Technische Universität München
TUM School of Computation, Information and Technology

On the Algorithmic Impact of Scientific Computing on Machine Learning

Severin Maximilian Reiz

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	1. Prof. Dr. Hans-Joachim Bungartz
	2. Prof. George Biros
	3. Prof. Dr. Felix Dietrich

Die Dissertation wurde am 24.01.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 17.06.2024 angenommen.

Contents

Acknowledgements

My biggest achievement with this thesis is earning the respect of my professor, colleagues, and friends. I sincerely thank everyone I have worked with for the scientific discussions. I feel that I often voice critical thoughts, which sometimes is beneficial, but it may not be easy to always cope with that.
First, I would like to thank my Professor Hans-Joachim Bungartz, for supporting me throughout all phases of my doctoral journey. When I had an idea or a proposal, he always supported it; when I was in doubt or a difficult phase, he always suggested a solution. Additionally, thanks to him, I always felt secure regarding funding, although my academic work (research, teaching, and chair work) phase for this thesis was relatively long. Furthermore, I am thankful for my second advisor, Prof. George Biros, whom I thankfully visited several times during my work. My thesis would have taken another route if your collaboration had not inspired me.
Second, I want to express my gratitude to my collaborators, colleagues, and the thoughtful academic environment. A big thanks goes to the group of TUM-I5 (mls) and all critical thinkers and technology-interested students for enriching discussions. I appreciate their feedback and the cynic humor that often comes alongside.
Third, a big appreciation goes to my colleagues and friends who helped proofread the thesis, namely Friedrich, Michael, Keerthi, and Benni. Also, I thank my girlfriend Lena for all the support she has given me, especially in the last phase of my PhD. Academic work often keeps the mind busy, and compared to most industry jobs (and the career progress there), PhD'ing was sometimes a time of deprivation. I enjoyed the flexibility in the last six years with my long-lasting friends, family, and sports colleagues.
Lastly, the academic environment was very thought-stimulating, and I found many friends alongside the academic life. Thankfully, we contemporaries live in a time and place, where society allows funding for such scientific and personal developments.

Abstract

In scientific computing and machine learning, *matrices* and matrix multiplications are often the backbone of algorithms. An implementation with a naive dense matrix suffers from a memory and runtime complexity of $\mathcal{O}(N^2)$. On the one hand, researchers often restrict their implementation to sparse matrices, which, however, may not be sufficient to capture the whole space in modeling; on the other hand, a domain-specific approximation scheme suffers from generality.

One approach for a symbiosis of computational efficiency and generality are hierarchical matrices (or short $\mathcal{H}$-matrices), as they promise $\mathcal{O}(N \log N)$ complexity, despite being general in their algebraic construction. Algorithms using $\mathcal{H}$-matrices are shown to be advantageous originally for the discretization of elliptic partial differential equations or related integral equations. Hence, with the code `GOFMM` developed along this thesis, we extend geometry-oblivious applicability and overcome the existing limiting quadratic complexity, outperforming the exact multiplication using Intel MKL `DGEMM`, `ScaLAPACK`, or the approximate one with the popular structured matrix package `STRUMPACK` for a wide range of problems. In addition, we offer an interface to Python and couple `GOFMM` with the diffusion maps code `datafold`. We show weak and strong scaling for matrices up to $200K \times 200K$ up to 128 Intel "Skylake" nodes (with 6144 cores) of SuperMUC-NG. With `GOFMM`, it is possible to integrate hierarchical approximation in many fields in a problem-agnostic way, allowing memory and runtime benefits, especially for large-scale systems.

Another approach is to avoid setting up the matrix entirely. With `Newton-CG` developed along this thesis, we study second-order optimization for large-scale neural networks. It does not set up the Hessian matrix but rather relies only on matrix-vector products with the Hessian. In our study, we apply it to a potpourri of cases from regression, variational auto-encoders, Bayesian neural networks, image classification, and natural language processing. `Newton-CG` performs on par or better for several cases, running almost as fast as the state-of-the-art optimizers *SGD* and *Adam*. For the other cases, it is only slightly worse. We implement a data-parallel scheme with Horovod and apply `Newton-CG`, for example, on a ResNet50 architecture with 16M unknowns using an NVIDIA DGX-1 with 8 A100 GPUs. `Newton-CG` is non-intrusive in the sense that it can be easily integrated into any deep-learning data pipeline.

In summary, with this thesis, we overcome the quadratic memory and runtime complexity of matrices for many scenarios. We *enable* users to integrate these approximations non-intrusively in their applications with the publicly available parallel frameworks `GOFMM` and `Newton-CG`.

PART I

INTRODUCTION

No one is so crazy that he can't find someone even more crazy who understands him.
Friedrich Nietzsche (1844 - 1900)

1 Introduction

Scientific computing and machine learning (ML) are widely used in today's computational software world: simulations of engineering problems or physics, kernel ridge regression of point clouds, or classification problems. Different types of techniques exist, ranging to numerical treatment of partial differential equations, Gaussian processes, graph operators, and neural networks in various forms. Deep learning models are particularly prominent and are commonly used for a wide variety of applications, notably in safety-critical fields such as autonomous driving and medical image processing, natural language processing (GPT), and ML for scientific computing.

With humankind constantly pushing limits to go larger, the underlying techniques have become increasingly compute-intensive. While *hardware* can keep pace with compute requirements to some degree (partly by using accelerators), *software* techniques using intelligent compute-efficient approximation algorithms gain immense focus. In many cases, at the core of these methods are linear algebra methods, most notably *matrices.* Large matrices have tremendous memory and compute demands. Nevertheless, they are needed in the numerical treatment for simulation models. Large matrices can also occur in optimization techniques for deep learning models.

This thesis is *two-fold*, where both approximate large matrices: *First*, we start with $\mathcal{H}$-matrix algorithms, and *second*, we cover a second-order optimization algorithm for neural networks.

1.1 $\mathcal{H}$-matrices

The numerical treatment of large *dense* matrices of size $N \times N$ suffers from $\mathcal{O}(N^2)$ or even higher cost concerning storage and computational operations. With the ubiquity of needs to go larger, this complexity becomes a severe issue. In many simulation algorithms, even a matrix-vector multiplication turns out to be a computational bottleneck. Hierarchical matrices (or short $\mathcal{H}$-matrices) promise almost linear cost, i.e. $\mathcal{O}(N \log N)$, for matrix operations (multiplication, inversion, decomposition) [Hac15, Beb08]. This concept is rooted in employing low-rank compression techniques on block partitions of the matrix.

Our approach: In this thesis, we describe a novel tunable algorithm called `GOFMM` for the approximation of dense symmetric positive definite (SPD) matrices. It can be used for compressing a dense matrix and accelerating matrix-vector multiplication operations or pseudo-inverses.

Let $K \in \mathbb{R}^{N\times N}$ be a dense SPD matrix, i.e. $K = K^T$ and $x^T K x > 0,\ \forall x \in \mathbb{R}^N, x \neq 0$. We construct an approximation $\tilde{K}$ such that $\|\tilde{K} - K\| \leq \epsilon \|K\|$, where ϵ is a user-defined error tolerance. We define a matrix $\tilde{K}$ with *hierarchical low-rank structure*, i.e., $\tilde{K}$ is an $\mathcal{H}$-Matrix, by (more details in Chapter 2 and Chapter 6)

$$\tilde{K} = D + S + UV, \tag{1.1}$$

where D is *block-diagonal* with *every block being an $\mathcal{H}$-Matrix*, U and V are *low rank* approximations, and S is *sparse*. At the base of the recursive definition, the blocks of D are small dense matrices. An $\mathcal{H}$-Matrix *matvec* requires $\mathcal{O}(N \log N)$ work, while the complexity constant depends on the rank of U and V.

One important observation is that *this hierarchical low-rank structure is not invariant to row and column permutations.* While state-of-the-art frameworks, e.g., `HODLR` and `STRUMPACK`, use lexicographic ordering, we introduce a distance notion for ordering and sampling. We exploit the fact that SPD matrices can be constructed by scalar products of unknown Gram vectors ϕ, namely $K_{ij} = \langle \phi_i, \phi_j \rangle$. Therefore, the distance between rows/columns i and j can be computed by

$$||\phi_i - \phi_j||_2^2 = \langle \phi_i - \phi_j, \phi_i - \phi_j \rangle = \, = \underbrace{\langle \phi_i, \phi_i \rangle}_{K_{ii}} - 2 \underbrace{\langle \phi_i, \phi_j \rangle}_{K_{ij}} + \underbrace{\langle \phi_j, \phi_j \rangle}_{K_{jj}} \, .$$

Note that this does not require the Gram vectors ϕ, but only three entries of the matrix K_{ij}.

Limitations: Current frameworks utilizing $\mathcal{H}$-matrices predominantly employ $\mathcal{H}-$arithmetic in the context of elliptic partial differential equations, leveraging a beneficial lexicographic ordering resulting from the modeling approach, and hence, do not require the geometry-oblivious Gram-distance. Although similar matrices are rare in real-world applications, it is quite common to find matrices that can be approximated *arbitrarily* well by an $\mathcal{H}$-Matrix. In a nuanced approach, $\mathcal{H}-$arithmetic seeks to uncover underlying structures within a matrix, projecting selected data points to capture the essence of the entire set. As a result, these matrices are also occasionally referred to as *data-sparse.*

Constructing an $\mathcal{H}$-matrix requires low-rank decompositions on partitions of the matrix. A naive approach makes compute effort much higher than a straightforward matrix-vector multiplication. The pre-factor of $\mathcal{O}(N \log N)$ depends not only on the rank of the off-diagonals but also on the construction technique of the low-rank representation. While we use several heuristics to control these costs, there is a potential compromise on accuracy to an extent where it becomes challenging to ensure guaranteed tolerance.

1.2 Neural networks

Over the past decade, machine learning (ML) and deep learning (DL) methods have become the most successful learning algorithms in a wide variety of tasks. They are widespread in many applications and ubiquitous in industry or academia. In the context of scientific computing, ML methods offer a way to construct cheap and efficient surrogate models with similar accuracy to existing models.

However, the reasons behind their success (as well as their failures in some respects) are largely unexplained. For almost all methods, numerical optimization is necessary to tune the parameters or hyperparameters of the corresponding method. It is widely believed that the success of deep learning is not just due to the deep architecture of the models but also due to the behavior of the optimization algorithms used for training them. Even though numerical optimization is a comparably mature field that offers many solution approaches, the optimization problem associated with real-world, large-scale ML scenarios is non-trivial and computationally very demanding: The dimensionality of the underlying spaces is high, the amount of parameters to be optimized is enormous, and the cost function (the loss) is typically mathematically complicated being non-convex and possessing many local optima and saddle points in general (e.g., the ResNet50 scenario discussed in the thesis has about 16 million degrees of freedom in the form of corresponding weights). Additionally, the performance of a method typically depends not only on the ML approach (network) but also on the scenario of the application (dataset).

From the zoo of different optimization techniques, certain first-order methods, such as the stochastic gradient descent (SGD), have been very popular and represent the de facto fallback in many cases. In short, a deep learning model with depth D comprises of a chained function $f(W, x) = f^{(D)}(\dots f^{(2)}(f^{(1)}(W, x)))$. A network loss function $L : \mathbb{R}^n \rightarrow \mathbb{R}$ is applied after the last layer in relation to the ground truth. The optimization problem consists of finding the weights $\mathbf{W}$ associated with all functions $f^{(i)}$. The SGD with mini-batches computes intermediate weights $\mathbf{W}_k$ in iteration k with weight update d_k via $\mathbf{W}_k = \mathbf{W}_{k-1} - \underbrace{\alpha \nabla L(\mathbf{W}_{k-1})}_{d_k}$ where $\nabla L(\mathbf{W}_{k-1})$ denotes the gradient of the total loss L w.r.t. the weights $\mathbf{W}$, and α is the so-called hyperparameter learning-rate (for details see Chapter 4).

Adaptive moments (*Adam*) approaches include momentum terms in the update (with several additional hyperparameters) and are state-of-the-art for many applications. However, these algorithms are first-order, memory-bound due to low computational intensity and lack in concise mathematical foundations, explanations, and suggestions for hyperparameter tuning. We propose a second-order approach with higher computational intensity and more intuitive hyperparameters.

Our approach: Looking beyond first-order methods, second-order methods, however, come at the price of evaluating the Hessian matrix of the problem, which typically is way too costly for real-world large-scale ML scenarios. The Newton-Raphson method for second-order schemes reads $H_L(\mathbf{W}_k)d^k = -\nabla L(\mathbf{W}_k)$, where H_L is the Hessian matrix for the loss function L, $\nabla L(\mathbf{W}_k)$ the gradient, and d_k the weight update for $\mathbf{W}_k$ associated with the network. In terms of size, $H_L \in \mathbb{R}^{N \times N}$ corresponds to the amount of parameters optimized for this network. Due to its size, it is infeasible to construct the dense Hessian; hence, we solely use the Hessian matrix-vector product $H_L(\mathbf{W})s$. We use the Pearlmutter trick, i.e. we can also multiply an arbitrary vector to the gradient already and then differentiate the product, i.e. $H_L(\mathbf{W})s = \nabla_w(\nabla_w L(\mathbf{W}) \cdot s)$ [Pea94]. For details we refer to Chapter 4 and Chapter 7.

With this, we can solve the Newton-Raphson equation with the conjugate gradients (CG) iterative linear Krylov solver at costs similar to *SGD* and *Adam* per step. By theory, we should suffice with fewer steps.

Limitations: With this trick, we can exceed the existing limitations for second-order Hessian matrices. However, some other issues remain. The optimization behavior is very problematic and dependent on network architecture. The optimization surface is very high-dimensional (corresponding to the amount of parameters of the network), and non-convexity and ill-conditioned states are likely to appear.

For this, we implemented a second-order optimization algorithm with tunable features. We applied it to several large-scale networks (up to 16 million parameters) on an NVIDIA A100 DGX-1 machine. The Hessian product can also be used for explainability, network pruning, quantifying uncertainty, or a detailed analysis of the training process when necessary.

1.3 Main contributions

The main contributions of this thesis are *two-fold*: Hierarchical matrices ($\mathcal{H}$-matrices) and second-order optimization. Lastly, we add a miscellaneous contribution in terms of public outreach and science coordination.

$\mathcal{H}$-matrices: For hierarchical matrices we developed *novel matrix algorithms.* `GOFMM` is implemented in `C++` with distributed memory parallelism through the message-passing interface MPI. We show scalability up to 128 nodes on SuperMUC-NG for compression and evaluation. We applied it to several matrices arising from kernels of data clouds, partial differential equations, graphs, and more. We demonstrate that it is not a galactic algorithm, which means an algorithm has such a high time complexity constant that it never surpasses the performance of theoretically worse algorithms. In practice, we outperform the exact `MKL DGEMM` or the popular structured matrix package `STRUMPACK`. Next, we also interfaced `GOFMM` to be called directly from Python, integrating it to the diffusion maps code `datafold`. We use the fast matrix-vector (matvec) for an iterative eigenvalue solver.

Second-order optimization: We developed `Newton-CG`, a second-order training algorithm for large-scale neural networks. It has several parameters to steer convergence (Tikhonov regularization τ, Armijo step-size restriction, learning rate α, a learning rate scheduler). We parallelized it with `Horovod` using a data-parallel approach. We executed it on an NVIDIA A100 DGX-1 machine with up to 8 *GPU*s simultaneously. We applied it on networks with up to 16 million parameters from regression, variational autoencoder, bayesian neural networks, image classification, and natural language processing. It became evident that the saved epochs in convergence speed for `Newton-CG` or lower optima is very *problem-dependent.* We implemented it non-intrusively in the DL-affine Tensorflow framework, making it easy to integrate into existing pipelines.

Miscellaneous: This thesis makes several contributions related to each of the topics. The thesis was written in the context of the German Priority Program 1648 Software for Exascale Computing SPPEXA[1] with many fruitful engagements for science coordination in conferences and annual plenary meetings, benefiting this thesis as well as research by others. The program by the funding agency Deutsche Forschungsgemeinschaft is summarized in this book article [BNN$^+$20] and in SIAM News [RB20]. In addition, we launched the Android application TUMlens in an effort to reach out to the public community.

1.4 Structure of the thesis

As the thesis is two-fold, there are chapters in parallel threads. The theory of the $\mathcal{H}$-matrix block is explained in Chapter 2, followed by implementation and results in Chapter 6. The `Newton-CG` block is comprised of the two theory chapters, Chapter 3 for the deep learning basics and Chapter 4 for the optimization basics. The methods, implementation, and experimental results are discussed in Chapter 7. The computational setup, including clusters and the Android Smartphone application, is described in Chapter 5. We close with a conclusion and outlook in Chapter 8.

To get to the meat of the thesis, skip the introduction and the theory. For $\mathcal{H}$-matrix and `GOFMM` go immediately to Section 6.1, and the results in Section 6.4 and Section 6.5. For `Newton-CG` the algorithm implementation is explained in Section 7.1 and the following sections for the results. The conclusion (Chapter 8) summarizes the results and discusses them regarding outlook into the future.

[1]`http://www.sppexa.de/`

PART II

FOUNDATIONS, STATE OF THE ART, AND RELATED WORK

The tired body finds its pillow everywhere, but when the mind is tired, where should it rest?
Georg Büchner (1813 - 1837)

2 $\mathcal{H}$-Matrices

This doctoral thesis is (mainly) two-fold: the first topic develops algorithms in the field of hierarchical matrices (short: $\mathcal{H}$-Matrices) with the code `GOFMM`, and the second topic addresses algorithms for second-order neural network optimization, with the code `Newton-CG`. In this first theory chapter, we begin with the theory of $\mathcal{H}$-Matrices, the basis of the `GOFMM` code, which we coupled to other codes and applied it to a range of problems, including an iterative algorithm for eigenvalue computation (Arnoldi iteration).

This chapter explains numerical and computational algorithms for large-scale. The grand algorithmic challenge often is to make algorithms suitable for high-dimensions d and large-scale problem sizes N. For this, those algorithms must be parallel and scalable, which require high amounts of allowed parallel computations and little need of communicating in between. Most groundbreaking developments follow these algorithmic paradigms: 1. *hierarchic* employing the inner structure of the problem; 2. *recursive* also allowing finely grained parallelism for many cores with large N; 3. *dynamic-adaptive* to allow problem-agnosticism for potentially dynamically changing requirements. Examples include the multi-grid methods or other multi-level algorithms; the same algorithmic patterns are similarly followed for domain decomposition methods, octrees, and many more. The development of $\mathcal{H}$-matrices and `GOFMM` also follow the same pattern: hierarchic, recursive, and dynamic-adaptive.

We start with general $\mathcal{H}$-matrix algorithms (Section 2.1) in particular for a hierarchal off-diagonal low-rank format, a variant in the family of $\mathcal{H}$-matrices. An $\mathcal{H}$-matrix implementation can involve a singular value decomposition for compression, and we illustrate how this can be done hierarchically for off-diagonal blocks of a matrix. We also show that with this format, we can compute a faster hierarchical matrix multiplication, and we can use this $\mathcal{H}$-matrix format for a pseudo-inverse. Additionally, we review related work in linear algebra with $\mathcal{H}$-matrices.

We continue in Section 2.2 to explain the foundations of the geometry-oblivious fast multipole method, `GOFMM`, which we developed along the thesis. This `GOFMM` scheme also forms a variant of an $\mathcal{H}$-matrix implementation. We introduce the respective algorithms and explain the difference between the fast multipole and the hierarchical semi-separable format.

Lastly, we cover applications of hierarchical matrix algorithms in Section 2.3. We created the matrix cases ourselves to have complete control over analyzing the test case. We most excessively experimented with Gaussian kernel matrices, so we describe Gaussian kernel density estimation in detail. Furthermore, we identified a computational bottleneck in a representative manifold learning algorithm: the computation of eigenvalues. Hence, we address this issue with our `GOFMM` $\mathcal{H}$-matrix variant and couple these two algorithms in the result section. Consequently, we describe the two representative algorithms from the established field of numerical linear algebra, Arnoldi, and the more recent manifold learning algorithm, diffusion maps, a (non-)linear dimensionality reduction algorithm. Eigendecomposition tends to be the computational bottleneck of such algorithms, and we address this using an iterative eigendecomposition method, namely the Arnoldi method. This coupling scheme opens myriad possibilities for other algorithms of the respective fields.

We close with a summary at the end of the chapter, i.e. in Section 2.4.

2.1 Hierarchical matrix algorithms

Domain-specific compression algorithms, such as the one by Barnes and Hut [BH86] and the later fast multipole method [GR87], are widely used in computational physics, especially molecular dynamics. $\mathcal{H}$-matrices are an algebraic generalization, and they originate in matrices for the discretization of elliptic partial differential equations and, integral equations and other problems. In this section, we introduce these $\mathcal{H}$-matrices, explain how the singular value decomposition can be used hierarchically, and how a hierarchical matrix multiplication and pseudo-inverse can be computed. We also state some related work, other approaches, and other software packages.

2.1.1 Algebraic compression algorithms

Let us look at a general matrix-vector product

$$u = Kw$$

where $u \in \mathbb{R}^N$ are called potentials, $K \in \mathbb{R}^{N \times N}$ the system matrix and $w \in \mathbb{R}^N$ the weights.

$\mathcal{H}$-matrices is an umbrella term for a family of different $\mathcal{H}$-matrix variants, and here we first introduce the **hierarchical** off-diagonal low-rank matrix (HODLR) variant. We wish to approximate K in a recursive definition by a **hierarchical** matrix of the form

$$K \approx \tilde{K} := D + UV \ ,$$

where D is a block-diagonal matrix, whose blocks can recursively again be a **hierarchical**, U, V a block low-rank approximation matrix representing off-diagonal blocks.

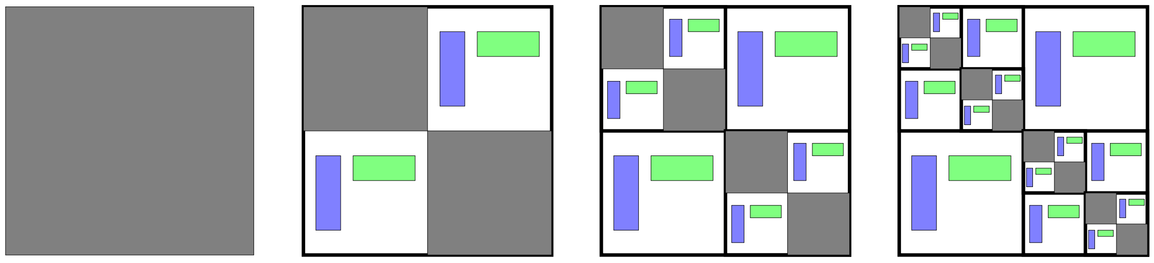


Figure 2.1: Visualization of a HODLR variant of a hierarchical matrix ($\mathcal{H}$-matrix) of the form $\tilde{K} := D + UV$. Off-diagonal blocks are hierarchically represented by a low-rank approximation, visualized by the blue and green submatrices. On the **left**: fully dense (hierarchy level 0); on the **right**: $\mathcal{H}$-matrix with hierarchy level 3. For details on the HODLR variant, see also Figure 2.2 and the section around it.

Figure 2.1 shows a visual definition of an $\mathcal{H}$-matrix, of an HODLR to be exact. The hierarchical construct can be imagined as structural compressions of matrix blocks. This scheme reduces the number of entries significantly, as seen by the ratio of whitespace on the right $\mathcal{H}$-matrix in relation to the fully dense on the left. In some sense, the structural compressions of blocks lead to thin and wide matrices. So, to some extent, this can be seen as "important" vectors (blue) that are multiplied by a projection matrix (green) to get an approximation of the block (gray off diagonals of the upper level, left). Consequentially, such matrices are sometimes referred to as *data-sparse* since we can find important vectors, “skeletons”, which represent the cluster.

We chose the above definition for $\mathcal{H}$-matrices, particularly HODLR, as it is the most catchy variant. However, as mentioned, there are more classes of low-rank structured matrices. Later, we will mention hierarchical semi-separable matrices (HSS), which are similar to HODLR, as they also assume hierarchical low rank on the off-diagonals. However, the low-rank representation of off-diagonals in HSS matrices is constructed by a nested basis from the lower level [Li21]. Hence, they are not as “catchy” and easy to visualize.

The Barnes and Hut [BH86] algebraic variant or the fast multipole method (FMM, by Greengard and Rokhlin [GR87]) also have a nested basis in the construction of the low-rank off-diagonals. However, they additionally have a bucket of geometrically near-interactions that are kept track of with, e.g., Verlet-lists (for details, see Section 2.2). In an algebraic view, this can be resembled by an additional sparse matrix. For constructing an FMM, we can first subtract the sparse matrix from the full dense matrix, then perform a HODLR $\mathcal{H}$-matrix-variant on the rest. Hence, we define a `FMM` of the form

$$K \approx \tilde{K} := D + UV + S$$

where D, U, V are matrices as above, and S a sparse matrix, representing points in off-diagonals that are nevertheless regarded important such that they need to be multiplied directly. The terms “near”/“far” originate in geometric tree-based approximation algorithms, i.e. Barnes-Hut [BH86] and FMM.

Hence, using an FMM, we can use this additive splitting and write

$$u \approx \tilde{K}w = (D + UV + S)w \ ,$$

where we can split the multiplications and add them afterward, i.e., we mathematically use the distributive law. Furthermore, we wish that $\tilde{K}$ has less direct entries than K and, thus, reduces storage and speeds up arithmetic operations. Any algorithm that computes $\tilde{K}$ from K, where $\tilde{K}$ is cheaper than K, can be called a compression algorithm. [GTY97, Beb08] For details on compression, see Subsection 2.1.2 and for splitted multiplication, see Subsection 2.1.3.

Similarly, we can use this splitting approach in computing a solution of N linear algebraic equations

$$Kx = b \ ,$$

where K is dense and nonsingular, $b \in \mathbb{R}^N$ given and x a N-dimensional solution vector. *One* approach is to calculate a pseudo-inverse solution by a compressed $\tilde{K}$, for details see Subsection 2.1.4. *Another* approach can use a fast matrix-vector product in iterative solution methods (preconditioned conjugate gradient [S$^+$94], GMRES [SS86], Arnoldi iteration [Arn51] etc.). The number of iterations in the solver depends on the condition of the matrix; preconditioning is indispensable, but for certain problems, finding a suitable preconditioner can be difficult; for further details, see Subsection 2.3.3. Almost always, a bottleneck of iterative solvers lies in memory and runtime expensive matrix-vector multiplications. [GTY97, AD13]

In general, these ideas of matrix compression are based on the observation that matrices can often not be represented by global low-rank, but it is reported that for many model problems “some” blocks can be compressed with a low-rank decomposition (for scenarios, see Subsection 2.1.5).

2.1.2 Low-rank representation

For every matrix $K \in \mathbb{R}^{N \times N}$ we can compute a singular value decomposition (SVD). An SVD is the “golden standard” and textbook method of compression algorithms due to tight theoretic bounds. It reads

$$K = U\Sigma V^T \ ,$$

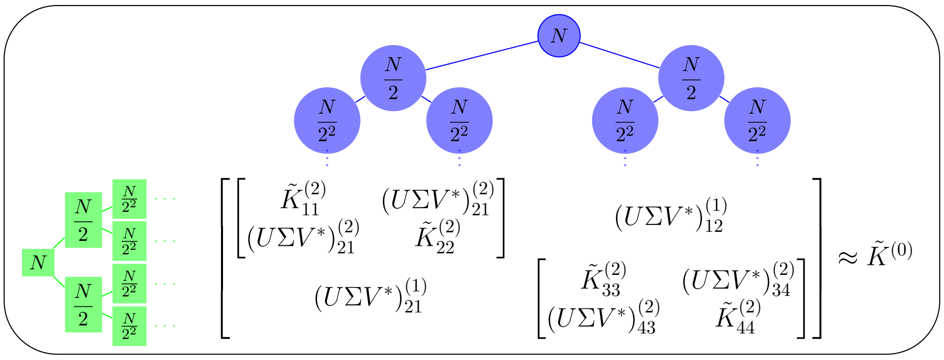
where $U,\ V \in \mathbb{R}^{N\times N}$ are orthogonal matrices, and $\Sigma \in \mathbb{R}^{N\times N}$ is a diagonal matrix, whose diagonal entries are the singular values σ_i, i.e. $\Sigma_{ii} = \sigma_i$ in the ordering $\sigma_1 \geq \sigma_2 \geq \dots$.

We call a matrix *low-rank* with rank r if we can compute a singular value decomposition, where $\sigma_r \leq \epsilon$ for a small $\epsilon > 0$ and $r \ll N$. We can then write a rank-r approximation G by cutting off at some location r, and write

$$G := U\Sigma_r V^T \text{ with } (\Sigma_r)_{ii} = \begin{cases} \sigma_i & \text{if } i \leq r \\ 0 & \text{otherwise} \end{cases} .$$

We can estimate the compression error by the Eckart-Young-Mirsky theorem in terms of the $(r+1)^{th}$ singular value, i.e. $||K - G|| \leq \sqrt{\sigma_{r+1} + \dots + \sigma_N} \leq \sigma_{r+1}$ [Hac15, Kre].

In Figure 2.2, we show the structure of a so-called HODLR matrix (remember from the previous section, hierarchical off-diagonal low-rank). Note that a block-diagonal matrix $\tilde{K}$ can again be a HODLR matrix, an $\mathcal{H}$-matrix-variant, e.g. the left-upper $K_{11}^{(2)}$ from tree level (2) with diagonal index 11. For example, in this case, in Figure 2.2, a full square matrix needs N^2 computations for dense multiplication. In HODLR, each level of the tree contains $N/2^l \cdot 2^l = N$ numbers; on each level, we need around $N \cdot r$ computations for the off-diagonals, given a fixed approximation rank of r. We have at most $\log N$ levels, i.e. $\log N$ summands, leading to $\mathcal{O}(N \log N)$ overall. The on-diagonal computations are just in the order of $\mathcal{O}(N)$.



N
$\frac{N}{2}$
$\frac{N}{2^2}$
$\tilde{K}_{11}^{(2)}$
$(U\Sigma V^*)_{21}^{(2)}$
$\tilde{K}_{22}^{(2)}$
$(U\Sigma V^*)_{12}^{(1)}$
$(U\Sigma V^*)_{21}^{(1)}$
$\tilde{K}_{33}^{(2)}$
$(U\Sigma V^*)_{34}^{(2)}$
$(U\Sigma V^*)_{43}^{(2)}$
$\tilde{K}_{44}^{(2)}$
$\approx \tilde{K}^{(0)}$

Figure 2.2: Recursive definition of the **hierarchical** off-diagonal low-rank representation (HODLR). Binary trees on top and left are formed for the off-diagonal contributions. One tree would be sufficient, but for the unsymmetrical case, this allows more flexibility. Note that for a level (1) tree (holding $\frac{N}{2}$ entries), the nodes hold itself, i.e., the off-diagonal, and its children, i.e. the on-diagonal, leading two sub-matrices. The on-diagonal is then split for the next level. We assume a fixed rank-r for the off-diagonals in each node, and due to some induced compression errors, we only write "$\approx$" instead of "$=$".

We can set the stopping criterion of this recursive structure either as maximum tree-level depth or from a leaf-node-size `m`, which we use mostly. A recursion depth can be $K_{11}^{(l)}$ consisting of a single element (`m`=1); however, we stop at a leaf node size `m` ≥ 1 due to practical reasons, such as limited benefit of rank-r on small matrices. Theoretically, for small sizes, there is no benefit of rank-r and even disadvantages due to construction effort. AVX vector units and cache lines need to be considered for numerical calculations on a computer. Thus, depending on the problem and the computational resource, values like `m` $= 64$ are beneficial.

2.1.3 Hierarchical matrix multiplication

Considering the task of a matrix-vector product, we have a system matrix $K \in \mathbb{R}^{N \times N}$, which we multiply with a weight vector $w \in \mathbb{R}^N$ to get a potential $u \in \mathbb{R}^N$, i.e.

$$u = K \cdot w \ .$$

Let us now decompose the matrix K in blocks of the form $K = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix}$ and the vectors in blocks of the form $u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ and $w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$.

We can then formulate the block-wise matrix-vector product separated with commas as

$$u_{11} := K_{11} w_1 \ , \ u_{12} := K_{12} w_2 \ , \ u_{21} := K_{21} w_1 \ , \ u_{22} := K_{22} w_2 \ .$$

Finally, the intermediate terms need to be summed up, leading to $u_1 = u_{11} + u_{12}$ and $u_2 = u_{21} + u_{22}$, resulting in the upper resultant vector u. [Hac15]

The benefit of this simple additive arithmetic, again the distributive law, is limited if the matrices are dense and fully populated, as the complexity is still $\mathcal{O}(N^2)$.

In HODLR, however, we assume that the blocks $K_{12}, K_{21} \in \mathbb{R}^{N/2 \times N/2}$ can be approximated with a low-rank representation, $(U\Sigma V^*)^{(1)}_{12}, (U\Sigma V^*)^{(1)}_{21}$. With a rank-r approximation, the multiplication with a vector is only of complexity $2 \cdot N/2 \cdot r = N \cdot r$, where we hope for $r \ll N/2$. We have a maximum of $\log_2 N$ levels. For a fixed rank-r, this results in $\mathcal{O}(N \log_2 N)$ complexity.

This block-wise multiplication is the core of the evaluation algorithm. We specify details for `GOFMM`, the algorithm developed along the thesis, later in Section 2.2 and in Section 6.1. In Section 6.4 we show empirical data showing this $\mathcal{O}(N \log N)$ computational complexity for `GOFMM`.

2.1.4 Hierarchical pseudo-inverse

As mentioned before, and provoked in the introductory paragraph of Subsection 2.1.1, we can use a splitting approach also in computing a direct solution of N linear algebraic equations

$$Kx = b$$

where K is dense and nonsingular, $b \in \mathbb{R}^N$ given and x a N-dimensional solution vector.

Again, let us split our matrix in four blocks,

$$K = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \ .$$

With this, we can compute the inverse using the so-called Sherman-Morisson-Woodbury (SMW) formula that is also widely used by other fields or works. [DYBM23, BG99] Let us use this significant property of a block-wise splitting and write the inverse of K as

$$K^{-1} = \begin{bmatrix} K_{11}^{-1} + K_{11}^{-1} K_{12} S^{-1} K_{21} K_{11}^{-1} & -K_{11}^{-1} K_{12} S^{-1} \\ -S^{-1} K_{21} K_{11}^{-1} & S^{-1} \end{bmatrix}$$

where $S = K_{22} - K_{21} K_{11}^{-1} K_{12}$ is the so-called Schur complement. With a hierarchical matrix, $\mathcal{H}$-matrix, we refer to this as $\mathcal{H}$-inverse.

This SMW formula is an exact formula, and the proof can be found in literature, see e.g. [Rie92, Hac15]. It is well known that computing an inverse numerically is instable [Huc96]. In our HODLR setting, we use approximations of matrix blocks in addition as a new source of error, e.g. on the off-diagonal block K_{12}. Hence, we sometimes call this pseudo-inverse as a note of caution caused by numerical instabilities. Others use such a pseudo-inverse for hierarchical preconditioning and smoothening with some iterative method, e.g. [LBG06]. We have done experiments on a preconditioned conjugate gradient scheme as well [YRB19].

2.1.5 Hierarchical matrix applications

Wolfgang Hackbusch, his colleagues, and collaborators have driven many fundamental research projects in this area for around 20 years. They have found use cases for elliptic differential equations with a geometric motivation and for integral equations with a Galerkin discretization or a collocation method. In [LBG06] $\mathcal{H}$-preconditioning was done on advection/ diffusion problems. In [Hac15] Hackbusch already mentions that many interesting kernel functions depend on the distance, $\mathcal{K}(x-y)$, similar to radial basis function kernels for point clouds, which we also describe in Subsection 2.3.1 and employ in Section 6.4. The area evolved, and some collaborators from this peer group show runtime advantages of a high-performance block low-rank representation (BLR), another $\mathcal{H}$-matrix variant, also for elliptic equations. [INK18]

A more recent and more high-performance-oriented development is the structured matrix package `STRUMPACK` [RLGN16], which is a software library providing linear algebra routines and linear system solvers for sparse and dense rank matrices. It was developed at NERSC and is now integrated into the Exascale Computing Project (ECP) with nice software integrations by the US Department of Energy. Several groups are involved, e.g., in [GS22], a multifrontal factorization algorithm is developed for regular mesh PDE problems or test matrices from sparse matrix collections. In [LXG$^+$21], the focus was butterfly factorization with randomized matrix-vector multiplications.

To summarize, we gave a non-exhaustive description of some related work, mainly $\mathcal{H}$-matrices [Hac15] and the structured matrix package `STRUMPACK` [RLGN16]. In the following Section 2.2, we show the foundations of our package `GOFMM`, which we compare to `STRUMPACK` in Subsection 6.4.2. Our method finds a matrix re-ordering by design, and others often assume a given suitable partition.

2.1.6 Summary

In this section, named hierarchical matrix algorithms, we introduced concepts of $\mathcal{H}$-matrix algebra for matrix multiplication and computing an $\mathcal{H}$-inverse.

We started by describing general algebraic compression algorithms and how they relate to geometric analytic approximations from computational physics by Barnes and Hut [BH86]. With this, we discussed a low-rank decomposition, namely SVD. We explained how it can be used hierarchically on off-diagonal blocks, leading to a *hierarchical off-diagonal low-rank* scheme.

Next, we explained how block-wise matrix-vector multiplication can lead to lower complexities, assuming a HODLR structure. In addition, we showed that we can use the Sherman-Morrison-Woodbury formula with the Schur complement to compute the inverse of a block matrix. In HODLR with low-rank blocks, this also outlines a favorable algorithm.

Finally, we showed some $\mathcal{H}$-matrix applications and related work in the field. This includes mainly $\mathcal{H}$-matrices from [Hac15] and the structured matrix package `STRUMPACK` [RLGN16].

2.2 Metric trees, neighbor search, and skeletonization

After having introduced general $\mathcal{H}$-matrix variants in the previous section, in this section, we focus on the algorithms for the geometry-oblivious fast multipole method (`GOFMM`), which we developed along the thesis. We start with the geometric predecessor `ASKIT`, explaining the geometry-oblivious distance notion and the algebraic low-rank representation. We discuss our randomized neighbor computation algorithm and differentiate between the fast multipole method (FMM) and hierarchically semi-separable (HSS) representation.

2.2.1 Geometric tree algorithms

Tree structures that represent an underlying geometric setting are often called geometric trees. Examples are octrees and kd-trees - two structures we will briefly introduce and compare in the first paragraph.

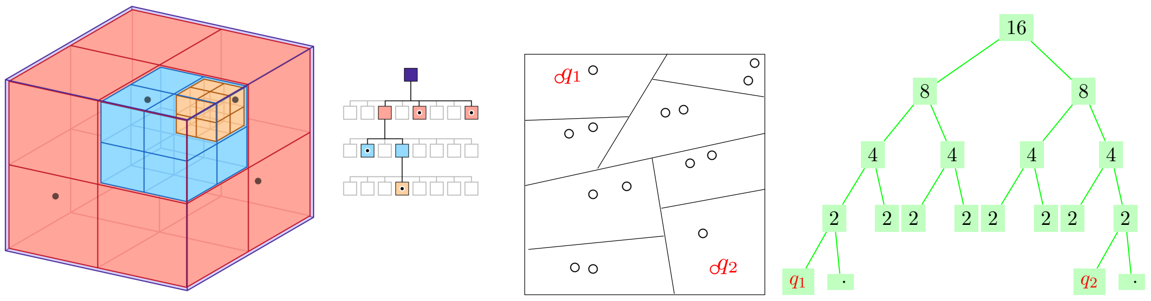
We continue this section with a motivation from the N-body problem, which forms the ground for `ASKIT`: *Approximate kernel independent treecode* and other fast multipole method codes. Our approach is an extension of `ASKIT`, but nevertheless, our idea of `GOFMM` is also sparked by the N-body problem. After the N-body problem motivation, we describe a hierarchical interaction calculation algorithm, like the multipole expansions used in computational physics for molecular dynamics. We will cover textbook methods for algebraic compression algorithms in the next Subsection 2.2.2.

Octrees vs. k-dimensional trees

For computational fluid dynamics, molecular dynamics, or other simulations in 3 dimensions, the domain is often decomposed using an **octree**, see Figure 2.3 (a). In molecular dynamics, for example, the hierarchic view comes from the fact that molecules are much more likely to interact with near molecules. The so-called butterfly effect, that an event on one side of the domain has large immediate effects on the other side, is quite unlikely, compared to near interactions, i.e., through collisions or near force fields, if the model is chosen well. A slower, time-dependent transport is possible in most molecular dynamics models through some particle exchange layers, so-called ghost cells, or similar methods.

With this analogy in mind, we also decompose the geometric space in Barnes-Hut schemes or fast multipole methods that we use for the thesis. However, this octree splitting is unsuitable for high dimensions, as an octree grows with dimensions ($2^d = 8$ for three dimensions). We would result in $2^4 = 16$ dimensions for a 4-dimensional tree. We have some randomized point clouds in high dimensional space in our setting, often $d > 100$. A randomized binary **k-dimensional tree** is more suitable for such high-dimensional point clouds, and thus, popular in data analysis. For simplicity, we illustrate a k-d tree, short for k-dimensional, where here we have $k = 2$, with random projection axes in Figure 2.3 (b). The point cloud is decomposed in an equalized split, so the 16 points are in buckets of 8 after the first split, i.e. in level. We split the point buckets subsequently, and for the query points, which are q_1 and q_2, we go to a level-four tree. The points $q_1$andq_2 are quite far away, and with a very high probability, those points are also in oppositely located leaves in the binary tree. Recall that this also fits our assumption that near points are put in near buckets, where interactions are possibly high, and far points are placed in far-away tree nodes.

Therefore, a k-dimensional binary tree with some randomized projection axes works similarly well and expands nicely to high dimensions. The tree size does not depend on the spatial dimension d, but rather on the number of points or the desired depth level of the tree.



(a) Octree taken from [Dul24]

(b) Randomized k-dimensional tree, expanded to three levels and four levels for query points q_1, q_2

Figure 2.3: Difference between a structured octree and a randomized k-d tree.

Geometric motivation from N-body problems

N-body methods are used in many disciplines: simulations of celestial gravitational, Coulomb, and Lennard-Jones potentials; in waves and scattering or fluids and transport; in data analysis, a similar problem occurs in scientific machine learning, geostatistics, and image analysis. [Rei17]

Given a set of N targets $x_i \in \mathbb{R}^d$, a set of N sources $x_j \in \mathbb{R}^d$, weights w_j and a kernel function $\mathcal{K}(x_i, x_j)$ (some pairwise potential), we want to compute a target potential u_i,

$$u_i = u(x_i) = \sum_{j=1}^{N} \mathcal{K}(x_i, x_j) w_j \ .$$

This operation can be viewed as a matrix-vector product with complexity $O(N^2)$ and is often the computational bottleneck of such simulations.

We want to reduce this complexity by exploiting geometrical information. This can be done with analytical methods for classical Barnes-Hut in a hierarchical structure; for example, Near-Lists are kept in Linked cells or in Verlet lists. In a “matrix” view (i.e., algebraic), this relates to exploiting possible low-rank blocks in matrix K
($K_{ij} = \mathcal{K}(x_i, x_j)$). We rewrite

$$u_i = \sum_{p \in \texttt{Near}_i} K_{ip} w_p + \sum_{p \in \texttt{Far}_i} K_{ip} w_p \ ,$$

where $\texttt{Near}_i$ is a set of near points, whose contributions need to be regarded individually and $\texttt{Far}_i$ being the set of far points, whose contributions can sufficiently be computed by an analytic approximation, or algebraically, by a low-rank approximation. [Rei17]

Hierarchical interaction calculation algorithm

The idea of such approximations relies on the observation that an interaction force with respect to a far-away individual point may be approximately neglected; however, the force with respect to a far-away cluster of points may not be neglected but can be approximated. Thus, we want to group far-away points in clusters hierarchically. In Subsection 2.2.4, we discuss *near-far pruning* criteria.

In this way, we will calculate for a single point i some *interactions* from `Near` points $j_{\texttt{Near}}$ individually, but *interactions* from `Far` clusters ($\texttt{Far}_{j'} \subset \{x_j\}_{j \in \texttt{Far}}$) only approximately (we aggregate weights at skeletons or center of mass of clusters, for example, for gravitational fields).

In celestial simulations, one can assume that we represent a clustered formation of stars (or solar systems, galaxies) in terms of their accumulated mass located at their center of mass. For points (comets, planets, stars) in different clusters, we approximate the incoming gravitational force from that cluster by its aggregated mass, e.g. at its center of mass. In this context, this represents a pseudo-point since we do not necessarily have a data point located at the center of mass. While this is not a problem per se, alternatively, we can use (one or more) actual data points serving as "skeletons" at which we aggregate the mass of its underlying cluster. [Rei17]

2.2.2 Low-rank approximations for GOFMM

In this section, we relate the geometric tree algorithms from the previous section to Gram matrices. We show algebraic compression algorithms for `GOFMM`, namely a reduced singular value decomposition for reference and the interpolative decomposition that we use. We also relate the decompositions to randomized linear algebra, which is the state-of-the-art approach used in recent literature, with which our interpolative decomposition could be replaced in the future.

As already sparked in Chapter 1 in the geometric-oblivious fast multipole method (`GOFMM`), we exploit the fact that SPD matrices can be seen as a Gramian matrix. Any Gramian can be constructed by scalar products of unknown Gram vectors ϕ, namely

$$K_{ij} = \langle \phi_i, \phi_j \rangle \ .$$

With this in mind, we can express the above distance algorithms without the actual geometric distance, but in terms of Gram distance. In a kernel matrix, a data point (source) generates a row of a matrix, and a potential can be formed in columns at any target point. We can compute the distance between rows and columns i and j with the Euclidean **Gram-ℓ_2** distance

$$||\phi_i - \phi_j||_2^2 = \langle \phi_i - \phi_j, \phi_i - \phi_j \rangle = \ = \underbrace{\langle \phi_i, \phi_i \rangle}_{K_{ii}} - 2 \underbrace{\langle \phi_i, \phi_j \rangle}_{K_{ij}} + \underbrace{\langle \phi_j, \phi_j \rangle}_{K_{jj}} \ .$$

Note that this does not require the Gram vectors ϕ, but only 3 entries of the matrix K_{ij}. Gram vectors are not unique; they can be computed from a given matrix, e.g., with a Cholesky-decomposition $K = LL^T$. Computing them is far too expensive and not necessary for our purposes.

We observed sometimes limited capabilities of the Near/far-field Euclidean assumption in Gram-vector spaces, so we also developed a **Gram-angle** criterion. Although this may appear arbitrary, the idea is sparked by an observation relating the rank of an off-diagonal block to the degree of orthogonality between sets of Gram vectors. Consider disjoint index sets $\alpha, \beta \subset \mathcal{I}$ and the corresponding matrix of interactions $K_{\alpha\beta}$. If we define ϕ_α to be the matrix with columns $\{\phi_i\}_{i\in\alpha}$, and define $\{\phi_j\}_{j\in\beta}$ similarly, then $K_{\alpha\beta} = \phi_\alpha^T \phi_\beta$. We may view $K_{\alpha\beta}$ as a projection of ϕ_β onto the span of ϕ_α, so the rank of $K_{\alpha\beta}$ is equal to the dimension of the projection. Thus, we require a measure of distance that clusters Gram vectors to form orthogonal subspaces rather than small volumes as above. [Rei17]

With the law of cosines, $c^2 = a^2 + b^2 - 2ab\cos\gamma$, we can express the angular distance again by three entries of the matrix, i.e.

$$\cos(\sphericalangle(\phi_i, \phi_j)) = \frac{\langle \phi_i, \phi_j \rangle}{\|\phi_i\| \cdot \|\phi_j\|} = \frac{\langle \phi_i, \phi_j \rangle}{\sqrt{\langle \phi_i, \phi_i \rangle \langle \phi_j, \phi_j \rangle}} \ .$$

With these **two** defined **Gram** distances, we can apply the previous far-field approximation algorithms with respect to Gram vector distance. With this in mind, we will cover algebraic compression in the next two paragraphs. In Subsection 2.2.3, we then explain how we compute Gram-neighbors, and in Subsection 2.2.4, we define our near-far pruning criterion for Gramians.

Reduced singular value decomposition

In our hierarchical matrix, we first compute compressions on the leaf level. With rank-reveling decomposition, we compute "important" columns, so-called *skeletons* that are aggregate "clusters". Those are passed upward in the tree, and we obtain a very skinny matrix by agglomerating some columns through rank-revealing factorization. We use the terms thin, narrow, or skinny interchangeably. The textbook reference decomposition for such matrices would be a reduced singular value decomposition.

