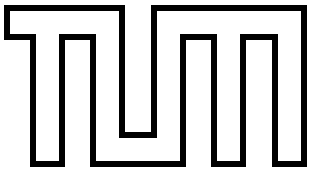


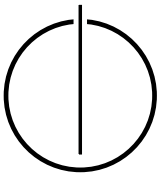
FAKULTÄT FÜR MATHEMATIK

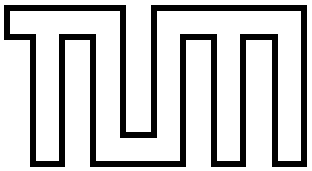
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Mathematics

Efficient and Scalable Linear Solver for Kernel Matrix Approximations Using Hierarchical Decomposition

Zhuoling Li





FAKULTÄT FÜR MATHEMATIK

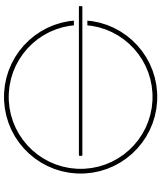
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Mathematics

Efficient and Scalable Linear Solver for Kernel Matrix Approximations Using Hierarchical Decomposition

Effizienter und skalierbarer linearer Solver für Kernel-Matrix-Approximationen unter Verwendung hierarchischer Zerlegung

Author:	Zhuoling Li
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Keerthi Gaddameedi, M.Sc., Severin Reiz, M.Sc.
Date:	01.12.2023



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 01.12.2023 Zhuoling Li

Acknowledgements

I am deeply grateful to Prof. Hans-Joachim Bungartz, my supervisor, for granting me the opportunity to explore the assigned topic. Alongside, I extend my heartfelt appreciation to my advisors, Keerthi Gaddameedi and Severin Reiz, whose unwavering support, invaluable guidance, and expert mentorship have been instrumental in shaping the course of this project. The contributions from my colleagues at LRZ, especially their steadfast hardware support, have significantly enriched this research. My heartfelt appreciation extends to my parents for their enduring love, encouragement, and sacrifices throughout my academic journey. Furthermore, I extend my sincere gratitude to my friends in various cities, whose unwavering support, understanding, and encouragement were paramount in this academic pursuit. Despite the geographical distance, their consistent encouragement, uplifting words, and understanding during challenging phases were immensely meaningful and contributed significantly to the success of this endeavor.

Abstract

This study presents an innovative approach that harnesses the power of the Geometry-Oblivious Fast Multipole Method (GOFMM) to compute and approximate kernel matrices derived from Convolutional-Neural-Network-equivalent Gaussian Processes (CNN-GPs). The primary objective is to devise an efficient and scalable linear solver specifically tailored to handle the intricacies associated with kernel matrix approximations within the domain of CNN-GPs. By leveraging hierarchical decomposition techniques, particularly the GOFMM approach, this research aims to significantly enhance both the computational efficiency and the accuracy in approximating kernel matrices in the context of CNN-GPs. The findings from extensive experiments underscore the accuracy of the proposed methodology, showcasing substantial improvements with complex CNN-GP architectures. Additionally, the scalability analysis demonstrates the method's robustness in handling various problem sizes, highlighting its potential versatility and applicability across a myriad of domains. The results of this study offer a promising avenue for enhancing the accuracy and computational efficiency of kernel matrix approximation, particularly in CNN-GP context, thereby facilitating advancements in various real-world applications demanding efficient processing of large-scale data.

Zusammenfassung

Diese Studie präsentiert einen innovativen Ansatz, der die Leistung der Geometry-Oblivious Fast Multipole Method (GOFMM) nutzt, um Kernmatrizen zu berechnen und anzunähern, die aus Convolutional-Neural-Network-equivalent Gaussian Processes (CNN-GPs) abgeleitet sind. Das Hauptziel besteht darin, einen effizienten und skalierbaren linearen Solver zu entwickeln, der speziell darauf abzielt, die Feinheiten der Kernmatrix-Approximationen im Bereich der CNN-GPs zu bewältigen. Durch die Nutzung hierarchischer Zerlegungstechniken, insbesondere des GOFMM-Ansatzes, zielt diese Forschung darauf ab, sowohl die Rechenleistung als auch die Genauigkeit bei der Approximation von Kernmatrizen im Kontext von CNN-GPs signifikant zu verbessern. Die Ergebnisse umfangreicher Experimente unterstreichen die Genauigkeit der vorgeschlagenen Methodik und zeigen erhebliche Verbesserungen bei komplexen CNN-GP-Architekturen auf. Darüber hinaus zeigt die Skalierbarkeitsanalyse die Robustheit der Methode bei der Bewältigung verschiedener Problemgrößen und unterstreicht deren potenzielle Vielseitigkeit und Anwendbarkeit in einer Vielzahl von Bereichen. Die Ergebnisse dieser Studie bieten einen vielversprechenden Ansatz zur Verbesserung der Genauigkeit und Rechenleistung der Kernmatrix-Approximation, insbesondere im Kontext von CNN-GPs, was Fortschritte in verschiedenen realen Anwendungen ermöglicht, die eine effiziente Verarbeitung großer Datenmengen erfordern.

Contents

Acknowledgements vii

Abstract ix

Zusammenfassung xi

1 Introduction 1

2 Theoretical Background 2
2.1 Preliminaries . . . 2
2.1.1 Convolution . . . 2
2.1.2 Non-linear Functions . . . 2
2.1.3 Kernel Matrix . . . 2
2.1.4 SPD Matrices . . . 3
2.2 Convolutional Neural Networks . . . 3
2.3 Gaussian Processes . . . 4
2.4 CNNs as Gaussian Processes . . . 5
2.4.1 A 2D Convolutional Network with Gaussian Prior . . . 6
2.4.2 CNN-equivalent GP Kernels . . . 7
2.5 Hierarchical Matrix Approximation . . . 8
2.5.1 FMM . . . 9
2.5.2 GOFMM . . . 9

3 Efficient and Scalable Linear Solver for Kernel Matrix Approximations Using Hierarchical Decomposition 13
3.1 Methodology . . . 13
3.1.1 Problem Statement . . . 13
3.1.2 Our Approach: Linear Solver for Kernel Matrix Approximation . . . 13
3.2 Implementation . . . 14
3.2.1 Main Architecture . . . 14
3.2.2 CNN-GP Implementation . . . 15
3.2.3 Parameter Tuning . . . 17

4 Experiments and Results 19
4.1 Experimental Setup . . . 19
4.1.1 Datasets . . . 19
4.1.2 Evaluation Metric . . . 19
4.1.3 CNN-GP Architectures . . . 20
4.2 Accuracy Evaluation . . . 21

4.3 Performance and Scaling 23
4.3.1 Hardware and Software Environment 23
4.3.2 Strong Scaling 25
4.3.3 Weak Scaling 26

5 Conclusion **29**

Bibliography **32**

1 Introduction

The field of machine learning and computational sciences has witnessed the growing prominence of kernel methods, leveraging kernel matrices as fundamental entities. These matrices play a pivotal role in various applications, facilitating non-linear transformations and enabling diverse computational techniques. However, resolving linear equations involving large-scale kernel matrices poses a formidable computational challenge, particularly when efficiency and scalability are imperative.

This thesis is dedicated to addressing the need for an efficient and scalable linear solver tailored specifically for kernel matrix approximations. Emphasizing the utilization of hierarchical decomposition from GOFMM (Geometry-oblivious Fast Multipole Method) [YLRB17] for this purpose, the research confronts the complexities inherent in handling kernel matrices derived from the CNN-GP framework [GAR19]. At its core, this work endeavors to devise a robust linear solver by harnessing the capabilities of GOFMM for approximating kernel matrices originating from CNN-GP. The study aims to augment computational efficiency and scalability, critical for addressing the intricacies and scale of modern data applications, while accommodating the properties of CNN-GP-derived matrices.

The significance of this work lies in its potential to propel computational methodologies within the domain of kernel-based methods. By introducing an optimized linear solver tailored explicitly for kernel matrix approximations derived from CNN-GP, this study anticipates profound advancements in efficiency and scalability across diverse applications, encompassing machine learning and computational sciences.

The thesis is structured into distinct chapters, each contributing uniquely to the exploration of the proposed linear solver. Chapter 2 extensively elucidates the theoretical foundations essential for understanding kernel matrix approximations and CNN-GPs. Chapter 3 delves into the methodologies and practical implementation aspects, detailing our main approach and the integration of GOFMM and CNN-GPs. In Chapter 4, the experimental setup, accuracy evaluation, and performance and scaling analysis of conducted experiments are presented, offering a comprehensive analysis and validation of the proposed linear solver. Finally, Chapter 5 encapsulates the thesis by summarizing key findings, contributions, and implications derived from the research, offering insights into potential avenues for future advancements.

2 Theoretical Background

2.1 Preliminaries

2.1.1 Convolution

The mathematical definition of the convolution function, often denoted as $(f * g)$, describes how one function is combined with another through a mathematical operation. The discrete form of convolution, often used in digital signal processing and image processing, is defined as follows:

$$(f * g)[n] = \sum_{k=-\infty}^{\infty} f[k] \cdot g[n-k] \tag{2.1}$$

where:

- $f[n]$ and $g[n]$ are two discrete sequences.
- $(f * g)[n]$ represents the convolution of f and g at position n.
- The summation $\sum$ computes the sum of the product of the elements of the two sequences as k varies.

