INAUGURAL-DISSERTATION
zur
Erlangung der Doktorwürde
der
Gesamtfakultät für Mathematik, Ingenieur- und Naturwissenschaften
der
RUPRECHT-KARLS-UNIVERSITÄT
HEIDELBERG

vorgelegt von

Kreis, Leonie, M. Sc.

aus Mannheim

Tag der mündlichen Prüfung:

Nonlinear Optimization Techniques Applied to Neural Network Training

Betreuer: Prof. Dr. Roland Herzog

Abstract

This thesis explores some aspects of how techniques from classical, deterministic, nonlinear optimization can be adapted to address the challenges posed by optimization problems that arise in neural network training. Such problems can be formulated as large-scale, stochastic and non-convex parameter estimation problems. Three main contributions are presented. First, we explore a multilevel optimization technique inspired by Multigrid Optimization (MG/Opt) for stochastic optimization problems. Starting with a method tailored to strongly convex quadratic objective functions, and continuing with a version suited to non-convex objective functions, stochastic MG/Opt variants are proposed and analyzed. For strongly convex quadratic problems, transfer operators suited to the Stochastic Gradient Descent (SGD) method are proposed and analyzed theoretically. For non-convex problems, novel convergence results are provided for stochastic bi-level formulations using SGD with both fixed and diminishing step sizes. This analysis highlights both the potential benefits and the limitations of the approach. The practical applications of multilevel structures in neural networks are also discussed. However, not all hierarchical constructions yield practical improvements. Furthermore, multigrid optimization in highly non-convex, stochastic settings remains only partially understood. Second, Sensitivity-based Layer Insertion (SensLI), a sensitivity-based procedure for adaptive layer insertion, is introduced. SensLI formulates network expansion as a constrained sensitivity analysis problem and produces a simple, general selection criterion for promising insertion positions. Numerical experiments on several architectures demonstrate that SensLI can efficiently increase model capacity during training. A theoretical comparison of the computational cost of the layer selection process with that of other adaptive methods is provided and shows its efficiency. Additionally, the method is extended to layer widening. Third, we propose a layer-wise preconditioning framework based on Frobenius-type inner products on the spaces of linear maps. This framework can handle predefined inner products based on prior knowledge of the layer spaces as well as data-driven inner products that adapt during training. We present a covariance-driven construction for the inner products on layer spaces and their resulting preconditioner, which equip layer spaces with non-Euclidean structures that describe the distribution of the layer data. This preconditioner is strongly similar to the Kronecker-Factored Approximate Curvature (K-FAC) method. Numerical studies critically assess the empirical benefits and limitations of this covariance-based preconditioner in practice.

Overall, the results demonstrate the value of classical optimization ideas for training neural networks, provided they are carefully tailored to the stochastic, high-dimensional, and non-convex nature of modern models. The thesis concludes with recommendations for future research, including a tighter integration of theory and experiments and a deeper understanding of the circumstances under which specific hierarchies, preconditioners, or insertion rules improve training.

Zusammenfassung

Diese Arbeit untersucht, wie ausgewählte Techniken aus der klassischen, deterministischen, nichtlinearen Optimierung angepasst werden können, um die Herausforderungen zu bewältigen, die sich aus Optimierungsproblemen beim Training neuronaler Netze ergeben. Solche Probleme lassen sich als hochdimensionale, stochastische und nichtkonvexe Parameter-Schätzungsprobleme formulieren. Es werden drei wesentliche Beiträge vorgestellt. Zunächst untersuchen wir eine von MG/Opt inspirierte mehrstufige Optimierungstechnik für stochastische Optimierungsprobleme. Ausgehend von einer Methode, die auf stark konvexe quadratische Zielfunktionen zugeschnitten ist, und fortfahrend mit einer Version, die für nichtkonvexe Zielfunktionen geeignet ist, werden stochastische MG/Opt-Varianten vorgeschlagen und analysiert. Für stark konvexe quadratische Probleme werden Transferoperatoren vorgeschlagen, die für die SGD-Methode geeignet sind, und theoretisch analysiert. Für nichtkonvexe Probleme werden neue Konvergenzergebnisse für stochastische zweistufige Formulierungen unter Verwendung von SGD mit festen und abnehmenden Schrittweiten bereitgestellt. Diese Analyse hebt sowohl die potenziellen Vorteile als auch die Grenzen des Ansatzes hervor. Die praktischen Anwendungen von mehrstufigen Strukturen in neuronalen Netzen werden ebenfalls diskutiert. Allerdings führen nicht alle hierarchischen Konstruktionen zu praktischen Verbesserungen. Darüber hinaus ist die Mehrgitter-Optimierung in nichtkonvexen, stochastischen Umgebungen nach wie vor nur teilweise verstanden. Zweitens wird SensLI vorgestellt, ein sensitivitätsbasiertes Verfahren zur adaptiven Einfügung von Schichten. SensLI formuliert die Netzwerkerweiterung als ein Problem der Sensitivitätsanalyse und liefert ein allgemeines Auswahlkriterium für vielversprechende Einfügungspositionen. Numerische Experimente mit mehreren Architekturen zeigen, dass SensLI die Modellkapazität während des Trainings effizient erhöhen kann. Ein theoretischer Vergleich der Rechenkosten des Schicht-Auswahlprozesses mit denen anderer adaptiver Methoden wird bereitgestellt und zeigt dessen Effizienz. Darüber hinaus kann die Methode auf die Schichtverbreiterung ausgeweitet werden. Drittens schlagen wir einen schichtweisen Vorkonditionierungsansatz vor, der auf Frobenius-artigen Innenprodukten auf den Räumen linearer Abbildungen basiert. Dieser Rahmen kann sowohl vordefinierte Innenprodukte auf der Grundlage von Vorwissen über die Schichträume als auch datengesteuerte Innenprodukte verarbeiten, die sich während des Trainings anpassen. Wir stellen eine kovarianzbasierte Konstruktion für die inneren Produkte auf Schichträumen und den daraus resultierenden Vorkonditionierer vor, der Schichträume mit nicht-euklidischen Strukturen ausstattet, die die Verteilung der Schichtdaten beschreiben. Dieser Vorkonditionierer ähnelt stark der K-FAC-Methode. Numerische Studien bewerten die Vorteile und Grenzen dieses kovarianzbasierten Vorkonditionierers in der Praxis.

Insgesamt zeigen die Ergebnisse den Wert klassischer Optimierungskonzepte für das Training neuronaler Netze, sofern sie sorgfältig auf die stochastische, hochdimensionale und nichtkonvexe Natur moderner Modelle zugeschnitten sind. Die Arbeit schließt mit Empfehlungen für die zukünftige Forschung, darunter eine engere Verbindung von Theorie und Experimenten, realistische Kostenmodelle für stochastische Methoden und ein tieferes Verständnis der Umstände, unter denen bestimmte Hierarchien, Vorkonditionierer oder Einfügungsregeln das Training verbessern.

Danksagung

Ich danke zunächst meinem Doktorvater Roland für die zuverlässige Betreuung, die vielen Anregungen und die Möglichkeit, dieses Projekt durchzuführen. Mein besonderer Dank gilt Ihno und meiner Schwester Franka für ihre beständige mentale Unterstützung, Geduld und offenen Ohren während der gesamten Zeit. Evelyn danke ich für den fachlichen Austausch und dafür, dass sie stets an mich geglaubt hat und mich ermutigt und unterstützt hat. Frederik und Anton danke ich für den fachlichen Austausch und die gute Zusammenarbeit.

Mein Dank geht außerdem an meine Kolleginnen und Kollegen aus der Gruppe - Viktor, Manuel, Hannah, Nico, Johannes, Georg, Jeannette und Karina - für die zahlreichen hilfreichen Diskussionen und die angenehme Arbeitsatmosphäre. Ein besonderer Dank geht auch an Maxi für die vielen langen Spaziergänge, die motivierenden Gespräche und seine beständige Unterstützung. Den "Chemikern", insbesondere Melina M., Tobi und Sebastian, danke ich für die vielen motivierenden Momente und Gespräche. Den "Mathematikern", vor allem Christian und Julian, danke ich für anregende Gespräche an der Kaffeemaschine und die gemeinsamen Mensazeiten.

Schließlich danke ich meinen Eltern für ihre fortwährende Unterstützung und Rückendeckung während meines Studiums und dieser Promotion.

Contents

List of Acronyms

PDE	Partial Differential Equation
ODE	Ordinary Differential Equation
FNN	Feedforward Neural Network
ResNet	Residual Neural Network
CNN	Convolutional Neural Network
ReLU	Rectified Linear Unit
MaxPool	Max-Pooling
GD	Gradient Descent
SGD	Stochastic Gradient Descent
MSE	Mean Squared Error
CE	Cross-Entropy
lr	learning rate
MG/Opt	Multigrid Optimization
cgc	coarse grid correction
SensLI	Sensitivity-based Layer Insertion
NAS	Neural Architecture Search
KKT	Karush-Kuhn-Tucker
LICQ	Linear Independence Constraint Qualification
IFT	Implicit Function Theorem
FLOP	Floating Point Operation
NGD	Natural Gradient Descent
FIM	Fisher Information Matrix
K-FAC	Kronecker-Factored Approximate Curvature
SMW	Sherman-Morrison-Woodbury
QN	Quasi-Newton
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno
GGN	Generalized Gauss-Newton
HFO	Hessian-Free Optimization

1. Introduction

The success of deep learning in various application domains over the last years has led to a widespread adoption of neural networks for tasks such as image classification, natural language processing, and reinforcement learning and to a frequent application of neural networks in scientific computing and engineering as well.

At its core, a supervised learning task which aims to find a mapping $g_{\text{exact}}\colon X \longrightarrow Y$ from a vector space X to a vector space Y with neural networks as the model class is formulated as a parameter estimation problem. A typical optimization problem of this form is given by

$$\min_{\theta \in \Theta} \frac{1}{D} \sum_{j=1}^{D} \mathcal{L}(g(\theta, x_{0,j}), y_{\text{label}j}),$$

where D is the number of training data points with features $x_{0,j} \in X$ and associated labels $y_{\text{label}j} \in Y$, and $g(\theta, \cdot)\colon X \longrightarrow Y$ is a neural network with parameters $\theta \in \Theta$ and $\mathcal{L}\colon Y \times Y \longrightarrow \mathbb{R}_{\geq 0}$ is a loss function measuring the discrepancy between predicted outputs $g(\theta, x_{0,j})$ and true labels $y_{\text{label}j}$.

Training neural networks on this supervised learning task corresponds to solving the above optimization problem, which is done with iterative optimization methods. The optimization problem is typically high-dimensional, since the optimization variables are comprised of the parameters of the neural network and has a highly non-convex objective function. Furthermore, due to the use of data samples the optimization problem has an underlying stochastic nature, which needs to be taken into account when choosing optimization methods for neural network training.

Commonly used optimization methods for neural network training are first-order methods, such as SGD and its variants and extensions. These methods are simple to implement and often use Euclidean geometry. Further, they are treating the neural network architecture as fixed during training. While these methods are effective in many settings, they do not exploit richer geometric or multilevel structure that could, from an optimization perspective, potentially improve convergence and robustness.

Hence, the aim of this thesis is to investigate how selected techniques from classical nonlinear deterministic optimization such as multilevel optimization, preconditioning, sensitivity analysis and adjustable models can be adapted to the challenges of neural network training, with the goal of improving convergence, robustness, and adaptability in these stochastic, non-convex learning problems. The techniques need to be tailored to the neural network structure and adapted to the stochastic nature of the problem; at the same time the methods should remain efficient and scalable to large models and datasets.

1.1. Main Contributions

In this thesis, we present three main contributions that adapt classical nonlinear optimization techniques to the challenges of neural network training:

We develop multilevel optimization methods based on the MG/Opt method for stochastic optimization problems. Starting with the construction of hierarchical optimization problems for strongly convex quadratic objective functions and restriction and prolongation operators suited for SGD, we examine a stochastic variant of MG/Opt tailored to quadratic problems and examine it theoretically. As a next step, we propose a stochastic version of MG/Opt adapted to the non-convex and stochastic setting arising in neural network training, providing convergence proofs for stochastic bi-level formulations with fixed and diminishing step sizes and exploring potential practical implementations and applications in neural network training. One other way to exploit the multilevel structure of neural networks which is naturally obtained by their layer-wise composition is to build up and expand a small baseline network during training, which we do in the SensLI framework. We formulate SensLI via constrained sensitivity analysis, which allows us to identify promising positions in the network architecture for layer insertion based on sensitivity information. We demonstrate the effectiveness of SensLI through a series of numerical experiments on various neural network architectures, showing that it can lead to improved performance and efficiency compared to training fixed architectures. Further, we extend the approach to layer widening. Finally, we introduce a layer-wise preconditioning framework based on Frobenius-type inner products on spaces of linear maps that unifies predefined and data-driven inner products. We investigate covariance-driven approaches to build the preconditioners and relate our framework to Kronecker-Factored Approximate Curvature (K-FAC). Numerical experiments demonstrate the effectiveness and limitations of the proposed preconditioning strategies in improving convergence during neural network training.

1.2. Structure

The structure of this thesis is as follows. After the introduction of necessary fundamental concepts, the main methodological contributions are presented in the subsequent chapters.

The fundamental concepts and their notation used throughout the thesis are introduced in Chapter 2.

Chapter 3 is devoted to MG/Opt methods for stochastic optimization problems. Here, classical multigrid ideas are adapted to the stochastic and non-convex setting of deep learning. The chapter discusses the construction of hierarchical optimization problems, analyzes convergence properties, and explores practical aspects of applying MG/Opt in neural network training.

Following this, Chapter 4 introduces the SensLI framework, which provides a depth-adaptive approach to neural network training. This chapter details the theoretical foundations of SensLI, describes its algorithmic implementation, and demonstrates its effectiveness through a series of numerical experiments on various neural network architectures.

Chapter 5 focuses on the development of preconditioners based on Frobenius-type inner products for optimization algorithms used in training Feedforward Neural Networks (FNNs). The chapter presents the mathematical formulation of these preconditioners, and investigates covariance-driven approaches to build the preconditioners. Further, it evaluates their performance through numerical experiments, highlighting their potential and limitations to improve convergence during neural network training.

Finally, Chapter 6 concludes the thesis with a summary of the main findings, a discussion of their implications for neural network training, and an outlook on potential directions for future research. Additional technical details, proofs, notation and supplementary results are provided in the appendices, while references to relevant literature are collected at the end of the thesis.

Each of the three main chapters contains its own section discussing related work and a discussion.

2. Fundamental Concepts

In this chapter we introduce selected fundamental concepts from statistics, deep learning and optimization that are used throughout this thesis. The introduction to necessary concepts from statistics and optimization can be found among others e.g. in textbooks such as [71] for statistics and [98, 11] for optimization. We remark that the presentation of deep learning concepts in this chapter is not exhaustive and serves as a basic introduction to understand the subsequent chapters. We refer to [42] for a comprehensive introduction to deep learning.

2.1. Statistical Concepts

In this section we introduce selected fundamental concepts from statistics that are used throughout this thesis to provide stochastic convergence results in Chapter 3 and construct data-driven preconditioners in Chapter 5.

We consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is the sample space, $\mathcal{F}$ is the σ-algebra of events and $\mathbb{P}$ is the probability measure. We consider random variables on this probability space, e.g. to model stochastic gradients or data samples. Each random variable is a measurable mapping

$$X\colon (\Omega, \mathcal{F}, \mathbb{P}) \to (V, \mathcal{B}(V)),$$

where V is a finite-dimensional vector space equipped with the Borel σ-algebra $\mathcal{B}(V)$. In the following, we shorten the notation and write $X\colon \Omega \to V$. We start by introducing the notion of expectation and covariance.

Definition 2.1 (Expectation of X). *The expectation of a random variable $X\colon \Omega \to V$ taking values in a vector space $V = \mathbb{R}^n$ is defined as*

$$\mathbb{E}[X] = \int_\Omega X(\omega) d\mathbb{P}(\omega), \tag{2.1}$$

if the integral exists.

Definition 2.2 (Covariance of X). *The covariance of a random variable $X\colon \Omega \to V$ taking values in a finite-dimensional vector space $V = \mathbb{R}^n$ is defined as*

$$Cov(X) = \mathbb{E}\left[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^T\right], \tag{2.2}$$

if the expectation exists and is finite.

It is also useful to introduce the notion of conditional expectation, which is the expectation of a random variable given some known information $\mathcal{G}$. In this thesis, we do use the notation of the conditional expectation with respect to other random variables, such as $\mathbb{E}[X|Y]$. The associated σ-algebra is then the one generated by the random variable Y. For a more precise characterization of conditional expectation and independence we refer the reader to [71].

Definition 2.3 (Conditional Expectation)**.** *The conditional expectation of a random variable $X\colon \Omega \longrightarrow V$ given a σ-algebra $\mathcal{G} \subseteq \mathcal{F}$ is defined as the random variable $\mathbb{E}[X|\mathcal{G}]$ that satisfies*

$$\int_A \mathbb{E}[X|\mathcal{G}](\omega)d\mathbb{P}(\omega) = \int_A X(\omega)d\mathbb{P}(\omega) \quad \textit{for all } A \in \mathcal{G}.$$

Next, we define independence of a random variable and a σ-algebra.

Definition 2.4 (Independence of a Random Variable and a σ-algebra)**.** *A random variable $X\colon \Omega \longrightarrow V$ is said to be independent of a σ-algebra $\mathcal{G} \subseteq \mathcal{F}$ if for all $A \in \mathcal{G}$ and any $B \in \sigma(X)$, it holds that*

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B).$$

Here, $\sigma(X)$ denotes the σ-algebra generated by the random variable X.

In order to obtain convergence results for sequences of random variables, we need the following properties of the conditional expectation and different forms of convergence.

Lemma 2.5 (Properties of the Conditional Expectation)**.** *Let $X\colon \Omega \longrightarrow V$ and $Y\colon \Omega \longrightarrow V$ be random variables and $\mathcal{G} \subseteq \mathcal{F}$ be a σ-algebra. Then the following properties hold:*

(*i*) *Linearity: $\mathbb{E}[aX + bY|\mathcal{G}] = a\mathbb{E}[X|\mathcal{G}] + b\mathbb{E}[Y|\mathcal{G}]$ for all $a, b \in \mathbb{R}$.*

(*ii*) *Taking out what is known: If X is $\mathcal{G}$-measurable, then $\mathbb{E}[XY|\mathcal{G}] = X\mathbb{E}[Y|\mathcal{G}]$.*

(*iii*) *Tower property: If $\mathcal{H} \subseteq \mathcal{G} \subseteq \mathcal{F}$ are σ-algebras, then $\mathbb{E}[\mathbb{E}[X|\mathcal{G}]|\mathcal{H}] = \mathbb{E}[X|\mathcal{H}]$.*

(*iv*) *If X is independent of $\mathcal{G}$, then $\mathbb{E}[X|\mathcal{G}] = \mathbb{E}[X]$.*

Lemma 2.6 (Total Expectation)**.** *Let $X\colon \Omega \longrightarrow V$ be a random variable and $\mathcal{G} \subseteq \mathcal{F}$ be a σ-algebra. Then the following property holds:*

$$\mathbb{E}[\mathbb{E}[X|\mathcal{G}]] = \mathbb{E}[X].$$

The next definition introduces the notion of independent and identically distributed (i.i.d.) random variables, which are used to construct stochastic gradient estimates in Chapter 3 and for SGD methods, and data-driven preconditioners in Chapter 5.

Definition 2.7 (i.i.d. random variables)**.** *A sequence of random variables $\{X_k\}_{k=1}^{\infty}$ where $X_k\colon \Omega \longrightarrow V$, is independent and identically distributed (i.i.d.) if each X_k has the same probability distribution as the others and all are mutually independent.*

In the following, the families of random variables $\{X_k\}_{k=1}^N$ for $N \in \mathbb{N} \cup \{\infty\}$ have the form $X_k : \Omega \to V$.

Given i.i.d. random variables $\{X_k\}_{k=1}^D$, the empirical mean is defined as

$$\bar{X}_D = \frac{1}{D} \sum_{k=1}^{D} X_k. \tag{2.3}$$

The covariance matrix estimator for the i.i.d. random variables $\{X_k\}_{k=1}^D$ with values in $V = \mathbb{R}^n$ is defined as

$$\hat{\mathrm{Cov}}_D(X) = \frac{1}{D} \sum_{k=1}^{D} (X_k - \bar{X}_D)(X_k - \bar{X}_D)^T, \tag{2.4}$$

where $\bar{X}_D$ is the empirical mean defined in (2.3).

There are several forms of convergence for stochastic objects such as sequences of random variables. We show the two most common forms of convergence used in this thesis in Chapter 3.

Definition 2.8 (Convergence in Expectation). *A sequence of random variables $\{X_k\}_{k=1}^\infty$ with values in $V = \mathbb{R}$ converges in expectation to a random variable X if*

$$\lim_{k \to \infty} \mathbb{E}[|X_k - X|] = 0$$

and the absolute moments $\mathbb{E}[|X_k|]$ and $\mathbb{E}[|X|]$ exist for all k.

The definition can be extended to random variables with values in any finite-dimensional vector space V by replacing the absolute value with a norm on V.

Definition 2.9 (Almost Sure Convergence). *A sequence of random variables $\{X_k\}_{k=1}^\infty$ converges almost surely (a.s.) to a random variable X if*

$$\mathbb{P}(\{\omega \in \Omega : \lim_{k \to \infty} X_k(\omega) = X(\omega)\}) = 1.$$

Remark 2.10 (Relations between Forms of Convergence). *Almost sure convergence is the strongest form of convergence for random variables. It implies convergence in expectation, the converse is not true.*

The following definition and result are used to prove almost sure convergence of certain sequences of random variables in Chapter 3.

Definition 2.11 (Supermartingale). *A sequence of real-valued random variables $\{X_k\}_{k=1}^\infty$ is called a supermartingale with respect to a sequence of σ-algebras $\{\mathcal{F}_k\}_{k=1}^\infty$, if for all k,*

(*i*) *X_k is $\mathcal{F}_k$-measurable,* (*ii*) $\mathbb{E}[|X_k|] < \infty$,

(*iii*) *and the supermartingale property holds:*

$$\mathbb{E}[X_{k+1}|\mathcal{F}_k] \leq X_k.$$

Theorem 2.12 (Supermartingale Convergence Theorem [28, Theorem 5.2.9]). *If $\{X_k\}_{k=1}^{\infty}$ is a non-negative supermartingale, then*

$$\lim_{k\to\infty} X_k = X \textit{ almost surely} \quad \textit{and} \quad \mathbb{E}[X] \leq \mathbb{E}[X_0].$$

2.2. Deep Learning

Deep learning is a subfield of machine learning. It focuses on learning representations of data through models composed of multiple processing layers which are called neural networks. The functions realized by neural networks form a specific hypothesis or model class. A model class is a set of functions $g(\theta, \cdot)$ parametrized by some set of parameters $\theta \in \Theta$; in the simplest form the neural network functions g are parametrized by the finite-dimensional weights W_i and biases b_i appearing in the i-th layer of the neural network.

At its core, deep learning can be viewed as an optimization problem, or more precisely, a parameter estimation problem, where the goal is to find optimal parameters and associated network function that minimize a chosen metric $\mathcal{L}$ on given data $(x_{0,j}, y_{\text{label}j})_{j=1\dots,D}$. The exact mapping we want to learn is generally unknown, hence we can only approximate it with e.g. a neural network function. The power of deep learning arises from its ability to learn complex patterns and representations from large datasets while being computationally relatively efficient, enabling state-of-the-art performance in tasks such as image recognition, natural language processing, and many others.

Neural networks as hypothesis classes can be utilized, among others, for unsupervised learning tasks, where the model learns to find patterns in data which has no explicit labels, or for reinforcement learning tasks, where the model learns to make decisions based on feedback from an environment. However, the most common tasks in deep learning are supervised learning tasks, where the model is trained on labeled data to learn a mapping from inputs to outputs. In the following chapters we focus on supervised learning tasks.

This section introduces the overall setting, in which the contributions of this thesis are developed. We start by introducing different types of neural network architectures in Section 2.2.1, followed by the derivation of a general optimization problem given in supervised learning tasks in Section 2.2.2. Lastly, we examine the structure of the derivatives and Euclidean gradients of the neural network architectures in Section 2.2.2.

2.2.1. Neural Network Architectures

All neural network architectures are composed of *layers*, each consisting of a set of *neurons* which are also sometimes called *nodes* and a propagation function, that process input data and pass it to the next layer. The number of layers in a neural network is called the *depth* of the neural network. The number of neurons in a layer of the neural network is called the *width* of the layer. There exist different types of neural network architectures, such as fully-connected Feedforward Neural Networks (FNNs), Residual Neural Networks (ResNets), Convolutional Neural Networks (CNNs), and many more. In the following, we describe common neural network architectures, such as FNNs, ResNets and CNNs.

Fully-Connected Feedforward Neural Networks

A fully-connected Feedforward Neural Network (FNN) is the most classical neural network architecture and widely used in various applications. An illustrative example of a fully-connected FNN is shown in Figure 2.1.

In a fully-connected FNN, each layer consists of an affine linear transformation of the input data, followed by an activation function. The forward pass of data through a fully-connected FNN with L layers is given by the equations

$$y_i := W_i(x_{i-1}) + b_i \quad \text{for } i = 1, \dots, L, \tag{2.5}$$

$$x_i := \sigma_i(y_i) \quad \text{for } i = 1, \dots, L-1, \tag{2.6}$$

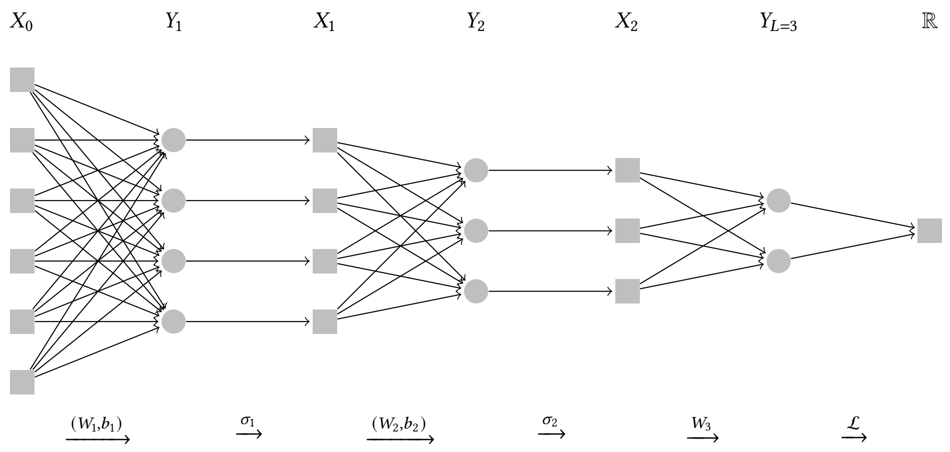


Figure 2.1.: An illustration of the notation for a fully-connected feedforward neural network with $L = 3$ layers.

for some input data $x_0 \in X_0$. Finally, the output of the neural network is given by y_L, where the last layer of the FNN consists only of the affine linear transformation without an activation function. In some scenarios, the bias b_i is omitted in the last layer as well, depending on the application. One layer is usually considered to be the combination of the affine linear transformation and the activation function, hence consisting of two parts: the pre-activation features $y_i \in Y_i$ and the post-activation features $x_i \in X_i$. The dimensions of the layer spaces are given by n_{Y_i} for the pre-activation features $y_i \in Y_i$ and n_{X_i} for the post-activation features $x_i \in X_i$, where n_{Y_i} is the number of neurons in the layer. Usually, the pre-activation space Y_i has the same dimension as the post-activation space X_i because the activation function is applied element-wise. The input space is given by X_0 with dimension n_{X_0}, the output space is given by Y_L with dimension n_{Y_L}. The layers which are not the input or output layer are called *hidden layers*. The parameters

$$W_i \in \mathbb{R}^{n_{X_i} \times n_{X_{i-1}}} \text{ for } i = 1, \ldots, L$$

are called the *weight matrices*, which are multiplied to the layer input. More generally they can be interpreted as objects $\in L(X_{i-1}, Y_i)$, where $L(U, V)$ denotes the space of linear maps from the vector space U to the vector space V, i.e. a linear operator. The *bias vector*

$$b_i \in \mathbb{R}^{n_{X_i}} \text{ for } i = 1, \ldots, L-1$$

is added to the product of weight matrix and layer input. The *activation function* is a nonlinear function

$$\sigma_i \colon Y_i \longrightarrow X_i \text{ for } i = 1, \ldots, L-1.$$

Often, the activation function consists of a one-dimensional function $\sigma \colon \mathbb{R} \longrightarrow \mathbb{R}$ applied element-wise to the pre-activation features. The activation function introduces nonlinearity to the neural network, which is crucial for the neural network to be able to learn complex patterns in the data. If the activation function is linear as well, the neural network would be equivalent to a single linear transformation. For shorter notation, we call the fully-connected FNN simply the FNN in the following.

Common activation functions which are applied element-wise are the *Rectified Linear Unit (ReLU)* activation function

$$\sigma(x) = \operatorname{ReLU}(x) := \max(0, x),$$

the sigmoid activation function (more correctly called *logistic* activation function)

$$\sigma(x) := \frac{1}{1 + \exp(-x)},$$

and the *hyperbolic tangent* activation function

$$\sigma(x) = \tanh(x) := \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}.$$

In order to generate different neural network functions which form the hypothesis classes for learning tasks, the parameters of the neural network have to be modified. These parameters are called *trainable parameters* and are used to adapt the neural network to the data at hand. The trainable parameters of a FNN are the weight matrices W_i and the bias vectors b_i for $i = 1, \ldots, L-1$. The set of all trainable parameters is denoted by

$$\theta = (W_i, b_i)_{i=1}^{L} \in \Theta,$$

where Θ is the space containing all possible values of all trainable parameters.

The *network function* for a parameter $\theta \in \Theta$ is the function that maps the input data to the output data, given by

$$g(\theta, \cdot) : X_0 \to Y_L, \quad x_0 \mapsto g(\theta, x_0) := y_L,$$

cf. (2.5). The modification of either width or depth of the neural network changes the hypothesis class of functions that can be represented by the neural network and hence also the number and structure of the trainable parameters.

There exist theoretical works that show the approximation capabilities of FNNs, depending on width, depth and activation function. Results of this form are called *universal approximation theorems* and show that FNNs can approximate any continuous function on a compact set to arbitrary precision, given sufficient width and depth.

The case for FNNs with only one hidden layer of arbitrary width was studied first. We mention here only a few of the many results in this setting. As the earliest version, [20] showed that a FNN with a single hidden layer and a sigmoid activation function can approximate any continuous function on a compact set to arbitrary precision, given sufficient width. [63] extended this result by proving that a FNN with a single hidden layer and a continuous activation function can approximate any continuous function on a compact set to arbitrary precision, given sufficient width. These results were further extended in [104] by showing that a FNN with a single hidden layer is a universal approximator if and only if the activation function is not a polynomial. The case for FNNs with fixed width and arbitrary depth was studied later. [86] showed that a ReLU-FNN with a fixed width of $n_{X_0} + 1$ (n_{X_0} is the dimension of the input space) and arbitrary depth can approximate any continuous function on a compact set to arbitrary precision, given sufficient depth. These results display that FNNs are universal function approximators, meaning that they can approximate any continuous function on a compact set to arbitrary precision, given sufficient width and depth.

Residual Neural Networks

The Residual Neural Network (ResNet) architecture is a type of neural network that uses skip connections, or residual connections in the propagation function between layers and aims to address the vanishing gradient problem that can occur in deep neural networks in order to stabilize training of very deep networks. The architecture was first introduced in [55] and has since become a popular architecture for deep learning tasks, especially in computer vision. The key idea behind ResNets is to allow the network to learn residual functions, which are the differences between the input and output of a layer, rather than the output itself.

A specific residual network architecture was proposed in [50] and has a strong mathematical interpretation. The residual network is the parametrized function

$$g(\theta, \cdot)\colon X_0 \to X_{L+1}, \quad x_0 \mapsto x_{L+1}$$

with the decomposed form

$$x_1 = W_1 x_0, \tag{2.7a}$$
$$x_i = x_{i-1} + h\,\sigma(W_i x_{i-1} + b_i), \tag{2.7b}$$
$$x_{L+1} = W_{L+1} x_L \tag{2.7c}$$

for all $i = 2, \dots, L$. All intermediate weight matrices $W_i \in \mathbb{R}^{n_{X_i} \times n_{X_{i-1}}}$ are square for $i = 2, \dots, L$ with widths $n_{X_1} = \dots = n_{X_L}$, while the initial and terminal weights have dimensions $W_1 \in \mathbb{R}^{n_{X_1} \times n_{X_0}}$ and $W_{L+1} \in \mathbb{R}^{n_{X_L} \times n_{X_i}}$. The bias vectors $b_i \in \mathbb{R}^{n_{X_i}}$ are also all of the same dimension. The activation function has the same role as in the FNN case, but is applied only to the output of the affine linear transformation in each layer, as shown in Equation (2.7b). Usually, the same activation function is used in each layer; accordingly, we do not denote it with an index i here. The parameter $h > 0$ is a discretization parameter that can be used to control the strength of the residual connections. The trainable parameters of the ResNet are comprised in the set

$$\theta = (W_1, (W_i, b_i)_{i=2}^{L}, W_{L+1}) \in \Theta.$$

The forward pass through this ResNet architecture up to x_L (2.7b) for an input $W_1 x_0 \in X_0$ can be interpreted as an explicit Euler discretization (cf. [52]) of a Ordinary Differential Equation (ODE) with a specific structure and initial value $W_1 x_0$ with discretization parameter h over the time interval $[0, Lh]$.

The associated continuous initial value problem has the form

$$\dot{x}(t) = \sigma(W(t)x(t) + b(t)) \quad \text{for } t \in [0, Lh], \tag{2.8}$$
$$x(0) = W_1 x_0, \tag{2.9}$$

where $W(t)$ is a time-dependent weight matrix, i.e. a function $W\colon [0, Lh] \to \mathbb{R}^{n_{X_1} \times n_{X_1}}$ and $b(t)$ is a time-dependent bias vector, i.e. a function $b\colon [0, Lh] \to \mathbb{R}^{n_{X_1}}$.

We additionally consider a further extension of the architecture inspired by [50] with propagation function

$$g(\theta, \cdot)\colon X_0 \to X_{L+1}, \quad x_0 \mapsto x_{L+1}$$

of the decomposed form

$$x_1 = W_1 x_0, \tag{2.10a}$$
$$x_i = x_{i-1} + W_i^{(2)}\,\sigma(W_i^{(1)} x_{i-1} + b_i), \tag{2.10b}$$
$$x_{L+1} = W_{L+1} x_L \tag{2.10c}$$

for all $i = 2, \dots, L$ employing two weight matrices per layer. The discretization parameter h is not present here, as it can be absorbed in the weight matrix $W_i^{(2)}$.

All intermediate weight matrices $W_i^{(1)}, W_i^{(2)} \in \mathbb{R}^{n_{X_i} \times n_{X_{i-1}}}$ are square for $i = 2, \dots, L$ with constant widths $n_{X_1} = \dots = n_{X_L}$, while the initial and terminal weights have dimensions $W_1 \in \mathbb{R}^{n_{X_1} \times n_{X_0}}$ and

$W_{L+1} \in \mathbb{R}^{n_{x_L} \times n_{x_i}}$. The bias vectors $b_i \in \mathbb{R}^{n_{x_i}}$ are also all of the same dimension.

Again, all weight matrices and biases comprise the set of trainable parameters

$$\theta = (W_1, (W_i^{(1)}, W_i^{(2)}, b_i)_{i=1}^{L}, W_{L+1}) \in \Theta.$$

The activation function has the same role as in the FNN, but is applied only to the output of the first affine linear transformation in each layer, as shown in Equation (2.10b).

The ResNet architecture is particularly useful for training very deep networks, as it helps to mitigate the vanishing gradient problem, where gradients become very small as they are backpropagated through many layers, making it difficult for the network to learn.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of neural network that is particularly well-suited for processing grid-like data, such as images. They are designed to automatically and adaptively learn spatial hierarchies of features from input data, making them highly effective for tasks such as image classification, object detection, and segmentation. A Convolutional Neural Network (CNN) consists of multiple layers, including convolutional layers, pooling layers, and fully-connected layers. The convolutional layers apply convolutional operations to the input of the layer (2.11a), while the pooling layers reduce the spatial dimensions of the data (2.11b), and the fully-connected layers perform classification or regression tasks (2.11d) and (2.11e).

We consider CNN architectures inspired by VGG networks, cf. [116], with L_1 convolutional layers and L_2 fully-connected layers. The baseline CNN architecture we use is shown in Figure 2.2.

The propagation function $g(\theta, \cdot)$ of the CNN has the decomposed form

$$\tilde{y}_k = \sigma(K_k \circledast \tilde{x}_{k-1} + b_k) \quad \text{for } k = 1, \dots, L_1, \tag{2.11a}$$

$$\tilde{x}_k = \text{MaxPool}(\tilde{y}_k) \quad \text{for } k = 1, \dots, L_1, \tag{2.11b}$$

$$x^{L_1} = \text{Flatten}(\tilde{x}_{L_1}), \tag{2.11c}$$

$$x_{k+1} = \sigma(W_k x_k + b_k^{\text{fc}}) \quad \text{for } k = L_1, \dots, L_1 + L_2 - 2, \tag{2.11d}$$

$$x_{L_1+L_2} = W_{L_1+L_2} x_{L_1+L_2-1}. \tag{2.11e}$$

The tensor $\tilde{x}_0 \in \mathbb{R}^{m_0 \times d_0 \times d_0}$ denotes the network's input with m_0 being the number of channels and $d_0 \times d_0$ the spatial dimensions of each channel. The features $\tilde{x}_k \in \mathbb{R}^{m_k \times d_k \times d_k}$ for $k = 1, \dots, L_1$ are the output of the convolutional layers, where m_k is the number of channels and $d_k \times d_k$ are the spatial dimensions of each channel. The convolutional kernels $K_k \in \mathbb{R}^{3 \times 3 \times m_k \times m_{k-1}}$ are applied to the input features $\tilde{x}_{k-1}$ with padding and stride values of 1, so that the spatial dimensions of $\tilde{x}_{k-1}$ are preserved. The bias vectors $b_k \in \mathbb{R}^{m_k}$ are applied channel-wise and the activation function σ is applied element-wise. The operation MaxPool is the Max-Pooling operation (cf. [107]) with a kernel size of 2×2 and stride 2. The operation Flatten transforms the tensor of dimension $m_{L_1} \times d_{L_1-1} \times d_{L_1-1}$ to a vector of dimension $m_{L_1} d_{L_1-1}^2 =: d_{L_1}$, so that the features of the last convolutional layer $\tilde{x}_{L_1}$ are transformed to a

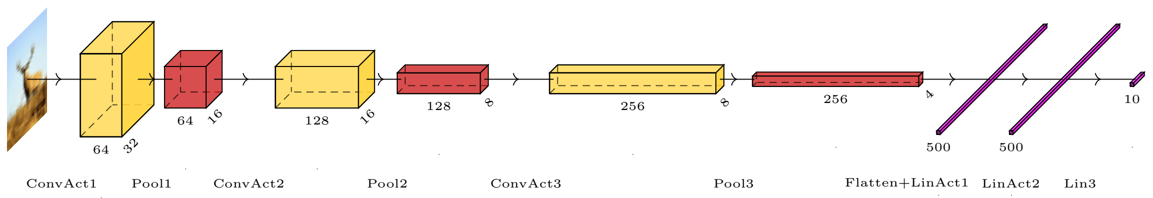


Figure 2.2.: An illustration of an exemplary VGG-inspired CNN architecture suited for the CIFAR-10 data set [76].

vector $x^{L_1} \in \mathbb{R}^{d_{L_1}}$. The features $x_k \in \mathbb{R}^{d_k}$ for $k = L_1, \dots, L_1 + L_2 - 1$ are the features of the fully-connected layers, where $d_k \coloneqq \frac{d_{k-1}}{2}$ is the dimension of the k-th fully-connected layer. The output of the CNN is given by $x_{L_1+L_2} \in \mathbb{R}^{d_{L_1+L_2}} = \mathbb{R}^c$, where c is the number of classes in the classification task. The trainable parameters θ comprise the kernels $K_k \in \mathbb{R}^{3\times 3\times m_k \times m_{k-1}}$ for $k = 1, \dots, L_1$, the bias vectors $b_k \in \mathbb{R}^{m_k}$ for $k = 1, \dots, L_1$, and the weight matrices $W_k \in \mathbb{R}^{d_{k+1}\times d_k}$ and bias vectors $b_k^{\text{fc}} \in \mathbb{R}^{d_{k+1}}$ of the fully-connected layers for $k = L_1, \dots, L_1 + L_2 - 1$. The set of all trainable parameters is given by

$$\theta = (K_k, b_k)_{k=1}^{L_1} \cup (W_k, b^{\text{fc}_k})_{k=L_1}^{L_1+L_2-1} \in \Theta.$$

For a more detailed introduction to CNNs, we refer the reader to [42, Chapter 9].

2.2.2. Supervised Learning

Every deep learning task relies on the availability of data. If all data are only available without a corresponding label, the learning task is called *unsupervised learning*. If some labels are available, but not for all data, the learning task is called *semi-supervised learning*. Finally, if all data is available with a corresponding label, the learning task is called *supervised learning*. For different types of learning problems, different optimizers and loss functions are used. In this thesis, we focus on supervised learning tasks, where the goal is to learn a mapping (within the hypothesis class of networks) from input data to output based on labeled training data. The training data consists of pairs of input features and output features, where the input features are the data points and the output features are the corresponding labels.

The nature of the supervised learning task can be categorized into two main types: *classification* problems and *regression* problems. An example for a classification problem is the task of assigning images into different categories, such as recognizing handwritten digits or identifying objects in images. In this case, the output features are discrete labels representing the categories. An example for a regression problem is the task of predicting a continuous value, such as the price of an object based on its features. In this case, the output features are continuous values.

As an example, in image classification tasks, the input features consist of images, such as those depicting handwritten digits, while the output features are the corresponding class labels, i.e., *digit 0* or *digit 1*. The goal is to find a collection of trainable parameters $\theta^* \in \Theta$ and the associated neural network function $g(\theta^*, \cdot) : X_0 \to Y_L$, such that this network can predict the class label of a new image depicting a new handwritten digit, which was not part of the training data.

To model the goal of the learning task, we need a *loss function*

$$\mathcal{L} : Y_L \times Y_L \to \mathbb{R}$$

which is able to quantify the difference between the predicted output of the network and the true output given by the label of the data. The appropriate loss function depends on the nature of the supervised learning task.

For regression tasks, a common loss function is the *Mean Squared Error (MSE)* loss, which is defined as

$$\mathcal{L}_{\text{MSE}}(y_{\text{pred}}, y_{\text{label}}) := \frac{1}{2}\|y_{\text{pred}} - y_{\text{label}}\|^2, \tag{2.12}$$

where y_{pred} is the predicted output of the network and y_{label} is the true output given by the label of the data and $\|\cdot\|$ is a norm on the output space which is suited for the task at hand such as, most commonly, the Euclidean norm or e.g. Sobolev norms [62].

For classification tasks, the most exact loss function is the *0-1 loss*, which is defined as

$$\mathcal{L}_{\text{0-1}}(y_{\text{pred}}, y_{\text{label}}) := \begin{cases} 0 & \text{if } y_{\text{pred}} = y_{\text{label}}, \\ 1 & \text{if } y_{\text{pred}} \neq y_{\text{label}}. \end{cases}$$

However, the 0-1 loss function is not even continuous and hence not suitable for training neural networks from an optimization perspective. A popular alternative loss function is the *Cross-Entropy (CE)*-loss, which is defined as

$$\mathcal{L}_{\text{CE}}(y_{\text{pred}}, y_{\text{label}}) := -\sum_{i=1}^{c}(y_{\text{label}})_i \log((y_{\text{pred}})_i), \tag{2.13}$$

where $(y_{\text{pred}})_i$ is the predicted probability of class i and $(y_{\text{label}})_i$ is the true probability of class i and c is the number of classes. The CE-loss measures the difference between the predicted probability distribution and the true probability distribution of the classes. It is often used in combination with the *softmax* activation function in the output layer of the neural network, which converts the raw output of the network into a probability distribution over the classes. The softmax function is defined as

$$\text{softmax}(y_{\text{pred}})_i := \frac{\exp((y_{\text{pred}})_i)}{\sum_{j=1}^{c}\exp((y_{\text{pred}})_j)}, \tag{2.14}$$

where $(y_{\text{pred}})_i$ is the raw output of the network (e.g. y_L for FNNs, see (2.5)) for class i and c is the number of classes. The softmax function ensures that the predicted probabilities sum to 1, making them interpretable as probabilities. The CE-loss is then used to train the network to produce accurate class probabilities for the input data. The final model output for classification tasks is hence given by $\text{softmax}(g(\theta, x_0))$ for an input $x_0 \in X_0$ and gives the predicted class probabilities. The biggest class probability is then used to determine the predicted class label.

There are other types of supervised learning tasks, such as *autoencoder* training (cf. [61]), where the goal is to learn a compressed representation of the input data. For this task, the used neural network

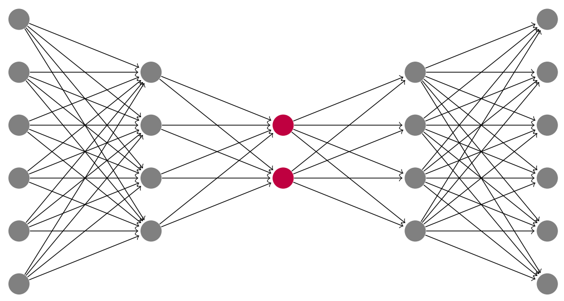


Figure 2.3.: An illustration of an exemplary FNN autoencoder architecture with the bottleneck layer colored in red.

is called an *autoencoder* and consists of an encoder followed by a decoder. Between the encoder and decoder, there is a bottleneck layer that forces the network to learn a compressed representation of the input data. An example for an autoencoder architecture is shown in Figure 2.3. In this case, the input features and output features are the same, and the loss function is typically the MSE-loss.

$$\mathcal{L}_{\mathrm{MSE}}(g(\theta, x_0), x_0) := \frac{1}{2}\|g(\theta, x_0) - x_0\|^2.$$

Autoencoders can be used for tasks such as dimensionality reduction, de-noising, and anomaly detection. Generally, training autoencoders is a more complex task than training networks for classification or regression, as the network has to learn a compressed representation of the input data while still being able to reconstruct it accurately. We are going to use autoencoder training as one of the training tasks in Chapter 5.

Next, we formulate the supervised learning task as an optimization problem. The goal is to find a set of parameters $\theta \in \Theta$ such that the expected loss of the network output with these parameters over the data of interest is minimized. This is done by minimizing the *expected loss* which is also referred to as the *exact risk* given by

$$f(\theta) := \mathbb{E}_{X,Y}[\mathcal{L}(g(\theta, X), Y)] \tag{2.15}$$

over the set of all trainable parameters Θ. The minimization problem is unconstrained and can be formulated as:

$$\text{Minimize} \quad f(\theta) \quad \text{over } \theta \in \Theta. \tag{2.16}$$

The expectation $\mathbb{E}_{X,Y}$ is taken w.r.t. to the joint distribution of the random variables X (with values in X_0) and Y (with values in Y_L) which represent, e.g. in image classification, the images and their associated class labels (or class label probabilities).

However, since the joint distribution of the random variables X and Y is rarely known, we cannot compute the objective function of the optimization problem (2.16) or its derivatives.

In practice, we have only access to a finite set of labeled training data consisting of D pairs

$$(x_{0,j}, y_{\mathrm{label}j}) \text{ for } j = 1, \dots, D,$$

which is a finite set of samples from the joint distribution of the random variables X and Y. We replace the expectation in (2.16) by the empirical mean over the training data, which allows us to approximate the expected loss f by the *empirical loss* or *empirical risk* given by

$$\hat{f}(\theta) := \frac{1}{D}\sum_{j=1}^{D} \mathcal{L}(g(\theta, x_{0,j}), y_{\text{label}j}). \tag{2.17}$$

We introduce the notation

$$\hat{f}(\theta) := \frac{1}{D}\sum_{j=1}^{D} f_j(\theta) \quad \text{with } f_j(\theta) := \mathcal{L}(g(\theta, x_{0,j}), y_{\text{label}j}), \tag{2.18}$$

i.e. the empirical risk $\hat{f}(\theta)$ is the average of the individual losses $f_j(\theta)$.

It is often beneficial to add a regularization term to the empirical risk in order to avoid overfitting and to improve the generalization abilities of the network. This leads to the *regularized empirical risk* given by

$$\hat{f}_\lambda(\theta) := \hat{f}(\theta) + \lambda \cdot \|\theta\|^2. \tag{2.19}$$

Here, $\lambda > 0$ is a regularization parameter and $\|\theta\|$ is a suitable norm on the parameter space Θ. The regularization term $\lambda \cdot \|\theta\|^2$ penalizes large parameter values, which can lead to overfitting on the training data.

To summarize, for classification tasks the most exact formulation of the training goal is given by the optimization problem

$$\text{Minimize} \quad \mathbb{E}_{X,Y}[\mathcal{L}_{0\text{-}1}(g(\theta, X), Y)] \quad \text{over } \theta \in \Theta. \tag{2.20}$$

However, to account for the fact that we do not have access to the joint distribution of the random variables X and Y and to work with a differentiable loss function, we use the following optimization problem in practice:

$$\text{Minimize} \quad \frac{1}{D}\sum_{j=1}^{D} \mathcal{L}_{\text{CE}}(\text{softmax}(g(\theta, x_{0,j})), y_{\text{label}j}) + \lambda \cdot \|\theta\|^2 \quad \text{over } \theta \in \Theta. \tag{2.21}$$

These modifications to the optimization problems introduce different types of errors. The first type of error is the *estimation error* which arises from the fact that we only have access to a finite set of training data D and hence cannot compute the exact expected loss (2.16). This error is related to the generalization error in the literature and overfitting. The second type of error is the *optimization error* which arises from the fact that the optimization problem (2.21) is non-convex and highly complex, making it nearly impossible to find a global minimum of the training problem (2.21). Classical optimization methods for non-convex problems can only approximate local minima or even saddle points, especially since the loss landscape of deep learning problems is highly complex. The necessity of using the CE-loss function instead of the 0-1 loss function leads to an error as well. The third type of error is the *approximation error* which arises from the choice of the model class $g(\theta, \cdot)$. The model class $g(\theta, \cdot)$ is typically a neural network with a finite number of parameters, which can only approximate the true function g^* to a certain degree. This leads to an approximation error in the training problem

(2.21). The broader the model class is chosen, as e.g. for neural networks more layers are chosen or wider layers are used, the more approximation capacity the model class has and hence the smaller the approximation error is, see e.g. [20, 86]. Before training, it is not clear how large the model class has to be chosen in order to achieve a sufficient approximation of the true function g^*.

Backpropagation

Common iterative optimization methods used to solve the optimization problem (2.21) require the computation of the derivative of the loss function $\mathcal{L}$ (for a selected subset of training data) with respect to the trainable parameters θ of the network. Because of the special structure of the neural network, the computation of the gradient can be done efficiently using the *backpropagation* algorithm (cf. [110, 26] for some first works). The backpropagation algorithm consists of two main steps: the forward pass and the backward pass.

In the forward pass, the input data is passed through the network to compute the output features. The forward pass through the network for a given input $x_0 \in X_0$ (neglecting the last pass through the loss function) is shown in Section 2.2.1, more specifically for e.g. FNNs in (2.5).

In the backward pass, the derivatives of the loss function with respect to the trainable parameters are computed using the chain rule of calculus. The derivatives of the loss function with respect to the output features are computed and then propagated back through the network while computing the derivatives w.r.t. the hidden features to compute the derivatives with respect to the trainable weight matrices and biases. We describe the backward pass for a FNN in the following. For simplicity, we consider the case for the loss function computed for one training datum and without regularization. Extending to a loss computed for a mini-batch of training data and including regularization is straightforward. The backward pass through the network for one individual training loss f_j (defined in (2.18)) is derived by the following equations.

The chain rule produces connections between the derivatives of the pre- and post-activation features by

$$\frac{\partial f_j(\theta)}{\partial x_{i-1}}(\cdot) = \frac{\partial f_j(\theta)}{\partial y_i}(W_i\,\cdot) \quad \text{for } i = 1, \dots, L, \tag{2.22}$$

$$\frac{\partial f_j(\theta)}{\partial y_i}(\cdot) = \frac{\partial f_j(\theta)}{\partial x_i}(\sigma_i'(y_i)\,\cdot) \quad \text{for } i = 1, \dots, L-1, \tag{2.23}$$

with the dual objects $\frac{\partial f_j(\theta)}{\partial x_i} \in X_i^*$ for $i = 1, \dots, L$ and $\frac{\partial f_j(\theta)}{\partial y_i} \in Y_i^*$ for $i = 1, \dots, L-1$.

The derivatives of f_j w.r.t. the trainable parameters lie in

$$\frac{\partial f_j(\theta)}{\partial W_i} \in L(Y_i^*, X_{i-1}^*) \quad \text{for } i = 1, \dots, L, \tag{2.24}$$

$$\frac{\partial f_j(\theta)}{\partial b_i} \in Y_i^* \quad \text{for } i = 1, \dots, L. \tag{2.25}$$

They have the evaluations

$$\frac{\partial f_j(\theta)}{\partial W_i}(\delta W) = \frac{\partial f_j(\theta)}{\partial y_i}(\delta W x_{i-1}) \quad \text{for } i = 1, \ldots, L, \tag{2.26}$$

$$\frac{\partial f_j(\theta)}{\partial b_i}(\delta b) = \frac{\partial f_j(\theta)}{\partial y_i}(\delta b) \text{ with } \quad \text{for } i = 1, \ldots, L, \tag{2.27}$$

for all $\delta W \in L(X_{i-1}, Y_i)$ and $\delta b \in Y_i$.

In most cases, the spaces Y_i and X_{i-1} are Euclidean spaces of the type $\mathbb{R}^n$ for some $n \in \mathbb{N}$, i.e. finite-dimensional vector spaces with the Euclidean inner product. The backpropagated Euclidean gradients of the pre- and post-activation features are consequently given by

$$\begin{aligned} \nabla_{x_{i-1}} f_j(\theta) &= W_i^T \nabla_{y_i} f_j(\theta) \quad \text{for } i = 1, \ldots, L, \\ \nabla_{y_i} f_j(\theta) &= \sigma_i'(y_i)^T \nabla_{x_i} f_j(\theta) \quad \text{for } i = 1, \ldots, L-1. \end{aligned}$$

Generally, the transposes can be understood of adjoints with respect to the Euclidean inner products on the layer spaces.

The Euclidean gradient of $\mathcal{L}$ w.r.t. a weight matrix W and a bias vector b is given by

$$\nabla_{W_i} f_j(\theta) = \nabla_{y_i} f_j(\theta) x_{i-1}^T, \tag{2.28}$$

$$\nabla_{b_i} f_j(\theta) = \nabla_{y_i} f_j(\theta). \tag{2.29}$$

The Euclidean gradient of the loss function f_j w.r.t. a weight matrix W_i is given by the product of the backpropagated gradient of the pre-activation features $\nabla_{y_i} f_j(\theta)$ and the post-activation features x_{i-1}^T and hence has a rank of at most one when the loss w.r.t. one training sample is computed. The specific structure of the gradient (2.28) will be exploited in Chapter 5.

It can also be determined by the solution of the quadratic minimization problem

$$\text{Minimize } f_j'(\theta)\delta W + \frac{1}{2}(\delta W, \delta W)_F \text{ over } \delta W \in \mathbb{R}^{n_{Y_i} \times n_{X_{i-1}}},$$

where $(\cdot,\cdot)_F$ is the Frobenius inner product on the space of matrices

$$(A, B)_F := \operatorname{trace}(A^T B) \quad \text{for } A, B \in \mathbb{R}^{n_{Y_i} \times n_{X_{i-1}}}. \tag{2.30}$$

Consequently, the Euclidean gradient of the loss function $\hat{f}$ w.r.t. the trainable parameters θ is given by

$$\begin{aligned} -\nabla_{W_i} \hat{f}(\theta) &= -\frac{1}{D} \sum_{j=1}^{D} \nabla_{y_i} f_j(\theta) x_{i-1}(j)^T, \\ -\nabla_{b_i} \hat{f}(\theta) &= -\frac{1}{D} \sum_{j=1}^{D} \nabla_{y_i} f_j(\theta), \end{aligned}$$

where $x_{i-1}(j)$ are the post-activation features of the $i-1$-th layer computed during the forward pass for the j-th training datum. The Euclidean gradient of the loss function $\hat{f}$ w.r.t. the trainable parameters θ is given by the sum of the gradients of the individual losses f_j w.r.t. the trainable parameters θ and hence has a rank of at most D.

Data Sets

In the following, we describe the data sets which we use in this thesis for numerical experiments.

Spiral Data Set The spiral data set is a synthetic data set that is often used as a very simple problem for classification tasks. The version used in this work consists of 600 data points with labels, where each data point is composed of a two-dimensional feature vector $x_j \in \mathbb{R}^2$ and a label that indicates whether it belongs to the red or blue spiral, i.e., $y_j \in \{\binom{0}{1}, \binom{1}{0}\} \subset \mathbb{R}^2$. For training, we use 450 data points, while the remaining 150 data points form the test set. An illustration of the spiral data set is shown in Figure 2.4. The spiral data set presented here is implemented by us. It can be made more complex and hence challenging by adding noise to the data points or by increasing the number of spirals or by decreasing the distance between the start of the spirals in the center.

MNIST [84]. The MNIST dataset contains 28×28 grayscale images of handwritten digits and corresponding digit labels (0-9). It consists of 60,000 training images and 10,000 test images. It is a widely used benchmark for small-scale image classification and for basic experiments in representation learning and autoencoder training, as e.g. in the benchmark in [61]. An example subset of images is shown in Figure 2.5. In this thesis MNIST is employed both as a classification benchmark and as a target for autoencoder experiments in Chapter 5.

CIFAR-10 [77]. CIFAR-10 consists of 32×32 color images (i.e. 3 channels) grouped into ten object classes and is a standard benchmark for convolutional network performance in small-scale image classification. It contains 50,000 training images and 10,000 test images. Here we use the standard data augmentation techniques before training. A selection of examples is shown in Figure 2.6. We use CIFAR-10 to evaluate methods on a dataset suited for convolutional architectures.

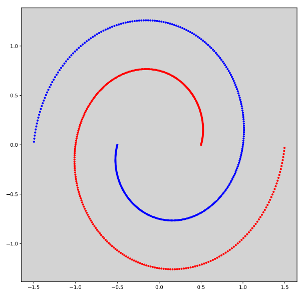


Figure 2.4.: Illustration of the spiral data set.

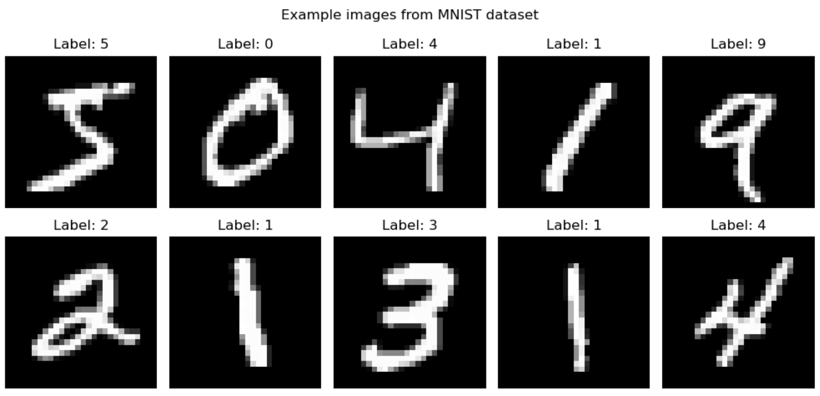


Figure 2.5.: Example images from the MNIST dataset.

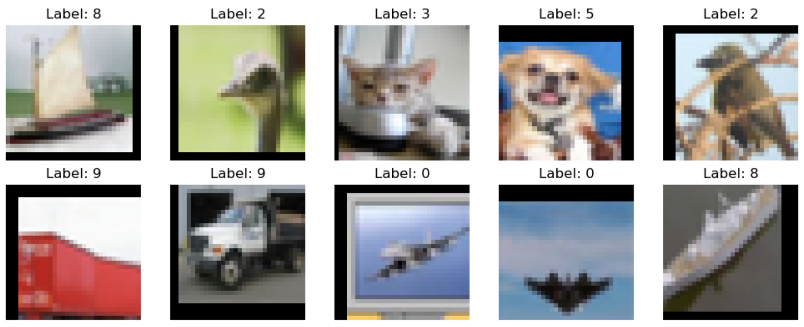


Figure 2.6.: Example images from the CIFAR-10 dataset.

2.3. Optimization

In this section, we give an overview of selected first-order optimization methods that can be used to solve the optimization problem (2.21) in deep learning. Usually, stochastic methods are used to solve the optimization problem (2.21) in deep learning, but we also consider some deterministic methods as a foundation for the analysis. Despite the fact that a deep learning optimization problem (2.21) is always non-convex, we examine the behavior of some methods for strongly convex problems (and sometimes even quadratic problems) as well, as they are often used as a starting point for the analysis of non-convex problems or are the only case where convergence to a global minimum can be guaranteed by theory. The methods presented here are used to develop new methods in the main contributions of this thesis in Chapter 3 and Chapter 4 and Chapter 5.

The section is structured as follows. We introduce the Gradient Descent (GD) method in Section 2.3.1 as the deterministic method, on which stochastic gradient methods are based. We present its convergence behavior for fixed step sizes in the strongly convex and non-convex case. In Section 2.3.2, we introduce the Stochastic Gradient Descent (SGD) method, which is the core method used in deep learning. We analyze its convergence behavior for fixed step sizes in the strong convex and non-convex case and also consider the case of Robbins-Monro step sizes [109] in the non-convex case. We also introduce the stochastic gradient method with momentum, which is a common variant of the SGD method. Finally, we consider sensitivity analysis in Section 2.3.3 for optimization problems subject to constraints.

2.3.1. Gradient Descent (GD)

Let us assume that the space of parameters Θ is a Hilbert space with an inner product $(\cdot,\cdot)$. In the following, we denote the associated norm by

$$\|\cdot\| := \sqrt{(\cdot,\cdot)}$$

and all quantities which depend on the choice of the inner product or norm are meant with respect to this inner product or norm, respectively. In the following, all assumptions and results are stated with respect to this inner product and norm, if not stated otherwise. We consider the optimization problem

$$\text{Minimize } f(\theta) \quad \text{over } \theta \in \Theta, \tag{2.31}$$

where $f : \Theta \to \mathbb{R}$ is a differentiable function, which is bounded from below. The GD method is an iterative optimization method that updates the parameters θ in the direction of the negative gradient (w.r.t. the inner product $(\cdot,\cdot)$) of the objective function f. The update rule for the GD method is given by

$$\theta^{k+1} = \theta^k - \alpha_k \nabla f(\theta^k), \tag{2.32}$$

where $\alpha_k > 0$ is the step size and $\nabla f(\theta^k)$ is the gradient (w.r.t. $(\cdot,\cdot)$) of the objective function f at the current parameters θ^k. The GD method is a first-order optimization method, as it only uses the first derivative of the objective function f to update the parameters. For more details on the GD method, we refer the reader to e.g. [98].

