FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN

FROM NUMERICAL OPTIMIZATION TO DEEP LEARNING AND BACK

Thomas Frerix

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften
(Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Nils Thuerey
Prüfer der Dissertation:	1.	Prof. Dr. Daniel Cremers
	2.	Prof. Tom Goldstein, Ph.D.

Die Dissertation wurde am 17.01.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15.06.2022 angenommen.

PREFACE

In the past decade, deep learning has created impressive success stories [34]. Arguably, however, few of the major concepts that led to these successes are fundamentally new. Many originate from ideas that already appeared in the 1980s and 1990s. A few examples are convolutional neural networks as the core building blocks of today's computer vision applications [35], backpropagation as the standard training procedure for neural networks [44], and automatic differentiation concepts behind modern libraries [24]. What has changed in the past 10 years? Certainly, some effective neural network architecture elements were presented, such as activation functions [32, 13], residual blocks [26], and the attention mechanism [4]. On top of that, novel neural network optimization algorithms, such as Adam [31], were added to the deep learning stack. However, two main developments stand out: advances in parallel computing hardware and modern automatic differentiation libraries. While the former has significantly increased the number of floating-point operations per second, which is a necessity for large deep learning models, the latter enables broad accessibility of this technology and allows for short development cycles. Today, everyone has access to powerful parallel processing hardware via cloud computing and may start experimenting with deep learning systems by using freely available automatic differentiation frameworks [10, 42, 38].

These advances have catalyzed the development of novel algorithms at the intersection of deep learning and numerical optimization. For example, developing neural network optimization algorithms has become much easier by exploiting the principles behind existing automatic differentiation libraries, as we have performed in [18]. Also, the design of neural network components as differentiable building blocks that can be incorporated in a plug-and-play fashion into existing models is facilitated, such as our constraint module presented in [19]. The formulation of numerical optimization objectives is equally eased: only the objective function must be constructed, whereas derivatives are provided by the automatic differentiation engine. It allows to experiment with complex objective functions, such as in [17], where our optimization objective includes a differential equation solver.

Still, these fields face numerous challenges. For example, deep learning systems are to a great extent based on heuristics, and designing a well-performing system requires intuition as opposed to being based on a theoretical foundation. Such models are then employed as black-box components in complex algorithmic pipelines, often without any guarantees. On a general level, this lack of theoretical principles leads to a research procedure dominated by trial and error, sarcastically termed "graduate student descent" in allusion to the gradient descent optimization algorithm. In terms of large-scale numerical optimization, science and engineering present various difficult non-convex optimization problems that challenge today's most powerful supercomputers. For decades, researchers have improved solution strategies for such problems. Deep learning provides the opportunity to revisit these approaches from a data-driven perspective and may push the frontier of scientific computing algorithms.

This publication-based dissertation comprises four publications that tackle different challenges at the intersection of deep learning and numerical optimization. It begins with a concise introduction to numerical optimization, deep learning, and automatic differentiation, all of

which provide the basis for the presented research. It is intended to be accessible with a background in mathematics as taught in any of the STEM subjects. Subsequently, a summary of the publications as a guide to the research contributions and a conclusion with open questions that are raised by this dissertation are presented. The original publications are reprinted in the appendix.

Despite being the single author of this dissertation, I use the plural pronoun *we* throughout the text to show that my understanding is based on a multitude of sources and would not have come into existence in isolation. Quite the opposite is the case, and I am grateful for the many opportunities of fruitful interaction with my co-authors, my colleagues from our research group, and our scientific community.

Thomas Frerix
Munich, December 2021

ABSTRACT

This dissertation presents research at the intersection of deep learning and numerical optimization. In recent years, the development in these fields has led to impressive results on various computational tasks. This progress is catalyzed by advances in parallel computation hardware and automatic differentiation libraries, which enable new algorithms and their applications in an increasing number of domains by writing differentiable codes, namely, parameterized programs that can be optimized by derivative-based optimization. Despite these success stories, numerous challenges remain.

The non-convex optimization problem for training a deep learning system is commonly approached with heuristic first-order optimization, which leaves much room for the development of neural network optimization algorithms. We contribute a framework to derive deep learning optimization algorithms from an alternative view of the classical backpropagation algorithm. We analyze one such method in detail, Proximal Backpropagation, which takes proximal steps instead of gradient steps for certain components of the model. Our approach allows to examine similar algorithms with an analogous analysis.

Another challenge is that deep learning models are often used as black-box functions within a large algorithmic framework, without providing any guarantees about their predictions. However, in many scenarios, prior knowledge about the learning task is available, for example about the geometry or the physics of a problem. Incorporating this information into the model is desirable, as otherwise the training procedure is data-inefficient and may even produce detrimental results. Such prior knowledge can often be formulated as constraints on model predictions. We develop a module to enforce constraints on the neural network output via a differentiable parameterization of the feasible set in the specific case of homogeneous linear inequality constraints.

The above approaches draw from ideas in numerical optimization to improve deep learning. Vice versa, deep learning systems may be used to advance classical optimization procedures for non-convex problems emerging from applications in science and engineering. We demonstrate one such approach in the context of variational data assimilation. Here, we improve the optimizability of the nonlinear least-squares problem by learning an approximate inverse to the observation operator. We use this learned inverse to transform the objective function and to improve the initialization of the optimizer. The outlined method may be used as a template to improve solvers for nonlinear inverse problems.

In the final publication, we develop a continuous relaxation of a combinatorial optimization problem with applications in numerical linear algebra, namely, the approximate factorization of an orthogonal matrix as a product of Givens matrices. This setting may serve as a stepping stone toward further improvements using deep learning.

The research contributions are framed by an introduction to deep learning, numerical optimization, and automatic differentiation, all of which provide the basis for the presented publications.

ZUSAMMENFASSUNG

Diese Dissertation präsentiert Forschung an der Schnittstelle von Deep Learning und Numerischer Optimierung. Die Entwicklung in diesen Bereichen hat in den letzten Jahren zu beeindruckenden Resultaten bei einer Vielzahl von Anwendungen geführt. Dies wird durch Fortschritte bei parallelen Rechnerarchitekturen und Softwarebibliotheken zum Automatischen Differenzieren katalysiert. So werden neue Algorithmen und deren Anwendung in einer zunehmenden Anzahl von Feldern ermöglicht, indem man differenzierbare Programme schreibt, d. h. parametrisierte Programme, die ableitungsbasiert optimiert werden können. Trotz dieser Erfolgsgeschichten bleiben zahlreiche Herausforderungen.

Das nicht-konvexe Optimierungsproblem zum Trainieren eines Deep Learning Systems wird üblicherweise mit heuristischer Optimierung erster Ordnung angegangen, was viel Raum für die Entwicklung von Optimierungsalgorithmen für Neuronale Netze lässt. Wir tragen ein Prinzip zur Formulierung von Deep Learning Optimierungsalgorithmen bei, welches auf einer alternativen Sichtweise auf den klassischen Backpropagation-Algorithmus basiert. Wir analysieren eine spezifische Methode im Detail, Proximal Backpropagation, die für bestimmte Komponenten des Modells proximale Schritte anstelle von Gradientenschritten anwendet. Unser Ansatz erlaubt es, ähnliche Algorithmen mit einer analogen Herangehensweise zu untersuchen.

Eine weitere Herausforderung beim Deep Learning besteht darin, dass solche Modelle oft als Black-Box Funktionen innerhalb eines größeren algorithmischen Systems verwendet werden, ohne Garantien für ihre Vorhersagen. In vielen Szenarien ist jedoch Vorwissen über die entsprechende Anwendung vorhanden, beispielsweise über die Geometrie oder die Physik des Problems. Es ist wünschenswert, diese Informationen in das Modell zu integrieren, da ansonsten der Trainingsprozess datenineffizient ist und sogar schädliche Ergebnisse erzeugen könnte. Wir entwickeln ein Modul, um solche Bedingungen für die Vorhersage des Neuronalen Netzes zu forcieren, indem wir eine differenzierbare Parametrisierung der zulässigen Menge für den speziellen Fall von homogenen linearen Ungleichungen formulieren.

Die obigen Ansätze entwickeln Deep Learning Komponenten basierend auf Ideen der Numerischen Optimierung. Umgekehrt können Deep Learning Systeme verwendet werden, um klassische Optimierungsverfahren für nicht-konvexe Probleme weiterzuentwickeln, welche die Natur- und Ingenieurwissenschaften hervorbringen. Wir demonstrieren einen solchen Ansatz im Kontext der variationsbasierten Datenassimilation. Hier verbessern wir die Optimierbarkeit eines nichtlinearen kleinste Quadrate Ansatzes, indem wir eine approximative Inverse des Beobachtungsoperators lernen. Wir verwenden diese gelernte Inverse, um die Zielfunktion zu transformieren und die Initialisierung des Optimierers zu verbessern. Diese Methode kann als Vorlage verwendet werden, um Lösungsansätze für nichtlineare inverse Probleme zu verbessern.

In einer abschließenden Veröffentlichung entwickeln wir eine kontinuierliche Relaxierung eines kombinatorischen Optimierungsproblems in der numerischen Linearen Algebra, nämlich die approximative Faktorisierung einer orthogonalen Matrix als Produkt von Givens-Matrizen. Dies kann als Ansatz für weitere Verbesserungen durch Deep Learning dienen.

Die Grundlage für die Forschungsbeiträge bilden eine Einführung in Deep Learning, Numerische Optimierung und Automatisches Differenzieren.

CONTENTS

I INTRODUCTION

This introduction develops the major concepts in numerical optimization, deep learning, and automatic differentiation, all of which form the foundation for the publications associated with this dissertation. Training a deep learning model requires to optimize a non-convex function and hence the theory of numerical optimization provides the basis for the development of novel deep learning training algorithms, such as the one presented in [18]. Such large-scale optimization algorithms are based on the efficient evaluation of derivatives, which can be provided by the framework of automatic differentiation.

We begin by introducing the setting for smooth non-convex optimization and arrive at L-BFGS as a default optimization algorithm for this setting, which we use in [17]. This lays out the roadmap for many fundamental notions, which we encounter en passant.

For the introduction to deep learning, we develop the basic components of a deep learning system and highlight the role of gradient-based optimization for its training. In particular, we outline the idea of stochastic optimization and present Adam as a default optimizer for deep learning models. We close by demonstrating the decisive role of parallel computation hardware for deep learning.

In practice, the principles of automatic differentiation are often hidden behind the API of a modern software library. Nevertheless, the fundamental principles are necessary to understand how novel algorithms can be efficiently implemented by using existing libraries. We derive the two common methods based on the chain rule of multivariate calculus, forward- and reverse-mode automatic differentiation. Finally, we outline how implicit differentiation can be used to differentiate through certain complex procedures.

1.1 Numerical Optimization

Numerical optimization aims to find the optimal point and its associated value of a mathematical function under specified constraints by means of an iterative procedure. We focus on unconstrained minimization of an *objective function* $f : \mathbb{R}^n \longrightarrow \mathbb{R}$:

$$\min_{x \in \mathbb{R}^n} f(x) \tag{1}$$

This setting permeates many problems in scientific computing and engineering. In particular, it is a core building block of machine learning systems [9].

Throughout this section, we assume that f is bounded from below to guarantee that a global minimizer exists and that f is twice continuously differentiable, i.e., that the Hessian $\nabla^2 f$ is element-wise continuous. It follows from Schwarz's theorem that the Hessian is symmetric and hence has real eigenvalues. This introduction is inspired by [40], which we recommend for further reading.

1.1.1 Stationary Points and (Non-)Convexity

We begin with the notions that describe an optimization problem and its solution. A *stationary point* x of the function f is characterized by the first-order criterion of a vanishing gradient, $\nabla f(x) = 0$. Such stationary points can be further subdivided by a second-order criterion based on the Hessian. A point x with $\nabla f(x) = 0$ is a

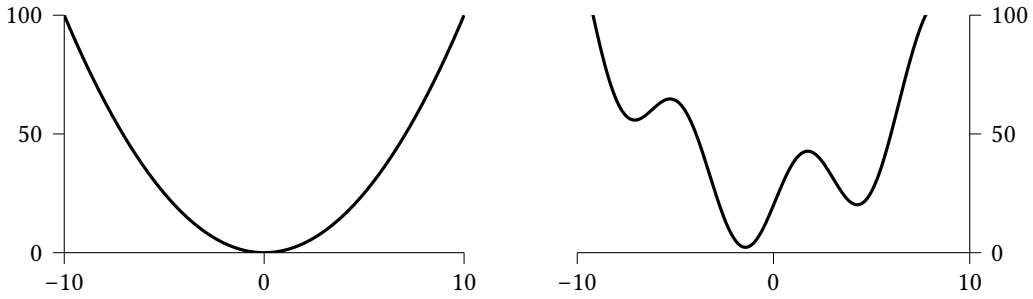
- *(local) maximum* if $\nabla^2 f(x)$ has only non-positive eigenvalues (negative semi-definite)
- *(local) minimum* if $\nabla^2 f(x)$ has only non-negative eigenvalues (positive semi-definite)
- *saddle point* if $\nabla^2 f(x)$ has positive and negative eigenvalues (indefinite)

A function f is *convex*, if and only if the Hessian is everywhere positive semi-definite, which implies that every local minimum is also a global minimum. This makes globally minimizing a convex function comparatively simple as we only have to find a stationary point. Hence, the only quality criterion of an optimizer for a convex objective function is how fast it converges to a stationary point.

Minimizing a *non-convex* function on the other hand can be difficult. An optimizer might get stuck at an unfavorable stationary point, which could be a poor local minimum that is not global or a saddle point. Consequently, there are two relevant quality criteria for the optimizer. First, the quality of the stationary point, i.e., how close the function value at this point is compared with the function value at the global minimum (which is of course not known). Secondly, how fast it converges to a stationary point. Figure 1 juxtaposes a convex and a non-convex function.

1.1.2 First- and Second-Order Optimization

In this section, we develop and analyze optimization algorithms based on first and second derivatives of the objective function. When examining Figure 1b, we have the whole picture of the objective function over the domain of interest available, and it might seem like a simple task to pick out a local or even global minimum. However, an iterative optimizer has only *local* information around the current iterate at hand to make a decision about its next step. This



(a) $f(x) = x^2$ (convex) **(b)** $f(x) = 20\sin(x) + x^2 + 20$ (non-convex)

Figure 1: Comparison of a convex and a non-convex function. While for the convex function every local minimum is a global minimum, the non-convex function has local maxima and local minima in addition to the global minimum.

situation is comparable to hiking downhill in the dark. Here, a sensible strategy is to build a local model of the surrounding landscape to make a decision about the next step.

To formalize this approach, we consider iterative *descent algorithms* of the form

$$x_{k+1} = x_k + \alpha_k p_k \ , \tag{2}$$

where α_k is a step size in a *descent direction* p_k at iteration k. A step direction p_k is a descent direction if $-p_k^T \nabla f(x_k) > 0$, i.e., if p_k is strictly within $\pm 90^\circ$ of the negative gradient. In this case, it is possible to reduce the objective function, which can be deduced from a first-order Taylor expansion around x_k,

$$f(x_k + \alpha p_k) - f(x_k) = \alpha \nabla f(x_k)^T p_k + \mathcal{O}(\alpha^2) \ . \tag{3}$$

By definition of the descent direction p_k, we have $\nabla f(x_k)^T p_k < 0$. For sufficiently small α, this first-order term dominates the quadratically vanishing error term. We can therefore always find a step size α such that $f(x_{k+1}) < f(x_k)$. An algorithm is *globally convergent* if it converges to a local minimum from any initial condition x_0. In contrast, a *locally convergent* algorithm may only converge when initialized from a restricted set of points, e.g., sufficiently close to a local minimum. The *convergence rate* of an update scheme of the form (2) describes the progress of an optimizer per update iteration toward a *local minimizer* x_*. Convergence rates are often provided as worst-case scenarios and performance may be better in practice.

Common numerical optimization algorithms build a local approximation of the objective function via a Taylor expansion. According to Taylor's theorem, a function can be expressed in a neighborhood of a point x as

$$f(x + p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T \nabla^2 f(x) p + \mathcal{O}(\|p\|^3) \ . \tag{4}$$

A *first-order* optimization algorithm uses only the linear term as an approximation to the function. We can deduce that the step direction that locally most reduces the objective function is in the direction of the negative gradient, $p = -\nabla f(x)$. The resulting optimization algorithm is *gradient descent*, which iterates

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) \ . \tag{5}$$

If the step sizes α_k are chosen appropriately, then this iteration is globally convergent to a stationary point and achieves a linear convergence rate, i.e.,

$$\|x_{k+1} - x_*\| \leq C_{GD}\|x_k - x_*\| \text{ for some constant } C_{GD} \in (0, 1) . \tag{6}$$

A *second-order* method approximates the objective function up to the quadratic term. The step direction that locally most reduces the objective function is $p = -\nabla^2 f(x)^{-1}\nabla f(x)$, which is obtained by minimizing the quadratic approximation to $f(x + p) - f(x)$ with respect to p under the assumption that $\nabla^2 f(x)$ is positive definite. The resulting optimization algorithm in this case is *Newton's algorithm* to find a root of the function's gradient,

$$x_{k+1} = x_k - \alpha_k \nabla^2 f(x_k)^{-1}\nabla f(x_k) . \tag{7}$$

Positive definiteness of the Hessian ensures that the step direction is a descent direction, since then $-p_k^T \nabla f(x_k) = \nabla f(x_k)^T \nabla^2 f(x_k)^{-1}\nabla f(x_k) > 0$. If this condition is not satisfied, modifications to the Hessian are necessary. For example, one might cut off the spectrum at a small positive number or apply more advanced modifications that yield a better conditioned matrix [15]. However, this might reduce convergence speed since the quality of the second-order approximation is reduced. Close enough to a local minimum, Newton's algorithm has a quadratic convergence rate, i.e.,

$$\|x_{k+1} - x_*\| \leq C_N\|x_k - x_*\|^2 \text{ for some constant } C_N \in (0, 1) . \tag{8}$$

Comparing the convergence rates of these two algorithms demonstrates the expected: using the more accurate second-order model improves convergence. This makes it desirable to use a second-order algorithm for optimization. However, such a convergence rate analysis only captures optimization progress as a function of iterations and not as a function of computational operations. In fact, a full Newton iteration (7) may be computation- and memory-expensive. First, one has to calculate second derivatives of the objective function. Secondly, inverting the Hessian matrix (or, numerically favorable, solving the linear system) for large problems becomes infeasible as this generally requires $\mathcal{O}(n^3)$ operations in n dimensions. Thirdly, explicitly constructing the Hessian matrix for high-dimensional problems quickly requires too much memory.

1.1.3 Line Search: Globalization of Convergence

If the step size is not chosen appropriately, both gradient descent and Newton's method may not converge because they overshoot the local minimum, or they may converge slowly because step sizes are too small. One approach to yield global convergence of both algorithms for most functions is via a *line search* to find a suitable step size α_k along the previously computed search direction p_k. We want to mention that counterexamples can be constructed for gradient descent [8] and Newton's method [29], which, however, are negligible in practice.

A line search analyzes the one-dimensional function

$$\phi(\alpha) := f(x_k + \alpha p_k) . \tag{9}$$

Exact line search for the global minimizer of ϕ is generally very costly, and instead an inexact line search is typically performed. Just requiring $f(x_k + \alpha p_k) < f(x_k)$ is not enough to guarantee

convergence to a stationary point. To state an explicit counterexample, we take $f(x) = x^2 - 1$ and follow a descent sequence of iterates $x_k = \sqrt{(k+1)/k}$ for $k \in \{1, 2, \dots\}$. Then, $f(x_k) = 1/k$ converges to zero, but the function's minimum is $f(0) = -1$. Hence, we need to guarantee a sufficient decrease per iteration. To formulate appropriate conditions, we begin with the first-order Taylor expansion (3). If we denote the remainder of quadratic order by $\mathcal{R}(\alpha) \in \mathcal{O}(\alpha^2)$, then for $c_1 \in (0, 1)$ we can find an α such that $\mathcal{R}(\alpha)/\alpha \leq (c_1 - 1)\nabla f(x_k)^T p_k$ (recall that $\nabla f(x_k)^T p_k < 0$ by definition of a descent direction). This leads to a bound on the decrease of the objective function. Requiring that the step size α satisfies this bound for a given constant $c_1 \in (0, 1)$ leads to the *Armijo condition*,

$$f(x_k + \alpha p_k) - f(x_k) \leq c_1 \alpha \nabla f(x_k)^T p_k, \quad c_1 \in (0, 1) \,. \tag{10}$$

This condition by itself does not suffice, since it is satisfied by arbitrarily small α. To rule out such small step sizes, we consider the change in f along the search direction when varying α. This slope is given by $\phi'(\alpha) = \nabla f(x_k + \alpha p_k)^T p_k$, with $\phi'(0) < 0$, since p_k is a descent direction. We want to follow the descent direction at least until f is less decreasing by a factor c_2, i.e., $\phi'(\alpha) \geq c_2 \phi'(0)$. This condition is called the *curvature condition* as it describes a change in slope of the function ϕ along the search direction. In terms of the function f it may be stated as

$$\nabla f(x_k + \alpha p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k, \quad c_2 \in (c_1, 1) \,. \tag{11}$$

Together, conditions (10) and (11) are referred to as the *Wolfe conditions*. The restriction $0 < c_1 < c_2 < 1$ guarantees that both conditions can be satisfied at the same time.

A simpler line search procedure that guarantees (10) and avoids small step sizes may be formulated as follows. A step size is tested for condition (10) and if it is not satisfied, the step size is multiplied by a factor $\tau \in (0, 1)$. The algorithm is commonly started with a step size $\alpha = 1$ and thus unreasonably small step sizes are ruled out. Since it starts with a large step, which is subsequently reduced, the approach is called *backtracking line search*. Backtracking line search suffices to render gradient descent and Newton's method (with possible Hessian modification) globally convergent, apart from practically irrelevant counterexamples [8, 29]. This is a common procedure for Newton's method, for which the step size $\alpha = 1$ is eventually always accepted close enough to a local minimum. On the other hand, gradient descent is rarely implemented with a line search algorithm in practice, since the additional function evaluations render this procedure uneconomical. Instead, some heuristic schedule for annealing the step size is typically chosen.

Algorithm 1 Backtracking line search

Require: $c_1 \in (0, 1)$, $\tau \in (0, 1)$, $\alpha > 0$; typical values are $c_1 = 10^{-4}$, $\tau = 1/2$, $\alpha = 1$
  while $f(x_k + \alpha p_k) > f(x_k) + c_1 \alpha \nabla f(x_k)^T p_k$ **do**
    $\alpha \leftarrow \tau \alpha$
  end while
  return α

1.1.4 Quasi-Newton Methods

Quasi-Newton methods are designed to retain some effectiveness of a second-order method while ameliorating the issues of Newton's method: the cost of computing second derivatives, a Hessian that might not be positive-definite, and the cost of inverting the Hessian. A quasi-Newton optimizer works with an approximate second-order model of the form

$$m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p \ , \tag{12}$$

where B_k is a symmetric positive-definite matrix, which is updated at every iteration. The matrix B_k is modeled to approximate the Hessian and hence the quadratic model (12) is an approximation to the second-order Taylor expansion (4). The model (12) is convex and its minimizer can be explicitly computed as

$$p_k = -B_k^{-1} \nabla f(x_k) \ . \tag{13}$$

The matrix B_k is now determined by requiring some properties of the model m_k and different requirements would lead to different quasi-Newton methods. Here, we outline the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update. Besides symmetry and positive-definiteness of B_k, we require that the model's gradient aligns with the objective function's gradient at two consecutive iterations. Additionally, among all those matrices which satisfy these conditions, we want to obtain the matrix which is closest to the previous iterate. Finally, instead of updating the matrix B_k, we directly update its inverse matrix $B_k^{\text{inv}} := B_k^{-1}$. Altogether, this yields the update rule:

$$\begin{cases} B_k^{\text{inv}} &= V_k^T B_{k-1}^{\text{inv}} V_k + \rho_k s_k s_k^T \\ V_k &= \mathbb{I} - \rho_k y_k s_k^T \\ \rho_k &= 1/(s_k^T y_k) \\ s_k &= x_k - x_{k-1} \\ y_k &= \nabla f(x_k) - \nabla f(x_{k-1}) \end{cases} \tag{14}$$

We refer to [40] for a detailed derivation of this update rule. Iteration (14) can be implemented with matrix-vector and outer products. It follows that the computational complexity of an update in n dimensions is in $\mathcal{O}(n^2)$. We can see that the algorithm uses only first-order information to build the quadratic model and does not rely on second derivatives. Nevertheless, the optimizer has a superlinear convergence rate [40]. BFGS works well in practice, but one decisive problem remains for large-scale optimization: the matrix B^{inv} in (14) is explicitly retained in memory, which becomes quickly prohibitive. In contrast, memory-efficient algorithms employ a *matrix-free* iteration step, i.e., they compute the matrix-vector product (13) without explicitly constructing the iteration matrix in memory. Limited-Memory-BFGS (L-BFGS) does exactly this kind of update while sacrificing accuracy of the quadratic model. The optimizer retains the m most recent vectors (s_i, y_i) and constructs an approximation $\hat{B}_k^{\text{inv}}$ to B_k^{inv} by iteratively applying (14) to a provided initial matrix $\hat{B}_k^{\text{inv},0}$. Crucially, this matrix is never explicitly constructed, but the result of the matrix-vector product (13) can directly be computed. We refer to [40] for implementation details, but mention that a $\hat{B}_k^{\text{inv}}$ update can be efficiently implemented at an $\mathcal{O}(mn)$ computational cost. L-BFGS achieves only a linear worst-case convergence rate, but typically works better in practice [30].

BFGS and L-BFGS require a line search along the direction (13) that satisfies the Wolfe conditions (Section 1.1.3). Backtracking line search is not sufficient, as the curvature condition is required to guarantee that B_k^{inv} remains positive definite throughout the iterations.

L-BFGS is a good generic choice for a large-scale, smooth, and non-convex optimization problem. We use the optimizer in [17] to solve an optimization problem in the context of variational data assimilation.

1.1.5 Numerical Performance

We demonstrate the concepts outlined here by a numerical comparison of the introduced optimizers. To this end, we optimized the (averaged) n-dimensional Rosenbrock function [43], a classical test problem for non-convex optimization,

$$f(z) = \frac{1}{n-1} \sum_{i=1}^{n-1} 100(z_{i+1} - z_i^2)^2 + (1 - z_i)^2 \,, \quad z = (z_1, \dots, z_n) \in \mathbb{R}^n \,. \tag{15}$$

The average here implies that the function's scale is independent of the dimension. We use $n = 1000$, for which the computational complexity for each iteration of the algorithms becomes important. It is a difficult test problem, since the minimum lies in a narrow valley, where there exist directions with very small and directions with very large curvature. This can be described by the absolute value ratio of the Hessian's largest and smallest eigenvalue, the *condition number* at a point $z \in \mathbb{R}^n$. The condition number at the minimum is about 3600. To effectively deal with this situation, an optimizer needs to take into account curvature information. Below we describe the implementation details for all algorithms, as these highlight some important ideas for their efficient implementation.

Gradient Descent

We implemented the update (5) with a backtracking line search as outlined in Algorithm 1. The line search yields a convergent algorithm and effective progress per iteration. However, the additional function evaluations incur an extra cost. Therefore, more economical (but less effective) heuristics for the step size selection, such as an annealing schedule, are often chosen in practice. We specified an initial step size $\alpha = 10^{-1}$ to the line search algorithm to avoid wasting too many function evaluations on the line search and chose the other line search parameters as the defaults described in Algorithm 1.

Newton's Method

As for gradient descent, we implemented the update (7) with a backtracking line search (Algorithm 1) with initial step size $\alpha = 1$ and default parameters. To ensure a descent direction, we modified the possibly non-positive definite Hessian by cutting off the eigenvalues at 10^{-6}.

BFGS

We implemented the update (14) with a Wolfe line search that guarantees that the matrix B_k^{inv} remains positive definite throughout the iterations. We initialized the update scheme with a modified Hessian whose eigenvalues are cut off at 10^{-3}.

L-BFGS

As for BFGS, we implemented the L-BFGS update with a Wolfe line search. We retained 10 vectors (s_i, y_i) to approximate the BFGS matrix. The initial matrix $\hat{B}_k^{\text{inv},0}$ is crucial for good performance of the optimizer and hence should be explicitly mentioned here. Following the recommendation in [40], we initialized as

$$\hat{B}_k^{\text{inv},0} = \frac{s_k^T y_k}{y_k^T y_k} \mathbb{I} \,. \tag{16}$$

The intention behind this scaling is to have $\hat{B}_k^{\text{inv},0}$ on the same scale as the inverse Hessian.

Results

We initialized all methods with a random vector for which entries were sampled from a standard normal distribution and ensured that all optimizers converged to the same global minimizer, which is the vector of all ones, $z_i = 1 \forall i \in \{1, \dots, n\}$. The optimization progress as a function of iterations and as a function of time is shown in Figure 2 on a log scale. The top row shows long term performance of the algorithms while the bottom row demonstrates their initial behavior.

Observing the optimization progress as a function of iterations (top left), we can indeed see that the theoretical linear convergence rate (6) for gradient descent and the quadratic convergence rate (8) for Newton's method translate into much more effective optimization for the latter. Gradient descent requires many more iterations to converge, which is cut off in the plot to retain a separation of the other methods. Progress for the initial iterations of Newton's algorithm is slow (bottom left). In this regime, the Hessian is not positive definite and our second-order model with Hessian modification has limited validity. Here, only small step sizes are accepted by the backtracking line search. In fact, we can observe that for this specific problem and initial condition the gradient direction is a better descent direction than the modified Newton direction. BFGS and L-BFGS both eventually converge super-linearly. Recalling their initialization, BFGS starts with a modified Newton update, whereas L-BFGS starts with a (scaled) gradient descent update and only slowly accumulates curvature information. This explains that L-BFGS is initially closer to gradient descent, while BFGS is initially closer to Newton's method.

As a function of time (top right), we can see that the quasi-Newton methods demonstrate the best overall performance. Newton's algorithm requires most time per iteration as second derivatives are computed and the Hessian needs to be inverted. In addition, the line search is costly with a poor second-order model, since many backtracking steps are required to then only yield a small step size. Eventually, the step size of $\alpha = 1$ is directly accepted without additional line search steps. It is this regime of optimization where Newton's method exhibits a quadratic convergence rate. BFGS and L-BFGS have a favorable computational complexity per iteration. In particular the latter quickly yields fast optimization progress (bottom right).

Overall, the results support the statement that L-BFGS is the method of choice in a large-scale setting as it exploits curvature information at a comparatively low computational cost and the matrix-free implementation is memory-efficient. We employ L-BFGS to solve a difficult non-convex optimization problem in [17].

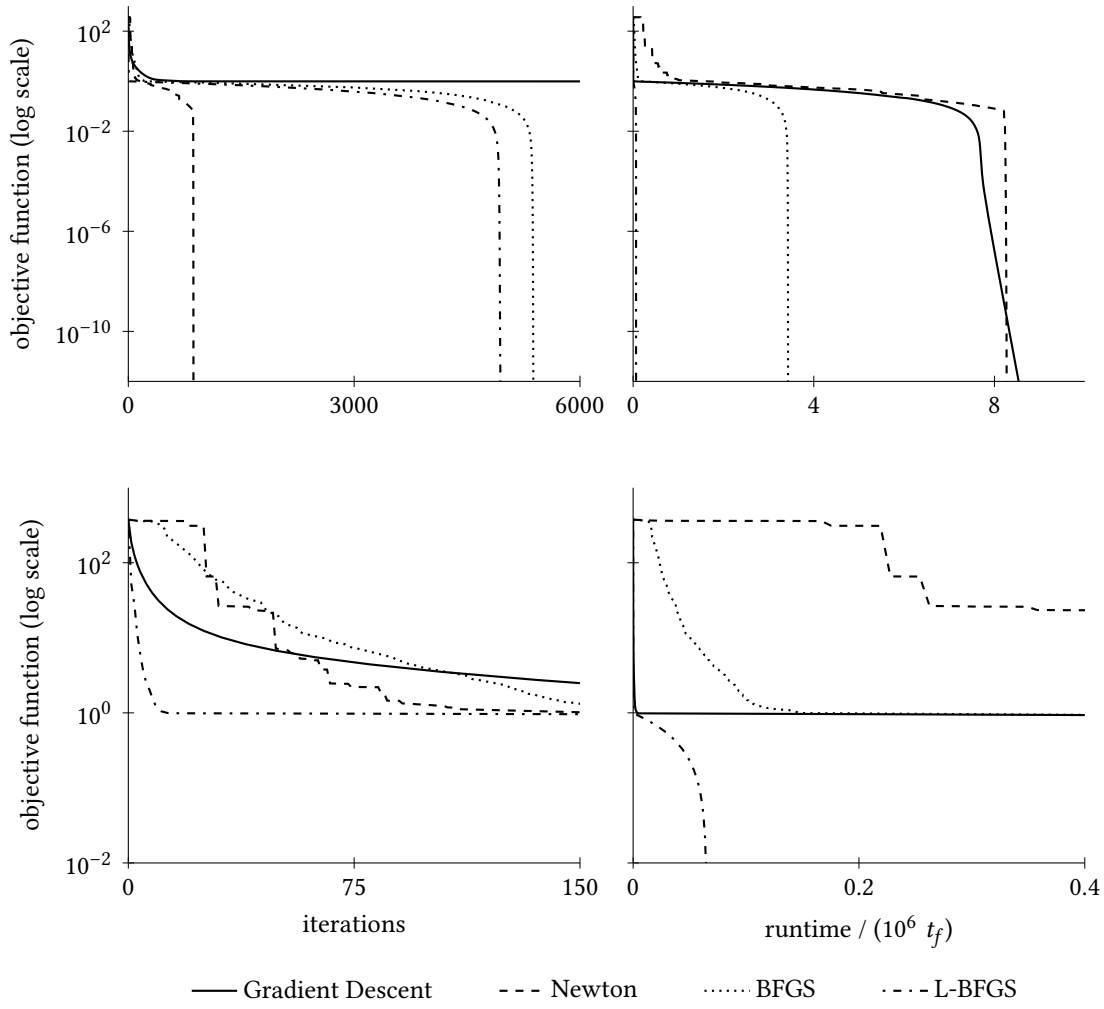


Figure 2: Comparison of different optimization algorithms for minimizing the averaged Rosenbrock function (15) in $n = 1000$ dimensions. Optimization progress is shown as a function of iterations (left) and as a function of runtime (right). The top and bottom row show the algorithms' long and short term behavior, respectively. Runtime is measured in units of the average objective function execution time, t_f. Note that the iteration and runtime plots do *not* show the same optimization time on their x-axis. The iteration plots show a shorter optimization time to highlight a separation of the various algorithms.

1.2 Deep Learning

In this section, we introduce the major concepts for building a deep learning model and define the language to describe them. An overview of various types of deep learning models may be found in [34] and the rich history of the field is summarized in [46]. Deep learning as a field is evolving at a fast pace. Here, we present how the term is currently used in research and practice, while we are aware that some concepts will likely be superseded a few years from now.

A *neural network* represents a parameterized function (or *model*) with the defining property that this function is composed of a sequence of elementary components, termed *layers*. For the purpose of this thesis, neural networks are used for *machine learning*: the parameters are optimized such that the model interpolates an unknown function based on samples of this function, called the *training data*. Finding a function by optimizing the parameters with respect to a suitably designed loss is called the *training phase* of the learning algorithm. After training, the model can be used on new samples, which is called the *inference phase* of the learning algorithm. The goal is to optimize performance for these new samples and a model is said to *generalize* well, if it yields good performance for these *test data*.