2.1.2 Non-linear Functions

- **ReLU** (**Re**ctified **L**inear **U**nit) [Aga18]

  $$ReLU(x) = max(0, x) \tag{2.2}$$

  The ReLU function outputs x for $x > 0$ and 0 for $x \leq 0$

- **Sigmoid**

  $$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

  The sigmoid function maps inputs to a range between 0 and 1, which can be interpreted as probabilities.

2.1.3 Kernel Matrix

A kernel matrix, often denoted as K, is a symmetric matrix whose entries represent the pairwise similarities or inner products between data points in a high-dimensional feature space. Formally, for a dataset $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$, the kernel matrix K is a matrix of kernel functions between x_i and x_j:

$$K = \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{pmatrix} \tag{2.4}$$

and

$$k(x_i, x_j) := \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \tag{2.5}$$

where k is an inner product of $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_j)$. The basis (or feature) function $\phi(\mathbf{x})$ represents the mapping of the data point $\mathbf{x}$ into the high-dimensional feature space. It's important to note that this mapping allows for non-linear transformations, enabling the capture of complex relationships that may not be discernible in the original feature space.

2.1.4 SPD Matrices

A symmetric positive definite (SPD) matrix A is a square matrix with following properties:

- Symmetry: $A^T = A$
- Positive definiteness: $\forall x \in \mathbb{R}^n$ with $x \neq 0$: $x^T A x > 0$.

The positive definiteness of an SPD matrix ensures that it is invertible and has real and positive eigenvalues.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs), also known as ConvNets or Convolutional Networks, are a class of deep neural networks desinged primarily for processing structured grid data, such as images and video. CNNs are characterized by their architecture, which is specifically designed to capture local patterns and hierarchical features in data. The key components of a CNN are basically listed as follows:

Convolutional layers perform convolution operations on the input data with learnable filters or kernels that helps identify local patterns in the data. Given an input image or a feature map as a 2D matrix, a filter slides over the input. The convolution operation in CNNs can be mathematically expressed as follows:

$$(f * g)(i, j) = \sum_m \sum_n f(i + m, j + n) * g(m, n) \tag{2.6}$$

where:

- $f(i, j)$ represents the value at position (i, j) in the input.
- $g(m, n)$ represents the filter values.
- The double summation considers all positions of the filter relative to (i, j).

Pooling layers are used to reduce the spatial dimensions of the feature maps after convolution. Common pooling techniques like max pooling or average pooling aim for retaining the most important information while reducing computational complexity. For example, at each position in the input feature map F, max pooling selects the maximum value within a local region, and the pooled output feature map P can be calculated as follows:

$$P(i, j) = max_{m,n} F(i \cdot p + m, j \cdot p + n) \tag{2.7}$$

where p is the pool size and (m, n) iterates over the local region.

Fully Connected (FC) layers connect neurons from the previous layers to the ones in the subsequent layers. They help in capturing global patterns and making decisions based on local features extracted by the earlier layers. A non-linear activation function is applied element-wise afterwards to introduce non-linearity. Let X be the input to a FC layer with n neurons, and each neuron is associated with a weight W_i and a bias b_i, the output of the FC layer can be expressed as:

$$Y = \sigma(W \cdot X + b) \tag{2.8}$$

where:

- W is a weight matrix of size $n \times m$, where n is the number of neurons and m is the dimension of the input X.
- σ is a non-linear activation function (e.g., ReLU or sigmoid).

2.3 Gaussian Processes

The seminal work that laid the foundation for using Gaussian processes in modern machine learning is typically attributed to [WB98] in the late 1990s. As its core, a **Gaussian Process (GP)** is a collection of random variables, any finite number of which have a joint Gaussian distribution [Ras04]. In mathematical terms, a GP is defined as follows:

Let X be a set of input values (often representing time or space) and Y be a set of output values. A Gaussian Process is then defined as a probability distribution p over functions $f(X)$ such that for any finite subset of input values $X = \{x_1, x_2, \cdots, x_n\}$, the corresponding output values $Y = \{f(x_1), f(x_2), \cdots, f(x_n)\}$ follow a multivariate Gaussian distribution. This can be expressed as:

$$p(f(X)) = \mathcal{N}(\mu(X), K_{X,X'}) \tag{2.9}$$

where:

- $\mu(X) \in \mathbb{R}^n = \{\mu(x_1), \cdots, \mu(x_n)\}$ is the mean function representing the mean value of the GP at input values X.
- $K_{X,X'} \in \mathbb{R}^{n \times n}$ is the covariance matrix representing the covariance between the GP values at input values X and X': $(K_{X,X'})_{ij} := k(x_i, x_j)$, where k is the kernel function.

Figure 2.1 illustrates an example of Gaussian process regression:

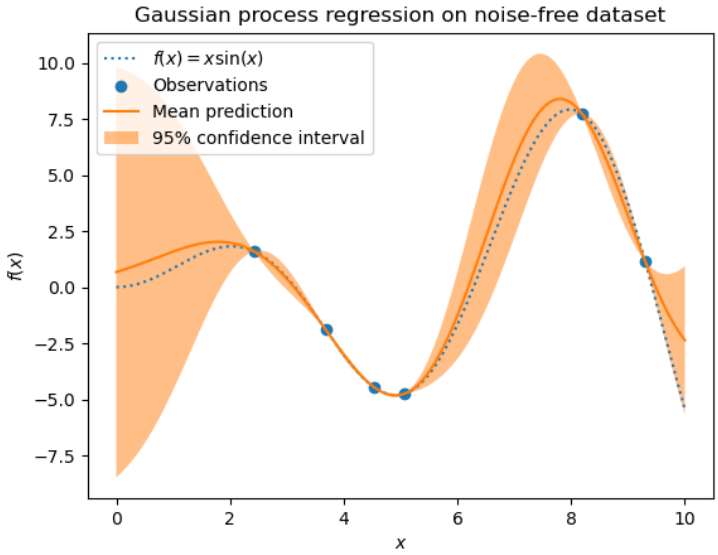


Figure 2.1: Example of Gaussian process regression: the dotted line represents the function $f(x)$. The orange line is the mean function $\mu(x)$ of the GP, and blue circles are the observed data points. [1]

GPs are non-parametric models, meaning that they can fit a wide range of functions without a fixed number of parameters. However, the choice of the mean and kernel functions, as well as their parameters, is crucial in defining the behavior of the GP and is typically determined from the data. Different kernels encode different assumptions about the functions, and selecting the appropriate kernel is often problem-dependent.

2.4 CNNs as Gaussian Processes

Since GPs are defined over an infinite-dimensional space of functions, researchers have been exploring their application for modeling complex and non-linear relationships between input and output values. For example, [WHSX16] combines their non-parametric flexibility with structural characteristics of deep neural networks. Specifically, they employ GPs to model the data, wherein the kernel function is constructed as a base kernel defined from a deep convolutional neural network (CNN) [LBD+89]. The deep kernel hyperparameters (including the base kernel parameters and the weights of the network) will then be learned by maximizing the log marginal likelihood $\mathcal{L}$ of the GPs. Nonetheless, an overfitting issue may arise as a consequence of the large number of kernel parameters.

To avoid bringing additional parameters from CNNs, [GAR19] shows that an arbitrarily deep CNN with an appropriate prior is equivalent to a GP, if each hidden layer has an infinite number of convolutional filters. In this case, the equivalent kernel contains only the parameters of the original CNN, addressing the issue raised in [WHSX16].

[1] Scikit-Learn. "Gaussian Process Regression (GPR) on Noisy Targets." Retrieved from https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_noisy_targets.html.

2.4.1 A 2D Convolutional Network with Gaussian Prior

Given an input image $\mathbf{X}$ with size $C \times H \times W$, where C is the number of the channels, H the height and W width, respectively.