In the following we state selected assumptions and convergence results for the GD method with fixed step sizes in the strongly convex and non-convex case. We denote the optimal solution of the optimization problem (2.31) by $\theta^* \in \Theta$ and the initial parameters by $\theta^0 \in \Theta$. The iterates of the GD method are denoted by $\theta^k \in \Theta$ for $k \in \mathbb{N}$.

Assumption 2.13 (L-Smooth Objective Function). *The objective function $f : \Theta \to \mathbb{R}$ is L-smooth, i.e. there exists a constant $L > 0$ such that*

$$\|\nabla f(\theta_1) - \nabla f(\theta_2)\| \le L\|\theta_1 - \theta_2\|$$

for all $\theta_1, \theta_2 \in \Theta$.

Assumption 2.14 (Objective Function Bounded from Below). *The objective function $f : \Theta \to \mathbb{R}$ is bounded below, i.e. there exists a constant $f_{inf} \in \mathbb{R}$ such that*

$$f(\theta) \ge f_{inf}$$

for all $\theta \in \Theta$.

Assumption 2.15 (Fixed Step Sizes). *The step size is fixed for all iterations*

$$\alpha_k = \alpha > 0 \quad \textit{for all } k \in \mathbb{N},$$

and satisfies the condition

$$\alpha \leq \frac{2}{L},$$

where L is a Lipschitz-smoothness constant of f, i.e. a Lipschitz constant of the gradient of the objective function f.

Assumption 2.16 (Strong Convexity of the Objective Function). *The objective function $f : \Theta \longrightarrow \mathbb{R}$ is μ-strongly convex, i.e. there exists a constant $\mu > 0$ such that*

$$f(\theta_1) \geq f(\theta_2) + (\nabla f(\theta_2), \theta_1 - \theta_2) + \frac{\mu}{2}\|\theta_1 - \theta_2\|^2$$

for all $\theta_1, \theta_2 \in \Theta$.

Theorem 2.17 (Convergence of GD for Strongly Convex Functions with Fix Step Sizes). *Let assumptions 2.13 to 2.16 hold. Then,*

$$\|\theta^k - \theta^*\|^2 \leq (1 - \alpha\mu)^k \, \|\theta^0 - \theta^*\|^2.$$

In particular, if $\alpha = \frac{2}{L}$, then

$$\|\theta^k - \theta^*\|^2 \leq \left(1 - \frac{2\mu}{L}\right)^k \|\theta^0 - \theta^*\|^2.$$

Hence, the convergence rate is linear in the sense that

$$\|\theta^k - \theta^*\|^2 \leq C\left(1 - \frac{2\mu}{L}\right)^k \quad \textit{for some constant } C > 0.$$

The optimal step size is given by $\alpha = \frac{2}{L}$ and leads to the best convergence rate.

This result is a special case derived from [38, Theorem 3.6].

Theorem 2.18 (Convergence of GD for Non-Convex Functions with Fix Step Sizes). *Let assumptions 2.13 to 2.15 hold. Then,*

$$\sum_{k=1}^{\infty} \|\nabla f(\theta^k)\|^2 < \infty$$

and hence

$$\lim_{k \to \infty} \|\nabla f(\theta^k)\| = 0.$$

This result is a special case derived from [11, Theorem 4.8].

2.3.2. Stochastic Gradient Descent (SGD)

Stochastic optimization methods are used to solve optimization problems where the objective function f can not be directly evaluated, e.g. because it is given by an expectation over an unknown probability distribution as in (2.15). Alternatively, stochastic optimization methods are used to solve optimization problems where the objective function is too expensive to compute exactly or often, because it is given by a sum over a large number of samples. Stochastic optimization methods differ from deterministic optimization methods in the aspect that they use random samples to estimate the objective function and its gradient and use them to update the parameters. The most common stochastic optimization method is the SGD method, which is the stochastic version of the Gradient Descent method introduced in Section 2.3.1.

Example 2.19 (Machine Learning Loss Function). *In machine learning, we aim to minimize and learn a general objective function, such as*

$$f(\theta) := \mathbb{E}_{X,Y}[\mathcal{L}(g(\theta, X), Y)]$$

for a model class (not necessarily a neural network) $g(\theta, \cdot)$ with parameters $\theta \in \Theta$ and a loss function $\mathcal{L} : Y_L \times Y_L \longrightarrow \mathbb{R}$. Since the distribution $\mathbb{P}$ of X, Y is unknown, the available training data $(x_{0,j}, y_{labelj})_{j=1,\cdots,D}$ is used to estimate the gradient of the loss function. The empirical ('known') loss function is given by

$$\hat{f}(\theta) = \frac{1}{D} \sum_{j=1}^{D} \mathcal{L}(g(\theta, x_{0,j}), y_{labelj}).$$

Commonly, the gradient estimates are constructed with mini-batches of the training data, i.e.

$$G(\theta^k, \xi_k) := \frac{1}{B} \sum_{j \in S_B^k} \nabla \mathcal{L}(g(\theta^k, x_{0,j}), y_{labelj}),$$

where $\xi_k := (x_{0,j}, y_{labelj})_{j \in S_B^k}$ is a mini-batch of size B of the training data. The random variable ξ_k comprises the randomness by which the training data is sampled of the mini-batch at iteration k. The mini-batch can also consist of a single training datum ($B = 1$). Here, $S_B^k \subseteq \{1, \ldots, D\}$ is the set of indices of the training data used in the mini-batch at iteration k.

The SGD update [109] is given by

$$\theta^{k+1} = \theta^k - \alpha_k G(\theta^k, \xi_k),$$

where $\alpha_k > 0$ is the step size. The random variable ξ_k represents the randomness of the gradient estimate at iteration k and $G(\theta^k, \xi_k)$ is an (unbiased) gradient estimate of the gradient $\nabla f(\theta^k)$.

For the convergence analysis of stochastic gradient methods, we need to make some assumptions on the objective function and the gradient estimates.

First we assume that f is differentiable and satisfies assumptions 2.13 and 2.14.

Assumption 2.20 (Samples for Gradient Estimates). *The random variables ξ_k from which samples are used to compute estimates of the gradients of the objective function f are independent and identically distributed (i.i.d.). We have access to an unlimited number of i.i.d. random variables ξ_k and their samples.*

Remark 2.21 (Access to Unlimited Number of Samples). *For the convergence theory of SGD, we assume that we have access to an unlimited number of i.i.d. random variables ξ_k and their samples. This would imply that we have an infinite number of training data samples available. In practice, the number of training data samples is finite and hence this assumption is violated. Each epoch of training reuses the training data samples in a different order, which can be seen as a randomization of the training data.*

Assumption 2.22 (Unbiased Gradient Estimate). *The gradient estimate $G(\theta^k, \xi_k)$ is an unbiased estimate of the exact gradient $\nabla f(\theta^k)$, i.e.*

$$\mathbb{E}_{\xi_k}[G(\theta, \xi_k)] = \nabla f(\theta) \quad \textit{for all } \theta \in \Theta.$$

Remark 2.23 (Definition of Exact Objective Function). *The objective function f can be the exact objective function, i.e. the expectation* (2.15)*, or the empirical objective function, i.e. the empirical risk* (2.17) *over all training data. The gradient estimates computed using mini-batches of the training data are unbiased estimates of the exact gradient of both choices of the objective function. When methods such as the Stochastic Variance Reduced Gradient (SVRG) [69] are developed, where access to the exact gradient is required, the objective function averaging over the complete training data is considered to be the exact objective function, i.e.*

$$f(\theta) \coloneqq \frac{1}{D} \sum_{j=1}^{D} \mathcal{L}(g(\theta, x_{0,j}), y_{labelj}).$$

We also need to make some assumptions that lets us control the variance of the gradient estimates. A common assumption is given by the following.

Assumption 2.24 (Bounded Variance of Gradient Estimates). *There exist constants $M, M_G > 0$ such that*

$$\mathbb{E}_{\xi_k}[\|G(\theta, \xi_k)\|^2] \leq M^2 + M_G \|\nabla f(\theta)\|^2$$

for all $\theta \in \Theta$.

Remark 2.25 (Variance Bounds). *The variance of the gradient estimate $G(\theta, \xi_k)$ can be bounded in different ways. An alternative bound to assumption 2.24 is given by*

$$\mathbb{E}_{\xi_k}[\|G(\theta, \xi_k)\|^2] \leq M^2 + M_G \|\nabla f(\theta)\|^2 + M_f\,(f(\theta) - f_{min})$$

for all $\theta \in \Theta$.

Assumption 2.26 (Fixed Step Size). *The step size is fixed for all iterations*

$$\alpha_k = \alpha > 0 \quad \textit{for all } k \in \mathbb{N},$$

and satisfies the condition

$$\alpha \leq \frac{1}{LM_G},$$

where L is a Lipschitz-smoothness constant of f, i.e. a Lipschitz constant of the gradient of the objective function f, and M_G is the constant from assumption 2.24.

We first show the convergence of SGD for strongly convex objective functions (assumption 2.16) with fixed step sizes.

Theorem 2.27 (Convergence of SGD for Strongly Convex Objective Function with Fixed Step Sizes). *Let assumptions 2.13, 2.14, 2.16, 2.20, 2.22, 2.24 and 2.26 hold. Then, the expected optimality gap satisfies the following inequality for all $k \in \mathbb{N}$:*

$$\begin{aligned}\mathbb{E}[f(\theta^k) - f(\theta^*)] &\leq \frac{\alpha LM}{2\mu} + (1 - \alpha\mu)^k \left(f(\theta_0) - f(\theta^*) - \frac{\alpha LM}{2\mu}\right) \\ &\to \frac{\alpha LM}{2\mu} \quad \textit{as } k \to \infty.\end{aligned}$$

From [11, Theorem 4.6].

Theorem 2.28 (Convergence of SGD for Non-Convex Objective Function with Fixed Step Sizes). *Let assumptions 2.13, 2.14, 2.20, 2.22, 2.24 and 2.26 hold. Then,*

$$\mathbb{E}[\sum_{k=1}^{K} \|\nabla f(\theta^k)\|^2] \leq K\alpha LM + \frac{2(f(\theta_0) - f(\theta^*))}{\alpha},$$

and hence

$$\frac{1}{K}\sum_{k=1}^{K} \mathbb{E}[\|\nabla f(\theta^k)\|^2] \to \alpha LM \quad \textit{as } K \to \infty.$$

From [11, Theorem 4.8].

Assumption 2.29 (Robbins-Monro Step Sizes [109]). *The step size is diminishing, i.e. there exists a sequence $(\alpha_k)_{k\in\mathbb{N}}$ such that*

$$\alpha_k \to 0 \quad \textit{as } k \to \infty,$$

and

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

Additionally, the step sizes satisfies the condition

$$0 \leq \alpha_k \leq \frac{1}{LM_G},$$

where L is a Lipschitz-smoothness constant of f.

Theorem 2.30 (Convergence of SGD for Non-Convex Objective Function with Diminishing Step Sizes)**.** *Let assumptions 2.13, 2.14, 2.20, 2.22, 2.24 and 2.29 hold. Then,*

$$\mathbb{E}\left[\frac{1}{A_k}\sum_{k=1}^{K}\alpha_k\|\nabla f(\theta^k)\|^2\right] \to 0 \quad \text{as } K \to \infty,$$

and hence

$$\liminf_{k\to\infty} \mathbb{E}[\|\nabla f(\theta^k)\|^2] = 0.$$

From [11, Theorem 4.9 and Theorem 4.10].

Theorem 2.31 (Almost Sure Convergence of SGD for Non-Convex Objective Function with Diminishing Step Sizes)**.** *Let assumptions 2.13, 2.14, 2.20, 2.22, 2.24 and 2.29 hold. Then,*

$$\liminf_{k\to\infty} \|\nabla f(\theta^k)\|^2 = 0 \quad \text{almost surely.}$$

This is shown e.g. in [127, Theorem 2.1].

The SGD method has the following advantages and difficulties. On the one hand, the biggest power of SGD is that it can be used to solve optimization problems with unknown objective functions, requiring only (unlimited) access to unbiased estimates of the gradient. Additionally, the method is cheaper than evaluating the true gradient at each iteration and can help to escape local minima in very non-convex landscapes by introducing noise.

This however also leads to difficulties, since the stochastic nature of the method introduces noise, which can make its behavior harder to detect and analyze. The convergence theory of SGD is weaker than that of gradient descent (in the strongly convex setting), as it is only able to guarantee convergence to a plateau around the minimizer and not to the exact minimizer. Furthermore, the method introduces additional hyperparameters concerning the computation of the gradient estimates (e.g. mini-batches), which can lead to additional tuning effort. Finally, the method is sensitive to the choice of the step size, which can lead to divergence or slow convergence if not chosen properly.

A well-known variant of the SGD method is SGD with momentum [105], which is a common method used in deep learning. The update rule for SGD with momentum is given by

$$\begin{aligned}\theta^{k+1} &:= \theta^k - \alpha_k m^k,\\ m^k &:= G(\theta^k, \xi_k) + \mu^k m^{k-1},\end{aligned}$$

where $\mu^k \in [0,1)$ are the momentum parameters and m^k is the momentum term. We set $m^{-1} = 0$. The momentum parameter μ^k controls the amount of momentum that is added to the update. It is a weighted average of the previous updates and helps to smooth the updates and reduce the noise introduced by the stochastic gradient estimates. Adding this momentum term can help to escape local minima and saddle points in the optimization landscape.

There exists convergence results for SGD with momentum, but only the case of convex objective functions.

Theorem 2.32 (Convergence of SGD with Momentum for Convex Objective Function). *Assume that the objective function $\hat{f}$ is a sum of convex functions, i.e.*

$$\hat{f}(\theta) = \sum_{j=1}^{D} f_j(\theta),$$

where each f_j is convex and differentiable. Additionally, we assume that the gradient of each f_j is Lipschitz continuous with constant L_{mb}, i.e.

$$\|\nabla f_j(\theta_1) - \nabla f_j(\theta_2)\| \leq L_{mb}\|\theta_1 - \theta_2\|$$

for all $\theta_1, \theta_2 \in \Theta$ and $\hat{f}$ is bounded from below by $f_{inf} \in \mathbb{R}$. Additionally, we set

$$M^2 := \sup_{\theta^* \in argmin\ \hat{f}} \mathbb{E}_\xi[\|\nabla G(\theta^*, \xi)\|^2].$$

Then, the iterates of SGD with momentum satisfy the following bound

$$\mathbb{E}[\hat{f}(\theta^k)] - f_{inf} \leq \frac{\|\theta^0 - \theta^*\|^2}{\eta(k+1)} + 2\eta M^2,$$

when

$$\alpha_k = \frac{2\eta}{k+1}, \quad \mu^k = \frac{k}{k+1} \quad \text{and } \eta \leq \frac{1}{4L_{mb}}.$$

This result is taken from [113, Corollary 25].

2.3.3. Sensitivity Analysis for Constrained Optimization Problems

The following subsection introduces sensitivity analysis for constrained optimization problems. This approach will be used as a basis for the sensitivity analysis developed for the SensLI method in Section 4.3.1. Let us consider the equality constrained optimization problem

$$\min_{\theta \in \Theta} f(\theta) \quad \text{subject to } c(\theta) = 0, \tag{2.33}$$

where $f : \Theta \longrightarrow \mathbb{R}$ is a differentiable objective function and $c : \Theta \longrightarrow \mathbb{R}^{n_{\text{new}}}$ is a differentiable constraint function containing n_{new} constraints.

The *Lagrange function* of the constrained problem (2.33) is given by

$$\mathcal{L}(\theta, \lambda) := f(\theta) + \lambda^{\top} c(\theta), \tag{2.34}$$

where $\lambda \in \mathbb{R}^{n_{\text{new}}}$ are the Lagrange multipliers.

The *Karush-Kuhn-Tucker (KKT) conditions* are given by the equations describing stationary points of the Lagrange function (2.34)

$$\frac{\partial \mathcal{L}(\theta, \lambda)}{\partial \theta} = 0,$$
$$\frac{\partial \mathcal{L}(\theta, \lambda)}{\partial \lambda} = 0,$$

which translate to

$$-\nabla_{\theta} f(\theta) = c'(\theta)^T \lambda, \tag{2.35}$$
$$c(\theta) = 0. \tag{2.36}$$

A point (θ^*, λ^*) satisfying the KKT conditions (2.35) and (2.36) is called a KKT point. A more detailed introduction to constrained optimization can e.g. be found in [98, Chapter 12].

Since the KKT conditions are necessary optimality conditions only under certain regularity conditions, we need to make some assumptions on the constraint function c. A common constraint qualification is the *Linear Independence Constraint Qualification (LICQ)*, which states that the gradients of the active constraints are linearly independent at the current feasible point θ. More formally, the LICQ states that the Jacobian of the constraint function c has full row rank at the current feasible point θ, i.e.

$$\nabla c_j(\theta) \text{ for } j = 1, \cdots, n_{\text{new}} \text{ are linearly independent.} \tag{2.37}$$

For more details see e.g. [98, Section 12].

The critical cone (coinciding with the linearizing cone) of the equality constraints at the KKT point θ is given by

$$T_c(\theta) = \{\delta \in \Theta \mid c_j'(\theta)\delta = 0 \quad \forall j = 1, \cdots, n_{\text{new}}\}. \tag{2.38}$$

To study the sensitivity of the solution of the optimization problem with respect to perturbations in the constraint function, we consider a family of perturbed optimization problems

$$\min_{\theta \in \Theta} f(\theta) \quad \text{subject to } c_{\epsilon}(\theta) = 0, \tag{2.39}$$

where $c_{\epsilon} : \Theta \longrightarrow \mathbb{R}^n_{\text{new}}$ is a family of differentiable constraint functions parametrized by $\epsilon \in \mathbb{R}^n_{\text{new}}$ via

$$c_{\epsilon}(\theta) = c(\theta) - \epsilon\Delta$$

for some $\Delta \in \mathbb{R}^n_{\text{new}}$.

The Lagrange function of the perturbed optimization problem is given by

$$\mathcal{L}(\theta, \lambda, \epsilon) \coloneqq f(\theta) + \lambda^\top c_\epsilon(\theta). \tag{2.40}$$

We study the sensitivity of the solution of the perturbed optimization problem with respect to the perturbation scale ϵ.

Theorem 2.33 (Behavior of Sensitivities w.r.t. Perturbations in the Constraints)**.** *Let (θ^*, λ^*) be a KKT point of the unperturbed optimization problem and let the LICQ* (2.37) *hold at θ^*. Assume that $\nabla^2_\theta \mathcal{L}(\theta^*, \lambda^*)$ is strictly positive definite on the critical cone $T_c(\theta^*)$* (2.38) *at point θ^*. Then, there exists a neighborhood U around $\epsilon = 0$ such that for each $\epsilon \in U$, there exists a unique KKT point $(\theta^*_\epsilon, \lambda^*_\epsilon)$ of the perturbed optimization problem* (2.39)*. Additionally, the KKT point $(\theta^*_\epsilon, \lambda^*_\epsilon)$ is a differentiable function of ϵ in U and satisfies*

$$\frac{\mathrm{d}}{\mathrm{d}\epsilon} f(\theta^*_\epsilon)\bigg|_{\epsilon=0} = -\lambda^{*\top}\Delta,$$

and hence

$$f(\theta^*_\epsilon) = f(\theta^*) - \epsilon\lambda^{*\top}\Delta + o(|\epsilon|).$$

This was shown in e.g. [32, Theorem 3.3.2] and [67, Theorem 2.24]. The Implicit Function Theorem (IFT) (see e.g. [33, Chapter 8]) is the essential tool in the proof. The theorem states that for a KKT point θ^* the sensitivity of the optimal objective function value with respect to perturbations in the constraint function is given by the Euclidean inner product of the Lagrange multipliers λ^* and the perturbation direction $\epsilon\Delta$.

2.4. The MG/Opt Method

The MG/Opt method is an optimization framework inspired by multigrid methods for Partial Differential Equations and is designed for solving deterministic optimization problems with a hierarchical structure. It can be applied to a wide range of optimization problems, among other things to neural network training. We introduce the MG/Opt method and its convergence properties as a basis for the stochastic MG/Opt methods developed in Chapter 3. The section is structured as follows. We start by giving a short introduction the multigrid method for PDEs in Section 2.4.1, which served as inspiration for the MG/Opt method. After that, we introduce the basic methodology of the MG/Opt method in Section 2.4.2 and present the general convergence theory of the MG/Opt framework for deterministic optimization problems.

2.4.1. Multigrid Methods for PDEs

Multigrid methods are a class of iterative methods used to solve linear and nonlinear systems of equations, especially those arising from discretizing PDEs on a family of grids with different resolutions. The main idea of multigrid methods is to solve the problem on multiple levels of discretization, where each level has a different grid resolution and error representation. A multigrid method consists of two main components: smoothing steps and the coarse grid corrections (cgcs). The smoothing steps are used to reduce the high-frequency errors in the solution, while the coarse grid corrections are used to reduce the low-frequency errors in the solution after the smoothing steps. While multigrid methods were originally developed for solving linear PDEs, they can also be applied to nonlinear PDEs as e.g. proposed in [13]. An example for a multigrid method is the V-cycle multigrid method, which is a common multigrid method for solving linear and nonlinear PDEs. For a more detailed tutorial on multigrid methods for PDEs, we refer the reader to [14] and [51]. The multigrid optimization method is developed by transferring the concepts of the V-cycle multigrid method for nonlinear PDEs as in [13] to stationary conditions of optimization problems. Naturally, the MG/Opt method was originally used to solve optimal control problems with PDEs, where the optimization problems are defined on different grids (fine and coarse) and the prolongation and restriction operators are defined as interpolation and restriction operators as usual for PDEs, respectively.

2.4.2. Multigrid Optimization

First, we introduce the general MG/Opt method algorithm described in [95]. After that, we present the general convergence theory of MG/Opt. For ease of notation, we restrict ourselves to two-level MG/Opt, where we only have a fine and a coarse optimization problem. To extend the theory to multiple levels, iterative arguments can be used.

The MG/Opt framework

To apply the MG/Opt method, we need hierarchical optimization problems

$$\min_{\theta\in\Theta} f(\theta) \qquad \text{and} \qquad \min_{\bar\theta\in\Theta_H} f_H(\bar\theta),$$

with differentiable objective functions $f\colon \Theta \to \mathbb{R}$, $f_H\colon \Theta_H \to \mathbb{R}$. Typically, the fine parameter space Θ is a higher-dimensional space than the coarse parameter space Θ_H or the fine optimization problem is more complex in some other form. We assume that both spaces are finite-dimensional inner product spaces with inner products $(\cdot,\cdot)_\Theta$ and $(\cdot,\cdot)_{\Theta_H}$, respectively. Further, we assume that the objective functions f and f_H are bounded from below by $f_{inf}, f_{H inf} \in \mathbb{R}$, respectively.

To relate both problems, we need the ability to prolongate information between the optimization problems by defining the linear *prolongation operator* $\mathcal{P}_p$ and the linear *restriction operator* $\mathcal{R}_p$

$$\mathcal{R}_p : \Theta \to \Theta_H, \tag{2.41}$$

$$\mathcal{P}_p : \Theta_H \to \Theta, \tag{2.42}$$

for parameters of the fine and coarse optimization problem, respectively. Additionally, we define a second linear restriction operator

$$\mathcal{R}_d : \Theta^* \to \Theta_H^*,$$

which restricts the residuals of the fine optimization problem to the coarse optimization problem, i.e. $\mathcal{R}_d f'(\theta) \in \Theta_H^*$. Since we work with gradients instead of derivatives in the following, we construct a restriction operator for the gradients $\mathcal{R}_g\colon \Theta \to \Theta_H$. It is defined by the restriction operator for derivatives $\mathcal{R}_d$ by the Riesz isomorphism as

$$\langle \mathcal{R}_d f'(\theta), \bar\theta\rangle = (\mathcal{R}_g \nabla f(\theta), \bar\theta) \quad \forall f'(\theta) \in \Theta^*, \bar\theta \in \Theta_H.$$

The method uses the coarse level optimization problem to compute an update direction for the fine-level optimization problem. The update direction is called the *coarse grid correction (cgc)*. It is expected that this update is not as exact as a fine-level update, but it can be much cheaper to compute.

The *cgc e* at a current point $\theta^k \in \Theta$ is constructed by iterations on the coarse grid correction optimization problem

$$\min_{\bar\theta\in\Theta_H} f_H(\bar\theta) + \bar v_k \bar\theta =: h_k(\bar\theta), \tag{2.43}$$

i.e. an additional linear term is added to the coarse optimization problem, given by

$$\bar v_k := \mathcal{R}_d(f'(\theta^k)) - f_H'(\bar\theta^k). \tag{2.44}$$

The cgc objective function h inherits its differentiability from f_H. The computation starts at the initial iterate $\bar\theta^k := \mathcal{R}_p(\theta^k)$ which is the restriction of the current fine parameter and performs parameter updates on it. Note that the cgc objective function h_k depends on the current iteration number k via the current fine iterate θ^k. When it is clear from the context, we drop the iteration index k in the notation of the cgc objective function.

The cgc is then given by the prolongation of the difference of end and initial coarse parameters

$$e \coloneqq \mathcal{P}_p(\bar{\theta}^*_{cgc} - \mathcal{R}_p(\theta^k)), \tag{2.45}$$

where $\bar{\theta}^*_{cgc}$ is a stationary point of the cgc optimization problem or an approximation of it. The cgc is then used to update the current fine iterate using a step size α^k_{cgc}.

Sometimes it is beneficial to add a regularization term to the cgc objective function (2.43) to ensure that the cgc does not deviate too much from the current fine parameter. This can be done by adding a quadratic regularization term to the cgc objective function, leading to

$$h_\lambda(\bar{\theta}) = f_H(\bar{\theta}) + \bar{v}_k\bar{\theta} + \frac{\lambda}{2}\|\bar{\theta} - \mathcal{R}_p(\theta^k)\|^2, \tag{2.46}$$

where $\lambda > 0$ is the regularization parameter.

The MG/Opt framework is given in Algorithm 2 and an illustration of the method is given in Figure 2.7. It is based on the method proposed in [95].

Algorithm 2 MG/Opt algorithm in general form

```
Require: Initial guess $\theta^0$, (iterative) optimizer for fine problem, cgc step size $\alpha_{cgc} > 0$, convergence
    criterion, criterion for cgc, restrictions $\mathcal{R}_p, \mathcal{R}_d$, prolongation $\mathcal{P}_p$, procedure to compute cgc
  while fine problem not converged do
    while stopping criterion for cgc not met or not converged do
      Iterate on the fine problem with fine-level optimizer until stopping criterion is met, current
      iterate $\theta^k$
    end while
    Compute cgc $e$ and update $\theta^{k+1} \coloneqq \theta^k + \alpha_{cgc}e$
  end while
```

The cgc is used to update the fine parameters, but not in each iteration, rather only after a certain number of direct iterations on the fine problem. It is common to implement a condition into the algorithm which decides whether the cgc should be computed or not. This condition can be based on the number of iterations, or on the convergence of the fine-level optimizer and the expected behavior of the current gradient of the fine objective function restricted to the coarse space.

2.4.3. Convergence Analysis of MG/Opt

In this section, we present general convergence results for the MG/Opt method. We assume for the following analysis that the fine parameter space Θ and the coarse parameter space Θ_H are finite-dimensional real vector spaces of the form $\Theta = \mathbb{R}^{n_\Theta}$ and $\Theta_H = \mathbb{R}^{n_{\Theta_H}}$ with $n_{\Theta_H}, n_\Theta \in \mathbb{N}$ and $n_{\Theta_H} \leq n_\Theta$. The convergence results presented here give a first impression of the techniques available for analyzing

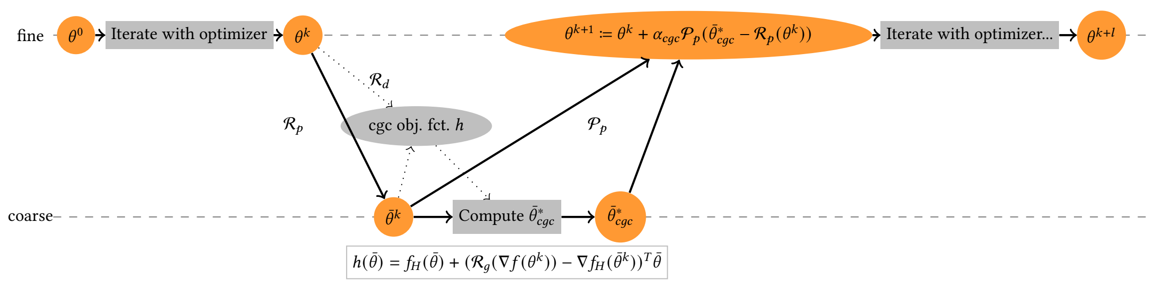

fine
θ^0
Iterate with optimizer
θ^k
$\theta^{k+1} := \theta^k + \alpha_{cgc}\mathcal{P}_p(\bar{\theta}^*_{cgc} - \mathcal{R}_p(\theta^k))$
Iterate with optimizer...
θ^{k+l}
$\mathcal{R}_d$
$\mathcal{R}_p$
cgc obj. fct. h
$\mathcal{P}_p$
coarse
$\bar{\theta}^k$
Compute $\bar{\theta}^*_{cgc}$
$\bar{\theta}^*_{cgc}$
$h(\bar{\theta}) = f_H(\bar{\theta}) + (\mathcal{R}_g(\nabla f(\theta^k)) - \nabla f_H(\bar{\theta}^k))^T\bar{\theta}$

Figure 2.7.: One loop of the MG/Opt algorithm.

the MG/Opt method. They serve as a foundation for the stochastic MG/Opt methods developed in Chapter 3. In the following, we assume that the parameter spaces satisfy $\Theta = \mathbb{R}^{n_\Theta}$ and $\Theta_H = \mathbb{R}^{n_{\Theta_H}}$ with $n_{\Theta_H}, n_\Theta \in \mathbb{N}$ and $n_{\Theta_H} \leq n_\Theta$. First it is necessary to investigate under which conditions the coarse grid correction (cgc) is a descent direction of the fine objective function at the current fine iterate. Then it is possible to derive convergence results for the MG/Opt method.

First-order coherence of the coarse grid correction The cgc is defined in a way such that the first iteration with a gradient update on the cgc problem is a descent direction of the fine objective function at the current fine iterate. This property is called *first-order coherence* and ensures that the iterations on the cgc problem start by roughly following the restricted fine gradient.

Lemma 2.34 (First-Order Coherence of Coarse Grid Correction). *If only one gradient descent step (with step size $\bar{\alpha}$) on the cgc problem is performed, then the coarse grid correction e satisfies*

$$e = -\bar{\alpha}\mathcal{P}_p(\mathcal{R}_g(\nabla f(\theta^k))).$$

Proof. Because only one gradient descent step on the cgc problem is performed, we have

$$\bar{\theta}^*_{cgc} = \bar{\theta}^k - \bar{\alpha}\nabla h(\bar{\theta}^k),$$

and hence

$$e = \mathcal{P}_p(\bar{\theta}^*_{cgc} - \bar{\theta}^k) = -\bar{\alpha}\mathcal{P}_p(\mathcal{R}_g(\nabla f(\theta^k))),$$

since the prolongation is linear and $\nabla h(\bar{\theta}^k) = \nabla f_H(\bar{\theta}^k) + \mathcal{R}_g(\nabla f(\theta^k)) - \nabla f_H(\bar{\theta}^k) = \mathcal{R}_g(\nabla f(\theta^k))$. □

Remark 2.35 (Number of Steps on the Coarse Grid Correction Problem). *Note that it is computationally inefficient to only perform one gradient descent step on the cgc, as the cgc problem is more expensive to construct than the with one iteration generated cgc update e, as it uses the evaluation of the fine gradient at the current iterate. Generally, there is no knowledge on how many steps on the cgc problem are the most effective to the best of my knowledge.*

Descent Direction Property of the Coarse Grid Correction Generally, the cgc update is not necessarily a descent direction of the fine objective function at the current fine iterate. However, there are some cases, where it can be shown that the cgc is a descent direction of the fine objective at the current fine iterate θ^k. In the following we list some of these cases.

Theorem 2.36 (Coarse Grid Correction as Descent Direction for Approximate Stationary Point $\bar{\theta}^*_{cgc}$). *Let f_H be μ_H-strongly convex and twice differentiable. Additionally, let $\mathcal{R}_g = \mathcal{P}_p^T$. If the cgc is computed approximately, i.e. $\nabla h(\bar{\theta}^*_{cgc}) = r$, and $\mathcal{R}_g(\nabla f(\theta^k)) \neq 0$, then the cgc $e = \mathcal{P}_p(\bar{e})$ with $\bar{e} := \bar{\theta}^*_{cgc} - \bar{\theta}^k$ satisfies*

$$\nabla f(\theta^k)^T e \leq -(\mu_H + \lambda)\|\bar{e}\|^2 + r^\top \bar{e},$$

as long as $\bar{\theta}^*_{cgc} \neq \bar{\theta}^k$. Then, the following scenarios guarantee that e is a descent direction of the fine objective f at θ^k:

- *If the cgc is computed exactly, i.e. $r = 0$.*
- *If f_H is not strongly convex, but a large enough regularization term with $\lambda > 0$ is added to the cgc objective as in (2.46) and $r = 0$.*
- *If the cgc is computed approximately, but the residual r is sufficiently small, i.e. $r^\top \bar{e} < (\mu_H + \lambda)\|\bar{e}\|^2$.*

The proof can be found in [75] and for accessibility in Appendix B.1.

Remark 2.37 (Restriction Operator for Parameters). *There are no assumptions on the restriction operator $\mathcal{R}_p$ for the parameters in Theorem 2.36. For the theory it does not need to have any relation to $\mathcal{R}_d$ or $\mathcal{P}_p$. However, it seems natural to choose $\mathcal{R}_p$ such that $\mathcal{R}_p\mathcal{P}_p = \mathbb{1}$. If we have a prolongation operator $\mathcal{P}_p$ with full rank, choosing $\mathcal{R}_p$ as the pseudo-inverse $(\mathcal{P}_p^T\mathcal{P}_p)^{-1}\mathcal{P}_p^T$ of $\mathcal{P}_p$ satisfies $\mathcal{R}_p\mathcal{P}_p = \mathbb{1}$.*

Theorem 2.38 (Approximate Coarse Grid Correction as Descent Direction for Strongly Convex Coarse Objective). *Let f_H be μ_H-strongly convex and twice differentiable. Additionally, let $\mathcal{R}_g = \mathcal{P}_p^T$. If the cgc is computed such that*

$$h(\bar{\theta}^*_{cgc}) \leq h(\bar{\theta}^k),$$

and $\mathcal{R}_g(\nabla f(\theta^k)) \neq 0$, then the cgc $e = \mathcal{P}_p(\bar{e})$ is a descent direction of the fine objective f at θ^k, i.e.

$$\nabla f(\theta^k)^T e \leq -t\mu_H\|\bar{e}\|^2,$$

for some $0 < t \leq 1$.

The proof can be found in [75, Theorem 5] and for accessibility in Appendix B.1.

For a non-convex coarse objective function we can not guarantee that the cgc is a descent direction of the fine objective even if the cgc is computed with a stationary point $\bar{\theta}^*_{cgc}$. Adding a regularization term helps to mitigate the problem. The results presented here serve as a basis for developing the stochastic MG/Opt methods presented in Chapter 3.

Convergence of MG/Opt Method Now that we have established criteria for when e is a descent direction of f at θ^k, we can show convergence results for the MG/Opt method.

Theorem 2.39 (General Global Convergence of MG/Opt). *Assume that e is always a descent direction of f at θ^k (either by being in a suitable problem setting or by only accepting the cgc update if it indeed is a descent direction). Further, assume that α_{cgc} is always chosen such that the cgc update $\theta^{k+1} = \theta^k + \alpha_{cgc}e$ yields a point with $f(\theta^{k+1}) \leq f(\theta^k)$. If the optimization algorithm for the fine problem converges globally and takes at least one step between consecutive coarse grid updates, then the MG/Opt algorithm converges globally in the same sense.*

The theorem is given in [95, Theorem 1].

Theorem 2.40 (Convergence of MG/Opt with Angle Condition). *Let f be L-smooth. Assume that e is always a descent direction of f at θ^k either by assuming the setting of Theorems 2.36 and 2.38 or by only accepting the cgc update if it indeed is a descent direction. Further, assume that the angle condition*

$$\cos(\angle(e, \nabla f(\theta^k))) \geq \eta$$

holds for some $\eta > 0$. If you use an optimization algorithm for the classical iterations on the fine optimization problem that uses a descent direction of the fine objective and the Wolfe line search conditions (see e.g. [98, Chapter 3.1]) to determine the step sizes for every iteration, then

$$\lim_{k\to\infty} \|\nabla f(\theta^k)\|^2 = 0.$$

From [98, Theorem 3.2 and following text].

Remark 2.41 (Number of Iterations on the Coarse Grid Correction Problem). *Note that in Theorems 2.39 and 2.40 there are no assumptions on how many iterations are performed on the cgc problem to compute e. It is only necessary that at least one iteration is performed between two consecutive coarse grid corrections. However, the number of iterations on the cgc problem influences the quality and the computational effort of the cgc and hence the convergence speed of the MG/Opt method in practice.*

Extensions of the MG/Opt Method There exists multiple works that extend the MG/Opt method to more complex settings. For example, [96] extends the method to optimization problems with equality constraints. [9] apply the MG/Opt method to optimization problems with PDE constraints. There exists also work from [131] constructing a novel backtracking line search strategy tailored to MG/Opt to ensure that the coarse grid corrections are always descent directions. Also, [8] proposes a step size strategy for strict convex problems ensuring global convergence of the method. [102] extends the method to non-smooth convex optimization problems. There exists also variants like [94] which extend the theory to optimization with manifolds. It is also possible to define multiplicative cgc update models instead of the additive one used in MG/Opt and hybrids of these, as e.g. done in [72].

2.5. Preconditioning for Matrices and Linear Maps

Preconditioning is a technique used to improve the convergence of iterative methods for solving linear systems of equations or for optimization problems. For $\Theta = \mathbb{R}^{n_\Theta}$, this is typically done by selecting a suitable (non-Euclidean) inner product on the space of optimization variables which leads to a preconditioned gradient. In the following we give a short introduction to preconditioning for matrices and linear maps in real Hilbert spaces, which is the basis for the preconditioning method presented in Chapter 5.

The section is structured as follows. After introducing basic notation and definitions for Hilbert spaces, we present Frobenius-type inner products on the space of linear maps between two Hilbert spaces, which were developed in [58]. The gradient of an object w.r.t. this inner product leads to a preconditioned gradient. Next, we give an introduction to the Fisher Information Matrix (FIM) and Natural Gradient Descent (NGD) method which employs a specific preconditioning technique and can be applied to parameter estimation problems. Finally, we introduce the popular Kronecker-Factored Approximate Curvature (K-FAC) preconditioning method which is based on the NGD method and is tailored to neural network training. We introduce it here as we compare it to the covariance-driven preconditioners presented in Chapter 5.

Let us assume that we have two real Hilbert spaces $\mathcal{H}_1$ and $\mathcal{H}_2$ with inner products $(\cdot,\cdot)_1$ and $(\cdot,\cdot)_2$, respectively. The norm on $\mathcal{H}_i$ induced by the inner product is given by

$$\|x\|_i \coloneqq \sqrt{(x,x)_i}$$

for $i = 1, 2$ and $x \in \mathcal{H}_i$.

The (topological) dual space $\mathcal{H}_i^*$ of $\mathcal{H}_i$ is the space of bounded linear functionals on $\mathcal{H}_i$, i.e. linear maps $\ell\colon \mathcal{H}_i \to \mathbb{R}$ that are continuous with respect to the norm on $\mathcal{H}_i$.

The Riesz representation theorem (see e.g. [4, Theorem 6.42]) states that for every continuous linear functional $\ell \in \mathcal{H}_i^*$, there exists a unique element $x_\ell \in \mathcal{H}_i$ such that

$$\ell(y) = (x_\ell, y)_i \quad \forall y \in \mathcal{H}_i.$$

This element x_ℓ is called the Riesz representer of ℓ. The Riesz map $\mathcal{R}_i\colon \mathcal{H}_i \to \mathcal{H}_i^*$ is defined by

$$\mathcal{R}_i(x) = \ell_x,$$

where $\ell_x \in \mathcal{H}_i^*$ is the continuous linear functional defined by

$$\ell_x(y) = (x, y)_i \quad \forall y \in \mathcal{H}_i.$$

The Riesz map is an isometric isomorphism between $\mathcal{H}_i$ and $\mathcal{H}_i^*$, i.e.

$$\|\ell\|_{\mathcal{H}_i^*} = \|x_\ell\|_i \quad \forall \ell \in \mathcal{H}_i^*.$$

The inverse of the Riesz map is called the dual Riesz map $\mathcal{R}_i^{-1}\colon \mathcal{H}_i^* \to \mathcal{H}_i$ and is defined by

$$\mathcal{R}_i^{-1}(\ell) = x_\ell. \tag{2.47}$$

The inner product on the dual space $\mathcal{H}_i^*$ is defined by

$$(\ell_1, \ell_2)_{\mathcal{H}_i^*} := (\mathcal{R}_i^{-1}(\ell_1), \mathcal{R}_i^{-1}(\ell_2))_i \quad \forall \ell_1, \ell_2 \in \mathcal{H}_i^*.$$

A more detailed introduction to Hilbert spaces and the Riesz representation theorem can be found in many textbooks, e.g. [4].

Definition 2.42 (Hilbert Space Adjoint). *Let $S\colon \mathcal{H}_1 \to \mathcal{H}_2$ be a continuous linear map between the two Hilbert spaces. The Hilbert space adjoint $S^*\colon \mathcal{H}_2 \to \mathcal{H}_1$ of S is defined by*

$$(Sx, y)_2 = (x, S^*y)_1 \quad \forall x \in \mathcal{H}_1, y \in \mathcal{H}_2.$$

Definition 2.43 (Dual Map). *Let $S\colon \mathcal{H}_1 \to \mathcal{H}_2$ be a continuous linear map between the two Hilbert spaces. The dual map $S'\colon \mathcal{H}_2^* \to \mathcal{H}_1^*$ of S is defined by*

$$\langle S'\ell, x\rangle_1 = \langle \ell, Sx\rangle_2 \quad \forall x \in \mathcal{H}_1, \ell \in \mathcal{H}_2^*.$$

In Hilbert spaces, the relation between the dual map and the Hilbert space adjoint is given by

$$S' = \mathcal{R}_1 S^* \mathcal{R}_2^{-1}. \tag{2.48}$$

2.5.1. Frobenius-Type Inner Products

The following subsection on Frobenius-type inner products is based on [58].

Let

$$S_i\colon \mathcal{H}_1 \to \mathcal{H}_2 \quad \text{for} \quad i = 1, 2$$

be continuous linear maps between the two real Hilbert spaces. The space of continuous linear maps from $\mathcal{H}_1$ to $\mathcal{H}_2$ is denoted by $\mathcal{L}(\mathcal{H}_1, \mathcal{H}_2)$.

The generalized Frobenius-type inner product is defined by

$$(S_1, S_2)_{\mathcal{H}_1 \to \mathcal{H}_2} := \operatorname{trace}(\mathcal{R}_{\mathcal{H}_1}^{-1} S_1' \mathcal{R}_{\mathcal{H}_2} S_2),$$

where S_i' is the Banach space adjoint of S_i with respect to the inner products on $\mathcal{H}_1$ and $\mathcal{H}_2$ and the trace operator is defined on $\mathcal{H}_1$. The maps $\mathcal{R}_{\mathcal{H}_1}\colon \mathcal{H}_1 \to \mathcal{H}_1^*$ and $\mathcal{R}_{\mathcal{H}_2}\colon \mathcal{H}_2 \to \mathcal{H}_2^*$ are the Riesz maps for the spaces $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively. In order to ensure that the trace is well-defined, we assume from now on that the operators $\mathcal{R}_{\mathcal{H}_1}^{-1} S_1' \mathcal{R}_{\mathcal{H}_2} S_2$ are trace-class, see e.g. [18] for more details.

The space $\mathcal{L}(\mathcal{H}_1, \mathcal{H}_2)$ is a Hilbert space as well. The dual space of $\mathcal{L}(\mathcal{H}_1, \mathcal{H}_2)$ is denoted by $\mathcal{L}(\mathcal{H}_1, \mathcal{H}_2)^*$

and consists of continuous linear functionals on $\mathcal{L}(\mathcal{H}_1, \mathcal{H}_2)$. The Riesz map on the space of linear maps w.r.t. the Frobenius-type inner product is given by

$$\begin{aligned}\mathcal{R}_{\mathcal{H}_1\to\mathcal{H}_2}\colon \mathcal{L}(\mathcal{H}_1, \mathcal{H}_2) &\to \mathcal{L}(\mathcal{H}_1, \mathcal{H}_2)^* \\ S &\mapsto \Big(T \mapsto \operatorname{trace}(\mathcal{R}_{\mathcal{H}_1}^{-1} S' \mathcal{R}_{\mathcal{H}_2} T)\Big).\end{aligned}$$

If the Hilbert spaces $\mathcal{H}_1$ and $\mathcal{H}_2$ are finite-dimensional and equipped with orthonormal bases, the inner products can be represented by matrices $\mathbb{H}_1$ and $\mathbb{H}_2$, respectively. A linear map $S\colon \mathcal{H}_1 \to \mathcal{H}_2$ can be represented by a matrix M such that

$$S(x) = Mx \quad \forall x \in \mathcal{H}_1.$$

Keep in mind that the representation of S by M and $\mathbb{H}_1$, $\mathbb{H}_2$ depend on the choice of the orthonormal bases of $\mathcal{H}_1$ and $\mathcal{H}_2$. The Frobenius-type inner product can then be expressed as

$$(M_1, M_2)_{\mathcal{H}_1\to\mathcal{H}_2} = \operatorname{trace}(\mathbb{H}_1^{-1} M_1^T \mathbb{H}_2 M_2),$$

where trace is the trace operator on the space of matrices.

2.5.2. Natural Gradient Descent and the Fisher Information Matrix

In this section, x is the random variable with values in the input space X_0 and y is the random variable with values in the label space Y_L, which represent the input features and label features, respectively. Accordingly, their distributions represent the data distribution from which the training data is sampled. Again, $\theta \in \Theta = \mathbb{R}^{n_\Theta}$ are the parameters of the prediction model $g\colon \Theta \times X_0 \to Y_L$. We assume a probabilistic model for the conditional distribution of y of the form

$$p(y|x, \theta) = p(y|g(\theta, x)),$$

where $p(y|\cdot)$ is an exponential family (see e.g. [128, Chapter 9.13.3]) with natural parameters in Y and $g : X \times \Theta \to Y$ is a prediction function parametrized by $\theta \in \Theta$. Given D i.i.d. training samples $(x_{0,j}, y_{\text{label}j})_{j=1}^D$, we want to minimize the loss function

$$\hat{f}(\theta) \coloneqq \sum_{j=1}^{D} \mathcal{L}(g(x_{0,j}, \theta), y_{\text{label}j}) = -\sum_{j=1}^{D} \log p(y_{\text{label}j} | g(x_{0,j}, \theta)).$$

Definition 2.44 (Score Function). *The score function is defined as the Euclidean gradient or the vector of partial derivatives of the log-likelihood function w.r.t. the parameters θ*

$$s(\theta) \coloneqq (\frac{\partial}{\partial\theta} \log p(y|g(\theta, x)))^T = \nabla_\theta \log p(y|g(\theta, x)). \tag{2.49}$$

Lemma 2.45. *Assume regularity conditions on the density $p(y|g(\theta, x))$ which ensure that differentiation* *and integration can be interchanged. Then it holds that* $\mathbb{E}[s(\theta)] = 0$.

Definition 2.46 (Fisher Information Matrix (FIM))**.** *The FIM is defined as the expected outer product of the score function*

$$F(\theta) := \mathbb{E}\left[s(\theta)s(\theta)^T\right] = \mathbb{E}\left[\nabla_\theta \log p(y|g(\theta,x))\nabla_\theta \log p(y|x,\theta)^T\right]. \tag{2.50}$$

For more details on the score and the FIM we refer to [40].

Assuming the regularity conditions from Lemma 2.45, the FIM is the covariance matrix of the score function, since $\mathbb{E}[s(\theta)] = 0$.

Natural Gradient Descent [1] is an optimization method that uses the FIM to compute the natural gradient.

Definition 2.47 (Natural Gradient Descent)**.** *The natural gradient is defined as*

$$\nabla_\theta^{nat}\hat{f}(\theta) = F(\theta)^{-1}\nabla_\theta^{eucl}\hat{f}(\theta).$$

The NGD update is then given by

$$\theta^{k+1} = \theta^k - \alpha_k\nabla_\theta^{nat}\hat{f}(\theta^k) = \theta^k - \alpha_k F(\theta^k)^{-1}\nabla_\theta^{eucl}\hat{f}(\theta^k),$$

where $\alpha_k > 0$ *is the learning rate (lr).*

Provided some specific loss functions, the FIM captures partial second-order information about the underlying statistical model, and it coincides with the Generalized Gauss-Newton (GGN) matrix (see e.g. [79, Chapter 3]). The most prominent loss functions with this property are the Mean Squared Error (MSE)-loss for regression tasks and the Cross-Entropy (CE)-loss for classification tasks, see Section 2.5.3 for details.

Preconditioning with the FIM changes the information geometry of the problem, because the distance between two points in parameter space is not measured in the Euclidean sense, but in the sense of the Kullback-Leibler divergence [78] of the probability distributions $p(y|g(\theta,x))$. For more details on this relation we refer readers to information geometry literature such as [1].

Applying NGD in machine learning was first proposed by [1]. In the context of neural networks, [112] showed relations between the FIM and the GGN. Among others, [82] and [125], derived optimization methods tailored to neural networks based on the GGN.

2.5.3. Statistical Approximations of the Fisher

In this subsection we start by giving examples for loss functions that lead to a probabilistic model for the data and clarify some notation on the FIM which was highlighted [79].

We show that the loss functions MSE-loss and CE-loss are structured such that there is a distribution $p(y|g(\theta,x))$ satisfying

$$\mathcal{L}(g(\theta,x),y) = -\log p(y|g(\theta,x)).$$

The MSE-loss (2.12) for regression tasks leads to a distribution of the form

$$\begin{aligned} &\log p(y|g(\theta,x)) = -\frac{1}{2}\|y-g(\theta,x)\|^2 \\ \rightarrow \quad &p(y|g(\theta,x)) = \exp\left(-\frac{1}{2}\|y-g(\theta,x)\|^2\right), \end{aligned}$$

which corresponds to a normal distribution with mean $g(x,\theta)$ and variance 1. For more details on the distribution we refer the reader to [71, p.45].

The CE-loss (2.13) for classification tasks leads to a distribution of the form

$$\begin{aligned} &\log p(y=c|g(\theta,x)) = -\log(\text{softmax}(g(\theta,x))_c) \\ \rightarrow \quad &p(y=c|g(\theta,x)) = \text{softmax}(g(\theta,x))_c = \frac{\exp(g(\theta,x)_c)}{\sum_{c'=1}^{C}\exp(g(\theta,x)_{c'})}. \end{aligned}$$

For these loss functions, the FIM coincides with the Generalized Gauss-Newton (GGN) approximation of the Hessian, see e.g. [112].

Generally (and as defined above), the FIM is defined as

$$F(\theta) = \mathbb{E}_{x,y\sim p(x,y|\theta)}\left[\nabla_\theta \log p(y|g(\theta,x))\nabla_\theta \log p(y|g(\theta,x))^T\right],$$

where $p(y|g(\theta,x))$ is the probability density function of the data y given the input x and the parameters θ. Keep in mind that $p(x,y \mid \theta) = p(x)p(y|g(\theta,x))$. This is what is called the *exact Fisher Information Matrix* in statistics.

Often, the distribution of the inputs $p(x)$ is not known, hence we approximate it empirically by sampling from the data distribution of x, i.e. the training data $\{x_{0,j}\}_{j=1}^{D}$. Then the FIM is defined as

$$F(\theta) = \frac{1}{D}\sum_{j=1}^{D}\mathbb{E}_{y\sim p(y|g(x_{0,j},\theta))}\left[\nabla_\theta \log p(y|g(x_{0,j},\theta))\nabla_\theta \log p(y|g(x_{0,j},\theta))^T\right].$$

This version is called the exact FIM in machine learning literature, but the empirical FIM in statistics literature.

The *empirical Fisher Information Matrix* in machine learning is defined as

$$F(\theta) = \frac{1}{D}\sum_{j=1}^{D}\nabla_\theta \log p(y_{\text{label}j}|g(x_{0,j},\theta)s)\nabla_\theta \log p(y_{\text{label}j}|g(x_{0,j},\theta))^T,$$

where D is the number of samples. The notation "empirical" is used here since for both in- and output the available (training) data is used. However, this can not be interpreted as a FIM directly, since the distribution of the output comes from $p(y \mid x_{0,j})$ instead of $p(y \mid x_{0,j}, \theta)$, which does not depend on the model parameters θ. In [79] the authors show severe limitations of the empirical (ML notation) FIM to advance training of neural networks, because it is not able to capture second-order information.

We consider the machine learning terminology (which is also used describing K-FAC), in denoting exact and empirical Fisher Information Matrices in the following.

Finally, we describe how to sample from the distribution $p(y \mid g(x_{0,j}, \theta))$ for the exact FIM for the two most prominent loss functions in machine learning.

Example 2.48 (1D MSE-loss). *Sample from the normal distribution $\mathcal{N}(g(x_{0,j}, \theta), 1)$ with mean $g(x_{0,j}, \theta)$ and variance 1 to get a sample for y_j. Then compute the gradient $\nabla_\theta \log p(y_j|g(x_{0,j}, \theta))$ for the sampled y_j.*

Example 2.49 (CE-loss). *Sample from the categorical distribution with probabilities given by the softmax of the model output $g(x_{0,j}, \theta)$ to get a sample for y_j. Then compute the gradient $\nabla_\theta \log p(y_j|g(x_{0,j}, \theta))$ for the sampled y_j.*

2.5.4. Kronecker-Factored Approximate Curvature (K-FAC)

Next we describe the derivation of the most popular approximation of the FIM tailored to neural networks, the K-FAC method, which was proposed in [91]. For deep learning problems, the FIM is often too large to be computed and stored. Hence, the K-FAC preconditioner makes approximations to the FIM tailored to the structure of neural networks.

In the following, we neglect biases and consider only the weight matrices W_i of the layers for FNNs. The vectorized gradient of the loss function w.r.t. all trainable parameters (i.e. all weight matrices) w.r.t. one training datum has the form

$$\nabla_\theta f_j(\theta) = \nabla_\theta f_j(\theta) = [d_1^T, d_2^T, \dots, d_L^T]^T,$$

where d_i is the vectorized Euclidean gradient of the loss function w.r.t. the weight matrix W_i of the i-th layer, i.e.

$$d_i = \mathrm{vec}(\nabla_{y_i} f_j(\theta) x_{i-1}^T) = x_{i-1}^T \otimes \nabla_{y_i} f_j(\theta).$$

Here, $\nabla_{y_i} f_j(\theta)$ is the Euclidean gradient of the loss function w.r.t. the pre-activation layer y_i and $\otimes$ is the Kronecker product.

The blocks corresponding to layers of the FIM are given by

$$F_{ik} = \mathbb{E}\left[d_i d_k^T\right] = \mathbb{E}\left[x_{i-1} x_{k-1}^T \otimes \nabla_{y_i} f_j(\theta) \nabla_{y_k} f_j(\theta)^T\right].$$

Approximation 1: We assume that the input and output of each layer are **independent**.

$$F_{ik} \approx \mathbb{E}[x_{i-1}x_{k-1}^T] \otimes \mathbb{E}[\nabla_{y_i} f_j(\theta)\nabla_{y_k} f_j(\theta)^T] =: X_{i-1,k-1} \otimes G_{i,k} =: \tilde{F}_{ik}.$$

This assumption is made out of necessity, since otherwise the FIM blocks F_{ik} are not efficiently computable in practice. The quantities are not independent in practice, however.

Approximation 2: We consider only the blocks on the **diagonal**, i.e. the interactions in between a layer and not between different or neighboring layers.

$$\tilde{F} := \operatorname{diag}(X_{0,0} \otimes G_{1,1},\ X_{1,1} \otimes G_{2,2}, \dots,\ X_{L-1,L-1} \otimes G_{L,L}).$$

Then, the inverse of the approximated FIM is given by

$$\tilde{F}^{-1} := \operatorname{diag}(X_{0,0}^{-1} \otimes G_{1,1}^{-1},\ X_{1,1}^{-1} \otimes G_{2,2}^{-1}, \dots,\ X_{L-1,L-1}^{-1} \otimes G_{L,L}^{-1}).$$

Now, $\tilde{F}^{-1}\nabla_\theta f(\theta)$ has L layer components

$$X_{i-1,i-1}^{-1} \otimes G_{i,i}^{-1} d_i = X_{i-1,i-1}^{-1} \otimes G_{i,i}^{-1} \operatorname{vec}(\nabla_{y_i} f_j(\theta) x_{i-1}^T)$$

with

$$X_{i-1,i-1}^{-1} \otimes G_{i,i}^{-1} d_i = \operatorname{vec}(G_{i,i}^{-1} \nabla_{y_i} f_j(\theta) x_{i-1}^T X_{i-1,i-1}^{-1}).$$

The K-FAC weight gradient for one training datum update is given by

$$\nabla_{W_i}^{\text{K-FAC}} f_j(\theta) = G_{i,i}^{-1} \nabla_{y_i} f_j(\theta) x_{i-1}^T X_{i-1,i-1}^{-1}.$$

The K-FAC weight gradient update for a mini-batch of size B is given by

$$\nabla_{W_i}^{\text{K-FAC}} \hat{f}(\theta) = G_{i,i}^{-1} \frac{1}{B} \sum_{j=1}^{B} \nabla_{y_i} f_j(\theta) x_{i-1}(j)^T X_{i-1,i-1}^{-1}, \tag{2.51}$$

where $x_{i-1}(j)$ is the post-activation features of the $i-1$-th layer for the j-th training datum.

The expectations $X_{i-1,i-1}$ and $G_{i,i}$ are approximated by the empirical means over a batch of N samples, i.e.

$$X_{i-1,i-1} \approx \frac{1}{N} \sum_{j=1}^{N} x_{i-1}(j) x_{i-1}(j)^T,$$

and get updated over the SGD iterations by exponential averaging as an example. For more details see Section 2.5.3.

Generally, the derivation (2.51) is already presented in [59]. The novelty in [91] are the technical additions to make the K-FAC preconditioner more efficient in training. We report some of them here.

- **Update frequency:** Recompute FIM approximation (and inversion) every 20 iterations via exponential averaging.
- **Damping strategy:** Use variable damping terms

$$G_{i,i} \leftarrow G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda + \eta})I \text{ and } X_{i-1,i-1} \leftarrow X_{i-1,i-1} + +\pi_i(\sqrt{\lambda + \eta}).$$

  λ is adapted in a Levenberg-Marquardt style (see e.g. [98, Chapter 10]), η is a regularization factor, and $\pi_i = \sqrt{\frac{\|X_{i-1,i-1} \otimes I\|}{\|I \otimes G_{i,i}\|}}$.
- **Rescaling:** Use the learning rate

$$\alpha^* = -\frac{\nabla_\theta \mathcal{L}(\theta)\delta}{\delta^T(F + (\lambda + \eta)I)\delta} \text{ with } \delta = \tilde{F}^{-1}\nabla_\theta \mathcal{L}(\theta).$$

- **Adding momentum:** Use an update with momentum (instead of δ) of the form

$$d = \alpha^* \delta + \mu d_0,$$

  where μ is the momentum factor and d is the update direction and d_0 is the update from the last iteration.

The K-FAC preconditioner became popular in [91]. The basic idea of its layer-wise structure approximation, was first used in [59] and also in [83, 90, 99, 106] before the K-FAC preconditioner was introduced.

There exist several works that use similar approximations of the FIM or GGN matrix for neural networks. The authors of [59] propose a block-diagonal approximation w.r.t. layers of the FIM for neural networks with a predefined fixed damping term which is added to ensure invertibility. [83] uses a block-diagonal approximation of the GGN matrix for neural networks as well, but with blocks corresponding to neurons instead of layers. Hessian-Free Optimization (HFO) for deep learning is proposed in [90], which is also based on the GGN matrix and uses CG iterations to avoid inversion without reusing info from the last iterations in the updating of the curvature. The work [99] introduces the Riemannian Natural Gradient Descent method for neural networks, which is based on the FIM and uses a block-diagonal approximation w.r.t. layers as well. It uses no damping since this would destroy the invariance properties of the algorithm. [106] refines the ideas above by developing an efficient online method to update the curvature information. There also exist some technical improvements to the K-FAC preconditioner, e.g. [5, 101, 22].

3. Multigrid Optimization (MG/Opt) for Neural Network Training

In this chapter we propose new stochastic variants of the MG/Opt method [95] for two levels accompanied by stochastic convergence results. By using stochastic gradient estimates instead of exact gradients, the method can be more easily applied to stochastic optimization problems compared to its deterministic version. First we develop a stochastic MG/Opt method in the strongly convex quadratic setting and identify restriction and prolongation operators suited to SGD iterations, where we can prove convergence of iterates in expectation under strong assumptions. Then we propose a second version tailored to the more general, non-convex setting, which is more relevant for neural network training. Naturally, this version leads to weaker convergence results. Nevertheless, we are able to show new stochastic convergence results in the non-convex setting for fixed and diminishing step sizes. Because of the use of stochastic gradient estimates, e.g. mini-batch gradients, the method can be applied to the training of neural networks. Further, we discuss potential applications and recent similar approaches to neural network training and limitations of the proposed method.

3.1. Introduction

Multigrid methods have long been recognized as powerful and efficient techniques for solving Partial Differential Equations, particularly in the context of large-scale scientific computing. Their ability to accelerate convergence by leveraging hierarchical problem structures has inspired researchers to explore their applicability beyond classical PDEs, including general optimization problems. The MG/Opt method, originally proposed by Nash [95], extends the multigrid V-cycle, well-known from nonlinear PDE solvers such as Full Approximation Storage (FAS), to general non-convex optimization problems that possess a hierarchical structure. For an introduction to MG/Opt and its convergence theory in the deterministic setting, we refer to Section 2.4.

With the rapid development of deep learning, the question naturally arises whether MG/Opt can be adapted for the training of neural networks, whose training often involves large-scale, non-convex optimization. Neural network architectures inherently exhibit hierarchical structures, e.g. by their depth or varying resolutions in Convolutional Neural Networks. However, neural network training typically relies on stochastic gradient estimates, such as those obtained from mini-batch Stochastic Gradient Descent (SGD), rather than exact gradients. This necessitates modifications to classical MG/Opt variants to ensure their theoretical soundness in the stochastic setting. An introduction to SGD and its convergence properties is provided in Section 2.3.2.

The chapter investigates the possibilities and limitations of applying MG/Opt in the context of stochastic optimization. We develop new stochastic variants of MG/Opt for two levels tailored to SGD, starting with the strongly convex quadratic setting and identifying suitable restriction and prolongation operators for this setting. Subsequently, we extend the method and its analysis to stochastic non-convex problems. In both settings, the methods are accompanied by a theoretical analysis, leading to new stochastic convergence results. Especially in the non-convex setting, we are able to show the first almost sure convergence result for MG/Opt with stochastic gradient estimates.