Since we mostly compute low-rank decompositions for narrow matrices, i.e. $N << M$, we want to additionally discuss a reduced form of the singular value decomposition (SVD). For a matrix $G \in \mathbb{R}^{M\times N}$, we are seeking a SVD decomposition

$$G = U\Sigma V^*$$

where $U \in \mathbb{R}^{M\times N}$, $V \in \mathbb{R}^{N\times M}$ are unitary matrices and $\Sigma \in \mathbb{R}^{M\times N}$ is a non-negative diagonal matrix with singular values on the diagonal, i.e. ($\Sigma_{ii} = \sigma_i$). They are, in fact, the non-negative square roots of eigenvalues to GG^* or G^*G. A regular singular value decomposition can be calculated by successive Householder transformations on A using an intermediate bidiagonal matrix $J \in \mathbb{R}^{m\times n}$ and a subsequent diagonalization of J. We usually use a QR decomposition of A for a reduced thin version. Overall, the computation of the singular value decomposition requires $\mathcal{O}(MN^2)$ operations (where $M \geq N$). [GK65, Beb08, Rei17]

Using this approach, we can approximate for a $(N-q)\times q$ block of matrix $K \in \mathbb{R}^{N\times M}$, its off-diagonal term, let us call it $G \in \mathbb{R}^{(N-q)\times q}$. This leads, among other matrices, to $U \in \mathbb{R}^{(N-q)\times q}$. Assuming that N is large (and q is fairly small), this approach becomes very unfavorable in terms of memory because we need to save the basis U, V for the decomposition. [Rei17]

Interpolative decomposition

Roughly speaking, in an *interpolative decomposition*, given a matrix G of rank k, the goal is to find k columns of the matrix, such that they form a suitable basis for the remaining columns. With this, we can use the columns as a nested basis for the parent.

This approach gives us benefits in terms of memory storage and saves operations, leading to a complexity of $\mathcal{O}(kMN)$. For any matrix $G \in \mathbb{R}^{M\times N}$ of rank k there exists a subset of columns from G building $G_{col} \in \mathbb{R}^{M\times k}$ and a projection matrix $P \in \mathbb{R}^{k\times N}$, such that

$$G_{col}P = G \ .$$

Additionally, some subset of P consists of the $k \times k$ identity matrix and P is not too large. [LWM+07, Rei17]

Let us suppose the rank of the matrix G is $s > k$, and $k \leq N, M$. We can find a $G_{col} \in \mathbb{R}^{M\times s}$ and $P \in \mathbb{R}^{s\times N}$, such that

$$G_{col}P \approx G \ ,$$

and can estimate the error as

$$\|G_{col}P - G\|_2 \leq \sigma_{s+1}\sqrt{s(n-s)+1} \ ,$$

where σ_{s+1} is the $(s+1)$st singular value of G. [LWM+07, Rei17]

An interpolative decomposition can be calculated using a rank-revealing QR decomposition. [Beb08] We decompose G into [Rei17]

$$G\Pi = QR \ ,$$

where $Q \in \mathbb{C}^{M\times N}$ has orthonormal columns, $R \in \mathbb{R}^{N\times N}$ is upper triangular and $\Pi \in \mathbb{R}^{N\times N}$ is a permutation matrix from a pivoted QR. We can split that into

$$G\Pi = QR = \begin{bmatrix} Q_{left} & Q_{right} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \approx \begin{bmatrix} Q_{left} & Q_{right} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & \cancel{R_{22}} \end{bmatrix}$$

where $Q_{left} \in \mathbb{R}^{N\times s}$, $R_{11} \in \mathbb{R}^{s\times s}$ (and appropriate other sizes), whereas we can guarantee that $\|R_{22}\| = \mathcal{O}(\sigma_{s+1})$. We approximate by disregarding R_{22} and simplify to

$$G\Pi \approx \begin{bmatrix} Q_{left}R_{11} & Q_{left}R_{12} \end{bmatrix} = Q_{left} \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}$$

We now write

$$G_{col} = Q_{left}R_{11}$$

where we term G_{col} as the first s columns of $G\Pi$; since Π represents a permutation matrix G_{col} turns out to be a subset of columns of G. We refer to the source points corresponding to the columns G_{col} as the skeleton points to this node. Using the approximation assumption

$$G_{col}P \approx G\Pi$$

we rewrite

$$Q_{left}R_{11}P = Q_{left} \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}$$

and simplify this expression to

$$R_{11}P = \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}$$

It remains to solve a system of linear equations for $P \in \mathbb{R}^{s\times n}$; it appears that P involves an $s \times s$ identity matrix, namely (in MATLAB notation, where the backslash operator ($\backslash$) denotes solving a system of linear equation,) [Rei17]

$$P = \begin{bmatrix} Id_{s\times s} & R_{12}\backslash R_{11} \end{bmatrix}.$$

In conclusion, we calculate a low-rank factorization of a matrix block using the interpolative decomposition; we have bounded the error in terms of the $(s + 1)$st singular value. However, those rank-revealing factorizations are less reliable than the SVD [Beb08].

In literature, interpolative decompositions are numerically often computed using randomized approaches. We use the rank-revealing QR decomposition for simplicity and better theoretic guarantees. An asymptotically more efficient algorithm that involves randomized linear algebra is described here [LWM$^+$07], which could replace our version in the future. The randomized interpolative decomposition is used in the software framework STRUMPACK [RLGN16].

2.2.3 Neighbor calculation using randomized k-d trees

We get approximate *skeletons*, the “important” columns, by a rank-revealing interpolative decomposition. To compute this decomposition on all rows of the matrix would be too expensive. Leverage scores are a theoretical measure of the importance of rows, but they are too expensive to compute in practice. As a heuristic, we use the assumption that close points are important and far points can be agglomerated. By this, we can significantly reduce the number of rows for the low-rank decomposition. We randomly sample from “neighbors”, i.e. important rows.

To compute neighbors, we use randomized k-d trees (RKDT). A k-d tree is a high-dimensional binary tree for points in k-dimensional space. [DECM98] Gram vectors are usually very high dimensional, as they are possibly in the N-dimensional space. Usually, there is an underlying structure, leading to lower-dimensional subspaces. An RKDT is known to adopt to intrinsic dimension and, therefore, the algorithm of choice for finding neighbors in high-dimensional space.

We split our index sets $\mathcal{I}$ hierarchically at a median pivot in projection to some random direction $\phi_p - \phi_k$. This can be done with either two Gram notions ℓ_2) and $\triangleleft$, as the user desires. It splits the index set into into two children *left* and *right*, $\mathcal{I} = \mathcal{I}_l + \mathcal{I}_r$ according to

(ℓ_2) a fictive orthogonal hyper-plane defined by the median,

($\triangleleft$) or fictive hyper-cones defined by a (roughly) equal-sized split.

We do this hierarchically to a leaf node size of roughly the number of desired neighbors `k` (user parameter), i.e. for the RKDT `m`=`k`. For each point on the leaf level, buckets of candidate neighbors are formed (all points that were in a leaf together). This process of getting candidate neighbors can be repeated several times, employing the randomized splits and getting more potential candidate neighbors.

For all points, we compute distances to all their candidate neighbors exhaustively. We sort them and form a list of `k` nearest neighbors. We sample from all neighbors of the respective leaf for the relevant rows.

It is suggested to over-sample a few rows too many to make sure to catch the interactions (exploration). We suggest sampling about twice as many rows as the maximum allowed rank of the off-diagonals, $\mathtt{s_m ax}$ (user parameter).

In summary, we calculate neighbors approximately using a randomized k-dimensional tree, short RKDT. From these neighbor lists, we sample rows that capture somewhat the most important behavior, i.e., resemble the leverage scores. Therefore, we reduce the complexity of low-rank decompositions by reducing the size of the respective matrices.

2.2.4 FMM vs. HSS representation

In the results chapter and in $\mathcal{H}$-matrix literature, the terms `FMM` and `HSS` often occur. `FMM` stands for fast multipole method and `HSS` for hierarchical semi-separable matrices. In `GOFMM`, we use this *user-defined* criterion for Near/Far pruning, i.e., to decide whether a point is near or far. In this section, we clarify the two $\mathcal{H}$-matrix variants.

A fast multipole method (`FMM`) originates from molecular dynamics in computational physics. There, one defines a cut-off radius at which one needs to compute an exact `Near` field interaction. There are different variants of keeping track of them (e.g., Verlet lists), but the goal is always to approximately accurately catch all near points, the ones that are within a *user-defined* distance range. All points inside this circle/ball are regarded individually, and all points outside are considered for the far field and approximated.

Recall that we introduced `GOFMM` as a hierarchical matrix variant of the form (see also Equation 1.1)

$$\tilde{K} = D + UV + S \ ,$$

where D is *block-diagonal* with *every block being an $\mathcal{H}$-Matrix*, U and V are *low rank* approximations, and S is *sparse*.

By an equalized split, some points on the border may have some points that are within the cut-off radius inside its leaf, but some may belong to another leaf despite being near (inside the cut-off radius). In `GOFMM`, we resemble those `Near`-lists with a sparse "update" matrix S.

However, the distances considered are in Gram vector space, and thus in a very high-dimensional space. Euclidean distances in high dimensions become less meaningful, and often, all points are relatively equally far away. Our remedy is to refrain to neighbor lists, with which we can roughly keep track of the ratio of `Near`-interactions. We define all points that are neighbors to each other as being regarded in the near field and all others as being in the far field. The same heuristic notion has been made in `ASKIT` [MB15], which is also a high-dimensional fast multipole method and served as an inspiration for `GOFMM`.

In regular $\mathcal{H}$-matrix variants, usually, it is assumed that the close interactions are sufficiently covered by the low-rank factors U, V. We let the user decide whether to follow this trust-worthy assumption or whether to improve the multiplication approximation by additional `Near`-points. Hence, a user can decide on the parameter `FMM` or `HSS`.

The separate sparse interaction matrix S complicates the pseudo-inverse calculation significantly. For reference, see the Sherman-Morisson-Woodbury in Subsection 2.1.4. This Schur complement strategy would need to be done again, with multiplications of dense and sparse matrices, resulting in dense matrices. We avoid this complication and allow the pseudo inverse only for `HSS`.

2.2.5 Summary

In Section 2.2, we introduced the foundations necessary for the `GOFMM` $\mathcal{H}$-matrix variant. We started in Figure 2.2.1 by a geometric hierarchical interaction calculation algorithm, which is implemented in the *approximate kernel independent treecode* (ASKIT), which is a predecessor of `GOFMM`.

Next, we introduced in Subsection 2.2.2 the Gram notion for the geometry-oblivious fast multipole method, called `GOFMM`, which was developed along this thesis. We defined a Gram-ℓ_2 and a Gram-angle criterion. We also describe the interpolative decomposition, which we use for computing low-rank representations of the off-diagonals.

In Subsection 2.2.3, we explain our neighbor calculation algorithm using randomized k-dimensional trees. In high-dimensions, and the Gram vector space is high dimensional by nature for most last matrices, plain distances are not very meaningful. Hence, as a remedy, we use neighbors for an estimate of near and far point lists. We use the neighbor sets for randomly sampling rows and for near-far pruning.

Lastly, in Subsection 2.2.4, we explain the difference between the `FMM` (fast multipole method) $\mathcal{H}$-matrix variant and the pure `HSS` (hierarchical semi-separable) version. Whether to use `FMM` or `HSS` can be chosen by the user, i.e. in a user parameter in `GOFMM`. Since we are working in very high dimensions, we do not support a user-defined cut-off radius; instead, we approximately define the near-field from aggregated neighbor lists, and consider everything else in the far field. Using `FMM` can enhance the multiplication accuracy. Note that a pseudo-inverse can only be computed with the `HSS` variant for practical reasons in `GOFMM`.

2.3 Applications of matrix algorithms

Following the theoretical foundations of $\mathcal{H}$-matrices and `GOFMM`, we dedicate this section to scenarios where $\mathcal{H}$-matrices can show significant benefits. Here, we describe *two* different examples, as they are representative methods for the field of statistics and manifold learning combined with numerical linear algebra.

We created the test cases ourselves to have complete control over the analysis. We experimented excessively with Gaussian kernel matrices. The most significant advantage of Gaussian kernel matrices is the ability to compare the *geometry-oblivious* scheme using matrices only with the *geometric version* using point clouds.

In a second step, we look at a recent manifold learning algorithm called diffusion maps, where an eigendecomposition is the computational bottleneck. We address this issue and look at the traditional methods from the established field of numerics. For example, we have chosen the Arnoldi iteration since it involves only matrix-vector multiplications. Hence, we describe the field of manifold learning and numerics *vividly* by exemplary algorithms of the field. This combination opens myriad possibilities for other coupling schemes. Therefore, we describe two representative examples from manifold learning and numerical linear algebra. We have coupled them using the $\mathcal{H}$-matrix-variant `GOFMM`. Consequentially, in Subsection 2.3.2, we describe the diffusion maps algorithm and show that the computational bottleneck is the eigendecomposition. In Subsection 2.3.3, we explain an iterative eigendecomposition algorithm, where we used `GOFMM`'s fast matrix-vector product.

2.3.1 Gaussian kernel density estimation

Since we use Gaussian kernel matrices excessively in the results section, we define the Gaussian kernel density estimation (KDE) separately in this theory section. We also list a few other kernel functions for point clouds, with the advantage that we can compare the geometry-aware variant *approximate kernel independent treecode* (ASKIT) to it.

Assume that we have point cloud data of the form $X_1, ..., X_n \in \mathbb{R}^d$ that are independently, identically distributed samples from some distribution with probability density function(PDF) $\widehat{f_h}(x)$. Our goal is to estimate the unknown density $f(x)$, with the means of statistics. Then, its *kernel density estimator* can be written as [Par62, Nad65]

$$\hat{f_n}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right) , \tag{2.1}$$

where h is the bandwidth for the kernel, i.e. characterizing the skewness of the curve, and $K : \mathbb{R}^d \to \mathbb{R}$ a given kernel function. The probability density at x sums up all local kernel density at x centering at each different X_i. The kernel function can be chosen according to the statistical distribution. One of the most common examples is the Gaussian distribution [Nad65], that is

$$K(x) = \frac{\exp\left(-\|x\|^2/2\right)}{v_{1,d}}, \quad v_{1,d} = \int \exp\left(-\|x\|^2/2\right) ,$$

defining the Gaussian KDE together with Equation 2.1. In practice, we compute a static kernel matrix of data point clouds. For our purposes, we also evaluate the kernel functions at the same points X_i, leading to a square symmetric matrix. We can also calculate the value at x for a locally fitted Gaussian curve. In the end, we sum up all values of x for the estimated PDF. [Ge20]

There are several other kernels that can have some advantages, depending on the dataset. Examples are the quadratic ($K(x, y) = (x^T y)$), inverse quadratic($K(x, y) = 1/(x^T y)$) or polynomial ($K(x, y) = (x^T y)^d$), the Helmholtz kernel ($K(x, y) = e^{2\pi i\kappa|x-y|}/|x - y|$) and many others.

Kernel functions can be formed from any point cloud data. For high-dimensional datasets, the hope is to find a lower-dimensional substructure. For practical reasons, we sample point clouds from the multivariate Gaussian distribution in 6 dimensions for most of our cases in a random order.

Note that we can also run the geometric code ASKIT as a reference. `GOFMM` often finds a similar partition; other $\mathcal{H}$-matrix codes often assume a suitable ordering given, and hence, `GOFMM` has advantages for such randomly-ordered point cloud kernels.

2.3.2 Diffusion maps

Diffusion maps, a non-linear method for reducing dimensionality in data clouds, estimate information regarding the data manifold without any prior knowledge of the underlying function or data geometry. In contrast to approaches like isomaps that directly rely on Euclidean or geodesic distances, Diffusion maps employ an affinity or similarity matrix generated through a kernel function with only positive and symmetric values. Similar to Gaussian KDE, given a dataset $X = \{x_1, x_2, x_3, \ldots, x_n\}$ and e.g., a Gaussian kernel function, a similarity matrix can be computed as [GRNB23]

$$W_{ij} \;=\; w(i,j) \;= e^{\frac{-||x_i - x_j||_2^2}{\sigma^2}} \;,$$

with x_i, x_j as data points and σ^2 for the bandwidth of the kernel around the point x_i. This is similar to the point-kernel matrix generation from above (K03-K09) but more versatile using the code `datafold`.

As outlined in Algorithm 2.1, the similarity matrix, which is computed inside `datafold`, is normalized with the density Q (degree of vertex) and the density parameter α to capture the influence of the data distribution on our approximations. When $\alpha = 0$, the density greatly influences how the underlying geometry is captured; for $\alpha = 1$, it is pretty independent of the geometry. For these reasons, normalization is done with $\alpha = 1$, and a Markov chain obtains the transition probabilities.

Algorithm 2.1 Diffusion Maps [CL06]

```
1: Compute $W_{ij}$                                                  ▷ Similarity matrix
2: Compute normalized weights $W_{ij}^{\alpha} = \frac{W_{ij}}{Q_i^{\alpha} \cdot Q_j^{\alpha}}$     ▷ $Q^{\alpha}$: Influence of density
3: Define Markov chain $P_{ij} = \frac{W_{ij}^{\alpha}}{Q_i^{\alpha}}$                  ▷ P: Transition matrix
4: Perform $t$ random walks to obtain $P^t$
5: Perform eigendecomposition on $P^t$             ▷ $\lambda_r$: eigenvalues, $\psi_r$: eigenvectors
6: Lower dimension $d(t) = \max\{\ l : \lambda_l^t < \delta\lambda_1^t\ \}$    ▷ $\delta$: Predetermined precision factor
```

The transition matrix P^t is set up by performing random walks for time steps t. For exposing the underlying lower dimension of the dataset, an eigendecomposition is necessary. As for complexities, and reported in practice, this is the most expensive part of the algorithm. For a dense matrix, a direct eigendecomposition requires $\mathcal{O}(N^3)$ operations, while for sparse systems, usually iterate approximate eigendecompositions like the Arnoldi iteration are used. [GRNB23]

2.3.3 Arnoldi iteration

In Subsection 6.5.3, we compute an eigendecomposition of the transition matrix in `datafold` with `GOFMM`. Hence, in this section, we describe the theoretical foundations of this algorithm.

An Arnoldi method belongs to the class of iterative methods that compute eigenvalues and are an extension of the so-called power methods. The power method involves multiplying a matrix with a random vector iteratively and progressively until the dominant components persist, representing the most significant eigencomponents, i.e. of the form $Ax, Ax^2, Ax^3, \ldots$. We use the Implicitly restarted Arnoldi method, which is a slight variation. [GRNB23]

Algorithm 2.2 Algorithm - k-step ArnoldiFactorization(A,x) [Sor92]

```
1: $x_1 \leftarrow \frac{x}{||x||}$                                  ▷ Computes first Krylov vector $x_1$
2: $w \leftarrow Ax_1$                                  ▷ Computes new candidate vector
3: $\alpha_1 \leftarrow x_1^H w$
4: $r_1 \leftarrow w - \alpha_1 x_1$
5: $X_1 \leftarrow [x_1]$                                  ▷ Orthonormal basis of Krylov subspace
6: $H_1 \leftarrow [\alpha_1]$                                  ▷ Upper Hessenberg matrix
7: for all $j = 1...k-1$ do          ▷ For $k$ steps, compute orthonormal basis $X$ and projection
8:     $\beta_j \leftarrow ||r_j||$ ; $x_{j+1} \leftarrow \frac{r_j}{\beta_j}$
9:     $X_{j+1} \leftarrow [X_j, x_{j+1}]$ ; $\hat{H}_j \leftarrow \begin{bmatrix} H_j, & \beta_j e_j^T \end{bmatrix}^T$          ▷ $e_j$ is unit standard coordinate basis
10:    $z \leftarrow Ax_j$
11:    $h \leftarrow X_{j+1}^H z$;  $r_{j+1} \leftarrow z - X_{j+1} h$          ▷ Gram-Schmidt Orthogonalization
12:    $H_{j+1} \leftarrow [\hat{H}_j, h]$
13: end for
```

We list the Arnoldi method in Algorithm 2.2. Note that given a start vector x_1 only matrix-vector products with the matrix A occur. The matrix H is an upper Hessenberg matrix whose eigenvalues converge to the ones of A. If we reduce the complexity of a dense matrix product Ax, we result in a lower overall complexity, depending on the number of steps.

In fact, we use the **Implicitly restarted Lanczos method**, a variant of the Arnoldi factorization. Arnoldi also works for non-hermitian matrices, while a Lanczos requires symmetry. We use the SciPy variant for the implementation of Lanczos methods. [GRNB23]

2.4 Summary

Chapter 2 builds the theoretical foundations for the the developed `GOFMM` framework, and the results in Chapter 6.

We started this chapter with an introduction to hierarchical matrix algorithms in Section 2.1. After defining the $\mathcal{H}$-matrix variant, we showed the textbook singular value decomposition to compute a low-rank representation. We demonstrated how this can be used hierarchically on matrix off-diagonals. Then, we described how a block-wise multiplication scheme for so-called *hierarchical off-diagonal low-rank* matrices can lead to $\mathcal{O}(N \log N)$ computational complexity. With the Sherman-Morisson-Woodbury formula, we can compute an approximate inverse of a block-wise matrix. The applications of $\mathcal{H}$-matrices are versatile. Early use cases include discretizations of elliptic partial equations and integral equations. A more recent software package is the structured matrix package `STRUMPACK`.

In the next Section 2.2, we formulated a geometric N-body problem that also occurs in computational physics, simulations, and data analysis. These geometric hierarchical algorithms are used for the predecessor of `GOFMM`, called *approximate kernel independent treecode* (`ASKIT`). We then introduced the geometry-oblivious distance notion for `GOFMM`, as the Gram-ℓ_2 or Gram-angle distance. We explained the interpolative decomposition, the low-rank approximation for `GOFMM`. We moved on to the neighbor computations using randomized trees and specified the difference between a fast multipole method (`FMM`) and a hierarchical semi-separable format of an $\mathcal{H}$-matrix. Details on the implementation of `GOFMM` can also be found in Section 6.1.

This chapter's last Section 2.3 explained the foundations of our $\mathcal{H}$-matrix application cases. First, we created several matrices by point clouds, for example, using the Gaussian kernel density estimation. This allows us to compare the geometric-oblivious to the geometry-aware variant.

Second, we showed how classical and dusty numerical linear algebra can impact recent data-learning algorithms. Therefore, we explained the diffusion maps algorithms from `datafold`, a representative manifold learning algorithm. The computational bottleneck is the eigendecomposition of the transition matrix. To address this issue, we looked in the field of linear algebra. The Arnoldi method is a representative algorithm for the iterative eigendecomposition. Therefore, we vividly discuss the foundations of the Arnoldi method, which paves the way for using it together with `GOFMM` in the implementation in Subsection 6.2.1. This opens myriad possibilities for more iterative algorithms and other computation-hungry applications.

In summary, this Chapter 2 forms the theoretical foundation of $\mathcal{H}$-matrices for `GOFMM`. The implementations and results using this $\mathcal{H}$-matrix variant are explained in Chapter 6.

Essentially, all models are wrong, but some are useful.

George Box (1919 - 2013)

3 Artificial neural networks

This chapter deals with the second topic of the thesis, namely neural network optimization. It explains the theory behind neural networks for deep learning, hence, it builds the application case for the later Chapter 4 called *Optimization methods*. Results based on this theory are shown in Chapter 7.

In Figure 3.1 we display a common data engineering pipeline. A considerable effort is spent on data pre-processing, including labeling, data cleaning, normalization and data augmentation. A data scientist then searches for the right neural network architecture by experience or using some heuristics. Then a model is trained and packaged. Validating the model with the domain task is important, and once accepted, the model can be deployed and monitored. This is an iterative process, which often needs several cycles of development. Note that enormous compute resources are spent globally on *hyperparameter* tuning, including neural architecture search as well as training parameters. Since such data pipelines are very widespread in engineering, any slight improvement can have a large benefit on the overall system.

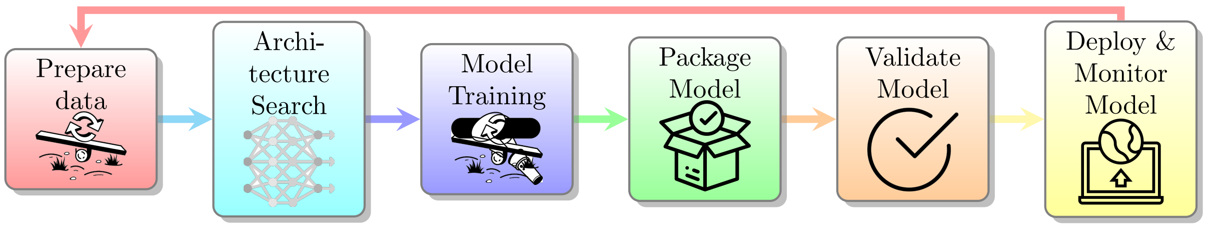


Figure 3.1: Data engineering pipeline. Often, this is an iterative process where considerable effort is spent on data preparation, validation, and model deployment, which we do not describe in this chapter.

This chapter starts with the basic theory of neural networks in Section 3.1, then explains some more sophisticated operations in neural network layers in Section 3.2. Section 3.3 explains how uncertainty - be it in the inputs or in parameter space - can be incorporated in neural networks. Furthermore, we continue with Section 3.4 by explaining the network architectures later used with a second-order optimizer in this thesis. We conclude with as summary (Section 3.5).

3.1 Scientific computing for deep learning

The field of machine learning has many subfields, and the most prominent division is formed by supervised and unsupervised learning. Supervised learning is a subfield of machine learning, where the targets of the samples are known. Unsupervised algorithms, however, find clusters and classes by themselves without this knowledge. [B$^+$95]

In this chapter, we restrict the theory to supervised learning and describe only popular working technologies. Essentially, machine learning describes parametric function approximation. We begin with a one-layer linear regression model in Subsection 3.1.1 and move towards a multilayer perceptron in Subsection 3.1.2.

3.1.1 Linear regression

The motivation of regression tasks is to predict one or more continuous target variables given a d-dimensional input data vector, $x \in \mathbb{R}^d$. With datasets $D = \{(x_i, y_i)\}_{i=1}^N$, where $x \in \mathbb{R}^D$ and $y \in \mathbb{R}$, where x are the input features and y are the target outputs.

The simplest feed-forward model is a linear combination of the input variables

$$f(x_i, W) = w_0 + \sum_{i=1}^{N} (w_i^{(d)} x_i) \ .$$

We then form an error function, e.g. the square loss, which we call l_{sq} for a single sample, and a capitalized L_{sq} for the sum of it, i.e.

$$L_{sq} = \sum_{i=1}^{N} l_{sq} = \frac{1}{2} \sum_{i=1}^{N} (f(x_i, W) - y_i)^2 \ .$$

To find the appropriate weights W, we use optimization algorithms to minimize the loss function, which we describe in Chapter 4.

3.1.2 Multilayer perceptron

Consider a feed-forward neural network defined as the parameterized function $f(X, W)$. The function f is composed by vector-valued functions $f^{(d)},\ d = 1, \dots, D$, which represent each one layer in the network of depth D, in the following way: $f(x) = f^{(D)}(f^{(D-1)}(\dots f^{(2)}(f^{(1)}(x)))$ [RNB23].

The function corresponding to a network layer (d) and the output of the j-th neuron are computed via

$$z_j^{(d)} = \begin{bmatrix} z_1^{(d)} \\ z_2^{(d)} \\ \vdots \\ z_{M^{(d)}}^{(d)} \end{bmatrix} \text{ and } z_j^{(d)} = f_j^{(d)}(z^{(d-1)}) = \phi\Big(\sum_{i=1}^{M^{(d-1)}} (w_{ji}^{(d)} f_i^{(d-1)}) + w_{j0}^{(d)} \Big)$$

with activation function ϕ and weights w. All weights w are comprised in a large matrix $W^{(d)} \in \mathbb{R}^{M^{(d)} \times M^{(d-1)}}$ which represents parameters for f. We denote the aggregation of all weights with the tensor $\mathbf{W}$, i.e. $\mathbf{W} = [W^{(0)}, W^{(1)}, \dots W^{(D)}]$. Hence, we sometimes write $f(X, \mathbf{W})$, where $\mathbf{W}$ is not an input but a model parameter that the function depends on. Note that each W can be of different sizes; hence, it is a list of different matrices with potentially different sizes.

For a supervised learning task, the optimization problem consists now of finding weights $\mathbf{W}$ such that a given loss function l will be minimized for given training samples X, Y,

$$\min_{\mathbf{W}} L(X, Y, \mathbf{W}) \ . \tag{3.1}$$

A prominent example of a loss function for binary classification is the categorical cross-entropy

$$L_{entr}(X, Y, \mathbf{W}) := - \sum_{i=1}^{\mathtt{N}} y_i \log(f^{(D)}(X, \mathbf{W})) \ .$$

Note that only the last layer function $f^{(D)}$ of the network directly shows up in the loss, but all layers are indirectly relevant due to the optimization for all weights in all layers [RNB23].

Stochasticity: Optimizers look at stochastic *mini batches* of data, i.e. disjoint collections of data points. The union of all mini batches will represent the whole training data set. The reason for considering data in chunks of mini-batches and not in total is that the backpropagation in larger neural networks will face severe issues w.r.t. memory. Hence, the mini-batch loss function is now defined by

$$L_{entr}(x, y, \mathbf{W}) := -\sum_{i=1}^{\text{batch-size}} y_i \log(f^{(D)}(X, \mathbf{W})) \ ,$$

where the mini batch is varied in each optimization step in a round-robin manner [RNB23].

Nonlinear Activation Function: After finishing the previous linear or convolutional operation, one then applies a nonlinear activation function ϕ. The output from the previous linear operation is passed to a nonlinear activation function [YNDT18]. Some examples are *sigmoid*, *tanh*, *softmax* and *rectified linear unit* (ReLU). ReLU, a widespread activation function is $\phi(x) = max(0, x)$. Sigmoid is chosen to be advantageous for differentiation, i.e. $\phi(x) = \frac{1}{1+e^{-x}}$. Softmax is commonly used in the last layer, as it normalizes the exponential of the vector z, i.e. $\phi_i(x) = \frac{e^{z_i}}{\sum_{j=0}^{M} e^{z_j}}$. In multi-class classification, another advantage of softmax in the final layer is that it can be seen as a probability of belonging in a certain category. [LBH15]

3.2 Convolutional neural networks

A multilayer perceptron maps the input linearly to the next layer; non-linearities can be achieved with a nonlinear activation function. This, however, is especially for images not the most efficient way. Computers need to capture information intelligently to be efficient and accurate. Similar to image compression algorithms, a convolution is an efficient way to grasp pixel semantics. Therefore, convolution operations have become imperative for image processing and have been shown to be vital for image classification with deep learning. *Convolutional Neural Networks* (CNN) are an artificial neural network layer with at least one convolutional layer. [GBC16b] These layers occur in most image processing and image recognition pipelines. CNNs apply, besides other ingredients, convolution kernels of different sizes in different layers in a sliding window approach to extract features. For a brief introduction to CNN, e.g. see [LBH15]. For example, they also occur in many layers of the prominent ResNet-50 network structure: It has 50 layers in total, with around 10 with convolutions; the interesting part of ResNets are, however the *skip connections* to avoid the problem of diminishing gradients, that we deal with later in Subsection 3.4.2.

Here, we lay the foundations of CNNs, also depicted in Figure 3.2. The input picture of the car with its contours can be well captured with a convolution, as shown in the figure. The figure also shows that many convolutions and pooling are performed in the early layers, which they call feature learning. It is followed by the so-called classification block with flatten and fully-connected layers, and in the last layer a Softmax normalization, as described previously. Therefore, we start this section with the convolutional operator in Subsection 3.2.1, how a convolution is applied to a pixel image. We move pooling layers in Subsection 3.2.2, a dropout layer in Subsection 3.2.3, and close with the fully connected layer in Subsection 3.2.4.

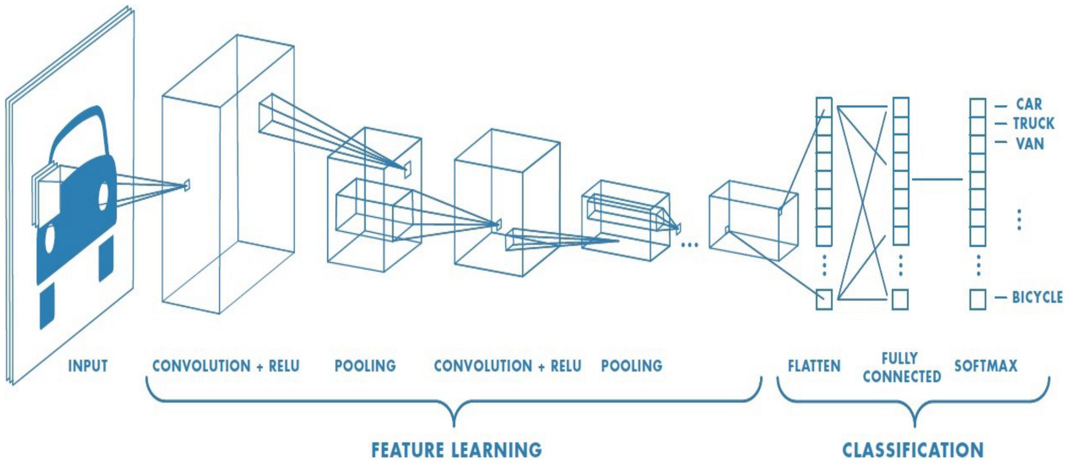


Figure 3.2: Convolutional Neural Network Architecture Example. Figure taken from [Sah].

3.2.1 Convolutional layer

A *convolutional layer* consists of a convolution operation or a small tensor multiplication. In general, the convolution is an operation on two functions I and K; the first, I, is the color intensity (pixels), and the latter, K, the chosen kernel. It is defined by

$$S(t) = (I * K)(t) = \int I(a)K(t-a)da \ .$$

If we use a 2D discrete pixel image I as input with a 2D discrete kernel K, we can write this two-dimensional discrete convolution as a matrix multiplication with

$$S(i,j) = (I * K)(i,j) = \sum_x \sum_y I(x,y)K(i-x, j-y) \ .$$

Color images have at least a channel for intensity of red, green, and blue at each pixel position (hence, the abbreviation RGB). Assume that each image is a 3D-tensor and $V_{i,j,k}$ describes the value of channel i at row j and column k. Then we can write our kernel as a 4D-tensor with $K_{l,i,j,k}$ denoting the connection strength (weight) between a unit in input channel i and output channel l at an offset of k rows and l columns between input and output [RNB23].

A kernel is basically a sliding window of smaller size than the whole picture that moves over the whole image. This allows features, such as a contour change, to be captured in one or more dimensions in the hidden layer. The network architecture designer defines the so-called kernel size and the stride, to characterize the kernels:

- **Kernel size:** Basically the size of the moving window. The network designer has empirically chosen values, and, to our knowledge, there is no rigid theoretical explanation on choosing the values. The kernel size is usually square and an odd number, typically such as 3×3, and sometimes 5×5 or 7×7.

- **Stride:** The stride is the offset between overlapping different kernel windows. Naturally, this defines, therefore, the number of kernels for this layer, and hence, the width of the feature layer. Usually, several convolution layers are stacked, so the network designer chooses the depth of the feature map.

A similar occurence can be seen in the previously described Figure 3.2, especially in the block of feature learning. The contours of the car object are best captured with a convolution.

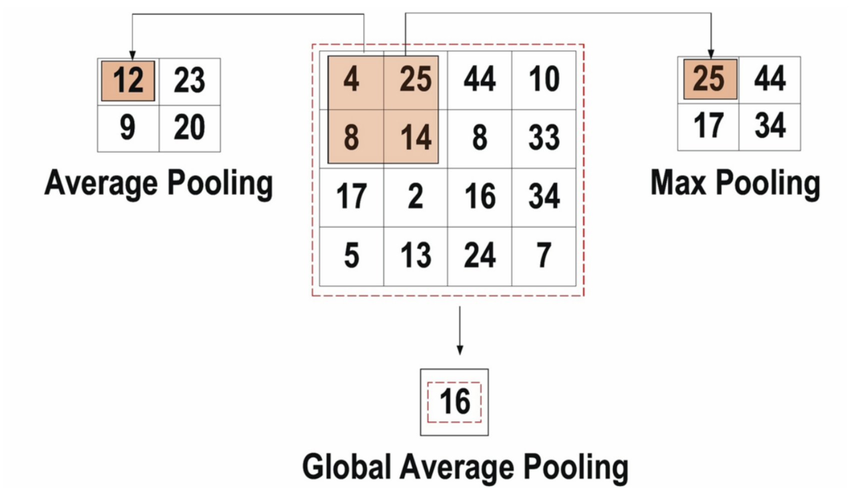


Figure 3.3: Three types of pooling operations. Figure extracted from [AZH+21].

To sum up, the main objective of this operation in the CNN architecture is to extract features. It starts from low-level features such as edges, orientation, and colors to high-level features such as object classification, recognition, and segmentation. The first layers capture the low-level features, while adding layers will capture high-level features.

3.2.2 Pooling layer

The main aim of the pooling layer is to downsample and diminish the feature map's dimensionality. It lowers the number of parameters and also simplifies the model's complexity [ON15]. A visual explanation could involve the observation that in real-life images, not all image regions are equally important, but rather, essential sections could be pooled. The pooling is done mathematically by taking a portion of pixels from one layer and reducing it to just one pixel in the next layer. There are two types of pooling:

- **Max Pooling:** Max pooling is the most popular and empirically best technique. It extracts a portion from the input activation values (sometimes called feature map), and it outputs the maximum value, discarding all other values [YNDT18]. The most common filter size is 2×2, with a stride equal to 2. Of course, the height and width can change here, but the depth often remains unchanged. Moreover, with this setup, the feature map is reduced by a factor of 2 per dimension.

- **Average Pooling:** Average pooling extracts a portion from the input (sometimes called feature map) and calculates the average value from all values. The output result is the average of all values in the relevant section. Similarly, with this setup, the feature map is reduced by a factor of 2 per dimension.

The process is visualized and summarized in Figure 3.3. The relevant section is the 2×2 section in the left upper corner. The average of the left upper corner is roughly 12; the digits after the comma are cut off. The other numbers of average pooling display the other 4 sections. Max pooling for the upper left corner yields 25, and the other numbers for the other sections, respectively. These 2D operations are eliminating both 3/4 of the parameters, respectively. The global average of the whole field yields 16.

3.2.3 Dropout layer

In most networks, dropout layers can also be found, which are necessary and state-of-practice. We only loosely define it here, as it is not a mathematical operation in the feed-forward evaluation mode. A dropout layer is only used for training, and it is a simple way to prevent overfitting. It basically thins out the fully-connected net by stochastically dropping some connections during training. This allows other features to be strongly trained, which could have faded in a full training process.

For evaluation, a dropout layer is void, i.e. the fully-connected net is evaluated regarded without a dropout. For training, a dropout improves regularization and prevent overfitting. [SHK+14]

3.2.4 Fully-connected layer

After performing all the convolutional and pooling layers, the output feature map is flattened (converted to a 1D array). This array is then often connected to one or more fully-connected layers, similar to the multilayer perceptron model in Subsection 3.1.2. A weight matrix is used to connect each input node to each output node. As visualized in Figure 3.2, this process concludes the feature extraction phase, and we have the flattened output feature map. We then enter the classification phase with many fully-connected layers. Activation functions, such as ReLU or sigmoid, are formed around each fully-connected layer, in the same way as the multilayer perceptron. In some sense, a fully-connected layer is the same as a layer in the multilayer perceptron model. In the last layer, there is usually the same number of neurons in dimensions for the fully connected layer as classes in our task.

As mentioned for activation functions in Subsection 3.1.2, in the last layer for multi-class classification we often apply a *softmax* function (also occurs in Figure 3.2). We expect a difference between the nonlinear activation function values on the last fully-connected layer and the output class feature. The shear value is irrelevant, but instead we need to put the value in relation to other values on this level, to get a rough probability of a class. Softmax helps by producing class probabilities; it outputs values ranging from 0 to 1 with a summation of all values equal to 1. Thus, the activation function of the last layer is usually a softmax function for the multi-class classification task. [YNDT18]

3.3 Uncertainty in deep learning

One of the biggest challenges in all areas of machine learning is deciding on an appropriate model complexity. Models with too low complexity will not fit the data well, while models possessing high complexity will generalize poorly and provide bad prediction results on unseen data, a phenomenon widely known as *overfitting*. Three commonly deployed strategies to counteract this problem are (1) hold-out or cross-validation on one hand, where part of the data is kept from training in order to optimize hyperparameters of the respective model that correspond to model complexity, (2) controlling the effective complexity of the model by inducing a penalty term on the loss function on the other hand, or (3) a dropout layer (mentioned before in Subsection 3.2.2). The second approach is known as regularization. Another approach is applying Bayesian techniques on neural networks, such as weights as a probability distribution. A dropout layer, approach (3)m can also be seen as a Bayesian approximation. [B+95]

Deep learning is used for safety-critical fields such as medical image processing and autonomous driving, where we need to quantify uncertainty reliably. This theory section aims to introduce the Bayesian techniques for our purposes of Bayesian neural networks, for which we run second-order optimization later in Chapter 7. We start this section by introducing the different types of uncertainty (Subsection 3.3.1), and move towards the frequentists approach in Subsection 3.3.2 and the necessary variational inference (Subsection 3.3.3), introduce the so-called implementation type Bayes by Backpropagation (Subsection 3.3.4) .

3.3.1 Types of uncertainty

In general, there are two types of uncertainty. *Aleatoric* uncertainty captures noise in the data, possibly induced, for example, by measurement inaccuracy. Thus, it cannot be reduced by increasing the amount of training data but can be treated by frequentist neural networks, i.e., a probabilistic network that is applied several times. On the other hand, *epistemic* uncertainty can be understood as the uncertainty about which model is appropriate, which has been difficult to judge in computer vision. The model uncertainty can be reduced by additional data and, equally important, by quantifying the epistemic uncertainty for unseen data, out-of-data samples that are not properly represented by the training data can be identified as such. Probabilistic approaches to deep learning exist for both types of uncertainty. Aleatoric uncertainty can be modeled by placing a distribution on the output of the model [KG17], while for epistemic uncertainty, one would place a prior distribution on the network weights, as described in Subsection 3.3.3. In [KG17] Kendall and Gal also identify scenarios where either of the two types of uncertainty is especially important. They argue that *aleatoric* uncertainty should be treated in situations with large amounts of data, where epistemic uncertainty is sufficiently small, as well as in real-time applications where Monte Carlo sampling is too expensive. In contrast, *epistemic* uncertainty should be carefully treated in cases of small data sets with sparse data and in safety-critical situations, where out-of-data samples need to be recognized. However, they also emphasize that aleatoric and epistemic uncertainty can be taken into account simultaneously and showcase how this can be achieved. [KG17]

For the sake of this thesis, this motivated us for Bayesian neural networks, with which we can approach *epistemic* model uncertainty and *aleatoric* with a frequentist's approach.

3.3.2 Frequentist approach of treating prediction uncertainty

The ultimate goal of modeling uncertainty is to express the uncertainty in the prediction in a meaningful and comprehensive way. First, multiple forward passes and, hence, predictions are conducted for each sample and averaged over the class probabilities, catching measurement errors and, thus, aleatoric uncertainty. Steinbrener [SPP20] suggests to use these predictions to compute quantiles and check whether all predictions in the considered credible interval vote for the same class. If that is not the case, they consider the prediction to be unconfident. Moreover, we are interested in the ratio of predictions in relation to the overall prediction in order to receive a continuous metric of confidence. Another straightforward approach is to set a limit to the average class probability

$$p \geq \frac{1}{\text{number of classes}}$$

mark overall predictions below p as unconfident and check if a low-class probability coincides with a high standard deviation among the predictions. Note that it is not sufficient to consider a *single* prediction of a Bayesian neural network to judge its confidence. Information about confidence can only be provided by an ensemble of several predictions together with an additional quantity, such as standard deviation or a confidence interval. [Wei21, G^{+}16]

In addition, Kendall and Gal gradually remove samples with low prediction confidence from the test set in order to verify the correlation between confidence and accuracy. [KG17]

3.3.3 Variational inference

Due to the large number of weights in deep learning models and the activation nonlinearities, the exact weights posterior is intractable. Therefore, a family of tractable surrogate distributions is considered, and eventually, the member that incarnates the best approximation is picked as surrogate posterior. A common choice for the family of distributions are parametric distribution, where the parameters remain to be determined, e.g. Gaussians with variable mean and variance. [B+95]

In order to deduce a measure for the goodness of approximation, start by considering the evidence $p(D)$.

The log-evidence $\log p(D)$ can be decomposed as

$$\log p(\mathcal{D}) = L(q(w|\theta)) + D_{KL}(q(w|\theta)p(w|\mathcal{D}))$$

where $q(w|\theta)$ is a member of a fixed family of parametric distributions,

$$L(q(w|\theta)) := \int q(w|\theta)\log(\frac{p(w,\mathcal{D})}{q(w|\theta)})dw$$

and

$$D_{KL}(q(w|\theta)p(w|\mathcal{D})) := -\int q(w|\theta)\log(\frac{p(w|\mathcal{D})}{q(w|\theta)})dw$$

denotes the Kullback-Leibler divergence from $p(w|\mathcal{D})$ to $q(w|\theta)$.

The first term of the decomposition is referred to as evidence lower bound (ELBO). From $p(D)$ being constant, it also follows that maximizing the ELBO is equivalent to minimizing the Kullback-Leibler divergence that will serve as an objective to determine the best surrogate posterior. [B+95] We use this ELBO term in the loss function for our implementation in the framework Tensorflow Probability for Subsection 7.2.4.

3.3.4 Bayes by backpropagation

Bayes by Backprop is a variational inference scheme for learning the posterior distribution of the weights of a neural network. It assumes that weights are drawn from Gaussian probability distributions with mean and variance, i.e. $\mathcal{N}(\theta|\mu,\sigma^2)$. In Bayes by Backprop (BBB) the variational free energy is minimized. The posterior consists of the likelihood and the evidence, resulting in a sum of the log-likelihood and a Kullback-Leibler divergence term. [BCKW15]

In this subsection, we follow the BBB approach and consider probability distributions over the weights of a neural network [B+95], specifically the posterior distribution of the weights given the data $p(W|\mathcal{D})$. Applying Bayes' theorem yields

$$p(W|\mathcal{D}) = \frac{p(\mathcal{D}|W)p(W)}{p(\mathcal{D})},$$

where $p(\mathcal{D})$ serves as normalization constant and is referred to as evidence, $p(\mathcal{D}|W)$ is called the likelihood function and $p(W)$ is the prior probability that captures prior assumptions about the distribution of the weights. The Bayesian prior is mathematically similar to L_2-regularization [VVMA19]. Due to the fact that the concept of regularization already comes with the Bayesian framework, an important benefit of BNNs is that they are robust to overfitting without the need to hold out a validation set [SLL19], which makes them suitable for small data sets. Furthermore, since the network weights are probabilistic, it is possible to assign a measure of confidence to predictions and, as a consequence, circumvent overconfident predictions for unseen data that is not represented well by the training data.

We use the implementations from the `Tensorflow Probability` framework, see also Subsection 7.1.2. In a nuanced approach, instead of convolutions and dense layers, we use TensorFlow Probability's Dense Variational and Flipout layers (1D/2D, respectively). DenseVariational implements a dense multiplication layer with a weight probability distribution identical to the Bayes by Backprop approach [tfpb]. Flipout implements a convolutional layer of the aforementioned Flipout estimator by Wen et al. [tfpa, WVB+18]. In image recognition, often 2D-convolutional layers occur. In order to include a Bayesian term in a regular visual network, we re-factored such convolutions with a 2D-Flipout estimator.

3.3.5 Summary

In summary, in this section, we started with the different types of uncertainty, epistemic uncertainty introduced by the model and aleatoric uncertainty, e.g., by input measurement noise. We show that we can quantify this prediction uncertainty by a frequentist approach, that is, with a histogram of forward runs. A variational inference model gives us an estimation to get the best approximation function to the intractable posterior. For Bayesian neural networks, we limit ourselves to the Bayes by backdrop approach, which sets a probabilistic distribution of mean and variance on the network weights. With variational inference, we minimize the likelihood and evidence terms to get the optimal probability distribution of network weights, allowing us to estimate uncertainty and other interesting analysis tasks.

3.4 Network architectures

In this section, we describe some more sophisticated network architectures with their theory, i.e., the architectures for which we apply second-order optimization in Chapter 7. We do not dwell on the theory for finding a suitable network design, as this involves the history of developments in the quickly growing field. We refer to the initial models with their mechanisms occasionally. In this section, however, we focus on popular network architectures that we used for second-order optimization in Chapter 7, including the network features and their consequences.

We start with shallow networks with regression and convolutions in Subsection 3.4.1, and move towards the residual nets for image classification in Subsection 3.4.2. In this section, we also describe the most complicated network design in the transformer architecture in Subsection 3.4.3. We close, as always, with a summary, mentioning the building blocks of the network architectures (Subsection 3.4.5).

Due to the model complexity of computer vision and natural language processing architecture structures, their subsections are more lengthy. We regard the simple models as equally important for second-order optimization, as we can find relevant optimization behavior patterns. Similar building blocks can be found in ResNets and transformers, as well as their derived models.

3.4.1 Shallow networks

We handcrafted small architectures for small datasets to test our algorithms and find optimization behavior. We started with one-layer networks, similar to regression problems. We also experimented with convolutional neural nets for some image classifications, which we describe afterwards.

Regression problems

A linear regression problem is mathematically equivalent to a one-layer network with respective weight matrix multiplication and activation functions. For linear regression, typically, a least-square problem is solved; for neural networks usually, gradient-based algorithms are used. Even though the algorithm for finding the weights is often different, the evaluation in the feedforward mode is equivalent.

The literature on shallow neural networks is vast; here, we give a general introduction. In [BCFW21], a theoretical convergence analysis was done for one-layer networks by showing strong convexity and convergence for infinitely wide networks. We do not dwell on the results, because its practicability is limited. As most algorithms depend heavily on the network architecture and the dataset, we refrain to empirical measurements for `Newton-CG`.

