputs are in the 2D plane: The first coordinate is sampled uniformly between -3 and 5, the second between -4 and 1. The hyperparameters of the RBF kernel are `lengthscale = 1.0`, `outputscale = 10.0` for Figures 5.3 and 5.4 and `outputscale = 20.0` for Figure 5.1.

Details Figure 5.1. In this figure, we compare two versions of our algorithm: IterNCGP-Chol without recycling and IterNCGP-CG with recycling and with compression ($R = 10$). Both runs were conducted on a CPU. The computation of the NLL loss is *not* included in the run time measurement. A description of how the NLL loss can be computed for arbitrary C is given in Appendix A.3.3.

Details Figure 5.2. For Figure 5.2, we compute a sequence of *precise* Newton steps by using IterNCGP with unit vector actions and $j \leq N$ solver iterations. Note that the Newton linear system is N-dimensional, *i.e.* we actually obtain $\boldsymbol{f}_i$ as defined by Equation (5.3).

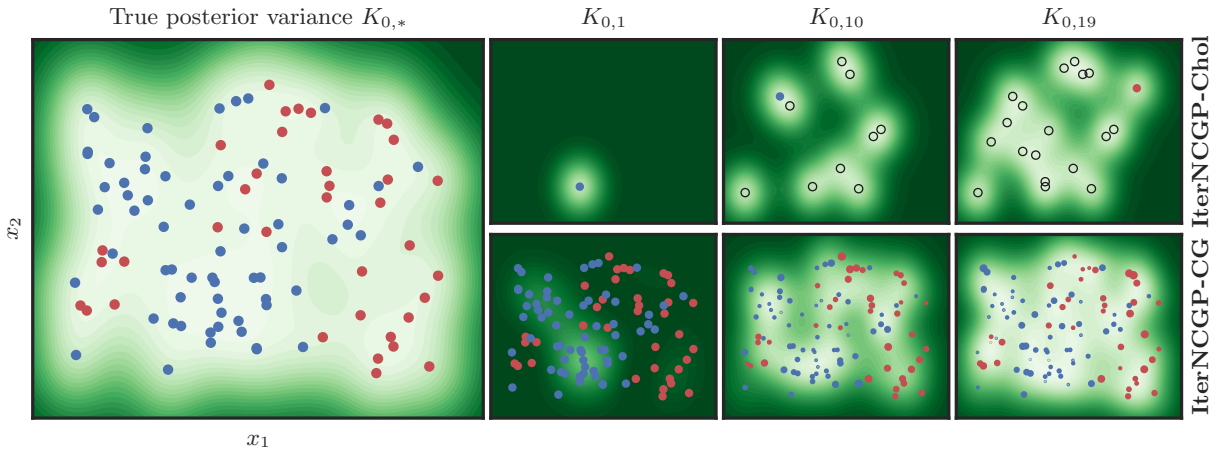


Figure A.1: Different Policies of IterNCGP Applied to GP Classification. *(Left)* The true posterior covariance $K_{0,*}$ ( ) for a binary classification task (●/●). *(Right)* Current estimate of the posterior covariance after 1, 10, and 19 iterations with the unit vector policy *(Top)* and the CG policy *(Bottom)*. Shown are the data points selected by the policy in this iteration with the dot size indicating their relative weight. For IterNCGP-Chol, data points are targeted one by one and previously used data points are marked with (○).

Details Figure 5.3. In this plot, we compare unit vector actions (IterNCGP-Chol) and residual actions (IterNCGP-CG) for the first Newton step ($i = 0$) at three solver iterations $j \in \{1, 10, 19\}$. The true posterior mean function $m_{0,*}$ and covariance function $K_{0,*}$ are computed by using IterNCGP-Chol and $j \leq N = 100$ iterations. Figure A.1 shows the covariance functions corresponding to the mean functions in Figure 5.3.

The actions are visualized by scaling the training data points according to their *relative weight*: First, we take the absolute value of the action vector $\boldsymbol{s}$ from Algorithm 5.2 (element-wise) and then scale its entries linearly such that the smallest entry is 0 and the largest is 1.

Details Figure 5.4. In this figure, we show the initial mean function $m_{1,0}$ and covariance function $K_{1,0}$ in the *second* Newton step for different buffer sizes $R \in \{0, 1, 3, 10\}$. We use CG actions for the first Newton step and let the solver run until convergence (this takes 19 iterations).

A.3.2 Poisson Regression

In Section 5.5.1, we apply IterNCGP to Poisson regression to demonstrate our algorithm's ability to generalize to other (log-concave) likelihoods and to explore the trade-off between the number of (outer loop) mode-finding steps and (inner loop) solver iterations.

Poisson Likelihood. We consider count data $\boldsymbol{y} \in \mathbb{N}_0^N$ that is assumed to be generated from a Poisson likelihood with unknown positive rate $\lambda \colon \mathbb{X} \to \mathbb{R}_{>0}$. Modeling λ with a GP which may

take positive *and negative* values, would therefore be inappropriate. However, we can use a GP for the log-rate $f : \mathbb{X} \to \mathbb{R}$ and regard this as the unknown latent function. With $\boldsymbol{\lambda} := \lambda(\boldsymbol{X}) = \exp(f(\boldsymbol{X})) = \exp(\boldsymbol{f})$, the likelihood is given by

$$p(\boldsymbol{y} \mid \boldsymbol{f}) = \prod_{n=1}^{N} \frac{\lambda_n^{y_n} \exp(-\lambda_n)}{y_n!}.$$

The gradient and (inverse) Hessian of the log-likelihood can be derived in closed form, see Appendix A.1.5.

Data & Model. First, we create $\boldsymbol{X}$ by linearly spacing $N = 100$ points between 0 and 1. For the count data $\boldsymbol{y}$, we sample from a GP with zero mean and a GPyTorch [41] RBF-kernel with `lengthscale = 0.1` and `outputscale = 5.0`. That GP f represents the log-Poisson rate. We then draw counts from a Poisson distribution with rate $\lambda(\boldsymbol{x}_n) = \exp(f(\boldsymbol{x}_n))$ for each data point in the training set. In this experiment, we conduct multiple IterNCGP runs on different training sets. These sets are created by re-drawing from the Poisson distributions with the same rates, *i.e.* the underlying GP for the log-rate does *not* change. Our NCGP's prior uses the same RBF kernel to avoid model mismatch.

IterNCGP-CG Approaches. We consider IterNCGP-CG with four different schedules: A fixed budget of 100 iterations is distributed uniformly over 5, 10, 20 or 100 outer loop steps (see Algorithm 5.1), which limits the number of inner loop iterations (see Algorithm 5.2) to $j \leq 20, 10, 5$ or 1. For each schedule, we perform 10 runs with different training sets, see above. Each run uses recycling without compression. For this experiment, the convergence tolerance in Algorithm 5.1 is set to $\delta = 0.001$. All runs are performed on a single NVIDIA GeForce RTX 2080 Ti 12 GB GPU.

Tracking of Performance Metrics. As a performance metric, we use the NLL loss. The computation of the NLL loss for the test and training set is *not* included in the run time reported in the results. For the NLL loss, we approximate the integral from Section 5.2.3 with MC samples: For each test datum $\boldsymbol{x}_\diamond$, we draw 10^5 MC samples from $\mathcal{N}\big(m_{i,j}(\boldsymbol{x}_\diamond), K_{i,j}(\boldsymbol{x}_\diamond, \boldsymbol{x}_\diamond)\big)$, map those samples $\{f_{\diamond,k}\}_{k=1}^{10^5}$ through the likelihood $p(y_\diamond \mid f_{\diamond,k})$ and average. This yields a loss value for $\boldsymbol{x}_\diamond$ and we obtain the training/test NLL loss by averaging these loss values for all data points in the training/test set.

Approximate Rate Distribution. Using IterNCGP-CG for the Poisson regression problem results in a sequence of posteriors $\mathcal{GP}(m_{i,j}, k_{i,j})$. By drawing MC samples from those posterior GPs and mapping them through the exponential, we obtain an approximated (skewed) distribution for the rate λ. In Figure 5.5 *(Right)*,

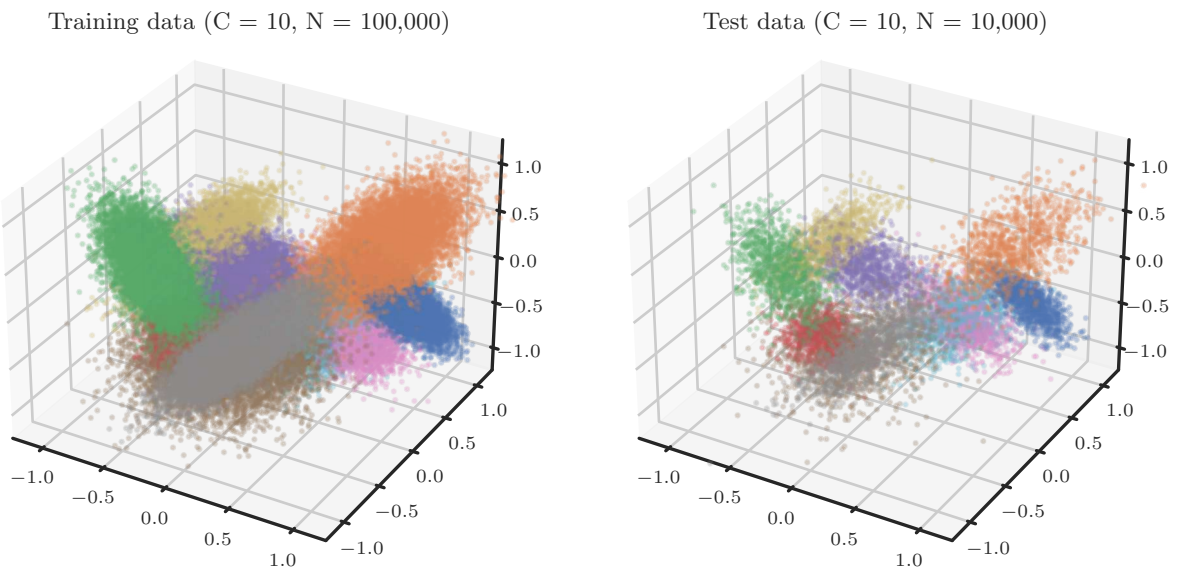


Figure A.2: Gaussian Mixture Training and Test Data. Training data *(Left)* and test data *(Right)* for the first run. Note that the underlying Gaussians are the same for training and test set and for all runs.

we report its median and a 95 % confidence interval between the 2.5 % and 97.5 % percentile.

A.3.3 Large-Scale GP Multi-Class Classification

In this experiment, we empirically evaluate IterNCGP on a large-scale GP multi-class classification problem to exhibit its scalability. We also investigate the impact of compression on performance.

Data. We consider a Gaussian mixture problem with $C = 10$ classes in 3D. For each class, we sample a mean vector uniformly in $[-1, 1]^3$ and a positive definite covariance matrix. For the covariance matrix, we first create a 3×3 matrix $\boldsymbol{C}$ with entries between 0 and 1 (sampled uniformly) and compute the eigenvectors $\boldsymbol{U}$ of $\boldsymbol{C}\boldsymbol{C}^\top$. Then, we create three eigenvalues $\{\lambda_d\}_{d\in\{1,2,3\}}$ uniformly between 0.001 and 0.1 and form the covariance matrix from the eigenvectors $\boldsymbol{U}$ and these eigenvalues, *i.e.* $\boldsymbol{U}\,\mathrm{diag}(\lambda_1, \lambda_2, \lambda_3)\,\boldsymbol{U}^\top$. For each class, 10^4 data points are sampled from the respective Gaussian distribution. This amounts to $N = 10^5$ data points in total. For testing, $N_\diamond = 10^4$ data points are used (10^3 per class).

Our benchmark (Figure 5.6) uses multiple runs for each method. The runs differ in the seed that is used to sample from the Gaussians, *i.e.* the training and test set are different for each run (the underlying Gaussians distributions remain the same). Both the training and test set used in the first run are shown in Figure A.2.

Model. We use a softmax likelihood (see Appendix A.1.6 for the details on the pseudo-inverse $\boldsymbol{W}^+$) and assume independent GPs for the C outputs of the latent function. Each of these GPs uses the

zero function as the prior mean and a Matérn($\frac{3}{2}$) kernel. We use the KeOps [16] version of the GPyTorch [41] kernel with `lengthscale = 0.05` and `outputscale = 0.05`.

SoD Approaches. For the SoD approaches, we create a random subset of the training data (sampling without replacement) of a specific subset size N_{sub}. We then explicitly form $\hat{\boldsymbol{K}}(\boldsymbol{f}_i)$ for every Newton step and compute its Cholesky decomposition via PyTorch's [111] `torch.linalg.cholesky` (instead of using IterGP in Algorithm 5.1 to ensure a competitive baseline implementation). In our experiment, we use four different subset sizes $N_{\text{sub}} \in \{250, 500, 1000, 2000\}$. Each setting is run five times with different random seeds (see above).

SVGP. SVGP [60, 138] is a commonly used variational method for approximative inference in non-conjugate GPs. We use GPyTorch's [41] SVGP implementation and optimize the ELBO for 10^4 seconds using Adam with batch size = 1024. The learning rate $\alpha \in \{0.001, 0.01, 0.05\}$ and the number of inducing points $U \in \{1000, 2500, 5000, 10000\}$ are tuned via grid search (using only a single run). We use U/C inducing points per class ($C = 10$ classes) and initialize them as a random subset of the training data. Within the given run time budget, SVGP performs between 6000 ($U = 1000$) and 600 ($U = 10000$) epochs.

For each of the 12 runs, we extract 6 performance indicators: lowest training/test NLL loss during training, highest training/test accuracy during training and training/test expected calibration error (ECE) at the very end of training. For each of those 6 categories, we determine those two runs with the best performance. This procedure results in a set of 3 runs in total, that are considered the "best" runs for SVGP. Only for those 3 settings, we perform 5 runs each and report their mean performance in Figure 5.6.

IterNCGP-CG Approaches. For comparison, we apply our matrix-free algorithm IterNCGP with residual actions to the *full* training set. We use two configurations: The first one uses recycling *without* compression (*i.e.* $R = \infty$), the second one uses recycling *with* compression ($R = 10$). The number of solver iterations per step is limited by $j \leq 5$. For both settings, we perform 5 runs with different random seeds (see above) and report their mean performance in Figure 5.6.

Tracking of Performance Metrics. As performance metrics, we use accuracy, the negative log-likelihood (NLL) loss and the expected calibration error (ECE) [81] on both the training and test set. The computation of those six metrics is *not* included in the run time reported in the results (Figure 5.6). First, we compute the predictive mean $m_{i,j}(\boldsymbol{x}_\diamond)$ and marginal variance $\text{diag}(K_{i,j}(\boldsymbol{x}_\diamond, \boldsymbol{x}_\diamond))$ (see Equations (5.8) and (5.9)) for each test input $\boldsymbol{x}_\diamond$. Then, we

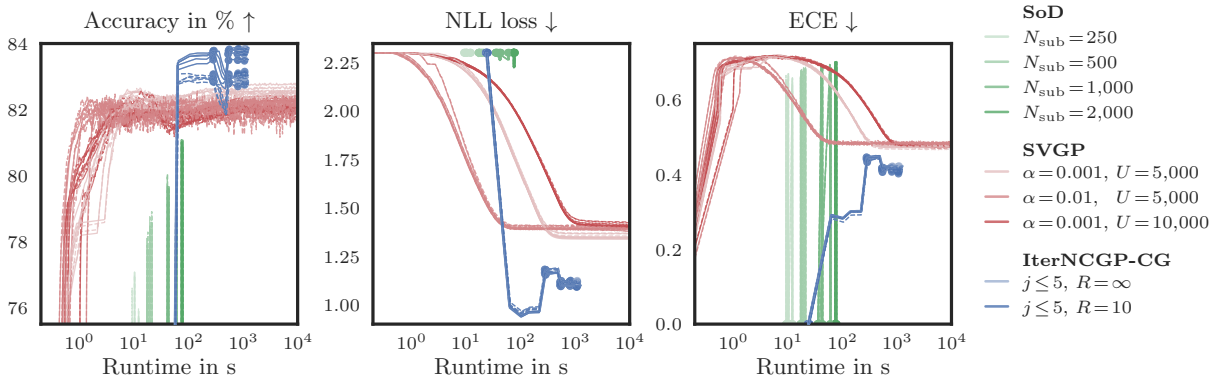


Figure A.3: Large-Scale GP Classification. Same as Figure 5.6, but showing all runs individually (instead of just their average).

use the probit approximation [91, 131] to obtain the predictive probabilities

$$\boldsymbol{\pi}_\diamond = \operatorname{softmax}\left(\frac{m_{i,j}(\boldsymbol{x}_\diamond)}{\sqrt{1+\frac{\pi}{8}\operatorname{diag}(K_{i,j}(\boldsymbol{x}_\diamond,\boldsymbol{x}_\diamond))}}\right) \in \mathbb{R}^C,$$

where the vector division is defined element-wise. This is an approximation of the integral from Section 5.2.3. All three performance metrics are based on the predictive probabilities.

- **Accuracy.** The prediction for $\boldsymbol{x}_\diamond$ is given by $\arg\max_c([\boldsymbol{\pi}_\diamond]_c)$, *i.e.* by the class with the largest predictive probability. The accuracy is defined as the ratio of correctly classified data.
- **NLL Loss.** The NLL loss for $\boldsymbol{x}_\diamond$ is defined as the log-probability for the actual class $y_\diamond$, *i.e.* $\log([\boldsymbol{\pi}_\diamond]_{y_\diamond})$. We obtain the NLL training and test loss by averaging the individual loss values for the entire training/test set.
- **ECE.** The expected calibration error (ECE) [81] is a measure for the calibration of the predictive probabilities. It groups the probabilities of the predicted classes (*i.e.* the classification confidences) into bins and, within these bins, compares the average confidence with the actual accuracy. We use `MulticlassCalibrationError` from `torchmetrics` [33] with default parameter `n_bins=15`.

Individual Runs. Figure 5.6 shows the *average* performance for each of the nine methods/variants over five runs. In order to show, that the observed performance differences are not due to chance, we show the *individual* runs in Figure A.3.

A.3.4 GP Multi-Class Classification on MNIST

To demonstrate IterNCGP's applicability to real-world data sets, we perform an additional experiment (similar to the experiment described in Appendix A.3.3) on a subset ($N_{\text{sub}} = 20{,}000$) of the

MNIST [85] data set. In the following, we describe the experiment and results in more detail.

Remark. For the experiment on synthetic data, we use KeOps on top of GPyTorch for fast kernel-matrix multiplies. However, KeOps scales poorly with the data dimension ($D = 28^2 = 784$ for MNIST)[1]. We therefore revert to GPyTorch's standard implementation. This implementation of the kernel matrix-vector product is fast but causes out-of-memory errors for large data sets, which is why we limit our benchmark to $N_{\text{sub}} = 20{,}000$ training data. To fully realize the potential of our method using KeOps, it might be advisable to apply it only to problems with data dimensions smaller than around 100.

1: https://www.kernel-operations.io/keops/_auto_benchmarks/plot_benchmark_high_dimension.html (accessed May 2024)

Data & Model. We use 20,000 training and 10,000 test images from the MNIST data set and the softmax likelihood. Our model for the latent function is a multi-output GP which uses $C = 10$ independent GPs, each of which is equipped with a Matérn($\frac{3}{2}$) kernel.

Kernel Hyperparameters. As a first step, we determine suitable hyperparameters (the outputscale and lengthscale) for the Matérn($\frac{3}{2}$) kernel by running GPyTorch's SVGP implementation. We use 1000 inducing points per class, a batch size of 1024 and optimize the ELBO using Adam with learning rate 0.001 for 30 epochs (this results in `lengthscale = 1.550934` and `outputscale = 0.451591`). Note that choosing hyperparameters with SVGP may give SVGP an advantage in what performance it can reach, making it a competitive baseline.

SVGP and IterNCGP-CG Approaches. We compare IterNCGP-CG and SVGP both using the same fixed kernel hyperparameters (see above). IterNCGP-CG is applied with recycling and $R = \infty$, *i.e.* without compression. We exclude compression since both IterNCGP runs converge within three iterations, see below. The number of inner loop iterations is limited by $j \leq 1$ or $j \leq 5$, *i.e.* two runs are performed. For the SVGP approach, we optimize the ELBO using Adam with batch size 1024 for 200 seconds. The number of inducing points and the learning rate are tuned via grid search. As in Appendix A.3.3, we use $U \in \{1000, 2500, 5000, 10000\}$ inducing points and three different learning rates $\alpha \in \{0.001, 0.01, 0.05\}$. All 14 runs are performed on a single NVIDIA GeForce RTX 2080 Ti 12 GB GPU.

Results. The results are shown in Figure 5.7. They show the two IterNCGP runs and the best four SVGP runs (these include the best two runs for each of the six performance metrics training/test accuracy/NLL/ECE). Our observations are mostly consistent with the results on the synthetic data (Figure 5.6): Both IterNCGP runs significantly outperform the best SVGP runs in terms of NLL loss

and accuracy. Only the ECE achieved by IterNCGP is slightly worse than for SVGP. However, as explained in Section 5.5.2, a small ECE on its own is not conclusive since we can easily construct a classifier with perfect ECE by randomly sampling predictions.

Additional Material for Chapter 6 B

B.1 Mathematical Details

B.1.1 Reducing the GGN Eigenvalue Problem to the Gram Matrix

For Equation (6.4), consider the left hand side of the GGN's characteristic polynomial $\det(\boldsymbol{G} - \lambda \boldsymbol{I}_P) = 0$. Inserting the ViViT factorization (Equation (6.3)) and using the matrix determinant lemma yields

$$
\begin{aligned}
&\det\left(-\lambda \boldsymbol{I}_P + \boldsymbol{G}\right) \\
&= \det\left(-\lambda \boldsymbol{I}_P + \boldsymbol{V}\boldsymbol{V}^\top\right) && \text{(Low-rank structure (6.3))} \\
&= \det\left(\boldsymbol{I}_{NC} + \boldsymbol{V}^\top(-\lambda \boldsymbol{I}_P)^{-1}\boldsymbol{V}\right)\det(-\lambda \boldsymbol{I}_P) && \text{(Matrix determinant lemma)} \\
&= \det\left(\boldsymbol{I}_{NC} - \frac{1}{\lambda}\boldsymbol{V}^\top\boldsymbol{V}\right)(-\lambda)^P \\
&= \left(-\frac{1}{\lambda}\right)^{NC} \det\left(\boldsymbol{V}^\top\boldsymbol{V} - \lambda \boldsymbol{I}_{NC}\right)(-\lambda)^P \\
&= (-\lambda)^{P-NC} \det\left(\tilde{\boldsymbol{G}} - \lambda \boldsymbol{I}_{NC}\right). && \text{(Gram matrix)}
\end{aligned}
$$

Setting the above expression to zero reveals that the GGN's spectrum decomposes into $P - NC$ zero eigenvalues and the Gram matrix spectrum obtained from $\det(\tilde{\boldsymbol{G}} - \lambda \boldsymbol{I}_{NC}) = 0$.

B.1.2 Relation Between GGN and Gram Matrix Eigenvectors

Assume the nontrivial Gram matrix spectrum $\tilde{\mathbb{S}}_+ = \{(\lambda_k, \tilde{\boldsymbol{u}}_k) | \lambda_k \neq 0, \tilde{\boldsymbol{G}}\tilde{\boldsymbol{u}}_k = \lambda_k \tilde{\boldsymbol{u}}_k\}_{k=1}^K$ with orthonormal eigenvectors $\tilde{\boldsymbol{u}}_j^\top \tilde{\boldsymbol{u}}_k = \delta_{jk}$ (δ represents the Kronecker delta) and $K = \operatorname{rank}(\boldsymbol{G})$. We now show that $\boldsymbol{u}_k = {}^1/\sqrt{\lambda_k}\boldsymbol{V}\tilde{\boldsymbol{u}}_k$ are normalized eigenvectors of $\boldsymbol{G}$ and inherit orthogonality from $\tilde{\boldsymbol{u}}_k$.

To see the first, consider right-multiplication of the GGN with $\boldsymbol{u}_k$, then expand the low-rank structure,

$$
\begin{aligned}
\boldsymbol{G}\boldsymbol{u}_k &= \frac{1}{\sqrt{\lambda_k}}\boldsymbol{V}\boldsymbol{V}^\top\boldsymbol{V}\tilde{\boldsymbol{u}}_k && \text{(Equation (6.3) and definition of } \boldsymbol{u}_k\text{)} \\
&= \frac{1}{\sqrt{\lambda_k}}\boldsymbol{V}\tilde{\boldsymbol{G}}\tilde{\boldsymbol{u}}_k && \text{(Gram matrix)}
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda_k \frac{1}{\sqrt{\lambda_k}} \boldsymbol{V} \tilde{\boldsymbol{u}}_k && \text{(Eigenvector property of } \tilde{\boldsymbol{u}}_k\text{)} \\
&= \lambda_k \boldsymbol{u}_k \,.
\end{aligned}
$$

Orthonormality of the $\boldsymbol{u}_k$ results from the Gram matrix eigenvector orthonormality,

$$
\begin{aligned}
\boldsymbol{u}_j^\top \boldsymbol{u}_k &= \left(\frac{1}{\sqrt{\lambda_j}} \tilde{\boldsymbol{u}}_j^\top \boldsymbol{V}^\top \right) \left(\frac{1}{\sqrt{\lambda_k}} \boldsymbol{V} \tilde{\boldsymbol{u}}_k \right) && \text{(Definition of } \boldsymbol{u}_j, \boldsymbol{u}_k\text{)} \\
&= \frac{1}{\sqrt{\lambda_j \lambda_k}} \tilde{\boldsymbol{u}}_j^\top \tilde{\boldsymbol{G}} \tilde{\boldsymbol{u}}_k && \text{(Gram matrix)} \\
&= \frac{\lambda_k}{\sqrt{\lambda_j \lambda_k}} \tilde{\boldsymbol{u}}_j^\top \tilde{\boldsymbol{u}}_k && \text{(Eigenvector property of } \tilde{\boldsymbol{u}}_k\text{)} \\
&= \delta_{jk} \,. && \text{(Orthonormality)}
\end{aligned}
$$

B.2 Experimental Details

Throughout this section, we use the notation introduced in Section 6.3 (see Table B.1). The code used for the experiments is available at `https://github.com/f-dangel/vivit-experiments`.

GGN Spectra (Figure 6.1a). To obtain the spectra of Figure 6.1a 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}. Figures B.1 and B.2 provide the spectra for all used architectures with both the full GGN and a per-layer block-diagonal approximation.

Table B.1: Notation for Curvature Approximations. The notation is introduced in Section 6.3. This table recapitulates the abbreviations (referring to the approximations introduced in Section 6.2.4) 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 [148]). 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.

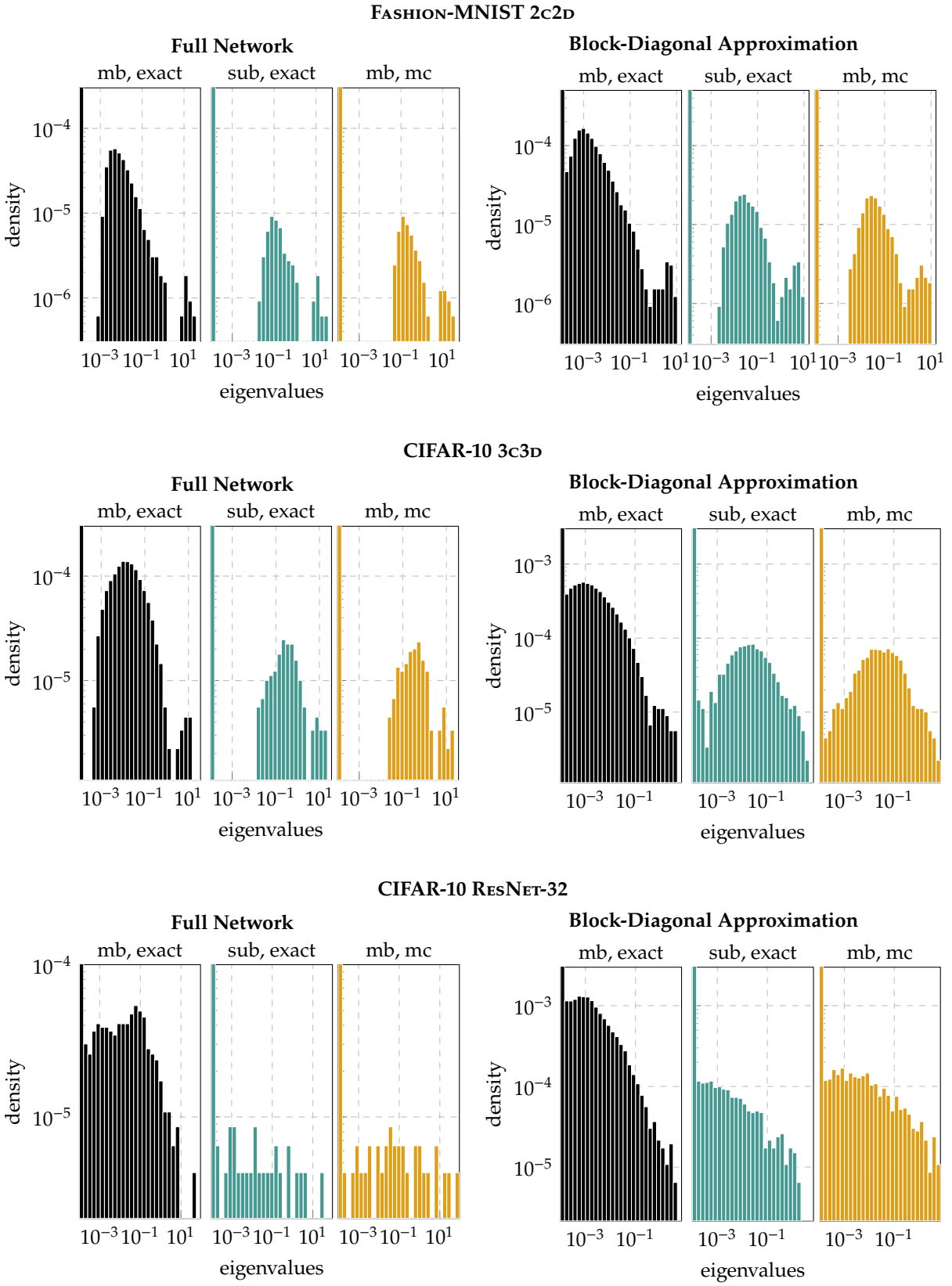


Figure B.1: GGN Spectra of Different Architectures Under ViViT's Approximations. Left and right columns contain results with the full network's GGN and a per-layer block-diagonal approximation, respectively. The column labels are explained in Table B.1.

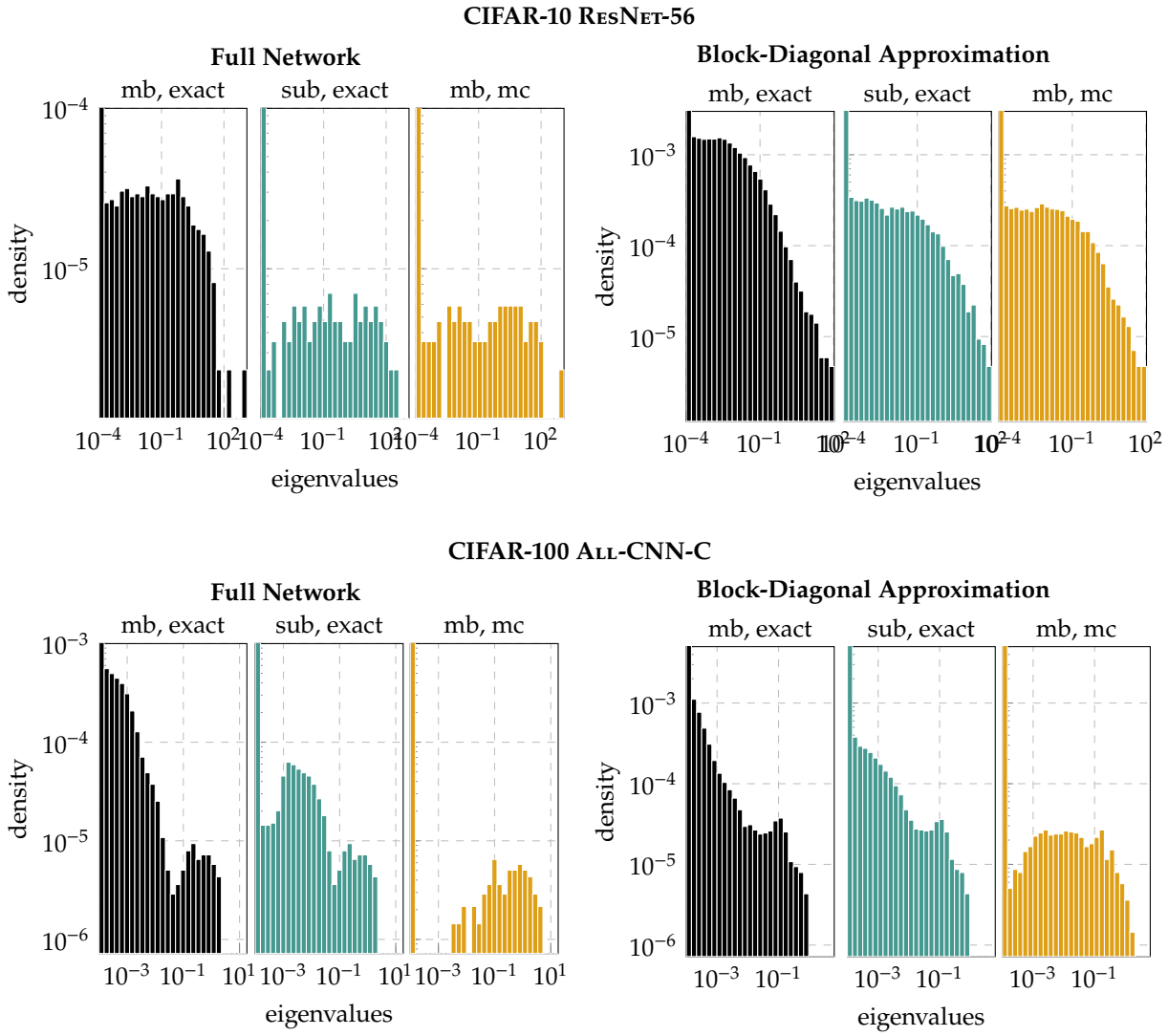


Figure B.2: GGN Spectra of Different Architectures Under ViViT's Approximations. Left and right columns contain results with the full network's GGN and a per-layer block-diagonal approximation, respectively. The column labels are explained in Table B.1.

B.2.1 Performance Evaluation

Hardware Specifications. Results in this section 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.

Architectures. We use untrained deep convolutional and residual networks from DeepOBS [123] and [64]. If a net has batch normalization layers, we set them to evaluation mode. Otherwise, the loss would not obey the sum structure of Equation (6.1). 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 residual networks 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 [148]). The MC approximation is always evaluated with a single sample ($M = 1$).

Memory Performance (Critical Batch Sizes). Two tasks are considered (see Section 6.3.1):

1. **Computing Eigenvalues:** Compute the nontrivial eigenvalues $\{\lambda_k \,|\, (\lambda_k, \tilde{\boldsymbol{u}}_k) \in \tilde{\mathbb{S}}_+\}$.
2. **Computing the Top Eigenpair:** Compute the top eigenpair $(\lambda_1, \boldsymbol{u}_1)$.

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 $[1; 32768]$.

Figures B.5 to B.14a,b present the results. As described in Section 6.2.3, computing eigenvalues is more memory-efficient than computing eigenvectors and exhibits larger critical batch sizes. In line with the description in Section 6.2.4, 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, we find that 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 [145] 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.

Figures B.5 to B.14c,d 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.

Power Iteration with Relaxed Hyperparameters. When ViViT's approximations are used for computing eigenvalues, a power iteration with relaxed hyperparameters might be an alternative. We thus extend the run time experiment from Section 6.3.1 in the following way: We consider the task of computing the top-10 eigenpairs of the mini-batch GGN on CIFAR-10 3c3d, with the same batch size $N = 128$, on GPU using both ViViT and the power iteration. The resulting approximations of the eigenvalues $\hat{\lambda}_k$, $k = 1, \dots, 10$ are compared with the exact eigenvalues λ_k, $k = 1, \dots, 10$ using

$$1 - \frac{1}{10} \sum_{k=1}^{10} \frac{|\hat{\lambda}_k - \lambda_k|}{|\lambda_k|}$$

as a measure of accuracy. We run the power iteration with 20 different convergence tolerances varying on a logarithmic grid from 10^{-5} to 10^{-1}, and disable termination due to exceeded iterations. Each setting is repeated 20 times, and we report the best run time and corresponding accuracy in Figure B.3.

