TECHNISCHE UNIVERSITÄT
CHEMNITZ

Dissertation

On the Efficient Utilization of Dense Nonlocal Adjacency Information in Graph Neural Networks

Von der Fakultät für Mathematik der
Technischen Universität Chemnitz genehmigte
Dissertation zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Dominik Bünger

Gutachter:	Prof. Dr. Martin Stoll, Technische Universität Chemnitz Prof. Dr. Lars Ruthotto, Emory University Prof. Dr. Philip Knight, University of Strathclyde
Tag der Einreichung:	September 14, 2021
Tag der Verteidigung:	December 8, 2021
Veröffentlichungsjahr:	2021
Zitierlink:	`https://nbn-resolving.org/urn:nbn:de:bsz:ch1-qucosa2-769384`

Bünger, Dominik: On the Efficient Utilization of Dense Nonlocal Adjacency Information In Graph Neural Networks

Dissertation an der Fakultät für Mathematik der Technischen Universität Chemnitz

Dissertation, 2021, 166 Seiten

Das Werk – ausgenommen das Logo der TU Chemnitz und sämtliche Abbildungen – steht unter der Creative-Commons-Lizenz *Attribution - 4.0 International* (CC BY 4.0)
<https://creativecommons.org/licenses/by/4.0/>

Referat:

In den letzten Jahren hat das Teilgebiet des Maschinellen Lernens, das sich mit Graphdaten beschäftigt, durch die Entwicklung von spezialisierten Graph-Neuronalen Netzen (GNNs) mit mathematischer Begründung in der spektralen Graphtheorie große Sprünge noch vorn gemacht. Zusätzlich zu natürlichen Graphdaten können diese Methoden auch auf Datensätze ohne Graphen angewendet werden, indem man einen Graphen künstlich mithilfe eines definierten Adjazenzbegriffs zwischen den Samplen konstruiert. Nach dem neueste Stand der Technik wird jedes Sample mit einer geringen Anzahl an Nachbarn verknüpft, um gleichzeitig das dünnbesetzte Verhalten natürlicher Graphen nachzuahmen, die Stärken bestehender GNN-Methoden auszunutzen und quadratische Abhängigkeit von der Knotenanzahl zu verhinden, welche diesen Ansatz für große Datensätze unbrauchbar machen würde.

Die vorliegende Arbeit beleuchtet die alternative Konstruktion von vollbesetzten Graphen basierend auf Kernel-Funktionen. Dabei quantifizieren die Verknüpfungen eines jeden Samples explizit die Ähnlichkeit zu allen anderen Samplen. Deshalb enthält der Graph eine quadratische Anzahl an Kanten, die die lokalen und nicht-lokalen Nachbarschaftsinformationen beschreiben. Obwohl dieser Ansatz in anderen Kontexten wie der Lösung partieller Differentialgleichungen ausgiebig untersucht wurde, wird er im Maschinellen Lernen heutzutage meist wegen der dichtbesetzten Adjazenzmatrizen als unbrauchbar empfunden. Aus diesem Grund behandelt ein großer Teil dieser Arbeit numerische Techniken für schnelle Auswertungen, insbesondere Eigenwertberechnungen, in wichtigen Spezialfällen, bei denen die Samples durch niedrigdimensionale Vektoren (wie z.B. in dreidimensionalen Punktwolken) oder durch kategoriale Attribute beschrieben werden.

Weiterhin wird untersucht, wie diese dichtbesetzten Adjazenzinformationen in Lernsituationen auf Graphen benutzt werden können. Es wird eine eigene transduktive Lernmethode vorgeschlagen und präsentiert, eine Version eines Graph Convolutional Networks (GCN), das auf die spektralen und räumlichen Eigenschaften von dichtbesetzten Graphen abgestimmt ist. Schließlich wird die Anwendung von Kernel-basierten Adjazenzmatrizen in der Beschleunigung der erfolgreichen Architektur "PointNet++" umrissen.

Im Verlauf der Arbeit werden die Methoden in ausfürlichen numerischen Experimenten evaluiert. Zusätzlich zu der empirischen Genauigkeit der Neuronalen Netze liegt der Fokus auf wettbewerbsfähigen Laufzeiten, um die Berechnungs- und Energiekosten der Methoden zu reduzieren.

Schlagworte: Numerische Lineare Algebra, Fourier-Analysis, Maschinelles Lernen, Graph Neuronale Netze

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Dissertation

On the Efficient Utilization of Dense Nonlocal Adjacency Information in Graph Neural Networks

Dissertation approved by the
Department of Mathematics at Technische Universität Chemnitz
for the aquirement of the degree

doctor rerum naturalium (Dr. rer. nat.)

submitted by

Dominik Bünger

Reviewers:	Prof. Dr. Martin Stoll, Technische Universität Chemnitz Prof. Dr. Lars Ruthotto, Emory University Prof. Dr. Philip Knight, University of Strathclyde
Date of submission:	September 14, 2021
Date of defense:	December 8, 2021
Year of publication:	2021
Citation link:	`https://nbn-resolving.org/urn:nbn:de:bsz:ch1-qucosa2-769384`

Abstract

Over the past few years, graph learning – the subdomain of machine learning on graph data – has taken big leaps forward through the development of specialized Graph Neural Networks (GNNs) that have mathematical foundations in spectral graph theory. In addition to natural graph data, these methods can be applied to non-graph data sets by constructing a graph artificially using a predefined notion of adjacency between samples. The state of the art is to only connect each sample to a low number of neighbors in order to simultaneously mimic the *sparse* behavior of natural graphs, play into the strengths of existing GNN methods, and avoid quadratic scaling in the number of nodes that would make the approach infeasible for large problem sizes.

In this thesis, we shine light on the alternative construction of kernel-based fully-connected graphs. Here the connections of each sample explicitly quantify the similarities to *all* other samples. Hence the graph contains a quadratic number of edges which encode local and non-local neighborhood information. Though this approach is well studied in other settings including the solution of partial differential equations, it is typically dismissed in machine learning nowadays because of its dense adjacency matrices. We thus dedicate a large portion of this work to showcasing numerical techniques for fast evaluations, especially eigenvalue computations, in important special cases where samples are described by low-dimensional feature vectors (e.g., three-dimensional point clouds) or by a small set of categorial attributes.

We then continue to investigate how this dense adjacency information can be utilized in graph learning settings. In particular, we present our own proposed transductive learning method, a version of a Graph Convolutional Network (GCN) designed towards the spectral and spatial properties of dense graphs. We furthermore outline the application of kernel-based adjacency matrices in the speedup of the successful PointNet++ architecture.

Throughout this work, we evaluate our methods in extensive numerical experiments. In addition to the empirical accuracy of our neural network tasks, we focus on competitive runtimes in order to decrease the computational and energy cost of our methods.

Contents

List of Figures iv

List of Tables v

List of Algorithms v

1 Introduction 1

2 An Overview of Graphs in Learning Settings 5
2.1 Graph basics . . . 5
2.2 Types of graphs . . . 8
2.2.1 Assortative vs. disassortative edges . . . 8
2.2.2 Directed vs. undirected edges . . . 9
2.2.3 Sparse graphs with edges intrinsic to the data . . . 9
2.2.4 "Real-world" vs "constructed" graphs . . . 10
2.2.5 Local neighborhood graphs . . . 11
2.2.6 Kernel-based edge weights . . . 11
2.2.7 Fully connected graphs . . . 13
2.2.8 Clique expansions of hypergraphs . . . 14
2.3 The Graph Laplacian Operator . . . 16
2.3.1 Definitions and basic properties . . . 17
2.3.2 Laplacian eigenvalue problems and spectral properties . . . 18
2.3.3 Smoothness, noise, and clustering information . . . 21
2.3.4 Examples of graphs with varying spectral properties . . . 24
2.3.5 Motivation via differential operators . . . 27
2.3.6 Motivation via Minimum Cut problems . . . 29
2.4 Graph Convolution . . . 32
2.4.1 Background on Fourier transforms and convolution . . . 32
2.4.2 The graph Fourier transform . . . 34
2.4.3 Convolution and spectral filters on graphs . . . 35

3 Numerical Computation Schemes 39
3.1 Approximating kernel-based matrix-vector products . . . 39
3.1.1 The Fast Fourier Transform . . . 41
3.1.2 The Non-equispaced Fast Fourier Transform . . . 42

3.1.3 NFFT-based fast summation . . . 49
3.1.4 Comparison to the Fast Gauss Transform . . . 54
3.2 Eigenvalue computation with Krylov methods . . . 55
3.2.1 Basics . . . 56
3.2.2 The Lanczos process . . . 58
3.2.3 The Krylov-Schur method . . . 60
3.3 Eigenvalue computation for the normalized Laplacian . . . 62
3.3.1 Matrix shifts and Wielandt deflation . . . 62
3.3.2 Effect of inexact degree computation . . . 65
3.3.3 Specialized computations for hypergraphs from categorial data 72
3.4 Alternative eigenvalue algorithm: The Nyström method . . . 75
3.4.1 The traditional Nyström extension . . . 75
3.4.2 NFFT-based accelerated Nyström . . . 77
3.5 Numerical evaluations . . . 77
3.5.1 Accuracy and runtime of eigenvalue computations . . . 80
3.5.2 Application to spectral clustering of images by pixel color . . . 83
3.5.3 Semi-supervised learning by a phase field method . . . 86
3.5.4 Conclusion . . . 87

4 Graph Convolutional Networks for fully connected graphs 89
4.1 Background of Graph Convolutional Networks . . . 89
4.1.1 Neural Networks . . . 89
4.1.2 Supervised classification . . . 92
4.1.3 Graph Neural Networks for Transductive Learning . . . 94
4.1.4 Convolution and spectral filters as learned parameters . . . 97
4.1.5 General design principles for filter functions . . . 99
4.1.6 Brief history of Graph Convolutional Network architectures . . 103
4.1.7 The role of loops . . . 105
4.2 The Pseudoinverse Graph Convolutional Network architecture . . . 105
4.2.1 Challenges of dense graphs . . . 105
4.2.2 Pseudoinverse spectral filters . . . 107
4.2.3 Low-rank approach . . . 108
4.2.4 Pseudoinverse filters in Graph Convolutional Networks . . . 111
4.2.5 Two-layer architecture . . . 114
4.3 Techniques for special cases . . . 115
4.3.1 Hypergraphs from categorial data . . . 115
4.3.2 Hypergraph regularization of sparse graphs . . . 117
4.4 Experiments . . . 118
4.4.1 Implementational details . . . 118
4.4.2 Baseline methods . . . 120
4.4.3 Semi-supervised point classification in 3D point clouds . . . 121

4.4.4 Hypergraph data sets from categorical attributes 121
4.4.5 Limitations: Sparse graph data sets 123
4.4.6 Rank dependency 125
4.4.7 Analysis of learned weight entries 128
4.4.8 Conclusion 128

5 Dense adjacency matrices in PointNet variants **131**
5.1 Original architectures for point cloud classification 131
5.1.1 Problem setting 131
5.1.2 The PointNet scheme 133
5.1.3 PointNet++ with Set Abstraction Modules 134
5.2 Proposed modifications 138
5.2.1 Single-layer perceptrons as local components 138
5.2.2 Add pooling and adjacency matrices 139
5.2.3 Other pooling operations 140
5.2.4 Nonlocal pooling and fully connected adjacency matrices . . . 143
5.2.5 SoftPointNet method 144
5.3 Experimental results 145
5.3.1 Training process & implementation details 145
5.3.2 Experiments on the ModelNet data sets 146
5.3.3 Interpretation and conclusion 150