Convolutional Nets

For small datasets in data analysis, a small network is usually sufficient. *Convolutional nets* are networks that involve at least one convolution layer. A convolutional first layer can greatly improve accuracy for datasets involving small images. Usually, 2D convolutions are applied on images, followed by some lower dimensions. A similar approach is followed for the so-called U-Net architecture [LSD15], which is now very popular in network design. The underlying foundation is a dimensionality reduction using several layers to a latent space with skip connections. We used self-crafted convolution nets that we stacked together for completeness. Our aim was, however, to address popular large models, as described in the next section.

3.4.2 Computer vision networks

Our goal was to test the algorithms on state-of-the-art networks that are popular in computer vision. We describe Residual Nets (short: ResNets) and the adaptions and extensions MobileNet and InceptionNet in separate sections, which in Chapter 7 we use for second-order optimization. We also look at the NasNet, a strategy for neural architecture search.

ResNet

Residual Networks (ResNet) are one of the most prominent architectures in deep learning for Computer vision; it was introduced in 2015 by Shaoqing Ren, Kaiming He, Jian Sun, and Xiangyu Zhang [HZRS16]. In order to solve more complex tasks, the trend is to make architectures deeper by adding more layers to the architecture, which leads to improving performance and accuracy of classification and recognition tasks, among other side effects. However, one side-effect of adding more layers (i.e., deeper networks) is that training becomes harder. The so-called problem of *vanishing gradients* appears, which means the gradient for parameters deeply hidden in the network becomes embarrassingly small, so it becomes very slow in training. This leads to saturation and degrading of accuracy. [HZRS16]

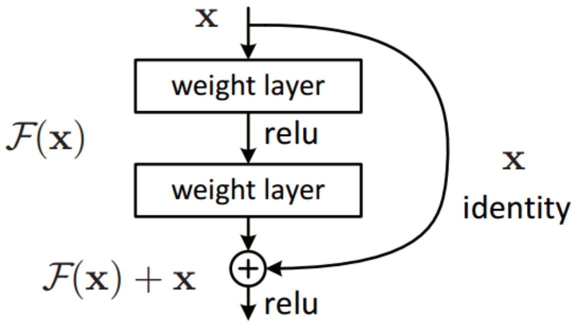


Figure 3.4: Residual learning: ResNet building block. Figure adapted from [HZRS16].

ResNets overcome the problem of *vanishing gradients* mentioned above by using the Residual blocks (in each ResNet layer), i.e., adding another bypass connection. In Figure 3.4, the residual block ResNet is shown, where we see a direct connection that allows us to skip one or more layers in the model. This is the core part of the ResNet part, sometimes also called *identity shortcut connection*, or just *skip connection*, and addresses the problem of vanishing gradients. [HZRS16] These skip connections tackle the vanishing gradient problem by doing two main things:

1. Allowing the gradient to flow through a shortcut path.
2. Skipping layers that affect the performance by regularization.

As shown in the figure, the residual block is formed by skipping a layer or more in between using the skip connection method. Stacking these residual blocks together forms the ResNet.

ResNets consist of these residual blocks, which open a family of different architectures associated with it. The popular models are usually denoted by their depth, for example. The first version of a ResNet was ResNet-34, which has 34 layers. Its main idea was to insert the shortcut connection to turn the plain network into a residual network. The plain network is inspired by the VGG networks, which have a convolution network of 3×3 filters [HZRS16]. If we compare VGG and ResNet, we deduce that ResNet has lower complexity with fewer filters. The ResNet had the same number of filters for each identical output feature map size and inversely scaled the number of filters to the feature map size. When the feature map size is halved, the number of filters is doubled, as this maintains the time complexity per layer.

The shortcut connections are used directly through addition, as the input and output dimensions are the same. Two implementations are possible, zero-padding or addition projection shortcuts. The first would increase the dimension, while the latter uses an addition to match the output size. [Rui18]

There are other updated versions of ResNet, such as the popular ResNet50, ResNet101, and ResNet152. ResNet50, for example, is based on the basic ResNet model but has one major difference. It uses a stack of 3 layers instead of 2 originally used in the basic ResNet model. This decreases the time taken to train the layers and results in better accuracy than the basic ResNet model. ResNets are a popular image classification benchmark for optimizers; [YGS+21] used a ResNet18 on ImageNet, which reached similar values regarding accuracy and speed compared to our ResNet50 model.

MobileNetV2

MobileNetV2 is a CNN model that was developed by Google. It is an enhancement model that was built and developed after the original, very similar model MobileNet. However, MobileNetV2 model uses inverted residual blocks with bottleneck features [SHZ+18]. As observed from its name, MobileNet models were introduced to be used to do deep learning tasks on the mobile device itself without the need to send the data to the server to do the work and then send the results back to the mobile device. To accomplish this, the size of the network and the complexity cost are reduced, resulting in the name MobileNet. Thus, in the original MobileNet model, depth-wise separable convolution is introduced, which helps in reducing the size and cost, and is suitable for any device with low computational power, such as mobile devices. In MobileNetV2, the performance and the memory efficiency is better than the original model by introducing the inverted residual blocks.

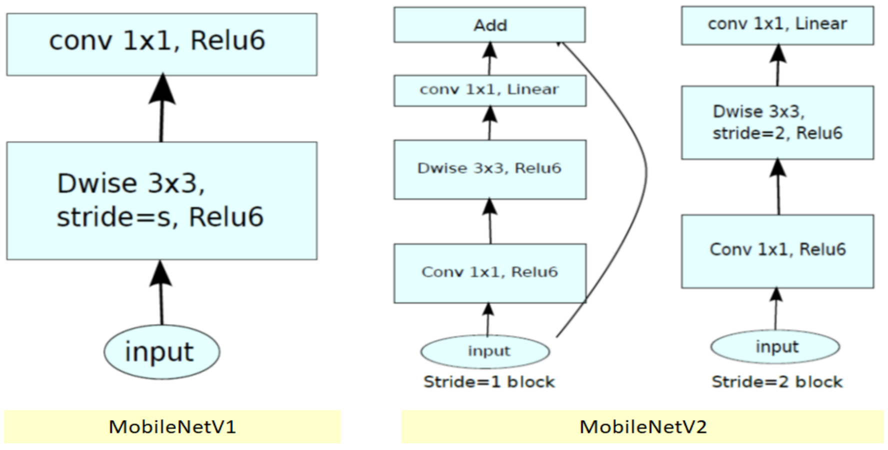


Figure 3.5: Comparison of convolutional blocks between MobileNetV1 and MobileNetV2 models. Figure taken from [SHZ+18].

There are two types of blocks in MobileNetV2; one is a residual block with a stride of 1, and the other uses stride 2 for downsampling. Each type of blocks contains 3 layers [SHZ+18]. These layers are represented as follows:

1. **First Layer:** 1×1 convolution with ReLU6 (Relu6 caps the linearity at 6; robust when used with low-precision computation).
2. **Second Layer:** depthwise convolution works by applying a single convolutional filter per input channel. So, a lightweight filtering is achieved.
3. **Third Layer:** 1×1 convolution without non-linearity (ReLU6) to reduce the depth and cap the increase of linear influence.

The overall architecture of MobileNetV2 is summarized in Table 3.1.

To summarize, this model developed by Google is an efficient improvement for devices with low computational power, such as mobile phones. They allow the performance of some ML tasks on the device without a process in the backend in a cloud. MobileNetV2 is similar to the original MobileNetV1 model but has some modifications in convolutional kernel sizes that enhance the model's accuracy.

InceptionV3

InceptionV3 is a CNN deep learning model that was also developed by a team from Google. This model is an upgrade and a superior version of the original and basic model "Inception".

Table 3.1: Overall Architecture of MobileNetV2 model. "bottleneck" is one of the upper mentioned conv blocks with a linear layer before the summation, see Figure 3.5.

Input	**Operator**	expansion factor	output channel	sequence of layers	stride
224^2 x 3	conv2d	-	32	1	2
112^2 x 32	bottleneck	1	16	1	1
112^2 x 16	bottleneck	6	24	2	2
56^2 x 24	bottleneck	6	32	3	2
28^2 x 32	bottleneck	6	64	4	2
14^2 x 64	bottleneck	6	96	3	1
14^2 x 96	bottleneck	6	160	3	2
7^2 x 160	bottleneck	6	320	1	1
7^2 x 320	conv2d 1x1	-	1280	1	1
7^2 x 1280	avgpool 7x7	-	-	1	-
1 x 1 x 1280	conv2d 1x1	-	k	-	-

The idea is to make the model wider rather than deeper by having parallel layers by applying more than one filter of different sizes to the same level. Overfitting of the data can be avoided by using this technique. The InceptionV1 model is composed of a block of 4 parallel layers:

1. 1×1 convolution
2. 3×3 convolution
3. 5×5 convolution
4. 3×3 max pooling

A problem with this approach is that using 5×5 convolution is pretty expensive; it requires high computational power and, thus, high runtime. To overcome this problem, the developers added a 1×1 convolutional layer above [SLJ+15].

InceptionV3 model was released in 2015 as an advanced version of InceptionV1 model. It consists of 42 layers and has a lower error rate compared to the InceptionV1 and InceptionV2 models, which makes it an optimized version of these predecessors. The major enhancements for this model, as per the authors of the work, are:

1. **Factorization into Smaller Convolutions**: The key factor of the InceptionV1 model is size reduction; InceptionV3 strengthens this by factorizing the large convolutions into smaller ones. Thus, the 5×5 convolution that was used in InceptionV1, as mentioned before, is replaced by two 3×3 convolutional layers. This reduces parameters, and therefore, the computational cost is reduced [SVI+16].

2. **Spatial Factorization into Asymmetric Convolutions**: Further reduction in the convolutions was proposed by the authors by asymmetric convolutions to make the model more efficient. (*Asymmetric convolutions* are of form $n \times 1$). Thus, the 3×3 convolutions is replaced by 1×3 convolution followed by 3×1 convolution. The result of this step is that the network is cheaper, and thus the computational cost [SVI+16].

   Figure 3.6 shows the inception module after applying the two enhancement techniques. This shows the 4 parallel layers and the filter concatenation.

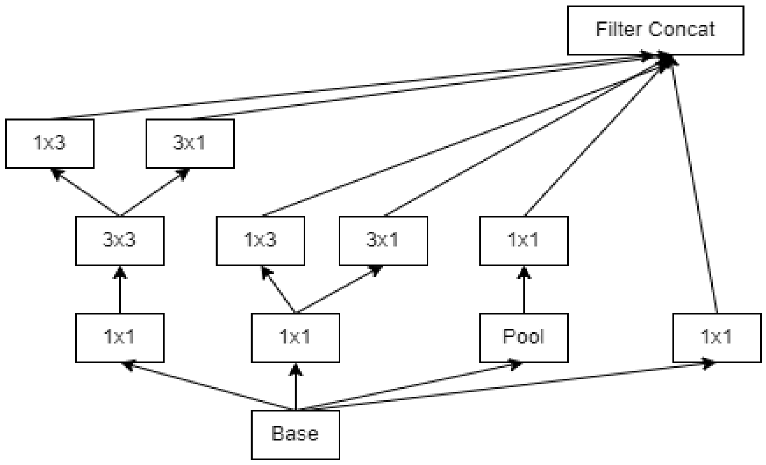


Figure 3.6: Module 3: Inception modules with expanded filter bank outputs. Figure taken from [SVI+16].

There are two additional features, i.e., *Auxiliary Classifiers* and *Grid Size Reduction*, that we do not further discuss here. In summary, the InceptionV3 model is made up of 42 layers, which makes it higher than that compared to InceptionV1 and InceptionV2. The efficiency is increased compared to both. Table 3.2 shows the complete architecture of this model [SVI+16], where the output size of each module is the input size of the next one.

Table 3.2: Outline of the InceptionV3 network architecture

type	patch size /stride	input size
conv	$3 \times 3/2$	299x299x3
conv	$3 \times 3/1$	149x149x32
conv padded	$3 \times 3/1$	147x147x32
pool	$3 \times 3/2$	147x147x64
conv	$3 \times 3/1$	73x73x64
conv	$3 \times 3/2$	71x71x80
conv	$3 \times 3/1$	35x35x192
3xInception	Module 1	$35 \times 35 \times 288$
5xInception	Module 2	$17 \times 17 \times 768$
2xInception	Module 3	$8 \times 8 \times 1280$
pool	8×8	8x8x2048
linear	logits	$1 \times 1 \times 2048$
softmax	classifier	$1 \times 1 \times 1000$

Outlook: NasNet

Last, we present an architecture for *Neural Architecture Search* that we did not use for image classification but is used for finding the right architecture. *Neural Architecture Search Network* (NasNet) was introduced again by a Google team in 2017. The main objective of NasNet is a gradient-based method for finding the best CNN architecture by considering it as a reinforcement learning problem. In reinforcement learning, a reward is given to the system whenever an action is done, achieving an acceptable answer or data. The main idea of NasNet is searching for the best combination of everything, including the number of layers, filter size, output channels , etc. In this context, on a given dataset, the accuracy of the searched architecture is considered the reward after the search action is performed.

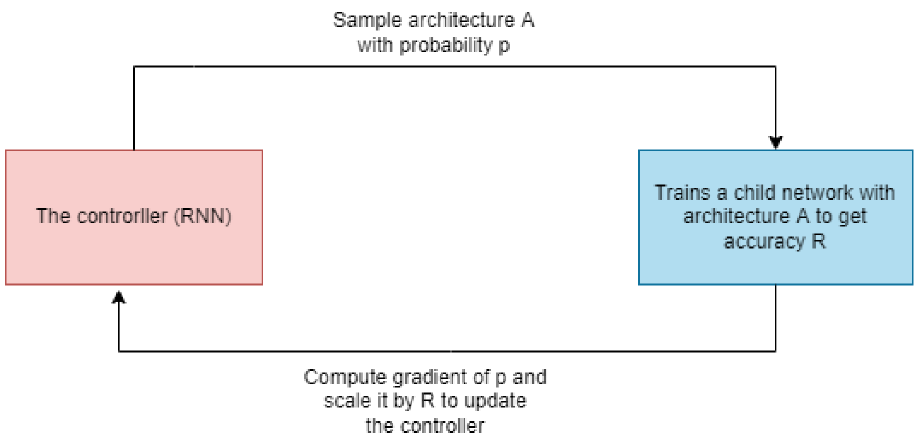

Sample architecture A with probability p
The controller (RNN)
Trains a child network with architecture A to get accuracy R
Compute gradient of p and scale it by R to update the controller

Figure 3.7: An overview of Neural Architecture Search. Figure taken from [ZL16].

As shown in Figure 3.7, NasNet consists of two main components, which are the *controller* and the *child network*. The objective of the controller is to use a recurrent network to generate a string (variable-length string) that specifies the structure and connectivity of a neural network. The child network trains the network on the data and results in an accuracy, which is considered as *reward*. Then, the gradient is computed based on this reward, and the controller is updated. In the following iteration, the architectures with higher accuracies will be granted higher probabilities by the controller. Thus, we can say that the controller learns to improve its search over time [ZL16].

Summary

In this section we introduced the architectures ResNet, MobilenetV2, and InceptionV3 that we used for second-order optimization in Chapter 7, in particular for Subsection 7.2.5. These network architectures are tweaked with stacked blocks and skip-connection layers, in order to address the problem of vanishing gradients, computational efficiency and high accuracy.

We also briefly outlook into the NasNet architecture, which uses reinforcement learning to find the best architecture regarding depth, width and layer structures.

3.4.3 Natural language processing architectures

Natural language processing (NLP) is another popular application of machine learning besides Computer vision. NLP has gained popularity in recent years with Google Translation, the recommendation system for movies or online shops, social media sentiment analysis, advertising, and, most recently, generative pre-trained transformer (GPT) and its variants. NLP uses mathematical approaches to understand and manipulate human language. Among these approaches, deep learning (or neural network) approaches have already shown the dominating performance over traditional computer linguistic approaches across many different NLP tasks in the past decade.

In the following sections, we will introduce the basics of NLP (e.g., Word embedding) and then the most popular neural networks in machine translation used in this thesis, the transformer architecture.

Word embedding

In word embedding, single words are presented with a vector in high-dimensional space, i.e., sentences are point clouds with positions. [JM09] Each word is mapped to a fixed embedding; its real-valued vector encodes the word's semantic meaning. The underlying idea is to capture semantic relationships between words, i.e., "close" words in vector space possess a similar meaning. For the purpose of illustration, we can define arbitrary example mappings for the four words, "mom", "dad" , "happy" and "sad": "mom" to $(0.2, 0.2, 0.4)$, "dad" to $(0.2, 0.1, 0.4)$, "happy" to $(-0.4, -0.5, -0.2)$ and "sad" to $(-0.4, -0.3, -0.3)$. The specific values in the vectors are not standardized and can be created using various techniques for word embeddings. From an embedding space, a mathematical function like cosine similarity can be used to show the vectors' distance and distinguish the clusters. In grammar and semantics "mom", "dad" are close nouns. "happy" and "sad" are adjectives, and in semantic text analysis they are the critical keyword to present states or customers' feedback. In machine translation, the task is often to predict the next word based on the previous words in the sentence. For example, given a predicting sentence, "she is my ...," It is intuitive to find the "..." around the noun and person cluster instead of the adjective cluster.

There are two classes of word embedding methods: sparse and dense. Sparse vector models have high dimensionalities from $20,000$ to $50,000$. Most of the elements in the vector are zeros, and standard methods are term frequency-inverse document frequency(**tf-idf**) [Ram99] and positive pointwise mutual information (**PPMI**) [NN94]. On the contrary, dense vector models have much lower dimensionalities of 50 to $1,000$, with most elements being non-zero. Popular and open-source dense word embedding models are word2vec from Google [MCCD13], GloVe from Stanford University [PSM14], and fastText from Facebook [BGJM17], which are fine-trained on much text data. On a different note, these methods can also be used for the concept of word analogies. For example, reflecting a gender relationship analogy, "man : woman : son : daughter". A machine can do reasoning like a human to understand "a man is to a woman as a son is to a daughter." is a classic example of recognizing patterns and relationships within language. Some of these embeddings are so powerful that the vector representation of the daughter is very similar to that of $v(son) - v(man) + v(woman)$.

Dense word embeddings are more popularly used in practice than sparse ones since short dimensionality may be easier to use for features in deep learning. The dense vector generalizes better than sparse ones, storing explicit counts. In both, sparse or dense, word embeddings, word and document similarities are computed by some dot product function between vectors. In practice, word embedding is often not built from scratch, but existing ones are heavily used, which large NLP companies have already crunched and optimized.

In Subsection 7.2.6, we use the pre-trained **fasttext** word embedding in our model. It has also proven highly useful in text classification, semantic analysis, and neural machine translation (NMT) [QSF+18]. This topic has been part of Yi-Han Hsieh's Master Thesis advised in the course of this PhD thesis. [Hsi21]

Transformer

The transformer is the primary model in our NLP implementation. The model comes from the most influential paper on NLP, written by a Google group in 2017, called *Attention is All you Need* [VSP+17]. The transformer made an enormous success in machine translation and is now the backbone of the generative pre-trained transformer models (GPT). A transformer is a sequence-to-sequence (Seq2Seq) model and entirely relies on the self-attention mechanism without any complex recurrent or convolutional neural networks, as, for example, recurrent neural networks (RNN) or LSTM [HS97]. Seq2Seq models have mainly two components, i.e.,

Encoder and **Decoder**. In between, a vector called **context vector** connects each other. The encoder and the decoder for basic models have traditionally been RNN-based networks such as LSTM, or GRU. In more advanced models, like Transformer and BERT [DCLT18], Encoder and Decoder optimizations include self-attention and feed-forward neural Networks, and there are no recurrent structures.

Figure 3.8 shows the famous overview transformer's architecture. Like most other competitive models, a transformer architecture consists of an encoder-decoder structure. The interesting part, and what makes it superior, is inside the encoder and the decoder. The transformer computes a representation of a given sequence by paying attention to different positions of the input sequence. Due to the properties of the multi-head self-attention mechanism, in transformer models *parallelism* can be employed for efficient training.

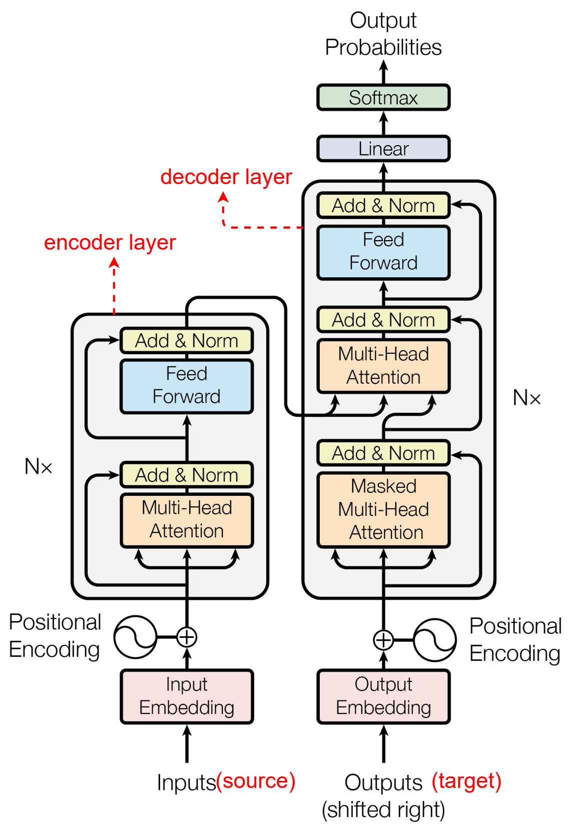


Figure 3.8: The Transformer - model architecture. Figure adapted from [VSP^{+}17].

Next, we sequentially explain the concepts behind the ingredients of the transformer model: 1. the input and outputs, 2. the positional encoding, 3. the encoder/decoder structure, and 4. the attention mechanism with the multi-head self-attention.

1. Inputs and Outputs: Before handling the complex structure inside, let us first state the inputs and outputs of the encoder and the decoder. The raw input of the encoder is a sentence with words that is then pre-processed with a word embedding. Pre-processing involves converting the source and target sequence(or sentence) into the vector presentation with positional encoding and a word-embedding layer. We have explained word embedding in Subsection 3.4.3 and cover positional encoding in a later paragraph.

Since a transformer only uses the self-attention mechanism, it finds the context by itself, and we do not need to specify a context vector, as is required in many other NLP models.

We have used the transformer for a Portuguese-to-English translation. The source is the raw input of the encoder, and the target is the raw input of the decoder. For training, we have source and target available; for testing, the goal is to find the target by itself. For example, in the Portuguese-to-English translation, the source is the tokenized Portuguese sentence, and the target is the tokenized English sentence.

2. Encoder & Decoder: The **encoder** consists of stacked identical **encoder layers**. Each encoder unit has two sub-layers. The first sub-layer is a multi-head self-attention layer. The second sub-layer is a **Position-wise Feed-Forward Networks (FFN)** consisting of two fully-connected layers with a ReLU activation in between.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Here, FFN adds nonlinearities from the self-attention; without these activations, it would just be a weighted average. In addition, since each vector of the later mentioned multi-head attention mechanism is independent of each other, parallelization can be applied here, which is an excellent advantage over the recurrent structures in other Seq2Seq models.

Moreover, a residual connection is added around each of the two sub-layers, followed by layer normalization. So, the output of each sub-layer is $LayerNorm(x + SelfAttention(x))$ or $LayerNorm(x + FFN(x))$. These additional tricks make the loss landscape smoother for more accessible training.

The **decoder** also consists of stacked identical **decoder layers**. Each sub-layer also uses a residual connection and a layer normalization. Besides, the decoder unit has an additional sub-layer between the first and second sub-layers. This additional sub-layer takes both the encoder's output and the first sub-layer's output. Slightly different from the first sub-layer encoder unit, a **masking** is added inside the scaled dot-product attention. See paragraph 4. multi-head attention. The idea of masking ensures that the decoder *does not peek into the future* and only relies on the known outputs at previous positions when predicting a sequence. So, it masks out (setting to $-\infty$) all sequences after position i when predicting position i.

The number of encoder and decoder layers, FFN's inner dimensionality, and the number of heads are the main hyper-parameters in the transformer model. We have denoted them as `num_layers`, `dff`, and `num_heads` in our implementation. `d_model` is determined by the word embedding layer. Check the Appendix B for the hyperparameters JSON file and other implementation details.

3. Positional Encoding: Since, for reasons of performance, there is no recurrence and no convolution in the self-attention mechanism, we also lack built-in position information. Therefore, it is required to encode the position information of the sentence before it goes into the self-attention layer, see also Figure 3.9.

Firstly, the source and the target sentences are tokenized and converted into the vectors in the d_{model} space through the corresponding embedding layers. Then, we sum up a unique positional vector and an embedding vector since both dimensions are the same (d_{model}). Finally, tokens will be closer in the d_{model} space based on the similarity of meaning and their position in the sentence. This process is shown in the Figure 3.9.

Besides, Figure 3.10 shows the visualization of final vectors. The function which produces the positional vector can be hand-crafted or learned from data. We follow the suggested word-embedding and language task as in [tut]. We use the functions that have also been used in the

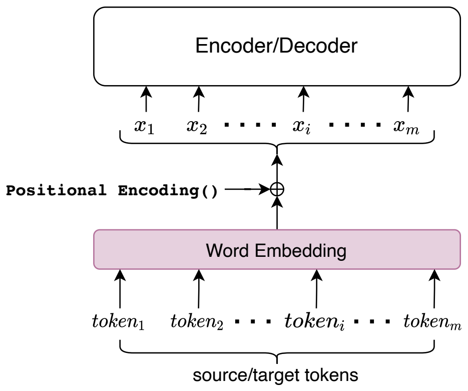


Figure 3.9: Pre-processing of words (tokens) with positional encoding and word embedding [Hsi21].

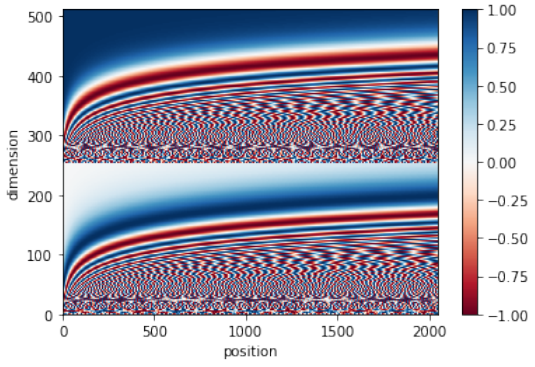


Figure 3.10: Tokens of the positional encoding in $d_{model} = 512$ space and length of sentence $= 2048$. The x axis is the position index in the sentence and the y axis is the dimension index in $d_{model} = 512$ space. Figure generated using [tut].

original paper [VSP+17],

$$PE_{pos,2i} = sin\left(pos/1000^{2i/d_{model}}\right), \text{and}$$

$$PE_{pos,2i+1} = cos\left(pos/1000^{2i/d_{model}}\right) \quad ,$$

where *pos* is the position and i is the dimension of the word embedding. This is displayed in Figure 3.10, generated with the corresponding Tensorflow Keras tutorial [tut].

4. Multi-head self-attention: The foundation behind this layer is the attention mechanism that in different variants, has appeared in other architectures before (Additive attention: [BCB16], [LPM15], Self-attention: [CDL16]). In the transformer architechture [VSP+17], this multi-head self-attention mechanism is incorporated in the so-called **Scaled Dot-Product Attention** term, and it depends on query vectors (stacked in a matrix $\boldsymbol{Q}$), key vectors ($\boldsymbol{K}$)) and query vectors $\boldsymbol{Q}$). A **Self-Attention Layer** relies entirely on its input sequence (unlike additive attention), and it uses the self-attention mechanism to calculate all the hidden states' attention, i.e., the alignment score w.r.t each other. We call the single-head attention function $Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$.

The motivation for Multi-head attention is that we want to look at multiple places in the sentence at once in parallel. So, the multi-head mechanism runs through the scaled dot-product attention multiple times in parallel rather than only computing the attention once.

$$\begin{aligned} MultiHead(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= Concat(head_1, ..., head_h)\boldsymbol{W}_O \\ \text{where } head_i &= Attention(\boldsymbol{Q}\boldsymbol{W}_{Q,i}, \boldsymbol{K}\boldsymbol{W}_{K,i}, \boldsymbol{V}\boldsymbol{W}_{V,i}) \end{aligned} \tag{3.2}$$

where h is is the number of attention heads, i ranges from 1 to h and $d_k = d_v = d_{model}/h$. $\boldsymbol{W}_{Q,i} \in \mathbb{R}^{d_{model} \times d_k}$, $\boldsymbol{W}_{K,i} \in \mathbb{R}^{d_{model} \times d_k}$, $\boldsymbol{W}_{V,i} \in \mathbb{R}^{d_{model} \times d_v}$, and $\boldsymbol{W}_{O,i} \in \mathbb{R}^{hd_v \times d_v}$ are parameter matrices to be learned. In the model, h is also one of the hyperparameters.

Since the heads reduce the dimensions of parameter matrices, the total computational cost is similar to that of single-head attention with full dimensionality [VSP+17].

Summary

A very popular branch of deep learning is natural language processing. The transformer architecture is superior to many other models, including a sequence-to-sequence model with an encoder/decoder structure. A *big advantage* is the multi-head attention mechanism that allows simultaneous focus on more parts of the sequence, allowing parallelism and a high range of capturing information.

In Subsection 7.2.6, we applied second-order optimization on the transformer architecture for a translation task.

3.4.4 Transfer learning

Recent progress in architectures and algorithms, including innovations like residual blocks and attention layers and the exponentially increasing amount of available data, clearly strongly supports the black-box deep learning-based approach to hand-crafted expert systems, especially in the field of image classification or natural language processing. Given the *feature hierarchy* in neural architectures of for word embeddings, i.e. the gained *knowledge*, a new use case opens up: transfer learning.

Transfer learning (TL) deals with applying already gained knowledge for generalization to a different but related domain [YCBL14, Li]. Generating an adequately sized and well-annotated dataset tailored to a particular task in the domains of image classification or natural language processing is a resource-intensive, expensive, and time-consuming process. Consequently, developers in academia and smaller companies often work with significantly smaller training and testing data sets compared to the renowned datasets, such as CIFAR and ImageNet, or the WebText for GPT2 [KH+09]. Furthermore, the training process is time-consuming and relies heavily on specialized, high-performance hardware, particularly in terms of memory. Since modern ResNets for image classification require around 2-3 weeks to train on ImageNet (depending on the power of the computational resources), re-starting this process from scratch for each single slightly modified task is hardly efficient. Transformer architectures consume even more time and require even more resources. The concept of transfer learning involves utilizing the knowledge acquired by a pre-trained model to enhance its performance in the specific task we are focused on. This allows for reducing the cost of pre-processing and augmenting a new dataset, and reduces training time and deploying the model. [YCBL14]

We will now explore the *two* most essential techniques for implementing transfer learning, for example, in image classification. The model highly profits if the model is pre-trained on a large dataset that has many classes and generalizes well to many tasks, e.g. ImageNet.

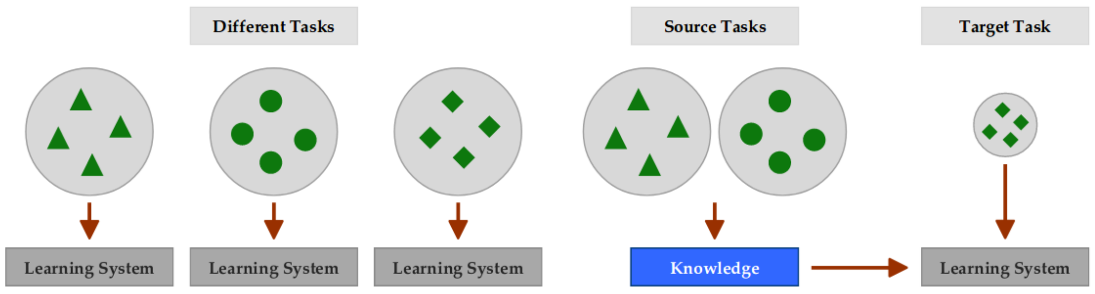


Figure 3.11: Visualization of the transfer learning approach. Left: classical, right: transfer learning. Figure based on [PY09].

Fixed feature extractor

In almost all our models, the final layer is a fully-connected layer that produces an output vector representing the scores of the classes (for ImageNet 1000 and for NLP, this depends on model hyperparameters). A fixed feature extractor has been created by eliminating this layer while keeping the rest of the hidden layers unchanged. [Li, SRASC14] Maintaining the model's feature hierarchy allows us to subsequently utilize this extractor on the training set for our specific task of interest. This leads to a multidimensional vector that holds knowledge in some sort of latent space, i.e., the values of the activation functions of the network's final hidden layer for each input image. Following the iteration through all training examples and applying a ReLU activation function to the resulting CNN codes, we train a fresh linear classifier, like softmax or SVM. This classifier takes the place of the previously removed output layer. In summary, the fixed feature extractor method enables us to utilize the pre-trained model's weights to extract features ready-to-use. These features are then employed to train a classifier tailored to our specific task of interest.

Fine-Tuning

Fine-tuning enables the transfer of our new model to a different domain by utilizing the pre-trained weights and biases as the *initialization* for the backpropagation process. The advantage of this technique is that it selectively updates the parameters where it needs to. With this, it adjusts the model's feature hierarchy and makes it capable of generalizing to the new task. Typically, we want to keep the generic, low-level features while adapting and updating the higher-level features to suit the context of the new domain. However, while a higher prediction accuracy can be observed with fine-tuning, one must treat the result with caution: the effectiveness depends on the size and similarity of the new training data to the initial dataset. Due to the high risk of overfitting, this technique is error-prone, especially for small datasets. [Li]

Benefits

As with any other task in machine learning, it is hard to put a set of defined rules that are applicable. However, we can list some points that help us decide when to use transfer learning. Similar to many tasks in machine learning, it is challenging to define a rigid suggestion when TL should be used. However, we list a few guidelines that assist in determining when to employ transfer learning.

1. Infeasible to train from scratch, as there is no access to or not enough training data available.
2. There already exists a universal model that has been optimized on a huge amount of data from a comparable task.
3. When we have two similar tasks that have the same kind of input.

Transfer learning can enhance deep learning in three key aspects. First, the initial accuracy in the current task can be quickly reached by re-using knowledge transferred from a previous task without additional learning. Second, the convergence speed of learning the new task can be accelerated using knowledge from the prior model, in contrast to training from scratch. Third, TL may benefit from the transferred knowledge as opposed to no knowledge from scratch. [TS10]

We can also discuss the benefits of transfer learning with Figure 3.12 from [TS10].

- **Higher start:** The initial performance of the model is higher than by not employing transfer learning.
- **Higher slope:** Regarding the convergence speed of the model, accuracy improves quicker than training from scratch.
- **Higher asymptote:** The final accuracy of the transfer learning model is much higher than without transfer learning.

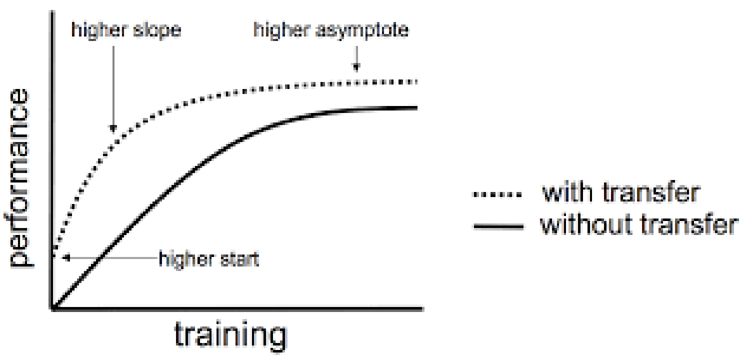


Figure 3.12: Three ways in which transfer might improve learning Figure taken from [TS10].

TL allows us to create complex models for specific tasks with limited amounts of labeled training data by re-using knowledge from a pre-trained model. Since this strategy is notably robust, it is considered to be a driving factor in the field of deep learning [Ng16]. We presented two strategies, a fixed feature extractor and fine-tuning, that have been reported to perform well on many different visual tasks [Li,SRASC14,YCBL14]. Since sharing research contributions and full digital models is a common practice in DL, a wide variety of pre-trained models are available, which can be utilized for *transfer learning*. However, it should be noted that the underlying model must be suitable for the new task of interest when applying TL.

3.4.5 Summary

In this section, we introduced the intricate neural network architectures on which we will apply second-order optimization in Chapter 7.

We started with shallow networks, which boil down to a linear regression problem, or with convolution nets, occurring in hand-crafted U-Net-type architectures (Subsection 3.1.1). For computer vision, the family of Residual nets(ResNets) and derived models have been most popular. ResNets have skip-connections in the residual block, and MobileNets have improved computational efficiency and inception nets. We explained these architectures, for which we will run second-order optimization in the results chapter, in Subsection 3.4.2. Lastly, we covered the transformer model for natural language processing, which we used for second-order optimization of a Portuguese-English translation task in Subsection 7.2.6. We explained the details of the transformer architecture, which is also the backbone of most translation tasks, and the generative pre-trained transformer. In Subsection 3.4.3, we explain the word embedding and the encoder/decoder structure, with the input/output sequences, the positional encoding, and the attention mechanism.

In the next Subsection 3.4.4, we explain the field of *transfer learning.* With a fixed feature extractor or fine-tuning, knowledge from existing models can be re-used, and a new target task can be addressed more efficiently.

In summary, the focus of this section is explaining the details of the deep network architectures that we used for second-order optimization in Chapter 7.

3.5 Summary

This theory Chapter 3 called artificial neural networks lays out the foundation of neural networks for the results Chapter 7.

We started this chapter in Section 3.1 with the basics of artificial neural networks from a scientific computing context. We showed how deep learning models are formed with the multilayer perceptron model.

We moved on in Section 3.2 to define the different layers in a convolutional neural network. Besides the convolution layer, poling and drop-out often occur, as well as fully-connected layers.

A grand challenge in machine learning is to decide on model quality and to estimate the uncertainty, especially for safety-critical fields. After explaining the artificial neural networks, in Section 3.3 we described the types of uncertainty, aleatoric and epistemic, and variational inference. In an approach called Bayes-by-Backprop, a probability distribution is placed around the network weights. With such a trained model, we can treat the prediction uncertainty in what we refer to as Bayesian neural networks.

In the next Section 3.4, we explained the intricate architecture design used for data analysis, computer vision, and natural language processing. We start with shallow networks for data analysis, such as linear regression and convolutions, and move towards computer vision. There, we outlined the advantages of *skip connections* in residual networks and showed the details of ResNets, MobileNets, and InceptionNets. As for natural language processing, we explained the transformer architecture with an encoder/decoder design. The attention mechanism was a breakthrough in the field, leading to technologies like the generative pre-trained transformer (GPT). Lastly, we showed that with transfer learning, existing models can be re-used efficiently to address a new target task. This opens a family of derived models to the previous intricate pre-trained architectures.

In summary, in this chapter, we introduced deep learning for the purpose of second-order optimization in this thesis. The state-of-the-art model architectures are becoming very sophisticated, with some allowing uncertainty estimation. The remarkable expansion of available data has led to breakthroughs in image classification with residual nets and natural language processing with the transformer architecture.

In the next Chapter 4, we will cover optimization methods, in particular for neural networks.

A mathematical formula should never be "owned" by anybody! Mathematics belongs to God.

Donald Knuth (*1938)

4 Optimization methods

This chapter, called "Optimization methods," introduces the theory of the second part of the thesis, neural network optimization (Chapter 7). Optimization is nowadays quite virulently used, some referring to the mature field in mathematics of (non-)convex optimization, others referring to parameter search, and others claiming that machine learning is just parameter optimization or statistics. Here, we refer to the theory of deep learning optimizers, which limits the context to suitable methods; this Chapter 4 builds the foundation for our results on a neural network optimization in Chapter 7. We begin with the common optimization approaches for deep learning (DL) in Section 4.1, and describe mainly *stochastic gradient descent* in Subsection 4.1.1 and *Adam* in Subsection 4.1.2. Next, we lay out the theory for the proposed second-order optimizer `Newton-CG` in Section 4.2. We close with a summary (Section 4.3).

Our approach follows the approach of Newton-Krylov methods; a survey for this is in literature comprised of books by Knoll and Keyes [KK04] or by Meister [Mei11], and extended for many applications [WDE07, WN⁺99, Ste83]. It is generally suggested to first use strategies of conventional lower-order iteratively (pre-training), and once the optimization process is sufficiently close to a minimum, address the Newton-type method around a trust region in a ball of convergence.

4.1 State-of-the-art deep learning optimization approaches

Let's recall that the neural network layer consists of a linear combination of the input variables,

$$f(x_i, W) = w_0 + \sum_{i=1}^{N} (w_i^{(d)} x_i) \ .$$

In Deep learning (DL), network layer functions are intricately linked through a chaining process, i.e., $f(x) = f^{(D)}(\dots f^{(2)}(f^{(1)}(x)))$. We then form an error function, e.g., the square loss,

$$L_{sq} = \frac{1}{2} \sum_{i=1}^{N} (f(x_i, W) - y_i)^2 \ .$$

To find the appropriate weights W, we use optimization algorithms to minimize the loss function. For a supervised learning task, the optimization problem consists now of finding all weights W comprised in a weight tensor $\mathbf{W}$ that minimize a given loss function l for given training samples X, Y:

$$\min_{\mathbf{W}} L(X, Y, \mathbf{W}) \ . \tag{4.1}$$

A prominent example of a loss function for binary classification is the categorical cross-entropy

$$L_{entr}(X, Y, \mathbf{W}) := -\sum_{l=1}^{\mathtt{N}} y_i \log(f^{(D)}(X, \mathbf{W})) \ .$$

Note that only the last layer function $f^{(D)}$ of the network directly shows up in the loss, but all layers are indirectly relevant (due to the chaining $f(x) = f^{(D)}(f^{(D-1)}(\dots)$ [RNB23]. Hence, the optimization addresses all weights in all layers.

DL optimization differs slightly from pure optimization, as it measures losses indirectly to some training sets, and it may be intractable. In contrast, pure optimization, where minimizing the function f (or rather a global $F = \sum_{i=0}^{N} f$ as the sum of all samples) directly of the function is the goal itself, in contrast to the chained objective function L. [GBC16a]

4.1.1 Stochastic gradient descent

In order to solve the optimization problem (4.1), different first-order methods exist (for a survey, see, e.g., [GBC16a]). For deep learning methods, the algorithmic choice is limited, as many data chunks need to be processed (memory intensive), and higher order methods are generally prohibitively expensive with the amounts of optimization parameters (memory and compute expensive). Therefore, the first order is the state of practice, despite being low accurate. Because we know from optimization literature that higher-order methods often prevail and are state-of-practice in other fields, we later propose and try a lumped second-order scheme.

The pure *gradient descent* without momentum is computing intermediate weights $\mathbf{W}_k$ in iteration k via

$$\mathbf{W}_k = \mathbf{W}_{k-1} - a_{k-1} \nabla L(\mathbf{W}_{k-1}) \ ,$$

where $\nabla L(\mathbf{W}_{k-1})$ denotes the gradient of the total loss function L w.r.t. the weights $\mathbf{W}$ and a_{k-1} is the so-called learning-rate.

Computing this gradient globally (w.r.t. all data points) is very expensive because it requires differentiating the model for every sample in the entire dataset, requiring memory access to the whole dataset and all intermediate layers.

Stochasticity

DL Optimizers look at stochastic *mini batches* of data, i.e. disjoint collections of data points. The union of all mini batches will represent the whole training data set. The reason for considering data in chunks of mini-batches and not in total is that the back-propagation in larger neural networks will face severe issues w.r.t. memory. Hence, the mini-batch loss function for binary classification, where the mini-batch is varied in each optimization step in a round-robin manner, is now defined by

$$L(x, y, \mathbf{W}) := -\sum_{i=1}^{\text{batch-size}} y_i \log(f^{(D)}(X, \mathbf{W})) \ .$$

Note that the *stochastic gradient descent* (SGD) includes stochasticity by changing the loss function to the input of a specific mini-batch of data, i.e. using parts of the global sum, which may result in different behavior. Each mini-batch of data provides a noisy estimator of the average gradient over all data points, hence the term stochastic. Technically, this is realized by switching the mini-batches in a round-robin manner to reach the full dataset (one full sweep is called an epoch; frequently, more than one epoch of iterations is necessary to achieve quality in the optimization) [RNB23].

Another motivation for using stochasticity in DL stems from the observation that optimizers often get stuck in a local minimum or in saddle points. Stochasticity helps in escaping from local minima or saddle points. Multicore architectures are usually underutilized, and hence, in a data-parallel approach, one can run different mini-batches embarrassingly parallel.

4.1.2 Adaptive moments

The family of *Adaptive moments* (Adam) methods updates weight values by moving averages of the gradient s_k with estimates of the 1^{st} moment (the mean) and the 2^{nd} raw moment (the uncentered variance).

The approach called *AdaGrad* performs well for some but not all deep learning models. It directly uses these estimators:

$$W_{k+1} = W_k - \alpha_k \frac{s_k}{\delta + \sqrt{r_k}} \ , \tag{4.2}$$

where s_k is the gradient g, $r_k = r_{k-1} + g \odot g$ and δ is a small constant.

The *Adam* method corrects for the biases in the estimators by using the estimators $\hat{s_k} = \frac{s_k}{1-\beta_1^k}$ and $\hat{r_k} = \frac{r_k}{1-\beta_2^k}$ instead of s_k and r_k. Good default settings for the tested machine learning problems described in this paper are $a_0 = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\delta = 10^{-8}$.

4.1.3 Other strategies for neural network optimization

Several other gradient-based optimization algorithms exist, prominent in DL. RMSprob falls into the class of adaptive learning rates and generalizes AdaGrad for non-convex settings by changing the gradient accumulation into an exponentially weighted moving average. The Nesterov's Accelerated Gradient Method is suited for large fluctuations and allows faster convergence for convex problems compared to SGD. [SBC15]

The approach of *Quasi-Newton methods* (with the most prominent as Broyden-Fletcher-Goldfarb-Shanno, BFGS) is to approximate the Hessian inverse with a low-rank update matrix. Like the method of conjugate gradients, the BFGS algorithm iterates with line searches towards the second-order curvature. Unlike conjugate gradients, however, the approach's success is in a greater range around the local minimum. Thus, relative to conjugate gradients, BFGS has the advantage that it can spend less time refining each line search. On the other hand, the BFGS algorithm must store the inverse Hessian matrix ($\mathcal{O}(N^2)$ memory), making BFGS impractical for most modern deep learning models that typically have millions of parameters. The L-BFGS algorithm adapts limited memory, stopping at a sequence of vectors representing the projection. [GBC16a]

A whole other class of optimization algorithms is defined by the Consensus-based optimization (CBO) methods. The idea involves several starting values (in a Monte Carlo fashion) and chooses the best guess with a certain refinement. It is a derivative-free optimization method for minimizing non-convex and non-smooth functions globally. [FKR22] Currently, it is not very popular for DL.

4.1.4 Summary

We have introduced several exciting learning algorithms, SGD, AdaGrad, and Adam, just to name a few important ones for this thesis. They are mostly based on auto-differentiating the system,i.e., they compute the gradient. All of them are facing the challenge of optimizing deep models by adapting the learning rate for each model parameter (a process called hyperparameter

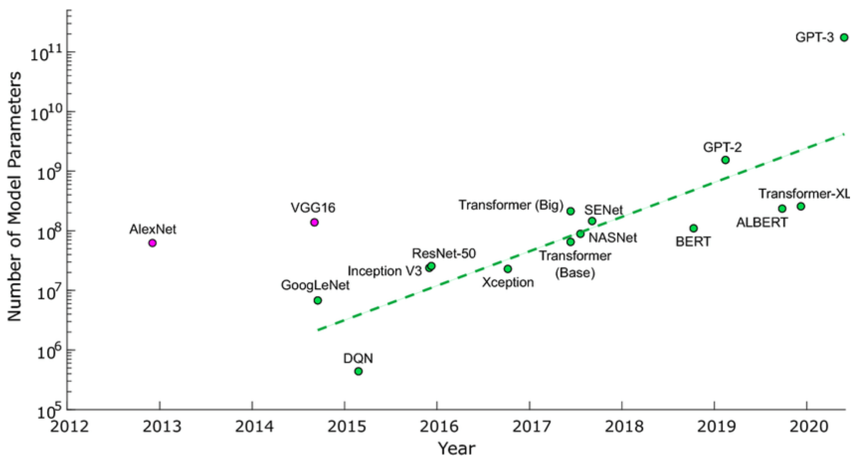


Figure 4.1: Increasing number of model parameters each year for prominent DL architectures. Figure taken from [BSH+21].

tuning). In practice, much compute time is spent finding these hyperparameters. At this point, a natural question poses: choosing the best optimization algorithm for which scenario? While we refer to benchmark studies, e.g. Descending through a crowded valley – benchmarking deep learning optimizers [SSH21], or other benchmarks [DSN+23]; in this thesis, we later introduce yet another training algorithm. Our ultimate goal is to have more intuitive (mathematically-motivated) hyperparameters.

4.2 2^{nd}-order optimization for deep learning

The second-order optimizer implemented and used for the results of this work consists of a Newton scheme with a matrix-free conjugate gradient (CG) solver for the linear systems of equations arising in each Newton step [RNB23]. The effect of the Hessian on a given vector (i.e., matrix-vector multiplication result) is realized via the so-called Pearlmutter approach and avoids setting up the Hessian explicitly.