Algorithm 1: A 2D Convolutional Network

Input: Image $\mathbf{X} \in \mathbb{R}^{C \times H \times W}(C : \#channels, H : height, W : width)$
Output: Last activations $\mathbf{A}^{(L+1)}(\mathbf{X})$

Function `2d_conv(X)`:
  // Iterate over all channels per layer
  for $i \leftarrow 1$ *to* $C^{(1)}$ **do**
    $a_i^{(1)}(\mathbf{X}) := b_i^{(1)}\mathbf{1} + \sum_{j=1}^{C} \mathbf{W}_{i,j}\mathbf{x}_j$ // Activations at layer 0
    for $l \leftarrow 1$ *to* L **do**
      $\mathbf{a}_i^{(l+1)}(\mathbf{X}) := b_i^{(l+1)}\mathbf{1} + \sum_{j=1}^{C^{(l)}} \mathbf{W}_{i,j}^{(l+1)}\phi(\mathbf{a}_j^{(l)}(\mathbf{X}))$ // Activations at other layers
  $\mathbf{A}^{(L+1)}(\mathbf{X}) := (a_1^{(L+1)}(x), \cdots, a_{C^{(L+1)}}^{(L+1)}(x))$ // Output activations

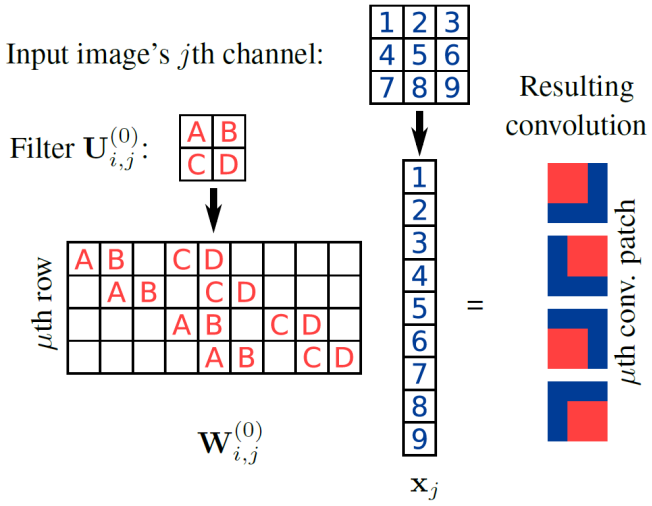


Figure 2.2: The convolutional filter $\mathbf{U}_{i,j}^{(0)} \in \mathbb{R}^{2\times2}$ performs on the input image $\mathbf{x}_j \in \mathbb{R}^{3\times3}$ on the jth channel. $\mathbf{W}_{i,j}^{(0)} \in \mathbb{R}^{(2\times2)\times(3\times3)}$ as a weight matrix stores the kernel parameters and transforms $\mathbf{U}_{i,j}^{(0)}$ into a matrix such that it can be multiplied with $\mathbf{x}_j$. The μth row represents the convolutional patch (red squares) where the filter is applied to (Here: A 2×2 filter applied on a 3×3 image leads to 4 patches). [GAR19]

A prior distribution is defined over functions by making the kernel parameters and biases independently Gaussian distributed, i.e., for each layer l, channels i, j and locations within the filter x, y:

$$U_{i,j,x,y}^{(l)} \sim \mathcal{N}(0, \sigma_w^2 / C^{(l)}) \tag{2.10}$$

$$b_i^{(l)} \sim \mathcal{N}(0, \sigma_b^2) \tag{2.11}$$

2.4.2 CNN-equivalent GP Kernels

[GAR19] prove that the output of the CNN defines a GP indexed by the inputs $\mathbf{X}$ and $\mathbf{X}'$, by applying the multivariate Central Limit Theorem (CLT), referring to the proof in [LBN+18] and [dGMRH+18]. The corresponding GP kernels are computed in an iterative way through the layers of the CNN. The kernel at layer 1 is initialized as follows:

$$K_\mu^{(1)}(\mathbf{X}, \mathbf{X}') = \sigma_b^2 + \frac{\sigma_w^2}{C^{(0)}} \sum_{i=1}^{C(0)} \sum_{v \in \mu th \ patch} X_{i,v} X_{i,v}^{'} \tag{2.12}$$

And at each layer $l+1$, the kernel is computed by the element-wise convariance of the activations from the last layer and the parameters of the Gaussian distribution:

$$K_\mu^{(l+1)}(\mathbf{X}, \mathbf{X}') = \sigma_b^2 + \sigma_w^2 \sum_{v \in \mu th \ patch} V_v^{(l)}(\mathbf{X}, \mathbf{X}') \tag{2.13}$$

where:

$$V_v^{(l)}(\mathbf{X}, \mathbf{X}') = \mathbb{E}[\phi(A_{j,v}^{(l)}(\mathbf{X}))\phi(A_{j,v}^{(l)}(\mathbf{X}'))] \tag{2.14}$$

is the element-wise covariance of the activations that is independent of the channel.

Note that the right-hand side of Eq. (2.14) can be computed in closed form with the choices of ϕ. For example, in the case of a ReLU nonlinearity (i.e., $\phi(x) = max(0, x)$), the covariance could be computed as:

$$V_v^{(l)}(X, X') = \frac{\sqrt{K_v^{(l)}(X, X') K_v^{(l)}(X', X'))}}{\pi} (sin\theta_v^{(l)} + (\pi - \theta_v^{(l)}) cos\theta_v^{(l)}) \tag{2.15}$$

where $\theta_v^{(l)} = cos^{-1}(K_v^{(l)}(X, X') / \sqrt{K_v^{(l)}(X, X) K_v^{(l)}(X', X')}$.

We go through all L layers of the CNN, compute intermediate kernel values and covariance at each layer and obtain in the end the output kernel $K^{(L+1)}$. The flowchart in Figure 2.3 illustrates the whole procedure of the computation, which is documented step by step in Algorithm 2.

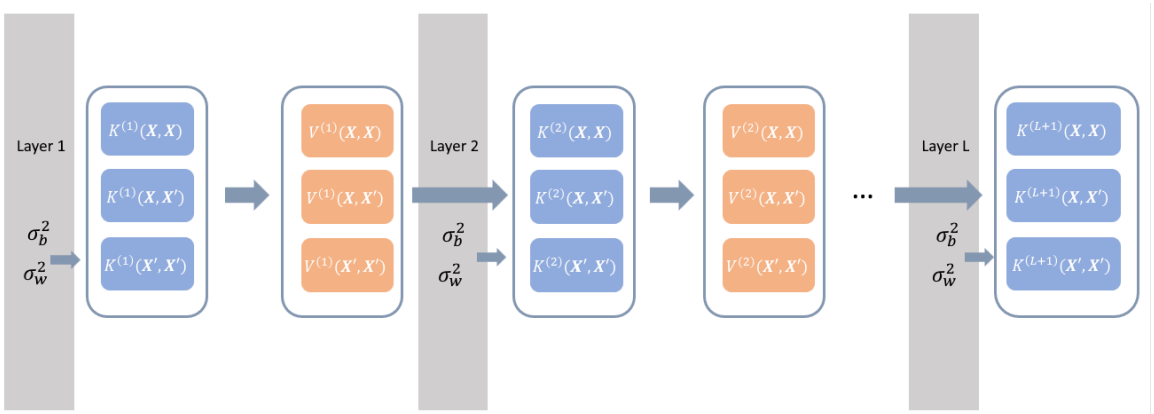


Figure 2.3: Flowchart of kernel computation in ConvNet

Algorithm 2: Compute ConvNet Kernel [GAR19]

Input: Image $\mathbf{X}, \mathbf{X}' \in \mathbb{R}^{C\times H\times W}$($C$: *#channels*, H : *height*, W : *width*)
Output: Kernel matrix $K^{(L+1)}(\mathbf{X}, \mathbf{X}')$

Function `conv_kernel(`$\mathbf{X}, \mathbf{X}'$`):`
  // Iterate over all locations
  for $\mu \in \{1, 2, ..., H^{(1)}W^{(1)}\}$ **do**
    Compute the variance of the first layer $K_\mu^{(1)}(\mathbf{X}, \mathbf{X})$, $K_\mu^{(1)}(\mathbf{X}, \mathbf{X}')$ and $K_\mu^{(1)}(\mathbf{X}', \mathbf{X}')$ using Eq. (2.12)
  for $l \leftarrow 1$ *to* L **do**
    for $\mu \in \{1, 2, ..., H^{(l)}W^{(l)}\}$ **do**
      Compute the covariance for each data pair $V_v^{(l)}(\mathbf{X}, \mathbf{X}')$ using Eq. (2.15)
    for $\mu \in \{1, 2, ..., H^{(l)}W^{(l)}\}$ **do**
      Compute the variance of the last layer $K_\mu^{(l+1)}(\mathbf{X}, \mathbf{X})$, $K_\mu^{(l+1)}(\mathbf{X}, \mathbf{X}')$, $K_\mu^{(l+1)}(\mathbf{X}', \mathbf{X}')$ using Eq. (2.13)
Output $K^{(L+1)}(\mathbf{X}, \mathbf{X}')$.

2.5 Hierarchical Matrix Approximation

Hierarchical matrix approximation is a technique used in numerical linear algebra to represent large matrices with a hierarchical low-rank structure.

Definition 2.5.1 *A matrix $\tilde{K} \in \mathbb{R}^{N\times N}$ is said to have a* ***hierarchical low-rank structure*** *or be an* ***$\mathcal{H}$-matrix*** *[Hac15], if:*

$$\tilde{K} = D + S + UV \tag{2.16}$$

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

2.5.1 FMM

The Fast Multipole Method (FMM) is a numerical technique designed for computing pairwise interactions in a system of N particles. Obviously the direct computation requires $\mathcal{O}(N^2)$ operations, while the FMM achieves only $\mathcal{O}(N)$ operations.

Classical FMM and its variations leverage quad-trees or oct-trees for hierarchical subdivision of the computation domain, often referred to as "tree code" algorithms. The tree structure enables adaptive refinement for non-uniform particle distributions, making these schemes well-suited for multi-core and parallel computing platforms.