The chapter is structured as follows. Section 3.2 surveys existing work on multigrid optimization for stochastic settings such as neural networks, highlighting developments and theoretical results, including multilevel trust-region methods and already existing stochastic coarse-level correction schemes. Section 3.3 develops stochastic versions of MG/Opt for two-level problems, starting with the strongly convex quadratic setting where typical convergence can be shown under strong assumptions. Subsequently, we propose a stochastic MG/Opt variant for non-convex problems for SGD gradients estimates and analyze the variant theoretically, leading to new stochastic convergence results. This variant is tailored to parameter estimation problems and can be applied to neural network training. Section 3.4 explores the application of stochastic MG/Opt to neural network training. We discuss different hierarchical strategies, such as varying network depth (e.g., finer discretizations in ResNets), varying resolution of data, and varying the variance of gradient estimates. Section 3.5 discusses strengths and weaknesses of the proposed methods and the effects of hierarchy selection.

3.2. Related Work

There already exist several works which apply multigrid optimization to the training of deep Residual Neural Networks (ResNets). There exists a variety of work around the group of Rolf Krause [12, 35, 126, 73] which applies multigrid optimization to machine learning problems, especially the training of deep ResNets. The group focuses on developing numerically well-performing methods for the training of deep neural networks without necessarily establishing theoretical convergence guarantees. A multilevel approach to training a machine learning model is proposed in [12] by constructing a hierarchy of models of varying sample size and apply the approach to logistic regression tasks. A multilevel minimization framework for training ResNets is developed in [35]. The authors develop a method from MG/Opt and construct a hierarchy of neural network training problems by interpreting a suitable class of ResNets as an explicit Euler discretization of an initial value problem and vary the number of layers through changing the time step size. The method is applied to MNIST (see Section 2.2.2) classification and a speed-up in training compared to the classical SGD method is observed. The approach is further extended in [126], where an additional line search strategy for the coarse grid correction is considered. The authors state that their method can also be used to prune a network. The work [73] develops a stochastic version of the multilevel trust-region (RMTR) method, which provably converges globally for non-convex problems. It is able to adapt the mini-batch size during training and include curvature information via a limited-memory Symmetric-Rank-1 update. The method is then applied to ResNet training. The work is based on the (deterministic) RMTR method proposed in [46] and expanded to a stochastic setting. RMTR is also expanded to handling bound constraints in [44] and its convergence theory is extended in [47]. The theory in [46] takes into account that the SGD method is used for neural network training, i.e. stochastic gradient estimates are used, but requires that access to the exact gradient is available. Finally, [45] develops an objective-function-free multilevel trust-region method for non-convex stochastic optimization solely relying on gradient and function estimates.

A different idea to incorporate multigrid ideas into stochastic optimization is taken in [89], where a variance reduction technique for SGD is proposed by creating a hierarchy of problems with varying batch sizes. Also, [122] examined training neural networks with coarse-level correction schemes.

The work [68] develops a MG/Opt variant for multilevel training of deep Convolutional Neural Networks inspired by [73]. It combines classical SGD theory for non-convex problems with the coarse grid correction of MG/Opt, but also requires access to the exact gradient. It is developed to handle arbitrarily many levels and relies on checking an angle condition in the method. This work is the closest to our approach; we discuss similarities and differences in more detail after presenting our method and its convergence analysis.

3.3. Stochastic MG/Opt for Two Levels

The aim is to develop a stochastic variant of the MG/Opt method for two levels, which then can be applied to machine learning problems such as neural network training with accompanied theoretical convergence results. Since the MG/Opt method is deduced from a method which was originally developed for solving linear systems, it is easier to consider strongly convex quadratic optimization problems first in the context of stochastic gradient estimates. Subsequently, we develop a stochastic variant tailored to non-convex problems and analyze its convergence properties. We refer the reader to Section 2.4 for the general notation, and an introduction to MG/Opt and SGD.

3.3.1. Strongly Convex Quadratic Setting

We begin by considering the setting where we aim to solve a quadratic symmetric positive definite (s.p.d.) optimization problem using MG/Opt. We assume that we have restriction and prolongation operators already defined between the parameter space $\Theta = \mathbb{R}^{n_\Theta}$ of the optimization problem and a coarser space $\Theta_H = \mathbb{R}^{n_{\Theta_H}}$ with dimensions n_Θ and n_{Θ_H}, respectively, where $n_{\Theta_H} \leq n_\Theta$. The first goal is to construct a quadratic coarse optimization problem out of the fine optimization problem using the given restriction and prolongation operators. The original optimization problem then corresponds to the fine optimization problem in MG/Opt and the constructed coarse optimization problem can be used as the coarse optimization problem in MG/Opt. As a next step, we examine the properties of the constructed coarse optimization problem and propose a variant of the MG/Opt method in this setting when SGD is used as a smoother. Thereafter, we define specific restriction and prolongation operators tailored to SGD iterations in the quadratic setting. Finally, we analyze the convergence of the proposed MG/Opt variant in expectation. This requires strong assumptions on the problem setting and the used operators, which we weaken in closing of this subsection.

Suppose we have a high-dimensional unconstrained optimization problem with a quadratic objective function. The fine optimization problem has the form

$$\min_{\theta \in \Theta} \frac{1}{2} \theta^T A_{\text{fine}} \theta - b_{\text{fine}}^T \theta,$$

where $A_{\text{fine}} \in \mathbb{R}^{n_\Theta \times n_\Theta}$ is a symmetric positive definite matrix and $b_{\text{fine}} \in \mathbb{R}^{n_\Theta}$ is a vector.
We additionally assume that we have access to linear restriction and prolongation operators

$$\mathcal{R}_p : \Theta \to \Theta_H,$$
$$\mathcal{P}_p : \Theta_H \to \Theta,$$

which can be used to transfer parameters and gradients between the fine and coarse space.

Construction of Coarse Optimization Problem To construct the coarse optimization problem, we can use the already available restriction operator $\mathcal{R}_p$ and prolongation operator $\mathcal{P}_p$ to define a coarse matrix A_{coarse} and a coarse vector b_{coarse}. The coarse quadratic optimization problem can be constructed as

$$\min_{\bar{\theta} \in \Theta_H} \frac{1}{2} \bar{\theta}^T A_{\text{coarse}} \bar{\theta} - b_{\text{coarse}}^T \bar{\theta}, \tag{3.1}$$

where

$$A_{\text{coarse}} \coloneqq \mathcal{R}_p A_{\text{fine}} \mathcal{P}_p \in \mathbb{R}^{n_{\Theta_H} \times n_{\Theta_H}}, \tag{3.2}$$
$$b_{\text{coarse}} \coloneqq \mathcal{R}_p b_{\text{fine}} \in \mathbb{R}^{n_{\Theta_H}}. \tag{3.3}$$

The natural question is whether the constructed coarse optimization problem (3.1) is a strongly convex optimization problem as well. We show that the constructed coarse matrix A_{coarse} inherits symmetry and positive definiteness (s.p.d.) from the fine matrix A_{fine} under standard conditions on restriction and prolongation.

We introduce the following standard assumption on the restriction and prolongation operators.

Assumption 3.1 (Restriction and Prolongation Operators). *The restriction operator $\mathcal{R}_p$ and the prolongation operator $\mathcal{P}_p$ are linear operators such that*

$$\mathcal{R}_p = c\mathcal{P}_p^T$$

for some $c > 0$. Additionally, the prolongation operator $\mathcal{P}_p$ has full rank, i.e. rank$(\mathcal{P}_p) = \dim(\Theta_H) \eqqcolon n_{\Theta_H}$.

Lemma 3.2 (Coarse Matrix is s.p.d.). *If A_{fine} is symmetric and positive definite (s.p.d.) and assumption 3.1 holds, then A_{coarse} as defined in (3.2) is s.p.d..*

Proof. A_{coarse} is symmetric since A_{fine} is symmetric and $\mathcal{R}_p = c\mathcal{P}_p^T$.
Next we show the positive definiteness. Let $\bar{\theta} \neq 0$ be arbitrary. By construction of the coarse matrix

A_{coarse} and assumption 3.1 it holds that

$$\begin{aligned}\bar{\theta}^T A_{\text{coarse}} \bar{\theta} &= c\bar{\theta}^T \mathcal{P}_p{}^T A_{\text{fine}} \mathcal{P}_p \bar{\theta} \\ &= c(\mathcal{P}_p \bar{\theta})^T A_{\text{fine}} \mathcal{P}_p \bar{\theta} \\ &= c\theta^T A_{\text{fine}} \theta > 0\end{aligned}$$

for $\theta \coloneqq \mathcal{P}_p \bar{\theta}$. The last inequality holds since A_{fine} is s.p.d. and $\mathcal{P}_p$ has full rank. □

The constructed coarse optimization problem (3.1) leads to a special property of the coarse grid correction (2.45) in MG/Opt. We are going to show that if the coarse grid correction is computed with a stationary point of the cgc objective, then the cgc e satisfies a specific equation involving the restricted fine gradient. This property can be used in the convergence analysis of MG/Opt in the following.

Lemma 3.3 (Property of Constructed Coarse Matrix). *If $\bar{\theta}^*_{cgc}$ is a stationary point of the cgc objective h, which is used to compute the cgc e as defined in* (2.45)*, then*

$$\mathcal{R}_g(\nabla f(\theta^k)) = -\mathcal{R}_p A_{fine} e.$$

Proof.

$$\begin{aligned}\mathcal{R}_p A_{\text{fine}} e &= \mathcal{R}_p A_{\text{fine}} \mathcal{P}_p (\bar{\theta}^*_{cgc} - \mathcal{R}_p(\theta^k)) \\ &= A_{\text{coarse}} (\bar{\theta}^*_{cgc} - \mathcal{R}_p(\theta^k)) \\ &= A_{\text{coarse}} \bar{\theta}^*_{cgc} - b_{\text{coarse}} - A_{\text{coarse}} \mathcal{R}_p(\theta^k) + b_{\text{coarse}} \\ &= \nabla f_H(\bar{\theta}^*_{cgc}) - \nabla f_H(\bar{\theta}^k) \\ &= -\mathcal{R}_g(\nabla f(\theta^k))\end{aligned}$$

where the last equality holds because $\bar{\theta}^*_{cgc}$ is a stationary point of the cgc objective h (2.43). □

Remark 3.4. *For non-stationary points $\bar{\theta}^*_{cgc}$ it holds that*

$$\mathcal{R}_p A_{fine} e = -\mathcal{R}_g(\nabla f(\theta^k)) + \nabla h(\bar{\theta}^*_{cgc}).$$

MG/Opt Variant with SGD as Iterative Method In the following, we consider the strongly convex quadratic setting and construct the coarse grid optimization problem as described above in (3.1). The proposed algorithm for the MG/Opt variant in this strongly convex setting is given in pseudocode in Algorithm 4. It is extended to use gradient estimates instead of exact gradients wherever possible.

Algorithm 4 MG/Opt algorithm for quadratic strongly convex fine objective function with SGD

```
Require: Initial guess $\theta^0$, $\xi_k$ i.i.d., $\alpha > 0$, $\alpha_{cgc} > 0$, $\kappa < 1$, $\mathcal{R}_p, \mathcal{R}_g$, $\mathcal{P}_p$, procedure to compute $e$ exactly
  for $k = 0, 1, 2, \dots$ do
    if $\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 > \kappa\|\nabla f(\theta^k)\|^2$ and last cgc update not computed at iteration $k-1$ then
      Compute cgc $e$ by solving (2.43) exactly
      $\theta^{k+1} := \theta^k + \alpha_{cgc}e$
    else
      $\theta^{k+1} := \theta^k - \alpha G(\theta^k, \xi_k)$
    end if
  end for
```

Remark 3.5 (Necessity of Exact Gradient Access)**.** *The results in Theorem 3.12, Corollary 3.13 and Corollary 3.14 are going to rely on access to the exact gradient $\nabla f(\theta^k)$ to ensure that the condition*

$$\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 > \kappa\|\nabla f(\theta^k)\|^2$$

(cf. Algorithm 4) is satisfied. Checking the κ-condition can not be performed with a gradient estimate $G(\theta^k, \xi_k)$ because we aim at using this condition for a convergence in expectation (Definition 2.8) result.

Developing Restriction and Prolongation Operators Suited for SGD in the Strongly Convex Quadratic Setting To achieve good performance of MG/Opt, the choice of the restriction and prolongation operators should, as it is the case for multigrid methods for PDEs, take the choice of the iterative method used into account. In the following, we thus develop and analyze a specific choice of restriction and prolongation operator suited to using SGD as the iterative method in MG/Opt. Hence, we are interested in the behavior of the SGD iterates in the strongly convex quadratic setting along the iterations.

In the quadratic strongly convex setting, [132] states that SGD converges along the large eigenvalue directions of A_{fine} when a moderate (i.e. not too small) learning rate is used. Contrary to the behavior of SGD, GD follows the small eigenvalue directions first. For small learning rates, SGD converges along the directions of the smaller eigenvalues first, as is GD.

The behavior described in [132] can be observed numerically in Figure 3.1 and Figure 3.2. In the figures, we compare the loss of MG/Opt using only one coarse grid correction update at SGD iteration 50 computed with three different choices for the restriction operator $\mathcal{R}_p$ for a quadratic fine problem. We compare the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the largest eigenvalues, the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the smallest eigenvalues and a random restriction operator. Additionally, the cgc update is only computed if the condition

$$\|\mathcal{R}_g\nabla f(\theta^k)\| \geq \kappa_1\|\nabla f(\theta^k)\| \text{ and } \|\mathcal{R}_g\nabla f(\theta^k)\| \geq \kappa_2 \tag{3.4}$$

is satisfied for some $\kappa_1, \kappa_2 > 0$. The cgc is computed by taking 50 SGD steps on the coarse problem starting from $\mathcal{R}_p(\theta^k)$. Additionally, we consider the loss of SGD without a cgc update at iteration 50.

We perform the experiment with both the fixed moderate learning rate 0.01 and the small learning rate 0.001, respectively. For more details on the experimental setup, we refer to Appendix B.2. We observe in Figure 3.1 that for the moderate learning rate, the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the smallest eigenvalues leads to the fastest decrease of the objective function after the cgc update at iteration 50. This shows that the SGD method has first eliminated the error parts associated to the large eigenvalues. In Figure 3.2, for the small learning rate, the opposite behavior can be observed, as is expected from [132]. Further, we observe that for the small learning rate scenario, the cgc is not computed for the restriction operator *smallev*, since the condition (3.4) for computing the cgc is not even satisfied. Hence, neglecting computational resources, we propose that a suitable restriction operator $\mathcal{R}_p$ consists of rows of the eigenvectors of the fine matrix A_{fine} corresponding to the largest or smallest eigenvalues, depending on the learning rate. Since we are mainly interested in using SGD with small learning rates, we focus on the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the largest eigenvalues in the following. We set $\mathcal{R}_g := \mathcal{R}_p$ and $\mathcal{P}_p := \mathcal{R}_p^T$.

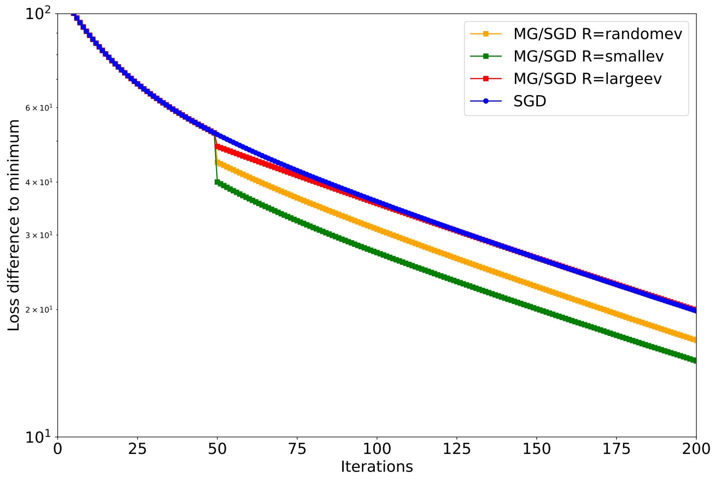


Figure 3.1.: Comparison of the loss of the MG/Opt method with one coarse grid correction after 50 SGD iterations for different choices restriction operator $\mathcal{R}_p$ for a quadratic fine problem with fixed (moderate) lr $\alpha = 0.01$. We compare the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the largest eigenvalues (red), the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the smallest eigenvalues (green) and a restriction operator using a random selection of eigenvectors (orange). Additionally, we plot the loss of SGD with lr $\alpha = 0.01$ (blue) without a cgc update. Up until iteration 50 all methods behave the same, since no cgc update is performed before.

Definition 3.6 (Restriction and Prolongation Suited for SGD)**.** *The restriction operator $\mathcal{R}_p : \mathbb{R}^{n_\Theta} \to \mathbb{R}^{n_{\Theta_H}}$ restricting parameters to a coarse space of dimension n_{Θ_H} consists of rows of the n_{Θ_H} eigenvectors of the fine matrix A_{fine} corresponding to the largest eigenvalues. We set $\mathcal{R}_g := \mathcal{R}_p$ and $\mathcal{P}_p := \mathcal{R}_p^T$.*

Lemma 3.7 (Restriction Operator with Eigenvectors as Rows)**.** *Consider the restriction operator from Definition 3.6. Then, $\mathcal{P}_p\mathcal{R}_g$ is the orthogonal projection (w.r.t. the Euclidean inner product) onto the subspace spanned by the eigenvectors of A_{fine}, corresponding to the n_{Θ_H} largest eigenvalues. Also, $\mathcal{R}_p\mathcal{P}_p = id$.*

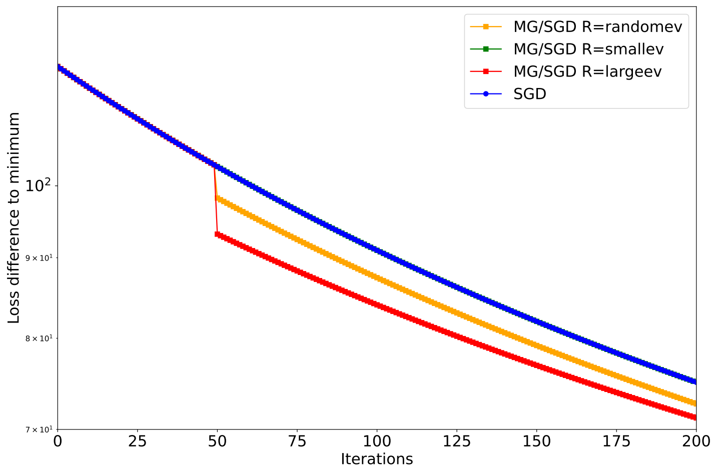


Figure 3.2.: Comparison of the loss of the MG/Opt method with one coarse grid correction after 50 SGD iterations for different choices restriction operator $\mathcal{R}_p$ for a quadratic fine problem with fixed (small) lr $\alpha = 0.001$. We compare the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the largest eigenvalues (red), the restriction operator consisting of rows of the eigenvectors of A_{fine} corresponding to the smallest eigenvalues (green) and a restriction operator using a random selection of eigenvectors (orange). Additionally, we plot the loss of SGD with lr $\alpha = 0.001$ (blue) without a cgc update. Up until iteration 50 all methods behave the same, since no cgc update is performed before.

Remark 3.8 (Restriction Suited for SGD). *The restriction proposed in Definition 3.6 is too expensive to construct in practice, since eigenvectors of A_{fine} are needed. Hence, we only consider this case for theoretical purposes. In the small learning rate regime, the restriction operator should consist of rows of the eigenvectors of A_{fine} corresponding to the largest eigenvalues. Hence, it could be possible to approximate it by using a few iterations of the power method. This approximation is not examined here.*

Analysis of the Coarse Grid Correction with Restriction and Prolongation in Definition 3.6

Because of the special structure of prolongation and restriction in Definition 3.6, we have additional information about the relation between the fine and coarse optimization problem and the coarse grid correction.

Lemma 3.9 (Relation between Fine and Coarse Optimization Problem for Restriction Defined in Definition 3.6). *It additionally holds that*

1. *A_{fine} and $\mathcal{P}_p\mathcal{R}_g$ commute, i.e. $A_{fine}\mathcal{P}_p\mathcal{R}_g = \mathcal{P}_p\mathcal{R}_g A_{fine}$ and $A_{coarse}^{-1} = \mathcal{R}_p A_{fine}^{-1}\mathcal{P}_p$.*
2. *$\mathcal{R}_p\theta^* = \bar{\theta}^* = \bar{\theta}^*_{cgc}$, i.e. the minimizer of the coarse problem (3.1) and cgc problem (2.43) is the restriction of the minimizer of the fine problem.*

Further, if $\bar{\theta}^*_{cgc}$ is a stationary point of the cgc objective h, which is used to compute the cgc e as defined in (2.45), then

3. $(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{fine}e = 0.$
4. $e = \mathcal{P}_p\mathcal{R}_g(\theta^* - \theta^k).$

Proof. The first point follows from the fact that $\mathcal{P}_p\mathcal{R}_g$ is the orthogonal projection onto the subspace spanned by eigenvectors of A_{fine} (cf. Lemma 3.7). Further, it holds that

$$\begin{aligned}A_{\text{coarse}}\mathcal{R}_pA_{\text{fine}}^{-1}\mathcal{P}_p &= \mathcal{R}_pA_{\text{fine}}\mathcal{P}_p\mathcal{R}_pA_{\text{fine}}^{-1}\mathcal{P}_p = \mathcal{R}_pA_{\text{fine}}A_{\text{fine}}^{-1}\mathcal{P}_p = \mathcal{R}_p\mathcal{P}_p = \mathbb{1},\\ \mathcal{R}_pA_{\text{fine}}^{-1}\mathcal{P}_pA_{\text{coarse}} &= \mathcal{R}_pA_{\text{fine}}^{-1}\mathcal{P}_p\mathcal{R}_pA_{\text{fine}}\mathcal{P}_p = \mathcal{R}_pA_{\text{fine}}^{-1}A_{\text{fine}}\mathcal{P}_p = \mathcal{R}_p\mathcal{P}_p = \mathbb{1},\end{aligned}$$

which shows that $A_{\text{coarse}}^{-1} = \mathcal{R}_pA_{\text{fine}}^{-1}\mathcal{P}_p$.

For the second point, because of the quadratic structure it holds that $\theta^* = A_{\text{fine}}^{-1}b_{\text{fine}}$ and hence $\mathcal{R}_p\theta^* = \mathcal{R}_pA_{\text{fine}}^{-1}b_{\text{fine}}$. At the same time it holds that $\bar{\theta}^* = A_{\text{coarse}}^{-1}b_{\text{coarse}} = (\mathcal{R}_pA_{\text{fine}}\mathcal{P}_p)^{-1}\mathcal{R}_pb_{\text{fine}}$.
In the first point we have shown that $(\mathcal{R}_pA_{\text{fine}}\mathcal{P}_p)^{-1} = \mathcal{R}_pA_{\text{fine}}^{-1}\mathcal{P}_p$ and hence

$$\bar{\theta}^* = (\mathcal{R}_pA_{\text{fine}}\mathcal{P}_p)^{-1}\mathcal{R}_pb_{\text{fine}} = \mathcal{R}_pA_{\text{fine}}^{-1}\mathcal{P}_p\mathcal{R}_pb_{\text{fine}} = \mathcal{R}_pA_{\text{fine}}^{-1}b_{\text{fine}} = \mathcal{R}_p\theta^*,$$

where the last equality holds because A_{fine} and $\mathcal{P}_p\mathcal{R}_g$ commute.

Secondly, since $\bar{v}_k^\top = \mathcal{R}_g(A_{\text{fine}}\theta^k + b_{\text{fine}}) - (A_{\text{coarse}}\mathcal{R}_p\theta^k + \mathcal{R}_pb_{\text{fine}}) = 0$, the cgc objective h has the same minimizer as the coarse problem (3.1).

For the third point, we use the first point and show $\mathcal{P}_p\mathcal{R}_gA_{\text{fine}}e = A_{\text{fine}}e$ which is equivalent to the statement. It holds that

$$\begin{aligned}\mathcal{P}_p\mathcal{R}_gA_{\text{fine}}e &= A_{\text{fine}}\mathcal{P}_p\mathcal{R}_ge\\ &= A_{\text{fine}}\mathcal{P}_p\mathcal{R}_p\mathcal{P}_p(\bar{\theta}_{cgc}^* - \mathcal{R}_p(\theta^k))\\ &= A_{\text{fine}}\mathcal{P}_p(\bar{\theta}_{cgc}^* - \mathcal{R}_p(\theta^k))\\ &= A_{\text{fine}}e.\end{aligned}$$

For the fourth point, we use

$$\begin{aligned}e &= \mathcal{P}_p(\bar{\theta}_{cgc}^* - \mathcal{R}_p(\theta^k))\\ &= \mathcal{P}_p(\mathcal{R}_p(\theta^*) - \mathcal{R}_p(\theta^k))\\ &= \mathcal{P}_p\mathcal{R}_g(\theta^* - \theta^k)\end{aligned}$$

where we have used the second point in the second equality. □

Now we show how the distance to the minimizer of the fine objective decreases with the cgc update.

Theorem 3.10 (Decrease of Distance to Minimizer with Coarse Grid Correction Update)**.** *Assume that the restriction operator is defined as in Definition 3.6 and let $\theta^{k+1} = \theta^k + e$ be with an exact computation of the cgc. Then, the distance to the minimizer of the fine objective decreases, i.e.*

$$\|\theta^{k+1} - \theta^*\|^2 \leq \|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)(\theta^k - \theta^*)\|^2.$$

Remark 3.11. *The result in Theorem 3.10 is independent of the used optimizer and its stochasticity which was used to compute e since we assume that the cgc problem is exactly solved.*

Proof. The squared norm of the distance to the minimizer of the fine objective after the cgc update can be computed as

$$\begin{aligned}\|\theta^{k+1} - \theta^*\|^2 &= \|\theta^k + e - \theta^*\|^2 \\ &= \|\theta^k - \theta^* + \mathcal{P}_p\mathcal{R}_g(\theta^* - \theta^k)\|^2 \\ &= \|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)(\theta^k - \theta^*)\|^2\end{aligned}$$

where we have used the fourth point of Lemma 3.9 in the second equality. □

There is an alternative way to show a decrease of the distance to the minimizer of the fine objective with the cgc update, which we state in the following.

Theorem 3.12 (Decrease of Distance to Minimizer with Coarse Grid Correction Update with κ Condition)**.** *Assume that the restriction operator is defined as in Definition 3.6 and $\theta^{k+1} = \theta^k + e$ with an exact computation of the cgc. Further, e has been computed since $\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 > \kappa\|\nabla f(\theta^k)\|^2$ (cf. Algorithm 4). Then, the distance to the minimizer of the fine objective decreases for $\kappa > 1 - \frac{\mu^2}{L^2}$, i.e.*

$$\|\theta^{k+1} - \theta^*\|^2 \leq \frac{L^2}{\mu^2}(1-\kappa)\|\theta^k - \theta^*\|^2,$$

where μ is the strong convexity constant and L a Lipschitz constant of the gradient of the fine objective function.

Proof. Use

$$\mu^2\|\theta^k - \theta^*\|^2 \leq \|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 \leq L^2\|\theta^k - \theta^*\|^2$$

and

$$\|\nabla f(\theta^{k+1}) - \nabla f(\theta^*)\|^2 = \|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + 2\alpha_{cgc}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) + \alpha_{cgc}^2\|A_{\text{fine}}e\|^2.$$

Then, we compute

$$\begin{aligned}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) &= (\nabla f(\theta^k) - \nabla f(\theta^*), \mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e) + (\nabla f(\theta^k) - \nabla f(\theta^*), (\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e)\\ &= (\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e)\\ &= -(\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k)))\\ &= -\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2\end{aligned}$$

and

$$\begin{aligned}\|A_{\text{fine}}e\|^2 &= \|\mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e\|^2 + \|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e\|^2\\ &= \|\mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e\|^2\\ &= \|\mathcal{P}_p\mathcal{R}_g\nabla f(\theta^k)\|^2.\end{aligned}$$

Hence, we can conclude that

$$\begin{aligned}\|\theta^{k+1} - \theta^*\|^2 &\leq \frac{1}{\mu^2}\|\nabla f(\theta^{k+1}) - \nabla f(\theta^*)\|^2\\ &= \frac{1}{\mu^2}(\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + 2\alpha_{cgc}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) + \alpha_{cgc}^2\|A_{\text{fine}}e\|^2)\\ &= \frac{1}{\mu^2}(\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 - 2\alpha_{cgc}\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 + \alpha_{cgc}^2\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2)\\ &\leq \frac{1}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 - \kappa\frac{2\alpha_{cgc}}{\mu^2}\|\nabla f(\theta^k)\|^2 + \kappa\frac{\alpha_{cgc}^2}{\mu^2}\|\nabla f(\theta^k)\|^2\\ &= \frac{1}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 - \kappa\frac{2\alpha_{cgc}}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + \kappa\frac{\alpha_{cgc}^2}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2\\ &= \frac{1}{\mu^2}(1-\kappa)\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 \text{ for } \alpha_{cgc} = 1,\\ &\leq \frac{L^2}{\mu^2}(1-\kappa)\|\theta^k - \theta^*\|^2.\end{aligned}$$

□

While the results from Theorem 3.12 are weaker than the results from Theorem 3.10, they have the advantage that they can be better extended to settings with weakened assumptions, which is what we want to do in the following. First, we can show a result for the case where the cgc is only computed approximately.

Corollary 3.13 (Decrease of Distance to Minimizer with Approximate Coarse Grid Correction Update with κ Condition). *Assume that the restriction operator is defined as in Definition 3.6 and $\theta^{k+1} = \theta^k + e$ is computed with an approximate computation of the cgc. Further, e has been computed since*

$$\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 > \kappa\|\nabla f(\theta^k)\|^2$$

(cf. Algorithm 4). Then, the distance to the minimizer of the fine objective changes for $\kappa > 1 - \frac{\mu^2}{L^2}$ by the

relation

$$\|\theta^{k+1} - \theta^*\|^2 \leq \frac{L^2}{\mu^2}(1-\kappa)\|\theta^k - \theta^*\|^2 + \frac{C_k(r)}{\mu^2},$$

where $C_k(r)$ depends on $r \coloneqq \nabla h(\bar{\theta}^*_{cgc})$.

The color orange indicates the terms that differ from Theorem 3.12. For a small enough residual r, the term $C_k(r)$ becomes small. Accordingly, for an approximate solution of the cgc problem with a small residual, we still are able to get a decrease of the distance to the minimizer of the fine objective.

Proof. Use

$$\mu^2\|\theta^k - \theta^*\|^2 \leq \|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 \leq L^2\|\theta^k - \theta^*\|^2$$

and

$$\|\nabla f(\theta^{k+1}) - \nabla f(\theta^*)\|^2 = \|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + 2\alpha_{cgc}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) + \alpha_{cgc}^2\|A_{\text{fine}}e\|^2.$$

Then, we compute

$$\begin{aligned}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) &= (\nabla f(\theta^k) - \nabla f(\theta^*), \mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e) + (\nabla f(\theta^k) - \nabla f(\theta^*), (\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e)\\ &= (\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e)\\ &= -(\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_d(\nabla f(\theta^k))) + (\nabla f(\theta^k), \mathcal{P}_p r)\\ &= -\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 + (\nabla f(\theta^k), \mathcal{P}_p r)\end{aligned}$$

for $r \coloneqq \nabla h(\bar{\theta}^*_{cgc})$ and

$$\begin{aligned}\|A_{\text{fine}}e\|^2 &= \|\mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e\|^2 + \|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e\|^2\\ &= \|\mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e\|^2\\ &= \|\mathcal{P}_p\mathcal{R}_g\nabla f(\theta^k)\|^2 + \|\mathcal{P}_p r\|^2 + 2(\mathcal{P}_p\mathcal{R}_g\nabla f(\theta^k), \mathcal{P}_p r).\end{aligned}$$

Hence, we can conclude that

$$\begin{aligned}&\|\theta^{k+1} - \theta^*\|^2 \leq \frac{1}{\mu^2}\|\nabla f(\theta^{k+1}) - \nabla f(\theta^*)\|^2\\ &= \frac{1}{\mu^2}(\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + 2\alpha_{cgc}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) + \alpha_{cgc}^2\|A_{\text{fine}}e\|^2)\\ &= \frac{1}{\mu^2}(\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 - 2\alpha_{cgc}\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 + \alpha_{cgc}^2\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2\\ &+2\alpha_{cgc}(\nabla f(\theta^k), \mathcal{P}_p r) + \alpha_{cgc}^2\|\mathcal{P}_p r\|^2 + 2\alpha_{cgc}^2(\mathcal{P}_p\mathcal{R}_g\nabla f(\theta^k), \mathcal{P}_p r))\\ &\leq \frac{1}{\mu^2}(1-\kappa)\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + \frac{1}{\mu^2}(2(\nabla f(\theta^k), \mathcal{P}_p r) + \|\mathcal{P}_p r\|^2 + 2(\mathcal{P}_p\mathcal{R}_g\nabla f(\theta^k), \mathcal{P}_p r)) \text{ for } \alpha_{cgc} = 1,\\ &\leq \frac{L^2}{\mu^2}(1-\kappa)\|\theta^k - \theta^*\|^2 + \frac{C_k(r)}{\mu^2},\end{aligned}$$

for $C_k(r) \coloneqq 2(\nabla f(\theta^k), \mathcal{P}_p r) + \|\mathcal{P}_p r\|^2 + 2(\mathcal{P}_p\mathcal{R}_g\nabla f(\theta^k), \mathcal{P}_p r)$. □

We can also extend Theorem 3.12 to the case where the restriction operator only has to have orthonormal rows, to weaken the strong assumption from Definition 3.6.

Corollary 3.14 (Decrease of Distance to Minimizer with Coarse Grid Correction Update Using κ Condition with Restriction with Orthonormal Rows)**.** *Assume that the restriction operator has orthonormal rows, set $\mathcal{R}_g := \mathcal{R}_p, \mathcal{P}_p := \mathcal{R}_p^T$ and $\theta^{k+1} = \theta^k + e$ with an exact computation of the cgc, in which e had been computed since $\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 > \kappa\|\nabla f(\theta^k)\|^2$ (cf. Algorithm 4). Then, the distance to the minimizer of the fine objective decreases for $\kappa > (1 - \frac{\mu^2}{L^2})/(1+\eta)$, i.e.*

$$\|\theta^{k+1} - \theta^*\|^2 \leq \frac{L^2}{\mu^2}(1 - \kappa(1+\eta))\|\theta^k - \theta^*\|^2,$$

if we additionally assume that

$$\|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{fine}e\|^2 \leq \eta\|\mathcal{P}_p\mathcal{R}_g A_{fine}e\|^2 \text{ and } (\nabla f(\theta^k), (\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{fine}e) \leq \eta(\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_g A_{fine}e),$$

for some $\eta > 0$.

The color purple indicates the parts that differ from Theorem 3.12 because of the more general restriction and prolongation. The parameter η measures how well the restriction and prolongation pair approximate the ideal case from Definition 3.6. The term $\|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e\|$ is zero in the ideal case and needs to be dominated by $\|\mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e\|$ in the more general case. The same is needed for the inner products $(\nabla f(\theta^k), (\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e)$ and $(\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e)$. Smaller values of η lead to better bounds.

Proof. Use

$$\mu^2\|\theta^k - \theta^*\|^2 \leq \|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 \leq L^2\|\theta^k - \theta^*\|^2$$

and

$$\|\nabla f(\theta^{k+1}) - \nabla f(\theta^*)\|^2 = \|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + 2\alpha_{cgc}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) + \alpha_{cgc}^2\|A_{\text{fine}}e\|^2.$$

Then, we compute

$$\begin{aligned}(\nabla f(\theta^k) - \nabla f(\theta^*), A_{\text{fine}}e) &= (\nabla f(\theta^k) - \nabla f(\theta^*), \mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e) + (\nabla f(\theta^k) - \nabla f(\theta^*), (\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e)\\ &\leq -(1+\eta)(\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k)))\\ &= -(1+\eta)\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2\end{aligned}$$

and

$$\begin{aligned}\|A_{\text{fine}}e\|^2 &= \|\mathcal{P}_p\mathcal{R}_g A_{\text{fine}}e\|^2 + \|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{\text{fine}}e\|^2\\ &\leq (1+\eta)\|\mathcal{P}_p\mathcal{R}_g\nabla f(\theta^k)\|^2.\end{aligned}$$

Hence, we can conclude that

$$\begin{aligned}
&\|\theta^{k+1} - \theta^*\|^2 \le \frac{1}{\mu^2}\|\nabla f(\theta^{k+1}) - \nabla f(\theta^*)\|^2 \\
&= \frac{1}{\mu^2}(\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 - 2\alpha_{cgc}(1+\eta)\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2 + \alpha_{cgc}^2(1+\eta)\|\mathcal{P}_p\mathcal{R}_g(\nabla f(\theta^k))\|^2) \\
&\le \frac{1}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 - \kappa(1+\eta)\frac{2\alpha_{cgc}}{\mu^2}\|\nabla f(\theta^k)\|^2 + \kappa(1+\eta)\frac{\alpha_{cgc}^2}{\mu^2}\|\nabla f(\theta^k)\|^2 \\
&= \frac{1}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 - \kappa(1+\eta)\frac{2\alpha_{cgc}}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 + \kappa(1+\eta)\frac{\alpha_{cgc}^2}{\mu^2}\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 \\
&= \frac{1}{\mu^2}(1 - \kappa(1+\eta))\|\nabla f(\theta^k) - \nabla f(\theta^*)\|^2 \text{ for } \alpha_{cgc} = 1, \\
&\le \frac{L^2}{\mu^2}(1 - \kappa(1+\eta))\|\theta^k - \theta^*\|^2.
\end{aligned}$$

□

Remark 3.15 (More Detailed Upper Bound in Corollary 3.14). *The additional assumptions in Corollary 3.14 (in color) can be refined to*

$$\|(\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{fine}e\|^2 \le \eta_1\|\mathcal{P}_p\mathcal{R}_gA_{fine}e\|^2 \textit{ and } (\nabla f(\theta^k), (\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)A_{fine}e) \le \eta_2(\nabla f(\theta^k), \mathcal{P}_p\mathcal{R}_gA_{fine}e)$$

and $(\mathcal{P}_p\mathcal{R}_gA_{fine}e, (\mathbb{1} - \mathcal{P}_p\mathcal{R}_g)(A_{fine}e)) \le \eta_3\|\mathcal{P}_p\mathcal{R}_gA_{fine}e\|^2$ *which results in a more specific bound.*

$$\|\theta^{k+1} - \theta^*\|^2 \le \frac{L^2}{\mu^2}(1 - \kappa\frac{(1+\eta_1)^2}{(1+\eta_2+2\eta_3)})\|\theta^k - \theta^*\|^2,$$

with $\alpha_{cgc} := \frac{1+\eta_1}{1+\eta_2+2\eta_3}$ *and* $\kappa > (1 - \frac{\mu^2}{L^2})\frac{1+\eta_2+2\eta_3}{(1+\eta_1)^2}$ *for* $1 + \eta_2 + 2\eta_3 > 0$ *and* $(1 + \eta_2 + 2\eta_3)(1 - \frac{\mu^2}{L^2}) < (1 + \eta_1)^2$.

Discussion on Applicability of Results The theoretical developments above rely on very strict assumptions and therefore remain largely a theoretical exercise, even though they are tailored to utilize the smoothing properties of (stochastic) gradient methods for strongly convex quadratic problems. The presented analysis aims to exploit MG/Opt-specific structure rather than providing a theory that covers, for example, arbitrary descent directions. In particular, many of the constructions one would like to use in order to obtain a clear result, most notably restriction and prolongation operators tailored to the fine-level Hessian depend on fine-level curvature information that is as expensive to obtain as solving the original fine problem. While it is possible to weaken these strong assumptions in places such as Corollary 3.13 and Corollary 3.14, the practical applicability of the results remains limited. The classical conditions on restriction and prolongation $\mathcal{R}_g = c\mathcal{P}_p^T$ (for some $c > 0$) yield at best results like Theorem 2.36 that only guarantees the coarse grid correction to be a descent direction and do not deliver more informative theoretical results even for quadratic problems. The work above however serves as a first step towards understanding MG/Opt with (stochastic) gradient methods as iterative methods in strongly convex quadratic optimization.

Extending the framework to stochastic gradients introduces further difficulties from a computational perspective, mainly due to the construction of the coarse matrix and vector. A stochastic coarse matrix and right-hand side faithful to the fine problem would require repeated access to fine-level matrix quantities, so no stochastic gradient of the coarse problem is available which is computationally less expensive to compute than the gradient on the fine problem, and at the same time, preserves the deterministic arguments. As a consequence, we found no natural, low-cost route to an equally simple stochastic algorithm even in the strongly convex quadratic setting. In the non-convex regime the available convergence statements are naturally weaker, which both complicates and, in a sense, also opens opportunities for obtaining insightful results.

3.3.2. Non-Convex Setting

We proceed to extend the MG/Opt framework to non-convex stochastic optimization problems, which are prevalent in machine learning applications such as training deep neural networks. Neural networks have natural hierarchical structures that can be exploited by multigrid methods. We aim to develop a stochastic variant of MG/Opt that leverages SGD iterates to efficiently tackle non-convex stochastic optimization problems with theoretical stochastic convergence guarantees. In this subsection, we present a stochastic version of MG/Opt using SGD iterates, which is tailored to non-convex stochastic optimization problems. We start by outlining the algorithmic framework. Then, we introduce further assumptions needed to establish convergence results for the proposed method. Finally, we present convergence theorems for the proposed stochastic MG/Opt method. First, we consider fixed step sizes and then extend the results to diminishing step sizes. The convergence results for diminishing step sizes are, to the best of our knowledge, the first of their kind for multigrid optimization methods applied to stochastic non-convex problems.

Let $(\Theta, (\cdot,\cdot)), (\Theta_H, (\cdot,\cdot)_H)$ be finite-dimensional Hilbert spaces with induced norms $\|\cdot\|, \|\cdot\|_H$. We consider the optimization problems

$$\min_{\theta \in \Theta} f(\theta) \quad \text{and} \quad \min_{\bar\theta \in \Theta_H} f_H(\bar\theta)$$

with differentiable non-convex objective functions $f\colon \Theta \to \mathbb{R}$ and $f_H\colon \Theta_H \to \mathbb{R}$.

Algorithmic Framework The MG/Opt algorithm tailored to using SGD iterations in the non-convex setting is summarized in Algorithm 5. Direct iterations on the fine problem are computed using stochastic gradient updates with fixed learning rate α and stochastic gradients $G_f(\theta^k, \xi_k)$. The cgc is computed if the condition $\|\mathcal{R}_g \nabla f(\theta^k)\|^2 > \kappa \|\nabla f(\theta^k)\|^2$ for some predefined $\kappa \in (0,1)$ is satisfied and the most recent cgc was not computed in the previous iteration. The cgc is computed by performing K SGD updates on the cgc function, where K is a predefined number of iterations. The SGD updates on the cgc function are performed on a regularized cgc problem with objective function

$$h_k(\bar\theta) := f_H(\bar\theta) + (\mathcal{R}_g \nabla f(\theta^k) - \nabla f_H(\bar\theta^k), \bar\theta) + \frac{\lambda}{2}\|\bar\theta - \bar\theta^k\|^2_{\Theta_H}$$

with fixed learning rate $\bar\alpha$ and stochastic gradient samples $G_{cgc,k}(\bar\theta_i^k, \xi_{k_i})$ of $\nabla h_k(\bar\theta_i^k)$. The cgc is then computed as $e := \mathcal{P}_p(\bar\theta^*_{cgc} - \bar\theta^k)$, where $\bar\theta^*_{cgc}$ is the final iterate of the SGD updates on the cgc function and $\bar{\theta}^k = \mathcal{R}_p \theta^k$ is the restricted fine iterate at iteration k.

Algorithm 5 MG/Opt algorithm for non-convex problems with GD

```
Require: Initial guess $\theta^0$, $0 < \alpha$, $0 < \bar{\alpha}$, $\alpha_{cgc}$, $\kappa \in (0,1)$, $\mathcal{R}_p, \mathcal{R}_d$, $\mathcal{P}_p$, $\lambda$, $K \geq 1$ iterations on cgc
  for $k = 0, 1, 2, \dots, J$ do
    if $\|\mathcal{R}_g \nabla f(\theta^k)\|^2 > \kappa \|\nabla f(\theta^k)\|^2$ and no cgc update for $k-1$ then
      $\bar{\theta}_0^k := \bar{\theta}^k = \mathcal{R}_p \theta^k$
      for $i = 0, 1, \dots, K$ do
        $\bar{\theta}_{i+1}^k := \bar{\theta}_i^k - \bar{\alpha}\, G_{cgc,k}(\bar{\theta}_i^k, \xi_{k_i})$
      end for
      $\bar{\theta}_{cgc}^* := \bar{\theta}_{K+1}^k$
      $e := \mathcal{P}_p(\bar{\theta}_{cgc}^* - \bar{\theta}^k)$
      $\theta^{k+1} := \theta^k + \alpha_{cgc} e$
    else
      $\theta^{k+1} := \theta^k - \alpha G_f(\theta^k, \xi_k)$
    end if
  end for
with $h_k(\bar{\theta}) := f_H(\bar{\theta}) + \langle \mathcal{R}_d f'(\theta^k) - f_H'(\bar{\theta}^k), \bar{\theta} \rangle + \frac{\lambda}{2}\|\bar{\theta} - \bar{\theta}^k\|_{\Theta_H}^2 = f_H(\bar{\theta}) + (\mathcal{R}_g \nabla f(\theta^k) - \nabla f_H(\bar{\theta}^k), \bar{\theta}) + \frac{\lambda}{2}\|\bar{\theta} - \bar{\theta}^k\|_{\Theta_H}^2$.
```

Assumptions for Convergence Analysis In order to give a convergence analysis of Algorithm 5, we need to make some assumptions on the stochastic gradients, the objective functions, the restriction and prolongation operators, the cgc iterates, and the regularization parameter for the cgc.

We start with the following assumptions on the fine and coarse objective functions.

Assumption 3.16 (Fine and Coarse Objective Functions). *We assume the following properties of the fine and coarse objective functions:*

- *The fine objective f is bounded below by $f_{inf} > -\infty$, i.e. $f(\theta) \geq f_{inf}$ for all $\theta \in \Theta$.*
- *The fine objective f is differentiable on Θ and $L-$smooth, i.e.*

$$\|\nabla f(\theta) - \nabla f(\theta')\|^2 \leq L^2 \|\theta - \theta'\|^2$$

  for all $\theta, \theta' \in \Theta$.
- *The coarse objective f_H is twice differentiable on Θ_H and $\bar{L}-$smooth, i.e.*

$$\|\nabla f_H(\bar{\theta}) - \nabla f_H(\bar{\theta}')\|^2 \leq \bar{L}^2 \|\bar{\theta} - \bar{\theta}'\|^2$$

  for all $\bar{\theta}, \bar{\theta}' \in \Theta_H$.

The restriction and prolongation operators for the parameters and the restriction operator for the derivative information need to satisfy the standard assumptions on restriction and prolongation operators (see also Section 2.4.3), which we summarize in the following assumption.

Assumption 3.17 (Restriction and Prolongation Operators). *For restriction and prolongation we consider linear operators*

$$\begin{aligned}
&\mathcal{P}_p \colon \Theta_H \to \Theta,\\
&\mathcal{R}_p \colon \Theta \to \Theta_H,\\
&\mathcal{R}_d \colon \Theta^* \to \Theta_H^*, \quad \textit{inducing}\ \mathcal{R}_g \colon \Theta \to \Theta_H.
\end{aligned}$$

- *There exists a* $c > 0$ *such that* $\langle \mathcal{R}_d \ell, \bar{\theta} \rangle = c \langle \ell, \mathcal{P}_p \bar{\theta} \rangle$ *for all* $\ell \in \Theta^*, \bar{\theta} \in \Theta_H$.
- *Working with gradients this translates to* $(\mathcal{R}_g \theta, \bar{\theta})_H = c(\theta, \mathcal{P}_p \bar{\theta})$ *for all* $\theta \in \Theta, \bar{\theta} \in \Theta_H$.
- *We use the notation* $P^2 := \|\mathcal{P}_p\|^2$ *and* $\rho^2 := \|\mathcal{R}_g\|^2$ *for the operator norms.*

In order to account for the non-convexity of the problem we add a regularization term to the cgc function

$$h(\bar{\theta}) := f_H(\bar{\theta}) + (\mathcal{R}_g \nabla f(\theta^k) - \nabla f_H(\mathcal{R}_p \theta^k))^T \bar{\theta} + \frac{\lambda}{2} \|\bar{\theta} - \bar{\theta}^k\|_{\Theta_H}^2, \tag{3.5}$$

where $\lambda > 0$ is a regularization parameter for the cgc and $\bar{\theta}^k$ is the current coarse grid iterate.

We make the following assumption on the regularization parameter.

Assumption 3.18 (Regularization Parameter). *We assume that* $\lambda > 0$ *is the regularization parameter for the cgc with* $\lambda > \bar{L} + \tau$ *for some* $\tau > 0$.

We make assumptions on the stochastic gradients of the fine and coarse objective functions, which are standard for analyzing SGD methods.

Assumption 3.19 (Stochastic Gradients). *Assume that it is possible to generate i.i.d. realizations* $\xi_0, \xi_1, \ldots$ *of the random variable* ξ.

- *Assume there exists an oracle* $G_f(\theta, \xi)$ *which returns an unbiased estimate of* $\nabla f(\theta)$*, i.e.*
$$\mathbb{E}[G_f(\theta, \xi)] = \nabla f(\theta).$$
- *Assume there exists an oracle* $G_{f_H}(\bar{\theta}, \xi)$ *which returns an unbiased estimate of* $\nabla f_H(\bar{\theta})$*, i.e.*
$$\mathbb{E}[G_{f_H}(\bar{\theta}, \xi)] = \nabla f_H(\bar{\theta}).$$
- *Every possible mini-batch gradient of the coarse objective is Lipschitz-smooth w.r.t.* $\bar{L}_{mb}$*, i.e.*
$$\|G_{f_H}(\bar{\theta}, \xi) - G_{f_H}(\bar{\theta}', \xi)\|^2 \le \bar{L}_{mb}^2 \|\bar{\theta} - \bar{\theta}'\|^2$$
for all $\bar{\theta}, \bar{\theta}' \in \Theta_H$ *and for all* ξ.
- *Let*
$$\mathbb{E}[\|G_f(\theta, \xi)\|^2] \le M^2 + M_G \|\nabla f(\theta)\|^2,$$

and

$$\mathbb{E}[\|G_{f_H}(\bar{\theta}, \xi)\|^2] \leq \bar{M}^2 + \bar{M}_G \|\nabla f_H(\bar{\theta})\|^2.$$

The gradient estimate for the cgc function is given by

$$G_{cgc,k}(\bar{\theta}, \xi_k) \coloneqq G_{f_H}(\bar{\theta}, \xi_k) + \mathcal{R}_g G_f(\theta^k, \xi_k) - G_{f_H}(\mathcal{R}_p \theta^k, \xi_k) + \lambda(\bar{\theta} - \bar{\theta}^k). \tag{3.6}$$

We assume that the learning rates are fixed and have some upper bounds.

Assumption 3.20 (Learning Rates)**.** *We assume that the learning rates α for the SGD updates of the fine objective function, α_{cgc} for the cgc updates of the fine objective function and $\bar{\alpha}$ for the SGD updates for the cgc objective function are fixed with*

$$\begin{aligned} 0 < \alpha &\leq \frac{1}{LM_G}, \\ A_k \coloneqq K\bar{\alpha} &\leq \frac{\nu}{4\rho^2 \bar{L}_{mb}}, \\ 0 < \alpha_{cgc} &\leq \frac{c\tau d_1 \kappa}{2LP^2 A_k^2 \rho^4 M_G^2}. \end{aligned}$$

Moreover, assume that A_k is constant for all cgc updates.

We assume that the iterations on the cgc problem decrease the cgc objective function.

Assumption 3.21 (Decrease on Coarse Grid Correction Objective)**.** *We assume that the final iteration on the cgc problem has a smaller function value of the cgc objective than the initial iterate, i.e.*

$$h(\bar{\theta}^*_{cgc}) \leq h(\bar{\theta}^k) - \epsilon$$

for some $\epsilon \in \mathbb{R}$.

We need to make some technical assumptions on the cgc iterates in order to control the behavior of the cgc updates.

Assumption 3.22 (Lower Bound for Coarse Grid Corrections)**.** *Assume that there exists a $d_1 > 0$ such that*

$$d_1 \|\mathcal{R}_g(\nabla f(\theta^k))\|^2 \leq \mathbb{E}[\|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \mid \theta^k]$$

for all cgc updates.

The next assumption is taken from [68] and is a condition on the cgc iterates.

Assumption 3.23 (Control of the Coarse Grid Correction Iterates). *Assume that there exists a $v > 0$ such that*

$$v\mathbb{E}[\|\bar{\theta}^j_{cgc,k} - \bar{\theta}^k\|^2 \mid \theta^k] \leq \mathbb{E}[\|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \mid \theta^k]$$

for all cgc updates and all iterates $j = 1, \cdots, K$.

Main Result for MG/Opt with SGD in the Non-Convex Setting With these assumptions in place, we can state the main convergence result for MG/Opt with SGD and fixed learning rates in the non-convex setting.

Theorem 3.24 (Main Result with SGD as Solver for Non-Convex Problems). *Let assumptions 3.16 to 3.23 hold. Then the iterates of the algorithm Algorithm 5 satisfy*

$$\mathbb{E}[\frac{1}{J+1}\sum_{k=0}^{J}\|\nabla f(\theta^k)\|^2] \leq \frac{\mathbb{E}[f(\theta^0)] - f_{inf}}{(J+1)\min[\alpha\frac{1}{2}, \alpha_{cgc}\frac{c\tau d_1\kappa}{2}]} + \frac{\max[\frac{\alpha^2 LM^2}{2}, \frac{\alpha_{cgc}^2 2LA_k^2\rho^4 M^2\|P\|^2}{2} - \alpha_{cgc}c\epsilon]}{\min[\alpha\frac{1}{2}, \alpha_{cgc}\frac{c\tau d_1\kappa}{2}]}$$

$$\to \frac{\max[\frac{\alpha^2 LM^2}{2}, \frac{\alpha_{cgc}^2 2LA_k^2\rho^4 M^2\|P\|^2}{2} - \alpha_{cgc}c\epsilon]}{\min[\alpha\frac{1}{2}, \alpha_{cgc}\frac{c\tau d_1\kappa}{2}]}$$

for $J \to \infty$.

All notations used in the theorem are defined in this chapter and are summarized in Appendix A. Before we prove the theorem, we need to establish auxiliary lemmas which are used in the proof. We start with a basic lemma on the decrease of general quadratic expressions, which will be used later on for the learning rates.

Lemma 3.25 (Decrease for General Quadratic Expressions). *Let*

$$g(\alpha) := -a\alpha + b\alpha^2$$

with $a, b > 0$. Then the optimal α is minimizing the expression given by

$$\alpha^* = \frac{a}{2b}$$

and the minimum is given by

$$g(\alpha^*) = -\frac{a^2}{4b}.$$

Additionally, it holds for $0 < \alpha \leq \alpha^*$ that

$$g(\alpha) \leq -\alpha\frac{a}{2}$$

and for $0 < \alpha < \frac{a}{b}$ that

$$g(\alpha) < 0.$$

Proof.

$$\begin{aligned} g'(\alpha) &= -a + 2b\alpha \\ g''(\alpha) &= 2b > 0. \end{aligned}$$

The minimum is at $\alpha^* = \frac{a}{2b}$ and

$$\begin{aligned} g(\alpha^*) &= -a\frac{a}{2b} + b\left(\frac{a}{2b}\right)^2 \\ &= -\frac{a^2}{2b} + \frac{a^2}{4b} \\ &= -\frac{a^2}{4b}. \end{aligned}$$

For $0 < \alpha$ we have

$$\begin{aligned} g(\alpha) &\leq -\alpha\frac{a}{2} \\ \Leftrightarrow \alpha^2 b &\leq \alpha\frac{a}{2} \\ \Leftrightarrow \alpha &\leq \frac{a}{2b}. \end{aligned}$$

For $0 < \alpha$ we have

$$\begin{aligned} g(\alpha) &< 0 \\ \Leftrightarrow -a\alpha + b\alpha^2 &< 0 \\ \Leftrightarrow b\alpha^2 &< a\alpha \\ \Leftrightarrow \alpha &< \frac{a}{b}. \end{aligned}$$

Thus, the claim is proven. □

We bound the decrease of the fine objective function for a SGD step. This is a well-known result for SGD steps, which we state here for completeness.

Lemma 3.26 (Decrease of Fine Objective Function for SGD Updates)**.** *With assumption 3.16, assumption 3.19 and assumption 3.20,*
we get

$$\mathbb{E}[f(\theta^{k+1}) - f(\theta^k)] \leq -\frac{\alpha_k}{2}\mathbb{E}[\|\nabla f(\theta^k)\|^2] + \frac{L}{2}\alpha_k^2 M^2.$$

The proof is a standard result for SGD steps, see e.g. [11, Lemma 4.4] We repeat it here to highlight the techniques used.

Proof. We start with the classical bound for the fine objective function.

$$\begin{aligned}
\mathbb{E}[f(\theta^{k+1}) - f(\theta^k) \mid \theta^k] &= \mathbb{E}[f(\theta^{k+1}) \mid \theta^k] - f(\theta^k) \\
&\leq \mathbb{E}[\langle \nabla f(\theta^k), \theta^{k+1} - \theta^k\rangle \mid \theta^k] + \frac{L}{2}\mathbb{E}[\|\theta^{k+1} - \theta^k\|^2 \mid \theta^k] \\
&= -\mathbb{E}[\langle \nabla f(\theta^k), \alpha_k G_f(\theta^k, \xi_k)\rangle \mid \theta^k] + \frac{L}{2}\alpha_k^2 \mathbb{E}[\|G_f(\theta^k, \xi_k)\|^2 \mid \theta^k] \\
&= -\langle \nabla f(\theta^k), \alpha_k \mathbb{E}[G_f(\theta^k, \xi_k) \mid \theta^k]\rangle + \frac{L}{2}\alpha_k^2 \mathbb{E}[\|G_f(\theta^k, \xi_k)\|^2 \mid \theta^k] \\
&= -\langle \nabla f(\theta^k), \alpha_k \nabla f(\theta^k)\rangle + \frac{L}{2}\alpha_k^2 \mathbb{E}[\|G_f(\theta^k, \xi_k)\|^2 \mid \theta^k] \\
&\leq -\alpha_k \|\nabla f(\theta^k)\|^2 + \frac{L}{2}\alpha_k^2 (M^2 + M_G \|\nabla f(\theta^k)\|^2)
\end{aligned}$$

Using Lemma 3.25 with $a = 1$ and $b = \frac{L}{2}M_G$, we get

$$\mathbb{E}[f(\theta^{k+1}) - f(\theta^k) \mid \theta^k] \leq -\frac{\alpha_k}{2}\|\nabla f(\theta^k)\|^2 + \frac{L}{2}\alpha_k^2 M^2.$$

Taking the expectation w.r.t. θ^k gives

$$\mathbb{E}[f(\theta^{k+1}) - f(\theta^k)] \leq -\frac{\alpha_k}{2}\mathbb{E}[\|\nabla f(\theta^k)\|^2] + \frac{L}{2}\alpha^{k,2} M^2.$$

□

We show that under the above assumptions, the cgc update is a descent direction for the fine objective function at the current fine iterate θ^k. Since the iterates are random objects, we first show that this claim holds in expectation conditioned on the current fine iterate.

Lemma 3.27 (Coarse Grid Correction is a Descent Direction for Fine Objective Function). *With assumptions 3.16 to 3.18 and assumption 3.21, we get*

$$\langle \nabla f(\theta^k), \mathbb{E}[e \mid \theta^k]\rangle \leq -c\tau \mathbb{E}[\|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \mid \theta^k] - c\epsilon.$$

Proof. We use assumption 3.21 as a starting point and the definition of the cgc objective function (3.5).