4.2.1 Pearlmutter approach

In Figure 4.1, the increasing network sizes over the years are plotted. It seems that with more computing power available, the networks (and the tasks) are also growing. The explicit setup of the Hessian is memory-expensive due to the quadratic dependence on the model parameters (weights); e.g., a 16M×16M matrix requires about 1 TB of memory. The Hessian arises as a matrix of size $N \times N$, and like most linear algebra computations, exact direct computation (which are sufficient but never necessary for NN problems) cost $\mathcal{O}(N^3)$ time. [YGKM20]. In order not to be only theoretically useful, a method should be checked against the more-or-less new architectures.

We can obtain "cheap" access to the problem's curvature information by computing the Hessian-vector product [RNB23]. This method is called Fast Exact Multiplication by the Hessian H (see [Pea94], e.g.). Specifically, it computes the Hessian vector product Hs for any s in just **two** (instead of the number of weights N) backpropagations (i.e. automatic differentiations for 1st derivative components).

For our formulation of the problem, it is defined as:

$$H_L(\mathbf{W})s = \begin{pmatrix} \sum_{i=1}^{n} s_i \frac{\delta^2}{\delta w_1 \delta w_i} L(\mathbf{W}) \\ \sum_{i=1}^{n} s_i \frac{\delta^2}{\delta w_2 \delta w_i} L(\mathbf{W}) \\ \vdots \\ \sum_{i=1}^{n} s_i \frac{\delta^2}{\delta w_n \delta w_i} L(\mathbf{W}) \end{pmatrix} = \begin{pmatrix} \frac{\delta}{\delta w_1} \sum_{i=1}^{n} s_i \frac{\delta}{\delta w_i} L(\mathbf{W}) \\ \frac{\delta}{\delta w_2} \sum_{i=1}^{n} s_i \frac{\delta}{\delta w_i} L(\mathbf{W}) \\ \vdots \\ \frac{\delta}{\delta w_n} \sum_{i=1}^{n} s_i \frac{\delta}{\delta w_i} L(\mathbf{W}) \end{pmatrix} = \nabla_w (\nabla_w L(\mathbf{W}) \cdot s)$$

We apologize for the slight abuse of notation. H is the Hessian matrix (or H_L with respect to the loss function L), the $\mathcal{H}$ symbol refers to the Hierarchical matrix from the chapter before. The Pearlmutter approach in algorithmic form is shown in Algorithm 4.3.

Algorithm 4.3 Pearlmutter

```
Require: $X, Y, \mathbf{W}, s$:   Compute $H_L s = \nabla_w(\nabla_w L(\mathbf{W}) \cdot s)$
Require: $W_0$:   Initial estimate for $\mathbf{W}$.
1: $g_0 \leftarrow$ gradient($L(\mathbf{W})$)                          ▷ Back-Prop
2: intermediate $\leftarrow$ matmul($g_0, s$)                ▷ Matrix-Multiplication
3: $Hs \leftarrow$ gradient(intermediate)                         ▷ Back-Prop
4: return $Hs$
```

The resulting formula is both efficient and numerically stable [Pea94]. In the implementation, we denote this building block as **Pearlmutter**. By the multiplication with a vector, we get the directional second derivative of the function. In mathematical terms, this falls under the Gateaux derivative, which is a generalization of the concept of directional derivatives in differential calculus. For differentiable functions, it also coincides with the Fréchet-derivative, for which there exists a nice theory in literature.

4.2.2 Newton's method

Recall the Newton-Raphson equation

$$H_L(\mathbf{W}^k)d^k = -\nabla L(\mathbf{W}^k) \tag{4.3}$$

for the network loss function $L : \mathbb{R}^n \to \mathbb{R}$, where $\mathbf{W}$ is the vector of network weights and $\mathbf{W}^k$ the current iterate of Newton's method to solve for the update vector d^k. The size of the Hessian is $\mathbb{R}^{n \times n}$, which becomes infeasible to store with state-of-the-art weight parameter ranges of ResNets (or similar).

The weighted update step is thus,

$$\mathbf{W}_k = \mathbf{W}_{k-1} - \alpha_{k-1} d_k \ .$$

We define the regularized Newton equation later in Equation 4.4], and the algorithm is outlined later in the implementation chapter in Algorithm 7.5.

4.2.3 Ingredients of proposed 2^{nd}-order optimizer: Newton-CG

This section introduces a few methods we need for our `Newton-CG` algorithm. We list the ingredients here as we need them; they do not follow a hierarchy from a field, but rather, they explain examples of different fields.

Conjugate gradient step

Since we have a matrix-vector product available without setting up the full matrix (with **Pearlmutter**), we employ an iterative solver scheme that requires matrix products only. We therefore employ a few (inaccurate) CG-iterations to solve Newton's regularized equation (4.4), thus resulting in an approximated Newton method. The standard CG-algorithm is e.g. described in [S+94]; in particular, no direct matrix access is required since the algorithm relies only on products with vectors.

Algorithm 4.4 Conjugate Gradients

```
Require: $A$, $b$:                                   ▷ We want to solve $Ax = b$ for $x$.
Require: $x_0$:                                      ▷ Initial estimate for $x$.
1: $r_0 \leftarrow Ax_0 - b$, $p_0 \leftarrow -r_0$, $k \leftarrow 0$
2: while $r_k$ too large do
3:     $\alpha_k \leftarrow \frac{r_k^\top r_k}{p_k^\top A p_k}$          ▷ Compute step size
4:     $x_{k+1} \leftarrow \alpha_k p_k$                 ▷ Apply step
5:     $r_{k+1} \leftarrow r_k + \alpha_k A p_k$          ▷ Compute new residual
6:     $\beta_{k+1} \leftarrow \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$   ▷ Factor, such that $p_{k-1}$ and $p_k$ are conjugate w.r.t. A
7:     $p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k$    ▷ Compute new step direction
8:     $k \leftarrow k + 1$
9: end while
```

Tikhonov Regularization

Recall the Newton equation

$$H_L(\mathbf{W}_k)d_k = -\nabla L(\mathbf{W}_k)$$

for a neural network loss function $L : \mathbb{R}^n \to \mathbb{R}$, where $\mathbf{W}$ is the vector of network weights and $\mathbf{W}_k$ the current iterate. Due to the large number of weights in practical neural networks, the Hessian may be ill-conditioned, i.e.

$$\kappa(H_L) := \frac{\lambda_{max}}{\lambda_{min}} \gg 1$$

with $\lambda_{min/max}$ being the minimal/maximal eigenvalue of H_L. In a geometric view, this behavior resembles starting values, where the function leads to a very far point due to a high/low slope (saddle point).

Since the Hessian frequently has a high condition number, which implies near-singularity and provokes imprecision, a general ansatz is to apply regularization techniques to counteract a bad condition. A common choice is Tikhonov regularization. To this end, a multiple of the unit matrix is added to the Hessian of the loss function such that the regularized system is given by

$$(H(\mathbf{W}_k) + \tau I)d_k = H(\mathbf{W}_k)d_k + \tau I d_k = -\nabla L(\mathbf{W}_k) \tag{4.4}$$

Note that for large τ the solution will converge to a fraction of the negative gradient $-\nabla L(\mathbf{W}^k)$, similar to a stochastic gradient descent method.

Armijo step size restriction

Line-search methods offer an intuitive approach to find suitable step lengths. For stability purposes, it helps to introduce a step size restriction. The Armijo condition reads

$$L(\mathbf{W}_{k+1}) - L(\mathbf{W}_k) \leq \nabla L\left(\mathbf{W}_k\right) p_k \ .$$

If fulfilled, it falls back to the SGD update step, the negative gradient $-\nabla L\left(\mathbf{W}_k\right)$. Note that $\gamma \to 1$ leads to a greedy step selection, but it may cause stagnation.

Line-search methods offer an intuitive approach to finding suitable step lengths.

It can be shown that the Armijo step size exists for every descent direction. A proof can be found in [UU12] alongside further step size rules. The different building bricks can then be combined in any optimization algorithm. [Suk20]

Stochasticity and dampening

As autodifferentiation with all samples is infeasible in DL (w.r.t. memory), we also must relax the proposed 2^{nd}-order optimizer with this constraint. There are voices that argue that with stochasticity, the advantages of a 2^{nd}-order optimizer are decreased, as for each batch, a different surface is optimized. While this is a worthwhile consideration, mini-batching (stochasticity) was necessary for our implementation of large architectures and data sets.

The requirements additionally include a learning-rate size parameter. While due to stochasticity, the loss surfaces change significantly for different batches, adding a dampening to the Newton scheme significantly improved convergence behavior. Note that state-of-the-art optimizers include a schedule with changing learning rates. For `Newton-CG` we also require the learn-rate scheduler to be at fair comparison.

Complexity

The method described above requires $O(bn)$ operations for the evaluation of the gradient, where n is the number of network weights and b is the size of the mini-batch. The Pearlmutter trick requires two back-propagations instead of one for SGD/Adam. In addition, for the solution of the Newton-like equation $O(2mbn)$ is needed, where m is the number of iterations conducted by the `CG` solver until a sufficient approximation to the solution is reached. Although the second-order optimizer requires more work than ordinary gradient descent, it may still be beneficial since, under the conditions that it promises local superlinear (at best quadratic) convergence, i.e. $\exists \gamma \in (0,1), l \geq 0$, such that

$$||\mathbf{W}^{k+1} - \mathbf{W}^{*}|| \leq \gamma ||\mathbf{W}^{k} - \mathbf{W}^{*}|| \quad \forall k > l$$

where $\mathbf{W}^{\star}$ is a local minimum (see [Suk20, RNB23]).

4.2.4 Related work on 2^{nd}-order optimization for deep learning

Optimization literature is vast, and this includes many variants for convex or non-convex optimization problems. Newton-CG is not new; it has been applied to other optimization problems [WDE07], and the theory is vast [WN+99, Ste83]. Some variants include Newton-GMRES [BM01], which extends the trust regions space using a GMRES solver.

Related work for training supervised learning optimization problems in the last few years is very fast, and it is a challenge to keep track of the newest developments. A thorough review of 0ptimization methods in large-scale machine learning is given in [BCN18]. This led (amongst other reports, conferences, and workshops) to a vivid discussion about next-generation optimizers for large-scale machine learning, including the balance between diminishing noise with stochasticity and incorporating second-order derivative approximations. The most popular and useful method for approximate second-order neural network training is KFAC [MG15]. It avoids setting up the Hessian by using a likelihood function for the Fisher information matrix. An efficient implementation is ChainerKFAC [OTU$^+$19]. To name a few others (no claim for completeness), AdaHessian incorporates second-order information using the Hutchinson method and sets the learning rate according to this Hessian eigenvalue estimate. [YGS$^+$21] Newton-MR is a very interesting study of convexity and inexactness conditions; they use a variant of a Newton-GMRES and especially report Newton-MR's advantage of an increased ball-of-convergence and compare a few methods. [RLXM22] Other work employs the classical BFGS update formula in its limited memory form for neural networks. [BHNS16]

In contrast, the Hessian itself may be indefinite in many cases. A remedy would be classical Gauss-Newton methods, especially for least squares loss functions; there are also several works on this. [BRB17,GZDH20] A complete other approach works on the generalization of optimizers, making theoretic proofs why SGD generalizes better than ADAM. [ZFM$^+$20]

4.3 Summary

We start in Section 4.1 by recalling the optimization problem for deep learning. We explain the state-of-the-art optimizers, in particular SGD and Adam, with many details and perks. We also touch a few other optimizers, but essentially all of them require some non-intuitive optimization parameters (hyperparameters). Hence, in practice, a lot of compute time is spent on naive hyperparameter tuning.

We continue in the next Section 4.2 by explaining theory on 2^{nd}-order optimization for Deep Learning. We begin with the Pearlmutter approach, a scheme to get a fast Hessian matrix-vector multiplication with just two back-propagation steps. We move towards the Newton method, explaining the ingredients of our proposed `Newton-CG` optimizer, including the conjugate gradient method, Tikhonov regularization, Armijo step size restriction, stochasticity, and dampening. The second-order methods promise a similar complexity to state-of-the-art first-order methods, promising (at best) quadratic convergence and, hence, fewer steps. This motivates many benefits with higher computational efficiency regarding memory-boundness, scenarios with sparse data, or parallel execution.

In the last few years, there was a vivid discussion in the field of 2^{nd}-order optimization for deep learning. We review related work in Subsection 4.2.4. At this point, a natural question is: "what is the best optimization algorithm for which scenario?" For this, on the one hand, we refer to benchmark studies of deep learning optimizers, including hyperparameter tuning, e.g. [SSH21, DSN$^+$23]. On the other hand, our approach is to compare later in the implementation and results chapter Chapter 7 the proposed `Newton-CG` algorithm to the state-of-the-art optimizers `SGD` and `Adam` for various scenarios, offering a nice guide for the most suitable optimizer.

PART III

METHODS, IMPLEMENTATION, AND EXPERIMENTAL RESULTS

Citius, altius, fortius – communiter (Latin, in English: faster, higher, stronger – together)

Motto of the Olympic Games

5 Computational setup

This chapter explains the computational resources used for the experiments in this thesis. The main objective is to show that several models can be run on several hierarchies of computer architectures, and even intricate models can be evaluated on edge devices like smartphones.

In this chapter, we use the hierarchy from Figure 5.1 for ordering in this chapter. In history, when there was the wave of computational simulations in the early 2000's, much emphasis on compute infrastructure lay on clusters. Nowadays, with the upcoming Internet of things with edge computing, designing software also for small systems and AI applications is imperative, i.e. from edge device to HPC.[1]. Results for $\mathcal{H}$-matrices and Newton-CG are described later; however, we use this chapter especially for describing our showcasing project TUM-lens, which we also describe beyond computational setup. Hence, it appears longer for the phone application, since we intertwine computational setup and results for the Android application TUM-lens, whereas for the others, we dedicate separate results chapters.

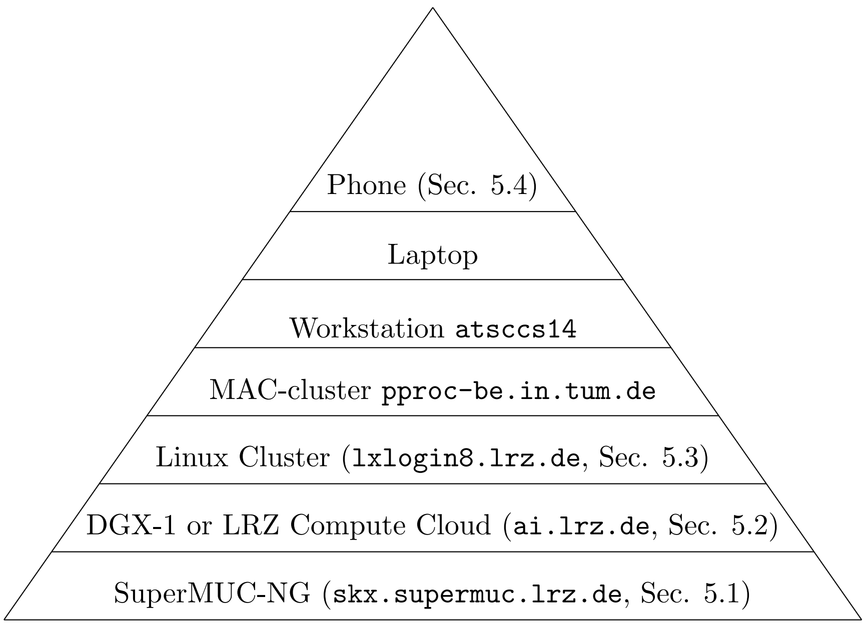

Phone (Sec. 5.4)
Laptop
Workstation atsccs14
MAC-cluster pproc-be.in.tum.de
Linux Cluster (lxlogin8.lrz.de, Sec. 5.3)
DGX-1 or LRZ Compute Cloud (ai.lrz.de, Sec. 5.2)
SuperMUC-NG (skx.supermuc.lrz.de, Sec. 5.1)

Figure 5.1: Contiguous compute architectures of various size of random access memory and FLOPS; the `teletype` typesetting denotes the `ssh` address.

For the first part of the thesis, $\mathcal{H}$-Matrices and `GOFMM`, we used SuperMUC-NG (Section 5.1) and the Linux cluster (Section 5.3). For performance runs of the deep learning part, we employed several GPU workstations, as well as NVIDIA's DGX-1 with 8 A100 GPUs (Section 5.2).

[1] According to the US National Science Foundation (NSF) visions and goals for the NSF-AI Institute ICICLE (Intelligent Cyberinfrastructure with Computational Learning in the Environment, see `icicle.osu.edu`)

We implemented an Android application, TUM-Lens (Section 5.4), to showcase image classification, object detection, and sign language recognition, running on a little edge device, like a smartphone.

5.1 SuperMUC-NG

SuperMUC-NG is Leibniz Supercomputing Centre's flagship cluster (abbreviation from German Leibniz Rechenzentrum: *LRZ*), currently #40 on the TOP-500 list[2]. It is currently the second largest in Germany after JUWELS Booster-System from Forschungszentrum Jülich (# 18) and only slightly ahead of Hawk from HLRS (# 42). It has in total 311,040 compute cores with a main memory of 719 TB and a peak performance of 26.9 PetaFlop/s. All compute nodes of SuperMUC-NG are Intel Xeon Skylake processors equipped with fast 100 Gbit/s OmniPath network interconnects. There exists also a Phase-2, where some compute nodes are equipped with 2 GPUs each. Currently, it is not accessible to HPC users, as they are struggling with power consumption of the GPU nodes [SW23]. Hence, we did not use it in this thesis.

Login is recommended through a secure shell inside the Münchener Wissenschaftsnetz (`MWN`) or with a virtual private network (VPN) through `skx.supermuc.lrz.de`. We have requested compute quota through a test project for `GOFMM`, which was granted for 300,000 compute core hours. The machine and documentation details can be found here [3]. We conducted our `GOFMM` experiments on the Skylake partitions of LRZ and TACC[4]. Each node on both machines has two 24-core Intel Xeon Platinum 8160 "Skylake" processors, which have two AVX-512 units. We use this system for Chapter 6.

5.2 Multiple GPU: DGX-1 for deep learning

For the deep learning part of the thesis, we used several GPU architectures for testing and prototyping the *Tensorflow* part; at first, a chair workstation with a NVIDIA Tesla P100 and later a Quadro RTX 5000. We intended to measure the performance more accurately and hence moved towards LRZ's AI system[5]. Compute jobs are submitted through `SLURM`, and we use this system for Chapter 7.

Nvidia DGX systems are designed to accelerate deep learning applications with improved GPU-GPU communication. For high performance, they are sometimes referred to as "Supercomputer in a box", as they consist of 4 to 16 Nvidia Tesla GPUs. The big advantage is fast memory transfer, as an NVLink mesh network connects GPUs. We display the different versions shortly in the following Table 5.1.

Table 5.1: GPU Accelerators and Their Specifications

Accelerator	**Name**	**Memory Bandwidth**	**Release Year**
P100	Pascal	720.00E0 $\mathrm{GB\,s^{-1}}$	~2016
V100	Volta	900.00E0 $\mathrm{GB\,s^{-1}}$	~2018
A100	Ampere	2039.00E0 $\mathrm{GB\,s^{-1}}$	~2020
H100	Hopper	3072.00E0 $\mathrm{GB\,s^{-1}}$	~2022

[2] Nov. 2023

[3] `https://doku.lrz.de/supermuc-ng-10745965.html`

[4] Texas advanced Compute Center with Stampede 2

[5] `https://doku.lrz.de/lrz-ai-systems-11484278.html`

Note that starting with the Ampere A100, they are equipped with a TPU unit. The TPUs are employed especially when using NVIDIA GPU containers (NGC)[6]; hence, it is recommended to be used by LRZ. In fact, NGC is a software hub for GPU-optimized applications, especially for tensor computational frameworks like Tensorflow or Horovod. The corresponding NGC container includes NVLink communications, and NGC used the MLPerf as HPC benchmark.

5.3 Linux cluster

LRZ also hosts a smaller system called Linux Cluster with an older segment CoolMUC-2 and a newer KNL segment CoolMUC-3. CoolMUC-2 cluster[7] has 812 nodes with 64GB memory per node. The nodes have 28-way Intel Xeon E5-2690 v3 ("Haswell") CPUs and FDR14 Infiniband interconnects. CoolMUC-3[8] has 148 nodes with 64 cores per node and 4 hyper threads per core. It has 64-way Knight's Landing (KNL) 7210-F many-core processors for parallel/vector processing and is equipped with Intel Omnipath OPA1 interconnects between the nodes.

We use this system for Chapter 6. Accuracy measurements that require an OpenMP number of threads less than 28 were performed on CoolMUC-2 and the rest on CoolMUC-3. Compute jobs are submitted through `SLURM`. We have used several queues with different features, such as maximum runtime or nodes. The memory or maximum runtime limits the size of the respective run.

5.4 Smartphone: TUM-lens

During the work with students in our seminar "Computational Aspects of Machine Learning", as well as other courses and theses, the question arose how image classifications are implemented on a smartphone. Usually, they upload the images to a server, do postprocessing there, and use the uploaded data to improve their models. A few years ago, there was a considerable debate around federated learning, i.e. to train on multiple local devices and aggregate on a global model [LFTL20]. With this in mind, we investigated the possibility of running some of our models in an IPython notebook, classifying a Laptop's webcam stream using the TensorFlow-Slim image classification model library (SLIM[9]). Afterwards, with the Bachelor Thesis of Max Jokel, we started TUM-lens, an application for local live image classification [Jok20]. After that, there followed several other student thesis theses [Dre21, Alh22, Kar22]. As we do not dedicate a separate result chapter for this, we explain the results of the edge device here in this section.

We started with a Java implementation, moved to Kotlin, and added feature after feature. Even with our first prototype, we were impressed by the short inference time despite the large models. The code is available in LRZ's GitLab[10] and GitHub[11]. The Android application can be downloaded in the Google Play Store here[12].

In the following, we explain the features and three different modules: image classification, object detection, and sign language recognition. The first milestone is image classification in Subsection 5.4.1. We continued in development with an object detection mechanism in Subsection 5.4.2. There, application size became an issue, which we solved using a dynamic

[6]`https://doku.lrz.de/5-using-nvidia-ngc-containers-on-the-lrz-ai-systems-10746648.html`
[7]`https://doku.lrz.de/coolmuc-2-11484376.html`
[8]`https://doku.lrz.de/coolmuc-3-11484375.html`
[9]`https://github.com/tensorflow/models/tree/master/research/slim`
[10]`https://gitlab.lrz.de/exaniml/tum-lens`
[11]`https://github.com/severin617/tum-lens`
[12]`https://play.google.com/store/apps/details?id=com.maxjokel.lens`

model zoo. Lastly, we added a different feature, namely translating sign language locally using the DGS Korpus dataset (Subsection 5.4.3).

5.4.1 Image classification

Image classification and object detection are vital tasks in the computer vision field. We show a short introduction of those tasks in Figure 5.2 showing the differences. In short, image classification maps an image to a certain probability of classes; we do neural network training for this task later in the thesis in Subsection 7.2.5. Object detection works by surrounding each object of interest in a given image or video with a boundary box, providing it with a label. Object detection combines two tasks: image classification and object localization (surrounding the object in the image with a bounding box).

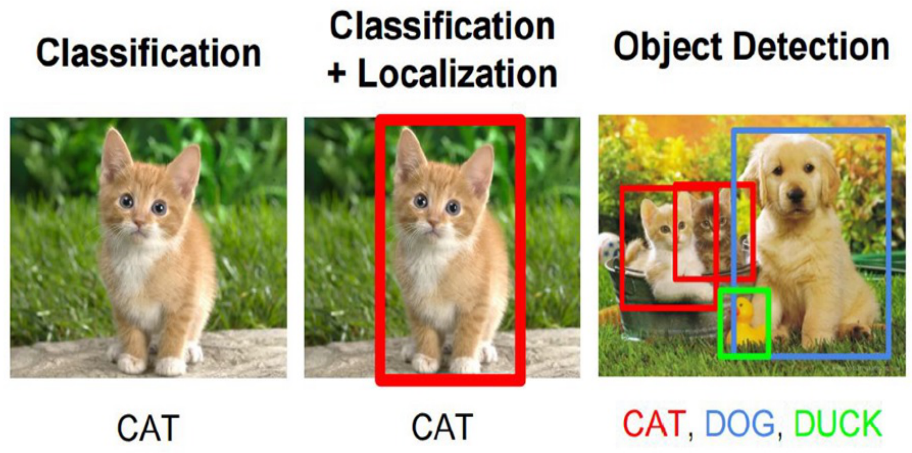


Figure 5.2: Difference between image classification, object localization, and object detection Figure taken from [Tir].

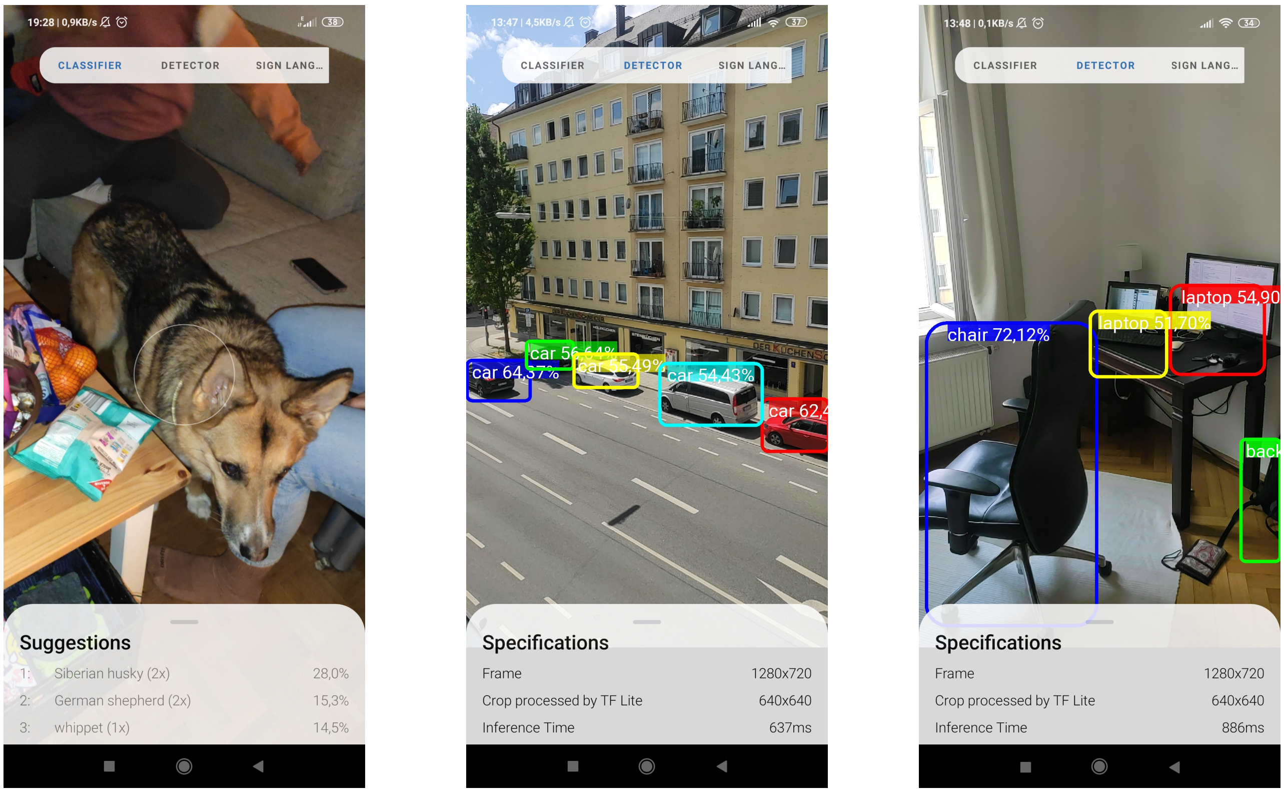
Image classification was our first implementation using Java. In the modern version, we have migrated to Kotlin. We used CameraX to capture the camera stream and allowed a CameraSelector to switch between the front and back cameras.

We crop and downsize the frame to a quadratic image of size 64×64 (similar as in ImageNet). In a loop, we evaluate a Tensorflow lite model that is trained with ImageNet or another dataset. Hence, evaluation is run on each frame with an inference time of around 50ms, 20 frames per second. There are several other features to be set, but the main objective was to keep it simple and interchangeable. To anticipate an example from later, in Figure 5.3 (a), we tested it on different dog breeds, as dog pictures are widely prominent in ImageNet, a dataset with 1000 classes consisting of 1.3m images. For mixed breeds, it can help in suggesting the dog's origins if, e.g., the dog owner does not know its origin. There are distinct visual features like nose, tail, ears, etc. that can help with this task.

5.4.2 Object detection

Object detection is another vital task in the computer vision field. As shown in the previous Figure 5.2, object detection is an image classification task combined with object localization (surrounding the object in the image with a bounding box). There are two types of object detection, and they are as follows:

- **Single-class object detection:** Detecting a single class instance in an image.



Suggestions

1:	Siberian husky (2x)	28,0%
2:	German shepherd (2x)	15,3%
3:	whippet (1x)	14,5%

Specifications

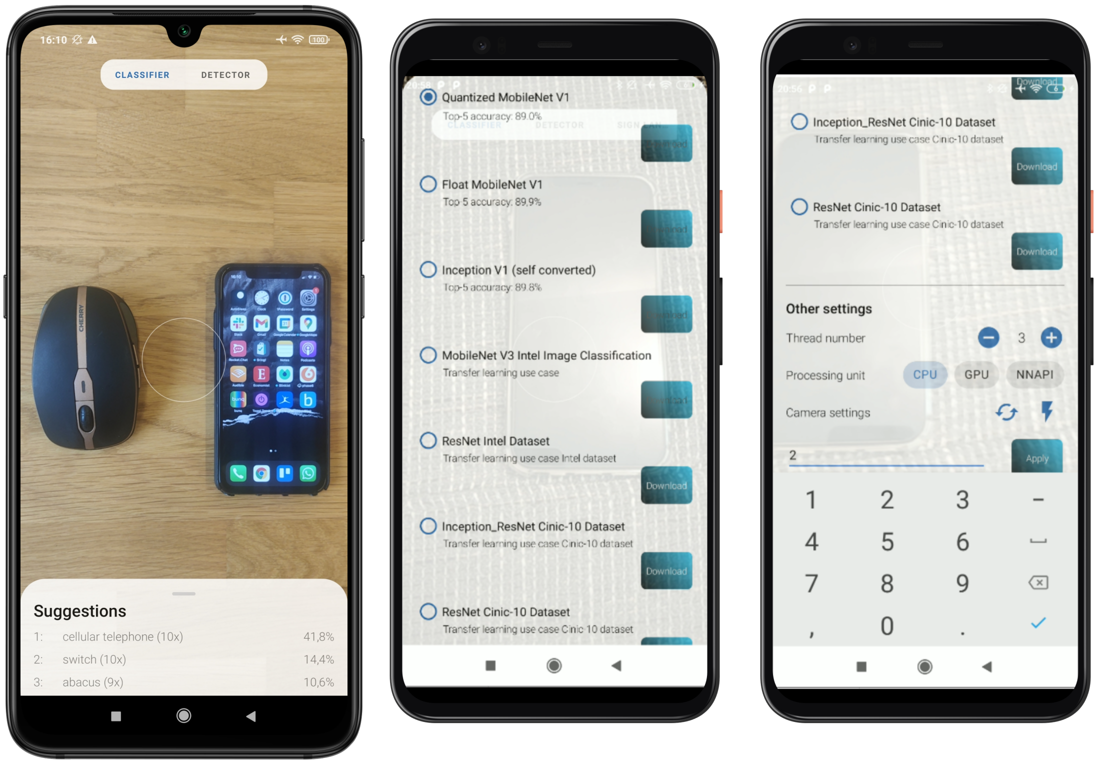
Frame	1280x720
Crop processed by TF Lite	640x640
Inference Time	637ms

Specifications

Frame	1280x720
Crop processed by TF Lite	640x640
Inference Time	886ms

(a) Image classification **(b)** Traffic **(c)** Home workplace

Figure 5.3: Screenshots of live image classification and object detection in TUM-Lens, developed along the thesis.



Suggestions

1:	cellular telephone (10x)	41,8%
2:	switch (10x)	14,4%
3:	abacus (9x)	10,6%

Quantized MobileNet V1
Top-5 accuracy: 89.0%

Float MobileNet V1
Top-5 accuracy: 89.9%

Inception V1 (self converted)
Top-5 accuracy: 89.8%

MobileNet V3 Intel Image Classification
Transfer learning use case

ResNet Intel Dataset
Transfer learning use case Intel dataset

Inception_ResNet Cinic-10 Dataset
Transfer learning use case Cinic-10 dataset

ResNet Cinic-10 Dataset
Transfer learning use case Cinic-10 dataset

Other settings
Thread number: 3
Processing unit: CPU GPU NNAPI
Camera settings
2

(a) Image classification (b) Different models (c) Settings and delay time

Figure 5.4: Screenshots of the object selection and settings screen

- **Multi-class object detection:** Identifying the labels of all the objects in an image.

Nowadays, object detection is used in many applications, such as self-driving cars, tracking objects, face detection, face recognition, activity recognition, medical imaging, automatic image annotation, and many more applications in the computer vision field. It is a vital topic in a visual identification activity. [VNRP19, PPR18] It works by iteratively placing bounding boxes around it on different parts of the image and refine. The most prominent one is Objectron[13], which builds the foundation of our module. It follows the following objectives:

1. **Verification:** verify that the object is present in the image.
2. **Detection & Localization:** detect the object in the image and then localize its specific location.
3. **Classification:** select potential categories for the object before classifying it.
4. **Naming:** identify the location of the objects in the image and their labels
5. **Description:** describe the object's relationships and actions in line with the image context

We integrated it in TUM-Lens using another module to be selected in the top pane. As expected, the object detection is a bit slower, and the inference time is around 600ms, with a

[13]https://github.com/google-research-datasets/Objectron/

frame rate of around 2fps. In Figure 5.3 (b) and (c), we show the object identification screens. In the scenario of traffic in (b), it can be used to count cars or in autonomous driving scenarios. In (c), we simply exposed it to the home workplace, showing different objects like a chair, laptop, or backpack.

In Figure 5.4 (a), we see two objects in the image. TUM-lens suggests celluar phone with 41% accuracy; clearly, it does not capture the second object, a mouse. Note that image classification is not very suitable for this picture as there are two objects shown. Object detection would be more reasonable.

Settings

In Figure 5.4, we display the image classification and the settings pane. In Figure 5.4 (b), we see the different models like inception net that was trained on the CINIC-10 dataset. Below, you can also set the thread number and processing unit (NNAPI abbreviates from Neural Network API and uses respective accelerators if equipped in the phone). The user can also switch between cameras, use the flash, or insert some delay for slower (and more readable) results.

In Figure 5.4 (c) we show the **model zoo**. For each model, the weights must be stored on the device, which makes up the biggest portion of the application's file size. One larger model needs around 20MB of storage despite compression. We moved towards an online location[14] for models that can be downloaded on-the-fly with the small blue buttons.

5.4.3 Sign language detection

More students were interested in the project after publishing the smartphone app in the Google Play Store. We came about sign language recognition from the DGS Korpus (**D**eutsche **G**ebärden**s**prache). Max Karpfinger came up with a pipeline that does not involve the direct image pixels. Instead, from a picture of two hands, he uses MediaPipe to get the location of fingertips, knuckles, etc., as an output vector. From this, he trained a fairly small convolutional net on the DGS Korpus database towards their labels. [Kar22] Thereafter, we integrated the pipeline in a new module in TUM-lens.

In Figure 5.5, we selected the sign language recognition in the upper white pane. In Figure 5.5 (a), you can select the models, again to be downloaded from the server to save app size. It also shows that it cannot detect the left and right hand, as this is required for the analysis.

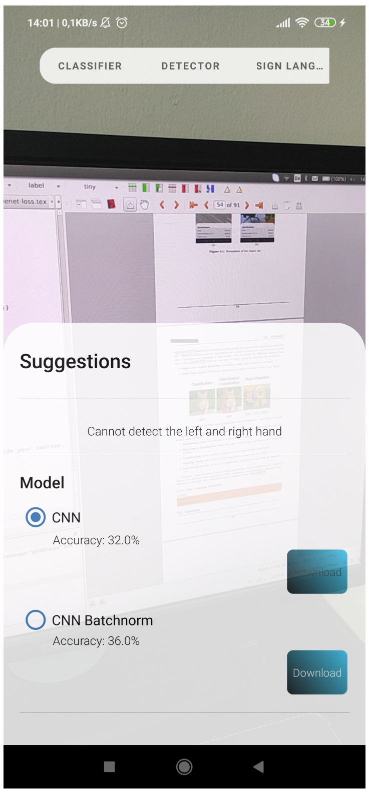
In Figure 5.5 (b), I (the writer, doctoral candidate) show the gesture for staying calm. The suggestions predict this with high certainty. The $ sign denotes gestures, in contrast to normal "words" that are without (see the second option RUND).

In summary, we do not reach very high accuracy as the data is too small, and the models are not developed mature enough. However, it shows that those applications are handy and can be adapted to various tasks in the future.

5.5 Summary

In summary, we have employed compute infrastructure on various levels. The `GOFMM` and $\mathcal{H}$-matrix-arithmetic were run on supercomputers and clusters. They have many compute nodes, Skylake, Knights Landing, and Haswell nodes. The significant advantage is the fast Infiniband interconnections for memory connections, which are necessary to show scalability for distributed memory parallelism using the message passing interface (`MPI`).

[14] `https://www5.in.tum.de/~reiz/osama/`



Suggestions

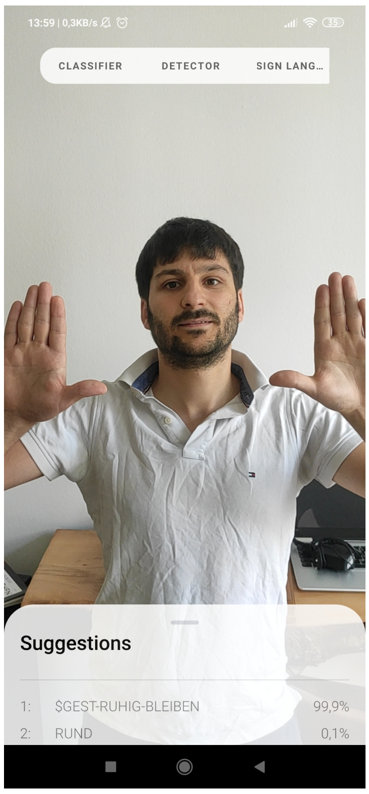
Cannot detect the left and right hand

Model

CNN [✓]
Accuracy: 32.0%

CNN Batchnorm
Accuracy: 36.0%

(a) Settings and model selection. The sign recognition requires two detect two hands to return results



Suggestions

1:	$GEST-RUHIG-BLEIBEN	99,9%
2:	RUND	0,1%

(b) $ denotes gesture; app suggests gesture ”ruhig bleiben”, ”remain calm” wit 99% certainty. German language due to DGS data labeling in German

Figure 5.5: Screenshots of the sign language detection

For deep learning, there are different requirements. A multi-threaded GPU is very beneficial; with Horovod, we also used multiple GPUs. Memory transfer between CPU-GPU or GPU-GPU usually dominates the runtime. NVIDIA DGX-1 machines are very beneficial, as they are optimized for memory transfer. In addition, we used the NGC containers, which are optimized for such machines. Hence, `Newton-CG` can be run on any GPU setting; our production runs are performed on an NVIDIA DGX-1.

Lastly, for showcasing purposes, we implemented a demonstrator TUM-lens, a live local image classification Android application. It offers image classification, object detection, and sign language recognition. For smaller software sizes, we employ an online model zoo. Some of its models are trained with `Newton-CG`. TUM-lens is available in the Google Playstore[15].

[15] `https://play.google.com/store/apps/details?id=com.maxjokel.lens`

The aim of the technique of hierarchical matrices is to perform all matrix operations (including matrix-matrix multiplication, inversion, and LU decomposition) in almost linear cost, which means $\mathcal{O}(N \log^* N)$ including logarithmic factors.

Wolfgang Hackbusch (*1948)

6 $\mathcal{H}$-Matrices

Dense SPD matrices are often the computational bottleneck of an algorithm; they appear in partial differential equations, integral equations, and Hessian operators for statistical learning in diffusion maps or graph operators. With geometric information, setting up a dense matrix is avoided, e.g., by restricting to local support to get sparse matrices in, e.g., partial differential equations, or by clustering points and using, e.g., geometric distance cut-off. The foundations of $\mathcal{H}$-matrices are explained in Chapter 2.

It is not possible to use geometric information when only given a black-box matrix. `GOFMM` introduces a row/column distance metric using the ℓ_2 or angular distance in Gram vector space. This scheme can be done for every matrix, reducing memory requirements and computational cost of operations such as the matrix-vector multiplication from $O(N^2)$ to $O(NlogN)$. We also allow with `GOFMM` a pseudo-inverse factorization. We applied it to many matrices and offer also a Python version to be used in other codes. Therefore, we identified the computational bottleneck of the diffusion maps algorithm and enabled the eigenvector decomposition with matrix-vector products from `GOFMM`.

In Figure 6.1, we show random Gaussian data points. If you were to introduce a radial basis function (RBF) kernel on the data, it would be natural to approximate (or disregard) the data from far away clusters. `GOFMM` tries to do this in an algebraic sense for an RBF-kernel matrix.

This chapter presents the methods, implementation, and results of the $\mathcal{H}$-Matrices/`GOFMM` part. We start by explaining the implementation of `GOFMM` and the `SWIG` interface for Python, as well as the test cases (Section 6.1). We then explain how our code performs with the geometric kernels (Section 6.4). Section 6.4.1 explains the distributed-parallel results and compares it to another $\mathcal{H}$-Matrix code, `STRUMPACK` and a Gauss-Newton Hessian multiplication and factorization. We continue with the Python interface (Section 6.5), which is formed by the applications to the diffusion map manifold operator code `datafold`. We close with a summary in Section 6.6.

6.1 Implementation: Geometry-oblivious fast multipole method

In Chapter 2, we have reviewed the theoretic foundations of $\mathcal{H}$-matrices for the geometry-oblivious fast multipole method. Our code is also called similar, to which we refer to with `GOFMM`. In this Section 6.1, we give details about the implementation of `GOFMM`.

We start in Subsection 6.1.1 by recalling the *geometry-oblivious* Gram distance notion and outline the splitting of indices in near and far-field.

Next, in Subsection 6.1.2, we explain the ingredients for `GOFMM`. We go through the steps of `GOFMM` pseudocode, where we start with a (1) static reordering tree and move to (2) neighbor computation. We then perform the (3) compression, (4) multiplication, and (5) factorization.

In Subsection 6.1.3, we give additional remarks for parameters to steer the approximation accuracy and acknowledge joint contributions.

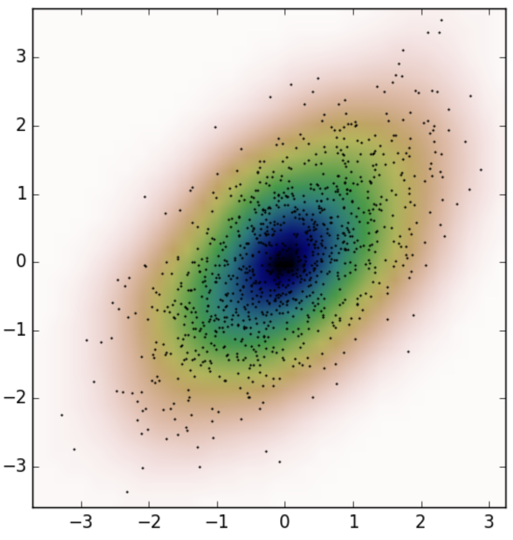


Figure 6.1: 2D **Gaussian** kernel density estimation plot with `GOFMM`. Black dots are the original data points, while colored contours are their density approximation. It is natural to approximate interactions of **far-away** points; with no points given for a SPD matrix, `GOFMM` enables **algebraic approximation** using a Gram-Vector distance metric, trying to resemble **far-field approximation**.

6.1.1 Gram distance notion

Any SPD Matrix can be constructed by scalar products of unknown Gram vectors ϕ, namely

$$K_{ij} = \langle \phi_i, \phi_j \rangle$$

where ϕ are unknown Gram vectors. The Gram vectors can be computed with a Choleksy factorization (they are not unique), but this is not necessary for our purposes.

In this Gram-vector space, we can compute distances between rows i and j and sort them according to Gram-distance without any knowledge of the underlying geometry (i.e., geometric structure on how the matrix was created). For most applications, local support and lexicographic ordering are beneficial for the (sparsity) structure of the matrix. In detail, we formulate **two** distances strategies: Gram-ℓ_2 and Gram-angle (introduced in [CLRB17]), given by

- Gram-ℓ_2

$$\begin{aligned} ||\phi_i - \phi_j||_2^2 &= \langle \phi_i - \phi_j, \phi_i - \phi_j \rangle = \\ &= \underbrace{\langle \phi_i, \phi_i \rangle}_{K_{ii}} - 2 \underbrace{\langle \phi_i, \phi_j \rangle}_{K_{ij}} + \underbrace{\langle \phi_j, \phi_j \rangle}_{K_{jj}} , \end{aligned}$$

- and Gram-angle $\sphericalangle$

$$\cos(\sphericalangle(\phi_i, \phi_j)) = \frac{\langle \phi_i, \phi_j \rangle}{\|\phi_i\| \cdot \|\phi_j\|} = \frac{\langle \phi_i, \phi_j \rangle}{\sqrt{\langle \phi_i, \phi_i \rangle \langle \phi_j, \phi_j \rangle}} .$$

Note that Gram ℓ_2 or the angle $\sphericalangle$ between rows or columns can be computed by only **3** entries of the respective matrix, so it's very *cheap* compared to many distance computations in a high-dimensional Euclidean geometric space.

Let us assume that interactions (kernels) between points degrade with (Gram-)distance. Such an assumption/approximation is used in *far-field* approximation algorithms, splitting the points into close and far away points. Those approximations are commonly used in Molecular Dynamics for gravitational field or force calculations, like the Barnes Hut [BH86] and fast multipole method [GR87]. Hence, we split the summation,

$$u_i = \sum_{p \in \text{Near}_i} K_{ip} w_p + \sum_{p \in \text{Far}_i} K_{ip} w_p$$

where Near_i is a set of near points, whose contributions need to be regarded individually, and Far_i being the set of far points, which contributions can sufficiently be computed by a low-rank approximation. Since plain distance is less meaningful in high dimensions, we often disregard the exact distance value but rather decide upon Near and Far estimates by counting the neighbors in respective leaves.

6.1.2 GOFMM ingredients

In the following, we go through the five steps of: 1. Reordering, 2. Neighbors, 3. Compression, 4. Multiplication and 5. Factorization. They basically form the pseudocode for `GOFMM` and are displayed in Figure 6.2. In the following, we describe those steps and refer to the Theory in Section 2.2 and the sections around it, where necessary.

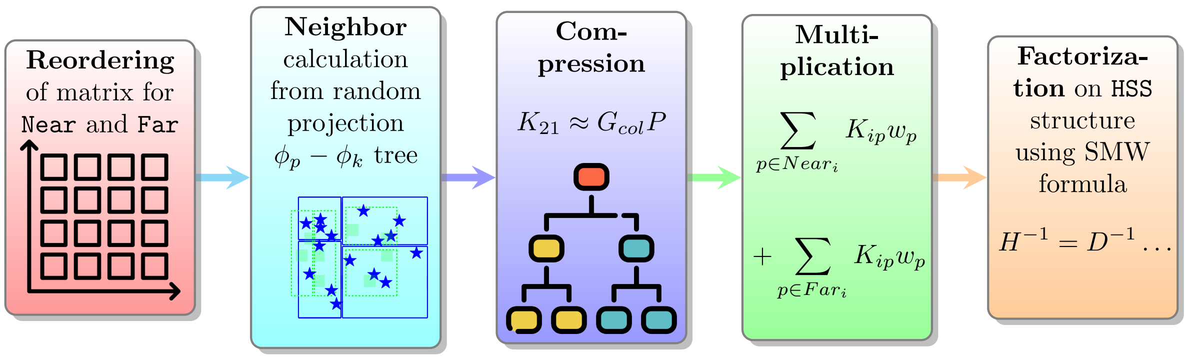

Reordering of matrix for `Near` and `Far`
Neighbor calculation from random projection $\phi_p - \phi_k$ tree
Com-pression
$K_{21} \approx G_{col} P$
Multi-plication
$\sum_{p \in Near_i} K_{ip} w_p$
$+ \sum_{p \in Far_i} K_{ip} w_p$
Factoriza-tion on `HSS` structure using SMW formula
$H^{-1} = D^{-1} \dots$

Figure 6.2: Pseudo-code of different steps in `GOFMM`

1. Reordering

In the reordering step, we first calculate an index p, as ϕ_p being farthest away from the center of mass $\hat{\phi}$ of corresponding points in the tree node

$$\phi_p = \underset{j}{argmax} \left\| \hat{\phi} - \phi_j \right\|$$

where we sort then by the *Gram*-distance to the approximate center of mass $\hat{\phi}$ by

$$\left\| \hat{\phi} - \phi_j \right\|^2 = \left\| \sum_{i=1}^{N} \phi_i - \phi_j \right\|^2 = \sum_{i=1}^{N} \|\phi_i\|^2 + \|\phi_j\|^2 - 2 \sum_{i=1}^{N} < \phi_i, \phi_j > .$$

We find ϕ_p as the furthest point ϕ_p in *Gram*-distance by taking the maximum maximum (argmax) of all rows j. Instead of summing i until N for the exact center, in practice, we only compute the distance to an approximate center by summing over a random subset; hence, we find ϕ_p. Subsequently, we find the index k to ϕ_k being on the opposite end of the distribution, i.e., being far-most away from ϕ_p. We then sort the other elements according to this projection axis $\phi_p - \phi_k$ and split by the median for a static hierarchical tree ordering.