In this setting, computing the exact top-10 eigenvalues with ViViT is faster than approximating them with a power iteration, even when using the largest tested tolerance. Also, when using ViViT with approximations through sub-sampling and MC-approximation,

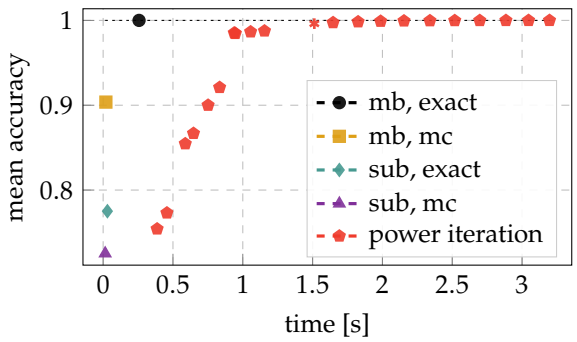


Figure B.3: Accuracy per Run Time Performance. Comparison of ViViT and its approximations to the power iteration for computing the top-10 eigenvalues for the 3c3d architecture ($P = 895\,210$) on CIFAR-10 ($C = 10$). For the power iteration, different tolerances ranging from 10^{-5} to 10^{-1} are used. The red star shows the result for the default tolerance.

these runs require less run time than the power iteration at similar accuracy. In the plot, we also highlighted the power iteration run with PyHessian's default convergence parameters (red star marker). It seems to yield a relatively good trade-off between time and accuracy for the power iteration.

These results show that, also in terms of "accuracy per run time", ViViT is superior to a power iteration in the mini-batch setting due to its increased parallelism.

Power Iteration on the GGN vs. Power Iteration on the GGN Gram Matrix. In the run time evaluation of our method (Section 6.3.1) we compute the k leading GGN eigenpairs by computing the full GGN Gram matrix spectrum and discarding all but the leading eigenpairs. Here, we present additional results where we exchange the full diagonalization by a power iteration with identical convergence hyperparameters as the baseline; a power iteration on the GGN.

Figure B.4 visualizes the comparison for the same setting as Figure 6.2b in the main text. In case of no approximations (mb, exact) where the Gram matrix dimension is largest, the power iteration can further reduce the run time shown in Figure 6.2b. However, for the GGN approximations through sub-sampling or MC approximation, the power iteration on the (rather small) Gram matrix, deteriorates performance in comparison to the results reported in Figure 6.2b as the number of leading eigenvalues increases. In this regime (small Gram matrix, many requested eigenvalues), the simplistic power iteration can require more matrix-vector products than a sophisticated eigensolver that computes the full spectrum. As in the main text, these findings show that ViViT (even in the exact version) outperforms the power iteration for $k \geq 2$.

Note on CIFAR-100 (Large C). For data sets with a large number of classes, like CIFAR-100 ($C = 100$), computations with the exact GGN are costly. In particular, constructing the Gram matrix $\tilde{\boldsymbol{G}}$ has

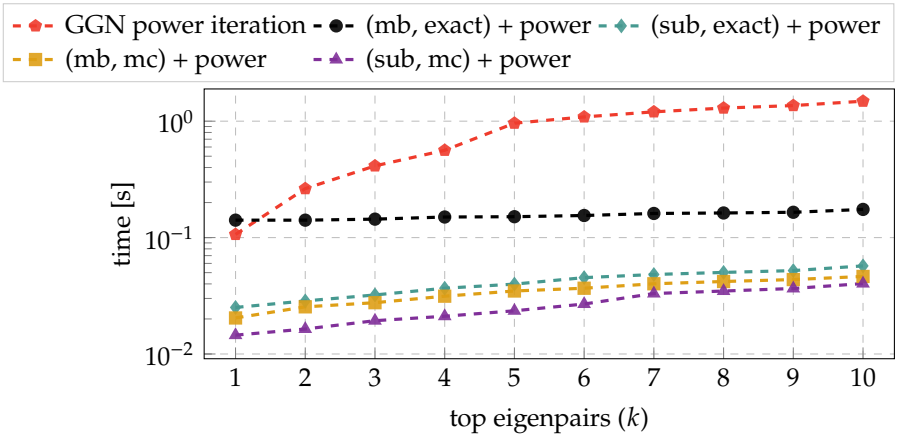


Figure B.4: Power Iteration on the GGN vs. Power Iteration on the Gram Matrix. The figure considers the same setting as Figure 6.2b (3c3d on CIFAR-10 and GPU). However, we exchange the full diagonalization of the GGN's Gram matrix by a power iteration with identical convergence hyperparameters as the baseline power iteration on the GGN (at most 100 matrix-vector producst per eigenvalue; stop if the eigenvalue estimate's relative change is less than 10^{-3}).

quadratic memory cost in C, and its eigendecomposition has cubic cost in time with C (see Section 6.2.3).

As a result, the exact computation only works with batch sizes smaller than DeepOBS' default ($N = 256$ for CIFAR-100, see Figures B.13 and B.14a,b). 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 B.14d.

For such data sets, the approximations proposed in Section 6.2.4 are essential to reduce costs. The most effective approximation to eliminate the scaling with C is using an MC approximation. Figures B.13 and B.14 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 $\boldsymbol{G}$,

$$-(\boldsymbol{G} + \delta\boldsymbol{I})^{-1}\boldsymbol{g} = \sum_{k=1}^{K} \frac{-\gamma_k}{\lambda_k + \delta}\boldsymbol{u}_k + \sum_{k=K+1}^{P} \frac{-\gamma_k}{\delta}\boldsymbol{u}_k \,. \tag{B.1}$$

It corresponds to a Newton update along nontrivial eigendirections that uses the first- and second-order directional derivatives described in Section 6.2.2 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 (B.1) as Newton step. As described in Section 6.2.3, we can perform the weighted sum in the

Gram matrix space, rather than the parameter space, by computing

$$\sum_{k=1}^{K} \frac{-\gamma_k}{\lambda_k + \delta} \boldsymbol{u}_k = \sum_{k=1}^{K} \frac{-\gamma_k}{\lambda_k + \delta} \frac{1}{\sqrt{\lambda_k}} \boldsymbol{V} \tilde{\boldsymbol{u}}_k = \boldsymbol{V} \left(\sum_{k=1}^{K} \frac{-\gamma_k}{(\lambda_k + \delta)\sqrt{\lambda_k}} \tilde{\boldsymbol{u}}_k \right).$$

This way, only a single vector needs to be transformed from Gram space into parameter space.

Table B.2 shows the critical batch sizes for the Newton step computation (first term on the right-hand side of Equation (B.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

(a)

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

Block-Diagonal Approximation

(b)

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) (d)

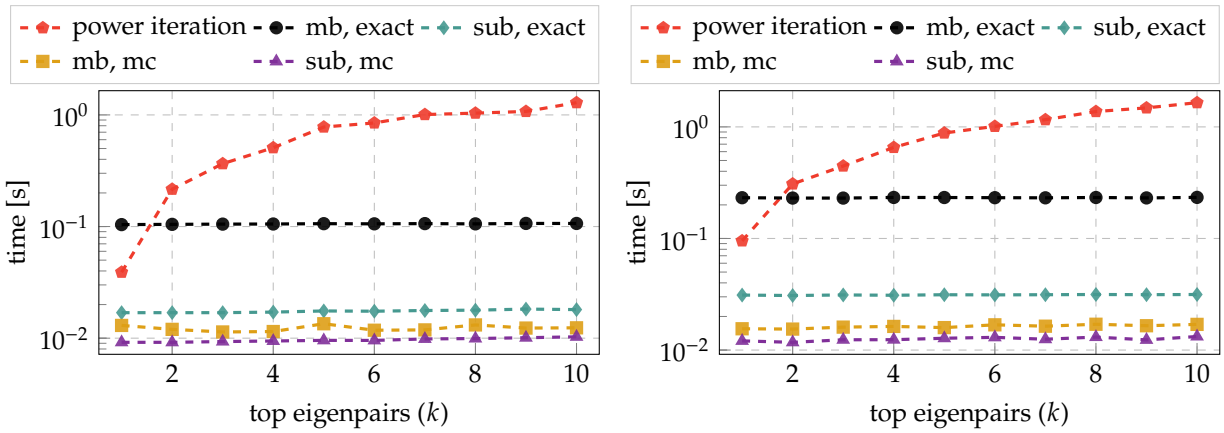


Figure B.5: GPU Memory and Run Time Performance for the 2c2d Architecture on Fashion-MNIST. Left and right columns show results with the full network's GGN ($P = 3\,274\,634$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (a, b) Critical batch sizes N_{crit} for computing eigenvalues 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 $N = 128$.

Full Network

(a)

N_{crit} (eigenvalues)

GGN^{Data}	mb	sub
exact	4224	21164
mc	24064	> 32768

N_{crit} (top eigenpair)

GGN^{Data}	mb	sub
exact	3276	21295
mc	24064	> 32768

Block-Diagonal Approximation

(b)

N_{crit} (eigenvalues)

GGN^{Data}	mb	sub
exact	4416	21453
mc	23723	> 32768

N_{crit} (top eigenpair)

GGN^{Data}	mb	sub
exact	3276	20378
mc	23457	> 32768

(c) (d)

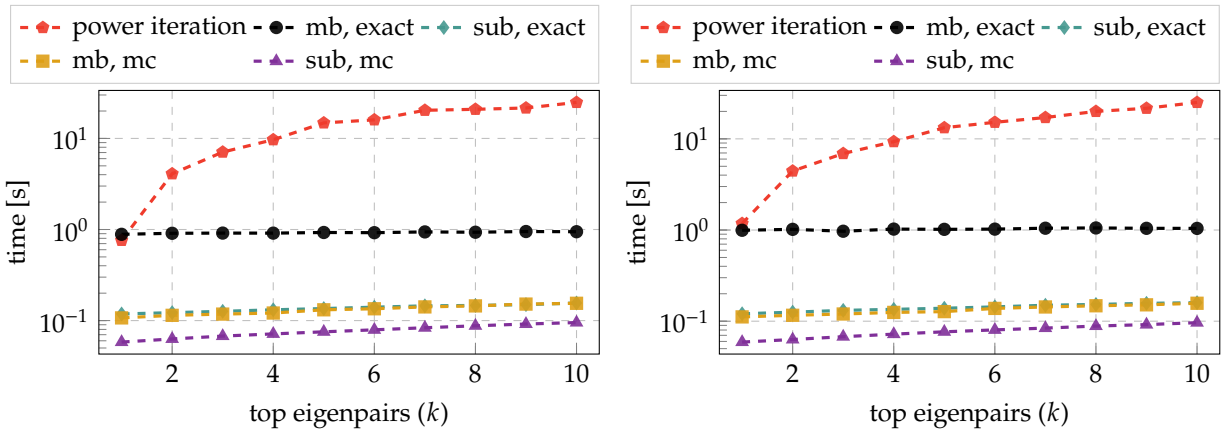


Figure B.6: CPU Memory and Run Time Performance for the 2c2d Architecture on Fashion-MNIST. Left and right columns show results with the full network's GGN ($P = 3\,274\,634$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (a, b) Critical batch sizes N_{crit} for computing eigenvalues 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 $N = 128$.

Full Network

(a)

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

Block-Diagonal Approximation

(b)

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) (d)

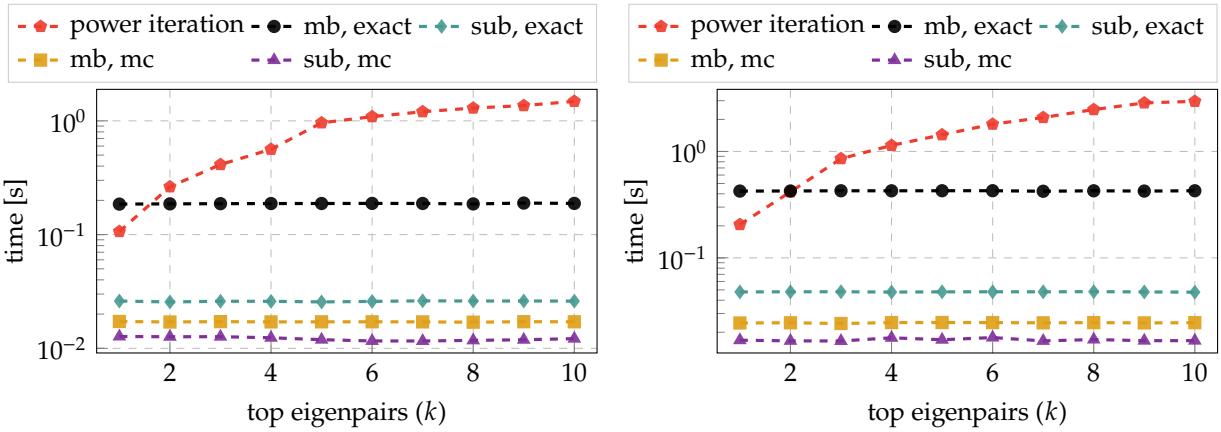


Figure B.7: GPU Memory and Run Time Performance for the 3c3d Architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($P = 895\,210$, $C = 10$) and a per-layer block-diagonal approximation, respectively. **(a, b)** Critical batch sizes N_{crit} for computing eigenvalues 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)

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

Block-Diagonal Approximation

(b)

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) (d)

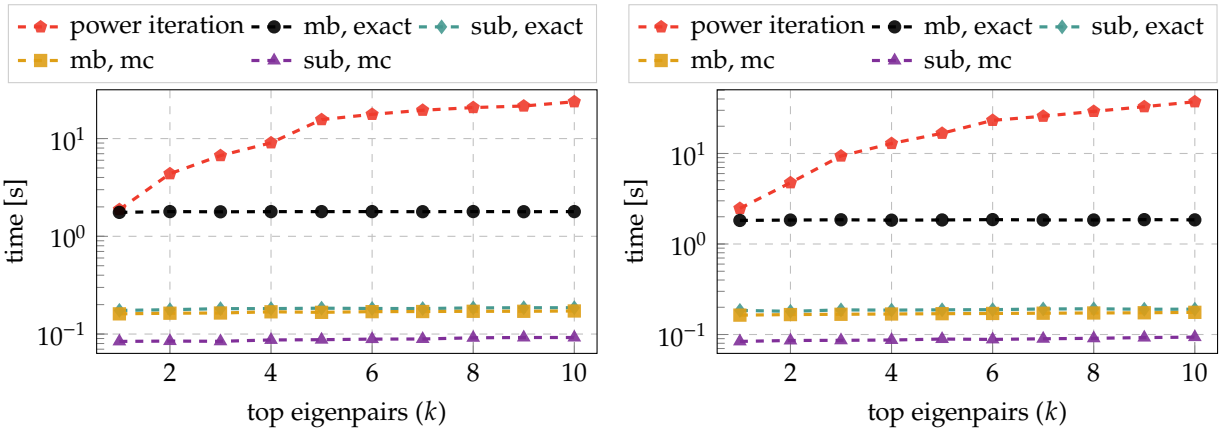


Figure B.8: CPU Memory and Run Time Performance for the 3c3d Architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($P = 895\,210$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (a, b) Critical batch sizes N_{crit} for computing eigenvalues 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) N_{crit} (eigenvalues)

GGN^{Data}	mb	sub
exact	1054	2052
mc	2064	2271

N_{crit} (top eigenpair)

GGN^{Data}	mb	sub
exact	364	1360
mc	1536	2176

Block-Diagonal Approximation

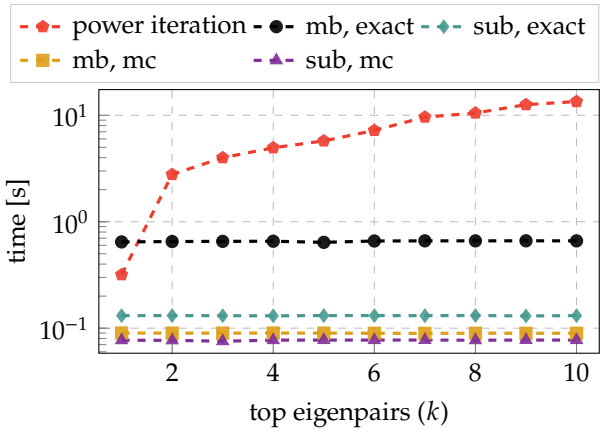
(b) N_{crit} (eigenvalues)

GGN^{Data}	mb	sub
exact	1061	2048
mc	2064	2271

N_{crit} (top eigenpair)

GGN^{Data}	mb	sub
exact	1040	2048
mc	2064	2271

(c)



(d)

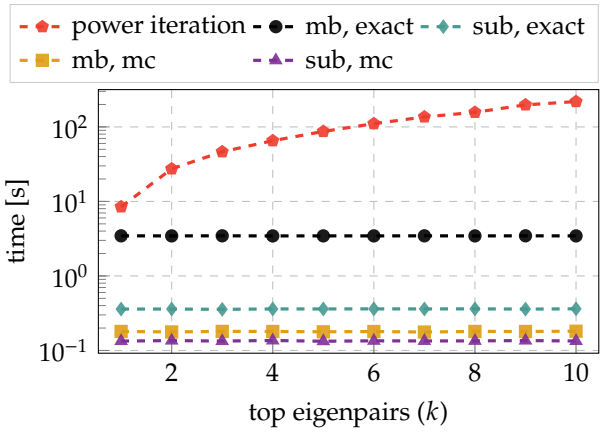
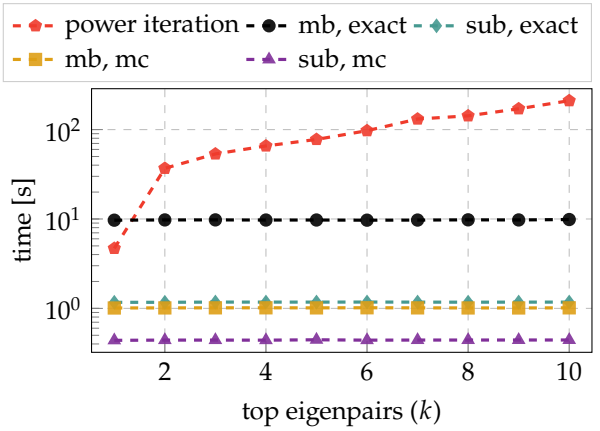


Figure B.9: GPU Memory and Run Time Performance for the ResNet-32 Architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($P = 464\,154$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (a, b) Critical batch sizes N_{crit} for computing eigenvalues 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)



Block-Diagonal Approximation

(d)

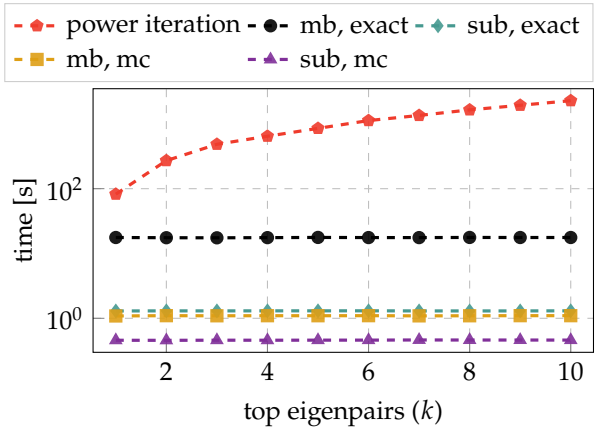


Figure B.10: CPU Memory and Run Time Performance for the ResNet-32 Architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($P = 464\,154$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (c, d) Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full Network

(a)

N_{crit} (eigenvalues)

GGN^{Data}	mb	sub
exact	837	1247
mc	1262	1312

N_{crit} (top eigenpair)

GGN^{Data}	mb	sub
exact	217	765
mc	896	1240

Block-Diagonal Approximation

(b)

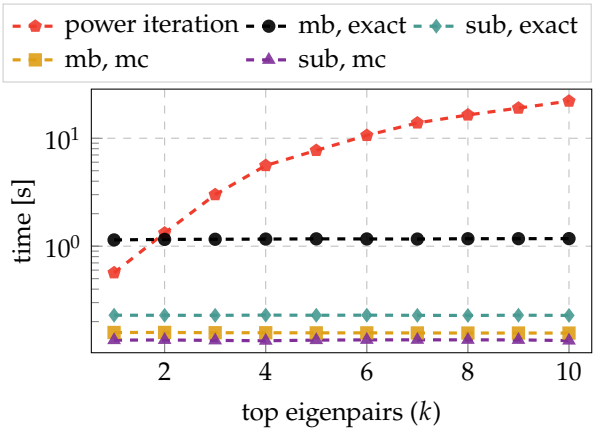
N_{crit} (eigenvalues)

GGN^{Data}	mb	sub
exact	688	1247
mc	1232	1259

N_{crit} (top eigenpair)

GGN^{Data}	mb	sub
exact	383	1247
mc	1232	1255

(c)



(d)

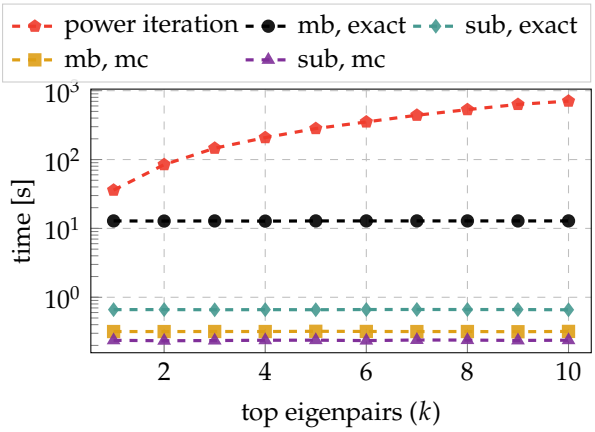
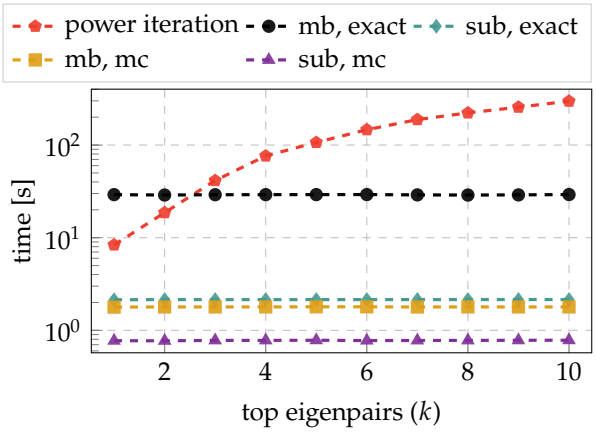


Figure B.11: GPU Memory and Run Time Performance for the ResNet-56 Architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($P = 853\,018$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (a, b) Critical batch sizes N_{crit} for computing eigenvalues 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)



Block-Diagonal Approximation

(d)

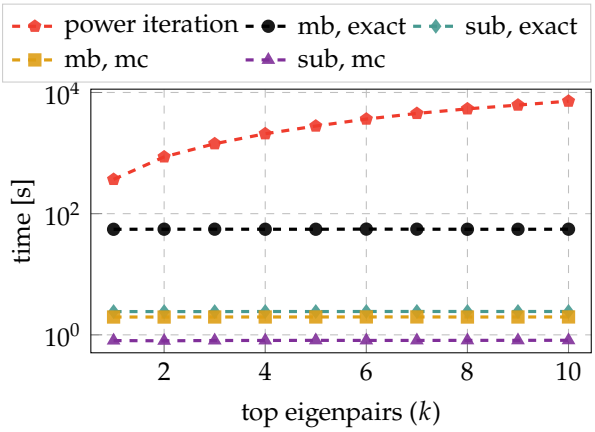


Figure B.12: CPU Memory and Run Time Performance for the ResNet-56 Architecture on CIFAR-10. Left and right columns show results with the full network's GGN ($P = 853\,018$, $C = 10$) and a per-layer block-diagonal approximation, respectively. (c, d) Run time comparison with a power iteration for extracting the k leading eigenpairs using a mini-batch of size $N = 128$.

Full Network

(a)

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)

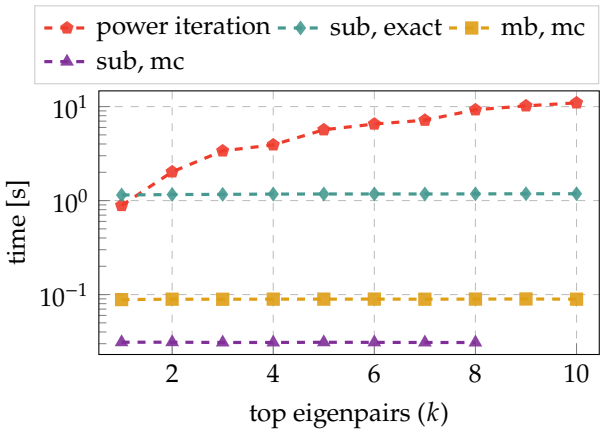
N_{crit} (eigenvalues)

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)



(d)

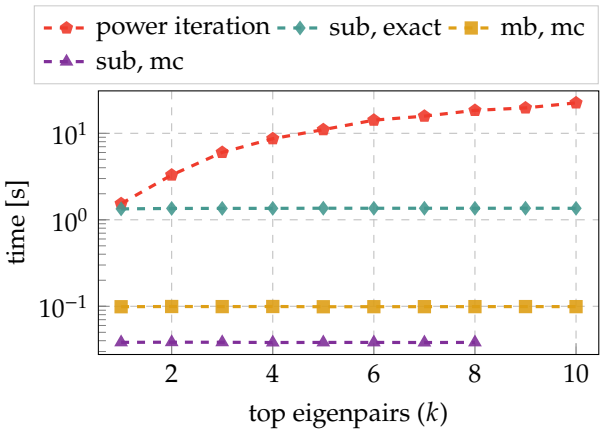


Figure B.13: GPU Memory and Run Time Performance for the All-CNN-C Architecture on CIFAR-100. Left and right columns show results with the full network's GGN ($P = 1\,387\,108$, $C = 100$) and a per-layer block-diagonal approximation, respectively. (a, b) Critical batch sizes N_{crit} for computing eigenvalues 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)

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)

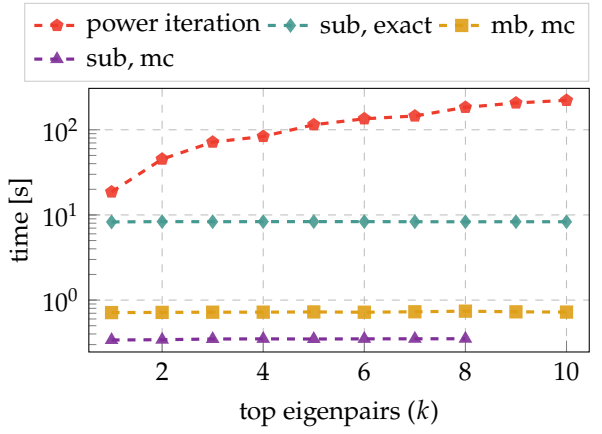
N_{crit} (eigenvalues)

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)



(d)

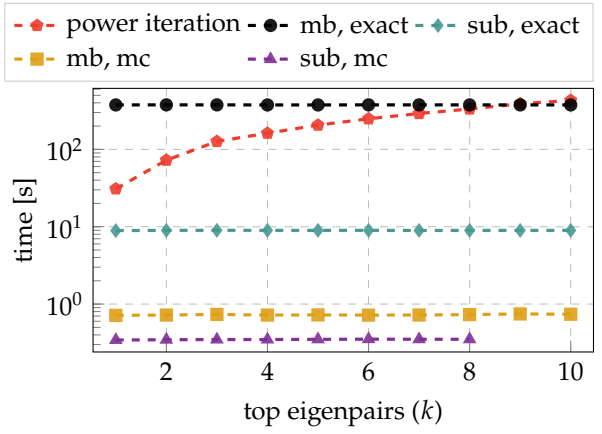


Figure B.14: CPU Memory and Run Time Performance for the All-CNN-C Architecture on CIFAR-100. Left and right columns show results with the full network's GGN ($P = 1\,387\,108$, $C = 100$) and a per-layer block-diagonal approximation, respectively. (a, b) Critical batch sizes N_{crit} for computing eigenvalues 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 B.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)			N_{crit} (CPU)		
GGN^{Data}	mb	sub	GGN^{Data}	mb	sub
exact	66	159	exact	202	487
mc	362	528	mc	1107	1639

Block-Diagonal Approximation

N_{crit} (GPU)			N_{crit} (CPU)		
GGN^{Data}	mb	sub	GGN^{Data}	mb	sub
exact	68	159	exact	210	487
mc	368	528	mc	1137	1643

CIFAR-10 3C3D

Full Network

N_{crit} (GPU)			N_{crit} (CPU)		
GGN^{Data}	mb	sub	GGN^{Data}	mb	sub
exact	208	727	exact	667	2215
mc	1055	1816	mc	3473	5632

Block-Diagonal Approximation

N_{crit} (GPU)			N_{crit} (CPU)		
GGN^{Data}	mb	sub	GGN^{Data}	mb	sub
exact	349	795	exact	1046	2423
mc	1659	2112	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)			N_{crit} (CPU)		
GGN^{Data}	mb	sub	GGN^{Data}	mb	sub
exact	13	87	exact	43	309
mc	640	959	mc	2015	2865

Block-Diagonal Approximation

N_{crit} (GPU)			N_{crit} (CPU)		
GGN^{Data}	mb	sub	GGN^{Data}	mb	sub
exact	35	135	exact	95	504
mc	1079	1536	mc	3360	3920

Table B.3: Hyperparameter Settings for Training Runs. For both SGD and Adam, we report their learning rates α (taken from the baselines in [28] 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.

Problem	SGD	Adam	Batch size	Train epochs
Fashion-MNIST 2c2d	$\alpha \approx 2.07 \cdot 10^{-2}$	$\alpha \approx 1.27 \cdot 10^{-4}$	$N = 128$	100
CIFAR-10 3c3d	$\alpha \approx 3.79 \cdot 10^{-3}$	$\alpha \approx 2.98 \cdot 10^{-4}$	$N = 128$	100
CIFAR-10 ResNet-32	$\alpha \approx 6.31 \cdot 10^{-2}$	$\alpha \approx 2.51 \cdot 10^{-3}$	$N = 128$	180
CIFAR-100 All-CNN-C	$\alpha \approx 4.83 \cdot 10^{-1}$	$\alpha \approx 6.95 \cdot 10^{-4}$	$N = 256$	350

B.2.2 Training of Neural Networks

Procedure. We train the following DeepOBS [123] 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 [28] (see Table B.3).

We also train a residual network ResNet-32 [54] 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 [123], 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 B.3. The training metrics training/test loss/accuracy for all eight test problems are shown in Figure B.15 and B.16.

B.2.3 GGN vs. Hessian

Checkpoints. During training of the neural networks (see Appendix B.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 6.3.2: For the set of orthonormal eigenvectors $\{\boldsymbol{u}_c^{\boldsymbol{A}}\}_{c=1}^{C}$ to the C largest eigenvalues of some symmetric matrix $\boldsymbol{A}$, let $\boldsymbol{P}^{\boldsymbol{A}} = (\boldsymbol{u}_1^{\boldsymbol{A}}, \dots, \boldsymbol{u}_C^{\boldsymbol{A}})(\boldsymbol{u}_1^{\boldsymbol{A}}, \dots, \boldsymbol{u}_C^{\boldsymbol{A}})^\top$. As in Gur-Ari et al. [50], the overlap between two subspaces $\mathcal{E}^{\boldsymbol{A}} = \operatorname{span}(\boldsymbol{u}_1^{\boldsymbol{A}}, \dots, \boldsymbol{u}_C^{\boldsymbol{A}})$

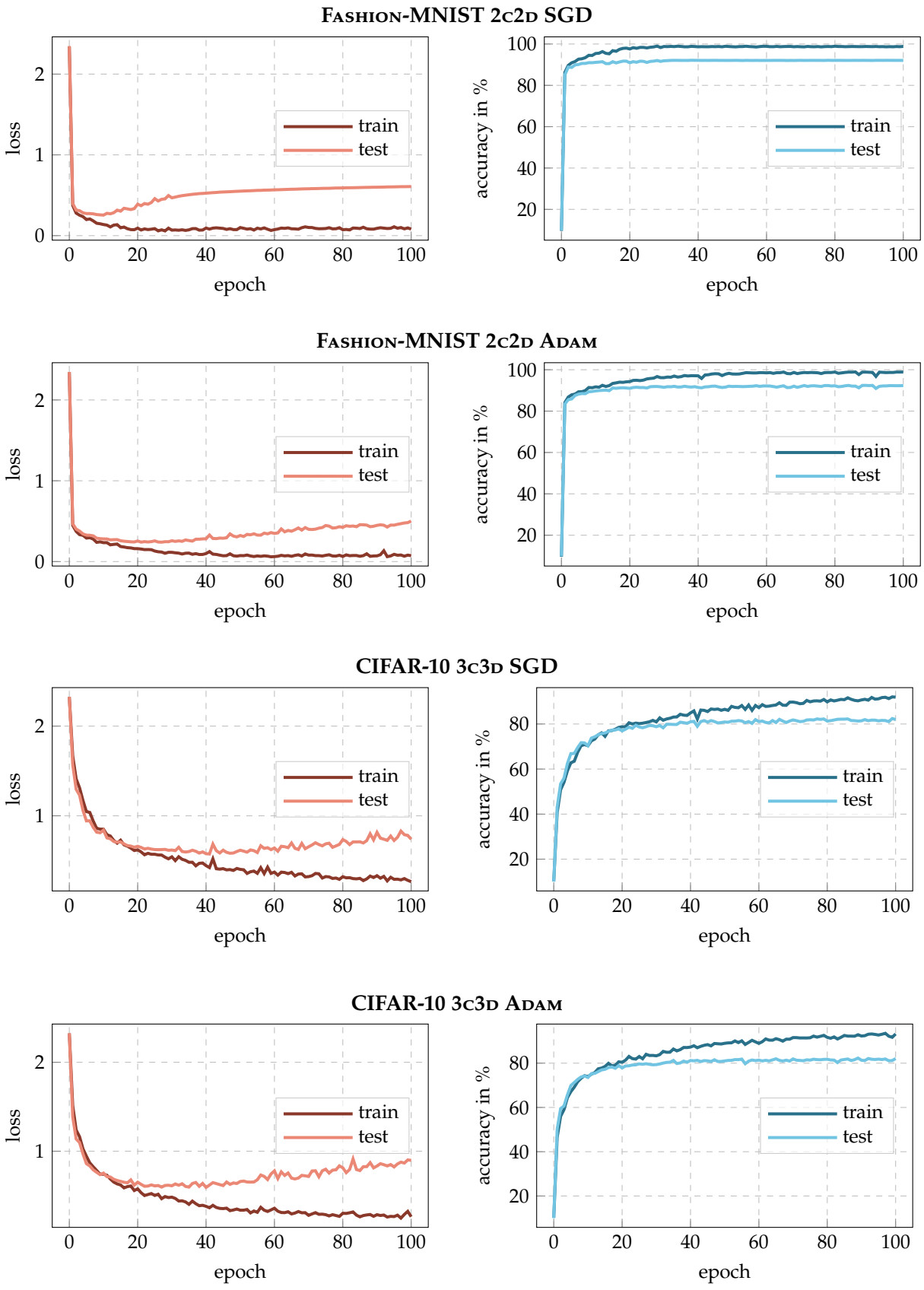


Figure B.15: Training Metrics (1). Training/test loss/accuracy for all test problems.