We consider an expectation w.r.t. the current fine iterate θ^k.

$$
\begin{aligned}
&\mathbb{E}\left[h(\bar\theta^*_{cgc}) \mid \theta^k\right] - h(\bar\theta^k) && \leq -\epsilon \\
\Leftrightarrow &\mathbb{E}\left[f_H(\bar\theta^*_{cgc}) - f_H(\bar\theta^k) + \langle \mathcal{R}_d f'(\theta^k) - f'_H(\bar\theta^k), \bar\theta^*_{cgc} - \bar\theta^k\rangle + \lambda\|\bar\theta^*_{cgc} - \bar\theta^k\|^2 \mid \theta^k\right] && \leq -\epsilon \\
\Leftrightarrow &\mathbb{E}\left[f_H(\bar\theta^*_{cgc}) - f_H(\bar\theta^k) \mid \theta^k\right] \\
&+ \mathbb{E}\left[(\mathcal{R}_g \nabla f(\theta^k), \bar\theta^*_{cgc} - \bar\theta^k) \mid \theta^k\right] \\
&- \mathbb{E}\left[(\nabla f_H(\bar\theta^k), \bar\theta^*_{cgc} - \bar\theta^k) \mid \theta^k\right] \\
&+ \lambda\mathbb{E}\left[\|\bar\theta^*_{cgc} - \bar\theta^k\|^2 \mid \theta^k\right] && \leq -\epsilon \\
\Leftrightarrow &\mathbb{E}\left[f_H(\bar\theta^*_{cgc}) - f_H(\bar\theta^k) \mid \theta^k\right] \\
&+ \frac{1}{c}\mathbb{E}\left[(\nabla f(\theta^k), \mathcal{P}_p(\bar\theta^*_{cgc} - \bar\theta^k)) \mid \theta^k\right] \\
&- \mathbb{E}\left[(\nabla f_H(\bar\theta^k), \bar\theta^*_{cgc} - \bar\theta^k) \mid \theta^k\right] \\
&+ \lambda\mathbb{E}\left[\|\bar\theta^*_{cgc} - \bar\theta^k\|^2 \mid \theta^k\right] && \leq -\epsilon
\end{aligned}
$$

where in the last step we used the fact that $c\mathcal{R}_g = \mathcal{P}_p$. Rearranging the terms and using $c > 0$ yields

$$
\begin{aligned}
(\nabla f(\theta^k), \mathbb{E}[e \mid \theta^k]) \leq\, & c\mathbb{E}\left[(\nabla f_H(\bar\theta^k), \bar\theta^*_{cgc} - \bar\theta^k) \mid \theta^k\right] \\
& - c\mathbb{E}\left[f_H(\bar\theta^*_{cgc}) - f_H(\bar\theta^k) \mid \theta^k\right] \\
& - c\lambda\mathbb{E}\left[\|\bar\theta^*_{cgc} - \bar\theta^k\|^2 \mid \theta^k\right] \\
& - c\epsilon.
\end{aligned} \tag{3.7}
$$

To bound the second term (3.7) on the right-hand side of the inequality, we use the mean value theorem (see e.g. in [80]) for the coarse objective function. The mean value theorem for functions with several variables states that there exists a random variable t whose realizations lie in $(0, 1)$ (t is a random variable since it depends on $\bar\theta^*_{cgc}$ and hence is not measurable w.r.t. θ^k) such that with (the random variable)

$$\bar\theta_t := \bar\theta^k + t(\bar\theta^*_{cgc} - \bar\theta^k)$$

holds that

$$
\begin{aligned}
\mathbb{E}\left[f_H(\bar\theta^*_{cgc}) - f_H(\bar\theta^k) \mid \theta^k\right] &= \mathbb{E}\left[\langle f'_H(\bar\theta_t), \bar\theta^*_{cgc} - \bar\theta^k\rangle \mid \theta^k\right] \\
&= \mathbb{E}\left[(\nabla f_H(\bar\theta_t), \bar\theta^*_{cgc} - \bar\theta^k) \mid \theta^k\right].
\end{aligned}
$$

Hence,

$$
\begin{aligned}
(\nabla f(\theta^k), \mathbb{E}[e \mid \theta^k]) \leq\, & c\mathbb{E}\left[(\nabla f_H(\bar\theta^k) - \nabla f_H(\bar\theta_t), \bar\theta^*_{cgc} - \bar\theta^k) \mid \theta^k\right] \\
& - c\lambda\mathbb{E}\left[\|\bar\theta^*_{cgc} - \bar\theta^k\|^2 \mid \theta^k\right] \\
& - c\epsilon.
\end{aligned}
$$

We again want to transform the first term on the right-hand side of the inequality using the mean value theorem on the coarse gradient. The mean value theorem for functions with several variables and multiple image dimensions states that there exists a random variable t^* whose realizations lie in

$(0, t)$ such that

$$\begin{aligned}\mathbb{E}\left[(\nabla f_H(\bar{\theta}^k) - \nabla f_H(\bar{\theta}_t), \bar{\theta}^*_{cgc} - \bar{\theta}^k) \mid \theta^k\right] &= \mathbb{E}\left[(\bar{\theta}^k - \bar{\theta}_t, \nabla^2 f_H(\bar{\theta}_{t^*})(\bar{\theta}^*_{cgc} - \bar{\theta}^k)) \mid \theta^k\right] \\ &= -\mathbb{E}\left[(\bar{\theta}_t - \bar{\theta}^k, \nabla^2 f_H(\bar{\theta}_{t^*})(\bar{\theta}^*_{cgc} - \bar{\theta}^k)) \mid \theta^k\right] \\ &= -\mathbb{E}\left[(\bar{\theta}^*_{cgc} - \bar{\theta}^k, t\nabla^2 f_H(\bar{\theta}_{t^*})(\bar{\theta}^*_{cgc} - \bar{\theta}^k)) \mid \theta^k\right].\end{aligned}$$

Hence,

$$\begin{aligned}(\nabla f(\theta^k), \mathbb{E}[e \mid \theta^k]) \leq &- c\mathbb{E}\left[(\bar{\theta}^*_{cgc} - \bar{\theta}^k, t\nabla^2 f_H(\bar{\theta}_{t^*})(\bar{\theta}^*_{cgc} - \bar{\theta}^k)) \mid \theta^k\right] \\ &- c\lambda\mathbb{E}\left[\|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \mid \theta^k\right] \\ &- c\epsilon \\ \leq &c\mathbb{E}\left[(\bar{\theta}^*_{cgc} - \bar{\theta}^k, (-t\nabla^2 f_H(\bar{\theta}_{t^*}) - \lambda \mathrm{id})(\bar{\theta}^*_{cgc} - \bar{\theta}^k)) \mid \theta^k\right] \\ &- c\epsilon.\end{aligned}$$

Since f_H is $\bar{L}-$smooth, we have

$$\min_{\bar{\theta} \in \Theta_H} \left(\lambda_{min}(\nabla^2 f_H(\bar{\theta}_{t^*}))\right) \geq -\bar{L},$$

and it is $t \in (0, 1)$. Keep in mind that t is a random variable and cannot be taken out of the expectation. Using assumption 3.18 we conclude

$$(\nabla f(\theta^k), \mathbb{E}[e \mid \theta^k]) \leq -c\tau\mathbb{E}\left[\|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \mid \theta^k\right] - c\epsilon.$$

□

We slightly modify [68, Lemma 3.17] to get a bound on the variance of the cgc update.

Lemma 3.28 (Upper Bound for the Variance of the Coarse Grid Correction). *Take assumption 3.16, assumption 3.19, assumption 3.20, assumption 3.22 and assumption 3.23.*
Then

$$\mathbb{E}[\|e^k\|^2 \mid \theta^k] \leq 2A_k^2\rho^4 M^2 + 2A_k^2\rho^4 M_G\|\nabla f(\theta^k)\|^2$$

with $\rho^2 := \|\mathcal{R}_g\|^2$.

Proof. We denote the iterates of the cgc step by $\bar{\theta}^j_{cgc,k}$ for $j = 0, \ldots, K$. We remind the reader that $\bar{\theta}^0_{cgc,k} := \bar{\theta}^k = \mathcal{R}_p\theta^k$. Because SGD updates are used on the cgc optimization problem, we have

$$\bar{\theta}^j_{cgc,k} = \bar{\theta}^{j-1}_{cgc,k} - \bar{\alpha}^j_k G_{cgc,k}(\bar{\theta}^{j-1}_{cgc,k}, \xi^j_k)$$

for $j = 1, \dots, K$. We can rewrite the cgc update as

$$\begin{aligned} e^k &= \mathcal{P}_p \left(\bar{\theta}^K_{cgc,k} - \bar{\theta}^0_{cgc,k}\right) \\ &= \mathcal{P}_p \sum_{j=1}^{K} \left(-\bar{\alpha}^j_k G_{cgc,k}(\bar{\theta}^{j-1}_{cgc,k}, \xi^j_k)\right). \end{aligned}$$

Hence, conditioned on the current fine iterate θ^k, we have

$$\mathbb{E}[\|e^k\|^2 \mid \theta^k] = \mathbb{E}[\|\mathcal{P}_p \sum_{j=1}^{K} \left(-\bar{\alpha}^j_k G_{cgc,k}(\bar{\theta}^{j-1}_{cgc,k}, \xi^j_k)\right)\|^2 \mid \theta^k] \leq A_k \sum_{j=1}^{K} \bar{\alpha}^j_k \rho^2 \mathbb{E}[\|G_{cgc,k}(\bar{\theta}^{j-1}_{cgc,k}, \xi^j_k)\|^2 \mid \theta^k].$$

Now we consider the squared norm of the stochastic gradients $G_{cgc,k}(\bar{\theta}^{j-1}_{cgc,k}, \xi^j_k)$ of the cgc objective function. It is

$$\begin{aligned} \|G_{cgc,k}(\bar{\theta}^{j-1}_{cgc,k}, \xi^j_k)\|^2 &\leq 2\|\mathcal{R}_g G_f(\theta^k, \xi^j_k)\|^2 + 2\|G_{f_H}(\bar{\theta}^0_{cgc,k}, \xi^j_k) - G_{f_H}(\bar{\theta}^{j-1}_{cgc,k}, \xi^j_k)\|^2 \\ &\leq 2\rho^2 \|G_f(\theta^k, \xi^j_k)\|^2 + 2\bar{L}^2_{mb} \|\bar{\theta}^0_{cgc,k} - \bar{\theta}^{j-1}_{cgc,k}\|^2 \\ &\leq 2\rho^2 \|G_f(\theta^k, \xi^j_k)\|^2 + 2\bar{L}^2_{mb} \nu^{-1} \|e^k\|^2 \end{aligned}$$

where we used assumption 3.17 and assumption 3.19 in the second step and assumption 3.23 in the last step. Using assumption 3.19 again, we can derive an upper bound for the first term $2\rho^2\|G_f(\theta^k, \xi^j_k)\|^2$. Inserting this into the expectation yields

$$\begin{aligned} \mathbb{E}[\|e^k\|^2 \mid \theta^k] &\leq A_k \sum_{j=1}^{K} \bar{\alpha}^j_k \rho^2 \left(2\rho^2 \mathbb{E}[\|G_f(\theta^k, \xi^j_k)\|^2 \mid \theta^k] + 2\bar{L}^2_{mb} \nu^{-1} \mathbb{E}[\|e^k\|^2 \mid \theta^k]\right) \\ &\leq A_k \sum_{j=1}^{K} \bar{\alpha}^j_k \rho^2 \left(2\rho^2 M^2 + 2\rho^2 M_G \|\nabla f(\theta^k)\|^2 + 2\bar{L}^2_{mb} \nu^{-1} \mathbb{E}[\|e^k\|^2 \mid \theta^k]\right). \end{aligned}$$

Rearranging the inequality yields

$$\frac{1}{2}\mathbb{E}[\|e^k\|^2 \mid \theta^k] \leq \left(1 - A^2_k \rho^2 2 \frac{\bar{L}_{mb}}{\nu}\right) \mathbb{E}[\|e^k\|^2 \mid \theta^k] \leq 2A^2_k \rho^4 M^2 + 2A^2_k \rho^4 M_G \|\nabla f(\theta^k)\|^2.$$

It holds that

$$\left(1 - A^2_k \rho^2 2 \frac{\bar{L}_{mb}}{\nu}\right) \geq \frac{1}{2} \Leftrightarrow A^2_k \rho^2 2 \frac{\bar{L}_{mb}}{\nu} \leq \frac{1}{2} \Leftrightarrow A^2_k \leq \frac{\nu}{4\rho^2 \bar{L}_{mb}},$$

which is exactly the condition from assumption 3.20. Thus, we have shown the claim. □

No we can show decrease of the fine objective function for a cgc step.

Lemma 3.29 (Decrease of Fine Objective Function for Coarse Grid Correction Step). *Let assumptions 3.16* *to 3.22 hold. Then,*

$$\mathbb{E}[f(\theta^{k+1}) - f(\theta^k)] \leq -\frac{\alpha^k_{cgc}}{2}(c\tau d_1\kappa)\mathbb{E}[\|\nabla f(\theta^k)\|^2] + \frac{L}{2}\alpha^{k,2}_{cgc}P^2 2A_k^2\rho^4 M^2 - \alpha^k_{cgc}c\epsilon.$$

Proof. We start by conditioning the expectation on the current fine iterate θ^k to bound the difference in terms of the fine gradient at the current fine iterate $\nabla f(\theta^k)$. Once we have derived a bound, we can use assumption 3.19 to bound the expectation thereafter. We start with the classical bound for the fine objective function.

$$\begin{aligned}\mathbb{E}[f(\theta^{k+1}) - f(\theta^k) \mid \theta^k] &= \mathbb{E}[f(\theta^{k+1}) \mid \theta^k] - f(\theta^k)\\ &\leq \mathbb{E}[(\nabla f(\theta^k), \theta^{k+1} - \theta^k) \mid \theta^k] + \frac{L}{2}\mathbb{E}[\|\theta^{k+1} - \theta^k\|^2 \mid \theta^k]\\ &= \mathbb{E}[(\nabla f(\theta^k), \alpha^k_{cgc}e^k) \mid \theta^k] + \frac{L}{2}\alpha^{k,2}_{cgc}\mathbb{E}[\|e^k\|^2 \mid \theta^k]\\ &= (\nabla f(\theta^k), \alpha^k_{cgc}\mathbb{E}[e^k \mid \theta^k]) + \frac{L}{2}\alpha^{k,2}_{cgc}\mathbb{E}[\|e^k\|^2 \mid \theta^k].\end{aligned}$$

Now we can use the result from Lemma 3.27 to bound the inner product and Lemma 3.28 to bound the expectation of the squared norm of the cgc update. We obtain

$$\begin{aligned}\mathbb{E}[f(\theta^{k+1}) - f(\theta^k) \mid \theta^k] &\leq -\alpha^k_{cgc}c\tau\mathbb{E}[\|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \mid \theta^k] - \alpha^k_{cgc}c\epsilon + \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4(M^2 + M_G\|\nabla f(\theta^k)\|^2)\\ &\leq -\alpha^k_{cgc}c\tau d_1\|\mathcal{R}_g\nabla f(\theta^k)\|^2 - \alpha^k_{cgc}c\epsilon + \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4(M^2 + M_G\|\nabla f(\theta^k)\|^2),\end{aligned}$$

where we used assumption 3.22 in the last bound. Using the algorithmic condition (see Algorithm 5) that the cgc update is only performed if

$$\|\mathcal{R}_g\nabla f(\theta^k)\|^2 \geq \kappa\|\nabla f(\theta^k)\|^2,$$

we can derive an upper bound for the first term using κ. We get

$$\begin{aligned}\mathbb{E}[f(\theta^{k+1}) - f(\theta^k) \mid \theta^k] &\leq -\alpha^k_{cgc}c\tau d_1\|\mathcal{R}_g\nabla f(\theta^k)\|^2 - \alpha^k_{cgc}c\epsilon + \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4(M^2 + M_G\|\nabla f(\theta^k)\|^2)\\ &\leq -\alpha^k_{cgc}c\tau d_1\kappa\|\nabla f(\theta^k)\|^2 - \alpha^k_{cgc}c\epsilon + \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4(M^2 + M_G\|\nabla f(\theta^k)\|^2).\end{aligned}$$

Using the assumptions on the learning rate α_{cgc} from assumption 3.20 and Lemma 3.25 with $a = c\tau d_1\kappa$, $b = \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4 M_G$ we can continue with

$$\begin{aligned}\mathbb{E}[f(\theta^{k+1}) - f(\theta^k) \mid \theta^k] &\leq -\alpha^k_{cgc}c\tau d_1\kappa\|\nabla f(\theta^k)\|^2 - \alpha^k_{cgc}c\epsilon + \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4(M^2 + M_G\|\nabla f(\theta^k)\|^2)\\ &\leq -\alpha^k_{cgc}\frac{c\tau d_1\kappa}{2}\|\nabla f(\theta^k)\|^2 - \alpha^k_{cgc}c\epsilon + \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4 M^2.\end{aligned}$$

Now taking the total expectation over the current fine iterate θ^k yields

$$\begin{aligned}\mathbb{E}[f(\theta^{k+1}) - f(\theta^k)] \leq &- \alpha^k_{cgc}\frac{c\tau d_1\kappa}{2}\mathbb{E}[\|\nabla f(\theta^k)\|^2]\\ &- \alpha^k_{cgc}c\epsilon\\ &+ \frac{L}{2}\alpha^{k,2}_{cgc}2A_k^2\rho^4M^2.\end{aligned}$$

□

With the help of the lemmas we have shown so far, we can now prove the main theorem of this section.

Proof of Theorem 3.24. Depending on whether the update is a cgc update or a SGD update, we have two different cases. In the first case, we use Lemma 3.29 to bound the decrease in the fine objective function. In the second case, we use Lemma 3.26 to bound the decrease in the fine objective function. This yields

$$\mathbb{E}[f(\theta^{k+1}) - f(\theta^k)] \leq \begin{cases} -\frac{\alpha_{cgc}}{2}(c\tau d_1\kappa)\mathbb{E}[\|\nabla f(\theta^k)\|^2] + \frac{L}{2}\alpha^2_{cgc}P^2 2A_k^2\rho^4M^2 - \alpha_{cgc}c\epsilon, & \text{if cgc update}\\ -\frac{\alpha}{2}\mathbb{E}[\|\nabla f(\theta^k)\|^2] + \frac{L}{2}\alpha^2M^2, & \text{if SGD update.}\end{cases}$$

Now for constant step sizes α and α_{cgc} and a constant A_k, we can derive the upper bound

$$\begin{aligned}f_{inf} - \mathbb{E}[f(\theta^0)] &\leq \mathbb{E}[f(\theta^{J+1}) - f(\theta^0)]\\ &= \sum_{k=0}^{J}\mathbb{E}[f(\theta^{k+1}) - f(\theta^k)]\\ &\leq -\min(A,B)\sum_{k=0}^{J}\mathbb{E}[\|\nabla f(\theta^k)\|^2] + (J+1)\max(C,D),\end{aligned}$$

with

$$\begin{aligned}A &:= \frac{\alpha_{cgc}}{2}c\tau d_1\kappa,\\ B &:= \frac{\alpha}{2},\\ C &:= \frac{L}{2}\alpha^2_{cgc}M^2P^2 2A_k^2\rho^4,\\ D &:= \frac{L}{2}\alpha^2M^2.\end{aligned}$$

Rearranging the inequality yields

$$\begin{aligned}\frac{1}{J+1}\sum_{k=0}^{J}\mathbb{E}[\|\nabla f(\theta^k)\|^2] &\leq \frac{f_{inf} - \mathbb{E}f(\theta^0)}{\min(A,B)(J+1)} + \frac{J+1}{J+1}\frac{\max(C,D)}{\min(A,B)}\\ &= \frac{f_{inf} - \mathbb{E}f(\theta^0)}{\min(A,B)(J+1)} + \frac{\max(C,D)}{\min(A,B)}.\end{aligned}$$

Examining the limit for $J \to \infty$ we get

$$\liminf_{J\to\infty} \frac{1}{J+1} \sum_{k=0}^{J} \mathbb{E}[\|\nabla f(\theta^k)\|^2] \to \frac{\max(C,D)}{\min(A,B)}.$$

□

Extension to Convergence with Diminishing Step Sizes Since in practice the restriction to fixed step sizes is often not desirable, we now extend the convergence analysis to a broader class of step sizes, the Robbins-Monro type diminishing step sizes [109]. For the next analysis we change the algorithm by replacing fixed step sizes α, α_{cgc} with diminishing step sizes α_k, α_{cgck} at iteration k. We replace assumption 3.20 with

Assumption 3.30 (Diminishing Learning Rates). *We assume that the learning rates α_k, α_{cgck} and $\bar{\alpha}$ are bounded by*

$$\begin{aligned} 0 < \alpha_k &\leq \frac{1}{LM_G}, \\ A_k := K\bar{\alpha} &\leq \frac{\nu}{4\rho^2 \bar{L}_{mb}}, \\ 0 < \alpha_{cgck} &\leq \frac{c\tau d_1 \kappa}{2LP^2 A_k^2 \rho^4 M^2}. \end{aligned}$$

A_k does not have to be constant for all cgc updates. We additionally assume that

$$\sum_{k \text{ SGD step}} \alpha_k = \infty, \quad \sum_{k \text{ SGD step}} \alpha_k^2 < \infty, \text{ and}$$
$$\sum_{k \text{ cgc step}} \alpha_{cgck} = \infty, \quad \sum_{k \text{ cgc step}} \alpha_{cgc}{}_k^2 < \infty.$$

Theorem 3.31 (Convergence for Diminishing Step Sizes). *Replacing assumption 3.20 by assumption 3.30 and assuming $\epsilon \geq 0$ in assumption 3.21 in the assumptions of Theorem 3.24, we get*

$$\liminf_{k\to\infty} \mathbb{E}[\|\nabla f(\theta^k)\|^2] \to 0.$$

Proof of Theorem 3.31. We start with the same bound as in the proof of Theorem 3.24:

$$\mathbb{E}[f(\theta^{k+1}) - f(\theta^k)] \leq \begin{cases} -\frac{\alpha_{cgck}}{2}(c\tau d_1\kappa)\mathbb{E}[\|\nabla f(\theta^k)\|^2] + \frac{L}{2}\alpha_{cgc}{}_k^2 P^2 2A_k^2\rho^4 M^2 - \alpha_{cgck} c\epsilon, & \text{if cgc update} \\ -\frac{\alpha_k}{2}\mathbb{E}[\|\nabla f(\theta^k)\|^2] + \frac{L}{2}\alpha_k^2 M^2, & \text{if SGD update.} \end{cases}$$

Then, for diminishing step sizes α_k and α_{cgck} and a possibly non-constant A_k, we get the upper bound

$$\begin{aligned} f_{inf} - \mathbb{E}[f(\theta^0)] &\leq \mathbb{E}[f(\theta^{J+1}) - f(\theta^0)] \\ &= \sum_{k=0}^{J} \mathbb{E}[f(\theta^{k+1}) - f(\theta^k)] \\ &\leq - \left(\sum_{k=0}^{J} \gamma_k \mathbb{E}[\|\nabla f(\theta^k)\|^2] \right) + C_J - D_J, \end{aligned}$$

with

$$\begin{aligned} \gamma_k &:= \begin{cases} \frac{\alpha_{cgck}}{2}(c\tau d_1 \kappa), & \text{if cgc update} \\ \frac{\alpha_k}{2}, & \text{if SGD update.} \end{cases} \\ C_J &:= \sum_{k=0}^{J} \gamma_k^2 \max(A, B), \\ D_J &:= \sum_{k=0}^{J} \alpha_{cgck} c \epsilon \mathbb{1}_{\text{cgc step}} \geq 0, \end{aligned}$$

where

$$\begin{aligned} A &:= \frac{LM^2}{2} P^2 \rho^2 \frac{\nu}{2\bar{L}_{mb}} \frac{1}{c^2 \tau^2 d_1^2 \kappa^2}, \\ B &:= \frac{LM^2}{2}. \end{aligned}$$

We examine the sum of the γ_k's by

$$\Gamma_J := \sum_{k=0}^{J} \gamma_k.$$

Using assumption 3.30 we can deduce that

$$\liminf_{J \to \infty} \Gamma_J = \infty.$$

At the same time holds that

$$\sum_{k=0}^{\infty} \gamma_k^2 < \infty.$$

Hence,

$$\mathbb{E}\left[\frac{1}{\Gamma_J} \sum_{k=0}^{J} \gamma_k \|\nabla f(\theta^k)\|^2 \right] \leq \frac{C_J}{\Gamma_J} + \frac{\mathbb{E}[f(\theta^0) - f_{inf}]}{\Gamma_J}.$$

Examining the limit for $J \to \infty$ we get

$$\mathbb{E}\left[\frac{1}{\Gamma_J}\sum_{k=0}^{J}\gamma_k\|\nabla f(\theta^k)\|^2\right] \to 0. \tag{3.8}$$

To show the claim of Theorem 3.31 we make an argument via contradiction. Assume that there exists an $\eta > 0$ such that

$$\liminf_{k\to\infty}\mathbb{E}[\|\nabla f(\theta^k)\|^2] > \eta.$$

This implies that there exists a J_0 such that for all $k \geq J_0$ it holds that

$$\mathbb{E}[\|\nabla f(\theta^k)\|^2] \geq \eta + \epsilon$$

for some $\epsilon > 0$. Hence, for $J \geq J_0$ we have

$$\begin{aligned}\mathbb{E}\left[\frac{1}{\Gamma_J}\sum_{k=0}^{J}\gamma_k\|\nabla f(\theta^k)\|^2\right] &= \frac{1}{\Gamma_J}\sum_{k=0}^{J_0}\mathbb{E}\left[\gamma_k\|\nabla f(\theta^k)\|^2\right] + \frac{1}{\Gamma_J}\sum_{k=J_0+1}^{J}\mathbb{E}\left[\gamma_k\|\nabla f(\theta^k)\|^2\right] \\ &\to \eta + \epsilon\end{aligned}$$

for $J \to \infty$ since $\gamma_k \to 0$ for $k \to \infty$ and $\sum_{k=0}^{\infty}\gamma_k = \infty$. This gives the contradiction to the previous result (3.8). □

With the assumption from Theorem 3.31 we can also show that the lim inf of the squared norm of the fine gradient converges almost surely to zero (Definition 2.9).

Theorem 3.32 (Almost Sure Convergence for Diminishing Step Sizes). *Using the assumptions from Theorem 3.31 we obtain*

$$\liminf_{k\to\infty}\|\nabla f(\theta^k)\|^2 \to 0 \quad \textit{almost surely.}$$

This result is based on similar ideas as in [127, Theorem 2.5].

Proof of Theorem 3.32. Define the sequences

$$\beta_k \coloneqq \gamma_k\|\nabla f(\theta^k)\|^2$$

and

$$\eta_k \coloneqq f(\theta^k) + \sum_{i=k}^{\infty}\gamma_i^2\tilde{C}$$

with $\tilde{C} := \max(\frac{LM^2}{2}P^2\rho^2\frac{\nu}{2\bar{L}_{mb}}\frac{1}{c^2\tau^2 d_1^2\kappa^2}, \frac{LM^2}{2})$. Conditioned on the current fine iterate θ^k, we have

$$\begin{aligned}\mathbb{E}[\eta_{k+1} \mid \theta^k] = \mathbb{E}[f(\theta^{k+1}) \mid \theta^k] + \sum_{i=k+1}^{\infty} \gamma_i^2 \tilde{C} &\leq f(\theta^k) + \sum_{i=k+1}^{\infty} \gamma_i^2 \tilde{C} - \gamma_k \|\nabla f(\theta^k)\|^2 + \gamma_k^2 \tilde{C} \\ &= \eta_k - \beta_k.\end{aligned}$$

Hence, we have

$$\mathbb{E}[\beta_k] \leq \mathbb{E}[\eta_k] - \mathbb{E}[\eta_{k+1}],$$

and

$$\mathbb{E}[\eta_{k+1} - f_{inf} \mid \theta^k] \leq \eta_k - f_{inf} - \beta_k \leq \eta_k - f_{inf},$$

since $\beta_k \geq 0$. Hence,

$$0 \leq \mathbb{E}[\eta_k - f_{inf}] \leq \eta_1 - f_{inf} < \infty,$$

since $f(\theta^0) < \infty$ and $\sum_{k=0}^{\infty} \gamma_k^2 < \infty$. Thus, $\eta_k - f_{inf}$ is a supermartingale (Definition 2.11). Theorem 2.12 states that a non-negative supermartingale converges almost surely to a limit and the expectation of the limit is smaller than or equal to the expectation of the starting random variable. Applying this to the supermartingale $\eta_k - f_{inf}$ yields

$$\lim_{k\to\infty} \mathbb{E}[\eta_k - f_{inf}] = \eta_\infty - f_{inf} = 0 \ a.s. \quad \text{and} \quad \mathbb{E}[\eta_\infty] \geq \mathbb{E}[\eta_1].$$

Now,

$$\begin{aligned}\mathbb{E}[\sum_{k=0}^{\infty} \beta_k] &= \sum_{k=0}^{\infty} \mathbb{E}[\beta_k] \\ &\leq \sum_{k=1}^{\infty} (\mathbb{E}[\eta_k] - \mathbb{E}[\eta_{k+1}]) < \infty.\end{aligned}$$

Hence, we have that

$$\sum_{k=0}^{\infty} \beta_k = \sum_{k=0}^{\infty} \gamma_k \|\nabla f(\theta^k)\|^2 < \infty \ a.s..$$

Since $\sum_{k=0}^{\infty} \gamma_k = \infty$ by assumption 3.30, the result follows. □

Corollary 3.33 (MG/Opt with GD as Iterative Method for Non-Convex Problems). *Let assumptions 3.16 to 3.18 and 3.21 to 3.23 hold. Additionally, let assumption 3.20 be satisfied for $M_G = 1, \bar{L}_{mb} = \bar{L}$. Then the iterates of the algorithm Algorithm 5 with full gradients instead of stochastic gradients satisfy*

$$\frac{1}{J+1}\sum_{k=0}^{J} \|\nabla f(\theta^k)\|^2 \leq \frac{f(\theta^0) - f_{inf}}{(J+1)\min[\alpha\frac{1}{2}, \alpha_{cgc}\frac{c\tau d_1 \kappa}{2}]} \to 0$$

for $J \to \infty$.

This result follows directly from Theorem 3.24 by setting the stochastic gradients equal to the full gradients.

Discussion on Convergence Results and Assumptions The presented results are restricted to the two-level case; nevertheless, we provide the first convergence result for MG/Opt with stochastic gradients in the non-convex setting for diminishing step sizes and the first almost sure convergence result thereof. Additionally, we leverage specific properties of the cgc updates to guarantee that the cgc updates e are descent directions for the fine objective function, instead of using techniques for general descent directions for the cgc updates, which allows us to get a more specific understanding of the cgc updates in this context.

The variant of MG/Opt proposed in [68] is the most closely related work to ours. Unlike our approach, it employs an angle condition

$$\frac{-(\nabla f(\theta^k), e)}{\|\nabla f(\theta^k)\| \|e\|} \geq \mu,$$

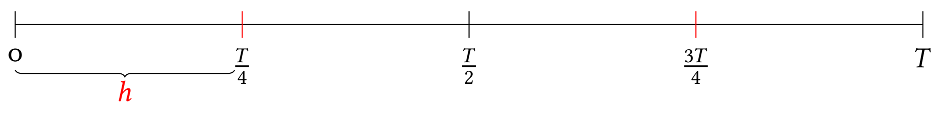
for some $\mu \in (0, 1]$, instead of the regularization (assumption 3.18) and the cgc descent (assumption 3.21), which we employ in order to guarantee convergence.

Some assumptions used are strong and therefore require discussion. While the most assumptions are standard, such as assumptions 3.16, 3.17 and 3.19, others are more specific to our analysis. Assumption 3.21 requires the coarse grid correction to yield a decrease in the coarse objective function in expectation, which is a reasonable assumption for optimization methods. Assumption 3.18 introduces a regularization term in the cgc objective function to ensure strong convexity, which is necessary for our analysis but poses a challenge in practical applications since there are scenarios where the regularization parameter λ must be chosen without knowledge of how large it must be. The work [68] first proposed the assumptions giving us control over the cgc iterates in assumptions 3.22 and 3.23. These assumptions are crucial for our analysis, but cannot be verified by the algorithm or proven in general.

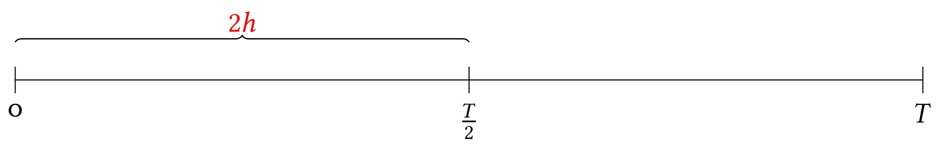
Note that the coarse objective function and the restriction operator for the parameters can be chosen very freely. The parameter κ can be selected freely to steer the method by influencing the frequency of cgc updates. A higher value of κ leads to less frequent cgc updates, while a lower value results in more frequent cgc updates. Also, the choice of the coarse objective function remains flexible in this framework, allowing for various strategies to be employed.

3.4. Stochastic MG/Opt for Neural Network Training

There are many possibilities to include multigrid optimization into neural network training. An overview of existing work was given in Section 3.2. Different versions of multilevel optimization methods can be used to train neural networks. Additionally, it differs by the choice of the fine and coarse objective functions, i.e. the hierarchical training problems and the choice of the restriction and



(a) Fine time discretization with time step h.



(b) Coarse time discretization with time step h.

Figure 3.3.: Time discretization on $[0, T]$ for ResNet training problems.

prolongation operators. In the following, we present different ideas for hierarchical training problems and suitable restriction and prolongation operators.

Example 3.34 (ResNets as Discretized ODEs). *When considering the architecture of a special ResNet described in* (2.7)*, we can see that the ResNet is interpreted as a discretization of a ODE. The continuous training problem is given by*

$$\begin{aligned}
&\min_{x(\cdot),u(\cdot),W_1,W_{L+1}} \frac{1}{D}\sum_{j=1}^{D} \mathcal{L}(W_{L+1}x_j(T;u(T)), y_{labelj}) \\
&\text{s.t. } x_j(t) = W^2(t)\sigma(W^1(t)x_j(t) + b(t)) && \forall j = 1,\dots,D,\ \forall t \in [0,T] \\
&\quad x_j(0) = W_1\tilde{x_{0,j}} && \forall j = 1,\dots,D.
\end{aligned}$$

Here, $u(t) := (W^1(t), W^2(t), b(t))$ are the time-dependent weights and biases of the ResNet. More specifically, the ResNets are an explicit Euler discretization (cf. [52]) of the ODE (2.8) *with fixed time step h. Given a network with $L-1$ residual layers of this form (assuming that $L-1 > 2$ is even), we can define a coarse ResNet with $(L-1)/2$ residual layers and doubling the time step to $2h$.*

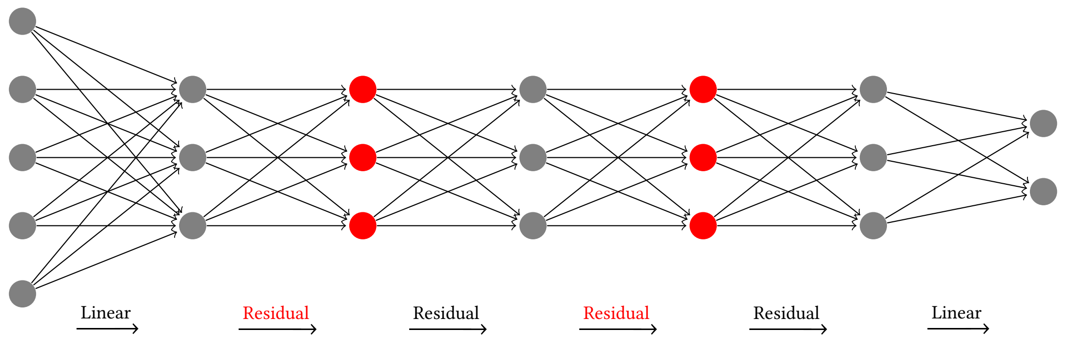
The fine problem is given by the training problem of a ResNet with $L-1$ residual layers and time step h:

$$\min_{W_1,u^2,\dots,u^L,W_{L+1}} \frac{1}{D}\sum_{j=1}^{D} \mathcal{L}(W_{L+1}x_{L,j}, y_{labelj}) \tag{3.9}$$

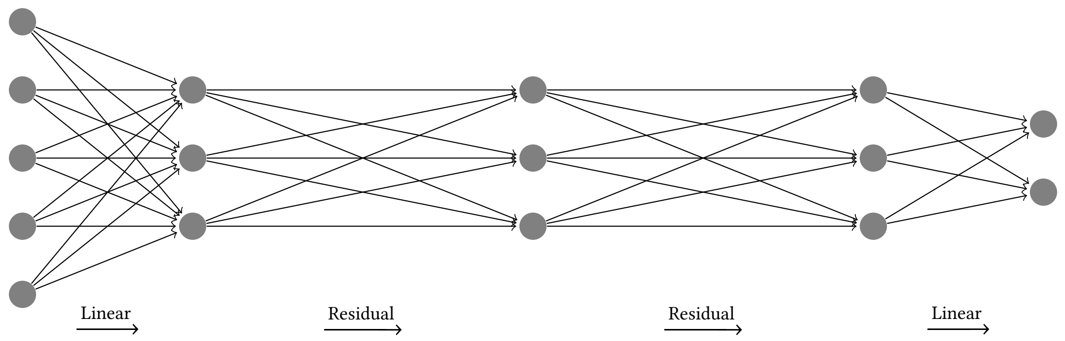
$$\text{s.t. } x_{i,j} = x_{i-1,j} + hW_i^2\sigma(W_i^1 x_{i-1,j} + b_i) \qquad \forall j = 1,\dots,D,\ \forall i = 2,\dots,L \tag{3.10}$$

$$x_{1,j} = W_1 x_{0,j} \qquad \forall j = 1,\dots,D. \tag{3.11}$$

Here, $u^i := (W_i^1, W_i^2, b_i)$ are the weights and biases of the $i-1$-th residual layer.



(a) Fine ResNet with L layers and discretization step size h. The additional residual layers in the fine ResNet are marked in red.



(b) Coarse ResNet with $\frac{L+1}{2}$ layers and discretization step size $2h$.

Figure 3.4.: Hierarchy by varying depth of ResNets. The additional residual layers in the fine ResNet are marked in red.

The coarse problem is given by the training problem of a ResNet with $\frac{L-1}{2}$ residual layers and time step $2h$:

$$\min_{W_1,u^2,\dots,u^{\frac{L+1}{2}},W_{\frac{L+3}{2}}} \frac{1}{D}\sum_{j=1}^{D} \mathcal{L}(W_{\frac{L+3}{2}} x_{\frac{L+1}{2},j}, y_{labelj}) \tag{3.12}$$

$$\text{s.t. } x_{i,j} = x_{i-1,j} + 2hW_i^2\sigma(W_i^1 x_{i-1,j} + b_i) \qquad \forall j = 1,\dots,D,\ \forall i = 2,\dots,\frac{L+1}{2} \tag{3.13}$$

$$x_{1,j} = W_1 x_{0,j} \qquad \forall j = 1,\dots,D. \tag{3.14}$$

Here, $u^i := (W_i^1, W_i^2, b_i)$ are the weights and biases of the $i-1$-th residual layer. Suitable restriction and prolongation operators for the parameters can be defined by standard coarsening strategies for time-dependent problems, e.g. injection or full-weighting for restriction and linear interpolation for prolongation. A schematic illustration of the fine and coarse ResNet architectures is given in Figure 3.4.

Example 3.35 (Constant Interpolation as Prolongation)**.** *Coming from the ODE interpretation of ResNets, a constant interpolation can be used as prolongation operator* (2.41)*. Then the matrix representing the* *prolongation operator is given by*

$$\mathcal{P}_p = \begin{pmatrix} \mathbb{I} & 0 & 0 & \cdots & 0 & 0 \\ \mathbb{I} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \mathbb{I} & 0 & \cdots & 0 & 0 \\ 0 & \mathbb{I} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \mathbb{I} & 0 \\ 0 & 0 & 0 & \cdots & 0 & \mathbb{I} \\ 0 & 0 & 0 & \cdots & 0 & \mathbb{I} \end{pmatrix},$$

where $\mathbb{I}$ is the identity matrix of the size of u^i, i.e. the parameters of one residual layer, which has the dimension $2n_{X_i}^2 + n_{X_i}$. The prolongation operator simply copies the parameters of one residual layer of the coarse ResNet to the two corresponding residual layers of the fine ResNet. The restriction operator is then given by the scaled transpose of the prolongation operator, i.e.

$$\mathcal{R}_g = \mathcal{R}_p = \frac{1}{2}\mathcal{P}_p^T.$$

The factor $\frac{1}{2}$ ensures that the restriction of a prolongated parameter vector is the same as the original parameter vector, i.e. $\mathcal{R}_p\mathcal{P}_p = \mathbb{I}$. These operators were used in [49, 50] to train ResNets with a multigrid optimization method.

Example 3.36 (Linear Interpolation as Prolongation). *Another possibility for the prolongation operator* (2.41) *is a linear interpolation. The matrix representing the prolongation operator is then given by*

$$\mathcal{P}_p = \begin{pmatrix} \mathbb{I} & 0 & 0 & \cdots & 0 & 0 \\ \frac{1}{2}\mathbb{I} & \frac{1}{2}\mathbb{I} & 0 & \cdots & 0 & 0 \\ 0 & \mathbb{I} & 0 & \cdots & 0 & 0 \\ 0 & \frac{1}{2}\mathbb{I} & \frac{1}{2}\mathbb{I} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \mathbb{I} & 0 \\ 0 & 0 & 0 & \cdots & \frac{1}{2}\mathbb{I} & \frac{1}{2}\mathbb{I} \\ 0 & 0 & 0 & \cdots & 0 & \mathbb{I} \end{pmatrix}.$$

The prolongation operator linearly interpolates the parameters of two neighboring residual layers of the coarse ResNet to the parameters of one residual layer of the fine ResNet. The restriction operator is then given by the scaled transpose of the prolongation operator, i.e.

$$\mathcal{R}_g = \mathcal{R}_p = \frac{1}{2}\mathcal{P}_p^T.$$

The factor $\frac{1}{2}$ is used to preserve the scale of the parameters.

Example 3.37 (Levels of Variance of Gradient Estimates). *The coarse problem is set to be the same as the fine problem and the parameter spaces of the fine and the coarse problem must be the same. The coarse level differs from the fine level by using a gradient estimate with a higher variance for the iteration* *updates, e.g. by using a smaller mini-batch size which leads to a cheaper but more noisy gradient estimate. This was proposed in [89] to train neural networks with a multigrid optimization method. Restriction and prolongation operators can be chosen as the identity as described in Example 3.38.*

Example 3.38 (Trivial Restriction and Prolongation)**.** *For varying levels of variance, the parameter space of the fine and coarse problem are the same. Hence, we can use the identity as restriction and prolongation operator* (2.41).

Example 3.39 (CNNs with Varying Resolution)**.** *When considering Convolutional Neural Networks (CNNs), we can define a hierarchy of networks and associated training problems by varying the resolution of the input data which then implies varying resolution of the feature maps in the hidden convolutional layers. Input data with a lower resolution is obtained by applying Max-Pooling* (2.11) *to the original high-resolution input data. The channel numbers of the feature maps and the number of layers can be kept the same for the different levels of resolution. Then, also the convolutional kernels are the same size for the different levels of resolution. Hence, the coarse CNN has no fewer parameters than the fine CNN, but the data and feature maps have a lower resolution leading to a smaller computational cost per forward and backward pass. Because of the same dimensions of the coarse and fine parameter space, the restriction and prolongation operators can be chosen as the identity as described in Example 3.38.*

There are further possibilities to define hierarchies for CNNs training problems based on varying resolution, e.g. by varying the size of the convolutional kernels. As the size of the convolutional kernels directly influences the number of parameters of the network, this leads to different dimensions of the parameter spaces of the fine and coarse problem and hence non-trivial restriction and prolongation operators must be defined. An example for this approach is given in [68].

3.5. Discussion

In this chapter, we proposed new MG/Opt variants for stochastic optimization and discussed their convergence properties and limitations. We started with analyzing the smoothing properties of SGD in the strongly convex quadratic case in Section 3.3.1. Based on these findings, we develop suitable prolongation and restriction operators and a variant of MG/Opt along with convergence theory for the stochastic quadratic setting. This theoretical analysis requires strong assumptions on the prolongation and restriction operators as well as on the cgcs to ensure convergence. We discuss possibilities to weaken the assumptions along with the adapted theoretical results. Subsequently, we consider the setting with non-convex objective functions and presented a stochastic two-level MG/Opt variant and convergence results tailored to this setup in Section 3.3.2. This setting is particularly relevant for neural network training applications. We show proofs of convergence of the new method under standard assumptions for stochastic gradients and additional assumptions specific to the MG/Opt framework for fixed and diminishing step sizes. We show convergence for diminishing step sizes in expectation and almost sure convergence, which are, to the best of our knowledge, the first convergence result for MG/Opt of this form. Finally, we discussed various possibilities to define hierarchies of neural network training problems and suitable restriction and prolongation operators in Section 3.4.

In the context of neural network training, i.e. stochastic optimization of non-convex objective functions, the behavior of optimization methods is less clear than in the convex deterministic case due to the highly complex loss landscape. Especially, the smoothing properties of different optimization methods like SGD in this setting are not known yet. Hence, examining the effects of different hierarchies and restriction and prolongation operators on the training performance of multigrid optimization methods both theoretically and numerically is an important topic for future research.

Numerical experiments are omitted here, since [68] observes that for both hierarchies, either by varying depth or by varying resolution, MG/Opt, more precisely the cgc updates do not improve the training performance w.r.t. loss and test accuracy over computational time. They propose to use the original coarse problem instead of the cgc problem as constructed in (2.43) to get an accelerated training process using a coarse level. In this setting, they observe speed-ups in training time compared to standard training without multigrid hierarchy, which is most pronounced for the hierarchy by varying depth. Leaving out the correction term in the cgc (2.43) leads to a different algorithm than MG/Opt and hence convergence properties of the MG/Opt method do not hold anymore. While this modification is not explored theoretically by [68], it seems that the training problem of the coarse network and the fine network are connected via some kind of natural coherence of the respective gradients. The hierarchy by varying the variance of the gradient estimates as described in Example 3.37 which was proposed in [89], was not applied to complicated non-convex learning problems such as neural networks yet, but only to simple convex learning problems.

Finally, we would like to point out that the above theory for the non-convex stochastic setting applies to a broad range of hierarchical training problems, especially coarse objective functions f_H. Additional to the hierarchies presented in Section 3.4, it is possible to pose an auxiliary problem involving the trainable parameters as the coarse objective function, which is different from the fine objective function, e.g. by adding further regularization terms, and it is not necessary to use a neural network training problem on the coarse level. Future research is needed to explore the potential of such approaches both theoretically and numerically.

An alternative way to exploit the multilevel structure of neural networks naturally obtained by their layer-wise composition is to build up and expand a small baseline network by adding layers during training, which we examine in the next chapter. This idea is inspired by the process called *nested iterations*, where one starts to iteratively solve a PDE on a coarse grid and then prolongates the solution to a finer grid to use it as an initial guess for further iterations. We will use a well-known technique from nonlinear constrained optimization, called *sensitivity analysis*, to identify suitable layers to be added to the network during training.

4. Sensitivity-based Layer Insertion (SensLI)

This chapter is an extended version of the article [57] with the title "Sensitivity-based Layer Insertion for Neural Networks". It introduces the Sensitivity-based Layer Insertion (SensLI) method for adaptively inserting new layers into neural networks during training. The chapter differs from the article since it elaborates on the theory of the sensitivity analysis behind the SensLI method in more depth in Section 4.3.1 than the original article, and extends the method to layer widening in Section 4.4. Finally, we discuss advantages and disadvantages of the SensLI method among other findings of this chapter in Section 4.5.

4.1. Introduction

Selecting an appropriate neural network architecture for supervised learning tasks remains a central challenge in machine learning. In practice, the architecture (determined by the number of layers, their types, and widths) is often chosen based on experience, intuition, or trial and error. This ad-hoc approach can lead to suboptimal models and inefficient training. To address this challenge, the field of Neural Architecture Search (NAS) has emerged, aiming to automate the process of finding suitable architectures prior to training. However, NAS algorithms are typically computationally expensive and may not scale well to large datasets or complex tasks. A survey of existing NAS methods can be found e.g. in [29].

As an alternative to exhaustive architecture search, adaptive methods which modify the network structure during training have gained attention. In this chapter, we present a general framework for Sensitivity-based Layer Insertion (SensLI), a depth-adaptive approach that can be viewed as an automated hyperparameter search for the network's depth. SensLI leverages sensitivity analysis of the loss function with respect to virtual weights associated with all potential new layers, enabling informed decisions about where and when to insert new layers. This method requires only moderate computational effort and is applicable to a wide range of architectures, including fully-connected Feedforward Neural Networks (FNNs), Residual Neural Networks (ResNets), and Convolutional Neural Networks (CNNs).

The core idea behind SensLI is inspired by adaptive grid refinement in numerical methods for Partial Differential Equations. In adaptive grid refinement, one starts with a coarse grid and refines it in regions where the solution lacks accuracy, guided by indicators that predict the effectiveness of refinement. New grid points are typically initialized by interpolating surrounding points, ensuring the solution process is not disrupted. This adaptive strategy focuses computational resources where they are most needed, leading to more efficient solutions compared to uniformly fine grids.

Transferring this concept to neural networks, SensLI interprets each layer as analogous to a grid point. In FNNs, layers can be seen as one-dimensional grid points, while in ResNets, layers correspond to time grid points in a discretized ODE solved by explicit Euler methods. SensLI inserts new layers in regions of high network activity, as determined by sensitivity analysis, thereby adaptively increasing the network's capacity where it is most beneficial.

A key distinction between neural networks and PDEs is that, in the latter, grid points are the primary quantities of interest and can be directly manipulated. In neural networks, however, layers are not directly changeable; instead, the trainable parameters, weights and biases, define the transitions between layers. Hence, the direct application of interpolation techniques used in PDEs is not straightforward and requires modification.

4.2. Related Work

Finding a suitable architecture for a neural network for data and the associated parameter estimation problem representing a supervised learning task is challenging. Neural Architecture Search is a field of research that aims to automate the search for a suitable architecture before the actual training starts. These methods are often very computationally expensive and hence not generally applicable in practice to larger problems. An alternative idea to NAS is to start with a baseline neural network and then modify the architecture during training. When the baseline network is a large network and the architecture during training is modified by removing neurons or layers, this is often referred to as pruning. Contrary, when the baseline network is a small network and its architecture is modified by adding neurons or layers during training, this is often referred to as growing neural networks or constructive neural networks. Of course, it is also possible to combine the two strategies by both adding and removing neurons or layers during training.

In the following, we focus on methods for expanding the neural network architecture during training, which main ideas are not new, but date back several decades. Earliest methods include a method called Cascade Correlation proposed in [31] which builds a uniquely structured architecture during training and expands it, and the Restricted Coulomb Energy (RCE) network proposed in [65]. The authors from [120] propose a new modifiable network architecture and its expansion called SONN (Self-Organizing Neural Network). Another popular early method for widening neural networks is dynamic node creation, see [3]. Other methods include [54], which adds hidden units adaptively during training, [136] which proposes a node splitting algorithm, and [53] which proposes meiosis networks, which grow new nodes based on the uncertainty of the trainable parameters.

For modern standard neural network architectures and training methods, there are two practical ways to expand a given neural network architecture. It is possible to expand a neural network by inserting new layers into the network. These layers can be of all kinds, e.g., fully-connected layers, convolutional layers or residual layers. Alternatively, it is possible to expand a neural network by adding new neurons to already existing layers, which is often referred to as layer widening. This includes the increase of the number of channels in a convolutional layer as well. Layer widening is a more popular strategy compared to layer insertion, due to its simpler structure and lower computational cost. Expanding a neural network architecture in those ways naturally leads to additional trainable parameters in the expanded network architecture, where the exact number of new parameters depends on the type of layer and the number of neurons added.

Methods in the literature vary in terms of how, where, and when additional neurons are inserted into neural network architectures. They also differ in terms of their computational cost and the types of architectures they can be applied to. We provide a tabular overview of relevant literature in Table 4.1.

Net2Net [16] and NetMorph [129] address the problem of effective initialization of newly added neurons following layer insertion. The Net2Net method focuses on maintaining the function represented by the network by inserting a neutral layer initialized as an identity mapping, while the second method NetMorph emphasizes a more flexible approach to layer insertion and initialization using convolutions. The method called Gradmax, which was introduced in [30], focuses on initialization techniques for layer widening that utilize singular value decomposition, with an emphasis on optimizing training dynamics rather than achieving an immediate reduction in the objective function. Introduced in [130], the method AutoGrow prioritizes automation of the network expansion process rather than speeding up training, and systematically evaluates various empirical strategies for initialization and triggering layer insertion. The splitting steepest descent method from [134] and its continuation, the Firefly architecture, introduced in [133], both present approaches that alternate between optimizing the networks parameters and modifying the architecture by widening layers or inserting residual layers (only for Firefly) during training. NeST, introduced in [21], presents a layer widening approach that combines gradient-based neuron growth with magnitude-based pruning. The MorphNet strategy which was proposed in [43], introduces resource constraints while it adaptively shrinks and expands the network architecture during training. Finally, the strategies introduced in [88] utilize several initialization strategies like orthogonal weight initialization together with specific triggering mechanisms also based on activations or gradients to widen layers during training.

While most of the above methods are developed by researchers from the machine learning community and naturally are mainly heuristic in nature, there are also some methods developed by researchers which focus more on existing mathematical and analytical foundations of invented methods. The method proposed in [124] utilizes information from the objective function to widen layers from a functional analytical background employing functional derivatives, while [92] employs the natural gradient used classical in information geometry to expand the neural network. AdaNet, introduced in [19], simultaneously learns both network structures and weights, providing data-driven theoretical guarantees. For ResNets, automated layer insertion methods that exploit the neural ODE perspective (cf. [49]) have been explored, as in [15] and [25]. The very recent work [60] proposes a layer-wise adaptive construction method for neural ODEs from an optimal control perspective, and applies it classification tasks with well-known data sets.

Table 4.1.: Comparison of selected methods for expanding neural networks. Indicating whether layer insertion or widening was considered, whether the question on how, where and when to insert were answered (cf. Section 4.3.2), whether the network expansion was executed during training and which architectures were examined out of FNN, ResNet and CNN.

	Layer					**During**	**Architecture**		
Method	**Insertion**	**Widening**	**How?**	**Where?**	**When?**	**training?**	**FNN**	**ResNet**	**CNN**
Net2Net [16]	✓	✓	✓	×	×	×	✓	×	✓
NetMorph [129]	✓	✓	✓	×	×	×	✓	×	✓
Firefly [133]	✓	✓	✓	✓	×	✓	✓	✓	✓
Autogrow [130]	✓	×	✓	✓	✓	✓	×	×	✓
SENN [92]	✓	✓	✓	✓	✓	✓	✓	✓	×
ConvSENN [2]	✓	✓	✓	✓	✓	✓	×	×	✓
MorphNet [43]	×	✓	×	×	✓	×	✓	✓	✓
NeST [21]	×	✓	✓	✓	✓	✓	✓	✓	✓
Splitting [134]	×	✓	✓	✓	✓	×	✓	×	×
GradMax [30]	×	✓	✓	×	×	✓	✓	×	✓
[88]	×	✓	✓	✓	✓	✓	✓	×	×
[124]	×	✓	✓	✓	✓	✓	✓	✓	✓
SensLI	✓	(✓)	✓	✓	✓	✓	✓	✓	✓

4.3. Sensitivity-Based Layer Insertion

We first show the necessary sensitivity analysis which is needed for the SensLI method later on in Section 4.3.1. Then we describe the overall SensLI method in Section 4.3.2 and give numerical results in Section 4.3.3. We close with a section dedicated to the extension of the method to layer widening in Section 4.4 and concluding remarks in Section 4.5.

4.3.1. Sensitivity Analysis for SensLI Algorithm

For the following analysis we assume that we have a baseline neural network g from a neural network architecture (i.e. FNN, ResNet, CNN) where it is possible to insert an identity layer at an arbitrary position in the baseline neural network by selecting the initialization values of the weights and biases of the new layer in a suitable way. In Section 4.3.2 we explain how this can be done for FNNs, ResNets and CNNs in detail. For the sake of convenience of the subsequent discussion, we treat the parameters for now as column vectors, although in reality they consist of matrices and vectors. The extended neural network with an additional layer is denoted by g_{ext}. We can keep the new layers of the extended neural network to remain inactive by fixing the weights and biases of the new layers to their initialized values. Adding constraints to the training problem of the extended neural network allows the baseline network to be transparently embedded into its extended version of the network.

Due to this fact, training the baseline neural network via the unconstrained optimization problem

$$\min_{\theta_{\text{base}} \in \Theta} \hat{f}_{\text{base}}(\theta_{\text{base}}) \quad \text{with} \quad \hat{f}_{\text{base}}(\theta_{\text{base}}) = \frac{1}{D} \sum_{j=1}^{D} \mathcal{L}(g(\theta_{\text{base}}, x_{0,j}), y_{\text{label}j}) \tag{4.1}$$

can be imitated by training the extended network via the (equality) constrained optimization problem

$$\min_{\theta_{\text{ext}} \in \Theta_{\text{ext}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \quad \text{with} \quad \hat{f}_{\text{ext}}(\theta_{\text{ext}}) = \frac{1}{D} \sum_{j=1}^{D} \mathcal{L}(g_{\text{ext}}(\theta_{\text{ext}}, x_{0,j}), y_{\text{label}j}) \tag{4.2}$$

$$\text{s.t. } c(\theta_{\text{ext}}) = 0, \tag{4.3}$$

where the constraints represent the initialization conditions of the new parameters,

$$c(\theta_{\text{ext}}) \coloneqq M\theta_{\text{ext}} - m. \tag{4.4}$$

The constraints have an affine linear form with the matrix M and vector m chosen such that the parameters of the new layers are fixed to their initialized values. The matrix M is of the form

$$M = \begin{pmatrix} 0 & \text{id} \end{pmatrix},$$

where the identity matrix id selects the parameters of the new layers and the zero matrix 0 selects the parameters of the baseline neural network for the extended parameters

$$\theta_{\text{ext}} = \begin{pmatrix} \theta_{\text{base}} \\ \theta_{\text{new}} \end{pmatrix}.$$

Hence, the constraints affect only the new parameters. The vector m of the dimension of the new parameters θ_{new} contains the (vectorized) initialized values of the new parameters. With the constraints satisfied, the objective functions of the baseline and extended neural network are equal, i.e. $f_{\text{ext}}(\theta_{\text{ext}}) = f_{\text{base}}(\theta_{\text{base}})$. Running a training algorithm on the baseline network (4.1), can be viewed as running the same algorithm on the extended network, as long as the additional weights are constrained to their respective values.

Now we aim to apply well-established techniques from sensitivity analysis as described in Section 2.3.3 to the constrained optimization problem (4.2) in order to get information about the influence of the new layers on the training process by considering a version of the constrained optimization problem as in (2.39), which incorporates perturbations in the constraints.

The perturbed optimization problem reads

$$\begin{aligned} \text{Minimize} \quad & \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \\ \text{s.t.} \quad & M\theta_{\text{ext}} - m = \epsilon\Delta \end{aligned} \tag{4.5}$$

with some perturbation vector Δ and perturbation parameter $\epsilon > 0$.

Remark 4.1. *The perturbation vector Δ can be interpreted as a direction in which the new layers are allowed to deviate from their initialized values, i.e. an update direction such as the negative gradient* *for the new parameters. The perturbation parameter ϵ then scales this direction as a step size. It is also possible to perturb the constraints with different terms as $\epsilon\Delta$.*

The classical setting of sensitivity analysis in nonlinear programming (cf. Section 2.3.3) examines the sensitivity of the objective function value to perturbations in the constraints at a local minimizer or at least a KKT point (see (2.35) and (2.36)) of the unperturbed problem, where $\epsilon = 0$. The backbone of sensitivity analysis is the Implicit Function Theorem (IFT) applied to the KKT conditions of the perturbed problem.

One crucial difference in our setting is that θ_{base} is generally not a minimizer for the baseline training problem (4.1), since we do not train the neural network to full convergence before inserting a new layer and want to be able to apply the sensitivity analysis at any point during training. Hence, we rather assume to be in a feasible point, which is an approximation of a local minimizer.

Because the actual training algorithm runs on the baseline network, it does not employ the additional, virtual parameters θ_{new}. These parameters only get inserted when it is time to extend the network, and they are getting chosen to satisfy the constraint $M\theta_{\text{ext}} - m = 0$ at insertion time exactly, which makes θ_{ext} feasible.

Hence, we need to extend the classical sensitivity analysis to the setting where a residual $r = \nabla \hat{f}_{\text{base}}(\theta_{\text{base}})$ remains at the time of the sensitivity analysis. We propose the following sensitivity theorem for this setting, which extends the classical sensitivity result in Theorem 2.33.

Theorem 4.2. *Consider the family of perturbed equality constrained optimization problems* (2.39) *with the specific constraints described in* (4.4)*. Let $(\hat{\theta}_{ext}, \hat{\lambda}) \in \Theta_{ext} \times \mathbb{R}^{n_{new}}$ be a point satisfying*

- *$c(\hat{\theta}_{ext}) = 0$ (feasibility),*
- *$\nabla^2_{\theta_{ext}} \mathcal{L}(\hat{\theta}_{ext}, \hat{\lambda}, 0)$ is strictly positive definite (or invertible) on the cone $T_c(\hat{\theta}_{ext})$, (as defined in* (2.38) *and* (2.40)*).*

Then there exist open neighborhoods U of $\epsilon = 0$ and V of $(\hat{\theta}_{ext}, \hat{\lambda})$ and a unique continuously differentiable function

$$\begin{pmatrix} \theta_{ext} \\ \lambda \end{pmatrix} : U \to V,$$

where the components of the functions w.r.t. θ_{ext} and λ at point $\epsilon \in U$ are taken as $\theta_{ext}(\epsilon)$ and $\lambda(\epsilon)$. The function satisfies

- *$\binom{\theta_{ext}}{\lambda}(0) = \binom{\hat{\theta}_{ext}}{\hat{\lambda}}$,*
- *$c_\epsilon(\theta_{ext}(\epsilon)) = 0 \ \forall \epsilon \in U$,*
- *$\nabla_{\theta_{ext}} \mathcal{L}(\theta_{ext}(\epsilon), \lambda(\epsilon), \epsilon) = s \ \forall \epsilon \in U$ for $s := \nabla_{\theta_{ext}} \mathcal{L}(\hat{\theta}_{ext}, \hat{\lambda}, 0)$,*
- *$\nabla^2_{\theta_{ext}} \mathcal{L}(\theta_{ext}(\epsilon), \lambda(\epsilon), \epsilon)$ is strictly positive definite (or invertible) on $T_{c_\epsilon}(\theta_{ext}(\epsilon)) \ \forall \epsilon \in U$.*

Further, it holds that

$$\hat{f}_{ext}(\theta_{ext}(\epsilon)) = \hat{f}_{ext}(\hat{\theta}_{ext}) - \epsilon\hat{\lambda}^T\Delta + \epsilon s^T \frac{\partial\theta_{ext}}{\partial\epsilon}(0) + o(|\epsilon|). \tag{4.6}$$

This result is based on the IFT applied to shifted KKT conditions of the perturbed problem (4.5) at $\epsilon = 0$. The Taylor expansion of the objective function value at the point $\hat{\theta}_{\text{ext}}$ in (4.6) differs from the classical counterpart (Theorem 2.33) by the additional term $\epsilon s^T \frac{\partial\theta_{\text{ext}}}{\partial\epsilon}(0)$ in (4.6), which appears due to the fact that $\hat{\theta}_{\text{ext}}$ is not a KKT point of the unperturbed problem.

Remark 4.3.
- *The residual $r = \nabla\hat{f}_{base}(\theta_{base})$ of the baseline training problem appears in the sensitivity result via the vector $s = \binom{r}{0}$, because the sensitivity analysis is applied to a modified objective function $\hat{f}_{ext}(\theta_{ext}) - s^{\mathsf{T}}\theta_{ext}$.*
- *With this approach, $s = \binom{r}{0}$ has only potentially non-zero entries which correspond to the baseline network parameters θ_{base}, hence s contains no information about the new layers. Consequently, the term containing s in (4.6) is not relevant for the selection of the new layer.*
- *Sometimes the Lagrange multipliers $\hat{\lambda}$ are also referred to as shadow prices in the literature, since they can be interpreted as the price of relaxing a constraint.*
- *We can interpret the second term as a competing term to the shadow prices which displays how much the value of the objective function can be improved by iterating further on the baseline network, whereas the sensitivities display how we can decrease the value of the objective function by relaxing the constraints on the new layers, hence allowing them to be updated.*
- *In theory, it would be possible to use the second term as an indicator on whether to stop training, but it is expensive which makes it infeasible in practice. The term $\frac{\partial\theta_{ext}}{\partial\epsilon}(0)$ can be computed by solving a system of linear equations given by the IFT. Additionally, we would need to compute the Hessian of f_{ext}, which is too expensive and can not be computed for arbitrarily large problems.*

Proof. In order to apply sensitivity analysis, the current point $\hat{\theta}_{\text{ext}}$ is treated as a local solution, or at least a stationary point, of a problem with modified objective

$$\hat{f}_{\text{base}}(\theta_{\text{base}}) - r^{\mathsf{T}}\theta_{\text{base}} = \hat{f}_{\text{ext}}(\theta_{\text{ext}}) - s^{\mathsf{T}}\theta_{\text{ext}}$$

with $s = \binom{r}{0}$. With the modified objective, $\hat{\theta}_{\text{ext}}$ becomes a stationary point of the extended problem (4.5) (at $\epsilon = 0$). Consequently, $\hat{\theta}_{\text{ext}}$ is a KKT point with uniquely defined Lagrange multiplier vector $\hat{\lambda}$ satisfying

$$\nabla\hat{f}_{\text{ext}}(\theta_{\text{ext}}) - \left[\begin{smallmatrix} r \\ 0 \end{smallmatrix}\right] + M^{\mathsf{T}}\lambda = 0, \tag{4.7a}$$

$$M\theta_{\text{ext}} - m = 0. \tag{4.7b}$$

Taking into account the partitioning $\theta_{\text{ext}} = \binom{\theta_{\text{base}}}{\theta_{\text{new}}}$ and the structure of $M = [0, \text{id}]$, we can write (4.7) as

$$\nabla_{\theta_{\text{base}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) - r = 0, \tag{4.8a}$$

$$\nabla_{\theta_{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) + \lambda = 0, \tag{4.8b}$$

$$\theta_{\text{new}} - m = 0. \tag{4.8c}$$

(4.8b) implies that the Lagrange multipliers are given by

$$\hat{\lambda} = -\nabla_{\theta_{\text{new}}} \hat{f}_{\text{ext}}(\hat{\theta}_{\text{ext}}).$$

Since the constraints are linear and the constraint Jacobian M has linearly independent rows and hence full row rank, the LICQ holds. Now we can apply Theorem 2.33 and get the following results. There exist open neighborhoods U of $\epsilon = 0$ and V of $(\hat{\theta}_{\text{ext}}, \hat{\lambda})$ and a unique continuously differentiable function

$$\begin{pmatrix} \theta_{\text{ext}} \\ \lambda \end{pmatrix} : U \to V,$$

where the components of the functions w.r.t. θ_{ext} and λ at point $\epsilon \in U$ are taken as $\theta_{\text{ext}}(\epsilon)$ and $\lambda(\epsilon)$. The function satisfies

- $\binom{\theta_{\text{ext}}}{\lambda}(0) = \binom{\hat{\theta}_{\text{ext}}}{\hat{\lambda}}$,
- $c_\epsilon(\theta_{\text{ext}}(\epsilon)) = 0 \ \forall \epsilon \in U$,
- $\nabla_{\theta_{\text{ext}}} \mathcal{L}(\theta_{\text{ext}}(\epsilon), \lambda(\epsilon), \epsilon) = s \ \forall \epsilon \in U$ for $s := \nabla_{\theta_{\text{ext}}} \mathcal{L}(\hat{\theta}_{\text{ext}}, \hat{\lambda}, 0)$,
- $\nabla^2_{\theta_{\text{ext}}} \mathcal{L}(\theta_{\text{ext}}(\epsilon), \lambda(\epsilon), \epsilon)$ is strictly positive definite (or invertible) on $T_{c_\epsilon}(\theta_{\text{ext}}(\epsilon)) \ \forall \epsilon \in U$.