2. Neighbor calculation for sampling rows

To compute **neighbors**, instead of an exhaustive search, we use a random k-d tree algorithm using a random projection axis (in literature, often RKDT). In each iteration and for each hierarchical split, we select a random direction $\phi_p - \phi_k$ (p and k are not the ones from above; they are random here), and split a tree node using our two Gram notions ℓ_2) and $\lhd$ into two children *left* and *right*

(ℓ_2) using a fictive orthogonal hyper-plane defined by the median,

($\lhd$) or fictive hyper-cones defined by a (roughly) equal-sized split.

We compute distances exhaustively from the set of candidate neighbors (all points that were in a leaf together) and sample rows from the `k` nearest neighbors.

3. Compression

For a hierarchical off-diagonal low-rank decomposition, we need to compute respective low-rank decompositions. The most accurate bound for errors can be achieved with a singular value decomposition ($G \approx U\Sigma V^*$); however, one must **store** matrices U, Σ, V in memory. As described in Section 2.2, an interpolative decomposition instead uses $G \approx G_{col}P$, where G_{col} is a subset of columns of G. We can bound the error for a rank s estimation with some factors, as described in Section 2.2. The big **advantage** of the **interpolative decomposition** is that we only need to store the indices of the respective columns (assuming we have access to entries of G) and a $s \times N$ projection matrix P. Note that the projection matrix P consists of the first columns of the identity matrix ($s \times s$), through which we can also save some storage. In summary, an interpolative decomposition gives us **memory benefits**.

Thus, we look at a hierarchical split matrix of the form

$$\begin{aligned} K &= \begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} \\ K_{21}^{(1)} & K_{22}^{(1)} \end{bmatrix} = \begin{bmatrix} K_{11}^{(1)} & [\hat{G}_{col}P]_{12}^{(1)} \\ (\hat{G}_{col}P)_{21}^{(1)} & K_{22}^{(1)} \end{bmatrix} \\ &= \begin{bmatrix} \begin{bmatrix} K_{11}^{(2)} & (\hat{G}_{col}P)_{21}^{(2)} \\ (\hat{G}_{col}P)_{21}^{(2)} & K_{22}^{(2)} \end{bmatrix} & (\hat{G}_{col}P)_{12}^{(1)} \\ (\hat{G}_{col}P)_{21}^{(1)} & \begin{bmatrix} K_{33}^{(2)} & (\hat{G}_{col}P)_{34}^{(2)} \\ (\hat{G}_{col}P)_{43}^{(2)} & K_{44}^{(2)} \end{bmatrix} \end{bmatrix} = \dots , \end{aligned} \tag{6.1}$$

where we approximate the off-diagonal G using an interpolative decomposition, $G \approx G_{col}P$ with G_{col} as the first s columns of the off-diagonal. We do this recursively to form a hierarchy, hence $\mathcal{H}$-matrix.

A reader may have found that we use $\hat{G}$ instead of G. In order to keep the computational effort low, limiting the matrix sizes of the off-diagonals is vital. On the lowest leaf level with node size `m` we use the matrix of size $k \times m$, where $k \approx 2m$. k are sampled **rows** from neighbors. Alternatively, we can also read them user-given from memory, or additional rows can be sampled if there aren't enough.

As described in Section 2.2, we use a **pivoted QR** to compute the interpolative decomposition, $G\ \Pi = Q\ R$ with Π that reshuffles the rows (pivot element). The projection matrix P is computed by a **triangular solve** with $R_{11}P = [R_{11}R_{12}]$. As for adaptive rank selections (see also Section 2.2), we employ the *rank revealing QR* factorization and choose an approximation rank as the limit when the diagonal entry R_{ii} falls below a user-given `tol` (see Section 2.2 for the connection between R_{ii} and the singular value for correspondence to tolerance). Furthermore, if we cannot reach the desired tolerance, we cap off at the integer s_{max}. The selected columns are then called skeletons, and $\hat{G}$ consists of the required rows (from neighbors or similar) and columns (from skeletons).

Hence, on a **leaf** level we do for **both** neighbors and skeletons:

- `neigh`: Sample rows from neighboring indices
$$\mathcal{N}_\alpha = \cup_i \mathcal{N}_i \text{ where } i \in \alpha$$
- `skel`: Select skeletons and save
$$\mathcal{S}_\alpha = \cup_i \mathcal{S}_i \text{ where } i \in \mathcal{S}_\alpha \ .$$

On **interior** levels, we must use an additional heuristic to choose neighbors and skeletons. For neighbors, we simply exempt nodes that are already inside. For skeletons, we merge the two children. It would be too expensive to do this more accurately since approaching the root, the interpolative decompositions would get prohibitively expensive. Hence, we build *skeletons* from the lower level.

- Use columns from children $\mathcal{S}_{\text{left}} \cup \mathcal{S}_{\text{right}}$
- Sample rows from children skeleton neighbors $\mathcal{N}^s_\alpha = (\mathcal{N}_{left} \cup \mathcal{N}_{right}) \backslash (\texttt{Nodes}_{left} \cup \texttt{Nodes}_{right})$

Hence, we compute the ID on $\hat{G}$ consisting of required rows (from neigbors or similar) and columns (from skeletons). We call the compression phase also *skeletonization.*

The advantage of our approach is the good and easy rank estimate. Many researchers refrain to randomized approaches, which we did not include due to difficulties in handling these libraries; we solely rely on LAPACK, which does not include Randomized Matrix Algorithms. In the future, one can use other methods like randomized approximate matrix decompositions. [HMT11]

4. Multiplication

After having computed a compression, we can then employ this to save arithmetic operations by using the approximation. We differ between the hierarchical semi-separable scheme (`HSS`) and, inspired by geometric schemes in molecular dynamics, the fast multipole method (`FMM`). In an *HSS* scheme, we follow a matrix multiplication from Equation 6.1. In a multiplication $u_i = \sum_{p \in Near_i} K_{ip} w_p + \sum_{p \in Far_i} K_{ip} w_p$ the near field is considered to be the diagonal blocks from Equation 6.1, respectively, and the far field is the off-diagonals, respectively. We sometimes refer to vector w as charges.

In an `FMM` scheme for a geometric low-dimensional case (up to 3D), often a cut-off radius is used to determine the Near and far-field particles. Verlet lists are used to keep track of the Near and Far field contributions. As we are in high dimensions, this distance becomes less meaningful. Instead, we determine Near and Far fields by neighbors. If one neighbor falls into another leaf, this leaf is added to the near-list. We limit this sometimes with a parameter budget, i.e. that we allow a certain percentage of Near-field (e.g. 10%); this compromises accuracy, of course.

In order to use the far-field approximation, we first need to accumulate charges at skeletons. We start at the leaf and go to the root in a post-order traversal. We then compute the skeleton-skeleton interactions in any order, followed by a pre-order traversal to come from skeleton potentials to the leaf node. We then add the near-field interactions.

This follows the approach from `ASKIT`, a predecessor of `GOFMM`; a more detailed analysis can be found in [MXT+15a].

5. Factorization

In `GOFMM`, we do matrix splitting of the form $H = D + UV + S$ where D, UV, S are block-diagonal, low-rank, and sparse matrices. We differ between `HSS` (hierarchical semi-separable, no sparse corrections) and `FMM`, while `FMM` expects corrections involving a distance notion of particles near the edge of two leaves (ghost region), i.e. for us with a neighbor based metric. This is resembled in the sparse correction matrix S. For this factorization, we only allow `HSS`, as sparse corrections would require more intricate treatment using the Schur complement.

We have applied the Sherman-Morrison-Woodbury formula on blocks of the matrix for approximate inversion. Given $H = D + UV$ we can write the Sherman-Morrison-Woodbury (SMV) as

$$H^{-1} = D^{-1} - D^{-1}U(I + VD^{-1}U)VD^{-1},$$

where D is a (block) diagonal matrix and is much easier to invert.

We see that only (approximate) inversions of smaller matrices are needed. We have done this through partial pivoted LU Factorizations on blocks of the matrix; we see that many blocks are zero, as we follow the publication [YRB19], a direct $\mathcal{H}$-matrix-inversion in `GOFMM`. It uses the so-called ULV-factorization; a similar work recently by Deshmukh, Yokota et al. [DYBM23] recently for $\mathcal{H}$-matrices.

6.1.3 Additional remarks

We tested GOFMM for several test matrices. We assembled the test cases ourselves to control properties and analyze behavior; they are described in Section 6.3.

Parameters and approximation error

`GOFMM` allows different *distance metrics*: Gram-ℓ^2, Gram-angle, and geometric-ℓ^2. For *compression*, we set parameters

- m (leaf node size),
- s (maximum rank),
- τ (accuracy tolerance used to adaptively select the rank), and
- κ (number of neighbors used for S and for sampling).

We also allow the parameter `budget` where the user can define a desired percentage of direct evaluations - to control cost vs. accuracy. The relative error ϵ_2 is computed by a Frobenius norm towards a (sampled) exact *matvec* (see Equation 6.2).

Throughout, we use relative error ϵ_2 defined as the following

$$\epsilon_2 = \|\tilde{K}w - Kw\|_F / \|Kw\|_F, \text{ where } w \in \mathbb{R}^{N \times r}. \tag{6.2}$$

This metric requires $\mathcal{O}(rN^2)$ work; to reduce the computational effort, we instead sample 100 rows of K. In all tables and plots, we use "`Comp`" and "`Eval`" to refer to the compression and evaluation time in seconds.

Joint contributions and acknowledgement:

`GOFMM` and the further development `MPI-GOFMM` was joint work with George Biros' group, namely Chenhan Yu, James Levitt, Chao Chen, and George Biros. I, Severin Reiz, tested the geometry-oblivious distance metric and the algorithms on black box SPD matrices from an idea by George Biros. We analyzed eigenvalue spectra of off-diagonal low ranks with a theoretic Python script with different numbering schemes. Chenhan did the C++ OpenMP and MPI-based implementation of `MPI-GOFMM` and the large performance runs. In `MPI-GOFMM`, I implemented the matrix partitioning schemes, as well as neighbor indexing for approximate nearest neighbors, optimization using the `std::sort` and a coarse formulation of the factorization. For the accuracy test, I added an automatic latex table generator and flags in main for Gram-ℓ^2, Gram-angle, and geometric-ℓ^2 argument flags. James introduced the Gram-angle criterion and advanced a lot on the geometric-ℓ^2 schemes. George laid out many theoretical explanations, isolated some theoretically strange behaviors, and thereby smoothened some bugs. He also wrote the matrix generation scripts in *MATLAB*. For the analysis of the Gauss-Newton Hessians, Chao Chen has written the autoencoder PyTorch implementation, including the sampling scheme for matrix generation.

`MPI-GOFMM` is implemented in `C++` with `MPI` (`MPI_THREAD_MULTIPLE` is required) and `OpenMP`. `MPI-GOFMM`'s only dependencies are multi-threaded `BLAS/LAPACK` and `MPI` (we used the Intel MPI library on TACC/LRZ respectively)[1]. All runs are performed in single precision. In [CRB18] we estimated the computational efficiency by FLOPS counts for a matrix multiplication to 27% to peak performance.

6.2 Implementation: GOFMM Python interface

To analyze `GOFMM` (and its Hierarchical matrix approximation scheme), it is cumbersome to store it in dense memory in the proper format and then apply the `GOFMM C++` code. Besides, other offered interfaces (like point-cloud or MLP that generate entries on-the-fly, etc) are also not handy for another application code. Our ultimate goal was to allow `GOFMM` in `numpy` directly. Especially in order to couple it to `datafold`, we required a `python` interface for `GOFMM`. The ultimate goal is to allow for broader applicability of `GOFMM`.

6.2.1 Python bindings using SWIG

A popular wrapper for this is the **Simplified Wrapper and Interface Generator** (SWIG[2]). It is a tool to wrap code written in `C++` to scripting languages, e.g. `Python`. It is available in github[3] and can be installed with pip[4].

In order to wrap a `C++` code with `SWIG` for `Python` simply a header needs to be written (e.g., `tools.i`); the `C++` code needs to be compiled, the `SWIG` script linked and then a python library (in addition to executable and shared object file) is generated. The Python library can then be imported, and the linked methods can be executed from there.

It was necessary to adapt the `GOFMM C++` classes and the compile process, where we modified `test_gofmm.cpp` and `gofmm.hpp` (see appendix Listing A.1). In detail, we created a `GOFMM` tree class (`gofmmTree`) to encapsulate the functionality and wrap the command line helper.

[1]See the repository `https://github.com/severin617/hmlp-1`

[2]`https://swig.org`

[3]`https://github.com/severin617/gofmm_swig_python`

[4]`https://pypi.org/project/gofmm1/`

Afterwards, we wrote the bindings file `tools.i` with the algorithms we intended to wrap. The general setting of SWIG and other wrapper snippets are included in the file *tools.i*. We list an example of the matrix loading in Listing A.2. With `SWIG`, we successfully transported the augmented `GOFMM` data structure with a modified container and matrix operations from C++ to Python. A big challenge was to interface the 2D arrays from numpy to the SPD matrix class from `GOFMM`, which we managed by a standard numpy snippet suggested online.

In the compile process, `SWIG` is then executed to initialize the bindings; afterwards, the cpp files are compiled and linked with the flags from `GOFMM` (using g++). The biggest challenges was to find the necessary include flags which required several development cycles.

Listing 6.1: Compile process of the SWIG bindings

```
swig -o toolswrap.cpp -c++ -python tools.i

g++ -o tools_wrap.os -c -I/usr/include/python3.8 -I/home/getianyi/.local/lib/
    python3.8/site-packages/numpy/core/include/ -I../gofmm/ -I../include/ -I../
    frame/ -I ../frame/base/ -I../frame/containers/ toolswrap.cpp -fPIC

g++ -O3 -fopenmp -m64 -fPIC -D_POSIX_C_SOURCE=200112L -fprofile-arcs -ftest-
    coverage -fPIC -DUSE_BLAS -mavx -std=c++11  -lpthread -fopenmp -lm -L/usr/
    local/lib/ -lopenblas tools_wrap.os  -o _tools.so -shared  -Wl,-rpath,./
    build: libhmlp.so -Wl,-rpath,./build: libhmlp.so -Wl,-rpath,/usr/local/lib -
    lblas -Wl,-rpath,/usr/local/lib -llapack
```

From Python, the wrapped `GOFMM` functions are now easy to call. Simply, the library needs to be imported, and next, the class `gofmmTree` can be used; at first, it must be initialized with the executable and the setting. Note that we use a class called `denseSpd`, which is a converted 2D-numpy array to the `GOFMM` matrix structure. It is the user's responsibility to check for SPD-ness beforehand. Out of curiosity, we also tried `GOFMM` on non-spd matrices. Although the Gram distance assumptions are not given, they sometimes yield acceptable results. We cannot give guarantees as we do not know what happens theoretically without Gram distances. `GOFMM` and the `SWIG` interface are research projects still under development with regular updates; we do not exercise a production/release cycle.

In Python (see Listing 6.2), we import the library tools and can do a static compression using `gofmmTree` per constructor. `mul_denseSPD` performs the multiplication with the vector `wData`.

Listing 6.2: Python include for using GOFMM functions

```
gofmmCalculator = tools.gofmmTree(self.executable, self.spdSize,
                                  self.m,
                                  self.k, self.s, self.nrhs,
                                  self.stol, self.budget,
                                  self.distance, self.matrixtype,
                                  self.kerneltype, self.denseSpd)

# return a 1D array which is a 2D matrix flattened row-wise
c = gofmmCalculator.mul_denseSPD(self.wData, self.lenMul)
c.resize(self.spdSize, self.nrhs)
```

A `numpy` reshape is necessary for getting the expected numpy matrix multiplication normal output (with correct shape). The reason is that there is no default typemap for 2D array as an output although there is one as an input. As a result, we need to resize the 1D flattened array into 2D afterwards (see Listing 6.2).

6.2.2 GOFMM integration into datafold

In this Subsection 6.2.2, we explain how we used the SWIG Python bindings in `datafold`. We first motivate the hierarchical approximation for diffusion maps. For simplicity, we created a Charlicloud container, a docker-like container for portability. We explain how we have employed the LinearOperator class of `SciPy` and close with a summary.

Hierarchical approximations for diffusion maps

In the diffusion maps algorithm listed in the Theory chapter (see Subsection 2.3.2), the eigendecomposition on the transition matrix is the most expensive part of the algorithm. As for the standard form, it scales with $\mathcal{O}(N^3)$ where N is the data dimension. The eigendecomposition is used to obtain the underlying lower dimension of the dataset. We perform an Arnoldi iteration with the linear operator class for finding eigenvectors and eigenvalues. There, we accelerate the matrix-vector multiplications using hierarchical matrix approximations. Assuming we have around $\mathcal{O}(N \log N)$ `GOFMM` multiplication cost, this results in roughly $\mathcal{O}(N^2 \log N)$.

We either **(1)** use the Arnoldi-iteration with `GOFMM` matvec from `scipy`'s eigsh (explained below) or **(2)** we store the matrix in memory and run `GOFMM` separately. For stategy (1) the advantage is the handiness in integration, strategy (2) allows bigger cases.

Charliecloud

This is collaborative work for Keerthi Gaddameedi's Master's Thesis [Gad22]. To improve software usability, namely not having to deal with compilation repeatedly, we created a docker file with commands to install all the runtime dependencies followed by installation of `GOFMM` and `datafold`. LRZ Linux cluster only offers such images to be used with Charliecloud, as docker is not suitable for cluster settings. Hence, we converted the docker image of `GOFMM+datafold` to Charliecloud according to the manual. This conversion to a charliecloud image involves the command `ch-builder2tar <docker-image> /dir/to/save`. Then, the charliecloud image is exported to the Linux cluster and unpacked with the command `ch-tar2dir <charliecloud-image> /dir/to/unpack`. Once the compressed image is unpacked, the environment variables are set, and `GOFMM` is compiled. Finally, the SWIG interface file is compiled to generate Python versions of `GOFMM C++` functions.

Figure 6.3 shows the function dependencies of our implementation in a UML diagram. Note that we used the class FullMatrix, where we specified a few static variables used for the `GOFMM` integration. This extends the LinearOperator class of `scipy.sparse.linalg`. We describe this in the following.

LinearOperator

`SciPy` [VGO+20] is open-source, free software for Python with modules for common tasks of scientific computing from linear algebra, (sparse) solvers, interpolation, etc. Usually, the direct matrix operations are included in `numpy`, and the linear algebra algorithms are gathered in `SciPy`. It contains seven array/matrix classes suitable for different types of representations, such as Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Coordinate format (COO), etc. It also accommodates methods to build various kinds of sparse matrices and has the algebra algorithms in `linalg`. In the submodule `linalg`, we find an abstract interface named `LinearOperator` for iterative solvers to perform matrix-vector products. This interface consists of methods such as `matmat(x)`, `matvec(x)`, `transpose(x)` for matrix-matrix multiplication, matrix-vector multiplication, and transposition of a matrix. A concrete subclass of `LinearOperator` can be instantiated with `matvec(x)` or `matmat(x)` methods and with attributes for `shape` and `dtype`. Depending on the type of matrices at hand, corresponding matvec methods may also be implemented.

A class called `FullMatrix` is derived from `LinearOperator` interface which belongs to `scipy.sparse.linalg` package (see Figure 6.3).

`scipy.sparse.linalg` provides algorithms for computing matrix inverses, norms, decompositions, and linear system solvers. The functionality we are interested in is the matrix decompositions. In Table 6.1, we list the different decomposition algorithms in the module. The method we use to decompose diffusion maps from `datafold` is `scipy.sparse.linalg.eigsh` [VGO+20]. It requires either an `ndarray`, a sparse matrix, or `LinearOperator` as parameters (which we use). It optionally takes the number of desired eigenvalues and eigenvectors k. The algorithm solves the eigenproblem $Ax[i] = \lambda_i x[i]$ and returns two arrays - λ for eigenvalues and k eigenvectors $X[:, i]$, where i is the column index corresponding to the eigenvalue.

`scipy.sparse.linalg.eigsh` uses the iterative Krylov-type method called *implicitly restarted Lanczos* (see Subsection 2.3.3) method to solve the system for eigenvalues and vectors. It is a wrapper for the ARPACK functions SSEUPD and DSEUPD, which implement the implicitly restarted Lanczos method to solve the system for eigenvalues and vectors [(LA07]. Although `SciPy` usually calls this for sparse settings, we overload it and use a dense interface. For us, this was the easiest way to force `SciPy` to use an iterative scheme with *matvecs*.

Table 6.1: Matrix Factorizations in `scipy.sparse.linalg`.

scipy.sparse.linalg.**eigs**	Computes eigenvalues and vectors of square matrix
scipy.sparse.linalg.**eigsh**	Computes eigenvalues and vectors of real symmetric or complex Hermitian matrix
scipy.sparse.linalg.**lobpcg**	Locally Optimal Block Preconditioned Conjugate Gradient Method
scipy.sparse.linalg.**svds**	Partial Singular Value Decompositions
scipy.sparse.linalg.**splu**	LU decomposition of sparse square matrix
scipy.sparse.linalg.**spilu**	Incomplete LU decomposition of sparse square matrix
scipy.sparse.linalg.**SuperLU**	LU decomposition of a sparse matrix

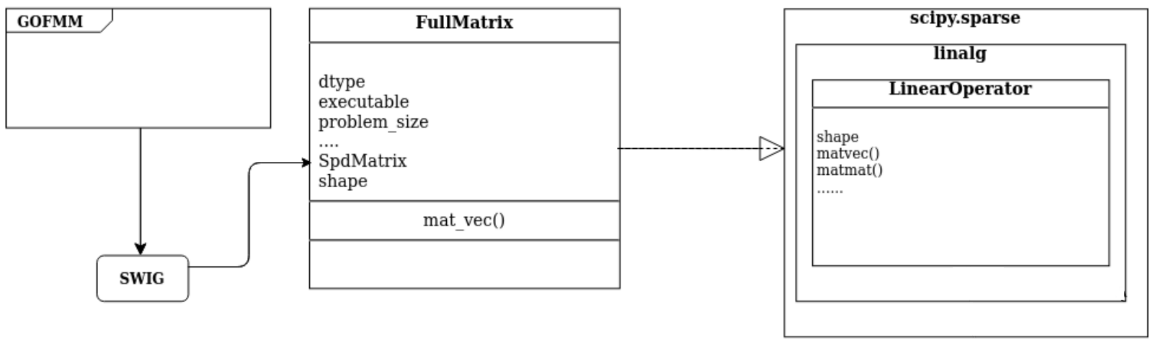


Figure 6.3: UML Diagram of `datafold` with `GOFMM` integration. `GOFMM` FullMatrix class extends the LinearOperator class and, thus, is used in the implicitly restarted Lanczos method of scipy.sparse.linalg.eigsh. The figure is joint work with Keerthi Gadameedi [Gad22].

In summary, our pipeline can analogously be extended to any other of the algorithms above. We are using the `GOFMM` fast matvecs through the `GOFMM` Python interface and apply on matrices `scipy.sparse.linalg.eigsh`. By this, we employ the implicitly restarted Lanczos method to solve the system for eigenvalues and vectors with $\mathcal{H}$-matrix approximation.

Summary

In this Subsection 6.2.2, we started with a complexity analysis showing that we can reduce the eigendecomposition complexity to $\mathcal{O}(N^2 \log N)$ with the fast `GOFMM` *matvec*. For portability, we created a Charliecloud image of the code, similar to a docker container for HPC clusters. We also discuss the function dependencies in a UML diagram for the coupling of `GOFMM` and `datafold`. Furthermore, we reviewed SciPy's LinearOperator class and explained how we can use different decomposition algorithms. We interfaced the matrix vector multiplication with `GOFMM` and used the *implicitly restarted Lanczos* using SciPy's `linalg.eigsh`. Our approach is versatile, and we could use `GOFMM` for any other iterative algorithm implemented in SciPy in a straightforward way.

6.3 Test matrices

`GOFMM` only works on SPD-matrices; this restriction, however, still includes many matrices from a wide field of scientific computing applications. Without claiming completeness, we study a potpourri of cases covering different properties in order to see benefits and pitfalls. To have full control over the properties, we assembled the matrices ourselves. In the first step in this subsection, we explain how we implement the Input/Output. Then, we dive into the generation of the synthetic test matrices (and kernel matrices from public datasets) as they were tested in [CLRB17, CRB18, YRB19]. Next, we describe the Gauss-Newton Hessians analyzed in [CRY$^+$21]. We close by explaining the matrices from diffusion maps [GRNB23].

6.3.1 Input/Output implementations

`GOFMM` supports both double and single precision; it requires binary storage format of float-after-float (4 bytes per entry, or 8 for double precision). Hence, the file size is $N \times N \times 4$ (or 8 for double precision). For the dense interface, `GOFMM` uses the `std::ifstream`, the standard input stream class, to operate on files. We then use `open`, `read`, and `close` from the beginning with the sizeof(float) $\times N^2$. We primarily use single precision since we suffice with approximate error rates in the range of 1E-3 to 1E-5, where a double precision format would not add any further accuracy.

For saving a matrix in MATLAB, we can use the file I/O with `fopen` and `fclose`, and we use `fwrite` with the option `'single'` (for writing an IEEE single 32-bit float). In Python, we use the file I/O with `open` (option `"wa"`, means write append), `write` (a for single (32-bit)/double (64-bit) float) and `close`. Note that for large matrices, we can write the matrix in batches, i.e., opening the file and closing it, writing row after row. We also experimented with writing dense Hessians from convolutional nets to memory, called `pythoncnn`[5]. In `pythoncnn`, we rely on this batched write (row-by-row) in order to allow for bigger cases.

[5]`https://github.com/severin617/pythoncnn`

6.3.2 Synthetic data and kernel matrices

This section deals with the dense interface to GOFMM, mostly generated with a MATLAB script [6]. The underlying idea is to cover a range of applications occurring in the simulation of partial differential equations, integral equations, kernels from statistical learning, or graph operators. We generated 27 matrices emulating different problems.

In Table 6.2, we list a range of *dense* matrices that were used for analysis for GOFMM. They appear in some publications [CLRB17, CRB18, YRB19], so we kept the notation as is, and some do not necessarily appear in this thesis. This covers PDE-problems (such as K01-K03, K11-18) and kernel matrices of pint clouds (K04-K10) or kernel matrices (K-) from datasets (K-MNIST, K-COVTYPE, K-HIGGS, K-SUSY). We also analyzed a few dense graph operators (G01-G05).

In detail for kernel matrices, GOFMM also offers to load data points from disk and use them *only* for the kernel evaluation. We do not use them for geometric repartitioning or sampling. In this case, every K_{ij} evaluation has a computational cost of $\mathcal{O}(1)$, the evaluation of a Gaussian kernel using points x_i and x_j.

6.3.3 Gauss-Newton Hessian

For an analysis of Hessians of neural networks, we computed dense Gauss-Newton Hessian(GNH) matrices corresponding to the weights at the end of training. For the convolutional nets (CNN), we used the Python Tensorflow script called `pythoncnn` and `fcnn`[7] using PyTorch. The MLP networks are implemented in MATLAB, and training of networks is performed with an SGD library[8] in MATLAB[9].

Let us now define the *Jacobian* of x^L (the output vector at level L) and the gradient and Gauss-Newton Hessian of F.

$$\begin{aligned} J_i &= \nabla_w F(x_i^L), \\ g &= \nabla_w \textstyle\sum_i^n f_i = \sum_i^n J_i^T q_i \\ H &= \textstyle\sum_{i=1}^n J_i^T Q_i J_i = J^T Q J, \end{aligned}$$

where J is a of dimension dn-by-N, with J_i being its i_{th} d-by-N block row and batch size n; and Q is block-diagonal matrix with n blocks of size d-by-d, each block being equal to Q_i. We remark that the Gauss-Newton Hessian is not the same as the true Hessian as in the coming Chapter 7.

Regarding the network architecture test, we are very flexible in the design of the experiments. We list the cases in Table 6.3 below. The most significant focus lay on autoencoder neural networks. Note that we chose simple examples for the GNH, but they can be extended to more intricate and much bigger networks. Evaluating the Gauss-Newton Hessian is computationally, and memory-expensive. For this purpose, joint work with Chao Chen and George Biros led to a sampling scheme for the multilayer perceptron in [CRY+21], which we do not describe here.

6.3.4 Datafold matrices

For completeness, we also list the cases for the Python interface from Section 6.5. Those are created with `numpy`, `scipy`, `scikit-learn` and `datafold`.

In Table 6.4, we list the cases; the first is used for the parameter studies, the latter two for Arnoldi-iteration for the iterative eigendecomposition.

[6] Joint work with George Biros
[7] Joint work with Chao Chen
[8] `https://github.com/hiroyuki-kasai/SGDLibrary`
[9] Joint work with George Biros

Table 6.2: Kernel matrices that are used for this thesis. As the numbering suggests, we analyzed many more cases which are not dealt with here, as they do not add any necessary new insight.

Identifier	Description
K01	Forward 2D Poisson operator
K02	2D regularized inverse Laplacian squared, resembling the Hessian operator of a PDE-constrained optimization problem. Laplacian is discretized using a 5-stencil finite-difference scheme with Dirichlet boundary conditions on a regular grid.
K03	Same setup with the oscillatory Helmholtz operator and 10 points per wavelength
K04-K06	Kernel matrices in six dimensions: Gaussians with different bandwidths, narrow and wide
K07	Kernel matrices in six dimensions: Laplacian Green's function
K08-K10	Kernel matrices in six dimensions: Quadratic, inverse quadratic, and polynomial kernel
K11	Inverse squared 1D variable coefficient Poisson problem operator
K12-14	2D advection-diffusion operators on a regular grid with highly variable coefficients
K15 & 16	2D pseudo-spectral advection-diffusion-reaction operators with variable coefficients
K17	3D pseudo-spectral operator with variable coefficients.
K18	inverse squared Laplacian in 3D with variable coefficients.
G01–G05	Inverse graph Laplacian of the **powersim**, **poli_large**, **rgg_n_2_16_s0**, **denormal**, and **conf6_0-8x8-30** graphs from UFL (`http://yifanhu.net/GALLERY/GRAPHS/search.html`)
K-MNIST	Gaussian kernel matrix with bandwidth h; 60K data points, 780D, digit recognition; see `http://yann.lecun.com/exdb/mnist/`
K-COVTYPE	Gaussian kernel matrix with bandwidth h; 100K data points, 54D, cartographic variables; see `https://archive.ics.uci.edu/dataset/31/covertype)`
K-HIGGS	Gaussian kernel matrix with bandwidth h; 500K data points, 28D, physics; see `https://archive.ics.uci.edu/dataset/280/higgs`
K-SUSY	Kernel matrix with data from high energy physics; see `https://archive.ics.uci.edu/ml/datasets/SUSY`

Table 6.3: Gauss-Newton Hessian matrices that are used for this thesis. The naming has been chosen to be consistent with the publications, e.g. [CRY+21].

Identifier	#weights	layers	Description
H02	100k	768→100→200→10	Hessian from ReLU neural network; trained to 95% accuracy on mnist [LeC98]
gnh_mnist	16.5k	768→25→10	Similarly trained on mnist [LeC98]; the numerical rank of its Hessian is small its numerical rank is low, less than 100.
gnh_cifar10	62.5k	3072→20	CIFAR-10 dataset [KH+09] from 32 × 32 RGB images. Trained with batch size 32 to about 60% accuracy. Its numerical rank to 1E-4 $\|H\|_2$ is r=1500 with n = 500; and$r$ = 4302 with n = 5000.
gnh_enco1	15k	784→10→784	Autoencoder for the MNIST data set. Here we have subsampled the 28-by-28 images
gnh_enco2	54k	3072→10→3072	Autoencoder for CIFAR-10 dataset [KH+09]
gnh_enco3	121k	3072→20→3072	Autoencoder for CIFAR-10 dataset [KH+09]

In addition, we also have scripts for a running `datafold` with a uniform distribution[10] and with swiss-roll[11], which can be found in the Github repository.

6.3.5 Parameter selection and accuracy metrics.

Similar to the SVD for arbitrary matrices, both work complexity and accuracy cannot be simultaneously guaranteed. We can control accuracy and complexity via the parameters m (leaf node size), s (maximum rank), τ (adaptive tolerance), κ (number of neighbors), *budget* (a key parameter for amount of direct evaluations and for switching between HSS and FMM) and partitioning (**Kernel**, **Angle**, **Lexicographic**, **geometric**, **random**). We use m =256–512; on average this gives good overall runtime. The adaptive tolerance τ, reflects the error of the subsampled block and may not correspond to the output error ϵ_2. Depending on the problem, τ may underestimate the rank. Similarly, this may occur in `HODLR`, `STRUMPACK`, and `ASKIT`. We use τ between 1E-2 and 1E-7, $s = m$, $k = 32$ and 3% budget. To enforce a HSS approximation, we use 0% budget. The Gaussian bandwidth values are taken from [MXT+15b] and produce optimal learning rates.

6.4 Results: GOFMM for kernel matrices

Dense SPD matrices are often the computational bottleneck of an algorithm; they appear in many areas of scientific computing. By nature, an exact dense matrix-vector multiplication is of complexity $\mathcal{O}(N^2)$, leading to a memory size limit for `DGEMM` at around $35,000 \times 35,000$ on a single "Skylake" node (see Figure 6.4). Using hierarchical approximations (tree-code) without `MPI`-parallel code `GOFMM` we theoretically have around $\mathcal{O}(N \log N)$ complexity. In this section, we show for which matrix sizes our algorithm, `GOFMM`, outperforms the conventional matrix multiplication, sacrificing accuracy. For clarity, we trade *accuracy* for *lower complexity* and for big sizes, therefore *lower runtime.*

[10]`https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html`
[11]`https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_swiss_roll.html`

Table 6.4: Datafold test matrices. Combination of the framework GOFMM, and *extends* datafold by offering a hierarchical variant for the eigendecomposition. The software setup allows reproducibility and portability.

Identifier	size	Description
py_10D_Gaussian_KDE	4096	Gaussian KDE generated with a python script for variable sizes, with numpy's cdist. Variable size of 1024×1024 up to 4096×4096
py_MNIST	8192	The MNIST database (Modified National Institute of Standards and Technology database) [LeC98] is a large database of handwritten digits that is commonly used for training various image processing systems. MNIST has a testing sample size of 10,000 and a training size of 60,000, where each sample has 784 dimensions. Kernel matrix generated with datafold and can be used up to 32,000 (due to max 60,000 images)
py_scurve	16384	3D S-curve dataset[12] is generated using scikit − learn

We tested this in Figure 6.4 on a **K04** type matrix: We multiplied a Gaussian Kernel matrix from a synthetic point cloud with a right-hand-side matrix of size $N - by - 1000$. We used only a single node in order not to defray different effects. For GOFMM, we added the one-time compression time and evaluation (multiplication). We can see that for this case GOFMM outperforms the dense exact case starting around $N = 6000$. Of course, the right-hand-side size of size $N - by - 1000$ is beneficial for this analysis, however: we can see the quadratic growth of the exact multiplication (blue crosses MKL dgemm) vs. the $N \log N$ behavior (see Figure 6.4). The break-even point for GOFMM comp+eval may be shifted depending on the matrix case or on the number of right-hand-sides. Still, the different scaling effects are becoming increasingly significant, and the benefit grows with larger matrix sizes.

Summarizing, GOFMM cannot ensure both accuracy and computational effort. However, hoping for good $\mathcal{H}$-approximability, we can assume that starting around size 6000 and 1000 right hand sides, GOFMM can *outperform* an exact multiplication. The benefit grows with larger matrix sizes.

6.4.1 Performance measurements: weak and strong scaling

Having discovered that a hierarchical matrix approximation has a break-even point, it is necessary to see what the limit on the size is and whether it scales across multiple nodes. Scaling to distributed memory is imperative for allowing larger matrices since not the whole but a smaller block needs to be stored on a node. With an exact method (dgemm, in distributed memory ScaLAPACK) almost no communication is necessary; with tree-codes like GOFMM, more nodes also mean significantly more communication overhead.

We have looked at performance measurements, including FLOP counts in [CLRB17,CRB18, YRB19]. Here, we ran experiments on the Skylake partition of SuperMUC-NG; at largest, it runs there on dense matrices of *200k* (k denotes one thousand, i.e. ,000). In the first subsubsection, we explain our *weak scaling* measurements; in the second subsubsection, the *strong scaling* results.

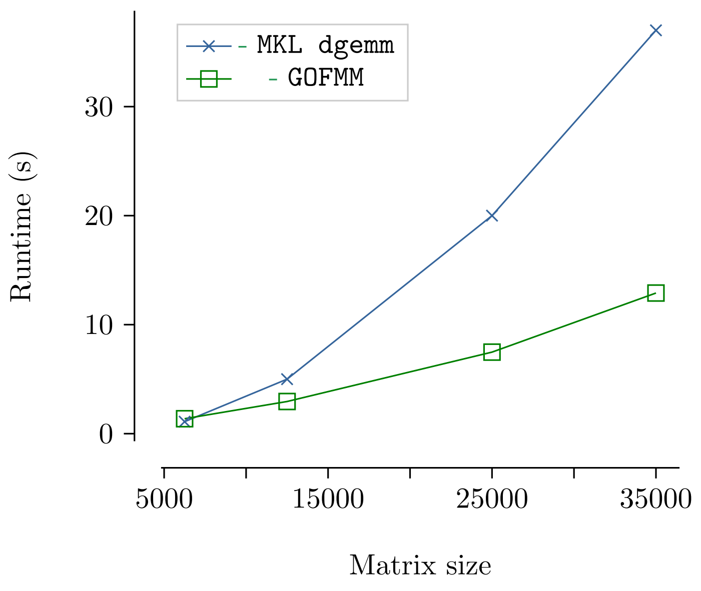


Figure 6.4: Matrix multiplication of a square $N \times N$ matrix with number of right hand sides `nrhs` $= 1000$ on a single node of SuperMUC (Skylake). Comparison of a simple C-script using `MKL DGEMM` and `GOFMM`. The *break-even* point (if we allow approximations) is at about 6000×6000 for `nrhs` $= 1000$.

Weak scaling

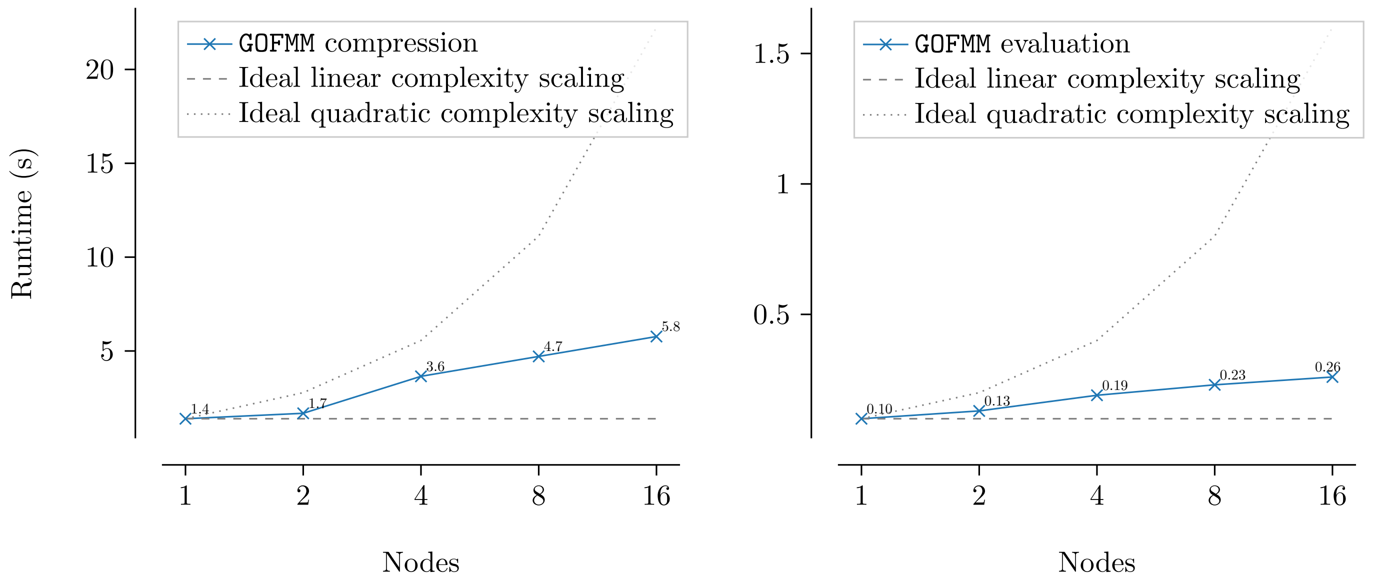
In a weak scaling run, the computational effort per core/node (computational resource) is kept constant. Hence, in linear complexity algorithms, this corresponds to a `constant` problem size per core/node. An ordinary matrix multiplication scales quadratically, so (for weak scaling) doubling the core/node would result in a $\sqrt{2}$-times global problem size in order to keep the computational work per node constant. This would result in uneven sizes, and `GOFMM` can exhibit issues for such problem sizes. In general, multiple of leaf-node-size is beneficial $m \times 2^l$ with l levels. Note that we double the problem size in the following when doubling the computation resource. This is also indicated through the dotted and dashed ideal scaling lines in Figure 6.5.

If we had perfect linear scaling, we expect this to be constant across all weak scaling runs; additional runtime would be caused by communication overhead. We assume `GOFMM` is nearly $\mathcal{O}(N \log N)$. In theory, we hope `GOFMM` to be $\mathcal{O}(N r_O)$ where r_O is the off-diagonal rank – with a certain adaptive rank selection and a certain accuracy this is increasing with problem size. Now, if we compare this to a normal matrix-matrix multiplication, having quadratic complexity, *doubling* the problem size results in 4-times the computational work, and with *doubling* compute nodes, this results in *twice* as long runtime. This is shown by the dotted "Ideal quadratic complexity scaling" line; the logarithm line is almost identical to the dashed linear complexity line.

We looked at runtimes in seconds of the one-time matrix compression and the multiplication with a vector of N-by-512 in Figure 6.5 (For parameters see[13]). For one node, the problem size comprised of 6.25k-by-6.25k. Compression took *1.4s* and evaluation (with 512 right-hand-sides) *0.10s*.

Runtime scaling of `GOFMM` compression in Figure 6.5 **(a)** ranges nearly between linear $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$. Although with this inaccurate complexity estimate, we cannot measure the parallel efficiency (and communication overhead) of `GOFMM`, it still shows us: with increasing matrix size,

[13]`GOFMM` parameters: `m` $= 768$, `s` $= 768$, `stol` $= 1E-3$, `k` $= 64$



(a) Compression time in seconds for a problem size of N-by-N roughly 6.25k per node, i.e. 6.25k-by-6.25k for 1 node and 12.5k-by-12.5k for 2 nodes.

(b) Multiplication time in seconds with a N-by-512 random right-hand-side

Figure 6.5: Weak scaling measurements of a Gaussian kernel matrices generated synthetically with 6-D point clouds with roughly 6.25k per node, i.e. 2 nodes corresponding to 12.5k-by-12.5k, 16 nodes to $\sim$ 100k-by-100k. Memory and the exact multiplication of a dense matrix scales quadratically; hence, the 2-node problem would correspond to 4 times as memory/computationally expensive (dotted). Log scale on the x-axis, linear scale on the y-axis. Next to the data cross is the runtime in seconds. Results run on Skylake partition of SuperMUC-NG. Each node has 48 cores, 16 Nodes, hence corresponding to 768 cores.

the difference between quadratic complexity and `GOFMM` is increasing. $\mathcal{H}$-matrix approximation is very *beneficial* for large matrices compared to an exact dense multiplication.

Figure 6.5 **(b)** shows the matrix-multiplication (we also call it evaluation) runtimes of `GOFMM` with increasing problem size and increasing nodes. The runtime for 1 node and a problem size of 6.25k is *0.10s*, and for 16 nodes $6.25k \cdot 16 \approx 100k$ about *0.26s*. Assuming $\mathcal{O}(N \log N)$, ideal scaling would result in $0.10s \cdot \log(16) \approx 0.12s$. Instead, *0.26s* suggests about 50% parallel efficiency.

In summary, we show weak scaling results of `GOFMM` until 16 nodes, 768 cores, and still see about 50% parallel efficiency for `GOFMM` in Figure 6.5.

The corresponding results from this plot to Figure 6.4 are 25k, corresponding to the 4-node weak scaling run in Figure 6.5. In the weak scaling run, we use roughly half as many right-hand sides, and we see *20s*, so for DGEMM on 500 `nrhs`, we assume around *10s* on a single node; `GOFMM` takes *3.8s* on 4 nodes (compression + eval time). If we extrapolate the *10s* with ideal scaling to 4 nodes, we expect around *2.5s*. Hence, this still slightly favors the optimized MKL `DGEMM` 4 node run due to the assumed ideal scaling of the `DGEMM`. However, if more right-hand sides are given (1000 as in the example of Figure 6.4), `GOFMM` can outperform `DGEMM`.

Similarly, we see at large matrices (above *25k*) `GOFMM` reaches orders of magnitude improvements against an exact `DGEMM` multiplication, which suffers from quadratic complexity and `GOFMM` has $N \log N$. Due to more communication in`GOFMM`, however, we must expect degrading parallel efficiency significantly above 16 nodes or 768 cores.

Strong scaling

In a strong scaling experiment, one leaves the problem size constant while increasing the computational resources (nodes). Hence, a constant problem size can become more challenging to scale as computational work per node diminishes while adding communication overhead.

In Figure 6.6, we show strong scaling measurements for `GOFMM` compression and evaluation for a 100k-by-100k synthetic kernel matrix. `GOFMM` can work with dense matrices of 200k starting at 2 nodes, prohibiting this for this analysis.

In Figure 6.6 on the **left** we see the one-time compression time (For parameter see[14]). For size 100k and this 6D random Gaussian kernel matrix on one node compression takes around *13s*; for multiplication with a 100k-by-512 vector (right) *1.35s*. The parallel efficiency starts from our baseline 1-node (100%), ranging until 4% in compression and 11% in evaluation on 128 nodes (6144 cores). Especially for compression, there is no runtime gain beyond 16 nodes. Runtime for 128 nodes is similar to runtime for 16 nodes. Having a limit on maximum acceptable efficiency is not unusual for a parallel code; also, increases in runtimes are possible as communication times are increasing. For this reason, we highlight multiplication runtime scaling [CRB18] and accept the one-time compression cost.

In Figure 6.6 on the **right**, we also see diminishing parallel efficiency up to 128 nodes. With 16 nodes, we see about 52% parallel efficiency reported, which means a runtime about 8 times smaller than on a single node. Interestingly, with theoretic calculations (assuming $N \log N$) complexity in weak scaling, we compute a similar number. Note that for the weak scaling analysis, we also have a 100k matrix for 16 nodes.
Especially above 64 nodes, the matrix multiplication evaluation time also does not decrease with more resources; still, with 64 nodes, our runtime is 16 times lower than on a single node.

In summary, we performed strong scaling measurements of `GOFMM` up to 128 nodes of SuperMUC (Skylake), see Figure 6.6. We see adequate scaling up to 16 nodes, and with a dense matrix of 100k, significant runtime benefits over theoretic dense exact multiplication.

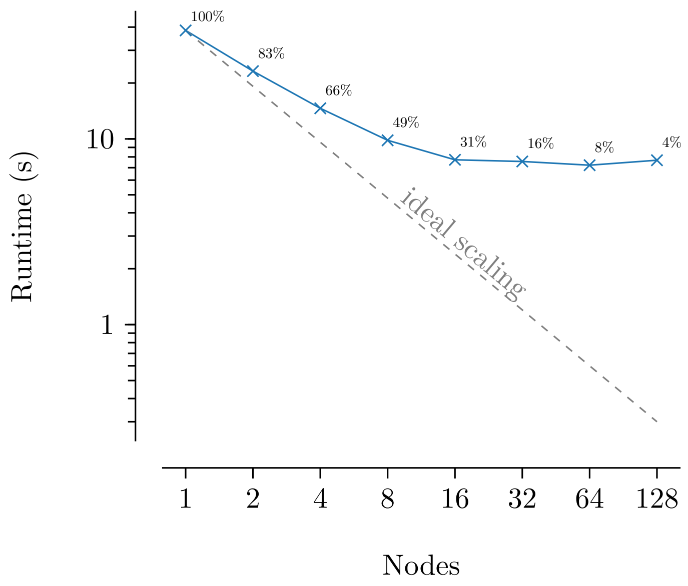
6.4.2 Timing comparison to ScaLAPACK and STRUMPACK

While good scaling results are vital to show good efficiency of the method and its implementation, it does not provide any details on whether other methods are superior. We here at least compare `GOFMM` to an exact `ScaLAPACK` (distributed version of a `DGEMM`) and the structured matrix package `STRUMPACK` (STRUctured Matrices PACKage) [RLGN16], which to our knowledge is the most competitive distributed memory software for dense rank-structured matrices.