6 Summary & Theses **153**

Bibliography **157**

List of Figures

3.1 Representation of the Gaussian window function variants for $d = 1$ . . 46
3.2 Example of the shifted window function values for the Gaussian window function . . . 47
3.3 The regularized Gaussian kernel K_R with $\varepsilon_B > 0$ . . . 51
3.4 Empirical measurements of various error components for the graph Laplacian for different Gaussian parameter values . . . 72
3.5 Visualization of the spiral data set from Example 3.19 with $n = 2\,000$ points . . . 80
3.6 Comparison of accuracies and runtimes for Laplacian eigenvalue computation on the spiral data set with Gaussian adjacency matrix . . . 82
3.7 Results of color-based image segmentation via spectral clustering and k-means using the NFFT-based Krylov-Schur method and the Nyström method . . . 85
3.8 The first ten eigenvalues of $\mathcal{L}_{\text{sym}}$ for the fully-connected graph with Gaussian weights and scaling parameter $\sigma = 90$ for Figure 3.7 . . . 86
3.9 Comparison of average classification rates with the phase field method on relabeled spiral data sets . . . 88
4.1 Visualization of a perceptron with two input neurons and three output neurons . . . 91
4.2 Visualization of selected convolutional filter functions of our proposed form . . . 112
4.3 Misclassification rate development over different split sizes on the UCI categorical data sets . . . 125
4.4 Average classification results for the sparse citation graphs transformed into centroid hypergraphs with categorical regularization. . . . 126
4.5 Misclassification rate and runtime development over different ranks r. 127
5.1 Visualization of one exemplary point cloud from the “chair” category in the ModelNet40 data set . . . 132
5.2 Visualization of the point sequence in three consecutive SA modules for a 2D point cloud . . . 135

List of Tables

2.1 Spatial and spectral properties of the graphs from Examples 2.18, 2.19, and 2.20 26
4.1 Effect of regularization on the citation graphs from Example 2.18 . . . 119
4.2 Average classification results on the Oakland data sets 122
4.3 Average classification results on the UCI categorical data sets 124
4.4 Average classification results for the sparse citation graphs 126
4.5 Average absolute weight matrix entries for the three separate filter parts 129
5.1 Average classification results for the ModelNet10 data set 147
5.2 Average classification results for the ModelNet40 data set 149

List of Algorithms

3.1 The forward Non-equispaced Fast Fourier Transform (NFFT) 47
3.2 NFFT-based fast summation 52
3.3 Setup of approximate matrix-vector products with the adjacency matrix of a kernel-based fully-connected graph 55
3.4 Rayleigh-Ritz method based on a Krylov decomposition 58
3.5 The Lanczos process for eigenvalue approximation 59
3.6 Krylov-Schur method for eigenvalue approximation 63
3.7 Eigenvalue approximation for $\mathcal{L}_{\text{sym}}$ with Wielandt deflation 66
3.8 NFFT-based accelerated Nyström-Gaussian method for Laplacian eigenvalue approximation 78
5.1 PointNet++ Set Abstraction (SA) layer 136

1 Introduction

The field of machine learning allows its users to tackle a variety of problems that must be deemed straight up unsolvable with conventional programming. In various fields including computer vision, natural language processing, medicine, bioinformatics, and finance, there are systems whose underlying internal mechanics are either unknown or cannot be described mathematically.

As a basic example, the task of distinguishing cats and dogs in a given picture is one that humans can solve with ease. However, if you are challenged to explain exactly how you come to your conclusion only based on the color values in each pixel, you would probably fail – you simply *see* the object in a way that transcends the raw data. For the same reason, we cannot teach our method to a computer via explicit step-by-step instructions in the same way that we write conventional algorithms.

In such a setting, machine learning methods like neural networks can *learn* to exhibit the desired behavior from provided examples that describe the ground truth. This training process generally requires both pre-labeled data and computational power on a large scale. For that reason, the rise of neural networks has only started around the turn of the century despite having been known for decades prior [5, 41].

From the point of view of an applied mathematician, learning methods may sometimes feel a bit lacking because so much of their success is due to heuristics and empirical observations that are difficult to analyze mathematically. For example, although the training process is formalized as an optimization problem, most specialized numerical techniques (e.g., second order methods or line search) are rarely applicable due to the high parameter dimension. Furthermore, the behavior of any neural network is strongly dependent on the training data and to a smaller degree on the random parameter initialization as well as hyperparameters that are often picked by experience. Therefore performance guarantees and practical error bounds are generally not obtainable and the influence of design choices on the outcome can often not be quantified in a general, abstract manner. Considering that, the training process in practice is still comparably robust in the end, but it is still not fully understood which structural particularities enable this feat. As a result, new methods are generally judged on their performance on certain benchmark data sets, which can only provide hints at their potential and practicability.

Despite all this, mathematics has contributed a lot to the improvement of neural networks and continues to do so. Maybe even because of the lack of quantification, mathematical derivations play an important role in the development of new approaches. There is great desire for interpretability within the machine learning

community, and if a method is based on techniques with provable benefits in abstract settings, this helps build trust in its stability and practical viability even if the final transfer to the actual learning setting is still heuristic. In addition, resource efficiency of learning algorithms is a major area of research in times where the high energy consumption of large scale network training raises environmental concerns. Techniques that are known from fields like numerical linear algebra and model order reduction can help reduce computational costs in special cases.

One important entry point for mathematical contributions are data-specific specializations. In theory, the simplest form of neural network, the multi-layer perceptron (MLP), is already flexible enough to solve any reasonable task sufficiently well if provided with enough training data and computational capacity (cf. the universal approximation theorem [5]). In practice though, the best results are obtained by models that are designed specifically to exploit the specific structure of the task at hand. For example, the regular grid structure of images can be utilized in Convolutional Neural Networks (CNNs), which represent the backbone of recent advancements in computer vision [96]. While the same effect can in theory be achieved by fully connected MLP layers, convolutional layers offer better interpretability and empirical payoff for a fraction of the computational cost.

In this work, we will focus on *graphs*, a different kind of data structure that allows for mathematical motivation and efficient computations [29]. Graphs are also known as *networks*, but we will only use the former name here due to the nomenclature overlap with neural networks.[1] They are essentially collections of entities called *nodes* that are equipped with a notion of adjacency in the form of *edges*. The nodes can encode any real-life or technical object like biological or physical samples, people in a community, products in a warehouse, or even pixels or point coordinates. An edge is a connection between two such nodes that represents some form of relationship, which may either be intrinsic to the data or an artificial indicator of similarity. Graphs are sufficiently flexible to be used for complex data representation in various domains, but most importantly, they are still abstract enough to enable sophisticated qualitative and quantitative statements in the mathematical domain of graph theory.

In terms of machine learning, theoretical understanding of given graph structures can be utilized by specialized neural network models collected under the umbrella term Graph Neural Networks (GNNs, [65]). Here, conventional analytical and computational concepts like smoothness and graph convolution can be used to incorporate the graph structure into the flexible neural network framework.

In a large portion of applications, the graph is structured in such a way that each node is only connected to a small number of neighbors. This allows the imple-

[1] In turn, neural networks can also be viewed as *computational graphs*, where nodes represent computed quantities and directed edges represent building blocks of computations. To avoid confusion, we will always use *graphs* to refer to data and *networks* to refer to a computational model.

mentation to exploit matrix sparsity in order to maintain applicability even for large problem sizes. Since the important benchmark problems all show this behavior, most methods are skewed towards these sparse graphs and designed to benefit from their special theoretical properties. As a result, when edge sets are constructed based on node feature similarity, this is commonly also done in a sparse way.

However, there are certain applications that favor a different approach. We will here discuss two main examples: three-dimensional point clouds as created by, e.g., LiDAR scans [76], and data sets where each sample is described by a comparably low number of discrete features. In both cases, the conventional sparse representation is possible but requires additional model assumptions and discards non-local information. Instead, we can construct *dense, fully connected* graphs where each node is adjacent to all other nodes, e.g., via radial basis function kernels that also play important roles in kernel methods [88], partial differential equations [35], and other numerical fields. As a result, the number of edges grows quadratically with the data set size and there are a lot of edges with lower weights that encode non-local similarities over larger distances. Taking advantage of this information requires specialized methods, but can ultimately improve the prediction performance. Similar techniques are also known in computer vision [21, 94] and image processing [38].

The main purpose of this thesis is to highlight how dense and fully connected graphs can be utilized in machine learning settings. We showcase the potential benefit of density for exemplary classification tasks. For machine learning researchers, the main deterrent of this approach is the computational cost. Therefore we provide a complete framework of efficient approximations that yield the same computational complexity as sparse graphs by comprising methods from Fourier analysis and numerical linear algebra. A further important contribution is the code developed for the experiments, most notably two publicly available Python packages that are aimed to make dense graphs more accessible for future research. We hope that our results convince more researchers to use fully connected graphs in new settings in the future.

This thesis is structured as follows. Chapter 2 gives background information on the graph setting and summarizes crucial concepts of spatial and spectral graph theory that will be utilized throughout this thesis. Chapter 3 deals with numerical techniques that are useful for the efficient computation of certain quantities of interest for graphs, most notably the spectrum of the graph Laplacian operator. In Chapter 4, we derive, present, and evaluate our proposed Graph Neural Network method, the pseudoinverse graph convolutional network. Furthermore, we discuss a related setting in supervised learning in Chapter 5 and derive an approach to reduce the computational cost of the popular PointNet++ network. Finally, Chapter 6 contains a summary of our findings.

Chapters 3 and 4 partially contain content that has been published in [2] and [3], respectively, with much additional background and explanations. Chapter 5 is part of ongoing work that will be submitted for publication in the future. An additional result of our research is currently in preparation, but does not fit in thematically with

the remainder of this thesis [1].

2 An Overview of Graphs in Learning Settings

This chapter lays the groundwork for the whole thesis by providing fundamental terms and concepts of mathematical graph theory, which makes this thesis largely self-contained. Due to the importance of linear algebra for our work, we put an increased emphasis on matrix-related concepts, already starting with the first definitions. Hence some aspects may differ from other works in the field. For a more comprehensive introduction that includes other viewpoints, see, e.g., [29].

2.1 Graph basics

To start things off, we give abstract definitions of the central structures of our setting, starting with the graph itself.

2.1 Definition. *(General graph framework)*
(a) A graph is a tuple $G = (V, \mathcal{A})$ of a finite, non-empty, ordered *node set* or *vertex set* $V = \{v_1, \dots, v_n\}$ and an *adjacency matrix* $\mathcal{A} = (a_{ij})_{i,j=1}^{n} \in \mathbb{R}^{n \times n}$ with non-negative entries $a_{ij} \geq 0$. Throughout this work, we identify V as $\{1, \dots, n\}$ and the nodes by their indices.

(b) For i, j with $a_{ij} > 0$, we say that node j is *adjacent* to node i and a_{ij} indicates the *strength* of the adjacency (cf. Section 2.2.1).

(c) A graph is called *unweighted* if $a_{ij} \in \{0, 1\}$ for all $i, j \in V$. Otherwise it is called *weighted* and $a_{ij} > 0$ typically encodes the *strength* of the adjacency (cf. Section 2.2.1).