Since $\theta_{\text{ext}}(\epsilon)$ is continuously differentiable, we can apply the chain rule to get

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\epsilon} f(\theta_{\text{ext}}(\epsilon)) &= \nabla_{\theta_{\text{ext}}} f(\theta_{\text{ext}}(\epsilon))^T \frac{\partial \theta_{\text{ext}}}{\partial \epsilon}(\epsilon) \\
&= (-c'(\theta_{\text{ext}}(\epsilon))^T \lambda(\epsilon) + s)^T \frac{\partial \theta_{\text{ext}}}{\partial \epsilon}(\epsilon) \\
&= -(c'(\theta_{\text{ext}}(\epsilon))^T \lambda(\epsilon))^T \frac{\partial \theta_{\text{ext}}}{\partial \epsilon}(\epsilon) + s^T \frac{\partial \theta_{\text{ext}}}{\partial \epsilon}(\epsilon) \\
&= -\lambda(\epsilon)^\top \Delta + s^T \frac{\partial \theta_{\text{ext}}}{\partial \epsilon}(\epsilon),
\end{aligned}$$

which shows that

$$\left. \frac{\mathrm{d}}{\mathrm{d}\epsilon} f(\theta_{\text{ext}}(\epsilon)) \right|_{\epsilon=0} = -\hat{\lambda}^\top \Delta + s^T \frac{\partial \theta_{\text{ext}}}{\partial \epsilon}(0).$$

The Taylor expansion of $f_{\text{ext}}(\theta_{\text{ext}}(\epsilon))$ around $\epsilon = 0$ then yields

$$\hat{f}_{\text{ext}}(\theta_{\text{ext}}(\epsilon)) = \hat{f}_{\text{ext}}(\hat{\theta}_{\text{ext}}) - \epsilon \hat{\lambda}^T \Delta + \epsilon s^T \frac{\partial \theta_{\text{ext}}}{\partial \epsilon}(0) + o(|\epsilon|),$$

where $\frac{\partial \theta_{\text{ext}}}{d\epsilon}(0) := (\frac{\partial \theta_{\text{ext},i}}{d\epsilon}(0))_{i=1,..,\dim(\Theta_{\text{ext}})}$, which concludes the proof. □

An alternative interpretation is that $\hat{\lambda}$ as proposed above pushes the derivative of the Lagrange function to zero on the orthogonal complement of the (linearized) equality constraints. This is possible not only in KKT points but at arbitrary points. If additionally the derivatives in the kernel of the linearized equality constraints are zero, we are in a KKT point.

We can infer a more specific result from Theorem 4.2 which is specifically tailored to the sensitivity analysis needed for the SensLI method.

Corollary 4.4 (Sensitivities as Comparison for Layer Effectiveness)**.** *Let the current parameters in training $\theta_{ext} \in \Theta_{ext}$ satisfy*

$\nabla^2_{\theta_{ext}} \hat{f}_{ext}(\theta_{ext})$ *exists and is strictly positive definite (or invertible) on* $ker(M) = \{\theta_{ext} \in \Theta_{ext} \mid \theta_{ext} = \begin{pmatrix} \theta_{base} \\ 0 \end{pmatrix}\}$.

Then, the first-order change of the objective function $\hat{f}_{ext}$ w.r.t. a perturbation of the constraints in direction $\epsilon\Delta$ is given by

$$\frac{d}{d\epsilon} f_{ext}(\theta_{ext}(\epsilon))\big|_{\epsilon=0} = \nabla_{\theta_{new}} \hat{f}_{ext}(\hat{\theta}_{ext})^T \Delta + s^T \frac{\partial \theta_{ext}}{\partial \epsilon}(0),$$

where s is defined as in Theorem 4.2 and hence

$$\hat{f}_{ext}(\theta_{ext}(\epsilon)) = \hat{f}_{ext}(\hat{\theta}_{ext}) + \epsilon \nabla_{\theta_{new}} \hat{f}_{ext}(\hat{\theta}_{ext})^T \Delta + \epsilon s^T \frac{\partial \theta_{ext}}{\partial \epsilon}(0) + o(|\epsilon|). \tag{4.9}$$

Proof. In the following, we check that the assumptions of Theorem 4.2 are satisfied by the specific constraints describing the layer insertion in (4.2).

- $c(\hat{\theta}_{\text{ext}}) = 0$ (feasibility) is fulfilled by construction of $\hat{\theta}_{\text{ext}}$ out of the baseline parameters θ_{base} and the initialized new parameters θ_{new}.
- The critical cone is given by $T_c(\hat{\theta}_{\text{ext}}) = \ker(M)$.
- $\nabla^2_{\theta_{\text{ext}}} \mathcal{L}(\hat{\theta}_{\text{ext}}, \hat{\lambda}, 0) = \nabla^2_{\theta_{\text{ext}}} \hat{f}_{\text{ext}}(\hat{\theta}_{\text{ext}})$ because of the linearity of the constraints and by assumption $\nabla^2_{\theta_{\text{ext}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}})$ is strictly positive definite or invertible on $\ker(M)$.

Inserting the explicit form of the Lagrange multipliers $\hat{\lambda} = -\nabla_{\theta_{\text{new}}} \hat{f}_{\text{ext}}(\hat{\theta}_{\text{ext}})$ from (4.8b) into the result of Theorem 4.2 concludes the proof. □

Remark 4.5. *The result above covers only the setting where a twice differentiable network function f_{ext} is used, which relies on the differentiability of the activation function. Further, the theory holds only at points θ_{base} with a strictly positive definite Hessian on the critical cone* (2.38)*. In practice however, we can use this theory as a heuristic also for ReLU activation functions and at arbitrary points.*

Corollary 4.4 states that the first-order change of the objective function w.r.t. a relaxation of the constraints associated to an inserted layer in the direction of $\epsilon\Delta$ can be predicted by the inner product of the corresponding Lagrange multipliers $\hat{\lambda}$ and the perturbation direction $\epsilon\Delta$. An update of the new weight matrix parameters in the direction of the negative gradient for GD leads to the quantity

$$-\epsilon\hat{\lambda}^T\Delta = -\epsilon\nabla_{\theta_{\text{new}}}\hat{f}_{\text{ext}}(\hat{\theta}_{\text{ext}})^{\mathsf{T}}\nabla_{\theta_{\text{new}}}\hat{f}_{\text{ext}}(\hat{\theta}_{\text{ext}}) = -\epsilon\|\nabla_{\theta_{\text{new}}}\hat{f}_{\text{ext}}(\hat{\theta}_{\text{ext}})\|^2,$$

which is the first-order predicted decrease in the objective function value, when allowing the new layer to be updated in the direction of the negative gradient instead of forcing it to remain inactive.

Consequently, the Euclidean norm

$$\|\nabla_{\theta_{\text{new}}}\hat{f}_{\text{ext}}(\theta_{\text{ext}})\|^2 \tag{4.10}$$

provides a notion of merit of inserting the layer with parameters θ_{new}. Hence, the most effective layer according to the Taylor expansion are the ones with the largest norm of the associated sensitivities and gets chosen to be inserted into the network. However, this prediction makes only a local statement; the new chosen layer does not have to be the one which leads globally to the fastest convergence. It is also possible to scale the sensitivities with the number of parameters in the layer to avoid a bias towards layers with many parameters.

The Taylor expansion (4.9) makes an averaged statement of the influence of the perturbation of all equality constraints. In order to use the Taylor expansion to select a new layer, we consider the Lagrange multipliers associated to each layer separately and averaged within each layer. Often, we only take the weight of the layer into account, neglecting the bias term parameters. It is also possible to examine the Lagrange multipliers associated to each parameter in a weight matrix individually, but allowing only selected parameters in a weight matrix to change during training is not efficient w.r.t. computational effort and memory consumption because of the backpropagation and structure of the network.

Remark 4.6. *The discussion of a KKT point would not be sufficient for this method. In practice, an exact minimizer is not found and additionally, in the case for FNNs it would cause problems, because of the backtracking structure of the network. If $\hat{\theta}_{ext}$ is part of a KKT point, then it follows that $\hat{\lambda} = 0$ since*

$$\nabla_{W_i}\hat{f}_{ext}(\theta_{ext}) = \nabla_{W_{i+1}}\hat{f}_{ext}(\theta_{ext})\ W_{i+1}^T\ \sigma'(y_i) = 0,$$
$$\nabla_{b_i}\hat{f}_{ext}(\theta_{ext}) = \nabla_{b_{i+1}}\hat{f}_{ext}(\theta_{ext})\ W_{i+1}^T\ \sigma'(y_i) = 0,$$

for the new layer parameters W_i, b_i (weights and biases of layer i). This is true independent of the position of the new layer in the network and the propagated training data. Hence, the network is not able to profit from the new layer, since the gradient is zero anyway. This problem does not occur for every neural network architecture, e.g. for ResNet architectures.

4.3.2. Method

In order to develop a method to expand a neural network either by inserting layers or only neurons into an already existing layer during a training process, three questions arise:

- **Where** to insert the new layer in the network?

- **How** to initialize the parameters of the new layer?
- **When** to insert the new layer during training?

The method we propose, SensLI, answers the first two questions from an optimization perspective. The sensitivity analysis described above is used to answer the first question, i.e. where to insert the new layer. We answer the third question only by a heuristic based on the computed sensitivities. This is not surprising, since even considering a standard training process of a neural network from a mathematical perspective, it is not clear when to stop the training process, because of the non-convexity and stochasticity of the process, see e.g. [139].

Now we describe the mechanics of the SensLI method [57] in some more detail.

Initialization:

Suppose that we are at some arbitrary point during training of a baseline neural network with parameters θ_{base}. Now a new hidden layer is added to the network at some arbitrary position. Inserting the new layer results in additional trainable parameters θ_{new} being incorporated into the network, which become part of the extended set of parameters $\theta_{\text{ext}} = \binom{\theta_{\text{base}}}{\theta_{\text{new}}}$ containing the old parameters of the baseline network θ_{base} and the new parameters of the inserted layer θ_{new}.

The reason for inserting a layer into the neural network is to allow the extended network to represent a richer space of functions than the baseline network. In order to make use of the expanded space of functions, we initialize the newly added trainable parameters θ_{new} with two goals in mind:

(1) The new trainable parameters should not disrupt the state of the already trained parameters. More specifically, the propagation function $g_{\text{ext}}(\theta_{\text{ext}}, \cdot)$ of the extended network with the extended set of parameters θ_{ext} should be identical to the propagation function $g(\theta_{\text{base}}, \cdot)$ of the baseline network for the current baseline parameters θ_{base} in training at least on the set of the available data.

(2) The new trainable parameters should be initialized in a way that they can be trained with standard methods to contribute to the training process. Because typical training methods are first-order methods, the loss function $\mathcal{L}(g_{\text{ext}}(\theta_{\text{ext}}, x_{0,j}), y_{\text{label}j})$ at a typical (training) data point $(x_{0,j}, y_{\text{label}j})$ should have non-zero gradient components w.r.t. the new parameters θ_{new} after initialization. As a consequence, the training algorithm can make use of the new parameters immediately, since it is able to modify them in the next training step.

For feedforward ReLU-networks as introduced in (2.5), we employ the initialization from [16] to insert a new hidden layer (at layer index $i \geq 2$). This initialization sets the following values

$$W_i := \text{id}_{n_{X_i} \times n_{X_{i-1}}}, \quad b_i := 0 \in \mathbb{R}^{n_{X_i}} \tag{4.11}$$

for the new parameters $\theta_{\text{new}} = [W_i, b_i]$. Notice that, with this initialization, the newly added layer has the same width as its predecessor, i.e. $n_{X_i} = n_{X_{i-1}}$. Generally, it is not possible to insert a layer with a smaller width than its predecessor while satisfying item (1) due to information loss. However, inserting a layer with a larger width would be possible by adding extra inactive neurons, i.e. neurons that do not change the output of the layer, which is not explored here.

The initialization in (4.11) results in the identity propagation through the layer since the ReLU activation satisfies $\sigma \circ \sigma = \sigma$ and hence

$$\begin{aligned} x^+ &:= g^+_{\text{ext,part}}(\theta_{\text{new}}, x_{i-1}) = \operatorname{ReLU}(\mathrm{id}_{n_{X_i} \times n_{X_{i-1}}} x_{i-1} + 0) \\ &= \operatorname{ReLU}(x_{i-1}) = x_{i-1} \end{aligned}$$

holds. The last equality is true because x_{i-1} itself is the output of the previous ReLU layer and thus a vector with non-negative components. Hence, the overall propagation function of the extended network is identical to the one of the baseline network at the current point in training, i.e. $g_{\text{ext}}(\theta_{\text{ext}}, x_0) = g(\theta_{\text{base}}, x_0)\ \forall x_0 \in X_0$, which satisfies item (1). Examining the gradients of the new layer and abbreviating $\hat{f}_{\text{ext},j}(\theta_{\text{ext}}) := \mathcal{L}(g_{\text{ext}}(\theta_{\text{ext}}, x_{0,j}), y_{\text{label}j})$ for a data point $(x_{0,j}, y_{\text{label}j})$, we obtain

$$\nabla_{W_i} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) = (W_{i+1})^\intercal\, \nabla_{W_{i+1}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}), \tag{4.12a}$$

$$\nabla_{b_i} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) = (W_{i+1})^\intercal\, \nabla_{b_{i+1}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}). \tag{4.12b}$$

Thus, (4.12) implies that we can expect a non-zero gradient w.r.t. the additional parameters θ_{new} after the initialization of the new layer as long as the next layer is not degenerate, i.e. W_{i+1} is not the zero matrix and $\nabla_{W_{i+1}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}})$ or $\nabla_{b_{i+1}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}})$ is not the zero matrix.

To realize an identity layer initialization in FNNs with general activation functions, different from ReLU, the activation function may need to be parametrized as described in [129].

For ResNets as introduced in (2.10), the initialization is less technical than for FNN architectures due to the structure of the residual blocks to ensure item (1). We again satisfy this condition by initializing the propagation realized by the newly added layer to be the identity function. We propose to insert a layer after the i-th layer and initialize its parameters $\theta_{\text{new}} = [W_i^{(1)}, W_i^{(2)}, b_i]$ using

$$\begin{aligned} &W_i^{(1)} \in \mathbb{R}^{n_{X_i} \times n_{X_i}} \text{ and } b_i \in \mathbb{R}^{n_{X_i}} \text{ arbitrary}, \\ &W_i^{(2)} := 0 \in \mathbb{R}^{n_{X_i} \times n_{X_i}}. \end{aligned} \tag{4.13}$$

Then the partial propagation function $g^+_{\text{ext,part}}$ realized by the inserted layer is indeed the identity:

$$\begin{aligned} x^+ &:= g^+_{\text{ext,part}}(\theta_{\text{new}}, x_{i-1}) = x_{i-1} + W_i^{(2)}\, \sigma(W_i^{(1)} x_{i-1} + b_i) \\ &= x_{i-1} + 0\, \sigma(W_i^{(1)} x_{i-1} + b_i) = x_{i-1}. \end{aligned}$$

Hence, the overall propagation function of the extended network is identical to the one of the baseline network at the current point in training, i.e. $g_{\text{ext}}(\theta_{\text{ext}}, x_0) = g(\theta_{\text{base}}, x_0)\ \forall x_0 \in X_0$, which satisfies item (1).

Calculating the gradients of the new layer leads to

$$\nabla_{W_i^{(1)}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) = \sigma'(W_i^{(1)} x_{i-1} + b_i)^\top \, 0 \, \nabla_{x_i} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) \, (x_{i-1})^\top = 0, \tag{4.14a}$$

$$\nabla_b \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) = \sigma'(W_i^{(1)} x_{i-1} + b_i)^\top \, 0 \, \nabla_{x_i} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) = 0, \tag{4.14b}$$

$$\nabla_{W_i^{(2)}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) = \nabla_{x_i} \hat{f}_{\text{ext},j}(\theta_{\text{ext}}) \, \sigma(W_i^{(1)} x_{i-1} + b_i)^\top, \tag{4.14c}$$

where gradients here are evaluated with respect to the Euclidean and Frobenius inner products, respectively. Moreover, $x_i = x_i(j)$ is the value of the input $x(j)$, propagated to the i-th hidden layer. The index j indicates that we consider the loss at a specific data point (x_j, y_j) and the index i is the layer index.

Again, we take item (2) of the loss having non-zero gradient w.r.t. the additional parameters into account in the choice of the initialization as well. This leads to further requirements on the initialization in (4.13) which has some freedom in the choice of $W_i^{(1)}$ and b_i. From (4.14c), we see that we need to initialize the new weight $W_i^{(1)}$ and bias b_i such that $\sigma(W_i^{(1)}(\cdot) + b_i)$ is not systematically zero, which depends on the employed activation function σ. For instance, for popular choices of the activation function σ like tanh or leaky ReLU, we can choose $W_i^{(1)} = \text{id}$ and $b_i = 0$. Note that the choice of $W_i^{(1)}$ will determine the scale or norm of the resulting gradient $\nabla_{W_i^{(2)}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}})$. For a fair comparison between different layer positions later on it is recommended to initialize the inner weight matrix $W_i^{(1)}$ in the same range (or even as the same matrix) for all layers, to not influence the magnitudes of $\nabla_{W_i^{(2)}} \hat{f}_{\text{ext},j}(\theta_{\text{ext}})$ by the choice of this initialization, cf. (4.14c). We also infer from (4.14) that the inner weight matrix $W_i^{(1)}$ and bias vector b_i will only start to receive non-zero updates from the second instead of first (mini-batch) gradient step on.

For the CNNs as described in (2.11), we also concentrate on the ReLU activation function and on the convolutional layers of the network. We now want to find an initialization of the kernel $K \in \mathbb{R}^{3\times3\times m\times m}$ and the bias $b_i \in \mathbb{R}^m$ so that

$$K \circledast X + b_i = X.$$

For all kernel indices $i, j = 1, 2, 3$ and channels $k, \ell = 1, \ldots, m$, we set

$$K_{i,j,k,\ell} \coloneqq \begin{cases} 1 & \text{if } i = j = 2 \text{ and } k = \ell, \\ 0 & \text{otherwise,} \end{cases} \tag{4.15a}$$

$$b_i \coloneqq 0 \quad \text{for } i = 1, \ldots, m. \tag{4.15b}$$

Then the forward propagation through the layer is indeed the identity function, as can be seen by direct calculation as for the FNN, cf. (4.11). As in the case for FNNs, we choose to insert a convolutional layer with the same number of channels as its predecessor. Inserting a layer with fewer channels is not possible while satisfying item (1) due to information loss. Inserting a layer with more channels is possible by adding extra inactive channels, i.e. channels that do not change the output of the layer. Additionally, similar to the FNN initialization, it would be possible to modify this initialization to different activation functions. We do not explore these options here.

The initializations for the different architectures do not cause symmetry problems in the kernels and weight matrices, as can be observed in Figure 4.1 for CNNs, Figure 4.2 for FNNs and Figure 4.3 for

ResNets.

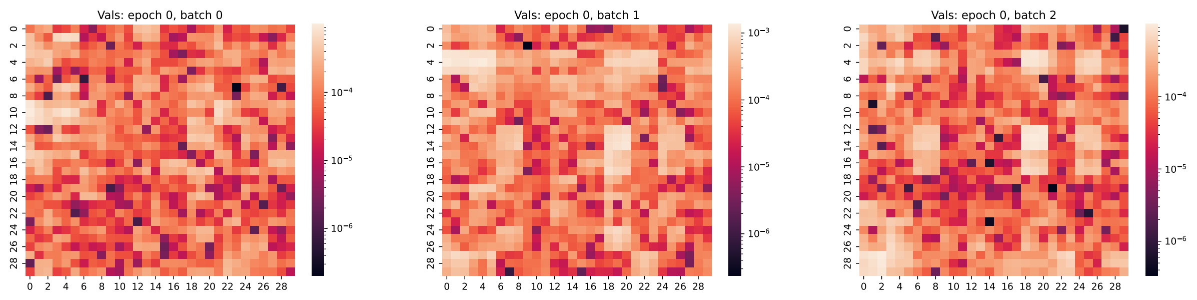


Figure 4.1.: Heatmaps showing the entries of absolute value of the (Euclidean) gradient of a convolutional kernel, flattened to 2 dimensions, for the first three mini-batch SGD iterations after insertion. The kernel is newly inserted into the CNN following the initialization described in (4.15).

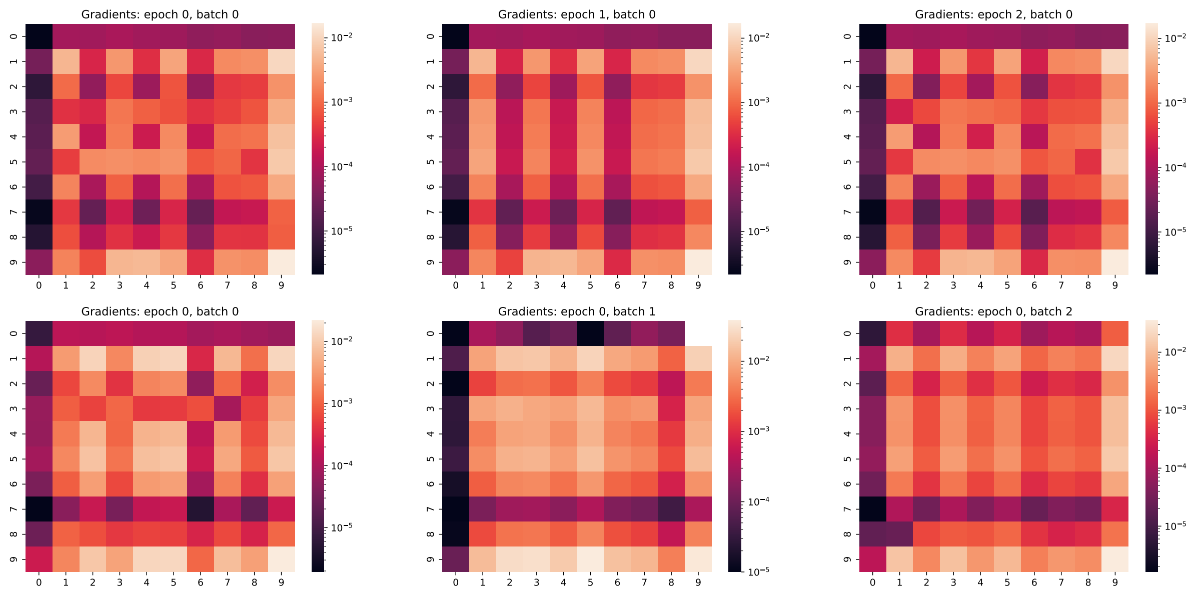


Figure 4.2.: Heatmaps showing the entries of the absolute values of the (Euclidean) gradient of a weight matrix, for the first three iterations after insertion. The weight matrix is newly inserted into a ReLU-FNN following the initialization described in (4.11). Top row: full-batch training. Bottom row: mini-batch training.

Remark 4.7 (Differentiation to [16]). *While we use the initialization proposed in [16] for FNNs (and CNNs), we use it to grow new layers during the training process and not at some converged point, which is proposed in the original paper.*

Remark 4.8. *We have proposed initialization strategies for networks that refine the network (by inserting new layers in the network) satisfying the two requirements items (1) and (2). We do not use linear*

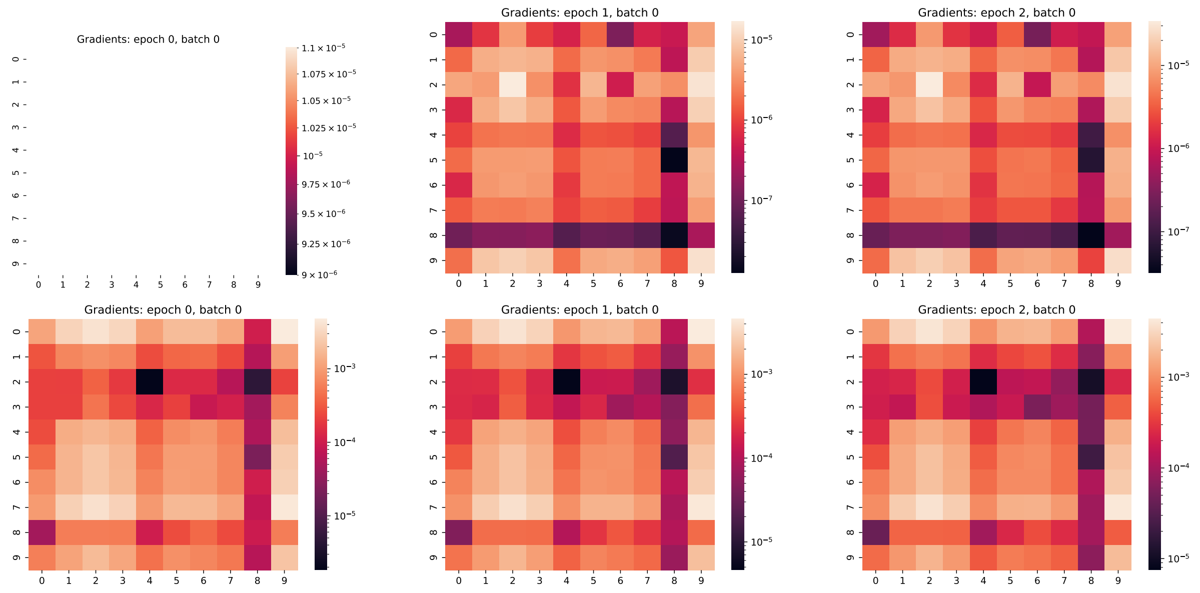


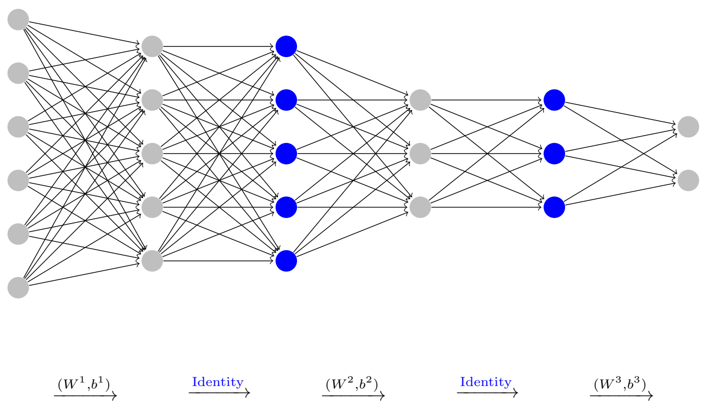
Figure 4.3.: Heatmaps showing the entries of the absolute values of the (Euclidean) gradient of the weight matrices of a residual layer, for the first three iterations after insertion. The weight matrices are newly inserted into a ResNet following the initialization described in (4.13). Top row: inner weight matrix W_i^1. Bottom row: outer weight matrix W_i^2. Full-batch gradient descent (GD) is used as optimizer. The top left heatmap is white, since the gradient is zero as expected by (4.14a).

interpolation as in grid refinement for differential equations, since the layers (or more exactly the values of the nodes of the layers) of a neural network are not the quantities we can and need to initialize, but rather the weight and bias parameters determining the transition function of the new layer.

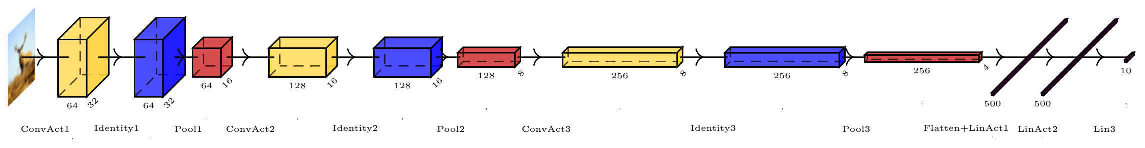
Selection of Layer Position:

The choices of initialization proposed above for FNNs, cf. (4.11), ResNets cf. (4.13), and CNNs, cf. (4.15) make it possible to insert multiple new hidden layers. More specifically, a new identity layer can be inserted after each hidden layer which already exists in the baseline network for FNNs, after each convolutional layer for CNNs, and a new residual layer can be inserted in each position in a ResNet where the width (of the preceding layer) of the current insertion point allows for a new residual layer, see Figure 4.4 for an illustration.

In the following, we describe the SensLI procedure to find the position for a newly inserted layer predicted to be most effective from sensitivity analysis. The layer insertion procedure makes no requirements on the state of the training process and can be applied at any point during training. Consequently, the procedure can be repeated multiple times during a training process to gradually expand the neural network. Additionally, it can be used to insert multiple layers at once, although we focus on selecting a single layer for insertion at a time in the following description of the method.


$\xrightarrow{(W^1,b^1)}$
$\xrightarrow{\text{Identity}}$
$\xrightarrow{(W^2,b^2)}$
$\xrightarrow{\text{Identity}}$
$\xrightarrow{(W^3,b^3)}$

(a) Feedforward Neural Network.



(b) Convolutional Neural Network.

Figure 4.4.: Examples for possible locations for layer insertion (blue) following the initialization described in (4.11) for FNNs and (4.15) for CNNs.

To find the most effective position for the new layer to be inserted, we have developed a notion of merit for its insertion at a particular location using the sensitivity analysis described in Section 4.3.1.

For simplicity, the description in Section 4.3.1 was referring to an arbitrary number of new parameters and the merit of their insertion into the network. In fact, the analysis does not change when several layers are selected to be added simultaneously. We now propose to insert a new layer at all possible positions in the network simultaneously in order to select the most effective one. An example is given in Figure 4.4. This results in a fully-extended network with parameters $\theta_{\text{ext}} = \binom{\theta_{\text{base}}}{\theta_{\text{new}}}$ containing the old parameters of the baseline network θ_{base} and the new parameters

$$\theta_{\text{new}} = \begin{pmatrix} \theta_{\text{new}}^{(1)} \\ \vdots \\ \theta_{\text{new}}^{(L)} \end{pmatrix}$$

of all L newly added layers as subvectors. The same structure will be inherited by the sensitivities

$$\lambda = -\nabla_{\theta_{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}). \tag{4.16}$$

We can therefore evaluate the norm, (4.10), separately for each chunk of parameters pertaining to a particular layer, and compare them. In our implementation, we are using the Frobenius norm of the partial gradient w.r.t. the weight matrix W_i,

$$\|\nabla_{W_i}\hat{f}_{\text{ext}}(\theta_{\text{ext}})\|_F^2, \tag{4.17}$$

as our final notion of merit of inserting a layer in case of a ReLU-FNN. The impact of the bias vector is disregarded. In case of a ResNet, we use (4.17) with the outer weight matrix W_i^2 instead of W_i

$$\|\nabla_{W_i^2}\hat{f}_{\text{ext}}(\theta_{\text{ext}})\|_F^2.$$

The reason we can disregard the impact of W_i^1 is that, according to (4.14a), $\nabla_{W_i^1}\hat{f}_{\text{ext}}(\theta_{\text{ext}})$ is initially equal to zero.

Note that even though the sensitivity analysis is based on the formulation of training as a constrained optimization problem (4.2), we do not actually need to compute costly iterations on the constrained optimization problem in the method we propose. Thankfully it is possible to compute the Lagrange multipliers (4.16) of the constrained problem by a forward and backward pass on the fully-extended network without the need to formulate the constrained problem explicitly and iterate with a costly optimization method suited for constrained optimization problems.

Remark 4.9. *Training a neural network via a constrained optimization problem is possible and done in a different setting with neural networks, e.g. in [36]. Formulating the training as a constrained optimization problem has the advantage that certain properties can be guaranteed by the constraints, but increases the number of optimization parameters significantly and needs special optimization methods for constrained problems. In our case it would increase the number of optimization variables roughly by factor 1.5 of the new, extended network, which is a significant increase.*

Remark 4.10 (Parallels to Adaptive Grid Refinement)**.** *We want to find a suitable position in the neural network where the new layer should be inserted. The SensLI method uses an idea which is also used in adaptive grid refinement and aims to insert new layers in regions where the neural network shows high activities. Being able to insert a new layer at any position in the neural network, i.e. between any two existing layers, can be seen as similar to being able to refine a one-dimensional grid between each two already existing grid points.*

In Convolutional Neural Networks (CNNs), the gradient with respect to the kernel, $\nabla_K\hat{f}_{\text{ext}}(\theta_{\text{ext}})$, is a tensor of higher dimensionality compared to the gradient with respect to the weight matrix encountered in a Feedforward Neural Network (FNN). For this reason, it is appropriate to select a norm that is well-adapted to the structural properties of the kernel tensor. In Appendix B.3, we provide a comparison of the behavior of various norms. For the subsequent analysis, we will utilize the operator norm as defined in (B.1c). At each stage of training, we identify the layer for which the merit indicator is maximized and designate this layer as the single candidate for insertion. Additionally, we note that, in the context of FNNs, introducing a scaling factor of $\frac{1}{n_{X_i}^2}$ can be beneficial for rendering the merit indicator comparable across layers with differing computational complexities.

The process of evaluating the Frobenius norm, as specified in (4.17), is both straightforward and computationally efficient. To compute the required gradients, we temporarily halt the training procedure and construct a fully-extended network from the baseline network in which new layers are inserted at every possible location. The weights and biases of the network are then set by copying the current values of θ_{base} into their respective positions, while the newly introduced weights and biases are initialized according to the procedure outlined in Section 4.3.2. Subsequently, we perform a single evaluation of $\hat{f}_{\text{ext}}(\theta_{\text{ext}})$ and, in the same forward-backward pass, compute the gradients with respect to all parameters, while none of the weights are updated during this process. This operation is equivalent to executing a full-batch gradient descent step with a learning rate of zero. The merit quantity described in (4.17) can then be conveniently assessed for each layer individually.

When employing a mini-batch training algorithm, the gradient $\nabla_{W_i}\hat{f}_{\text{ext}}(\theta_{\text{ext}})$ is not immediately available at once. To address this, we carry out a complete epoch of mini-batch Stochastic Gradient Descent (SGD) steps, again with a zero learning rate, in order to obtain the sensitivities $\nabla_{W_i}\hat{f}_{\text{ext}}(\theta_{\text{ext}})$ over the whole training data and to compute their norm as specified in (4.17). Alternatively, one could opt to balance accuracy and computational efficiency by evaluating the sensitivities on a sufficiently large mini-batch rather than the entire dataset. Once the largest norm among all candidate new layers has been identified, we discard the temporary fully-extended network, proceed to insert the selected layer into the baseline model, initialize its weights and biases as previously described in Section 4.3.2, and then resume the training process.

When to Insert a New Layer:

The question on when (*after which iteration*) to insert a new layer is best in the training process is generally hard to answer from a mathematical perspective. This stems from the fact that the classical training process of neural networks consists of stochastic iterations on a highly non-convex stochastic optimization problem, for which only weak local convergence results exist under some assumptions which are not necessarily satisfied in practice. Even for a standard training problem it is not clear from a mathematical perspective when to stop the training process, see e.g. [139]. Thus, the question when to insert a new layer in the training process is even more difficult to answer and remains without a satisfying answer. Even though training the smaller network for as long as possible can save resources, training it for too long can be lead to inefficient parameter updates and even may prevent the extended network from fully leveraging the added layer, as e.g. for FNNs as discussed in Remark 4.6. Our numerical experiments show that the time in which a layer is inserted can have a strong impact on the training process, which makes sense in the light of the highly non-convex optimization landscape, see e.g. Figure 4.12.

To the best of our knowledge, currently existing state-of-the-art methods to decide the insertion time during training are heuristic at regular intervals or manually tuned, as in [133, 92].

We propose to evaluate the merit of inserting a new layer at regular intervals during training, and to insert the layer with the highest sensitivity when

$$\frac{\|\nabla_{W_i}\hat{f}_{\text{ext}}(\theta_{\text{ext}})\|^2}{\frac{1}{\#\mathcal{W}}\sum_{W\in\mathcal{W}}\|\nabla_W\hat{f}_{\text{ext}}(\theta_{\text{ext}})\|^2} \geq \tau \tag{4.18}$$

holds with $\tau \geq 1$. Here, $\mathcal{W}$ is the set containing all weight matrices or kernels already present in the baseline model. The threshold criterion compares a suitable norm of the sensitivities of the new layers with norm of the sensitivities of the already existing layers (more precisely their weight matrices or kernels). When the ratio between the norms is too small, no new layer is inserted, since we expect the new layer to only have a small impact on following training process of the neural network. This however is only a heuristic and is not guaranteed to be optimal. In Figure 4.8, we observe that already for $\tau = 1$ the heuristic in (4.18) actually accepts and rejects layer insertions in practice. This heuristic allows the potential insertion of more than one layer during training, if the sensitivities of more than one layer are sufficiently large. It can also happen that no layer is inserted during a selection phase, if the sensitivities of all potential new layers are too small. Additionally, the choice of τ allows one to control the number of layers inserted during training.

Summary of the SensLI Algorithm

We summarize the steps of the SensLI algorithm in Algorithm 6.

Algorithm 6 Sensitivity-based Layer Insertion (SensLI)

```
1: Start with a baseline architecture $g$ (either a FNN, ResNet, or CNN as described in Section 2.2.1) and initial parameters $\theta_0$.
2: Train on the baseline network for $K_0$ epochs, starting from $\theta_0$ (with a chosen optimizer): $\theta_{\text{curr}} \coloneqq \text{Train}(g, \theta_0, K_0)$.
3: while Termination criterion not met do
4:     Fully extend the current network by inserting identity layers at all possible positions: $g_{\text{ext}} \coloneqq \text{Extend}(g)$.
5:     Initialize the fully-extended network with $\theta_{\text{ext}}$ as described in Section 4.3.2: $\theta_{\text{ext}} \coloneqq \text{InitExtended}(g_{\text{ext}}, \theta_{\text{curr}})$.
6:     Compute sensitivities w.r.t. the new weight matrices over the whole training data by performing a backward pass for every training data point as described in Section 4.3.1: $\lambda \coloneqq \text{SensComp}(g_{\text{ext}}, \theta_{\text{ext}})$.
7:     Select whether and where to insert a layer by comparing the norms of the sensitivities and checking the threshold in (4.18): $g, \theta_0 \coloneqq \text{SelectNewNetwork}(g, \theta_{\text{curr}}, \lambda)$.
8:     Train on the current network for $K_{\text{curr}}$ epochs, starting from $\theta_0$ (with a chosen optimizer): $\theta_{\text{curr}} \coloneqq \text{Train}(g, \theta_0, K_{\text{curr}})$.
9: end while
10: return Extended network $g$ with parameters $\theta_{\text{curr}}$.
```

4.3.3. Numerical Experiments

We perform experiments with the architectures described above, namely FNNs, ResNets and CNNs. For the FNN and ResNet architectures, we consider a spiral data set for binary classification, cf. Figure 2.4. For the numerical experiments with CNN architectures we need a more complex data set consisting of images. Hence, we employ the CIFAR-10 data set, cf. Figure 2.6. For more information on the data sets used, we refer to Section 2.2.2. A validation set is not needed in our experiments, as we do not conduct explicit hyperparameter search. Instead, our depth-adaptive layer insertion approach effectively serves as an automated search for the optimal network depth.

All detailed information on the experimental setup is provided in the appendix of [57]. We use full-batch gradient descent in several experiments to facilitate clearer interpretation of the results. While mini-batch SGD (with momentum) introduces additional variability, it can yield more competitive outcomes, particularly for larger datasets. For this reason, we also present results from experiments using mini-batch training. In all figures, the points of layer insertion are marked by vertical dotted lines.

Our implementation of the SensLI method is available on `GitHub`[1] and is based on `PyTorch` [103].

Fixed-Architecture Training vs SensLI

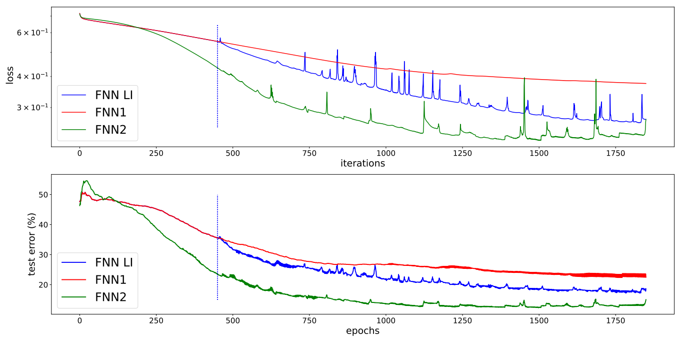
In this subsection, we compare the performance of the SensLI method to that of fixed-architecture training of the baseline network of the same architecture and to an extended architecture which is the resulting architecture generated by SensLI after layer insertion. We first consider experiments with only a single layer insertion for FNNs, ResNets and CNNs. As a further step, we will examine the behavior of SensLI with repeated layer insertions as in Figure 4.8.

Insertion of a Single Layer

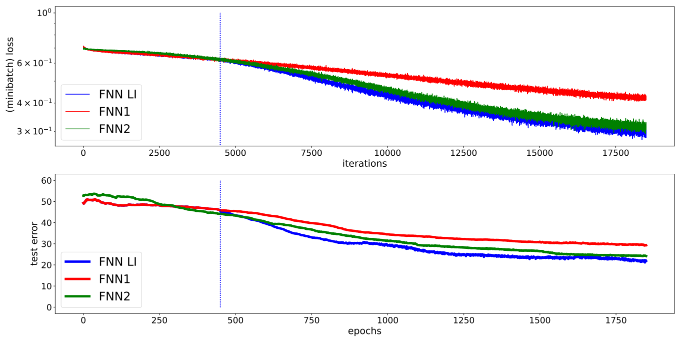
We evaluate SensLI on FNN, ResNet, and CNN architectures (denoted as FNN LI, ResNet LI, CNN LI) and compare its performance to both the corresponding baseline models (FNN1, ResNet1, CNN1) and the extended architectures resulting from the insertion of one selected layer (FNN2, ResNet2, CNN2). After we examine the performance w.r.t. iteration count, we will also compare the computational cost of the different training processes.

We account for the effects of random initialization by averaging results over multiple training runs in Figure 4.5. To ensure comparability, we fix both the iteration at which a layer is inserted and the learning rate across all runs. By restoring the random seed, we guarantee identical initializations for both the baseline and SensLI architectures trainings. As a result, the loss and error trajectories are consistent up to the point of layer insertion, apart from the variability introduced by mini-batch selection.

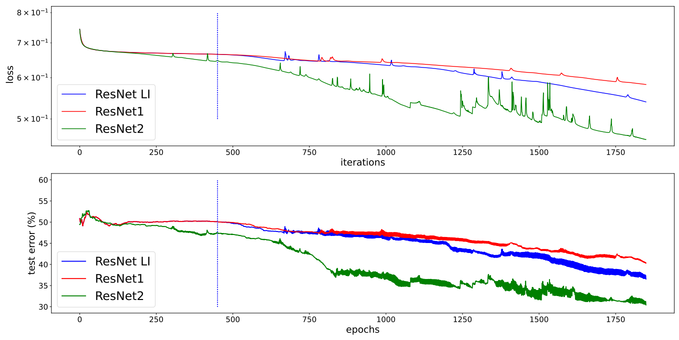
[1]`https://github.com/mathemml/SensLI`



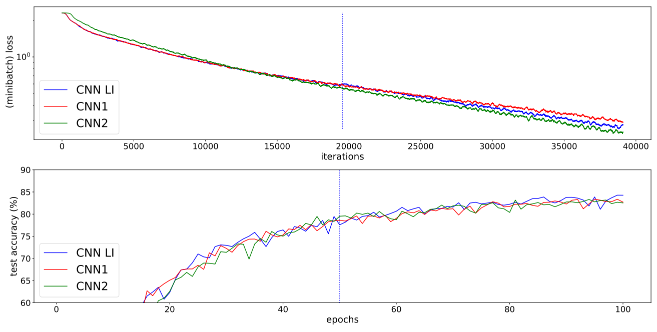
(a) FNN, full-batch GD.



(b) FNN, mini-batch SGD.



(c) ResNet, full-batch GD.

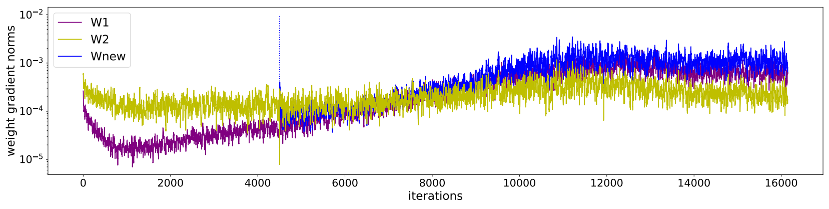


(d) CNN, mini-batch SGD with momentum.

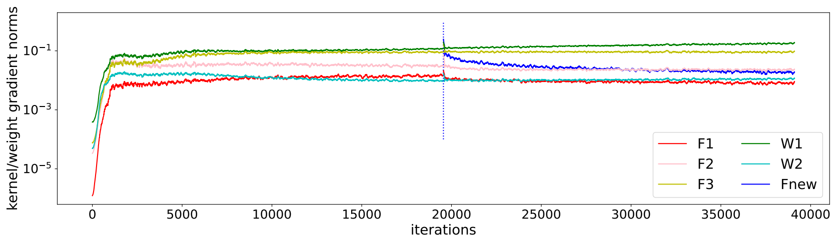
Figure 4.5.: Comparison of layer insertion and fixed-architecture training for FNNs with full-batch (a) and mini-batch SGD (b), ResNets with full-batch (c) and CNNs with mini-batch SGD. We show the loss (top) and test error (bottom) averaged over 30 (FNN), 40 (ResNet) and 10 (CNN) runs, respectively. These experiments are included in the `GitHub` repository as Exp2–Exp5.

Across all architectures, SensLI consistently results in a faster reduction of the loss w.r.t. iteration count compared to fixed-architecture training on the baseline network. This demonstrates that the newly inserted layer actively contributes to the learning process. The improvement is most notable for FNNs, particularly when using mini-batch SGD, where SensLI even surpasses the performance of training on the extended network FNN2. In contrast, the benefit of layer insertion is less pronounced for ResNets and CNNs. This can be attributed to the fact that ResNet1 and CNN1 baseline architectures are inherently more expressive than FNN1, and the absolute as well as relative increase in the number of parameters after layer insertion is smaller for these models. When comparing SensLI to fixed-architecture training on the extended networks, a more rapid loss decay is observed only for FNNs with mini-batch SGD. Nevertheless, SensLI is more computationally efficient in all cases, as shown in Table 4.4.

It should be noted that, for the sake of comparability, the insertion epoch was fixed, which may restrict the effectiveness of SensLI, since the optimal insertion time can depend on the specific initialization. Therefore, SensLI may not achieve its full potential in every averaged training run. Additionally, because the number of parameters in the extended CNN2 model is only 1.4 % greater than in the baseline CNN1 model (Table 4.3), only minor differences in accuracy are to be expected. The only small effect of layer insertion for the CNN architecture training is also observable when examining the norms of the layer-wise gradients during training, cf. Figure 4.6. Figure 4.6 shows one exemplary run using SensLI for a FNN with mini-batch SGD and a CNN with mini-batch SGD with momentum, respectively. Generally, we observe that the gradient norm values (measured in the Frobenius norm) of the newly inserted weight matrices lie within a comparable range of the norms of pre-existing trainable values, i.e., no vanishing or exploding gradient problems are encountered. However, while for the FNN architecture, the new gradient exceeds the other gradients, which agrees with the pronounced effect of layer insertion visible in Figure 4.5b, for the CNN, the gradient norm values of the newly inserted kernel are not dominating.



(a) FNN, mini-batch SGD.



(b) CNN, mini-batch SGD with momentum.

Figure 4.6.: Behavior of layer-wise gradients during training for a FNN with mini-batch SGD (a) and a CNN with mini-batch SGD (b). We show the Frobenius norm of the gradients of the trainable parameters in the networks over iteration count (newly inserted weight matrix in blue). The experiments can be found in the `GitHub` repository as Exp4 and Exp5.

Table 4.2.: Comparison of Floating Point Operations (FLOPs) per training sample for Figure 4.7.

Architecture	**FLOPs**	**thereof FLOPs for SensLI evaluation**
ResNet1	90 000	
ResNet2	252 000	
ResNet LI	188 712	1512, approx. 0.8 %

In this comparison, the baseline architectures have the same number of parameters as SensLI before any layer insertion, while the extended architectures match the parameter count of SensLI after layers have been inserted, as detailed in Table 4.3.

Table 4.3.: Number of network parameters during training.

Epochs	**FNN LI**	**ResNet LI**	**Epochs**	**CNN LI**
0–449	27	33	0–49	2 674 816
450–end	57	54	50–end	2 711 744

Even though a network with more parameters has a higher computational cost per iteration than a smaller network, the number of parameters of a neural network does not directly relate linearly with the computational cost of a forward-backward pass through the network. Hence, as a more precise theoretical comparison, we provide approximate counts of Floating Point Operations (FLOPs) for the training process of the different architectures, cf. Table 4.4. We also highlight the amount of FLOPs needed for SensLI evaluation (during the expansion step of SensLI between training parts) to get a better understanding of the relative effort required by the expansion of the networks through SensLI.

Table 4.4.: Comparison of FLOPs per training sample for Figure 4.5.

Architecture	**FLOPs**	**thereof FLOPs for SensLI evaluation**
FNN1	222 000	
FNN2	499 500	
FNN LI	432 270	270, approx. 0.06 %
ResNet1	333 000	
ResNet2	532 000	
ResNet LI	484 488	288, approx. 0.05 %
CNN1	25 196 544 000	
CNN2	47 845 785 600	
CNN LI	37 452 607 488	931 442 688, approx. 2.5 %

For the larger CNN architecture, SensLI evaluation incurs higher computational cost because of the increased number of parameters in the extended layers. At the same time, the reduction in FLOPs from CNN2 to CNN LI is significantly greater than the corresponding reductions observed in the smaller FNN and ResNet architectures.

The above experiments using only one layer insertion illustrate that a SensLI can outperform fixed-architecture training even on the extended network, but in some scenarios only has a slight effect. We now examine the training performance of SensLI with repeated layer insertions during training.

Repeated Insertion of Layers

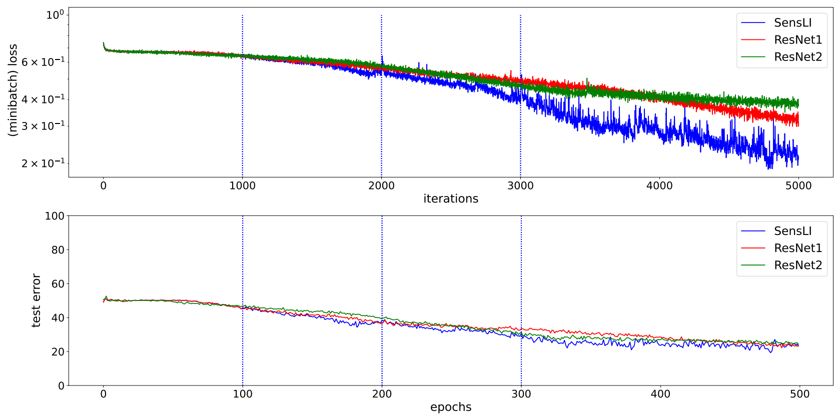


Figure 4.7.: Comparison of SensLI and fixed-architecture training for ResNets with mini-batch SGD. SensLI inserts a layer three times in the training process, indicated by vertical lines. We show the loss (top) and test error (bottom) averaged over 30 (ResNet) runs. The experiments can be found in the `GitHub` repository as Exp13.

First we consider a ResNet architecture with three repetitions of layer insertion, cf. Figure 4.7. The ResNet architecture is trained with mini-batch SGD and the learning rate is fixed to 0.01 for all runs. After 100, 200 and 300 epochs, we let SensLI insert a new layer.

On average, SensLI achieves better performance than fixed-architecture training on the baseline ResNet1 and even surpasses the extended ResNet2 architecture. This demonstrates that SensLI effectively utilizes the additional capacity provided by the inserted layers, while avoiding the tendency to get trapped in local minima—a phenomenon observed in fixed-architecture training on ResNet2. When comparing the results in Figure 4.5 to those in Figure 4.7 and Figure 4.8, it becomes evident that the benefits of SensLI are even more pronounced when multiple layers are inserted during training. Moreover, SensLI requires less training time for the ResNet architecture with three layer insertions than fixed-architecture training on the extended ResNet2, as shown in Table 4.2.

In Figure 4.8, we compare SensLI with three repeated layer insertions applied to a CNN architecture with training on the extended CNN from the outset. The CNN is again trained with mini-batch SGD with momentum and a fixed learning rate. We observe that SensLI achieves a lower training loss and higher test accuracy than fixed-architecture training on the extended CNN2. SensLI completes 200 epochs in 1659 s, which is only 71.5 % of the time required by the extended CNN (2320 s). Let us point out that SensLI only inserts two layers instead of the three initially planned, as the sensitivity threshold in (4.18) was not met during the second insertion attempt. This indicates that the heuristic

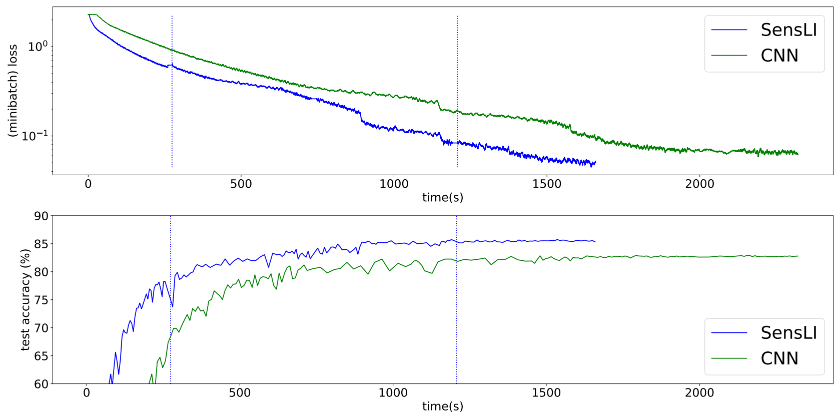


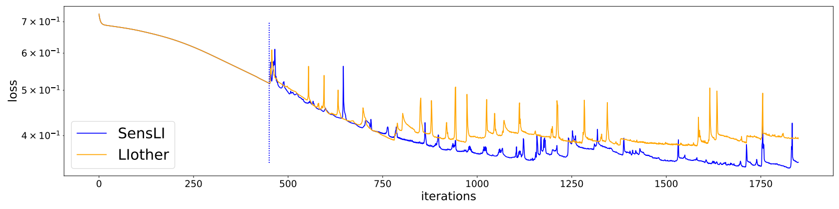
Figure 4.8.: Training a CNN on the CIFAR-10 data set with multiple layer insertions (SensLI), compared to training the extended CNN from the beginning, plotted over time in seconds. We display the training loss (top) and test accuracy (bottom). SensLI is executed every 50 epochs, i.e., 3 times throughout the training run, but decides against the second layer insertion, because the threshold in (4.18) was not met. Hence, only 2 layer insertions take place, which are indicated by vertical lines. This experiment is included in the `GitHub` repository as Exp1 and the detailed experiment setup is documented in the appendix of [57].

in (4.18) effectively prevents unnecessary layer insertions that would have minimal impact on the training process.

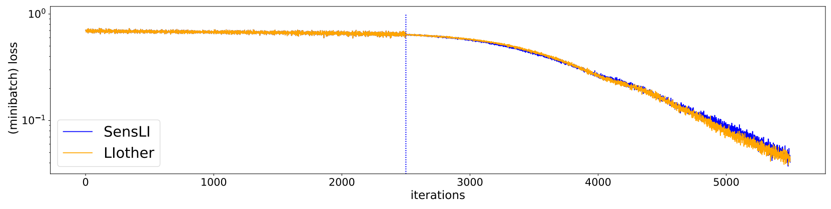
For a theoretical comparison, we estimate the total number of FLOPs per training data point, summed over all epochs. Fixed-architecture training on the extended CNN consumes approximately $3 \times 46\,996\,684\,800$ FLOPs per data point, while SensLI uses about $33\,063\,616\,512 \times 3$ FLOPs per data point, of which $914\,460\,672 \times 3$ FLOPs are attributed to the layer insertion evaluation—roughly 2.8 % of the overall training cost. The ratio of FLOPs per training data point (SensLI/CNN = 70 %) closely matches the observed ratio of total computational time, with minor deviations likely due to additional time required for initialization and setup.

Importance of the Layer Insertion Position

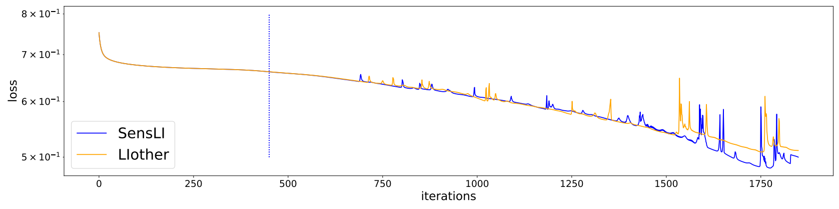
The proposed SensLI method inserts a new layer at the position where the layer is the most effective to reduce the loss function, based on local predictions from sensitivity analysis. In order to validate this strategy, we study the training performance for different layer positions. We compare inserting a new layer at the position indicated by SensLI to inserting the layer at the position with the smallest sensitivity indicator (LIother). We perform this comparison for FNN, ResNet and CNN architectures, using the merit indicators defined in (4.17) for the FNN and ResNet architectures with respect to W_i or W_i^2 and (B.1c) for the CNN architecture, respectively. In order to isolate the effect of layer placement, we consider layer insertions at a fixed epoch in this comparison and use baseline architectures with limited depth and constant width across the hidden layers, which leads to that all potential layers for insertion have the same number of parameters, cf. the appendix of [57, Appendix, Exp6-9]. Additionally, we start by considering full-batch gradient descent first and then consider mini-batch SGD (with momentum)



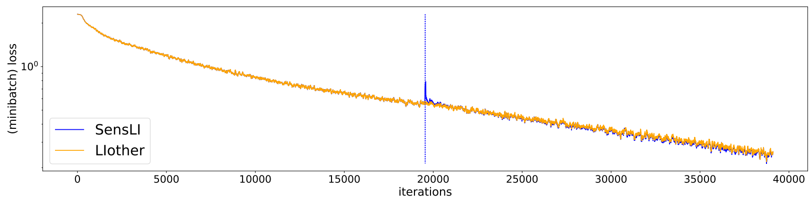
(a) FNN, full-batch GD.



(b) FNN, mini-batch SGD.



(c) ResNet, full-batch GD.



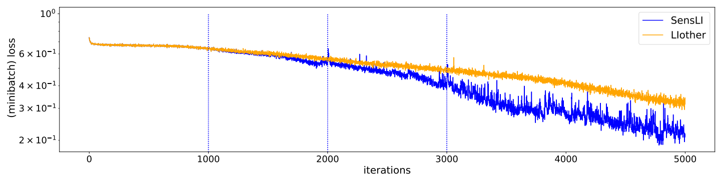
(d) CNN, mini-batch SGD with momentum.

Figure 4.9.: Comparison of layer insertion at positions given by the largest (SensLI) and the smallest (LIother) of the merit indicators. The indicators for each layer are given by (4.17) w.r.t. W_i for FNNs (a and b), (4.17) w.r.t. W_i^2 for ResNets (c), and (B.1c) for CNNs (d). We show the loss over iteration count, averaged over 30 (FNN, ResNet) and 7 (CNN) training runs. These experiments can be found in the `GitHub` repository as Exp6, Exp8, Exp7 and Exp9.

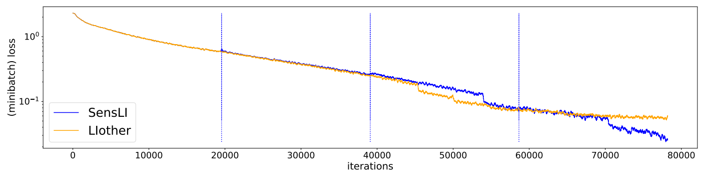
which introduces additional noise, which is not accounted for in the local sensitivity analysis.

In the experiment shown in Figure 4.9, we observe that inserting a layer at the position indicated by SensLI leads slightly to a faster reduction in training loss compared to inserting the layer at the position with the lowest sensitivity indicator (LIother) directly after the insertion and also in the long run. This effect is most pronounced for the FNN and ResNet architectures trained with full-batch gradient descent, cf. Figure 4.9a and Figure 4.9c. At the same time, we also observe that the noise introduced by mini-batch SGD can dominate the local effects of SensLI, as seen in Figure 4.9d and to some extent in Figure 4.9b.

Considering multiple layer insertions in Figure 4.10 as a second step, the advantage of SensLI over



(a) ResNet, mini-batch SGD.



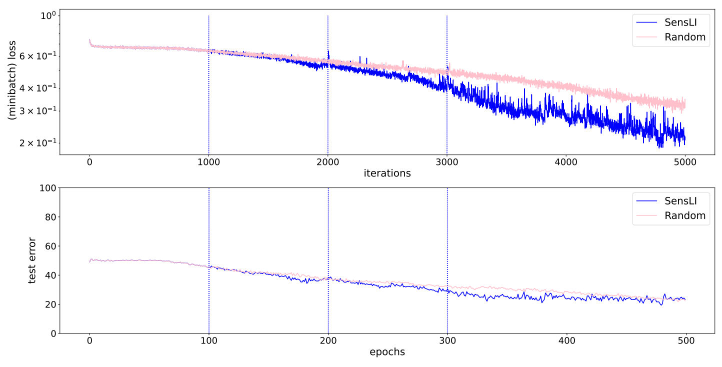
(b) CNN, mini-batch SGD.