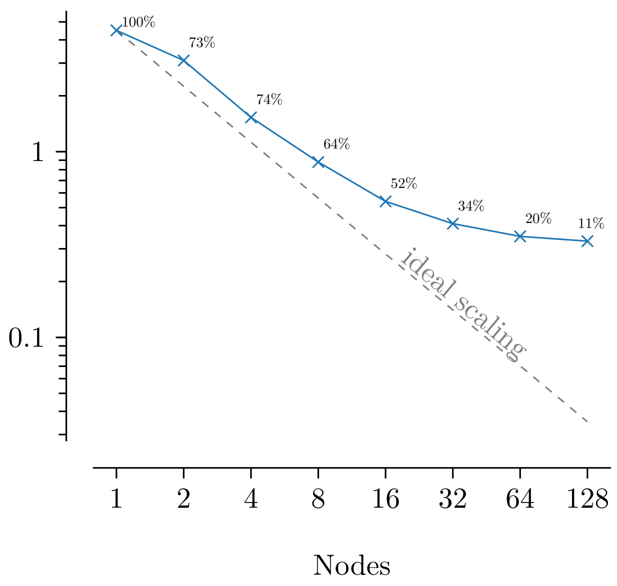
In Table 6.5, we compare overall wallclock timings for ScaLAPAck, `STRUMPACK`, and `GOFMM`. For a generic setup, we used 4 Skylake nodes of Stampede2. We multiplied several test matrices (**K**** have size 102k, **G03** 89k, **H02** 98k) with a matrix of size N-by-2048. While the runtimes of `ScaLAPACK` are independent of the case (no approximation), `STRUMPACK` and `GOFMM` vary significantly depending on the matrix structure. `STRUMPACK` stuggles with almost all of the cases[15], while `GOFMM` outperforms both others for all kernel matrices (**K****) and the graph Laplacian **G03**. With the Hessian (H02) it also struggles, and there a dense exact multiplication is faster.

This comparison is not to show general superiority to `STRUMPACK`; it simply lays out a few examples where `GOFMM`'s method proves to be better and shows that `GOFMM` is very competitive. The matrix K** is ordered randomly, as the points are randomly generated. `GOFMM` with the Gram distance metric seems to be beneficial, whereas `STRUMPACK` does not reorder, which seems

[14] `GOFMM` parameters: `m` $= 768$, `s` $= 768$, `stol` $= 1E - 3$, `k` $= 64$
[15] For `STRUMPACK` we used the lexicographic ordering



(a) Compression time in seconds up to leaf node size of 768. Non-ideal scaling due to diminshing parallelism close to root.



(b) Multiplication time in seconds with a 100k-by-512 random right-hand-side

Figure 6.6: Strong scaling measurements of a Gaussian kernel matrices generated synthetically with 6-D point clouds, all roughly of size 100k-by-100k. Next to the data cross, we annotate the parallel efficiency in percent. Results run on Skylake partition of SuperMuc-NG. Each node has 48 cores, 128 Nodes hence corresponds to 6144 cores.

to be not very beneficial. We are optimistic that for their testsuit with Poisson/convection, it is state-of-the-art [GLR+16]. However, we have no access to it for comparison. The ultimate goal is simply to provoke other ideas, solicit feedback, and give alternatives.

Table 6.5: Comparison between `ScaLAPACK`, `STRUMPACK`, and `MPI-GOFMM` for several different matrices. All results are on four Stampede-2 nodes. All the matrices are roughly 100k-by-100k. We report the time in seconds to do a dense matrix-matrix multiplication with ScaLAPACK (no approximation), and then the accuracy, compression time, and multiplication time for `STRUMPACK` and `MPI-GOFMM` respectively. The error ϵ_2 is defined in both cases from a rough Froebenius norm, see Section 6.1.

	`ScaLAPACK`	`STRUMPACK`			MPI-GOFMM		
case	multiplication	ϵ_2	compression	multiplication	ϵ_2	compression	multiplication
K04	4.1	0.17	224	7.5	1.7E-05	1.69	0.09
K07	4.0	0.02	15.8	3.3	1.0E-04	1.45	0.04
K11	4.1	0.01	93.6	2.2	7.9E-06	1.52	0.04
K12	4.1	0.11	222	4.3	6.6E-05	1.64	0.05
G03	2.8	0.10	**33.5**	2.1	7E-05	**1.44**	0.04
H02	3.9	0.09	**19.6**	5.3	7.0E-04	**11.65**	0.57

6.4.3 Gauss–Newton Hessian matrix

With the evolution of deep learning, training time takes up increasing portions of computing time globally. If this could be sped up only slightly, this would have a significant outcome. Hence, we propose the research question of whether matrix approximation can be applied to deep learning Hessians. Our goal here is to empirically show a pattern that for these small academic networks, `GOFMM` can compress Gauss-Newton Hessians, that in the future it can be used for model analysis, and if surfaces are smooth enough, be used in second-order optimization.

Hence, we applied `GOFMM` to several Gauss-Newton Hessian matrices in the following. The most prominent method in the current deep learning second-order community is KFAC [MG15], which resembles a block-diagonal Fisher matrix. It avoids setting up the Hessian and approximates it with Kroenecker products. On the one hand, this study can be seen as orthogonal to KFAC, simply trying to find structure, allowing for approximations. On the other hand, we also observed that many training Hessians have significant rank and thus cannot be compressed globally – by a single SVD – and a hierarchical treatment is necessary (hierarchical off-diagonal low-rank).

There are several settings and parameters in `GOFMM` to control accuracy and computational effort. The leaf node size `m` steers the depth of the binary hierarchical tree structure. Variable `s` is the maximum hierarchical compression rank, where `stol` is used as a local tolerance measure. Variable `k` nearest neighbors are used for sampling (in `FMM` also for sparse corrective terms). For clarity, and order to not distract the reader, we defined static test conditions that we used for *forward multiplication* and approximate *HSS-factorization* computation [16].

In the multiplication case, we distinguish between `HSS` and `FMM`. *Hierarchically Semi-Separable* (`HSS`) approximation ($H = D + UV$) use no sparse corrective terms, whereas in `FMM` the sparse correction terms $H = D + UV + S$ are included. The compression effort is similar between `HSS` and `FMM`; for the evaluation phase, `GOFMM` has a parameter for the maximum percentage of sparse corrections. Naturally, we allow 0% and 10% sparse corrections for `HSS` and `FMM`, respectively.

Measurements in Table 6.6 and Table 6.7 are obtained from *four* nodes of "Stampede 2" resulting in a total of 192 cores.

Matrix-Vector Multiplication

We analyzed several Gauss-Newton Hessians (GNH) from exemplary deep learning networks (implementation details can be found in Section 6.3). Our results here evolved in the course of work in [CRY+21]. In addition to the `GOFMM` analysis of GNHs here, some aspects of the publication also involve sampling, which we do not describe here[17].

The Table 6.6 shows the multiplication results of GNHs of variable sizes N. We differ between parameters `HSS`/`FMM` and `low` of `high` accuracy. At first, GNHs are compressible, and we find for all cases suitable parameter settings to achieve 3 digits of accuracy ($\epsilon_F \leq 1.3E-3$). We also see that the `HSS` format performs quite well for almost all cases and we see that the `FMM` format sometimes helps 1 digit in accuracy ϵ_F.

In general, better approximations (smaller errors) can be obtained through either `high` accuracy configuration; additionally, including `FMM` format with sparse corrections (more storage) helps slightly. Notice that this setting keeps full diagonal blocks (due to the assumption) and compresses only off-diagonal blocks, so further compression could be achieved if diagonal blocks are also compressed. `FMM` requires more accesses to the matrix, which we denote with %K as the percentage of entries accessed from the matrix. As described above, we usually limit this to around 10% with FMM; however, due to uneven neighbor/leaf distributions, this lies somewhere between 20-30% for `FMM`.

Let us look at the biggest case, 20 in Table 6.6: Compared to Table 6.5 we see similar

[16]`GOFMM` parameters:
`low` accuracy: `m` $= 128$, `s` $= 128$, `stol` $= 1E-3$, `k` $= 64$
`high` accuracy: `m` $= 1024$, `s` $= 2048$, `stol` $= 1E-5$, `k` $= 64$

[17]In collaboration with Chao Chen

compression and multiplication runtimes (one higher, one lower due to desired accuracy). GNHs are a bit harder to compress than Radial Basis Function kernel matrices. Nevertheless, if multiplied to many right hand sides, as e.g. in an iterative solver, it can be beneficial.

Our method targets the regime in which the rank is significant ($\text{rank}(H) \geq \frac{1}{4}N$) and hence H is *not globally* low-rank, but H is nearly *hierarchical off-diagonal low-rank*. In summary, we see that $\mathcal{H}$-matrix-arithmetic allows compression of GNHs with about 3-5 digits of accuracy. In addition, we reported in [CRY$^+$21] that $\mathcal{H}$-matrix-approximation performs much better than global low-rank (randomized SVD)[18].

Table 6.6: Timing and approximation of full GNH matrices that are precomputed with 1000 data points ($n = 1000$ in Eq. (6.3.3)). $\mathcal{H}$-matrix approximations are computed with low- and high-accuracy settings, as well as `FMM/HSS` format. The HSS format with low accuracy is usually the fastest scheme with the most compression. Measurements are run on 4 nodes of Stampede2, resulting in 192 cores.

			Parameters		GOFMM results			
#	net	N	`scheme`	`acc`	t_{Comp} (in s)	t_{Mult} (in s)	%K	ϵ_F
1	gnh_mnist	16.5k	`HSS`	`low`	0.44E0	0.02E0	0.4%	2.3E−2
2			`HSS`	`high`	1.81E0	0.07E0	4.78%	1.6E−2
3			`FMM`	`low`	0.48E0	0.08E0	19.17%	1.6E−2
4			`FMM`	`high`	2.84E0	0.08E0	23.98%	1.4E−4
5	gnh_cifar10	62.5k	`HSS`	`low`	1.20E0	0.03E0	0.60%	6.2E−2
6			`HSS`	`high`	26.29E0	0.26E0	9.46%	2.4E−3
7			`FMM`	`low`	1.34E0	0.57E0	29.06%	1.5E−2
8			`FMM`	`high`	27.16E0	0.91E0	35.74%	1.3E−3
9	gnh_enco1	15k	`HSS`	`low`	0.62E0	0.03E0	2.1%	1.6E−1
10			`HSS`	`high`	8.93E0	0.13E0	27.3%	4.4E−4
11			`FMM`	`low`	0.59E0	0.04E0	24.5%	1.2E−1
12			`FMM`	`high`	8.99E0	0.12E0	29.2%	5.6E−4
13	gnh_enco2	55k	`HSS`	`low`	0.23E0	0.02E0	0.48%	1.4E−1
14			`HSS`	`high`	9.34E0	0.16E0	3.53%	1.4E−4
15			`FMM`	`low`	1.14E0	0.21E0	18.2%	1.2E−1
16			`FMM`	`high`	14.50E0	0.33E0	21.3%	9.4E−5
17	gnh_enco2	121k	`HSS`	`low`	2.81E0	0.04E0	0.25%	1.4E−1
18			`HSS`	`high`	37.91E0	0.63E0	3.42%	8.8E−5
19			`FMM`	`low`	2.92E0	1.20E0	20.02%	9.2E−2
20			`FMM`	`high`	36.94E0	2.02E0	31.4%	1.5E−4

Linear Solver for HSS

A GNH often appears in Newton's equation, where it is required to solve a system of linear equations. In the previous subsection, we mentioned that a fast hierarchical matrix multiplication can be used in an iterative solver since it involves matrix products. In this section, we look at an approximate solver, sometimes so coarse to be called a preconditioner. Using $\mathcal{H}$-matrix-arithmetic, there is a numerically cheap way to compute the (approximate) inverse of the GNH. However, as typical for (approximate) inverses, this may be numerically instable.

[18]In collaboration with Chao Chen

In this section, we look at HSS inversion using GOFMM; in Table 6.7, we have tabulated the factorization time, fraction of accessed entries %K, and error ϵ_F. The error measures ϵ_F are computed by a sampled normed error $H * H^{-1}$ and the identity matrix.

We observe that the low accuracy HSS schemes provide reasonable error measures for multiplication (see previous Table 6.6); however, for the factorization (Table 6.7) the errors for low accuracy are quite off (see # $\geq 1E+2$) (23, 25, 27, 29). It seems that we see numerical instabilities. The high accuracy schemes provide reasonable accuracies (usually $\leq 1E-3$ in 22, 26, 28, 30). The factorization times for the high accuracy runs, however, are orders of magnitudes higher (runs 22, 24, 26, 28, 30). This cost is a serious concern and may prohibit GNH treatment; however, if one keeps the GNH constant for a few steps or only uses it for smoothing, this cost may be acceptable.

Table 6.7: Timing and preconditioning for small networks, where full GNH matrices are used. The HSS format with high accuracy leads to good preconditioners. Measurements are run on 4 nodes of Stampede2, resulting in 192 cores.

			Parameters		GOFMM results		
#	net	N	scheme	acc	t_{Fact} (in s)	%K	ϵ_F
21	mnist	16.5k	HSS	low	0.06E0	0.41%	8.9E−4
22			HSS	high	2.48E0	5.1%	1.2E−6
23	cifar10	65.5k	HSS	low	0.25E0	0.59%	7.2E+4
24			HSS	high	32.45E0	9.4%	3.5E−1
25	enco	15k	HSS	low	0.06E0	2.1%	2.6E+3
26			HSS	high	7.79E0	27.3%	8.8E−4
27	enco	54k	HSS	low	0.07E0	0.45%	2.4E2
28			HSS	high	11.60E0	4.3%	8.1E−5
29	enco	121k	HSS	low	0.25E0	0.25%	4.3E+3
30			HSS	high	38.15E0	3.5%	1.1E−4

6.4.4 Summary of GOFMM results on SPD matrices

As a motivation, we began with the quadratic complexity of matrix-vector multiplication. While the break-even point of a hierarchical compression depends on the problem setting and the number of right-hand-sides, we reported for a Gaussian kernel matrix and 1000 right-hand-sides. We saw a break-even at a matrix size of 6k, and with the quadratic increase, the runtime difference between exact and hierarchical becomes tremendous.

Next, we showed weak and strong scaling results for matrices with a maximum size of $100k \times 100k$. With a sweet spot at around 16 nodes, we get around 30% parallel efficiency for the one-time compression, while we get 50% for the multiplication. GOFMM shows, especially for iterative multiplication, huge benefits. Additionally, GOFMM outperforms the exact ScaLAPACK, and the structured matrix package STRUMPACK for several cases (Table 6.5).

We continue with a a modern neural network application, namely Gauss-Newton Hessians (GNH). We saw that a hierarchical semi-separable (HSS) or a fast multipole method (FMM) can compress several pre-trained GNHs. For multiplication, we easily achieve several digits of accuracy, whereas for factorization, some numerical instabilities occur for low accuracy approximations.

In summary, we saw several benefits of $\mathcal{H}$-matrix-arithmetic on **multiplication** and **factorization** of several interesting benchmark problems. While GOFMM (with the geometric-oblivious distance notion) shows competitive results, we do not claim completeness. There is active research on several $\mathcal{H}$-matrix-schemes for other applications, like structured matrix package `STRUMPACK` (STRUctured Matrices PACKage) [RLGN16] or the most recently published ULV factorization by Deshmukh, Yokota et al. [DYBM23].

6.5 Results: Python interface

In addition to the runtime measurements of the C++ implementation, we implemented a *Python interface* to allow users to call `GOFMM` directly on numpy matrices. This primarily serves the purpose of quick prototyping and error checking. We implemented this with the simple wrapper interface generator called `SWIG` (see Subsection 6.2.1 for details). We also offer this in a separate container for reproducibility.

The experiments were performed on the CoolMUC-2 cluster of the Leibniz Supercomputing Centre[5]. It has 812 28-way Intel Xeon E5-2690 v3 ("Haswell") based nodes with 64GB memory per node and FDR14 Infiniband interconnect.

We start with a study on the impact of the `GOFMM` parameters, mainly `leafSize` and approximation `rank`. For the matrix multiplication, this is covered in Subsection 6.5.1, for the pseudo-inverse in Subsection 6.5.2.

Secondly, we cover the combination of `GOFMM` and `datafold` in Subsection 6.5.3, results we published in [GRNB23]. Here, we use `GOFMM` for an eigendecomposition, i.e. we use the matrix product of `GOFMM` to perform an iterative Arnoldi-based eigendecomposition. We start with the scurve dataset, where we compare the result of `scipy`'s exact product with the `GOFMM` approximate one. Then, we cover the MNIST dataset. At the end, as always, we close this section with a summary.

6.5.1 Accuracy measurements of the matrix-vector multiplication

In this subsection, we use a kernel density estimation case with a Gaussian radial basis function. We use 2D data points, as it is suitable for `GOFMM` and it is visually representative.

The inputs are randomly generated from a 2D normal distribution, and depicted in Figure 6.1. Our goal is to reassemble these data points *geometry-oblivious* and classify them according to their neighboring density. As usual, first, we do the compression using `skeletonize`. Second, we use the fast matrix-vector multiplication using `evaluate` directly from Python by calling the SWIG-generated function.

In Table 6.8, we tabulate the root squared errors from multiplication with a randomly drawn vector (normal distribution) of size $N \times 1$.[19] We use moderate size matrices of 1024, 2048, 4096. Rank means the maximum off-diagonal compression rank. If we allow only a maximum rank of 8, `GOFMM` always fails to provide adequate accuracy.

We also see in the table that with a higher allowed maximum rank, we achieve the desired accuracy. This can also be achieved with small `leafSizes`, i.e. the tree is deep.

In summary, for a matrix-vector multiplication `GOFMM` is fairly accurate, and hence, we can confirm correctness and suggest its use. We used the same interface for the eigendecompositions in Subsection 6.5.3.

6.5.2 Factorization using the pseudo-inverse

Similarly, we also checked the accuracy in the Python interface for the pseudo-inverse, which we also briefly covered in Subsection 6.4.3.

[19] For other settings, we use `k`=64 and `stol`= 1E-5.

Table 6.8: Root squared error of MATVEC on a 2D Gaussian KDE. Blank results for 1024 leaf size for the first set (size 1024), as this would mean no compression.

Size: 1024 x 1024						
`leafSize` / `maxRank`	8	32	128	256	512	
8	3.48	1.85	1.68	1.39	1.08	
32	1.18E-4	7.78E-5	6.44E-5	6.23E-5	1.28E-4	
128	6.00E-5	5.81E-5	6.30E-5	5.85E-5	8.60E-4	
256	6.47E-5	6.64E-5	6.38E-5	5.57E-5	7.08E-4	
Size: 2048 x 2048						
`leafSize` / `maxRank`	8	32	128	256	512	1024
8	3.94	1.93	1.73	1.58	1.44	8.54E-1
32	1.54E-4	8.24E-5	7.80E-5	7.55E-5	1.16E-4	1.46E-4
128	7.79E-5	7.85E-5	7.46E-5	7.44E-5	7.98E-4	9.15E-5
256	8.03E-5	8.24E-5	7.94E-5	8.07E-5	8.08E-5	8.83E-5
512	8.18E-5	7.99E-5	8.56E-5	8.09E-5	8.34E-5	8.86E-5
1024	7.55E-5	7.54E-5	7.60E-5	7.56E-5	7.73E-5	8.32E-5
Size: 4096 x 4096						
`leafSize` / `maxRank`	8	32	128	256	512	1024
8	3.58	1.93	1.82	1.78	1.65	1.29
32	9.29E-5	1.12E-4	8.73E-5	8.45E-5	1.27E-4	1.25E-4
128	8.53E-5	8.58E-5	8.75E-5	8.42E-5	8.94E-5	8.77E-5
256	8.81E-5	9.21E-5	8.85E-5	8.54E-5	8.61E-5	8.43E-5
512	8.62E-5	8.52E-5	8.61E-5	8.50E-5	8.65E-5	8.84E-5

Table 6.9: Root squared error of the approximate inverse on 2D Gaussian KDE

Size: 1024 x 1024						
maxRank \ leafSize	8	32	128	256	512	
8	3.75E-2	1.85E-2	1.57E-2	1.49E-2	1.21E-2	
32	3.61E-5	3.60E-5	4.00E-5	5.16E-5	7.09E-5	
128	6.43E-5	6.55E-5	6.61E-5	6.60E-5	7.41E-5	
256	3.31E-4	3.31E-4	3.31E-4	3.31E-4	3.31E-4	
Size: 2048 x 2048						
maxRank \ leafSize	8	32	128	256	512	1024
8	5.93E-2	2.71E-2	2.22E-2	1.84E-2	1.57E-2	1.32E-2
32	3.76E-5	3.67E-5	4.06E-5	5.30E-5	7.33E-5	1.04E-4
128	6.87E-5	6.97E-5	6.99E-5	6.93E-5	7.73E-5	1.01E-4
256	9.38E-5	9.41E-5	9.28E-5	9.33E-5	9.23E-5	1.03E-4
512	1.28E-4	1.27E-4	1.29E-4	1.28E-4	1.29E-4	1.29E-4
1024	6.55E-4	6.55E-4	6.55E-4	6.55E-4	6.55E-4	6.55E-4
Size: 4096 x 4096						
maxRank \ leafSize	8	32	128	256	512	1024
8	8.58E-2	4.00E-2	3.21E-2	3.02E-2	3.06E-2	2.21E-2
32	3.73E-5	3.74E-5	4.14E-5	5.37E-5	7.25E-5	1.05E-4
128	7.15E-5	7.07E-5	7.14E-5	7.03E-5	7.89E-5	1.01E-4
256	9.87E-5	9.89E-5	9.87E-5	9.87E-5	9.95E-5	1.07E-4
512	1.33E-4	1.33E-4	1.33E-4	1.33E-4	1.33E-4	1.33E-4

In Table 6.9, we again varied `leafSize` and the maximum approximation rank. We measured the root mean squared error with the Froebenius norm to $||K * K^{-1} - I||_F$. Surprisingly, we are not so much affected by the numerical instabilities of the inverse as they occurred in Subsection 6.4.3. In general[20] the lower ranks or `leafSize`s seem to not interfere too much with the errors. The errors range from 1E-2 to 1E-5, with the lower errors more towards larger maximum ranks.

An error of around 1E-4 for an inverse is relatively low, and we can confirm the correctness of the pseudo-inverses.

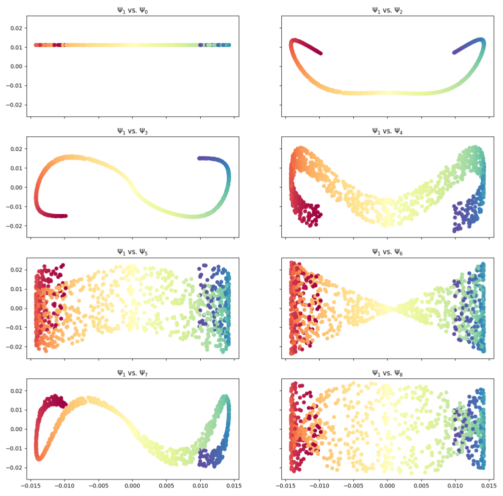
6.5.3 Eigenvector decomposition for datafold

In the following, we observe an interesting application: in the diffusion maps algorithm (Algorithm 2.1), the eigendecomposition is the computational bottleneck. Our goal is to use $\mathcal{H}$-matrix-arithmetic to reduce this for a dense matrix.

In this subsection, we used `GOFMM`'s matrix-vector multiplication for an iterative eigendecomposition. As described in Subsection 6.2.1, we overloaded the LinearOperator class with GOFMM's multiplication.

We start with the 3D S-curve dataset and show then the results for MNIST. These results are also published in [GRNB23].

[20]Remember `stol`=1E-5


(a) `scipy`

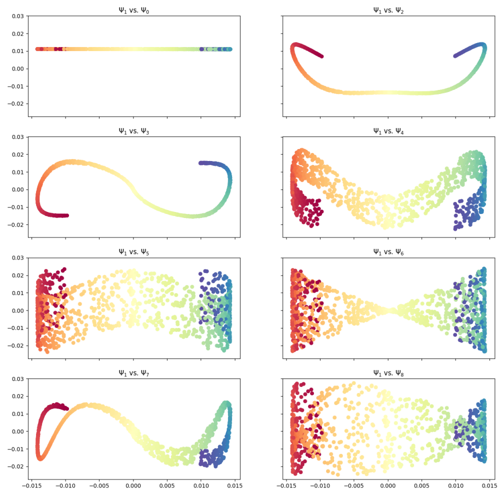

(b) `GOFMM`

Figure 6.7: Eigenvector comparison for the S-curve dataset

S-curve

A 3D S-curve dataset[21] is generated using `scikit-learn` [PVG+11] with 16384 points in the dataset. A 3D S-curve has an underlying intrinsic dimension of 2, and we apply a diffusion maps algorithm to compute this. Since our focus lies in the eigendecompositions of the kernel matrix, eigenpairs are computed using two solvers. The first set of values is computed using the `scipy` solver, and these are taken as reference values. The approximations of our `GOFMM` `matvec` implementation are computed [22]. We computed the difference between the exact and the GOFMM eigenvalues in the Froebius norm. For this case, the overall error is observed to be in the range of $9E-4$.

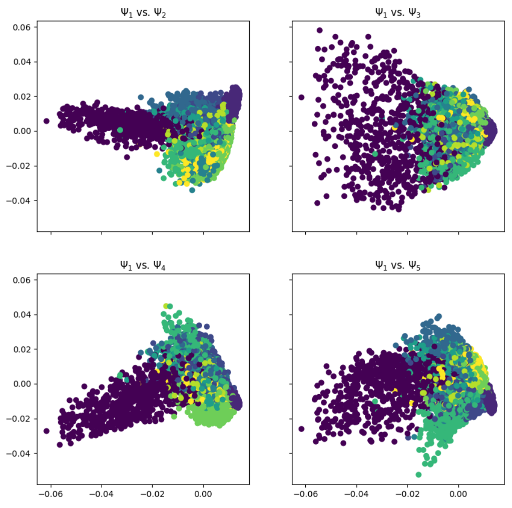
We can compare the embeddings obtained from both solvers by fixing the first non-trivial eigenvector and comparing it to the other eigenvectors. Eigenvector comparison for both `scipy` solver and `GOFMM` can be observed to be qualitatively very similar in Figure 6.7.

MNIST

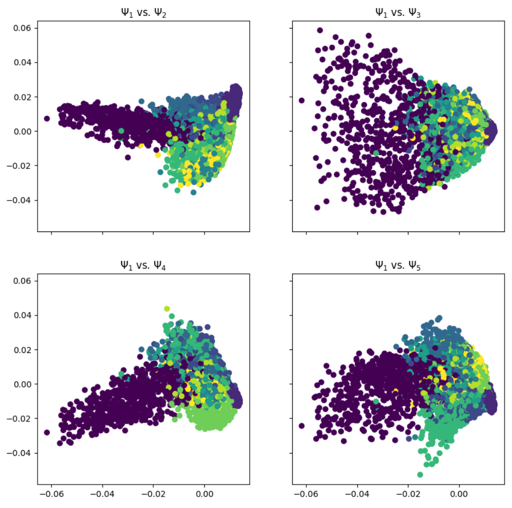
The MNIST database (Modified National Institute of Standards and Technology database) [LeC98] is a large database of handwritten digits that is commonly used for training various image processing systems. MNIST has a testing sample size of 10,000 and a training size of 60,000 where each sample has 784 dimensions.
For a matrix size of 8192, the eigenvector comparison[23] for both solvers look qualitatively similar as visible in Figure 6.8. The Frobenius norm of the difference of the first five eigenvalues is also in the range of $1E-4$.

The parameters required to obtain the results we obtained in the course of the work show that the approach is very problem-dependent. If we use a `low`-accuracy *matvec*, we run into

[21]`https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_s_curve.html`
[22]`GOFMM` parameters: $\texttt{m} = 512$, $\texttt{s} = 512$, $\texttt{stol} = 1E-7$
[23]`GOFMM` parameters: $\texttt{m} = 512$, $\texttt{s} = 512$, $\texttt{stol} = 1E-7$



(a) `scipy`



(b) `GOFMM`

Figure 6.8: Eigen vector comparison for MNIST

divergence of the eigenvalues. As already mentioned in [CLRB17], problems with very dense kernel matrices are well-suited for hierarchical approaches.

6.5.4 Summary of GOFMM Python integration using SWIG

In summary, we implemented Python bindings of the `C++` codebase using `SWIG`. We performed accuracy measurements of the matrix-vector multiplication (*matvec*, see Subsection 6.5.1) and factorization (see Subsection 6.5.2).

Firstly, we varied the leaf node size and the maximum approximation rank for 2D synthetic Gaussian kernel matrices. We saw that the maximum allowed rank has the most significant influence, and also, a very deep tree, i.e., a small leaf node size, allows for errors of around 1E-4. We expected good accuracies for the *matvec* but also observed good factorization accuracies in the same range.

Secondly, in Subsection 6.5.3, we applied the *matvec* for an iterative eigendecomposition for the application `datafold`, a diffusion maps code. We compared its eigenvectors qualitatively to `scipy`'s exact multiplication and saw no significant difference for the datasets S-curve and MNIST. Quantitatively, we measured the Froebenius norm on the first 5 eigenvectors, reaching accuracies of 9E-4 and 1E-4, respectively.

In summary, we offer a Python version of the `GOFMM` C++ code frame - which is versatile and can be used for prototyping and finding other application cases. It can also be used in a container, where we employed Charliecloud for usage on LRZ's Linux Cluster.

6.6 Summary

In summary, we started in Section 6.1 by explaining the methods and the implementation of `GOFMM`. In addition, we cover the Python bindings using `SWIG`, as well as the overloading of the `LinearOperator` class to be used in iterative `scipy` algorithms, in particular in conjunction with datafold. Also, we explain the test matrices ranging from synthetic data and kernel matrices to Gauss-Newton Hessians from neural networks to diffusion maps matrices.

In a second step (Section 6.4), we see that GOFMM with $\mathcal{O}(N \log N)$ behavior can be significantly faster for matrices above 6.5k than a classical exact scheme, like MKL's DGEMM. We perform weak and strong scaling measurements on LRZ's SuperMUC-NG up to 128 nodes (6144 cores). On islands of 16 nodes, we reach about 30% parallel efficiency for the one-time compression and around 50% for multiplication. We also compare against a popular $\mathcal{H}$-matrix-arithmetic code called STRUMPACK and the distributed exact version called ScaLAPACK. We observed for most cases – from several kernel matrices over graph Laplacians – GOFMM shows significantly *lower runtime* and better *accuracy* compared with STRUMPACK.

We continue with Gauss-Newton Hessians (GNH), where we saw that a hierarchical semi-separable (HSS) or a fast multipole method (FMM) can compress several pre-trained GNHs. For multiplication, we easily achieve several digits of accuracy, whereas for factorization, some numerical instabilities occur for low accuracy approximations.

In Section 6.5, we checked the correctness of *multiplication* and *factorization* using a parameter study, where we varied GOFMM parameters. We continue with analyzing matrices from diffusion maps. We applied GOFMM's *matvec* in an iterative eigendecomposition for the application datafold, reaching above 1E-3 accuracy.

In summary, we described and analyzed GOFMM, showing benefits in *accuracy* and *lower runtime* and their interplay. Those benefits become more significant for large matrices. Including Python bindings, with this work, we show several interesting applications, such as kernels, Hessians, and diffusion maps.

Intelligence is the ability to avoid doing work,
yet getting the work done.
Linus Torvalds (*1969)

7 2^{nd}-order optimizer for artificial neural networks

This thesis consists of *two* major result chapters: The first is Chapter 6 $\mathcal{H}$-matrix, the second is Chapter 7 2^{nd}-order optimizer for Artificial Neural Networks. A 2^{nd}-order scheme is beneficial since it promises local quadratic convergence in the vicinity of a minimum (we often relax this to superlinear convergence).

2^{nd}-order algorithms for neural networks *were infeasible* due to quadratic growth of memory requirements of setting up the Hessian with growing number of network parameters; with *this* novel tunable algorithm called `Newton-CG` we *enable* a second-order scheme *approximately* by only requiring the Hessian-matrix vector product.

`Newton-CG` requires about twice as much compute work as an ordinary gradient descent in the best case (1 CG-iteration). A 2^{nd}-order optimizer increases the computational intensity on data (memory), i.e. flops per byte. Since many data algorithms are memory-bound, additional computations (as required by `Newton-CG`) are “free”. Hence, this extra computing work often does not increase the time to solution.

We used existing and prominent deep learning models and our newly implemented versatile optimizer `Newton-CG` (with many features) on them.

In this Chapter 7, the *implementation methods* and *results* of the second-order neural network optimization are explained. We start with the implementations, including variants of the optimizer `Newton-CG` and the network architectures using an auto-differentiation framework; we also explain the datasets (Section 7.1). We continue with *results* for a *global convergence analysis*, where we show convergence plots for various neural network architectures and observe a very *problem-dependent* behavior (Section 7.2). Then, we employ data-parallelism and analyze *results* for *parallel* performances for `Newton-CG` (Section 7.3). We close this chapter with a summary in Section 7.4.

7.1 Implementation: Newton-CG

Recall the Tikhonov-regularized Newton-Raphson equation

$$(H_L(\mathbf{W}_k) + \tau I)d_k = -\nabla L(\mathbf{W}_k)$$

for the network loss function $L : \mathbb{R}^n \to \mathbb{R}$. $\mathbf{W}_k$ is the current iterate vector of network weights, ∇L the gradient of the loss function and H_L its Hessian; the goal is to solve for the update vector d_k, resulting in $\mathbf{W}_{k+1} = \mathbf{W}_k + \alpha d_k$ with learning-rate α. Many Hessians are not symmetric positive-definite, so we use a Tikhonov regularization that adds a scaled identity matrix, i.e. $H+\tau I$. The size of the equation system is $\mathbb{R}^{N\times N}$, with the number of weights N, which becomes infeasible to store with state-of-the-art weight parameter ranges of ResNets (or similar).

Instead of allocating the dense Hessian matrix, we use Hessian information only through the fast *matvec* with the Pearlmutter trick (see Subsection 4.2.1). To solve the regularized Newton's equation (see also Equation 4.4 from the theory section), we use the iterative solution method conjugate gradients (CG). Such a method is often called Newton-CG; when referring to our code we use `Newton-CG`, except for headings. Our `Newton-CG` method is summarized in Algorithm 7.5.

Algorithm 7.5 `Newton-CG`

```
Require: $\mathbf{W}_0$:   Starting point
Require: $\tau$:   Tikhonov regularization/damping factor
1: $k \leftarrow 0$
2: while $\mathbf{W}_k$ not converged do
3:     $k \leftarrow k + 1$
4:     $p_k \leftarrow$ CG$((H + \tau I), -\nabla L(\mathbf{W}_k))$      ▷ Tikhonov: Approx $(H + \tau I) p_k = -\nabla L(\mathbf{W}_k)$
5:     if $\nabla L(\mathbf{W}_k)^\top p_k > \gamma$ then $p_k \leftarrow -\nabla L(\mathbf{W}_k)$      ▷ Armijo: Feasibility check.
6:     end if
7:     $\alpha_k \leftarrow \alpha$      ▷ Compute or use a given step size.
8:     $\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} + \alpha_k p_k$
9: end while
```

Note that we added the Armijo feasibility check. If the conditions are fulfilled, then we fall back to the gradient descent update step. See Subsection 4.2.3 for details.

7.1.1 Newton-CG: optimizer features

Neural networks are becoming more sophisticated, and with it, the number of parameters in neural networks grows rapidly (see also Figure 4.1 and the section around it). As the optimization domain corresponds to the number of parameters in a neural network, training them is a very high-dimensional optimization task. One suffers with *non-convexity* and *non-smoothness*, with data-batching one deals with *stochasticity*, and *hyperparameter tuning* with the sheer *millions of weights*. Such problems often tend to be difficult to optimize, and implementations struggle with *data scarcity*. Our goal here is a *novel tunable algorithm* that satisfies our wish list, and apply it to a broad range of problems.

The optimizer aims to be *fast and reliable* and to be able to work on a *reduced number of weights*. We reach the former by a mathematical regularization and the latter by layer-wise optimization (e.g., standard practice for TensorFlow optimizers). With `Python` outer-loop scripting, we can do automated *hyperparameter tuning*, perform sensitivity analysis or uncertainty quantification, and transfer this to other datasets (transfer learning) or generalize this e.g., in a smartphone application. We can integrate this and achieve better for *explainabilty*.

In Table 7.1 we list the features of our `Newton-CG` implementation. They can be used in an outer loop for hyperparameter tuning or similar. However, one goal was also to *reduce* the number of hyperparameters. A devil's advocate may rightfully comment that it seems that we *exchanged* the hyperparameters of in-practice optimizers (`Adam`, `SGD`: lr, $\beta_{1,2}$ with others, see Table 7.1). However, the hyperparameters from `Newton-CG` have strong mathematical foundations and can be chosen with knowledge from *numerical optimization*; in turn, they can also be *auto-tuned* – on a small subset of the problem or according to a table for a corresponding optimization task (regression, auto-encoder, computer vision or transformer). We refer to the very problem-dependent behavior, tabulated in Table 7.3. This allows for a look-up table on which problem requires which algorithm and hyperparameter set.

A suitable setting starts with a first-order algorithm and then gradually moves towards a more exact second-order. `Newton-CG` allows for this with regularization parameter τ.

Table 7.1: `Newton-CG` optimizer features: Optional arguments; if not specified, defaults are used. `cg-tolerance` and `max cg-steps` are necessary to either *limit* computational effort or *enforce* a certain tolerance. `Newton-CG` cannot ensure *both* computational budget and accuracy.

Optional Parameters	Typical Range	Comment
`learning-rate`	$1 \times 10^{-5} - 1$	Pure Newton uses 1. Decrease for better stability.
`learning-rate scheduler`	`exponential decay`, `cyclic lr`	Arbitrary function. Many common optimizers rely heavily on this feature.
`Tikhonov regularization`	$1 \times 10^{-7} - 1$	If high, converges to `SGD`.
`cg-tolerance`	$1 \times 10^{-7} - 1 \times 10^{-3}$	Stopping criterion for CG – whichever is reached first.
`max cg-steps`	$1 - 20$	Stopping criterion for CG – whichever is reached first.
`armijo step-size restriction`	$0.01 - 1$	Acceptance criterion at which Newton-step is accepted or `SGD` fall-back is used.

7.1.2 Automatic differentiation framework

We previously experimented with a MATLAB implementation for multilayer perceptron (*MLP*) (autoencoder) networks[1] without convolutions in [CRB18, CRY+21]; we precomputed the Hessian, stored it in memory and analyzed it, e.g. with the dense matrix interface of `GOFMM`.

Since more recent networks often involve convolutions, where backpropagation is not as straightforward as in *MLP*, we also experimented with a custom precomputed Hessians (via `TensorFlow`), showing good `GOFMM` approximation behavior. However, we were restricted in size because we again stored the matrix as a file in memory with quadratic complexity; additionally, this procedure did not support a feedback into the training process for weights (second-order optimizer). Therefore, the restriction (and goal at the same time) was to stay within the code framework *without* an extensive interface to another library.

Inspired by iterative method literature and autodifferention, we then came up with a *mat-vec* only second-order scheme called `Newton-CG`, without using `GOFMM`. The `Newton-CG` optimization strategy is independent of the implementation and, of course, is suitable in any setting where **(1)** second-order is beneficial and **(2)** storing Hessians is infeasible w.r.t. memory consumption. We (again) choose the *differentiation* framework `TensorFlow`, since it is widely used in the community and fits our requirements. For an integrated survey and outreach, the `Newton-CG` was implemented as *TensorFlow optimizer*; using `pip`, it is public and can be installed and applied to most machine learning pipelines[2]; thus, it *outreaches* to the deep learning community and is *reproducible.*

[1]Written in collaboration with Chao Chen and George Biros

[2]For installation instructions, please follow the newest version from `https://github.com/severin617/Newton-CG`

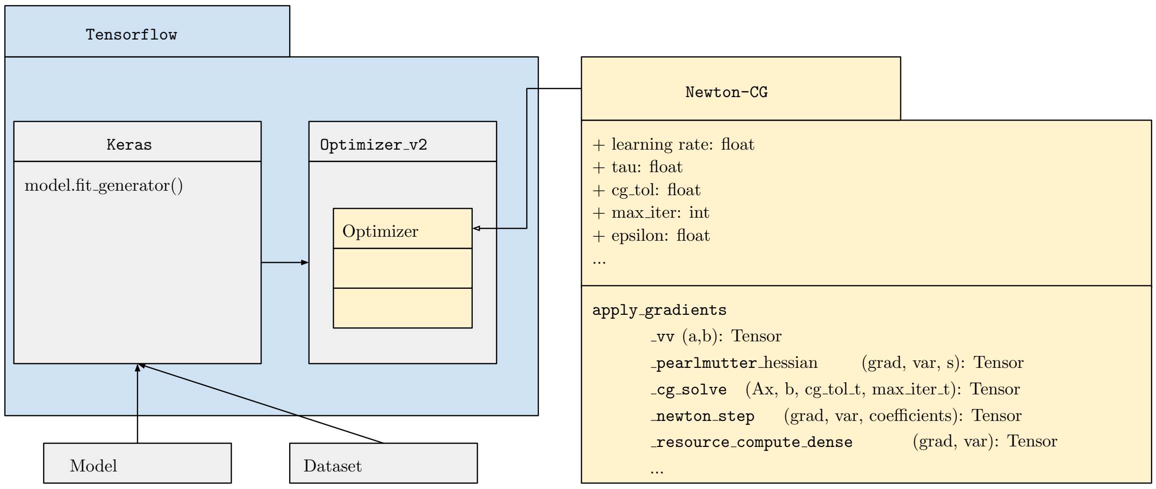


Figure 7.1: UML sketch of class `Newton_CG` inheriting from base `tf.python.keras.optimizer_v2.Optimizer_v2`. It can be applied to any sequential model and dataset. For recursive structures, we limit the depth, as for Subsection 7.2.6.

TensorFlow optimizer

The TensorFlow programming model consists of two main steps: (1) Define computations in the form of a "stateful dataflow graph" and (2) execute this graph. At the heart of model training in TensorFlow lies the *Optimizer*; we used `tf.python.keras.optimizer_v2.Optimizer_v2` subclassed similar to different optimization algorithms (e.g., we compared to `Adam` or `SGD`). The base class handles the two main steps of optimization: `compute_gradients()` and `apply_gradients()`. We re-use the same computational graph for backpropagation of the intermediate gradient-vector product as described by [YGKM20].

When applying the gradients, for each variable that is optimized, the method `resource_compute_dense(grad, var)` is called with the variable and its (earlier computed) gradient. In this method, the update step for this variable is computed. It has to be overwritten by any subclassing optimizer. We implemented two versions of our optimizer: one inheriting from the optimizer in `tf.train` and one inheriting from the Keras `Optimizer_v2`. The latter is shown in Figure 7.1. The constructor accepts the learning rate (plus optionally learning rate scheduler) as well as the `Newton-CG` hyperparameters: regularization factor τ (in code written plain `tau`), the `CG`-convergence-tolerance and the maximum number of CG iterations, see Table 7.1. Internally, the parameters are converted to tensors and stored as `python` object attributes. The main logic happens in the above mentioned `resource_compute_dense(grad, var)` method. You can find the implementation here[3]).

Data parallelism with Horovod

Horovod is a data-parallel distributed training framework (open source) for TensorFlow, Keras, PyTorch, and Apache MXNet, that scales a training script up to many GPUs using `MPI` [SB18].

The following Table 7.2 summarizes the data and model parallelism in the context of neural network optimization. In the first step, we apply Horovod for the data-parallelization of the second-order `Newton-CG` approach. Here, data-parallelization describes the same operations performed in parallel on different batches of data. In the second step, the whole algorithms can

[3]`https://github.com/severin617/Newton-CG/blob/main/newton_cg/newton_cg.py#L127`

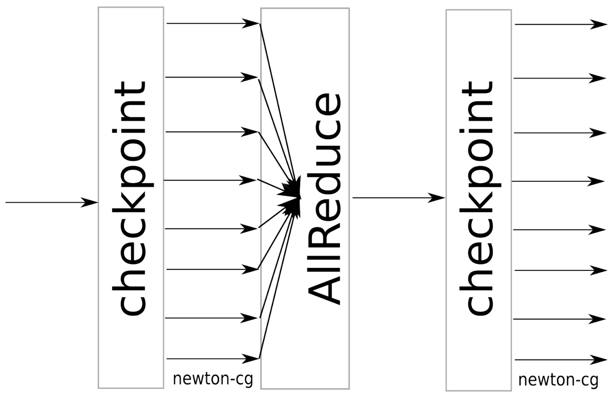


Figure 7.2: Data parallelism approach: arrows correspond to a compute worker (*GPU*). **(1)** 8 GPUs compute (`Newton-CG`) individual weight update for corresponding batch (8 arrows), and **(2)** accumulate it with all others (AllReduce, 8 arrows point towards a single point); **(3)** accumulate result (weight update) is applied to whole network (single arrow). Step (3) is not equivalent to a weight update of an 8-times as big data-batch. Hence, the parallel algorithm is slightly different from the sequential version. Similar in a red-black Gauß-Seidel iteration (parallel) than sequential (only black) Gauß-Seidel.

be parallelized through model-parallelism, which—in contrast to data-parallelism—performs different operations in parallel on the same data. We did not pursue the latter due to sufficient parallel efficiency and wait for further studies to determine whether we see a breakthrough of Newton-CG in terms of the number of steps or accuracy (or similar 2nd order algorithms).

Table 7.2: Comparison of data and model parallelism

Data parallelism	**Model parallelism**
Operations performed on different batches of data.	Parallel operations performed on same data (in identical batch).

In order to show the applicability of the proposed second-order optimizer for real-world large-scale networks, it is necessary for parallel optimization computations to allow for faster runtimes. We decided to use the relatively simple and prominent strategy of *data-parallelism*. Data-parallel strategies distribute data across different compute units, and each unit operates on the data in parallel. So in our setting, we compute different `Newton-CG` steps on i different mini-batches in parallel, and the resulting update vectors are accumulated using an *Allreduce* (see Figure 7.2). Note that this is different to e.g. a i-times as big batch or i-times as many steps since this would use an updated weight when computing gradient information via backpropagations. In a smoothly defined function, this could converge to a similar minimum; however, due to stochasticity, this may not (i.e., the parallel algorithm is slightly different from the serial version).

In academic research, some groups have switched to *PyTorch* or *Jax*; *TensorFlow* is portable since it uses the same backend. Figure 7.3 shows that TensorFlow, PyTorch or Jax all use the autodifferentiation backend called XLA (or OpenXLA). Hence, this should allow for performance portability, and the underlying exact framework is ephemeral. OpenXLA allows for different back-end implementations and device optimizations. In some scripts, we need to modify the TensorFlow device list with `xla_gpu` inside `tf.python.keras.optimizer_v2.Optimizer_v2` – in order to force GPU usage.

For **best** performance, we suggest using NVIDIA NGC Containers because they are optimized for NVIDIA GPUs and updated by NVIDIA on a regular basis.[4] Most test runs, and the ones for **best** performances, were conducted on the *Leibniz Rechenzentrum (LRZ)* AI System DGX-1 A100 Architecture with 8 NVIDIA Tesla A100 and 80 GB per GPU (see Section 5.2). Jobs are submitted with SLURM in the respective partition, nodes, and specifications.

TensorFlow Probability

Second-order may benefit not only in situations where the optimization not only involves scalar/vector parameters but probability distributions around the *weight parameters*. Continuity of optimization objectives (like a probability distribution) may favor a 2^{nd}-order scheme. To this end, we looked at Bayesian Neural Networks (BNN) in Subsection 7.2.4. In the implementation, we aimed to use synergies in the development effort, hence to stay within the *TensorFlow* framework. For such tasks, there exists TensorFlow Probability (TFP), a library built on top of TensorFlow itself; TFP inherits from TensorFlow, and thus it is possible to use the same previous optimizers (among others) for probabilistic models, and for our purpose, Bayesian Neural Networks.

Regarding **Bayesian network layers** in TensorFlow Probability, dense layers in neural networks are replaced by TensorFlow Probability's Dense Variational; and for the convolutional layer TensorFlow Probability's Flipout layers (1D/2D respectively). DenseVariational implements a dense multiplication layer with a weight probability distribution (see Bayes by Backprop) [tfpb]. Flipout implements a convolutional layer of the aforementioned Flipout estimator by Wen et al. [tfpa, WVB+18]. In image recognition, often 2D-convolutional layers occur. In order to include a Bayesian term in a regular visual network, we re-factored such convolutions with a 2D-Flipout estimator.