(d) A graph is called *undirected* if $a_{ij} = a_{ji}$ for all $i, j \in V$, i.e., the adjacency matrix is symmetric with $\mathcal{A} = \mathcal{A}^T$. Otherwise, it is called *directed*.

Unless stated otherwise, the letter n will always refer to the number of nodes in a graph. In this thesis, we will almost exclusively work with undirected graphs.

2.2 Definition. *(Edges)*
(a) In any graph $G = (V, \mathcal{A})$, a *directed edge* is a tuple $\vec{e} = (i, j) \in V \times V$ such that $a_{ij} > 0$. Here i and j are called the *start* and *end* nodes of edge $\vec{e}$. The set of all edges is called the *edge set* $\vec{E} \subset V \times V$.

(b) A *loop* is an edge connecting a node with itself, $\vec{e} = (i, i)$ with $a_{ii} > 0$. Graphs

without loops are called *loop-free*.

(c) In undirected graphs, an *undirected edge* is a subset of V with one or two elements, $e = \{i, j\}$, such that $a_{ij} = a_{ji} > 0$. For loops, we write $e = \{i, i\}$ instead of $\{i\}$ for notational consistency. The set of all undirected edges is called the *undirected edge set* E. For non-loops, the directed edges $(i, j), (j, i) \in \vec{E}$ are the two *directions* of $e = \{i, j\}$.

(d) The *edge weight map* w assigns to each directed edge $\vec{e} = (i, j) \in \vec{E}$ its associated adjacency matrix entry $w(\vec{e}) = a_{ij} > 0$, called the *edge weight* of $\vec{e}$. In undirected edges, w can also be evaluated for undirected edges, $w(\{i, j\}) = a_{ij} > 0$. In unweighted graphs, all edge weights are one, $w \cong 1$.

Any graph $G = (V, \mathcal{A})$ can be alternatively characterized through $G = (V, \vec{E}, w)$ with the directed edge set $\vec{E}$ and the edge weight map w. For unweighted graphs, w can be omitted. For undirected graphs, the undirected edge set E is sufficient.

In fact, this alternative characterization is more common in the literature, partly because the edges often have a direct connection to the real-world application and thus offer a natural description of the graph, cf. Section 2.2.4.

It is a common convenience in the literature to stretch the notation of indexing for vectors and matrices to allow the use of edges as indices as well as edge sets as exponents for product sets. This way we can avoid having to work with an explicit ordering all the time. Let us formally introduce this notation.

2.3 Remark. *(Indexing with edges)*
By assuming an arbitrary, but fixed ordering of the edges, $\vec{E} = \{\vec{e}_1, \dots, \vec{e}_{|\vec{E}|}\}$, we can exploit the usual vector notation and agree that $\boldsymbol{x} \in \mathbb{R}^{\vec{E}}$ is identical to $\mathbb{R}^{|\vec{E}|}$ but directly using the edges as indices,

$$x_{\vec{e}_j} \mathrel{\widehat{=}} x_j \quad \forall\, j = 1, \dots, |E|.$$

This way, the weight map $w : \vec{E} \to \mathbb{R}_{>0}$ of a graph can be equivalently understood as a vector

$$\boldsymbol{w} = \big(w_{\vec{e}}\big)_{\vec{e} \in \vec{E}} \in \mathbb{R}^{\vec{E}}_{>0} \quad \text{with} \quad w_{\vec{e}} \mathrel{\widehat{=}} w_{(i,j)} \mathrel{\widehat{=}} w(\vec{e}) \mathrel{\widehat{=}} w(i, j) \quad \text{for } \vec{e} = (i, j) \in \vec{E}.$$

In undirected graphs, we can analogously write $\boldsymbol{w} \in \mathbb{R}^{E}$. Throughout this work, we will typically use edge indexing where applicable.

At some occasions, we will need to sum over all nodes j adjacent to a central node i. For that, we introduce the notation of *neighborhoods*.

2.4 Definition. *(Neighborhoods)*
For each node $i \in V$ in a graph, we define the following two sets of adjacent nodes:

- the set of neighbors from *outgoing* edges $\mathcal{N}_i^{\text{out}} = \{j : a_{ij} > 0\}$,
- the set of neighbors from *incoming* edges $\mathcal{N}_i^{\text{in}} = \{j : a_{ji} > 0\}$.

In undirected graphs, both sets are identical and we write $\mathcal{N}_i = \mathcal{N}_i^{\text{out}} = \mathcal{N}_i^{\text{in}}$.

Next up, node degrees are a simple *centrality measure* that show how many connections a given node has, possibly taking into account the edge weights.

2.5 Definition. *(Node degrees)*
For each node $i \in V$ of a directed graph, we define

- the unweighted outgoing node degree $|\mathcal{N}_i^{\text{out}}|$,
- the unweighted incoming node degree $|\mathcal{N}_i^{\text{in}}|$,
- the weighted outgoing node degree $d_i^{\text{out}} = \sum_{j \in \mathcal{N}_i^{\text{out}}} w(i,j) = \sum_{j=1}^{n} a_{ij}$,
- the weighted incoming node degree $d_i^{\text{in}} = \sum_{j \in \mathcal{N}_i^{\text{in}}} w(j,i) = \sum_{j=1}^{n} a_{ij}$,
- the diagonal outgoing node degree matrix $\mathcal{D}^{\text{out}} = \text{diag}(d_1^{\text{out}}, \dots, d_n^{\text{out}})$,
- the diagonal incoming node degree matrix $\mathcal{D}^{\text{in}} = \text{diag}(d_1^{\text{in}}, \dots, d_n^{\text{in}})$,

For undirected graphs, outgoing and incoming degrees are identical and we write $d_i = d_i^{\text{in}} = d_i^{\text{out}}$ and $\mathcal{D} = \mathcal{D}^{\text{in}} = \mathcal{D}^{\text{out}}$.

The weighted outgoing and incoming node degrees are hence equal to the row and column sums of the adjacency matrix $\mathcal{A}$, given by

$$[d_1^{\text{out}}, \dots, d_n^{\text{out}}]^T = \mathcal{A}\mathbf{1} \quad \text{and} \quad [d_1^{\text{in}}, \dots, d_n^{\text{in}}]^T = \mathcal{A}^T\mathbf{1}.$$

In this thesis, we will typically refer to the unweighted degrees as *neighbor counts* and to the weighted degrees simply as *degrees*.

The final basic concept that we are going to introduce here are signals and node features. Signals are the central subject of the field of signal processing on graphs [90], which lays the foundation for many important concepts in graph learning. See Section 2.4 for more details.

2.6 Definition. *(Signals on graphs and node features)*
(a) A signal on a graph is a function that assigns a real number to each node, which can be written either as $f : V \to \mathbb{R}$ or $\boldsymbol{f} = (f_i)_{i=1}^n \in \mathbb{R}^n$ with $f(i) \mathrel{\widehat{=}} f_i$.

(b) Given a linear feature space $\mathcal{F}$, a set of *node features* for a graph consists of one feature vector $\boldsymbol{x}_i \in \mathcal{F}$ for each node $i \in V$.

(c) For an m-dimensional Euclidean feature space $\mathcal{F} = \mathbb{R}^m$, a *node feature matrix*

$X = \begin{bmatrix} \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \end{bmatrix}^T \in \mathbb{R}^{n \times m}$ collects the node feature vectors as its rows. The m columns of the matrix each constitute a signal on the graph.

2.2 Types of graphs

Now that we have covered the basic definitions, let us dive into the different variations of graphs that are found in common learning tasks. Graphs can be used to model data in a multitude of settings and applications, and that diversity is reflected in their properties and layout as well as in the way these graphs should be analyzed and interpreted.

2.2.1 Assortative vs. disassortative edges

An important factor in the analysis of graphs for data science is the interpretation of the edge set. If the existence of an edge between two nodes indicates that these nodes share some properties or are similar in some other way, we speak of *assortative* edges. In the context of node classification, this means that two connected nodes can be expected to be more likely to belong to the same class than two nodes without connection. This concept is also known as *homophily* [68].

The opposite is the case of *disassortative* edges or *heterophily*, where connection indicates dissimilarity. The most extreme example would be the binary classification task of recovering the two components of a bipartite graph, where all edges have one endpoint from each component. Interpretations of disassortative edges include expressed dislike in social networks, e.g., through users "blocking" each other's content.

For weighted graphs, this notion is expanded to the edge weights. By design of the weighted adjacency matrix, an edge weight of zero would be identical to no connection at all. Hence a large positive weight $w(e) \gg 0$ encodes a strong connection, i.e., a high degree of similarity and proximity in the assortative case or a large distance and discrepancy in the disassortative case.

While purely disassortative graphs are highly atypical in applications, a mixture of both types of edges may be more common. This kind of information can be encoded in so-called *signed* graphs, a generalization where the usual assortative edge set with positive weights is complemented by disassortative edges that are given negative weights. Signed graphs present their own challenges for dedicated methods and are beyond the scope of this work (see [93] for an overview).

In most data science applications, however, typically the only relevant case is the assortative one, which we hence focus on throughout this work.

2.2.2 Directed vs. undirected edges

One of the first and crucial questions when analyzing a given graph is whether adjacency of one node to another also implies adjacency in the other direction. The answer to that question impacts the analysis from a very early stage. It even changes how edges should be represented mathematically, as a tuple or an edge (cf. Definition 2.2).

For implementation purposes, this may not seem to make that much of a difference because for the adjacency matrix we need to store one entry for each direction of an undirected edge anyway. However, as soon as quantities like node degrees are involved, the impact of whether we need to differentiate between incoming and outgoing degrees (cf. Definition 2.5) becomes apparent.

As for applications, there are different use cases both for directed and undirected graphs. Directedness occurs wherever edges encode a concrete action *from* one data point *to* another, e.g., one user following or contacting another in a social network. Undirectedness on the other hand occurs when edges encode properties of the general relationship between two data points, e.g., whether two publications share a co-author in a citation graph.

Whether or not directed edges should be included in the data should however not only depend on the available data, but also on the learning task at hand. In some tasks, the edge direction plays an important role, for example in flow problems or for the identification of central or highly influential nodes. For the analysis of connectivity, clustering, or classification, on the other hand, it only matters whether two nodes are connected at all and a possible direction cannot be utilized anyway. This is also reflected in the fact that one of the central tools for these tasks, the graph Laplacian, is generally only defined for undirected graphs (cf. Section 2.3). In that context, often a directed edge that only exists in one direction can be thought of as an undirected edge with half the weight.

For that reason, since this thesis focuses on the latter type of tasks, we will generally assume graphs to be undirected.

2.2.3 Sparse graphs with edges intrinsic to the data

The origins of graph learning are deeply connected with their applications on graphs that are purely given by the data in the sense that not only the nodes describe real-world entities, but also every single edge represents a real-world connection. In social networks, edges typically describe acquaintance between people. In citation graphs, two nodes are connected by an edge if they share a co-author.

In these scenarios, if the network is large enough, the case of two random nodes being connected is far less likely than them *not* being connected, since the number of edges per node is limited by real-world factors independent of the network size. No

matter how large the population of a social network is, the average user will still only have connections to a certain number of people. No matter how many publications a citation graph covers, the average number of authors per paper as well as other publications per author remain the same. This means that the average node degree is asymptotically constant, the total number of edges grows linearly with the network size, and the adjacency matrix is sparse. For that reason, we refer to these as sparse graphs.