2.5.2 GOFMM

GOFMM (**G**eometry-**O**blivious **FMM**) [YLRB17], is an algorithm performing a low-rank approximation of dense symmetric positive definite (SPD) matrices. Given an arbitrary SPD matrix K, GOFMM constructs a hierarchically low-rank matrix $\tilde{K}$ that approximates K, such that:

$$u = Kw \approx \tilde{K}w, \quad for\ w \in \mathbb{R}^N \tag{2.17}$$

where $K_{ij} = K(x_i, x_j)$. The main algorithm of GOFMM adopts a tree-based structure. For every tree node α, a node β is said to be *far* to α if $K_{\beta\alpha}$ is low-rank and *near* otherwise. Here are several key notations outlined as follows:

- $N(\alpha)$: neighbour list of node α
- $Near(\alpha)$: near interaction list of node α
- $Far(\alpha)$: far interaction list of node α
- $MortonID(i)$: a bit array that codes the path from the root to the leaf node

GOFMM usually consists of two stages: ***compression*** and ***evaluation***.

Compression

The compression part is composed of following processes: creating node neighbour lists and near lists of a node α, finding far list of α and performing low-rank approximation.

Given a treenode α that contains a set of matrix indices, its two children l and r will divide the indices evenly as shown in Figure 2.4.

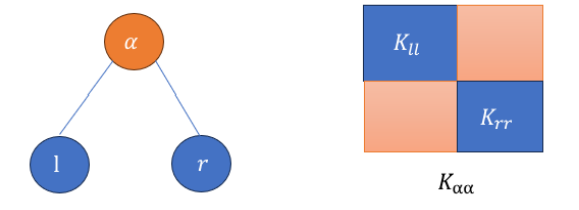


Figure 2.4: A treenode α contains a set of matrix indices and two children nodes l and r divide them evenly ($\alpha = l \cup r$).

Combining with Eq. (2.16), we can say that $\tilde{K}$ can be compressed recursively such that:

$$\tilde{K}_{\alpha\alpha} = \begin{bmatrix} \tilde{K}_{ll} & 0 \\ 0 & \tilde{K}_{rr} \end{bmatrix} + \begin{bmatrix} 0 & S_{lr} \\ S_{rl} & 0 \end{bmatrix} + \begin{bmatrix} 0 & UV_{lr} \\ UV_{rl} & 0 \end{bmatrix}$$

where l and r are the children nodes of α.

Algorithm 3 demonstrates the complete compression process. The functions comprising its components are detailed below:

Algorithm 3: COMPRESS(K) [YLRB17]

```
for each randomized tree do
    SPLI(α)                                   // split α into l and r
    ANN(α)                                    // update N_α with KNN(K_α)
SPLI(α)
Near(β) = LEAFNEAR(β)                         // Get Near(β) using Algo. (4)
Far(β) = FINDFAR(β, root)                     // Find Far(β) using Algo. (5)
MERGEFAR(α)          // Merge Far(l), Far(r) to Far(α) using Algo. (6)
// Low-rank approximation
α̃, P_α̃α or P_α̃[l̃r̃] = SKELETON(α)             // Skeletonization using Algo. (7)
∀α ∈ Near(β) : K_βα = K(β, α)
∀α ∈ Far(β) : K_β̃α̃ = K(β̃, α̃)
```

LEAFNEAR() constructs the near interaction list $Near(\beta)$ using the neighbour list $N(\beta)$.

Algorithm 4: LEAFNEAR(β)[YLRB17]

```
return Near(β) = {MortonID(i) : ∀i ∈ N(β)}
```

FINDFAR() finds the far interaction list $Far(\beta)$.

Algorithm 5: FINDFAR($\beta = leaf$, α)[YLRB17]

```
if α ∩ Near(β) ≠ ∅ then
    FINDFAR(β, l)
    FINDFAR(β, r)
else
    Far(β) = Far(β) ∪ α
```

MERGEFAR() involves combning common nodes found in two children lists $Far(l)$ and $Far(r)$. These shared nodes are extracted from the children lists and included in their parent list $Far(\alpha)$.

Algorithm 6: MergeFar(α) [YLRB17]

```
MergeFar(l)
MergeFar(r)
Far(α) = Far(l) ∩ Far(r)
Far(l) = Far(l)\Far(α)
Far(r) = Far(r)\Far(α)
```

The Skeleton() function basically consists of two parts: skeletonization that selects $\tilde{\alpha}$, and interpolation that computes $P_{\tilde{\alpha}[\tilde{l}\tilde{r}]}$.

Algorithm 7: Skeleton(α) [YLRB17]

```
if α is leaf then
    return [α̃, P_{α̃α} = ID(α)]
[l̃,] = Skeleton(l)
[r̃,] = Skeleton(r)
return [α̃, P_{α̃[l̃r̃]}] = ID([l̃r̃])
```

At the end of the compression phase, we can cache all $K_{\beta\alpha}$ in $Near(\beta)$, and all $K_{\tilde{\beta}\tilde{\alpha}}$ in $Far(\beta)$, which can significantly decrease the time required for evaluating and collecting submatrices.

Evaluation

GOFMM computes the value of u in Eq. (2.17). The goal is to approximate each *matvec* (matrix-vector multiplication) u_β with all α in $Near(\beta)$.

Algorithm 8: Evaluate(u, ω) [YLRB17]

```
if α is leaf then
    ω̃_α = P_{α̃α} ω_α                          // Compute w̃
else
    ω̃_α = P_{α̃[l̃r̃]} [ω̃_l; ω̃_r]
ũ_β = Σ_{α∈Far(β)} K_{β̃α̃} ω̃_α                 // Apply skeleton basis K_{β̃α̃}
if α is leaf then
    u_β = P^T_{β̃β} ũ_β
else
    [ũ_l; ũ_r]_+ = P_{β̃[l̃r̃]} [ũ_β]            // Accumulate ũ
u_β = Σ_{α∈Near(β)} K_{βα} ω_α                 // Accumulate matvec to u
```

The initial step involves computing the skeleton weights, denoted as $\tilde{\omega}$, for every leaf and inner node. Subsequently, the skeleton basis $K_{\tilde{\beta}\tilde{\alpha}}$ is utilized to aggregate the skeleton potentials $\tilde{u}$ alongside their respective skeleton weights. Finally, for a leaf node, the computation of the *matvec* u_β is directly achieved by accumulating its children. In contrast,

for a non-leaf node, each *matvec* u_β involving all α in $Near(\beta)$ is computed and subsequently accumulated into u_β.

3 Efficient and Scalable Linear Solver for Kernel Matrix Approximations Using Hierarchical Decomposition

3.1 Methodology

3.1.1 Problem Statement

Linear solving involves finding solutions to linear equations, and in the context of kernel matrices, it primarily revolves around solving linear systems in the form of:

$$K\mathbf{x} = \mathbf{b} \tag{3.1}$$

where:

- $K \in R^{N \times N}$ is the kernel matrix where each entry $K_{ij} := k(x_i, x_j)$ is the kernel function between i-th and j-th data points (N is the number of the data points).
- $\mathbf{x} \in R^N$ represents the solution vector.
- $\mathbf{b} \in R^N$ represents the target values or observations.

A kernel matrix K can grow substantially in size as N increases. This poses computational and memory challenges when storing and manipulating large kernel matrices. Therefore, developing efficient methods of linear solving for kernel matrices becomes crucial, i.e., computing the solution vector $\mathbf{x}$ with numerical stability.

We are aiming for constructing a kernel matrix approximation $\tilde{K}$ of the raw K, such that the error of the solution of the linear solving is sufficiently small, i.e.,

$$\|x - \tilde{x}\| \to 0 \tag{3.2}$$

with

$$x = K^{-1}b, \quad \tilde{x} = \tilde{K}^{-1}b \tag{3.3}$$

3.1.2 Our Approach: Linear Solver for Kernel Matrix Approximation

The algorithm aims to derive a solution for solving linear equations concerning the kernel matrix approximation $\tilde{K}$. GOFMM serves as a solid foundation for approximating the kernel matrix. While the typical approach involves initializing the kernel matrix of a diffusion map with a Gaussian kernel, we now explore the potential to leverage GOFMM for the kernel matrix generated by CNN-GP. The algorithm's illustration is outlined below:

Algorithm 9: Linear Solver for Kernel Matrix Approximation

Input: $X \in \mathbb{R}^{N \times C \times H \times W}$, $b \in \mathbb{R}^N$

Output: Linear solution for kernel matrix approximation $\tilde{x} \in \mathbb{R}^N$

1 $K_{xx} = \text{CONV\_KERNEL}(X, X)$ // Generate kernel matrix with Algo. (2)

2 $K_{approx} = \text{GOFMM}(K_{xx})$ // Kernel matrix approximation with GOFMM

3 **return** $\tilde{x} = K_{approx}^{-1} b$ // Approximated linear solution

With an input minibatch X sized (N,C,H,W), we construct a CNN designed to mirror an equivalent GP, followed by computing the kernel matrix associated with this GP (see Algorithm 2). GOFMM is then employed to approximate the kernel matrix through hierarchical decomposition. The resulting $\tilde{x}$ represents the linear solution derived from the kernel matrix approximation K_{approx} and target vector b. Note that the kernel approximation K_{approx} in this context is assumed to possess full rank, enabling the inversion process. In the end, we expect to attain $\tilde{x}$ with an error small enough in comparison to the raw linear solution $x = K^{-1}b$.

3.2 Implementation

This section provides an in-depth exploration of the implementation phase, offering insights into how theoretical concepts were translated into a functional system. It covers the main architecture of our approach, CNN-GP implementation, and crucial parameters utilized within our approach.