The *capacity* of the model describes the complexity of the underlying function that can be expressed. If we allowed all possible functions as interpolants, infinitely many of them would perfectly fit the training data. However, these may produce an arbitrarily large error on test data points. Hence, we need to restrict the capacity of the model to allow only certain types of interpolants, at best by incorporating prior knowledge about the underlying function to be approximated. This process is called *regularization*. *Explicit regularization* modifies the loss function, e.g., by adding a term that favors certain solutions. An example is an L_2-norm penalty for the neural network parameters, so-called weight-decay, which favors models with smaller parameters. In contrast, *implicit regularization* describes the notion that various aspects of the architecture design and training process may implicitly restrict the capacity of the model. Regularization may be used to impose prior knowledge about the task, a *bias*, onto the deep learning model. Among all candidate functions that are a good fit to the data, the optimization process is then biased toward those with desired properties. Altogether, machine learning with neural networks consists of optimization with suitable regularization.

In recent years, neural network models with the following properties have emerged:

P1 They contain many layers, a property referred to as *deep*.

P2 *Stochastic first-order optimization* is used to optimize the model parameters.

P3 They are *over-parameterized*, i.e., the models contain more parameters than there are training data points.

This setting in combination with a set of heuristics for effective training of such models is commonly referred to as *deep learning*. A trend toward such models can be seen from the history of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [45]. Table 1 summarizes the winning models for the image classification task. Starting in 2012, deep learning models have dominated the challenge and have continued to reduce the classification error. All of these models are significantly over-parameterized compared with the 1.2 million training images. Furthermore, we can deduce a tendency toward deeper models.

YEAR	MODEL	L	$P/10^6$	P/N
2012	AlexNet [32]	9	60	50
2013	ZFNet [51]	8	63	53
2014	GoogLeNet [48]	22	7	6
2015	ResNet-152 [26]	152	60	50
2016*	ResNeXt-101 [50]	101	83	69
2017	SENet-154 [28]	154	115	96

Table 1: ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winning models for image classification between 2012 and 2017. The training data set contains 1.2 million images (N) with the task of classifying these into 1000 classes. Reported parameter numbers (P) and parameterization factors (P/N) are rounded. L denotes the number of layers. All models are over-parmeterized with a tendency toward deeper models. *: The 2016 winner was the Trimps-Soushen model. However, the authors did not publish their approach and hence we report here the closely following runner-up model.

From a classical optimization perspective, one cannot expect good performance in this setting at first sight. **P1** suggests that the function to be optimized is non-convex with possibly many poor local optima and saddle points. **P2** means that this non-convex function is optimized with a first-order optimizer, which may get stuck at an unfavorable stationary point and for which optimization progress may be slow. **P3** implies that it is possible to perfectly fit the training data and hence optimizing to a local optimum on this training data might result in unfavorable generalization.

Nevertheless, deep learning has yielded impressive results in practice. Even though there is currently no good theory to explain this success, recent research attempts to shed light on this discrepancy between classical theory and empirical results [52, 5, 2, 41, 36]. In the following, we summarize the common steps of designing a deep learning model.

1.2.1 Formalizing the Deep Learning Problem

Here, we establish a notation and formalize the major types of machine learning. A deep learning algorithm interpolates an unknown function $f : \mathcal{X} \to \mathcal{Y}$ with a θ-parameterized model f_θ. Information about f is provided by samples $(x, y) \sim \mathbb{D}$ from this function according to a data distribution $\mathbb{D}$. The learning task is to solve the following optimization problem:

$$\min_{\theta \in \mathbb{R}^p} \mathbb{E}_{(x,y)\sim\mathbb{D}}[\mathcal{L}(f_\theta; x, y)] \quad , \tag{17}$$

where $\mathcal{L}(f_\theta; x, y)$ is a loss function that is designed for a specific setting (Section 1.2.3).

In a *classification* task, we predict a label $\hat{y} \in \mathcal{Y}$ for a data point $x \in \mathcal{X}$. An example of classification is to predict a label for the content of an image. Here, the set $\mathcal{Y}$ contains all possible labels and $\mathcal{X} \subset \mathbb{R}^{a \cdot b}$ represents all possible images on an $a \times b$ pixel grid.

In a *regression* task, $\mathcal{Y}$ is a continuous space. An example is an auto-encoder, where the neural network architecture should predict the input itself, while there exists an intermediate layer of smaller dimension than the input. In this setting, the neural network is tasked to find an average compression of the data.

For the purpose of this introduction, we focus on *supervised learning*, for which one is provided with both x and y and the task is to predict y from x. Both examples above fall into

this category. For completeness, we mention the concept of *unsupervised learning*, where no direct correspondence y is provided for x. Instead, the machine learning algorithm exploits a relationship between different input data points. An example is a clustering algorithm, which assigns a common label to data points that are close with respect to some metric.

1.2.2 Compiling a Deep Learning Architecture

The *architecture* of a deep learning model describes the composition of parameterized elements. Commonly, a component of the architecture consists of a linear (or, more precisely, affine) layer, an element-wise nonlinear function, and an aggregation operation. Here, we describe common choices for each of these elements. More formally, we want to represent a θ-parameterized function $f_\theta : \mathcal{X} \to \mathcal{Y}$ as a sequence of layers, $f_\theta = f^1_{\theta_1} \circ \cdots \circ f^L_{\theta_L}$, where each layer represents a function $f^l_{\theta_l}$ with its parameters θ_l. We denote intermediate values of the l-th layer by $a^l = f^l_{\theta_l}(a^{l-1})$, the layer's *activations*.

Linear layers are of the form $a^{l+1} = Wa^l + b$ where W is a *weight*-matrix and b a *bias*-vector of suitable size. The matrix W may have no structure (fully-connected layer) or may have additional structure. The most common structured linear layer is a convolutional layer. Such a layer has the signature $\mathrm{Conv}(c_{\text{in}}, c_{\text{out}}, s_f)$ and applies a different convolutional filter of size $s_f \times s_f$ to each combination of the input channels c_{in} and output channels c_{out}. These filters are the trainable parameters of the layer. Convolution is a translation equivariant operation, i.e., if $\mathcal{T}$ is a translation operator, then $\mathrm{Conv}(\mathcal{T}(x)) = \mathcal{T}(\mathrm{Conv}(x))$. This property matches well with images, where a differently positioned object should induce equivalently shifted activations. As a consequence, convolutional architectures are the most common building block of deep learning models for computer vision. The choice of nonlinearity is crucial for optimization performance. If smoothness of the parameterized function is not a requirement, then the rectified linear unit (ReLU) [32] is often a good choice. Otherwise, a smooth approximation to this function can be chosen, such as the exponential linear unit (ELU) [13] or the sigmoid-weighted linear unit (SiLU) [14]. These activation functions are visualized in Figure 3. The most common aggregation operation is some form of normalization of the neural network activations. A prominent example is batch normalization, which normalizes the activations to zero mean and unit variance across a mini-batch (see Section 1.2.4 for mini-batch optimization).

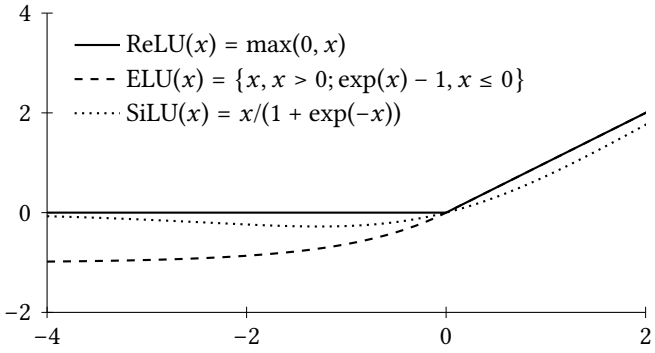


Figure 3: Common choices for nonlinear activation functions. ReLU is an effective choice if smoothness is not required. Otherwise, ELU or SiLU can be chosen as smooth approximations to ReLU.

A modern deep learning architecture is composed of many of such smaller modules, which define the class of possible functions that can be represented. Table 1 lists the number of layers for successful computer vision models with the tendency toward more layers, i.e., deeper architectures.

1.2.3 Designing a Loss Function

In the following, we derive the typical loss functions for a classification task and a regression task. As outlined in Section 1.2.1, in a classification task the learning target is discrete, whereas in a regression task the learning target is continuous.

Classification

To assign a class $\hat{y} \in \mathcal{Y}$ to an input $x \in \mathcal{X}$ for a classification problem, we predict a discrete distribution $p_\theta(\hat{y}|x)$, i.e., we assign a probability to each class $\hat{y}$ that the input x belongs to this class. The neural network maps an input x to a vector $f_\theta(x) \in \mathbb{R}^K$, where K is the number of classes and every entry of this vector assigns a *score* to the respective class. One way of turning this score vector into a discrete probability distribution via a differentiable operation is by applying a softmax operation,

$$p_\theta(\hat{y}|x) = \operatorname{softmax}(f_\theta(x))_{k(\hat{y})} = \frac{\exp([f_\theta(x)]_{k(\hat{y})})}{\sum_{y' \in \mathcal{Y}} \exp([f_\theta(x)]_{k(y')})} \quad \forall \hat{y} \in \mathcal{Y} \quad . \tag{18}$$

Here, $k(\hat{y})$ is the index associated with the class $\hat{y}$. This distribution is then compared with the ground truth distribution $q(\hat{y}|y) = \mathbb{I}_{\{\hat{y}=y\}}$, which is equal to one at the correct label and zero otherwise. The parameters θ are optimized to move the distribution p_θ closer to q by minimizing the Kullback-Leibler divergence from q to p_θ,

$$\begin{aligned} \mathrm{KL}(q||p_\theta) &= -\sum_{\hat{y} \in \mathcal{Y}} q(\hat{y}|y) \log\left(\frac{p_\theta(\hat{y}|x)}{q(\hat{y}|y)}\right) && (19) \\ &= \underbrace{-\sum_{\hat{y} \in \mathcal{Y}} q(\hat{y}|y) \log(p_\theta(\hat{y}|x))}_{\text{cross entropy } H(q,p_\theta)} + \underbrace{\sum_{\hat{y} \in \mathcal{Y}} q(\hat{y}|y) \log(q(\hat{y}|y))}_{\text{independent of } \theta} && (20) \end{aligned}$$

Hence, minimizing the first term, called the *cross entropy* $H(q, p_\theta)$ between distributions q and p_θ, forces p_θ to approximate q:

$$\mathcal{L}(\theta; x, y) := -\sum_{\hat{y} \in \mathcal{Y}} q(\hat{y}|y) \log(p_\theta(\hat{y}|x)) = -\log(p_\theta(y|x)) \, . \tag{21}$$

Cross-entropy has the interesting property that it compensates the exponential scaling of the softmax (18) in the following sense. If $p(s) = \operatorname{softmax}(s)$ is the discrete probability distribution vector obtained as a softmax over a score vector s, then $\nabla_s H(q, p(s)) = p(s) - q$ for any discrete probability distribution vector q. This implies that gradient descent on this loss induces a change in the scores proportional to the discrepancy of the probability vectors.

Regression

To arrive at a loss function for the regression problem, we assume that the unknown continuous function $f : \mathcal{X} \longrightarrow \mathcal{Y}$ to be interpolated can only be accessed via *noisy* samples. This assumption can be modeled by $y = f(x) + \varepsilon$, where ε is a random variable with zero mean. This means that for provided (x, y) there is a probability $p(\hat{y}(\theta); x, y)$, centered around a mean y, that a predicted vector $\hat{y}(\theta) = f_\theta(x)$ is the true underlying value, $p(\hat{y}(\theta); x, y) = \mathbb{P}[\hat{y}(\theta) = f(x)]$. Different assumptions for the noise distribution ε lead to different loss functions. Here, we follow the most common assumption that ε is normally distribution with unit variance, $\varepsilon \sim \mathcal{N}(0, \mathbb{I})$. We want to find the parameters θ_* that yield the most likely prediction $\hat{y}(\theta)$, an approach appropriately termed *maximum likelihood*:

$$\theta_* = \underset{\theta \in \mathbb{R}^p}{\operatorname{argmax}}\, p(\hat{y}(\theta); x, y) \; . \tag{22}$$

We may equivalently minimize the negative log-likelihood, which yields a linear least-squares loss,

$$\mathcal{L}(\theta; x, y) := \frac{1}{2} \| f_\theta(x) - y \|_2^2 \; . \tag{23}$$

Analogous to classification with a cross-entropy loss, this formulation also exhibits the property that the gradient is linear in the discrepancy between the prediction and the fitting target, $\nabla_{f_\theta(x)} \mathcal{L} = f_\theta(x) - y$.

1.2.4 Training a Deep Learning Model

In this section, we outline the training procedure of a deep learning model, i.e., the phase of optimizing the parameters of the model for a given data distribution, and we present Adam as a default choice to train a deep learning model. In practice, we cannot directly solve (17), since we only have access to a finite data set $\mathcal{D}$ of samples from $\mathbb{D}$. Instead, we approximate the expected value with the empirical mean,

$$\mathcal{L}(\theta; \mathcal{D}) := \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \mathcal{L}(f_\theta; x, y) \; . \tag{24}$$

However, we are not interested in the loss on the available samples $\mathcal{D}$, but only in the generalization to new samples $(x, y) \sim \mathbb{D}$. This is the central philosophical difference between a machine learning task and a pure optimization task. To model this aspect, we divide the available samples up into a training, a validation, and a test data set, $\mathcal{D} = \mathcal{D}_{\text{train}} \bigcup \mathcal{D}_{\text{val}} \bigcup \mathcal{D}_{\text{test}}$. We still take minimization steps on the training loss $\mathcal{L}(\theta; \mathcal{D}_{\text{train}})$, but not to global optimality. Instead, we also monitor the validation loss $\mathcal{L}(\theta; \mathcal{D}_{\text{val}})$. We continue minimizing $\mathcal{L}(\theta; \mathcal{D}_{\text{train}})$ until $\mathcal{L}(\theta; \mathcal{D}_{\text{val}})$ starts to increase. At this point there is no further benefit to optimizing the training loss since any further minimization only overfits to the training data. This principle is called *early stopping*. The authors of [7] suggest that early stopping has the same effect as explicit L_2-norm regularization of the parameters θ. To simulate the final model performance on unseen data, the model is evaluated on the test set $\mathcal{D}_{\text{test}}$. To avoid that the model is tuned toward these test data, it is explicitly separated from the model development process.

A first-order optimizer is commonly used for training. To compute gradients of the loss with respect to the model parameters, reverse-mode automatic differentiation is used (Section 1.3). Since an error in the final loss is propagated in a backward pass of the model, the procedure is referred to as *backpropagation*.

Deep learning models are trained with huge amounts of data, which cannot fit into memory. Consequently, the training loss cannot be evaluated for all samples at the same time. Therefore, the loss is evaluated only for a smaller *mini-batch* $\mathcal{B} \subset \mathcal{D}_{\text{train}}$ of samples at a time. A closer look at the optimization objective function (24) reveals that even though we optimize with respect to the parameters θ, the objective function is determined by the specific samples in a single batch. Hence, evaluating the objective function batch-wise, implies that minimization steps are taken on a different function each time. If these functions were not related, then there would not be any hope for a meaningful optimization algorithm. However, the mini-batches are sampled independently and uniformly with replacement from all available samples. We denote this sampling distribution by $\mathcal{U}$. The single sample gradient $\nabla\mathcal{L}(\theta; x, y)$ then becomes a random variable, and we want to construct an estimator for the full-batch gradient based on sampled mini-batch gradients. This concept is called *stochastic optimization*. We exploit linearity of the expected value and the property that $\text{Var}[Z_1 + Z_2] = \text{Var}[Z_1] + \text{Var}[Z_2]$ for uncorrelated random variables Z_1 and Z_2. It follows that the mini-batch gradient is an unbiased estimator and that its variance is inversely proportional to the batch size:

$$\mathbb{E}_{\mathcal{B}\sim\mathcal{U}(\mathcal{D}_{\text{train}})}[\nabla\mathcal{L}(\theta; \mathcal{B})] = \nabla\mathcal{L}(\theta; \mathcal{D}_{\text{train}}) \tag{25}$$

$$\text{Var}_{\mathcal{B}\sim\mathcal{U}(\mathcal{D}_{\text{train}})}[\nabla\mathcal{L}(\theta; \mathcal{B})] = \frac{1}{|\mathcal{B}|}\text{Var}_{(x,y)\sim\mathcal{U}(\mathcal{D}_{\text{train}})}[\nabla\mathcal{L}(\theta; x, y)] \tag{26}$$

An *epoch* of training on a sequence of batches is completed once every sample is expected to have been used once. If there are N samples in the data set and we sample batches of size B, then this is the case after N/B batches. The batch size is typically chosen according to two criteria. First, the computational hardware determines the available amount of memory and parallelism. The batch size should be chosen to exploit the parallel architecture (Section 1.2.5). Secondly, the size of the mini-batch determines the amount of noise in the gradient computation as becomes evident through (26). On the one hand, this deteriorates the information provided about the full-batch gradient. On the other hand, noisy mini-batch gradients have a regularizing effect. Intuitively, the noise introduced to the gradient computation prevents the algorithm from settling in a local minimum or saddle point of the training loss. The trade-off here is between convergence speed and regularization strength. While large batches yield a faster converging optimization algorithm, smaller mini-batches avoid getting stuck at unfavorable stationary points.

The template algorithm for stochastic first-order optimization is stochastic gradient descent (SGD). For a sequence of batches $\mathcal{B}_k \subset \mathcal{D}_{\text{train}}$ with associated gradients $g_k = \nabla\mathcal{L}(\theta_k; \mathcal{B}_k)$, the update equation for parameters θ_k with step size α_k is

$$\theta_{k+1} = \theta_k - \alpha_k g_k \ . \tag{27}$$

SGD has several disadvantages. First, a step size sequence $\{\alpha_k\}_{k=1}^{\infty}$ has to be chosen. If the step sizes are too large, then the optimization progress may be initially fast, but may prevent the algorithm from converging. On the other hand, if the step sizes are too small, this may preclude any reasonable optimization progress to begin with. Heuristics, such as a schedule for annealing

the step size, are typically employed. Secondly, the gradients' variance may be very high, which significantly slows down convergence.

Various optimizers have been proposed in recent years to improve over SGD for neural network training. The currently most broadly used optimizer is Adam [31], which stands for adaptive moment estimation. Adam builds an exponential moving average of the gradients' mean m_k and (uncentered) variance v_k at each iteration k. The update rule is:

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k \tag{28}$$

$$v_k = \beta_2 v_{k-1} + (1 - \beta_2) g_k^2 \tag{29}$$

$$\hat{m}_k = m_k / (1 - \beta_1^k) \tag{30}$$

$$\hat{v}_k = v_k / (1 - \beta_2^k) \tag{31}$$

$$\theta_{k+1} = \theta_k - \alpha \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \varepsilon} \tag{32}$$

Here, the hyperparameters $\beta_1, \beta_2 \in [0, 1)$ trade-off the current mini-batch gradient with the current moment estimate, and the hyperparameter α is a fixed scale of the step size. In step (32), a small number $\varepsilon > 0$ is added to avoid division by zero. The optimizer initializes m_0, v_0 to all zeros. It is desirable in stochastic optimization to construct an unbiased estimator of the full-batch objective function as in (25) for SGD. Let $\mathcal{U}$ be the i.i.d. uniform mini-batch sampling distribution, then the element-wise expected value can be explicitly derived as $\mathbb{E}_{g_k \sim \mathcal{U}}[m_k] = (1 - \beta_1^k)\mathbb{E}_{g_k \sim \mathcal{U}}[g_k]$ (and analogously for v_k). It follows that the steps (30) and (31) correct for the bias introduced by zero-initialized iterations (28) and (29). The parameter update (32) displays that the step taken in the estimated gradient direction is modified by a factor $1/\sqrt{\hat{v}_k}$. Hence, a smaller step is taken when the uncentered gradient variance is large in comparison to the gradient mean. As a consequence, the algorithm adapts its step size in the course of optimization. Crucially, the algorithm avoids tedious hyperparameter tuning, since the default choices $\alpha = 10^{-3}, \beta_1 = 0.9, \beta_2 = 0.999$ often already work well in practice for the commonly used neural network architectures and data sets. These constants imply that the current batch only marginally contributes to the current moment estimates. A noteworthy property is that the update parameters may be chosen independent of the overall scale of the objective function, since a scalar multiple of the objective function cancels in the update (32).

1.2.5 Hardware Acceleration

A computer's central processing unit (CPU) is a generic computational device to support a broad range of computational tasks. *Hardware acceleration* describes the notion of task-specific computation hardware, which might not be as broadly applicable, but on the other hand is very efficient at a particular task. A strong catalyst for recent success stories in deep learning and automatic differentiation codes is the exploitation of parallel computation hardware, notably the graphical processing unit (GPU). While a CPU supports a wide range of instructions on a few cores at a relatively high clock speed, a GPU only supports a few instructions for floating-point arithmetic on many cores at a lower clock speed. This renders GPUs extremely efficient at one particular task: floating-point matrix multiplication. This becomes obvious by explicitly writing down the matrix-multiplication of two matrices $A, B \in \mathbb{R}^{n\times n}$,

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \,. \tag{33}$$

Computing an element of the resulting matrix requires floating-point multiplication and summation. Additionally, all elements are independent and can therefore be computed in parallel. The GPU may efficiently execute these computations in parallel on its many cores.

To train a deep learning model, we need to compute many such operations. The application of a linear layer and an element-wise nonlinearity may both be executed in parallel. As we detail in Section 1.3, computing derivatives of such a model equally requires parallelizable matrix operations. Therefore, GPUs (and similar parallel processing architectures) are especially apt at training deep learning models.

To compare CPU- and GPU-efficiency, we benchmarked both hardware types for the square matrix multiplication (33). The results are shown in Figure 4. For the specific hardware tested, we deduce a performance gain of GPU over CPU of a factor of $10 - 70$. A factor of 50, for example, would imply that a computation runs in 10 minutes as opposed to running overnight. This difference becomes particular striking when taking into account that a training algorithm for a deep learning model is not just transferred from a drawing board to code, but instead many iterations of trial and error are typically necessary. Consequently, such drastic hardware acceleration not only reduces execution time for the final model, but also significantly reduces development time.

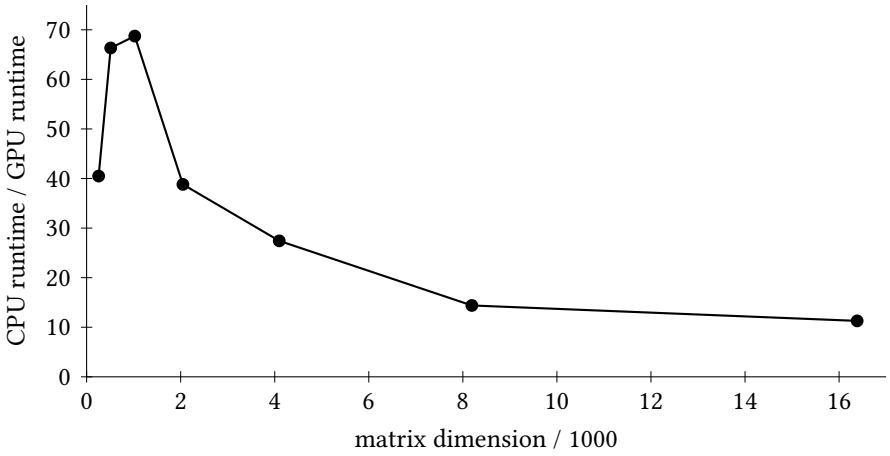


Figure 4: Comparison of CPU vs. GPU performance for the multiplication of two square matrices. We implemented the matrix multiplication using JAX [10], which employs XLA JIT-compilation to the respective hardware. We compared the runtime between execution on 24 Intel Xeon E5-2643 v3 CPUs and execution on an NVIDIA Titan X GPU. As a performance measure, we used the average runtime of 100 matrix multiplications for random matrices with i.i.d. sampled entries from a standard normal distribution. The CPU/GPU runtime quotient as a function of the matrix dimension first increases up to a factor of approximately 70 and then decreases again to a factor of approximately 10. For the smaller matrices, the GPU's parallel architecture cannot be fully exploited and the fewer but faster CPU cores yield a relatively high throughput. For the very large matrices, the GPU's parallel processing capabilities are saturated, and additional computations are executed sequentially. Even in this saturated regime, the GPU architecture is roughly a factor 10 faster than the CPU architecture. The peak relative performance of the GPU occurs when its parallel processing capabilities can be fully exploited.

1.3 Automatic Differentiation

The foregoing sections have demonstrated the central role of derivatives for numerical optimization and the training of deep learning models. Hence, efficient algorithms to compute derivatives in a large-scale setting are of paramount importance. This can be provided by the framework of *automatic differentiation*, the evaluation of *exact* derivatives by means of an algorithmic procedure. Here, *exact* means up to numerical errors, but without approximation errors. In contrast, one may construct an approximation to the derivative by using function evaluations (e.g., finite differences), an approach often termed *numerical differentiation*.

Modern automatic differentiation libraries provide this functionality while hiding the implementation from the user. However, the design of novel features for deep learning models and the efficient implementation of optimization algorithms require a deeper understanding of the main principles. For the implementation of the algorithms presented in the associated publications, we used the automatic differentiation libraries JAX [10] and PyTorch [42]. While they differ in their details, both share the same fundamental concepts to compute derivatives, which we outline in this section. To implement our neural network training algorithm Proximal Backpropagation in [18], we modify the backward pass of reverse-mode automatic differentiation (Section 1.3.2). In [19], we develop an approach to enforce a type of hard constraint on neural network activations. The modularity of automatic differentiation codes allows us to implement our approach as a differentiable module, which may be easily incorporated in existing models. In [17], we design a complex objective function that requires a gradient with respect to the initial state of a differential equation. We can build on JAX's implementation based on [12], which is why there is no need for a cumbersome implementation of this gradient from scratch. Automatic differentiation codes often contain parallelizable computations such as matrix multiplications or element-wise vector operations. In Section 1.2.5, we demonstrate the significant impact of hardware accelerators for such systems. All codes for the associated publications were executed on a graphical processing unit (GPU).

1.3.1 Computational Graphs

A computation can be represented by a directed, acyclic *computational graph*. Figure 5 shows a computational graph for a simple function, in which each *edge* of the graph denotes an elementary computation and each *node* (or *vertex*) is associated with the numerical value of the computation at this stage. Such a graph admits a topological ordering, i.e., the nodes may be sorted in a sequence in which each node depends only on previous nodes. A vertex v receives computation results from its *parent* vertices and provides its own computation result to its *child* vertices. To establish a notation for graph traversal, we denote the list of all vertices of a computational graph G by $\mathcal{V}(G)$, the set of all parents of a vertex v by $\mathcal{P}(v)$, and the set of all its children by $\mathcal{C}(v)$. For example, for the graph depicted in Figure 5, $\mathcal{P}(v_3) = \{v_1, v_2\}$ and $\mathcal{C}(v_3) = \{v_5\}$.

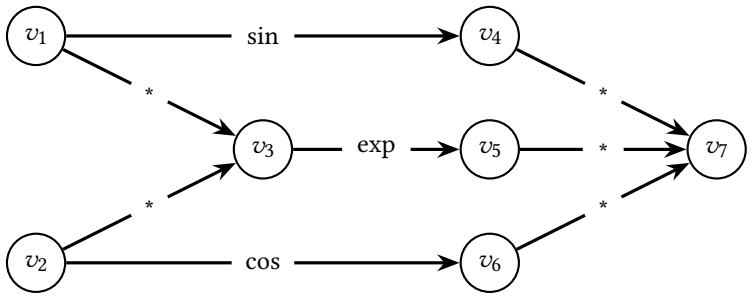


Figure 5: Computational graph with vertices enumerated as v_i, in which each edge represents an elementary computation. Altogether, they evaluate $f(x) = \sin(x_1)\cos(x_2)\exp(x_1 x_2)$ for an input $x = (x_1, x_2) \in \mathbb{R}^2$.

1.3.2 Forward- and Reverse-Mode Automatic Differentiation

To introduce the two main modes of automatic differentiation, we assume a sequential computational graph, in which nodes are aggregated to *layers* and every layer depends only on its directly preceding layer,

$$x \mapsto y^1 \mapsto \cdots \mapsto y^L = f(x) \,. \tag{34}$$

Explicitly, we have $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m, f = f^L \circ f^{L-1} \circ \cdots \circ f^1$. We denote intermediate variables as $y^l = f^l(y^{l-1}) \in \mathbb{R}^{n_l}$, with $y^0 = x$ and $y^L = f(x)$. Note that conceptually this sequential model is not a restriction. For a general computational graph, nodes that skip a level of the topological ordering could be copied to obtain a sequential dependence. For example, the computation depicted in Figure 5 may be represented in sequential form as shown in Figure 6. This would of course introduce a memory overhead and would not lead to a sensible implementation for a general computational graph.

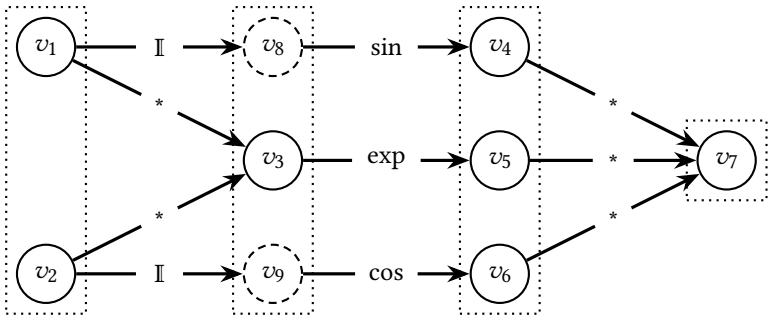


Figure 6: Computational graph from Figure 5 represented in sequential form. The additional vertices v_8, v_9 (dashed circles) are added by an identity operation from v_1, v_2. The layers of the model may now be aggregated (dotted rectangles) such that every layer depends only on its directly preceding layer.

Automatic differentiation is based on the chain rule of multivariate calculus. The difficulty of presenting this subject is more a matter of notation than conceptual depth and hence we make sure to carefully specify our notation. For a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m, x \mapsto f(x)$, we denote the Jacobian by $\partial f / \partial x$ with elements

$$\left(\frac{\partial f}{\partial x}\right)_{ij} = \frac{\partial f_i}{\partial x_j} \,. \tag{35}$$

Here, x_j and f_i are the component variables of the input and output, respectively. The chain rule for the composition of two differentiable operations, $x \in \mathbb{R}^n \mapsto y \in \mathbb{R}^a \mapsto f(x) \in \mathbb{R}^m$, may then be written as

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} \,. \tag{36}$$

This product is in general a matrix-matrix product. Computation of the partial derivatives can thus be reduced to computing derivatives of more elementary intermediate functions. To keep the notation concise, we overload function names and the names of their output variable. For example, y in (36) represents the intermediate variable $y \in \mathbb{R}^a$ and the function $y : \mathbb{R}^n \to \mathbb{R}^a$.

For the general sequential composition (34) we have two choices of applying the chain rule. We can either substitute in *reverse-mode* or in *forward-mode*:

$$\frac{\partial f}{\partial x} = \boxed{\frac{\partial f}{\partial y^{L-1}}}\frac{\partial y^{L-1}}{\partial x} \quad \text{(reverse-mode)} \tag{37}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y^{1}}\boxed{\frac{\partial y^{1}}{\partial x}} \quad \text{(forward-mode)} \tag{38}$$

In both cases, the boxed part can be directly evaluated, while the other part is further substituted to arrive at the same fully expanded expression,

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y^{L-1}}\frac{\partial y^{L-1}}{\partial y^{L-2}} \cdots \frac{\partial y^{2}}{\partial y^{1}}\frac{\partial y^{1}}{\partial x} \,. \tag{39}$$

Note that with this notation, the evaluation order in reverse-mode is from left to right, while the evaluation order in forward-mode is from right to left. Even though both approaches arrive at this same expression, the computational complexity of the evaluation order differs. To clarify this aspect, we consider a composition with two intermediate variables,

$$x \in \mathbb{R}^n \mapsto y \in \mathbb{R}^a \mapsto z \in \mathbb{R}^b \mapsto f(x) \in \mathbb{R}^m \,. \tag{40}$$

The chain rule can be explicitly written as:

$$\frac{\partial f}{\partial x} = \left(\frac{\partial f}{\partial z}\frac{\partial z}{\partial y}\right)\frac{\partial y}{\partial x} \quad \text{(reverse-mode)} \tag{41}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z}\left(\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\right) \quad \text{(forward-mode)} \tag{42}$$

Here, the parentheses enclose the operation that is first evaluated in the respective case. To compute this Jacobian $\partial f/\partial x$, reverse-mode requires $(mab + man)$ multiplications, while forward-mode requires $(nab + mbn)$ multiplications. It follows that reverse-mode is favorable if $(m - n)/mn < (b - a)/ab$. Generalizing from this case, we could compute the number of arithmetic operations necessary for the sequential computation (39) and then determine which mode to use. However, pure forward- and reverse-mode automatic differentiation are only the two extremes of a possible evaluation order. In general, an optimal evaluation order might constitute a particular order of intermediate matrix products. Determining this optimal evaluation order is an NP-complete problem [39].

1.3.3 Implementing Automatic Differentiation

Implementing the chain rule directly as a sequence of matrix multiplications as in (39) has several disadvantages. First, this requires to store the matrices explicitly in memory, which becomes quickly prohibitive in a large-scale setting. Secondly, without explicitly implementing sparse matrix multiplication, a matrix product might have a significant overhead, since many common operations have a sparse Jacobian. Examples are element-wise operations, for which the Jacobian is diagonal.

To avoid these issues, we consider general directed, acyclic computational graphs (Figure 5) and require a topological ordering of the graph. We associate a variable v_i with every vertex of the graph and a Jacobian $\partial v_i / \partial v_j$, where v_j is a parent vertex to its child vertex v_i. In contrast to (34), each vertex now may have multiple parents from which it receives a function argument and multiple children to which it passes its computation result. We do not construct a full Jacobian, but instead only a Jacobian-vector product (JVP) or a vector-Jacobian product (VJP). We consider once more our example (40) with two intermediate variables to demonstrate how a JVP or VJP of the overall function may be computed from the same type of operation on intermediate variables. A VJP of a vector $q \in \mathbb{R}^m$ and JVP of a vector $p \in \mathbb{R}^n$ with the Jacobian $\partial f / \partial x \in \mathbb{R}^{m \times n}$ can be expanded by the chain rule as:

$$\mathbb{R}^n \ni q^T \left(\frac{\partial f}{\partial x} \right) = \sum_k \frac{\partial y_k}{\partial x} \left(\sum_j \frac{\partial z_j}{\partial y_k} \left(\sum_i \frac{\partial f_i}{\partial z_j} q_i \right) \right) \qquad \text{(VJP)} \tag{43}$$

$$\mathbb{R}^m \ni \left(\frac{\partial f}{\partial x} \right) p = \sum_j \frac{\partial f}{\partial z_j} \left(\sum_k \frac{\partial z_j}{\partial y_k} \left(\sum_i \frac{\partial y_k}{\partial x_i} p_i \right) \right) \qquad \text{(JVP)} \tag{44}$$

Both formulations have a recursive structure: once a part in parentheses is evaluated, one is left with the same operation at the previous/subsequent layer. In the following, we outline how to compute VJPs and JVPs for general computational graphs using reverse- and forward-mode automatic differentiation, respectively.