Figure 4.10.: Comparison of layer insertion at positions given by the largest (SensLI) and the smallest (LIother) of the merit indicators. A layer is inserted three times each in the training process. We show the loss over iteration count and test error over epochs, averaged over 30 (ResNet) and 7 (CNN) training runs. These experiments can be found in the `GitHub` repository as Exp13 and Exp14.

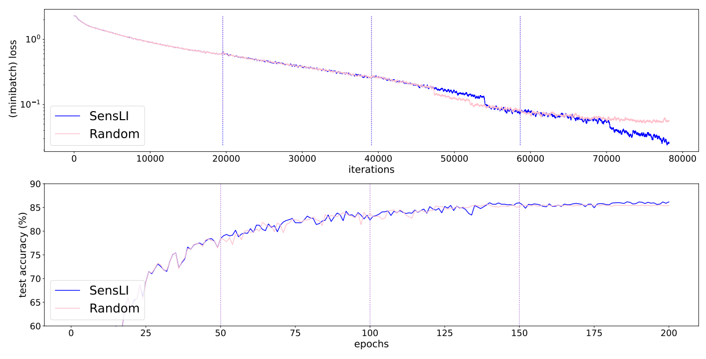
LIother becomes more pronounced again, even when using mini-batch SGD. This is also true for the long run and not only directly after the layer insertion. We consider a ResNet architecture in Figure 4.10a and a CNN architecture in Figure 4.10b and compare again against LIother, which inserts each new layer at the position with the smallest sensitivity indicator. Since we average over multiple runs, we plot loss over iterations instead of time. We observe that SensLI is more effective than LIother to reduce the training loss, even in the presence of noise from mini-batch training.

Comparison against Random Layer Insertion

We compare SensLI to a random layer insertion strategy, which randomly selects a layer position. For this comparison, we again consider multiple layer insertions during training for a ResNet and a CNN architecture, respectively. We average over multiple training runs to account for the effects of random initialization and random layer insertion. In Figure 4.11, we observe that SensLI outperforms the random insertion strategy in both cases.



(a) ResNet, mini-batch SGD.



(b) CNN, mini-batch SGD.

Figure 4.11.: Comparison of layer insertion at positions given by the largest of the merit indicators (SensLI) or random positioning (Random). Layer insertion is executed three times in the training process, indicated by dotted vertical lines. We show the loss over iteration count and test error over epochs, averaged over 30 (ResNet) and 7 (CNN) training runs. These experiments can be found in the `GitHub` repository as Exp13 and Exp14.

Comparison of Layer Insertion Points

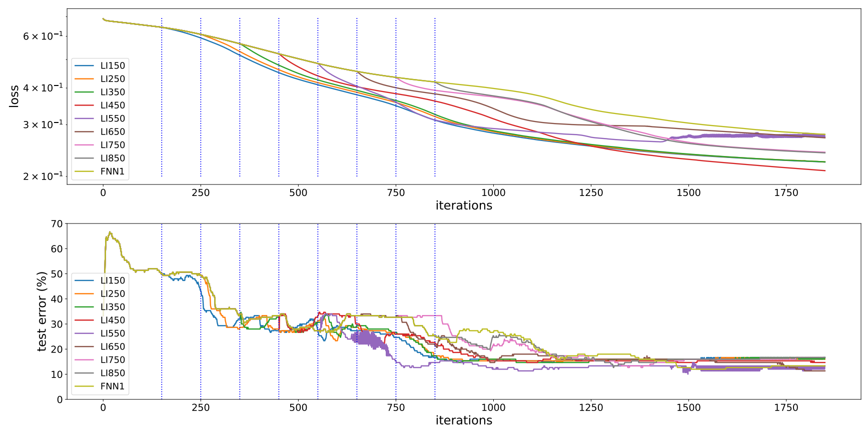
In this experiment, we analyze how the timing of layer insertion affects the training process. Throughout all experiments, we restrict ourselves to inserting a single layer to isolate the effect of the insertion timing. By restoring the random seed, we ensure that the initialization for every run in which a layer is inserted matches that of the baseline network (FNN1 or CNN1). As a result, when employing full-batch gradient descent, the training trajectories for all runs are identical to those of FNN1 up to the point of layer insertion.

We consider three training algorithms: full-batch SGD, mini-batch SGD, and mini-batch SGD with momentum. For each algorithm, we compare the training histories of the fixed-architecture baseline network (FNN1 or CNN1) with those of networks where an additional layer is inserted at various stages of training. Specifically, we investigate eight different insertion points in each setup: for FNN full-batch SGD, layers are inserted at iterations $150, 250, \dots, 850$; for FNN mini-batch SGD, at epochs $50, 100, \dots, 400$; and for CNN mini-batch SGD with momentum, at epochs $10, 20, \dots, 80$. To maintain comparability, identical hyperparameters are used for all layer insertions. Further details of the experimental setup are provided in the appendix of [57].

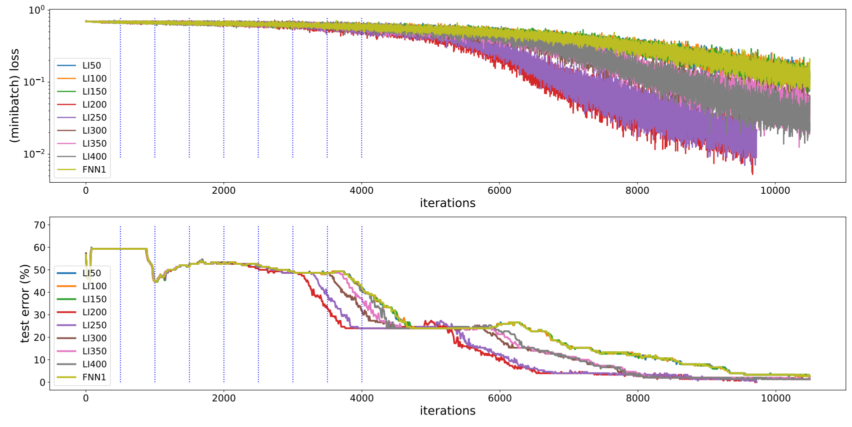
The resulting training histories are depicted in Figure 4.12. This experiment highlights that determining the optimal timing for layer insertion is not straightforward, likely due to the complex interplay of various factors like stochasticity and non-convexity during training. Notably, inserting a layer at any of the eight tested points during training yields better results than omitting the second hidden layer altogether (FNN1/CNN1). However, it is evident that inserting a layer too late in the training process diminishes its effectiveness.

For the FNN full-batch SGD scenario and the specific random instance considered, the most beneficial insertion point among the eight tested is after 450 iterations. In the FNN mini-batch SGD case, the training histories display a more monotonic behavior, with earlier layer insertions leading to faster loss reduction.

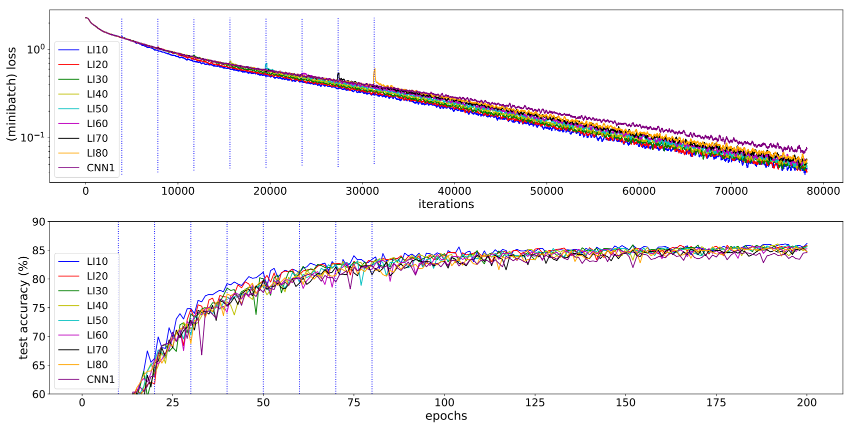
In contrast, for the CNN mini-batch SGD with momentum, the training histories are more erratic, and the timing of layer insertion has a less pronounced effect. Additionally, we observe that inserting a layer later in training tends to disrupt the training process more significantly. It is also worth noting that, for the CNN, the method consistently selects the same position for layer insertion at each tested epoch.



(a) FNN, full-batch GD.



(b) FNN, mini-batch SGD.



(c) CNN, mini-batch SGD with momentum.

Figure 4.12.: Comparison of layer insertion at different iterations (indicated by vertical lines) as described in Section 4.3.2. We show the loss and test error over iteration count. These experiments can be found in the `GitHub` repository under the name Exp10 and Exp11 for the ReLU-FNN and Exp12 for the CNN.

4.3.4. Comparison of SensLI to Other Layer Insertion Methods

In this section, we present a comparison between SensLI and other informed methods for growing networks which are able to handle layer insertion. Several approaches in the literature address the question of optimal layer placement, notably SENN [92], Firefly [133], and Autogrow [130], as summarized in Table 4.1. However, it is important to note that Autogrow relies on random initialization, which does not constitute an informed strategy. Consequently, our comparison focuses on SENN and Firefly. Conducting a fair numerical comparison between these methods is challenging for several reasons. For instance, SENN's implementation of layer insertion for CNNs is restricted to DenseNet architectures [64], whereas in Firefly, the process of layer insertion cannot be separated from layer widening within its framework. Due to these limitations, we instead provide a theoretical analysis of the computational effort required for network expansion, which demonstrates that SensLI is considerably less demanding in terms of computational resources.

The SensLI algorithm operates by performing a single full-batch forward and backward pass on a fully-extended network, where new layers are inserted at all possible positions. The selection of the layer to be inserted is then based on the norm of the gradient with respect to the variables of each candidate layer, as detailed in algorithm 7. Both SENN and Firefly also employ the strategy of considering layer insertion at all possible positions, but they differ in their mechanisms for selecting the new layer.

In the case of SENN, the procedure involves executing N random weight initializations, each followed by M iterations. This results in a computational effort equivalent to $N \times M$ (potentially large) mini-batch forward and backward passes for a network extended by one layer. For typical parameter choices, such as $N = 110$, $M = 300$ and a mini-batch size of $B = 1000$ out of $T = 50000$ training data points, this amounts to $N \times M \times B/T = 600$ full-batch forward and backward passes on the partially extended network, as described in [92, Appendix B]. For each random initialization, a natural expansion score is computed, which is approximated using a K-FAC approximation (cf. Section 2.5.4) of the Fisher matrix. The final selection of the layer position is made by comparing the expansion scores across all possible positions, meaning that the total number of forward and backward passes must be multiplied by the number of candidate layer positions.

Firefly, on the other hand, simultaneously optimizes both the initialization and the positions of new neurons. The method performs M full-batch gradient descent iterations on the fully-extended network, which contains new layers at all possible positions and includes additional variables for each new neuron. Each optimization step requires a full-batch forward and backward pass, and the authors indicate that only a few iterations are typically sufficient, i.e. $M \leq 10$. The selection of new neurons is then based on the values of these optimization variables.

In summary, SensLI achieves its goal (selecting a layer for insertion) with just one full-batch forward and backward pass on the fully-extended network. In contrast, SENN requires several hundred full-batch forward and backward passes on only partially-extended networks considering each possible layer position separately, while SensLI needs only one such pass on the fully-extended network. Firefly involves multiple optimization steps, each necessitating a full-batch forward and backward pass. No theory currently is able to show whether any of these methods achieves fewer iterations to a given loss or superior test accuracy. Hence, it is unclear how these methods perform against each other in practice, beyond computational effort. This analysis highlights the significant computational advantage of SensLI over these alternative methods.

4.4. Extension of SensLI to Layer Widening

The sensitivity-based approach which forms the basis of SensLI can be used for layer widening as well. We show the extension for FNNs here, but the approach can be adapted to other architectures and activation functions. Note that, generally, due to its structure, layer widening is less restrictive than layer insertion, allowing for more flexibility in the choice of the activation function.

Assume that we have two consecutive layers with weight matrices and biases $W_1 \in \mathbb{R}^{n_1 \times n_0}, b_1 \in \mathbb{R}^{n_1}$ and $W_2 \in \mathbb{R}^{n_2 \times n_1}, b_2 \in \mathbb{R}^{n_2}$, respectively. The forward propagation through the layers is given by

$$\begin{aligned} y_2 &\coloneqq W_1 x_1 + b_1, \quad & x_2 &\coloneqq \sigma(y_2), \\ y_3 &\coloneqq W_2 x_2 + b_2, \quad & x_3 &\coloneqq \sigma(y_3). \end{aligned}$$

Now we describe the change in the architecture when we widen the layer y_2 and consequently also x_2. For simplicity, we consider the case of adding one neuron, but the approach can be extended to adding multiple neurons. We add one neuron to y_2, which increases the size of W_1 and b_1 to $W_1^+ \in \mathbb{R}^{(n_1+1)\times n_0}, b_1^+ \in \mathbb{R}^{n_1+1}$ and the size of W_2 to $W_2^+ \in \mathbb{R}^{n_2 \times (n_1+1)}$. Note that the size of y_3 and x_3 remains unchanged and hence the dimension of the bias b_2 also remains unaffected. An illustration of the layer widening is given in Figure 4.13.

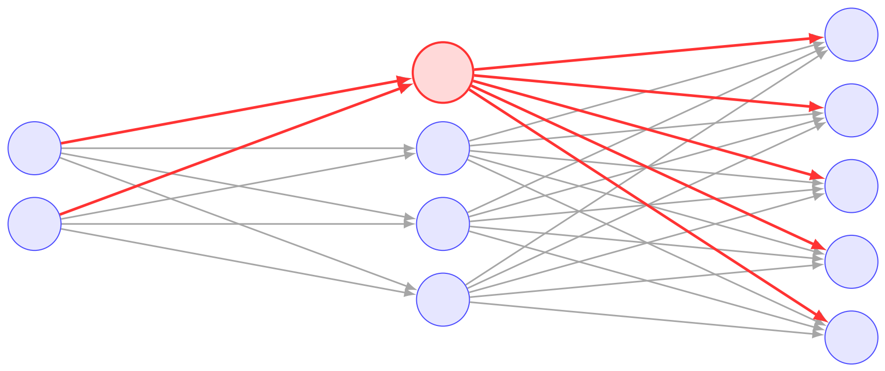


Figure 4.13.: Illustration of layer widening by adding one neuron (highlighted in red) to a hidden layer in a FNN.

The forward propagation through the widened layers is given by

$$\begin{aligned} y_2^+ &\coloneqq W_1^+ x_1 + b_1^+, \quad & x_2^+ &\coloneqq \sigma(y_2^+), \\ y_3^+ &\coloneqq W_2^+ x_2^+ + b_2, \quad & x_3^+ &\coloneqq \sigma(y_3^+), \end{aligned}$$

with $y_2^+ = \binom{y_2}{y_{\text{new}}}$, $x_2^+ = \binom{x_2}{x_{\text{new}}}$ and

$$W_1^+ = \begin{pmatrix} W_1 \\ W_1^{\text{new}} \end{pmatrix}, \quad b_1^+ = \begin{pmatrix} b_1 \\ b_1^{\text{new}} \end{pmatrix}, \quad W_2^+ = \begin{pmatrix} W_2 & W_2^{\text{new}} \end{pmatrix}.$$

To ensure that the output of the network remains unchanged after widening,

$$y_3^+ = y_3$$

must hold. Neuron splitting is possible and often used in the literature for this purpose, as e.g. in [136, 134]. However, this approach is not suited for our sensitivity-based approach. Hence, we aim to leave the baseline parameters unchanged and initialize the additional new parameters such that the output remains unchanged. This leads to the following conditions for the additional parameters given by

$$0 = W_2^{\text{new},i} \sigma(W_1^{\text{new}} x_1 + b_1^{\text{new}}) \quad \forall i = 1, \ldots, n_2.$$

The gradients of the extended network w.r.t. the extended parameters W_1^+, b_1^+, W_2^+ are given by

$$\begin{aligned} \nabla_{W_2^+} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \nabla_{y_3^+} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) x_2^{+\top} = \nabla_{y_3^+} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \begin{pmatrix} x_2^\top & x_{\text{new}} \end{pmatrix} \\ &= \nabla_{y_3^+} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \begin{pmatrix} x_2^\top & \sigma(W_1^{\text{new}} x_1 + b_1^{\text{new}}) \end{pmatrix} \in \mathbb{R}^{n_2 \times (n_1+1)}, \\ \nabla_{W_1^+} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \nabla_{y_2^+} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) x_1^\top = \begin{pmatrix} \nabla_{y_2} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \\ \nabla_{y_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \end{pmatrix} x_1^\top \in \mathbb{R}^{(n_1+1) \times n_0} \\ \nabla_{b_1^+} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \begin{pmatrix} \nabla_{y_2} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \\ \nabla_{y_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \end{pmatrix} \in \mathbb{R}^{n_1+1}, \end{aligned}$$

and the gradients w.r.t. the new neuron of the widened layer are given by

$$\begin{aligned}\nabla_{x_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \nabla_{y_3} \hat{f}_{\text{ext}}(\theta_{\text{ext}})^\top W_2^{\text{new}}, \\ \nabla_{y_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \nabla_{x_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \sigma'(y_2^{\text{new}}) = \nabla_{y_3} \hat{f}_{\text{ext}}(\theta_{\text{ext}})^\top W_2^{\text{new}} \sigma'(y_2^{\text{new}}).\end{aligned}$$

We propose the initializations

$$W_2^{\text{new}} = (0 \dots 0)^\top \in \mathbb{R}^{n_2 \times 1}, \quad W_1^{\text{new}} \in \mathbb{R}^{1 \times n_0} \neq 0, \quad b_1^{\text{new}} = 0 \in \mathbb{R}, \tag{4.19}$$

which ensure that the output of the network remains unchanged after widening. With this initialization, the gradients w.r.t. the new parameters at the first iteration of training after widening are given by

$$\begin{aligned}\nabla_{W_1^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \nabla_{y_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) x_1^\top = \nabla_{y_3} \hat{f}_{\text{ext}}(\theta_{\text{ext}})^\top W_2^{\text{new}} \sigma'(y_2^{\text{new}}) x_1^\top = 0, \\ \nabla_{b_1^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \nabla_{y_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) = x_1^\top = 0, \\ \nabla_{W_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) &= \nabla_{y_3} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) x_2^{\text{new}} = \nabla_{y_3} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \sigma(W_1^{\text{new}} x_1 + b_1^{\text{new}}).\end{aligned}$$

While the gradients w.r.t. the new parameters W_1^{new} and b_1^{new} are zero at the first iteration after widening, the gradient w.r.t. W_2^{new} is non-zero in general, depending on the choice of W_1^{new} and activation function σ. Would the weight matrix W_1^{new} be initialized with zeros as well, the gradient w.r.t. W_2^{new} would also be zero at the first iteration after widening, if an activation function with $\sigma(0) = 0$ is used, e.g. ReLU. Hence, we propose a non-zero initialization of W_1^{new}, e.g. a non-zero scalar. We point out again that the choice (4.19) is not unique and when comparing different inserted neurons, the same choice should be used to ensure comparability.

Remark 4.11 (Alternate Initialization Strategy). *Another possibility to initialize the new parameters is*

$$W_2^{new} \neq 0$$
$$\textit{and} \quad W_1^{new}, \quad b_1^{new}, \quad \textit{such that} \quad \sigma(W_1^{new} x_1 + b_1^{new}) = 0,$$

i.e. the new parameters are initialized such that the new neuron is in the saturated regime of the activation function. However, this approach is less flexible and not possible for all activation functions, e.g. ReLU, when the activation function satisfies $\sigma'(0) = 0$. Hence, we do not pursue this approach further.

We can now insert new neurons in all layers of the neural network and compare their effectiveness. Then, we can select the number of neurons we want to insert at one point in time and select the most effective ones. Imitating training on the baseline network can be done by constraining the values of the new parameters as described in Section 4.3.1 in the training on the fully-extended problem. Consequently, the effectiveness of a new neuron can be measured, as for layer insertion, by the square of the norm of the gradient w.r.t. the new parameters $W_1^{\text{new}}, b_1^{\text{new}}, W_2^{\text{new}}$ following the sensitivity analysis given in Section 4.3.1. Since the gradients w.r.t. W_1^{new} and b_1^{new} are zero at the first iteration after widening with the choice (4.19), the effectiveness of the new neuron is measured by

$$\|\nabla_{W_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}})\|^2 = \|\nabla_{y_3} \hat{f}_{\text{ext}}(\theta_{\text{ext}}) \sigma(W_1^{\text{new}} x_1 + b_1^{\text{new}})\|^2.$$

The position of the layer to be widened is then selected by comparing the effectiveness of a new neuron at all possible positions. Note that the choice of W_1^{new} influences the effectiveness of the new neuron and hence the choice of the position.

The decision of when to widen the network can be made similarly as for layer insertion, cf. Section 4.3.2. Hence, it would be possible to use a threshold strategy similar as described in Section 4.3.2, namely if

$$\frac{\|\nabla_{W_2^{\text{new}}} \hat{f}_{\text{ext}}(\theta_{\text{ext}})\|^2}{\frac{1}{\#\mathcal{W}} \sum_{W \in \mathcal{W}} \|\nabla_W \hat{f}_{\text{ext}}(\theta_{\text{ext}})\|^2} \geq \tau \tag{4.20}$$

holds with $\tau \geq 1$. Here, $\mathcal{W}$ is the set of all columns of all weight matrices in the current network which are consecutive weight matrices of a widened layer.

Although we have described the extension of SensLI to layer widening for the case of adding a single neuron, the approach can be extended to adding multiple neurons either per layer or spread between multiple layers as well. As a next step, the proposed extension of SensLI to layer widening should be validated numerically, which we leave for future work.

4.5. Discussion

The Sensitivity-based Layer Insertion (SensLI) approach has the advantage of conceptual and computational simplicity. When training is suspended for a potential insertion, a temporary fully-extended network is built by inserting identity-initialized layers at all candidate positions and the gradient of the extended objective is evaluated once by backpropagation. This makes SensLI computationally cheaper than alternatives such as SENN and Firefly, while still covering the questions of where, how and when to insert a layer. Unlike methods that rely on costly candidate selection or uninformed random initializations, SensLI uses a clear selection criterion rooted in sensitivity analysis for nonlinear programming and can be applied to residual, fully-connected feedforward and convolutional layers. Although the experiments focus on inserting a single layer at a time, the proposed merit indicators can be used to insert multiple layers concurrently and to extend the approach to layer widening.

The idea for SensLI is inspired by adaptive grid refinement techniques used in the numerical solution of PDEs. In that context, the quantities of interest are typically spatially distributed fields, which can be directly manipulated by refining or coarsening the computational grid. In contrast, neural networks are parametrized by high-dimensional weight and bias vectors, which determine the propagation functions between layers and not the values of the neurons in a layer themselves. Hence, there is no natural way to interpolate these parameters when inserting a new layer, nor is there a discretization parameter analogous to grid spacing in PDEs, except in specially designed architectures like ResNets. SensLI overcomes these challenges by initializing new layers as identity mappings, ensuring that the insertion does not disrupt the current function of the network. The method formulates training as a constrained optimization problem, where the parameters of new layers are fixed at their initialized values. This allows the use of Lagrange multipliers as sensitivity indicators, revealing which constraints and thus which potential layer insertions are most restrictive to loss descent. The layers associated with the largest sensitivity norms are selected for insertion, enabling the network to grow adaptively during training.

The method is based on a first-order, local prediction, yet global improvements in loss decay and test accuracy are observed empirically. SensLI can outperform training on extended architectures from the outset, particularly in CNNs, by reducing computational effort and training time, as exemplary shown in Figure 4.8. This outcome is notable because sensitivity analysis cannot, in principle, predict long-term effects of insertion or guarantee which insertion yields the fastest eventual convergence or best test accuracy.

Several limitations must be acknowledged. SensLI's selection is inherently local and therefore offers no guarantee that a chosen insertion will be beneficial in the long run. The supporting theory assumes smoothness conditions that are not satisfied for ReLU activations, and extending rigorous validation to stochastic, non-convex training is challenging. Consequently, no general proof of benefit for layer insertion is provided or even exists to the best of our knowledge; the evidence presented here is empirical and could be extended to further validate the approach.

An alternative way to improve neural network training besides allowing the architecture to adapt during training is to employ more informed optimization algorithms. In this context, preconditioned gradient methods are of particular interest, as they can be implemented with moderate computational overhead while potentially improving convergence significantly. To examine the behavior of preconditioned SGD updates for neural network training problems, we analyze the use of a layer-wise preconditioner in the next chapter. Because of the stochastic nature of the training algorithms and the non-convexity of the underlying optimization problems, we consider preconditioners based on covariance information generated in the forward and backward passes of the network.

5. Layer-wise preconditioning for Neural Network Parameters

In this chapter, we introduce a layer-wise preconditioning method for Feedforward Neural Networks based on the Frobenius-type inner products introduced in Section 2.5.1, which we call *Frobenius-type preconditioning* and which relies on inner products on the layer spaces. In most cases the layer spaces are not equipped with non-canonical inner products, hence we explore data-driven choices of inner products on the layer spaces based on covariance information of the forward and backward pass. The resulting preconditioner has a strong connection to Kronecker-Factored Approximate Curvature (K-FAC) [91], even though the derivation is different. We discuss differences to K-FAC and evaluate the numerical performance of Frobenius-type preconditioning in comparison to unpreconditioned Stochastic Gradient Descent (SGD) in varying scenarios. The aim of the numerical experiments is to critically assess in which settings the preconditioning is able to improve the loss decrease w.r.t. iterations and computation time during training, and to understand the behavior of the preconditioner during training. The idea for the construction of the preconditioner evolved from the article [58] with the title "Frobenius-type norms and inner products of matrices and linear maps with applications to neural network training".

5.1. Introduction

Preconditioning in deep learning has been a topic of significant interest due to its potential to stabilize and accelerate the training process. Various methods have been proposed, ranging from Quasi-Newton (QN) methods, such as Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) and trust-region methods (see e.g. [98, Chapter 6 and 2.2]), to Hessian-Free Optimization and Natural Gradient Descent (NGD), which leverages the Fisher Information Matrix (FIM). Approximations of NGD, like K-FAC [91], have also gained popularity. Additionally, diagonal preconditioning methods, such as Adagrad [27] and Adam [70], are widely used and can be interpreted as parameter-wise rescaling/preconditioning or adaptive learning rate schedules. Other layer-wise techniques with a similar aim as preconditioning, including batch normalization [66], weight normalization [111], and spectral normalization [93], have been explored to improve convergence. Other normalization methods, such as layer normalization [6] and group normalization [135], also contribute to this domain.

We introduce a general framework for layer-wise preconditioning of Feedforward Neural Networks based on inner products on the layer spaces, which includes many existing methods as special cases. Layer-wise preconditioning is more accurate than diagonal preconditioning methods since it captures interactions between parameters within a layer, while being less computationally expensive than full preconditioning on the parameter space. We introduce a layer-wise preconditioning method for

Feedforward Neural Networks based on the Frobenius-type inner product, which we call *Frobenius-type preconditioning*. After establishing this general framework, we explore specific data-driven choices of inner products based on covariance information, which eliminates the need for predefined inner products on the layer spaces. We assess the numerical performance of Frobenius preconditioning in comparison to unpreconditioned Stochastic Gradient Descent (SGD) and discuss differences to K-FAC (cf. Section 2.5.4).

5.2. Related Work

A common method for layer-wise preconditioning in neural network training is K-FAC [91], which approximates NGD (cf. Section 2.5.2) by a Kronecker-factored block-diagonal approximation of the FIM. Although K-FAC is one of the most popular layer-wise preconditioners, there is a body of related work that develops both extensions of K-FAC and alternative layer-wise preconditioning strategies. Below we first review direct extensions and practical variants of K-FAC, and then survey other approaches that use covariance or curvature information for layer-wise preconditioning. EK-FAC, proposed in [39], is a fast approximate NGD in a Kronecker-factored eigenbasis. A coordinate-free construction of scalable NGD, which is formulated in a coordinate-free manner and constructs a Riemannian metric where the natural gradient update equals the K-FAC update is given in [87]. To make K-FAC more efficient, [119] propose SK-FAC, which is a training method for neural networks with faster Kronecker-factored approximate curvature. The method TENGraD, which is proposed in [118] introduces a time-efficient NGD method with exact FIM block inversion in the block-diagonal approximation. In [117], a method called WoodFisher is proposed, which estimates the inverses of the FIM blocks via rank-one updates. The authors of [100] develop K-FAC further to get a more practical algorithm for large-scale problems. Finally, [137] shows convergence results for K-FAC applied to two-layer ReLU-networks.

Other related work not directly related to K-FAC includes [48], which introduces Shampoo, a method for layer-wise preconditioning of trainable parameters of a neural network architecture when training with stochastic gradient descent using a matrix preconditioner that is separated into left and right matrices, and uses (weight) gradient-based updates for both matrices. Shampoo is closely related to the full version of AdaGrad and convergence guarantees in the convex setting are provided. A modification of the neural network architecture called natural neural networks is proposed in [24], which employs a whitening matrix and a centering vector in each layer (in the forward pass) to whiten (shift and rescale) the activations. These new elements are updated every few iterations by the newly available data. The concept of natural neural networks is extended in [34] by introducing a bi-directional whitening approach, which whitens the layers in both forward and backward passes. Both approaches aim to improve the training dynamics by incorporating covariance information.

There also exists work to incorporate curvature information into the layer-wise preconditioning. Since the FIM and the GGN matrix are closely related for common training problems (cf. Section 2.5.2), the approaches to incorporate curvature information are similar to the methods for approximating the FIM. The method proposed in [10] follows a similar approach as K-FAC, but instead of approximating the FIM, the authors approximate the GGN matrix in a layer-wise Kronecker-factored manner using low-rank approximations. Backpropagation is partially replaced with forward-mode automatic differentiation to compute the preconditioned gradient in [56]. To efficiently compute the preconditioned gradient,

[108] adds a Levenberg-Marquardt style damping term and employs the Sherman-Morrison-Woodbury (SMW) formula (cf. [115]). The work [41] develops a Kronecker-factored BFGS method (KBFGS) to train deep neural networks. Considering big mini-batch settings, [37] show that the stochastic GGN method is very efficient. Other work on layer-wise preconditioning that is more loosely related to the topic include i.e. [17, 7, 97, 138, 85, 81, 74].

A very recent work that proposes a framework of preconditioned matrix norms to develop new preconditioned gradient methods for neural network training is [123]. Contrary to our approach, they focus on developing a preconditioner based on an extension of the Adam method, while we focus on layer-wise preconditioning based on covariance-driven inner products.

5.3. Frobenius-type Preconditioning for Neural Networks

In this section, we introduce a layer-wise preconditioning method for Feedforward Neural Networks (FNNs) based on the Frobenius-type inner product, which we call *Frobenius-type preconditioning*. We start by deriving the preconditioned gradient for an exemplary layer of a FNN and extend this to a general framework for layer-wise preconditioning in Section 5.3.1. Then, we discuss covariance-driven inner products on the layer spaces in Section 5.3.2 and how they can be used to define Frobenius-type preconditioners. Finally, we discuss numerical observations of the behavior of Frobenius-type preconditioning with covariance inner products in Section 5.3.3.

5.3.1. General Framework

We derive an approach to precondition the updates of the weight matrices of the neural network training process in a layer-wise fashion. The main idea is to utilize inner products on the layer spaces to induce inner products on the space of weights.

As a reminder of (2.5), the forward pass of a FNN through the i-th layer is given by

$$y_i := W_i x_{i-1} \quad \text{and} \quad x_i := \sigma(y_i).$$

For now, we neglect the bias term in (2.5) to examine linear maps without an affine shift. The associated layer spaces of x_i and y_i are X_i, a post-activation space, and Y_i, a pre-activation space.

Feedforward Neural Networks can be interpreted as a combination of (affine) linear maps between the layer spaces

$$S_i : X_{i-1} \to Y_i \tag{5.1}$$

and activation maps

$$\sigma : Y_i \to X_i.$$

We assume in the following that the layer spaces X_{i-1} and Y_i are Hilbert spaces with inner products

$$(\cdot,\cdot)_{X_{i-1}} \quad \text{and} \quad (\cdot,\cdot)_{Y_i}$$

and associated Riesz maps (2.47)

$$\mathcal{R}_{X_{i-1}} : X_{i-1} \longrightarrow X_{i-1}^* \text{ and } \mathcal{R}_{Y_i} : Y_i \longrightarrow Y_i^*.$$

We derive the Frobenius-type preconditioned gradient introduced in Section 2.5.1 of the loss $\hat{f}$ of a FNN w.r.t. the linear map S_i defined in (5.1) of the neural network. For the remainder of this section, we assume that the trace operator is defined and finite for the involved operators, such that the Frobenius-type inner product is well-defined. We start by giving a general result and then discuss the finite-dimensional case, which is most relevant in practice.

Lemma 5.1 (General Frobenius-Type Preconditioned Gradient). *The gradient of the loss for one training datum w.r.t. the Frobenius-type inner product induced by the inner products on the layer spaces X_{i-1} and Y_i, is given by*

$$\nabla_{S_i}^{X_{i-1}\to Y_i} f_j(\theta) = \mathcal{R}_{Y_i}^{-1}\left(\frac{\partial f_j(\theta)}{\partial y_i}\right)\mathcal{R}_{X_{i-1}}(x_{i-1}) \in L(X_{i-1}, Y_i), \tag{5.2}$$

which maps $x \in X_{i-1}$ to

$$\nabla_{S_i}^{X_{i-1}\to Y_i} f_j(\theta)(x) = \underbrace{\mathcal{R}_{Y_i}^{-1}\left(\frac{\partial f_j(\theta)}{\partial y_i}\right)}_{\in Y_i}\underbrace{\left[\mathcal{R}_{X_{i-1}}(x_{i-1})\right](x)}_{\in\mathbb{R}} \in Y_i,$$

where $\frac{\partial f_j(\theta)}{\partial y_i} \in Y_i^*$ is the derivative of the loss function f_j w.r.t. the pre-activation features y_i of the i-th layer and $x_{i-1} \in X_{i-1}$ are the post-activation features of the $(i-1)$-th layer. We call $\nabla_{S_i}^{X_{i-1}\to Y_i} f_j(\theta)$ the ***Frobenius-type preconditioned gradient*** *of the loss function f_j w.r.t. the linear map S_i.*

Proof. We start at the derivative of the loss function f_j w.r.t. the linear map S_i to determine the general gradient. Let $\delta S \in L(X_{i-1}, Y_i)$ be a variation of the linear map S_i. Then the derivative of the loss function f_j w.r.t. the linear map S_i in direction δS, (see also (2.26)), is given by

$$\langle \frac{\partial f_j(\theta)}{\partial S_i}, \delta S\rangle = \frac{\partial f_j(\theta)}{\partial S_i}(\delta S) = \underbrace{\frac{\partial f_j(\theta)}{\partial y_i}(\delta S x_{i-1})}_{\in\mathbb{R}} = \operatorname{trace}\left(\frac{\partial f_j(\theta)}{\partial y_i}(\delta S x_{i-1})\right) = \operatorname{trace}\left(x_{i-1}\frac{\partial f_j(\theta)}{\partial y_i}\delta S\right),$$

where the last equality holds since the trace is invariant under cyclic permutations. The gradient $g \in L(X_{i-1}, Y_i)$ must satisfy

$$\langle \frac{\partial f_j(\theta)}{\partial S_i}, \delta S\rangle = (g, \delta S)_{X_{i-1}\to Y_i} \quad \forall \delta S \in L(X_{i-1}, Y_i).$$

Here, the Frobenius-type inner product $(\cdot,\cdot)_{X_{i-1}\to Y_i}$ reads

$$(g, \delta S)_{X_{i-1}\to Y_i} = \operatorname{trace}\left(\mathcal{R}_{X_{i-1}}^{-1} g' \mathcal{R}_{Y_i}\delta S\right) = \operatorname{trace}\left(\mathcal{R}_{X_{i-1}}^{-1}\mathcal{R}_{X_{i-1}} g^* \mathcal{R}_{Y_i}^{-1}\mathcal{R}_{Y_i}\delta S\right) = \operatorname{trace}\left(g^*\delta S\right).$$

It follows, that the Hilbert space adjoint $g^* \in L(Y_i, X_{i-1})$ (see (2.42) for a definition) of the gradient g is given by

$$g^* = x_{i-1}\frac{\partial f_j(\theta)}{\partial y_i}.$$

Here, the dependency on the inner products is not visible in the expression of the Hilbert space adjoint g^*, due to the definition of the Frobenius-type inner product. The dual map $g' \in L(Y_i^*, X_{i-1}^*)$ (see (2.43) for a definition) of the gradient g is then given by

$$g' = \mathcal{R}_{X_{i-1}} x_{i-1}\frac{\partial f_j(\theta)}{\partial y_i}\mathcal{R}_{Y_i}^{-1}.$$

From this we can deduce that the gradient has the form

$$g = \left(\mathcal{R}_{Y_i}^{-1}\frac{\partial f_j(\theta)}{\partial y_i}\right)(\mathcal{R}_{X_{i-1}} x_{i-1}),$$

which evaluated at a point $x \in X_{i-1}$ equals to

$$g(x) = \left(\mathcal{R}_{Y_i}^{-1}\frac{\partial f_j(\theta)}{\partial y_i}\right)(\mathcal{R}_{X_{i-1}} x_{i-1})(x) \in Y_i.$$

□

In classical machine learning, the layer spaces are finite-dimensional vector spaces of the type $\mathbb{R}^n$ and the linear map S_i can be represented by a weight matrix W_i (w.r.t. the canonical basis of $\mathbb{R}^n$). For the finite-dimensional case, given orthonormal bases $(\psi_j)_{j=1,\dots,n_{X_{i-1}}}$ for X_{i-1} and $(\phi_j)_{j=1,\dots,n_{Y_i}}$ for Y_i, the inner products on the layer spaces X_{i-1} and Y_i can be represented by symmetric and positive definite matrices $R_{X_{i-1}}$ and R_{Y_i}, such that

$$(x_1, x_2)_{X_{i-1}} = x_1^T R_{X_{i-1}} x_2 \text{ and } (y_1, y_2)_{Y_i} = y_1^T R_{Y_i} y_2. \tag{5.3}$$

This allows to represent the preconditioned gradient as follows.

Lemma 5.2 (Frobenius-Type Preconditioned Gradient in Finite-Dimensional Layer Spaces). *Assume that the layer spaces are equipped with orthonormal bases as introduced above. The gradient w.r.t. the Frobenius-type inner product induced by the inner products on the finite-dimensional layer spaces X_{i-1} and Y_i (w.r.t. the bases $(\psi_j)_{j=1,\dots,n_{X_{i-1}}}$ and $(\phi_j)_{j=1,\dots,n_{Y_i}}$) for one training datum w.r.t. the weight matrix W_i, is given by*

$$\nabla_{W_i}^{X_{i-1}\to Y_i} f_j(\theta) = R_{Y_i}^{-1}\nabla_{y_i} f_j(\theta) x_{i-1}^T R_{X_{i-1}} \tag{5.4}$$

with the notation of (5.3)*. Here, $\nabla_{y_i} f_j(\theta) \in \mathbb{R}^{n_{Y_i}}$ and $x_{i-1} \in \mathbb{R}^{n_{X_{i-1}}}$ are the coordinate representations of $\frac{\partial f_j(\theta)}{\partial y_i}$ and x_{i-1} w.r.t. the chosen bases of the layer spaces.*

Proof. The dual of the pre-activation space Y_i^* is spanned by the basis $(\bar{\phi}_i)_{i=1,..,n_{Y_i}}$ and the post-activation space X_{i-1} is spanned by the basis $(\psi_j)_{j=1,..,n_{X_{i-1}}}$. We can express the pre-activation dual object $\frac{\partial f_j(\theta)}{\partial y_i}$

and the post-activation features x_{i-1} as

$$\frac{\partial f_j(\theta)}{\partial y_i} = \sum_{i=1}^{n_{Y_i}} \beta_i \bar{\phi}_i =: \beta^T \bar{\phi}, \tag{5.5}$$

$$x_{i-1} = \sum_{j=1}^{n_{X_{i-1}}} \alpha_j \psi_j =: \alpha^T \psi. \tag{5.6}$$

β are the coefficients of the dual features $\frac{\partial f_j(\theta)}{\partial y_i}$ w.r.t. the basis of the dual of the pre-activation space Y_i^* and α are the coefficients of the post-activation features x_{i-1} w.r.t. the basis of the post-activation space X_{i-1}. The matrix representing the Riesz map $\mathcal{R}_{X_{i-1}}$ in the basis $(\psi_j)_{j=1,..,n_{X_{i-1}}}$ of the post-activation space X_{i-1} can be constructed by

$$R_{X_{i-1}} = \begin{pmatrix} \langle \mathcal{R}_{X_{i-1}}\psi_1 , \psi_1\rangle & \langle \mathcal{R}_{X_{i-1}}\psi_1 , \psi_2\rangle & \cdots & \langle \mathcal{R}_{X_{i-1}}\psi_1 , \psi_{n_{X_{i-1}}}\rangle \\ \langle \mathcal{R}_{X_{i-1}}\psi_2 , \psi_1\rangle & \langle \mathcal{R}_{X_{i-1}}\psi_2 , \psi_2\rangle & \cdots & \langle \mathcal{R}_{X_{i-1}}\psi_2 , \psi_{n_{X_{i-1}}}\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathcal{R}_{X_{i-1}}\psi_{n_{X_{i-1}}} , \psi_1\rangle & \langle \mathcal{R}_{X_{i-1}}\psi_{n_{X_{i-1}}} , \psi_2\rangle & \cdots & \langle \mathcal{R}_{X_{i-1}}\psi_{n_{X_{i-1}}} , \psi_{n_{X_{i-1}}}\rangle \end{pmatrix},$$

where

$$(R_{X_{i-1}})_{i,j} := \langle \mathcal{R}_{X_{i-1}}\psi_i , \psi_j\rangle = (\psi_i, \psi_j)_{X_{i-1}}.$$

Note that the matrix $R_{X_{i-1}}$ depends on the choice of basis of the post-activation space X_{i-1} and is symmetric and positive definite (spd), inheriting these properties from the fact that every inner product must be symmetric and positive definite. Let

$$x_1, x_2 \in X_{i-1} \text{ with coefficients } \alpha_1, \alpha_2 \in \mathbb{R}^{n_{X_{i-1}}}$$
$$\text{and representations} \quad x_1 = \sum_{j=1}^{n_{X_{i-1}}} \alpha_{1,j}\psi_j, x_2 = \sum_{j=1}^{n_{X_{i-1}}} \alpha_{2,j}\psi_j.$$

Leveraging the linearity of the Riesz map, the following connection holds for x_1, x_2:

$$\begin{aligned} \langle \mathcal{R}_{X_{i-1}}(x_1) , x_2\rangle &= \langle \mathcal{R}_{X_{i-1}}(x_1) , \sum_{j=1}^{n_{X_{i-1}}} \alpha_{2,j}\psi_j\rangle \\ &= \sum_{j=1}^{n_{X_{i-1}}} \alpha_{2,j} \langle \mathcal{R}_{X_{i-1}}(x_1) , \psi_j\rangle \\ &= \sum_{j=1}^{n_{X_{i-1}}} \alpha_{2,j} \langle \mathcal{R}_{X_{i-1}}(\sum_{i=1}^{n_{X_{i-1}}} \alpha_{1,i}\psi_i) , \psi_j\rangle \\ &= \sum_{j=1}^{n_{X_{i-1}}} \alpha_{2,j} \sum_{i=1}^{n_{X_{i-1}}} \alpha_{1,i} \langle \mathcal{R}_{X_{i-1}}(\psi_i) , \psi_j\rangle \\ &= \sum_{j=1}^{n_{X_{i-1}}} \alpha_{2,j} \sum_{i=1}^{n_{X_{i-1}}} \alpha_{1,i} (R_{X_{i-1}})_{i,j} \\ &= \alpha_1^T R_{X_{i-1}} \alpha_2. \end{aligned}$$

Hence, we can interpret the Riesz map $\mathcal{R}_{X_{i-1}}$ as an inner product on the post-activation space X_{i-1} and

$$\mathcal{R}_{X_{i-1}}(x_1) = \alpha_1^T R_{X_{i-1}}. \tag{5.7}$$

In the same manner we can derive a Riesz matrix R_{Y_i} for the pre-activation space Y_i using the associated primal basis $\{\phi_i\}_{i=1}^{n_{Y_i}}$ of the pre-activation space Y_i.

$$R_{Y_i} = \begin{pmatrix} \langle \mathcal{R}_{Y_i}\phi_1 , \phi_1\rangle & \langle \mathcal{R}_{Y_i}\phi_1 , \phi_2\rangle & \cdots & \langle \mathcal{R}_{Y_i}\phi_1 , \phi_{n_{Y_i}}\rangle \\ \langle \mathcal{R}_{Y_i}\phi_2 , \phi_1\rangle & \langle \mathcal{R}_{Y_i}\phi_2 , \phi_2\rangle & \cdots & \langle \mathcal{R}_{Y_i}\phi_2 , \phi_{n_{Y_i}}\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathcal{R}_{Y_i}\phi_{n_{Y_i}} , \phi_1\rangle & \langle \mathcal{R}_{Y_i}\phi_{n_{Y_i}} , \phi_2\rangle & \cdots & \langle \mathcal{R}_{Y_i}\phi_{n_{Y_i}} , \phi_{n_{Y_i}}\rangle \end{pmatrix},$$

where

$$(R_{Y_i})_{i,j} \coloneqq \langle \mathcal{R}_{Y_i}\phi_i , \phi_j\rangle = (\phi_i, \phi_j)_{Y_i}.$$

The matrix M constructed by

$$(M)_{i,j} \coloneqq \langle \bar{\phi}_i , \mathcal{R}_{Y_i}^{-1}(\bar{\phi}_j)\rangle$$

is indeed the inverse of the Riesz matrix R_{Y_i}, i.e.

$$M = R_{Y_i}^{-1}.$$

To see this, use the property of the dual basis that $\langle \bar{\phi}_i , \phi_j\rangle = \delta_{i,j}$, where $\delta_{i,j}$ is the Kronecker delta function and that

$$\langle \ell , y\rangle = \langle \ell , \mathcal{R}_{Y_i}^{-1}\mathcal{R}_{Y_i} y\rangle \quad \forall \ell \in Y_i^*, \forall y \in Y_i.$$

Let

$$\begin{aligned} &\ell_1, \ell_2 \in Y_i^* \text{ with coefficients } \beta_1, \beta_2 \in \mathbb{R}^{n_{Y_i}} \\ &\text{and representations } \ell_1 = \sum_{i=1}^{n_{Y_i}} \beta_{1,i}\bar{\phi}_i, \quad \ell_2 = \sum_{i=1}^{n_{Y_i}} \beta_{2,i}\bar{\phi}_i. \end{aligned}$$

Leveraging the linearity of the Riesz map, the following connection holds for ℓ_1, ℓ_2:

$$\begin{aligned}
&\langle \ell_2 , \mathcal{R}_{Y_i}^{-1}(\ell_1)\rangle \\
&= \langle \ell_2 , \sum_{i=1}^{n_{Y_i}} \beta_{1,i}\mathcal{R}_{Y_i}^{-1}(\bar{\phi}_i)\rangle \\
&= \sum_{i=1}^{n_{Y_i}} \beta_{1,i}\langle \ell_2 , \mathcal{R}_{Y_i}^{-1}(\bar{\phi}_i)\rangle \\
&= \sum_{i=1}^{n_{Y_i}} \beta_{1,i}\langle \sum_{j=1}^{n_{Y_i}} \beta_{2,j}\bar{\phi}_j , \mathcal{R}_{Y_i}^{-1}(\bar{\phi}_i)\rangle \\
&= \sum_{i=1}^{n_{Y_i}} \beta_{1,i} \sum_{j=1}^{n_{Y_i}} \beta_{2,j}\langle \bar{\phi}_j , \mathcal{R}_{Y_i}^{-1}(\bar{\phi}_i)\rangle \\
&= \sum_{i=1}^{n_{Y_i}} \beta_{1,i} \sum_{j=1}^{n_{Y_i}} \beta_{2,j}(M)_{j,i} \\
&= \beta_2^T R_{Y_i}^{-1}\beta_1.
\end{aligned}$$

Hence, we can interpret the Riesz map $\mathcal{R}_{Y_i}^{-1}$ as an inner product on the dual space of the pre-activation space Y_i^* and

$$\mathcal{R}_{Y_i}^{-1}(\ell_1) = R_{Y_i}^{-1}\beta_1.$$

Using the notation from (5.5), (5.6) and Lemma 5.2, we can express the gradient of the loss function w.r.t. the weight matrix W_i as

$$-\nabla_{W_i} f_j(\theta) = -R_{Y_i}^{-1}\beta\alpha^T R_{X_{i-1}},$$

where $R_{X_{i-1}}$ is the Riesz matrix associated with the basis on the post-activation space X_{i-1} and R_{Y_i} is the Riesz matrix associated with the basis on the pre-activation space Y_i. This shows the representation in (5.4).

□

Remark 5.3 (Preconditioned Gradient of the Bias Vector). *The Frobenius-type preconditioned gradient w.r.t. the bias vector b_i of the i-th layer can be derived in a similar manner. It only depends on the inner product of the pre-activation space Y_i and is given by*

$$\nabla_{b_i} f_j(\theta) = R_{Y_i}^{-1}\beta = R_{Y_i}^{-1}\nabla_{y_i} f_j(\theta).$$

An alternative approach to compute the Frobenius-type preconditioned gradient (or any preconditioned gradient in general) is to find the minimizer of the following optimization problem. The negative Euclidean gradient of the loss function w.r.t. the weight matrix W_i is the solution of the optimization problem

$$\text{Minimize } \frac{\partial f(\theta)}{\partial W_i}\delta W + \frac{1}{2}(\delta W, \delta W)_F \text{ over } \delta W,$$

where $(\cdot,\cdot)_F$ is the Frobenius inner product.

Using the more general Frobenius-type inner product instead of the classical Frobenius inner product in the second term of the objective function, we can get the negative Frobenius-type preconditioned gradient $-G$ as described in Lemma 5.1 as the solution of the optimization problem

$$\text{Minimize } \frac{\partial f(\theta)}{\partial W_i}\delta W + \frac{1}{2}(\delta W, \delta W)_{X_{i-1}\to Y_i} \text{ over } \delta W.$$

This is true because the optimization problem is convex and the solution fulfills the first-order optimality condition

$$\frac{\partial f(\theta)}{\partial W_i}\delta W + (\delta W, -G)_{X_{i-1}\to Y_i} = 0 \;\forall\; \delta W,$$

which is equivalent to

$$\Leftrightarrow \langle \frac{\partial f(\theta)}{\partial W_i}, \delta W \rangle = (\delta W, G)_{X_{i-1}\to Y_i} \;\forall\; \delta W,$$

which is the definition of the gradient w.r.t. the inner product $(\cdot,\cdot)_{X_{i-1}\to Y_i}$.

There are many possibilities for equipping the layer spaces with inner products, and thereby inducing inner products on the space of linear maps. Different choices of inner products on the layer spaces X_{i-1} and Y_i induce different Frobenius-type preconditioners. A classical choice are *predefined inner products* on the layer spaces which are fixed during training, given by the knowledge that the finite-dimensional layer space has a specific a-priori known structure. These inner products are generally independent of the training data and the optimization variable θ.

Example 5.4 (Sobolev Inner Products). *For machine learning problems with a specific task such as learning a Partial Differential Equation solution operator, non-Euclidean inner products resembling discretized Sobolev inner products (see e.g. [62]) can be used on the first and last layer spaces X_0, Y_L to incorporate a-priori information. The layer spaces of the hidden layers often do not have a known structure, hence Euclidean inner products are used on the hidden layer spaces X_{i-1}, Y_i for $i = 2, \dots, L-1$.*

Even though predefined inner products exist and can be leveraged in some training cases, often underlying structures are unknown or too complex to be incorporated in a fixed inner product. Also, predefined inner products might not be able to capture relevant structures of the layer spaces which change during training.

5.3.2. Preconditioning with Covariance Information

An alternative to predefined inner products, which circumvents the issues of no or insufficient a-priori knowledge about the layer spaces and their changing structure during training are *data-driven inner products* which change with the iterations on the layer spaces, given by e.g. covariance or curvature information of the data. In this section we consider inner products on the layer spaces driven by covariance information of the layer data.

We examine inner products on the layer spaces X_{i-1} and Y_i which are constructed by available data on the spaces. The easiest available data on the layer spaces is covariance information of the layer data, more specifically samples of the random variable representing the layer data on X_{i-1} and Y_i, which is induced by the distribution of the training data and the current optimization variable θ. We consider the data samples of the forward and backward pass which appear in the Euclidean gradient of the weight matrix (2.28) and use it to compute covariance estimates of the random variables representing the layer data. The covariance estimates are then used to define inner products on the layer spaces.

Since a sample or realization of the random variable x_{i-1} is generated by the forward pass through the neural network, it lies in the layer space X_{i-1}. A realization of the random variable $\frac{\partial f_j(\theta)}{\partial y_i}$ is generated by the forward and backward pass through the neural network and has values in the dual space of the layer space Y_i. Because of the different structure of the samples, we examine both layer spaces separately. We start with the post-activation layer space X_{i-1} and then consider derivative information on the pre-activation layer space Y_i.

Inner Product on the Post-Activation Space X_{i-1}

Before we construct the covariance-based inner product on the layer space X_{i-1}, we shortly recall the definition of the covariance operator for a zero-mean random variable in the general infinite-dimensional case in order to understand the connection of the covariance to inner products on the layer space.

For a zero-mean random variable x with values in the space X, the covariance operator (provided it exists and is finite) is given by

$$\mathbb{E}[xx^{**}] \in \mathcal{L}(X^*, X),$$

with bilinear form

$$c(\ell_1, \ell_2) = \mathbb{E}[\ell_1 x x^{**} \ell_2] = \mathbb{E}[\ell_1(x) x^{**}(\ell_2)] = \mathbb{E}[\ell_1(x)\ell_2(x)]$$

for $\ell_1, \ell_2 \in X^*$. Here, x^{**} is the second dual of x. The covariance operator is a symmetric positive semi-definite operator, i.e. for all $\ell \in X^*$ it holds that

$$\langle \ell, \mathbb{E}[xx^{**}]\ell \rangle = \mathbb{E}[\ell(x)^2] \geq 0,$$

and acts one the dual space X^* of X. For more details on covariance operators, we refer the reader to e.g. [121]. This implies that, given additional positive definiteness, the covariance of x is an inner product on the dual space X_{i-1}^*. Accordingly, we define the inner product on X_{i-1} as the inverse covariance of x. Keep in mind that in order to use the covariance operator as an inner product, it has to be invertible, i.e. positive definite, which is not guaranteed in general for covariance operators. In order to ensure invertibility, we add a small positive multiple of a positive definite base metric to the covariance operator.

In finite dimensions, when x is a random variable with values in $\mathbb{R}^{n_{x_{i-1}}}$, the covariance operator can be represented by a covariance matrix (2.2), which has the form

$$\operatorname{Cov}(x) \coloneqq \mathbb{E}[xx^T].$$

For a random variable x with non-zero mean, the covariance operator is not a linear operator anymore, but an affine linear operator and still acts on the dual space X^* of X. The associated covariance matrix (2.2) is given by

$$\operatorname{Cov}(x) \coloneqq \mathbb{E}[(x - \mathbb{E}[x])(x - \mathbb{E}[x])^T],$$

where $\mathbb{E}[x] \in X$ is the expectation (or mean) (2.1) of the random variable x. A simple base metric to ensure positive definiteness is the Euclidean inner product represented by a scaled identity matrix $\gamma \mathrm{id}$ with a small positive regularization parameter $\gamma > 0$. For a random variable with non-zero mean, an alternative to the covariance is to use the second moment $\mathbb{E}[xx^T]$ as an inner product on the dual space X. For a zero-mean random variable, the second moment and the covariance coincide.

We now apply this information about the covariance operator to define an inner product on the finite-dimensional post-activation layer space X_{i-1} using the random variable representing the post-activation features x_{i-1}. Since the exact covariance is generally not computable in practice, we formulate empirical approximations of the covariance using samples $x_{i-1}(j)$ of the random variable x_{i-1}, where j is the sample index.

We start by fixing the trainable parameters θ. The covariance matrix of the random variable x_{i-1} is given by

$$\operatorname{Cov}(x_{i-1}) = \mathbb{E}\left[(x_{i-1} - \mathbb{E}[x_{i-1}])(x_{i-1} - \mathbb{E}[x_{i-1}])^T\right],$$

since x_{i-1} is generally a random variable with non-zero mean. The expectation is taken w.r.t. the distribution of x_{i-1} and hence is not exactly computable in practice. The exact covariance can be approximated by the covariance matrix estimator (2.4)

$$\hat{\operatorname{Cov}}(x_{i-1}) \coloneqq \frac{1}{N} \sum_{j=1}^{N} (x_{i-1}(j) - \overline{x}_{i-1})(x_{i-1}(j) - \overline{x}_{i-1})^T \tag{5.8}$$

over a batch of N i.i.d. samples drawn from the distribution of x_{i-1}, where $x_{i-1}(j)$ is the j-th sample of the batch. The quantity $\overline{x}_{i-1}$ denotes the empirical mean (2.3) over the batch of samples

$$\overline{x}_{i-1} \coloneqq \frac{1}{N} \sum_{j=1}^{N} x_{i-1}(j). \tag{5.9}$$

The samples are taken from the distribution of the post-activation layer $p(x_{i-1} \mid x_0, \theta)$ which is induced by the distribution of the input training data x_0 denoted by $p(x_0)$ and by the current parameters θ and the forward pass through the neural network. In order to draw a sample $x_{i-1}(j)$ from the distribution of the post-activation layer, we first draw a sample of the input data $x_{0,j}$ by sampling from training data input distribution $p(x_0)$ and then compute the forward pass for $x_{0,j}$ with the current parameters θ up until the post-activation layer X_{i-1} to get the sample $x_{i-1}(j)$. Note that the distribution of x_{i-1} depends only on the distribution of the input data and the current optimization variable θ and not on the distribution of the labels, which will not be the case for the pre-activation layer Y_i.

To ensure invertibility of the matrix representing the inner product, we finally set

$$R^{-1}_{X_{i-1}}(\theta) \coloneqq \left(\hat{\mathrm{Cov}}(x_{i-1}) + \gamma I\right)^{-1}. \tag{5.10}$$

The inner product (5.10) changes with each iteration since x_{i-1} depends on the optimization variable θ (and on the mini-batch samples). There exist various possibilities how to deal with this changing inner product in practice.

The first and most straightforward option is to recompute the covariance matrix from scratch in each iteration using the current mini-batch of samples and invert it for the inner product. The samples needed for the covariance computation are already generated during the forward pass of the current mini-batch. At iteration k, this leads to

$$R^{(k)}_{X_{i-1}} \coloneqq \left(\hat{\mathrm{Cov}}_{\text{mini-batch from iteration k}}(x_{i-1}) + \gamma I\right)^{-1}. \tag{5.11}$$

This ensures that the inner product is always up-to-date with the current optimization variable θ but can be computationally expensive. Alternatively, one could also perform a forward pass with a different mini-batch of input data samples just for the covariance computation. But, this would increase the computational cost even more.

When one assumes that the optimization variable θ changes only slightly in each iteration, we can also reuse covariance information from previous iterations to save computational resources and get a potentially more stable estimate of the covariance matrix. An option is to perform an exponential averaging of the covariance matrix from the current mini-batch and the covariance matrix from the last iteration. Here, $\beta \in [0,1]$ is a hyperparameter controlling the influence of the new covariance estimate from the current mini-batch. At iteration k, this leads to

$$R^{(k)}_{X_{i-1}} \coloneqq \left((1-\beta)\hat{\mathrm{Cov}}_{\text{used in last iteration}}(x_{i-1}) + \beta\hat{\mathrm{Cov}}_{\text{mini-batch from iteration k}}(x_{i-1}) + \gamma I\right)^{-1}.$$

Alternatively we can perform exponential averaging of the inverse covariance matrix from the current mini-batch and the inverse covariance matrix from the last iteration. Here, $\beta \in [0,1]$ is again a hyperparameter controlling the influence of the new inverse covariance estimate from the current mini-batch. At iteration k, this leads to

$$R^{(k)}_{X_{i-1}} \coloneqq (1-\beta)R^{(k-1)}_{X_{i-1}} + \beta\left(\hat{\mathrm{Cov}}_{\text{mini-batch from iteration k}}(x_{i-1}) + \gamma I\right)^{-1}.$$

For both exponential averaging options, the initialization of the covariance matrix in the first iteration $R^{(0)}_{X_{i-1}}$ can be done by computing the covariance from the first mini-batch. Additionally, it is also possible to update the covariance matrix only every K iterations to save computational resources for all options mentioned above.

After having established an inner product on the post-activation layer space X_{i-1}, we now consider the pre-activation layer space Y_i and define an inner product there using covariance information of the backward data on the pre-activation layer. It turns out that this covariance data forms an inner product on the double dual space of the pre-activation layer space Y_i, contrary to the post-activation layer.

Inner Product on the Pre-Activation Space Y_i

Before we construct the covariance-based inner product on the layer space Y_i, we shortly recall the definition of the covariance operator for a zero-mean random variable with values in the dual space in order to understand the connection of the covariance to inner products on the layer space.

For a zero-mean random variable ℓ with values in the dual space Y^*, the covariance operator is given by

$$\mathbb{E}[\ell\ell^{**}] \in \mathcal{L}(Y^{**}, Y^*),$$

with bilinear form

$$c(\hat{y}_1, \hat{y}_2) = \mathbb{E}[\hat{y}_1(\ell)\ell^{**}(\hat{y}_2)] = \mathbb{E}[\langle \ell, y_1\rangle\langle \ell, y_2\rangle]$$

for $\hat{y}_1, \hat{y}_2 \in Y_i^{**}$ and their primal representatives $y_1, y_2 \in Y_i$. We see that the covariance operator acts on the double dual space Y_i^{**}. Using the isomorphism between the double dual and the primal space, we deduce that the covariance of ℓ, given additional positive definiteness, is an inner product on the primal space Y_i, in contrast to the post-activation layer case where the covariance formed an inner product on the dual space. Accordingly, we define the inner product on Y_i as the covariance of ℓ. Keep in mind that, also in this case, in order to use the covariance operator as an inner product, it has to be invertible, i.e. positive definite, which is not guaranteed in general. In order to ensure invertibility, we add a small positive multiple of a positive definite base metric to the covariance operator.