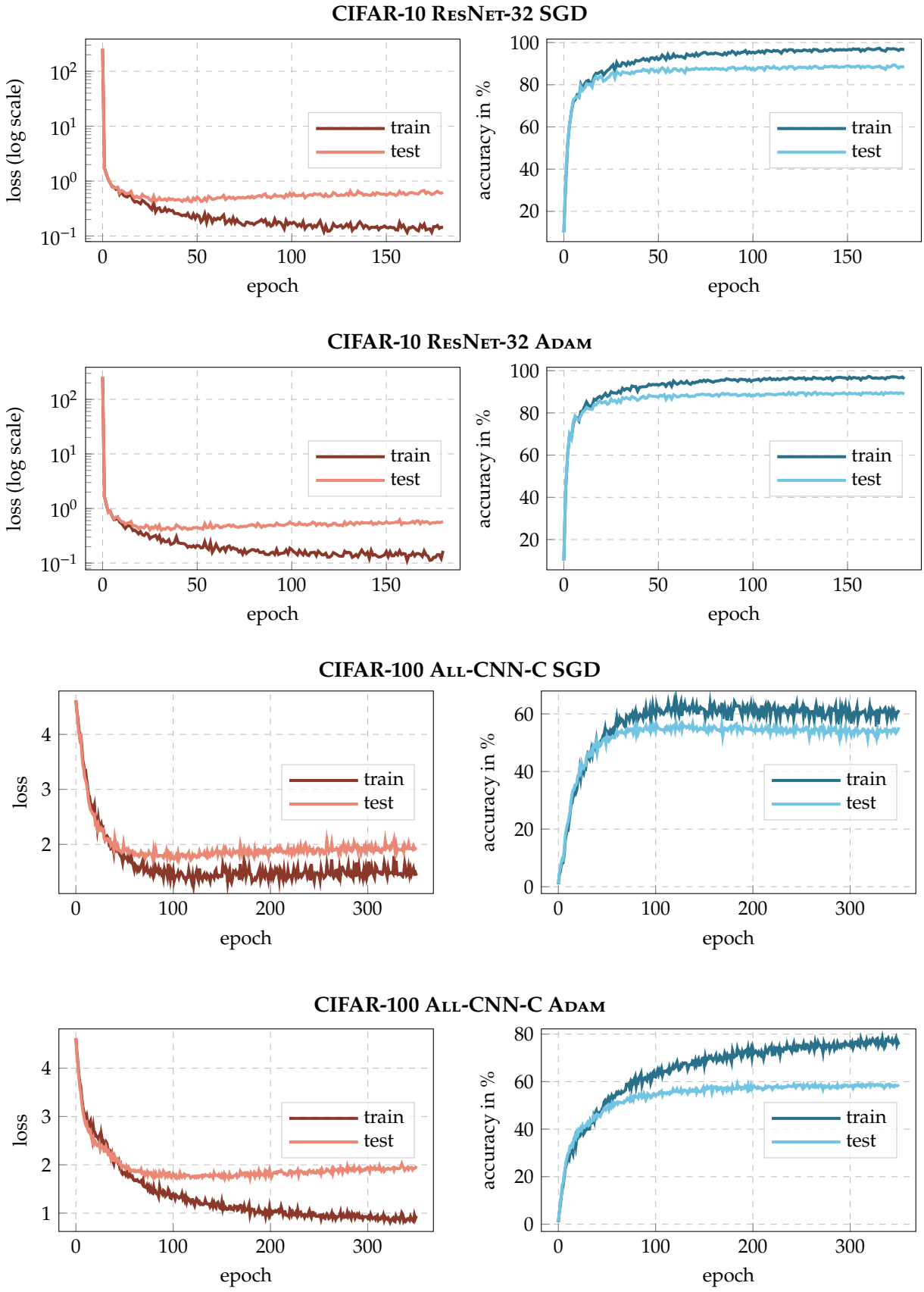


Figure B.16: Training Metrics (2). Training/test loss/accuracy for all test problems.

and $\mathcal{E}^B = \text{span}\,(\boldsymbol{u}_1^B, \ldots, \boldsymbol{u}_C^B)$ of the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is defined by

$$\text{overlap}(\mathcal{E}^A, \mathcal{E}^B) = \frac{\text{Tr}\,(\boldsymbol{P}^A \boldsymbol{P}^B)}{\sqrt{\text{Tr}\,(\boldsymbol{P}^A)\,\text{Tr}\,(\boldsymbol{P}^B)}} \in [0, 1]\,.$$

The overlap can be computed efficiently by using the trace's cyclic property: It holds $\text{Tr}\,(\boldsymbol{P}^A \boldsymbol{P}^B) = \text{Tr}\,(\boldsymbol{W}^\top \boldsymbol{W})$ with $\boldsymbol{W} = (\boldsymbol{u}_1^A, \ldots, \boldsymbol{u}_C^A)^\top (\boldsymbol{u}_1^B, \ldots, \boldsymbol{u}_C^B) \in \mathbb{R}^{C \times C}$. Note that this is a small $C \times C$ matrix, whereas $\boldsymbol{P}^A, \boldsymbol{P}^B \in \mathbb{R}^{P \times P}$. Since

$$\begin{aligned}
&\text{Tr}\,(\boldsymbol{P}^A) \\
&= \text{Tr}\,((\boldsymbol{u}_1^A, \ldots, \boldsymbol{u}_C^A)(\boldsymbol{u}_1^A, \ldots, \boldsymbol{u}_C^A)^\top) \\
&= \text{Tr}\,((\boldsymbol{u}_1^A, \ldots, \boldsymbol{u}_C^A)^\top (\boldsymbol{u}_1^A, \ldots, \boldsymbol{u}_C^A)) && \text{(Cyclic property of trace)} \\
&= \text{Tr}\,(\boldsymbol{I}_C) && \text{(Orthonormality of the eigenvectors)} \\
&= C
\end{aligned}$$

(and analogous $\text{Tr}\,(\boldsymbol{P}^B) = C$), 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 B.17. 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.

B.2.4 Eigenspace Under Noise and Approximations

Procedure (1). We use the checkpoints and the definition of overlaps between eigenspaces from Appendix B.2.3. For the approximation of the GGN, we consider the cases listed in Table B.4.

For every checkpoint and case, we compute the top-C eigenvectors of the respective approximation to the GGN. The eigenvectors are

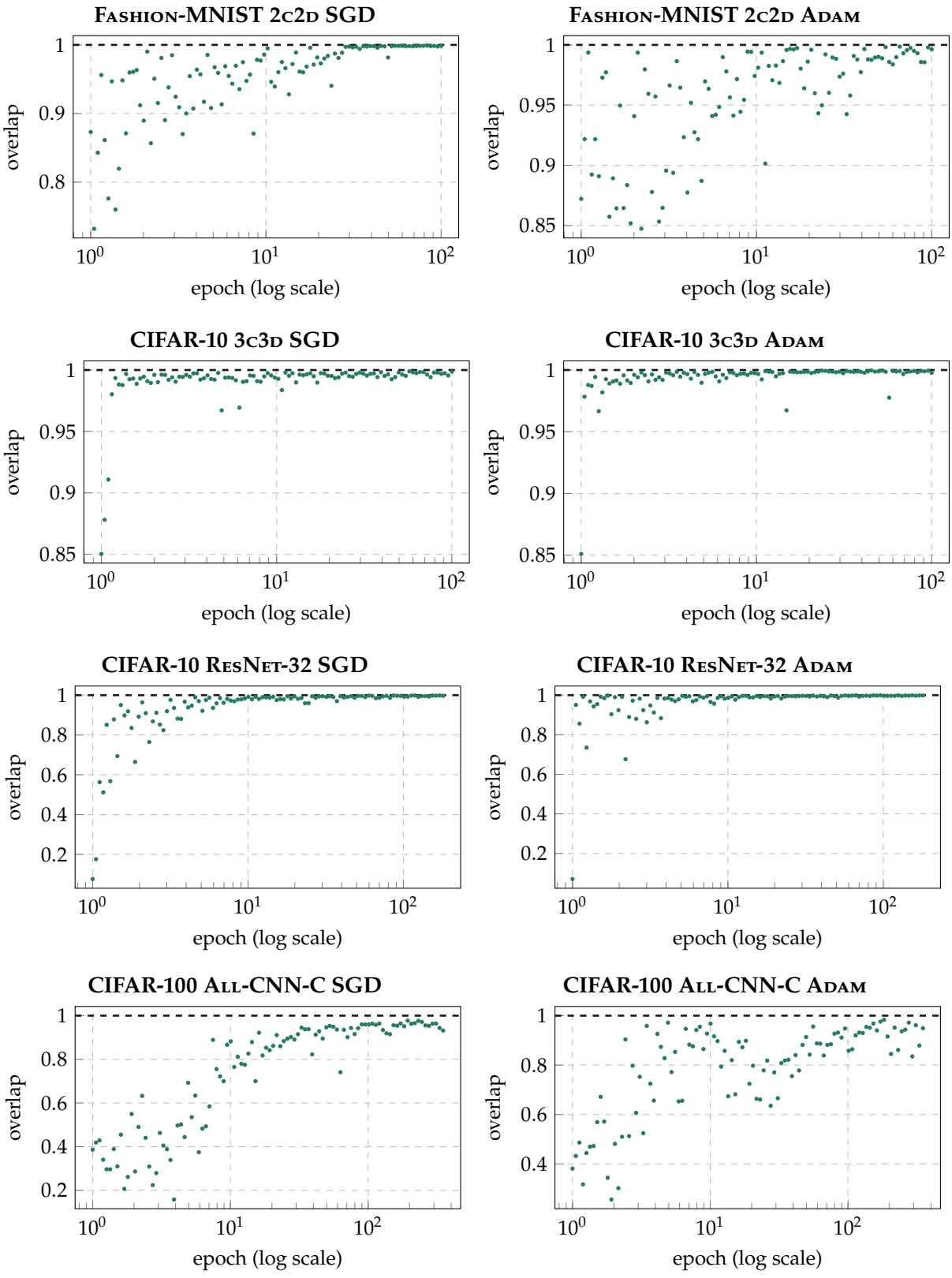


Figure B.17: Full-Batch GGN vs. Full-Batch Hessian. Overlap between the top-C eigenspaces of the full-batch GGN and full-batch Hessian during training for all test problems.

Table B.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

either computed directly using ViViT (by transforming eigenvectors of the Gram matrix into parameter space, see Section 6.2.1) or, if not applicable (because memory requirements exceed N_{crit}, see Section 6.3.1), using an iterative matrix-free approach. The overlap is computed in reference to the GGN's full-batch top-C eigenspace (see Appendix B.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 B.18 and B.19. 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 B.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.

Results (2). The results can be found in Figure B.20. 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

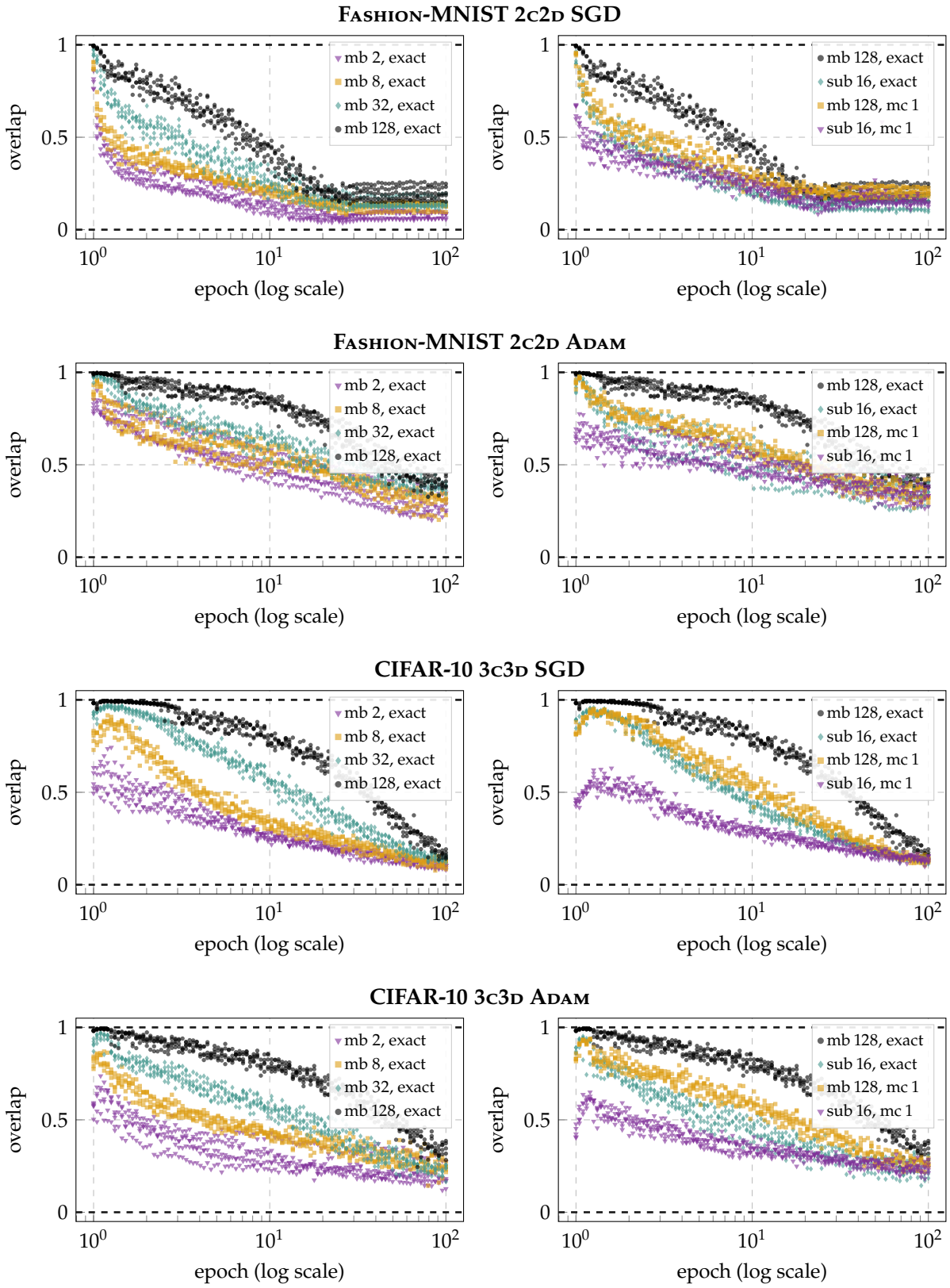


Figure B.18: ViViT vs. Full-Batch GGN (1). Overlap between the top-C eigenspaces of different GGN approximations and the full-batch GGN during training for all test problems. Each approximation is evaluated on 5 different mini-batches.

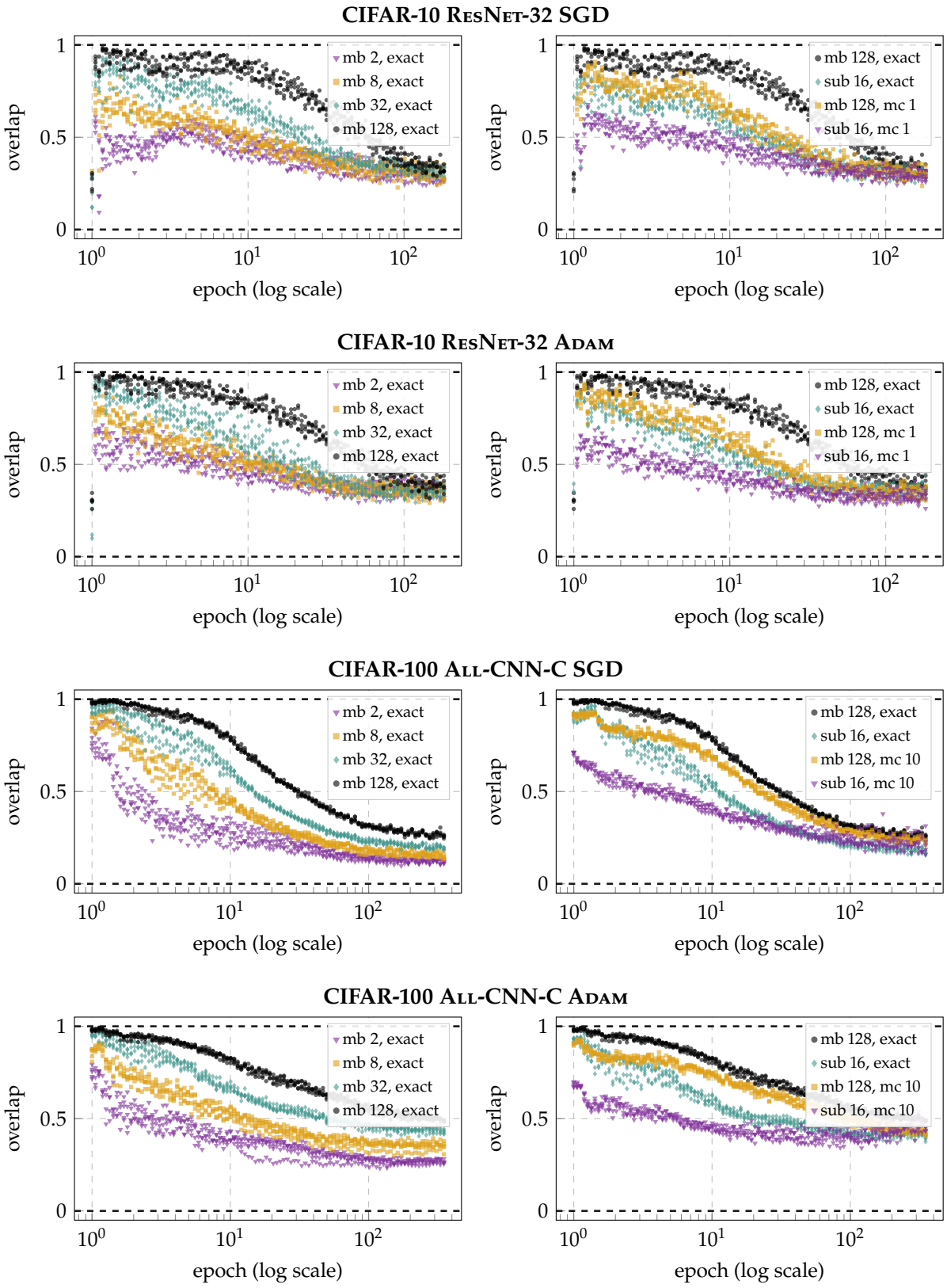


Figure B.19: ViViT vs. Full-Batch GGN (2). Overlap between the top-C eigenspaces of different GGN approximations and the full-batch GGN during training for all test problems. Each approximation is evaluated on 5 different mini-batches.

Table B.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

the CIFAR-100 All-CNN-C test problem, the MC approximation stands out particularly early from the other approximations and consistently yields higher overlaps with the mini-batch GGN.

B.2.5 Curvature Under Noise and Approximations

GGN and Hessian are predominantly used to locally approximate the loss by a quadratic model q (see Equation (6.6)). 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 B.5 gives an overview over the cases considered in this experiment.

Due to the large computational effort needed for the evaluation of full-batch directional derivatives, a subset of the checkpoints from Appendix B.2.3 is used for two test 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 $\{\boldsymbol{u}_k\}_{k=1}^{C}$ of the GGN approximation $\boldsymbol{G}^{(\text{ap})}$ either with ViViT or using an iterative matrix-free approach (as in Appendix B.2.4). The second-order directional derivative of the corresponding quadratic model along direction $\boldsymbol{u}_k$ is then given by $\lambda_k^{(\text{ap})} = \boldsymbol{u}_k^\top \boldsymbol{G}^{(\text{ap})} \boldsymbol{u}_k$ (see Equation (6.7)). As a reference, we compute the full-batch GGN $\boldsymbol{G}^{(\text{fb})}$ and the resulting directional derivatives along the same eigenvectors $\{\boldsymbol{u}_k\}_{k=1}^{C}$, i.e. $\lambda_k^{(\text{fb})} = \boldsymbol{u}_k^\top \boldsymbol{G}^{(\text{fb})} \boldsymbol{u}_k$. The average (over all

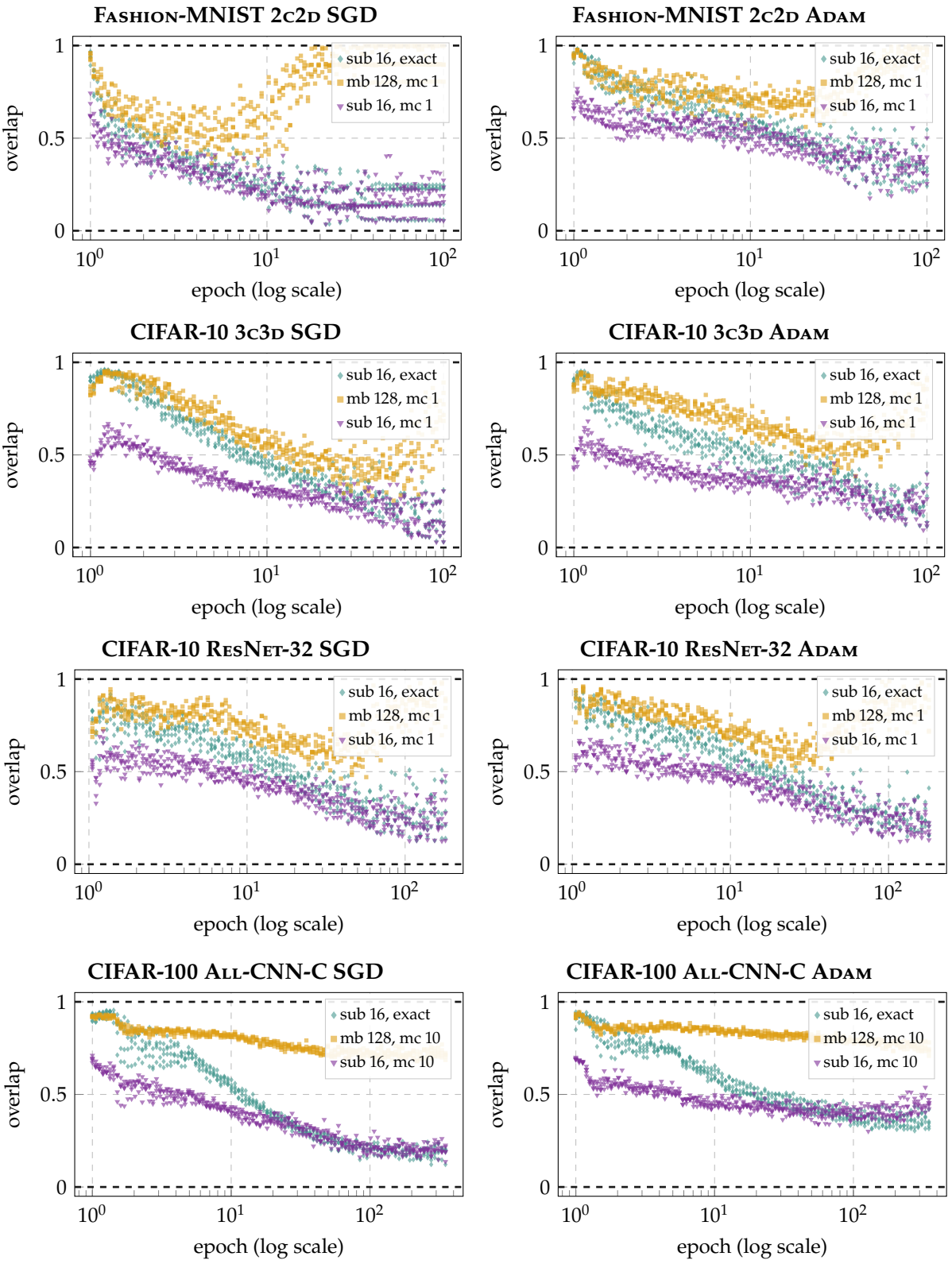


Figure B.20: ViViT vs. Mini-Batch GGN. Overlap between the top-C eigenspaces of different GGN approximations and the mini-batch GGN during training for all test problems. Each approximation is evaluated on 5 different mini-batches.

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)—except for the CIFAR-100 All-CNN-C test problem, where we perform only a single run to keep the computational effort manageable.

Results. The results can be found in Figure B.21. We observe similar results as in Appendix B.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.

B.2.6 Directional Derivatives

Procedure. We use the checkpoints from Appendix B.2.3. For every checkpoint, we compute the top-C eigenvectors of the mini-batch GGN (with a mini-batch size of $N = 128$) 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 (see Equation (6.6)) are given by Equation (6.8).

We use these directional derivatives $\{\gamma_{nk}\}_{n=1,k=1}^{N,C}$, $\{\lambda_{nk}\}_{n=1,k=1}^{N,C}$ to compute signal-to-noise ratios (SNRs) along the top-C eigenvectors. The curvature SNR along direction $\boldsymbol{u}_k$ is given by the squared sample mean divided by the empirical variance of the samples $\{\lambda_{nk}\}_{n=1}^{N}$, i.e.

$$\text{SNR} = \frac{\lambda_k^2}{1/N-1 \sum_{n=1}^{N} (\lambda_{nk} - \lambda_k)^2} \quad \text{where} \quad \lambda_k = \frac{1}{N} \sum_{n=1}^{N} \lambda_{nk} .$$

(and similarly for $\{\gamma_{nk}\}_{n=1}^{N}$).

Results. The results can be found in Figure B.22 and B.23. These plots show the SNRs in C distinct colors that are generated by linearly interpolating 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.

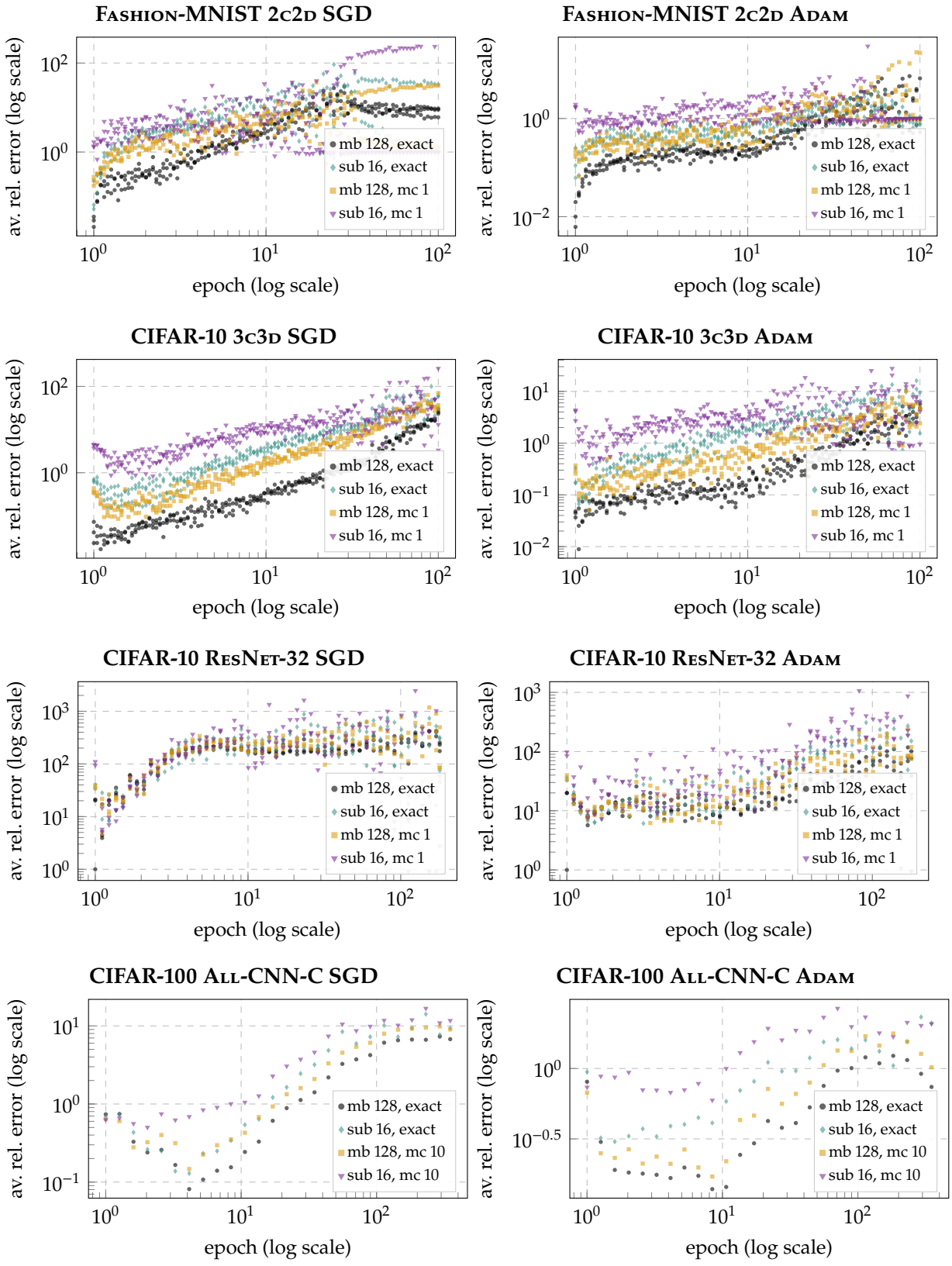


Figure B.21: ViViT's vs. Full-Batch Quadratic Model. Comparison between approximations to the quadratic model and the full-batch model in terms of the average relative error for the directional curvature during training for all test problems.

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.

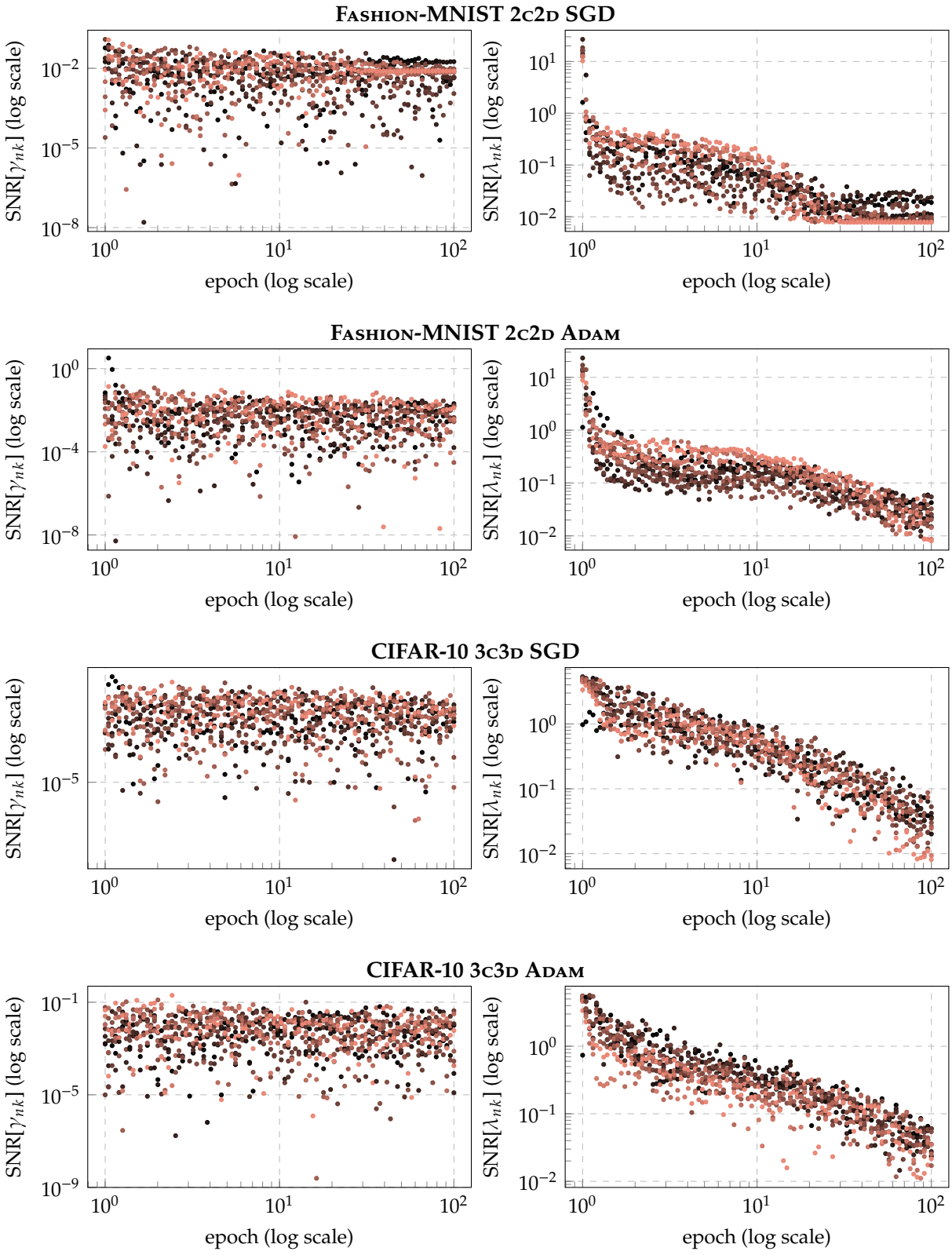


Figure B.22: Directional SNRs (1). SNR along each of the mini-batch GGN's top-C eigenvectors during training for all test problems. At fixed epoch, the SNR for the most curved direction is shown in ● and for the least curved direction in ●.

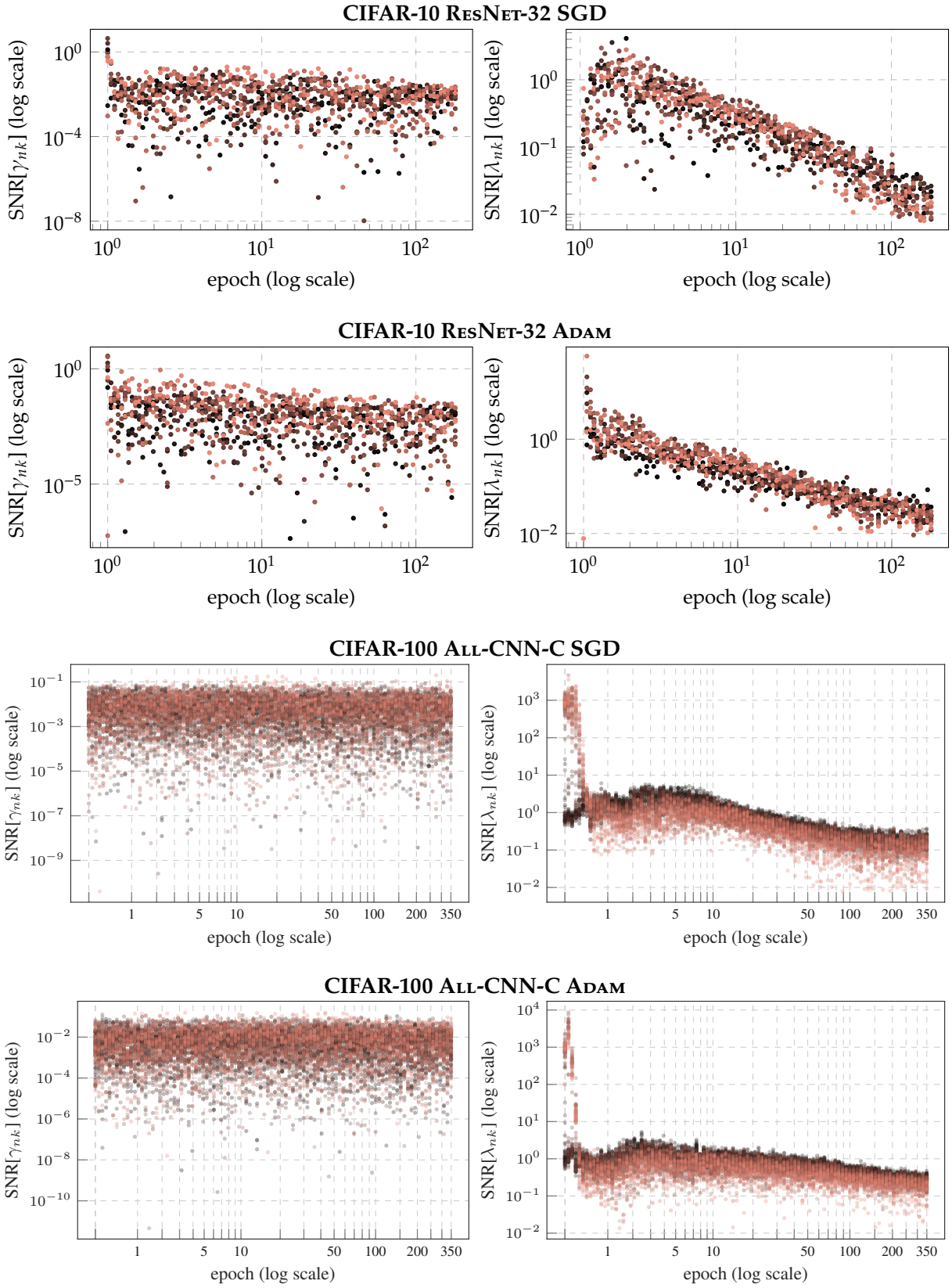


Figure B.23: Directional SNRs (2). SNR along each of the mini-batch GGN's top-C eigenvectors during training for all test problems. At fixed epoch, the SNR for the most curved direction is shown in ● and for the least curved direction in ●.