3.2.1 Main Architecture

The implementation of our approach is briefly described in the following pseudocode:

```
from full_matrix import FullMatrix
from cnn_gp.kernels import Sequential, Conv2d, ReLU
import scipy
import torch

cnn_gp = Sequential(
    Conv2d(),
    ReLU(),
    ...
) # initialize a CNN-GP model with specified layers

Kxx = cnn_gp(X,X) # generate kernel matrix of input batch X

kernel_OP = FullMatrix(Kxx, problem_size, ...) # kernel matrix
    approximation from GOFMM
x_gofmm = scipy.linalg.solve(kernel_OP,b) # approximated solution of
    the linear system
```

Listing 3.1: Pseudocode of our main approach

In the first stage, we initialize a CNN-GP model using a `Sequential` architecture that combines multiple convolutional layers (see line 6). After that, we feed the input minibatch X into the CNN-GP model to generate a kernel matrix of X. Note that in order to simplify the problem, we only consider the kernels between the data inside X itself in our implementation, meaning that taking two input of `cnn_gp` as the same. Once we get the kernel matrix, we use the `FullMatrix` class used for performing decompositions with GOFMM and obtain the approximation of the kernel matrix. The final outcome will be the solution of the linear system with the approximated kernel matrix.

3.2.2 CNN-GP Implementation

The CNN-GP model is structured as a `Sequential` class, organizing multiple convolutional layers in sequence. Within our implementation, we construct various instances of the CNN-GP by exploring different parameter combinations. Our primary focus is on the `Conv2d` and `ReLU` layers, both of which are subclasses of `NNGPKernel`. The following UML diagram illustrates the overall architecture of the CNN-GP implementation:

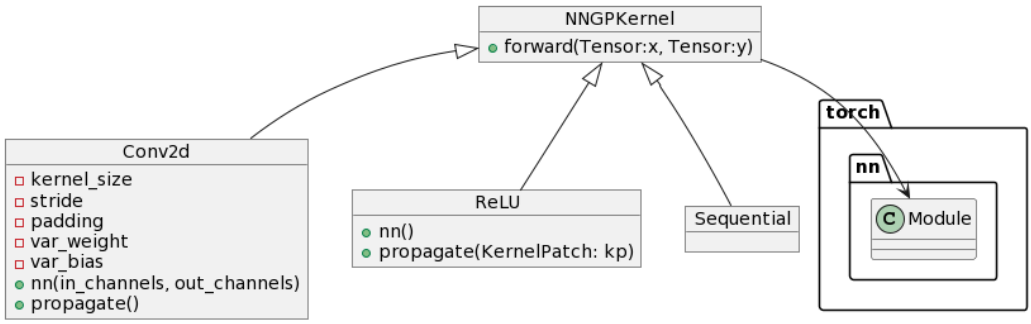


Figure 3.1: UML diagram of CNN-GP implementation.

- `NNGPKernel`: A `torch.nn.Module()` that constructs a covariance kernel of input minibatches x and (or) y for the GP.
- `Conv2d`: The parameters `kernel_size`, `stride`, `padding`, `var_weight` and `var_bias` are used to specify the kernels of convolutional layers. Note that a dense layer could also be implemented as a `Conv2d` with setting the `padding` to 0. `nn()` initializes neural networks that have the architecture corresponding to the kernel. `propagate()` computes the GP kernel through the convolutional layer.
- `ReLU`: The ReLU nonlinearity is used to numerically stabilise the covariance by clampling values.
- `Sequential`: The class combines multiple convolutional layers and sequences the `propagate()` and `nn()` accordingly.

In our implementation, a CNN-GP layer consists of a `Conv2d` and a `ReLU`. A `Conv2d` is initialized as follows:

```
class Conv2d(NNGPKernel):
    def __init__(self, kernel_size, stride=1, padding="same",
        dilation=1, var_weight=1., var_bias=0.):
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride
        self.dilation = dilation
        self.var_weight = var_weight
        self.var_bias = var_bias
        if padding == "same":
            self.padding = dilation*(kernel_size//2)
        else:
            self.padding = padding
```

Listing 3.2: Initialization of Conv2d() [GAR19]

And the entire model includes multiple CNN-GP layers where the last layer is a dense layer (`padding=0`), the below listing illustrates an example of a CNN-GP instance with a 3-layer convolutional network:

```
cnn_gp = Sequential(
    Conv2d(kernel_size=3),
    ReLU(),
    Conv2d(kernel_size=3, stride=2),
    ReLU(),
    Conv2d(kernel_size=7, padding=0)
)
```

Listing 3.3: Example of a 3-layer CNN-GP instance

Below is a comprehensive table detailing the parameters crucial for defining the entire network architecture.

	`kernel_size`	`stride`	`padding`	`dilation`
layer_1	3	1	1	1
layer_2	3	2	1	1
layer_3	7	1	0	1

Table 3.1: Example of a 3-layer CNN-GP instance

- `kernel_size`: The `kernel_size` parameter refers to the dimensions of the convolutional kernel.
- `stride`: The `stride` parameter determines the step size at which the convolutional kernel moves across the input data. For example, With a stride of 2, the kernel moves two pixels at a time along both height and width during the convolution operation.
- `padding`: The `padding` parameter involves adding extra pixels around the input data, helping preserve spatial dimensions after convolution. A padding of 1 adds one pixel of zero padding around the input, maintaining the output size.

- `dilation`: The `dilation` parameter controls the spacing between the kernel elements, influencing the receptive field and the stride of the convolution operation. A dilation value of 2 means there are gaps of one pixel between the elements of the kernel during convolution.

3.2.3 Parameter Tuning

Problem Size

The `problem_size` defines the size of the input minibatch and determines the dimension of the kernel matrix, scaling the approximation of GOFMM. The kernel matrix will be of size $N \times N$ if we set the `problem_size` to N. Consequently, the space complexity grows quadratically with a linear increase in the `problem_size`. This signifies that for each increment in the `problem_size`, the space required increases exponentially due to the quadratic relationship between size and space complexity. In order to explore how the performance of the approach influenced by the scaling, we set `problem_size` from 512 with a scale factor of 2, up to 16384.

```
X = torch.randn(problem_size, c, h, w) # input minibatch
Kxx = cnn_gp(X,X) # of size (problem_size, problem_size)
weights = np.ones((problem_size, num_rhs))
kernel_matrix_OP = FullMatrix(problem_size, weights, Kxx, ...)
```

Listing 3.4: Code snippet demonstrating the utilization of the `problem_size`

Input Resolution

The input resolution mainly refers to the height and width of the input minibatch X. Since the CNN-GP is designed to produce an output size of 1×1 (kernel matrix), it is essential that the input resolution aligns with this expectation to achieve the desired output. We take the example from Table 3.2:

	`kernel_size`	`stride`	`padding`	`dilation`	**input_size**	**output_size**
layer_1	3	1	1	1	14×14	14×14
layer_2	3	2	1	1	14×14	7×7
layer_3	7	1	0	1	7×7	1×1

Table 3.2: Input and output size of the example 3-layer CNN-GP

With an initial minibatch featuring an input resolution of 14×14, the initial layer's output maintains identical dimensions to the input owing to a padding value of 1 and a stride of 1. Subsequently, the second layer, employing a stride of 2, effectively halves the spatial dimension, resulting in an output size of 7×7. The third layer's application of a 7×7 kernel without padding on the 7×7 input ultimately yields a compressed 1×1 output due to the considerable kernel size in relation to the input dimensions.

Number of ConvNet Parameters

The number of parameters in the CNN refers to the total count of weights and biases across all layers within the network. These parameters play a pivotal role in defining the network's architecture and capacity. For a single Conv2d layer, the number of the parameters is the sum of the number of the weights and the number of the biases:

- num_parameters = num_weights + num_biases
- num_weights = `kernel_size` × `kernel_size` × `input_channels` × `output_channels`
- num_biases = `output_channels`

The total number of parameters of the entire ConvNet is obtained by summing the parameters across all layers. In the following table, we count the number of parameters for each layer of the example from Table 3.2:

	`kernel size`	`input channels`	`output channels`	num_weights	num_biases	num_params
layer_1	3	3	1	$3 \times 3 \times 3 \times 1 = 27$	1	28
layer_2	3	1	1	$3 \times 3 \times 1 \times 1 = 9$	1	10
layer_3	7	1	1	$7 \times 7 \times 1 \times 1 = 49$	1	50

Table 3.3: Number of parameters of the example 3-layer CNN-GP

The total number of the parameters is therefore $28 + 10 + 50 = 88$. This count gives an indication of the network's complexity, memory requirements, and learning capacity. Models with more parameters require more computation for each input sample, resulting in slower inference times.

Other Parameters

To ensure consistency across experiments for comparison, we maintain fixed values for other parameters, specifically utilized to initialize the `FullMatrix` for GOFMM approximation. These constants remain unchanged to facilitate comparative analyses.

```
max_leaf_node_size = 256
num_of_neighbors = 0
max_off_diagonal_ranks = 256
num_rhs = 1
user_tolerance = 1E-3
computation_budget = 0.00
distance_type = "kernel"
matrix_type = "dense"
kernel_type = "gaussian"
```

Listing 3.5: Other parameters

4 Experiments and Results

In this section, we delve into the empirical evaluation of our proposed methodology and implementation, presenting a comprehensive analysis of experiments conducted and the ensuing results.