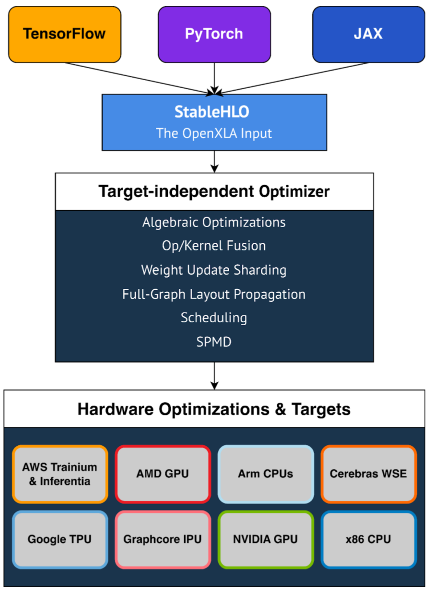


Figure 7.3: OpenXLA pipeline, taken from `https://opensource.googleblog.com/2023/03/openxla-is-ready-to-accelerate-and-simplify-ml-development.html`

[4]See `https://doku.lrz.de/5-using-nvidia-ngc-containers-on-the-lrz-ai-systems-10746648.html`

TensorFlow models

Our optimizer can be integrated into almost all *TensorFlow* scripts; it can be installed using `pip`[5] and applied to most machine learning pipelines; thus, it reaches out to the deep learning community and is reproducible.

The scripts that we used are collected on our general `Newton-CG` GitHub page. For the transformer case, we worked with several code bases outside this since the recurrent structure of the transformer architecture required loop unrolling and different things. We have had several git forks on the LRZ gitlab[6], but for reproducibility and outside reach, we migrated the most current version to GitHub. It can be found under `https://github.com/severin617/nlp-newton-cg`.

7.1.3 Datasets

In this section, we explain the datasets that we used in the results chapter. We decided to dedicate a section to it in order not to distract the reader too much in the global convergence analysis results section.

Life expectancy: Dataset involves life expectancy estimation influenced by several factors (e.g., country, infant deaths, diphtheria, GDP) [7]. Note that the dataset is relatively small (around 180 entries, where every entry belongs to a country).

Boston housing: The Boston Housing Dataset is publicly available, e.g. at Kaggle, or directly in TensorFlow. The governmental Census Service collected it in the Boston/Massachusetts area. It includes 14 factors, including crime rate, air quality, age, house features, price and population proportion.[8]

MNIST: MNIST is a collection of handwritten digits. It consists of 60000 pictures with 28×28 grey-scale pixels. We used 48000 samples for training, validated them on 12000, and finally tested them on 10000 images. The MNIST data was preprocessed by dividing every pixel of the grey-scale image by 255 in order to scale all values to $[0.0, 1.0]$.

Breast Cancer Wisconsin: The Breast Cancer Wisconsin dataset[9] contains features from cell nuclei from breast mass, we display an example picture in Figure 7.4. The data set is relatively small; it only contains 357 benign and 212 malignant samples. Per sample, it contains 30 features, such as geometrical descriptions of the nuclei (size, radius, contour), see also Figure 7.4.

[5]for the most current version on how-to, please refer to `https://github.com/severin617/Newton-CG`

[6]see `https://gitlab.lrz.de/exaniml/nlp-newton-cg`

[7]See `https://valueml.com/predicting-the-life-expectancy-using-tensorflow/`

[8]Dataset is available here `https://www.kaggle.com/code/prasadperera/the-boston-housing-dataset` or here `https://www.tensorflow.org/api_docs/python/tf/keras/datasets/boston_housing`

[9]Publicly available: `https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data`

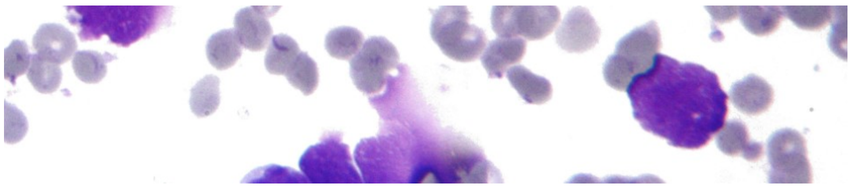


Figure 7.4: Example picture of the Breast Cancer Wisconsin (Diagnostic) Data Set. The data set is preprocessed; for each sample, there are 30 feature descriptors computed. We work on the 30 descriptor vectors, not on the images themselves. Not a single descriptor (size, contour, etc.) determines the target, so a non-expert cannot judge benign/malignant state. Figure taken from kaggle dataset URL in footnote above.

BreKHis: The breast cancer histopathology database BreaKHis [SOPH15] consists of 7909 histopathology images acquired by biopsy on 82 patients. The microscopic pictures are from different magnifying factors ($40 - 400\times$) and are 3-channel RGB 700×460 pixel pictures with 8-bit depth. Both benign and malignant samples are included and, moreover labeled as one of eight tumor types and assigned to an anonymous patient ID. [Wei21]

Benign tumors usually show slow growth and are considered relatively harmless. Malignant tumors are, however, in everyday language, equivalent to cancer. They are invasive, quickly grow and spread metastases, and cause death over the long run. It is, therefore, imperative to distinguish such cells.

Figure 7.5 shows in the top row four distinct benign types (adenosis, fibroadenoma, phyllodes tumor, and tubular adenona) and four malignant (carcinoma, lobular carcinoma, mucinous carcinoma and papillary carcinoma). Its form characterizes the cells and a medical doctor can diagnose the types on the way it looks under the microscope.

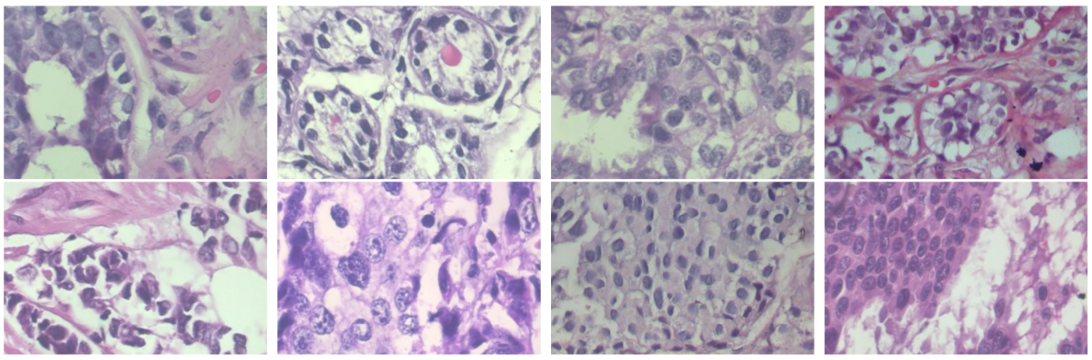


Figure 7.5: Histopathology images of eight types of breast tumors (first row: benign, second row: malignant). Figures taken from [SOPH15].

ImageNet: The ImageNetLarge Scale Visual Recognition Challenge 2012 (ILSVRC2012) alongside published the first version of ImageNet with around 14 million images, 1000 classes, and around 1300 images per class. The pictures can be scaled; we use square images of $4096 (64 \times 64)$ input units, whereas a `JPEG` format can be comfortably scaled, and therefore scaling to `PNG`-style pixels. This is our largest dataset that we also used for parallel performances; it is publicly available after accepting the (academic) usage terms.

CINIC10: CINIC10[10] contains CIFAR-10 images and a few selected from Imagenet. CIFAR, in contrast to MNIST, contains more realistic pictures of objects. CINIC10 contains 210,000 images downsampled to 32×32, and is considered a bridge between CIFAR and ImageNet, for benchmarking or testing. It can be split into train/validation/testing set accordingly.

Coffemugs: We also experimented with self-made photos of coffee mugs from the university chair kitchen. The challenges of real-world and editing for public use became evident: many photos (we only had 50) and preprocessing in the lighting, etc., are necessary for meaningful results. In addition, data augmentation would help significantly. While it works methodologically, we note the efforts of original authors of public datasets for their preprocessing work.

Intel image classification dataset: The dataset[11] was published for a challenge to distinguish similar looking landscape photos. It only has 6 classes of all similar-looking classes (landscape “scene” photos, classes are ’buildings’, ’forest’, ’glacier’, ’mountain’, ’sea’, ’street’).

fastText: fastText was developed and trained by Facebook on large text corpora, such as news articles, Common Crawl[12] and Wikipedia. Both Portuguese(cc.pt.300.vec) and English(cc.en.300.vec) dense pre-trained word embedding files contain ≈ 2 million word vectors with dimension 300. An example for the word vectors are:

```
'that' -0.0315 0.0328 ...  -0.0239 -0.0125 0.0045 0.0145 -0.0080
'with' 0.0149 -0.1152 ...  0.0758 -0.0099 -0.0632 -0.0068 -0.0139
```

7.1.4 Summary of implementation

The implementation here aims to show the feasibility of a 2^{nd}-order optimizer for arbitrary neural networks; this *enables* the data science community to reproduce this to their test cases. Our method is non-intrusive and can be integrated into existing ML pipelines. Other codes can use it in a plug-and-play fashion. We achieve feasibility through *algorithmic* approximation of a true Newton scheme, i.e. a coarse approximation using a few `CG`-steps; we do not rely on a high-performance implementation of `Newton-CG`.

We implemented a TensorFlow and Keras optimizer for our `Newton-CG` scheme with many features, like lr-scheduler, regularization τ, Armijo step size restriction, etc. This opens myriad possibilities for use in most TensorFlow models; it also allows for Bayesian Neural Networks using TensorFlow Probability or Transformer architectures by restricting the recursion depth. Our optimizer can be used on multiple *GPUs* with data-parallelism using `Horovod`.

In Subsection 7.1.3 we also described all datasets used for the result chapter.

7.2 Results: Newton-CG global convergence analysis

After implementing a novel algorithm, it is imperative to test it thoroughly, from a simple correctness check to large models. In many studies, authors simply show that their code is superior to some baseline - which could have been generated by cherry-picking the suitable results and hyperparameters. It is best practice to also report downfalls and “fairly” compare the *addendum* (the addition/modification to a common benchmark). For mathematical papers, a challenge often appears in the implementation/application to large-scale problems and datasets.

[10]Available here: `https://datashare.ed.ac.uk/handle/10283/3192`
[11]https://www.kaggle.com/puneet6060/intel-image-classification
[12]https://commoncrawl.org

Deep learning's "accuracy" is a random metric – it often jumps and is extremely sensitive. We focus on the validation loss, which in most cases relates but is more continuous.

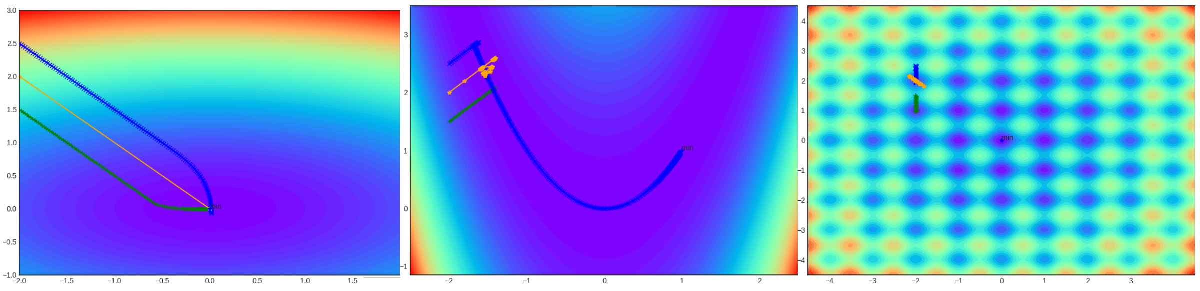
We have performed a comparative study for benchmarks from different fields. We observed that our `Newton-CG` algorithm is very **problem-dependent**. We start from the most basic function optimization – from a quadratic function to the Rosenbrock function from optimization literature.

The analysis of the 2nd order optimizer for artificial neural networks is divided into 5 fields: regression, autoencoders, Bayesian, vision and natural language processing. We have observed that the optimization behavior is very problem-dependent (see Table 7.3). *Disclaimer:* For some cases, the 2^{nd} order does not pay off, despite Armijo feasibility check and regularization. Our goal here is not to revolutionize an optimizer suitable for all settings; instead, we find benefits for machine learning benchmarks in order to suggest benefits for specific tasks.

In detail, for artificial neural networks, we move on to a shallow 1-layer network (basically regression) implemented with auto-differentiation (so we can use our TensorFlow `Newton-CG`), showing very promising results (++ in regression in Table 7.3). We had much hope for the variational auto-encoder since it involves many layers (non-linearities) where a second-order could outperform a first-order (SGD); `Newton-CG` results are subpar (○). Subsequently, expectations have been high for *Bayesian Neural Networks*, as optimization involves not only scalar parameters but probability distributions around the *weight parameters* (basically two parameters, mean and variance). Results vary depending on dataset and model architecture (○ to ++). For *image classification* simple convolutional netwrks work well for `Newton-CG` +, but deep ResNets/MobileNet/InceptionNet with ($\approx$ 50 layers) shows nice benefits +; once `Newton-CG` beats `Adam`, once `SGD`, hence it offers the best out of two worlds; however, transfer learning show mediocre behavior (- to ○). For natural language processing, a transformer architecture for a translation task, `Newton-CG` shows benefits in validation (prevents overfitting a bit +).

Table 7.3: Comparison of qualitative behavior of `Adam`, `SGD`, and `Newton-CG`.
As metrics we used a mixture of Loss, Stability and speed in terms of required epochs to reach a validation loss minimum

scenario description \ **optimizer**		`Adam`	`SGD`	`Newton-CG`
regression	life expectancy	- -	-	++
	Boston housing	+	- -	++
variational autoencoder	mnist	+	+	○
bayesian nn	mnist	○	+	+
	wisc	++	+	○
	BreaKHis	-	-	++
image-classification	resnet-50 imagenet	○	○	○
	mobile-net	○	+	+
	inception-net	+	○	+
	transfer learning	+	○	-
natural language	transformer	○	○	+



(a) objective function $x^2 + 4y^2$ (b) Rosenbrock function $(1-x)^2 + 100(y-x^2)^2$ (c) Rastrigin function

Figure 7.6: Optimization steps for Gradient decent (in green), Adam (in blue), and `Newton-CG` (in orange). Starting values are varied for better visibility, (2,-1.5), (2,-2.5), and (2,-2), respectively. Use zoom to see the optimal point marked tiny with min.

7.2.1 Test functions: Quadratic, Rosenbrock and Rastrigin

We use this section to check the correctness of the implementation and application to low-dimensional function optimization. Therefore, we wrote a Python script that outputs the results of a Hessian with unit vectors, essentially returning the Hessian itself. For functions without a simple symbolic Hessian, we had a central difference scheme to compute (approximate) single entries of the Hessian. We use the official TensorFlow optimizers.

Next, we tested whether our approach can find the minimum of a 2D-quadratic function after one step, compared to Adam/SGD, which struggle to find the right direction if its iso-lines are an ellipsoid. We constructed the 2D quadratic objective function $x^2 + 4y^2$, and checked reconstruction of the 2×2 Hessian to its analytical form: $\begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$). As expected, after one Newton step, our approach reaches the minimum (see Figure 7.6 (a)). Stochastic gradient descent boils down to normal gradient descent in these normal function optimizations. Adam also, respectively, does not profit from higher-order gradients and needs many steps (similar to SGD).

A challenging benchmark function in optimization is the quartic *Rosenbrock* function $(1-x)^2 + 100(y-x^2)^2$. Due to its shape, it is sometimes also referred to as the "banana" function. In the subplot (b), a blue banana may be recognized(Figure 7.6), or on a bigger 3D plot, one sees the banana shape better; the minimum is marked small at $(1, 1)$. We observe that SGD and `Newton-CG` oscillate and do not find the minimum $(1, 1)$; Adam finds it after ≈ 500 steps. This is not surprising, as many optimization algorithms fail for this challenging function.

The Rastrigin function $f(x) = An + \sum_{i=0}^{n}[x_i^2 - Acos(2\pi x_i)]$ is a (locally) non-convex function from mathematical optimization, a "bumpy parabola surface" with the minimum in $(1, 1)$. For the plot (Figure 7.6) we use constants $A = 10$ and $n = 2$. We observe that with naive parameter choice, `SGD`, `Adam`, and `Newton-CG` get stuck in a local minimum. Stochasticity (*mini-batches*) can improve this behavior. We also allow for hyperparameter optimization and used the popular Python package hyperopt [BYC13]. A completely different approach is Consensus-based optimization (CBO). It yields significantly better results for the Rastrigin function and would eventually reach the minimum by sufficiently many sampling starting points [FKR22]. CBO is suitable for such cases with many local optima.

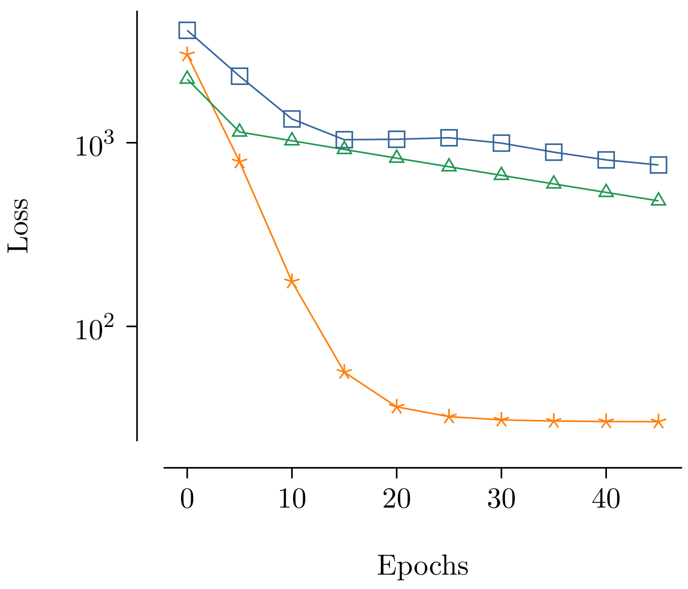


Figure 7.7: Semi-logarithmic loss plot for a 1-layer network training (regression) of the **life-expectancy** dataset. Hyperparameters were chosen as default values; no pretraining was necessary.

7.2.2 Regression

After a general function optimization, we switch to a shallow network scenario. In other words, we follow Occam's razor, i.e., problem solving with the simplest set of elements, or in other words, from *simple* to *complicated.* There is a vast literature on shallow network optimization and many theoretical results up to infinite-wide shallow networks [Uns23, BCFW21]; we simply interpret a regression problem as a 1-layer network and analyze this. We have a ReLU activation function and a square loss function. This is then similar to a least-square fit for a linear regression model.

Synthetic data: Shallow network

Regarding regression, there are many sources online – we chose life-exp involving life expectancy estimation (see Subsection 7.1.3, it includes several factors, such as country, infant deaths, diphtheria, GDP). We built a 1-layer network with a square loss function.

Figure 7.7 shows that `Newton-CG` outperforms the state-of-the-art optimizers `Adam` and `SGD` from TensorFlow significantly. Also, note the logarithmic Y-axis and the values of the scale. `Newton-CG` is better in orders of magnitude, the *superlinear*, or even *quadratic* convergence in contrast to `SGD` and `Adam` becomes visible, as well as a much lower final loss. Note that the dataset is fairly small, the computational cost is hence negligible. Loss is age estimation error; for a set of $\approx$ 180 entries, this corresponds to 0.5 years misestimation of life expectancy on average.

Boston housing dataset

We used a simple 1-layer network, with a single dense layer and a sigmoid activation function to Predict Boston Housing Prices. First, we experimented with a zero-weight initialization; however, due to much better applicability, we moved to a standard TensorFlow initializer with normally distributed random variables. For better clarity, we worked with fixed learning rates,

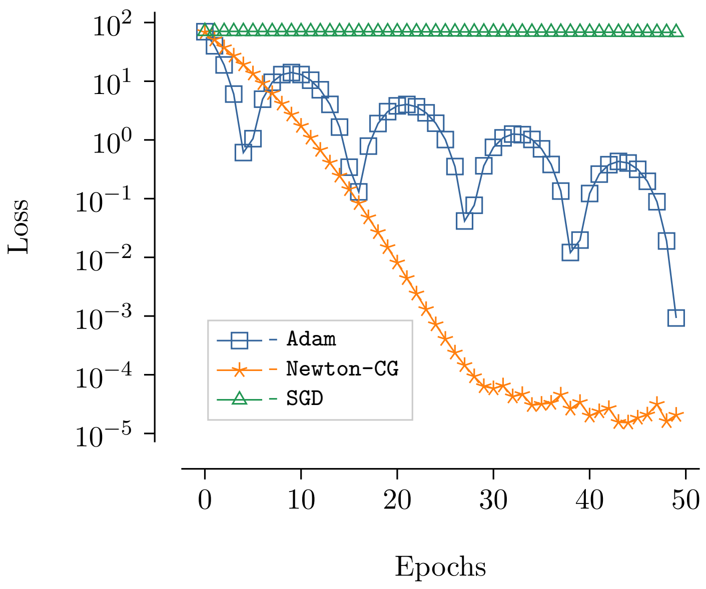


Figure 7.8: Predicting Boston housing prices with 3 different optimizers, `Adam`, `Newton-CG` and `SGD`. Surprisingly, Adam's loss seems to oscillate between local minima, `SGD` stuck. Due to **semi-log** plot `Newton-CG` outperforms the former by orders of magnitude.

e.g. 1E-3. Since the data is relatively small (506 data points), we can work on the full dataset at once, and no mini-batching is necessary.

Figure 7.8 shows the mean squared error loss. It can be observed that the stochastic gradient descent optimizer is stuck and is not suitable for training. Adam seems to find several local minima and stays there until hopping to the next. We were surprised by this result but were able to reproduce it. `Newton-CG` is a stable optimizer and very suitable for this simple regression (i.e. 1-layer) model.

7.2.3 Variational autoencoder

Variational Autoencoders (VAE) are widely used in many areas. They consist of input space, an encoder to a latent space, and a decoder to an output dimension.

We use our TensorFlow Keras version of the optimizers. We built the script with an input dimension (due to MNIST) of 784, and a very simple dense layer to intermediate dimension 64, and then to a latent dimension of 2. The decoder is the inverse, respectively. A loss, we use the sum of a binary cross entropy and a term for Kullback-Leibler (short: KL) divergence loss.

Figure 7.9 shows the optimization behavior of the three optimizers `Adam`, `Newton-CG` and `SGD`. The fastest and lowest results are yielded by `Adam`, `SGD` reaches a similar minimum but at a much slower pace. `Newton-CG` struggles with this problem. It results in a much higher validation loss at the end compared to the other two. A hyperparameter study on `learning-rate` and `tau` could help, but we observed struggles for `Newton-CG` in this variational autoencoder case. The motivation for 2^{nd}-order was continuity in the optimization domain of VAE; however, the results do not confirm this expectation.

7.2.4 Bayesian neural networks

Bayesian Neural Networks (BNN) are a class of probabilistic neural networks. In this thesis, we look at BNN models, where weights are not scalar values, but weights rather contain probability

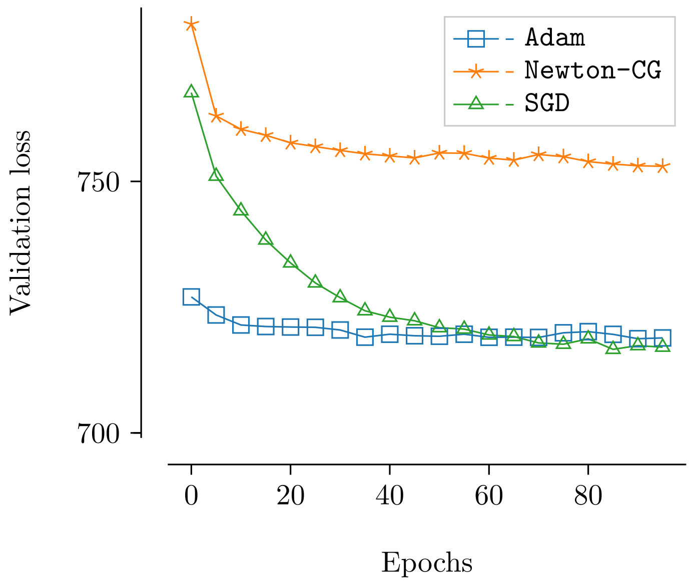


Figure 7.9: Validation loss of a simple variational autoencoder on the MNIST dataset for 100 Epochs. Note the **semi-logarithmic** plot and the scale limits; although `Newton-CG` looks much worse, the difference marginal.

distributions. From intuition, it is hard to explain a distinct weight as a scalar value for a randomly chosen network architecture (from literature or experience). BNNs with a probabilistic view of weights are, therefore, inspired by (stochastic) measurement/input data and the random nature of network architecture.

In BNN training, the goal is to find the mean and variance for all weights. The “forward-pass” can then deliver a probability distribution of the network output. Either the probability distribution is passed through the network. Another simpler (and faster) way is to sample from weight distributions, similar to normal forward evaluation. To get uncertainties, this process can be repeated multiple times, and judgment can be drawn from a histogram of output probabilities.

Parts of this analysis has also been published in our proceedings paper [RNB23].

MNIST Multilayer perceptron

The first experiments with Bayesian neural networks of this work were conducted on the MNIST database of handwritten digits [LeC98]. There are two types of BNNs trained, one as a multi-layer perceptron, the latter involving convolutions. We only show the results of the former here due to convolutions shown in architectures for the other dataset (BreaKHis in the following subsection).

The multilayer perceptron BNN consists of two dense variational layers mapping the input (784 dimensions) to 1024 (hidden dimension) and, finally, 10 output dimensions. It uses a ReLU and softmax activation functions. As an optimization objective function, we use the loss as a sum of cross entropy loss and KL divergence. We have experimented with a learning rate scheduler, yielding no significant improvements.

Figure 7.10 (a) shows the loss on the validation set for the three tested optimizers. On the one hand, we can see that `Newton-CG` behaves very similarly to stochastic gradient descent. This may be caused by a high regularization term, as for high Tikhonov regularization τ, `Newton-CG` is approximately equivalent to `SGD`. On the other hand, `Adam` seems to struggle regarding the validation loss.

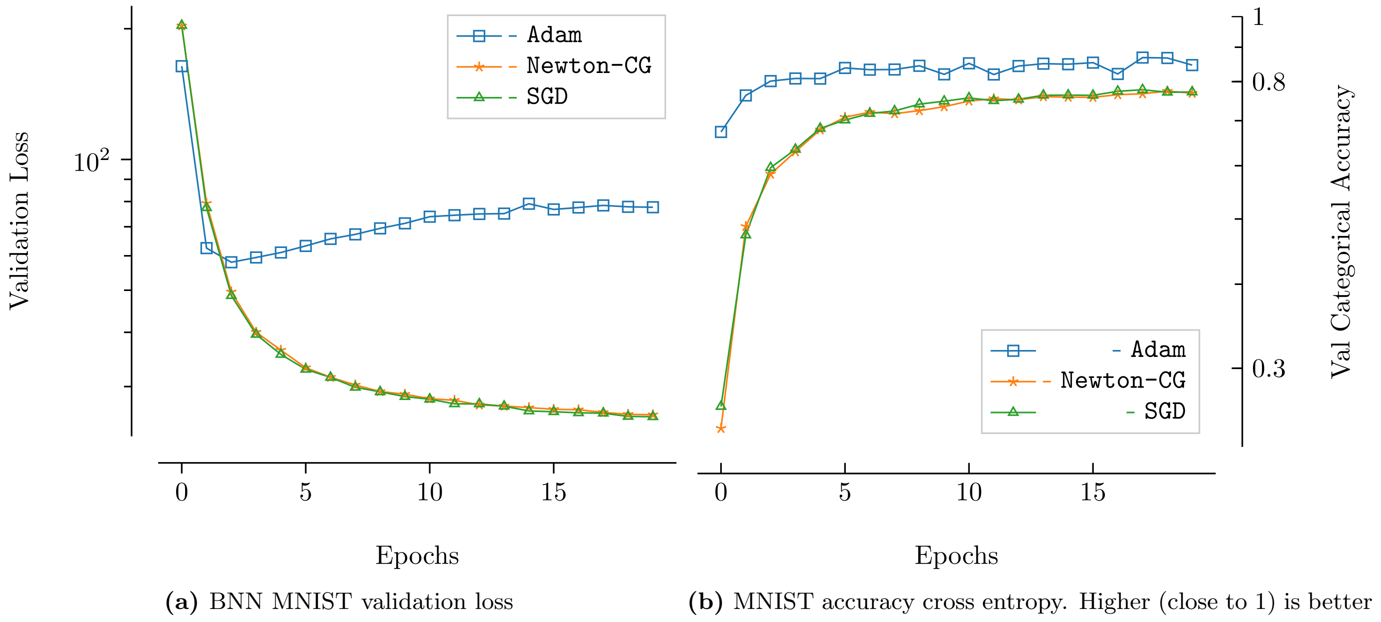


Figure 7.10: Validation loss of a BNN for MNIST. Note the semi-log scaling in both plots. Interestingly, accuracy and validation loss behavior differ for `Adam`.

The categorical cross entropy behavior shows a different response than the validation-loss behavior, see Figure 7.10 (b). `Adam` shows continuously better accuracy values than `Newton-CG` and `SGD`. Hence, training behavior cannot be concluded from loss only. Especially in this case, as loss is constructed from a sum of different terms (crossentropy loss and KL divergence).

Breast Cancer Wisconsin (Diagnostic) dataset

In a second step, we applied the `Newton-CG` optimizer in various settings to the Breast Cancer Wisconsin Diagnostic Data Set. Our initial motivation to apply BNN to medical imaging was due to the high degree of reliability through quantifying uncertainty in this area.

As said, we only display the dense network without convolutions here. It consists of two dense layers (soft plus activation) with sizes 512 and 2 (output: benign or malignant). It is initialized, and the posterior is built in a similar way as the previous Bayesian network. We use a batch size of 128, default hyperparameters, and for `Newton-CG`, a regularization of 10.

Figure 7.11 shows the validation loss of the multilayer perceptron model for the breast cancer data set. Adam shows the best (and stable) optimization behavior, whereas `Newton-CG` seems to struggle. This shows the limitation of the optimizer; a greater regularization would bring the loss to the stable green `SGD` line. However, let us reiterate that we used only default levels for hyperparameters and see limitations here. As the data set is small, runtimes are almost negligible for this case.

Breast Cancer Histopathological Database (BreaKHis)

Next, we use medical images from BreaKHis. This study uses a similar network architecture for the Non-Bayesian network (see Table 7.4). Convolutions have been replaced with the probabilistic flipout layers (2D) and dense layers with DenseVariational. The idea was to use a proven architecture and only add an uncertainty measure through the Bayesian Neural Network.

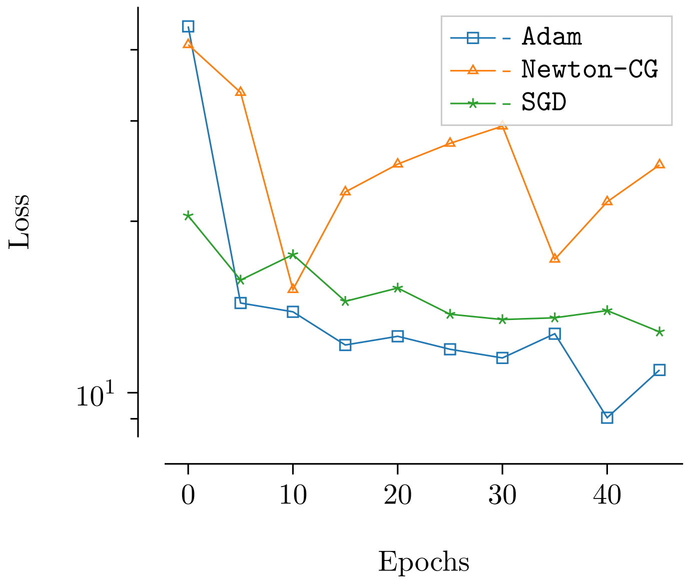


Figure 7.11: Breast cancer Wisconsin dataset multilayer percepton validation loss

Table 7.4: Architecture details of the network layers. CONV+POOL$_*$ are replaced with 2D Bayesian Flipout layers and Dense with Dense Variational. Adapted from Table II of [SOPH16].

	Layers					
	1	2	3		4	5
Type	CONV+POOL$_{max}$	CONV+POOL$_{avg}$	CONV+POOL$_{avg}$	Flatten	Dense	Dense
Channel	32	32	64	-	64	2
Filter Size	(3, 3)	(3, 3)	(3, 3)	-	-	-
Convolution Stride	(1, 1)	(1, 1)	(1, 1)	-	-	-
Pooling Size	(2, 2)	(2, 2)	(2, 2)	-	-	-
Pooling Stride	(2, 2)	(2, 2)	(2, 2)	-	-	-
Padding	SAME	SAME	SAME	-	-	-

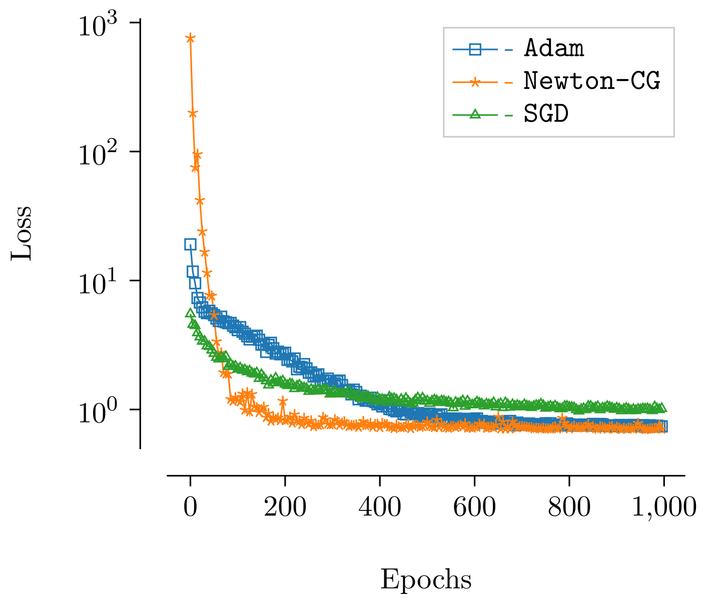


Figure 7.12: Validation cross-entropy as loss for a (convolutional) Bayesian Neural Network for the BreaKHis dataset.

In contrast to the above implementation with feed-forward layers, no learning rate scheduler is used. The initial learning rates are set to $\text{lr} = 0.001$ for `Adam` and $\text{lr} = 0.01$ for both `Newton-CG` and `SGD`. The KL-loss is re-weighted by dividing it by the size of the entire training set. We use a training batch size of 64 and a validation batch size of 8.

Figure 7.12 shows the validation loss behavior of the different optimizers. It can be observed that `Newton-CG` shows advantages compared to `SGD` and `Adam` in convergence speed and final loss. The final loss is similar to `Adam`, so we assume it finds a similar minimum.

7.2.5 Image classification

Image classification is a core area in deep learning and has paved the way for development efforts in machine learning. Hence, image classification lays out important benchmarks for optimizers, architectures, and so on. Therefore, we trained the image networks with data from Imagenet.

As for architectures, we followed the zoo of models from the official TensorFlow research repository, called TensorFlow-Slim image classification model library. We tested with many architectures from there, including smaller datasets and varied settings. In the following, we explain the necessary pretraining and then discuss different architectures(e.g., ResNet) and last-layer training (transfer learning).

ResNet50

Residual neural networks (or short ResNets) have played a prominent role in our analysis phase. Its core residual blocks with skip-connection layers (see Subsection 3.4.2) are used in many image classification architectures. ResNet50 refers to a 50-layer deep network. This, and similar networks from the TensorFlow Slim or Keras, are also used for the TUM-Lens showcasing application (see Subsection 5.4.1). Besides, due to long training time and stable behavior, it was our core case for parallel performances with Horovod. We used the Keras version of `Newton-CG` for the runs here.

Pretraining: Especially for the large image classification models, we saw that `Newton-CG` was diverging, and we needed some pretraining with an ordinary method. A second-order scheme can only perform well in convex regions, which are more likely sufficiently close to a minimum. If the optimizer "jumps" out of a current local minimum, it can also exhibit divergence. We try to tackle this with regularization and learning rate (schedulers). The ResNet-50 model was pretrained for 200 epochs to improve second-order convergence, with Imagenet and a `SGD` optimizer [13]. It gradually reached a validation loss of around 4.4 and a top-5 accuracy of 0.68.

Results: Figure 7.13 shows the optimization results for the ResNet-50 architecture trained for 50 epochs. With similar parameters[14] there is hardly any difference between `Adam`, `Newton-CG` and `SGD` here – it seems that the large dataset smoothens out differences and it only depends on data: more data, lower loss. While we found this result surprising, we are confident since we reproduced this behavior several times. It may be caused by pre-training, as it converges well in this region.

We compared the loss behavior to literature [YGS+21] and reached similar accuracies for a slightly different architecture (ResNet18 with 18 layers instead of 50 for ResNet50).

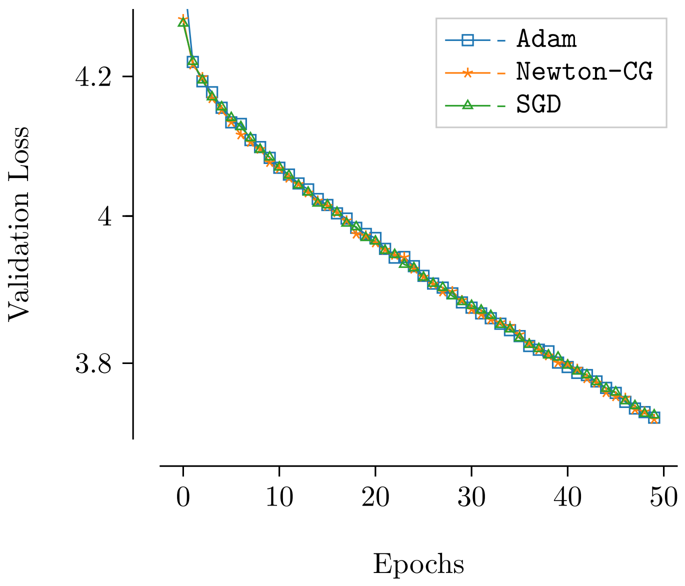


Figure 7.13: Full training Total validation loss of a ResNet50 architecture on ImageNet

MobileNetV2

Similarly, we applied the training process on other architectures for the same ImageNet dataset. As described in Subsection 3.4.2 MobileNetV2 is designed for *smaller cost* for *evaluation*, so that is suitable for a smartphone. Training of MobileNetV2 is still computationally expensive, and (full training) must be performed offline, e.g. on a cluster.

For this, we used the TensorFlow implementation of `Newton-CG` - which can be watched with Tensorboard. For postprocessing plotting, we used the tf eventfiles. *Pretraining* would be

[13]Following parameters were utilized in the pretraining: training/val-batch-size: 64, learning-rate: 0.001, momentum: 0.9, weight-decay: 0.00005. After each step, ten validation steps were used to calculate the top_5 accuracy, resulting in a final loss of 4.5332 and a final top_5 accuracy of 0.6800 after 2e5 steps.

[14]Following parameters were utilized: training/val-batch-size: 96, learning-rate: 0.001, momentum: 0.9, weight-decay exponential: 0.00005, for ADAM default $\beta_{1,2}$ values.

similarly expensive as for the ResNet. We circumvent this by using a pretrained *checkpoint* file from the *TensorFlow hub* that has been trained with SGD on 200 epochs. For training, we use the following parameters[15].

Figure 7.14 shows the optimization behavior of the validation loss for 40 epochs. While `Adam` is decreasing slowly, `SGD` (despite first-order without momentum) is outperforming `Adam` significantly. `Newton-CG` is performing similarly as `SGD` and hence, is outperforming `Adam` in convergence speed and final loss.

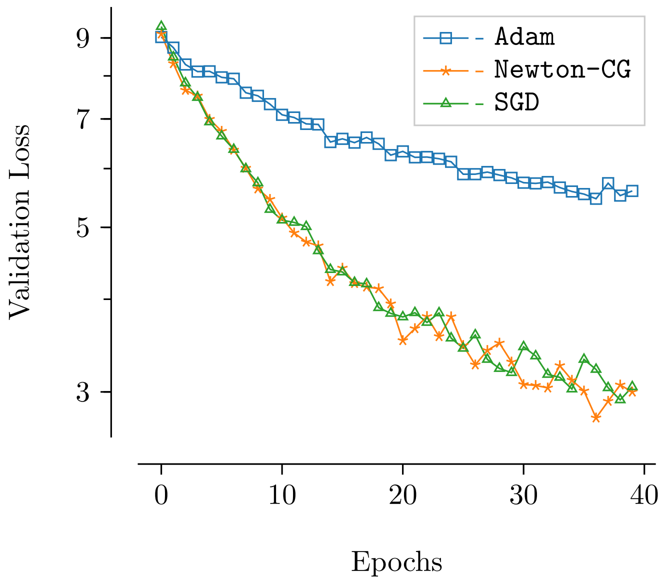


Figure 7.14: Total validation loss of fully training a MobileNetV2 architecture on ImageNet

Inception Net

As for a complete image classification analysis, we also report results for the InceptionV3 architecture. As explained in Subsection 3.4.2, an Inception Net has a skip-connection and a parallel track of convolutions of different sizes in a layer (the inception block) and augments them then to a longer vector. Respectively, other layers are chosen such that we reach the same output dimension, in our case 1000 classes in ImageNet.

As for pretraining, we used a *pretrained checkpoint* from the *TensorFlow Hub*, which has been trained with SGD on 200 epochs. Note that for implementation, similar to in MobileNet, we used the pure TensorFlow version of `Newton-CG` and restricted to training the last 5 layers to avoid memory problems. We use the same parameters as for the MobileNetV2[16].

Figure 7.15 shows the total for 40 epochs. `SGD` seems to be stuck in a local minima [17]. `Adam` and `Newton-CG` are significantly better. `Adam` gets slightly lower in final validation loss (note that the difference is small due to the logarithmic scale in the y-axis). `Newton-CG` is almost similarly good both in convergence speed and final validation loss value.

[15]Batch size of 96 and a learning rate of $1E-5$. As for `Newton-CG` we use regularization tau of $1E-2$ and allow a maximum of 20 cg-steps. Other parameters, e.g. those required by Adam or SGD, we use the default values from literature.

[16]e.g., learning rate $1E-5$

[17]Another set of hyperparameters could help. Manual tuning on a handful of configurations did not yield adequately better results.

While in MobileNet `Newton-CG` outperformed `Adam`, in InceptionV3, it outperformed `SGD`. It seems that the loss landscape of these both architectures is very suitable for `Newton-CG`, and for this, for these cases, `Newton-CG` combines the best of the two worlds (`Adam` and `SGD`).

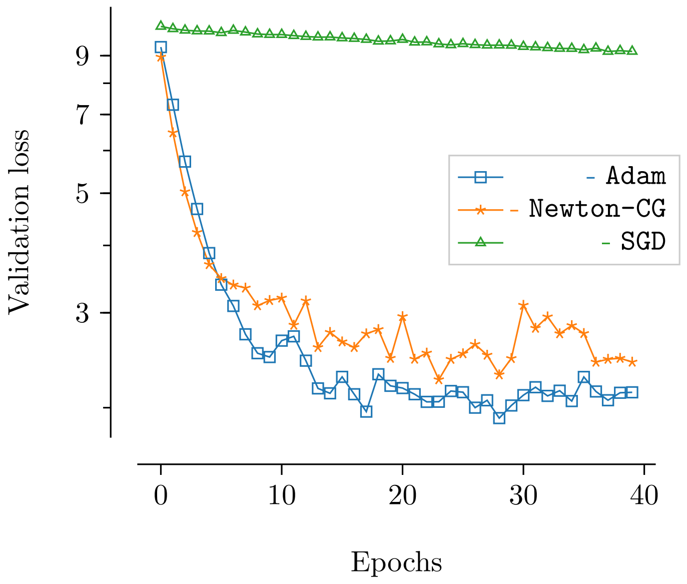


Figure 7.15: Total validation loss of a fully training a InceptionV3 architecture on ImageNet

Transfer learning

An efficient approach to tailor a given trained model to another task is transfer learning. From a *pre-trained* MobileNetV2, we added a last layer and thereby tuned it to a classification task for the CINIC10 data. The optimization results are shown in Figure 7.16. The total loss of `Newton-CG` at first decreases similarly as `SGD`. After about an epoch, `Adam` catches up and leads to a lower validation loss compared to `SGD`. `Newton-CG` suffers from a divergence of the current optimum starting at about 2.5 epochs. This subpar behavior could be caused by some stronger diverging effects caused by the Hessian iterative solver. A higher regularization can help, but second-order effects would smear out. This is another example that caution should be advised when dealing with higher-order methods for machine learning.

Additionally, we have performed several other transfer learning examples, such as the Intel image classification dataset. Training and evaluation work well; we have this architecture also on TUMlens. We also tried this approach on self-made photos of coffee mugs with mediocre results - preprocessing regarding lighting, rotation etc. of a dataset is imperative.

7.2.6 Natural language processing

With the rise of large language models (LLM), which form the basis of general purpose technology (GPT) (nowadays also reported in general news outlets), it is natural to also try the `Newton-CG` optimizer for natural language processing (NLP). After our results, we saw similar work done by others, using a second-order optimizer for language models [LLH+23]. The challenge for NLP optimizing lies in backdifferentiation with a recursive structure that may occur in these models. We again used our `Newton-CG` TensorFlow implementation and capped the recursion depth at 5 levels.

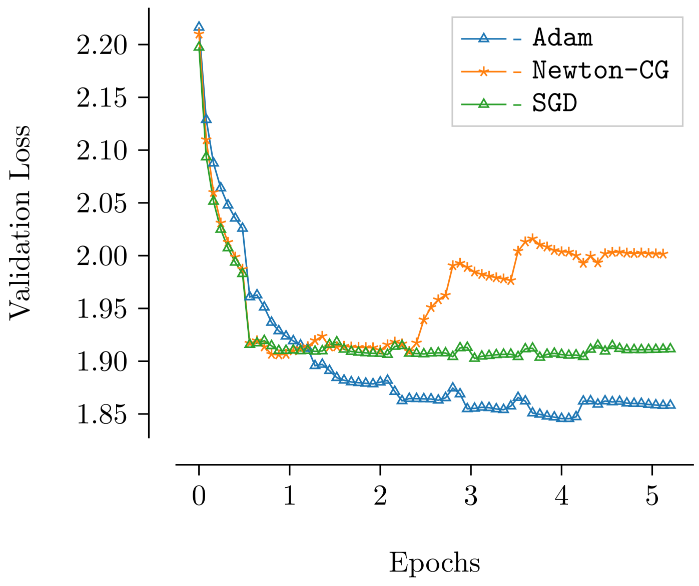


Figure 7.16: Validation loss in a transfer learning example for the CINIC10 dataset

Pre-trained word embedding

Word embeddings turn word tokens into vectors before entering the transformer, as mentioned in Subsection 3.4.3. Significant computing efforts are needed to train the word embedding layers from scratch (about 5 million parameters). For our study here, the vocabulary size of the tokenizer is set to 8000, and the dimension of the embedding layer in the transformer model is hence $\mathbb{R}^{8000\times300}$. We acknowledge that this number of trainable variables ($8000 \times 300 = 2.4m$) results in infeasible computing necessities and we chose a pre-trained word embedding from others [Hsi21,Mov22]. We also use pre-trained word embedding, since (1) TensorFlow requires a sparse weight update for this (*tf.keras.layers.Embedding*) and (2) our focus is on the transformer part.

We used the pre-trained word embedding from fastText [MGB+17]. For the natural language processing test case, we used the upper-mentioned translation task and employed the network architecture (transformer) in conjunction with Keras and our respective optimizer.[18]

We use the Transformer implementation with a flexible number of layers. For the results below, we used a single layer with 15 heads (see multi-head attention in the transformer model architecture). We use a learning rate of 1E-2 and for `Newton-CG` a regularization factor `tau`= 5.0. For the rest, we use default hyperparameters. In order to employ the full benefits of a second-order scheme, the starting value must be sufficiently close to a local minimum. We use a pre-training with Adam for 20 epochs with standard settings.

Results

In Figure 7.17, we show the optimization behavior of the transformer; on the left is the training loss, and on the right is the validation loss. We see that the training loss of `Adam` actually degrades (increases), but the validation loss decreases. This is surprising. Literature reports that `Adam` often generalizes better for nlp problems than `SGD` [ZKV+20]. This still does not explain the training-validation loss behavior but explains that the final validation is lower besides a higher final training loss than `SGD`. `Newton-CG` performs subpar to the other optimizers in these

[18]Code available here: `https://github.com/severin617/nlp-newton-cg`

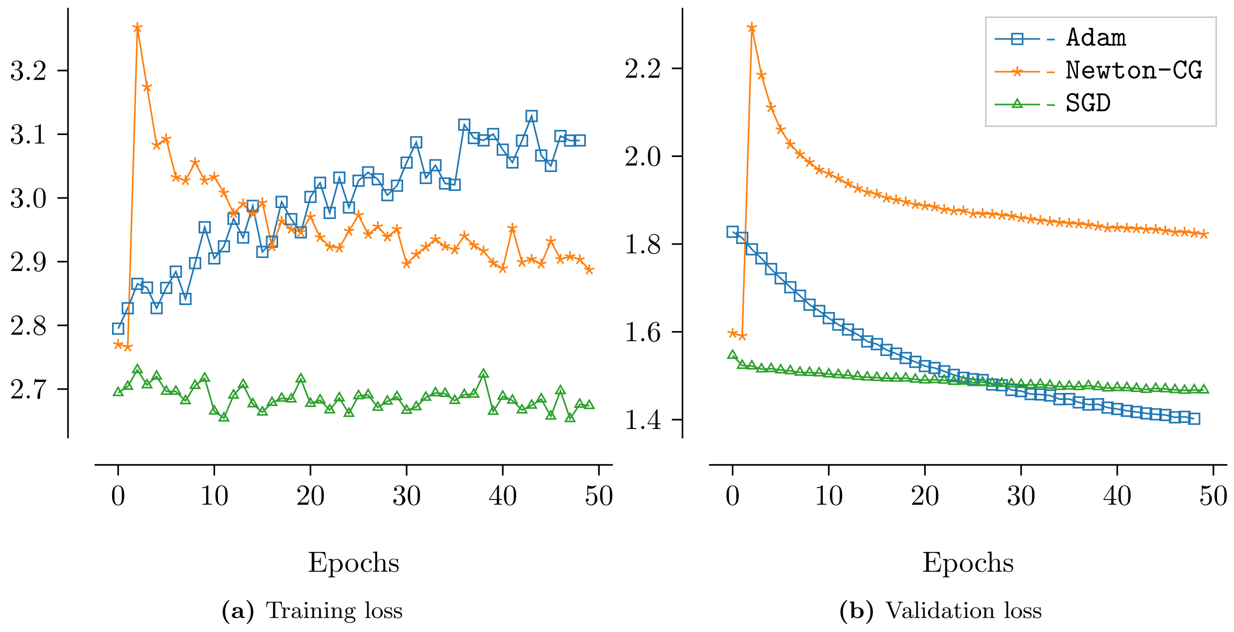


Figure 7.17: Transformer loss for natural language processing task comparing `Newton-CG` with `SGD` and `Adam`.

runs. BLEU scores are basically a measure for a sequence of several translated words. We found that for some validation BLEU scores, our `Newton-CG` can yield better results [Hsi21]. A final conclusion cannot be drawn, but rather, we can say that all have their benefits.