For numerical methods, this sparsity is directly connected to the scaling of memory and runtime efforts with growing data set size n. If sparsity is present in the way described above, the cost will remain proportional to n. While in other numerical fields this behaviour is more of a convenient special case, many graph learning methods rely on it because of its prevalence in benchmarks and applications.

Hence dedicated methods typically exhibit bad scaling if the graph was ever not sparse. If the average node degree were to scale with n, that considerably lowers the maximum number of nodes for the graph to still be storable in memory.

2.2.4 "Real-world" vs "constructed" graphs

Most tools in graph theory have been naturally defined and derived for graphs from real-world applications. The origins are clear: In order to analyze given graph data, we define auxiliary objects like the graph Laplacian matrix and investigate their properties.

However, the success of graph-based methods has led to the desire to enable this technique in applications where no explicit graph formulation is present. Here, the roles are interchanged: In order to utilize the analytical power of the graph Laplacian and similar tools, a graph is constructed artificially.

Note that this notion of artificiality does not mean that the underlying data itself is fictitious. Artificial, randomly created data and graphs are also important, e.g., for the theoretical analysis of graph methods. However, when we speak of constructed graphs here, we refer to the augmentation of real-world data with a graph, constructed based on given properties.

The common setting is that a set of n data points, or samples, are given, each described by a feature vector $\boldsymbol{x} \in \mathcal{F}$. The missing ingredient to form a graph is the edge set. As said earlier, we focus on assortative, undirected edges, so nodes i, j with a small feature distance $d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ should likely be connected by an edge, $\{i, j\} \in E$. The feature space $\mathcal{F}$ as well as the distance measure $d : \mathcal{F} \times \mathcal{F} \to \mathbb{R}_{\geq 0}$ can be arbitrary, though most applications will use a standard d-dimensional Euclidean vector space.

For this kind of data, the distance measure by itself already provides a notion of proximity and locality that can be utilized independent of any strict graph formulation. Still, the graph point of view allows for an additional level of analysis and

tools.

The following sections shall introduce techniques for creating artificial graphs for real-world applications.

2.2.5 Local neighborhood graphs

In order to construct artificial graphs for which existing graph learning methods produce good results, it is crucial to mimic certain properties of real-world graphs. As mentioned above, one decisive and often required feature is sparsity of the adjacency matrix. For that reason, neighborhood graphs constrain the node degrees by only connecting each node to its closest neighbors.

One way to construct this is to specify a radius $\rho > 0$ and connect two nodes $i \neq j$ if and only if their feature distance is not larger than ρ, $d(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq \rho$. This way, the graph neighborhood of node i consists exactly of the nodes whose feature vectors are contained in the Euclidean ball with radius ρ and center $\boldsymbol{x}_i$. The resulting graph is trivially undirected. Drawbacks are that no level of sparsity can be guaranteed unless ρ is chosen in a very specific way, which requires strong knowledge of the feature space population. Morever, if n grows and new node features are added from the same bounded subdomain of the feature space, node degrees will not be constant unless ρ is decreased with growing n.

An alternative method are k-nearest neighbor graphs, for which the integer parameter k directly determines the neighbor count of each node. For the simple approach, the edge (i, j) is created if and only if the feature $\boldsymbol{x}_j$ is one of the k closest features to $\boldsymbol{x}_i$. This generally results in directed, non-symmetric graphs, as the density around nodes i and j may differ and thus j may be one of the top-k neighbors of i but not vice versa. Strategies for symmetry enforcement include making all edges undirected, optionally with half weight, or only adding edges that pass the threshold in both directions. Another potential drawback is that by itself, this method gives all nodes similar centrality properties, no matter if they're outliers or situated in a dense area. A remedy for this can be given by additionally weighting the edges, as discussed in the upcoming section.

Furthermore, combinations of the two approaches are possible. By specifying a maximum radius as well as a maximum neighbor count, a specific level of sparsity is enforced while outliers are not connected to distant nodes just to fulfill the degree requirement.

2.2.6 Kernel-based edge weights

After an edge set is constructed in some way, it is in many cases beneficial to treat edges differently based on the distance they are spanning in the feature space. This can be achieved by generating edge weights that indicate the connection strengths.

Since we focus on assortative edges, the weights should be monotonically decreasing with the distance to incorporate that shorter distances cause stronger connections.

Edge weights are generally constructed based on a function that maps two nodes' features to the weight of their connecting edge,

$$K : \mathcal{F} \times \mathcal{F} \to \mathbb{R}_{\geq 0}.$$

These functions are called *kernels*, which play important roles in various fields of mathematics and learning methods in particular [12, 88]. Of course, a kernel value of 0 means that no edge should be created in the first place.

In the typical case with standard linear spaces, there is generally no prior knowledge about the meaning of different regions of the feature space, so the kernel should be unbiased with respect to the location and *invariant under translations*. Hence $K(\boldsymbol{x}, \bar{\boldsymbol{x}})$ should only depend on the relative position of $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$ to each other, $\boldsymbol{x} - \bar{\boldsymbol{x}}$.

Furthermore, since different directions in the feature space are equally indistinguishable without prior knowledge, in almost all cases only the length of this difference is used, i.e., the distance $d(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \|\boldsymbol{x} - \bar{\boldsymbol{x}}\|_2$. Kernels that rely only on this distance are called *rotationally invariant* or *radial basis function* (RBF) kernels.

Though Euclidean feature spaces and distance measures are the most important use case, other examples also have their applications. In non-linear feature spaces (e.g., for string features), the difference $\boldsymbol{x} - \bar{\boldsymbol{x}}$ might not exist but some other distance measure $d(\boldsymbol{x}, \bar{\boldsymbol{x}})$ can be used. In addition, some higher-dimensional linear feature spaces may have a richer intrinsic structure that would be lost by a simple Euclidean distance measure. One example of such feature data are time series, i.e., long sequences describing the evolution of some quantity over time. For this type of data, there exist more sophisticated distance measures like Dynamic Time Warping, an algorithm that respects small temporal shifts which would otherwise lead to large Euclidean distances. For details we refer to our manuscript [1] and the references therein. The ability to incorporate any application-specific measures is a strong advantage of kernel-based edge weight.

In this work, we will only look at kernels that are rotationally invariant. We will overload the notation for K and write

$$K(\boldsymbol{x}, \bar{\boldsymbol{x}}) = K(r) \quad \text{with} \quad r = d(\boldsymbol{x}, \bar{\boldsymbol{x}}).$$

2.7 Example. *(The Gaussian kernel)*
The most common rotationally invariant kernel is the Gaussian kernel

$$K(r) = \exp\left(-\frac{r^2}{\sigma^2}\right) \tag{2.1}$$

with a localization parameter σ. Sometimes different parameterizations are seen

with either $\exp(-r^2/(2\sigma^2))$ or $\exp(-\gamma r^2)$. We use the parameterization of (2.1) due to its use in the fast summation literature that lays the foundation for Section 3.1. The value of σ can be interpreted as the distance that yields common kernel break points, mimicking the normal distribution in stochastics with its Empirical Rule. Here, the decay of K can similarly be estimated via

$$K(0) = 1, \quad K(\sigma) \approx 0.37, \quad K(2\sigma) \approx 0.018, \quad K(3\sigma) \approx 0.00012.$$

This means that nodes with a feature distance of around σ still have a significant edge weight while a distance of 3σ yields a nearly vanishing edge weight.

Kernels with similar properties as the Gaussian kernel include the inverse multiquadric kernel, $K(r) = 1/\sqrt{r^2+1}$, or the sinc kernel, $K(r) = \sin(r)/r$ (continuous with $K(0) = 1$, but negative for $r > \pi$).

Other examples do not fit our framework of assortative edges because they are increasing with r, e.g., the multiquadric kernel $K(r) = \sqrt{r^2+1}$.

In some contexts, kernels with a singularity at $r = 0$ play an important role. These include the inverse kernels $K(r) = 1/r$ or $K(r) = 1/r^2$, the logarithmic kernel $K(r) = \log(r)$, or the thin-plate spline $K(r) = r^2 \log(r)$. The latter two are not applicable anyway because of the negativity, but the first two may be if the edge set does not contain loops and each two other connected nodes are guaranteed to have differing feature vectors.

All of these kernels can be parameterized and adjusted to different scalings just like the Gaussian kernel by replacing r with r/σ.

Another type of kernels are not rotationally invariant, but "radial" w.r.t. a different norm, like the "Laplacian RBF" kernel $K(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \exp(-\|\boldsymbol{x} - \bar{\boldsymbol{x}}\|_1/\sigma)$.

2.2.7 Fully connected graphs

If the weight kernel is sufficiently expressive, it is possible to skip the first step of edge selection and instead connect each node to every other node. As opposed to describing the neighborhood of each node via a discrete set of its closest neighbors, the localization information is then encoded in a significantly larger number of edges with a continuous weight decay.

By itself, this method creates edges for each pair of nodes i, j with $i \neq j$. This way, all off-diagonal entries of the adjacency matrix are determined via the kernel. One open choice is whether self-loops are also added with weight $K(0)$, which determines the constant value on the adjacency matrix's diagonal. Generally, the fully connected adjacency matrix has the form

$$\mathcal{A} = (a_{ij})_{i,j=1}^{n} \quad \text{with} \quad a_{ij} = \begin{cases} w_{\text{loop}} & \text{if } i = j, \\ K(\boldsymbol{x}_j - \boldsymbol{x}_i) & \text{else} \end{cases} \tag{2.2}$$

with the constant loop weight $w_{\text{loop}} \geq 0$.

For the Gaussian kernel (2.1), the localization parameter σ directly determines the range of adjacency matrix entries. Suppose that σ is equal to the diameter of the data set in the feature space, i.e., the maximum distance between two node features. Then the edge weights only range from 1 to $\exp(-1) \approx 0.37$. In particular there are no numerical zeros. In single precision, the first edge weights already reach numerical zero (10^{-8}) if σ is smaller than approximately one fourth of the diameter.[1]

In data science, the fully connected Gaussian graph is perhaps most famous for its usage in Spectral Clustering [75]. We will briefly describe this method in Remark 2.17.

If the loop weight is equal to $K(0)$, this structure of the adjacency matrix is called a *kernel matrix* (cf. Section 3.1 and Equation (3.1)). These matrices also appear in different fields and learning methods even without the graph context. The most famous example will be Kernel Machines like Support Vector Machines (see [88, 89] for an overview and details). The most crucial difference lies in the use of rectangular matrices: For two different data point sets $\boldsymbol{x}_i, \boldsymbol{y}_j$, a matrix with entries $K(\boldsymbol{x}_i, \boldsymbol{y}_j)$ can be used to transfer knowledge from one set of points to a new one. This can be done repeatedly for different sets of target points. From the graph viewpoint, however, we always think of one set of data points that is available from the start and we want to analyze the adjacency matrix as a whole.

Kernel matrices are also closely related to the theory of integral equations, which allows for theoretical analysis resulting in Mercer's theorem [69, 89]. It gives a nice characterization of kernels that always result in positive definite kernel matrices, which are also known as Mercer kernels.