B.3 Implementation Details

Layer View of Backpropagation. Consider a single layer $T^{(l)}_{\boldsymbol{\theta}^{(l)}}$ that transforms inputs $\boldsymbol{z}_n^{(l-1)} \in \mathbb{R}^{h^{(l-1)}}$ into outputs $\boldsymbol{z}_n^{(l)} \in \mathbb{R}^{h^{(l)}}$ by means of a parameter $\boldsymbol{\theta}^{(l)} \in \mathbb{R}^{P^{(l)}}$. During backpropagation for $\boldsymbol{V}$, the layer receives vectors $\boldsymbol{s}_{nc}^{(l)} = (\mathrm{J}_{\boldsymbol{z}_n^{(l)}} f_n)^\top \boldsymbol{s}_{nc}$ from the previous stage (recall $\nabla^2_f \ell_n = \sum_{c=1}^{C} \boldsymbol{s}_{nc}\boldsymbol{s}_{nc}^\top$). Parameter contributions $\boldsymbol{v}_{nc}^{(l)}$ to $\boldsymbol{V}$ are obtained by application of its Jacobian,

$$\begin{aligned}
\boldsymbol{v}_{nc}^{(l)} &= \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} f_n\right)^\top \boldsymbol{s}_{nc} \\
&= \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right)^\top \left(\mathrm{J}_{\boldsymbol{z}_n^{(l)}} f_n\right)^\top \boldsymbol{s}_{nc} && \text{(Chain rule)} \\
&= \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right)^\top \boldsymbol{s}_{nc}^{(l)} . && \text{(Definition of } \boldsymbol{s}_{nc}^{(l)}\text{)} \qquad \text{(B.2)}
\end{aligned}$$

Consequently, the contribution of $\boldsymbol{\theta}^{(l)}$ to $\boldsymbol{V}$, denoted by $\boldsymbol{V}^{(l)} \in \mathbb{R}^{P^{(l)} \times NC}$, is

$$\boldsymbol{V}^{(l)} = \frac{1}{\sqrt{N}} \begin{pmatrix} \boldsymbol{v}_{11}^{(l)} & \boldsymbol{v}_{12}^{(l)} & \dots & \boldsymbol{v}_{NC}^{(l)} \end{pmatrix} \qquad \text{(B.3)}$$

with $\boldsymbol{v}_{nc}^{(l)} = (\mathrm{J}_{\boldsymbol{\theta}^{(l)}} f_n)^\top \boldsymbol{s}_{nc}$.

B.3.1 Optimized Gram Matrix Computation for Linear Layers

Our goal is to efficiently extract $\boldsymbol{\theta}^{(l)}$'s contribution to the Gram matrix $\tilde{\boldsymbol{G}}$, given by

$$\tilde{\boldsymbol{G}}^{(l)} = {\boldsymbol{V}^{(l)}}^\top \boldsymbol{V}^{(l)} \in \mathbb{R}^{NC \times NC} . \qquad \text{(B.4)}$$

Gram Matrix via Expanding $\boldsymbol{V}^{(l)}$. One way to construct $\boldsymbol{G}^{(l)}$ is to first expand $\boldsymbol{V}^{(l)}$ (Equation (B.3)) via the Jacobian $\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}$, then contract it (Equation (B.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 $T^{(l)}_{\boldsymbol{\theta}^{(l)}}$ be a linear layer with weights $\boldsymbol{W}^{(l)} \in \mathbb{R}^{h^{(l)} \times h^{(l-1)}}$, *i.e.* $\boldsymbol{\theta}^{(l)} = \mathrm{vec}(\boldsymbol{W}^{(l)}) \in \mathbb{R}^{P^{(l)} = h^{(l)} h^{(l-1)}}$ with column stacking convention for vectorization,

$$T^{(l)}_{\boldsymbol{\theta}^{(l)}} : \quad \boldsymbol{z}_n^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{z}_n^{(l-1)} .$$

The Jacobian is

$$\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)} = {\boldsymbol{z}_n^{(l-1)}}^\top \otimes \boldsymbol{I}_{h^{(l)}} \,. \tag{B.5}$$

Its structure can be used to directly compute entries of the Gram matrix without expanding $\boldsymbol{V}^{(l)}$,

$$\begin{aligned}
&\left[\tilde{\boldsymbol{G}}^{(l)}\right]_{(nc)(n'c')} \\
&= {\boldsymbol{v}_{nc}^{(l)}}^\top \boldsymbol{v}_{n'c'}^{(l)} && \text{(Equation (B.4))} \\
&= {\boldsymbol{s}_{nc}^{(l)}}^\top \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_n^{(l)}\right) \left(\mathrm{J}_{\boldsymbol{\theta}^{(l)}} \boldsymbol{z}_{n'}^{(l)}\right)^\top \boldsymbol{s}_{n'c'}^{(l)} \\
&= {\boldsymbol{s}_{nc}^{(l)}}^\top \left({\boldsymbol{z}_n^{(l-1)}}^\top \otimes \boldsymbol{I}_{h^{(l)}}\right) \left({\boldsymbol{z}_{n'}^{(l-1)}}^\top \otimes \boldsymbol{I}_{h^{(l)}}\right)^\top \boldsymbol{s}_{n'c'}^{(l)} && \text{(Equation (B.5))} \\
&= {\boldsymbol{s}_{nc}^{(l)}}^\top \left({\boldsymbol{z}_n^{(l-1)}}^\top \boldsymbol{z}_{n'}^{(l-1)} \otimes \boldsymbol{I}_{h^{(l)}}\right) \boldsymbol{s}_{n'c'}^{(l)} && \text{(Equation (B.2))} \\
&= {\boldsymbol{z}_n^{(l-1)}}^\top \boldsymbol{z}_{n'}^{(l-1)} {\boldsymbol{s}_{nc}^{(l)}}^\top \boldsymbol{I}_{h^{(l)}} \boldsymbol{s}_{n'c'}^{(l)} && ({\boldsymbol{z}_n^{(l-1)}}^\top \boldsymbol{z}_{n'}^{(l-1)} \in \mathbb{R}) \\
&= \left({\boldsymbol{z}_n^{(l-1)}}^\top \boldsymbol{z}_{n'}^{(l-1)}\right) \left({\boldsymbol{s}_{nc}^{(l)}}^\top \boldsymbol{s}_{n'c'}^{(l)}\right) \,.
\end{aligned}$$

We see that the Gram matrix is built from two Gram matrices based on $\{\boldsymbol{z}_n^{(l-1)}\}_{n=1}^N$ and $\{\boldsymbol{s}_{nc}^{(l)}\}_{n=1,c=1}^{N,C}$, that require $\mathcal{O}(N^2)$ and $\mathcal{O}((NC)^2)$ memory, respectively. In comparison, the naïve approach via $\boldsymbol{V}^{(l)} \in \mathbb{R}^{P^{(l)} \times NC}$ scales with the number of weights, which is often comparable to P. For instance, the 3c3d architecture on CIFAR-10 has $P = 895{,}210$ and the largest weight matrix has $P^{(l)} = 589{,}824$, whereas $NC = 1{,}280$ during training [123].

Run Time Comparison. To evaluate the efficiency of our optimization for linear layers, we consider a multi-layer perceptron with $1024 \to 512 \to 256 \to 128 \to 64 \to 32 \to C = 10$ units, each activated by ReLU except for the last, and mean squared error as loss function. We set the batch size to $N = 128$ and randomly generate synthetic inputs and labels. The following tasks are considered (results are for GPU, reporting the smallest run time over 20 repetitions):

- T1: Mini-batch gradient computation: $t_{\text{grad}} \approx 0.774\,\text{ms}$ (1.0 x)
- T2: Naive computation of *explicit* $\boldsymbol{V}$: $t_{\boldsymbol{V}} \approx 9.25\,\text{ms}$ (11.9 x)
- T3: Naive computation of the Gram matrix $\tilde{\boldsymbol{G}}$ via *explicit* $\boldsymbol{V}$: $t_{\boldsymbol{V},\tilde{\boldsymbol{G}}} \approx 233\,\text{ms}$ (301 x)
- T4: Optimized computation of $\tilde{\boldsymbol{G}}$ via *implicit* $\boldsymbol{V}$: $t_{\boldsymbol{V},\tilde{\boldsymbol{G}},\text{opt}} \approx 4.28\,\text{ms}$ (5.5 x)

The main finding of this comparison is that, for this setting, **our optimized approach (T4) computes the Gram matrix >50 x faster than the naive approach (T3), and requires less than C gradient computations (T1)**. Detailed description:

- **Naive Computations of V (T2) and $\tilde{G}$ (T3) Are Slow.** From

the complexity analysis in Section 6.2.3, we expect $t_V \approx C t_{\text{grad}}$. Empirically, we observe a factor of 11.9. Despite the use of a GPU, performance does not improve. We believe this may be because the operation requires allocating and writing large amounts of data into memory for $\boldsymbol{V}$. Naively expanding this *explicit* $\boldsymbol{V}$, then computing $\tilde{\boldsymbol{G}}$ (T3), would cost $t_{V,G}/t_{\text{grad}} \approx 301$ gradients.

- **Our Optimized Approach to *implicitly* Work with $\boldsymbol{V}$ in Combination with Exploiting Structure in the Jacobian (T4) Is Fast.** With our optimizations, we observe $t_{V,\tilde{G},\text{opt}}/t_{\text{grad}} \approx 5.5$. This is a $> 50\,\text{x}$ speed-up over the naive approach (T3)! It also requires less than C gradient computations, i.e. almost 50% of the expected overhead is compensated by parallelism on the GPU.

In our work, we first extended the BackPACK package by the naive operations T2 and T3 described above. These are already superior to for-loop based approaches with PyTorch's built-in automatic differentiation because BackPACK implements vectorized vector-Jacobian products. On top of that, we then further *significantly* improved performance through structural tricks (T4), as shown above.

B.3.2 Implicit Multiplication with the Inverse (Block-Diagonal) GGN

Inverse GGN-Vector Products. A damped Newton step requires multiplication by $(\boldsymbol{G} + \delta \boldsymbol{I}_P)^{-1}$.[1] By means of Equation (6.3) and the matrix inversion lemma,

1: $\delta \boldsymbol{I}_P$ can be replaced by other easy-to-invert matrices.

$$
\begin{aligned}
&(\delta \boldsymbol{I}_P + \boldsymbol{G})^{-1} \\
&\quad = \left(\delta \boldsymbol{I}_P + \boldsymbol{V}\boldsymbol{V}^\top\right)^{-1} && \text{(Equation (6.3))} \qquad \text{(B.6)} \\
&\quad = \frac{1}{\delta}\left(\boldsymbol{I}_P + \frac{1}{\delta}\boldsymbol{V}\boldsymbol{V}^\top\right)^{-1} \\
&\quad = \frac{1}{\delta}\left[\boldsymbol{I}_P - \frac{1}{\delta}\boldsymbol{V}\left(\boldsymbol{I}_{NC} + \boldsymbol{V}^\top\frac{1}{\delta}\boldsymbol{V}\right)^{-1}\boldsymbol{V}^\top\right] && \text{(Matrix inversion lemma)} \\
&\quad = \frac{1}{\delta}\left[\boldsymbol{I}_P - \boldsymbol{V}\left(\delta \boldsymbol{I}_{NC} + \boldsymbol{V}^\top\boldsymbol{V}\right)^{-1}\boldsymbol{V}^\top\right] && \text{(Gram matrix)} \\
&\quad = \frac{1}{\delta}\left[\boldsymbol{I}_P - \boldsymbol{V}\left(\delta \boldsymbol{I}_{NC} + \tilde{\boldsymbol{G}}\right)^{-1}\boldsymbol{V}^\top\right]. && \text{(B.7)}
\end{aligned}
$$

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 $\boldsymbol{G} \approx \boldsymbol{G}_{\text{BDA}} =$

$\mathrm{diag}(\boldsymbol{G}^{(1)}, \boldsymbol{G}^{(2)}, \dots)$ with

$$\boldsymbol{G}^{(l)} = \boldsymbol{V}^{(l)} \boldsymbol{V}^{(l)\top} \in \mathbb{R}^{P^{(l)} \times P^{(l)}}$$

and $\boldsymbol{V}^{(l)}$ as in Equation (B.3). Then, inverse multiplication reduces to each block,

$$\boldsymbol{G}_{\mathrm{BDA}}^{-1} = \mathrm{diag}\left(\boldsymbol{G}^{(1)^{-1}}, \boldsymbol{G}^{(2)^{-1}}, \dots\right).$$

If again a damped Newton step is considered, we can reuse Equation (B.7) with the substitutions

$$\left(\boldsymbol{G}, P, \boldsymbol{V}, \boldsymbol{V}^\top, \tilde{\boldsymbol{G}}\right) \leftrightarrow \left(\boldsymbol{G}^{(l)}, P^{(l)}, \boldsymbol{V}^{(l)}, \boldsymbol{V}^{(l)\top}, \tilde{\boldsymbol{G}}^{(l)}\right)$$

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 (B.3).
2. Compute $\tilde{\boldsymbol{G}}^{(l)}$ using Equation (B.4).
3. Compute $\left(\delta \boldsymbol{I}_{NC} + \tilde{\boldsymbol{G}}^{(l)}\right)^{-1}$.
4. Apply the inverse in Equation (B.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 $l-1$.

Note that the above scheme should only be used for parameters that satisfy $P^{(l)} > NC$, *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 $\boldsymbol{G}_{\mathrm{BDA}}$.

Additional Material for Chapter 7 C

Below, we provide additional details on the mathematical derivations, describe the experimental setup and present additional results.

C.1 Mathematical Details

C.1.1 Directional Derivatives of a Quadratic

We claim in Section 7.2.1 that a cut r through the quadratic $q(\,\cdot\,;\mathcal{B})$ from $\boldsymbol{\theta}_\bullet \in \Theta \subseteq \mathbb{R}^P$ along the normalized direction $\boldsymbol{d}$ can be written as

$$\begin{aligned} r(\tau) &\coloneqq q(\boldsymbol{\theta}_\bullet + \tau\boldsymbol{d};\mathcal{B}) \\ &= \frac{1}{2}\tau^2 \boldsymbol{d}^\top \nabla^2 q(\boldsymbol{\theta}_\bullet;\mathcal{B})\,\boldsymbol{d} + \tau \boldsymbol{d}^\top \nabla q(\boldsymbol{\theta}_\bullet;\mathcal{B}) + \text{const.} \end{aligned} \tag{C.1}$$

Proof for Equation (C.1). Here, we provide the derivation for Equation (C.1). Let $\boldsymbol{\theta}_\bullet \in \Theta \subseteq \mathbb{R}^P$ be a point in parameter space and $\boldsymbol{d} \in \mathbb{R}^P$ a normalized direction, *i.e.* $\|\boldsymbol{d}\| = 1$. We consider the quadratic approximation $q(\boldsymbol{\theta};\mathcal{B})$ around $\boldsymbol{\theta}_0$ and evaluate it along the cut $\boldsymbol{\theta}_\bullet + \tau\boldsymbol{d}$ for $\tau \in \mathbb{R}$. We assume that the Hessian (or its approximation) $\boldsymbol{H}_\mathcal{B}$ is symmetric and obtain

$$\begin{aligned} r(\tau) &\coloneqq q(\boldsymbol{\theta}_\bullet + \tau\boldsymbol{d};\mathcal{B}) \\ &= \frac{1}{2}(\boldsymbol{\theta}_\bullet + \tau\boldsymbol{d} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}_\mathcal{B}(\boldsymbol{\theta}_\bullet + \tau\boldsymbol{d} - \boldsymbol{\theta}_0) \\ &\quad + (\boldsymbol{\theta}_\bullet + \tau\boldsymbol{d} - \boldsymbol{\theta}_0)^\top \boldsymbol{g}_\mathcal{B} + c_\mathcal{B} \\ &= \frac{1}{2}\tau^2 \boldsymbol{d}^\top \boldsymbol{H}_\mathcal{B}\boldsymbol{d} + \tau\boldsymbol{d}^\top \boldsymbol{H}_\mathcal{B}(\boldsymbol{\theta}_\bullet - \boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta}_\bullet - \boldsymbol{\theta}_0)^\top \boldsymbol{H}_\mathcal{B}(\boldsymbol{\theta}_\bullet - \boldsymbol{\theta}_0) \\ &\quad + \tau\boldsymbol{d}^\top \boldsymbol{g}_\mathcal{B} + (\boldsymbol{\theta}_\bullet - \boldsymbol{\theta}_0)^\top \boldsymbol{g}_\mathcal{B} + c_\mathcal{B} \\ &= \frac{1}{2}\tau^2 \boldsymbol{d}^\top \boldsymbol{H}_\mathcal{B}\boldsymbol{d} + \tau\boldsymbol{d}^\top(\boldsymbol{H}_\mathcal{B}(\boldsymbol{\theta}_\bullet - \boldsymbol{\theta}_0) + \boldsymbol{g}_\mathcal{B}) + \text{const.} \end{aligned}$$

Since $\nabla q(\boldsymbol{\theta}_\bullet;\mathcal{B}) = \boldsymbol{H}_\mathcal{B}(\boldsymbol{\theta}_\bullet - \boldsymbol{\theta}_0) + \boldsymbol{g}_\mathcal{B}$ and $\nabla^2 q(\boldsymbol{\theta}_\bullet;\mathcal{B}) \equiv \boldsymbol{H}_\mathcal{B}$, we arrive at Equation (C.1). □

Proof for Equation (7.5). Next, we show that the average directional slope/curvature over all mini-batches in the training set coincides with the directional slope/curvature of the full-batch quadratic. Let's assume that all M mini-batches $\{\mathcal{B}_{m'}\}_{m'=1}^M$ are disjoint, have

the same size $|\mathcal{B}_1|$ and that their union is the training set, *i.e.*

$$|\mathcal{B}_{m'}| = |\mathcal{B}_1| \;\; \forall m' \in \{1, \dots, M\} \quad \text{and} \quad \bigcup_{m'=1}^{M} \mathcal{B}_{m'} = \mathcal{D}. \tag{C.2}$$

This implies $M|\mathcal{B}_1| = |\mathcal{D}|$. It holds

$$\begin{aligned}
&\frac{1}{M} \sum_{m'=1}^{M} \underbrace{\partial_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})}_{\text{one ● and many ●}} \\
&= \frac{1}{M} \sum_{m'=1}^{M} \boldsymbol{u}_p^\top \nabla q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})
\end{aligned}$$

with $\nabla q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'}) = \boldsymbol{g}_{\mathcal{B}_{m'}}$ since $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}_\star$

$$\begin{aligned}
&= \boldsymbol{u}_p^\top \frac{1}{M} \sum_{m'=1}^{M} \nabla \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_\star; \mathcal{B}_{m'}) \\
&= \boldsymbol{u}_p^\top \left(\frac{1}{M} M \nabla r(\boldsymbol{\theta}_\star) + \frac{1}{M} \sum_{m'=1}^{M} \nabla \mathcal{L}(\boldsymbol{\theta}_\star; \mathcal{B}_{m'}) \right) \\
&= \boldsymbol{u}_p^\top \left(\nabla r(\boldsymbol{\theta}_\star) + \frac{1}{M} \sum_{m'=1}^{M} \frac{1}{|\mathcal{B}_{m'}|} \sum_{n \in \mathcal{B}_{m'}} \nabla \ell(f_{\boldsymbol{\theta}_\star}(\boldsymbol{x}_n), \boldsymbol{y}_n) \right) \\
&\overset{(C.2)}{=} \boldsymbol{u}_p^\top \left(\nabla r(\boldsymbol{\theta}_\star) + \frac{1}{M|\mathcal{B}_1|} \sum_{m'=1}^{M} \sum_{n \in \mathcal{B}_{m'}} \nabla \ell(f_{\boldsymbol{\theta}_\star}(\boldsymbol{x}_n), \boldsymbol{y}_n) \right) \\
&= \boldsymbol{u}_p^\top \nabla \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_\star; \mathcal{D}) \\
&= \underbrace{\partial_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{D})}_{\text{✚}}.
\end{aligned}$$

As the mini-batch curvature $\nabla^2 \mathcal{L}(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})$ is also an *average* over the samples in the mini-batch (this applies to both the actual Hessian and its GGN approximation), the same argument holds for the associated directional curvatures. For K-FAC, the derivation above does *not* hold as $(1/2)(\boldsymbol{K}_{\mathcal{B}} + \boldsymbol{K}_{\tilde{\mathcal{B}}}) \neq \boldsymbol{K}_{\mathcal{B} \cup \tilde{\mathcal{B}}}$. □

C.1.2 The Method of Conjugate Gradients (CG)

Using CG for Minimizing a Quadratic. The argmin of a quadratic $q(\cdot; \mathcal{B})$ is given by $\boldsymbol{\theta}_0 - \boldsymbol{H}_{\mathcal{B}}^{-1} \boldsymbol{g}_{\mathcal{B}}$ (assuming that $\boldsymbol{H}_{\mathcal{B}}$ is symmetric and positive definite), see Section 7.2.2. We can use the method of conjugate gradients (see Algorithm C.1) for computing the Newton step (or an approximation thereof) by setting $\boldsymbol{A} \leftarrow \boldsymbol{H}_{\mathcal{B}}$ and $\boldsymbol{b} \leftarrow -\boldsymbol{g}_{\mathcal{B}}$.

Properties of CG & Geometric Interpretations. In the following, we provide some important properties and a geometric interpretation of the quantities involved in the CG method.

Algorithm C.1: Method of Conjugate Gradients (CG), Based on [102, Alg. 5.2].

Input: Access to matrix-vector products $\boldsymbol{v} \mapsto \boldsymbol{A}\boldsymbol{v}$, where $\boldsymbol{A} \in \mathbb{R}^{P \times P}$ is symmetric and positive definite, right-hand side $\boldsymbol{b} \in \mathbb{R}^P$, convergence tolerance $\epsilon \in \mathbb{R}_{>0}$, maximum number of iterations $P_{\max} \in \mathbb{N}$, with $P_{\max} \leq P$

Output: Approximate solution $\boldsymbol{x}_p$ to the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$

Procedure: CG($\boldsymbol{A}, \boldsymbol{b}, \epsilon, P_{\max}$)

Initialize $\boldsymbol{x}_0 = \boldsymbol{0}$, $\boldsymbol{r}_0 \leftarrow -\boldsymbol{b}$, and $\boldsymbol{s}_0 \leftarrow -\boldsymbol{r}_0$
for $p = 0, 1, \ldots, P_{max}$ **do**
  if $p = P_{max}$ ***or*** $\|\boldsymbol{r}_p\|_2 \leq \epsilon$ **then** — Termination criteria
    return (approximate) solution $\boldsymbol{x}_p$
  $\boldsymbol{t}_p \leftarrow \boldsymbol{A}\boldsymbol{s}_p$ — Compute the matrix-vector product only once
  $\alpha_p \leftarrow \dfrac{\boldsymbol{r}_p^\top \boldsymbol{r}_p}{\boldsymbol{s}_p^\top \boldsymbol{t}_p}$ — Update magnitude $\alpha_p = \dfrac{\boldsymbol{r}_p^\top \boldsymbol{r}_p}{\boldsymbol{s}_p^\top \boldsymbol{A}\boldsymbol{s}_p}$
  Update $\boldsymbol{x}_{p+1} \leftarrow \boldsymbol{x}_p + \alpha_p \boldsymbol{s}_p$ — Update along search direction $\boldsymbol{s}_p$
  Update residual $\boldsymbol{r}_{p+1} \leftarrow \boldsymbol{r}_p + \alpha_p \boldsymbol{t}_p$ — Residual $\boldsymbol{r}_p = \boldsymbol{A}\boldsymbol{x}_p - \boldsymbol{b}$ computed via recursion
  $\beta_{p+1} \leftarrow \dfrac{\boldsymbol{r}_{p+1}^\top \boldsymbol{r}_{p+1}}{\boldsymbol{r}_p^\top \boldsymbol{r}_p}$
  $\boldsymbol{s}_{p+1} \leftarrow -\boldsymbol{r}_{p+1} + \beta_{p+1} \boldsymbol{s}_p$ — Construction of new (conjugate) search direction

- **Residual $\boldsymbol{r}_p$.** Let $\boldsymbol{x}_p := \boldsymbol{\theta}_p - \boldsymbol{\theta}_0$. The residual $\boldsymbol{r}_p$ can be written as $\boldsymbol{r}_p = \boldsymbol{A}\boldsymbol{x}_p - b = \boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}_p - \boldsymbol{\theta}_0) + \boldsymbol{g}_{\mathcal{B}} = \nabla q(\boldsymbol{\theta}_p; \mathcal{B})$, *i.e.* it coincides with the gradient of the quadratic at $\boldsymbol{\theta}_p$.
- **Update magnitude α_p.** Consider a cut through the quadratic from $\boldsymbol{\theta}_p$ into the direction $\boldsymbol{s}_p$, *i.e.* $r(\tau) = q(\boldsymbol{\theta}_p + \tau \boldsymbol{s}_p; \mathcal{B})$. Minimizing this 1D quadratic requires its first derivative to vanish. It holds (see Equation (C.1))

$$
\begin{aligned}
r'(\tau) = 0 \quad &\Leftrightarrow \quad \tau \boldsymbol{s}_p^\top \nabla^2 q(\boldsymbol{\theta}_p; \mathcal{B})\, \boldsymbol{s}_p + \boldsymbol{s}_p^\top \nabla q(\boldsymbol{\theta}_p; \mathcal{B}) = 0 \\
&\Leftrightarrow \quad \tau = -\frac{\boldsymbol{s}_p^\top \nabla q(\boldsymbol{\theta}_p; \mathcal{B})}{\boldsymbol{s}_p^\top \nabla^2 q(\boldsymbol{\theta}_p; \mathcal{B})\, \boldsymbol{d}_p} \\
&\qquad = -\frac{\boldsymbol{s}_p^\top \boldsymbol{r}_p}{\boldsymbol{s}_p^\top \boldsymbol{A}\, \boldsymbol{s}_p} \\
&\qquad = \frac{\boldsymbol{r}_p^\top \boldsymbol{r}_p}{\boldsymbol{s}_p^\top \boldsymbol{A}\boldsymbol{s}_p} \\
&\qquad = \alpha_p,
\end{aligned}
$$

where the equality $-\boldsymbol{s}_p^\top \boldsymbol{r}_p = \boldsymbol{r}_p^\top \boldsymbol{r}_p$ is due to Equation (5.14a) and Theorem 5.2 in [102]. So, the update magnitude α_p is chosen such that it minimizes the quadratic along the direction $\boldsymbol{s}_p$. Note that the update magnitude can also be written as a ratio of the negative directional slope and directional curvature at $\boldsymbol{\theta}_p$, *i.e.* a 1D directional Newton step. Let $\boldsymbol{d}_p := \boldsymbol{s}_p / \|\boldsymbol{s}_p\|$ denote the *normalized* search direction. It

holds

$$\begin{aligned}
\boldsymbol{x}_{p+1} &= \boldsymbol{x}_p + \alpha_p \boldsymbol{s}_p \\
&= \boldsymbol{x}_p - \frac{\boldsymbol{s}_p^\top \boldsymbol{r}_p}{\boldsymbol{s}_p^\top \boldsymbol{A}\, \boldsymbol{s}_p} \boldsymbol{s}_p \\
&= \boldsymbol{x}_p - \frac{\boldsymbol{d}_p^\top \boldsymbol{r}_p}{\boldsymbol{d}_p^\top \boldsymbol{A}\, \boldsymbol{d}_p} \boldsymbol{d}_p \\
&= \boldsymbol{x}_p - \underbrace{\frac{\partial_{\boldsymbol{d}_p} q(\boldsymbol{\theta}_d; \mathcal{B})}{\partial^2_{\boldsymbol{d}_p} q(\boldsymbol{\theta}_d; \mathcal{B})}}_{=:\, \tau_p} \boldsymbol{d}_p
\end{aligned}$$

i.e. $\tau_p = \alpha_p \|\boldsymbol{s}_p\|$. Via the shift $\boldsymbol{\theta}_p = \boldsymbol{\theta}_0 + \boldsymbol{x}_p$, we arrive at Equation (7.3).

- **Search Direction $\boldsymbol{s}_p$.** CG constructs the search directions to be conjugate, *i.e.* $\boldsymbol{s}_{p+1}^\top \boldsymbol{A}\, \boldsymbol{s}_i = 0$ for all $i \leq p$. Note that this property also applies to the normalized search directions.

Efficient Implementation of the Debiased CG Approach. For the debiased CG version, we need to re-evaluate the update magnitudes $\tilde{\tau}_p$ for a given set of search directions $\{\boldsymbol{d}_1, \dots \boldsymbol{d}_P\}$ on a second mini-batch $\tilde{\mathcal{B}}$ (see Equation (7.8)), *i.e.*

$$\begin{aligned}
\tilde{\tau}_p &= -\frac{\partial_{\boldsymbol{d}_p} q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}})}{\partial^2_{\boldsymbol{d}_p} q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}})} \\
&= -\frac{\boldsymbol{d}_p^\top \nabla q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}})}{\boldsymbol{d}_p^\top \nabla^2 q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}})\, \boldsymbol{d}_p} \\
&= -\frac{\boldsymbol{d}_p^\top (\boldsymbol{H}_{\tilde{\mathcal{B}}}(\tilde{\boldsymbol{\theta}}_p - \boldsymbol{\theta}_0) + \boldsymbol{g}_{\tilde{\mathcal{B}}})}{\boldsymbol{d}_p^\top \boldsymbol{H}_{\tilde{\mathcal{B}}}\, \boldsymbol{d}_p}
\end{aligned}$$

Implemented naively, this requires *two* matrix-vector products—one for the numerator and one for the denominator. However, we can use a recursive formula for the numerator:

$$\begin{aligned}
\nabla q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}}) &= \boldsymbol{H}_{\tilde{\mathcal{B}}}(\tilde{\boldsymbol{\theta}}_p - \boldsymbol{\theta}_0) + \boldsymbol{g}_{\tilde{\mathcal{B}}} \\
&= \boldsymbol{H}_{\tilde{\mathcal{B}}}(\tilde{\boldsymbol{\theta}}_{p-1} + \tilde{\tau}_{p-1} \boldsymbol{d}_{p-1} - \boldsymbol{\theta}_0) + \boldsymbol{g}_{\tilde{\mathcal{B}}} \\
&= \boldsymbol{H}_{\tilde{\mathcal{B}}}(\tilde{\boldsymbol{\theta}}_{p-1} - \boldsymbol{\theta}_0) + \boldsymbol{g}_{\tilde{\mathcal{B}}} + \tilde{\tau}_{p-1} \boldsymbol{H}_{\tilde{\mathcal{B}}}\, \boldsymbol{d}_{p-1} \\
&= \nabla q(\tilde{\boldsymbol{\theta}}_{p-1}; \tilde{\mathcal{B}}) + \tilde{\tau}_{p-1} \boldsymbol{H}_{\tilde{\mathcal{B}}}\, \boldsymbol{d}_{p-1}. \qquad \text{(C.3)}
\end{aligned}$$

So, if we store the gradient from the previous iteration $\nabla q(\tilde{\boldsymbol{\theta}}_{p-1}; \tilde{\mathcal{B}})$ and the Hessian vector product with the previous direction $\boldsymbol{H}_{\tilde{\mathcal{B}}}\, \boldsymbol{d}_{p-1}$, the current gradient can be computed *without* an additional matrix-vector product. At iteration p, we thus (i) compute the gradient $\nabla q(\tilde{\boldsymbol{\theta}}_p; \tilde{\mathcal{B}})$ recursively from the cached vectors $\nabla q(\tilde{\boldsymbol{\theta}}_{p-1}; \tilde{\mathcal{B}})$ and $\boldsymbol{H}_{\tilde{\mathcal{B}}}\, \boldsymbol{d}_{p-1}$ via Equation (C.3) (for $p = 0$, we have

$\nabla q(\tilde{\boldsymbol{\theta}}_0; \tilde{\mathcal{B}}) = \boldsymbol{g}_{\tilde{\mathcal{B}}}$), (ii) compute the Hessian-vector product with the current direction $\boldsymbol{H}_{\tilde{\mathcal{B}}} \boldsymbol{d}_p$, (iii) store both these vectors and (iv) compute the update magnitude $\tilde{\tau}_p$ (both the numerator and the denominator only require a simple dot product of two pre-computed vectors).

C.1.3 Laplace Approximation for Neural Networks

Derivation of the Laplace Approximation for Neural Networks

Preliminaries. The softmax function $\text{softmax} : \mathbb{R}^C \to \mathbb{R}^C$ is defined as

$$\text{softmax}(\boldsymbol{z}) \coloneqq \left(\frac{\exp(z_1)}{\sum_{c'=1}^{C} \exp(z_{c'})}, \dots, \frac{\exp(z_C)}{\sum_{c'=1}^{C} \exp(z_{c'})} \right).$$

It maps an arbitrary vector $\boldsymbol{z} \in \mathbb{R}^C$ to a vector whose entries are non-negative and sum up to one. So, the output of the softmax function can be interpreted as a probability distribution over C classes. With this, the cross-entropy loss for a single datum $(\boldsymbol{x}, \boldsymbol{y})$ is given by

$$\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) \coloneqq -\sum_{c=1}^{C} \boldsymbol{y}_c \cdot \log(\text{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}))_c). \tag{C.4}$$

Here, we assume that $\boldsymbol{y} \in \{0,1\}^C$ is a one-hot encoded vector representing the true class label. So, if $c_\star \in \{1, \dots, C\}$ is the correct class (*i.e.* $\boldsymbol{y}_{c_\star} = 1$), the cross-entropy loss is given by the negative logarithm of the probability the network assigns to this class $\ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) = -\log(\text{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}))_{c_\star})$. Finally, let

$$\text{Cat}(\boldsymbol{y}, \boldsymbol{p}) \coloneqq \prod_{c=1}^{C} \boldsymbol{p}_c^{\boldsymbol{y}_c} \tag{C.5}$$

denote the probability mass function of the categorical distribution, where $\boldsymbol{p} \in \mathbb{R}^C$ is a vector of probabilities (*i.e.* $\boldsymbol{p}_c \geq 0$ and $\sum_c \boldsymbol{p}_c = 1$), and $\boldsymbol{y} \in \{0,1\}^C$ is a one-hot encoded vector representing a class label.

Probabilistic Interpretation of the Regularized Loss. Recall from Equation (7.1) that the regularized loss function is given by

$$\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) = \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) + r(\boldsymbol{\theta}) \quad \text{with}$$
$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \frac{1}{N} \sum_{n \in \mathcal{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n).$$

We assume a classification problem that uses the cross-entropy loss (see Equation (C.4)) and an L^2 regularizer $r(\boldsymbol{\theta}) = \beta/2 \, \|\boldsymbol{\theta}\|_2^2$ with

parameter $\beta \in \mathbb{R}_{>0}$. We use one-hot encoded labels $\boldsymbol{y}_n \in \{0,1\}^C$ (with $\boldsymbol{y}_{nc}$, we denote the c-th entry of $\boldsymbol{y}_n$) and obtain

$$
\begin{aligned}
N \cdot \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) &= \sum_{n \in \mathcal{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n), \boldsymbol{y}_n) + \frac{N\beta}{2} \|\boldsymbol{\theta}\|_2^2 \\
&\overset{(C.4)}{=} - \sum_{n \in \mathcal{D}} \sum_{c=1}^{C} \boldsymbol{y}_{nc} \cdot \log(\operatorname{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))_c) \\
&\quad + \frac{N\beta}{2} \|\boldsymbol{\theta}\|_2^2 \\
&\overset{(C.5)}{=} - \sum_{n \in \mathcal{D}} \log\left(\operatorname{Cat}\left(\boldsymbol{y}_n, \operatorname{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\right)\right) \\
&\quad - \left(-\frac{1}{2} \boldsymbol{\theta}^\top (N\beta \cdot \boldsymbol{I})\, \boldsymbol{\theta}\right) \\
&= - \log \Bigg(\underbrace{\prod_{n \in \mathcal{D}} \operatorname{Cat}\left(\boldsymbol{y}_n, \operatorname{softmax}(f_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\right)}_{\text{likelihood } p(\mathbb{D} \mid \boldsymbol{\theta})} \Bigg) \\
&\quad - \log \Bigg(\underbrace{\mathcal{N}\left(\boldsymbol{\theta}; \boldsymbol{0}, \frac{1}{N\beta} \boldsymbol{I}\right)}_{\text{prior } p(\boldsymbol{\theta})} \Bigg) - Z,
\end{aligned}
$$

where $Z := {}^{P}/_{2} \log({}^{2\pi}/_{N\beta})$ absorbs the normalization constant of the Gaussian prior. The cross entropy loss $\mathcal{L}$ for the training set is thus connected to the negative log categorical likelihood and the regularizer can be seen as a negative log Gaussian prior over the parameters. Note that a similar derivation is possible for other loss functions as well, *e.g.* the MSE loss (which is equivalent to a negative log Gaussian likelihood).