4.1 Experimental Setup

4.1.1 Datasets

One dataset on which we evaluate our model is the MNIST dataset [LCB98], which consists of a collection of handwritten digits (0 through 9) that have been normalized and centered. Each image is a grayscale image with dimensions of 28×28 pixels, representing a single digit. In terms of classification, the dataset encompasses 10 distinct classes, each aligning with a digit ranging from 0 to 9.

- **Size**: The dataset contains a total of 60,000 training images and 10,000 test images.
- **Labeling**: Each image is associated with a ground truth label indicating the digit it represents.

In our experiment, we only sample the images from the training set with the size of `problem_size`. Each input data instance comprises a grayscale image of dimensions $1 \times 28 \times 28$. The associated label for each image ranges from 0 to 9, representing the digit depicted in the image.

4.1.2 Evaluation Metric

We evaluate the performance of the Algorithm 9 with the normalized error between the solutions of the raw kernel matrix and the approximated kernel matrix of GOFMM. The normalized error between N-dimensional solutions is defined as follows:

$$x_{error} := \frac{\|x - \tilde{x}\|_F}{\sqrt{N}} \tag{4.1}$$

where $x, \tilde{x} \in \mathbb{R}^N$ and N is the dimension of the kernel matrix. $\|\cdot\|_F$ denotes the Frobenius norm [Hig02] which is computed as the square root of the sum of the squares of all individual elements of the matrix sized $m \times n$:

$$\|\cdot\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} \tag{4.2}$$

To mitigate the impact introduced by the problem size N, we normalize the error by dividing the norm by the square root of the problem size. This normalization helps in

accounting for and scaling the error concerning the size of the problem, enabling a more standardized assessment.

4.1.3 CNN-GP Architectures

We performed experiments using various CNN-GP architectures to explore the impact of diverse input sizes. Initially, utilizing a randomly initialized input featuring 3 channels, we constructed the CNN-GP using convolutional kernels with smaller kernel sizes. As input resolutions decrease in size, the network tends to become less complex, characterized by a reduction in both parameters and layers.

Network	n_1	n_2
num_layers	3	3
layer 1	Conv2d(kernel_size=3)+ReLU	Conv2d(kernel_size=3)+ReLU
layer 2	Conv2d(kernel_size=3, stride=2)+ReLU	Conv2d(kernel_size=3, stride=2)+ReLU
layer 3	Conv2d(kernel_size=7, padding=0)	Conv2d(kernel_size=3, padding=0)
num_params	88	48
input_size	$3 \times 14 \times 14$	$3 \times 6 \times 6$

Network	n_3
num_layers	2
layer 1	Conv2d(kernel_size=2)+ReLU
layer 2	Conv2d(kernel_size=2, padding=0)
layer 3	-
num_params	18
input_size	$3 \times 2 \times 2$

Table 4.1: Different CNN-GP architectures on random 3-channel inputs

In the case of the MNIST dataset, the input image maintains a consistent size of (1, 28, 28). Consequently, we adapt the network architecture to accommodate these specific input dimensions. Additionally, we initialize the `var_weight` and `var_bias` using the prescribed values from [GAR19].

Network	n_5	n_6
num_layers	2	2
layer 1	Conv2d(kernel_size=3, var_weight * 7**2, var_bias) + ReLU	Conv2d(kernel_size=3, var_weight * 7**2, var_bias) + ReLU
layer 2	Conv2d(kernel_size=28, padding=0, var_weight * 7**2, var_bias)	Conv2d(kernel_size=14, padding=0, var_weight * 7**2, var_bias)
num_params	795	225
input_size	$1 \times 28 \times 28$	$1 \times 14 \times 14$

Network	n_7
num_layers	2
layer 1	Conv2d(kernel_size=3, var_weight * 7**2, var_bias) + ReLU
layer 2	Conv2d(kernel_size=7, padding=0, var_weight * 7**2, var_bias)
num_params	60
input_size	$1 \times 7 \times 7$

Table 4.2: Different CNN-GP architectures on MNIST dataset with fixed `var_bias`=7.86 and `var_weight`=2.79

The network n_5 serves as an illustration from the implementations of [GAR19]. When adjusting a similar architecture for smaller images, we tailor the last layer to fit the necessary output sizes. This adaptation results in a corresponding decrease in the number of parameters.

4.2 Accuracy Evaluation

We first evaluate our approach on randomly initialized input employing different CNN architectures. These networks exhibit distinct kernel sizes within their final layers, consequently leading to input sizes of (3, 14, 14) and (3, 6, 6) for n_1 and n_2, and (1, 28, 28), (1, 14, 14) and (1, 7, 7) for n_3, n_4 and n_5, respectively. Note that due to the memory limit on the cluster, certain complex networks with a larger number of weights could solely accommodate problem sizes up to 2048 or 4096. In the table, we signify these cases with a '-' to indicate unavailability due to memory constraints.

		CNN Architecture				
		n_1	n_2	n_5	n_6	n_7
num_weights		88	48	795	225	60
Problem size	512	6.62e-4	6.901e-1	1.73e-10	4.01e-9	1.02e-7
	1024	1.45e-2	3.16e0	3.49e-3	1.389e-1	9.3581e-1
	2048	2.542e1	1.119e2	-	4.929e-1	3.018e-1
	4096	-	1.15e2	-	-	4.946e0

Table 4.3: Normalized error x_{error} on random input

As observed in both types of networks ($\{n_1, n_2\}$ and $\{n_3, n_4, n_5\}$) from the table, it becomes apparent that as the number of weights decreases, signifying a simpler network architecture, the normalized error tends to increase. This trend indicates a proportionate relationship between the reduction in network complexity and the rise in normalized error across the tested configurations.

In order to strengthen this observation, our evaluation extends to the MNIST dataset, also exploring varied CNN-GP architectures. Notably, in the case of n_3, the network demonstrates simplicity, characterized by an extremely small number of parameters. As a consequence, across all instances, the kernel matrix fails to attain full rank. This occurrence is attributed to the specifics of the MNIST dataset, wherein the target values are constrained within a limited range. The complexity of CNN-GP becomes crucial in this context, potentially resulting in the generation of naive kernel matrices prone to linear dependency among rows, consequently rendering the matrix non-invertible.

	n_3					
problem size	512	1024	2048	4096	8192	16384
$rank(K)$	405	819	1611	3147	6139	11927

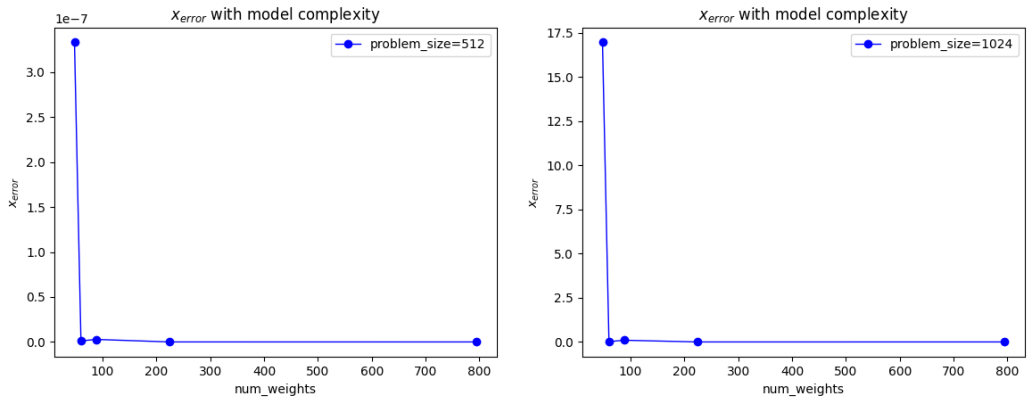
Table 4.4: Cases of n_3 (`num_params`=18): the kernel matrix with all problem sizes lack full rank.

This motivates us to explore more complex CNN-GP architectures on the MNIST dataset. Networks equipped with a minimum of 48 parameters are capable of handling matrices up to a size of 4096×4096. Except for the case of n_3, the outcomes derived from the MNIST dataset consistently exhibit a similar trend observed in the random input scenario, suggesting that more complex networks tend to yield smaller normalized errors:

		CNN Architecture				
		n_1	n_2	n_5	n_6	n_7
num_weights		88	48	795	225	60
problem size	512	2.83e-9	3.33e-7	2.10e-11	3.5e-11	1.19e-9
	1024	9.81e-2	1.696e1	1.66e-3	1.43e-3	1.90e-2
	2048	4.8408e4	-	-	1.50e2	2.57e2
	4096	-	-	-	-	5.122e3

Table 4.5: Normalized error x_{error} on MNIST dataset

To further examine the impact of CNN-GP model complexity on the normalized error x_{error}, we plot the error against different numbers of the weights of the network with different problem size:



(a) problem_size=512 (b) problem_size=1024

Figure 4.1: x_{error} on MNIST dataset with different model complexity: the x-axis represents the number of the weights in the CNN-GP model, while the y-axis denotes the normalized error x_{error}.