Notation

We explicitly distinguish a vertex variable v and its computed value $\boldsymbol{v}$ by bold face notation, which helps to highlight at what point in the algorithm a variable is computed. Further, we denote the set of all parent vertices of a vertex v by $\mathcal{P}(v)$ and the set of all child vertices by $\mathcal{C}(v)$. To describe a parent vertex of a vertex v, we use the letter w, i.e., a dependency $w \mapsto v$. Since a node may depend on several parent vertices, all of these must be evaluated first. To make this explicit, we write $\boldsymbol{v} = v(\boldsymbol{w} \in \mathcal{P}(v))$. We denote a Jacobian by $\partial v / \partial w$ and use the notation $\partial v / \partial \boldsymbol{w}$ to describe that this Jacobian is explicitly evaluated at a point $\boldsymbol{w}$. More so, we imply that *all* parents of v are already evaluated. The simple example $v(\boldsymbol{w}_1, \boldsymbol{w}_2) = \boldsymbol{w}_1^T \boldsymbol{w}_2$ demonstrates the necessity for this convention. Nodes are vector-valued and we assume that there is a single input node v_{in} and a single output node v_{out}. This clarifies the subsequent presentation and is without loss of generality, as one can always prepend an input node that distributes an input vector to different vertices and append an output node that aggregates the computation result into a single output vector.

Reverse-Mode Automatic Differentiation

A VJP is computed in reverse-mode by evaluating an expression of the form

$$\bar{\boldsymbol{w}} = \sum_{v \in \mathcal{C}(w)} \bar{\boldsymbol{v}}^T \left(\frac{\partial v}{\partial \boldsymbol{w}} \right) . \tag{45}$$

A node is implemented by providing a function v that computes the node's value from its parents' values, $\boldsymbol{v} = v(\boldsymbol{w} \in \mathcal{P}(v))$. Furthermore, the VJP-rule $q^T(\partial v/\partial w)$ for all $w \in \mathcal{P}(v)$ must be specified. A forward-pass evaluates the computational graph at a provided point $\boldsymbol{v}_{\text{in}} = \boldsymbol{x}$. The algorithm iterates over the graph in topological order and evaluates all nodes.

VJPs are computed in a *subsequent* backward pass over the topologically sorted graph. To this end, a vector $\bar{\boldsymbol{v}}_{\text{out}} = \boldsymbol{q}$ is provided for which the VJP should be computed. Now, we can iterate over the transposed graph to compute the quantities (45). Again, the topological ordering guarantees a viable computation order. This procedure is summarized in Algorithm 2. The nodes' values have to be stored to evaluate derivatives in the backward pass, which might incur a significant memory cost. One approach to alleviate the memory burden is by *checkpointing* [23]. Here, only the values of some nodes are stored in a forward pass and during the backward pass the computation is partially re-executed to obtain the missing values. This method therefore trades-off memory cost with computation cost. We apply checkpointing in [17] to train a memory-intensive model.

A full Jacobian may be computed by choosing $q = \boldsymbol{e}_k, k \in \{1, \dots, m\}$ as a standard basis vector. Then the vector $\boldsymbol{e}_k^T \boldsymbol{J}$ is the k-th row of the Jacobian $\boldsymbol{J}$ and thus the full Jacobian can be constructed row-wise. In particular, obtaining the gradient of a scalar function ($m = 1$), such as during training of a deep learning model, requires only a single VJP backward pass.

Forward-Mode Automatic Differentiation

A JVP is computed in forward-mode as a directional derivative of a vertex variable v with respect to the input vector $\boldsymbol{x} \in \mathbb{R}^n$ in a direction $\boldsymbol{p} \in \mathbb{R}^n$,

$$D_{\boldsymbol{p}} v(\boldsymbol{x}) := \left(\frac{\partial v}{\partial \boldsymbol{x}} \right) \boldsymbol{p} = \sum_{w \in \mathcal{P}(v)} \left(\frac{\partial v}{\partial \boldsymbol{w}} \right) D_{\boldsymbol{p}} w(\boldsymbol{x}) . \tag{46}$$

In addition to the node's function, every node needs to be provided with the JVP-rule $(\partial v/\partial w)p$ for all $w \in \mathcal{P}(v)$. In a *single* forward pass, the JVPs are evaluated simultaneously with the nodes' values. The algorithm is initialized with $\boldsymbol{v}_{\text{in}} = \boldsymbol{x}$ and $D_{\boldsymbol{p}} v_{\text{in}}(\boldsymbol{x}) = \boldsymbol{p}$. By iterating over the topologically sorted graph, expression (46) is iteratively computed. Again, the topological ordering guarantees that all parent vertices have completed their computation before their child vertices. This procedure is summarized in Algorithm 3. In contrast to computing a VJP in reverse-mode, values of past vertices can be discarded and hence the memory requirement does not scale with the graph size.

A full Jacobian may be computed by choosing the standard basis vectors, $p = \boldsymbol{e}_k, k \in \{1, \dots, n\}$. The vector $\boldsymbol{J}\boldsymbol{e}_k$ is the k-th column of the Jacobian $\boldsymbol{J}$, which hence can be constructed column-wise.

Algorithm 2 Reverse-mode automatic differentiation via VJPs

Require: computational graph G associated with a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$; input vector $\boldsymbol{x} \in \mathbb{R}^n$; vector $\boldsymbol{q} \in \mathbb{R}^m$ to evaluate a vector-Jacobian product $q^T(\partial f / \partial x)$

$G_s \longleftarrow \text{toposort}(G)$
for $v \in \mathcal{V}(G_s)$ **do** ▷ forward pass
  if $v = v_{\text{in}}$ **then**
    $\boldsymbol{v} \longleftarrow \boldsymbol{x}$
  else
    $\boldsymbol{v} \longleftarrow v(\boldsymbol{w} \in \mathcal{P}(v))$
  end if
  if $v = v_{\text{out}}$ **then**
    $\bar{\boldsymbol{v}} \longleftarrow \boldsymbol{q}$
  else
    $\bar{\boldsymbol{v}} \longleftarrow 0$
  end if
end for
for $v \in \text{reversed}(\mathcal{V}(G_s))$ **do** ▷ backward pass
  for $w \in \mathcal{P}(v)$ **do**
    $\bar{\boldsymbol{w}} \longleftarrow \bar{\boldsymbol{w}} + \bar{\boldsymbol{v}}^T \left(\frac{\partial v}{\partial \boldsymbol{w}}\right)$
  end for
end for
return $(\boldsymbol{v}_{\text{out}}, \bar{\boldsymbol{v}}_{\text{in}})$

Algorithm 3 Forward-mode automatic differentiation via JVPs

Require: computational graph G associated with a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$; input vector $\boldsymbol{x} \in \mathbb{R}^n$; vector $\boldsymbol{p} \in \mathbb{R}^n$ to evaluate a Jacobian-vector product $(\partial f / \partial x)p$

$G_s \longleftarrow \text{toposort}(G)$
for $v \in \mathcal{V}(G_s)$ **do** ▷ forward pass
  if $v = v_{\text{in}}$ **then**
    $\boldsymbol{v} \longleftarrow \boldsymbol{x}$
    $D_{\boldsymbol{p}} v(\boldsymbol{x}) \longleftarrow \boldsymbol{p}$
  else
    $\boldsymbol{v} \longleftarrow v(\boldsymbol{w} \in \mathcal{P}(v))$
    $D_{\boldsymbol{p}} v(\boldsymbol{x}) \longleftarrow 0$
    for $w \in \mathcal{P}(v)$ **do**
      $D_{\boldsymbol{p}} v(\boldsymbol{x}) \longleftarrow D_{\boldsymbol{p}} v(\boldsymbol{x}) + \left(\frac{\partial v}{\partial \boldsymbol{w}}\right) D_{\boldsymbol{p}} w(\boldsymbol{x})$
    end for
  end if
end for
return $(\boldsymbol{v}_{\text{out}}, D_{\boldsymbol{p}} v_{\text{out}}(\boldsymbol{x}))$

Remarks on the VJP-/JVP-based Implementation of Automatic Differentiation

In both modes, the function needs to be evaluated in a forward pass to compute derivatives. Hence, if a VJP/JVP is needed, the function value is obtained without additional cost. Using the VJP/JVP-primitives yields a matrix-free implementation of reverse- and forward-mode automatic differentiation, which avoids to store intermediate matrices explicitly in memory. Also, both methods require the same amount of arithmetic operations to compute a single VJP/JVP. This can be readily seen for the simple sequential example (40) and in the general case by counting operations in the innermost loop of Algorithm 2 and Algorithm 3. For $v \in \mathbb{R}^{n_v}$ we need $\sum_{v \in \mathcal{V}(G)} \sum_{w \in \mathcal{P}(v)} n_v n_w$ multiplications in both cases. However, computing a VJP requires two graph traversals and the storage of intermediate values as opposed to a single graph traversal with a constant memory cost for a JVP. If a full Jacobian is required, it may be computed row-wise via VJPs and column-wise via JVPs. Consequently, to compute the Jacobian of a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, a JVP-based computation is favorable for $m \gg n$ and $m \approx n$, while a VJP-based computation is favorable for $m \ll n$. In particular, when computing the gradient of a scalar function with $m = 1$, reverse-mode automatic differentiation should be applied. This is, for example, the case with gradient-based optimization in deep learning. A task that involves both modes is the computation of a Hessian matrix of a scalar function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. To compute the Jacobian, it is favorable to use reverse-mode to obtain a mapping $J : \mathbb{R}^n \longrightarrow \mathbb{R}^n$. Computing derivatives of this mapping to obtain the Hessian is then advantageous in forward-mode. We close this discussion by mentioning that there exist situations where a VJP/JVP implementation to obtain a full Jacobian would require more floating-point operations than working with explicit intermediate Jacobians. In such situations, the VJP/JVP implementation can be considered as a trade-off between arithmetic operations and memory consumption. To state an explicit example, consider the case of an L-layer sequential model (34) with dense Jacobians $J_l \in \mathbb{R}^{n_l \times n_{l-1}}$. The overall Jacobian is given by the product $J = J_L J_{L-1} \dots J_1$. A VJP/JVP effectively computes this product by prepending and appending the identity matrix, respectively:

$$\underbrace{(J_L J_{L-1} \dots J_1)}_{\text{direct}} \qquad \underbrace{\mathbb{I}_{n_L} (J_L J_{L-1} \dots J_1)}_{\text{VJP-based}} \qquad \underbrace{(J_L J_{L-1} \dots J_1) \mathbb{I}_{n_0}}_{\text{JVP-based}} \tag{47}$$

It follows that a VJP-based implementation incurs an additional cost of $n_L^2 n_{L-1}$ multiplications and a JVP-based implementation has an overhead of $n_0^2 n_1$ multiplications.

All our computational cost estimates count floating-point multiplications. In practice, however, exploiting a particular hardware accelerator is a decisive factor for an efficient implementation (Section 1.2.5).

1.3.4 Implicit Differentiation

To compute derivatives, we have so far explicitly followed the execution of the computational graph, either forward along the evaluation of the function itself (forward-mode) or backward in a separate pass after evaluation of the function (reverse-mode). For certain computations, this procedure is disadvantageous. Take for example an optimization problem such as the following regularized least-squares problem for given $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$,

$$x_*(\theta) = \operatorname*{argmin}_{x \in \mathbb{R}^n} f(\theta, x), \quad f(\theta, x) \coloneqq \frac{1}{2}\|Ax - b\|_2^2 + \frac{\theta}{2}\|x\|_2^2, \quad \theta > 0 \,. \tag{48}$$

Different regularization parameters θ yield different solutions x_*, and we may ask how the solution changes with a change in this parameter. We therefore consider the solution as a *function* of the regularization parameter, $x_*(\theta)$, and are interested in the derivative $\partial x_*/\partial\theta$. To compute this derivative, we could explicitly follow the computational graph of an iterative numerical solver. However, this computational graph may be very deep. Accumulating the derivatives would require to traverse the whole graph and could lead to accumulating numerical errors. Furthermore, if we applied reverse-mode automatic differentiation, we would have to store intermediate values and the algorithm's memory-requirement would be proportional to the depth of the graph.

However, there is a different way of obtaining the derivative $\partial x_*/\partial\theta$ without explicitly following the computation. This is possible by exploiting a more abstract condition about the solution, namely its first-order optimality condition. The optimal solution to this convex problem can be explicitly computed by setting the gradient of the objective function equal to zero,

$$\nabla_x f(\theta, x) = (A^T A + \theta \mathbb{I})x - A^T b = 0 \,. \tag{49}$$

Obtaining the solution requires solving this linear system for x. Since the matrix $(A^T A + \theta\mathbb{I})$ is positive definite and hence invertible for all $\theta > 0$, the solution can be explicitly written as

$$x_*(\theta) = (A^T A + \theta \mathbb{I})^{-1} A^T b \,. \tag{50}$$

We define the function $F(\theta, x) \coloneqq \nabla_x f(\theta, x)$ to characterize the solution by the condition

$$F(\theta, x_*(\theta)) = 0 \,. \tag{51}$$

To be precise, we use the notation $\mathrm{d}F/\mathrm{d}\theta$ to denote the total derivative, while we reserve $\partial F/\partial\theta$ for the partial derivative. We compute the total derivative of (51) with respect to the parameter θ to obtain

$$\frac{\mathrm{d}F}{\mathrm{d}\theta}(\theta, x_*(\theta)) = x_*(\theta) + (A^T A + \theta \mathbb{I})\frac{\partial x_*}{\partial \theta} = 0 \,. \tag{52}$$

Rearranging this equation yields the result,

$$\frac{\partial x_*}{\partial \theta} = -(A^T A + \theta \mathbb{I})^{-1} x_*(\theta) \,. \tag{53}$$

In particular, this Jacobian is independent of *how* the solution to the optimization problem was actually computed, since it only depends on the minimizer x_*.

We now generalize from this example to demonstrate how to differentiate through an implicit condition. Suppose we can characterize our computation by an implicit condition of a continuously differentiable function F,

$$F(\hat{\theta}, \hat{x}) = 0, \quad F : \mathbb{R}^p \times \mathbb{R}^n \longrightarrow \mathbb{R}^n \,. \tag{54}$$

We want to find an explicit *local parameterization* $x(\theta)$ around a point $\hat{\theta}$ for which $x(\hat{\theta}) = \hat{x}$ and $F(\hat{\theta}, \hat{x}) = 0$. In particular, we are interested in the derivative $\partial x / \partial \theta(\hat{\theta})$ at this point. To this end, we compute the total derivative of (54),

$$0 = \frac{\mathrm{d}F}{\mathrm{d}\theta}(\hat{\theta}, x(\hat{\theta})) = \underbrace{\frac{\partial F}{\partial \theta}(\hat{\theta}, \hat{x})}_{\mathbb{R}^{n \times p}} + \underbrace{\frac{\partial F}{\partial x}(\hat{\theta}, \hat{x})}_{\mathbb{R}^{n \times n}} \underbrace{\frac{\partial x}{\partial \theta}(\hat{\theta})}_{\mathbb{R}^{n \times p}} \,. \tag{55}$$

The solution for $\partial x / \partial \theta(\hat{\theta})$ may be found by solving the resulting linear system. More precisely, these are p linear systems with the same system matrix $\partial F / \partial x(\hat{\theta}, \hat{x})$. For uniqueness, we require this matrix to be non-singular. A singular matrix would imply that there are several values $\hat{x}$ that correspond to the value $\hat{\theta}$, i.e., no explicit local parameterization $x(\theta)$ is possible. This procedure is summarized by the *implicit function theorem*.

One approach to solve these systems numerically is by applying an LU-factorization of the system matrix at a computational cost of $\mathcal{O}(n^3)$ [49]. With this factorization at hand, solving each system requires $\mathcal{O}(n^2)$ operations, so in sum $\mathcal{O}(pn^2)$. Altogether, this implies a computational cost of $\mathcal{O}(n^3 + pn^2)$ operations to solve (55), which scales as $\mathcal{O}(n^3)$, if $p \in \mathcal{O}(n)$.

Derivatives that are computed via such an implicit condition may be used just as any derivative obtained from an explicit formulation for forward- and reverse-mode automatic differentiation by specifically defining a custom JVP-/VJP-rule.

II SUMMARY OF PUBLICATIONS

This publication-based dissertation comprises the following publications:

OPTIMIZERS FOR DEEP LEARNING

Thomas Frerix*, Thomas Möllenhoff*, Michael Moeller*, and Daniel Cremers. Proximal Backpropagation. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. 2018. (* denotes equal contribution)

CONSTRAINED DEEP LEARNING

Thomas Frerix, Matthias Nießner, and Daniel Cremers. Homogeneous Linear Inequality Constraints for Neural Network Activations. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2020

RELAXATION OF A COMBINATORIAL OPTIMIZATION PROBLEM

Thomas Frerix and Joan Bruna. Approximating Orthogonal Matrices with Effective Givens Factorization. In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 2019

DEEP LEARNING FOR VARIATIONAL DATA ASSIMILATION

Thomas Frerix, Dmitrii Kochkov, Jamie A. Smith, Daniel Cremers, Michael P. Brenner, and Stephan Hoyer. Variational Data Assimilation with a Learned Inverse Observation Operator. In: *Proceedings of the 38th International Conference on Machine Learning (ICML)*. 2021

Stated for completeness, but not part of this dissertation, the author contributed to:

Philip Haeusser, Thomas Frerix, Alexander Mordvintsev, and Daniel Cremers. Associative Domain Adaptation. In: *Proceedings of the International Conference on Computer Vision (ICCV)*. 2017

2.1 Proximal Backpropagation

Citation

Thomas Frerix*, Thomas Möllenhoff*, Michael Moeller*, and Daniel Cremers. Proximal Backpropagation. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. 2018. (* denotes equal contribution)

Author Contributions

The author of this dissertation significantly contributed to

- developing the main concepts
- implementing the algorithm
- evaluating the numerical experiments
- writing the paper

Summary

First-order optimization is at the core of training a deep learning model. Consequently, novel training algorithms may increase the speed of training (training efficiency) and improve the generalization properties of the model (training effectiveness). In this paper, we outline a general principle of obtaining neural network optimizers inspired by a quadratic penalty function, which couples the layers of a deep neural network by L_2-norm terms. Formally, this function may be written as

$$E(\theta, a, z) = L_y(\phi(\theta^{L-1}, a^{L-2})) + \sum_{l=1}^{L-2} \frac{\gamma}{2}\|\sigma(z^l) - a^l\|_2^2 + \frac{\rho}{2}\|\phi(\theta^l, a^{l-1}) - z^l\|_2^2 \,. \tag{56}$$

Here, the linear activations z^l are the output of an affine function ϕ and the nonlinear activations a^l are the output of a nonlinear activation function σ, where the index l counts the layers of a feed-forward neural network. The final neural network loss L_y depends only on the previous layer and a learning target y. In this formulation, the parameters θ *and* the activations a, z are optimization variables.

We do not directly minimize (56), but instead take sequential optimization steps on the quadratic penalty terms in a backward pass through the network. We prove that a gradient descent step on the original neural network loss is recovered for $\gamma = \rho = 1/\tau$ by taking gradient steps with step size τ on these quadratic terms. More generally, this point of view now allows to substitute these sequential gradient steps with other minimization steps yielding different optimization algorithms. We derive one such algorithm, *Proximal Backpropagation (ProxProp)*, which takes a proximal step (or implicit gradient step) instead of an explicit gradient step to update the parameters of the linear layers. A proximal step at iteration k with step size τ on a function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$x_{k+1} = \operatorname*{argmin}_{x \in \mathbb{R}^n} f(x) + \frac{1}{2\tau}\|x - x_k\|_2^2 \tag{57}$$

$$= x_k - \tau \nabla f(x_{k+1}) \,. \tag{58}$$

In other words, to obtain the next iterate, we minimize the objective function while remaining somewhat close to the current iterate. This trade-off is mediated by the step size τ. By setting the gradient of (57) equal to zero, we obtain (58). This is a gradient step for which the gradient is evaluated at the *next* iterate. Hence, the term *implicit* gradient step. In our case, the function f is one of the quadratic terms in (56) associated with the affine function ϕ. It follows that in order to perform the update, we have to solve a linear system. We demonstrate how to approximately solve this system efficiently using a conjugate gradient solver. While gradient descent is not guaranteed to converge with an arbitrary fixed step size, this iteration converges for any step size $\tau > 0$. Indeed, we demonstrate numerically that the algorithm is stable for large step sizes.

In our convergence analysis, we prove that the update direction of ProxProp is a descent direction and that fixed points of the update iteration are stationary points of the neural network loss function. It follows that ProxProp minimizes this loss. With the same proof strategy, it is possible to show that other update schemes based on this general principle minimize the neural network loss. Importantly, algorithms derived from this principle may be implemented with common deep learning frameworks by modifying the backward pass of a layer. Furthermore, the ProxProp update direction may be used with other optimizers that only assume to be provided with a descent direction.

We close by demonstrating that ProxProp yields comparable performance with gradient-based optimization.

2.2 Homogeneous Linear Inequality Constraints for Neural Network Activations

Citation

Thomas Frerix, Matthias Nießner, and Daniel Cremers. Homogeneous Linear Inequality Constraints for Neural Network Activations. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2020

Author Contributions

The author of this dissertation significantly contributed to

- developing the main concepts
- implementing the algorithm
- evaluating the numerical experiments
- writing the paper

Summary

Current deep learning systems are mostly unconstrained black-box systems. The data they are trained with, however, often have properties that are amenable to a mathematical description. For example, they may satisfy symmetric, geometric, or physical constraints. It is desirable to incorporate such constraints into deep learning systems for two reasons. First, properties should be guaranteed in applications where they are necessary to preclude detrimental behavior, e.g., in medical applications. Secondly, learning (an approximation to) these properties from scratch requires an enormous amount of training data, while such constraints often have a relatively simple mathematical formulation. For constrained optimization problems, a large body of theoretical results and optimization algorithms exists, which can provide inspiration for constrained deep learning. While hand-designed solutions for specific applications have been proposed, an efficient approach to incorporating generic constraint classes in deep learning models does not yet exist. Generally, two different approaches are considered to enforce constraints: soft constraints and hard constraints. In a soft constraint approach, a distance function $d(\cdot, C)$ from the feasible set C is explicitly added to the objective function,

$$\min_{x \in \mathbb{R}^n} f(x) + \lambda d(x, C) \, . \tag{59}$$

Here, the two components are traded-off via a parameter λ. Hence, the optimizer is drawn toward the feasible set, but there is no guarantee about constraint satisfaction of the result.

In contrast, a hard constraint approach directly searches for solutions over the feasible set,

$$\min_{x \in C} f(x) \, . \tag{60}$$

In this paper, we present a method to enforce a specific type of hard constraint on neural network activations. This algorithm guarantees that neural network activations satisfy homogeneous linear inequality constraints of the form

$$\{x \in \mathbb{R}^n \mid Ax \leq 0\}, \quad A \in \mathbb{R}^{m \times n} \, , \tag{61}$$

for which the m constraints are provided by specifying the matrix A.

We design a constraint parameterization layer whose output parameterizes the feasible set (61). This set can be represented by a polyhedral cone spanned by s rays $\{r_1, \dots, r_s\}$. This duality, named after Minkowski and Weyl, may be written as

$$\{x \in \mathbb{R}^n \mid Ax \leq 0\} = \left\{ \sum_{j=1}^{s} \mu_j r_j \;\middle|\; \mu_j \geq 0 \right\} . \tag{62}$$

The left-hand side is a natural way to specify such constraints by providing the matrix A, while the right-hand side provides a parameterization of the feasible set by varying the conic combination parameters $\mu_j \geq 0$. We can switch between these dual points of view via Fukuda's double description method. The number s of rays depends on the matrix A and is a priori not known.

Our algorithm proceeds in two stages starting with a matrix A that specifies the set of constraints. Prior to training, we compute the conic representation using the double description method. Then we incorporate this representation as a differentiable layer. While this procedure may incur a significant computational cost prior to training, the layer subsequently enforces these constraints at the cost of a fully-connected layer during training and inference. This is the major advantage compared with unconstrained training followed by a projection onto the feasible set, which requires solving a (convex) optimization problem during inference.

We consider generative modeling as a relevant application for constrained deep learning and demonstrate this idea by training a constrained variational autoencoder. Our experiments indicate that care must be taken when enforcing such hard constraints in neural network training. While evaluating the hard-constrained network during inference is faster than adding a final projection step, the loss converges slower compared with unconstrained training. This might be a general problem when hard constraints are combined with gradient-based optimization. Our intuition is that the constraints reduce the degrees of freedom during optimization, while current research suggests that over-parameterization helps the commonly used optimizers.

2.3 Approximating Orthogonal Matrices with Effective Givens Factorization

Citation

Thomas Frerix and Joan Bruna. Approximating Orthogonal Matrices with Effective Givens Factorization. In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 2019

Author Contributions

The author of this dissertation significantly contributed to

- developing the main concepts
- implementing the algorithm
- evaluating the numerical experiments
- writing the paper

Summary

Combinatorial optimization problems can be difficult to solve. One solution strategy is to formulate an easier to solve approximation to the original problem, a so-called relaxation. Rather than the discrete combinatorial problem, the relaxed problem is often continuous. In this paper, we propose a continuous relaxation of a combinatorial problem with applications in numerical linear algebra: effectively factorizing an orthogonal matrix.

Our motivating example for factorizing orthogonal matrices is the Fast Fourier transform (FFT), which speeds up Fourier transformation in n dimensions from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log(n))$, catalyzing a revolution in large-scale signal processing. Can such speed-ups be obtained for general orthogonal matrices? The formulation we analyze in this paper is a truncated factorization. Suppose an orthogonal matrix can be approximated with a product of N factors, where the computational cost of applying each factor is constant. Then, applying this approximation to a vector requires $\mathcal{O}(N)$ operations in contrast to the $\mathcal{O}(n^2)$ operations of a dense matrix-vector product. If the factorization is computed once and subsequently applied many times, it is sensible to first invest computational resources in finding such a factorization.

A special orthogonal matrix can be considered as a rotation in high-dimensional space. We analyze a factorization of an orthogonal matrix as a product of so-called Givens matrices. Each Givens matrix $G(i, j, \alpha)$ rotates in a two-dimensional (i, j)-subspace by an angle α. In n dimensions there exists a simple constructive algorithm to obtain a factorization with at most $n(n-1)/2$ factors, the dimension of the orthogonal group. However, finding the best approximation with a limited number of factors poses a difficult combinatorial optimization problem.

When considering the orthogonal group as a Lie group, multiplication with a Givens matrix $G(i, j, \alpha)$ can be viewed as a step of size α in the direction of the (i, j)-basis coordinate of the manifold. Consequently, if we start with a matrix U to be factorized and take coordinate steps

with Givens matrices G_k^T until we have reached the identity matrix (up to permutation), the transposed sequence of these Givens factors represents a factorization of the original matrix,

$$G_N^T \dots G_1^T U = \mathbb{I} \,. \tag{63}$$

Since the permutation matrices are the sparsest elements of the orthogonal group, it is sensible to choose the Givens factors by minimizing a sparsity-inducing objective function. We formulate a relaxation of the factorization problem by minimizing the matrix L_1-norm over the orthogonal group, which we optimize with a manifold coordinate descent algorithm.

We demonstrate that effective factorization in the sense of $N \in \mathcal{O}(n \log(n))$ for a generic orthogonal matrix is not possible. Therefore, we compare our algorithm with baseline methods on matrices for which we can control complexity by construction. Finally, we demonstrate the algorithm for factorizing the orthogonal matrix that diagonalizes the graph Laplacian. This matrix can be used to perform a graph Fourier transform, which is an important operation in graph signal processing.

A relaxation of the combinatorial problem with a continuous function can be the starting point for deep learning approaches. We attempted to directly learn an objective function, which yielded worse results than the L_1-norm heuristic. Replacing an objective function entirely with a deep learning model is generally difficult, since there is only little control over the mathematical properties of the learned function.

2.4 Variational Data Assimilation with a Learned Inverse Observation Operator

Citation

Thomas Frerix, Dmitrii Kochkov, Jamie A. Smith, Daniel Cremers, Michael P. Brenner, and Stephan Hoyer. Variational Data Assimilation with a Learned Inverse Observation Operator. In: *Proceedings of the 38th International Conference on Machine Learning (ICML)*. 2021

Author Contributions

The author of this dissertation significantly contributed to

- developing the main concepts
- implementing the algorithm
- evaluating the numerical experiments
- writing the paper

Summary

Weather forecasting is indispensable for modern society. At the same time, numerical weather prediction systems require enormous computational resources and to further improve forecasts novel algorithmic ideas for more efficient algorithms are necessary. A core principle behind such a prediction system is variational data assimilation. Here, a modeled dynamical system is fitted to data by varying its initial state. This allows to combine prior knowledge about the system with measurements. The dynamical system can subsequently be extrapolated into the future.

In this paper, we present an approach to improve variational data assimilation with deep learning. We use the following formulation of variational data assimilation:

$$J(x_0) = \sum_{t=0}^{T} \|\mathcal{H}(x_t) - y_t\|_2^2, \qquad x_{t+1} = \mathcal{M}(x_t) \tag{64}$$

Here, $\{x_0, \dots, x_T\}$ is the trajectory evolved through the dynamical system $\mathcal{M}$ from an initial state x_0 and $\mathcal{H}$ is the observation operator, which maps physical states x_t to observations (or measurements) y_t. The initial value problem (64) requires solving a difficult nonlinear least-squares problem.

To improve optimizability, we learn an approximate inverse h_θ to the observation operator $\mathcal{H}$, which is parameterized by trainable parameters θ. The deep learning model is trained to map a trajectory of measurements to a trajectory of corresponding physical states such that $h_\theta(y_0, \dots, y_T) \approx (x_0, \dots, x_T)$. We use this learned approximate inverse to improve two aspects of the optimization setting. First, we transform the fitting targets to modify the objective function,

$$\tilde{J}(x_0) = \sum_{t=0}^{T} \|x_t - h_\theta(y_t)\|_2^2, \qquad x_{t+1} = \mathcal{M}(x_t)\,. \tag{65}$$

Here, $h_\theta(y_t)$ denotes a single element of the inverted trajectory $h_\theta(y_0, \dots, y_T)$. The problem is now formulated in physics space instead of in observation space. This objective function is easier to optimize, since we do not have to optimize through the observation operator $\mathcal{H}$. Because (65) is only an approximate surrogate for (64), local minimizers of this transformed objective are not necessarily local minimizers of the original objective. We therefore employ a hybrid strategy that first optimizes the approximate surrogate function (65) and then refines the optimization result by minimizing the original function (64).

As a second application of the deep learning model, we use the first state of an inverted trajectory $h_\theta(y_0, \dots, y_T)$ to initialize the optimizer. Initialization is important for non-convex optimization. The initial state determines the local minimum to which the optimizer is attracted and close enough to this local optimum the problem appears locally convex.

We use L-BFGS as a generic non-convex optimizer and numerically evaluate the algorithm on two chaotic dynamical systems, the Lorenz96 model and a two-dimensional turbulent fluid flow. Our results suggest that far from a local minimum, initially optimizing the surrogate model is efficient, while with an already very good initialization provided by the learned inverse, it is more economical to directly optimize the original objective function.

Altogether, instead of replacing the forecasting system with a deep learning model, we retain the structure of the problem, in particular the exact physics model. We use deep learning only to improve the optimization procedure. This method may be used as a template to improve optimizability of nonlinear inverse problems.

III CONCLUDING REMARKS

3.1 Optimizers for Deep Learning

The current optimization techniques for deep learning employ the setting outlined in Section 1.2.4, i.e., stochastic optimization with first-order methods that use an adaptive step size scheme. However, frequently used data sets, neural network architectures, and optimizers have coevolved toward the current state of deep learning. It implies that changing one aspect of the deep learning setting independent of the others likely deteriorates performance, which is an impediment for fundamentally new ideas. However, it does not imply that a radically different deep learning setting will not work. In fact, several studies challenge this setting and question, for example, the decisive role of stochastic gradient noise [22] or the restriction to first-order methods [37]. In the following, we describe further research directions in neural network optimization that are suggested by the roadmap of this dissertation.

As a direct extension of our work presented in [18], novel neural network optimization algorithms can be derived from the layer-wise coupling point of view. Proximal Backpropagation remains relatively close to the classical backpropagation algorithm, as it replaces certain gradient steps with proximal steps. Further optimizers can be deduced by substituting the gradient steps with different minimization steps that allow a similar analysis.

Shifting toward optimizers that consider curvature information, quasi-Newton methods, such as L-BFGS, are successful solvers for large-scale non-convex optimization (Section 1.1.4). This factor raises the question as to how such optimizers can be employed for deep learning models, where stochastic mini-batch optimization is often required due to memory constraints. Such a stochastic quasi-Newton method must satisfy several requirements:

R1 The quadratic model matrix must capture curvature information about the full-batch objective function.

R2 The quadratic model matrix must remain positive definite to yield a descent direction.

R3 The optimization algorithm should find solutions with low generalization error instead of aggressively optimizing the training loss.

When the quadratic model matrix is updated on the basis of different mini-batches at each iteration, satisfying **R1** is difficult. Specifically, the (L-)BFGS updating scheme (14) is based on gradients evaluated at two different consecutive mini-batches. Propositions to ameliorate this issue include the construction of overlapping mini-batches [6] and the inference of curvature information based on intermittent Hessian evaluations [11]. In contrast to curvature estimation based on sampled mini-batches, the authors of [47] fix the mini-batches prior to optimization and then maintain a separate Hessian approximation for each of them. **R2** must be satisfied by a different method than a Wolfe line search, which is typically employed in the full-batch case. In the stochastic setting, such a line search does not imply the curvature condition (11), because it is based on different consecutive mini-batches. This issue is often circumvented by

simply skipping an update of the quadratic model matrix if the curvature condition is violated. Building further on these results, developing effective stochastic quasi-Newton methods for large-scale deep learning systems is desirable.

Parallel to this line of thinking toward stochastic higher-order optimizers, an improved understanding of the success of stochastic first-order optimizers in the over-parameterized setting can aid the further development of training algorithms. Empirical evidence based on model compression algorithms, such as knowledge distillation [27], indicates that over-parameterization helps the optimization process, but is not necessary for the representation of the learned function. In this approach, a small student neural network is trained on the output of a large teacher neural network. This small student network has better generalization properties compared with the same architecture trained directly on the raw data and may generalize comparably or even better than the large teacher network [21, 3]. Over-parameterized neural network models are often trained to perfect training accuracy and nevertheless generalize well [52]. These empirical findings contradict the classical bias-variance trade-off for model selection, and recent research challenges this notion for over-parameterized neural networks [5]. This observation of well-performing over-parameterized models is not new and has already been observed in the 1990s [33]. However, no comprehensive theory has been proposed to date. Specifically, the role of stochastic gradient descent for the successes of such systems remains elusive. A current stream of research focuses on the implicit bias of (stochastic) gradient descent, i.e., the tendency of an optimizer to converge to solutions with particular properties [2, 41, 36].

3.2 Constrained Deep Learning

Constraints on the output of a neural network can be implemented either as soft constraints or as hard constraints. The former adds a term to the loss function that penalizes a distance from the feasible set, typically an L_2-norm distance; the latter incorporates a structure into the architecture, such that the output of a layer is guaranteed to be feasible. Our experiments in [19] suggest that optimizing a hard-constrained deep learning model may be difficult. Our intuition here is that hard constraints restrict the model's degrees of freedom, thereby reducing the amount of over-parameterization, which is empirically found to help the optimization process [27]. By contrast, a soft constraint approach works well with gradient-based optimization and scales to the large-scale setting with many variables and constraints. However, a post-processing step (e.g., a projection) must be taken to guarantee constraint satisfaction with soft constraints.

To incorporate a generic class of hard constraints, to our knowledge, no practically efficient algorithm has been proposed to date. Two approaches to hard constraints appear in current research: parameterization of feasible sets and implicit differentiation through constraint-defining equations. Our contribution [19] is an example of the former approach. As an example of the latter method, the authors of [1] implicitly differentiate through the optimality conditions of a quadratic program, which comprises the projection onto a convex set as a special case. Toward a scalable method to incorporate hard constraints, further understanding on how they influence the optimization process is beneficial.