The derivation above shows that the (rescaled) regularized loss function can be interpreted as the negative unnormalized log-posterior of a Bayesian model

$$
N \cdot \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) \overset{c}{=} - \log p(\mathbb{D} \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) \overset{c}{=} - \log p(\boldsymbol{\theta} \mid \mathbb{D}) \quad \text{(C.6)}
$$

with Gaussian prior and categorical likelihood. With $\overset{c}{=}$, we denote equality up to an additive constant.

Training as MAP Estimation. Equation (C.6) allows re-interpreting the training procedure of a neural network as a maximum a posteriori (MAP) estimation problem since minimizing the regularized loss

$$
\underbrace{\arg\min_{\boldsymbol{\theta} \in \Theta} \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}, \mathcal{D})}_{=: \boldsymbol{\theta}_\star} = \arg\min_{\boldsymbol{\theta} \in \Theta} N \cdot \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}, \mathcal{D})
$$

$$\overset{(C.6)}{=} \arg\max_{\boldsymbol{\theta}\in\Theta} \log p(\boldsymbol{\theta} \mid \mathbb{D})$$
$$= \arg\max_{\boldsymbol{\theta}\in\Theta} p(\boldsymbol{\theta} \mid \mathbb{D})$$

is equivalent to maximizing the posterior $p(\boldsymbol{\theta} \mid \mathbb{D})$.

Laplace Approximation. The idea of the Laplace approximation is to approximate the posterior distribution $p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{D})$ with a Gaussian at the mode $\boldsymbol{\theta}_\star$ of the posterior, *i.e.* after training the network. For this, we approximate the log-posterior by a second-order Taylor expansion around the mode $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}_\star$, *i.e.*

$$\begin{aligned}\log p(\boldsymbol{\theta} \mid \mathbb{D}) &\overset{c}{=} -N \cdot \mathcal{L}_{\text{reg}}(\boldsymbol{\theta}; \mathcal{D}) \\ &\overset{(7.2)}{\approx} -N \cdot q(\boldsymbol{\theta}; \mathcal{D}) \\ &\overset{c}{=} -\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_\star)^\top (N \cdot \boldsymbol{H}_\mathcal{D})(\boldsymbol{\theta} - \boldsymbol{\theta}_\star),\end{aligned} \tag{C.7}$$

where we assumed that $\boldsymbol{\theta}_\star$ is a (local) minimum of the regularized loss function (*i.e.* $\boldsymbol{g}_\mathcal{D} = \nabla\mathcal{L}_{\text{reg}}(\boldsymbol{\theta}_\star; \mathcal{D}) = 0$) such that the linear term $(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{g}_\mathcal{D}$ in the quadratic approximation vanishes. The additive constants we did not state explicitly in Equation (C.7) turn into some multiplicative factor (denoted by Z^{-1}) when taking the exponential

$$p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \frac{1}{Z} \exp\left(-\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_\star)^\top (N \cdot \boldsymbol{H}_\mathcal{D})(\boldsymbol{\theta} - \boldsymbol{\theta}_\star)\right).$$

This immediately identifies Z as the normalization constant of the Gaussian, and we obtain the Laplace approximation

$$p(\boldsymbol{\theta} \mid \mathbb{D}) \approx \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{D}) \quad \text{with} \quad \boldsymbol{\Sigma}_\mathcal{D} \coloneqq (N \cdot \boldsymbol{H}_\mathcal{D})^{-1} = \frac{1}{N}\boldsymbol{H}_\mathcal{D}^{-1}. \tag{C.8}$$

Mini-Batch Version of the Laplace Approximation. In order to reduce the computational cost of the Laplace approximation, we can replace the full-batch quadratic $q(\boldsymbol{\theta}; \mathcal{D})$ in Equation (C.7) by a mini-batch quadratic $q(\boldsymbol{\theta}; \mathcal{B})$, *i.e.* we obtain $\boldsymbol{\Sigma}_\mathcal{B} = N^{-1}\boldsymbol{H}_\mathcal{B}^{-1}$.

Predictive Uncertainty via the Laplace Approximation. Ultimately, we want to use the Laplace approximation to equip the prediction for an unknown test input $\boldsymbol{x}_\diamond$ with uncertainty. Ideally, we would compute the expectation of the model likelihood under the approximate posterior, *i.e.*

$$\begin{aligned}p(\boldsymbol{y}_\diamond \mid \boldsymbol{x}_\diamond, \mathbb{D}) &= \int p(\boldsymbol{y}_\diamond \mid \boldsymbol{x}_\diamond, \boldsymbol{\theta})\, p(\boldsymbol{\theta} \mid \mathbb{D})\, \mathrm{d}\boldsymbol{\theta} \\ &\overset{(C.8)}{\approx} \int \mathrm{Cat}\big(\boldsymbol{y}_\diamond, \mathrm{softmax}(f_\boldsymbol{\theta}(\boldsymbol{x}_\diamond))\big)\, \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{B})\, \mathrm{d}\boldsymbol{\theta}.\end{aligned}$$

One way to approximate this integral is via Monte Carlo sampling, *i.e.*

$$p(\boldsymbol{y}_\diamond \mid \boldsymbol{x}_\diamond, \mathbb{D}) \approx \frac{1}{S} \sum_{s=1}^{S} \mathrm{Cat}\left(\boldsymbol{y}_\diamond, \mathrm{softmax}(f_{\boldsymbol{\theta}^{(s)}}^{\mathrm{lin}}(\boldsymbol{x}_\diamond))\right), \tag{C.9}$$

where $\boldsymbol{\theta}^{(s)} \sim \mathcal{N}(\boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{B})$. As suggested by Immer et al. [66] and Roy et al. [117], we use the linearized network $f_{\boldsymbol{\theta}}^{\mathrm{lin}}(\boldsymbol{x}) \coloneqq f_{\boldsymbol{\theta}_\star}(\boldsymbol{x}) + \nabla f_{\boldsymbol{\theta}_\star}(\boldsymbol{x})(\boldsymbol{\theta} - \boldsymbol{\theta}_\star)$, where $\nabla f_{\boldsymbol{\theta}_\star}(\boldsymbol{x}) \in \mathbb{R}^{C \times P}$ is the network's Jacobian at $\boldsymbol{\theta}_\star$.

Sampling from the K-FAC Laplace Approximation

The Monte Carlo (MC) approach from Equation (C.9) requires samples from the Gaussian $\mathcal{N}(\boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{B})$ (the following derivations work exactly the same when the full-batch LA based on the entire training set is used). However, $\boldsymbol{\Sigma}_\mathcal{B} \in \mathbb{R}^{P \times P}$ is often too large to be built explicitly in memory, and it requires inverting the $P \times P$ Hessian $\boldsymbol{H}_\mathcal{B}$ of the regularized loss function. One approach for that issue is to use the K-FAC curvature approximation $\boldsymbol{H}_\mathcal{B} = \nabla^2 \mathcal{L}(\boldsymbol{\theta}; \mathcal{B}) + \beta \boldsymbol{I} \approx \boldsymbol{K}_\mathcal{B} + \beta \boldsymbol{I}$, *i.e.* $\boldsymbol{\Sigma}_\mathcal{B} \approx N^{-1}(\boldsymbol{K}_\mathcal{B} + \beta \boldsymbol{I})^{-1}$. As we will see in the following, the K-FAC approximation enables us to sample efficiently from the corresponding LA.

Leveraging K-FAC's Block-Diagonal Structure. K-FAC is a block-diagonal curvature approximation, *i.e.* $\boldsymbol{K}_\mathcal{B} = \mathrm{blockdiag}_{l=1,\dots,L}(\boldsymbol{K}_\mathcal{B}^{(l)})$ and $\boldsymbol{K}_\mathcal{B} + \beta \boldsymbol{I}$ inherits this structure. The inverse of a block-diagonal matrix is also block-diagonal with the block inverses on the diagonal, *i.e.*

$$\begin{aligned}
\boldsymbol{\Sigma}_\mathcal{B} &\approx \frac{1}{N}(\boldsymbol{K}_\mathcal{B} + \beta \boldsymbol{I})^{-1} \\
&= \frac{1}{N}\left(\mathrm{blockdiag}_{l=1,\dots,L}(\boldsymbol{K}_\mathcal{B}^{(l)} + \beta \boldsymbol{I})\right)^{-1} \\
&= \mathrm{blockdiag}_{l=1,\dots,L}\Big(\underbrace{\frac{1}{N}(\boldsymbol{K}_\mathcal{B}^{(l)} + \beta \boldsymbol{I})^{-1}}_{=:\, \boldsymbol{\Sigma}^{(l)}}\Big) \\
&= \mathrm{blockdiag}_{l=1,\dots,L}(\boldsymbol{\Sigma}^{(l)}).
\end{aligned}$$

To draw a sample $\boldsymbol{v} \in \mathbb{R}^P$ from $\mathcal{N}(\boldsymbol{\theta}_\star, \boldsymbol{\Sigma}_\mathcal{B})$, we can thus simply sample from the block covariances $\boldsymbol{v}^{(l)} \sim \mathcal{N}\left(\boldsymbol{0}, \boldsymbol{\Sigma}^{(l)}\right)$, stack these samples and add the mean, *i.e.* $\boldsymbol{v} = \boldsymbol{\theta}_\star + (\boldsymbol{v}^{(1)}, \dots, \boldsymbol{v}^{(L)})$.

Leveraging the Blocks' Kronecker Structure. To sample from $\mathcal{N}\left(\boldsymbol{0}, \boldsymbol{\Sigma}^{(l)}\right)$, we can exploit the Kronecker structure of the blocks $\boldsymbol{K}_\mathcal{B}^{(l)} = \boldsymbol{A} \otimes \boldsymbol{B}$ (we omit the layer index l for $\boldsymbol{A}$ and $\boldsymbol{B}$ for brevity). First, we compute the eigendecompositions of the Kronecker factors, *i.e.*

$\boldsymbol{A} = \boldsymbol{U}_A \boldsymbol{S}_A \boldsymbol{U}_A^\top$ and $\boldsymbol{B} = \boldsymbol{U}_B \boldsymbol{S}_B \boldsymbol{U}_B^\top$. It follows

$$\begin{aligned}
\boldsymbol{K}_{\mathcal{B}}^{(l)} + \beta \boldsymbol{I} &= \boldsymbol{A} \otimes \boldsymbol{B} + \beta \boldsymbol{I} \\
&= \boldsymbol{U}_A \boldsymbol{S}_A \boldsymbol{U}_A^\top \otimes \boldsymbol{U}_B \boldsymbol{S}_B \boldsymbol{U}_B^\top + \beta \boldsymbol{I} \\
&= \underbrace{(\boldsymbol{U}_A \otimes \boldsymbol{U}_B)}_{=: \boldsymbol{U}} \underbrace{(\boldsymbol{S}_A \otimes \boldsymbol{S}_B)}_{=: \boldsymbol{S}} (\boldsymbol{U}_A \otimes \boldsymbol{U}_B)^\top + \beta \boldsymbol{I} \\
&= \boldsymbol{U} \boldsymbol{S} \boldsymbol{U}^\top + \beta \boldsymbol{I} \\
&= \boldsymbol{U} (\boldsymbol{S} + \beta \boldsymbol{I}) \boldsymbol{U}^\top.
\end{aligned}$$

$\boldsymbol{U} = \boldsymbol{U}_A \otimes \boldsymbol{U}_B$ forms an orthogonal eigenbasis of the block and the diagonal matrix $\boldsymbol{S} + \beta \boldsymbol{I} = \boldsymbol{S}_A \otimes \boldsymbol{S}_B + \beta \boldsymbol{I}$ contains the corresponding eigenvalues (see *e.g.* [43]). It follows

$$\begin{aligned}
\boldsymbol{\Sigma}^{(l)} &= \frac{1}{N} (\boldsymbol{K}_{\mathcal{B}}^{(l)} + \beta \boldsymbol{I})^{-1} \\
&= \frac{1}{N} \boldsymbol{U} (\boldsymbol{S} + \beta \boldsymbol{I})^{-1} \boldsymbol{U}^\top \\
&= \underbrace{\frac{1}{\sqrt{N}} \boldsymbol{U} (\boldsymbol{S} + \beta \boldsymbol{I})^{-1/2}}_{=: \boldsymbol{V}} \frac{1}{\sqrt{N}} (\boldsymbol{S} + \beta \boldsymbol{I})^{-1/2} \boldsymbol{U}^\top \\
&= \boldsymbol{V} \boldsymbol{V}^\top.
\end{aligned}$$

So, in order to draw a sample $\boldsymbol{v}^{(l)} \sim \mathcal{N}\left(\boldsymbol{0}, \boldsymbol{\Sigma}^{(l)}\right)$, we first draw from a standard Gaussian $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and then transform the sample via $\boldsymbol{v}^{(l)} = \boldsymbol{V} \boldsymbol{w}$. The resulting vector $\boldsymbol{v}^{(l)}$ has mean $\boldsymbol{0}$ and covariance $\boldsymbol{V} \boldsymbol{I} \boldsymbol{V}^\top = \boldsymbol{\Sigma}^{(l)}$. As Gaussians are closed under affine linear transformations, $\boldsymbol{v}^{(l)}$ is indeed distributed according to $\mathcal{N}\left(\boldsymbol{0}, \boldsymbol{\Sigma}^{(l)}\right)$.

The matrix-vector product $\boldsymbol{w} \mapsto \boldsymbol{V} \boldsymbol{w}$ can be computed efficiently without actually forming $\boldsymbol{V}$ in memory. The first step is to multiply with $(\boldsymbol{S} + \beta \boldsymbol{I})^{-1/2}$. Since this is a diagonal matrix, we can simply multiply the vector $\boldsymbol{w}$ element-wise with the inverse square root of the diagonal entries of $\boldsymbol{S} + \beta \boldsymbol{I}$. The second step is to multiply with $\boldsymbol{U} = \boldsymbol{U}_A \otimes \boldsymbol{U}_B$, which can be implemented efficiently by using the property $(\boldsymbol{U}_A \otimes \boldsymbol{U}_B) \operatorname{vec}(\boldsymbol{W}) = \operatorname{vec}(\boldsymbol{U}_B \boldsymbol{W} \boldsymbol{U}_A^\top)$. Finally, we scale by $N^{-1/2}$.

Summary. Drawing a sample from the K-FAC LA $\mathcal{N}\left(\boldsymbol{\theta}_\star, N^{-1} (\boldsymbol{K}_{\mathcal{B}} + \beta \boldsymbol{I})^{-1}\right)$ can be done without ever forming the blocks of the covariance matrix explicitly. Using properties of the Kronecker product, we can efficiently transform a sample from the standard Gaussian to a sample $\boldsymbol{v}^{(l)} \sim \mathcal{N}\left(\boldsymbol{0}, \boldsymbol{\Sigma}^{(l)}\right)$. Stacking the samples from all blocks and adding the mean $\boldsymbol{\theta}_\star$ yields a sample from the LA due to the block-diagonal structure of the covariance matrix.

Debiased K-FAC Laplace Approximation

The idea of the debiased K-FAC Laplace approximation is to construct one Kronecker-factored curvature matrix from two mini-batches $\mathcal{B}$ and $\tilde{\mathcal{B}}$.

Eigenbasis of a K-FAC Block. First, recall that the eigendecomposition of a block $\boldsymbol{K}_{\mathcal{B}}^{(l)}$ from a K-FAC matrix can be constructed from the eigendecompositions of the Kronecker factors (see *e.g.* [43]). Consider the l-th block from both K-FAC approximations $\boldsymbol{K}_{\mathcal{B}}^{(l)} = \boldsymbol{A} \otimes \boldsymbol{B}$ and $\boldsymbol{K}_{\tilde{\mathcal{B}}}^{(l)} = \boldsymbol{C} \otimes \boldsymbol{D}$ (we omit the layer index l for $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ for brevity). Again, let $\boldsymbol{A} = \boldsymbol{U}_A \boldsymbol{S}_A \boldsymbol{U}_A^\top$ and $\boldsymbol{B} = \boldsymbol{U}_B \boldsymbol{S}_B \boldsymbol{U}_B^\top$ denote the eigendecompositions of the Kronecker factors. It holds

$$
\begin{aligned}
\boldsymbol{K}_{\mathcal{B}}^{(l)} &= \boldsymbol{A} \otimes \boldsymbol{B} \\
&= \boldsymbol{U}_A \boldsymbol{S}_A \boldsymbol{U}_A^\top \otimes \boldsymbol{U}_B \boldsymbol{S}_B \boldsymbol{U}_B^\top \\
&= \underbrace{(\boldsymbol{U}_A \otimes \boldsymbol{U}_B)}_{=: \boldsymbol{U}} \underbrace{(\boldsymbol{S}_A \otimes \boldsymbol{S}_B)}_{=: \boldsymbol{S}} (\boldsymbol{U}_A \otimes \boldsymbol{U}_B)^\top \\
&= \boldsymbol{U} \boldsymbol{S} \boldsymbol{U}^\top
\end{aligned}
$$

$\boldsymbol{U} = \boldsymbol{U}_A \otimes \boldsymbol{U}_B$ forms an orthogonal eigenbasis of the block and the diagonal matrix $\boldsymbol{S} = \boldsymbol{S}_A \otimes \boldsymbol{S}_B$ contains the eigenvalues.

Re-Evaluation of the Directional Curvatures. For the debiased approach, we keep the block's eigenbasis $\boldsymbol{U}$, but instead of using the directional curvatures $\boldsymbol{S}$, we re-evaluate these measurements on the second mini-batch $\tilde{\mathcal{B}}$. First, consider the projection of the block $\boldsymbol{K}_{\tilde{\mathcal{B}}}^{(l)}$ onto the eigenvectors $\boldsymbol{U}$, *i.e.*

$$
\begin{aligned}
\boldsymbol{U}^\top \boldsymbol{K}_{\tilde{\mathcal{B}}}^{(l)} \boldsymbol{U} &= (\boldsymbol{U}_A \otimes \boldsymbol{U}_B)^\top (\boldsymbol{C} \otimes \boldsymbol{D})(\boldsymbol{U}_A \otimes \boldsymbol{U}_B) \\
&= \boldsymbol{U}_A^\top \boldsymbol{C}\, \boldsymbol{U}_A \otimes \boldsymbol{U}_B^\top \boldsymbol{D}\, \boldsymbol{U}_B .
\end{aligned}
$$

The debiased directional curvatures are on the diagonal of $\boldsymbol{U}^\top \boldsymbol{K}_{\tilde{\mathcal{B}}}^{(l)} \boldsymbol{U}$. For a square matrix $\boldsymbol{X} \in \mathbb{R}^{n \times n}$, let $\operatorname{Diag}(\boldsymbol{X})$ denote the operator that maps the matrix onto its diagonal, *i.e.* $\operatorname{Diag} : \mathbb{R}^{n \times n} \to \mathbb{R}^n$ with $\operatorname{Diag}(\boldsymbol{X})_i := X_{ii}$ for $i \in \{1, \dots, n\}$. It holds:

$$
\begin{aligned}
\operatorname{Diag}(\boldsymbol{U}^\top \boldsymbol{K}_{\tilde{\mathcal{B}}}^{(l)} \boldsymbol{U}) &= \operatorname{Diag}(\boldsymbol{U}_A^\top \boldsymbol{C}\, \boldsymbol{U}_A \otimes \boldsymbol{U}_B^\top \boldsymbol{D}\, \boldsymbol{U}_B) \\
&= \underbrace{\operatorname{Diag}(\boldsymbol{U}_A^\top \boldsymbol{C}\, \boldsymbol{U}_A)}_{=: \tilde{\boldsymbol{s}}_A} \otimes \underbrace{\operatorname{Diag}(\boldsymbol{U}_B^\top \boldsymbol{D}\, \boldsymbol{U}_B)}_{=: \tilde{\boldsymbol{s}}_B} \\
&= \tilde{\boldsymbol{s}}_A \otimes \tilde{\boldsymbol{s}}_B \\
&=: \tilde{\boldsymbol{s}} .
\end{aligned}
$$

Construction of a Debiased Block. Now that we have the eigenbasis $\boldsymbol{U}$ and the debiased directional curvatures $\tilde{\boldsymbol{s}}$, we construct

the debiased block. Let $\tilde{\boldsymbol{S}}_A \coloneqq \operatorname{diag}(\tilde{\boldsymbol{s}}_A)$, $\tilde{\boldsymbol{S}}_B \coloneqq \operatorname{diag}(\tilde{\boldsymbol{s}}_B)$ and

$$\tilde{\boldsymbol{S}} \coloneqq \operatorname{diag}(\tilde{\boldsymbol{s}}) = \operatorname{diag}(\tilde{\boldsymbol{s}}_A \otimes \tilde{\boldsymbol{s}}_B) = \operatorname{diag}(\tilde{\boldsymbol{s}}_A) \otimes \operatorname{diag}(\tilde{\boldsymbol{s}}_B) = \tilde{\boldsymbol{S}}_A \otimes \tilde{\boldsymbol{S}}_B.$$

The debiased block is given by $\boldsymbol{U}\tilde{\boldsymbol{S}}\boldsymbol{U}^\top$. It can be written as the Kronecker product of two matrices $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ since

$$\begin{aligned}\boldsymbol{U}\tilde{\boldsymbol{S}}\boldsymbol{U}^\top &= (\boldsymbol{U}_A \otimes \boldsymbol{U}_B)(\tilde{\boldsymbol{S}}_A \otimes \tilde{\boldsymbol{S}}_B)(\boldsymbol{U}_A \otimes \boldsymbol{U}_B)^\top \\ &= \underbrace{(\boldsymbol{U}_A \tilde{\boldsymbol{S}}_A \boldsymbol{U}_A^\top)}_{=:\,\tilde{\boldsymbol{A}}} \otimes \underbrace{(\boldsymbol{U}_B \tilde{\boldsymbol{S}}_B \boldsymbol{U}_B^\top)}_{=:\,\tilde{\boldsymbol{B}}}.\end{aligned}$$

This is important as it allows for efficient sampling as described in Appendix C.1.3.

Computational Cost. Since we need the eigendecompositions of the Kronecker factors $\boldsymbol{A} = \boldsymbol{U}_A \boldsymbol{S}_A \boldsymbol{U}_A^\top$ and $\boldsymbol{B} = \boldsymbol{U}_B \boldsymbol{S}_B \boldsymbol{U}_B^\top$ for sampling in any case (see Appendix C.1.3), the computational overhead for the debiased K-FAC approximation $\tilde{\boldsymbol{A}} \otimes \tilde{\boldsymbol{B}}$ consists of computing a K-FAC approximation $\boldsymbol{C} \otimes \boldsymbol{D}$ on another mini-batch, re-evaluating the directional curvatures $\tilde{\boldsymbol{s}}_A = \operatorname{Diag}(\boldsymbol{U}_A^\top \boldsymbol{C}\, \boldsymbol{U}_A)$ and $\tilde{\boldsymbol{s}}_B = \operatorname{Diag}(\boldsymbol{U}_B^\top \boldsymbol{D}\, \boldsymbol{U}_B)$ and finally computing $\tilde{\boldsymbol{A}} = \boldsymbol{U}_A \operatorname{diag}(\tilde{\boldsymbol{s}}_A)\, \boldsymbol{U}_A^\top$ and $\tilde{\boldsymbol{B}} = \boldsymbol{U}_B \operatorname{diag}(\tilde{\boldsymbol{s}}_B)\, \boldsymbol{U}_B^\top$.

From Block-Level to Full Matrix. So far, we have only considered the debiasing of a single block. However, correcting the blocks's eigenvalues is sufficient because they coincide with the eigenvalues of the full matrix (due to the block-diagonal structure). Let $\boldsymbol{u}^{(l)}$ denote an eigenvector of the l-th block $\boldsymbol{X}^{(l)}$ of some block-diagonal matrix $\boldsymbol{X} = \operatorname{blockdiag}_{l=1,\dots,L}(\boldsymbol{X}^{(l)})$ corresponding to the eigenvalue λ. Then, $\boldsymbol{u}^\top \coloneqq (\boldsymbol{0}^\top, \dots, \boldsymbol{u}^{(l)\top}, \dots, \boldsymbol{0}^\top)$ is an eigenvector of $\boldsymbol{X}$ corresponding to the same eigenvalue λ, because

$$\boldsymbol{X} \cdot \boldsymbol{u} = \operatorname{blockdiag}_{l=1,\dots,L}(\boldsymbol{X}^{(l)}) \cdot \boldsymbol{u} = \begin{pmatrix} \boldsymbol{X}^{(1)} \cdot \boldsymbol{0} \\ \vdots \\ \boldsymbol{X}^{(l)} \cdot \boldsymbol{u}^{(l)} \\ \vdots \\ \boldsymbol{X}^{(L)} \cdot \boldsymbol{0} \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} \\ \vdots \\ \lambda\, \boldsymbol{u}^{(l)} \\ \vdots \\ \boldsymbol{0} \end{pmatrix} = \lambda \boldsymbol{u}.$$

The eigenvalues of $\boldsymbol{X}$ thus coincide with the eigenvalues of its blocks; and $\boldsymbol{X}$'s eigenvectors can be constructed from the eigenvectors of the blocks by filling them up with zeros.

Connection to Equation (7.9): The equivalence between Equation (7.9) and the approach we describe above may not be obvious. We thus show here that the directional curvature of the debiased matrix $\hat{\boldsymbol{K}}$ along an eigenvector $\boldsymbol{u}$ of $\boldsymbol{K}_{\mathcal{B}}$ indeed coincides with the directional curvature $\boldsymbol{u}^\top \boldsymbol{K}_{\tilde{\mathcal{B}}}\, \boldsymbol{u}$ on $\tilde{\mathcal{B}}$.

More concretely, let $\hat{\boldsymbol{K}} = \operatorname{blockdiag}_{l=1,\dots,L}(\hat{\boldsymbol{K}}^{(l)})$ denote the debiased

K-FAC approximation constructed from $\boldsymbol{K}_{\mathcal{B}}$ and $\boldsymbol{K}_{\tilde{\mathcal{B}}}$ as described above. Also, let $\boldsymbol{u}^\top := (\boldsymbol{0}^\top, \dots, {\boldsymbol{u}^{(l)}}^\top, \dots, \boldsymbol{0}^\top)$ denote an eigenvector of $\boldsymbol{K}_{\mathcal{B}}$, where $\boldsymbol{u}^{(l)}$ is the ith eigenvector of $\boldsymbol{K}_{\mathcal{B}}^{(l)}$ (*i.e.* the ith column of $\boldsymbol{U}$). It holds

$$\begin{aligned}
\boldsymbol{u}^\top \hat{\boldsymbol{K}} \boldsymbol{u} &= {\boldsymbol{u}^{(l)}}^\top \hat{\boldsymbol{K}}^{(l)} \boldsymbol{u}^{(l)} \\
&= {\boldsymbol{u}^{(l)}}^\top \boldsymbol{U} \tilde{\boldsymbol{S}} \boldsymbol{U}^\top \boldsymbol{u}^{(l)} \\
&= \boldsymbol{e}_i^\top \tilde{\boldsymbol{S}} \boldsymbol{e}_i \\
&= \tilde{\boldsymbol{s}}_i \\
&= {\boldsymbol{u}^{(l)}}^\top \boldsymbol{K}_{\tilde{\mathcal{B}}}^{(l)} \boldsymbol{u}^{(l)} \\
&= \boldsymbol{u}^\top \boldsymbol{K}_{\tilde{\mathcal{B}}} \boldsymbol{u},
\end{aligned}$$

where $\boldsymbol{e}_i$ denotes the ith eigenvector.

C.1.4 Bias in the Directional Slope

Biased Directional Slopes Along Negative Gradient Directions. Assume that we are given a quadratic $q(\,\cdot\,; \mathcal{B})$ around the current parameters $\boldsymbol{\theta}_0$. We consider the negative normalized gradient direction $\boldsymbol{d} = -\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \cdot \|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\|^{-1}$ at some location $\boldsymbol{\theta}_\bullet$ evaluated on mini-batch $\mathcal{B}$. We have

$$\begin{aligned}
\underbrace{\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})}_{\bullet} &= \underbrace{\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet} + \|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\|(1 - \cos(\alpha)) \\
&\geq \underbrace{\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet},
\end{aligned} \tag{C.10}$$

where $\alpha := \angle(\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}), \nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}}))$, and we assumed $\|\nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})\| = \|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\|$ which is true at least in expectation. Projecting a gradient onto its negative direction—the right-hand side of the inequality—will *always* result in a directional slope that is ≤ 0 since

$$\begin{aligned}
\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \mathcal{B}) &= \boldsymbol{d}^\top \nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \\
&= -\frac{\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})^\top}{\|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\|} \nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}) \\
&= -\|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\| \\
&\leq 0
\end{aligned}$$

and $= 0$ only if $\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}) = \boldsymbol{0}$. Projecting a *different* gradient (of equal length) onto that direction—that is the left-hand side of the inequality—will result in a larger (possibly even positive) directional slope.

Proof of Equation (C.10). Equation (C.10) quantifies the bias in the directional slope along $\boldsymbol{d}$ as a function of the alignment of the two

mini-batch gradients. It holds

$$\begin{aligned}
&\underbrace{\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet} - \underbrace{\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})}_{\bullet} \\
&\quad = \boldsymbol{d}^\top \nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}) - \boldsymbol{d}^\top \nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}}) \\
&\quad = \underbrace{\|\boldsymbol{d}\|}_{=1}\, \|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\| \underbrace{\cos(\pi)}_{=-1} - \underbrace{\|\boldsymbol{d}\|}_{=1}\, \|\nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})\| \cos(\gamma) \\
&\quad = \|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\|(-1 - \cos(\gamma)),
\end{aligned}$$

where $\gamma = \measuredangle(\boldsymbol{d}, \nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}}))$, and we assumed that $\|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\| = \|\nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})\|$ in the last step (which is true, at least, in expectation). Next, we re-write γ as

$$\begin{aligned}
\gamma &= \measuredangle(\boldsymbol{d}, \nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})) \\
&= \measuredangle(-\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}), \nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})) \\
&= \pi - \underbrace{\measuredangle(\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B}), \nabla q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}}))}_{=:\alpha}.
\end{aligned}$$

It follows $-1 - \cos(\gamma) = -1 - \cos(\pi - \alpha) = \cos(\alpha) - 1$. Substituting this into the expression for the bias, we arrive at

$$\begin{aligned}
\underbrace{\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet} - \underbrace{\partial_{\boldsymbol{d}}\, q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})}_{\bullet} &= \|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\|(-1 - \cos(\gamma)) \\
&= \|\nabla q(\boldsymbol{\theta}_\bullet; \mathcal{B})\|(\cos(\alpha) - 1),
\end{aligned}$$

from which Equation (C.10) follows. □

The First CG Search Direction. Assume that the current parameters are $\boldsymbol{\theta}_0$, and we apply CG to the local quadratic approximation $q(\,\cdot\,; \mathcal{B})$. The very first search direction is the quadratic's normalized negative gradient $\boldsymbol{d}_0 = -\nabla q(\boldsymbol{\theta}_0; \mathcal{B}) \cdot \|\nabla q(\boldsymbol{\theta}_0; \mathcal{B})\|^{-1}$ at $\boldsymbol{\theta}_0$. This is exactly the situation we describe above, where $\boldsymbol{\theta}_\bullet$ is set to $\boldsymbol{\theta}_0$. Equation (C.10) thus explains the bias for the first CG direction.

The Subsequent CG Search Directions. For the subsequent CG search directions ($\boldsymbol{d}_p$ with $p \geq 1$), the situation is more complex: Each direction $\boldsymbol{d}_p$ is a linear combination of the current normalized negative residual $-\boldsymbol{r}_p$ (note that $\boldsymbol{r}_p$ coincides with the gradient $\nabla q(\boldsymbol{\theta}_p; \mathcal{B})$, see Appendix C.1.2) and the previous search direction $\boldsymbol{d}_{p-1}$ (see Appendix C.1.2) and the previous search direction $\boldsymbol{d}_{p-1}$ (see Algorithm C.1), *i.e.*

$$\begin{aligned}
\boldsymbol{d}_p &= \frac{\boldsymbol{s}_p}{\|\boldsymbol{s}_p\|} \\
&= \frac{1}{\|\boldsymbol{s}_p\|}(-\boldsymbol{r}_p) + \frac{\beta_p}{\|\boldsymbol{s}_p\|}\boldsymbol{s}_{p-1} \\
&= \underbrace{\frac{\|\boldsymbol{r}_p\|}{\|\boldsymbol{s}_p\|}}_{=:\eta_1 \geq 0} \frac{-\boldsymbol{r}_p}{\|\boldsymbol{r}_p\|} + \underbrace{\frac{\beta_p \|\boldsymbol{s}_{p-1}\|}{\|\boldsymbol{s}_p\|}}_{=:\eta_2 \geq 0} \boldsymbol{d}_{p-1}
\end{aligned}$$

$$= \eta_1 \frac{-\boldsymbol{r}_p}{\|\boldsymbol{r}_p\|} + \eta_2 \, \boldsymbol{d}_{p-1}.$$

The additional correction term ensures conjugacy. The directional slope along $\boldsymbol{d}_p$ thus also splits into two corresponding terms: The slope along the negative gradient direction and the slope along the previous search direction. It holds

$$\begin{aligned}
\partial_{\boldsymbol{d}_p} q(\boldsymbol{\theta}_p; \mathcal{B}) &= \boldsymbol{d}_p^\top \nabla q(\boldsymbol{\theta}_p; \mathcal{B}) \\
&= \eta_1 \frac{-\boldsymbol{r}_p}{\|\boldsymbol{r}_p\|}^\top \nabla q(\boldsymbol{\theta}_p; \mathcal{B}) + \eta_2 \, \boldsymbol{d}_{p-1}^\top \nabla q(\boldsymbol{\theta}_p; \mathcal{B}) \\
&= \eta_1 \partial_{-\boldsymbol{r}_p/\|\boldsymbol{r}_p\|} q(\boldsymbol{\theta}_p; \mathcal{B}) + \eta_2 \, \partial_{\boldsymbol{d}_{p-1}} q(\boldsymbol{\theta}_p; \mathcal{B}).
\end{aligned}$$

The bias in the first term—as explained by Equation (C.10)—also introduces a bias along the search direction $\boldsymbol{d}_p$.

C.1.5 Bias in the Directional Curvature

Derivation for Equation (7.6). Let $\boldsymbol{H}_{\mathcal{B}} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$ and $\boldsymbol{H}_{\tilde{\mathcal{B}}} = \tilde{\boldsymbol{U}}\tilde{\boldsymbol{\Lambda}}\tilde{\boldsymbol{U}}^\top$ denote the eigendecompositions of the mini-batch Hessians, where $\boldsymbol{U} = (\boldsymbol{u}_1, \dots, \boldsymbol{u}_P), \tilde{\boldsymbol{U}} = (\tilde{\boldsymbol{u}}_1, \dots, \tilde{\boldsymbol{u}}_P) \in \mathbb{R}^{P \times P}$ contain the orthonormal eigenvectors and $\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \dots, \lambda_P)$, $\tilde{\boldsymbol{\Lambda}} = \operatorname{diag}(\tilde{\lambda}_1, \dots, \tilde{\lambda}_P)$ the respective eigenvalues in descending order, *i.e.* $\lambda_1 \geq \dots \geq \lambda_P$ and $\tilde{\lambda}_1 \geq \dots \geq \tilde{\lambda}_P$. The directional curvature along one of $\boldsymbol{H}_{\mathcal{B}}$'s eigenvectors $\boldsymbol{u}_i$ on mini-batch $\mathcal{B}$ is given by the corresponding eigenvalue since

$$\underbrace{\partial^2_{\boldsymbol{u}_i} q(\boldsymbol{\theta}_\bullet; \mathcal{B})}_{\bullet} = \boldsymbol{u}_i^\top \boldsymbol{H}_{\mathcal{B}} \boldsymbol{u}_i = \boldsymbol{u}_i^\top \lambda_i \boldsymbol{u}_i = \lambda_i \|\boldsymbol{u}_i\|_2^2 = \lambda_i.$$

For $\tilde{\mathcal{B}}$, we obtain

$$\begin{aligned}
\underbrace{\partial^2_{\boldsymbol{u}_i} q(\boldsymbol{\theta}_\bullet; \tilde{\mathcal{B}})}_{\bullet} &= \boldsymbol{u}_i^\top \boldsymbol{H}_{\tilde{\mathcal{B}}} \boldsymbol{u}_i \\
&= \boldsymbol{u}_i^\top \left(\sum_{p=1}^{P} \tilde{\lambda}_p \tilde{\boldsymbol{u}}_p \tilde{\boldsymbol{u}}_p^\top \right) \boldsymbol{u}_i \\
&= \sum_{p=1}^{P} \tilde{\lambda}_p \underbrace{(\tilde{\boldsymbol{u}}_p^\top \boldsymbol{u}_i)^2}_{=: \Omega_{i,p}} \\
&= \sum_{p=1}^{P} \tilde{\lambda}_p \, \Omega_{i,p}.
\end{aligned}$$

Weights Sum Up to One. The weights $\Omega_{i,p}$ are non-negative and

sum up to one, *i.e.* $\sum_{p=1}^{P} \Omega_{i,p} = 1$. This is because

$$\sum_{p=1}^{P} \Omega_{i,p} = \sum_{p=1}^{P} (\tilde{\boldsymbol{u}}_p^\top \boldsymbol{u}_i)^2 = \|\tilde{\boldsymbol{U}}^\top \boldsymbol{u}_i\|_2^2 = \boldsymbol{u}_i^\top \underbrace{\tilde{\boldsymbol{U}}\tilde{\boldsymbol{U}}^\top}_{=\boldsymbol{I}} \boldsymbol{u}_i = \|\boldsymbol{u}_i\|_2^2 = 1 \tag{C.11}$$

where we used that $\tilde{\boldsymbol{U}}$ is an orthogonal matrix and that $\boldsymbol{u}_i$ is normalized.