When $Y = \mathbb{R}^{n_{Y_i}}$, the covariance operator can be represented by a covariance matrix, which has the form

$$\mathrm{Cov}(\ell) := \mathbb{E}[\ell^\top \ell],$$

where $\ell \in Y^*$ is interpreted as a row vector in $\mathbb{R}^{n_{Y_i}}$. Since we consider a setting where no known a-priori structure on Y_i exists yet, we use the Euclidean inner product on Y_i. Using the Riesz representer $y_\ell = \mathcal{R}_Y^{-1}(\ell) \in Y$ of the random variable ℓ, which is the Euclidean gradient of ℓ in this setting, we can equivalently express the covariance matrix in terms of y_ℓ as

$$\mathrm{Cov}(\ell) := \mathbb{E}[y_\ell y_\ell^T].$$

For a random variable ℓ with non-zero mean, the covariance is an affine linear operator. The associated covariance matrix (formulated in terms of the Euclidean gradient) is still given by

$$\mathrm{Cov}(\ell) := \mathbb{E}[(y_\ell - \bar{y}_\ell)(y_\ell - \bar{y}_\ell)^T],$$

where $\bar{y}_\ell := \mathbb{E}[y_\ell] \in Y_i$ is the mean of the random variable y_ℓ. A simple base metric to ensure positive definiteness is the Euclidean inner product represented by a scaled identity matrix $\gamma\mathrm{id}$ with a small positive regularization parameter $\gamma > 0$. For a random variable with a non-zero mean, an alternative to the covariance is to use the second moment

$$\mathbb{E}[y_\ell y_\ell^T]$$

as an inner product on the primal space Y_i. For a zero-mean random variable, the second moment and the covariance coincide.

Now we apply this to define an inner product on the finite-dimensional pre-activation layer space Y_i using the random variable representing the derivative information $\frac{\partial f(\theta)}{\partial y_i}$. Since the exact covariance is generally not computable in practice, we formulate empirical approximations of the covariance using samples $\nabla_{y_i} f(\theta)(j)$ of the random variable $\nabla_{y_i} f(\theta)$.

We start by fixing the trainable parameters θ. The covariance matrix of the random variable $\nabla_{y_i} f(\theta)$ is given by

$$\mathrm{Cov}(\nabla_{y_i} f(\theta)) = \mathbb{E}\left[(\nabla_{y_i} f(\theta) - \mathbb{E}[\nabla_{y_i} f(\theta)])(\nabla_{y_i} f(\theta) - \mathbb{E}[\nabla_{y_i} f(\theta)])^T\right],$$

since $\nabla_{y_i} f(\theta)$ is generally a random variable with non-zero mean . The expectation is taken w.r.t. the distribution of $\nabla_{y_i} f(\theta)$ and hence is not exactly computable in practice. The exact covariance can be approximated by the covariance matrix estimator

$$\hat{\mathrm{Cov}}(\nabla_{y_i} f(\theta)) := \frac{1}{N}\sum_{j=1}^{N}(\nabla_{y_i} f(\theta)(j) - \overline{\nabla_{y_i} f(\theta)})(\nabla_{y_i} f(\theta)(j) - \overline{\nabla_{y_i} f(\theta)})^T \tag{5.12}$$

over a batch of N i.i.d. samples drawn from the distribution of $\nabla_{y_i} f(\theta)$, where $\nabla_{y_i} f(\theta)(j)$ is the j-th gradient sample of the batch. The quantity $\overline{\nabla_{y_i} f(\theta)}$ denotes the empirical mean over the batch of samples

$$\overline{\nabla_{y_i} f(\theta)} := \frac{1}{N}\sum_{j=1}^{N} \nabla_{y_i} f(\theta)(j). \tag{5.13}$$

The samples are drawn from the distribution of the gradient on the pre-activation layer $p(\nabla_{y_i} f(\theta) \mid x_0, y_{\mathrm{label}}, \theta)$ which is induced by the distribution of the training data x_0, y_{label} and by the current parameters θ and the forward and backward pass through the neural network. We assume the following model of the training data distribution, which is often used for parameter estimation problems:

$$p_\theta(y \mid x) = p(y \mid g(\theta, x)),$$

i.e. we assume that the distribution of y is parametrized by θ through the model g. In the following, we use the relation $p_\theta(x, y) = p_\theta(x)p_\theta(y|x) = p(x)p(y \mid g(\theta, x))$. In order to draw a sample $\nabla_{y_i} f(\theta)(j)$ from the distribution of the gradient on the pre-activation layer, we first draw a sample of the input data $x_{0,j}$ by sampling from training data input distribution $p(x_0)$ and compute a forward pass for $x_{0,j}$ with the current parameters θ through the whole network to sample from the training data output distribution given the input and the model parameters $p(y|x_{0,j}, \theta) = p(y|g(\theta, x_{0,j}))$ to get a sample of the label data $y_{\mathrm{label}j}$ needed for the backward pass. Then, we compute the backward pass with the current parameters and model output given the sampled label (i.e. we replace the training label by the obtained sample) to get the sample $\nabla_{y_i} f(\theta)(j)$. Note that the distribution of $\nabla_{y_i} f(\theta)$ depends on the distribution of the input data, the labels, and the current optimization variable θ. In order to compute samples from $\nabla_{y_i} f(\theta)$, an additional backward pass is necessary compared to the post-activation layer case. A more detailed description of how to sample from the output distribution $p(y|x,\theta)$ is given in Section 2.5.3.

Remark 5.5 (Distribution of Label Data)**.** *It is also possible to assume that the training data distribution $p(x,y)$ is fixed and independent of the model parameters θ. Then, we can directly sample from the training data labels y_{labelj} instead of sampling from the output distribution $p(y|x,\theta)$.*

In order to ensure invertibility of the matrix representing the inner product, we finally set

$$R_{Y_i}(\theta) := \hat{\mathrm{Cov}}(\nabla_{y_i} f_j(\theta)) + \gamma I \tag{5.14}$$

and accordingly

$$R_{Y_i}^{-1}(\theta) := \left(\hat{\mathrm{Cov}}(\nabla_{y_i} f_j(\theta)) + \gamma I\right)^{-1}.$$

The inner product (5.14) changes with each iteration since $\nabla_{y_i} f_j(\theta)$ depends on the optimization variable θ. There exist various possibilities how to deal with this changing inner product in practice. The options are analogous to the ones described for the post-activation layer space X_{i-1}, wherefore we do not repeat them here, but refer to the last paragraph.

To summarize, the Frobenius-type data-driven preconditioned gradient w.r.t. the weight matrix W_i for one training datum is given by

$$\begin{aligned}\nabla_{W_i}^{X_{i-1}\to Y_i} f_j(\theta) &= R_{Y_i}^{-1} \nabla_{W_i}^{\mathrm{eucl}} f_j(\theta) R_{X_{i-1}} \\ &= \left(\hat{\mathrm{Cov}}(\nabla_{y_i}^{\mathrm{eucl}} f(\theta)) + \gamma_1 I\right)^{-1} \nabla_{y_i}^{\mathrm{eucl}} f_j(\theta) x_{i-1}^T \left(\hat{\mathrm{Cov}}(x_{i-1}) + \gamma_2 I\right)^{-1}.\end{aligned}$$

For a mini-batch of B training data, the Frobenius-type preconditioned gradient is given by

$$\nabla_{W_i}^{X_{i-1}\to Y_i} \hat{f}(\theta) = \left(\hat{\mathrm{Cov}}(\nabla_{Y_i}^{\mathrm{eucl}} f(\theta)) + \gamma_1 I\right)^{-1} \frac{1}{B} \sum_{j=1}^{B} \nabla_{Y_i}^{\mathrm{eucl}} f_j(\theta) x_{i-1}(j)^T \left(\hat{\mathrm{Cov}}(x_{i-1}) + \gamma_2 I\right)^{-1}. \tag{5.15}$$

Multiplying the respective gradient components by the inverse covariance suppresses gradient components aligned with high-variance directions while preserving or amplifying components aligned with low-variance directions.

Relation to Kronecker-Factored Approximate Curvature (K-FAC)

The covariance-driven Frobenius-type preconditioner (5.15) was inspired by inner products on the space of linear maps which were constructed with covariance information. The resulting preconditioner has a strong resemblance to the backbone of the well-known K-FAC method [91]. K-FAC is an approximation of the NGD preconditioner tailored to the neural network architecture which uses, among others, a Kronecker-factored approximation of the FIM. This leads to a preconditioned gradient which has the same basic structure as the Frobenius-type preconditioned gradient given in (5.15), but uses second moment estimates instead of covariance estimates for the preconditioning. Hence, K-FAC can be interpreted as a specific choice of the Frobenius-type preconditioner using (damped) second moment estimates as inner products on the layer spaces. Generally, the Frobenius-type preconditioning framework allows for a wider range of inner products on the layer spaces using covariance or second moment information for an example. Further, K-FAC adds various additional heuristics to work well in practice, such as a more intricate damping strategy, momentum, and rescaling of the update direction. A more detailed description of K-FAC and its derivation from the NGD method is given in Section 2.5.4.

Covariance Estimation

Updating the preconditioning matrices R_{Y_i} and $R_{X_{i-1}}$ over the course of training is generally possible in many ways. The proposed method above is only one possible way to do so. Generally, many works in the literature use covariance or second moment information to precondition the gradient in neural network training; some methods build on the K-FAC method and some were developed earlier. The main difference between these methods lies in the way how the covariance or second moment estimates are computed and updated in the course of training. We summarize and categorize the main aspects in how all methods differ in the following list:

- **Time window** for covariance data: current mini-batch, last K iterations, or all previous iterations.
- **Data source and size**: exact expectation, empirical mean over current mini-batch, or empirical mean over an external sample (specify N).
- **Update frequency**: every iteration, every K iterations, or a single estimate.
- **Structural approximation**: dense, low-rank plus diagonal, or diagonal.
- **Inversion strategy**: update inverse directly (e.g. rank updates) or recompute inverse from covariance each update.
- **Damping**: none, scalar damping, or adaptive damping.
- **Inclusion of mean**: covariance (centered) or second moment (uncentered).

Generally, covariance-driven Frobenius-type preconditioners are concerned with the same choices as listed above.

Remark 5.6 (Relation to Quasi-Newton (QN) Ideas)**.** *For Hessian-Free Optimization, QN methods are a successful approach to approximate the Hessian matrix if it is too expensive to compute it directly. The idea is to use the curvature information of the loss function to approximate the Hessian matrix, and the methods rely on the secant condition as a backbone for their updates. For covariance matrices there exists no comparable concept of a secant condition, hence these methods can not be applied directly in this context. There exists a result in [41] which formulates a secant condition for covariance matrices and applies QN updates to covariance matrix approximations. However, this turns out to be a more expensive way to compute the inverse of the covariance matrix using the Sherman-Morrison-Woodbury (SMW) formula [115]. Hence, we will not consider QN update ideas for the covariance matrices in this work.*

5.3.3. Numerical Experiments

In this subsection, we present numerical experiments for Feedforward Neural Network (FNN) training using the Frobenius-type covariance-based preconditioner described in Section 5.3.2. We compare the performance of unpreconditioned SGD and preconditioned SGD using the Frobenius-type covariance-based preconditioner. Since there is no theoretical proof of good functionality of the proposed preconditioner, we investigate whether it leads to an improved training performance compared to unpreconditioned SGD and for which tasks this may be the case. We perform experiments on classification tasks and an autoencoder task using the MNIST dataset. This will also give insights into whether the good performance of K-FAC in practice is really due to the covariance-based preconditioning or rather due to additional heuristics used in K-FAC. To understand the mechanics of the covariance-based preconditioner during training, we also analyze the behavior of the gradients and inner products induced by the covariance estimates over the course of training.

The experiments were implemented using `PyTorch` [103] and BackPACK [23]. The detailed experiment setup is given in Appendix B.4.

Classification Task with ReLU-FNN

We start by considering a classification task on the MNIST dataset (cf. Figure 2.5) with flattened input images of size $28 \times 28 = 784$. We use a FNN with the architecture $784 - 100 - 100 - 100 - 10$ with ReLU activations, i.e. with 3 hidden layers of width 100 each. The network is trained to minimize the CE-loss using mini-batch SGD with and without the Frobenius-type covariance-based preconditioner. A batch size of 100 is used for all training runs. This setup is similar to the one used in [34] and uses commonly employed hyperparameter choices.

We employ second moment estimates instead of covariance estimates for the preconditioning and use a fixed damping term with $\gamma = 0.01$ for both covariance estimates. The estimates are updated in each iteration using the current mini-batch as described in (5.11). We implemented the Frobenius-type covariance-based preconditioner using the BackPACK [23] extension for PyTorch [103] to compute the second moment estimates (without damping). This is because BackPACK does not provide an implementation to compute covariance estimates directly. Hence, the samples needed for the estimate on the pre-activation layer are drawn as described in Section 2.5.3.

In order to eliminate the variability of the decrease of the loss function that is due to learning rate schedules, we decided to run the trainings with fixed learning rates for both unpreconditioned SGD and preconditioned SGD. We train the network for 30 epochs, leading to $18,000$ iterations with batch size 100. We compare the performance of unpreconditioned SGD and preconditioned SGD for a grid of fixed learning rates. Then, we select the best learning rate for both methods based on the lowest training loss achieved in the last iterations of training and compare them against each other. In the following, we will refer to these best-performing learning rates as *champion learning rates*.

Considering the decrease of the loss functions w.r.t. iterations in Figure 5.1, we see that the Frobenius-type covariance-based preconditioner does not perform better than unpreconditioned SGD for this task. Both champion methods achieve a similar decrease of the loss function w.r.t. iterations, with the unpreconditioned method being even slightly better. Additionally, a high noise level can be observed for both methods. Since the computational cost per iteration is higher for the preconditioned method due to the covariance estimates, the unpreconditioned method outperforms the preconditioned method also w.r.t. wall-clock time.

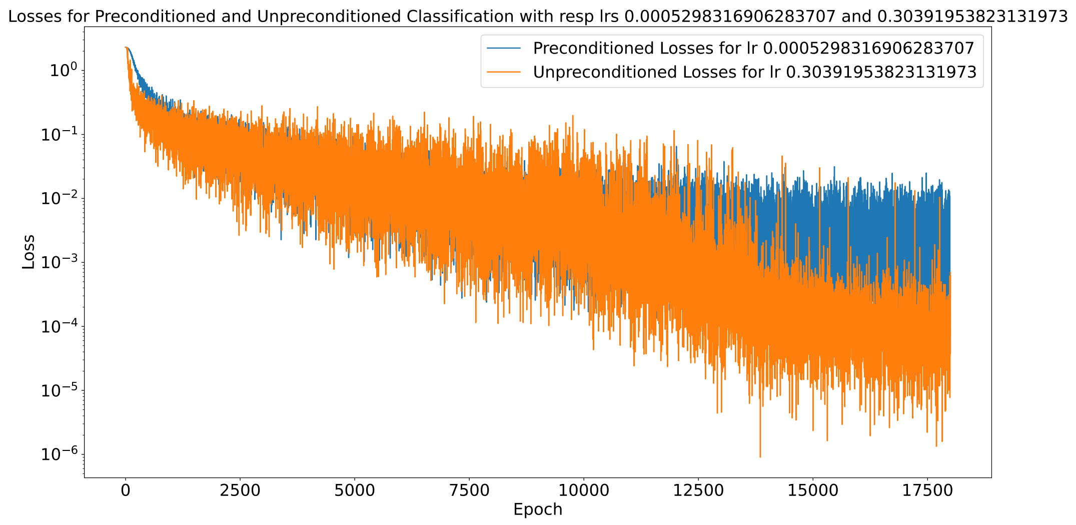


Figure 5.1.: Comparison of the champions of unpreconditioned SGD (learning rate $\alpha \approx 5 \times 10^{-4}$) and preconditioned SGD (learning rate $\alpha \approx 0.3$) using the Frobenius-type covariance-based preconditioner for MNIST classifier training. The figure shows loss vs iterations.

In order to get a deeper understanding of the mechanics of the covariance-based preconditioner during training, we analyze the behavior of the gradients and inner products induced by the covariance estimates over the course of training. Because each weight matrix contains many parameters, we summarize layer-wise behavior by monitoring gradient norms of the weight matrices instead of the full parameter vector norm or absolute values of individual parameters. We plot the norms of the weight gradients over the course of training iterations in Figure 5.2. For unpreconditioned SGD (top row), we see that the gradient norms first shortly increase and then decrease over training for all layers, with the first layer having the largest gradient norms. The Euclidean gradient norms (middle row) and the Frobenius preconditioned gradient norms (bottom row) of the preconditioned training display a higher noise level compared to the unpreconditioned training. Additionally, the scale of the gradient norms does not change significantly over training for all layers for both methods and the scale of the gradient norms of the preconditioned training is generally smaller than for the unpreconditioned training.

To understand the mechanics of the covariance-based preconditioner during training, we analyze the behavior of the eigenvalues of the inner products on the layer spaces which are induced by the covariance estimates over the course of training. The eigenvalues of the inner product matrices plotted over iterations for the champion of the preconditioned method are shown in Figure 5.3. We observe that the eigenvalues of the covariance matrices for the post-activation features (left column) and the pre-activation gradients (right column) behave differently over the course of training. For the covariance on the post-activation layer, the magnitude of the eigenvalues stays similar over training for all layers. The eigenvalues of the covariance matrix on the pre-activation layer show a different behavior. For all layers, most of the eigenvalues are close to zero during training while the two to three

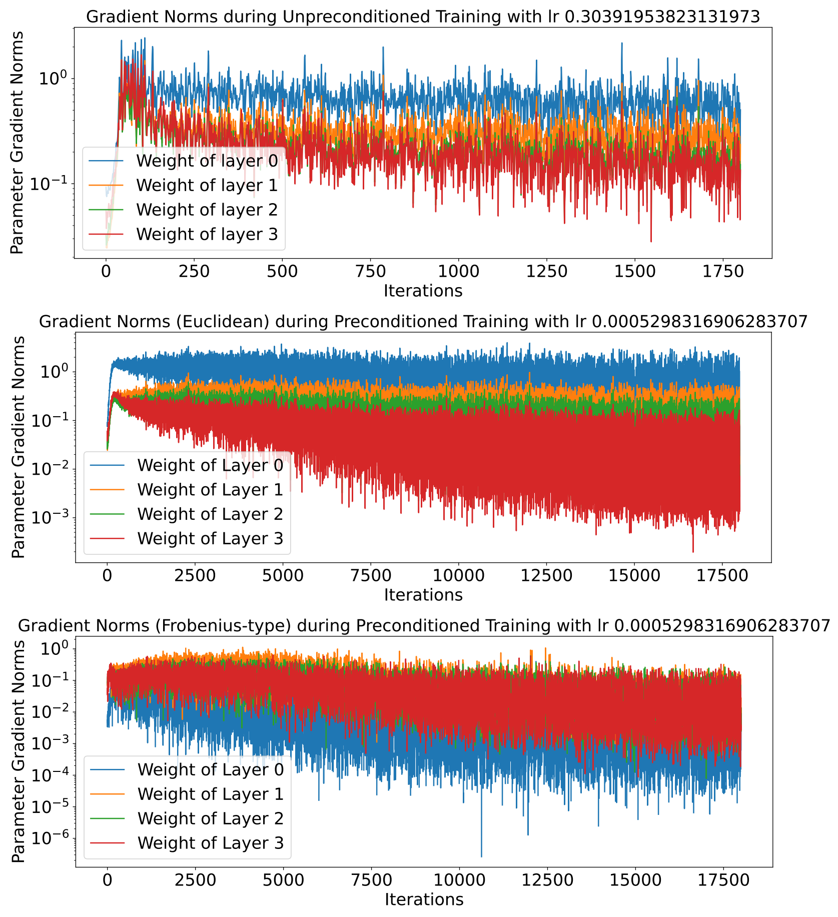


Figure 5.2.: Gradient norms for the training in Figure 5.1. Top: Euclidean gradient norms of unpreconditioned SGD (for learning rate $\alpha \approx 0.3$). Middle: Euclidean gradient norms of preconditioned training (learning rate $\alpha \approx 5 \times 10^{-4}$). Bottom: Frobenius preconditioned gradient norms of preconditioned training (learning rate $\alpha \approx 5 \times 10^{-4}$).

largest eigenvalues take significantly larger values in a very noisy and unstable manner. This suggests that the preconditioning in the pre-activation space is mostly ineffective and random since most directions are scaled by the identity due to the damping, while still having a high computational cost for estimating the covariance. The gradient norms displayed in Figure 5.2 suggest the same conclusion since the noise level of the gradients is high. In this experiment, the Frobenius-type covariance-based preconditioner is not able to improve loss decrease compared to unpreconditioned SGD over iteration count; even more, it is slightly worse. For the next experiment, we consider a setting where the Frobenius-type covariance-based preconditioner is able to improve training performance compared to unpreconditioned SGD. We are going to analyze the behavior of the gradients and inner products induced by the covariance estimates over the course of training for this setting as well and compare it to the previous experiment.

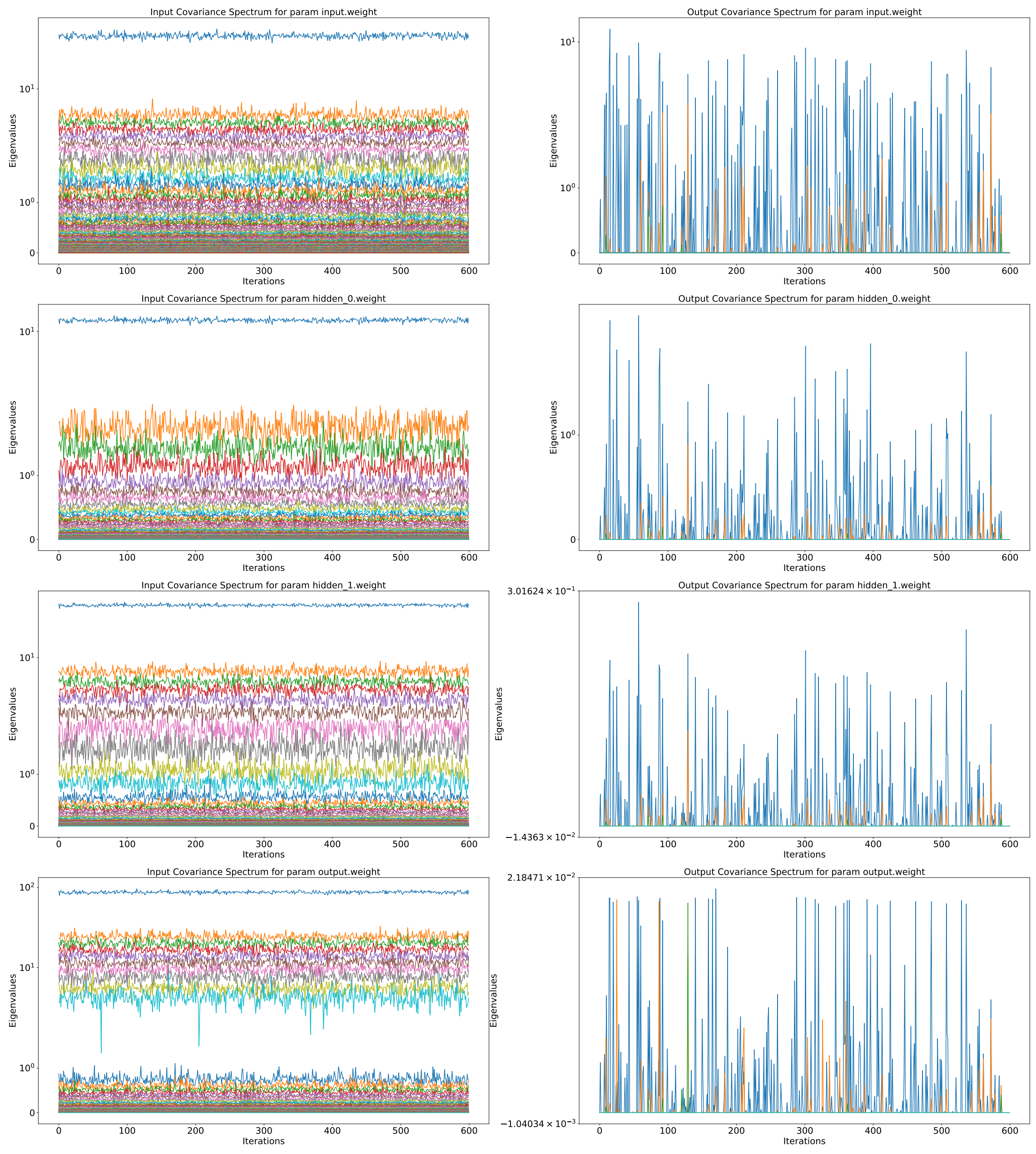


Figure 5.3.: Spectra of the covariance matrices for the preconditioned champion of the ReLU-FNN classification task on MNIST dataset with learning rate $\approx 5 \times 10^{-4}$. Left: Covariance of post-activation features. Right: Covariance of pre-activation gradients.

Autoencoder Task with Sigmoid-FNN

Next, we consider an autoencoder task (cf. Figure 2.3) on the MNIST dataset (cf. Figure 2.5) with flattened input images of size $28 \times 28 = 784$. We choose this task since it is more complex than the classification task and K-FAC [91] reported good results in the specific setting, which we aim to reproduce here. A fully-connected autoencoder with the architecture $784 - 1000 - 500 - 250 - 30 - 25 - 500 - 1000 - 784$ and sigmoid activations is used. We train the autoencoder to minimize the MSE-loss using mini-batch SGD with and without the Frobenius-type covariance-based preconditioner. Since good results are reported in [91] for large batch sizes, we use a batch size of 1000 for all our experiments. The preconditioner is implemented as described in the experiment above, cf. Section 5.3.3. A damping term of γid with $\gamma = 0.001$ is used for both covariance estimates for all weight matrices.

We perform experiments for a grid of fixed learning rates for both unpreconditioned SGD and preconditioned SGD using the Frobenius-type covariance-based preconditioner. Then we train for 30 epochs (leading to 1800 iterations with batch size 1000) and select the champion learning rate for both methods based on the lowest training loss achieved after 30 epochs. All experimental details are given in Appendix B.4.

We observe in Figure 5.4 that the loss over the iterations decreases significantly more for the preconditioned champion compared to the champion of unpreconditioned SGD. This shows that the Frobenius-type covariance-based preconditioner has the potential to significantly improve training performance compared to unpreconditioned SGD. After performing similarly to the unpreconditioned method in the first iterations, the preconditioned method is able to escape the minimum found by unpreconditioned SGD and to decrease the loss further at around iteration 140. Naturally, the preconditioned method has a higher computational cost per iteration due to the covariance estimates. Examining the loss over wall-clock time, the advantage of the preconditioned method shrinks, but it is still able to achieve a lower loss in the same amount of time compared to unpreconditioned SGD. This is, because unpreconditioned SGD in this case seems to fail to escape a minimum found after only a few iterations even when given significantly more time. The general behavior for the grid of learning rates for both preconditioned and unpreconditioned SGD is shown below in Figure 5.8 and Figure 5.9.

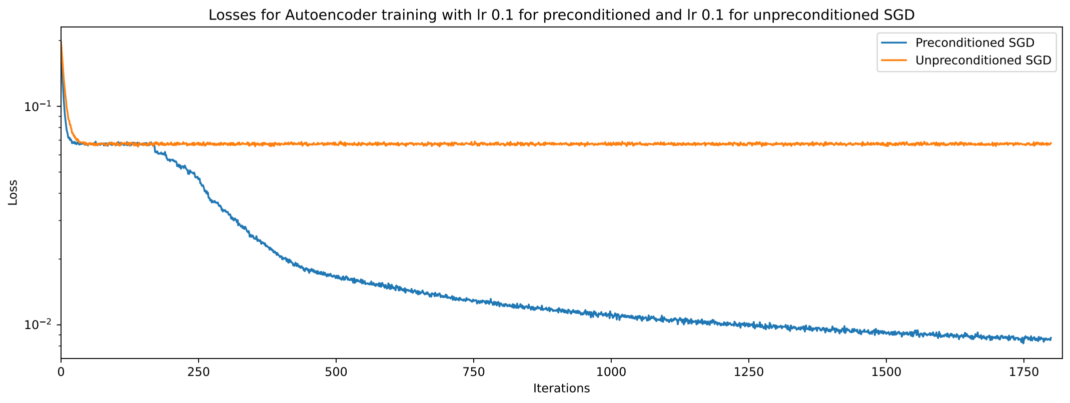


Figure 5.4.: Comparison of the champions of unpreconditioned SGD (learning rate $\alpha = 0.1$) and preconditioned SGD (learning rate $\alpha = 0.1$) using the Frobenius-type covariance-based preconditioner for MNIST autoencoder training. The figure shows loss vs iterations.

We examine the gradient norms w.r.t. the Euclidean inner product for unpreconditioned SGD and for the preconditioned method both w.r.t. the Euclidean inner product and the Frobenius-type inner product induced by the covariance estimates. We observe from Figure 5.5 that the values of the Euclidean gradient norms of unpreconditioned SGD stay at their respective order of magnitude over the whole training. In the beginning of the training all gradient norms decrease slowly in a similar fashion until the method starts to plateau. Then the gradient norms stagnate at their individual scale. This behavior is mirrored in the loss decrease shown in Figure 5.4 where the loss stagnates after a few iterations. For the preconditioned method, we observe a different behavior. The Euclidean gradient norms start at a similar scale as for unpreconditioned SGD and decrease, but then their behavior changes significantly at around iteration 140. This is the same iteration where the loss starts to decrease further for the preconditioned method in Figure 5.4. After this point, the Euclidean gradient norms change their scale and become more similar in their order of magnitudes across all layers. This indicates that the preconditioner is able to change the direction of the gradient significantly at this point and escape the minimum found by unpreconditioned SGD. At the same time, this is unexpected since the norms are not scaled by the dimensions of the weight matrices which indicates that the preconditioner is able to find large parameter values in all directions of the weight matrices without disrupting the training process. For the Frobenius-type gradient norms of the preconditioned method we observe a similar but more noisy picture as for the Euclidean gradient norms of preconditioned SGD.

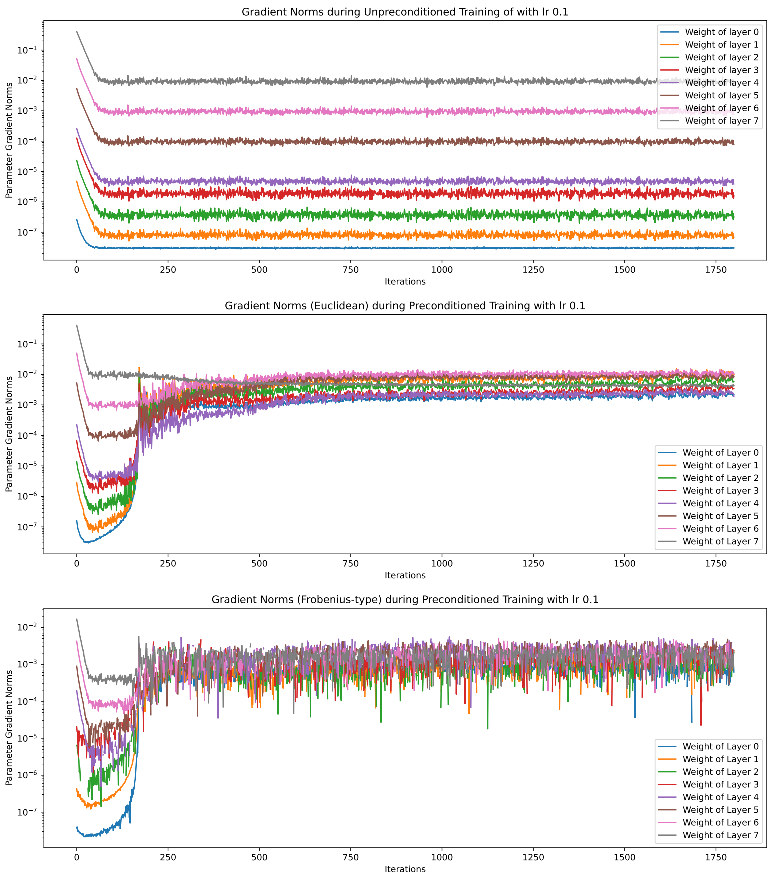


Figure 5.5.: Gradient norms for the training in Figure 5.4. Top: Euclidean gradient norms of unpreconditioned SGD (for learning rate $\alpha = 0.1$). Middle: Euclidean gradient norms of preconditioned training (learning rate $\alpha = 0.1$). Bottom: Frobenius preconditioned gradient norms of preconditioned training (learning rate $\alpha = 0.1$).

We also examine the spectra of the inner product matrices used for the preconditioning over the course of training. We tracked the spectra of all covariance matrices of all layers during training. The results for the champion learning rate are shown in Figures 5.6 and 5.7 for the encoder and decoder part of the autoencoder, respectively. We observe that the spectra of all covariance matrices change significantly over the course of training. The most outstanding observation is that the spectra of all covariance matrices of the pre-activation layer spaces increase their scale significantly at around iteration 140. This, again, coincides with the iteration where the loss starts to decrease further and where the Euclidean gradient norms change their scale. Also, comparing the spectra of the inner product matrices of the post-activation layer spaces to the pre-activation layer spaces, we observe that the spectra have different range of scales. In particular, the post-activation layer covariance matrices all have one very large eigenvalue compared to the rest of the spectrum which does not change significantly over the course of training, which can also be observed in the previous experiment in Figure 5.3. Finding the cause for the displayed behavior of the spectra is subject of future research.

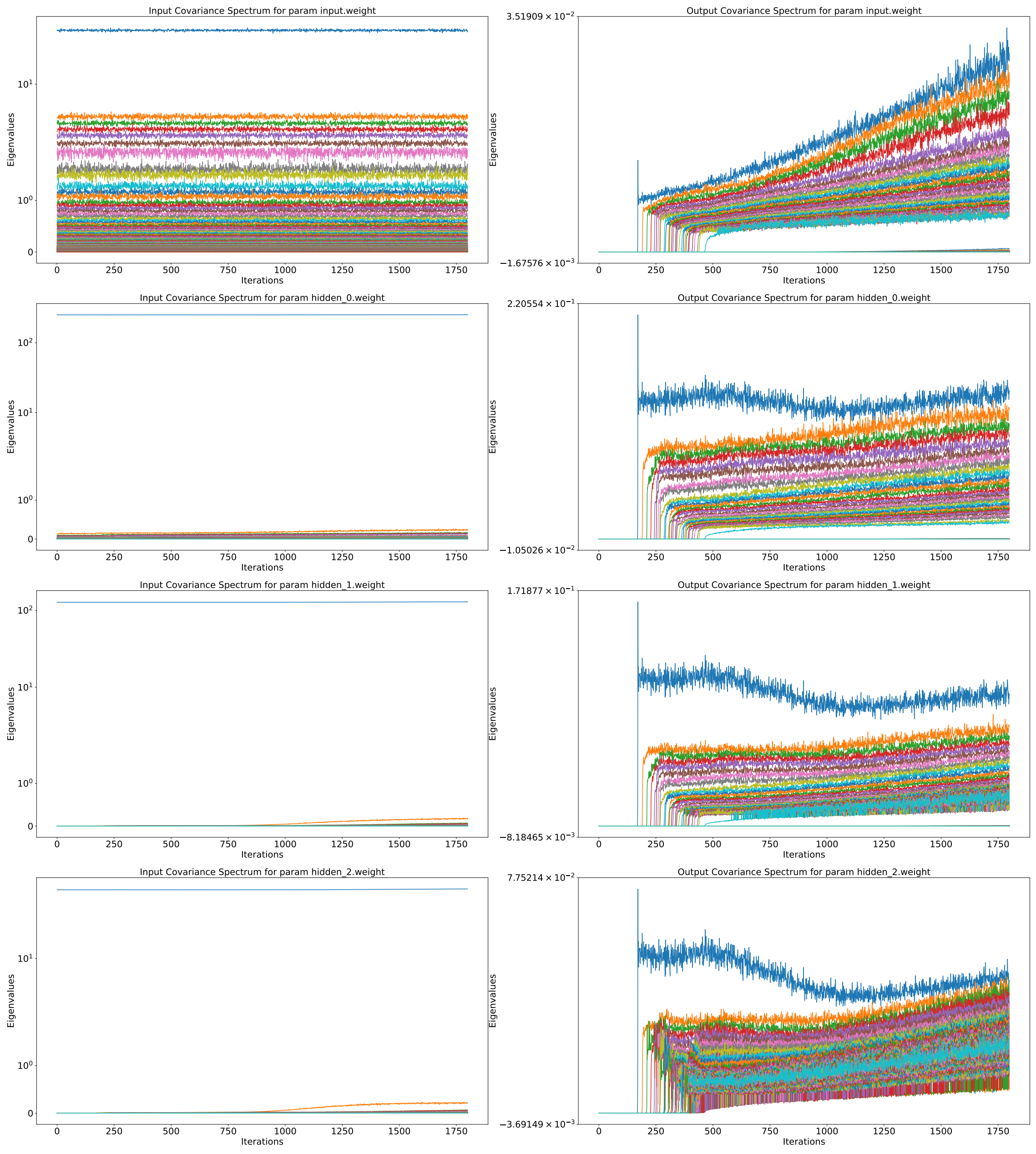


Figure 5.6.: Spectra of the covariance matrices for the champion for autoencoder training. Encoder. Left: Covariance of post-activation features. Right: Covariance of pre-activation gradients.

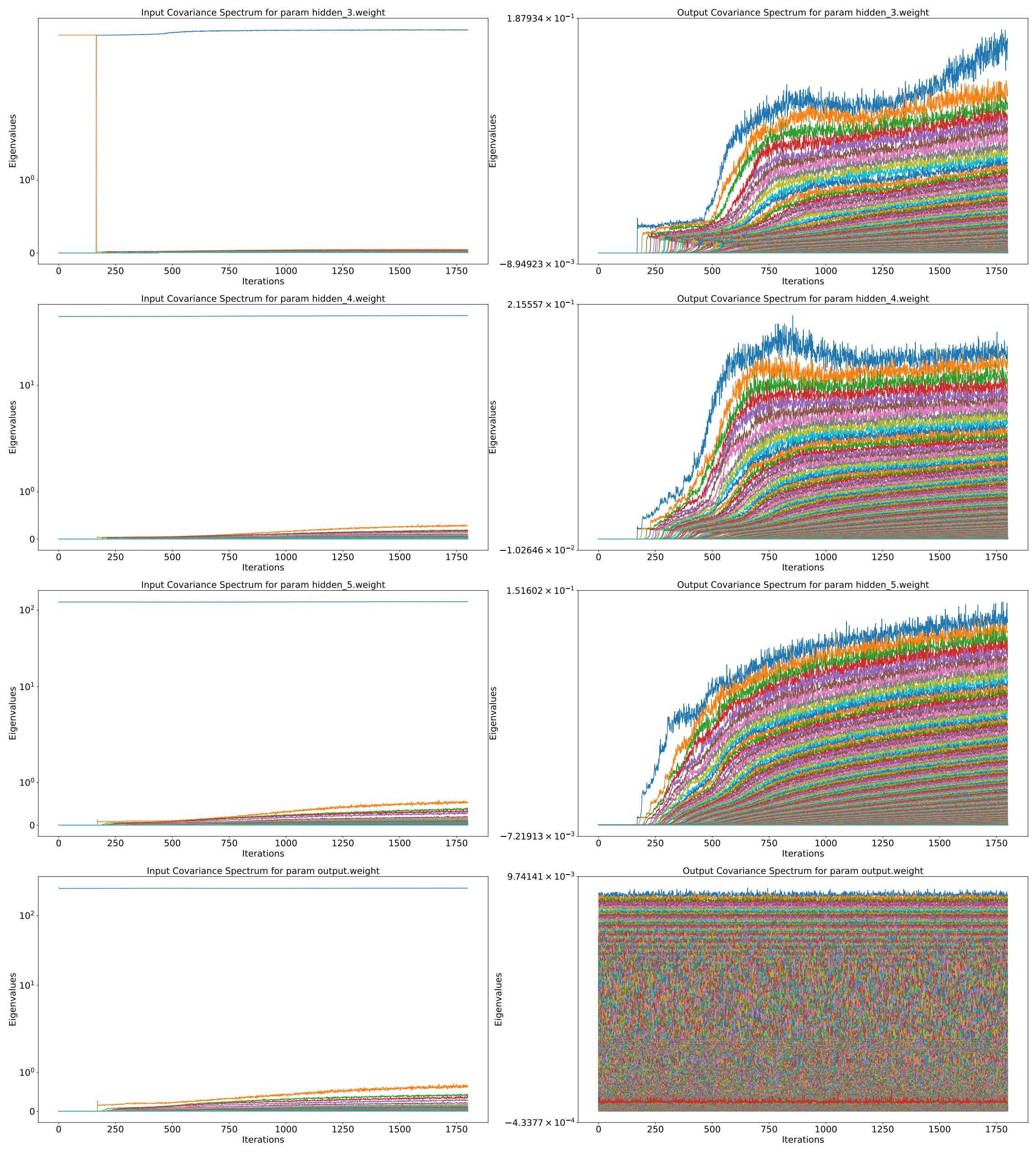


Figure 5.7.: Spectra of the covariance matrices for the champion for autoencoder training. Decoder. Left: Covariance of post-activation features. Right: Covariance of pre-activation gradients.

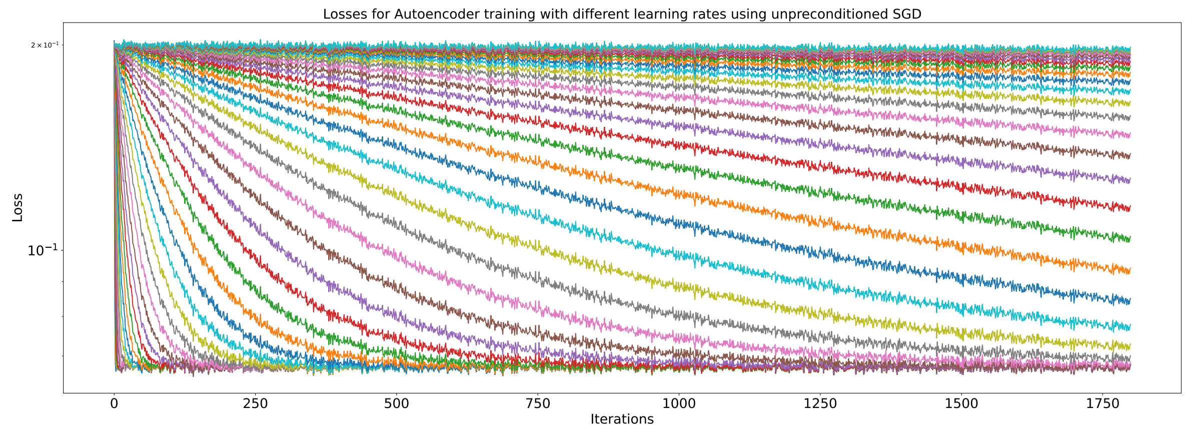


Figure 5.8.: The figure shows loss over iterations for unpreconditioned SGD runs for MNIST autoencoder training with different fixed learning rates.

The general performance of unpreconditioned SGD for different fixed learning rates on the autoencoder problem is shown in Figure 5.8. We observe that all learning rates lead to the approximately same minimal loss value and none of the learning rates used in training with unpreconditioned SGD are able to decrease the loss further within 30 epochs (and even longer). Larger learning rates reach this loss minimum faster, while smaller learning rates have a slower loss decay. This suggests that unpreconditioned SGD gets caught in a local minimum in this experiment setup for all learning rates tested. Of course, this behavior might not be observed when e.g. a different architecture or learning rate schedules are used.

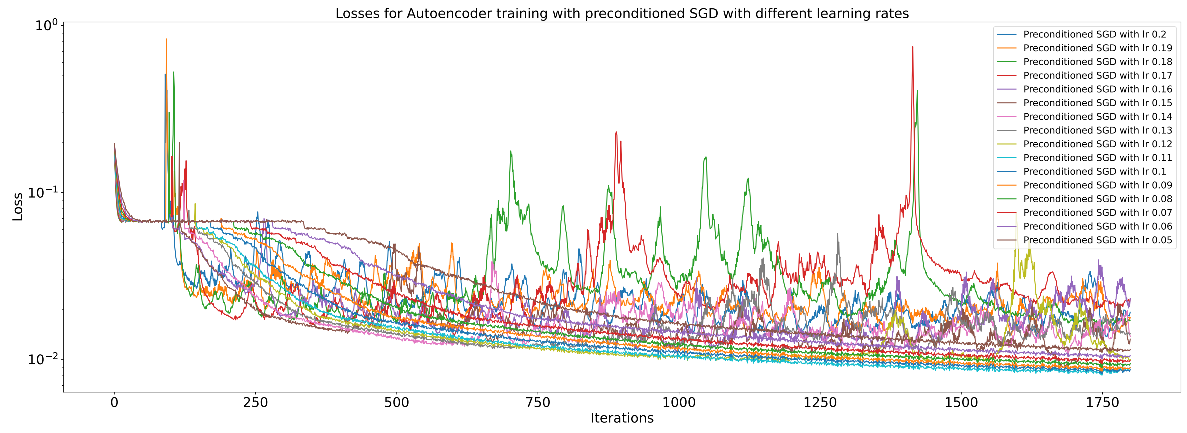


Figure 5.9.: The figure shows loss over iterations for preconditioned SGD runs for MNIST autoencoder training with different fixed learning rates.

Additionally, we examine the behavior of the preconditioned SGD for different fixed learning rates on this autoencoder problem for 30 epochs in Figure 5.9. When examining the range of the loss values in Figure 5.9, compared to Figure 5.8, it is prominent that the loss values take on different scales. We observe that for all displayed learning rates the preconditioned method is able to achieve a smaller loss than unpreconditioned SGD after 30 epochs. Training preconditioned SGD with larger learning rates leads to an eventually unstable behavior for later iterations, while smaller learning rates lead to a slower loss decay. This shows that the choice of the learning rate is important for the performance of the preconditioned method as well, but is robust in the sense that all learning rates tested lead to a better performance than unpreconditioned SGD in this experiment setup. Also, decreasing the learning rate over time might lead to an even better performance of the preconditioned method.

Compared to the setup in Section 5.3.3, the covariance-based preconditioner is able to improve training performance significantly for this autoencoder task. Also, the gradient norms and spectra of the covariance matrices display a more active behavior with significant changes of scale over the course of training compared to the previous classification task, where only few but erratic changes were observed. This suggests that the covariance-based preconditioner is able to adapt better to the optimization landscape in this experiment setup and is hence able to improve training performance significantly. At the same time, unpreconditioned SGD seems to struggle in this particular setup, probably due to the high complexity of the loss landscape.

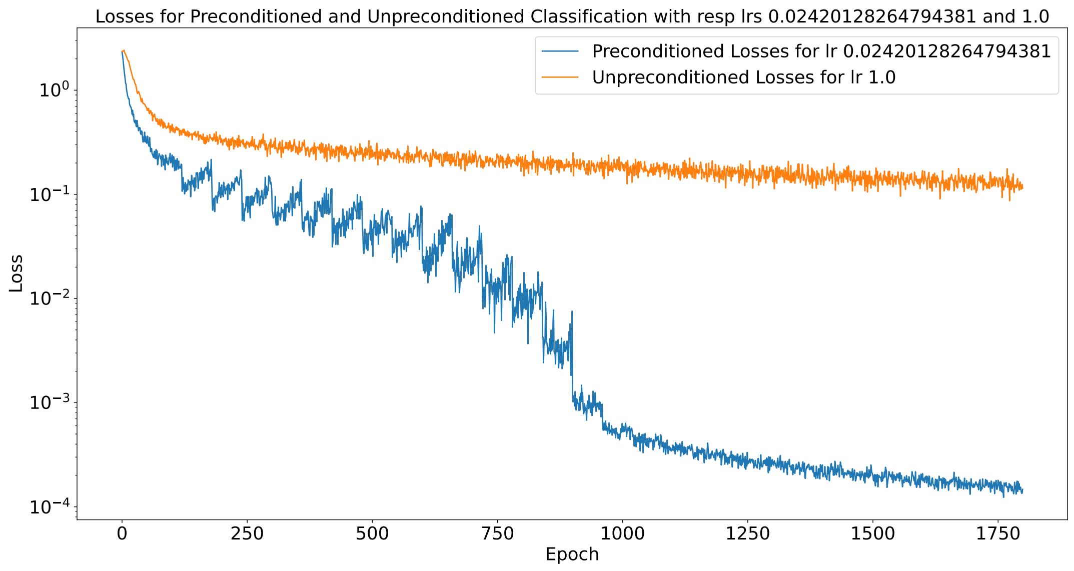
Classification Task with Sigmoid-FNN

Next, we examine a classification task on the MNIST dataset (Figure 2.5) with hyperparameters inspired by the autoencoder experiment in Section 5.3.3. Similar hyperparameters as in the autoencoder experiment are chosen to see whether preconditioned SGD can also improve training performance for classification tasks in this setting. We employ a shallow FNN architecture with widths $784-100-10$ with sigmoid activations. We choose a shallow architecture, because the covariance-based preconditioner is computationally expensive for large architectures and at the same time, unpreconditioned SGD can struggle to train shallow networks, because they are generally less over-parametrized than deeper networks and hence harder to optimize. To train the network, we minimize the CE-loss using mini-batch SGD with and without the Frobenius-type covariance-based preconditioner. We use a batch size of 1000, as in the autoencoder experiment. The preconditioner is implemented as described in Section 5.3.3.

We perform experiments for a grid of fixed learning rates for both unpreconditioned SGD and preconditioned SGD using the Frobenius-type covariance-based preconditioner. Then we train for 30 epochs (leading to 18,000 iterations with batch size 1000) and select the champion learning rate for both methods based on the lowest training loss achieved after 30 epochs.

We observe in Figure 5.10 that, contrary to the first classification experiment in Section 5.3.3, the loss over iterations decreases significantly more for the preconditioned method compared to unpreconditioned SGD, as is the case for the autoencoder training in Section 5.3.3. This observation even holds true for the first iterations of training. Examining the loss over wall-clock time, the advantage of the preconditioned method is less pronounced, but it is still able to achieve a significantly lower loss in the same amount of time compared to unpreconditioned SGD. Here, unpreconditioned SGD was computed for more epochs to show that the preconditioned method reaches a lower loss in the same time. Even for the longer time which unpreconditioned SGD was given, it is not able to reach the loss achieved by the preconditioned method.

Comparing the gradient norms w.r.t. the Euclidean inner product for unpreconditioned SGD and for the preconditioned method both w.r.t. the Euclidean inner product and the Frobenius-type inner product induced by the covariance estimates in Figure 5.11, we observe that the scale of the Euclidean gradient



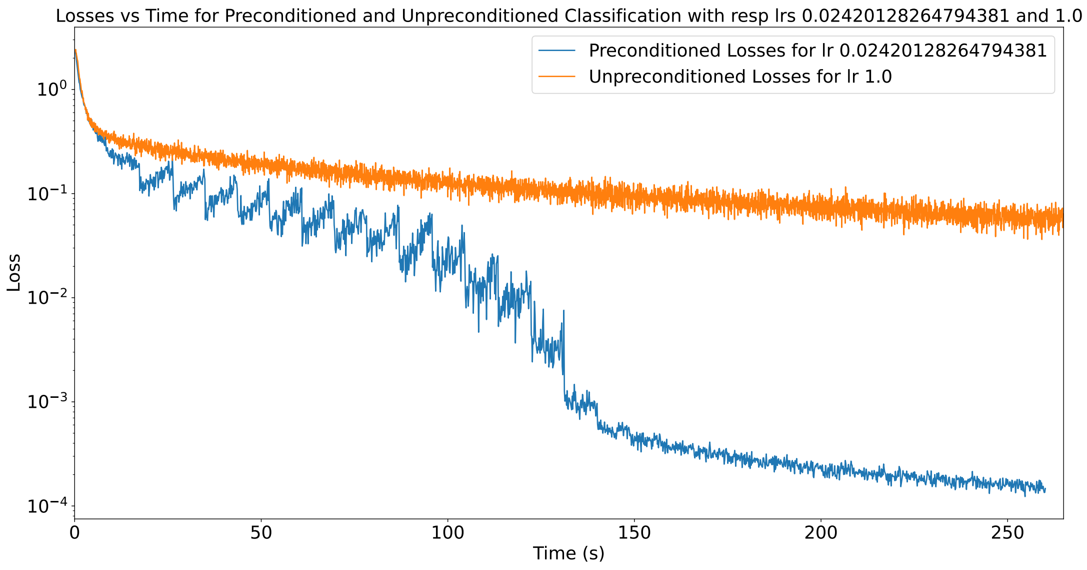


Figure 5.10.: Comparison of the champions of preconditioned SGD (learning rate $\alpha \approx 0.024$) and unpreconditioned SGD (learning rate $\alpha = 1.0$) using the Frobenius-type covariance-based preconditioner for MNIST autoencoder training. Top: Loss over iterations. Bottom: Loss over time.

norms of unpreconditioned SGD decreases less compared to the gradient norms of the preconditioned method, where the order of magnitude of both the Euclidean and Frobenius-type gradient norms of the preconditioned training decreases significantly over the course of training in accordance with the loss decrease. However, this behavior is completely different from the one observed in the autoencoder experiment in Figure 5.5.

Examining the spectra of the covariance matrices used for the preconditioning over the course of training in Figure 5.12, their behavior is different from the one observed in the autoencoder experiment in Figures 5.6 and 5.7 as well. The difference is most pronounced for the covariance matrices of the

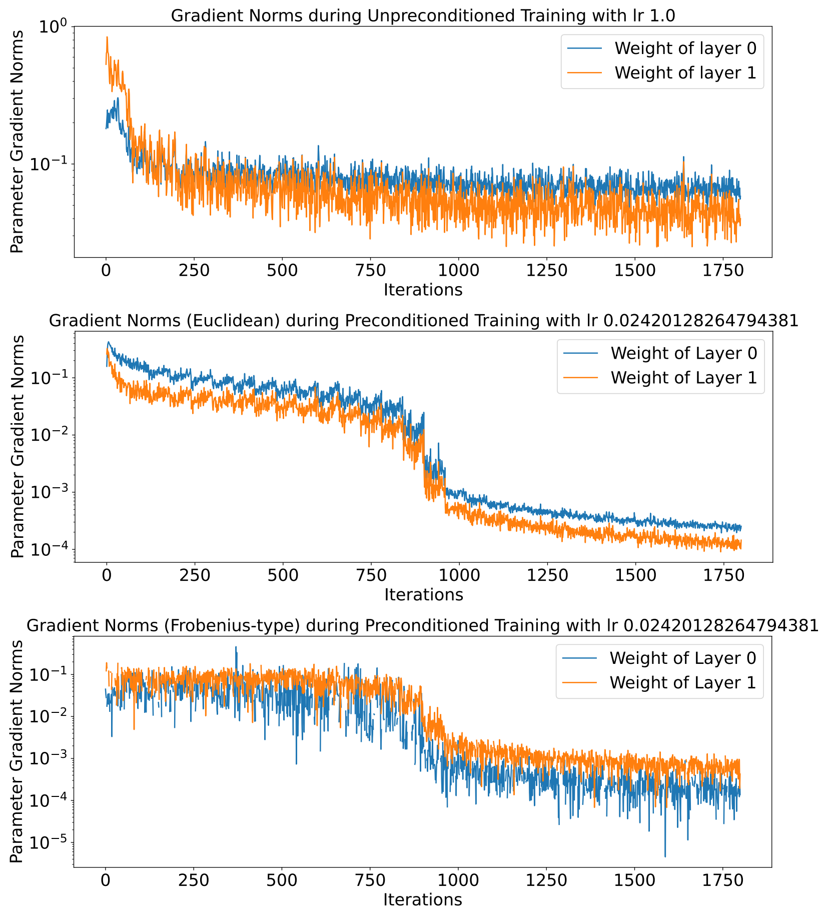


Figure 5.11.: Gradient norms for the training in Figure 5.10. Top: Euclidean gradient norms of unpreconditioned SGD (for learning rate $\alpha = 1.0$). Middle: Euclidean gradient norms of preconditioned training (learning rate $\alpha \approx 0.024$). Bottom: Frobenius preconditioned gradient norms of preconditioned training (learning rate $\alpha \approx 0.024$).

pre-activation layer spaces, where we do not observe a significant change in scale over the course of training, but a few peaks in the spectrum appear and disappear over the course of training.

In the above experiments we have seen that the Frobenius-type covariance-based preconditioner can significantly improve training performance compared to unpreconditioned SGD for certain tasks and settings. However, the behavior of the gradients and covariance spectra during training differs for the different tasks and settings, even when both tasks benefit from the covariance-based preconditioner. This indicates that the mechanics of the covariance-based preconditioner during training are not yet fully understood and require further investigation. In order to examine more exactly when and why the covariance-based preconditioner helps and when not, further experiments varying single hyperparameters such as the architecture, activation functions, batch size, and damping are necessary.

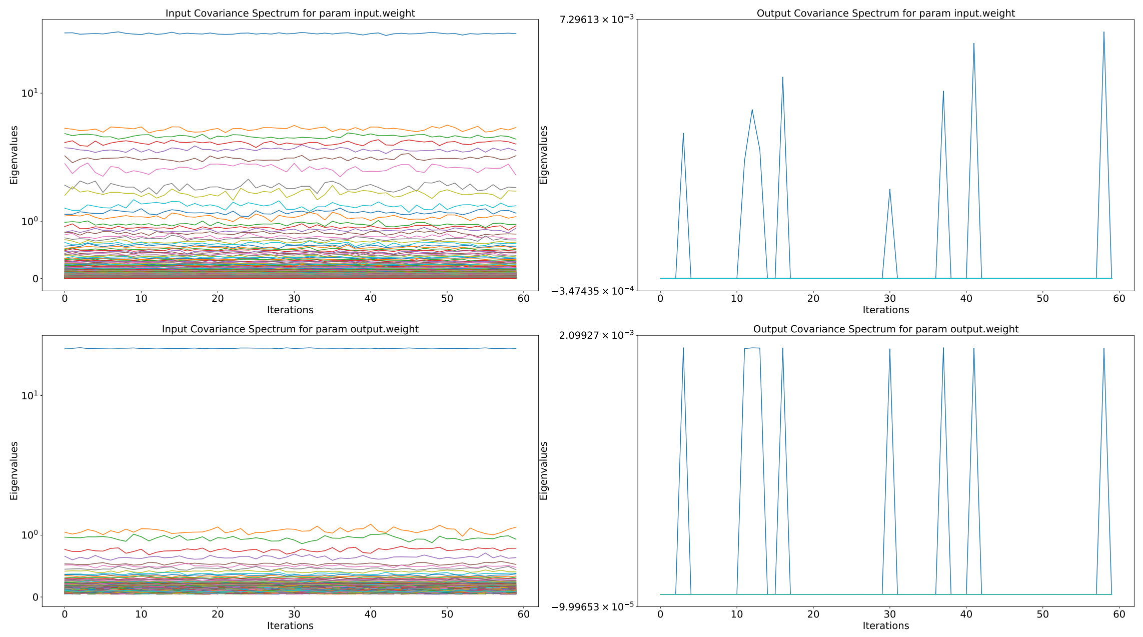


Figure 5.12.: Spectra of the covariance matrices for the preconditioned champion for classification training in Figure 5.10. Left: Covariance of post-activation features. Right: Covariance of pre-activation gradients.

5.4. Discussion

In this chapter we have developed a layer-wise preconditioning framework for Feedforward Neural Networks. Choosing layer-wise preconditioners instead of a global preconditioner for all weights of the neural network has the advantage that the preconditioned gradient w.r.t. the weight matrix W_i of the i-th layer can be computed independently of the other layers and is hence generally less computationally expensive. The preconditioning is based on an inner product on the space of linear maps between the post-activation spaces X_{i-1} and pre-activation spaces Y_i of each layer i, which is induced by the inner products on the layer spaces. The framework allows using predefined fixed inner products which arise from a-priori knowledge about the data or the task at hand, but also data-dependent inner products based on e.g. covariance information of the layer spaces in the course of training, which are updated online during training. In particular, we have presented a Frobenius-type covariance-based preconditioner which takes the varying stochastic structure of the data in the layer spaces into account. It uses covariance estimates of the post-activation features and pre-activation gradients as inner products on the layer spaces. This specific type of preconditioning coincides with the backbone of the well-known K-FAC method [91] when second moment estimates are used instead of covariance estimates.

We have conducted numerical experiments for Feedforward Neural Network training using the Frobenius-type covariance-based preconditioner on classification and autoencoder tasks on the MNIST dataset. We differ in our implementation from K-FAC because we do not use any additional heuristics such as momentum, learning rate schedules, or adaptive damping strategies (cf. Section 2.5.4) and train with fixed learning rates using mini-batch SGD. This allows us to isolate the effect of the covariance-based preconditioning on the training performance better. The experiments show that the covariance-based preconditioner has the potential to significantly improve training performance compared to unpreconditioned SGD for certain settings. This is most pronounced in scenarios where unpreconditioned SGD struggles to escape local minima and is in line with the good performance of K-FAC reported in the literature. However, the improved performance is not consistent across arbitrary tasks and settings. In particular, for the first classification experiment using a ReLU-FNN architecture, the covariance-based preconditioner was not able to outperform unpreconditioned SGD and can even lead to worse decrease in the loss value. Analyzing the behavior of the gradients of the weight matrices and inner products induced by the covariance estimates over the course of training, we observed different not yet understood behaviors for the different tasks. These observations give first insights into the mechanics of the covariance-based preconditioner during training. The behavior of loss decrease, gradient norms and spectra of the inner products on the layer spaces which was observed for the autoencoder task was the most comprehensible and interpretable. However, the behavior of the spectra of the covariance matrices which was observed for the second classification task with a shallow sigmoid-FNN architecture differed significantly from the autoencoder task, even though the covariance-based preconditioner led to an improved training performance compared to unpreconditioned SGD in both cases. This indicates that the mechanics of the covariance-based preconditioner during training are not yet fully understood and require further investigation. To our knowledge, no theoretical validation of the good functionality of covariance-based layer-wise preconditioners for neural network training exists yet, which would help to better understand their mechanics during training. Such is the case for the K-FAC method as well, where the good practical performance is not backed up by a theoretical understanding of its mechanics during training, due to the approximations involved in the derivation of the K-FAC method.

The computational cost of computing the Frobenius-type preconditioned gradient w.r.t. the weight matrix W_i in (5.4) depends on the cost of computing the products with the (inverses of) preconditioner matrices $R_{X_{i-1}}$ and $R_{Y_i}^{-1}$. If these matrices are dense, this can quickly become very expensive for wide layers with large widths $n_{X_{i-1}}$ and n_{Y_i} compared to the cost of computing the Euclidean gradient itself. Hence, it can be desirable, if possible, to design preconditioner matrices $R_{X_{i-1}}$ and $R_{Y_i}^{-1}$ that are sparse, low-rank (plus base metric), or have a multilevel structure to reduce the computational cost of computing the preconditioned gradient in some cases. When a Frobenius-type covariance-based preconditioner is better understood and performs well for a wider range of tasks and settings, it might be possible to exploit low-rank structures of the covariance matrices in the layer spaces to reduce the computational cost of computing the preconditioned gradients as future research.

6. Conclusion and Outlook

This thesis develops and analyzes adaptations of selected classical nonlinear optimization techniques for the training of neural networks, with a particular focus on stochastic, high-dimensional, and non-convex settings. While many ideas from deterministic nonlinear optimization remain conceptually relevant, their direct application to modern neural network training is often inadequate due to the stochastic nature of training data. The contributions presented here illustrate both the potential and the limitations of transferring such techniques to stochastic learning problems, and they highlight the need for careful methodological and theoretical adaptation.