3.3 Deep Learning for Variational Data Assimilation

Variational data assimilation combines physical modeling with measurements to make future predictions by solving a difficult nonlinear least-squares problem. The central question for improving optimizability with deep learning is to decide which elements of the classical algorithm should be modified by deep learning components. On the one side of the spectrum is the classical formulation without any deep learning, on the other side is an end-to-end deep learning prediction. We advocate to stay close to the former by *augmenting* the classical formulation with deep learning, rather than replacing it. Our approach presented in [17] is an example; it improves optimizability by modifying the least-squares fitting targets through a learned approximate inverse of the observation operator. The approach still uses the exact dynamical system, and resulting trajectories satisfy the physical laws governing the dynamics, which is an advantage over black-box predictions. However, this assumption can be further relaxed by only requiring that the exact dynamical system be used for extrapolation into the future and not during optimization. Instead, one may emulate the exact dynamical system $\mathcal{M}$ with an approximate deep learning model m_θ, i.e., optimizing the parameters θ on the basis of simulated trajectories, such that $m_\theta(x) \approx \mathcal{M}(x)$ for physical states x. Rather than solving the equations of motion for $\mathcal{M}$, one can generate an approximate trajectory by sequentially applying the model m_θ. Using m_θ instead of $\mathcal{M}$ in a loss function avoids the need to repeatedly solve the equations of motion during optimization, which is computationally expensive. In addition to being efficient to evaluate and backpropagate through, the emulated model can be trained to have favorable properties, such as smoother dependence of the trajectory on the initial state. This emulated model can then be used in a hybrid approach, where one first optimizes with the easier to optimize emulator and then fine-tunes with the exact model to obtain high-accuracy solutions.

3.4 From Numerical Optimization to Deep Learning and Back

This dissertation presents research at the intersection of deep learning and numerical optimization with the recurring theme that innovations in these domains are catalyzed by advances in parallel computation hardware and automatic differentiation libraries. This trend will continue and allow ever more complex end-to-end differentiable deep learning models and advanced optimization pipelines that contain such models.

Judging by the effort that has been invested into developing elaborate methods for large-scale non-convex optimization, it seems unlikely that heuristic first-order optimizers are the best conceivable optimization procedures for deep learning. Therefore, pursuing research on novel optimization procedures is worthwhile, even though the coevolved state of the deep learning stack will initially be difficult to outperform. Moreover, deep learning currently requires many heuristics with trial and error to arrive at a well-performing system. Any step toward a comprehensive theory of deep learning will be invaluable in guiding the system and optimizer design process. In the meantime, deep learning models are becoming an integral part of classical algorithmic pipelines, where they may be used to perform data-driven modeling of otherwise heuristic procedures. There is much room to reconsider algorithms in computational sciences and to augment them with deep learning components.

BIBLIOGRAPHY

[1] Brandon Amos and J. Zico Kolter. OptNet: Differentiable Optimization as a Layer in Neural Networks. In: *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 2017.

[2] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018.

[3] Jimmy Ba and Rich Caruana. Do Deep Nets Really Need to be Deep? In: *Advances in Neural Information Processing Systems*. Vol. 27. 2014.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In: *Proceedings of the 3rd International Conference on Learning Representations, (ICLR)*. 2015.

[5] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. In: *Proceedings of the National Academy of Sciences* 116.32 (2019), pp. 15849–15854.

[6] Albert S. Berahas, Jorge Nocedal, and Martin Takac. A Multi-Batch L-BFGS Method for Machine Learning. In: *Advances in Neural Information Processing Systems*. Vol. 29. 2016.

[7] Christopher M. Bishop. Regularization and complexity control in feed-forward networks. In: *Proceedings of the International Conference on Artificial Neural Networks*. 1995, pp. 141–148.

[8] Jérôme Bolte and Edouard Pauwels. Curiosities and counterexamples in smooth convex optimization. In: *Mathematical Programming* (2021).

[9] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. In: *SIAM Review* 60.2 (2018), pp. 223–311.

[10] James Bradbury et al. JAX: composable transformations of Python+NumPy programs. 2018. URL: http://github.com/google/jax.

[11] Richard H. Byrd, Samantha L. Hansen, Jorge Nocedal, and Yoram Singer. A Stochastic Quasi-Newton Method for Large-Scale Optimization. In: *SIAM Journal on Optimization* 26.2 (2016), pp. 1008–1031.

[12] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations. In: *Advances in Neural Information Processing Systems*. Vol. 31. 2018.

[13] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In: *Proceedings of the 4th International Conference on Learning Representations (ICLR)*. 2016.

[14] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. In: *Neural Networks* 107 (2018), pp. 3–11.

[15] Haw-ren Fang and Dianne P. O'Leary. Modified Cholesky algorithms: a catalog with new approaches. In: *Mathematical Programming* 115.2 (2008), pp. 319–349.

[16] Thomas Frerix and Joan Bruna. Approximating Orthogonal Matrices with Effective Givens Factorization. In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 2019.

[17] Thomas Frerix, Dmitrii Kochkov, Jamie A. Smith, Daniel Cremers, Michael P. Brenner, and Stephan Hoyer. Variational Data Assimilation with a Learned Inverse Observation Operator. In: *Proceedings of the 38th International Conference on Machine Learning (ICML)*. 2021.

[18] Thomas Frerix, Thomas Möllenhoff, Michael Moeller, and Daniel Cremers. Proximal Backpropagation. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. 2018.

[19] Thomas Frerix, Matthias Nießner, and Daniel Cremers. Homogeneous Linear Inequality Constraints for Neural Network Activations. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2020.

[21] Tommaso Furlanello, Zachary Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born Again Neural Networks. In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018.

[22] Jonas Geiping, Micah Goldblum, Phillip E. Pope, Michael Moeller, and Tom Goldstein. Stochastic Training is Not Necessary for Generalization. In: *Proceedings of the 10th International Conference on Learning Representations (ICLR)*. 2022.

[23] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. In: *Optimization Methods and Software* 1.1 (1992), pp. 35–54.

[24] Andreas Griewank. Evaluating Derivatives. First Edition. Society for Industrial and Applied Mathematics, 2000.

[25] Philip Haeusser, Thomas Frerix, Alexander Mordvintsev, and Daniel Cremers. Associative Domain Adaptation. In: *Proceedings of the International Conference on Computer Vision (ICCV)*. 2017.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[27] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the Knowledge in a Neural Network. In: *NIPS Deep Learning and Representation Learning Workshop*. 2015.

[28] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-Excitation Networks. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

[29] Florian Jarre and Philippe L. Toint. Simple examples for the failure of Newton's method with line search for strictly convex minimization. In: *Mathematical Programming* 158.1 (2016), pp. 23–34.

[30] Tobias L. Jensen and Moritz Diehl. An Approach for Analyzing the Global Rate of Convergence of Quasi-Newton and Truncated-Newton Methods. In: *Journal of Optimization Theory and Applications* 172.1 (2017), pp. 206–221.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In: *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. 2015.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing Systems*. Vol. 25. 2012.

[33] Steve Lawrence, C. Lee Giles, and Ah Tsoi. Lessons in Neural Network Training: Overfitting May be Harder than Expected. In: *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*. 1997.

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. In: *Nature* 521.7553 (2015), pp. 436–444.

[35] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object Recognition with Gradient-Based Learning. In: *Shape, Contour and Grouping in Computer Vision*. 1999, pp. 319–345.

[36] Siyuan Ma, Raef Bassily, and Mikhail Belkin. The Power of Interpolation: Understanding the Effectiveness of SGD in Modern Over-parametrized Learning. In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018.

[37] James Martens. Second-order Optimization for Neural Networks. PhD thesis. University of Toronto, 2016.

[38] Martín Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. URL: https://www.tensorflow.org/.

[39] Uwe Naumann. Optimal Jacobian accumulation is NP-complete. In: *Mathematical Programming* 112.2 (2008), pp. 427–441.

[40] Jorge Nocedal and Stephen J. Wright. Numerical Optimization. Second Edition. Springer, 2006.

[41] Samet Oymak and Mahdi Soltanolkotabi. Overparameterized Nonlinear Learning: Gradient Descent Takes the Shortest Path? In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 2019.

[42] Adam Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019.

[43] Howard H. Rosenbrock. An Automatic Method for Finding the Greatest or Least Value of a Function. In: *The Computer Journal* 3.3 (1960), pp. 175–184.

[44] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. In: *Nature* 323.6088 (1986), pp. 533–536.

[45] Olga Russakovsky et al. ImageNet Large Scale Visual Recognition Challenge. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252.

[46] Jürgen Schmidhuber. Deep learning in neural networks: An overview. In: *Neural Networks* 61 (2015), pp. 85–117.

[47] Jascha Sohl-Dickstein, Ben Poole, and Surya Ganguli. Fast large-scale optimization by unifying stochastic gradient and quasi-Newton methods. In: *Proceedings of the 31st International Conference on Machine Learning (ICML)*. 2014.

[48] Christian Szegedy et al. Going deeper with convolutions. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.

[49] Lloyd N. Trefethen and David Bau. Numerical Linear Algebra. Society for Industrial and Applied Mathematics, 1997.

[50] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

[51] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In: *Proceedings of the 13th European Conference on Computer Vision (ECCV)*. 2014.

[52] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In: *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. 2017.

PUBLICATIONS

Proximal Backpropagation

Thomas Frerix*, Thomas Möllenhoff*, Michael Moeller*, and Daniel Cremers. Proximal Backpropagation. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. 2018. (* denotes equal contribution)

PROXIMAL BACKPROPAGATION

Thomas Frerix[1*], **Thomas Möllenhoff**[1*], **Michael Moeller**[2*], **Daniel Cremers**[1]

thomas.frerix@tum.de
thomas.moellenhoff@tum.de
michael.moeller@uni-siegen.de
cremers@tum.de

[1] Technical University of Munich
[2] University of Siegen

ABSTRACT

We propose proximal backpropagation (ProxProp) as a novel algorithm that takes *implicit* instead of explicit gradient steps to update the network parameters during neural network training. Our algorithm is motivated by the step size limitation of explicit gradient descent, which poses an impediment for optimization. ProxProp is developed from a general point of view on the backpropagation algorithm, currently the most common technique to train neural networks via stochastic gradient descent and variants thereof. Specifically, we show that backpropagation of a prediction error is equivalent to sequential gradient descent steps on a quadratic penalty energy, which comprises the network activations as variables of the optimization. We further analyze theoretical properties of ProxProp and in particular prove that the algorithm yields a descent direction in parameter space and can therefore be combined with a wide variety of convergent algorithms. Finally, we devise an efficient numerical implementation that integrates well with popular deep learning frameworks. We conclude by demonstrating promising numerical results and show that ProxProp can be effectively combined with common first order optimizers such as Adam.

1 INTRODUCTION

In recent years neural networks have gained considerable attention in solving difficult correlation tasks such as classification in computer vision (Krizhevsky et al., 2012) or sequence learning (Sutskever et al., 2014) and as building blocks of larger learning systems (Silver et al., 2016). Training neural networks is accomplished by optimizing a nonconvex, possibly nonsmooth, nested function of the network parameters. Since the introduction of stochastic gradient descent (SGD) (Robbins & Monro, 1951; Bottou, 1991), several more sophisticated optimization methods have been studied. One such class is that of quasi-Newton methods, as for example the comparison of L-BFGS with SGD in (Le et al., 2011), Hessian-free approaches (Martens, 2010), and the Sum of Functions Optimizer in (Sohl-Dickstein et al., 2013). Several works consider specific properties of energy landscapes of deep learning models such as frequent saddle points (Dauphin et al., 2014) and well-generalizable local optima (Chaudhari et al., 2017a). Among the most popular optimization methods in currently used deep learning frameworks are momentum based improvements of classical SGD, notably Nesterov's Accelerated Gradient (Nesterov, 1983; Sutskever et al., 2013), and the Adam optimizer (Kingma & Ba, 2015), which uses estimates of first and second order moments of the gradients for parameter updates.

Nevertheless, the optimization of these models remains challenging, as learning with SGD and its variants requires careful weight initialization and a sufficiently small learning rate in order to yield a stable and convergent algorithm. Moreover, SGD often has difficulties in propagating a learning signal deeply into a network, commonly referred to as the vanishing gradient problem (Hochreiter et al., 2001).

[*] contributed equally

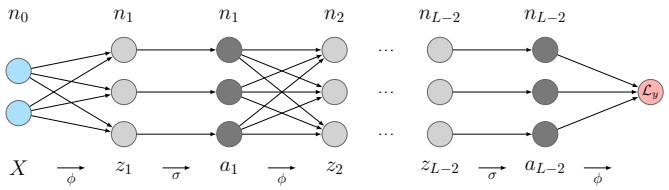


Figure 1: Notation overview. For an L-layer feed-forward network we denote the explicit layer-wise activation variables as z_l and a_l. The transfer functions are denoted as ϕ and σ. Layer l is of size n_l.

Training neural networks can be formulated as a constrained optimization problem by explicitly introducing the network activations as variables of the optimization, which are coupled via layer-wise constraints to enforce a feasible network configuration. The authors of (Carreira-Perpiñán & Wang, 2014) have tackled this problem with a quadratic penalty approach, the method of auxiliary coordinates (MAC). Closely related, (Taylor et al., 2016) introduce additional auxiliary variables to further split linear and nonlinear transfer between layers and propose a primal dual algorithm for optimization. From a different perspective, (LeCun, 1988) takes a Lagrangian approach to formulate the constrained optimization problem.

In this work, we start from a constrained optimization point of view on the classical backpropagation algorithm. We show that backpropagation can be interpreted as a method alternating between two steps. First, a forward pass of the data with the current network weights. Secondly, an ordered sequence of gradient descent steps on a quadratic penalty energy.

Using this point of view, instead of taking explicit gradient steps to update the network parameters associated with the *linear* transfer functions, we propose to use implicit gradient steps (also known as proximal steps, for the definition see (6)). We prove that such a model yields a descent direction and can therefore be used in a wide variety of (provably convergent) algorithms under weak assumptions. Since an exact proximal step may be costly, we further consider a matrix-free conjugate gradient (CG) approximation, which can directly utilize the efficient pre-implemented forward and backward operations of any deep learning framework. We prove that this approximation still yields a descent direction and demonstrate the effectiveness of the proposed approach in PyTorch.

2 Model and notation

We propose a method to train a general L-layer neural network of the functional form

$$J(\boldsymbol{\theta}; X, y) = \mathcal{L}_y(\phi(\theta_{L-1}, \sigma(\phi(\cdots, \sigma(\phi(\theta_1, X))\cdots)). \tag{1}$$

Here, $J(\boldsymbol{\theta}; X, y)$ denotes the training loss as a function of the network parameters $\boldsymbol{\theta}$, the input data X and the training targets y. As the final loss function $\mathcal{L}_y$ we choose the softmax cross-entropy for our classification experiments. ϕ is a linear transfer function and σ an elementwise nonlinear transfer function. As an example, for fully-connected neural networks $\theta = (W, b)$ and $\phi(\theta, a) = Wa + b\mathbf{1}$.

While we assume the nonlinearities σ to be continuously differentiable functions for analysis purposes, our numerical experiments indicate that the proposed scheme extends to rectified linear units (ReLU), $\sigma(x) = \max(0, x)$. Formally, the functions σ and ϕ map between spaces of different dimensions depending on the layer. However, to keep the presentation clean, we do not state this dependence explicitly. Figure 1 illustrates our notation for the fully-connected network architecture.

Throughout this paper, we denote the Euclidean norm for vectors and the Frobenius norm for matrices by $||\cdot||$, induced by an inner product $\langle\cdot,\cdot\rangle$. We use the gradient symbol ∇ to denote the transpose of the Jacobian matrix, such that the chain rule applies in the form "inner derivative times outer derivative". For all computations involving matrix-valued functions and their gradient/Jacobian, we uniquely identify all involved quantities with their vectorized form by flattening matrices in a column-first order. Furthermore, we denote by A^* the adjoint of a linear operator A.

Algorithm 1 - Penalty formulation of BackProp

Input: Current parameters $\boldsymbol{\theta}^k$.

// Forward pass.
for $l = 1$ **to** $L-2$ **do**
  $z_l^k = \phi(\theta_l^k, a_{l-1}^k)$, *// $a_0 = X$.*
  $a_l^k = \sigma(z_l^k)$.
end for

// Perform minimization steps on (3).
ⓐ grad. step on E wrt. (θ_{L-1}, a_{L-2})
for $l = L-2$ **to** 1 **do**
  ⓑ grad. step on E wrt. z_l and a_{l-1},
  ⓒ grad. step on E wrt. θ_l.
end for

Output: New parameters $\boldsymbol{\theta}^{k+1}$.

Algorithm 2 - ProxProp

Input: Current parameters $\boldsymbol{\theta}^k$.

// Forward pass.
for $l = 1$ **to** $L-2$ **do**
  $z_l^k = \phi(\theta_l^k, a_{l-1}^k)$, *// $a_0 = X$.*
  $a_l^k = \sigma(z_l^k)$.
end for

// Perform minimization steps on (3).
ⓐ grad. step on E wrt. (θ_{L-1}, a_{L-2}), Eqs. 8, 12.
for $l = L-2$ **to** 1 **do**
  ⓑ grad. step on E wrt. z_l and a_{l-1}, Eqs. 9, 10.
  ⓒ prox step on E wrt. θ_l, Eq. 11.
end for

Output: New parameters $\boldsymbol{\theta}^{k+1}$.

3 Penalty formulation of backpropagation

The gradient descent iteration on a nested function $J(\boldsymbol{\theta}; X, y)$,

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \tau \nabla J(\boldsymbol{\theta}^k; X, y), \tag{2}$$

is commonly implemented using the backpropagation algorithm (Rumelhart et al., 1986). As the basis for our proposed optimization method, we derive a connection between the classical backpropagation algorithm and quadratic penalty functions of the form

$$E(\boldsymbol{\theta}, \boldsymbol{a}, \boldsymbol{z}) = \mathcal{L}_y(\phi(\theta_{L-1}, a_{L-2})) + \sum_{l=1}^{L-2} \frac{\gamma}{2} \|\sigma(z_l) - a_l\|^2 + \frac{\rho}{2} \|\phi(\theta_l, a_{l-1}) - z_l\|^2. \tag{3}$$

The approach of (Carreira-Perpiñán & Wang, 2014) is based on the minimization of (3), as under mild conditions the limit $\rho, \gamma \to \infty$ leads to the convergence of the sequence of minimizers of E to the minimizer of J (Nocedal & Wright, 2006, Theorem 17.1). In contrast to (Carreira-Perpiñán & Wang, 2014) we do not optimize (3), but rather use a connection of (3) to the classical backpropagation algorithm to motivate a semi-implicit optimization algorithm for the original cost function J.

Indeed, the iteration shown in Algorithm 1 of forward passes followed by a sequential gradient descent on the penalty function E is equivalent to the classical gradient descent iteration.

Proposition 1. *Let $\mathcal{L}_y$, ϕ and σ be continuously differentiable. For $\rho = \gamma = 1/\tau$ and $\boldsymbol{\theta}^k$ as the input to Algorithm 1, its output $\boldsymbol{\theta}^{k+1}$ satisfies* (2)*, i.e., Algorithm 1 computes one gradient descent iteration on J.*

Proof. For this and all further proofs we refer to Appendix A. □

4 Proximal backpropagation

The interpretation of Proposition 1 leads to the natural idea of replacing the explicit gradient steps ⓐ, ⓑ and ⓒ in Algorithm 1 with other – possibly more powerful – minimization steps. We propose Proximal Backpropagation (ProxProp) as one such algorithm that takes *implicit* instead of *explicit* gradient steps to update the network parameters θ in step ⓒ. This algorithm is motivated by the step size restriction of gradient descent.

4.1 Gradient descent and proximal mappings

Explicit gradient steps pose severe restrictions on the allowed step size τ: Even for a convex, twice continuously differentiable, $\mathscr{L}$-smooth function $f : \mathbb{R}^n \to \mathbb{R}$, the convergence of the gradient

descent algorithm can only be guaranteed for step sizes $0 < \tau < 2/\mathscr{L}$. The Lipschitz constant $\mathscr{L}$ of the gradient ∇f is in this case equal to the largest eigenvalue of the Hessian H. With the interpretation of backpropagation as in Proposition 1, gradient steps are taken on quadratic functions. As an example for the first layer,

$$f(\theta) = \frac{1}{2}||\theta X - z_1||^2 \,. \tag{4}$$

In this case the Hessian is $H = XX^\top$, which is often ill-conditioned. For the CIFAR-10 dataset the largest eigenvalue is $6.7 \cdot 10^6$, which is seven orders of magnitude larger than the smallest eigenvalue. Similar problems also arise in other layers where poorly conditioned matrices a_l pose limitations for guaranteeing the energy E to decrease.

The proximal mapping (Moreau, 1965) of a function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as:

$$\text{prox}_{\tau f}(y) := \underset{x \in \mathbb{R}^n}{\text{argmin}} \; f(x) + \frac{1}{2\tau}||x - y||^2. \tag{5}$$

By rearranging the optimality conditions to (5) and taking $y = x^k$, it can be interpreted as an *implicit* gradient step evaluated at the new point x^{k+1} (assuming differentiability of f):

$$x^{k+1} := \underset{x \in \mathbb{R}^n}{\text{argmin}} \; f(x) + \frac{1}{2\tau}||x - x^k||^2 = x^k - \tau \nabla f(x^{k+1}). \tag{6}$$

The iterative algorithm (6) is known as the proximal point algorithm (Martinet, 1970). In contrast to explicit gradient descent this algorithm is *unconditionally stable*, i.e. the update scheme (6) monotonically decreases f for any $\tau > 0$, since it holds by definition of the minimizer x^{k+1} that $f(x^{k+1}) + \frac{1}{2\tau}||x^{k+1} - x^k||^2 \leq f(x^k)$.

Thus proximal mappings yield unconditionally stable subproblems in the following sense: The update in θ_l provably decreases the penalty energy $E(\boldsymbol{\theta}, \boldsymbol{a}^k, \boldsymbol{z}^k)$ from (3) for fixed activations $(\boldsymbol{a}^k, \boldsymbol{z}^k)$ for any choice of step size. This motivates us to use proximal steps as depicted in Algorithm 2.

4.2 ProxProp

We propose to replace explicit gradient steps with proximal steps to update the network parameters of the linear transfer function. More precisely, after the forward pass

$$\begin{aligned} z_l^k &= \phi(\theta_l^k, a_{l-1}^k), \\ a_l^k &= \sigma(z_l^k), \end{aligned} \tag{7}$$

we keep the explicit gradient update equations for z_l and a_l. The last layer update is

$$a_{L-2}^{k+1/2} = a_{L-2}^k - \tau \nabla_{a_{L-2}} \mathcal{L}_y(\phi(\theta_{L-1}, a_{L-2})), \tag{8}$$

and for all other layers,

$$z_l^{k+1/2} = z_l^k - \sigma'(z_l^k)(\sigma(z_l^k) - a_l^{k+1/2}), \tag{9}$$

$$a_{l-1}^{k+1/2} = a_{l-1}^k - \nabla \left(\frac{1}{2}\|\phi(\theta_l, \cdot) - z_l^{k+1/2}\|^2 \right)(a_{l-1}^k), \tag{10}$$

where we use $a_l^{k+1/2}$ and $z_l^{k+1/2}$ to denote the updated variables before the forward pass of the next iteration and multiplication in (9) is componentwise. However, instead of taking explicit gradient steps to update the linear transfer parameters θ_l, we take proximal steps

$$\theta_l^{k+1} = \underset{\theta}{\text{argmin}} \; \frac{1}{2}||\phi(\theta, a_{l-1}^k) - z_l^{k+1/2}||^2 + \frac{1}{2\tau_\theta}||\theta - \theta_l^k||^2. \tag{11}$$

This update can be computed in closed form as it amounts to a linear solve (for details see Appendix B). While in principle one can take a proximal step on the final loss $\mathcal{L}_y$, for efficiency reasons we choose an explicit gradient step, as the proximal step does not have a closed form solution in many scenarios (e.g. the softmax cross-entropy loss in classification problems). Specifically,

$$\theta_{L-1}^{k+1} = \theta_{L-1}^k - \tau \nabla_{\theta_{L-1}} \mathcal{L}_y(\phi(\theta_{L-1}^k, a_{L-2}^k)). \tag{12}$$

Note that we have eliminated the step sizes in the updates for z_l and a_{l-1} in (9) and (10), as such updates correspond to the choice of $\rho = \gamma = \frac{1}{\tau}$ in the penalty function (3) and are natural in the sense of Proposition 1. For the proximal steps in the parameters θ in (11) we have introduced a step size τ_θ which – as we will see in Proposition 2 below – changes the descent metric opposed to τ which rather rescales the magnitude of the update.

We refer to one sweep of updates according to equations (7) - (12) as *ProxProp*, as it closely resembles the classical backpropagation (BackProp), but replaces the parameter update by a proximal mapping instead of an explicit gradient descent step. In the following subsection we analyze the convergence properties of ProxProp more closely.

4.2.1 CONVERGENCE OF PROXPROP

ProxProp inherits all convergence-relevant properties from the classical backpropagation algorithm, despite replacing explicit gradient steps with proximal steps: It minimizes the original network energy $J(\boldsymbol{\theta}; X, y)$ as its fixed-points are stationary points of $J(\boldsymbol{\theta}; X, y)$, and the update direction $\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k$ is a descent direction such that it converges when combined with a suitable optimizer. In particular, it is straight forward to combine ProxProp with popular optimizers such as Nesterov's accelerated gradient descent (Nesterov, 1983) or Adam (Kingma & Ba, 2015).

In the following, we give a detailed analysis of these properties.

Proposition 2. *For $l = 1, \dots, L-2$, the update direction $\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k$ computed by ProxProp meets*

$$\theta_l^{k+1} - \theta_l^k = -\tau \left(\frac{1}{\tau_\theta} I + (\nabla \phi(\cdot, a_{l-1}^k))(\nabla \phi(\cdot, a_{l-1}^k))^* \right)^{-1} \nabla_{\theta_l} J(\boldsymbol{\theta}^k; X, y). \tag{13}$$

In other words, ProxProp multiplies the gradient $\nabla_{\theta_l} J$ with the inverse of the positive definite, symmetric matrix

$$M_l^k := \frac{1}{\tau_\theta} I + (\nabla \phi(\cdot, a_{l-1}^k))(\nabla \phi(\cdot, a_{l-1}^k))^*, \tag{14}$$

which depends on the activations a_{l-1}^k of the forward pass. Proposition 2 has some important implications:

Proposition 3. *For any choice of $\tau > 0$ and $\tau_\theta > 0$, fixed points $\boldsymbol{\theta}^*$ of ProxProp are stationary points of the original energy $J(\boldsymbol{\theta}; X, y)$.*

Moreover, we can conclude convergence in the following sense.

Proposition 4. *The ProxProp direction $\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k$ is a descent direction. Moreover, under the (weak) assumption that the activations a_l^k remain bounded, the angle α^k between $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k; X, y)$ and $\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k$ remains uniformly bounded away from $\pi/2$, i.e.*

$$\cos(\alpha^k) > c \geq 0, \qquad \forall k \geq 0, \tag{15}$$

for some constant c.

Proposition 4 immediately implies convergence of a whole class of algorithms that depend only on a provided descent direction. We refer the reader to (Nocedal & Wright, 2006, Chapter 3.2) for examples and more details.

Furthermore, Proposition 4 states convergence for any minimization scheme in step ⓒ of Algorithm 2 that induces a descent direction in parameter space and thus provides the theoretical basis for a wide range of neural network optimization algorithms.

Considering the advantages of proximal steps over gradient steps, it is tempting to also update the auxiliary variables a and z in an implicit fashion. This corresponds to a proximal step in ⓑ of Algorithm 2. However, one cannot expect an analogue version of Proposition 3 to hold anymore. For example, if the update of a_{L-2} in (8) is replaced by a proximal step, the propagated error does not correspond to the gradient of the loss function $\mathcal{L}_y$, but to the gradient of its Moreau envelope. Consequently, one would then minimize a different energy. While in principle this could result in an optimization algorithm with, for example, favorable generalization properties, we focus on minimizing the original network energy in this work and therefore do not further pursue the idea of implicit steps on a and z.

4.2.2 INEXACT SOLUTION OF PROXIMAL STEPS

As we can see in Proposition 2, the ProxProp updates differ from vanilla gradient descent by the variable metric induced by the matrices $(M_l^k)^{-1}$ with M_l^k defined in (14). Computing the ProxProp update direction $v_l^k := \frac{1}{\tau}(\theta_l^{k+1} - \theta_l^k)$ therefore reduces to solving the linear equation

$$M_l^k v_l^k = -\nabla_{\theta_l} J(\boldsymbol{\theta}^k; X, y), \tag{16}$$

which requires an efficient implementation. We propose to use a conjugate gradient (CG) method, not only because it is one of the most efficient methods for iteratively solving linear systems in general, but also because it can be implemented *matrix-free*: It merely requires the application of the linear operator M_l^k which consists of the identity and an application of $(\nabla\phi(\cdot, a_{l-1}^k))(\nabla\phi(\cdot, a_{l-1}^k))^*$. The latter, however, is preimplemented for many linear transfer functions ϕ in common deep learning frameworks, because $\phi(x, a_{l-1}^k) = (\nabla\phi(\cdot, a_{l-1}^k))^*(x)$ is nothing but a forward-pass in ϕ, and $\phi^*(z, a_{l-1}^k) = (\nabla\phi(\cdot, a_{l-1}^k))(z)$ provides the gradient with respect to the parameters θ if z is the backpropagated gradient up to that layer. Therefore, a CG solver is straight-forward to implement in any deep learning framework using the existing, highly efficient and high level implementations of ϕ and ϕ^*. For a fully connected network ϕ is a matrix multiplication and for a convolutional network the convolution operation.

As we will analyze in more detail in Section 5.1, we approximate the solution to (16) with a few CG iterations, as the advantage of highly precise solutions does not justify the additional computational effort in obtaining them. Using any number of iterations provably does not harm the convergence properties of ProxProp:

Proposition 5. *The direction $\tilde{v}_l^k$ one obtains from approximating the solution v_l^k of* (16) *with the CG method remains a descent direction for any number of iterations.*

4.2.3 CONVERGENCE IN THE STOCHASTIC SETTING

While the above analysis considers only the full batch setting, we remark that convergence of ProxProp can also be guaranteed in the stochastic setting under mild assumptions. Assuming that the activations a_l^k remain bounded (as in Proposition 4), the eigenvalues of $(M_l^k)^{-1}$ are uniformly contained in the interval $[\lambda, \tau_\theta]$ for some fixed $\lambda > 0$. Therefore, our ProxProp updates fulfill Assumption 4.3 in (Bottou et al., 2016), presuming the classic stochastic gradient fulfills them. This guarantees convergence of stochastic ProxProp updates in the sense of (Bottou et al., 2016, Theorem 4.9), i.e. for a suitable sequence of diminishing step sizes.

5 NUMERICAL EVALUATION

ProxProp generally fits well with the API provided by modern deep learning frameworks, since it can be implemented as a network layer with a custom backward pass for the proximal mapping. We chose PyTorch for our implementation[1]. In particular, our implementation can use the API's GPU compute capabilities; all numerical experiments reported below were conducted on an NVIDIA Titan X GPU. To directly compare the algorithms, we used our own layer for either proximal or gradient update steps (cf. step ⓒ in Algorithms 1 and 2). A ProxProp layer can be seamlessly integrated in a larger network architecture, also with other parametrized layers such as BatchNormalization.

5.1 EXACT AND APPROXIMATE SOLUTIONS TO PROXIMAL STEPS

We study the behavior of ProxProp in comparison to classical BackProp for a supervised visual learning problem on the CIFAR-10 dataset. We train a fully connected network with architecture $3072 - 4000 - 1000 - 4000 - 10$ and ReLU nonlinearities. As the loss function, we chose the cross-entropy between the probability distribution obtained by a softmax nonlinearity and the ground-truth labels. We used a subset of 45000 images for training while keeping 5000 images as a validation set. We initialized the parameters θ_l uniformly in $[-1/\sqrt{n_{l-1}}, 1/\sqrt{n_{l-1}}]$, the default initialization of PyTorch.

[1] `https://github.com/tfrerix/proxprop`

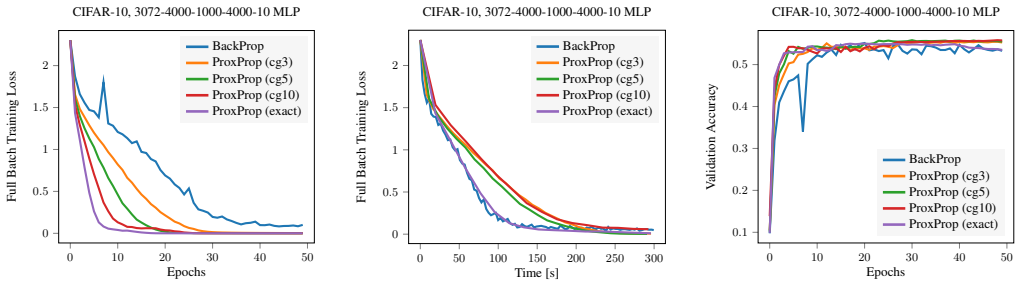


Figure 2: Exact and inexact solvers for ProxProp compared with BackProp. **Left:** A more precise solution of the proximal subproblem leads to overall faster convergence, while even a very inexact solution (only 3 CG iterations) already outperforms classical backpropagation. **Center & Right:** While the run time is comparable between the methods, the proposed ProxProp updates have better generalization performance ($\approx 54\%$ for BackProp and $\approx 56\%$ for ours on the test set).

Figure 2 shows the decay of the full batch training loss over epochs (left) and training time (middle) for a Nesterov momentum[2] based optimizer using a momentum of $\mu = 0.95$ and minibatches of size 500. We used $\tau_\theta = 0.05$ for the ProxProp variants along with $\tau = 1$. For BackProp we chose $\tau = 0.05$ as the optimal value we found in a grid search.

As we can see in Figure 2, using implicit steps indeed improves the optimization progress per epoch. Thanks to powerful linear algebra methods on the GPU, the exact ProxProp solution is competitive with BackProp even in terms of runtime.

The advantage of the CG-based approximations, however, is that they generalize to arbitrary linear transfer functions in a matrix-free manner, i.e. they are independent of whether the matrices M_l^k can be formed efficiently. Moreover, the validation accuracies (right plot in Figure 2) suggest that these approximations have generalization advantages in comparison to BackProp as well as the exact ProxProp method. Finally, we found the exact solution to be significantly more sensitive to changes of τ_θ than its CG-based approximations. We therefore focus on the CG-based variants of ProxProp in the following. In particular, we can eliminate the hyperparameter τ_θ and consistently chose $\tau_\theta = 1$ for the rest of this paper, while one can in principle perform a hyperparameter search just as for the learning rate τ. Consequently, there are no additional parameters compared with BackProp.

5.2 Stability for larger step sizes

We compare the behavior of ProxProp and BackProp for different step sizes. Table 1 shows the final full batch training loss after 50 epochs with various τ. The ProxProp based approaches remain stable over a significantly larger range of τ. Even more importantly, deviating from the optimal step size τ by one order of magnitude resulted in a divergent algorithm for classical BackProp, but still provides reasonable training results for ProxProp with 3 or 5 CG iterations. These results are in accordance with our motivation developed in Section 4.1. From a practical point of view, this eases hyperparameter search over τ.

τ	50	10	5	1	0.5	0.1	0.05	$5 \cdot 10^{-3}$	$5 \cdot 10^{-4}$
BackProp	–	–	–	–	–	0.524	0.091	0.637	1.531
ProxProp (cg1)	77.9	0.079	0.145	0.667	0.991	1.481	1.593	1.881	2.184
ProxProp (cg3)	94.7	0.644	0.031	$2 \cdot 10^{-3}$	0.012	1.029	1.334	1.814	2.175
ProxProp (cg5)	66.5	0.190	0.027	$3 \cdot 10^{-4}$	$2 \cdot 10^{-3}$	0.399	1.049	1.765	2.175

Table 1: Full batch loss for conjugate gradient versions of ProxProp and BackProp after training for 50 epochs, where "–" indicates that the algorithm diverged to `NaN`. The implicit ProxProp algorithms remain stable for a significantly wider range of step sizes.