As depicted in both plots of Figure 4.1, initially, as model complexity increases, x_{error} tends to decrease, indicating improved performance with a more complex model. This reduction in error signifies the model's ability to capture intricate patterns in the data.

4.3 Performance and Scaling

In this section, we delve into an analysis of the performance and scaling characteristics of the implemented methodologies. The focus revolves around evaluating the computational efficiency and scalability of the proposed approaches across varying problem sizes and network complexities.

4.3.1 Hardware and Software Environment

Hardware The experiments are performed on the compute node *lxlogin2* within the CoolMUC-2 cluster. The CoolMUC-2 cluster comprises 812 nodes, each featuring 64GB of memory and operating on 28-way Intel Xeon E5-2690 v3 ("Haswell") processors. Furthermore, the cluster employs FDR14 Infiniband interconnect technology to facilitate communication between nodes. [1]

Software While docker container is not supported by the Linux cluster due to admin rights, we use Charliecloud [PR17] that enables users to run containers on HPC systems without requiring administrative privileges or root access. Charliecloud converts a docker image into a Charliecloud image, subsequently exporting it to the Linux cluster for utilization. The container creates an integrated environment for utilizing GOFMM methods, ensuring the installation of necessary run-time dependencies. Within this environment, it employs a SWIG (Simplified Wrapper Interface Generator) interface within this environment to generate Python versions of GOFMM's C++ methods. Apart from the docker image,

[1] "Linux Cluster Segments". Retrieved from https://doku.lrz.de/coolmuc-2-11484376.html.

installing the complete package encompassing all CNN-GP implementation necessitates the installation of key components, particularly the framework PyTorch [PGM+19].

The computation time corresponding to the experiments conducted in the previous section are documented in the subsequent tables. Once more, we distinguish between cases involving random inputs and those employing input derived from the MNIST dataset.

		CNN Architecture				
		n_1	n_2	n_5	n_6	n_7
num_weights		88	48	795	225	60
problem size	512	38.50	18.12	63.83	21.25	16.58
	1024	176.20	168.88	540.43	160.98	154.75
	2048	1707.10	1341.35	-	1180.93	1088.61
	4096	-	-	-	-	5829.88

Table 4.6: Computation time of our approach on random input (s)

The findings clearly demonstrate that as the number of weights in the CNN-GP network increases while maintaining a constant problem size, there is a direct correlation with extended computation times. This relationship is underscored by the increased volume of operations required during the forward pass of the CNN-GP. Conversely, when examining the same CNN-GP network, a substantial rise in computation time occurs upon doubling the problem size due to the increased size of the kernel matrix for approximation. Across various networks, there is a gradual decrease in computation time, contrasting with an observed exponential growth when augmenting the problem size. This exponential increase signifies an accelerated rise in computation time as the problem size expands. The results on MNIST dataset below further validate this trend:

		CNN Architecture				
		n_1	n_2	n_5	n_6	n_7
num_weights		88	48	795	225	60
problem size	512	12.93	7.66	27.02	14.76	14.23
	1024	211.56	141.81	208.73	158.65	146.02
	2048	1532.86	-	-	1525.01	1483.59
	4096	-	-	-	-	10314.47

Table 4.7: Computation time of our approach on MNIST dataset (s)

We visualize the relationship between runtime, model complexity, and problem size using the results from the aforementioned table in the following figure:

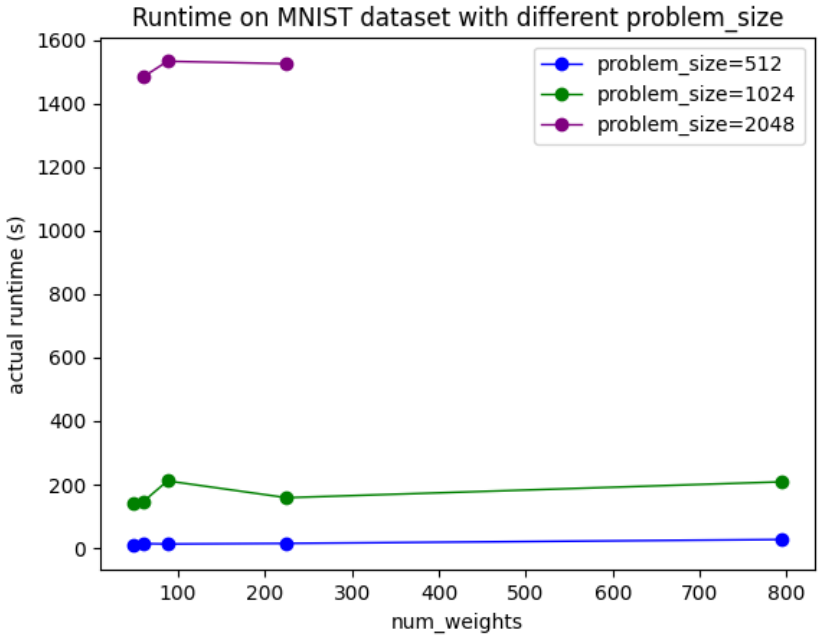


Figure 4.2: Runtime on MNIST dataset with different problem size

Figure 4.2 showcases a considerable disparity in runtime between problem sizes 1024 and 2048. By referencing the data from the above table, it is apparent that this gap is anticipated to widen further as the problem size escalates to 4096. Consequently, while both problem size and model complexity contribute to computation time, the problem size emerges as a relatively dominant factor. This is evident as it significantly influences the GOFMM process, which remains the most time-consuming aspect throughout the entirety of the procedure.

4.3.2 Strong Scaling

Strong scaling, in the realm of parallel computing, examines how the performance of the model envolves as the number of processors (`OMP_NUM_THREADS`) increases while keeping the problem size constant. Strong scaling assesses how effectively a model's performance improves as more processors are employed to solve a fixed-size problem. Here we fix the `problem_size` to 4096, and determine whether the runtime decreases as more processors are added to solve the computational task.

`problem_size`	4096					
`OMP_NUM_THREADS`	1	2	4	8	16	32
Runtime(s)	5.228e3	4.361e3	3.441e3	3.822e3	4.596e3	1.7296e4

Table 4.8: Runtime with strong scaling (`problem_size`=4096)

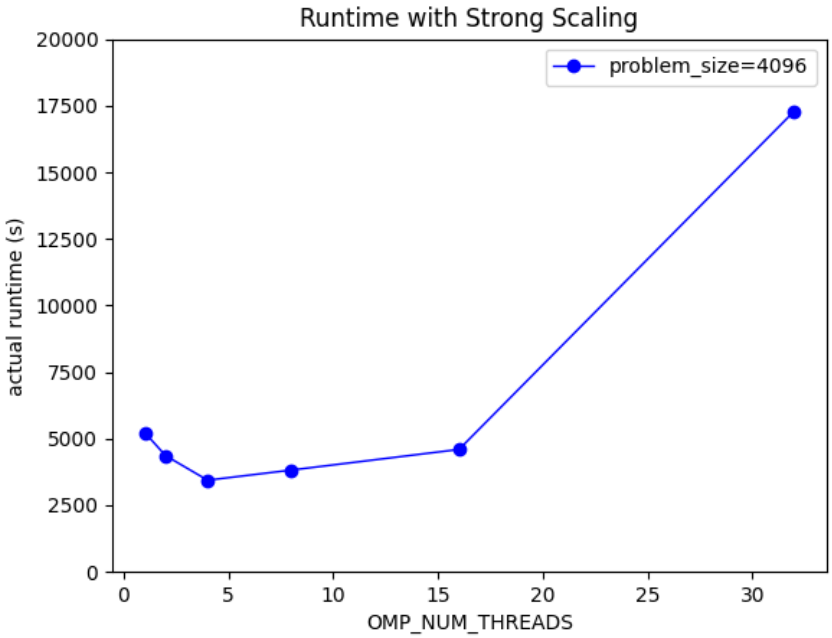


Figure 4.3: Runtime with strong scaling

The plot consistently depicts the runtime within the magnitude of 10^3 across 1 to 16 OMP threads. This stable runtime implies that scaling the computation by adding more threads does not significantly reduce the computation time per task, suggesting a saturation point in the system's ability to improve efficiency solely through thread addition. This phenomenon might arise from overhead or bottlenecks within the system as more threads are introduced.

4.3.3 Weak Scaling

Weak scaling is used to measure how a model's performance behaves when the problem size per thread (`LOAD_PER_THREAD`) remains constant, but the number of the threads (`OMP_NUM_THREADS`) increases. This evaluation aims to determine if the system's computational capacity grows proportionally alongside the allocated resources. In our context, we augment the `problem_size` by a factor of 2 and observe the runtime to completion. The aim of weak scaling is to maintain a balanced workload per thread as the computational resources expand. Note that in our scenario, the workload per processor does not stay constant; instead, it grows linearly with the number of threads due to the quadratic increase in matrix size:

$$\texttt{WORKLOAD\_PER\_THREAD} = \frac{\texttt{problem\_size} * (\texttt{scale\_factor})^2}{\texttt{OMP\_NUM\_THREADS}} \tag{4.3}$$

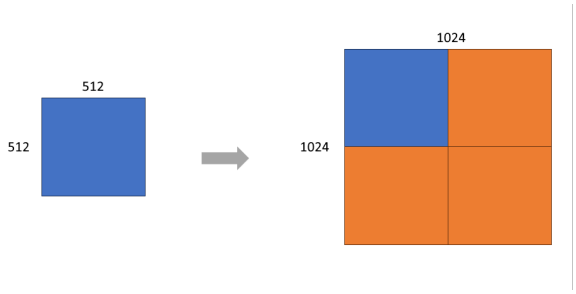


Figure 4.4: Quadratic increase of matrix size: when the problem size doubles, it leads to a workload increase of four times.