2.2.8 Clique expansions of hypergraphs

Neighborhood graphs or fully connected graphs are one way to introduce a notion of adjacency in data sets with features. Given a useful distance metric, they can even be used for discrete feature spaces. However, a sometimes simpler approach is to encode discrete node attributes by constructing a *hypergraph*.

Hypergraphs are a simple generalization of traditional graphs that appear naturally in various contexts [14]. Unlike traditional edges connecting exactly two nodes with each other, a hypergraph is made up of *hyperedges* that each specify a connection between any number of nodes [14, 57]. While there exists a generalization of *directed* hyperedges with designated source and target nodes, we will here only introduce the undirected case where a hyperedge is simply defined as a nontrivial subset of the node set, $\emptyset \neq e \subset V$, without any distinction among its members.

For their usage in data science tasks, suppose that given data point entities are described by discrete features where each feature vector entry refers to a categorial

[1] To be precise, $\exp(-D^2/\sigma^2) \leq 10^{-8} \iff \sigma/D \leq 1/\sqrt{-\ln(10^{-8})} \approx 0.233$, where D denotes the diameter.

property that can have one of K different non-numerical values. Then Euclidean distances between the feature vectors have no real-world meaning and will be misleading. In hypergraphs, one can model these properties by introducing K hyperedges that connect all data points that have the associated property value. If this is done for all feature entries, each node will be in exactly as many hyperedges as there are attributes.

Analogously to traditional graphs, we assume a fixed ordering of the hyperedge set $E = \{e_1, \dots, e_{|E|}\}$ and use that for hyperedge index expressions, cf. Remark 2.3. The whole structure can be given through the hypergraph's incidence matrix

$$\mathcal{H} = (h_{ie})_{i \in V, e \in E} \in \{0,1\}^{n \times E},$$

where each column $(h_{ie})_{i=1}^{n}$ is the indicator vector of e, i.e., $h_{ie} = 1$ if and only if node i is a member of hyperedge e. Optionally a hypergraph can be equipped with *hyperedge weights* w_e that we will store in a diagonal matrix $\mathcal{W} = \operatorname{diag}(w_{e_1}, \dots, w_{e_{|E|}})$.

Node degrees can be generalized straightforwardly as the sum of all hyperedge weights of hyperedges that contain the node in question,

$$d_i = \sum_{e \in E:\ i \in e} w_e = \sum_{e \in E} h_{ie} w_e.$$

Just like in traditional case, we collect the node degrees in a diagonal degree matrix $\mathcal{D} = \operatorname{diag}(d_1, \dots, d_n)$. In addition, we define the degree b_e of each hyperedge $e \in E$ as the unweighted number of contained nodes,

$$b_e = |e| = \sum_{i=1}^{n} h_{ie},$$

and also set up the hyperedge degree matrix $\mathcal{B} = \operatorname{diag}(b_{e_1}, \dots, b_{e_{|E|}}) \in \mathbb{R}^{E \times E}$.

Different ways of handling hypergraphs have been developed for different applications [103, 14, 17]. The approach we will look at here is in many ways the most simple – analyzing a hypergraph via a specially constructed graph known as the *clique expansion* of the hypergraph. The clique expansion graph is defined by its adjacency matrix

$$\mathcal{A} = \mathcal{H}\mathcal{W}\mathcal{B}^{-1}\mathcal{H}^T \quad \text{with} \quad a_{ij} = \sum_{\substack{e \in E: \\ i \in e,\ j \in e}} \frac{w_e}{b_e}. \tag{2.3}$$

As we see, two nodes are connected in the clique expansion iff they share membership in at least one hyperedge, and the edge weight is the sum of the associated hyperedge weights divided by their hyperedge degrees. Thus a connection in the clique expansion is strongest if there are multiple hyperedges that contain the two nodes, but not that many other nodes. The more nodes a hyperedge connects, the less it contributes to

the individual edge weights in the clique expansion. This behavior acknowledges that shared membership in a smaller hyperedge is more meaningful.

The node degrees of the clique expansion are identical to the hyperedge node degrees as defined above, which we can easily see via

$$\sum_{j=1}^{n} a_{ji} = \sum_{j=1}^{n} \sum_{e \in E} h_{ie} h_{je} \frac{w_e}{b_e} = \sum_{e \in E} h_{ie} \frac{w_e}{b_e} \sum_{j=1}^{n} h_{je} = \sum_{e \in E} h_{ie} w_e = d_i.$$

Clique expansion graphs have a few properties that are not commonly found in traditional graphs. First, due to the structure of (2.3), $\mathcal{A}$ must always be positive semi-definite and has a rank of at most $|E|$.

Second, an important part of the clique expansion are its loop weights, i.e., the diagonal adjacency entries $a_{ii} = \sum_{e \in E:\ i \in e} \frac{w_e}{b_e}$. Because the contributing hyperedges are a superset of those contributing to each a_{ij} for $j \neq i$, we see that $a_{ii} \geq a_{ji}$ for all i, j. Specifically, if we express a traditional loop-free graph with adjacency matrix $\tilde{\mathcal{A}}$ and degree matrix $\tilde{\mathcal{D}}$ as a hypergraph (i.e., $|e| = 2$ for all $e \in E$), then the clique expansion has the adjacency matrix $\mathcal{A} = \tilde{\mathcal{A}} + \tilde{\mathcal{D}}$ and degrees $\mathcal{D} = 2\tilde{\mathcal{D}}$.

Despite these minor particularities, clique expansion lets us treat hypergraphs just like regular graphs. In Section 3.3.3, we will discuss efficient numerical techniques for the special case that $\mathcal{A}$ has low rank.

2.3 The Graph Laplacian Operator

The graph Laplacian operator can be considered the most important tool for graph analysis and learning. It enables the investigation of graphs by means of linear algebra. The whole field of spectral graph theory revolves around the study of the graph Laplacian and its eigenvalues. Many different types of statements have been made about connections between the spectrum and combinatorial properties of the graph [19].

Unfortunately there is not just one definition of "the" graph Laplacian, but different variants targeted at varying use cases, problem formulations, and types of graphs. As briefly discussed in Section 2.2.2, we will only introduce the Laplacian for undirected graphs. For directed graphs, one would need to replace each occurrence of the node degree matrix $\mathcal{D}$ with either $\mathcal{D}^{\text{in}}$ or $\mathcal{D}^{\text{out}}$ from Definition 2.5, depending on the situation.

The operator is always linear, which makes it representable as a matrix, though some nonlinear generalizations exist (e.g., [17]). In the literature for traditional graphs, there are two main variants with competing popularity as well as multiple minor variations that are targeted more at specific problems. The two main definitions are a *combinatorial* or *non-normalized* and a *symmetric normalized* Laplacian, which only differ in the use of normalization with the node degrees. There are many

publications that use either only the non-normalized variant [90], only the normalized variant [19], or both [98].

In this section, we will first introduce both variants and their resulting eigenvalue problems and spectral properties. This is done to align this thesis with other works. Then we will discuss motivations and derivations for the matrices from different points of view in order to sharpen an intuition about where their useful features come from.

2.3.1 Definitions and basic properties

Let us start by defining the two main variants of the operator as well as two minor ones with practical applications.

2.8 Definition. *(Variants of the graph Laplacian)*
Consider a undirected graph with adjacency matrix $\mathcal{A}$ and diagonal degree matrix $\mathcal{D}$.
(a) The *combinatorial* or *non-normalized* graph Laplacian is defined as

$$\mathcal{L} = \mathcal{D} - \mathcal{A}. \tag{2.4}$$

(b) The *symmetric normalized* Laplacian is defined as

$$\mathcal{L}_{\text{sym}} = \mathcal{D}^{-1/2}\mathcal{L}\mathcal{D}^{-1/2} = I - \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}. \tag{2.5}$$

(c) The *signless* symmetric normalized Laplacian is defined as

$$\mathcal{L}_{\text{sym+}} = 2I - \mathcal{L}_{\text{sym}} = I + \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}. \tag{2.6}$$

(d) The *random walk* Laplacian is defined as

$$\mathcal{L}_{\text{rw}} = \mathcal{D}^{-1}\mathcal{L} = I - \mathcal{D}^{-1}\mathcal{A}. \tag{2.7}$$

The random walk Laplacian $\mathcal{L}_{\text{rw}}$ is a generally nonsymmetric matrix that acts as the probability matrix for random processes on the graph, which leads to its own special properties [19]. It is only listed here for completeness, as it is arguably as well known as the two main definitions but less impactful to our theory. The signless Laplacian will play an important role in Section 3.3.

Note that the normalization with $\mathcal{D}^{-1/2}$ or $\mathcal{D}^{-1}$ is only defined if no node degree is zero, i.e., each node is connected with at least one edge. For that reason, we will always assume that the graph has no isolated nodes, which would be impractical in learning settings anyway.

2.9 Proposition. *(Properties of the graph Laplacian variants)*
(a) The combinatorial graph Laplacian $\mathcal{L}$ is invariant under the addition of loops

with arbitrary weights. In other words, if two graphs differ only in the presence of loops or their weights, they share the same combinatorial graph Laplacian.

(b) The combinatorial graph Laplacian satisfies

$$\boldsymbol{f}^T \mathcal{L} \boldsymbol{f} = \frac{1}{2} \sum_{i,j=1}^{n} a_{ij} \left(f_i - f_j\right)^2 \geq 0 \qquad \forall\, \boldsymbol{f} \in \mathbb{R}^n. \tag{2.8}$$

(c) The symmetric normalized graph Laplacian satisfies

$$\boldsymbol{f}^T \mathcal{L}_{\text{sym}} \boldsymbol{f} = \frac{1}{2} \sum_{i,j=1}^{n} a_{ij} \left(\frac{f_i}{\sqrt{d_i}} - \frac{f_j}{\sqrt{d_j}}\right)^2 \geq 0 \qquad \forall\, \boldsymbol{f} \in \mathbb{R}^n. \tag{2.9}$$

(d) The signless symmetric normalized graph Laplacian satisfies

$$\boldsymbol{f}^T \mathcal{L}_{\text{sym+}} \boldsymbol{f} = \frac{1}{2} \sum_{i,j=1}^{n} a_{ij} \left(\frac{f_i}{\sqrt{d_i}} + \frac{f_j}{\sqrt{d_j}}\right)^2 \geq 0 \qquad \forall\, \boldsymbol{f} \in \mathbb{R}^n. \tag{2.10}$$

(e) $\mathcal{L}$, $\mathcal{L}_{\text{sym}}$, and $\mathcal{L}_{\text{sym+}}$ are all symmetric positive semi-definite.

Proof. For 2.9(a), note that increasing the value of a loop weight a_{ii} by α also increases the degree d_i. Hence the only changed value is $\mathcal{L}_{ii}^{\text{new}} = (d_i+\alpha)-(a_{ii}+\alpha) = \mathcal{L}_{ii}$.