Chapter 3 investigates multilevel optimization methods for stochastic optimization, motivated by the success of multigrid techniques in deterministic settings. We propose new MG/Opt variants tailored to stochastic objectives and analyze their convergence properties under different structural assumptions. The chapter begins with an analysis of the behavior of SGD iterates in the strongly convex quadratic case. This setting provides a tractable model problem that allows us to study how the behavior of stochastic gradient methods can be exploited within a multilevel framework. Based on these insights, we construct suitable restriction and prolongation operators and develop a stochastic MG/Opt variant for quadratic objectives, accompanied by a convergence analysis. The theoretical guarantees in this setting rely on strong assumptions regarding the choice of transfer operators and the structure of the coarse grid correction problems. While these assumptions ensure convergence, they also reveal the fragility of multilevel constructions in stochastic optimization and motivate the discussion of possible relaxations and alternative formulations.

Moving beyond the quadratic case, we consider non-convex objective functions, which are of primary relevance for neural network training. For this setting, we introduce a stochastic two-level MG/Opt variant and establish convergence results under standard assumptions on stochastic gradients, together with additional assumptions specific to the multilevel framework. We prove convergence in expectation and almost sure convergence for diminishing step sizes, which, to the best of our knowledge, constitute the first convergence results of this type for MG/Opt in a stochastic non-convex setting.

In addition to the theoretical analysis, the chapter discusses various possibilities for defining hierarchies of neural network training problems, including hierarchies based on network depth, resolution, or variance of gradient estimates. While the presented theory applies to a broad class of hierarchical objective functions, numerical evidence from the literature suggests that classical MG/Opt coarse grid correction updates do not necessarily improve training performance for realistic neural network problems. In particular, existing experiments indicate that accelerated training can sometimes be achieved by solving auxiliary coarse problems without the classical correction term, leading to algorithms that deviate from MG/Opt and for which the established convergence theory no longer applies. These observations underline both the promise and the current limitations of multilevel approaches for neural network training and point to a gap between theoretical convergence results and empirically successful heuristics.

Chapter 4 addresses a different aspect of adaptivity in neural network training, namely the question of how and where to modify network architectures during training. The proposed Sensitivity-based Layer Insertion (SensLI) method introduces a sensitivity-based approach to adaptive layer insertion that is conceptually simple, computationally efficient, and broadly applicable. SensLI formulates layer insertion as a constrained optimization problem, where candidate layers are initialized as identity mappings and temporarily frozen. By evaluating the sensitivity of the objective function with respect to these constraints, the method derives a clear and principled selection criterion for identifying promising insertion locations in the network.

A key advantage of SensLI is that it requires only a single additional backpropagation step to evaluate all candidate insertions simultaneously, making it computationally cheaper than alternative methods such as SENN or Firefly. Unlike heuristic or architecture-specific approaches, SensLI is rooted in classical sensitivity analysis for nonlinear programming and can be applied to fully-connected, residual, and convolutional layers. Although the numerical experiments focus on inserting a single layer at a time, the underlying merit indicators naturally extend to the insertion of multiple layers or to layer widening. While the method is based on a local, first-order analysis, empirical results demonstrate that SensLI can lead to improved loss decay, reduced training time, and better test accuracy, particularly in convolutional neural networks.

At the same time, several limitations must be acknowledged. The selection criterion is inherently local and cannot predict long-term effects of insertion or guarantee global improvements. The supporting theory assumes smoothness conditions that are violated by commonly used non-differentiable activation functions such as ReLU, and extending the analysis to fully stochastic, non-convex training remains challenging. Consequently, the evidence for the effectiveness of SensLI is empirical rather than theoretical, and further experimental validation and theoretical investigation are needed to better understand when and why adaptive layer insertion is beneficial.

Chapter 5 develops a general layer-wise preconditioning framework for Feedforward Neural Networks, motivated by interpreting weight matrices as linear maps between layer spaces and by equipping these spaces with suitable inner products. By focusing on layer-wise preconditioners rather than a single global preconditioner, the framework enables independent computation of preconditioned gradients for each layer, which can significantly reduce computational cost compared to global preconditioning. The framework encompasses both predefined inner products, based on prior knowledge about the task or data, and data-driven inner products that are updated online during training.

A central contribution of this chapter is the introduction and analysis of a Frobenius-type covariance-based preconditioner, which uses covariance estimates of post-activation features and pre-activation gradients to define inner products on the layer spaces, which adapt during training. This construction naturally accounts for the stochastic structure of the data and reduces to the core structure of the K-FAC method when second-moment estimates are used instead of covariance estimates.

Numerical experiments on classification and autoencoder tasks using the MNIST dataset demonstrate that covariance-based layer-wise preconditioning can substantially improve training performance over unpreconditioned SGD in certain settings, particularly when standard SGD struggles to escape poor local minima. At the same time, the experiments reveal that such improvements are not universal and depend strongly on the task, architecture, and activation functions. Detailed analysis of gradient behavior and the evolution of layer-wise inner products provides initial insights into the mechanics of covariance-based preconditioning, but also highlights that these mechanics are not yet fully understood.

The computational cost of covariance-based preconditioning remains a significant challenge, especially for wide layers with large dense covariance matrices. This motivates future work on exploiting sparsity, low-rank structure, or multilevel approximations of covariance matrices to reduce computational overhead. More fundamentally, the lack of a rigorous theoretical understanding of why methods such as K-FAC work well in practice remains an open problem, and addressing this gap is essential for the principled design of future preconditioners.

Taken together, the contributions of this thesis illustrate recurring themes in neural network training. Stochastic, non-convex optimization differs fundamentally from deterministic settings, and classical nonlinear optimization techniques must be adapted with care rather than applied unchanged. While empirically successful methods often exist, rigorous theoretical understanding frequently lags behind, due to the complexity of the loss landscape, the stochastic nature of training, and the high dimensionality of parameter spaces.

Future research should aim for a tighter integration of theoretical and empirical analysis. This includes extending optimization methods to stochastic regimes under realistic cost models, identifying the mechanisms that drive practical performance gains, and developing theoretical explanations for observed empirical behavior rather than relying solely on benchmark comparisons. A deeper understanding of when and why particular hierarchies, preconditioners, or adaptive insertion strategies are effective will be crucial for transforming the methods presented in this thesis into robust and widely applicable tools for neural network training.

Bibliography

[1] S.-i. Amari. “Natural gradient works efficiently in learning”. In: *Neural Computation* 10.2 (1998), pp. 251–276. DOI: 10.1162/089976698300017746.

[2] B. Appolinary, A. Deaconu, S. Yang, Qingze, and Li. *Self Expanding Convolutional Neural Networks*. 2024. arXiv: 2401.05686.

[3] T. Ash. “Dynamic node creation in backpropagation networks”. In: *Connection Science* 1.4 (1989), pp. 365–375. DOI: 10.1080/09540098908915647.

[4] S. Axler. *Linear Algebra Done Right*. 3rd ed. Springer, 2014. DOI: 10.1007/978-3-319-11080-6. URL: https://linear.axler.net/.

[5] J. Ba, R. Grosse, and J. Martens. “Distributed second-order optimization using kronecker-factored approximations”. In: *International conference on learning representations*. 2017.

[6] J. L. Ba, J. R. Kiros, and G. E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450.

[7] F. Benzing. “Gradient descent on neurons and its link to approximate second-order optimization”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 1817–1853.

[8] A. Borzì. “On the convergence of the MG/OPT method”. In: *PAMM* 5.1 (2005), pp. 735–736. DOI: 10.1002/pamm.200510342.

[9] A. Borzì and V. Schulz. “Multigrid methods for PDE optimization”. In: *SIAM Review* 51.2 (2009), pp. 361–395. DOI: 10.1137/060671590.

[10] A. Botev, H. Ritter, and D. Barber. “Practical Gauss-Newton optimisation for deep learning”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 557–565.

[11] L. Bottou, F. E. Curtis, and J. Nocedal. “Optimization methods for large-scale machine learning”. In: *SIAM Review* 60.2 (2018), pp. 223–311. DOI: 10.1137/16M1080173.

[12] V. Braglia, A. Kopaničáková, and R. Krause. *A multilevel approach to training*. 2020. arXiv: 2006.15602.

[13] A. Brandt. “Multi-level adaptive solutions to boundary-value problems”. In: *Mathematics of Computation* 31.138 (1977), pp. 333–390. DOI: 10.2307/2006422.

[14] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. Second. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000. DOI: 10.1137/1.9780898719505.

[15] B. Chang, L. Meng, E. Haber, F. Tung, and D. Begert. “Multi-level residual networks from dynamical systems view”. In: *International Conference on Learning Representations, ICLR 2018*. 2018. arXiv: 1710.10348. URL: https://openreview.net/forum?id=SyJS-OgR-.

[16] T. Chen, I. Goodfellow, and J. Shlens. *Net2Net: accelerating learning via knowledge transfer*. 2015. arXiv: 1511.05641.

[17] R. M. Clarke and J. M. Hernández-Lobato. *Studying K-FAC Heuristics by Viewing Adam through a Second-Order Lens*. 2023. arXiv: 2310.14963.

[18] J. Conway. *A Course in Operator Theory.* Graduate Studies in Mathematics. American Mathematical Society, 2000.

[19] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang. "AdaNet: adaptive structural learning of artificial neural networks". In: *Proceedings of the 34th International Conference on Machine Learning.* Ed. by D. Precup and Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, 2017, pp. 874–883. URL: http://proceedings.mlr.press/v70/cortes17a.html.

[20] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274.

[21] X. Dai, H. Yin, and N. K. Jha. "NeST: a neural network synthesis tool based on a grow-and-prune paradigm". In: *IEEE Transactions on Computers* 68.10 (2019), pp. 1487–1497. DOI: 10.1109/tc.2019.2914438.

[22] F. Dangel, S. Harmeling, and P. Hennig. "Modular block-diagonal curvature approximations for feedforward architectures". In: *International Conference on Artificial Intelligence and Statistics.* PMLR. 2020, pp. 799–808.

[23] F. Dangel, F. Kunstner, and P. Hennig. "Backpack: Packing more into backprop". In: *arXiv preprint arXiv:1912.10985* (2019).

[24] G. Desjardins, K. Simonyan, R. Pascanu, et al. "Natural neural networks". In: *Advances in neural information processing systems* 28 (2015).

[25] C. Dong, L. Liu, Z. Li, and J. Shang. "Towards adaptive residual network training: a neural-ODE perspective". In: *Proceedings of the 37th International Conference on Machine Learning.* Ed. by H. Daumé and A. Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 2616–2626. URL: https://proceedings.mlr.press/v119/dong20c.html.

[26] S. E. Dreyfus. "Artificial neural networks, back propagation, and the Kelley-Bryson gradient procedure". In: *Journal of Guidance, Control, and Dynamics* 13.5 (1990), pp. 926–928. DOI: 10.2514/3.25422.

[27] J. Duchi, E. Hazan, and Y. Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[28] R. Durrett. *Probability. Theory and Examples.* 5th ed. Cambridge University Press, 2019. DOI: 10.1017/9781108591034.

[29] T. Elsken, J. H. Metzen, and F. Hutter. "Neural architecture search: A survey". In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21.

[30] U. Evci, B. van Merriënboer, T. Unterthiner, M. Vladymyrov, and F. Pedregosa. *GradMax: growing neural networks using gradient information.* 2022. arXiv: 2201.05125.

[31] S. Fahlman and C. Lebiere. "The Cascade-Correlation learning architecture". In: *Advances in Neural Information Processing Systems.* Ed. by D. Touretzky. Vol. 2. Morgan-Kaufmann, 1989.

[32] A. V. Fiacco. *Introduction to Sensitivity and Stability Analysis in Nonlinear Programming.* New York: Academic Press, 1983.

[33] O. Forster. *Analysis 2.* 10th ed. Springer Fachmedien Wiesbaden, 2013. DOI: 10.1007/978-3-658-02357-7.

[34] Y. Fujimoto and T. Ohira. “A Neural Network model with Bidirectional Whitening”. In: (2017). DOI: 10.1007/978-3-319-91253-0_5. arXiv: 1704.07147.

[35] L. Gaedke-Merzhäuser, A. Kopaničáková, and R. Krause. “Multilevel minimization for deep residual networks”. In: ed. by D. Auroux, J.-B. Caillau, R. Duvigneau, A. Habbal, C. Malot, O. Pantz, L. Pronzato, L. Rifford, R. Ruelle, and C. Soresi. Vol. 71. EDP Sciences, 2021, pp. 131–144. DOI: 10.1051/proc/202171131. arXiv: 2004.06196.

[36] J. Gallego-Posada, J. Ramirez, A. Erraqabi, Y. Bengio, and S. Lacoste-Julien. “Controlled sparsity via constrained optimization or: How i learned to stop tuning penalties and love constraints”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 1253–1266.

[37] M. Gargiani, A. Zanelli, M. Diehl, and F. Hutter. *On the Promise of the Stochastic Generalized Gauss-Newton Method for Training DNNs.* 2020. arXiv: 2006.02409.

[38] G. Garrigos and R. M. Gower. *Handbook of convergence theorems for (stochastic) gradient methods.* 2023. arXiv: 2301.11235.

[39] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent. “Fast approximate natural gradient descent in a kronecker factored eigenbasis”. In: *Advances in neural information processing systems* 31 (2018).

[40] H.-O. Georgii. *Stochastik.* 2009. DOI: doi:10.1515/9783110215274.bm.

[41] D. Goldfarb, Y. Ren, and A. Bahamou. “Practical quasi-newton methods for training deep neural networks”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 2386–2396.

[42] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2016. URL: https://www.deeplearningbook.org.

[43] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. “MorphNet: fast & simple resource-constrained structure learning of deep networks”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition.* IEEE, 2018. DOI: 10.1109/cvpr.2018.00171. arXiv: 1711.06798.

[44] S. Gratton, M. Mouffe, P. L. Toint, and M. Weber-Mendonca. “A recursive ℓ_∞-trust-region method for bound-constrained nonlinear optimization”. In: *IMA Journal of Numerical Analysis* 28.4 (2008), pp. 827–861. DOI: 10.1093/imanum/drn034.

[45] S. Gratton, A. Kopaničáková, and P. L. Toint. “Multilevel Objective-Function-Free Optimization with an Application to Neural Networks Training”. In: *SIAM Journal on Optimization* 33.4 (2023), pp. 2772–2800. DOI: 10.1137/23m1553455.

[46] S. Gratton, A. Sartenaer, and P. L. Toint. “Recursive trust-region methods for multiscale nonlinear optimization”. In: *SIAM Journal on Optimization* 19.1 (2008), pp. 414–444. DOI: 10.1137/050623012.

[47] C. Gross and R. Krause. “On the Convergence of Recursive Trust-Region Methods for Multiscale Nonlinear Optimization and Applications to Nonlinear Mechanics”. In: *SIAM Journal on Numerical Analysis* 47.4 (2009), pp. 3044–3069. DOI: 10.1137/08071819x.

[48] V. Gupta, T. Koren, and Y. Singer. “Shampoo: Preconditioned stochastic tensor optimization”. In: *International Conference on Machine Learning.* PMLR. 2018, pp. 1842–1850.

[49] E. Haber and L. Ruthotto. *Stable architectures for deep neural networks.* 2017. arXiv: 1705.03341.

[50] E. Haber and L. Ruthotto. “Stable architectures for deep neural networks”. In: *Inverse Problems* 34.1 (2017), p. 014004. DOI: 10.1088/1361-6420/aa9a90. arXiv: 1705.03341.

[51] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer Berlin Heidelberg, 1985. DOI: 10.1007/978-3-662-02427-0.

[52] E. Hairer, S. P. Nørsett, and G. Wanner. “The Numerical Solution of Ordinary Differential Equations I: Nonstiff Problems”. In: *Solving Ordinary Differential Equations I*. Springer Berlin Heidelberg, 1993, pp. 129–353. DOI: 10.1007/978-3-540-78862-1_2.

[53] S. Hanson. “Meiosis networks”. In: *Advances in neural information processing systems* 2 (1989).

[54] D. Harris and T. D. Gedeon. “Adaptive insertion of units in feed-forward neural networks”. In: *4th Int. Conf. on Neural Networks and their Applications*. 1991.

[55] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016. DOI: 10.1109/cvpr.2016.90. arXiv: 1512.03385.

[56] J. F. Henriques, S. Ehrhardt, S. Albanie, and A. Vedaldi. “Small steps and giant leaps: Minimal newton solvers for deep learning”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 4763–4772.

[57] E. Herberg, R. Herzog, F. Köhne, L. Kreis, and A. Schiela. *Sensitivity-based layer insertion for residual and feedforward neural networks*. 2023. arXiv: 2311.15995.

[58] R. Herzog, F. Köhne, L. Kreis, and A. Schiela. *Frobenius-type norms and inner products of matrices and linear maps with applications to neural network training*. 2023. arXiv: 2311.15419.

[59] T. Heskes. “On “Natural” Learning and Pruning in Multilayered Perceptrons”. In: *Neural Computation* 12.4 (2000), pp. 881–901. DOI: 10.1162/089976600300015637.

[60] M. Hintermüller, M. Hinze, and D. Korolev. *Layerwise goal-oriented adaptivity for neural ODEs: an optimal control perspective*. 2026. arXiv: 2601.07397.

[61] G. E. Hinton and R. R. Salakhutdinov. “Reducing the Dimensionality of Data with Neural Networks”. In: *Science* 313.5786 (2006), pp. 504–507. DOI: 10.1126/science.1127647.

[62] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich. *Optimization with PDE Constraints*. Berlin: Springer, 2009. DOI: 10.1007/978-1-4020-8839-1.

[63] K. Hornik, M. Stinchcombe, and H. White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

[64] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. *Densely Connected Convolutional Networks*. 2018. arXiv: 1608.06993 [cs.CV]. URL: https://arxiv.org/abs/1608.06993.

[65] M. Hudak. “RCE networks: an experimental investigation”. In: *IJCNN-91 Seattle International Joint Conference on Neural Networks*. IEEE, 1991. DOI: 10.1109/ijcnn.1991.155290.

[66] S. Ioffe and C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167.

[67] K. Ito and K. Kunisch. *Lagrange Multiplier Approach to Variational Problems and Applications*. Vol. 15. Advances in Design and Control. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2008. DOI: 10.1137/1.9780898718614.

[68] I. Jacob. *Stochastic multilevel methods for deep learning*. Verlag Dr. Hut, 2025. DOI: 10.26083/TUPRINTS-00029564. URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/29564.

[69] R. Johnson and T. Zhang. “Accelerating stochastic gradient descent using predictive variance reduction”. In: *Advances in Neural Information Processing Systems.* Ed. by F. Pereira, C. Burges, L. Bottou, and K. Weinberger. Vol. 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper_files/paper/2013/file/ac1dd209cbcc5e5d1c6e28598e8cbbe8-Paper.pdf.

[70] D. P. Kingma and J. Ba. *Adam: a method for stochastic optimization.* 2014. arXiv: 1412.6980.

[71] A. Klenke. *Wahrscheinlichkeitstheorie.* Springer, 2006.

[72] A. Kopaničáková. *On the use of hybrid coarse-level models in multilevel minimization methods.* 2022. arXiv: 2211.15078.

[73] A. Kopaničáková and R. Krause. “Globally convergent multilevel training of deep residual networks”. In: *SIAM Journal on Scientific Computing* 45.3 (2022), S254–S280. DOI: 10.1137/21m1434076.

[74] A. Koroko, A. Anciaux-Sedrakian, I. B. Gharbia, V. Garès, M. Haddou, and Q. H. Tran. *Efficient Approximations of the Fisher Matrix in Neural Networks using Kronecker Product Singular Value Decomposition.* 2022. arXiv: 2201.10285.

[75] L. Kreis. “Multilevel Training of Residual Neural Networks”. M.Sc. Thesis. Heidelberg University, 2022.

[76] A. Krizhevsky. *Learning multiple layers of features from tiny images.* Technical Report. Toronto, Ontario, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[77] A. Krizhevsky, G. Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009). URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[78] S. Kullback and R. A. Leibler. “On information and sufficiency”. In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

[79] F. Kunstner, P. Hennig, and L. Balles. “Limitations of the empirical fisher approximation for natural gradient descent”. In: *Advances in neural information processing systems* 32 (2019).

[80] S. Lang. “The Mean Value Theorem”. In: *A First Course in Calculus.* Springer New York, 1986, pp. 159–180. DOI: 10.1007/978-1-4419-8532-3_5.

[81] N. Lawton, G. Ver Steeg, and A. Galstyan. “Deep Residual Partitioning”. In: ().

[82] N. Le Roux and A. W. Fitzgibbon. “A fast natural Newton method.” In: *ICML.* 2010, pp. 623–630.

[83] N. Le Roux, P.-A. Manzagol, and Y. Bengio. “Topmoumoute online natural gradient algorithm”. In: *Advances in neural information processing systems* 20 (2007).

[84] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[85] Y. Lu, M. Harandi, R. Hartley, and R. Pascanu. *Block Mean Approximation for Efficient Second Order Optimization.* 2018. arXiv: 1804.05484.

[86] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. “The expressive power of neural networks: A view from the width”. In: *Advances in neural information processing systems* 30 (2017).

[87] K. Luk and R. Grosse. *A Coordinate-Free Construction of Scalable Natural Gradient.* 2018. arXiv: 1808.10340.

[88] K. Maile, E. Rachelson, H. Luga, and D. G. Wilson. "When, where, and how to add new neurons to ANNs". In: *Proceedings of the First International Conference on Automated Machine Learning*. Ed. by I. Guyon, M. Lindauer, M. van der Schaar, F. Hutter, and R. Garnett. Vol. 188. Proceedings of Machine Learning Research. PMLR, 2022, pp. 18/1–12. URL: https://proceedings.mlr.press/v188/maile22a.html.

[89] F. Marini, M. Porcelli, and E. Riccietti. *A multilevel stochastic regularized first-order method with application to finite sum minimization*. 2024. arXiv: 2412.11630.

[90] J. Martens et al. "Deep learning via hessian-free optimization." In: *Icml*. Vol. 27. 2010, pp. 735–742.

[91] J. Martens and R. Grosse. "Optimizing neural networks with Kronecker-factored approximate curvature". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by F. Bach and D. Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 2408–2417. URL: http://proceedings.mlr.press/v37/martens15.html.

[92] R. Mitchell, R. Menzenbach, K. Kersting, and M. Mundt. *Self-expanding neural networks*. 2023. arXiv: 2307.04526.

[93] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida. *Spectral Normalization for Generative Adversarial Networks*. 2018. arXiv: 1802.05957.

[94] S. Müller, S. Petra, and M. Zisler. "Multilevel Geometric Optimization for Regularised Constrained Linear Inverse Problems". In: (2022). arXiv: 2207.04934.

[95] S. G. Nash. "A multigrid approach to discretized optimization problems". In: *Optimization Methods and Software* 14.1-2 (2000), pp. 99–116. DOI: 10.1080/10556780008805795.

[96] S. G. Nash. "Properties of a class of multilevel optimization algorithms for equality-constrained problems". In: *Optimization Methods and Software* 29.1 (2013), pp. 137–159. DOI: 10.1080/10556788.2012.759571.

[97] B. Neyshabur, R. Tomioka, R. Salakhutdinov, and N. Srebro. *Data-Dependent Path Normalization in Neural Networks*. 2015. arXiv: 1511.06747.

[98] J. Nocedal and S. J. Wright. *Numerical Optimization*. 2nd ed. New York: Springer, 2006. DOI: 10.1007/978-0-387-40065-5.

[99] Y. Ollivier. *Riemannian metrics for neural networks I: feedforward networks*. 2013. arXiv: 1303.0818.

[100] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, C.-S. Foo, and R. Yokota. "Scalable and Practical Natural Gradient for Large-Scale Deep Learning". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.1 (2022), pp. 404–415. DOI: 10.1109/tpami.2020.3004354.

[101] 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*. 2018. arXiv: 1811.12019.

[102] P. Parpas. "A multilevel proximal gradient algorithm for a class of composite optimization problems". In: *SIAM Journal on Scientific Computing* 39.5 (2017), S681–S701. DOI: 10.1137/16m1082299.

[103] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, 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". In: *Advances in Neural Information Processing Systems.* Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. NeurIPS'19. Curran Associates, Inc., 2019. URL: https://papers.nips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[104] A. Pinkus. "Approximation theory of the MLP model in neural networks". In: *Acta Numerica* 8 (1999), pp. 143–195. DOI: 10.1017/s0962492900002919.

[105] B. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17. DOI: https://doi.org/10.1016/0041-5553(64)90137-5.

[106] D. Povey, X. Zhang, and S. Khudanpur. *Parallel training of DNNs with Natural Gradient and Parameter Averaging.* 2014. arXiv: 1410.7455.

[107] M. Ranzato, Y.-L. Boureau, Y. Cun, et al. "Sparse feature learning for deep belief networks". In: *Advances in neural information processing systems* 20 (2007).

[108] Y. Ren and D. Goldfarb. *Efficient Subsampled Gauss-Newton and Natural Gradient Methods for Training Neural Networks.* 2019. arXiv: 1906.02353.

[109] H. Robbins and S. Monro. "A stochastic approximation method". In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407. DOI: 10.1214/aoms/1177729586.

[110] F. Rosenblatt et al. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms.* Vol. 55. Spartan books Washington, DC, 1962.

[111] T. Salimans and D. P. Kingma. *Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks.* 2016. arXiv: 1602.07868.

[112] N. N. Schraudolph. "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent". In: *Neural Computation* 14.7 (2002), pp. 1723–1738. DOI: 10.1162/08997660260028683.

[113] O. Sebbouh, R. M. Gower, and A. Defazio. "Almost sure convergence rates for stochastic gradient descent and stochastic heavy ball". In: *Conference on Learning Theory.* PMLR. 2021, pp. 3935–3971.

[114] H. Sedghi, V. Gupta, and P. M. Long. *The Singular Values of Convolutional Layers.* 2018. arXiv: 1805.10408.

[115] J. Sherman and W. J. Morrison. "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix". In: *The Annals of Mathematical Statistics* 21.1 (1950), pp. 124–127.

[116] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* 2014. arXiv: 1409.1556.

[117] S. P. Singh and D. Alistarh. "Woodfisher: Efficient second-order approximation for neural network compression". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 18098–18109.

[118] S. Soori, B. Can, B. Mu, M. Gürbüzbalaban, and M. M. Dehnavi. *TENGraD: Time-Efficient Natural Gradient Descent with Exact Fisher-Block Inversion.* 2021. arXiv: 2106.03947.

[119] Z. Tang, F. Jiang, M. Gong, H. Li, Y. Wu, F. Yu, Z. Wang, and M. Wang. “Skfac: Training neural networks with faster kronecker-factored approximate curvature”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2021, pp. 13479–13487.

[120] M. Tenorio and W.-T. Lee. “Self-organizing network for optimum supervised learning”. In: *IEEE Transactions on Neural Networks* 1.1 (1990), pp. 100–110. DOI: 10.1109/72.80209.

[121] N. N. Vakhania, V. I. Tarieladze, and S. A. Chobanyan. “Covariance Operators”. In: *Probability Distributions on Banach Spaces.* Springer Netherlands, 1987, pp. 144–183. DOI: 10.1007/978-94-009-3873-1_3.

[122] N. Vater and A. Borzì. “Training Artificial Neural Networks with Gradient and Coarse-Level Correction Schemes”. In: *Machine Learning, Optimization, and Data Science.* Springer International Publishing, 2022, pp. 473–487. DOI: 10.1007/978-3-030-95467-3_34.

[123] A. Veprikov, A. Bolatov, S. Horváth, A. Beznosikov, M. Takáč, and S. Hanzely. *Preconditioned Norms: A Unified Framework for Steepest Descent, Quasi-Newton and Adaptive Methods.* 2025. arXiv: 2510.10777.

[124] M. Verbockhaven, S. Chevallier, and G. Charpiat. “Growing tiny networks: spotting expressivity bottlenecks and fixing them optimally”. In: 2023. URL: https://www.lri.fr/~gcharpia/Expressivity_bottlenecks_preprint.pdf.

[125] O. Vinyals and D. Povey. “Krylov subspace descent for deep learning”. In: *Artificial intelligence and statistics.* PMLR. 2012, pp. 1261–1268.

[126] C. von Planta, A. Kopaničáková, and R. Krause. *Training of deep residual networks with stochastic MG/OPT.* 2021. arXiv: 2108.04052.

[127] X. Wang, S. Ma, D. Goldfarb, and W. Liu. “Stochastic Quasi-Newton Methods for Nonconvex Stochastic Optimization”. In: *SIAM Journal on Optimization* 27.2 (2017), pp. 927–956. DOI: 10.1137/15m1053141.

[128] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference.* Springer Texts in Statistics. Springer New York, 2013. URL: https://books.google.de/books?id=qrcuBAAAQBAJ.

[129] T. Wei, C. Wang, Y. Rui, and C. W. Chen. “Network morphism”. In: *Proceedings of The 33rd International Conference on Machine Learning.* Ed. by M. F. Balcan and K. Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. PMLR, 2016, pp. 564–572. arXiv: 1603.01670. URL: https://proceedings.mlr.press/v48/wei16.html.

[130] W. Wen, F. Yan, Y. Chen, and H. Li. “AutoGrow: automatic layer growing in deep convolutional networks”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* ACM, 2020. DOI: 10.1145/3394486.3403126.

[131] Z. Wen and D. Goldfarb. “A line search multigrid method for large-scale nonlinear optimization”. In: *SIAM Journal on Optimization* 20.3 (2010), pp. 1478–1503. DOI: 10.1137/08071524x.

[132] J. Wu, D. Zou, V. Braverman, and Q. Gu. *Direction Matters: On the Implicit Bias of Stochastic Gradient Descent with Moderate Learning Rate.* 2020. arXiv: 2011.02538.

[133] L. Wu, B. Liu, P. Stone, and Q. Liu. “Firefly neural architecture descent: a general approach for growing neural networks”. In: *Advances in Neural Information Processing Systems.* Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 22373–22383. arXiv: 2102.08574. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/fdbe012e2e11314b96402b32c0df26b7-Paper.pdf.

[134] L. Wu, D. Wang, and Q. Liu. “Splitting steepest descent for growing neural architectures”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Vol. 32. NeurIPS’19. Curran Associates, Inc., 2019. arXiv: 1910.02366. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/3a01fc0853ebeba94fde4d1cc6fb842a-Paper.pdf.

[135] Y. Wu and K. He. *Group Normalization*. 2018. arXiv: 1803.08494.

[136] M. Wynne-Jones. “Node splitting: A constructive algorithm for feed-forward neural networks”. In: *Advances in neural information processing systems* 4 (1991).

[137] G. Zhang, J. Martens, and R. B. Grosse. “Fast convergence of natural gradient descent for over-parameterized neural networks”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[138] H. Zhang, C. Xiong, J. Bradbury, and R. Socher. *Block-diagonal Hessian-free Optimization for Training Neural Networks*. 2017. arXiv: 1712.07296.

[139] J. Zhang, H. Li, S. Sra, and A. Jadbabaie. “Neural network weights do not converge to stationary points: An invariant measure perspective”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 26330–26346.

A. Notation

Notation Deep Learning

The following notation is used throughout this thesis for deep neural networks, in particular Feedforward Neural Networks (FNNs). Notation may be adapted for specific architectures such as Convolutional Neural Networks (CNNs) or Residual Neural Networks (ResNets) as needed and is explained in the respective sections.

- Number of layers: L
- Input space: X_0 with inner product $(\cdot,\cdot)_{X_0}$ and dimension n_{X_0} with elements x_0
- Pre-activation spaces: Y_i for $i = 1, \dots, L$ with inner products $(\cdot,\cdot)$ and elements y_i
- Dimension of pre-activation spaces: n_{Y_i} for $i = 1, \dots, L$
- Post-activation spaces: X_i for $i = 1, \dots, L-1$ with inner products $(\cdot,\cdot)_{X_i}$ and elements x_i
- Dimension of post-activation spaces: n_{X_i} for $i = 1, \dots, L-1$
- Output space: Y_L with inner product $(\cdot,\cdot)_{Y_L}$ with dimension n_{Y_L} (is last pre-activation space)
- Dual layer spaces: Y_i^* for $i = 1, \dots, L$ with inner products $(\cdot,\cdot)_{Y_i^*}$ and X_i^* for $i = 0, \dots, L-1$ with inner products $(\cdot,\cdot)_{X_i^*}$
- Dual features: $\frac{\partial f_j(\theta)}{\partial y_i}$ and $\frac{\partial f_j(\theta)}{\partial x_i}$ representing the derivative of the objective function w.r.t. the pre- and post-activation features
- Riesz maps: $\mathcal{R}_{Y_i} \in L(Y_i, Y_i^*)$ for $i = 1, \dots, L$ and $\mathcal{R}_{X_{i-1}} \in L(X_i, X_i^*)$ for $i = 0, \dots, L-1$
- Weight matrices: $W_i \in L(X_{i-1}, Y_i)$ or $W_i \in \mathbb{R}^{n_{X_i} \times n_{X_{i-1}}}$ for $i = 1, \dots, L$
- Bias vectors: $b_i \in \mathbb{R}^{n_{X_i}}$ for $i = 1, \dots, L-1$
- Activation functions: $\sigma_i \colon Y_i \to X_i$ for $i = 1, \dots, L-1$
- All trainable parameters (FNN): $\theta = (W_i, b_i)_{i=1}^{L} \in \Theta$

Forward pass through network (FNN):

$$\begin{aligned} y_i &= W_i(x_{i-1}) + b_i \quad \text{for } i = 1, \dots, L \\ x_i &= \sigma_i(y_i) \quad \text{for } i = 1, \dots, L-1 \end{aligned}$$

- Network function: $g(\theta,\cdot) : X_0 \longrightarrow Y_L$
- Loss function: $\mathcal{L} : Y_L \times Y_L \longrightarrow \mathbb{R}$
- Training data set: D pairs $(x_{0,j}, y_{L,j})$ for $j = 1, \dots, D$
- Objective function: $f : \Theta \longrightarrow \mathbb{R}$ often given by $\hat{f}(\theta) = \frac{1}{D}\sum_{j=1}^{D} f_j(\theta)$ with

$$f_j(\theta) \coloneqq \mathcal{L}(g(\theta, x_{0,j}), y_{L,j})$$

Backward pass through network (FNN):

$$\begin{aligned} \frac{\partial f_j(\theta)}{\partial x_i} &\in X_i^* \quad \text{for } i = 1, \dots, L \\ \frac{\partial f_j(\theta)}{\partial y_i} &\in Y_i^* \quad \text{for } i = 1, \dots, L-1 \\ W_i^* &\in L(Y_i^*, X_{i-1}^*) \text{ is the adjoint of } W_i \in L(X_{i-1}, Y_i) \text{ for } i = 1, \dots, L. \end{aligned}$$

The chain rule produces connections between the derivatives of the pre- and post-activation features:

$$\begin{aligned} \frac{\partial f_j(\theta)}{\partial x_{i-1}}(\cdot) &= \frac{\partial f_j(\theta)}{\partial y_i}(W_i\ \cdot) \quad \text{for } i = 1, \dots, L \\ \frac{\partial f_j(\theta)}{\partial y_i}(\cdot) &= \frac{\partial f_j(\theta)}{\partial x_i}(\sigma_i'(y_i)\ \cdot) \quad \text{for } i = 1, \dots, L-1. \end{aligned}$$

The backpropagated Euclidean gradients of the pre- and post-activation features are given by:

$$\begin{aligned} \nabla_{x_{i-1}} f_j(\theta) &= W_i^T \nabla_{y_i} f_j(\theta) \quad \text{for } i = 1, \dots, L \\ \nabla_{y_i} f_j(\theta) &= \sigma_i'(y_i)^T \nabla_{x_i} f_j(\theta) \quad \text{for } i = 1, \dots, L-1. \end{aligned}$$

Generally, the transposes can be understood of adjoints with respect to the Euclidean inner products on the layer spaces.

Derivatives of f_j w.r.t. the trainable parameters:

$$\begin{aligned} \frac{\partial f_j(\theta)}{\partial W_i} &\in L(X_{i-1}, Y_i)^* = L(Y_i^*, X_{i-1}^*) \qquad \text{with } \frac{\partial f_j(\theta)}{\partial W_i}(\delta W) = \frac{\partial f_j(\theta)}{\partial y_i}(\delta W x_{i-1}) \quad \text{for } i = 1, \dots, L \\ \frac{\partial f_j(\theta)}{\partial b_i} &\in Y_i^* \qquad \text{with } \frac{\partial f_j(\theta)}{\partial b_i}(\delta b) = \frac{\partial f_j(\theta)}{\partial y_i}(\delta b) \quad \text{for } i = 1, \dots, L. \end{aligned}$$

Euclidean gradient of f_j w.r.t. a weight matrix W and bias vector b:

$$\begin{aligned} \nabla_{W_i} f_j(\theta) &= \nabla_{y_i} f_j(\theta) x_{i-1}^T \\ \nabla_{b_i} f_j(\theta) &= \nabla_{y_i} f_j(\theta). \end{aligned}$$

Notation SGD and MG/Opt

Symbol	**Space**	**Name**
Θ	-	fine optimization parameter space
n	$\mathbb{N}$	fine optimization parameter dimension
f	$\Theta \to \mathbb{R}$	fine objective function
θ	Θ	fine optimization parameter
α_k	$\mathbb{R}_{>0}$	step size (learning rate)
k	$\mathbb{N}_0$	iteration number
θ^0	Θ	initial fine parameter
θ^k	Θ	current (fine) parameter
L	$\mathbb{R}_{>0}$	Lipschitz constant (for gradient of f)
μ	$\mathbb{R}_{>0}$	strong convexity constant of f
ξ	-	random variable (for gradient estimate of f)
B	$\mathbb{N}$	batch size
$G_f(\cdot, \xi)$	Θ	gradient estimate for f
M, M_G	$\mathbb{R}_{>0}$	variance bounds for stochastic gradients
m^k	Θ	momentum term
μ^k	$[0, 1)$	momentum parameter

Table A.1.: Notation SGD

Symbol	**Space**	**Name**
Θ_H	-	coarse optimization parameter space
n_H	$\mathbb{N}$	coarse optimization parameter dimension
f_H	$\Theta_H \to \mathbb{R}$	coarse objective function
$\bar{\theta}$	Θ_H	coarse optimization parameter
$\mathcal{R}_p, \mathcal{R}_g$	$\Theta \to \Theta_H$	restriction operator (linear) for parameters and gradients
c, P, ρ	$\mathbb{R}_{>0}$	constants from assumption 3.18
$\mathcal{P}_p$	$\Theta_H \to \Theta$	prolongation operator (linear)
$\bar{\theta}^k$	Θ_H	current fine parameter restricted, $\bar{\theta}^k = \mathcal{R}_p \theta^k$
$\bar{\theta}^k_K$	Θ_H	coarse parameter at iteration K of iterations on h
$\bar{v}_k$	Θ_H^*	shift for the cgc objective function
h_k	$\Theta_H \to \mathbb{R}$	cgc objective function (sometimes simplified as h)
λ	$\mathbb{R}_{>0}$	regularization parameter
τ	$\mathbb{R}_{>0}$	constant for bound of regularization (cf. assumption 3.18)
$\bar{\theta}^*_{cgc}$	Θ_H	minimizer of the cgc objective (or approximate stationary point)
$\bar{e}$	Θ_H	coarse grid correction without prolongation, $\bar{e} = \bar{\theta}^*_{cgc} - \mathcal{R}_p \theta^k$
e	Θ	coarse grid correction, generated by MG/Opt via $e = \bar{e}$
$\bar{\alpha}$	$\mathbb{R}_{>0}$	step size for coarse objective
α_{cgc}	$\mathbb{R}_{>0}$	step size for the cgc
κ	$\mathbb{R}_{>0}$	threshold for the cgc
A_{fine}	$\mathbb{R}^{n \times n}$	matrix of fine quadratic problem
b_{fine}	$\mathbb{R}^n$	vector of fine quadratic problem
A_{coarse}	$\mathbb{R}^{n_H \times n_H}$	matrix of coarse quadratic problem, $A_{\text{coarse}} = \mathcal{R}_p A_{\text{fine}} \mathcal{P}_p$
b_{coarse}	$\mathbb{R}^{n_H}$	vector of coarse quadratic problem, $b_{\text{coarse}} = \mathcal{R}_p b_{\text{fine}}$
$G_{f_H}(\cdot, \xi)$	Θ_H	gradient estimate for f_H (random object)
$G_{cgc,k}(\cdot, \xi)$	Θ_H	gradient estimate for h at iteration k (random object)
$\bar{L}$	$\mathbb{R}_{>0}$	Lipschitz constant (for gradient of coarse objective function)
$\bar{\mu}$	$\mathbb{R}_{>0}$	strong convexity constant (for coarse objective function)
L_{mb}	$\mathbb{R}_{>0}$	Lipschitz constant (for stochastic gradients of fine objective function)
$\bar{M}, \bar{M}_G$	$\mathbb{R}_{>0}$	variance bounds for stochastic gradients of coarse objective function
K	$\mathbb{N}$	number of cgc iterations
ϵ	$\mathbb{R}$	decrease in cgc function value
d_1	$\mathbb{R}_{>0}$	lower bound for the cgc (defined in assumption 3.22)
v	$\mathbb{R}_{>0}$	control for cgc iterates (defined in assumption 3.23)

Table A.2.: Notation MG/Opt

Notation SensLI

Symbol	**Space**	**Name**
θ_{base}	Θ	parameters of baseline network
g	$\Theta \times X_0 \to Y_L$	baseline network
$\hat{f}_{\text{base}}$	$\Theta \to \mathbb{R}$	training objective function of baseline network
n_{ext}	$\mathbb{N}$	number of parameters of extended network
θ_{ext}	Θ_{ext}	parameters of extended network $\theta_{\text{ext}} = \begin{pmatrix} \theta^T & \theta_{\text{new}}^T \end{pmatrix}^T$
n_{new}	$\mathbb{N}$	number of additional parameters
θ_{new}	Θ_{new}	additional parameters of extended network
g_{ext}	$\Theta_{\text{ext}} \times X_0 \to Y_L$	extended network
$\hat{f}_{\text{ext}}$	$\Theta_{\text{ext}} \to \mathbb{R}$	training objective function of extended network
c	$\Theta_{\text{ext}} \to \mathbb{R}^{n_{\text{new}}}$	constraint function
$\mathcal{L}$	$\Theta_{\text{ext}} \times \mathbb{R}^{n_{\text{new}}} \to \mathbb{R}$	Lagrangian function
$T_c(\theta_{\text{ext}})$	-	critical cone of the constraints
ϵ	$\mathbb{R}_{>0}$	perturbation parameter
Δ	Θ_{new}	perturbation vector
λ	$\mathbb{R}^{n_{\text{new}}}$	Lagrange multipliers
M	$\mathbb{R}^{n_{\text{new}} \times n_{\text{ext}}}$	constraint matrix
m	$\mathbb{R}^{n_{\text{new}}}$	constraint vector (contains initializations for θ_{new})
$\mathcal{W}$	-	set of all weight matrices in a neural network

Table A.3.: Notation SensLI

Notation Frobenius Preconditioner

Symbol	**Space**	**Name**
$\mathcal{H}_1, \mathcal{H}_2$	-	real Hilbert spaces
$\mathcal{H}_i^*$	-	dual space of $\mathcal{H}_i$ for $i = 1, 2$
$\mathcal{R}_i$	$L(\mathcal{H}_i, \mathcal{H}_i^*)$	Riesz map for $\mathcal{H}_i$ for $i = 1, 2$
$L(\mathcal{H}_1, \mathcal{H}_2)$	-	space of linear maps from $\mathcal{H}_1$ to $\mathcal{H}_2$
S	$L(\mathcal{H}_1, \mathcal{H}_2)$	linear map from $\mathcal{H}_1$ to $\mathcal{H}_2$
S^*	$L(\mathcal{H}_2, \mathcal{H}_1)$	Hilbert space adjoint of S
S'	$L(\mathcal{H}_2^*, \mathcal{H}_1^*)$	dual map of S
$(\cdot, \cdot)_{\mathcal{H}_1 \to \mathcal{H}_2}$	-	Frobenius-type inner product on space of linear maps L($\mathcal{H}_1, \mathcal{H}_2$)
$F(\theta)$	$\mathbb{R}^{n_\Theta \times n_\Theta}$	Fisher Information Matrix (dep. on parameters θ)
$\tilde{F}(\theta)$	$\mathbb{R}^{n_\Theta \times n_\Theta}$	approximate Fisher Information Matrix by K-FAC
$\mathcal{R}_{X_i}$	$L(Y_i, Y_i^*)$	Riesz map for pre-activation space for layer i
$\mathcal{R}_{Y_i}$	$L(X_i, X_i^*)$	Riesz map for post-activation space for layer $i - 1$
$R_{X_{i-1}}$	$\mathbb{R}^{n_{X_{i-1}} \times n_{X_{i-1}}}$	inner product matrix for post-activation space for layer $i - 1$
R_{Y_i}	$\mathbb{R}^{n_{Y_i} \times n_{Y_i}}$	inner product matrix for pre-activation space for layer i
$\mathrm{Cov}(\cdot)$	-	exact covariance matrix
$\hat{\mathrm{Cov}}(\cdot)$	-	covariance matrix estimator
γ	$\mathbb{R}_{>0}$	damping parameter for preconditioner
$x_{i-1}(j)$	X_{i-1}	j-th post-activation feature sample
$\nabla_{y_i} f(\theta)(j)$	Y_i	j-th pre-activation feature gradient sample (with modified random label)

Table A.4.: Notation for Preconditioning

List of Figures

List of Tables

B. Appendix

B.1. Coarse Grid Correction as Descent Direction - Additional Proofs

Proof of Theorem 2.36. Using the definition of the coarse grid correction in (2.45) and the exactness of the coarse problem, we have

$$r = \nabla h(\bar{\theta}^*_{cgc}) = \nabla f_H(\bar{\theta}^*_{cgc}) + \mathcal{R}_g \nabla f(\theta^k) - \nabla f_H(\bar{\theta}^k) - \lambda \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right).$$

We start by bounding

$$\begin{aligned}
\nabla f(\theta^k)^\top e =& \nabla f(\theta^k)^\top \mathcal{P}_p \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right) \\
=& \left(\mathcal{R}_g \nabla f(\theta^k)\right)^\top \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right) \\
=& \left(\nabla f_H(\bar{\theta}^k) - \nabla f_H(\bar{\theta}^*_{cgc}) + r + \lambda(\bar{\theta}^*_{cgc} - \bar{\theta}^k)\right)^\top \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right) \\
=& - \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right)^\top (\nabla^2 f_H(\xi) + \lambda) \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right) + r^\top \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right) \\
\leq& -(\mu_H + \lambda) \|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 + r^\top \left(\bar{\theta}^*_{cgc} - \bar{\theta}^k\right),
\end{aligned}$$

where we used the mean value theorem in the last step to guarantee the existence of some ξ on the line segment between $\bar{\theta}^k$ and $\bar{\theta}^*_{cgc}$. □

Proof of Theorem 2.38. It is

$$h(\bar{\theta}^*_{cgc}) - h(\bar{\theta}^k) \leq 0.$$

This is equivalent to

$$\begin{aligned}
&0 \geq h(\bar{\theta}^*_{cgc}) - h(\bar{\theta}^k) \\
&\Leftrightarrow 0 \geq f_H(\bar{\theta}^*_{cgc}) - f_H(\bar{\theta}^k) + (\bar{\theta}^*_{cgc} - \bar{\theta}^k)^\top (\mathcal{R}_g \nabla f(\theta^k) - \nabla f_H(\bar{\theta}^k)) + \frac{\lambda}{2} \|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \\
&\Leftrightarrow -\mathcal{R}_g \nabla f(\theta^k)^\top (\bar{\theta}^*_{cgc} - \bar{\theta}^k) \geq f_H(\bar{\theta}^*_{cgc}) - f_H(\bar{\theta}^k) + (\bar{\theta}^*_{cgc} - \bar{\theta}^k)^\top (-\nabla f_H(\bar{\theta}^k)) + \frac{\lambda}{2} \|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \\
&\Leftrightarrow \nabla f(\theta^k)^\top e \leq (\bar{\theta}^*_{cgc} - \bar{\theta}^k)^\top \nabla f_H(\bar{\theta}^k) - f_H(\bar{\theta}^*_{cgc}) + f_H(\bar{\theta}^k) - \frac{\lambda}{2} \|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \\
&\Leftrightarrow \nabla f(\theta^k)^\top e \leq (\bar{\theta}^*_{cgc} - \bar{\theta}^k)^\top (\nabla f_H(\bar{\theta}^k) - \nabla f_H(\bar{\theta}_t)) - \frac{\lambda}{2} \|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \\
&\Leftrightarrow \nabla f(\theta^k)^\top e \leq -(\bar{\theta}^*_{cgc} - \bar{\theta}^k)^\top \nabla^2 f_H(\bar{\theta}_{\bar{t}}) t (\bar{\theta}^*_{cgc} - \bar{\theta}^k) - \frac{\lambda}{2} \|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2 \\
&\Leftrightarrow \nabla f(\theta^k)^\top e \leq -(\bar{\theta}^*_{cgc} - \bar{\theta}^k)^\top (t\mu_H + \frac{\lambda}{2} I)(\bar{\theta}^*_{cgc} - \bar{\theta}^k) \\
&\Leftrightarrow \nabla f(\theta^k)^\top e \leq -(t\mu_H + \frac{\lambda}{2}) \|\bar{\theta}^*_{cgc} - \bar{\theta}^k\|^2.
\end{aligned}$$

We used the mean value theorem in two steps to guarantee the existence of some $\bar{\theta}_{\bar{t}}$ on the line segment between $\bar{\theta}^k$ and $\bar{\theta}^*_{cgc}$ for $t \in (0,1)$ and the existence of some $\bar{\theta}_{\bar{t}}$ on the line segment between $\bar{\theta}^k$ and $\bar{\theta}_t = \bar{\theta}^k + t(\bar{\theta}^*_{cgc} - \bar{\theta}^k)$. □

B.2. Detailed Experiments Setup for MG/Opt

The code with which the experiments were computed is available on `GitHub`[1].

Strong Convex Quadratic Coarse Grid Correction with Moderate and Small Learning Rate

Construction of the objective functions: We construct the matrix A_{fine} by generating 50 random eigenvalues via `np.random.uniform(0.1, 0.5, 25)` and `np.random.uniform(1.0, 5.0, 25)`. Then, we generate a random orthogonal matrix Q via the command `scipy.stats.special_ortho_group.rvs(50)`. Finally, we set $A_{\text{fine}} = Q^\top \operatorname{diag}(\lambda_1, \ldots, \lambda_{50})\, Q$. We construct b_{fine} via `np.random.randn(50)`.

The restriction operators which we compare map to $\mathbb{R}^{25}$ and are constructed by

(i) *randomev*: a random selection of 25 eigenvectors of A_{fine} as rows,

(ii) *smallev*: a selection of the 25 eigenvectors of A_{fine} corresponding to the smallest eigenvalues as rows,

(iii) *largeev*: a selection of the 25 eigenvectors of A_{fine} corresponding to the largest eigenvalues as rows.

The prolongation operator is always the transpose of the restriction operator and $\mathcal{R}_p = \mathcal{R}_g$. The coarse

[1]`https://github.com/LeonieKreis/quadratic_multilevel`

problem is the constructed as described in Section 3.3.1.

The stochastic gradient estimates are constructed by adding Gaussian noise to the true gradient, i.e.
```
theta = np.random.uniform(-siga, siga,50),
eps = np.random.uniform(-sigb, sigb),
A = A + theta.T @ theta -2/3 * siga * np.eye(50),
b = b + eps ,
```
with `siga = sigb = 0.1`.

The gradient estimates of the coarse problem are constructed as the construction of the coarse quantities $A_{\text{coarse}}, b_{\text{coarse}}$. The setup is as follows:

- **Figures:** Figure 3.1, Figure 3.2
- **Iterations:** 50 SGD updates with fixed learning rate (0.01 for moderate and 0.001 for small learning rate). Then, a cgc step is computed with 50 SGD iterations on the coarse level with the same learning rates if a criterion is satisfied. The criterion reads

$$\|\mathcal{R}_g \nabla f(\theta^k)\| \geq \kappa_1 \|\nabla f(\theta^k)\| \text{ and } \|\mathcal{R}_g \nabla f(\theta^k)\| \geq \kappa_2,$$

  where $\nabla f(\theta^k)$ is the fine gradient at current iteration $k = 50$ and $\mathcal{R}_g$ is the restriction operator for the gradients. We set $\kappa_1 = 0.5$ and $\kappa_2 = 0.001$. The cgc update is then applied to the fine level with a step size of 1.0. After the cgc update, 150 SGD iterations are performed on the fine level with the learning rates used before the cgc step.
- **Smoothing in Figures:** None.

B.3. Sensitivity Norm Computation

In Convolutional Neural Networks (CNNs), the gradient with respect to the kernel, $\nabla_K \hat{f}_{\text{ext}}(\theta_{\text{ext}})$, is a tensor of higher dimensionality compared to the gradient of a weight matrix in, for example, a FNN. As a result, it is important to select a norm that is well-suited to the structure of the kernel tensor. Several options are available: using the Frobenius norm, as in (B.1a), analogous to the approach for fully-connected networks, is a rather uninformed choice. To facilitate comparison between layers with different kernel sizes, one can scale the Frobenius norm by the number of elements in the kernel tensor, as shown in (B.1b).

Alternatively, one may adopt a different perspective by considering the operator norm of the linear map A_K that represents the convolution operation with kernel K. Here, the indices i and j correspond to the spatial filter positions, while k and ℓ refer to the input and output channels, respectively. The operator norm, which is the largest singular value of A_K, is defined in (B.1c). This norm can be computed efficiently using the method described in [114], with a computational complexity of $O(d^2c^3)$, where d denotes the spatial size of the convolutional layers and c is the number of input or output channels.

To further refine the measurement of the impact of different output channels, one can compute the $2 \leftarrow 2$ norm over the input channels for each output channel individually, and then aggregate these

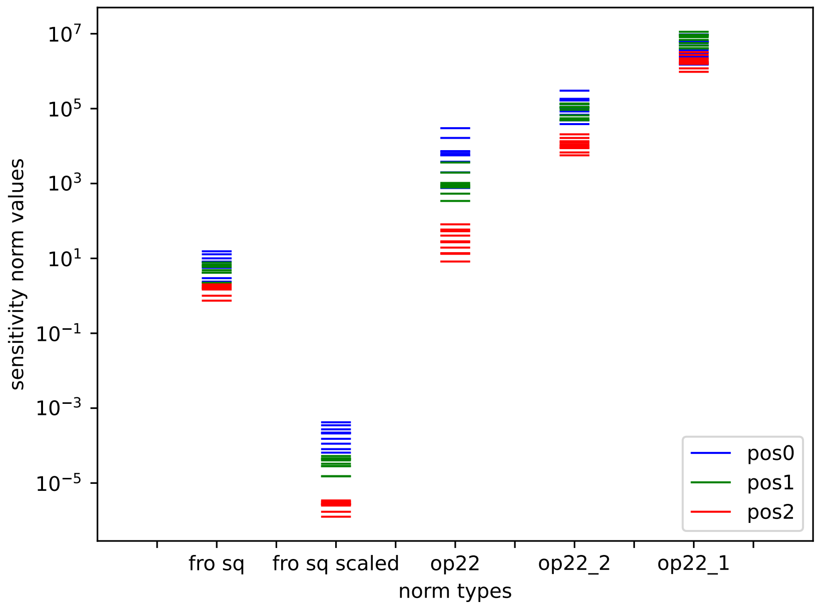


Figure B.1.: Comparison of different norms for the sensitivity of layer insertion in CNNs. From left to right we show $\|K\|_F^2$, (B.1a), $\|K\|_{F,\text{scaled}}^2$, (B.1b), $\|K\|_{2\leftarrow 2}^2$, (B.1c), $\|K\|_{2,2\leftarrow 2}^2$, (B.1d) and $\|K\|_{1,2\leftarrow 2}^2$, (B.1e). We display 10 runs and every line per norm per position indicates the result from one run. The experiment can be found in the GitHub repository as Exp5.

values by taking either the squared 2-norm, as in (B.1d), or the 1-norm, as in (B.1e), over the output channels. Altogether, we define

$$\|K\|_F^2 := \sum_{i,j,k,\ell} K_{i,j,k,\ell}^2, \tag{B.1a}$$

$$\|K\|_{F,\text{scaled}}^2 := \frac{1}{c_i\, c_j\, c_k\, c_\ell} \sum_{i,j,k,\ell} K_{i,j,k,\ell}^2, \tag{B.1b}$$

$$\|K\|_{2\leftarrow 2}^2 := \sup_{\|x\|_2=1} \left\|A_K x\right\|_2^2, \tag{B.1c}$$

$$\|K\|_{2,2\leftarrow 2}^2 := \sum_{\ell \text{ outchannels}} \left\| \sum_{k \text{ inchannels}} K_{:,:,k,\ell} \right\|_{2\leftarrow 2}^2, \tag{B.1d}$$

$$\|K\|_{1,2\leftarrow 2}^2 := \left(\sum_{\ell \text{ outchannels}} \left\| \sum_{k \text{ inchannels}} K_{:,:,k,\ell} \right\|_{2\leftarrow 2} \right)^2, \tag{B.1e}$$

where K is the kernel tensor, c_i and c_j are the spatial filter sizes, and c_k and c_ℓ are the number of input and output channels, respectively. Further, A_K is the matrix which represents the convolution operation with kernel K as a linear map.

We numerically compare the behavior of the different sensitivity norms in Figure B.1, using results from 10 runs of the same experiment. Further experimental details are provided in the appendix of [57]. In this setup, we analyze a CNN where an additional layer is inserted after 50 epochs. Across all runs, the training process is highly consistent, and position 0, i.e. the first possible position, consistently emerges as the most favorable location for layer insertion. The five norms differ in their orders of magnitude, and some do not clearly distinguish between the possible insertion positions, as seen in Figure B.1. Among the options, the operator norm stands out as the most effective for our approach, as it provides a clear separation between insertion positions. The Frobenius norm, when scaled by the number of elements in the kernel tensor, also serves as a reasonable alternative.

B.4. Detailed Experiments Setup for Frobenius-type Preconditioners

The repository which contains the code used to perform the experiments is available on `GitHub`[2].

Experiment 1 (Classification with ReLU-FNN)

- **Figures:** Figure 5.1, Figure 5.3
- **Data set:** MNIST, flattened images.
- **Data set size** 70 000.
- **Train/Test split** 60 000/10 000.
- **Batch size:** 100.
- **Architecture:** FNN classifier with widths 784-100-100-100-10.
- **Activation function:** ReLU.
- **Loss function:** Cross-Entropy.
- **Optimizer:** SGD with fixed learning rate for 30 epochs.
- **Learning rates for unpreconditioned training:** `np.logspace(0, -5, num=30)`
- **Learning rates for preconditioned training:** `np.logspace(0, -5, num=30)`
- **Covariance preconditioner:**
    - ▹ Second moment estimate as inner product, samples generated with BackPACK [23].
    - ▹ Recomputed every iteration with 100 samples, then applied to the gradient.
    - ▹ Damping parameter for preconditioners fixed to $\lambda = 0.01$.
- **Smoothing in Figures:** Moving average with window size 200.
- **Computation of gradient norms:** Euclidean norm with `torch.norm(p-euclid)`. Frobenius-type with `torch.sqrt(torch.trace(torch.mul(p-precond, p-euclid)))`
- **Computation of spectra:** Eigenvalues of the second moment matrices (not their inverses) with `torch.linalg.eigvals()`.
- Experiment files: `experiments/experiment_files/Exp_mnist_backpack_kfac_new_diss.py`

Experiment 2 (Autoencoder Training)

[2]`https://github.com/LeonieKreis/layerwise_preconditioning_for_neural_networks`

- **Figures:** Figure 5.4, Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8, Figure 5.9
- **Data set:** MNIST, flattened images.
- **Data set size** 70 000.
- **Train/Test split** 60 000/10 000.
- **Batch size:** 1000.
- **Architecture:** FNN autoencoder with widths 784-1000-500-250-30-250-500-1000-784.
- **Activation function:** Sigmoid.
- **Loss function:** Mean Squared Error.
- **Optimizer:** SGD with fixed learning rate for 30 epochs.
- **Learning rates for unpreconditioned training:** `np.logspace(0, -5, 50)`
- **Learning rates for preconditioned training:** `[ 0.2,0.19,0.18,0.17,0.16,0.15,0.14,0.13,0.12,0.11, 0.1, 0.09,0.08,0.07,0.06,0.05]`
- **Covariance preconditioner:**
  - ▹ Second moment estimate as inner product, samples generated with BackPACK [23].
  - ▹ Recomputed every iteration with 1000 samples, then applied to the gradient.
  - ▹ Damping parameter for preconditioners fixed to $\lambda = 0.001$.
- **Smoothing in Figures:** Moving average with window size 200.
- **Computation of gradient norms:** Euclidean norm with `torch.norm(p-euclid)`. Frobenius-type with `torch.sqrt(torch.trace(torch.mul(p-precond, p-euclid)))`
- **Computation of spectra:** Eigenvalues of the second moment matrices (not their inverses) with `torch.linalg.eigvals()`.
- Experiment files: `experiments/experiment_files/Exp_mnist_backpack_kfac_wide_autoencoder_systematic_longerkfac.py`, `experiments/experiment_files/Exp_mnist_backpack_kfac_wide_autoencoder_systematic_longerkfac2.py`, `experiments/experiment_files/Exp_mnist_backpack_kfac_wide_autoencoder_systematic_longerkfac3.py` and `experiments/experiment_files/Exp_mnist_backpack_kfac_wide_autoencoder_systematic_sgd_longer_finerlrs.py`

Experiment 3 (Classification with Shallow FNN)

- **Figures:** Figure 5.10, Figure 5.11, Figure 5.12
- **Data set:** MNIST, flattened images.
- **Data set size** 70 000.
- **Train/Test split** 60 000/10 000.
- **Batch size:** 1000.
- **Architecture:** FNN classifier with widths 784-100-10.
- **Activation function:** Sigmoid.
- **Loss function:** Cross-Entropy.
- **Optimizer:** SGD with fixed learning rate for 30 epochs.

- **Learning rates for unpreconditioned training:** `np.logspace(-5, 0, num=100)`
- **Learning rates for preconditioned training:** `np.logspace(-5, 0, num=100)`
- **Covariance preconditioner:**
  - ▹ Second moment estimate as inner product, samples generated with BackPACK [23].
  - ▹ Recomputed every iteration with 1000 samples, then applied to the gradient.
  - ▹ Damping parameter for preconditioners fixed to $\lambda = 0.001$.
- **Smoothing in Figures:** Moving average with window size 200.
- **Computation of gradient norms:** Euclidean norm with `torch.norm(p-euclid)`. Frobenius-type with `torch.sqrt(torch.trace(torch.mul(p-precond, p-euclid)))`
- **Computation of spectra:** Eigenvalues of the second moment matrices (not their inverses) with `torch.linalg.eigvals()`.
- Experiment files: `experiments/experiment_files/Exp_mnist_backpack_kfac_classification_systematic_longerkfac_finer.py`

Use of AI assistance Parts of the formulations of this thesis were drafted with the assistance of GitHub Copilot. All suggestions and draft text produced by the tool were reviewed, edited, and changed or verified by the author; the tool was mainly used to provide ideas for phrasings that the author adapted as needed. Final content, results and conclusions are the author's responsibility.