[2]PyTorch's Nesterov SGD for direction $\boldsymbol{d}(\boldsymbol{\theta}^k)$: $\boldsymbol{m}^{k+1} = \mu \boldsymbol{m}^k + \boldsymbol{d}(\boldsymbol{\theta}^k)$, $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \tau(\mu \boldsymbol{m}^{k+1} + \boldsymbol{d}(\boldsymbol{\theta}^k))$.

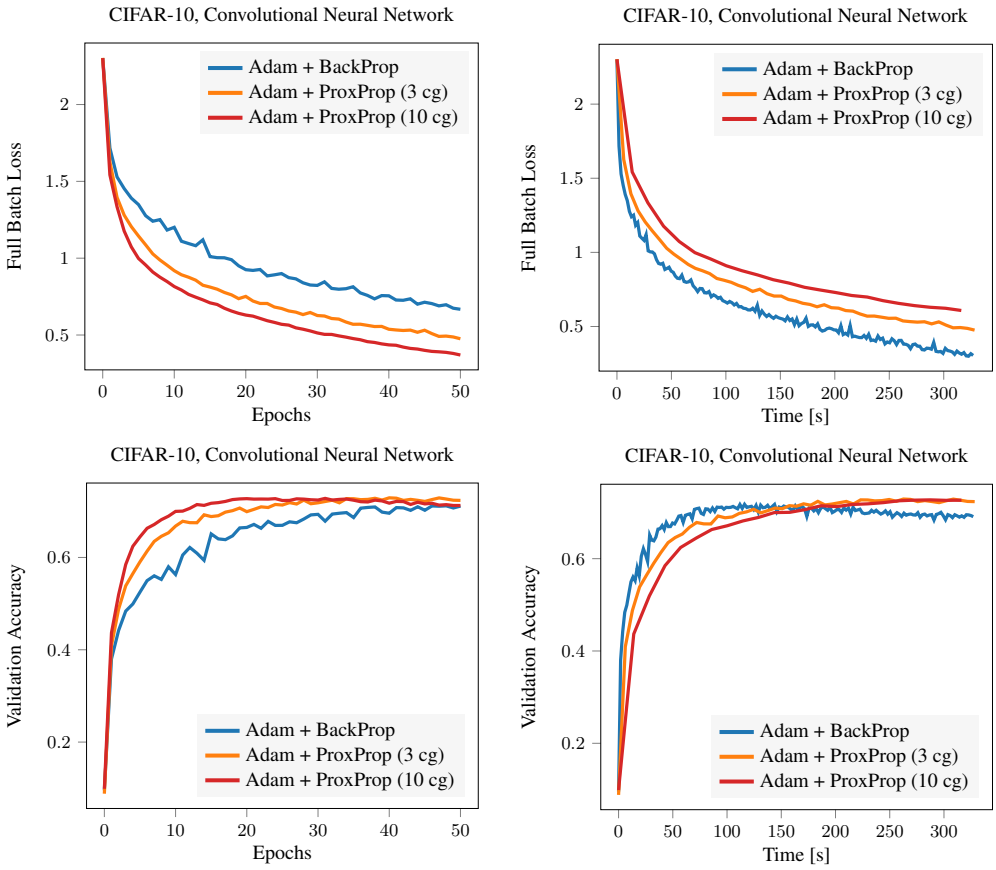


Figure 3: ProxProp as a first-order oracle in combination with the Adam optimizer. The proposed method leads to faster decrease of the full batch loss in epochs and to an overall higher accuracy on the validation set. The plots on the right hand side show data for a fixed runtime, which corresponds to a varying number of epochs for the different optimizers.

5.3 ProxProp as a first-order oracle

We show that ProxProp can be used as a gradient oracle for first-order optimization algorithms. In this section, we consider Adam (Kingma & Ba, 2015). Furthermore, to demonstrate our algorithm on a generic architecture with layers commonly used in practice, we trained on a convolutional neural network of the form:

$$\mathrm{Conv}[16 \times 32 \times 32] \to \mathrm{ReLU} \to \mathrm{Pool}[16 \times 16 \times 16] \to \mathrm{Conv}[20 \times 16 \times 16] \to \mathrm{ReLU}$$
$$\to \mathrm{Pool}[20 \times 8 \times 8] \to \mathrm{Conv}[20 \times 8 \times 8] \to \mathrm{ReLU} \to \mathrm{Pool}[20 \times 4 \times 4] \to \mathrm{FC} + \mathrm{Softmax}[10 \times 1 \times 1]$$

Here, the first dimension denotes the respective number of filters with kernel size 5×5 and max pooling downsamples its input by a factor of two. We set the step size $\tau = 10^{-3}$ for both BackProp and ProxProp.

The results are shown in Fig. 3. Using parameter update directions induced by ProxProp within Adam leads to a significantly faster decrease of the full batch training loss in epochs. While the running time is higher than the highly optimized backpropagation method, we expect that it can be improved through further engineering efforts. We deduce from Fig. 3 that the best validation accuracy (72.9%) of the proposed method is higher than the one obtained with classical backpropagation (71.7%). Such a positive effect of proximal smoothing on the generalization capabilities of deep networks is consistent with the observations of Chaudhari et al. (2017b). Finally, the accuracies on the test set after 50 epochs are 70.7% for ProxProp and 69.6% for BackProp which suggests that the proposed algorithm can lead to better generalization.

6 CONCLUSION

We have proposed proximal backpropagation (ProxProp) as an effective method for training neural networks. To this end, we first showed the equivalence of the classical backpropagation algorithm with an algorithm that alternates between sequential gradient steps on a quadratic penalty function and forward passes through the network. Subsequently, we developed a generalization of BackProp, which replaces explicit gradient steps with implicit (proximal) steps, and proved that such a scheme yields a descent direction, even if the implicit steps are approximated by conjugate gradient iterations. Our numerical analysis demonstrates that ProxProp is stable across various choices of step sizes and shows promising results when compared with common stochastic gradient descent optimizers.

We believe that the interpretation of error backpropagation as the alternation between forward passes and sequential minimization steps on a penalty functional provides a theoretical basis for the development of further learning algorithms.

REFERENCES

Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8), 1991.

Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

Miguel Á. Carreira-Perpiñán and Weiran Wang. Distributed optimization of deeply nested systems. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics, AISTATS*, 2014.

Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-SGD: Biasing gradient descent into wide valleys. In *Proceedings of the 5th International Conference on Learning Representations, ICLR*, 2017a.

Pratik Chaudhari, Adam Oberman, Stanley Osher, Stefano Soatto, and Guillame Carlier. Deep Relaxation: partial differential equations for optimizing deep neural networks. *arXiv preprint arXiv:1704.04932*, 2017b.

Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Proceedings of the 27th International Conference on Neural Information Processing Systems, NIPS*, 2014.

Sepp Hochreiter, Yoshua Bengio, and Paolo Frasconi. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In *Field Guide to Dynamical Recurrent Networks*. IEEE Press, 2001.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR*, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference of Neural Information Processing Systems, NIPS*, 2012.

Quoc V Le, Adam Coates, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of The 28th International Conference on Machine Learning, ICML*, 2011.

Yann LeCun. A theoretical framework for back-propagation. In *Proceedings of the 1988 Connectionist Models Summer School*, pp. 21–28, 1988.

James Martens. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning, ICML*, 2010.

Bernard Martinet. Régularisation d'inéquations variationnelles par approximations successives. *Rev. Francaise Inf. Rech. Oper.*, pp. 154–159, 1970.

Jean-Jacques Moreau. Proximité et dualité dans un espace hilbertien. *Bulletin de la Société mathématique de France*, 93:273–299, 1965.

Yurii Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2006.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Jascha Sohl-Dickstein, Ben Poole, and Surya Ganguli. Fast large-scale optimization by unifying stochastic gradient and quasi-newton methods. In *Proceedings of The 31st International Conference on Machine Learning, ICML*, 2013.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning, ICML*, 2013.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference of Neural Information Processing Systems, NIPS*, 2014.

Gavin Taylor, Ryan Burmeister, Zheng Xu, Bharat Singh, Ankit Patel, and Tom Goldstein. Training neural networks without gradients: A scalable ADMM approach. In *Proceedings of the 33rd International Conference on Machine Learning, ICML*, 2016.

APPENDIX

A THEORETICAL RESULTS

Proof of Proposition 1. We first take a gradient step on

$$E(\boldsymbol{\theta}, \boldsymbol{a}, \boldsymbol{z}) = \mathcal{L}_y(\phi(\theta_{L-1}, a_{L-2})) + \sum_{l=1}^{L-2} \frac{\gamma}{2}\|\sigma(z_l) - a_l\|^2 + \frac{\rho}{2}\|\phi(\theta_l, a_{l-1}) - z_l\|^2, \tag{17}$$

with respect to (θ_{L-1}, a_{L-2}). The gradient step with respect to θ_{L-1} is the same as in the gradient descent update,

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \tau \nabla J(\boldsymbol{\theta}^k; X, y), \tag{18}$$

since J depends on θ_{L-1} only via $\mathcal{L}_y \circ \phi$.

The gradient descent step on a_{L-2} in ⓐ yields

$$a_{L-2}^{k+1/2} = a_{L-2}^k - \tau \nabla_a \phi(\theta_{L-1}^k, a_{L-2}^k) \cdot \nabla_\phi \mathcal{L}_y(\phi(\theta_{L-1}^k, a_{L-2}^k)), \tag{19}$$

where we use $a_{L-2}^{k+1/2}$ to denote the updated variable a_{L-2} before the forward pass of the next iteration. To keep the presentation as clear as possible we slightly abused the notation of a right multiplication with $\nabla_a \phi(\theta_{L-1}^k, a_{L-2}^k)$: While this notation is exact in the case of fully connected layers, it represents the application of the corresponding linear operator in the more general case, e.g. for convolutions.

For all layers $l \leq L-2$ note that due to the forward pass in Algorithm 1 we have

$$\sigma(z_l^k) = a_l^k, \quad \phi(\theta_l^k, a_{l-1}^k) = z_l^k \tag{20}$$

and we therefore get the following update equations in the gradient step ⓑ

$$z_l^{k+1/2} = z_l^k - \tau\gamma \nabla\sigma(z_l^k)\left(\sigma(z_l^k) - a_l^{k+1/2}\right) = z_l^k - \nabla\sigma(z_l^k)\left(a_l^k - a_l^{k+1/2}\right), \tag{21}$$

and in the gradient step ⓒ w.r.t. a_{l-1},

$$\begin{aligned} a_{l-1}^{k+1/2} &= a_{l-1}^k - \tau\rho \nabla_a \phi(\theta_l^k, a_{l-1}^k) \cdot \left(\phi(\theta_l^k, a_{l-1}^k) - z_l^{k+1/2}\right) \\ &= a_{l-1}^k - \nabla_a \phi(\theta_l^k, a_{l-1}^k) \cdot \left(z_l^k - z_l^{k+1/2}\right). \end{aligned} \tag{22}$$

Equations (21) and (22) can be combined to obtain:

$$z_l^k - z_l^{k+1/2} = \nabla\sigma(z_l^k)\nabla_a \phi(\theta_{l+1}^k, a_l^k) \cdot \left(z_{l+1}^k - z_{l+1}^{k+1/2}\right). \tag{23}$$

The above formula allows us to backtrack the differences of the old z_l^k and the updated $z_l^{k+1/2}$ up to layer $L-2$, where we can use equations (21) and (19) to relate the difference to the loss. Altogether, we obtain

$$z_l^k - z_l^{k+1/2} = \tau \left(\prod_{q=l}^{L-2} \nabla\sigma(z_q^k)\nabla_a \phi(\theta_{q+1}^k, a_q^k) \right) \cdot \nabla_\phi \mathcal{L}_y(\phi(\theta_{L-1}^k, a_{L-2}^k)). \tag{24}$$

By inserting (24) into the gradient descent update equation with respect to θ_l in ⓒ ,

$$\theta^{k+1} = \theta^k - \nabla_\theta \phi(\theta_l^k, a_{l-1}^k) \cdot \left(z_l^k - z_l^{k+1/2}\right), \tag{25}$$

we obtain the chain rule for update (18). □

Proof of Proposition 2. Since only the updates for θ_l, $l = 1, \ldots, L-2$, are performed implicitly, one can replicate the proof of Proposition 1 exactly up to equation (24). Let us denote the right hand side of (24) by g_l^k, i.e. $z_l^{k+1/2} = z_l^k - g_l^k$ and note that

$$\tau \nabla_{\theta_l} J(\boldsymbol{\theta}^k; X, y) = \nabla_\theta \phi(\cdot, a_{l-1}^k) \cdot g_l^k \tag{26}$$

holds by the chain rule (as seen in (25)). We have eliminated the dependence of $\nabla_\theta \phi(\theta_l^k, a_{l-1}^k)$ on θ_l^k and wrote $\nabla_\theta \phi(\cdot, a_{l-1}^k)$ instead, because we assume ϕ to be linear in θ such that $\nabla_\theta \phi$ does not depend on the point θ where the gradient is evaluated anymore.

We now rewrite the ProxProp update equation of the parameters θ as follows

$$\begin{aligned}
\theta_l^{k+1} &= \operatorname*{argmin}_\theta \frac{1}{2}||\phi(\theta, a_{l-1}^k) - z_l^{k+1/2}||^2 + \frac{1}{2\tau_\theta}||\theta - \theta_l^k||^2 \\
&= \operatorname*{argmin}_\theta \frac{1}{2}||\phi(\theta, a_{l-1}^k) - (z_l^k - g_l^k)||^2 + \frac{1}{2\tau_\theta}||\theta - \theta_l^k||^2 \\
&= \operatorname*{argmin}_\theta \frac{1}{2}||\phi(\theta, a_{l-1}^k) - (\phi(\theta^k, a_{l-1}^k) - g_l^k)||^2 + \frac{1}{2\tau_\theta}||\theta - \theta_l^k||^2 \\
&= \operatorname*{argmin}_\theta \frac{1}{2}||\phi(\theta - \theta^k, a_{l-1}^k) + g_l^k||^2 + \frac{1}{2\tau_\theta}||\theta - \theta_l^k||^2,
\end{aligned} \tag{27}$$

where we have used that ϕ is linear in θ. The optimality condition yields

$$0 = \nabla\phi(\cdot, a_{l-1}^k)(\phi(\theta_l^{k+1} - \theta^k, a_{l-1}^k) + g_l^k) + \frac{1}{\tau_\theta}(\theta_l^{k+1} - \theta_l^k) \tag{28}$$

Again, due to the linearity of ϕ in θ, one has

$$\phi(\theta, a_{l-1}^k) = (\nabla\phi(\cdot, a_{l-1}^k))^*(\theta), \tag{29}$$

where *, denotes the adjoint of a linear operator. We conclude

$$\begin{aligned}
&0 = \nabla\phi(\cdot, a_{l-1}^k)(\nabla\phi(\cdot, a_{l-1}^k))^*(\theta_l^{k+1} - \theta^k) + \nabla\phi(\cdot, a_{l-1}^k) g_l^k + \frac{1}{\tau_\theta}(\theta_l^{k+1} - \theta_l^k), \\
&\Rightarrow \left(\frac{1}{\tau_\theta} I + \nabla\phi(\cdot, a_{l-1}^k)(\nabla\phi(\cdot, a_{l-1}^k))^*\right)(\theta_l^{k+1} - \theta_l^k) = -\nabla\phi(\cdot, a_{l-1}^k) g_l^k = -\tau \nabla_{\theta_l} J(\boldsymbol{\theta}^k; X, y),
\end{aligned} \tag{30}$$

which yields the assertion. □

Proof of Proposition 3. Under the assumption that θ^k converges, $\theta^k \to \hat{\theta}$, one finds that $a_l^k \to \hat{a}_l$ and $z_l^k \to \hat{z}_l = \phi(\hat{\theta}_l, \hat{a}_{l-1})$ converge to the respective activations of the parameters $\hat{\theta}$ due to the forward pass and the continuity of the network. As we assume $J(\cdot; X, y)$ to be continuously differentiable, we deduce from (30) that $\lim_{k\to\infty} \nabla_{\theta_l} J(\boldsymbol{\theta}^k; X, y) = 0$ for all $l = 1, \dots, L-2$. The parameters of the last layer θ_{L-1} are treated explicitly anyways, such that the above equation also holds for $l = L - 1$, which then yields the assertion. □

Proof of Proposition 4. As the matrices

$$M_l^k := \frac{1}{\tau_\theta} I + (\nabla\phi(\cdot, a_{l-1}^k))(\nabla\phi(\cdot, a_{l-1}^k))^* \tag{31}$$

(with the convention $M_{L-1}^k = I$) are positive definite, so are their inverses, and the claim that $\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k$ is a descent direction is immediate,

$$\langle \theta_l^{k+1} - \theta_l^k, -\nabla_{\theta_l} J(\boldsymbol{\theta}^k; Y, x)\rangle = \tau \langle (M_l^k)^{-1} \nabla_{\theta_l} J(\boldsymbol{\theta}^k; Y, x), \nabla_{\theta_l} J(\boldsymbol{\theta}^k; Y, x)\rangle. \tag{32}$$

We still have to guarantee that this update direction does not become orthogonal to the gradient in the limit $k \to \infty$. The largest eigenvalue of $(M_l^k)^{-1}$ is bounded from above by τ_θ. If the a_{l-1}^k remain bounded, then so does $\nabla\phi(\cdot, a_{l-1}^k)$ and the largest eigenvalue of $\nabla\phi(\cdot, a_{l-1}^k)\nabla\phi(\cdot, a_{l-1}^k)^*$ must be bounded by some constant $\tilde{c}$. Therefore, the smallest eigenvalue of $(M_l^k)^{-1}$ must remain bounded from from below by $(\frac{1}{\tau_\theta} + \tilde{c})^{-1}$. Abbreviating $v = \nabla_{\theta_l} J(\boldsymbol{\theta}^k; Y, x)$, it follows that

$$\begin{aligned}
\cos(\alpha^k) &= \frac{\tau\langle (M_l^k)^{-1} v, v\rangle}{\tau \|(M_l^k)^{-1} v\| \|v\|} \\
&\geq \frac{\lambda_{\min}((M_l^k)^{-1})\|v\|^2}{\|(M_l^k)^{-1} v\| \|v\|} \\
&\geq \frac{\lambda_{\min}((M_l^k)^{-1})}{\lambda_{\max}((M_l^k)^{-1})}
\end{aligned} \tag{33}$$

which yields the assertion. □

Proof of Proposition 5. According to (Nocedal & Wright, 2006, p. 109, Thm. 5.3) and (Nocedal & Wright, 2006, p. 106, Thm. 5.2) the k-th iteration x_k of the CG method for solving a linear system $Ax = b$ with starting point $x_0 = 0$ meets

$$x_k = \arg \min_{x \in \text{span}(b, Ab, \ldots, A^{k-1}b)} \frac{1}{2}\langle x, Ax \rangle - \langle b, x \rangle, \tag{34}$$

i.e. is optimizing over an order-k Krylov subspace. The starting point $x_0 = 0$ can be chosen without loss of generality. Suppose the starting point is $\tilde{x}_0 \neq 0$, then one can optimize the variable $x = \tilde{x} - \tilde{x}_0$ with a starting point $x_0 = 0$ and $b = \tilde{b} + A\tilde{x}_0$.

We will assume that the CG iteration has not converged yet as the claim for a fully converged CG iteration immediately follow from Proposition 4. Writing the vectors $b, Ab, \ldots, A^{k-1}b$ as columns of a matrix $\mathcal{K}_k$, the condition $x \in \text{span}(b, Ab, \ldots, A^{k-1}b)$ can equivalently be expressed as $x = \mathcal{K}_k\alpha$ for some $\alpha \in \mathbb{R}^k$. In terms of α our minimization problem becomes

$$x_k = \mathcal{K}_k\alpha = \arg \min_{\alpha \in \mathbb{R}^k} \frac{1}{2}\langle \alpha, (\mathcal{K}_k)^T A \mathcal{K}_k \alpha \rangle - \langle (\mathcal{K}_k)^T b, \alpha \rangle, \tag{35}$$

leading to the optimality condition

$$\begin{aligned} 0 &= (\mathcal{K}_k)^T A \mathcal{K}_k \alpha - (\mathcal{K}_k)^T b, \\ \Rightarrow x_k &= \mathcal{K}_k((\mathcal{K}_k)^T A \mathcal{K}_k)^{-1}(\mathcal{K}_k)^T b. \end{aligned} \tag{36}$$

Note that A is symmetric positive definite and can therefore be written as $\sqrt{A}^T\sqrt{A}$, leading to

$$(\mathcal{K}_k)^T A \mathcal{K}_k = (\sqrt{A}\mathcal{K}_k)^T(\sqrt{A}\mathcal{K}_k) \tag{37}$$

being symmetric positive definite. Hence, the matrix $((\mathcal{K}_k)^T A \mathcal{K}_k)^{-1}$ is positive definite, too, and

$$\begin{aligned} \langle x_k, b \rangle &= \langle \mathcal{K}_k((\mathcal{K}_k)^T A \mathcal{K}_k)^{-1}(\mathcal{K}_k)^T b, b \rangle \\ &= \langle ((\mathcal{K}_k)^T A \mathcal{K}_k)^{-1}(\mathcal{K}_k)^T b, (\mathcal{K}_k)^T b \rangle > 0. \end{aligned} \tag{38}$$

Note that $(\mathcal{K}_k)^T b$ is nonzero if b is nonzero, as $\|b\|^2$ is its first entry.

To translate the general analysis of the CG iteration to our specific case, using any number of CG iterations we find that an approximate solution $\tilde{v}_l^k$ of

$$M_l^k v_l^k = -\nabla_{\theta_l} J(\boldsymbol{\theta}^k; X, y) \tag{39}$$

leads to

$$\langle \tilde{v}_l^k, -\nabla_{\theta_l} J(\boldsymbol{\theta}^k; X, y) \rangle > 0,$$

i.e., to $\tilde{v}_l^k$ being a descent direction. □

B PROXIMAL OPERATOR FOR LINEAR TRANSFER FUNCTIONS

In order to update the parameters θ_l of the linear transfer function, we have to solve the problem (11),

$$\theta^{k+1} = \underset{\theta}{\text{argmin}} \ \frac{1}{2}||\phi(\theta, a^k) - z^{k+1/2}||^2 + \frac{1}{2\tau_\theta}||\theta - \theta^k||^2. \tag{40}$$

Since we assume that ϕ is linear in θ for a fixed a^k, there exists a matrix A^k such that

$$\text{vec}(\theta^{k+1}) = \underset{\theta}{\text{argmin}} \ \frac{1}{2}||A^k\text{vec}(\theta) - \text{vec}(z^{k+1/2})||^2 + \frac{1}{2\tau_\theta}||\text{vec}(\theta) - \text{vec}(\theta^k)||^2, \tag{41}$$

and the optimality condition yields

$$\text{vec}(\theta^{k+1}) = (I + \tau_\theta(A^k)^T A^k)^{-1}(\text{vec}(\theta^k) + (A^k)^T\text{vec}(z^{k+1/2})). \tag{42}$$

In the main paper we sometimes use the more abstract but also more concise notion of $\nabla\phi(\cdot, a^k)$, which represents the linear operator

$$\nabla\phi(\cdot, a^k)(Y) = \text{vec}^{-1}((A^k)^T \text{vec}(Y)). \tag{43}$$

To also make the above more specific, consider the example of $\phi(\theta, a^k) = \theta a^k$. In this case the variable θ may remain in a matrix form and the solution of the proximal mapping becomes

$$\theta^{k+1} = \left(z^{k+1/2}\,(a^k)^\top + \frac{1}{\tau_\theta}\theta^k\right)\left(a^k(a^k)^\top + \frac{1}{\tau_\theta}I\right)^{-1}. \tag{44}$$

Since $a^k \in \mathbb{R}^{n\times N}$ for some layer size n and batch size N, the size of the linear system is independent of the batch size.

Homogeneous Linear Inequality Constraints for Neural Network Activations

Thomas Frerix, Matthias Nießner, and Daniel Cremers. Homogeneous Linear Inequality Constraints for Neural Network Activations. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2020

Following the IEEE reuse permissions, we include the *accepted* version of the publication.

Homogeneous Linear Inequality Constraints for Neural Network Activations

Thomas Frerix thomas.frerix@tum.de
Matthias Nießner niessner@tum.de
Daniel Cremers cremers@tum.de

Technical University of Munich

Abstract

We propose a method to impose homogeneous linear inequality constraints of the form $Ax \leq 0$ on neural network activations. The proposed method allows a data-driven training approach to be combined with modeling prior knowledge about the task. One way to achieve this task is by means of a projection step at test time after unconstrained training. However, this is an expensive operation. By directly incorporating the constraints into the architecture, we can significantly speed-up inference at test time; for instance, our experiments show a speed-up of up to two orders of magnitude over a projection method. Our algorithm computes a suitable parameterization of the feasible set at initialization and uses standard variants of stochastic gradient descent to find solutions to the constrained network. Thus, the modeling constraints are always satisfied during training. Crucially, our approach avoids to solve an optimization problem at each training step or to manually trade-off data and constraint fidelity with additional hyperparameters. We consider constrained generative modeling as an important application domain and experimentally demonstrate the proposed method by constraining a variational autoencoder.

1. Introduction

Deep learning models [14] have demonstrated remarkable success in tasks that require exploitation of subtle correlations, such as computer vision [11] and sequence learning [20]. Typically, humans have strong prior knowledge about a task, e.g., based on symmetry, geometry, or physics. Learning such a priori assumptions in a purely data-driven manner is inefficient and, in some situations, may not be feasible at all. While certain prior knowledge was successfully imposed – for example translational symmetry through convolutional architectures [13] – incorporating more general modeling assumptions in the training of deep networks remains an open challenge. Recently, generative neural networks have advanced significantly [8, 10]. With such models, controlling the generative process beyond a data-driven, black-box approach is particularly important.

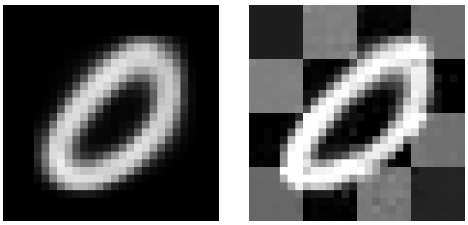


Figure 1. Samples drawn from a variational autoencoder trained on MNIST without constraints (left) and with a checkerboard constraint on the output domain (right). For a pixel intensity domain $[-1, 1]$, the checkerboard constraint forces the image tiles to have average positive or negative brightness.

In this paper, we present a method to impose prior knowledge through homogeneous linear inequality constraints of the form $Ax \leq 0$ on the activations of deep learning models. We directly impose these constraints through a suitable parameterization of the feasible set. The main two advantages of this approach are:

- The constraints are hard-constraints in the sense that they are satisfied at any point during training and inference.
- Inference on the constrained network incurs no overhead compared to unconstrained inference.

In summary, the main contribution of our method is a reparameterization that incorporates homogeneous linear inequality hard-constraints on neural network activations and allows for efficient test time predictions, i.e., our method is faster up to two orders of magnitude. The model can be optimized by standard variants of stochastic gradient descent. As an application in generative modeling, we demonstrate that our method is able to produce authentic samples from a variational autoencoder while satisfying the imposed constraints.

2. Related work

Various works have introduced methods to impose some type of hard constraint on neural network activations.

Márquez-Neila et al. [15] formulated generic differentiable equality constraints as soft constraints and employed a Lagrangian approach to train their model. While this is a principled approach to constrained optimization, it does not scale well to practical deep neural network models with their vast number of parameters. To make their method computationally tractable, a subset of the constraints is selected at each training step. In addition, these constraints are locally linearized; thus, there is no guarantee that this subset will be satisfied after a parameter update.

For the specific problem of weakly supervised segmentation, Pathak et al. [18] proposed an optimization scheme that alternates between optimizing the deep learning model and fitting a constrained distribution to these intermediate models. However, this method involves solving a (convex) optimization problem at each training step. Furthermore, the overall convergence path depends on how the alternating optimization steps are combined, which introduces an additional hyperparameter that must be tuned. Briq et al. [4] approached the weakly supervised segmentation problem with a layer that implements the orthogonal projection onto a simplex, thereby directly constraining the activations to a probability distribution. This optimization problem can be solved efficiently, but does not generalize to other types of inequality constraints.

OptNet, an approach to solve a generic quadratic program as a differentiable network layer, was proposed by Amos and Kolter [1]. OptNet backpropagates through the first-order optimality conditions of the quadratic program, and linear inequality constraints can be enforced as a special case. The formulation is flexible; however, it scales cubically with the number of variables and constraints. Thus, it becomes prohibitively expensive to train large-scale deep learning models.

Finally, several works have proposed handcrafted solutions for specific applications, such as skeleton prediction [21] and prediction of rigid body motion [5]. In contrast, to avoid laborious architecture design, we argue for the value of generically modeling constraint classes. In practice, this makes constraint methods more accessible for a broader class of problems.

Contribution In this work, we tackle the problem of imposing homogeneous linear inequality constraints on neural network activations. Rather than solving an optimization problem during training, we split this task into a *feasibility step* at initialization and an *optimality step* during training. At initialization, we compute a suitable parameterization of the constraint set (a polyhedral cone) and use the neural network training algorithm to find a good solution within this feasible set. Conceptually, we are trading-off computational cost during initialization to obtain a model that has no overhead at test time. The proposed method is implemented as a neural network layer that is specified by a set of homogeneous linear inequalities and whose output parameterizes the feasible set.

3. Linear constraints for deep learning models

We consider a generic L layer neural network F_θ with model parameters θ for inputs x as follows:

$$F_\theta(x) = f^{(L)}_{\theta_L}(\sigma(f^{(L-1)}_{\theta_{L-1}}(\sigma(\dots f^{(1)}_{\theta_1}(x)\dots)))), \quad (1)$$

where $f^{(l)}_{\theta_l}$ are affine functions, e.g., a fully-connected or convolutional layer, and σ is an elementwise non-linearity[1], e.g., a sigmoid or rectified linear unit (ReLU). In supervised learning, training targets y are known and a loss $\mathcal{L}_y(F_\theta(x))$ is minimized as a function of the network parameters θ. A typical loss for a classification task is the cross entropy between the network output and the empirical target distribution, while the mean-squared error is commonly used for a regression task. The proposed method can be applied to constrain any linear activations $z^{(l)} = f^{(l)}_{\theta_l}(a^{(l-1)})$ or non-linear activations $a^{(l)} = \sigma(z^{(l)})$. In most cases, one would like to constrain the output $F_\theta(x)$.

The feasible set for m linear inequality constraints in d dimensions is the convex polyhedron

$$\mathcal{C} \coloneqq \left\{ z \,\middle|\, Az \leq b, A \in \mathbb{R}^{m\times d}, b \in \mathbb{R}^m \right\} \subseteq \mathbb{R}^d \,. \quad (2)$$

A suitable description of the convex polyhedron $\mathcal{C}$ is obtained by the decomposition theorem for polyhedra.

Theorem 1 (Decomposition of polyhedra, Minkowski-Weyl)**.** *A set $\mathcal{C} \subset \mathbb{R}^d$ is a convex polyhedron of the form* (2) *if and only if*

$$\begin{aligned}\mathcal{C} &= \operatorname{conv}(v_1,\dots,v_n) + \operatorname{cone}(r_1,\dots,r_s) \\ &= \left\{ \sum_{i=1}^n \lambda_i v_i + \sum_{j=1}^s \mu_j r_j \,\middle|\, \lambda_i,\mu_j \geq 0, \sum_{i=1}^n \lambda_i = 1 \right\} \quad (3)\end{aligned}$$

for finitely many vertices $\{v_1,\dots,v_n\}$ and rays $\{r_1,\dots,r_s\}$.

Furthermore, $\mathcal{C} = \left\{z | Az \leq 0, A \in \mathbb{R}^{m\times d}\right\}$ if and only if

$$\mathcal{C} = \operatorname{cone}(r_1,\dots,r_s) \quad (4)$$

for finitely many rays $\{r_1,\dots,r_s\}$.

[1] Formally, σ maps between different spaces for different layers and may also be a different element-wise non-linearity for each layer. We omit such details in favor of notational simplicity.

The theorem states that an intersection of half-spaces (half-space or H-representation) can be written as the Minkowski sum of a convex combination of the polyhedron's vertices and a conical combination of some rays (vertex or V-representation). One can switch algorithmically between these two viewpoints via the double description method [7, 16], which we discuss in the following. Thus, the H-representation, which is natural when modeling inequality constraints, can be transformed into the V-representation, which can be incorporated into gradient-based neural network training.

In this paper, we focus on *homogeneous* constraints of the form (4), for which the feasible set is a polyhedral cone. Due to the special structure of this set, we can avoid to work with the convex combination parameters in (3), which is numerically advantageous (Section 3.5), and we can efficiently combine modeling constraints and domain constraints, such as a $[-1, 1]$-pixel domain for images (Section 3.3). Such a polyhedral cone is shown in Figure 2.

3.1. Double description method

The double description method converts between the half-space and vertex representation of a system of linear inequalities. It was originally proposed by Motzkin et al. [16] and further refined by Fukuda and Prodon [7].[2] Here, we are only interested in the conversion from H-representation to V-representation for homogeneous constraints (4),

$$\mathcal{H} \rightarrow \text{cone}(r_1, \ldots, r_s) \; . \tag{5}$$

The core algorithm proceeds as follows. Let the rows of A define a set of homogeneous inequalities and let $R = [r_1, \ldots, r_s]$ be the matrix whose columns are the rays of the corresponding cone. Here, (A, R) form a double description pair. The algorithm iteratively builds a double description pair (A^{k+1}, R^{k+1}) from (A^k, R^k) in the following manner. The rows in A^k represent a k-subset of the rows of A and thus define a convex polyhedron associated with R^k. Adding a single row to A^k introduces an additional half-space constraint, which corresponds to a hyperplane. If the vector $r_i - r_j$ for two columns r_i, r_j of R^k intersects with this hyperplane then this intersection point is added to R^k. Existing rays that are cut-off by the additional hyperplane are removed from R^k. The result is the double description pair (A^{k+1}, R^{k+1}). This procedure is shown in Figure 2.

Adding a hyperplane might drastically increase the number of rays in intermediate representations, which, in turn, contribute combinatorically in the subsequent iteration. In fact, there exist worst case polyhedra for which the algorithm has exponential run time as a function of the number of inequalities and the input dimension, as well as the number of rays [3, 6]. Overall, one can expect the algorithm to be efficient only for problems with a reasonably small number m of inequalities and dimension d.

[2] In our experiments we use `pycddlib`, which is a Python wrapper of Fukuda's `cddlib`.

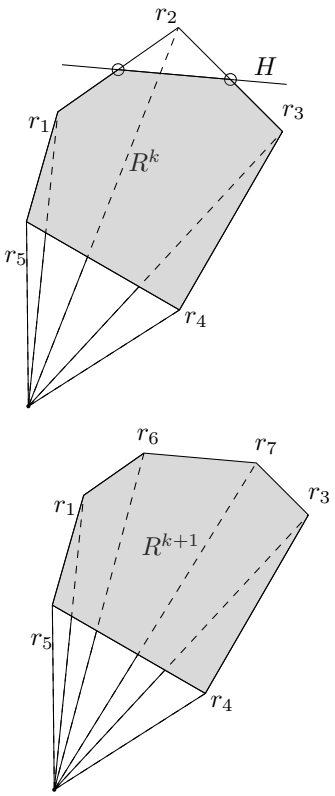


Figure 2. Diagram illustrating an iteration of the double description method. Adding a constraint to the k-constraint set A^k at iteration $k+1$ introduces a hyperplane H. The intersection points of H with the boundary of the current polyhedron R^k (marked by $\circ$) are added as rays r_6 and r_7 to the polyhedral cone. The ray r_2 is cut-off by the hyperplane H and is removed from R^k. The result is the next iterate R^{k+1}.