Proof of Equation (7.7). Here, we assume the simplified case where the spectra of $\boldsymbol{H}_{\mathcal{B}}$ and $\boldsymbol{H}_{\tilde{\mathcal{B}}}$ are identical, *i.e.* $\lambda_p = \tilde{\lambda}_p \; \forall p \in \{1, \dots, P\}$.

First, consider $\boldsymbol{u}_1$ and assume that $\boldsymbol{u}_1$ and $\tilde{\boldsymbol{u}}_1$ are *not* perfectly aligned, *i.e.* $\Omega_{1,1} = (\boldsymbol{u}_1^\top \tilde{\boldsymbol{u}}_1)^2 < 1$. It holds

$$\underbrace{\partial^2_{\boldsymbol{u}_1} q(\boldsymbol{\theta}_0, \tilde{\mathcal{B}})}_{\bullet} \overset{(7.6)}{=} \sum_{p=1}^{P} \lambda_p \Omega_{1,p} = \lambda_1 \Omega_{1,1} + \sum_{p=2}^{P} \lambda_p \Omega_{1,p}.$$

The sum can be bounded from above by putting all remaining weight $1 - \Omega_{1,1}$ (see Equation (C.11)) on the second-largest eigenvalue λ_2, *i.e.*

$$\begin{aligned} \underbrace{\partial^2_{\boldsymbol{u}_1} q(\boldsymbol{\theta}_0, \tilde{\mathcal{B}})}_{\bullet} &\leq \lambda_1 \Omega_{1,1} + \lambda_2 (1 - \Omega_{1,1}) \\ &\overset{(*)}{\leq} \lambda_1 \Omega_{1,1} + \lambda_1 (1 - \Omega_{1,1}) \\ &= \lambda_1 \\ &= \underbrace{\partial^2_{\boldsymbol{u}_1} q(\boldsymbol{\theta}_0, \mathcal{B})}_{\bullet}. \end{aligned}$$

The inequality $(*)$ turns into $<$ if the top two eigenvalues are separated, *i.e.* $\lambda_1 > \lambda_2$. The proof for $\boldsymbol{u}_P$ is similar. We assume that $\boldsymbol{u}_P$ and $\tilde{\boldsymbol{u}}_P$ are *not* perfectly aligned, *i.e.* $\Omega_{P,P} = (\boldsymbol{u}_P^\top \tilde{\boldsymbol{u}}_P)^2 < 1$. It follows

$$\underbrace{\partial^2_{\boldsymbol{u}_P} q(\boldsymbol{\theta}_0, \tilde{\mathcal{B}})}_{\bullet} \overset{(7.6)}{=} \sum_{p=1}^{P} \lambda_p \Omega_{P,p} = \sum_{p=1}^{P-1} \lambda_p \Omega_{P,p} + \lambda_P \Omega_{P,P}.$$

This time, we bound the sum from below by putting all remaining weight $1 - \Omega_{P,P}$ (see Equation (C.11)) on the second-smallest eigenvalue λ_{P-1}, *i.e.*

$$\begin{aligned} \underbrace{\partial^2_{\boldsymbol{u}_P} q(\boldsymbol{\theta}_0, \tilde{\mathcal{B}})}_{\bullet} &\geq \lambda_{P-1} (1 - \Omega_{P,P}) + \lambda_P \Omega_{P,P} \\ &\overset{(*)}{\geq} \lambda_P (1 - \Omega_{P,P}) + \lambda_P \Omega_{P,P} \\ &= \lambda_P \\ &= \underbrace{\partial^2_{\boldsymbol{u}_P} q(\boldsymbol{\theta}_0, \mathcal{B})}_{\bullet}. \end{aligned}$$

Again, the inequality $(*)$ turns into $>$ if the bottom two eigenvalues are separated, *i.e.* $\lambda_{P-1} > \lambda_P$. This concludes the proof of Equation (7.7). □

C.2 Experimental Details

In the following, we provide information on the experimental setup and give detailed instructions on how to replicate all empirical results.

C.2.1 Test Problems and Training Procedures

Throughout the paper, we use a series of test problems with different models, data sets, and training procedures which we describe in more detail in the following. We use DeepOBS [123] on top of PyTorch [111] as our general benchmarking framework as it provides easy access to a variety of data sets and model architectures.

Data. We use the data sets CIFAR-10 (with $C = 10$ classes) and CIFAR-100 (with $C = 100$ classes) [78]. Each data set contains 60 000 data points that are split into 40 000 training samples, 10 000 validation samples and 10 000 test samples. For the experiments on out-of-distribution (OOD) data, we create the data sets CIFAR-10-C and CIFAR-100-C, each containing 10 000 images, as described in [55]. In these data sets, each image is corrupted using one out of 15 different corruptions (chosen uniformly at random) at a specific severity level (a number between 1 and 5). We also use the ImageNet data set [32] which contains images from $C = 1000$ different classes.

Test Problems. All of the following test problems use the cross-entropy loss function.

(A) **All-CNN-C on CIFAR-100.** This test problem uses the All-CNN-C model architecture [132] (where we removed the dropout layers as explained in Section 7.3.1) and CIFAR-100 data. The training hyperparameters are taken from an existing benchmark [122]: We train the model with SGD (learning rate 0.171234) with batch size 256 for 350 epochs. Weight decay $\beta = 0.0005$ is used on the weights but not the biases of the model.

(B) **All-CNN-C on CIFAR-10.** This test problem is similar to (A) but uses the CIFAR-10 data set. The model is trained with SGD (learning rate 0.025, momentum 0.9) with batch size 256 for 350 epochs. The learning rate is reduced by a factor of 10 at epochs 200, 250 and 300, as suggested in [132]. Weight

decay $\beta = 0.001$ is used on the weights but not the biases of the model.

(C) **Wide ResNet 16-4 on CIFAR-100.** This is a test problem from DeepOBS. For details on the architecture, see [146]. We train this model with SGD (learning rate 0.1, momentum 0.9) with batch size 128 for 160 epochs. The learning rate is reduced by a factor of 5 after 60 and 120 epochs. Weight decay $\beta = 0.0005$ is used on the non-bias weights of the model.

(D) **ConvNet on CIFAR-10.** This test problem uses a simple convolutional neural network with variable depth d and width w (denoted by "model d-w" in Figures C.1 and C.15). The *first* block of the model consists of a convolutional layer (kernel size 5, padding 2) with 3 input and w output channels, a ReLU activation function, and a max-pooling layer (kernel size 2). The *last* block consists of a max-pooling layer (kernel size 2), a flatten layer, and a dense linear layer. In between those blocks, there are d *hidden* blocks each consisting of a convolutional layer (kernel size 5, padding 2) with w input and w output channels followed by a ReLU activation function. So, the depth d determines the number of hidden blocks, whereas w controls the number of parameters in the layers and is thus referred to as the width. We use $d \in \{1, 4, 7\}$ and $w \in \{32, 64, 128\}$ (resulting in 9 models) and train each model for 100 epochs on the CIFAR-10 data set using Adam with standard hyperparameters (learning rate 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$) with batch size 256. No weight decay is used for this test problem.

(E) **ResNet-50 on ImageNet.** This test problem uses the ResNet-50 model architecture [54] and the ImageNet data set. We use the IMAGENET1K_V1 weights pre-trained on ImageNet-1k using SGD (initial learning rate 0.1, momentum 0.9) with batch size 256 for 90 epochs. For the pre-training, the learning rate was set by a multistep scheduler which multiplied the learning rate by a factor of 0.1 after every 30 epochs. Additionally, a weight decay of $\beta = 0.0001$ was used on all weights apart from biases and the learned batch normalization weights.

(F) **ViT Little on ImageNet.** This test problem uses the ViT Little architecture of the Vision Transformer model family [35] on the ImageNet data set. We use the vit_little_patch16_reg4_gap_256.sbb_in1k weights pre-trained on ImageNet-1k using NadamW. For the exact hyperparameters used for the weights, refer to the HuggingFace Model Card.

During training, we store the model's parameters at 10 checkpoints spaced log-equidistantly between the first and last epoch. The training metrics for all test problems are shown in Figure C.1.

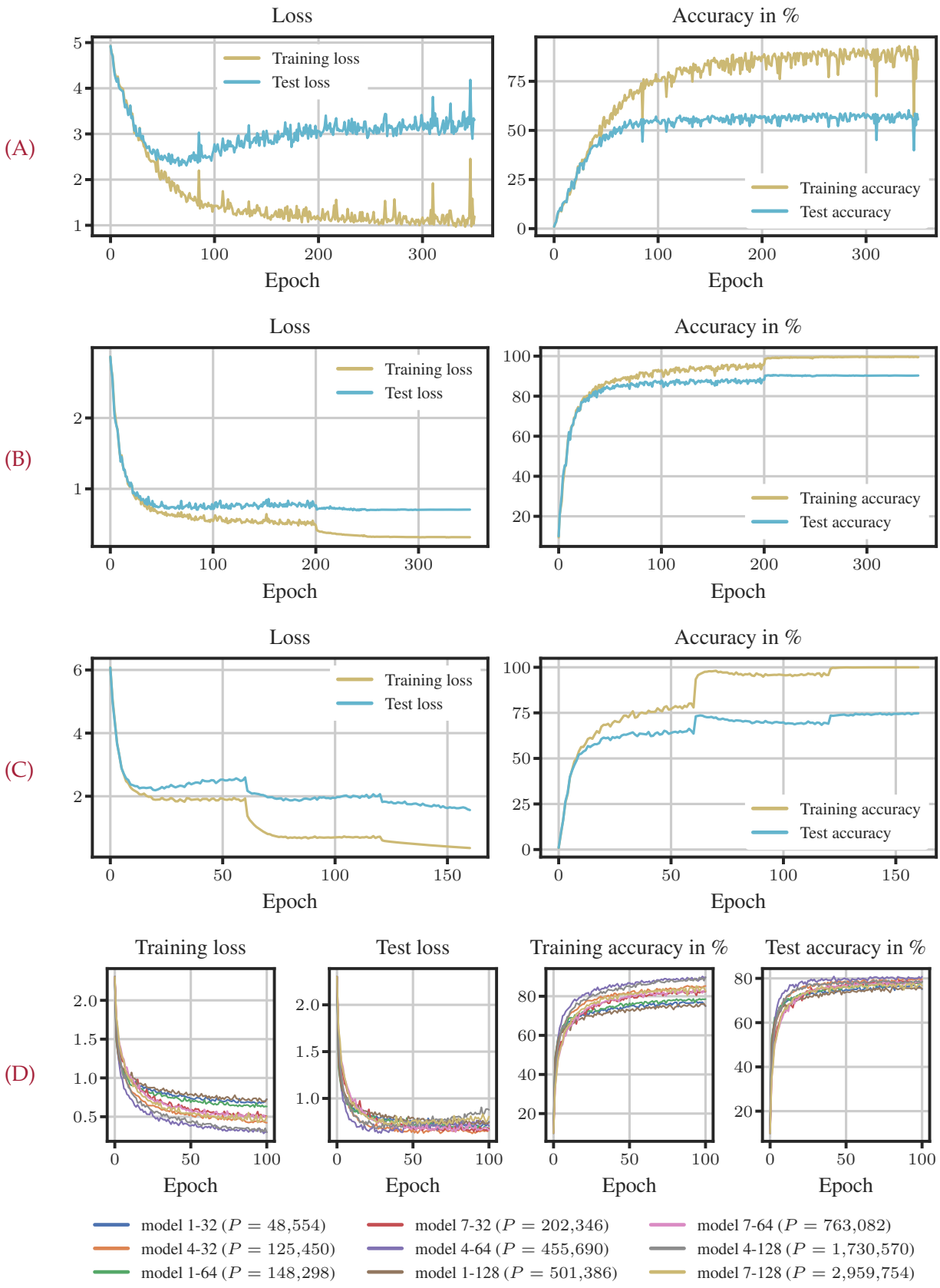


Figure C.1: Training Metrics for All Test Problems. The panels show the training/test loss/accuracy during training for all test problems (A) (All-CNN-C on CIFAR-100), (B) (All-CNN-C on CIFAR-10), (C) (Wide ResNet 16-4 on CIFAR-100) and (D) (ConvNet on CIFAR-10). Test problem (E) (ResNet-50 on ImageNet) is not shown since a pre-trained model is used here.

C.2.2 Matrix-Vector Products with the Curvature Matrix

Access to Curvature Matrices via BackPACK. In order to compute an eigendecomposition of the curvature matrix or apply the CG method to a quadratic, we need access to matrix-vector products with the curvature matrix $v \mapsto \boldsymbol{H}_{\mathcal{B}} \cdot v$. When products with the full-batch curvature matrix are required, we accumulate the mini-batch quantities over manageable chunks of the training set. Our implementation uses BackPACK [28] that provides access to products with the Hessian $v \mapsto \nabla^2 \mathcal{L}(\boldsymbol{\theta}; \mathcal{B}) \cdot v$ and GGN $v \mapsto \boldsymbol{G}_{\mathcal{B}} \cdot v$ of the empirical risk $\mathcal{L}$ as well as the K-FAC curvature approximation $\boldsymbol{K}_{\mathcal{B}}$.

Eigenvectors of the Curvature Matrix. Given access to matrix-vector products with the curvature matrix, we can construct an instance of a `scipy.sparse.linalg.LinearOperator` that can be used in `scipy.sparse.linalg.eigsh`.

C.2.3 Section 7.1 (Introduction): Figure 7.1

For the visual abstract in Figure 7.1, we use the fully trained ALL-CNN-C model from test problem (A) (see Appendix C.2.1). The experimental procedure below is repeated for 5 different mini-batches $\mathcal{B}_m$, $m \in \{0, 1, 2, 3, 4\}$ of size $|\mathcal{B}_m| = 512$ (we omit the index m in the following).

Experimental Procedure. We use the GGN curvature proxy $\boldsymbol{H}_{\mathcal{B}} \leftarrow \boldsymbol{G}_{\mathcal{B}} + \beta \boldsymbol{I}$ and compute its top two eigenvectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ (see Appendix C.2.2). In order to evaluate the quadratic $q(\boldsymbol{\theta}_\star; \mathcal{B})$ in the 2D space spanned by those eigenvectors efficiently, we use the following equation:

$$
\begin{aligned}
q(\boldsymbol{\theta}_\star + \tau_1 \boldsymbol{u}_1 + \tau_2 \boldsymbol{u}_2; \mathcal{B}) = &\frac{1}{2} \tau_1^2 [\boldsymbol{u}_1^\top \boldsymbol{H}_{\mathcal{B}} \boldsymbol{u}_1] \\
&+ \tau_1 \tau_2 [\boldsymbol{u}_1^\top \boldsymbol{H}_{\mathcal{B}} \boldsymbol{u}_2] \\
&+ \frac{1}{2} \tau_2^2 [\boldsymbol{u}_2^\top \boldsymbol{H}_{\mathcal{B}} \boldsymbol{u}_2] \\
&+ \tau_1 [\boldsymbol{u}_1^\top \boldsymbol{g}_{\mathcal{B}}] + \tau_2 [\boldsymbol{u}_2^\top \boldsymbol{g}_{\mathcal{B}}] + [c_{\mathcal{B}}].
\end{aligned}
$$

We can compute all the terms on the right-hand side in brackets *once*, store them, and then evaluate the quadratic for arbitrary values of τ_1 and τ_2 at basically no cost. The derivation for the full-batch quadratic $q(\boldsymbol{\theta}_\star + \tau_1 \boldsymbol{u}_1 + \tau_2 \boldsymbol{u}_2; \mathcal{D})$ is analogous.

C.2.4 Section 7.3 (The Shape of a Mini-Batch Quadratic): Figures 7.2, 7.3a and 7.3b

For the evaluation of the directional derivatives, we use the fully trained ALL-CNN-C model from test problem (A) (see Appendix C.2.1).

Three Settings. Throughout this section, we consider three different settings: (i) Quadratics that use the Hessian of the empirical risk, *i.e.* $\boldsymbol{H}_{\mathcal{B}} \leftarrow \nabla^2 \mathcal{L}(\boldsymbol{\theta}_\star; \mathcal{B}) + \beta \boldsymbol{I}$ at batch size 512, (ii) quadratics that use the Hessian's GGN approximation, *i.e.* $\boldsymbol{H}_{\mathcal{B}} \leftarrow \boldsymbol{G}_{\mathcal{B}} + \beta \boldsymbol{I}$ at batch size 512 and (iii) at batch size 2048. For each setting, the experimental procedure below is repeated for 3 mini-batches $\mathcal{B}_m$, $m \in \{0, 1, 2\}$. The prior precision β is set to 0.0005 (the same β was used for training, see Appendix C.2.1) and only acts on the weights of the model but not its biases.

Experimental Procedure (Biases). The experimental procedure consists of two steps: The computations of the directions on mini-batch $\mathcal{B}_m$ and the evaluation of the directional derivatives on *all* mini-batches (of the same size) in the training data set.

1. **Directions Based on $\mathcal{B}_m$.** First, we use the quadratic $q(\boldsymbol{\theta}_\star; \mathcal{B}_m)$ with the given curvature proxy and batch size to compute a set of 100 directions. This is either the top 100 eigenvectors of $\boldsymbol{H}_{\mathcal{B}}$ computed via `scipy.sparse.linalg.eigsh` (see Appendix C.2.2) or the first 100 CG search directions (see Appendix C.2.5). In the case of CG, we also store the trajectory of the iterates $\boldsymbol{\theta}_\star = \boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{100}$.
2. **Directional Derivatives on All Mini-Batches.** Finally, we evaluate the directional slope and curvature for all directions from step 1 on *all* mini-batches $\mathcal{B}_{m'}$, $m' \in \{0, \ldots, M-1\}$ in the training data set (where M denotes the number of mini-batches in the training data).
   - **Directional Derivatives Along Eigenvectors.** For an eigenvector $\boldsymbol{u}$ the directional slope and curvature are given by $\partial_{\boldsymbol{u}}\, q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})$ and $\partial^2_{\boldsymbol{u}}\, q(\boldsymbol{\theta}_\star; \mathcal{B}_{m'})$, respectively.
   - **Directional Derivatives Along CG Search Directions.** For CG, we evaluate the directional slope and curvature along each search direction $\boldsymbol{d}_p$ at the respective iterate $\boldsymbol{\theta}_p$ of the trajectory, *i.e.* $\partial_{\boldsymbol{d}_p}\, q(\boldsymbol{\theta}_p; \mathcal{B}_{m'})$ for the slope and $\partial^2_{\boldsymbol{d}_p}\, q(\boldsymbol{\theta}_p; \mathcal{B}_{m'})$ for the curvature. As the curvature $\nabla^2 q(\boldsymbol{\theta}; \mathcal{B}_{m'}) \equiv \boldsymbol{H}_{\mathcal{B}_{m'}}$ is independent of $\boldsymbol{\theta}$, this small distinction is irrelevant for the directional curvature. For the directional slope, however, it does make a difference.

Results (Biases). The results for setting (i) are shown in Figure C.2, for setting (ii) in Figures 7.2 and C.3 and for (iii) in Figure C.4. The biases in the directional slopes and curvatures can be observed across

all scenarios. The biases in the slopes are even more pronounced along the CG search directions than along the eigenvectors of the curvature matrix. The opposite holds for the curvature biases that tend to be larger along the eigenvectors. This is consistent with our explanations from Section 7.3.2. Both the biases in the slope and curvature decrease with increasing mini-batch size. For setting (i), CG encounters a search direction with negative curvature (indicating that, as expected, the curvature matrix $\boldsymbol{H}_{\mathcal{B}}$ is indefinite) and thus terminates already after 4 iterations.

Experimental Procedure (Overlaps). Next, we compute the overlaps between the eigenspaces of the curvature matrices on different mini-batches $\mathcal{B}$ and $\tilde{\mathcal{B}}$. More specifically, we compute $\Omega_{i,p} = (\boldsymbol{u}_i^\top \tilde{\boldsymbol{u}}_p)^2$, where $i, p \in \{1, \dots, 100\}$, $\boldsymbol{u}_i$ is an eigenvector of $\boldsymbol{H}_{\mathcal{B}}$ and $\tilde{\boldsymbol{u}}_p$ is an eigenvector of $\boldsymbol{H}_{\tilde{\mathcal{B}}}$. The values $\Omega_{i,p}$ are bounded between 0 ($\boldsymbol{u}_i$ and $\tilde{\boldsymbol{u}}_p$ are orthogonal) and 1 ($\boldsymbol{u}_i$ and $\tilde{\boldsymbol{u}}_p$ are identical, up to their sign). We apply a log-transform to $\Omega_{i,p}$. In Figure 7.3a and Appendix C.2.4, values below -8 (*i.e.* $\Omega_{i,p} \leq 10^{-8}$) are shown in black, values equal to 0 (*i.e.* $\Omega_{i,p} = 10^0 = 1$) are shown in white.

Results (Overlaps). The results are shown in Appendix C.2.4. They show that the eigenspaces of the curvature matrices are not perfectly aligned: Eigenvectors from $\mathcal{B}_m$ typically overlap with several eigenvectors from $\tilde{\mathcal{B}}$. The eigenspaces are more aligned at the larger batch size 2048 than at batch size 512. This seems reasonable since random vectors in high-dimensional spaces tend to be orthogonal to each other —less stochasticity (due to a larger batch size) thus leads to better alignment.

C.2.5 Section 7.6.1 (Debiased Conjugate Gradients): Figure 7.4

Here, we describe the experimental details for Figure 7.4 from Section 7.6.1. For the derivation and the mathematical details of the standard single-batch CG method and the debiased approach, see Appendix C.1.2. The experiment uses the fully trained All-CNN-C model from test problem (A) (see Appendix C.2.1) and the GGN curvature proxy $\boldsymbol{H}_{\mathcal{B}} \leftarrow \boldsymbol{G}_{\mathcal{B}} + \beta \boldsymbol{I}$ with $\beta = 0.0005$.

Experimental Procedure. The experimental procedure consists of two steps: The computation of the CG trajectories and the evaluation of the four performance metrics.

1. **Computation of the CG Trajectories.** For the single-batch CG approach, we use one mini-batch of size 1024, apply $K = 30$ CG iterations and store the trajectory $\boldsymbol{\theta}_\star = \boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_{30}$. For the debiased approach (details in Appendix C.1.2), we use two mini-batches of size 512 to construct the trajectory. As the debiased approach uses a total of 60 GGN-vector

Biases: Additional Results for $H_{\mathcal{B}} \leftarrow \nabla^2 \mathcal{L}(\theta_\star; \mathcal{B}) + \beta I$ (Batch Size 512).

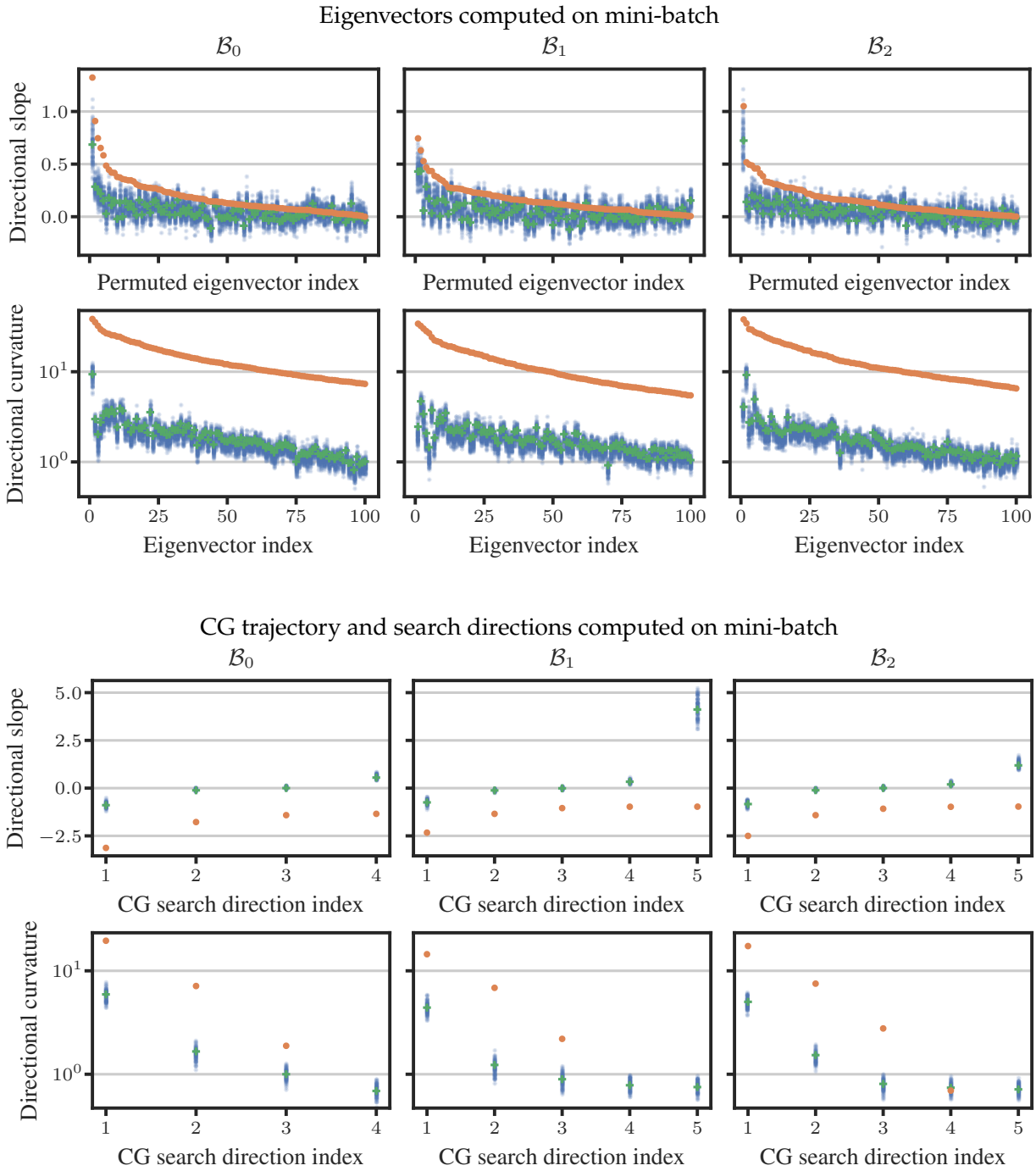


Figure C.2: Directional Slopes and Curvatures Are Biased. The experimental setting is similar to Figure 7.2 but uses the Hessian of the empirical risk, *i.e.* $H_{\mathcal{B}} \leftarrow \nabla^2 \mathcal{L}(\theta_\star; \mathcal{B}) + \beta I$ at batch size 512. The upper plot *(Top)* shows the directional derivatives along the top 100 eigenvectors of $H_{\mathcal{B}}$, the lower plot *(Bottom)* shows the directional derivatives along the first 100 CG search directions. For the top panel of the upper plot, we switch the order and sign of the eigenvectors such that the orange dots are all above zero and in descending order. There are strong systematic biases in the directional slopes and curvatures. The curvature biases are more pronounced along the eigenvectors, whereas the biases in the slope are larger along the CG directions.

Biases: Additional Results for $H_{\mathcal{B}} \leftarrow G_{\mathcal{B}} + \beta I$ (Batch Size 512).

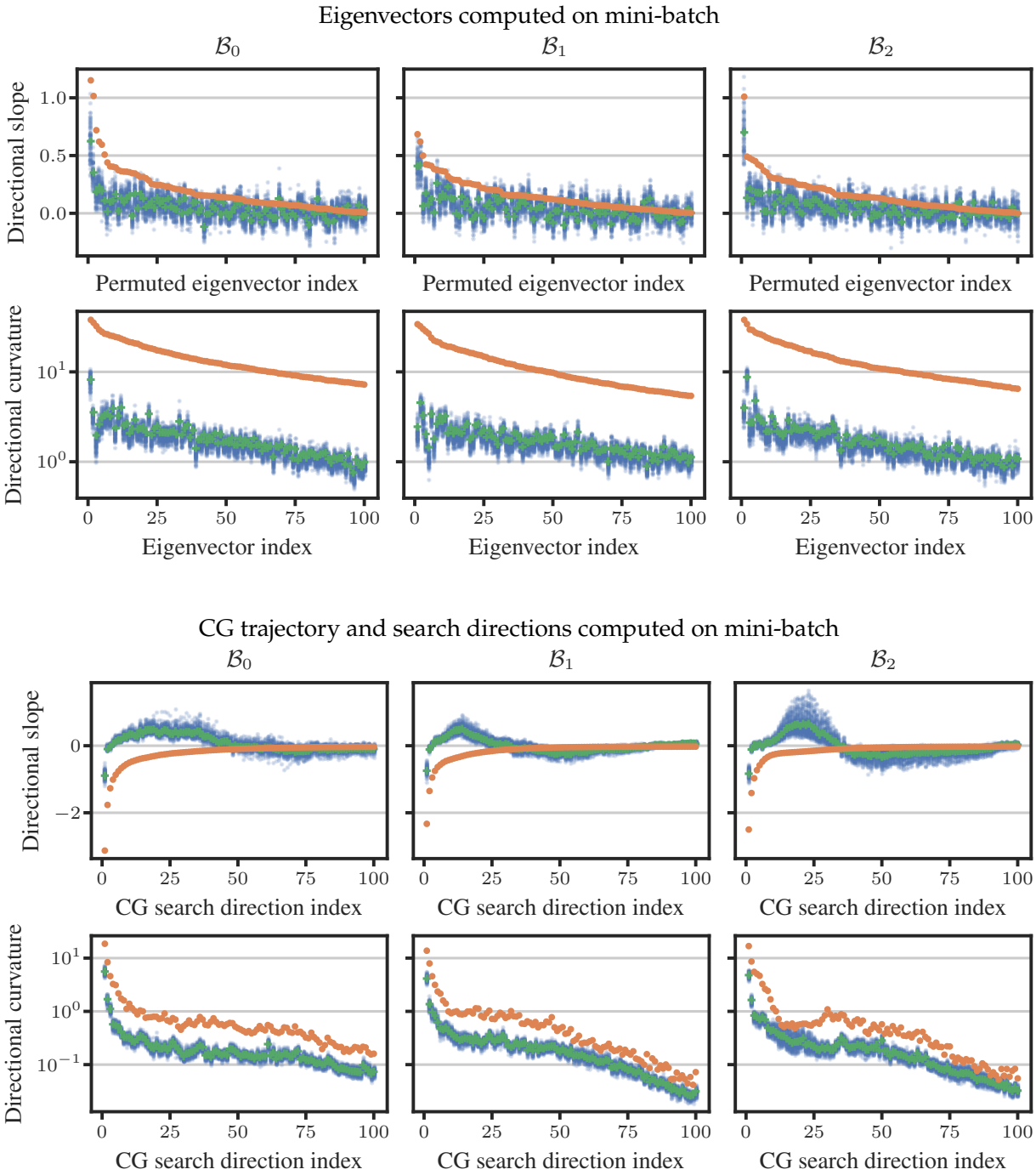


Figure C.3: Directional Slopes and Curvatures Are Biased. The experimental setting is the same as in Figure 7.2: We use the GGN curvature proxy $H_{\mathcal{B}} \leftarrow G_{\mathcal{B}} + \beta I$ at batch size 512. The upper plot *(Top)* shows the directional derivatives along the top 100 eigenvectors of $H_{\mathcal{B}}$, the lower plot *(Bottom)* shows the directional derivatives along the first 100 CG search directions. For the top panel of the upper plot, we switch the order and sign of the eigenvectors such that the orange dots are all above zero and in descending order. There are strong systematic biases in the directional slopes and curvatures. The curvature biases are more pronounced along the eigenvectors, whereas the biases in the slope are larger along the CG directions.

Biases: Additional Results for $H_{\mathcal{B}} \leftarrow G_{\mathcal{B}} + \beta I$ (Batch Size 2048).

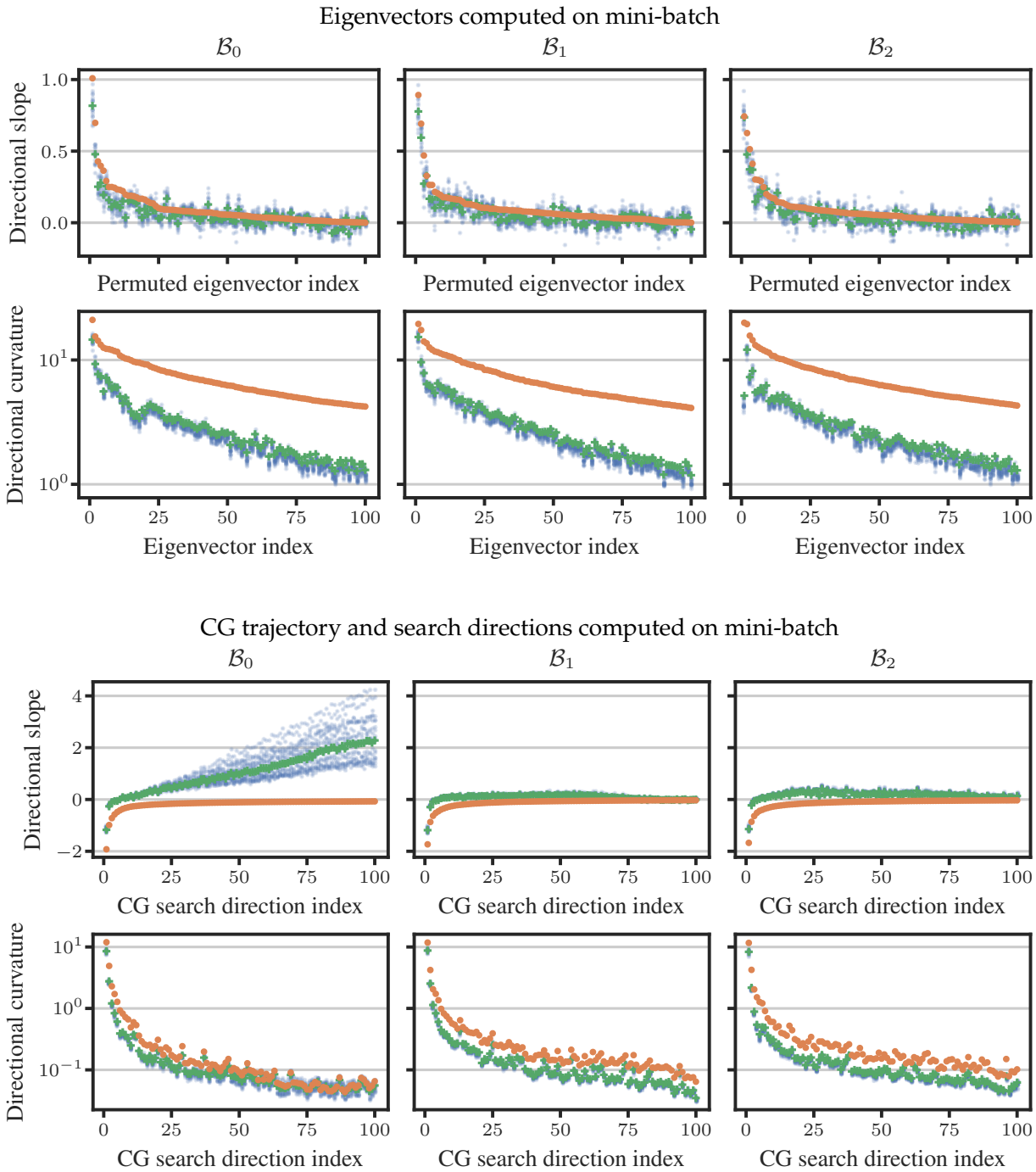


Figure C.4: Directional Slopes and Curvatures Are Biased. The experimental setting is similar to Figure 7.2: We use the GGN curvature proxy $H_{\mathcal{B}} \leftarrow G_{\mathcal{B}} + \beta I$ but batch size 2048. The upper plot *(Top)* shows the directional derivatives along the top 100 eigenvectors of $H_{\mathcal{B}}$, the lower plot *(Bottom)* shows the directional derivatives along the first 100 CG search directions. For the top panel of the upper plot, we switch the order and sign of the eigenvectors such that the orange dots are all above zero and in descending order. There are strong systematic biases in the directional slopes and curvatures. The curvature biases are more pronounced along the eigenvectors, whereas the biases in the slope are larger along the CG directions.