Despite the observed better generalization of `Adam` to `SGD`, [ZFM$^+$20] theoretically analyze the generalization gap and sharp minima. Their theoretic analysis yields that `SGD` generalizes better than `Adam`. Empirically, we cannot always confirm the theoretical results.

7.2.7 Summary

In summary, we showed the applicability and benefits of an approximate 2^{nd}-order scheme on a broad range of deep learning domains and large-scale architectures. We tabulated their quality in Table 7.3 with metrics convergence speed, stability, and final loss.

We measured predominately benefits of `Newton-CG` compared to `Adam` and `SGD` in neural network training. For example, in regression problems, in a Bayesian Neural Network for breast cancer diagnostics (including weight probability distributions), or in MobileNet or Inception-Net in computer vision. For some others, like other Bayesian Nets, a transfer learning example, or some NLP parts, we saw some disadvantages. Naturally, there are also limitations, since 2^{nd}-order is susceptible to instabilites. We observed the pattern that usually, in well-formed convergence regions, `Newton-CG` performs better than `ADAM` and `SGD` in trust/convergence regions, whereas in areas where the other two optimizers struggle, `Newton-CG` also struggles. For clarity, we can also refer back to the Table 7.3.

7.3 Results: Newton-CG with data-parallelism

In the previous Section 7.2, called global convergence analysis, we have seen that for several state-of-the-art exemplary cases, using second-order information for training neural networks is beneficial. In this section, we address the computational performance of Newton-CG.

Table 7.5: `Newton-CG` runtimes per epoch with batch-size 512, ResNet50 on ImageNet

	1 GPU	2 GPUs	4 GPUs	8 GPUs
NVIDIA A100 runtime	238s	121s	65s	37s
A100 parallel efficiency	100%	98.3%	91.5%	80.4%

In the first Subsection 7.3.1, we apply `Newton-CG` in a data-parallel way to multiple GPUs. With this, we observe strong scaling results for the `Newton-CG` implementation using `Horovod` for a ResNet50.

In the second Subsection 7.3.2, we compare the timings of `Newton-CG` to the regular state-of-the-art optimizers `Adam` and `SGD`. With this, we identify the additional cost for the benefit of a second-order optimization algorithm.

7.3.1 Parallel runs

Training with Keras and Horovod was used to show the applicability and scalability of the proposed second-order optimization. However, especially for large networks from image classification are notoriously hard to train from scratch. Our goal is to first use a first-order method (or a method of choice) and use second-order for fine-tuning only.

Exploiting parallelism allows for distributing work in case of failures (e.g. resilience), usage of modern compute architectures with accelerators, and ultimately, lower *time-to-solution*. All network architectures shown before can be run in parallel, in the data parallel approach explained in section 7.1.2.

For the following measurements, we fully trained the Keras ResNet50 model on the DGX-1 partition of the *LRZ*, since it is our biggest network model and, therefore, allows for the biggest parallelism gains (see Table 7.5)[19]. Note that the batch size is reduced with *GPUs*, in order to account for a similar problem to be solved when increasing the amount of *workers* (strong scaling). However, it cannot be entirely related to *strong scaling*, since the algorithm changes with workers/GPUs as explained in Figure 7.2, and the section around it. In a parallel setup, the loss is calculated for a smaller *mini-batch*, and then the *update* is accumulated. This is different to looking at a bigger *batch*, since the loss function is a different one (computed for *mini-batch* per *GPU* only).

7.3.2 Sequential timing comparisons

Similarly, for the parallel runtime analysis above, we conducted a single GPU performance study also for the two other optimizers. `SGD` and `Adam` take 255 seconds and 214 per epoch, respectively. This is in a similar runtime range as `Newton-CG` with 238s. We can assume that in this big scenario, the runtime is dominated by memory transfer, and the one vs. two backpropagations hardly make a difference.

We also tabulated the runtimes of the other runs in Table 7.6. We observe the pattern that the size of the dataset has the most considerable influence on time per epoch. Timings of optimizers are sometimes similar, and sometimes they differ significantly.

For some runs, we indicate < 1 as runtime. For us, such runtimes are negligible, and we expect that a user can be satisfied with the extra work from the `Newton-CG` scheme.

The number of CG iterations can increase runtimes proportionally by the number of iterations. Hence, for competitive timings with other optimizers, many CG-iterations should be

[19]On the LRZ cluster, we had to reduce to 20% training images for lower memory disk usage and 60% of the optimization layers, 30 layers for ResNet50.

avoided but suffice with a very coarse Newton approximation. We assume that in cases like MNIST, BreaKHis, or transformer, the computational cost of `Newton-CG` may be caused by extra `CG`-steps. We allow 20 by default, matching roughly the increase factor.

For similar runtimes between `Adam`, `SGD` and `Newton-CG` (like ResNet, MobileNet), we assume runtime is dominated by memory transfer of the dataset, not the backpropagation computations. Hence, sometimes we may afford an approximate second-order optimizer without longer runtimes, since we simply increase the computational intensity of the (already) loaded data. `Adam` and `SGD` seem to be *memory-bound* in these cases.

Table 7.6: Time-per-epoch in seconds of `Adam`, `SGD`, and `Newton-CG`

scenario description		Adam	SGD	Newton-CG
regression	life expectancy	< 1	< 1	< 1
	Boston housing	< 1	< 1	< 1
variational autoencoder	mnist	5	5	60
bayesian nn	mnist	1	1	16
	wisc	< 1	< 1	< 1
	breakhis	1	1	16
image-classification	resnet-50 imagenet	255	214	245
	mobile-net imagenet	≈ 720	≈ 690	≈ 780
	inception imagenet	60	38	100
	transfer learning	119	114	100
natural language	transformer	30	29	720

7.3.3 Summary

In addition to the convergence analysis from Section 7.2, where we show that `Newton-CG` is beneficial with respect to optimization speed, we enabled here in Section 7.3 our `Newton-CG` implementation also for Multi-GPU usage. We applied it to our biggest model, full training of a ResNet-50 Keras model on a DGX-1 with 8 GPUs. With our data-parallel approach, we achieved 80% parallel efficiency. That means, by using 8 GPUs instead of 1, we get our final result 6.43 times faster ($6.43/8 \approx 80\%$).

We have compared all sequential runtimes in Table 7.6. For some runs, we see similar runtimes of `Adam`, `SGD` and `Newton-CG` - in these cases, we can assume the algorithms are *memory-bound*, so we get the extra computations of 2^{nd}-order for free, i.e. they do not add to runtime. For small dataset models with < 1s runtime, we deem runtimes are negligible, and the gain from extra accuracy is more important. Some `Newton-CG` runtimes are, however, much longer since the number of CG-iterations adds computational costs proportionally.

Let us stress, the **two** goals of this section:

1. `Newton-CG` parallelizes well. With Horovod on 8 A100 GPUs in a NVIDIA DGX-1, we have achieved 80% parallel efficiency.

2. `Newton-CG` has small additional cost. As the analyzed algorithms are *memory-bound*, `Newton-CG` runtimes are hardly longer than for `SGD` or `Adam`.

Therefore, we believe that `Newton-CG` is very competitive to the state-of-the-art and offers to be an excellent alternative for future deep learning training algorithms.

7.4 Summary of `Newton-CG` for neural networks

In summary, we showed the applicability and benefits of an approximate 2^{nd}-order scheme on modern large-scale neural networks. True Newton schemes are limited in network size for analysis, as the memory requirements outgrow modern computing systems quickly. Our version, `Newton-CG`, applies a few CG steps to the Newton equation, so it only requires a Hessian matrix-vector product that we obtain cheaply by two backpropagations. Therefore, we were able to apply it to a broad range of deep-learning domains and large-scale architectures.

We measured predominately benefits of `Newton-CG` compared to `Adam` and `SGD` in neural network training. For example, we saw benefits in regression problems, in a Bayesian Neural Network for breast cancer diagnostics (including weight probability distributions), or in MobileNet or InceptionNet in computer vision. For some others, like other Bayesian Nets, a transfer learning example, or some NLP parts, we saw some challenges. This is natural, since 2^{nd}-order schemes are susceptible to instabilities. We observed the pattern that usually, in friendly convergence regions, `Newton-CG` performs well (inside trust/convergence regions). In contrast, in areas where the other two optimizers struggle, `Newton-CG` also struggles. For a summary, we refer to Table 7.3, which benchmarks all our scenarios and compares `Newton-CG` to other optimizers.

With Horovod, we parallelized the algorithm with data-parallelism to an 8 NVIDIA A100 GPU machine (DGX-1), achieving around 80% parallel efficiency. We also tabulated all other runtimes and observed sometimes only a small additional cost of the 2^{nd}-order scheme, but sometimes a large discrepancy between time-per-epoch due to a challenging region in optimization. We believe that these optimizers are memory-bound, and hence, this would allow for more computations without additional runtime. However, `Newton-CG` runtime scales proportionally with a number of `CG`-iterations, resulting sometimes in larger runtimes.

We show that `Newton-CG` is competitive to state-of-the-art optimizers with concepts like Tikhonov regularization and Armijo step size restriction. For regression problems, vision networks, or fine-tuning, it can be very beneficial. With that `Newton-CG`, or other 2^{nd}-order algorithms, are imperative in benchmarking neural network training algorithms, e.g. [DSN+23], or in the large study "Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers" [SSH21]. For PyTorch, the library ASDL offers some approximate second-order primitives[20] [OIY+23]. Second-order can also help in fine-tuning and model analysis for having fewer trainable parameters, e.g. using the low-rank adaption for GPT and large language models. [HSW+21]

[20] `https://github.com/kazukiosawa/asdl`

PART IV

CONCLUSION AND OUTLOOK

Truth is ever to be found in the simplicity and not in the multiplicity and confusion of things.
Isaac Newton (1642 - 1726)

8 Algorithmic impact of scientific computing on machine learning

This thesis made several contributions related to $\mathcal{H}$-matrices and second-order optimization for neural networks. After the previous result chapters, in this chapter, we *summarize*, *discuss*, and *conclude* the key findings. We start with a summary and discussion in Section 8.1 of both parts of the thesis, $\mathcal{H}$-Matrices and Newton-CG, and sketch out a possible outlook into the future in Section 8.2.

8.1 Conclusion

In Chapter 6 called $\mathcal{H}$-matrix we have analyzed many dense matrices and observed that `GOFMM` can outperform an exact `MKL DGEMM` or the structured matrix package `STRUMPACK`. In Chapter 7 called `Newton-CG`, we have observed a very problem-dependent behavior for several deep learning (DL) networks and seen cases where `Newton-CG` can outperform the state-of-the-art stochastic gradient descent (`SGD`) or adaptive moments (`Adam`). While we treat the $\mathcal{H}$-matrix and second-order optimization for neural networks as separate chapters in this thesis, there are several building blocks where the topics overlap. In particular, we have analyzed several Gauss-Newton Hessians from auto-encoder networks in Subsection 6.4.3. We have avoided the combination of a `GOFMM` treatment in the second-order training process due to runtime and memory overhead. While there are experiments with `GOFMM` to compute entries on the fly or save them in memory out-of-core, this creates tremendous overhead. On the one hand, `GOFMM` currently does not allow a matrix-vector product as matrix input. On the other hand, the DL community is very reluctant to outer-community change (with a software stack), and effects alter depending on the size and architecture. Hence, to analyze such problem-agnostic effects, we have developed `Newton-CG`, that only requires a *matvec* and avoids the expensive part of setting up and compression (required by `GOFMM`). We also refer to the concurrent goals in the simulation triangle, see Section 8.2 and Figure 8.1.

8.1.1 $\mathcal{H}$-Matrices with GOFMM

The numerical treatment of large-sized matrices, in particular, of fully populated matrices, suffers from quadratic or cubic cost concerning storage and matrix operations. To address this, on the one hand, setting up the matrix could be avoided by mechanistic modeling of the geometric domain and with domain-specific approximations, if necessary (white-box approach). We, on the other hand, developed a black-box approach that, given a matrix, finds structure, applies compression, and saves runtime for large matrices with a complexity of $\mathcal{O}(N \log N)$.

`GOFMM` uses a geometry-oblivious notion to compute distances for reordering and sampling. We have empirically demonstrated the complexity of $\mathcal{O}(N \log N)$. Depending on the use case, we saw around 1000 multiplications (called `nrhs`) benefits starting at $N = 6,000$. We performed weak and strong scaling measurements up to 128 nodes, finding a sweet spot around 16 nodes with about 50% parallel efficiency for multiplication. We also looked at a Gauss-Newton Hessian regarding multiplication and factorization. Next, we interfaced `GOFMM` to `numpy` and analyzed matrices from diffusion maps, computed with `datafold`. We have employed the `GOFMM`'s fast matrix-vector for an iterative Arnoldi eigendecomposition, achieving (depending on parameters) around $9E-4$ accuracy in the Froebenius norm.

In summary, we observed the promised $\mathcal{O}(N \log N)$ complexity for $\mathcal{H}$-matrices. The black-box modeling is very *non-intrusive* for simulation codes. `GOFMM` outperforms state-of-the-art methods for large sizes and some cases. In the future, with more high-dimensional domains in data science or other fields, we believe that more $\mathcal{H}$-type algorithms are used. `GOFMM` is an excellent candidate and a valuable resource for future development.

8.1.2 2^{nd}-order optimizer Newton-CG

In the second part of the thesis, we developed `Newton-CG`. We show benefits in training epochs for shallow networks, as well as some image classification and transformer architectures. With Bayesian neural networks, we explored weight uncertainties. We observed very problem-dependent behavior in some cases, where `Newton-CG` does not satisfy our high expectations. We parallelized `Newton-CG` with *Horovod*, a framework for data parallelism. With data-parallelism, we achieved around 80% parallel efficiency on 8 GPUs, and generally, `Newton-CG` is as fast as `SGD` and `Adam`.

This second-order analysis can accelerate neural network training and help in the explainability of deep learning architectures. Since it is written in Tensorflow for Python, it is *non-intrusive* and can be integrated in machine learning pipelines. It is common in machine learning to tune different hyperparameter sets to judge the validity of the network and training state, exhausting tremendous computing resources. We have experienced that `Newton-CG` is less prone to such effects, and `Newton-CG` can help in error estimation or be used for fine-tuning the optimization state.

In summary, we observed the pattern of `Newton-CG` that usually, in well-formed trust/convergence regions, it performs on-par or better than `ADAM` and `SGD`. In contrast, in areas where the other two optimizers struggle, `Newton-CG` also struggles.

However, limitations of the 2^{nd}-order optimizer for DL must be noted. The analyzed problems are very high-dimensional and, therefore, suffer from non-smoothness, potentially leading to the observed *problem-dependent* behavior. As there occurred several "AI winters", i.e. periods of reduced funding and interest in artificial intelligence research, currently, second-order optimization for neural networks is in winter times, while general AI is in summer times. There has been a spike in second-order optimization for DL around 2018, but currently, it seems to be adopted by the industry for fine-tuning models, for analysis, or in scenarios of sparse data. It is not practiced in mostly low-accurately trained current scenarios for academia. However, also in academia, another nature of problems could be needed, in particular when the problem ensures smoothness in the optimization surface, or another mathematical scheme could post-smoothen it. Then, this debate about the broad applicability of the 2^{nd}-order optimizer for DL training could be reinvigorated.

8.2 Outlook

This thesis has an interdisciplinary appeal: scholars from different computational subfields in computer science, mathematics, physics, or engineering will find it of particular interest [BRU+20].

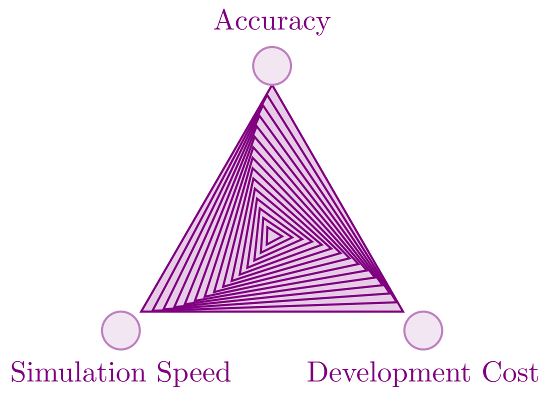


Figure 8.1: Concurrent goals in the simulation triangle. Accuracy vs. Simulation Speed vs. Development Speed. We added the inner rotating triangles, as there may be additional dimensions (computational resources, external factors) or skewness.

We embark on exploring innovative mathematical concepts and analyze them through implementation and on a wide range of problems. Our focus extends to addressing their parallel performances and optimizing computational efficiency.

Let us analyze the thesis with the concurrent goals in the simulation triangle, see Figure 8.1. It poses the question, how to adaptively approach a global minimum for resource requirements given a prescribed total target (accuracy and precision). The triangle may be interpreted as: `low` development cost often comes with `low` simulation speed and `low` accuracy. For achieving `high` accuracy in simulation models, either `high` simulation speed (long runtime) or `high` development cost (or a balance) must be spent. In other words, as economics claims, there is no free lunch. If the last performance percentages are tweaked through mechanistic modeling in a white-box approach, the approach is very problem-depending. With empirical modeling, we address a more problem-agnostic black-box approach that balances accuracy, simulation speed and development speed.

We have encountered several times the concurrency between accuracy and runtime and its interplay in this thesis. As changes take time, our methods in this thesis are not yet adopted widely, and changes at different levels may be necessary. We show that they *already* can outperform existing methods for some cases, but definitely are *already* very competitive for a wide range of problems. Our ideas and methods address pressing issues in the field, and maybe they are yet just a stone throw in the right direction. We strongly believe that the algorithms will be adopted in the coming years.

As for the future, on the one end, the application of `GOFMM` has been made accessible for experimentation, allowing users to assess the potential value of $\mathcal{H}$-matrices for incorporation into a simulation code. A swift evaluation can be performed using NumPy, while the `C++` codebase can be used for high-performance. On the other end, `Newton-CG` can be used to analyze a wide range of differentiable optimization problems in DL or other fields. It is non-intrusive and can be integrated into data pipelines in a plug-and-play fashion. There remain many interesting future problems where this second-order scheme can help in training, fine-tuning, or for analysis purposes.

Proprietary software is an injustice. Sharing is good, and with digital technology, sharing is easy.

Richard Stallman (*1953)

A SWIG Code compilation

The GOFMM C++ code originated in a collaboration with George Biros' group, in particular Chenhan Yu and James Levitt. In addition to our contributions there, we wrote a SWIG file to convert the GOFMM code from C++ into Python. To access it publicly, it's easiest to also use the GitHub version, `https://github.com/severin617/gofmm_swig_python`.

For using the C++ variant of GOFMM, we refer to the ReadMe for the GOFMM C++ version. The package includes other codes, hence he called it "High-Performance Machine Learning Primitives", on GitHub here `https://github.com/ChenhanYu/hmlp`.

A.1 GOFMM files

SWIG is a software tool that connects programs written in C or C++ with scripting languages such as Python or R. With SWIG, we are able to call the *main* function on the Python command line for example. However, one major obstacle for compiling the SWIG file is linking. We need to link the proper Python version to some indispensable shared object files in order to generate the Python wrapper. We added commentary to the linking part in the SWIG file *swig.sh* for reference below, which should work straightforwardly on LRZ's Linux cluster .

Now, we need to pack a user configuration for arguments at the command line into a vector of char pointer. We will explain this procedure in the framework of our augmented GOFMM prototype in Listing A.1.

Listing A.1: Adaptations of GOFMM in `test_gofmm.cpp`

```
class gofmmTree {
  /* This data structure gofmmTree enables storage of configuration
    parameters needed to execute mainly the following two operations:
    1. Mat-vec multiplication
    2. GOFMM Pseudo-inverse
  */
 private:
  SPDMATRIX_DENSE K;  // SPD Matrix
  std::vector<const char*> argv;  // holder for configuration parameters

  /* Configuration parameters */
  std::string executable;  // ./test_gofmm
  int n;  // problem size
  int m;  // maximum leaf node size
  int k;  // number of neighbors
  int s;  // maximum off-diagonal ranks
  int nrhs;  // number of right hand sides
  T stol;  // user tolerance
  T budget;  // user computation budget [0,1]
  std::string distance;  // distance type (geometry, kernel, angle)
  // spdmatrix type (testsuit, dense, ooc, kernel, userdefine)
  std::string matrixtype;
  std::string kerneltype;  // kernelmatrix type (gaussian, laplace)

 public:
  /* Constructors and destructors*/
  gofmmTree();  // default constructor

  // User-defined constructor: fill the inputs into the object fields above
  gofmmTree(std::string executableName, int n0, int m0, int k0, int s0,
            int nrhs0, T stol0, T budget0, std::string distanceName,
            std::string matrixtypeName, std::string kerneltypeName,
            SPDMATRIX_DENSE K0);

  // Convert all initialized configuration parameters to a vector (argv)
  void convert_to_vector();

  /* Operations: mul + inverse */
  void mul_denseSPD(DATA_s w, double* mul_numpy, int len_mul_numpy);
  void invert_denseSPD(T lambda, double* inv_numpy, int len_inv_numpy);
};
```

A.2 SWIG file `tools.i`

In order to wrap a code for Python, SWIG requires a tools.i files to specify which functions in particular need to be wrapped. A challenge was to get the right notation for numpy arrays, that you will find in the `tools.i` file on GitHub. We here list the necessary lines for the routine *load_denseSPD_from_console*

Listing A.2: SWIG snippet for `load_denseSPD_from_console`

```
%apply (float* IN_ARRAY2, int DIM1, int DIM2) \  // IN_ARRAY2: Input 2D
      {(float* numpyArr, int row_numpyArr, int col_numpyArr)}
```

Then, for the SWIG wrapper for *mul_denseSPD*, we also require integers and numpy arrays, see Listing A.3.

Listing A.3: SWIG snippet for *mul_denseSPD*

```
%apply (double* ARGOUT_ARRAY1, int DIM1) \  // ARGOUT: output array
      {(double* mul_numpy, int len_mul_numpy)}
```

ARGOUT_ARRAY1 specifies that the argument *mul_numpy* is a 1D output array. This means, it should be the return value of the routine. One may ask why we choose a 1D array as the return instead of the 2D since the multiplication Kw has the 2D type. The reason is that there is no default typemap for 2D array as an output although there is one as an input. As a result, we need to resize the 1D flattened array into 2D afterwards. This is easy to accomplish in Python.

A.3 Compilation and installation

In order to use SWIG please install it from their website: `https://swig.org/`. Then, the C++ version of GOFMM must be compiled, and then compiled and linked with the libraries of swig. For this, we wrote `swig.sh`, amongst others, see the GitHub page.

As mentioned in Section 6.1, it was cumbersome to find the respective files and link it, so it can be properly used. The linking flags for the Linux cluster are listed below, and can be used for reference. The link line advisor by Intel is also a great resource: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html`

Listing A.4: SWIG snippet for `swig.sh`

```
cp tools.i build/
cd build
swig -o toolswrap.cpp -c++ -python tools.i
#  Create a wrapper file for tools
# -I/usr/include/python3.8: for Python.h file
# -I/home/getianyi/.local/lib/python3.8/site-packages/numpy/core/include/: for
    arrayobject.h
g++ -o tools_wrap.os -c -I/usr/include/python3.8 -I/home/getianyi/.local/lib/
    python3.8/site-packages/numpy/core/include/ -I../gofmm/ -I../include/ -I../
    frame/ -I ../frame/base/ -I../frame/containers/ toolswrap.cpp -fPIC
# Create a shared library. Note: we must link the lapack and blas libs
# at run time together with hmlp in order to construct a python-portable
# _tools.so
g++ -O3 -fopenmp -m64 -fPIC -D_POSIX_C_SOURCE=200112L -fprofile-arcs -ftest-
    coverage -fPIC -DUSE_BLAS -mavx -std=c++11  -lpthread -fopenmp -lm -L/usr/
    local/lib/ -lopenblas tools_wrap.os  -o _tools.so -shared  -Wl,-rpath,./
    build: libhmlp.so -Wl,-rpath,./build: libhmlp.so -Wl,-rpath,/usr/local/lib -
    lblas -Wl,-rpath,/usr/local/lib -llapack
```

For the Python Package Index (PyPI) installation under a different Linux configurations, the user first must install all necessary dependencies such as the library `libblas.so`. Then, one can install it from `https://pypi.org/project/gofmm1/`. We cannot only depend on the PyPi installation, as we require `libhmlp.so`, which is bound to your architecture (Linux configuration). Software publishers usually have pre-compiled versions for all necessary operating systems under different "wheels"; since we are a small academic group, we cannot provide "wheels" for any operating system without great effort.

If you are reading this and you want to discuss upon it, I will buy you a coffee or a beer.

Severin Reiz (*1992)

B Newton-CG installation

For reproducibility, we also share Newton-CG freely, e.g. on GitHub, see `https://github.com/severin617/Newton-CG`. To increase outreach, a user can also get it from the Python Package Index, from here `https://test.pypi.org/simple/newton-cg/`.

B.1 Installation

To install Newton-CG directly, just do the following commands, as we recommended to use it with a new conda environment:

Listing B.1: Install `Newton-CG`

```
conda create -n tf1 python=3.7
conda activate tf1
pip install -r tensorflow==1.15 keras==2.3
pip install -i https://test.pypi.org/simple/ newton-cg==0.0.3
```

B.2 Usage

You can use easily load it and use the optimizer in any existing `tensorflow` or `keras` script, or also, for `tensorflow_probability` in Bayesian Neural Networks:

Listing B.2: Instantiate EHNewtonOptimizer (Efficient Hessian Newton)

```
import newton_cg as es
optimizer = es.EHNewtonOptimizer(
        learning_rate,
        tau=FLAGS.eso_tau,
        cg_tol=FLAGS.eso_cg_tol,
        max_iter=FLAGS.eso_max_iter)
```

You can use also a learning-rate scheduler, and call model.fit , which is a Tensorflow API to start the optimization process. You can also perform training step-by-step for each Newton-step.

Listing B.3: Adapt the learning rate and call model.fit

```
# lr scheduler
from clr_callback import CyclicLR
clr_trig2 = CyclicLR(mode='exp_range', base_lr=0.00001, max_lr=0.0001,
    step_size=1, gamma=0.5) #gamma=0.9994)

model.fit(data_X, data_Y,  epochs=1, callbacks=[clr_trig2])
```

For parallelization, you can use the data-parallel approach of `horovod`, like this:

Listing B.4: Parallel `Newton-CG` with Horovod

```
import newton_cg as es
import horovod as hvd
hvd.init()
# Horovod: pin GPU to be used to process local rank (one GPU per process)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = str(hvd.local_rank())
K.set_session(tf.Session(config=config))
opt = EHNewtonOptimizer(initial_lr)
es.opt = hvd.DistributedOptimizer(opt, compression=compression)
```

Beware that alongside `horovod` setup with `MPI` and suitable `CUDA` drivers is tedious. You may get a docker image from NVIDIA directly[1], but we had to nevertheless make major adaptions here and there. We refer to the documentation and help from stackoverflow, as the data science community is very large.

In the GitHub version, we also added the archictecures as Tensorflow/Keras script. For development, we have used the GitLab repository, i.e. `https://gitlab.lrz.de/tum-i05/public/keras-second-order-optmizer`.

The datasets for computer vision (`Imagenet`) and for NLP model (Portuguese-English Translation), requires large memory (and possibly small `batch-size`).

Also, the program could need a long time to run due to `Imagenet` size (You may use pre-trained checkpoint file that you find e.g. here: `https://github.com/tensorflow/models/tree/master/research/slim`). We, howerver, have trained from scratch and have the corresponding checkpoint files on our file systems (we can share upon request).

For natural language processing, we currently only have it on GitLab, which is public to anyone after accepting the terms and conditions, see `https://gitlab.lrz.de/tum-i05/public/nlp-newton-cg`. For some models it is required to unroll recurrent structures in the model (`RNN`) and using pre-trained word embeddings. Hyperparameters can be set in the JSON file called `config.json` , see here: `https://gitlab.lrz.de/tum-i05/public/nlp-newton-cg/-/blob/main/config.json`.

We suggest to start using Newton-CG, in the straightforward regression, small `MNIST` CNN or Bayesian Neural Network models, as they run fast and require less memory.

[1]ngc.nvidia.com

The secret to creativity is knowing how to hide your sources.

Albert Einstein (1879 - 1955)

Bibliography

[AD13] Sivaram Ambikasaran and Eric Darve. An $\mathcal{O}(N\ log(N))$ Fast Direct Solver for Partial Hierarchically Semi-Separable Matrices. *Journal of Scientific Computing*, 57(3):477–501, 2013. 21

[Alh22] Osama Alhartani. Transfer Learning and Dynamic Loading in TUM-Lens. Master's Thesis, Technical University of Munich, 2022. 73

[Arn51] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951. 21

[AZH$^+$21] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021. 41

[B$^+$95] Christopher M Bishop et al. *Neural networks for pattern recognition.* Oxford university press, 1995. 37, 42, 44

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. 55

[BCFW21] Kristian Bredies, Marcello Carioni, Silvio Fanzon, and Daniel Walter. Linear convergence of accelerated generalized conditional gradient methods. *arXiv preprint arXiv:2110.06756*, 2021. 46, 120

[BCKW15] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural network. In *International conference on machine learning*, pages 1613–1622. PMLR, 2015. 44

[BCN18] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018. 67

[Beb08] Mario Bebendorf. *Hierarchical matrices.* Springer, 2008. 13, 21, 28, 29

[BG99] Steffen Börm and Lars Grasedyck. H-Lib-a library for H-and H2-matrices. *Max Planck Institute for Mathematics in the Sciences.[Online]. Available: http://www. hlib. org*, 1999. 23

[BGJM17] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information, 2017. 52

[BH86] Josh Barnes and Piet Hut. A hierarchical $\mathcal{O}(N \log N)$ force-calculation algorithm. *Nature*, 324(6096):446–449, 1986. 20, 21, 24, 83

[BHNS16] R. H. Byrd, S. L. Hansen, Jorge Nocedal, and Y. Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016. 68

[BM01] Stefania Bellavia and Benedetta Morini. A globally convergent newton-gmres subspace method for systems of nonlinear equations. *SIAM Journal on Scientific Computing*, 23(3):940–960, 2001. 67

[BNN$^+$20] Hans-Joachim Bungartz, Wolfgang E. Nagel, Philipp Neumann, Severin Reiz, and Benjamin Uekermann. Software for exascale computing: Some remarks on the priority program sppexa. In Hans-Joachim Bungartz, Severin Reiz, Benjamin Uekermann, Philipp Neumann, and Wolfgang E. Nagel, editors, *Software for Exascale Computing - SPPEXA 2016-2019*, pages 3–18, Cham, 2020. Springer International Publishing. 16

[BRB17] Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical Gauss-Newton optimisation for deep learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 557–565. PMLR, 06–11 Aug 2017. 68

[BRU$^+$20] Hans-Joachim Bungartz, Severin Reiz, Benjamin Uekermann, Philipp Neumann, and Wolfgang E Nagel. *Software for exascale computing-SPPEXA 2016-2019*. Springer Nature, 2020. 138

[BSH$^+$21] Liane Bernstein, Alexander Sludds, Ryan Hamerly, Vivienne Sze, Joel Emer, and Dirk Englund. Freely scalable and reconfigurable optical hardware for deep learning. *Scientific reports*, 11(1):3144, 2021. 64

[BYC13] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013. 119

[CDL16] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading, 2016. 55

[CL06] Ronald R. Coifman and Stéphane Lafon. Diffusion maps. *Applied and Computational Harmonic Analysis*, 21:5–30, 2006. Special Issue: Diffusion Maps and Wavelets. 33

[CLRB17] D Yu Chenhan, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for Compressing of Dense Matrices. In *Proceedings of SC17*, 2017. 82, 91, 92, 95, 107

[CRB18] D Yu Chenhan, Severin Reiz, and George Biros. Distributed-memory hierarchical compression of dense spd matrices. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 183–197. IEEE, 2018. 87, 91, 92, 95, 98, 111

[CRY$^+$21] Chao Chen, Severin Reiz, Chenhan D Yu, Hans-Joachim Bungartz, and George Biros. Fast approximation of the gauss–newton hessian matrix for the multilayer perceptron. *SIAM Journal on Matrix Analysis and Applications*, 42(1):165–184, 2021. 91, 92, 94, 100, 101, 111

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 53

[DECM98] Amalia Duch, Vladimir Estivill-Castro, and Conrado Martinez. Randomized k-dimensional binary search trees. In *Algorithms and Computation: 9th International Symposium, ISAAC'98 Taejon, Korea, December 14–16, 1998 Proceedings 9*, pages 198–209. Springer, 1998. 29

[Dre21] David Drews. Implementing a Mobile App for Object Detection. Master's thesis, Technical University of Munich, 2021. 73

[DSN$^+$23] George E Dahl, Frank Schneider, Zachary Nado, Naman Agarwal, Chandramouli Shama Sastry, Philipp Hennig, Sourabh Medapati, Runa Eschenhagen, Priya Kasimbeg, Daniel Suo, et al. Benchmarking neural network training algorithms. *arXiv preprint arXiv:2306.07179*, 2023. 64, 68, 133

[Dul24] Saurab Dulal. octree construction and nearest neighborhood search(nns). `https://dulalsaurab.github.io/computerscience/octree-construction-and-nns/`, retrieved on Jan 14 2024. 26

[DYBM23] Sameer Deshmukh, Rio Yokota, George Bosilca, and Qinxiang Ma. O (n) distributed direct factorization of structured dense matrices using runtime systems. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 1–10, 2023. 23, 86, 103

[FKR22] Massimo Fornasier, Timo Klock, and Konstantin Riedl. Convergence of anisotropic consensus-based optimization in mean-field law. In Juan Luis Jiménez Laredo, J. Ignacio Hidalgo, and Kehinde Oluwatoyin Babaagba, editors, *Applications of Evolutionary Computation*, pages 738–754, Cham, 2022. Springer International Publishing. 63, 119

[G$^+$16] Yarin Gal et al. *Uncertainty in deep learning*. PhD thesis, University of Cambridge, 2016. 43

[Gad22] Keerthi Gaddameedi. Efficient and scalable kernel matrix approximation using hierarchical decomposition. Master's thesis, Technical University of Munich, 2022. 89, 90

[GBC16a] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 62, 63

[GBC16b] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`. 39

[Ge20] Tianyi Ge. Python software suite of geometric-oblivious fmm. Master's thesis, TUM informatics, 2020. 32

[GK65] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965. 28

[GLR$^+$16] Pieter Ghysels, Xiaoye S Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016. 99

[GR87] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987. 20, 21, 83

[GRNB23] Keerthi Gaddameedi*, Severin Reiz*, Tobias Neckel, and Hans-Joachim Bungartz. Efficient and scalable kernel matrix approximations using hierarchical decomposition. *accepted for publication in conference issue for BenchCouncil IC 2023 in Springer CCIS*, 2023. 33, 34, 91, 103, 105

[GS22] Pieter Ghysels and Ryan Synk. High performance sparse multifrontal solvers on modern GPUs. *Parallel Computing*, 110:102897, 2022. 24

[GTY97] SA Goreinov, EE Tyrtyshnikov, and A Yu Yeremin. Matrix-free iterative solution strategies for large dense linear systems. *Numerical linear algebra with applications*, 4(4):273–294, 1997. 21

[GZDH20] Matilde Gargiani, Andrea Zanelli, Moritz Diehl, and Frank Hutter. On the promise of the stochastic generalized gauss-newton method for training dnns. *arXiv preprint arXiv:2006.02409*, 2020. 68

[Hac15] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis.* Springer Series in Computational Mathematics 49. Springer-Verlag Berlin Heidelberg, 1 edition, 2015. 13, 22, 23, 24

[HMT11] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011. 85

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 52

[Hsi21] Yi-Han Hsieh. Second order training for natural language processing using newton-cg optimizer. Master's thesis, Technical University of Munich, Oct 2021. 52, 55, 129, 130

[HSW$^+$21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 133

[Huc96] Thomas Huckle. Sparse approximate inverses for preconditioning of linear equations. In *Conferentie van Numeriek Wiskundigen, Woudschoten, Zeist, The Netherlands.* Citeseer, 1996. 23

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 46, 47

[INK18] Akihiro Ida, Hiroshi Nakashima, and Masatoshi Kawai. Parallel hierarchical matrices with block low-rank representation on distributed memory computer systems. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 232–240, 2018. 24

[JM09] Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition.* Pearson Prentice Hall, Upper Saddle River, N.J., 2009. 52

[Jok20] Lukas Maximilian Jokel. Implementing a tensorflow-slim based android app for image classification. Bachelor thesis, Technical University of Munich, 2020. 73

[Kar22] Maximilian Karpfinger. Sign Language Translation on Mobile Devices. Master's thesis, Technical University of Munich, 2022. 73, 77

[KG17] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? *Advances in neural information processing systems*, 30, 2017. 43

[KH$^+$09] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009. 56, 94

[KK04] D.A. Knoll and D.E. Keyes. Jacobian-free newton–krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397, 2004. 61

[Kre] Daniel Kressner. Low rank approximation lecture 1. https://sma.epfl.ch/ anchp-common/lecture1.pdf. 22

[(LA07] Los Alamos Computer Science Institute (LACSI). Arpack software. http://lacsi.rice.edu/software/arpak/default.htm, 2007. 90

[LBG06] Sabine Le Borne and Lars Grasedyck. H-matrix preconditioners in convection-dominated problems. *SIAM journal on matrix analysis and applications*, 27(4):1172–1183, 2006. 24

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015. 39

[LeC98] Yann LeCun. The mnist database of handwritten digits. *Website*, 1998. 94, 95, 106, 122

[LFTL20] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. A review of applications in federated learning. *Computers & Industrial Engineering*, 149:106854, 2020. 73

[Li] Fei-Fei Li. Cs231n: Convolutional neural networks for visual recognition. 56, 57, 58

[Li21] X Sherry Li. Scalable and robust hierarchical matrix factorizations via randomization. In *Conference on Fast Direct Solvers (Online)*, 2021. 21

[LLH$^+$23] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training, 2023. 128

[LPM15] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015. 55

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015. 46

[LWM$^+$07] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172, 2007. 28, 29

[LXG$^+$21] Yang Liu, Xin Xing, Han Guo, Eric Michielssen, Pieter Ghysels, and Xiaoye Sherry Li. Butterfly factorization via randomized matrix-vector multiplications. *SIAM Journal on Scientific Computing*, 43(2):A883–A907, 2021. 24

[MB15] William B March and George Biros. Far-field compression for fast kernel summation methods in high dimensions. *Applied and Computational Harmonic Analysis*, 2015. 31

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. 52

[Mei11] Andreas Meister. *Numerik linearer gleichungssysteme*, volume 4. Springer, 2011. 61

[MG15] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015. 68, 100

[MGB$^+$17] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. *arXiv preprint arXiv:1712.09405*, 2017. 129

[Mov22] Danylo Movchan. Implementing a learning-rate scheduler in a newton-cg optimizer for deep learning. Bachelor's thesis, Technical University of Munich, Oct 2022. 129

[MXT$^+$15a] William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. A kernel-independent fmm in general dimensions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. 86

[MXT$^+$15b] William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. Robust treecode approximation for kernel machines. In *Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1–10, Sydney, Australia, August 2015. 94

[Nad65] EA Nadaraya. On non-parametric estimates of density functions and regression curves. *Theory of Probability & Its Applications*, 10(1):186–190, 1965. 32

[Ng16] Andrew Ng. Nuts and bolts of building ai applications using deep learning. *NIPS Keynote Talk*, 2016. 58

[NN94] Yoshiki Niwa and Yoshihiko Nitta. Co-occurrence vectors from corpora vs. distance vectors from dictionaries. In *COLING 1994 Volume 1: The 15th International Conference on Computational Linguistics*, 1994. 52

[OIY$^+$23] Kazuki Osawa, Satoki Ishikawa, Rio Yokota, Shigang Li, and Torsten Hoefler. ASDL: A Unified Interface for Gradient Preconditioning in PyTorch, 2023. 133

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015. 41

[OTU$^+$19] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019. 68

[Par62] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962. 32

[Pea94] Barak A Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1):147–160, 1994. 15, 64, 65

[PPR18] Ajeet Ram Pathak, Manjusha Pandey, and Siddharth Rautaray. Application of deep learning for object detection. *Procedia computer science*, 132:1706–1717, 2018. 76

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014. 52

[PVG$^+$11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 106

[PY09] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009. 57

[QSF$^+$18] Ye Qi, Devendra Singh Sachan, Matthieu Felix, Sarguna Janani Padmanabhan, and Graham Neubig. When and why are pre-trained word embeddings useful for neural machine translation?, 2018. 52

[Ram99] Juan Ramos. Using TF-IDF to Determine Word Relevance in Document Queries, 1999. 52

[RB20] Severin Reiz and Hans-Joachim Bungartz. SPPEXA: Software for Exascale Computing. *SIAM News*, Volume 53(Oct. 2020), 2020. 16

[Rei17] Severin Reiz. Black Box Hierarchical Approximations for SPD Matrices. Master's thesis, TUM, Apr 2017. betreuer: Levitt, James; pruefer: Biros, George; Bungartz, Hans-Joachim. 26, 27, 28, 29

[Rie92] Kurt S Riedel. A sherman–morrison–woodbury identity for rank augmenting matrices with application to centering. *SIAM Journal on Matrix Analysis and Applications*, 13(2):659–662, 1992. 23

[RLGN16] François-Henry Rouet, Xiaoye S Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions on Mathematical Software (TOMS)*, 42(4):1–35, 2016. 24, 29, 98, 103

[RLXM22] Fred Roosta, Yang Liu, Peng Xu, and Michael W. Mahoney. Newton-mr: Inexact newton method with minimum residual sub-problem solver. *EURO Journal on Computational Optimization*, 10:100035, 2022. 68

[RNB23] Severin Reiz, Tobias Neckel, and Hans-Joachim Bungartz. Neural nets with a newton conjugate gradient method on multiple gpus. In *Parallel Processing and Applied Mathematics: 14th International Conference, PPAM 2022, Gdansk, Poland, September 11–14, 2022, Revised Selected Papers, Part I*, pages 139–152. Springer, 2023. 38, 39, 40, 62, 64, 67, 122

[Rui18] Pablo Ruiz Ruiz. Understanding and visualizing resnets–towards data science. *en línea]*, 2018. 47

[S$^+$94] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994. 21, 66

[Sah] Sumit Saha. A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way. `https://towardsdatascience.com`. 40

[SB18] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018. 112

[SBC15] Weijie Su, Stephen Boyd, and Emmanuel J. Candes. A differential equation for modeling nesterov's accelerated gradient method: Theory and insights, 2015. 63

[SHK$^+$14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. 42

[SHZ$^+$18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018. 48

[SLJ$^+$15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. 49

[SLL19] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. A comprehensive guide to bayesian convolutional neural network with variational inference. *arXiv preprint arXiv:1901.02731*, 2019. 44

[SOPH15] Fabio A Spanhol, Luiz S Oliveira, Caroline Petitjean, and Laurent Heutte. A dataset for breast cancer histopathological image classification. *Ieee transactions on biomedical engineering*, 63(7):1455–1462, 2015. 116

[SOPH16] Fabio Alexandre Spanhol, Luiz S Oliveira, Caroline Petitjean, and Laurent Heutte. Breast cancer histopathological image classification using convolutional neural networks. In *2016 international joint conference on neural networks (IJCNN)*, pages 2560–2567. IEEE, 2016. 124

[Sor92] Danny C Sorensen. Implicit application of polynomial filters in ak-step arnoldi method. *Siam journal on matrix analysis and applications*, 13(1):357–385, 1992. 34

[SPP20] Jan Steinbrener, Konstantin Posch, and Jürgen Pilz. Measuring the uncertainty of predictions in deep neural networks with variational inference. *Sensors*, 20(21):6011, 2020. 43

[SRASC14] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014. 57, 58

[SS86] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986. 21

[SSH21] Robin M Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2021. 64, 68, 133

[Ste83] Trond Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM Journal on Numerical Analysis*, 20(3):626–637, 1983. 61, 67

[Suk20] Julian Suk. Application of second-order optimisation for large-scale deep learning. Masterarbeit, TUM, May 2020. 67

[SVI$^+$16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016. 49, 50

[SW23] David Schneller and Mario Wille. Personal conversation. *Personal Conversation*, Dec 2023. 72

[tfpa] tfp.layers.Convolution2DFlipout. `https://www.tensorflow.org/probability/api_docs/python/tfp/layers/Convolution2DFlipout`. Accessed 8-12-22. 45, 114

[tfpb] tfp.layers.DenseVariational. `https://www.tensorflow.org/probability/api_docs/python/tfp/layers/DenseVariational`. Accessed 8-12-22. 45, 114

[Tir] Nikhil Tirumala. Evolution of object detection networks. `https://cogneethi.com/evodn/object_detection_intro/`. 74

[TS10] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010. 58

[tut] TF NLP Transformer tutorial. `https://www.tensorflow.org/text/tutorials/transformer`. Accessed 8-12-23. 54, 55

[Uns23] Michael Unser. Ridges, neural networks, and the radon transform. *Journal of Machine Learning Research*, 24(37):1–33, 2023. 120

[UU12] Michael Ulbrich and Stefan Ulbrich. *Nichtlineare Optimierung.* Springer-Verlag, 2012. 67

[VGO$^+$20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, et al. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python.* Nature Methods, 2020. 89, 90

[VNRP19] Abdul Vahab, Maruti S Naik, Prasanna G Raikar, and SR Prasad. Applications of object detection system. *International Research Journal of Engineering and Technology*, 6(4):4186–4192, 2019. 76

[VSP$^+$17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. 52, 53, 55, 56

[VVMA19] Mariia Vladimirova, Jakob Verbeek, Pablo Mesejo, and Julyan Arbel. Understanding priors in bayesian neural networks at the unit level. In *International Conference on Machine Learning*, pages 6458–6467. PMLR, 2019. 44

[WDE07] Martin Weiser, Peter Deuflhard, and Bodo Erdmann. Affine conjugate adaptive newton methods for nonlinear elastomechanics. *Optimisation Methods and Software*, 22(3):413–431, 2007. 61, 67

[Wei21] Hanna Weigold. Second-order optimization methods for bayesian neural networks. Master's Thesis, Technical University of Munich, 2021. 43, 116

[WN$^+$99] Stephen Wright, Jorge Nocedal, et al. Numerical optimization. *Springer Science*, 35(67-68):7, 1999. 61, 67

[WVB$^+$18] Yeming Wen, Paul Vicol, Jimmy Ba, Dustin Tran, and Roger Grosse. Flipout: Efficient pseudo-independent weight perturbations on mini-batches. *arXiv preprint arXiv:1803.04386*, 2018. 45, 114

[YCBL14] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014. 56, 58

[YGKM20] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W Mahoney. Pyhessian: Neural networks through the lens of the hessian. In *2020 IEEE international conference on big data (Big data)*, pages 581–590. IEEE, 2020. 64, 112

[YGS$^+$21] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. Adahessian: An adaptive second order optimizer for machine learning. In *proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10665–10673, 2021. 47, 68, 126

[YNDT18] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9(4):611–629, 2018. 39, 41, 42

[YRB19] Chenhan D. Yu, Severin Reiz, and George Biros. Distributed o(n) linear solver for dense symmetric hierarchical semi-separable matrices. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 1–8, 2019. 24, 86, 91, 92, 95

[ZFM$^+$20] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven Chu Hong Hoi, et al. Towards theoretically understanding why sgd generalizes better than adam in deep learning. *Advances in Neural Information Processing Systems*, 33:21285–21296, 2020. 68, 130

[ZKV$^+$20] Jingzhao Zhang, Sai Praneeth Karimireddy, Andreas Veit, Seungyeon Kim, Sashank Reddi, Sanjiv Kumar, and Suvrit Sra. Why are adaptive methods good for attention models? *Advances in Neural Information Processing Systems*, 33:15383–15393, 2020. 129

[ZL16] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 51