3.2. Integration in neural network architectures

We parameterize the homogeneous form (4) via a neural network layer. This layer takes as input some (latent) representation of the data, which is mapped to activations satisfying the desired hard constraints. The algorithm is provided with the H-representation of linear inequality constraints, i.e., a matrix $A \in \mathbb{R}^{m \times d}$ for m constraints in d dimensions to specify the feasible set (4). At initialization, we convert this to the V-representation via the double description method (Section 3.1). This corresponds to computing the set of rays $\{r_1, \ldots, r_s\}$ to represent the polyhedral cone. During training, the neural network training algorithm is used to optimize within in the feasible set. There are two critical aspects in this procedure. First, as outlined in Section 3.1, the run-time complexity of the double description method may be prohibitive. Conceptually, the proposed approach allows for significant compute time at initialization

to obtain an algorithm that is very efficient at training and test time. Second, we must ensure that the mapping from the latent representation to the parameters integrates well with the training algorithm. We assume that the model is trained with gradient-based backpropagation, as is common for current deep learning applications. The constraint layer comprises a batch normalization layer and an affine mapping (fully-connected layer with biases) followed by the element-wise absolute value function that ensures the non-negativity required by the conical combination parameters. In theory, any function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ would fulfill this requirement; however, care must be taken to not interfere with backpropagated gradients.

3.3. Combining modeling and domain constraints

Domain constraints are often formulated as unit box constraints, $\mathcal{B} \coloneqq \{x \in \mathbb{R}^d | -1 \leq x_i \leq 1\}$, such as a pixel domain for images. Box constraints are particularly unfit to be converted using the double description method because the number of vertices is exponential in the dimension. Therefore, we distinguish *modeling constraints* and *domain constraints* and only convert the former into V-representation. Based on this representation, we obtain a point in the modeling constraint set, $x \in \mathcal{C}$. However, this point may not be in the unit box $\mathcal{B}$. To arrive at a point in the intersection $\mathcal{C} \cap \mathcal{B}$, we normalize x by its infinity norm if $x \notin \mathcal{B}$, $\hat{x} = x / \max\{\|x\|_\infty, 1\}$. Indeed, $\hat{x} \in \mathcal{C} \cap \mathcal{B}$ since scaling by a positive constant remains in the cone, i.e., if $x \in \mathcal{C}$, then $\alpha x \in \mathcal{C}\ \forall \alpha \geq 0$.

3.4. Applications of homogeneous constraints

A natural application of constraints of the form $Ax \leq 0$ is a parameterization of a set of binary classifiers. If each row a_i of A is such a binary classifier, then the method presented in this paper parameterizes the set $\{x | a_i^T x \leq 0\ \forall i\}$. Consequently, it can be guaranteed that neural network activations satisfy a set of binary criteria. Another domain is to express certain direct relations between neural network activations. Notably, one can guarantee mathematical properties such as monotonicity via $x_{i+1} \geq x_i$ and convexity via $x_{i+1} - 2x_i + x_{i-1} \geq 0$.

3.5. Extension to general linear constraints

The proposed method takes advantage of the special structure of a polyhedral cone to efficiently combine modeling and domain constraints (Section 3.3). General linear inequality constraints of the form $Ax \leq b$ without restrictions on A and b possibly require the conic and convex component of (3) for their V-representation. The main approach of this paper may be used in this case, i.e., our layer additionally needs to predict convex combination parameters. However, we observed slow convergence, which we ascribe to the simplex parameterization for the convex combination parameters. We used a softmax function $f(x)_i = \exp(x_i) / \sum_{j=1}^m \exp(x_j)$ to enforce the constraints $\lambda_i \geq 0, \sum_{i=1}^m \lambda_i = 1$ of the convex combination parameters in (3). This function has vanishing gradients when one x_i is significantly greater than the other vector entries. Furthermore, this most general setting does not allow for efficient incorporation of domain constraints, as this would require an efficient parameterization of the intersection of a general convex polyhedron and the unit box.

4. Numerical results

We compare the proposed *constraint parameterization* algorithm with an algorithm that trains without constraints, but requires a projection step at test time. We call this latter algorithm *test time projection*. The proposed algorithm optimizes over the feasible set, while the projection is restricted to yielding a solution on the boundary of that set. We analyze these algorithms in two different settings. In an initial experiment, we learn the orthogonal projection onto a constraint set to demonstrate properties of these algorithms. Here, the result can be compared to the optimal solution of the convex optimization problem. In a second experiment, consistent with our motivation to constrain the output of generative models, we apply these algorithms to a variational autoencoder. Finally, we evaluate the running time of inference for these problems and show that the proposed algorithm is significantly more efficient compared to the test time projection method.

We used the MNIST dataset [12] for both experiments (59000 training, 1000 validation, and 10000 test samples). We chose PyTorch [17] for our implementation[3] and all experiments were performed on a single Nvidia Titan X GPU. All networks were optimized with the Adam optimizer and we evaluated learning rates in the range $[10^{-5}, 10^{-3}]$. The initial learning rate was annealed by a factor of $1/2$ if progress on the validation loss stagnated for more than 5 epochs. We used OSQP [19] as an efficient solver to compute orthogonal projections.

Both experiments were performed with a checkerboard constraint with 16 tiles, where neighboring tiles are constrained to be on average either below or above pixel domain midpoint. For a $[-1, 1]$-pixel domain, the tiles' average intensity is positive or negative, respectively. The initial computational cost of converting these constraints into V-representation via the double description method is negligible (less than 1s). We observed that it is numerically advantageous to activate unit box scaling after the constraint parameterization model was initially optimized only with modeling constraints for a specified number of epochs.

One might consider OptNet [1] and an analogous version of the method introduced by Pathak et al. [18] as baselines.

[3] https://github.com/tfrerix/constrained-nets

However, these approaches incur a significant drawback for the setup presented in this paper as they are are computationally expensive at training time. An OptNet layer solves a generic quadratic program as a differentiable network layer, which scales cubically with the number of variables and constraints. The method by Pathak et al. [18] for the regression problems in this paper alternates between optimization steps in the network parameters via a variant of stochastic gradient descent and projecting the network output onto the constraints, which is computationally expensive.

4.1. Orthogonal projection onto a constraint set

We learn an orthogonal projection to demonstrate general properties of both algorithms. For given linear inequalities specified in H-representation, we solve the following problem:

$$\min_{z \in \mathbb{R}^d} \|z - y\|_2 \quad \text{s.t. } Az \leq 0 \qquad , \tag{6}$$

where y is an MNIST image. Here, the problem is convex; therefore, the global optimum can be readily computed and compared to the performance of the learning algorithms. In this setting, we can expect that training an unconstrained network with subsequent projection onto the constraint set at test time yields good results, which can be seen as follows. Let $\mathcal{P}_{\mathcal{C}}(y) \coloneqq \arg\min_{z \in \mathcal{C}} \|z - y\|_2$ be the orthogonal projection onto the constraint set $\mathcal{C}$ and denote the mean-squared error as $\mathcal{L}_y(x) \coloneqq \|x - y\|_2$. Both mappings are Lipschitz continuous with Lipschitz constant $L = 1$. Consequently, for an output $\hat{y}$ of an unconstrained model,

$$\begin{aligned} \left| \mathcal{L}_y(\mathcal{P}_{\mathcal{C}}(\hat{y})) - \mathcal{L}_y(\mathcal{P}_{\mathcal{C}}(y)) \right| &\leq \left\| \mathcal{P}_{\mathcal{C}}(\hat{y}) - \mathcal{P}_{\mathcal{C}}(y) \right\|_2 \\ &\leq \|\hat{y} - y\|_2 \ , \end{aligned} \tag{7}$$

where, by definition, the term $\mathcal{L}_y(\mathcal{P}_{\mathcal{C}}(y))$ is the optimal value of problem (6). The training algorithm fits $\hat{y}$ to y; therefore, projecting the unconstrained output $\hat{y}$ onto the constraint set will yield an objective value that is close to the optimal value of the constrained optimization problem.

To have a comparable number of parameters for both methods, we use a single fully-connected layer in both cases. For the unconstrained model, we employ an $FC(784, 784)$ layer, and for the constrained model we employ an $FC(784, n_r)$ layer with $n_r = 1552$ many rays to represent the constraint set in V-representation. Additionally, the constraint layer first applies a batch normalization operation [9]. Both models were optimized with an initial learning rate of 10^{-4}, which was annealed by a factor of 0.1 if progress on the validation loss stagnated for more than 5 epochs. The batch size was chosen to be 256. The unit box constraints were activated after 25 epochs. Additionally, the data for training the model with all constraints being active is shown. This mode eventually results in worse generalization. Figure 3 shows that the mean-squared validation objective for both algorithms converges close to the average optimum. The constraint parameterization method has a larger variance and optimality gap, which hints at the numerical difficulty of training the constrained network. To be precise, the best average validation error during training is within 9% of the optimum for the constraint parameterization method and within 1% of the optimum for the test time projection method. Figure 4 shows a test set sample and the respective output of the learned models.

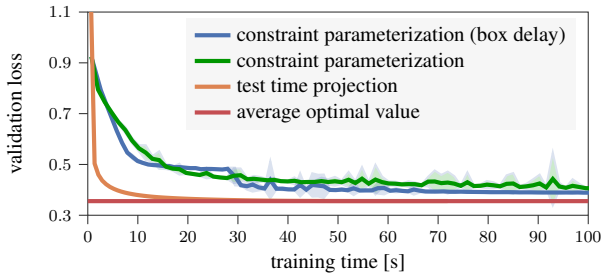


Figure 3. Mean-squared validation loss averaged over all pixels for 10 runs; shaded area denotes standard deviation. The objective function (6) is computed on a held-out validation set for the proposed constraint parameterization method and unconstrained optimization with subsequent test time projection. The average optimum over the validation set is obtained as a solution to a convex optimization problem. For the *box delay* curve, the box constraints are activated after 25 epochs (after $\sim$ 30s), which results in better generalization. The best average validation error during training is within 9% of the optimum for the constraint parameterization method with box constraint delay and within 1% of the optimum for the test time projection method.

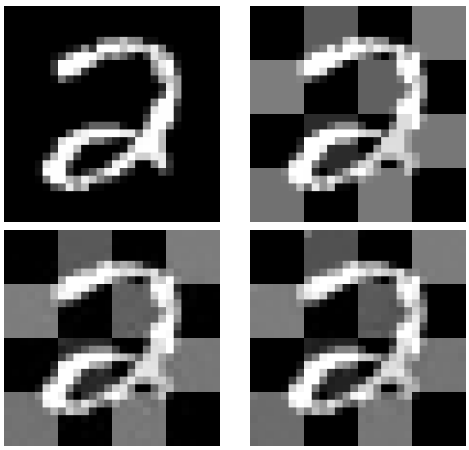


Figure 4. Learning to solve the orthogonal projection onto a constraint set as defined in (6). Top left: MNIST sample from a test set. Top right: optimal projection by solving a quadratic program. Bottom left: test time projection model inference. Bottom right: constraint parameterization model inference (ours).

Table 1. Inference time for test time projection (TTP) and constraint parameterization (CP) methods. Mean and standard deviation of running times are computed over 100 runs of 59000 samples with a batch size of 256.

METHOD	PROJECTION	VAE
TTP	82 ± 1 s	40 ± 1 s
CP (ours)	$\mathbf{0.46 \pm 0.02}$ s	$\mathbf{0.75 \pm 0.04}$ s

4.2. Constrained generative modeling

Variational autoencoders (VAE) are a class of generative models that are jointly trained to encode observations into latent variables via an encoder or inference network and decode observations from latent variables using a decoder or generative network [10]. We base our implementation on [2]. The model has a fully-connected architecture:

$$\begin{aligned}\text{encoder: } & FC(784, 256) - \text{ReLU} - FC(256, 2) \\ \text{decoder: } & FC(2, 256) - \text{ReLU} - FC(256, 784) \\ & - \text{sigmoid} - \text{constraint}\end{aligned}$$

Here, $\text{ReLU}(x) = \max(0, x)$ and the sigmoid non-linearity takes the form $\sigma(x) = 1/(1 + \exp(-x))$. In contrast to a standard VAE, we constrain the samples generated by the model to obey a checkerboard constraint. The model was optimized with an initial learning rate of 10^{-4}, which was annealed by a factor of 0.1 if progress on the validation loss stagnated for more than 5 epochs. The batch size was chosen to be 64. The model was trained for 200 epochs while the unit box constraints were activated after 100 epochs. To generate images, we sample the latent space prior $z \sim \mathcal{N}(0, I)$ and evaluate the decoding neural network (Figure 5). The model is able to sample authentic digits while obeying the checkerboard constraint.

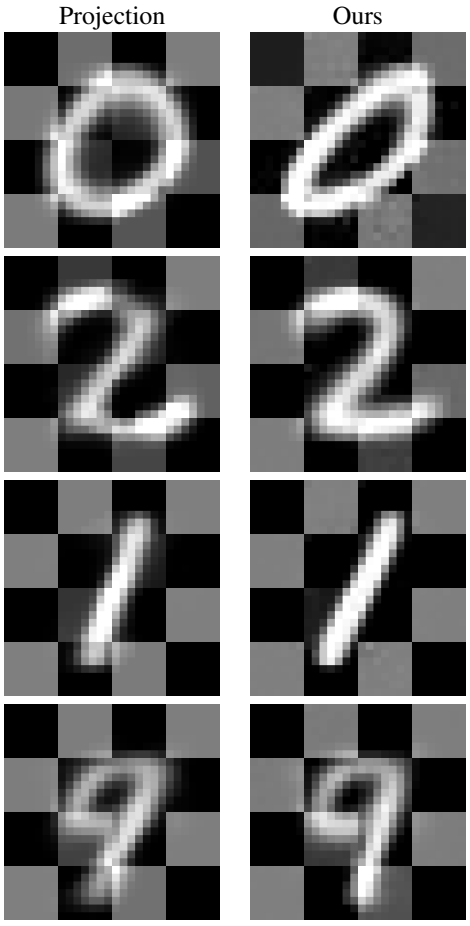


Figure 5. Samples from a constrained variational autoencoder trained with the test time projection method and our constraint parameterization method. The images represent authentic digits while satisfying the imposed checkerboard constraint. Inference is significantly faster using our method.

4.3. Fast inference with constrained networks

The main advantage of the proposed method over a simple projection method is a vast speed-up at test time. Since the constraint is incorporated into the neural network architecture, a forward pass has almost no overhead compared to an unconstrained network. On the other hand, for a network that was trained without constraints, a final projection step is necessary; this requires solving a convex optimization problem, which is relatively costly. Table 1 shows inference times for both models for the above numerical experiments. The constraint parameterization approach is up to two orders of magnitude faster at test time compared to the test time projection algorithm.

5. Conclusion

To combine a data-driven task with modeling constraints, we have developed a method to impose homogeneous linear inequality constraints on neural network activations. At initialization, a suitable parameterization is computed and subsequently a standard variant of stochastic gradient descent is used to train the reparameterized network. In this way, we can efficiently guarantee that network activations satisfy the constraints at any point during training. The main advantage of our method over simply projecting onto the feasible set after unconstrained training is a significant speed-up at test time of up to two orders of magnitude. An important application of the proposed method is generative modeling with prior assumptions. Therefore, we demonstrated experimentally that the proposed method can be used successfully to constrain the output of a variational autoencoder. Our method is implemented as a layer, which is simple to combine with existing and novel neural network architectures in modern deep learning frameworks and is therefore readily available in practice.

Acknowledgements

The authors would like to thank Thomas Möllenhoff, Erik Bylow, and Gideon Dresdner for fruitful discussions and valuable feedback on the manuscript. This work was supported by an Nvidia Professorship Award, the TUM-IAS Carl von Linde and Rudolf Mößbauer Fellowships, the ERC Starting Grant *Scan2CAD (804724)*, and the Gottfried Wilhelm Leibniz Prize Award of the DFG.

References

[1] Brandon Amos and J. Zico Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, pages 136–145, 2017.

[2] Tim Baumgärtner. VAE-CVAE-MNIST. `https://github.com/timbmg/VAE-CVAE-MNIST`, 2018. commit: e4ba231.

[3] David Bremner. Incremental convex hull algorithms are not output sensitive. *Discrete & Computational Geometry*, 21 (1):57–68, 1999.

[4] Rania Briq, Michael Moeller, and Juergen Gall. Convolutional Simplex Projection Network (CSPN) for Weakly Supervised Semantic Segmentation. *BMVC 2018*, 2018.

[5] Arunkumar Byravan and Dieter Fox. SE3-nets: Learning rigid body motion using deep neural networks. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

[6] Martin E. Dyer. The complexity of vertex enumeration methods. *Mathematics of Operations Research*, 8(3):381–402, 1983.

[7] Komei Fukuda and Alain Prodon. *Double description method revisited*, pages 91–111. Combinatorics and Computer Science: 8th Franco-Japanese and 4th Franco-Chinese Conference Brest, France, July 3–5, 1995 Selected Papers. Springer Berlin Heidelberg, 1996.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27 (NIPS 2014)*, pages 2672–2680. 2014.

[9] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pages 448–456. PMLR, 07–09 Jul 2015.

[10] Diederik Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations (ICLR 2014)*, 2014.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference of Neural Information Processing Systems (NIPS 2012)*, 2012.

[12] Yann LeCun, Corinna Cortes, and Christopher Burges. The MNIST database of handwritten digits. URL `http://yann.lecun.com/exdb/mnist/`.

[13] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[15] Pablo Márquez-Neila, Mathieu Salzmann, and Pascal Fua. Imposing hard constraints on deep networks: Promises and limitations. *First Workshop on Negative Results in Computer Vision, CVPR 2017*, 2017.

[16] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In *Contributions to the Theory of Games II*, volume 8 of *Ann. of Math. Stud.*, pages 51–73. Princeton University Press, 1953.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. *Autodiff Workshop, NIPS 2017*, 2017.

[18] Deepak Pathak, Philipp Krähenbühl, and Trevor Darrell. Constrained Convolutional Neural Networks for Weakly Supervised Segmentation. In *International Conference on Computer Vision (ICCV 2015)*, 2015.

[19] Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. OSQP: An Operator Splitting Solver for Quadratic Programs. *ArXiv e-prints*, 2017.

[20] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference of Neural Information Processing Systems (NIPS 2014)*, 2014.

[21] Xingyi Zhou, Xiao Sun, Wei Zhang, Shuang Liang, and Yichen Wei. Deep Kinematic Pose Regression. *Workshop on Geometry Meets Deep Learning, ECCV 2016*, 2016.

Approximating Orthogonal Matrices with Effective Givens Factorization

Thomas Frerix and Joan Bruna. Approximating Orthogonal Matrices with Effective Givens Factorization. In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 2019

Approximating Orthogonal Matrices with Effective Givens Factorization

Thomas Frerix [1] **Joan Bruna** [2]

Abstract

We analyze effective approximation of unitary matrices. In our formulation, a unitary matrix is represented as a product of rotations in two-dimensional subspaces, so-called Givens rotations. Instead of the quadratic dimension dependence when applying a dense matrix, applying such an approximation scales with the number factors, each of which can be implemented efficiently. Consequently, in settings where an approximation is once computed and then applied many times, such a representation becomes advantageous. Although effective Givens factorization is not possible for generic unitary operators, we show that minimizing a sparsity-inducing objective with a coordinate descent algorithm on the unitary group yields good factorizations for structured matrices. Canonical applications of such a setup are orthogonal basis transforms. We demonstrate numerical results of approximating the graph Fourier transform, which is the matrix obtained when diagonalizing a graph Laplacian.

1. Introduction

Unitary operators are ubiquitous in many areas, from numerical linear algebra to quantum computing and cryptography. Celebrated applications include the QR-decomposition and the diagonalization of symmetric matrices (Golub & Van Der Vorst, 2000). Without any assumptions on the structure of the matrix, applying a unitary transformation in d dimensions requires $\mathcal{O}(d^2)$ operations for the matrix-vector product. In scenarios where a given unitary operator needs to be intensively applied many times, using approximations that trade-off accuracy with a better scaling behavior in the dimension is desirable.

In this paper, we develop a method to compute approximations of unitary matrices in the form of Givens factorization (Givens, 1958). Givens rotations are localized in a two-dimensional subspace of predefined coordinates. Therefore, computations with Givens sequences scale with the number of factors and the computational cost for applying each factor can be kept low since efficient implementations are possible (Golub & Van Loan, 2012). Our main motivation comes from the success story of the Fast Fourier transform (FFT) (Cooley & Tukey, 1965), which brought down the computational cost of applying a Fourier transform to $\mathcal{O}(d\log(d))$ operations. This reduction led to a revolution in signal processing and was recognized by Sullivan & Dongarra (2000) as one of the most important algorithms of the 20th century. However, this speed-up relies on the fact that the classical Fourier transform is defined over a periodic grid, which provides many symmetries leveraged in the butterfly structure of the FFT.

These symmetries do not carry over to unstructured domains such as graphs and general unitary operators. In fact, using simple covering bounds, we show that generic unitary matrices require $\mathcal{O}(d^2/\log d)$ Givens factors to be effectively approximated. However, the question of approximating with fewer factors in the presence of structure remains open: given an element $U \in U(d)$, how to produce the best possible N-term sequence of Givens rotations $G_1 \dots G_N$ that minimizes $\left\| U - \prod_j G_j \right\|$?

Due to the combinatorial nature of selecting Givens subspaces, this is an NP-hard optimization problem. In this paper, we propose a relaxation based on sparsity-inducing norms over the unitary group. In essence, given a point $U \in U(d)$, we use the gradient flow of a potential function $f : U(d) \to \mathbb{R}$ to define a path that links U to its nearest signed permutation matrix, the sparsest elements of the group and thus the global minimizers of f. Then, our algorithm tries to approximately follow this path using coordinate descent with the Givens factors acting as generators of the group.

We validate our algorithm on a family of structured orthogonal operators, constructed with a planted random sequence of K Givens factors and demonstrate that effective approximation is possible in the regime $K = \mathcal{O}(d\log d)$. Finally, we apply our algorithm to approximate a graph Fourier transform (GFT), the orthogonal matrix obtained when di-

[1]Technical University of Munich [2]New York University. Correspondence to: Thomas Frerix <thomas.frerix@tum.de>, Joan Bruna <bruna@cims.nyu.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

agonalizing a graph Laplacian.

For ease of exposition, we restrict our discussion to approximating orthogonal group elements. However, this does not impose a restriction on the outlined approaches, as they equally apply to the complex unitary group as well as the real orthogonal group.

2. Related Work

Givens rotations were introduced by (Givens, 1958) to factorize the unitary matrix that transforms a square matrix into triangular form. The elementary operation of rotating in a two-dimensional subspace led to numerous successful applications in numerical linear algebra (Golub & Van Loan, 2012), in particular, for eigenvalue problems (Golub & Van Der Vorst, 2000). In this context, a Givens sequence factorizes a unitary basis transform, which is an operation of paramount importance to signal processing.

In contrast to signal processing on a Euclidean domain, recently there has been increased interest in signal processing on irregular domains such as graphs (Shuman et al., 2013; Bronstein et al., 2017). In this setting, Magoarou et al. (2018) considered a truncated version of the classical Jacobi algorithm (Jacobi, 1846) to approximate the orthogonal matrix that diagonalizes a graph Laplacian. Other notable strategies to efficiently approximate large matrices with presumed structure include multiresolution analysis (Kondor et al., 2014) and sparsity (Kyng & Sachdeva, 2016).

In quantum computation, approximate representation of unitary operators is a fundamental problem. Here, a unitary operation that performs a computation on a quantum state needs to be represented by or approximated with few elementary single- and two-qubit gates, ideally polynomial in the number of qubits. In the literature of quantum computing, a Givens rotation is commonly referred to as a two-level unitary matrix; a generic n-qubit unitary operator can be factorized in such two-level matrices with $\mathcal{O}\left(4^n\right)$ elementary quantum gates (Vartiainen et al., 2004).

An alternative viewpoint on Givens sequences was analyzed by Shalit & Chechik (2014). The authors considered manifold coordinate descent over the orthogonal group as sequentially applying Givens factors. Consequently, the minimizing sequence of this algorithm yields a Givens factorization of the initial orthogonal matrix.

In this work, we analyze information theoretic properties of approximating unitary matrices via Givens factorization. We then propose to minimize a sparsity-inducing objective via manifold coordinate descent in a regime where effective approximation is possible. Subsequently, we apply this approach to approximate the graph Fourier transform and demonstrate that the proposed method can find better sequences compared to a truncated Jacobi algorithm. This allows to efficiently transform a graph signal into the graph's approximate Fourier basis, an essential operation in graph signal processing.

3. Givens Factorization and Elimination

Givens matrices represent rotations in a two-dimensional subspace, while leaving all other dimensions invariant (Givens, 1958; Golub & Van Loan, 2012). Such a counterclockwise rotation in the (i, j)-plane by an angle α can be written as applying $G^T(i, j, \alpha)$, where

$$G(i,j,\alpha) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos(\alpha) & \cdots & \sin(\alpha) & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -\sin(\alpha) & \cdots & \cos(\alpha) & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \tag{1}$$

The trigonometric expressions appear in the i-th and j-th rows and columns. Any orthogonal matrix $U \in \mathbb{R}^{d\times d}$ that is a rotation, $U \in SO(d)$, can be decomposed into a product of at most $d(d-1)/2$ Givens rotations. In general, there exist many possible factorizations. If $U \in O(d) \setminus SO(d)$, then it cannot be represented directly by a sequence of Givens rotations. However, a factorization can be obtained up to permutation with a negative sign, e.g., by flipping two columns.

In numerical linear algebra, Givens factors are often used to selectively introduce zero matrix entries by controlling the rotation angle. This leads to a constructive factorization algorithm, which demonstrates a $d(d-1)/2$-factorization. To this end, we start with the matrix $U \in SO(d)$ and introduce zeros on the lower diagonal column-wise from left to right and bottom to top within every column. This is achieved by choosing the rotation subspace (i, j) and a suitable rotation angle to zero-out the matrix element (i, j). The elimination order is illustrated for $d = 4$ by

$$\begin{pmatrix} * & * & * & * \\ 3 & * & * & * \\ 2 & 5 & * & * \\ 1 & 4 & 6 & * \end{pmatrix} \tag{2}$$

After $N = d(d-1)/2$ steps, we have $G_N^T \dots G_1^T U = D$, where D is a diagonal matrix with $D_{kk} = -1$ for an even number of values and $D_{kk} = 1$ otherwise. This result can be reduced to the identity by selecting two subspaces with values $D_{ii} = D_{jj} = -1$ and applying a rotation by an angle $\alpha = \pi$. We refer to this algorithm by *structured elimination*.

Apart from this sign ambiguity, we consider factorizations in the broader sense up to signed permutation of the resulting matrix columns. To be explicit, the set of signed

permutation matrices is defined as $\mathcal{P}_d \coloneqq \{P \in \mathbb{R}^{d\times d} | P_{ij} \in \{-1, 0, 1\}, \sum_i |P_{ij}| = 1 \; \forall j, \sum_j |P_{ij}| = 1 \; \forall i\}$. For a matrix $U \in O(d)$, to measure approximation quality, we denote an approximation by $\hat{U}$ and use a symmetrized Frobenius norm criterion up to a signed permutation matrix as follows:

$$\left\|U - \hat{U}\right\|_{F,\mathrm{sym}} \coloneqq \min_{P\in\mathcal{P}_d} \left\|U - \hat{U}P\right\|_F . \tag{3}$$

The range of (3) over the orthogonal group is $[0, \sqrt{2d})$ as the maximum is obtained for the distance between Hadamard[1] matrices $H(d)$ and the identity with $\left\|H(d) - I\right\|_{F,\mathrm{sym}} / \sqrt{d} \to \sqrt{2}$ as $d \to \infty$. Since $\|A\|_F^2 = \mathbb{E}_{x\sim\mathcal{N}(0,I)} \left[\|Ax\|_2^2\right]$, the criterion measures the average approximation quality over random Gaussian vectors when applying $\hat{U}$ instead of U. The motivation for this definition is twofold. First, this definition allows us to discuss Givens factorizations of orthogonal matrices with negative determinant and henceforth we consider factorization over the orthogonal group $O(d)$ rather than the special orthogonal group $SO(d)$. Second, it enlarges the class of possible factorization algorithms to those that cannot distinguish between signed permutation matrices. Observe that since the cost of multiplying by a signed permutation matrix is $\mathcal{O}(d)$ (Knuth, 1998), the computational efficiency arguments in this paper are not affected by the permutation equivalence class as we are discussing approximations in the regime of $\mathcal{O}\left(d\log(d)\right)$ factors.

4. Information Theoretic Rate of Givens Representation

The elimination algorithm discussed in Section 3 guarantees to factorize any orthogonal matrix in at most $d(d-1)/2$ Givens factors, which corresponds to the dimension of the orthogonal group. Since each Givens factor is parametrized by a single angle, it immediately follows that exact Givens factorization for arbitrary elements $U \in O(d)$ necessarily requires $d(d-1)/2$ factors.

Hence, this leads to the question of approximate factorization: if one tolerates a certain error $\|U - \hat{U}\|_F \leq \epsilon$, is it possible to find approximations $\hat{U} = \prod_{n\leq N} G_n$ with $N = o(d^2)$, ideally with $N = \mathcal{O}(d\log d)$? A covering argument shows that generic orthogonal matrices in d dimensions require at least $\Theta(d^2/\log(d))$ Givens factors to achieve an ϵ-approximate factorization. We denote by μ the uniform Haar measure on the unitary group, which we normalize for each d, $\mu(U(d)) = 1$. For notational simplicity, we carry out the proof for the operator 2-norm. An analogous argument holds by replacing the operator 2-norm with the Frobenius norm while re-scaling the error by $\sqrt{d}$.

[1] A Hadamard matrix is an orthogonal matrix H whose entries satisfy $|H_{i,j}| = 1/\sqrt{d}$ for all i, j.

Lemma 1. *Let $\prod_{n\leq N} G_n$ be a product of Givens factors with rotation angles α_n and $\bar{G}_n$ be the respective perturbed factors with rotation angles $\alpha_n + \delta_n$ and perturbations $0 \leq \delta_n \leq \delta$. Then,*

$$\left\| \prod_{n\leq N} \bar{G}_n - \prod_{n\leq N} G_n \right\|_F \leq 2N\delta . \tag{4}$$

Proof. For any orthogonal matrices U, U', V, V', we have

$$\begin{aligned} \left\|U'V' - UV\right\|_F &= \left\|(U + U' - U)V' - UV\right\|_F \\ &\leq \left\|U(V' - V)\right\|_F + \left\|(U' - U)V'\right\|_F \\ &= \left\|V' - V\right\|_F + \left\|U' - U\right\|_F , \end{aligned} \tag{5}$$

by using the fact that the Frobenius norm is invariant to orthogonal matrix multiplication. By iterating this relation, we obtain

$$\left\| \prod_{n\leq N} \bar{G}_n - \prod_{n\leq N} G_n \right\|_F \leq \sum_{n\leq N} \left\|\bar{G}_n - G_n\right\|_F . \tag{6}$$

Since $\bar{G}_n$ and G_n rotate in the same subspace,

$$\left\|\bar{G}_n - G_n\right\|_F = 2\sqrt{1 - \cos(\delta_n)} . \tag{7}$$

Inequality (4) follows from $\sqrt{1 - \cos(\delta_n)} \leq \delta_n \leq \delta$. □

Theorem 1. *Let $\epsilon > 0$. If $N = o\big(d^2/\log(d)\big)$, then as $d \to \infty$,*

$$\mu\left(\left\{U \in U(d) \middle| \inf_{G_1\ldots G_N} \|U - \prod_n G_n\|_2 \leq \epsilon\right\}\right) \to 0 .$$

Proof. Consider an ϵ-covering of the unitary group, i.e., a discrete set $\mathcal{X}$ such that $\inf_{X\in\mathcal{X}} \|U - X\|_2 \leq \epsilon$ for all $U \in U(d)$. Since the manifold dimension of the unitary group is $d(d-1)/2$, we need $|\mathcal{X}| = \Theta(\epsilon^{-d(d-1)/2})$ many balls for that cover. Let $N \coloneqq N(d)$ be the number of available Givens factors for approximation at dimension d, and $\mathcal{A}_N = \{X \in U(d) | \inf_{G_1\ldots G_N} \|X - \prod_{n\leq N} G_n\|_2 \leq \epsilon/2\}$ denote the set of unitary operators which can be effectively approximated with N Givens terms. Now, suppose that $\mu(\mathcal{A}_N) \geq c > 0$, i.e., the set of group elements admitting an $\epsilon/2$-approximation has positive measure. This implies that any ϵ-cover of $\mathcal{A}_N$ must be of size $\Theta(\epsilon^{-d(d-1)/2})$. Let us build such an ϵ-cover.

If we discretize the rotation angle to a value $\delta > 0$, then there are $(d(d-1)/2\delta)$ many different quantized Givens factors, denoted by $\bar{G}_i$, and consequently $(d(d-1)/2\delta)^N$ many different sequences. It follows that if $\delta \coloneqq \frac{\epsilon}{4N}$, the discrete set $\mathcal{Y} = \{\prod_{n\leq N} \bar{G}_{i_n}\}$ containing all possible sequences of length N of quantized Givens rotations is an

ϵ-cover of $\mathcal{A}_N$. Indeed, by using Lemma 1 and the fact that the operator 2-norm is bounded by the Frobenius norm, we have $\forall\, X \in \mathcal{A}_N$,

$$\|X - \prod_{n\le N} \bar{G}_n\|_2 \le \|X - \prod_{n\le N} G_n\|_2 + 2N\delta \le \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon\,.$$

Since $|\mathcal{Y}| = \left(\frac{2d(d-1)N}{\epsilon}\right)^N$, it follows that

$$\left(\frac{2d(d-1)N}{\epsilon}\right)^N = \Theta(\epsilon^{-d(d-1)/2})\,,$$

which implies $N = \mathcal{O}\left(d^2/\log d\right)$. □

An immediate consequence of Theorem 1 is that generic effective approximation, i.e., with a number of factors $N = \mathcal{O}\left(d \log d\right)$, is information theoretically impossible. However, the situation may be entirely different for structured distributions of unitary operators. For that purpose, we develop an algorithm to obtain effective approximations based on sparsity-inducing norms.

5. Givens Factorization and Coordinate Descent on $O(d)$

In this section, we offer an alternative viewpoint presented by Shalit & Chechik (2014) that interprets Givens factorization as manifold coordinate descent on the orthogonal group over a certain potential energy.

The orthogonal group $O(d)$ is a matrix Lie group with associated Lie algebra $\mathfrak{o}(d) = \mathrm{Skew}(d) = \{X \in \mathbb{R}^{d\times d} | X = -X^T\}$, the set of $d \times d$ skew-symmetric matrices (Hall, 2003). The tangent space at an element U is $T_U O(d) = \{XU | X \in \mathrm{Skew}(d)\}$ and the Riemannian directional derivative of a differentiable function f in the direction $XU \in T_U O(d)$ is given by

$$D_X f(U) = \tfrac{\mathrm{d}}{\mathrm{d}\alpha} f(\mathrm{Exp}(\alpha X)U)\Big|_{\alpha=0}\,, \tag{8}$$

where $\mathrm{Exp} : \mathfrak{o}(d) \to O(d)$ is the matrix exponential. If we choose the basis $\{X_{ij} = e_i e_j^T - e_j e_i^T | 1 \le i \le j \le d\}$ for the tangent space, then $D_{X_{ij}} f(U)$ represents the directional derivative in such a coordinate direction. A coordinate descent algorithm uses a criterion to choose coordinates (i, j) and a step size (rotation angle) α to iteratively update

$$U^{k+1} = \mathrm{Exp}(-\alpha X_{ij}) U^k\,. \tag{9}$$

A greedy criterion determines the best descent on f by a search over all possible coordinate directions $\{X_{ij}\}_{i\le j\le d}$ with the optimal step size obtained by a line search.