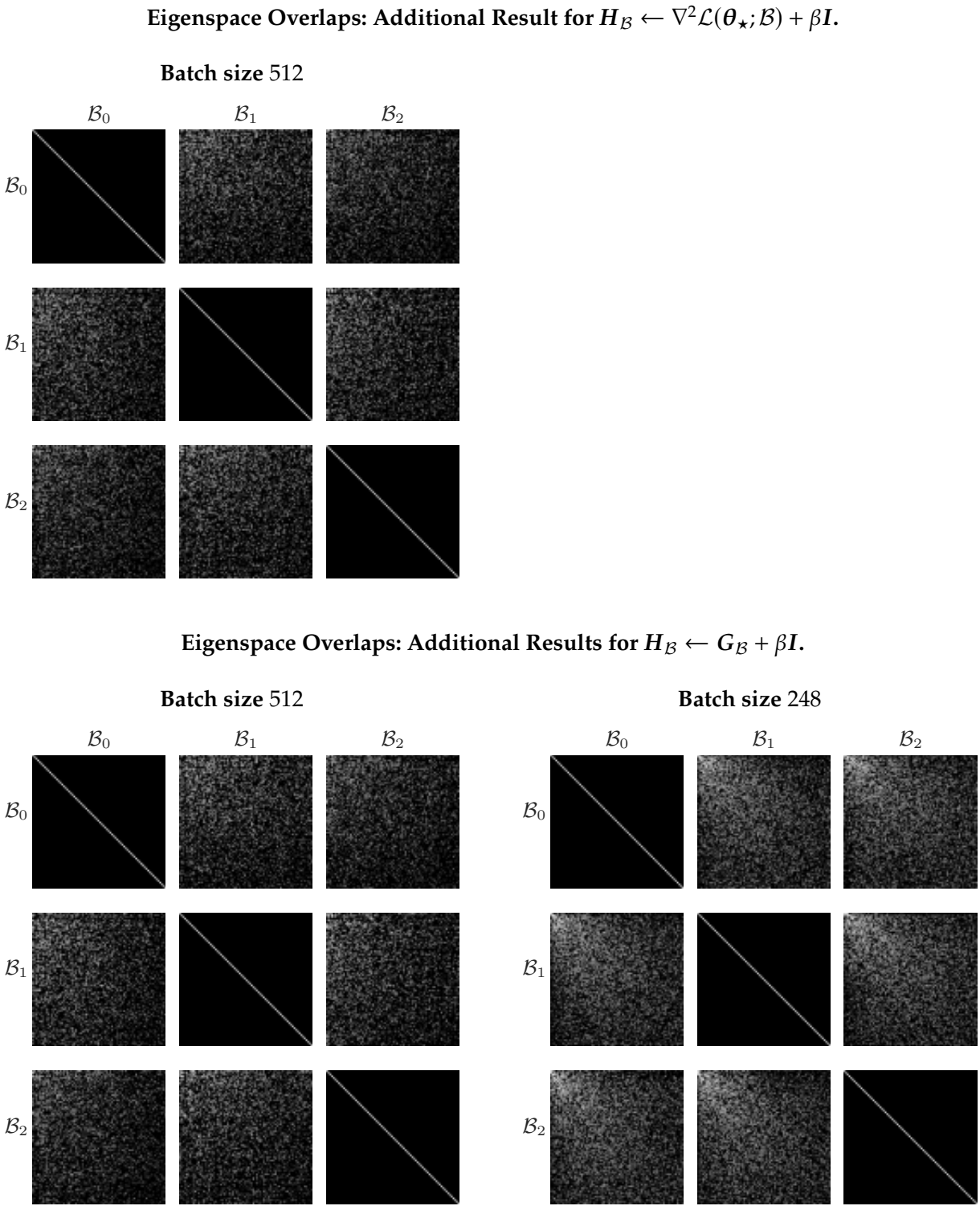


Figure C.5: In Practice, Eigenspaces Are Misaligned. The experimental setting is similar to Figure 7.3a. The top plot shows overlaps between three curvature matrices that use the Hessian of the empirical risk, *i.e.* $\boldsymbol{H}_{\mathcal{B}} \leftarrow \nabla^2 \mathcal{L}(\boldsymbol{\theta}_\star; \mathcal{B}) + \beta \boldsymbol{I}$ at batch size 512. The bottom plots use the GGN curvature proxy $\boldsymbol{H}_{\mathcal{B}} \leftarrow \boldsymbol{G}_{\mathcal{B}} + \beta \boldsymbol{I}$ at batch size 512 *(Left)* and 2048 *(Right)*. The weights $\Omega_{i,j}$ are shown as a 100×100 grayscale image (color ranges from black for $\Omega_{i,j} \leq 10^{-8}$ to white for $\Omega_{i,j} = 1$) for $m, m' \in \{0, 1, 2\}$. Clearly, the eigenspaces for different mini-batches are not perfectly aligned as eigenvectors from $\mathcal{B}_m$ overlap with several eigenvectors from $\mathcal{B}_{m'}$. At the larger batch size, the eigenspaces are more aligned.

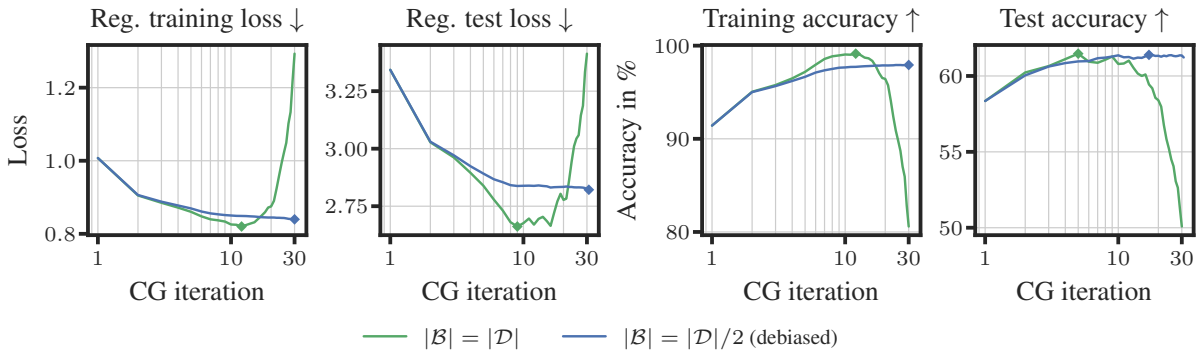
products (30 for the search directions and 30 for the update magnitudes) at half the computational cost (since the cost for the GGN-vector product scales linearly with the mini-batch size), the total cost of both approaches are comparable.
For both approaches, the above procedure is repeated for 5 different mini-batches/mini-batch pairs, such that 10 trajectories are computed in total.
We compute two additional trajectories: The standard CG trajectory for the full-batch quadratic $q(\cdot\,; \mathcal{D})$ and the trajectory for a debiased full-batch approach, where we use half the training data for the directions and the other half for the update magnitudes.

2. **Evaluation of Performance Metrics.** For each of the 12 trajectories, we evaluate four performance metrics: (i) the regularized loss $\mathcal{L}_{\text{reg}}$ (see Equation (7.1)) on the training set $\mathcal{D}$ and (ii) on the test set $\mathcal{D}_{\text{test}}$ as well as the accuracy (*i.e.* the relative number of correctly classified samples) on the (iii) training and (iv) test data set.

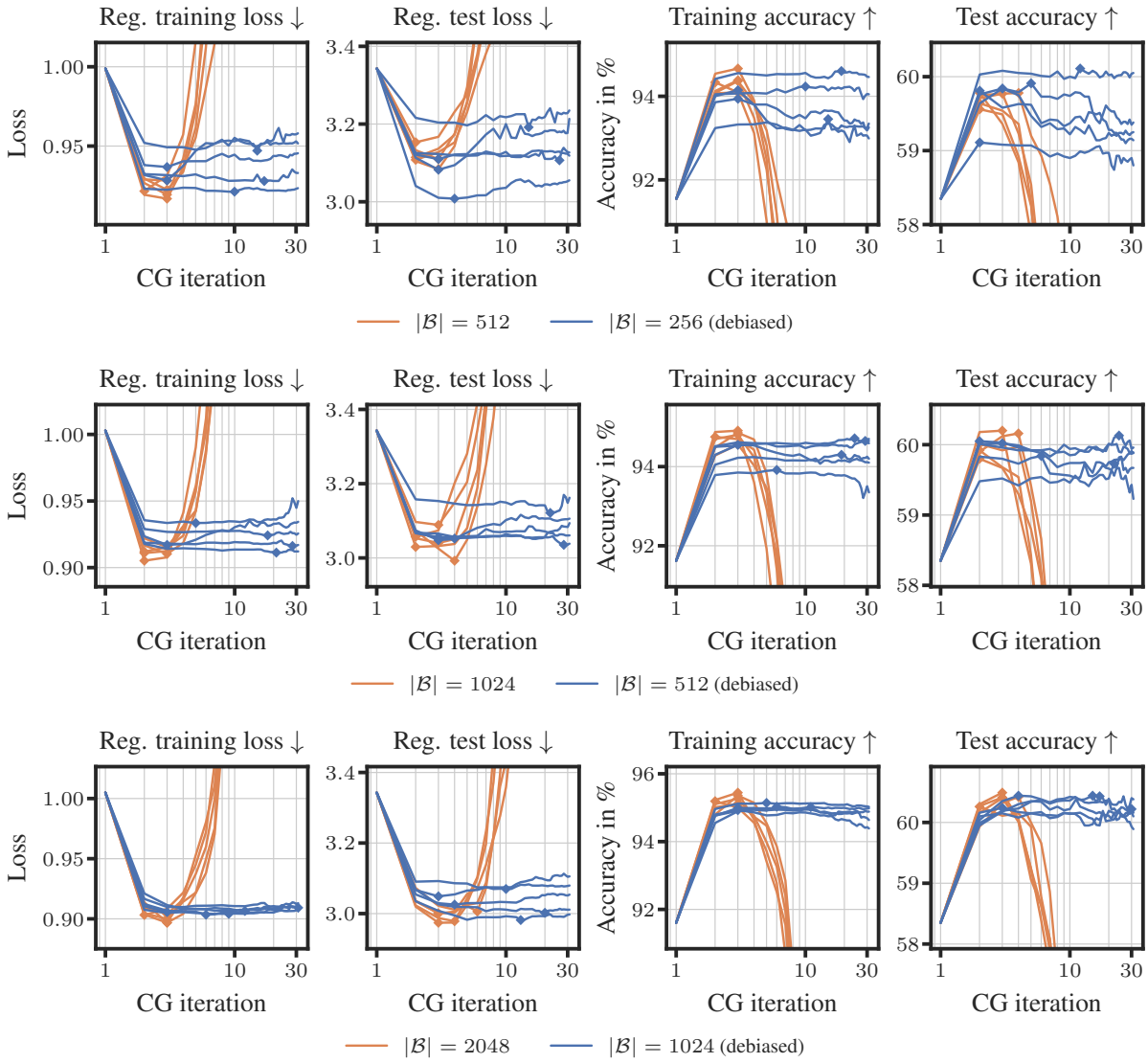
Results. The mini-batch results are shown in Figure 7.4 and discussed in Section 7.6.1. The results for the two full-batch CG variants are shown in Figure C.6a. Surprisingly, the full-batch CG approach diverges after roughly 10 iterations. If we consider the training data as a (large) sample from the data distribution, the full-batch CG approach can be seen as an instance of the mini-batch approach with a very large mini-batch size. Thus, it might also exhibit the associated biases we describe in Section 7.3.2. Another explanation would be the approximation error in Equation (7.2): $\mathcal{L}_{\text{reg}}(\cdot\,; \mathcal{D}) \approx q(\cdot\,; \mathcal{D})$. If the trajectory moves far away from $\boldsymbol{\theta}_\star$, it leaves the region where the approximation is valid (whereas the debiased full-batch version might take smaller steps and thus remains stable). We leave it to future work to investigate this further.

Additional Results. We also provide the results for two other mini-batch sizes $|\mathcal{B}| \in \{512, 2048\}$ (the debiased approach uses two mini-batches of size 256 and 1024, respectively). The results are shown in Figure C.6b.

Run Time and Memory Consumption. Our current implementation uses the naive approach that requires *two* matrix-vector products with the GGN for computing one debiased update magnitude (see Appendix C.1.2 for details). As this would skew the run time comparison, we refrain from providing concrete numbers here. However, as we describe in detail in Appendix C.1.2, it is possible to implement the debiased approach in a more efficient way that only requires one additional matrix-vector product for each CG direction to compute the debiased update magnitude. Using the debiased approach at half the mini-batch size, this brings the computational costs down to roughly the same level as the



(a) Comparison of Full-Batch CG Approaches. We use the setting from Figure 7.4. The full-batch approach (shown as —) applies standard CG to $q(\cdot\,; \mathcal{D})$ while the debiased approach (shown as —) uses one half of the training data for the directions and the other half for the update magnitudes. The markers ◆ and ◆ are placed at peak performance. Surprisingly, the full-batch run diverges. The debiased CG run is stable.



(b) Comparison of CG Approaches at Different Batch Sizes. We use the same experimental setting as in Figure 7.4 but consider different mini-batch sizes $|\mathcal{B}| \in \{512, 1024, 2048\}$ (from *(Top)* to *(Bottom)*). The debiased approach uses two mini-batches of size 256, 512, and 1024, respectively. Across all mini-batch sizes, the single-batch CG runs diverge quickly while our debiased approach maintains stability.

standard approach (assuming that the run time of a GGN-vector product scales linearly with the mini-batch size).

As explained in detail in Appendix C.1.2, the memory overhead for the efficiently implemented debiased CG approach is two additional vectors of size P. For the experimental setting used for Figure 7.4, the number of parameters is $P = 1\,387\,108$, which corresponds to 5.55 MB in single precision. The computational overhead incurred by the debiased approach is thus about 11.1 MB.

C.2.6 Section 7.6.2 (Debiased Laplace Approximation): Figure 7.5

Figure 7.5 shows a comparison of the vanilla model, the mini-batch K-FAC LA, the debiased version, and the full-batch LA. Here, we describe the experiment in more detail and present additional results, in particular on OOD data. For the derivation and the mathematical details of these approaches, see Appendix C.1.3. The experiment uses the fully trained ALL-CNN-C model on CIFAR-10 data from test problem (B) (see Appendix C.2.1).

Experimental Procedure. The experimental procedure consists of three steps: The computation of the K-FAC curvature approximations, the evaluation of the corresponding predictive class probabilities, and the computation of the performance metrics.

1. **K-FAC Curvature Approximations.** We consider a log-equidistant grid of 13 prior precisions β between 10^{-4} and 10^{0} and add $\beta = 10$.
   For each of those 14 values, we compute the K-FAC curvature approximation via BackPACK [28] (see Appendix C.2.2) using (i) a single batch of size 64, 256, and 1024, (ii) the two-batch debiased LA version (details in Appendix C.1.3) at batch sizes 32, 128, and 512, and (iii) the full-batch LA. For the latter, we accumulate mini-batch K-FAC approximations over the entire training set. As an additional baseline, we consider (iv) the vanilla model (without LA), which is independent of the prior precision β. For approaches (i) and (ii), we repeat the experiment for 5 different mini-batches/mini-batch pairs.
2. **Evaluation of Predictive Class Probabilities.** The first step in the evaluation of the performance metrics is the computation of the predictive uncertainty. For a given test data set of size $N_\diamond$, these can be represented as a matrix $\boldsymbol{P} \in \mathbb{R}^{N_\diamond \times C}$, where $\boldsymbol{P}_{n,c}$ is the probability that the n-th sample belongs to class c, *i.e.* rows of $\boldsymbol{P}$ sum to 1. In all experiments, we draw $S = 40$ MC samples $\{\boldsymbol{\theta}^{(s)}\}_{s=1}^{S}$ from the weight posterior following the procedure in Appendix C.1.3. The predictive class probabilities are then obtained via Equation (C.9). In the case

of the vanilla model without LA, the sum in Equation (C.9) collapses to a single term corresponding to the MAP model. The evaluation procedure above can be applied to arbitrary test data sets. We consider the training data, test data and CIFAR-10-C (*i.e.* OOD data sets at severity 1 to 5).

3. **Performance Metrics.** We consider the following performance metrics:
   - **Accuracy.** The predictive classes are obtained from $\boldsymbol{P}$ by extracting the class with the highest probability. The accuracy is the relative number of correctly classified samples.
   - **Negative Log-Likelihood (NLL).** For each datum in the test set, we compute the negative log-probability of the true class. We then average over all samples in the test data set. This coincides with the empirical risk from Equation (7.1), evaluated on the test data set.
   - **Expected Calibration Error (ECE).** The ECE is a measure of the calibration of the model's predictive probabilities. It groups the classification confidences (*i.e.* the maximum entry in each of $\boldsymbol{P}$'s rows) into bins, and within these bins, compares the average confidence with the actual accuracy. We use `MulticlassCalibrationError` from `torchmetrics` [33].
   - **AUROC.** For the OOD data sets, we provide the Area Under the Receiver Operating Characteristic curve (AUROC). Our goal is to distinguish in-distribution (ID) and out-of-distribution (OOD) samples using the model's uncertainty. We use the entropy of the predictive distribution $p(\boldsymbol{y}_\diamond \mid \boldsymbol{x}_\diamond, \mathbb{D})$ as our uncertainty estimate $u(\boldsymbol{x}_\diamond) \in \mathbb{R}$ for input $\boldsymbol{x}_\diamond$. The ground-truth binary labels are the ID/OOD indicators of the test inputs. Several thresholds ξ could be established to turn the scalar uncertainty estimates into binary predictions by using $\mathbf{1}_{\{u(\boldsymbol{x}_\diamond) > \xi\}}$. Instead of choosing a particular threshold and evaluating its accuracy, the AUROC metric directly evaluates $u(\cdot)$ by measuring the area under the plot of the true positive rate (TPR) against the false positive rate (FPR) for all possible thresholds.

Additional Experimental Details.

- **Uncertainty over Weights but Not Biases.** As the prior acts only on the weights of the network but not its biases (see Appendix C.2.1), we only consider the uncertainty over the weights in the LA. This slightly reduces the size of the covariance matrix as it excludes the bias parameters.
- **Single vs. Double Precision.** Although the K-FAC factors are positive semidefinite by construction, their numerical eigenvalues can be negative due to numerical inaccuracies. To balance precision and computational cost, we use double precision for all computations until drawing the MC samples and single precision afterwards.

Results. The results are presented in Figures 7.5, C.7 and C.8. For the single-batch and debiased approach (where we use 5 mini-batches/mini-batch pairs each), we report the average performance as a dot and the min/max as a vertical line. The performance of the vanilla model is shown as a horizontal line as its performance does not depend on β, as explained above.

Across all data sets and performance metrics, the debiased LA mimics the behavior of the full-batch approach much better than the single-batch LA (although both approaches use a comparable amount of computational resources). In particular, for small prior precisions, where the covariance matrix relies almost exclusively on the K-FAC curvature information, the debiased LA maintains a good performance in contrast to the single-batch approach.

Additional Results.

- **ResNet-50 on ImageNet.** To showcase the scalability of our debiased LA approach, we repeat the experiment for a ResNet-50 model on the ImageNet data set (test problem (E) in Appendix C.2.1). The results are shown in Figure C.9. Again, the debiased approach behaves similarly to the full-batch LA and maintains stability over the entire spectrum of prior precisions.
- **ViT Little on ImageNet.** Similar to the previous experiment, we also apply our LA debiasing approach to a ViT Little model on the ImageNet data set (test problem (F) in Appendix C.2.1) to ensure that it is beneficial for other architectures as well. The results are shown in Figure C.10. The debiased approach mimics the calibration behavior of the full-batch LA remarkably well and remains stable for lower prior precision values as well where the single-batch approach behaves erratically.

Run Time Analysis (Computational Overhead of Debiasing). We claim in Section 7.6.2 that, by using the debiased approach at

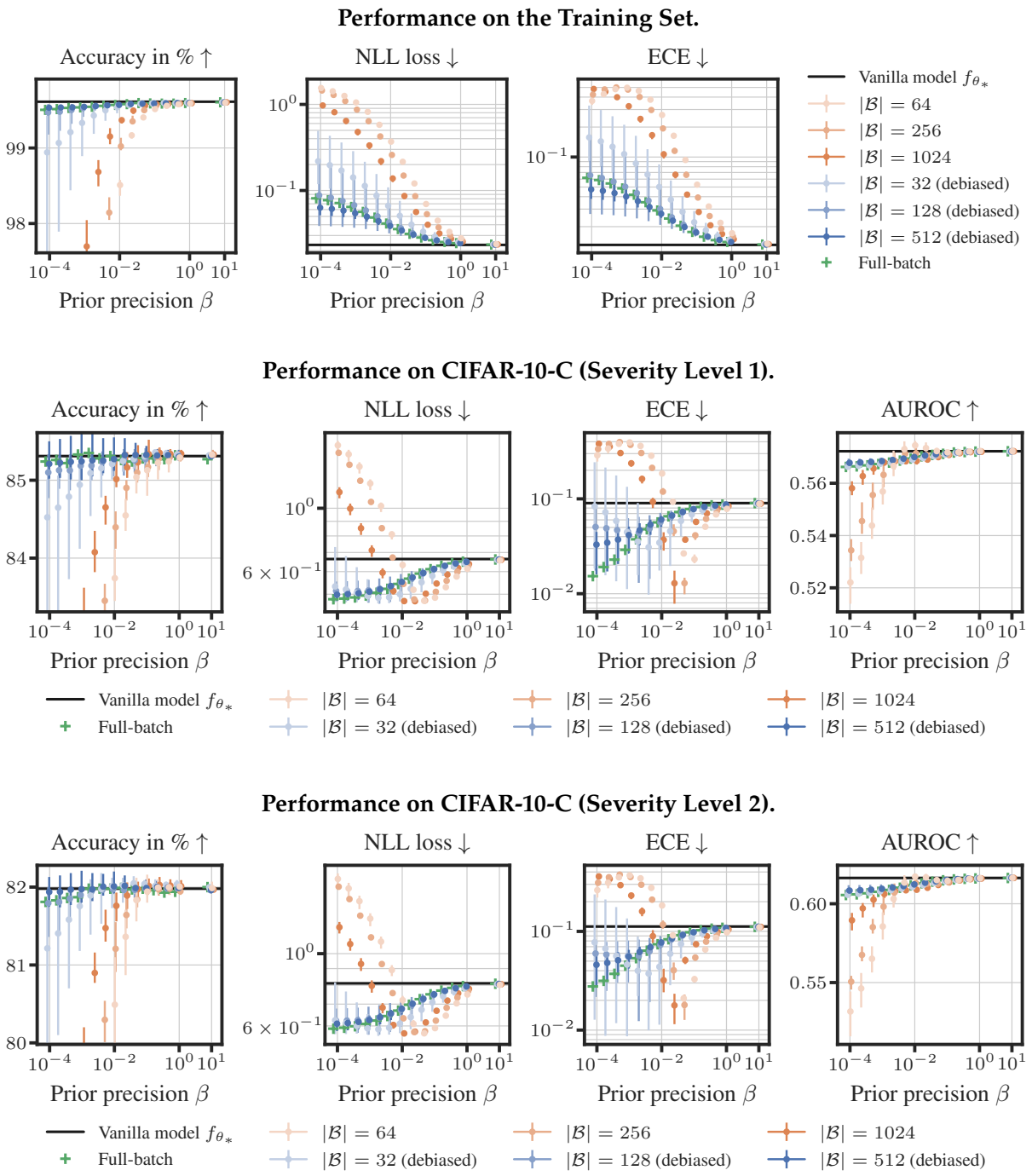


Figure C.7: Debiased LA Mimics the Full-Batch LA Very Well. The experimental setting is the same as in Figure 7.5, but we report results on additional data sets (training set and OOD data sets at severity level 1 and 2). For the OOD data sets, we report the AUROC metric in addition to the accuracy, NLL and ECE. In contrast to the single-batch approach, the debiased version mimics the behavior of the full-batch approach very well over the entire range of prior precisions and across data sets.

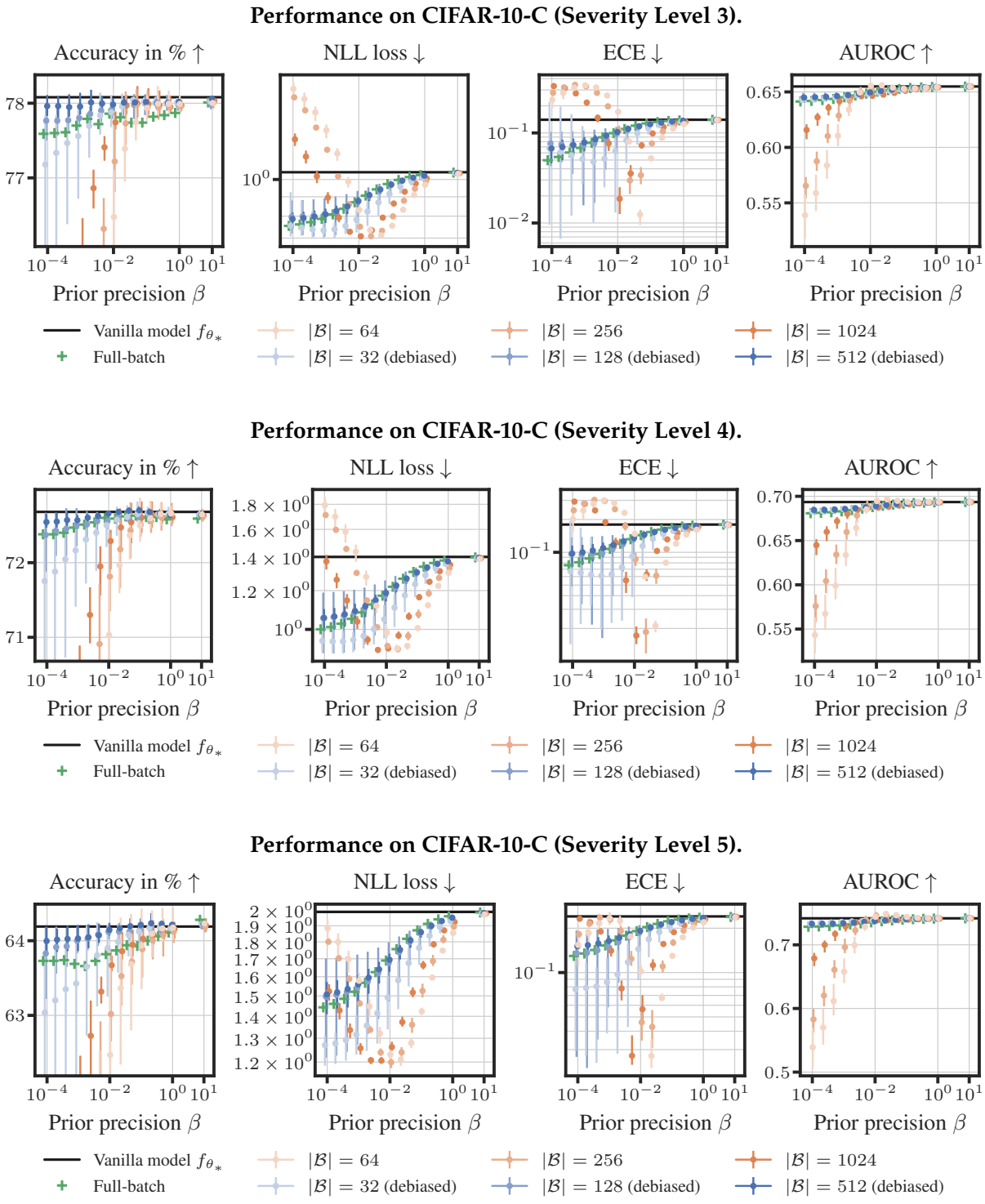


Figure C.8: Debiased LA Mimics the Full-Batch LA Very Well. The experimental setting is the same as in Figure 7.5, but we report results on additional data sets (OOD data sets at severity level 3, 4, and 5). We report the AUROC metric in addition to the accuracy, NLL, and ECE. In contrast to the single-batch approach, the debiased version mimics the behavior of the full-batch approach very well over the entire range of prior precisions and across data sets.

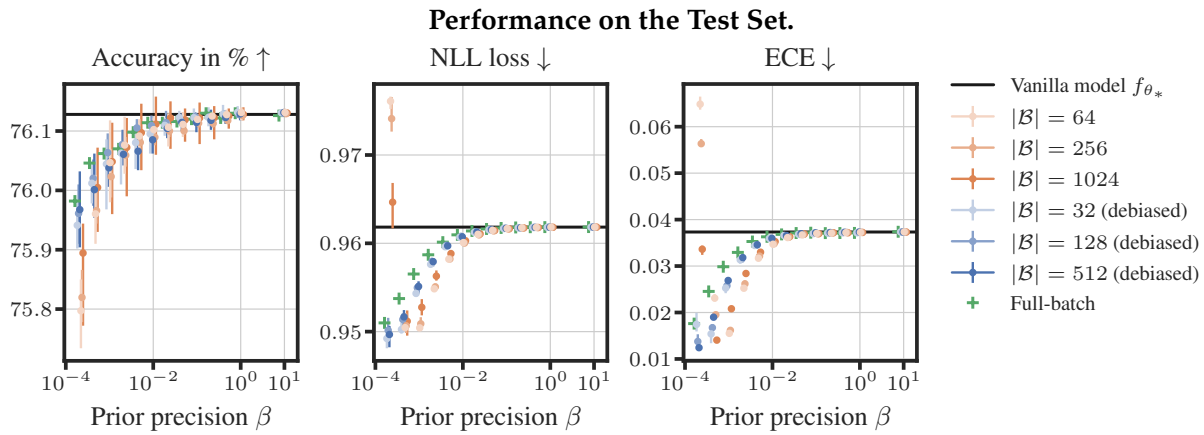


Figure C.9: Our Approach Scales to ResNet-50 on ImageNet. The experimental setting is the same as in Figure 7.5, but we use test problem (E) (see Appendix C.2.1) (ResNet-50 on ImageNet). The results are consistent with the findings on CIFAR-100: The debiased approach behaves similarly to the full-batch LA and maintains stability even for small prior precisions.

half the mini-batch size that is used by the single-batch approach, we obtain a fair comparison (more details in Section 7.4.2). Here, we substantiate this claim by providing a detailed run time comparison of the biased, debiased, and full-batch LA approaches on CIFAR-10.

Results (Run Times). We report the run times of the biased (▭), debiased (▭), and full-batch (▭) K-FAC Laplace approximation schemes in Figure C.11. We find that (i) both biased and debiased approaches are orders of magnitude cheaper than the full-batch LA, and (ii) the debiased approach introduces only negligible overhead and can, sometimes, even improve upon the run time of the biased approach.

Memory Consumption. The All-CNN-C model we use for the LA experiment in Section 7.6.2 has $P = 1\,368\,480$ trainable parameters (this amounts to 10.95 MB of memory in double precision). The K-FAC approximation requires storing 11 483 965 (91.87 MB) numbers to represent all Kronecker factors. The memory of a K-FAC approximation is thus roughly equivalent to that of 8.4 models. As the debiased approach is based on *two* such approximations in the case of a naive implementation, the overhead of debiasing is another 91.87 MB of memory. However, this can be significantly improved by building up the K-FAC approximations block by block: Having two corresponding blocks available (based on two different mini-batches) already suffices to compute the respective debiased block (see Appendix C.1.3). This way, we only have two blocks (represented by their Kronecker factors) in memory at the same time (instead of two entire K-FAC approximations) which greatly reduces the memory overhead of debiasing, down to the largest block. For the All-CNN-C model, this is two Kronecker factors of sizes 192×192 and 1728×1728 amounting to 3 022 848 numbers in total, *i.e.* the equivalent of 2.20 models or 24.18 MB

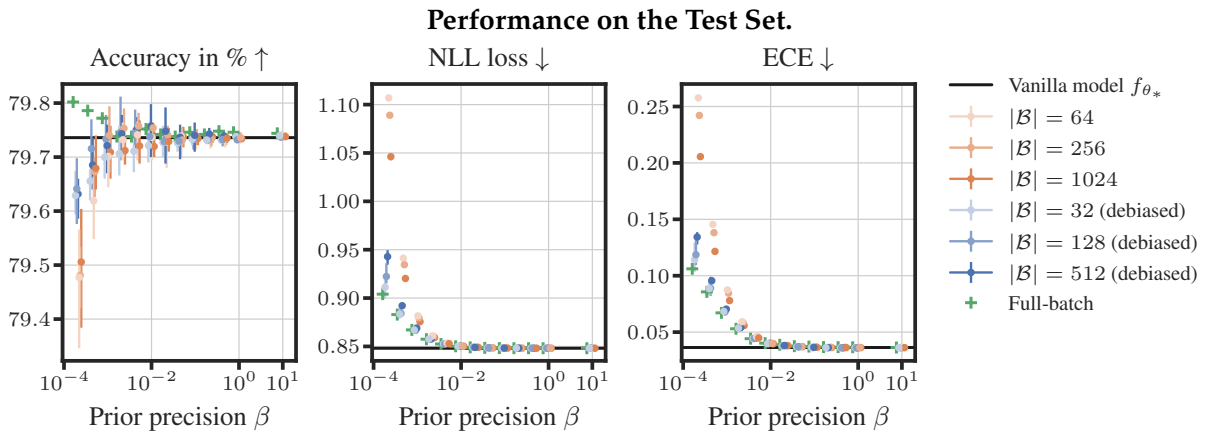


Figure C.10: Curvature Biases Are Also Present for the ViT Little Architecture on ImageNet. The experimental setting is the same as in Figure 7.5 and Figure C.9, but we use test problem (F) (see Appendix C.2.1) (ViT Little on ImageNet). Again, the debiased approach behaves similarly to the full-batch LA and maintains stability even for small prior precisions (where the covariance is almost exclusively based on the K-FAC curvature information).