For instance, doubling the `problem_size` from 512 to 1024 with a scale factor of 2 results in a quadrupling of the total matrix. With an increase of `OMP_NUM_THREADS` by a factor of 2, the workload per thread is 1024, increasing 2 times as before:

`problem_size`	512	1024	2048	4096	8192	16384
`OMP_NUM_THREADS`	1	2	4	8	16	32
`WORKLOAD_PER_THREAD`	512	1024	2048	4096	8192	16384

Table 4.9: Workload per thread

Thus, by observing the `problem_size` and `workload_per_thread`, we expect a linear increase in the computation time as `problem_size` and `OMP_NUM_THREADS` increase simutaenously. The increase will correspond to the problem size.

`problem_size`	512	1024	2048	4096	8192	16384
`OMP_NUM_THREADS`	1	2	4	8	16	32
Runtime(s)	1.538e1	1.360e2	1.670e4	3.822e3	1.906e4	1.216e5

Table 4.10: Runtime with weak scaling

Table 4.10 and Figure 4.5 illustrate an noteworthy contrast: while the workload per thread showcases a linear growth, the runtime displays an exponential rise, particularly from `OMP_NUM_THREADS` = 16 onwards. This resemblance to the observed behavior also in strong scaling is quite remarkable.

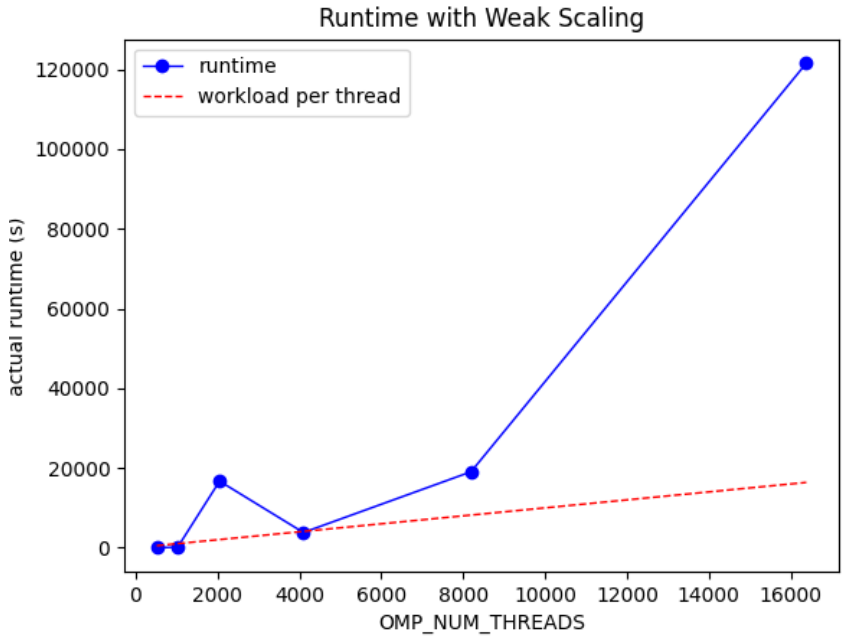


Figure 4.5: Runtime with weak scaling

The unexpected behavior observed in the plotted results of both strong and weak scaling could be attributed to various factors inherent to the experimental setup or the computational infrastructure employed. One conceivable factor could be the limitations of the hardware utilized for the experiments. It is only manageable to use 28 threads on the node *lxlogin2* of CoolMUC-2, such that the system reached its maximum capacity or encountered resource constraints near this threshold, hindering its ability to efficiently scale with the addition of more computational resources. Additionally, increased communication overhead among processors or threads could have introduced bottlenecks, causing inefficiencies as more resources were added. Furthermore, as the number of processors increased, synchronization and coordination between threads or nodes might have introduced overhead, affecting the overall performance.

5 Conclusion

In summary, this thesis has introduced a novel methodology employing the Geometry-Oblivious Fast Multipole Method (GOFMM) to compute and approximate kernel matrices from Convolutional Neural Network-equivalent Gaussian Processes (CNN-GPs). The primary objective was to craft an efficient and scalable linear solver tailored for managing the intricacies of kernel matrix approximations within CNN-GPs. Through the utilization of hierarchical decomposition techniques, notably GOFMM, this research significantly enhanced computational accuracy in approximating CNN-GP kernel matrices, especially with complex CNN-GP architectures. The methodology's application in extensive experimental scenarios highlighted nuanced behaviors in both strong and weak scaling. While initially anticipating linear or exponential relationships between problem size, workload per thread, and runtime, the observed deviations from these anticipated trends surfaced intriguing challenges. The unexpected behaviors were found to potentially stem from hardware limitations, where the system's capacity on certain nodes restricted scalability, encountering resource constraints or bottlenecks upon scaling beyond a certain threshold. Additionally, increased communication overhead among processors or threads further impacted scalability. These findings underscore the need for meticulous optimization efforts, considering potential reconfigurations of the computational environment, to achieve more favorable scaling behaviors. Addressing these challenges will pave the way for more efficient and scalable solutions, facilitating advancements across diverse applications reliant on large-scale data processing.

List of Figures

2.1 Example of Gaussian process regression: the dotted line represents the function $f(x)$. The orange line is the mean function $\mu(x)$ of the GP, and blue circles are the observed data points. [1] 5

2.2 The convolutional filter $\mathbf{U}_{i,j}^{(0)} \in \mathbb{R}^{2\times 2}$ performs on the input image $\mathbf{x}_j \in \mathbb{R}^{3\times 3}$ on the jth channel. $\mathbf{W}_{i,j}^{(0)} \in \mathbb{R}^{(2\times 2)\times(3\times 3)}$ as a weight matrix stores the kernel parameters and transforms $\mathbf{U}_{i,j}^{(0)}$ into a matrix such that it can be multiplied with $\mathbf{x}_j$. The μth row represents the convolutional patch (red squares) where the filter is applied to (Here: A 2×2 filter applied on a 3×3 image leads to 4 patches). [GAR19] 6

2.3 Flowchart of kernel computation in ConvNet 8

2.4 A treenode α contains a set of matrix indices and two children nodes l and r divide them evenly ($\alpha = l \cup r$). 9

3.1 UML diagram of CNN-GP implementation. 15

4.1 x_{error} on MNIST dataset with different model complexity: the x-axis represents the number of the weights in the CNN-GP model, while the y-axis denotes the normalized error x_{error}. 23

4.2 Runtime on MNIST dataset with different problem size 25

4.3 Runtime with strong scaling 26

4.4 Quadratic increase of matrix size: when the problem size doubles, it leads to a workload increase of four times. 27

4.5 Runtime with weak scaling 28

List of Tables

3.1 Example of a 3-layer CNN-GP instance 16
3.2 Input and output size of the example 3-layer CNN-GP 17
3.3 Number of parameters of the example 3-layer CNN-GP 18

4.1 Different CNN-GP architectures on random 3-channel inputs 20
4.2 Different CNN-GP architectures on MNIST dataset with fixed `var_bias`=7.86 and `var_weight`=2.79 21
4.3 Normalized error x_{error} on random input 21
4.4 Cases of n_3 (`num_params`=18): the kernel matrix with all problem sizes lack full rank. 22
4.5 Normalized error x_{error} on MNIST dataset 22
4.6 Computation time of our approach on random input (s) 24
4.7 Computation time of our approach on MNIST dataset (s) 24
4.8 Runtime with strong scaling (`problem_size`=4096) 25
4.9 Workload per thread 27
4.10 Runtime with weak scaling 27

Bibliography

[Aga18] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[dGMRH+18] Alexander G. de G. Matthews, Mark Rowland, Jiri Hron, Richard E. Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks, 2018.

[GAR19] Adrià Garriga-Alonso, Laurence Aitchison, and Carl Edward Rasmussen. Deep convolutional networks as shallow Gaussian processes. In *International Conference on Learning Representations*, 2019.

[Hac15] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49. 12 2015.

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition, 2002.

[LBD+89] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In *Neural Information Processing Systems*, 1989.

[LBN+18] Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S. Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes, 2018.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PR17] Reid Priedhorsky and Tim Randles. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[Ras04] Carl Edward Rasmussen. *Gaussian Processes in Machine Learning*, pages 63–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[WB98] C.K.I. Williams and D. Barber. Bayesian classification with gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1342–1351, 1998.

[WHSX16] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P. Xing. Deep kernel learning. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 370–378, Cadiz, Spain, 09–11 May 2016. PMLR.

[YLRB17] Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious fmm for compressing dense spd matrices, 2017.