A Givens factor can be interpreted as a coordinate descent step over the orthogonal group. This follows from the relation

$$\mathrm{Exp}(-\alpha X_{ij}) = G^T(i, j, \alpha)\,. \tag{10}$$

In $d = 3$, an explicit example of the correspondence between Lie algebra and Lie group elements is

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -\alpha \\ 0 & \alpha & 0 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix}\,. \tag{11}$$

Suppose we want to minimize a function f over the orthogonal group,

$$\min_{U\in O(d)} f(U)\,. \tag{12}$$

Then minimizing (12) with manifold coordinate descent iterations (9) yields a Givens factorization of the initial point U^0. A truncated sequence leads to an approximate factorization. From this viewpoint, the quality of a Givens factorization can be controlled by properties of the function f. In the following, we construct an objective function that results in approximate factorization with less than $\mathcal{O}(d^2)$ factors.

6. Sparsity-Inducing Dynamics

To factorize a matrix $U \in O(d)$ one may choose it as an initial value to problem (12) when minimizing a suitable potential function f with manifold coordinate descent. We want to find a factorization up to signed permutation of the matrix columns. As the signed permutation matrices are the sparsest orthogonal matrices, we consider an energy function that quickly enforces sparsity, the element-wise L_1-norm of a matrix,

$$f(U) \coloneqq d^{-1}\|U\|_1 = d^{-1} \sum_{i,j=1}^{d} \left|U_{ij}\right|\,. \tag{13}$$

Although f is convex in $\mathbb{R}^{d^2}$ (since it is a norm), due to the non-convexity of the domain, the problem $\min_{U\in O(d)} f(U)$ is non-convex . The landscape of f characterizes the class of orthogonal matrices that admit effective Givens approximation. It is easy to see that the global minima of f in $O(d)$ consist of signed permutation matrices, with $\min f(U) = 1$, and the global maxima are located at Hadamard matrices, with $\max f(U) = \sqrt{d}$. A more involved question concerning the presence or absence of spurious local minima of f is of interest. The following proposition partially addresses this question by showing that critical points of f are necessarily located at $U \in O(d)$ with some of its entries set to zero.

Proposition 1. *Let $x \in \mathbb{R}^{2\times d}$ and let*

$$R(\alpha) \coloneqq \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{14}$$

be a counter-clockwise rotation in the plane by an angle α. Consider the function $g(\alpha) \coloneqq \left\|R(\alpha)x\right\|_1$. Then, at every local minimum α^* of g there exist indices k, l such that $\left(R(\alpha^*)x\right)_{kl} = 0$.

Proof. We show equivalently that any stationary point α^* with $\left(R(\alpha^*)x\right)_{kl} \neq 0 \ \forall k, l$ is a local maximum. At any such point the function g is twice continuously differentiable and the second derivative is

$$\left.\frac{\partial^2 g}{\partial \alpha^2}\right|_{\alpha=\alpha^*} = -g(\alpha^*) < 0 \,. \tag{15}$$

Consequently, any stationary point under this assumption must be a local maximum. □

Proposition 1 implies that for a given subspace (i, j), the best rotation angle can be found by checking all axis transitions for the 2D points $(u_{ik}, u_{jk}), k \in \{1, \dots d\}$ and selecting the angle that most minimizes the objective among them. It also implies that any local minimum of f must correspond to an orthogonal matrix with at least d zeros placed at specific entries, such that no two rows or columns have the same support. Indeed, Proposition 1 implies that there exists a continuous path $t \mapsto U(t) = G(i, j, \alpha(t))$ with $\alpha(0) = 0$, generated by a Givens rotation of angle $\alpha(t)$, such that $f(U(t))$ is non-increasing at $t = 0$, provided one can find two rows or columns of U with the same support. However, this result does not exclude the possibility that f has spurious local minima at matrices U with the above special sparsity pattern. In fact, we conjecture that the landscape of f does have spurious local minima.

A manifold coordinate descent on the objective function f is explicitly stated in Algorithm 1. The crucial step involves optimizing this objective in the rotation angle α for a given subspace (i, j), which is a non-convex optimization problem. Nevertheless, the global optimum can be found as stated by Proposition 1. In d dimensions, this step requires d operations. Consequently, due to the squared dimension dependence of the double for-loop, a naive implementation of Algorithm 1 would require $\mathcal{O}\left(d^3\right)$ operations . However, applying the selected Givens factor in each step changes only two rows of the matrix; thus, in the subsequent iteration, only those pairs of rows that involve the previously modified ones need to be re-computed. These are $\mathcal{O}\left(d\right)$ rows and altogether the runtime of an iteration is $\mathcal{O}\left(d^2\right)$.

Algorithm 1 Coordinate descent on the L_1-criterion

Input: initial value $U^0 \in O(d), f(U) = \|U\|_1$
repeat
  for $i = 1$ **to** d **do**
    for $j = 1$ **to** d **do**
      if α^*_{ij} not up-to-date **then**
        $\alpha^*_{ij} = \operatorname{argmin}_\alpha f(G^T(i, j, \alpha)U^k)$
      end if
    end for
  end for
  $i^*, j^* = \operatorname{argmin}_{i,j} f(G^T(i, j, \alpha^*_{ij})U^k)$
  $U^{k+1} = G^T(i^*, j^*, \alpha^*_{i^*j^*})U^k$
until $\left\|U^{k+1} - I\right\|_{F,\text{sym}} < \varepsilon$ or $maxIter$ is reached

7. Numerical Experiments

7.1. Planted Models

Theorem 1 shows that we cannot expect to find good approximations to Haar-sampled matrices with less than $\mathcal{O}\left(d^2/\log(d)\right)$ Givens factors. Therefore, we focus on a distribution for which we can control approximability. We use the uniform distribution over the set $\{U \in SO(d) | U = G_1 \cdots G_K, G_k = G(i_k, j_k, \alpha_k)\}$, where each G_k is obtained by first sampling a subspace uniformly at random (with replacement), and then sampling the corresponding angle uniformly from $(0, 2\pi)$. We denote the resulting distribution by the K-planted distribution ν_K. While this distribution may be sparse in the number of Givens factors for $K \ll d(d-1)/2$, this does not imply that the resulting matrices are sparse. In fact, products of Givens matrices become dense quickly. It follows from the Coupon Collector's Lemma that matrices generated with $\Theta(d \log_2(d))$ Givens factors are already dense with high probability. To visualize this effect, Figure 1 shows the L_0-norm as a function of planted Givens factors.

We compare the following factorization algorithms. A *greedy baseline* iteratively finds the Givens factor that most minimizes the objective (3). The *structured elimination* algorithm described in Section 3 yields a sequence of Givens factors that eliminate matrix entries in the order (2) and is guaranteed to find a perfect factorization with $d(d-1)/2$ factors. Our sparsity-inducing algorithm minimizes the L_1-criterion (13) via a *manifold coordinate descent* scheme.[2]

In an initial experiment, we demonstrate the approximation effectiveness of these algorithms; the results are shown in Figure 2. They indicate that minimizing the L_1-criterion improves over directly minimizing the Frobenius norm (greedy

[2] An implementation of these algorithms can be found at https://github.com/tfrerix/givens-factorization

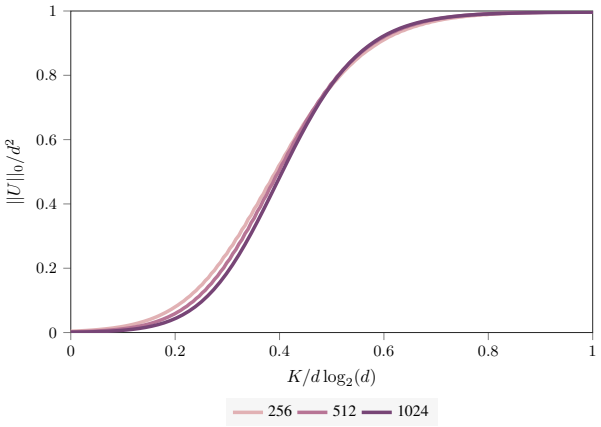


Figure 1. Average sparsity based on 100 samples of matrices drawn from the K-planted distribution over $SO(d)$ for increasing K. Standard deviation is negligible and not shown. Matrices become dense quickly as the number of planted Givens factors grows. In particular, matrices sampled from the $d\log_2(d)$-planted distribution are already dense.

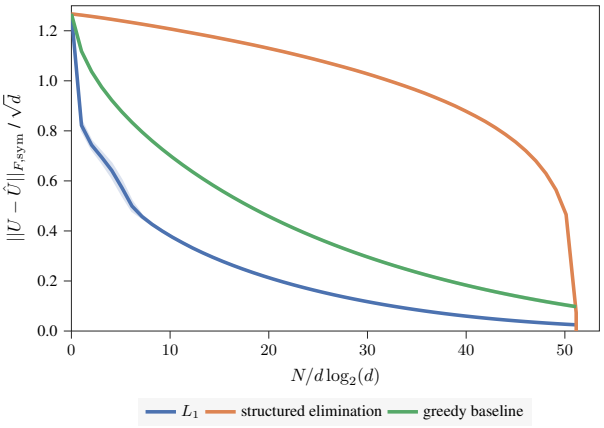


Figure 2. Average Frobenius norm approximation error in $d = 1024$ dimensions when factorizing 10 samples drawn from the $d\log_2(d)$-planted distribution over $SO(d)$ with $d(d-1)/2$ factors. Shaded area denotes standard deviation.

baseline). Next, we analyze the approximability of samples drawn from the K-planted distribution ν_K as a function of K. To obtain a Givens sequence, we factorize these samples with manifold coordinate descent on the L_1-objective (13). Along the optimization path, we define $N_\epsilon(U)$ as the number of Givens factors for which the normalized approximation error (3) is smaller than $\epsilon = 0.1$, i.e.,

$$N_\epsilon(U) := \min\left\{ N \,\middle|\, \frac{\|U - G_1 \dots G_N\|_{F,\mathrm{sym}}}{\sqrt{d}} < \epsilon \right\} \quad (16)$$

We refer to a Givens sequence with such $N_\epsilon(U)$ factors as an ϵ-factorizing sequence of U. In Figure 3, the sample average $N_\epsilon = n^{-1}\sum_{i=1}^n N_\epsilon(U_i)$ for $n = 10$ samples is shown as a function of K. We are interested in the rate at which N_ϵ grows for increasing K. The data in Figure 3 show that for $K = \alpha d\log_2(d)$ and $N_\epsilon = \beta d\log_2(d)$, the ratio β/α is not independent of d. For the shown dimension regime this implies that for $K = \mathcal{O}\left(d\log(d)\right)$, N_ϵ grows polynomial in d, albeit with small rate for few planted factors. To make this relation more precise, we extract the exponent η of a model $N_\epsilon \sim d^\eta$. Figure 4 shows that the growth is slightly superlinear in the few-factor regime and becomes quadratic towards $K = d\log_2(d)$. Analytically characterizing such growth is left for future work.

That said, our initial results suggest the existence of a computational-to-statistical gap for the recovery (or detection) of sparse planted Givens factors. Indeed, Theorem 1 proves that recovery with $K = \mathcal{O}\left(d^2/\log d\right)$ planted factors is information-theoretically possible, whereas our greedy recovery strategy is only effective for $K = \mathcal{O}\left(d\log d\right)$. The mathematical analysis of our coordinate descent algorithm in the regime where effective approximation is feasible is beyond the scope of the present paper. In particular, proving that $N_\epsilon = \mathcal{O}\left(d\log d\right)$ is sufficient when $K \lesssim d\log d$ remains an open question.

7.2. Application: Graph Fourier Transform

The method introduced in this paper is useful in situations where one at first computes an approximation to a unitary operator, which is subsequently applied many times. Hence, the trade-off between initial computation and approximation on the one hand and efficient application on the other hand is in favor of the latter. Canonical examples for this scenario are orthogonal basis transforms. In this paper, we draw motivation from the FFT, which yields a speed-up of applying a Fourier transformation over a regular grid domain from $\mathcal{O}(d^2)$ to $\mathcal{O}(d\log(d))$ time complexity (Cooley & Tukey, 1965). However, these speed-ups do not carry over when the domain is unstructured, such as general graphs. Here, we compute an effective approximation of the graph Fourier transformation (GFT). Consider a simple, undirected graph with degree matrix D and adjacency matrix A. The unnormalized graph Laplacian is defined as $L := D - A$, which is a positive semi-definite, symmetric matrix. The GFT is represented by the orthogonal matrix that diagonalizes L.

A baseline for our method is the Jacobi algorithm (Jacobi, 1846), which diagonalizes a symmetric matrix L by greedily minimizing the off-diagonal squared Frobenius norm,

$$\mathrm{off}(L) := \|L\|_F^2 - \sum_{k=1}^d L_{kk}^2 \,. \quad (17)$$

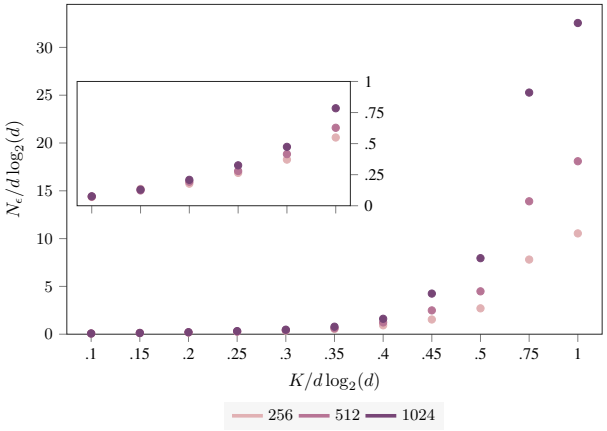


Figure 3. Average number of Givens factors necessary to factorize a K-planted matrix in $d \in \{256, 512, 1024\}$ dimensions up to desired accuracy as a function of K. Here, $\epsilon = 0.1$ is the accuracy as defined in expression (16). Note that the x-axis is shown with unequal spacing to highlight the relevant regime of the data. The inset plot shows a zoom of the first data points.

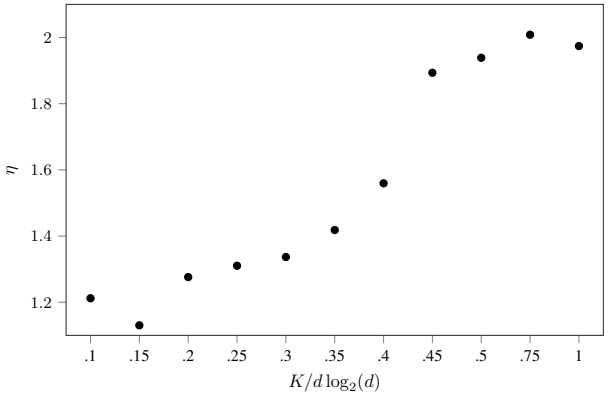


Figure 4. Polynomial growth rate η of the model $N_\epsilon \sim d^\eta$ as a function of the number of planted factors estimated from $d \in \{256, 512, 1024\}$. Note that the x-axis is shown with unequal spacing to highlight the relevant regime of the data.

This is achieved by zeroing-out the largest matrix element in absolute value at every iteration. To this end, a Givens matrix similarity transformation with a suitably chosen rotation subspace and rotation angle is applied. However, the Jacobi algorithm does not guarantee factorization in a finite number of steps; in particular, it may take more than $N = d(d-1)/2$ iterations. In fact, the algorithm converges linearly (Golub & Van Loan, 2012),

$$\mathrm{off}(L^{k+1}) \leq \left(1 - \frac{1}{N}\right) \mathrm{off}(L^k)\ . \tag{18}$$

If the iteration number k is large enough, quadratic convergence was shown by Schönhage (1964). Hence, the method is ineffective for small iteration numbers and in high dimensions. A truncated version of this algorithm was used by Magoarou et al. (2018) to obtain an approximation to the GFT. The objective (17) of the Jacobi method is motivated by approximating the spectrum of the symmetric matrix through the Gershgorin circle theorem (Gershgorin, 1931). However, we argue here that a criterion focused on approximating the eigenbasis of the symmetric matrix directly yields a more effective approximation to this orthogonal basis transformation. We consider the eigendecomposition $L = U\Lambda U^T$ and compute an approximation of the orthogonal matrix U with the algorithms outlined in Section 7.1. We demonstrate this procedure on Barabási-Albert random graphs and several real world graphs.

The Barabási-Albert model starts with n_0 unconnected vertices and iteratively adds vertices to the graph, which are connected to a number m of already existing ones with a probability proportional to the degree of these vertices.

Table 1. Construction of Barabási-Albert graphs. An n-vertex graph is constructed by choosing $n_0 = m_k$ initial vertices, then adding vertices and connecting them to m_k of already existing ones with a probability proportional to the degree of these vertices. m_k is chosen such that the number of resulting edges is approximatly $k \cdot 0.25n(n-1)/2$.

n	64	128	256	512	1024
m_1	54	109	218	437	874
m_2	36	69	136	267	528

This construction is known as preferential attachment and induces a scale-free degree distribution found in real world graphs (Barabási & Albert, 1999). The details of generating these graphs are described in Table 1. We approximate the corresponding graph Laplacians with $n\log_2(n)$ factors leading to the results shown in Figure 5. While our sparsity-inducing algorithm yields better factorizations in most cases, there exist scenarios, where the greedy baseline results in better approximations ($d \in \{512, 2014\}$ for $\sim 0.25n\log_2(n)$ edges). Finally, we demonstrate approximate factorization of the graph Laplacian of various real world graphs listed in Table 2. Our L_1-algorithm yiels the best factorization for the Minnesota, HumanProtein, and EMail graphs, while the greedy baseline algorithm is superior for the Facebook graph.

A simple strategy to improve the performance of our L_1 greedy method with mild computational overhead is to perform beam-search, which is beyond the scope of this paper. Overall, it remains an open question to more closely characterize the graphs for which our sparsity-inducing algorithm yields effective approximations of the GFT.

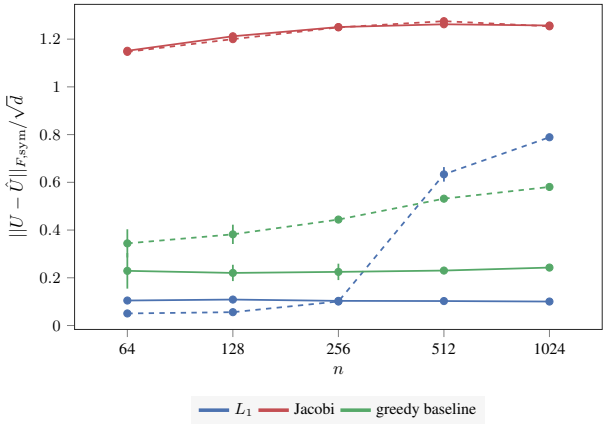


Figure 5. Approximate factorization of the graph Laplacian of n-vertex Barabási-Albert graphs with $n \log_2(n)$ factors. Data points are averages of 10 samples, vertical lines denote standard deviation. The solid (–) lines show factorizations of graphs with $\sim 0.5n(n-1)/2$ edges, while the dashed (- -) lines show factorizations of graphs with $\sim 0.25n(n-1)/2$ edges.

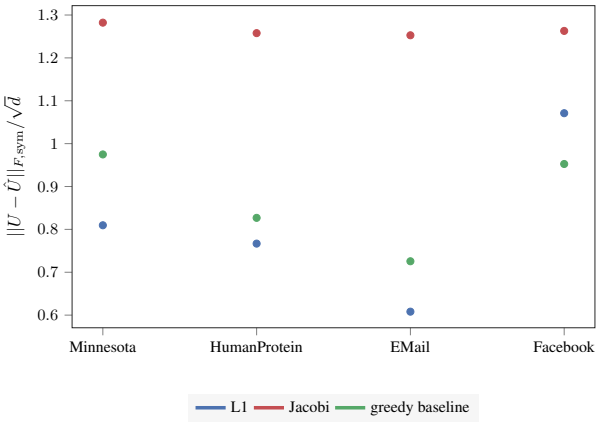


Figure 6. Approximate factorization of the graph Laplacian of various n-vertex real world graphs with $n \log_2(n)$ factors.

Table 2. GFT approximation for real world graphs with n vertices and n_e edges.

	n	n_e
MINNESOTA (Defferrard et al.)	2642	3304
HUMANPROTEIN (Rual et al., 2005)	3133	6726
EMAIL (Guimerà et al., 2003)	1133	5451
FACEBOOK (McAuley & Leskovec, 2012)	2888	2981

8. Discussion

We analyzed the problem of approximating orthogonal matrices with few Givens factors. While a perfect factorization in $\mathcal{O}\left(d^2\right)$ is always possible, an approximation with fewer factors is advantageous if the orthogonal matrix is applied many times. We showed that effective Givens factorization of generic orthogonal matrices is impossible and inspected a distribution of planted factors, which allows us to control approximability. Our initial results suggest that sparsity inducing factorization is promising beyond the sparse matrix regime. However, it remains an open problem to further characterize the matrices that admit effective factorization using manifold coordinate descent on an L_1-criterion.

This work opens up questions we believe are important both from a theoretical and an applied perspective. On the theory side, important problems arising from our analysis are: *(i)* a complete description of the landscape of $f(U) = \|U\|_1$ over the orthogonal and unitary groups, *(ii)* a precise classification of the detection threshold $K(d)$ below which it is possible to discriminate a K-planted sample from a Haar sample in polynomial time, and *(iii)* a guarantee that the proposed sparse Givens coordinate descent algorithm requires $N = \Theta(d \log d)$ terms for $K \leq Cd \log d$ for some constant $C > 0$. These questions suggest a learning approach whereby our sparsity promoting potential f would be replaced by a classifier f_θ trained to discriminate between K-planted and Haar distributions. From an applied perspective, the method allows to approximately invert a time-varying symmetric linear operator $H(t)$. Similar to the Woodbury formula for low-rank updates of an inverse, one could set up an approximate Givens factorization of the eigenbasis of $H(t_0)$, and update it efficiently at subsequent times. If successful, this could dramatically improve the efficiency of second-order optimization schemes, where $H(t)$ is the Hessian of a loss function.

Acknowledgements

The authors would like to thank Oded Regev for in-depth discussions and early feedback, as well as Kyle Cranmer and Lenka Zbedorova for fruitful discussions on the topic, and Yann LeCun for introducing us to the problem. This work was partially supported by NSF grant RI-IIS 1816753, NSF CAREER CIF 1845360, the Alfred P. Sloan Fellowship and Samsung Electronics.

References

Barabási, A.-L. and Albert, R. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34 (4):18–42, 2017.

Cooley, J. W. and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19: 297–301, 1965.

Defferrard, M., Martin, L., Pena, R., and Perraudin, N. Pygsp: Graph signal processing in python. description: https://www.cise.ufl.edu/research/sparse/matrices/Gleich/minnesota.html.

Gershgorin, S. A. Über die Abgrenzung der Eigenwerte einer Matrix. *Izv. Akad. Nauk. USSR Otd. Fiz.-Mat. Nauk*, (6):749–754, 1931.

Givens, W. Computation of Plain Unitary Rotations Transforming a General Matrix to Triangular Form. *Journal of the Society for Industrial and Applied Mathematics*, 6(1): 26–50, 1958.

Golub, G. H. and Van Der Vorst, H. A. Eigenvalue Computation in the 20th Century. *Journal of Computational and Applied Mathematics*, 123(1-2):35–65, 2000.

Golub, G. H. and Van Loan, C. F. *Matrix computations*. JHU Press, 4th edition, 2012.

Guimerà, R., Danon, L., Díaz-Guilera, A., Giralt, F., and Arenas, A. Self-similar Community Structure in a Network of Human Interactions. *Phys. Rev. E*, 68 (6):065103, 2003. retrieved from http://konect.uni-koblenz.de/networks/arenas-email.

Hall, B. *Lie Groups, Lie Algebras, and Representations*. Graduate Texts in Mathematics. Springer, 2003.

Jacobi, C. Über ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Journal für die reine und angewandte Mathematik*, 30:51–94, 1846.

Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd edition, 1998. Ex 5.2-10, p.80.

Kondor, R., Teneva, N., and Garg, V. Multiresolution matrix factorization. In *International Conference on Machine Learning (ICML 2014)*, pp. 1620–1628, 2014.

Kyng, R. and Sachdeva, S. Approximate Gaussian Elimination for Laplacians: Fast, Sparse, and Simple. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pp. 573–582, 2016.

Magoarou, L. L., Gribonval, R., and Tremblay, N. Approximate Fast Graph Fourier Transforms via Multilayer Sparse Approximations. *IEEE Transactions on Signal and Information Processing over Networks*, 4(2):407–420, 2018.

McAuley, J. and Leskovec, J. Learning to Discover Social Circles in Ego Networks. In *Advances in Neural Information Processing Systems (NIPS 2012)*, pp. 548–556. 2012. retrieved from http://konect.uni-koblenz.de/networks/ego-facebook.

Rual, J.-F., Venkatesan, K., Hao, T., Hirozane-Kishikawa, T., Dricot, A., Li, N., Berriz, G. F., Gibbons, F. D., Dreze, M., and Ayivi-Guedehoussou, N. Towards a Proteome-scale Map of the Human Protein–Protein Interaction Network. *Nature*, (7062):1173–1178, 2005. retrieved from http://konect.uni-koblenz.de/networks/maayan-vidal.

Schönhage, A. Zur quadratischen Konvergenz des Jacobi-Verfahrens. *Numerische Mathematik*, 6(1):410–412, 1964.

Shalit, U. and Chechik, G. Coordinate-Descent for Learning Orthogonal Matrices through Givens Rotations. In *Proceedings of the 31st International Conference on Machine Learning (ICML 2014)*, 2014.

Shuman, D. I., Narang, S. K., Frossard, P., Ortega, A., and Vandergheynst, P. The Emerging Field of Signal Processing on Graphs: Extending High-Dimensional Data Analysis to Networks and other Irregular Domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.

Sullivan, F. and Dongarra, J. Guest Editors' Introduction: The Top 10 Algorithms. *Computing in Science and Engineering*, 2:22–23, 2000.

Vartiainen, J. J., Möttönen, M., and Salomaa, M. M. Efficient Decomposition of Quantum Gates. *Phys. Rev. Lett.*, 92:177902, 2004.

Variational Data Assimilation with a Learned Inverse Observation Operator

Thomas Frerix, Dmitrii Kochkov, Jamie A. Smith, Daniel Cremers, Michael P. Brenner, and Stephan Hoyer. Variational Data Assimilation with a Learned Inverse Observation Operator. In: *Proceedings of the 38th International Conference on Machine Learning (ICML)*. 2021

Variational Data Assimilation with a Learned Inverse Observation Operator

Thomas Frerix [1 2] **Dmitrii Kochkov** [1] **Jamie A. Smith** [1] **Daniel Cremers** [2]
Michael P. Brenner [1 3] **Stephan Hoyer** [1]

Abstract

Variational data assimilation optimizes for an initial state of a dynamical system such that its evolution fits observational data. The physical model can subsequently be evolved into the future to make predictions. This principle is a cornerstone of large scale forecasting applications such as numerical weather prediction. As such, it is implemented in current operational systems of weather forecasting agencies across the globe. However, finding a good initial state poses a difficult optimization problem in part due to the non-invertible relationship between physical states and their corresponding observations. We learn a mapping from observational data to physical states and show how it can be used to improve optimizability. We employ this mapping in two ways: to better initialize the non-convex optimization problem, and to reformulate the objective function in better behaved physics space instead of observation space. Our experimental results for the Lorenz96 model and a two-dimensional turbulent fluid flow demonstrate that this procedure significantly improves forecast quality for chaotic systems.

1. Introduction

Variational data assimilation provides the basis for numerical weather prediction (ECMWF, 2019), integrating the non-linear partial differential equations describing the atmosphere. The core algorithm is an optimization problem for the initial state of the system, such that when the equations of motion are evolved over time, the resulting trajectories are close to the measurements. Continuing to evolve the physical system into the future then yields a forecast (Figure 1a). Over the last decades, these algorithms have led to a steady improvement in forecast quality, though further improvements are limited by computational resources. Data assimilation accounts for a significant fraction of the computational cost for numerical weather prediction. This restricts the amount of data that can be assimilated and only a small volume of available satellite data is utilized for operational forecasts (Bauer et al., 2015; Gustafsson et al., 2018).

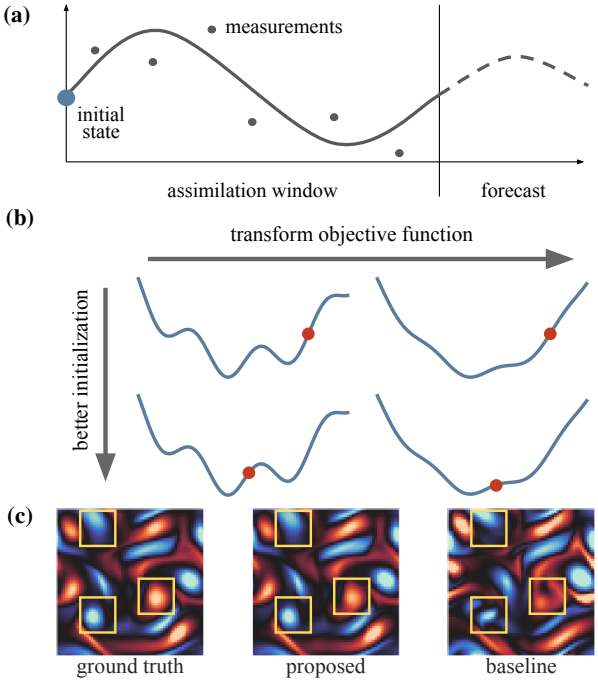


Figure 1. Overview of the proposed method. **(a)** The principle of variational data assimilation. The goal is to optimize for an initial state (large blue dot) of a physical system such that the evolution over an assimilation window (solid line) is close to measurements (small gray dots). The model is subsequently used to make predictions into the future (dashed line). **(b)** Improving optimizability of variational data assimilation. We use a learned inverse observation operator to better initialize the optimization problem (red dots) and to transform the non-convex objective function to be better behaved. **(c)** Data assimilation results of the proposed method compared with a traditional algorithm. Depicted is a vorticity prediction of a two-dimensional turbulent fluid flow. The proposed method more accurately captures vorticity features (yellow squares).

[1]Google Research [2]Technical University of Munich [3]Harvard University. Correspondence to: Thomas Frerix <thomas.frerix@tum.de>, Stephan Hoyer <shoyer@google.com>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

Weather forecasting systems are complex algorithmic pipelines (Bauer et al., 2015). Recent work has shown that in some cases forecasts can be improved by replacing the entire system with a machine learned prediction (see, e.g., (Sønderby et al., 2020; Ham et al., 2019)). This approach is very powerful, but physical models remain more accurate for global weather forecasting (Rasp et al., 2020). Moreover, they offer guarantees of generalization, interpretability and physical consistency because they are built upon well-known physical principles. In fact, some of the best pure machine learning approaches rely upon pre-training with simulation data due to insufficient historical observations (Rasp & Thuerey, 2020; Ham et al., 2019). Additionally, physical modeling facilitates the principled coupling of processes on different characteristic spatial and time scales, e.g., the atmosphere, ocean, and land surfaces, which is critical for complex forecasting applications (Bauer et al., 2015). Consequently, a more promising approach may be to *augment* physical models with machine learning (Watt-Meyer et al., 2021; Kochkov et al., 2021).

In this paper, we augment a traditional variational data assimilation algorithm with machine learning. We use the equations of motion to evolve the dynamical system, while machine learning is used only to improve the optimization problem for calculating the initial state. To this end, we learn an approximate inverse to the observation operator. Using this mapping, we provide an effective initialization scheme for the non-convex optimization problem and transform the objective function for the variational data assimilation problem to be better behaved (Figure 1b). We generate observational data from two model problems, a classical model for data assimilation introduced by Lorenz (Lorenz, 1995), and a turbulent fluid flow in two spatial dimensions. We demonstrate that the algorithm enhanced with machine learning leads to a substantial performance improvements over the baseline. Figure 1c shows an example of an improved forecast.

2. Related Work

Data assimilation is a suitable formalism for combining physical modeling with machine learning since large scale applications are characterized by rich physics and large amounts of data. Both approaches can be viewed in the framework of Bayesian inference (Geer, 2020). Machine learning approaches to modify the physical model for data assimilation include a learned correction to an approximate model (Farchi et al., 2020; Brajard et al., 2020a), training a machine learning model to completely emulate the physics (Brajard et al., 2020b), and learning a forcing term within the weak-constraint 4D-Var formulation (Bonavita & Laloyaux, 2020). In contrast, we use an exact physical model and modify the representation of observations using machine learning. Mack et al. (2020) formulate variational data assimilation in a latent space derived by training an autoeconder. The dimensionality reduction allows for significantly faster optimization. However, this approach loses physical guarantees for decoded states.

Integration of dynamical systems is a central component of data assimilation. However, simulating high-resolution dynamics quickly becomes computationally intractable. To ameliorate this issue, several recent works combine traditional numerical solvers with machine learning to obtain high-resolution physics from coarser simulations. MeshfreeFlowNet (Jiang et al., 2020) continuously parameterizes the spatial domain by learning an interpolation function for each grid cell. Um et al. (2020) incorporate a correction operator directly into the numerical solver and train this function to nudge an inaccurate solution towards a more accurate one. The authors of (Bar-Sinai et al., 2019; Zhuang et al., 2020) learn a discretization scheme for PDEs that better captures the unresolved physics, leading to improvements over ad hoc finite difference discretization methods. Using a fully differentiable computational fluid solver, Kochkov et al. (2021) demonstrate that with this approach the grid resolution can be reduced by an order of magnitude without sacrificing accuracy. Similarly, we use a fully differentiable solver for our model systems and our approach may therefore be complemented by such ideas.

Variational data assimilation requires solving a difficult optimization problem. Our approach of improving optimizability of this problem with machine learning can be contextualized with other works that employ machine learning to transform a physics constrained optimization problem. In the context of simulating mechanical materials, Beatson et al. (2020) approximate the inner problem of a bi-level optimization problem by a learned function, thus crucially reducing the the computation cost. To optimize photonic device designs, Kudyshev et al. (2021) train for a compressed design space with an adversarial autoencoder. This space is then explored using an evolutionary algorithm. Ackmann et al. (2020) learn a preconditioner to improve the solution of a linear system arising during the integration of a shallow-water model. As with our approach, the preconditioned system does not suffer from generalization issues of the machine learning model. We can guarantee a certain performance level by defaulting to a classical method. Various works improve optimization problems not with a component learned from training data, but by reparameterizing the objective function with a neural network architecture. The neural network here acts as an overparameterization with a specific inductive bias, e.g., convolutional neural networks for building hierarchical, multi-scale representations (Hoyer et al., 2019; Ulyanov et al., 2018) or fully-connected networks for continuous representations (Mildenhall et al., 2020).