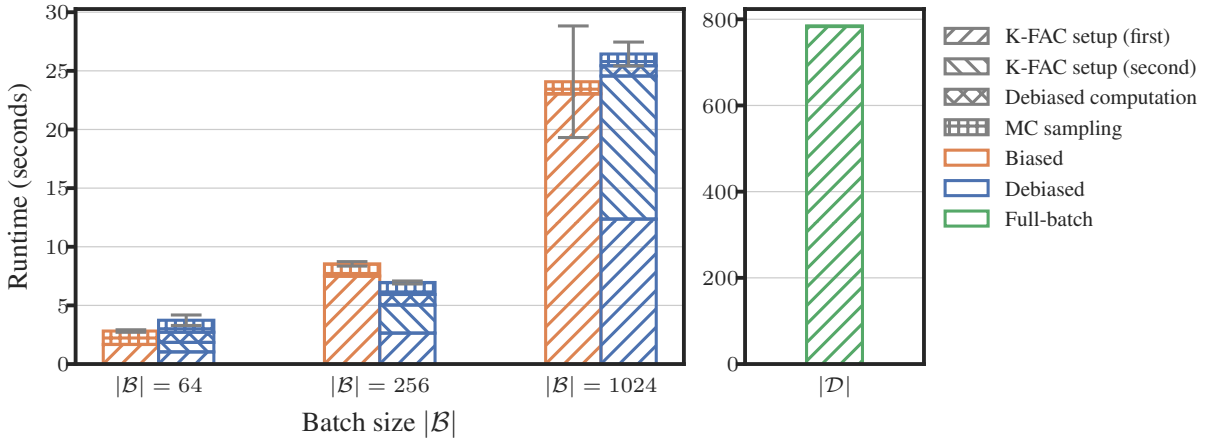


Figure C.11: The Computational Overhead of Debiasing Is Negligible for Laplace Approximations. We follow the experimental setup of Section 7.6.2 and provide a detailed run time comparison of the biased (▭), debiased (▭), and full-batch (▭) K-FAC Laplace approximation schemes on CIFAR-10. The total run time includes the construction of K-FAC's Kronecker factors (on one or two mini-batches for the biased and debiased approach, respectively), the debiasing computations (see Appendix C.1.3), and the Monte Carlo sampling of $S = 40$ weights. All reported run times are averages over 5 runs. The error bars for the total run time cover one standard deviation. The results show that the debiased approach introduces a negligible overhead. Compared to the full-batch variant, both mini-batch approaches are orders of magnitude faster. Note that the evaluation (i.e., the computation of the predictive uncertainty) is not included in the comparison, as the time requirements are the same for all approaches; however, this step requires the most computational resources.

in double precision. The statement that the debiased approach roughly doubles the memory consumption is thus a worst-case scenario.

C.2.7 Additional Experiment: Biases for a WideResNet 40-4 Model Architecture

Experimental Setting & Results. Here, we extend our analysis from Section 7.3.1 (details in Appendix C.2.4) to test problem (C) (see Appendix C.2.1): A Wide ResNet model on CIFAR-100. We use

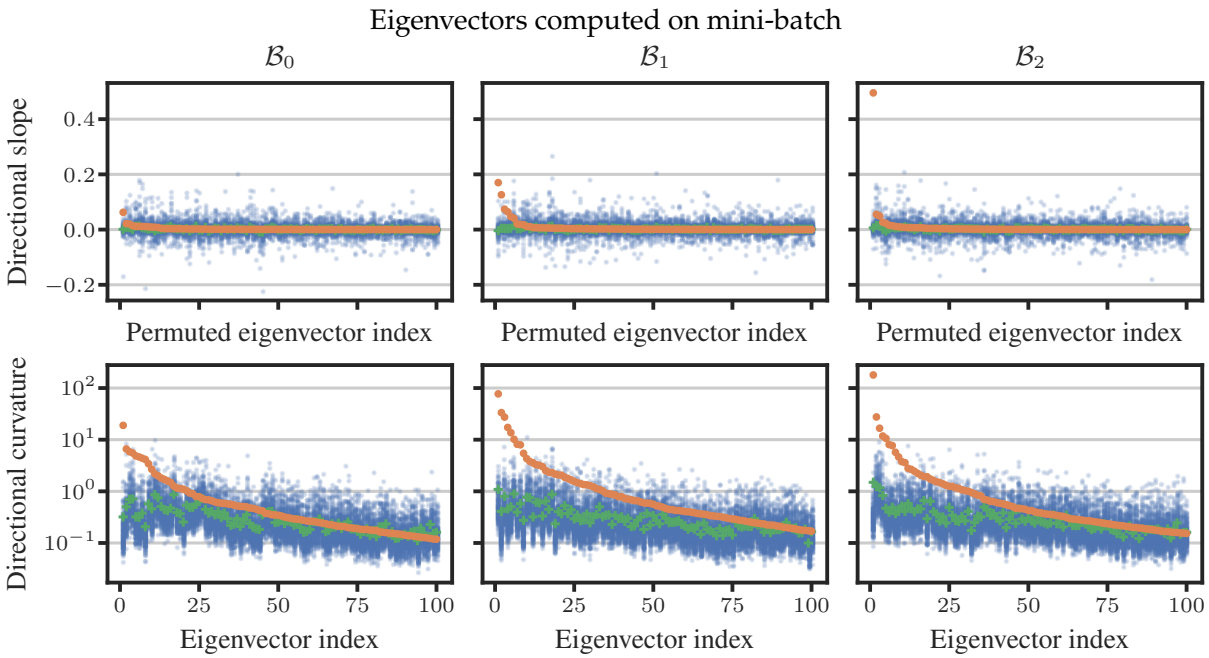


Figure C.12: Directional Slopes and Curvatures Are Biased. The experimental setting is similar to Figure 7.2, but we use test problem (C) (Wide ResNet on CIFAR-100) with the GGN curvature proxy $\boldsymbol{H}_{\mathcal{B}} \leftarrow \boldsymbol{G}_{\mathcal{B}} + \beta \boldsymbol{I}$ at batch size 256. For the top panel, we switch the order and sign of the eigenvectors such that the orange dots are all above zero and in descending order. There is a strong, systematic bias, particularly in the curvature: Computing the eigenvectors and directional curvatures on the same data results in over-estimation that is very pronounced along the first few eigenvectors but then decays more quickly than in test problem (A).

the GGN curvature proxy $\boldsymbol{H}_{\mathcal{B}} \leftarrow \boldsymbol{G}_{\mathcal{B}} + \beta \boldsymbol{I}$ at batch size 256 and compute the directional slopes and curvatures along the top 100 eigenvectors of $\boldsymbol{H}_{\mathcal{B}}$. The results are shown in Figure C.12. Along the first few top-curvature directions, the curvature biases are even larger for this test problem than for the All-CNN-C model on CIFAR-100, but then they decay more quickly.

C.2.8 Additional Experiment: K-FAC and the Dependence of the Biases on Mini-Batch Size

Experimental Setting. Here, we extend our analysis of test problem (A) (see Appendix C.2.1) from Section 7.3.1 to K-FAC and investigate the impact of the mini-batch size on the curvature biases.

Results & Discussion. Figure C.13 shows the directional curvatures of the K-FAC approximation for four different mini-batch sizes. When we compute the eigenvectors and directional curvatures on the same mini-batch, we observe a systematic curvature bias that decreases with increasing mini-batch size. There are two phenomena at play: With increasing mini-batch size, (i) the green crosses move upwards and (ii) the orange dots move downwards. Our intuition for this is as follows: With increasing mini-batch size, the eigenvectors become more meaningful such that, on other data, they also exhibit large curvature—this explains (i). Similarly,

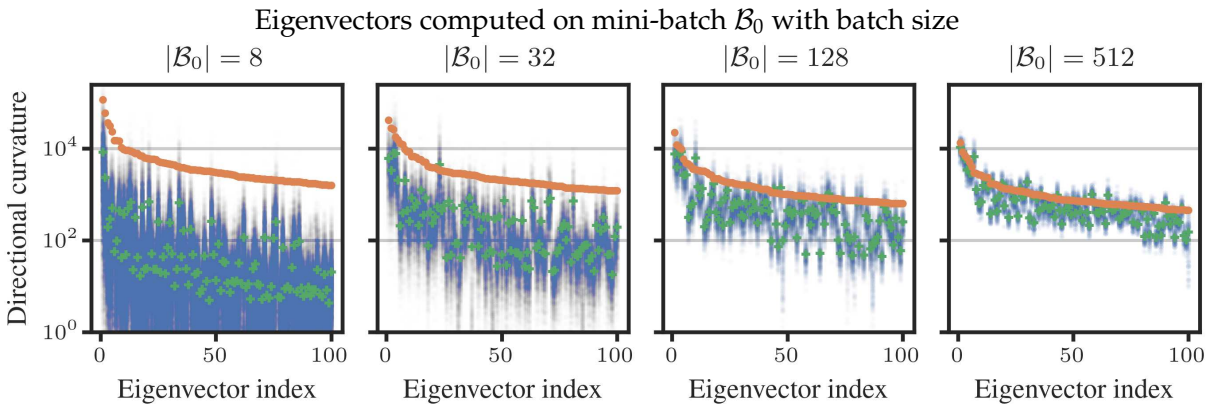


Figure C.13: Directional Curvatures with K-FAC. We use the CIFAR-100 data set with the fully trained All-CNN-C model. For each mini-batch size $\in \{8, 32, 128, 512\}$, we draw one mini-batch and compute the top 100 eigenvectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_{100}$ of the K-FAC matrix $\boldsymbol{K}_{\mathcal{B}_0}$. We show the directional curvatures $\boldsymbol{u}_p^\top \boldsymbol{K}_{\mathcal{B}_m} \boldsymbol{u}_p$, $p \in \{1, \dots, 100\}$ on the same mini-batch ($m = 0$) as ●, all other mini-batches in the training set ($m \in \{1, 2, \dots\}$) as ● and their average as ✚. Similar to Figure 7.2, we observe a systematic bias in the curvature. The bias decreases with increasing mini-batch size.

with increasing mini-batch size, it gets harder to find directions of *extreme* curvature as these directions have to exhibit large curvature on *all* data points within the mini-batch—this explains (ii).

C.2.9 Additional Experiment: Development of the Biases over the Course of Training

Experimental Setting & Results. Here, we investigate how the biases in the slope and curvature develop over the course of training for test problem (A) (see Appendix C.2.1). We use the same procedure as in Appendix C.2.4 but evaluate the biases at 10 different checkpoints during training (spread log-equidistantly between the first and last epoch). At each checkpoint, we draw 5 mini-batches, compute the top 100 eigenvectors of the corresponding GGN-based quadratics, and finally evaluate the *relative* errors in the directional slopes and curvatures (where the ground truth is the full-batch quadratic's slope/curvature). For each mini-batch, this distribution of 100 relative curvature and slope biases is represented as a dot (at the mean relative bias) and a vertical line ranging from the 25% to the 75% percentile in Figure C.14. While the biases in the slope remain relatively stable over the course of training, the biases in the curvature increase over more than 3 orders of magnitude up to a relative error of order 10 in epoch 349 (which is consistent with the results in Figure 7.2). This suggests that the eigenspaces of different curvature matrices become more and more misaligned as training progresses steadily increasing the need for effective debiasing strategies.

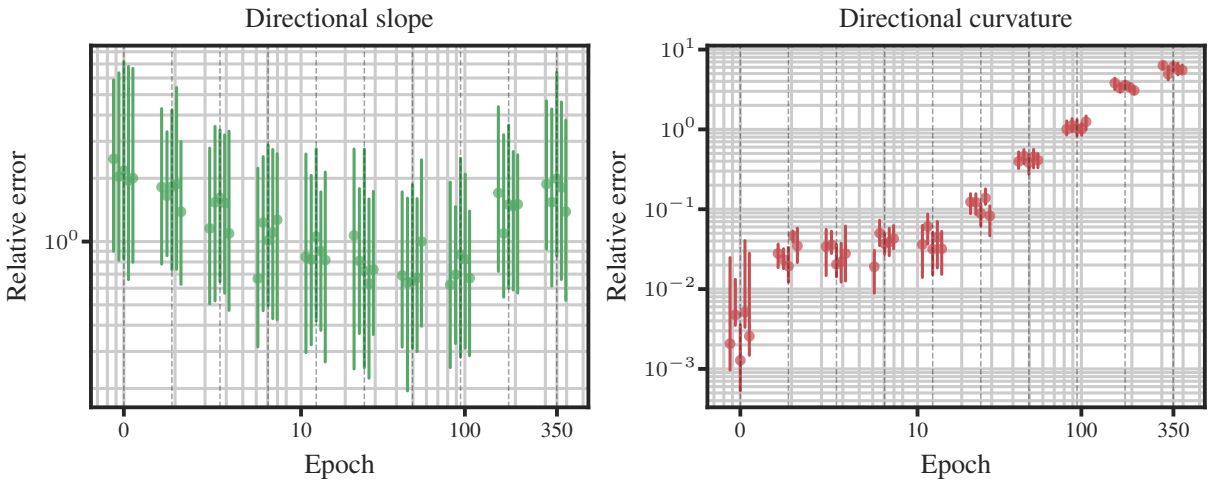


Figure C.14: Curvature Biases Increase over the Course of Training. We evaluate the relative slope and curvature biases at different checkpoints during training for the ALL-CNN-C model on CIFAR-100. 5 mini-batches are drawn per checkpoint. For each mini-batch, the relative biases are represented as a dot ● (at the mean relative bias) and a vertical line — ranging from the 25% to the 75% percentile. While the biases in the slope remain relatively stable over the course of training, the biases in the curvature increase (over more than 3 orders of magnitude).

C.2.10 Additional Experiment: The Curvature Bias Increases with P

In high-dimensional spaces, it becomes increasingly unlikely that random vectors are aligned. While the eigenspaces of the curvature matrices are not completely random, they are subject to noise. It is thus conceivable that their overlap decreases as the number of parameters P increases. If this hypothesis is true, the curvature biases should become more pronounced in large models.

Experimental Procedure & Results. To test this hypothesis, we use test problem (D) that implements a simple convolutional neural network with variable width and depth (for details, see Appendix C.2.1). For each fully trained model, we evaluate the *relative* error between $\partial^2_{u_p} q(\boldsymbol{\theta}_\star; \mathcal{B})$ and $\partial^2_{u_p} q(\boldsymbol{\theta}_\star; \mathcal{D})$ for the $\boldsymbol{G}_\mathcal{B}$'s top 100 eigenvectors at batch size 128. This procedure is repeated for 5 different mini-batches for each of the 9 models, resulting in a total of 45 error distributions (see Figure C.15). The results confirm our hypothesis: The relative errors tend to increase with the number of parameters P. In the massively overparameterized regime, the biases might thus become even more relevant and effective debiasing strategies are needed.

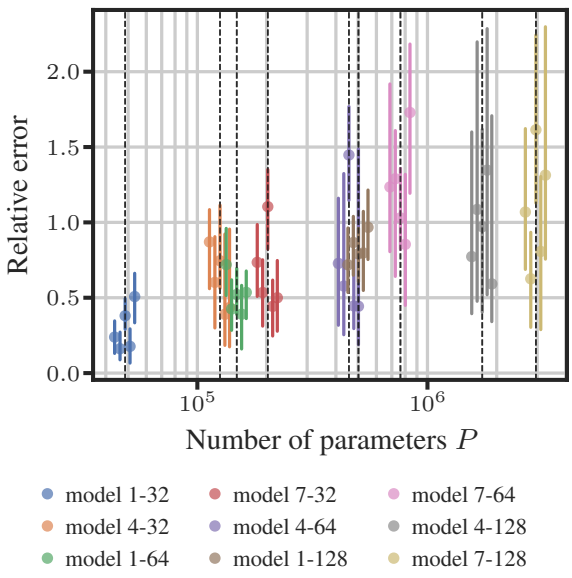


Figure C.15: Increasing Curvature Biases with P. We train 9 convolutional neural networks with different widths and depths for 100 epochs on the CIFAR-10 data set using Adam with standard hyperparameters. For the fully trained model, we evaluate the relative error $|\partial^2_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{B}) - \partial^2_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{D})| \cdot |\partial^2_{\boldsymbol{u}_p} q(\boldsymbol{\theta}_\star; \mathcal{D})|^{-1}$ for $\boldsymbol{G}_\mathcal{B}$'s top 100 eigenvectors at batch size $|\mathcal{B}| = 128$. 5 different mini-batches are used per model resulting in a total of 45 error distributions (each consisting of 100 numbers). These distributions are represented by their median (as a dot ●) and the 25% and 75% percentiles (as a line segment —) P. The 5 distributions for each model are slightly spread along the x-axis for better visibility. The experiment confirms our hypothesis: The relative errors tend to increase with the number of parameters P.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". 2015.

[2] 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". *arXiv* (2018).

[3] S.-i. Amari. "Natural Gradient Works Efficiently in Learning". *Neural Computation* (1998).

[4] A. Andreyev and P. Beneventano. "Edge of Stochastic Stability: Revisiting the Edge of Stability for SGD". *arXiv* (2025).

[5] R. Anil, V. Gupta, T. Koren, K. Regan, and Y. Inger. "Scalable Second Order Optimization for Deep Learning". *arXiv* (2021).

[6] F. L. Bauer. "Computational Graphs and Rounding Error". *SIAM Journal on Numerical Analysis* (1974).

[7] F. Benzing. "Gradient Descent on Neurons and its Link to Approximate Second-order Optimization". *International Conference on Machine Learning (ICML)*. 2022.

[8] B. Bilodeau, Y. Tang, and A. Stringer. "On the Tightness of the Laplace Approximation for Statistical Inference". *arXiv* (2022).

[9] C. M. Bishop. "Pattern Recognition and Machine Learning". *Springer*, 2006.

[10] A. Botev, H. Ritter, and D. Barber. "Practical Gauss-Newton Optimisation for Deep Learning". *International Conference on Machine Learning (ICML)*. 2017.

[11] S. Boyd and L. Vandenberghe. "Convex optimization". *Cambridge university press*, 2004.

[12] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. "JAX: composable transformations of Python + NumPy programs" (2020).

[13] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. 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. 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". *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

[14] 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).

[15] A. B. Chan and D. Dong. "Generalized Gaussian process models". *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2011.

[16] B. Charlier, J. Feydy, J. A. Glaunès, F.-D. Collin, and G. Durif. "Kernel Operations on the GPU, with Autodiff, without Memory Overflows". *Journal of Machine Learning Research* (2021).

[17] 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).

[18] J. Cockayne, C. Oates, I. C. Ipsen, and M. Girolami. "A Bayesian Conjugate Gradient Method". *Bayesian Analysis* (2019).

[19] J. Cockayne, C. Oates, T. J. Sullivan, and M. Girolami. "Bayesian probabilistic numerical methods". *SIAM Review* (2019).

[20] J. Cohen, S. Kaur, Y. Li, J. Z. Kolter, and A. Talwalkar. "Gradient Descent on Neural Networks Typically Occurs at the Edge of Stability". *International Conference on Learning Representations (ICLR)*. 2021.

[21] J. P. Cunningham, K. V. Shenoy, and M. Sahani. "Fast Gaussian process methods for point process intensity estimation". *International Conference on Machine Learning (ICML)*. 2008.

[22] G. Cybenko. "Approximation by superpositions of a sigmoidal function". *Mathematics of Control, Signals and Systems* (1989).

[23] M. Dagréou, P. Ablin, S. Vaiter, and T. Moreau. "How to compute Hessian-vector products?" *International Conference on Learning Representations (ICLR) Blogposts*. 2024.

[24] G. E. Dahl, F. Schneider, Z. Nado, N. Agarwal, C. S. Sastry, P. Hennig, S. Medapati, R. Eschenhagen, P. Kasimbeg, D. Suo, J. Bae, J. Gilmer, A. L. Peirson, B. Khan, R. Anil, M. Rabbat, S. Krishnan, D. Snider, E. Amid, K. Chen, C. J. Maddison, R. Vasudev, M. Badura, A. Garg, and P. Mattson. "Benchmarking Neural Network Training Algorithms". *arXiv* (2023).

[25] F. Dangel. "Backpropagation Beyond the Gradient". PhD thesis. Universität Tübingen, 2022.

[26] F. Dangel, R. Eschenhagen, W. Ormaniec, A. Fernandez, L. Tatzel, and A. Kristiadi. "Position: Curvature Matrices Should Be Democratized via Linear Operators". *arXiv* (2025).

[27] F. Dangel, S. Harmeling, and P. Hennig. "Modular block-diagonal curvature approximations for feedforward architectures". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2020.

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

[29] 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)* (2023).

[30] E. Daxberger, A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig. "Laplace Redux–Effortless Bayesian Deep Learning". *NeurIPS*. 2021.

[31] E. Daxberger, E. Nalisnick, J. U. Allingham, J. Antoran, and J. M. Hernandez-Lobato. "Bayesian Deep Learning via Subnetwork Inference". *International Conference on Machine Learning (ICML)*. 2021.

[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A large-scale hierarchical image database". *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2009.

[33] N. S. Detlefsen, J. Borovec, J. Schock, A. H. Jha, T. Koker, L. Di Liello, D. Stancl, C. Quan, M. Grechkin, and W. Falcon. "TorchMetrics - Measuring Reproducibility in PyTorch". *Journal of Open Source Software* (2022).

[34] 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 (NeurIPS)*. 2017.

[35] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". *International Conference on Learning Representations (ICLR)*. 2021.

[36] J. Duchi, E. Hazan, and Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *Journal of Machine Learning Research* (2011).

[37] R. Eschenhagen, A. Immer, R. E. Turner, F. Schneider, and P. Hennig. "Kronecker-Factored Approximate Curvature for Modern Neural Network Architectures". *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[38] A. Fernandez. *Skerch: Sketched matrix decompositions for PyTorch*. 2024.

[39] J. Frank and C. Vuik. "On the construction of deflation-based preconditioners". *SIAM Journal on Scientific Computing* (2001).

[40] F. Galton. "Regression towards mediocrity in hereditary stature." *The Journal of the Anthropological Institute of Great Britain and Ireland* (1886).

[41] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson. "GPyTorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration". *Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

[42] M. Gargiani, A. Zanelli, M. Diehl, and F. Hutter. "On the Promise of the Stochastic Generalized Gauss-Newton Method for Training DNNs". *arXiv* (2020).

[43] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent. "Fast Approximate Natural Gradient Descent in a Kronecker Factored Eigenbasis". *Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

[44] B. Ghorbani, S. Krishnan, and Y. Xiao. "An Investigation into Neural Net Optimization via Hessian Eigenvalue Density". *arXiv* (2019).

[45] I. J. Goodfellow, Y. Bengio, and A. Courville. "Deep Learning". 2016.

[46] D. Granziol, X. Wan, and T. Garipov. "Deep Curvature Suite". 2021.

[47] R. Grosse and J. Martens. "A Kronecker-factored approximate Fisher matrix for convolution layers". *arXiv* (2016).

[48] R. Guhaniyogi and D. B. Dunson. "Bayesian Compressed Regression". *Journal of the American Statistical Association* (2015).

[49] V. Gupta, T. Koren, and Y. Singer. "Shampoo: Preconditioned Stochastic Tensor Optimization". *International Conference on Machine Learning (ICML)*. 2018.

[50] G. Gur-Ari, D. A. Roberts, and E. Dyer. "Gradient Descent Happens in a Tiny Subspace". *arXiv* (2018).

[51] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". *SIREV* (2011).

[52] B. Hassibi and D. Stork. "Second order derivatives for network pruning: Optimal Brain Surgeon". *Advances in Neural Information Processing Systems (NeurIPS)*. 1992.

[53] K. He, X. Zhang, S. Ren, and J. Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *ICCV*. 2015.

[54] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[55] D. Hendrycks and T. Dietterich. "Benchmarking Neural Network Robustness to Common Corruptions and Perturbations". *International Conference on Learning Representations (ICLR)*. 2019.

[56] P. Hennig. "Probabilistic Interpretation of Linear Solvers". *SIAM Journal on Optimization* (2015).

[57] P. Hennig, M. A. Osborne, and H. P. Kersting. "Probabilistic Numerics: Computation as Machine Learning". *Cambridge University Press*, 2022.

[58] P. Hennig, M. A. Osborne, and M. Girolami. "Probabilistic numerics and uncertainty in computations". *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* (2015).

[59] J. Hensman, N. Fusi, and N. D. Lawrence. "Gaussian processes for Big data". *Conference on Uncertainty in Artificial Intelligence (UAI)*. 2013.

[60] J. Hensman, A. Matthews, and Z. Ghahramani. "Scalable Variational Gaussian Process Classification". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2015.

[61] M. R. Hestenes and E. Stiefel. "Methods of conjugate gradients for solving linear systems". *Journal of Research of the National Bureau of Standards* (1952).

[62] K. Hornik. "Approximation capabilities of multilayer feedforward networks". *Neural Networks* (1991).

[63] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". *Neural Networks* (1989).

[64] Y. Idelbayev. "Proper ResNet Implementation for CIFAR10/CIFAR100 in PyTorch". https://github.com/akamaster/pytorch_resnet_cifar10. 2018.

[65] 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.

[66] 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.

[67] A. Jacot, F. Gabriel, and C. Hongler. "Neural Tangent Kernel: Convergence and Generalization in Neural Networks". *Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

[68] J. M. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. "Highly accurate protein structure prediction with AlphaFold". *Nature* (2021).

[69] P. Kasimbeg, F. Schneider, R. Eschenhagen, J. Bae, C. S. Sastry, M. Saroufim, F. Boyuan, L. Wright, E. Z. Yang, Z. Nado, S. Medapati, P. Hennig, M. Rabbat, and G. E. Dahl. "Accelerating neural network training: An analysis of the AlgoPerf competition". *International Conference on Learning Representations (ICLR)*. 2025.

[70] R. E. Kass, L. Tierney, and J. B. Kadane. "The validity of posterior expansions based on Laplace's method". *Bayesian and Likelihood Methods in Statistics and Econometrics* (1990).

[71] 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.

[72] E. Khan, S. Mohamed, and K. P. Murphy. "Fast Bayesian Inference for Non-Conjugate Gaussian Process Regression". *Advances in Neural Information Processing Systems (NeurIPS)*. 2012.

[73] M. E. Khan, A. Immer, E. Abedi, and M. Korzepa. "Approximate Inference Turns Deep Networks into Gaussian Processes". *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[74] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". *International Conference on Learning Representations (ICLR)*. 2015.

[75] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al. "Overcoming catastrophic forgetting in neural networks". *Proceedings of the National Academy of Sciences* (2017).

[76] P. W. Koh and P. Liang. "Understanding Black-box Predictions via Influence Functions". *Proceedings of Machine Learning Research (PMLR)*. 2017.

[77] 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.

[78] A. Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". 2009.

[79] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". *Advances in Neural Information Processing Systems (NeurIPS)*. 2012.

[80] A. Krogh and J. A. Hertz. "A Simple Weight Decay Can Improve Generalization". *Advances in Neural Information Processing Systems (NeurIPS)* (1991).

[81] A. Kumar, P. S. Liang, and T. Ma. "Verified Uncertainty Calibration". *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[82] F. Kunstner, P. Hennig, and L. Balles. "Limitations of the empirical Fisher approximation for natural gradient descent". *Advances in Neural Information Processing Systems*. 2019.

[83] P. S. Laplace. "Mémoire sur la probabilité des causes par les évènements." *Mémoires de Mathématique et de Physique, Tome Sixieme* (1774).

[84] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition". *Neural Computation* (1989).

[85] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". *Proceedings of the IEEE* (1998).

[86] Y. LeCun, J. Denker, and S. Solla. "Optimal Brain Damage". *Advances in Neural Information Processing Systems (NeurIPS)*. 1989.

[87] Y. LeCun, P. Simard, and B. Pearlmutter. "Automatic Learning Rate Maximization by On-Line Estimation of the Hessian's Eigenvectors". *Advances in Neural Information Processing Systems*. Edited by S. Hanson, J. Cowan, and C. Giles. Volume 5. *Morgan-Kaufmann*, 1993.

[88] W. Lin, F. Dangel, R. Eschenhagen, J. Bae, R. E. Turner, and A. Makhzani. "Can We Remove the Square-Root in Adaptive Gradient Methods? A Second-Order Perspective". *International Conference on Machine Learning (ICML)*. 2024.

[89] D. C. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming* (1989).

[90] D. MacKay. "Bayesian Model Comparison and Backprop Nets". *Advances in Neural Information Processing Systems (NeurIPS)*. 1991.

[91] D. J. C. MacKay. "The Evidence Framework Applied to Classification Networks". *Neural Computation* (1992).

[92] D. J. MacKay. "Choice of Basis for Laplace Approximation". *Machine Learning* (1998).

[93] J. R. Magnus and H. Neudecker. "Matrix Differential Calculus with Applications in Statistics and Econometrics". Second. *John Wiley*, 1999.

[94] J. Martens. "Deep learning via Hessian-free optimization". *International Conference on Machine Learning (ICML)*. 2010.

[95] J. Martens. "New insights and perspectives on the natural gradient method". *arXiv* (2020).

[96] J. Martens, J. Ba, and M. Johnson. "Kronecker-factored Curvature Approximations for Recurrent Neural Networks". *International Conference on Learning Representations (ICLR)*. 2018.

[97] J. Martens and R. Grosse. "Optimizing Neural Networks with Kronecker-factored Approximate Curvature". *International Conference on Machine Learning (ICML)*. 2015.

[98] M. Matena and C. Raffel. "Merging Models with Fisher-Weighted Averaging". *arXiv* (2022).

[99] I. Murray. "Gaussian processes and fast matrix-vector multiplies". *Numerical Mathematics in Machine Learning Workshop (ICML)*. 2009.

[100] J. A. Nelder and R. W. M. Wedderburn. "Generalized Linear Models". *Journal of the Royal Statistical Society. Series A (General)* (1972).

[101] J. Nocedal. "Updating Quasi-Newton Matrices with Limited Storage". *Mathematics of Computation* (1980).

[102] J. Nocedal and S. Wright. "Numerical optimization". 2. ed. *Springer*, 2006.

[103] R. Novak, J. Sohl-Dickstein, and S. S. Schoenholz. "Fast Finite Width Neural Tangent Kernel". *International Conference on Machine Learning (ICML)*. 2022.

[104] C. Oates and T. J. Sullivan. "A modern retrospective on probabilistic numerics". *Statistics and Computing* (2019).

[105] D. Oktay, N. McGreivy, J. Aduol, A. Beatson, and R. P. Adams. "Randomized Automatic Differentiation". *International Conference on Learning Representations (ICLR)*. 2021.

[106] K. Osawa, S. Ishikawa, R. Yokota, S. Li, and T. Hoefler. "ASDL: A Unified Interface for Gradient Preconditioning in PyTorch". *arXiv* (2023).

[107] 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". *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[108] V. Papyan. "Measurements of Three-Level Hierarchical Structure in the Outliers in the Spectrum of Deepnet Hessians". *International Conference on Machine Learning (ICML)*. 2019.

[109] V. Papyan. "The Full Spectrum of Deepnet Hessians at Scale: Dynamics with SGD Training and Sample Size". *arXiv* (2019).

[110] M. L. Parks, E. d. Sturler, G. Mackey, D. D. Johnson, and S. Maiti. "Recycling Krylov Subspaces for Sequences of Linear Systems". *SIAM Journal on Scientific Computing* (2006).

[111] 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.

[112] B. A. Pearlmutter. "Fast Exact Multiplication by the Hessian". *Neural Computation* (1994).

[113] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods". *USSR Computational Mathematics and Mathematical Physics* (1964).

[114] C. E. Rasmussen and C. K. I. Williams. "Gaussian Processes for Machine Learning". *MIT Press*, 2006.

[115] H. Ritter, A. Botev, and D. Barber. "A Scalable Laplace Approximation for Neural Networks". *International Conference on Learning Representations (ICLR)*. 2018.

[116] H. Ritter, A. Botev, and D. Barber. "Online Structured Laplace Approximations for Overcoming Catastrophic Forgetting". *Advances in Neural Information Processing Systems*. 2018.

[117] H. Roy, M. Miani, C. H. Ek, P. Hennig, M. Pförtner, L. Tatzel, and S. Hauberg. "Reparameterization invariance in approximate Bayesian inference". *Advances in Neural Information Processing Systems (NeurIPS)*. 2024.

[118] H. Rue, S. Martino, and N. Chopin. "Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations". *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* (2009).

[119] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". *Nature* (1986).

[120] L. Sagun, L. Bottou, and Y. LeCun. "Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond". *arXiv* (2017).

[121] L. Sagun, U. Evci, V. U. Guney, Y. Dauphin, and L. Bottou. "Empirical Analysis of the Hessian of Over-Parametrized Neural Networks". *arXiv* (2018).

[122] R. M. Schmidt, F. Schneider, and P. Hennig. "Descending through a Crowded Valley – Benchmarking Deep Learning Optimizers". 2021.

[123] F. Schneider, L. Balles, and P. Hennig. "DeepOBS: A Deep Learning Optimizer Benchmark Suite". *International Conference on Learning Representations (ICLR)*. 2019.

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

[125] N. N. Schraudolph. "Fast curvature matrix-vector products for second-order gradient descent". *Neural computation* (2002).

[126] H.-J. M. Shi, T.-H. Lee, S. Iwasaki, J. Gallego-Posada, Z. Li, K. Rangadurai, D. Mudigere, and M. Rabbat. "A Distributed Data-Parallel PyTorch Implementation of the Distributed Shampoo Optimizer for Training Neural Networks At-Scale". *arXiv* (2023).

[127] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search". *Nature* (2016).

[128] S. P. Singh and D. Alistarh. "WoodFisher: Efficient Second-Order Approximation for Neural Network Compression". *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

[129] J. Sliwa, F. Schneider, N. Bosch, A. Kristiadi, and P. Hennig. "Efficient Weight-Space Laplace-Gaussian Filtering and Smoothing for Sequential Deep Learning". *arXiv* (2024).

[130] A. Spantini, A. Solonen, T. Cui, J. Martin, L. Tenorio, and Y. Marzouk. "Optimal low-rank approximations of Bayesian linear inverse problems". *SIAM Journal on Scientific Computing* (2015).

[131] D. J. Spiegelhalter and S. L. Lauritzen. "Sequential updating of conditional probabilities on directed graphical structures". *Networks* (1990).

[132] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. "Striving for Simplicity: The All Convolutional Net". *arXiv* (2015).

[133] L. Tatzel, P. Hennig, and F. Schneider. "Late-Phase Second-Order Training". *Has it Trained Yet? NeurIPS 2022 Workshop*. 2022.

[134] L. Tatzel, B. Mucsányi, O. Hackel, and P. Hennig. "Debiasing Mini-Batch Quadratics for Applications in Deep Learning". *International Conference on Learning Representations (ICLR)*. 2025.

[135] L. Tatzel, J. Wenger, F. Schneider, and P. Hennig. "Accelerating Non-Conjugate Gaussian Processes By Trading Off Computation For Uncertainty". *Transactions on Machine Learning Research (TMLR)* (2025).

[136] M. Telgarsky. "Benefits of depth in neural networks". *Proceedings of the 29th Annual Conference on Learning Theory (COLT)*. 2016.

[137] T. Tieleman, G. Hinton, et al. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". 2012.

[138] M. Titsias. "Variational learning of inducing variables in sparse Gaussian processes". *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2009.

[139] B. L. Trippe, J. H. Huggins, R. Agrawal, and T. Broderick. "LR-GLM: High-Dimensional Bayesian Inference Using Low-Rank Data Approximations". *International Conference on Machine Learning (ICML)*. 2019.

[140] K. A. Wang, G. Pleiss, J. R. Gardner, S. Tyree, K. Q. Weinberger, and A. G. Wilson. "Exact Gaussian processes on a million data points". *Advances in Neural Information Processing Systems (NeurIPS)* (2019).

[141] J. Wenger and P. Hennig. "Probabilistic Linear Solvers for Machine Learning". *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

[142] J. Wenger, G. Pleiss, P. Hennig, J. P. Cunningham, and J. R. Gardner. "Preconditioning for Scalable Gaussian Process Hyperparameter Optimization". *International Conference on Machine Learning (ICML)*. 2022.

[143] J. Wenger, G. Pleiss, M. Pförtner, P. Hennig, and J. P. Cunningham. "Posterior and Computational Uncertainty in Gaussian processes". *Advances in Neural Information Processing Systems (NeurIPS)*. 2022.

[144] J. Wenger, K. Wu, P. Hennig, J. R. Gardner, G. Pleiss, and J. P. Cunningham. "Computation-Aware Gaussian Processes: Model Selection And Linear-Time Inference". *Advances in Neural Information Processing Systems (NeurIPS)*. 2024.

[145] Z. Yao, A. Gholami, K. Keutzer, and M. Mahoney. “PyHessian: Neural Networks Through the Lens of the Hessian”. *arXiv* (2019).

[146] S. Zagoruyko and N. Komodakis. “Wide Residual Networks”. *arXiv* (2017).

[147] W. Zeng and R. Urtasun. “MLPrune: Multi-Layer Pruning for Automated Neural Network Compression”. 2019.

[148] H. Zhang, C. Xiong, J. Bradbury, and R. Socher. “Block-diagonal Hessian-free Optimization for Training Neural Networks”. *arXiv* (2017).

[149] L. Zhang, M. Mahdavi, R. Jin, T. Yang, and S. Zhu. “Random Projections for Classification: A Recovery Approach”. *IEEE Transactions on Information Theory* (2014).

[150] Y.-T. Zhou and R. Chellappa. “Computation of optical flow using a neural network”. *IEEE International Conference on Neural Networks*. 1988.