2.9(b) can be seen by reformulation of the binomial expression

$$\begin{aligned}\sum_{i,j=1}^{n} a_{ij} \left(f_i - f_j\right)^2 &= \sum_{i=1}^{n} f_i^2 \sum_{j=1}^{n} a_{ij} - 2 \sum_{i,j=1}^{n} a_{ij} f_i f_j + \sum_{j=1}^{n} f_j^2 \sum_{i=1}^{n} a_{ij} \\ &= 2 \sum_{i=1}^{n} f_i^2 d_i - 2 \sum_{i,j=1}^{n} f_i a_{ij} f_j = 2\left(\boldsymbol{f}^T \mathcal{D} \boldsymbol{f} - \boldsymbol{f}^T \mathcal{A} \boldsymbol{f}\right) = 2 \boldsymbol{f}^T \mathcal{L} \boldsymbol{f}.\end{aligned}$$

2.9(c) and 2.9(d) follow by replacing $\boldsymbol{f}$ with $\mathcal{D}^{-1/2}\boldsymbol{f}$ and switching the sign. For 2.9(e), the symmetricity is inherited from $\mathcal{A}$ because the graph is undirected, and the positive semi-definiteness is directly proven by the previous parts. □

2.3.2 Laplacian eigenvalue problems and spectral properties

The theoretical interpretability and practical applicability of the spectrum of the graph Laplacian operators can hardly be overstated. This section deals with the most important definitions and results. While the interpretation of the eigenvalues, particularly the Fiedler value, is backed up by strong mathematical theory and estimates [19], similar interpretations of the corresponding eigenvectors relies much more

on empirical observations and intuitions, especially regarding their smoothness and clustering capabilities. Nevertheless, these weak statements yield an important contribution to the understanding of why many graph learning methods work as well in practice as they do.

2.10 Definition. *(Laplacian eigenvalue problems)*
Consider an undirected graph. The following eigenvalue problems (EVPs) are posed to find eigenpairs consisting of an eigenvalue $\lambda \in \mathbb{C}$ and a nonzero eigenvector $0 \neq \boldsymbol{u} \in \mathbb{C}^n$ such that:

(a) The combinatorial Laplacian EVP: $\mathcal{L}\boldsymbol{u} = \lambda\boldsymbol{u}$

(b) The generalized Laplacian EVP: $\mathcal{L}\boldsymbol{u} = \lambda\mathcal{D}\boldsymbol{u}$

(c) The (symmetric) normalized Laplacian EVP: $\mathcal{L}_{\text{sym}}\boldsymbol{u} = \lambda\boldsymbol{u}$

(d) The random walk Laplacian EVP: $\mathcal{L}_{\text{rw}}\boldsymbol{u} = \lambda\boldsymbol{u}$

2.11 Proposition. *(Basic spectral properties)*
(a) The following are equivalent for $\lambda \in \mathbb{C}$ and $0 \neq \boldsymbol{u} \in \mathbb{C}^n$:

- $\lambda, \boldsymbol{u}$ is an eigenpair for the generalized EVP 2.10(b).
- $\lambda, \mathcal{D}^{1/2}\boldsymbol{u}$ is an eigenpair for the normalized EVP 2.10(c).
- $\lambda, \boldsymbol{u}$ is an eigenpair for the random walk EVP 2.10(d).
- $\tilde{\lambda} = 2 - \lambda,\ \tilde{\boldsymbol{u}} = \mathcal{D}^{1/2}\boldsymbol{u}$ is an eigenpair of the normalized signless Laplacian, $\mathcal{L}_{\text{sym}+}\tilde{\boldsymbol{u}} = \tilde{\lambda}\tilde{\boldsymbol{u}}$.

(b) All four problems in Definition 2.10 have n real-valued eigenpairs $\lambda_i \in \mathbb{R}$, $\boldsymbol{u}_i \in \mathbb{R}^n$ $(i = 1, \ldots, n)$ such that the eigenvectors $\boldsymbol{u}_i$ form a basis of $\mathbb{R}^n$.

(c) All eigenvalues of the combinatorial EVP 2.10(a) are non-negative. For the other three EVPs, all eigenvalues are contained in $[0, 2]$.

Proof. 2.11(a) can be seen easily by basic arithmetic reformulation and substitution. 2.11(b) follows from the symmetricity of $\mathcal{L}$ and $\mathcal{L}_{\text{sym}}$ together with 2.11(a). The bounds in 2.11(c) follow from Proposition 2.9(e): The non-negativity is due to $\mathcal{L}$ and $\mathcal{L}_{\text{sym}}$ being positive semi-definite, and the upper bound follows from the fact that $2 - \lambda$ must be an eigenvalue of the positive semi-definite matrix $\mathcal{L}_{\text{sym}+}$. □

When talking about the normalized Laplacian and an associated eigenpair $\lambda, \boldsymbol{u}$ of EVP 2.10(c), the vector $\boldsymbol{f} = \mathcal{D}^{-1/2}\boldsymbol{u}$ is sometimes called a *harmonic eigenfunction* of λ [19]. It is a corresponding eigenvector of the generalized EVP 2.10(b).

2.12 Proposition. *(The eigenvalue 0 and connected components)*
Consider an undirected graph with K connected components, i.e., the node set can be partitioned into $V = \bigcup_{k=1}^{K} V_k$ such that nodes i, j can be connected by a path if and only if they belong to the same component V_k.
(a) $\lambda_1 = 0$ is the smallest eigenvalue of all Laplacian EVPs from Definition 2.10, and its multiplicity is equal to K.

(b) The eigenspace of eigenvectors corresponding to $\lambda_1 = 0$ for EVP 2.10(c) is spanned by the weighted indicator vectors $\tilde{\boldsymbol{\chi}}_k$, $k = 1, \ldots, K$, with entries

$$(\tilde{\boldsymbol{\chi}}_k)_i = \begin{cases} \sqrt{d_i} & \text{if } i \in V_k, \\ 0 & \text{else.} \end{cases} \tag{2.11}$$

(c) The eigenspaces of eigenvectors corresponding to $\lambda_1 = 0$ for the other three EVPs are spanned by the indicator vectors $\boldsymbol{\chi}_k$, $k = 1, \ldots, K$, with entries

$$(\boldsymbol{\chi}_k)_i = \begin{cases} 1 & \text{if } i \in V_k, \\ 0 & \text{else.} \end{cases} \tag{2.12}$$

(d) If the graph is connected, $K = 1$, then the eigenspaces corresponding to $\lambda_1 = 0$ are spanned by the single vector $\tilde{\boldsymbol{\chi}}_1 = \mathcal{D}^{1/2}\mathbf{1}$ for EVP 2.10(c) or $\boldsymbol{\chi}_1 = \mathbf{1}$ for the other three EVPs.

Proof. Both $\mathcal{L}$ and $\mathcal{L}_{\text{sym}}$ are positive semi-definite (Proposition 2.9(e)) and can only have non-negative eigenvalues. We first take a look at EVP 2.10(c). According to the Courant-Fischer min-max theorem (cf. Proposition 2.15(b)), the smallest eigenvalue is 0 with eigenvector $\boldsymbol{f} \neq 0$ if and only if $\boldsymbol{f}^T \mathcal{L}_{\text{sym}} \boldsymbol{f} = 0$. Because that means that all summands of (2.9) must be zero, this is satisfied if and only if $\frac{f_i}{\sqrt{d_i}} = \frac{f_j}{\sqrt{d_j}}$ for all pairs i, j where $\mathcal{A}_{ij} > 0$. By transitivity, $\frac{f_i}{\sqrt{d_i}}$ must be constant on each connected component, so $f_i = c_k\sqrt{d_i}$ for all $i \in V_k$, with one constant c_k per component. For the whole vector, this is equivalent to

$$\boldsymbol{f} = \sum_{k=1}^{K} c_k \tilde{\boldsymbol{\chi}}_k \quad \Leftrightarrow \quad \boldsymbol{f} \in \operatorname{span}(\tilde{\boldsymbol{\chi}}_1,\ \ldots,\ \tilde{\boldsymbol{\chi}}_K),$$

proving part 2.12(a) for EVP 2.10(c) as well as part 2.12(b).

The corresponding results for EVPs 2.10(b) and 2.10(d) follow from Proposition 2.11(a) due to $\tilde{\boldsymbol{\chi}}_k = \mathcal{D}^{1/2}\boldsymbol{\chi}_k$. The zero eigenspace of EVP 2.10(a) is identical to that of EVP 2.10(b). Finally, if $K = 1$, part 2.12(d) follows with $\tilde{\boldsymbol{\chi}}_1 = \mathcal{D}^{1/2}\mathbf{1}$ and $\chi_1 = \mathbf{1}$. □

From now on, we will only talk about the symmetric normalized Laplacian $\mathcal{L}_{\text{sym}}$ instead of $\mathcal{L}$. Keep in mind that all of these concepts can be transferred easily to the non-normalized Laplacian. In fact, many publications do rather work without normalization. For the use in Graph Convolutional Networks, the normalized variant has proven to be beneficial [56], partly due to the boundedness of its spectrum, cf. Proposition 2.11(c).

For convenience, we introduce a dedicated notation for the spectrum of the normalized Laplacian. Unless stated otherwise, these identifiers will always refer to the eigenvalues and eigenvectors of $\mathcal{L}_{\text{sym}}$.

2.13 Definition. *(Ordered normalized Laplacian spectrum)*
We denote the n real-valued eigenpairs of $\mathcal{L}_{\text{sym}}$ by $\lambda_i \in [0, 2]$ and $\boldsymbol{u}_i \in \mathbb{R}^n$ for $i = 1, \dots, n$, such that

- $\mathcal{L}_{\text{sym}}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i$,
- the eigenvalues are sorted, $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n \leq 2$,
- the eigenvectors form an orthonormal basis of $\mathbb{R}^n$, $\boldsymbol{u}_i^T \boldsymbol{u}_j = \delta_{ij}$,
- the first eigenvector is fixed as $\boldsymbol{u}_1 = \mathcal{D}^{1/2}\mathbf{1}$, following Proposition 2.12.

All other eigenvectors are arbitrary representatives of the corresponding eigenspaces. We collect these eigenpairs in matrices

$$\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n) \in \mathbb{R}^{n \times n} \quad \text{and} \quad U = \big[\boldsymbol{u}_1, \, \dots, \, \boldsymbol{u}_n\big] \in \mathbb{R}^{n \times n}$$

that satisfy

$$\mathcal{L}_{\text{sym}} U = U\Lambda, \qquad U^T U = I.$$

2.3.3 Smoothness, noise, and clustering information

In optimal control problems on a continuous domain, it is often desired that the solution should be sufficiently *smooth*, i.e., sharp jumps and large gradients should be discouraged. This can be achieved by adding a Tikhonov regularization term to the objective function, for example using the Dirichlet energy term $\|\nabla f\|_2^2$ that is the squared Sobolev W_2^2 semi-norm [4].

On graphs, a simple generalization yields the energy term $\|\nabla \boldsymbol{f}\|_2^2 = \boldsymbol{f}^T \mathcal{L} \boldsymbol{f}$, as will be motivated in Section 2.3.5. Following (2.8), this expression is smaller if $\boldsymbol{f}$ has more similar values across edges, and increases if there are larger differences across edges. This concept is common in graph theory [46] and specifically signal processing on graphs [90, 15].

Again, we will use the normalized variant of the graph Laplacian to achieve the

same effect. For formal treatment, let us introduce the following two functionals on graph signals.