3. Variational Data Assimilation

The state of the art variational data assimilation algorithm is called 4D-Var (Bannister, 2017). It minimizes an objective function of the form

$$\begin{aligned} J(x_0) =& (x_0 - x^b)^T \mathbf{B}^{-1} (x_0 - x^b) \\ &+ \sum_{t=0}^{T} (\mathcal{H}(x_t) - y_t)^T \mathbf{R}^{-1} (\mathcal{H}(x_t) - y_t) \\ x_{t+1} =& \mathcal{M}(x_t). \end{aligned} \tag{1}$$

The goal is to produce a maximum likelihood estimate of the initial state x_0 of a trajectory $(x_1, \ldots, x_T)$ that is evolved through a physics model $\mathcal{M}$, given a sequence of observations $(y_1, \ldots, y_T)$. The observation operator $\mathcal{H}$, maps physical states into the space of observations. As an example, the physics model could be the Navier Stokes equations for evolving a weather system, and the observation operator could measure the state of the atmosphere at discrete weather stations. The loss $J(x_0)$ models the initial condition and conditional distribution of observations as a multi-variate normal distribution. The first term incorporates a guess for the initial state x_0 (the so-called background state x^b), where $\mathbf{B}$ is the background covariance matrix, representing the uncertainty about this assumption, i.e., $x_0 \sim \mathcal{N}(x^b, \mathbf{B})$. Similarly, the matrix $\mathbf{R}$ models the observation error covariance, i.e., $y_t \sim \mathcal{N}(\mathcal{H}(x_t), \mathbf{R})$. The simplifying assumption of Gaussian background and observation errors may suffer from model misspecification when applied to real-world data (Bocquet et al., 2010). Altogether, this amounts to solving a non-linear least squares problem. We denote the initialization of this optimization problem by *initial condition* and refer to an *initial state* to describe the first state x_0 of a physics trajectory.

4D-Var minimizes objective functions of the form (1) via gradient based optimization. To forecast a trajectory $(x_1, \ldots, x_{T'})$, the objective (1) is minimized to estimate x_0 over a fixed-length window of observations, the so-called assimilation window, which is shifted in time. The forecast state from a previous assimilation window becomes the background state for the next assimilation window. Minimizing (1) is a difficult optimization problem for various reasons (Andersson et al., 2005): First, the physics model $\mathcal{M}$ is in general non-linear or even chaotic, so that small changes in the initial state can lead to large changes in an integrated state. Secondly, the observation operator $\mathcal{H}$ that reduces information from physics trajectories to observations is usually non-invertible and possibly non-linear.

In what follows, we focus on the difficulty posed by the observation operator $\mathcal{H}$, and learn an approximate inverse to $\mathcal{H}$ that maps the observational data to the space of physical states. For simplicity, we focus our analysis on a fixed time horizon without a shifting assimilation window. Moreover, we neglect prior modeling of the initial state, so that we omit the first term in (1). Finally, we neglect an explicit observation noise model, i.e., we set $\mathbf{R}$ to be the identity matrix. This amounts to studying the following simplified version of the 4D-Var model:

$$J(x_0) = \sum_{t=0}^{T} ||\mathcal{H}(x_t) - y_t||_2^2, \qquad x_{t+1} = \mathcal{M}(x_t) \tag{2}$$

The method presented in this paper is not restricted to this setting, but equally applies to the general 4D-Var problem. However, to study the effect of the observation operator $\mathcal{H}$ on the optimization problem, additional aspects of the problem are not necessary.

4. Learning an Inverse Observation Operator

To be precise, we distinguish the space $\mathcal{P}$ of physical states or *physics space* and the space $\mathcal{O}$ of observations or *observation space*. The observation operator $\mathcal{H} : \mathcal{P} \to \mathcal{O}$ maps the physics space $\mathcal{P}$ to the observation space $\mathcal{O}$. The variational data assimilation objective (2) is formulated in observation space. However, the non-invertibility (and potential non-linearity) of $\mathcal{H}$ makes minimizing this objective difficult. The key idea of this paper is to parameterize an approximate inverse $h_\theta : \mathcal{O} \to \mathcal{P}$ and to use machine learning to train the parameters θ. The training target is to map observations to corresponding physical states, in our notation to obtain $h_\theta(y_t) \approx x_t$. While in practice there is ample training data from historical observations, in this work we revert to simulations for generating training data.

In order to exploit both spatial and temporal correlations, we construct a fully-convolutional architecture in space and time. Fully-convolutional architectures are natural for several reasons: they use local filters and therefore enforce the locality of the underlying equations of motion. Additionally, the number of parameters in a convolutional layer does not increase with input size. This is vital because typical physics models $\mathcal{M}$ are discretized over large grids.

We implement our models for the approximate inverse in JAX (Bradbury et al., 2018) and use Flax as neural network library, with the Adam optimizer (Kingma & Ba, 2015) and learning rate of 10^{-3} for training[1]. We also use JAX to implement differentiable simulators for the physics model $\mathcal{M}$ that arises inside the objective function (2) (Kochkov et al., 2021). Operational numerical weather prediction models similarly make use of differentiable simulators, where they are known as adjoint models. All models can be trained and optimized on a single NVIDIA V100 GPU.

We use the trained inverse observation operator for two aspects of the optimization problem. First, we map the

[1] https://github.com/googleinterns/invobs-data-assimilation

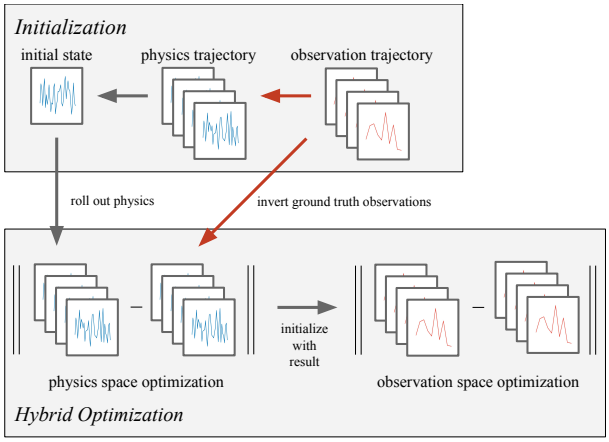


Figure 2. Variational data assimilation with a learned inverse observation operator. The learned inverse observation mapping is denoted by red, hollow arrows. We approximately invert an observation trajectory and choose its first state as an *initialization* of the non-convex optimization problem. The *hybrid optimization* approach first minimizes (3) in physics space and subsequently uses the optimization result to initialize refinement minimization of (2) in observation space.

earliest trajectory of observations to a trajectory in physics space and then use its first state as an initialization to the optimization problem. Secondly, we substitute (2) with a reformulated objective function in physics space:

$$\tilde{J}(x_0) = \sum_{t=0}^{T} ||x_t - h_\theta(y_t)||_2^2, \qquad x_{t+1} = \mathcal{M}(x_t) \quad (3)$$

An overview of this method is depicted in Figure 2. The objective functions (2) and (3) are not equivalent, rather we use (3) as a proxy for (2). As we will show in Section 5, minimizing (3) is a more benign optimization problem compared to minimizing (2). However, the caveat with minimizing (3) is that we can only expect h_θ to be an approximation that does not even guarantee to map to a physical state, i.e., a state that one could encounter under the statistically stationary dynamics of the model. As a consequence, we adopt a *hybrid* approach where we first minimize (3) and use the optimization result to initialize minimizing (2) for further refinement.

5. Results

We demonstrate this approach on two chaotic dynamical systems, the Lorenz96 model (Lorenz, 1995) and Kolmogorov flow (Chandler & Kerswell, 2013). As our baseline, we follow a common approach (Bannister, 2008) and assimilate in observation space over a set of uncorrelated variables, i.e., we minimize (2) after a whitening transformation to $\xi = C^{-1/2}x$, where C is the empirical covariance matrix over a set of 10^6 independent samples from the stationary distribution of the respective dynamical systems. The empirical covariance matrix might not be positive definite, an issue that is often encountered in applications (Tabeart et al., 2020). To ensure positive definiteness, we threshold the spectrum of C at 10^{-6}. To be precise, we solve

$$\min_{\xi_0} \sum_{t=0}^{T} ||\mathcal{H}(C^{1/2}\xi_t) - y_t||_2^2, \; \xi_{t+1} = C^{-1/2}\mathcal{M}(C^{1/2}\xi_t) \; . \quad (4)$$

We use L-BFGS (Nocedal & Wright, 2006) as an optimizer for assimilation, retaining a history of 10 vectors for the Hessian approximation. An optimization step in physics and observation space incurs a comparable computational cost, since the inverse observation operator is applied *prior* to optimization to modify the fitting targets.

We measure the quality of forecasts by an L_1 point-wise error metric ε between two states z_1, z_2:

$$\varepsilon(z_1, z_2) := \|z_1 - z_2\|_1/\gamma \; , \quad (5)$$

where we scale this metric to a *relative* error by dividing by a mean error γ of random independent states sampled from the stationary distribution of the dynamical system. This metric can be easily interpreted: an order unity error implies the average performance of a random evolution of the system. We compare the optimized forecasts with the evolution from a ground truth initial state on a set of 100 test trajectories.

5.1. Lorenz96 Model

The single-level Lorenz96 model (Lorenz, 1995) is a periodic, one-dimensional model where each grid point is evolved according to the equation of motion

$$\frac{dX_k}{dt} = -X_{k-1}(X_{k-2} - X_{k+1}) - X_k + F \; . \quad (6)$$

Here, the first term models advection, the second term represents a linear damping, and F is an external forcing. We choose a grid of size $K = 40$ and an external forcing $F = 8$, parameters where the system is chaotic with a Lyapunov time of approximately 0.6 time units. For an observation operator, we use subsampling. We integrate trajectories over an assimilation window of $T = 10$ time steps with a time increment of $\Delta t = 0.1$ time units starting from an initial condition in the statistically stationary regime, i.e., where $\sum_k X_k^2$ fluctuates around a constant mean value.

We now demonstrate how a learned inverse observation operator significantly improves forecast results by providing an effective initialization scheme for the non-convex optimization problem and by formulating a more benign objective function in physics space $\mathcal{P}$ instead of observation space $\mathcal{O}$.

LAYER	(T, X, C)
INPUT	(10, 10, 1)
CONV2D + BN + SILU	(10, 10, 128)
UPSAMPLE + CONV2D + BN + SILU	(10, 20, 64)
UPSAMPLE + CONV2D + BN + SILU	(10, 40, 32)
CONV2D + BN + SILU	(10, 40, 16)
CONV2D	(10, 40, 1)

Table 1. Fully-convolutional network for training the inverse observation operator for the Lorenz96 model. The table shows a layer with its respective output array dimensions time (T), space (X), and channel (C). The CONV2D layer applies periodic convolution in the space dimension and zero-padded convolution in the time dimension. The filter size for all convolutional layers is $(3, 3)$. BN denotes batch normalization. To upscale the grid by a factor of 2 in layers two and three, we use cubic interpolation. As a non-linearity we use the sigmoid-weighted linear unit (SILU), $\text{silu}(x) = x/(1 + \exp(-x))$.

As an observation operator for the following experiments, we observe every 4th grid point. To approximate the inverse observation operator, we train a fully-convolutional network as described in Table 1. We train on a dataset of 32000 independent observation trajectories with batch size 8 for 500 epochs.

For data assimilation, we compare two initialization schemes. The baseline *averaging initialization* scheme initializes the optimizer with the observed grid points and uses the average over a data set of independent states as an estimate of the unobserved grid points. This is equivalent to a least-squares fit of the unobserved grid points. The *inverse initialization* scheme uses the learned inverse observation operator to create the initialization. To this end, we map a sequence of observations to a physical trajectory and use its first state for initialization as depicted in Figure 2. Figure 3 shows a qualitative comparison of these two initialization schemes, demonstrating that the learned inverse mapping leads to a much more accurate initialization. We found that first optimizing for an initial condition from a previous assimilation window, as in 4D-Var, does not improve baseline initialization. We compare optimizing in observation space (baseline) with the hybrid approach of first optimizing in physics space and then refining the results in observation space. For a fair comparison, both optimization methods are limited to 500 optimization steps with the hybrid method assigning 100 of these steps to optimization in physics space and the remaining 400 steps to refinement in observation space. The forecast results are shown in Figure 4. Inverse initialization improves forecasts for observation space optimization compared with average initialization. For the inaccurate averaging initialization, hybrid optimization significantly improves forecast quality compared with observation space optimization. This suggests that by first optimizing in physics space, we obtain an initialization for refinement in observation space that is located at a favorable basin of attraction. Adding inverse initialization to the hybrid optimization approach leads to a small additional improvement.

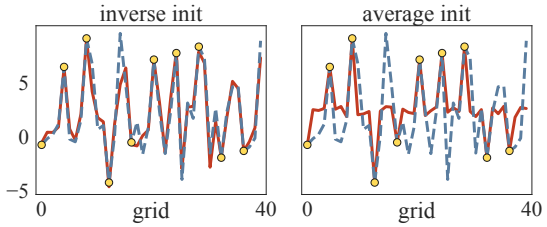


Figure 3. Comparison of initialization (solid red) with the ground truth initial state (dashed blue). The observed grid points are marked as yellow dots. The learned inverse observation mapping takes a trajectory of such subsampled states as input and generates the inverse initialization. Inverse initialization is much more accurate than averaging initialization.

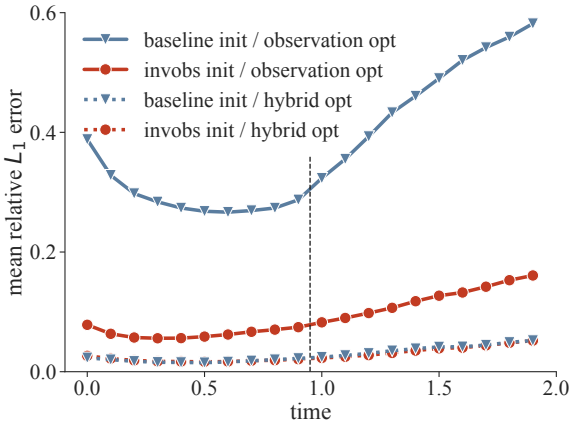


Figure 4. Forecast quality with a learned inverse observation operator for the Lorenz96 model. Quality measure is the L_1 forecast error relative to a random evolution of the system. Depicted is the mean error based on a sample of 100 trajectories. The vertical dashed line separates the assimilation window from the forecast window. Inverse initialization improves forecasts for observation space optimization compared with average initialization. For the inaccurate averaging initialization, hybrid optimization significantly improves forecast quality compared with observation space optimization. Adding inverse initialization to the hybrid optimization approach leads to a small additional improvement, which is significant with a p-value of $p < 10^{-4}$.

Figure 5 shows an example forecast of the system. The hybrid method initialized with the learned inverse mapping is able to capture the ground truth evolution of the system. In contrast, for the baseline method a visible approximation error remains throughout the system integration.

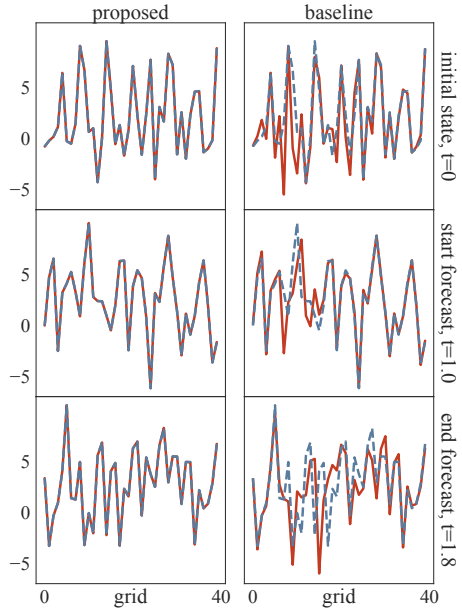


Figure 5. Forecast trajectory for the Lorenz96 model optimized from the initial conditions depicted in Figure 3. The forecast trajectory based on optimization with the inverse observation operator (inverse initialization, hybrid optimization) is qualitatively much closer to the ground truth evolution of the system than the baseline method (averaging initialization, observation space optimization).

LAYER	(T, X, Y, C)
INPUT	(10, 4, 4, 2)
CONV3D + BN + SILU	(10, 4, 4, 64)
UPSAMPLE + CONV3D + BN + SILU	(10, 8, 8, 32)
UPSAMPLE + CONV3D + BN + SILU	(10, 16, 16, 16)
UPSAMPLE + CONV3D + BN + SILU	(10, 32, 32, 8)
UPSAMPLE + CONV3D + BN + SILU	(10, 64, 64, 4)
CONV3D	(10, 64, 64, 2)

Table 2. Fully-convolutional network for training the inverse observation operator for Kolmogorov flow. The table shows a layer with its respective output array dimensions time (T), space (X and Y), and channel (C). The CONV3D layer applies periodic convolution in the two space dimensions and zero-padded convolution in the time dimension. The filter size for all convolutional layers is $(3, 3, 3)$. BN denotes batch normalization. We upsample the grid using bicubic interpolation by a factor of 2 and correspondingly halve the number of channels. As a non-linearity we use the sigmoid-weighted linear unit (SILU), $\mathrm{silu}(x) = x/(1 + \exp(-x))$.

5.2. Two-Dimensional Turbulence

Next, we study data assimilation for a two-dimensional turbulent fluid (Boffetta & Ecke, 2012). Machine learning in this setting requires modeling richer physics and poses a much more demanding computational problem. Furthermore, having in mind the application of data assimilation to numerical weather prediction, this class of models can be considered as the simplest approximation to modeling the flow of the atmosphere.

We consider the incompressible Navier-Stokes equation for a velocity field $\mathbf{u}$ and a pressure field p:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} - \nu\nabla^2\mathbf{u} + \nabla p - \mathbf{F} = 0 \qquad (7)$$
$$\nabla \cdot \mathbf{u} = 0\,,$$

where ν is the kinematic viscosity of the fluid. We choose the external forcing $\mathbf{F}$ to correspond to Kolmogorov flow (Chandler & Kerswell, 2013), with linear damping (Boffetta & Ecke, 2012) to ensure that the long-time behavior of the solution is statistically stationary:

$$\mathbf{F} = \sin(kx)\hat{\mathbf{x}} - \alpha\mathbf{u} \qquad (8)$$

For our experiments, we choose a domain $[0, 2\pi]^2$ with periodic boundary conditions, a wavenumber $k = 4$, a damping coefficient $\alpha = 0.1$, and a viscosity of $\nu = 10^{-2}$. We discretize the solution on a 64×64 grid and use standard numerical methods to solve the Navier-Stokes equation with a differentiable solver written in JAX (Kochkov et al., 2021). The Lyapunov time of the system is approximately 5.9 time units. Our flows are initialized with a random velocity field filtered with a spectral filter at a peak wavenumber 4, which is then integrated to a statistically stationary regime of the flow. We assimilate over trajectories of length $T = 10$ time steps, where the integration time between two such snapshots is $\Delta t \approx 0.18$, consisting of 25 internal solver integration steps. To save memory when computing gradients, we checkpoint the state from the forward pass only at each internal integration step rather than storing intermediate values (Griewank, 1994). This requires evaluating the forward pass twice, but reduces memory usage by two orders of magnitude.

We carry out data assimilation on the velocity field $\mathbf{u}$ of the flow. To analyze our forecasts, we use vorticity ω, the curl of the velocity field,

$$\omega := \left(\frac{\partial \mathbf{u}_y}{\partial x} - \frac{\partial \mathbf{u}_x}{\partial y}\right)\,. \qquad (9)$$

Vorticity describes the local direction of movement of the fluid. We visualize vorticity on a scale, which is cut off at $[-8, 8]$ with negative values (blue in Figures 6 and 9) denoting clockwise rotation and positive values (red in Figures 6 and 9) denoting counter-clockwise rotation.

We again use equidistant subsampling for the observation operator $\mathcal{H}$. In contrast to the Lorenz96 model the solution is smooth over the grid points, so we can use bicubic interpolation between observed grid points as the baseline initialization scheme. We now demonstrate the effect of

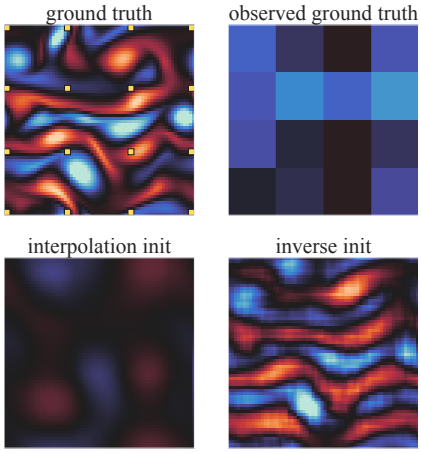


Figure 6. Comparison of initialization schemes with the ground truth initial state for Kolmogorov flow with a 16-subsampling observation operator (yellow dots). The learned inverse observation operator is trained on a trajectory of subsampled velocity fields. The observed ground truth vorticity exemplifies the amount of information of a single state of this trajectory. The trained model predicts a good approximation to the ground truth state only from a sequence of 4×4 points.

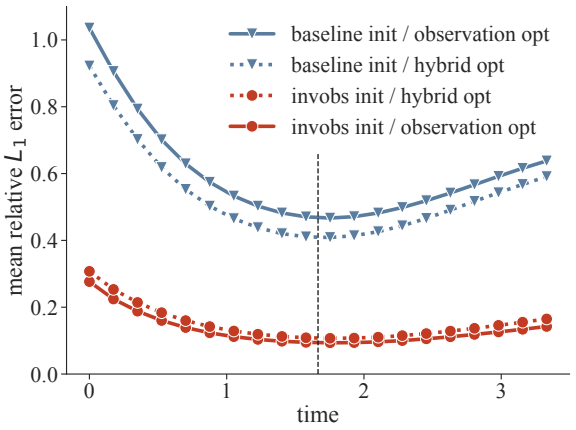


Figure 7. Data assimilation results for Kolmogorov flow using a learned inverse observation operator. Quality measure is the L_1 forecast error relative to a random evolution of the system. Depicted is the mean error based on a sample of 100 trajectories. Trajectories are obtained by evolving the initial states returned by corresponding optimization methods. Using the hybrid method for optimization improves assimilation quality with inaccurate interpolation initialization. The inverse initialization scheme significantly improves forecasts for observation space optimization. Adding inverse initialization to hybrid optimization does not improve performance. The difference between observation space optimization vs. hybrid optimization for both initialization schemes is significant with a p-value of $p < 10^{-8}$.

a learned inverse observation operator, when this operator observes every 16th grid point. For training, we employ a fully-convolutional network as shown in Table 2. We train on a dataset of 32000 independent observation trajectories with batch size 8 for 500 epochs.

For data assimilation, we compare bicubic interpolation as the baseline *interpolation initialization* with *inverse initialization* derived from the learned inverse observation operator, as depicted in Figure 2. Figure 6 compares these initialization methods with the ground truth initial state. As with the Lorenz96 model analyzed in Section 5.1, we also compare optimizing in observation space with the hybrid approach of first optimizing in physics space and subsequently refining this solution in optimization space. We limit both optimization methods to 1000 steps, with the hybrid approach using 100 of these steps to optimize in physics space and the remaining 900 to refine in observation space. Figure 7 shows the results, with three implications analogous to experiments on the Lorenz96 model. First, hybrid optimization improves assimilation quality even when using the inaccurate interpolation initialization. This implies that by first optimizing in physics space, we arrive at a favorable basin of attraction for optimization in observation space. Secondly, using inverse initialization for optimizing in observation space significantly improves forecasts. Finally, adding inverse initialization to hybrid optimization does not improve performance. This is presumably because the initialized state is already a good approximation to the optimization target, so there is no added advantage in optimizing in physics space. In contrast, it is more sensible to directly refine this state by optimizing in observation space. This effect becomes evident when analyzing how each of these optimization methods decreases the objective function in observation space during optimization, as depicted in Figure 8. For the less accurate interpolation initialization, a more favorable basin of attraction can be reached for some samples by first optimizing in physics space. However, since inverting the observation space trajectory only approximates the true trajectory, fitting against this target precludes progress after an initial phase of optimization steps. Hence, with a fixed budget of optimization steps there is a trade-off between finding a favorable basin of attraction by optimizing in physics space and finding a higher-accuracy solution by optimizing in observation space. Figure 9 qualitatively compares vorticity forecasts for the baseline method (interpolation initialization, observation space optimization) with our method based on the inverse observation operator for initialization and hybrid space optimization. The dominant features of the flow are visibly better predicted by the proposed method. Note that the structure of initial states differs from that of the following trajectories. Certain perturbations vanish quickly during the evolution of the system and are therefore not optimized to vanish in the initial state.

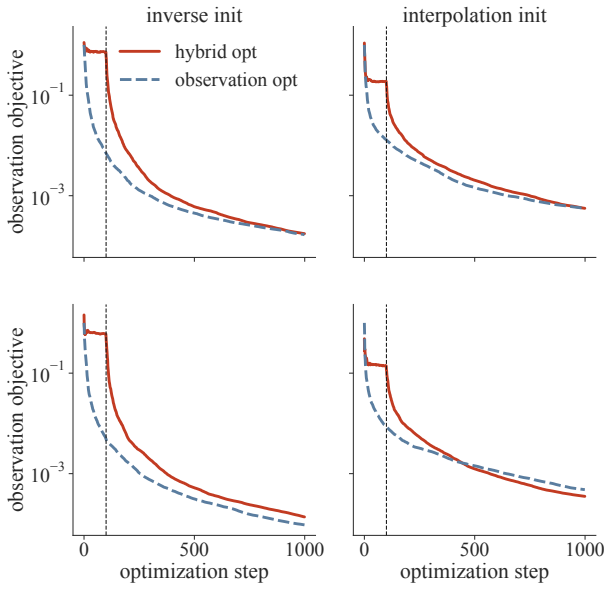


Figure 8. Observation space data assimilation objective (2) during optimization. Values are relative to the initial value on a log-scale for two different samples (rows). Depicted are observation space optimization (dashed blue) and hybrid optimization (solid red). For both methods, we evaluate the *same* observation space objective function along the optimization path. The vertical dashed line signifies the change from physics space to observation space in the hybrid method. For inaccurate interpolation initialization, a favorable basin of attraction can be reached for some samples by first optimizing in physics space. Inverse initialization provides such a good initial condition that there is no added advantage of first optimizing in physics space.

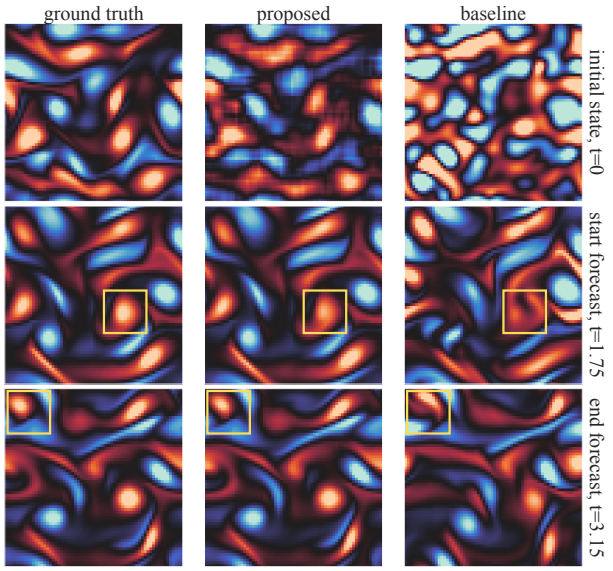


Figure 9. Vorticity snapshots of a forecast trajectory for Kolmogorov flow. The proposed method (inverse initialization, hybrid optimization) more accurately captures ground truth vorticity features (yellow squares) compared with the baseline (interpolation initialization, observation space optimization). Depicted are the initial state and snapshots from the start and end of the forecast window. Note that certain perturbations vanish quickly during the evolution of the system and are therefore not optimized to vanish in the initial state.

	LORENZ96		KOLMOGOROV	
	OBS	HYBRID	OBS	HYBRID
BASELINE	1	0.08	1	0.88
INVERSE	0.25	**0.07**	**0.20**	0.23

Table 3. Mean L_1 forecast error of the first forecast state. All values are relative to the baseline method for the respective model. The table compares both initialization schemes (baseline, inverse) and optimization methods (observation space, hybrid) for the Lorenz96 model and Kolmogorov flow. The best optimization setting is emphasized in bold face. Using the learned inverse observation operator improves optimizability for both models.

5.3. Result Summary

To summarize the results, we compare the relative performance of each optimization setting for the first forecast state, as depicted in Table 3. By using the learned inverse observation operator, the forecast error can be significantly reduced for both models. The relative merit of exploiting this operator for initialization and transformation of the objective function depends on the properties of the physical model. For the Lorenz96 model, hybrid optimization in addition to inverse initialization notably improves performance. For Kolmogorov flow, the learned inverse mapping already provides an extremely good initialization and hence optimizing in physics space does not further reduce the forecast error.

6. Conclusion

Data assimilation is the perfect problem class to explore the combination of physical modeling and machine learning since applications naturally involve rich physics and vast amounts of data. We demonstrate in this paper that a traditional variational data assimilation pipeline is improved by using a learned inverse observation operator. Exploiting this operator, we transform the 4D-Var optimization problem and show significantly enhanced forecast quality on two canonical chaotic models, the Lorenz96 model and a two-dimensional turbulent fluid flow. More broadly, our work shows that the core functionality of modern machine learning frameworks – support for automatic differentiation, hardware accelerators and deep learning – can advance research for data assimilation and other physics constrained optimization problems.

Acknowledgement

We thank Stephan Rasp, Casper Sønderby, and Jason Hickey for fruitful discussions on the topic. This work was conducted during an internship of T.F. at Google Research and was supported by the Munich Center for Machine Learning.

References

Ackmann, J., Düben, P. D., Palmer, T. N., and Smolarkiewicz, P. K. Machine-learned preconditioners for linear solvers in geophysical fluid flows. *arXiv 2010.02866*, 2020.

Andersson, E., Fisher, M., Hólm, E. V., Isaksen, L., Radnóti, G., and Trémolet, Y. Will the 4D-Var approach be defeated by nonlinearity? *ECMWF Technical Memoranda*, (479), 2005.

Bannister, R. N. A review of forecast error covariance statistics in atmospheric variational data assimilation. ii: Modelling the forecast error covariance statistics. *Quarterly Journal of the Royal Meteorological Society*, 134 (637):1971–1996, 2008.

Bannister, R. N. A review of operational methods of variational and ensemble-variational data assimilation. *Quarterly Journal of the Royal Meteorological Society*, 143 (703):607–633, 2017.

Bar-Sinai, Y., Hoyer, S., Hickey, J., and Brenner, M. P. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019.

Bauer, P., Thorpe, A., and Brunet, G. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, 2015.

Beatson, A., Ash, J., Roeder, G., Xue, T., and Adams, R. P. Learning composable energy surrogates for PDE order reduction. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Bocquet, M., Pires, C. A., and Wu, L. Beyond gaussian statistical modeling in geophysical data assimilation. *Monthly Weather Review*, 138(8):2997 – 3023, 2010.

Boffetta, G. and Ecke, R. E. Two-dimensional turbulence. *Annu. Rev. Fluid Mech.*, 44(1):427–451, 2012.

Bonavita, M. and Laloyaux, P. Machine learning for model error inference and correction. *Journal of Advances in Modeling Earth Systems*, 12(12):e2020MS002232, 2020.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Brajard, J., Carrassi, A., Bocquet, M., and Bertino, L. Combining data assimilation and machine learning to infer unresolved scale parametrisation. *arXiv 2009.04318*, 2020a.

Brajard, J., Carrassi, A., Bocquet, M., and Bertino, L. Combining data assimilation and machine learning to emulate a dynamical model from sparse and noisy observations: A case study with the lorenz 96 model. *Journal of Computational Science*, 44:101171, 2020b.

Chandler, G. J. and Kerswell, R. R. Invariant recurrent solutions embedded in a turbulent two-dimensional kolmogorov flow. *Journal of Fluid Mechanics*, 722:554–595, 2013.

ECMWF. *Data Assimilation*. Number 2 in IFS Documentation CY46R1. ECMWF, 2019.

Farchi, A., Laloyaux, P., Bonavita, M., and Bocquet, M. Using machine learning to correct model error in data assimilation and forecast applications. *arXiv 2010.12605*, 2020.

Geer, A. J. Learning earth system models from observations: machine learning or data assimilation? *ECMWF Technical Memoranda*, (863), 2020.

Griewank, A. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optim. Methods Softw.*, 1(1), 1994.

Gustafsson, N., Janjić, T., Schraff, C., Leuenberger, D., Weissmann, M., Reich, H., Brousseau, P., Montmerle, T., Wattrelot, E., Bučánek, A., Mile, M., Hamdi, R., Lindskog, M., Barkmeijer, J., Dahlbom, M., Macpherson, B., Ballard, S., Inverarity, G., Carley, J., Alexander, C., Dowell, D., Liu, S., Ikuta, Y., and Fujita, T. Survey of data assimilation methods for convective-scale numerical weather prediction at operational centres. *Quarterly Journal of the Royal Meteorological Society*, 144(713): 1218–1256, 2018.

Ham, Y.-G., Kim, J.-H., and Luo, J.-J. Deep learning for multi-year enso forecasts. *Nature*, 573(7775):568–572, 2019.

Hoyer, S., Sohl-Dickstein, J., and Greydanus, S. Neural reparameterization improves structural optimization. *arXiv 1909.04240*, 2019.

Jiang, C. M., Esmaeilzadeh, S., Azizzadenesheli, K., Kashinath, K., Mustafa, M., Tchelepi, H. A., Marcus, P., Prabhat, and Anandkumar, A. Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework. In *Proceedings of the International*

Conference for High Performance Computing, Networking, Storage and Analysis, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., and Hoyer, S. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021.

Kudyshev, Z. A., Kildishev, A. V., Shalaev, V. M., and Boltasseva, A. Machine learning–assisted global optimization of photonic devices. *Nanophotonics*, 10(1): 371–383, 2021.

Lorenz, E. N. Predictability: a problem partly solved. In *Seminar on Predictability*, pp. 1–18. ECMWF, 1995.

Mack, J., Arcucci, R., Molina-Solana, M., and Guo, Y.-K. Attention-based convolutional autoencoders for 3D-variational data assimilation. *Computer Methods in Applied Mechanics and Engineering*, 372:113291, 2020.

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. NeRF: Representing scenes as neural radiance fields for view synthesis. In *Computer Vision – ECCV 2020*, 2020.

Nocedal, J. and Wright, S. J. *Numerical Optimization*. Springer, 2nd edition, 2006.

Rasp, S. and Thuerey, N. Data-driven medium-range weather prediction with a ResNet pretrained on climate simulations: A new model for weatherbench, arxiv 2008.08626, 2020.

Rasp, S., Dueben, P. D., Scher, S., Weyn, J. A., Mouatadid, S., and Thuerey, N. Weatherbench: A benchmark data set for data-driven weather forecasting. *Journal of Advances in Modeling Earth Systems*, 12(11):e2020MS002203, 2020.

Sønderby, C. K., Espeholt, L., Heek, J., Dehghani, M., Oliver, A., Salimans, T., Agrawal, S., Hickey, J., and Kalchbrenner, N. Metnet: A neural weather model for precipitation forecasting. *arXiv 2003.12140*, 2020.

Tabeart, J. M., Dance, S. L., Lawless, A. S., Nichols, N. K., and Waller, J. A. Improving the condition number of estimated covariance matrices. *Tellus A: Dynamic Meteorology and Oceanography*, 72(1):1–19, 2020.

Ulyanov, D., Vedaldi, A., and Lempitsky, V. Deep image prior. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

Um, K., Brand, R., Fei, Y., Holl, P., and Thuerey, N. Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Watt-Meyer, O., Brenowitz, N. D., Clark, S. K., Henn, B., Kwa, A., McGibbon, J. J., Perkins, W. A., and Bretherton, C. S. Correcting weather and climate models by machine learning nudged historical simulations. *Earth and Space Science Open Archive*, 2021.

Zhuang, J., Kochkov, D., Bar-Sinai, Y., Brenner, M. P., and Hoyer, S. Learned discretizations for passive scalar advection in a 2-D turbulent flow. *arXiv 2004.05477*, 2020.