2.14 Definition. *(Smoothing functional & Rayleigh quotient)*
On an undirected graph, define for all signals $\boldsymbol{f} \in \mathbb{R}^n$
(a) the *smoothing functional*

$$S(\boldsymbol{f}) = \|\nabla(\mathcal{D}^{-1/2}\boldsymbol{f})\|_2^2 = \boldsymbol{f}^T \mathcal{L}_{\text{sym}} \boldsymbol{f}, \tag{2.13}$$

and

(b) the *Rayleigh quotient* w.r.t. $\mathcal{L}_{\text{sym}}$,

$$R(\boldsymbol{f}) = \frac{S(\boldsymbol{f})}{\|\boldsymbol{f}\|_2^2} = \frac{\boldsymbol{f}^T \mathcal{L}_{\text{sym}} \boldsymbol{f}}{\boldsymbol{f}^T \boldsymbol{f}}. \tag{2.14}$$

We say that signals $\boldsymbol{f} \in \mathbb{R}^n$ with a small value $S(\boldsymbol{f})$ are *smooth* and those with a large value are *noisy*. According to 2.9(c), smoothness indicates similar values in the large clusters. Noisy signals on the other hand are those with big jumps across highly weighted edges, which we also refer to as *high-frequency oscillations* (cf. Section 2.4).

Note that $S(\boldsymbol{f})$ scales quadratically with $\|\boldsymbol{f}\|_2$, so minimizing $S(\boldsymbol{f})$ as a standalone objective will simply push $\boldsymbol{f}$ to be zero. The Rayleigh quotient can be understood as a "relative" smoothness functional that avoids this problem. Signals with a small value $R(\boldsymbol{f})$ (or a small value $S(\boldsymbol{f})$ despite a fixed norm $\|\boldsymbol{f}\|_2$) can be expected to have meaningful nonzero entries on a significant part of the graph and those values are similar among clusters. Hence such signals are suitable for cluster detection and we say they *contain clustering information*. This excludes signals like the zero eigenvector $\boldsymbol{u}_1$, which is only capable of "detecting" the whole graph as one cluster.

We can use this notion to assess the eigenvectors of $\mathcal{L}_{\text{sym}}$. This analysis also provides the basis for graph convolution later on; see [15].

2.15 Proposition. *(Smoothness of eigenvectors)*
(a) The eigenvectors $\boldsymbol{u}_k$ $(k = 1, \dots, n)$ as in Definition 2.13 have the smoothness value and Rayleigh quotient

$$S(\boldsymbol{u}_k) = R(\boldsymbol{u}_k) = \boldsymbol{u}_k^T \, \mathcal{L} \, \boldsymbol{u}_k = \lambda_k,$$

so they are decreasing in smoothness for increasing k.

(b) A variant of the Courant-Fischer min-max theorem (for a proof see [51, Theorem 4.2.2 and Equation (4.2.7)]; also known as the Rayleigh-Ritz theorem [98]) gives the

characterization

$$\lambda_k = \underset{\boldsymbol{f}\in\mathbb{R}^n}{\arg\min}\ S(\boldsymbol{f}) = \boldsymbol{f}^T\mathcal{L}_{\text{sym}}\boldsymbol{f} \quad \text{s.t.} \quad \boldsymbol{f}^T\boldsymbol{u}_i(L) = 0\ \ \forall\ i<k, \quad \|\boldsymbol{f}\|_2 = 1$$

$$= \underset{\boldsymbol{f}\in\mathbb{R}^n}{\arg\min}\ R(\boldsymbol{f}) = \frac{\boldsymbol{f}^T\mathcal{L}_{\text{sym}}\boldsymbol{f}}{\boldsymbol{f}^T\boldsymbol{f}} \quad \text{s.t.} \quad \boldsymbol{f}^T\boldsymbol{u}_i(L) = 0\ \ \forall\ i<k, \quad \|\boldsymbol{f}\|_2 > 0.$$

For both problems, the eigenvector $\boldsymbol{u}_k$ is a minimizer. This means that $\boldsymbol{u}_k$ is the *smoothest* vector with norm 1 that does not have any components in direction of a previous eigenvector.

Naturally, the eigenvectors to the smallest eigenvalues (except for $\boldsymbol{u}_1$) are of special interest due to their smoothness and clustering capabilities. We call the first nonzero eigenvectors $\boldsymbol{u}_2, \boldsymbol{u}_3, \ldots$ the *informative* eigenvectors, though there is obviously no hard boundary separating them from the non-informative eigenvectors. The second eigenvector is particularly well known.

2.16 Definition. *(Algebraic connectivity and the Fiedler vector)*
(a) For any undirected graph, the second smallest eigenvalue $\lambda_2 \geq 0$ is called the *algebraic connectivity* or *Fiedler value* of the graph. It can also be referred to as the *spectral gap* or *eigengap* of $\mathcal{L}_{\text{sym}}$ or simply *the eigenvalue* of the graph.
Following Proposition 2.12(a), λ_2 is the smallest nonzero eigenvalue for connected graphs and zero for disconnected graphs. The smaller λ_2, the closer the graph is to being disconnected. Spectral graph theory establishes theoretical relationships between the combinatorial Fiedler value and properties of the graph [19].

(b) The eigenvector $\boldsymbol{u}_2$ corresponding to the Fiedler value is called the *Fiedler vector*. It is the smoothest vector orthogonal to $\boldsymbol{u}_1$ (cf. Proposition 2.15) and known to exhibit clustering information of the graph [29, 19].

We conclude this section with a few final observations about the use of eigenvectors for clustering tasks.

2.17 Remark.
(a) The algebraic connectivity is related to how many edges need to be removed for the graph to become disconnected [19]. Empirically, graphs with a sparse adjacency matrix have thus very small connectivities. With growing density of the graph, we expect λ_2 to grow.

(b) Splitting the graph into two parts based on the signs of the Fiedler vector's entries gives a simply binary clustering [98] (see also the upcoming motivation in Section 2.3.6).

(c) A more advanced development of Fiedler vector clustering is *Spectral Clustering*, where the first K eigenvectors of $\mathcal{L}_{\text{sym}}$ are computed and used as node features.

These K-dimensional feature vectors are then projected onto the sphere with radius one and passed to the simple Euclidean clustering tool *kmeans*. For more details, we refer to [98]. Spectral clustering will be utilized in Section 3.5.2.

(d) A larger gap between λ_K and λ_{K+1} indicates that there is no vector with similar suitability for clustering as $\boldsymbol{u}_K$ that doesn't contain components in direction of the first K eigenvectors. This may imply that the graph can be split into K similar clusters.

2.3.4 Examples of graphs with varying spectral properties

In order to illustrate how spectral properties like algebraic connectivity relate empirically to more tangible spatial properties, we give three examples of graphs from different backgrounds. These examples serve as empirical evidence of the observation in Remark 2.17 stating that sparser graphs tend to have smaller eigengaps λ_2.

The data set descriptions are partly taken from our paper [3].

2.18 Example. *(Citation graphs)*
A standard benchmark of semi-supervised graph learning methods (cf. Chapter 4) are the popular *citation graphs* Citeseer, Cora, and Pubmed.[2] In these directed, unweighted graphs, each node corresponds to a scientific publication and each edge $\vec{e} = (i, j)$ indicates that paper i cites paper j. The task is often to split the publications into their scientific fields. For this, the direction of an edge $\vec{e}$ is not important due to the chronological constraints: If paper i cites paper j, it must be published *after* j, but were it the other way around, j might quite likely have cited i. Hence we just treat all edges as undirected.
As stated in Section 2.2.3, real-world constraints lead to all these graphs being highly sparse. The full network statistics are given in Table 2.1. Note that there are only around two edges per node in all of these and the average degree (and the average number of entries in each row of $\mathcal{A}$) is around 4. Only for Pubmed, these few edges are arranged in a way that the graph is connected. The algebraic connectivity in that case is still almost zero, which states that the graph is very close to being disconnected. Citeseer and Cora, on the other hand, are simply not connected and thus have an algebraic connectivity of zero. But it's not just that there are many components without connection between one another, also the components themselves are only weakly connected. This is demonstrated by the fact that the gap between $\lambda_K = 0$ and $\lambda_{K+1} > 0$ is very small, with K denoting the number of connected components.
The data sets are typically associated with binary feature vectors for each publication that describe the occurrence of certain key words. A feature value of 1 implies that the associated word is contained in the paper. The feature dimension, i.e., the number of words in the dictionary, is $d = 3\,703$ for Citeseer, $d = 1\,433$ for Cora, and $d = 500$ for Pubmed.

2.19 Example. *(Fully connected Gaussian graphs for 3D point clouds)*
As stated in Section 2.2.4, there are many applications where constructing a graph based on data point features provides beneficial insight into the data set structure. One such example are collections of points in three-dimensional space called *point clouds*. These clouds often consist of points on the surface of a real-world object and can be generated, e.g., using LiDAR scans [76].
Here we use a part of a subset of the popular Oakland data set, as used in [73] and made available on the project website.[3] The data set consists of multiple point clouds that are typically used for point cloud segmentation [76]. We instead separate two point clouds from the data set and analyze them independently. We refer to these clouds as Oakland 1 (original training cloud) and Oakland 2 (original validation cloud).
We turn these points into graphs using the fully connected approach (cf. Section 2.2.7) with Gaussian edge weights (cf. Section 2.2.6), i.e., the edge connecting the points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ has the weight $\exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2/\sigma^2)$. As stated in these sections, the parameter σ controls the density of the graph. A small value for σ makes the adjacency matrix numerically sparse, while a value equal to the data set diameter will result in a minimum edge weight of $\frac{1}{\mathrm{e}}$.
The spectral and spatial properties are again listed in Table 2.1. The densities for $\sigma = 10$ are less than 100% because in single precision, we can count edge weights smaller than 10^{-8} as numerically zero. The eigenvalues were here approximated using the approach from Chapter 3. Note that despite the still quite high density and average degree, the connectivity for $\sigma = 10$ is very low. This is due to the special structure of the point clouds that consist of multiple well-separated regions with high concentration of points. Notice how much λ_K grows with the parameter σ, confirming that *denser* graphs have larger connectivities.

2.20 Example. *(Hypergraphs for categorial data)*
As our final example, we will look at three data sets from the UCI machine learning repository [27]. In their basic form, these data sets do not describe adjacency information, but only a set of attributes for each node. Because these features are categorial, we can turn the data sets into *hypergraphs* (cf. Section 2.2.8).
The Mushroom data set,[4] describes each of its $n = 8\,124$ samples by 21 categorical attributes (ignoring one with missing values). For each attribute, we create as many hyperedges as there are attribute values present, where each hyperedge connects all samples with a specific value.
The Covertype data set[5] contains a total of 581 012 samples from 7 classes, described

[2]The more common term is *citation network*, but we will stick with our convention to reserve the term network for neural networks.
[3]`http://www.cs.cmu.edu/~vmr/datasets/oakland_3d/cvpr09/doc/`

Table 2.1: Spatial and spectral properties of the graphs from Examples 2.18, 2.19, and 2.20. The symbols denote: σ – Gaussian parameter, n – nodes, $|E|$ – edges or hyperedges, d – feature dimension, Diam. – diameter (for point clouds), Dens. – density of $\mathcal{L}$, d_{avg} – average weighted node degree, K – connected components, λ_2 – algebraic connectivity.

Graph	σ	n	$\lvert E\rvert$	d	Diam.	Dens.	d_{avg}	K	λ_2
Citeseer		3 327	4 552	3 703		0.11%	2.74	438	0
Cora		2 708	5 278	1 433		0.18%	3.90	78	0
Pubmed		19 717	44 324	500		0.03%	4.50	1	0.0095
Oakland 1	10	36 932		3	112.1	87%	6 524	1	0.084
	100					100%	34 177	1	0.929
Oakland 2	10	91 515		3	126.9	70%	11 206	1	0.002
	100					100%	79 189	1	0.872
Mushroom		8 124	112	112		100%	21	1	0.67
Covertype45		12 240	104	104		95%	12	1	0.58
Covertype67		37 877	125	125		94%	12	1	0.59

by 10 continuous and 44 binary attributes. We follow the setup process from [47], dividing the value range of each continuous attribute into 10 equally sized bins and creating hyperedges that connect all samples with values in the same bin. For each binary attribute, we only create one hyperedge for those samples with a "true" value. Afterwards, we create two subhypergraphs by using only samples from classes 4 and 5, or 6 and 7. Because we remove all hyperedges with less than two nodes, the resulting hypergraphs have less than the original 144 hyperedges.
In both examples, all hyperedges have weight $w_e = 1$. After setting up the clique expansion's adjacency matrix as in (2.3), we analyze the resulting graph just like any other. The full information is again given in Table 2.1. The Laplacians are quite dense, which is again reflected in the large connectivity values, but recall that the rank of the adjacency matrix is only equal to the number of hyperedges $|E|$. This special structure will be exploitable in Section 3.3.3.

[4] `https://archive.ics.uci.edu/ml/datasets/Mushroom`
[5] `https://archive.ics.uci.edu/ml/datasets/Covertype`

2.3.5 Motivation via differential operators

As the name suggests, the graph Laplacian operator is related to the generalization of the Laplacian differential operator, which is at the heart of many important Partial Differential Equations (PDEs) like the heat equation and the wave equation. There, the Laplacian operator Δ is defined as the divergence of the gradient.

This section is loosely based on definitions in [16]. We assume an undirected graph without loops w.l.o.g. (cf. Proposition 2.9(a)). The underlying concepts of differential analysis are only summarized briefly for readers already familiar with them. In particular, we use common notation for function spaces without proper introduction. See, e.g., [30] for detailed definitions.

Generalization of vector fields. On open domains $\Omega \subset \mathbb{R}^d$, a vector field is a function F that maps each point $\boldsymbol{x} \in \Omega$ to a direction $F(\boldsymbol{x}) \in \mathbb{R}^d$. The Euclidean inner product, $\boldsymbol{\eta}^T F(\boldsymbol{x})$ defines a flow from point $\boldsymbol{x}$ in the direction of $\boldsymbol{\eta} \in \mathbb{R}^d$. $F(\boldsymbol{x})$ itself can be thought of as a main flow or steepest ascent direction. On manifolds, directions from a point $\boldsymbol{x}$ are restricted to the tangent space. On undirected graphs, however, the valid directions from node i are all directed edges starting at i. This set of directions is finite and doesn't allow addition or scaling like a linear space. Hence the flow from i in direction $\vec{e} = (i, j)$ can't be given through an inner product but must be specified explicitly.

Recall Definition 2.2, particularly the undirected edge set E and directed edge set $\vec{E}$. An *edge field* $\boldsymbol{F}$ thus contains the scalar flow F_{ij} and F_{ji} along both directions of each undirected edge $\{i, j\}$. In order to maintain axiomatic behavior of vector fields, the edge field must be antisymmetric, i.e., $F_{ij} = -F_{ji}\ \forall (i, j) \in \vec{E}$. The set of edge fields can be equipped with the inner product $\langle\!\langle \boldsymbol{F}, \tilde{\boldsymbol{F}} \rangle\!\rangle = \sum_{\{i,j\} \in E, i<j} F_{ij} \tilde{F}_{ij}$ to form a Hilbert space. With this, edge fields are vectors in $\mathbb{R}^E$ where $F_{\{i,j\}}$ contains the flow in the direction with $i < j$. These vectors can be canonically expanded to $\mathbb{R}^{\vec{E}}$.

Generalization of the gradient. The gradient ∇f of a smooth function $f : \Omega \to \mathbb{R}$ on an open domain $\Omega \subset \mathbb{R}^d$ is a vector field that serves two purposes: $(\nabla f)(\boldsymbol{x})$ is the direction of steepest ascent from point $\boldsymbol{x} \in \Omega$, and scalar products $\boldsymbol{\eta}^T (\nabla f)(\boldsymbol{x})$ yield the derivative in direction of $\boldsymbol{\eta} \in \mathbb{R}^d$. On manifolds, this can again be generalized using directional derivatives for elements of the tangent space. For functions defined on undirected graphs, $f : V \to \mathbb{R}$ or $\boldsymbol{f} \in \mathbb{R}^n$ (cf. Definition 2.6(a)), $\nabla \boldsymbol{f}$ must be an edge field such that $(\nabla \boldsymbol{f})_{ij}$ is the derivative at node i along edge $\vec{e} = (i, j)$. To get a notion of such a derivative, we can think of the affine linear continuation of $\boldsymbol{f}$ onto the edge $\vec{e}$. The slope of this continuation is constant but depends on the distance between i and j. Since this distance should be decreasing with increasing similarity (cf. Section 2.2.6), let us assign the distance $\mathrm{dist}(i, j) = 1/\sqrt{w(i, j)}$ based on the

edge weight $w(i,j)$. Hence the derivative of $\boldsymbol{f}$ at i along (i,j) can be viewed as

$$(\nabla \boldsymbol{f})_{ij} = \frac{f_j - f_i}{\operatorname{dist}(i,j)} = \sqrt{w(i,j)}(f_j - f_i).$$

This operation is clearly antisymmetric, so $\nabla \boldsymbol{f}$ is an edge field that can be described as a vector in $\mathbb{R}^E$. This way the gradient operator can be expressed as a matrix

$$\nabla \in \mathbb{R}^{E\times n} \quad \text{with} \quad \nabla_{e,i} = \begin{cases} \sqrt{w(i,j)} & \text{if } e = \{i,j\} \text{ with } j < i, \\ -\sqrt{w(i,j)} & \text{if } e = \{i,j\} \text{ with } j > i, \\ 0 & \text{else.} \end{cases}$$

Generalization of divergence. On open domains, the divergence $\operatorname{div} F$ of a vector field F is the density of the flux out of subvolumes of Ω following the vector field. Informally, $(\operatorname{div} F)(\boldsymbol{x})$ describes the difference between inward and outward flow to and from $\boldsymbol{x}$. Divergence is often described as the pointwise adjoint operator of the gradient, $\operatorname{div} = \nabla\cdot$, in that $\operatorname{div} F(\boldsymbol{x}) = (\nabla \cdot F)(\boldsymbol{x}) = \sum_{j=1}^{d} \frac{\partial F_j}{\partial x_j}(\boldsymbol{x})$. However, for a smooth function that is zero on the domain's border, $f \in \mathcal{H}_0^1(\Omega)$, and a vector field $F \in \mathcal{H}^1(\Omega)^d$, Gauss's theorem yields $\langle \nabla f, F\rangle_{L_2(\Omega)^d} = \langle f, -\operatorname{div} F\rangle_{L_2(\Omega)}$ [30]. Hence on these function spaces, div can be thought of as the negative adjoint operator of ∇ w.r.t. the L_2 inner product.

The divergence of an edge field $\boldsymbol{F}$ on an undirected graph is typically defined as

$$(\operatorname{div} \boldsymbol{F})_i = \sum_{j\in\mathcal{N}_i} \sqrt{w(i,j)} F_{ji},$$

a weighted sum of all outgoing and incoming flow components. Same as in the continuous case, a positive sign means that there is a stronger flow *towards* node i than *away* from it. However, now the divergence is the *positive* adjoint of the gradient, which can be seen in matrix notation via

$$\begin{aligned}(\nabla^T \boldsymbol{F})_i &= \sum_{e\in E} \nabla_{e,i} F_e = \sum_{\substack{j<i: \\ \{i,j\}\in E}} \sqrt{w(j,i)} F_{ji} - \sum_{\substack{j>i: \\ \{i,j\}\in E}} \sqrt{w(i,j)} F_{ij} \\ &= \sum_{j\in\mathcal{N}_i} \sqrt{w(i,j)} F_{ji} = (\operatorname{div} \boldsymbol{F})_i.\end{aligned}$$

Generalization of the Laplacian. For smooth functions $f \in \mathcal{H}^2(\Omega)$, the Laplacian operator is the divergence of the gradient, $\Delta f = \operatorname{div} \nabla f$. Due to the above adjoint relation, for functions that are zero on the boundary, we have $\langle f, \Delta f\rangle_{L_2(\Omega)} = -\langle \nabla f, \nabla f\rangle_{L_2(\Omega)^d} \leq 0$, so Δ is a negative semi-definite operator on $\mathcal{H}_0^2(\Omega)$ w.r.t. the L_2 norm.

Using the above generalizations of the gradient and divergence, the Laplacian can also be defined as the divergence of the gradient, yielding the symmetric positive semi-definite matrix $\Delta = \nabla^T \nabla$. The action of this operator is

$$(\Delta \boldsymbol{f})_i = \operatorname{div}(\nabla \boldsymbol{f})_i = \sum_{j \in \mathcal{N}_i} \sqrt{w(i,j)} (\nabla \boldsymbol{f})_{ji} = \sum_{j \in \mathcal{N}_i} \sqrt{w(i,j)} \sqrt{w(i,j)} (f_i - f_j)$$

$$= d_i f_i - \sum_{j \in \mathcal{N}_i} w(i,j) f_j = \sum_{j=1}^{n} \Delta_{ij} f_j \quad \text{with} \quad \Delta_{ij} = \begin{cases} d_i - a_{ii} & \text{if } i = j, \\ -a_{ij} & \text{else.} \end{cases}$$

Hence, with this choice, the motivated generalization of the graph Laplacian is indeed equal to the combinatorial graph Laplacian definition from 2.8(a), $\Delta = \mathcal{L} = \mathcal{D} - \mathcal{A}$.

Partial differential equations (PDEs) on graphs. From this motivation we can conjecture that the graph Laplacian creates similar dynamics on a graph domain as the original Laplacian does on continuous domains. For example, most simple diffusion equation is the homogeneous heat equation,

$$\frac{\partial}{\partial t} f(\boldsymbol{x}, t) = \Delta f(\boldsymbol{x}, t) \quad \forall\, \boldsymbol{x} \in \Omega,\, t \in [0, \infty),$$

with its graph equivalent $\frac{\partial}{\partial t} \boldsymbol{f}(t) = -\mathcal{L} \boldsymbol{f}(t)$. The sign change is due to the aforementioned difference in definiteness. The equation can be solved analytically as $\boldsymbol{f}(t) = \exp(-t\mathcal{L}) \boldsymbol{f}(0)$. Here $\exp(-t\mathcal{L})$ is known as the heat kernel. Some publications use $\mathcal{L}_{\text{rw}}$ or $\mathcal{L}_{\text{sym}}$ instead, which have similar diffusion properties [20, 19]. The heat kernel can also be used for graph neural networks; see, e.g., [58].

Differential operators of the kind $\operatorname{div}(g(x) \nabla f(x))$ can also be generalized, where g is a matrix-valued function that may describe material properties in the domain. This pointwise linear transformation of the gradient vector cannot be adopted one-to-one on graphs, but the gradient along each edge must be manipulated individually, as in $\operatorname{div}(\operatorname{diag}(\boldsymbol{g}) \nabla \boldsymbol{f})$. In neural network approaches, $\boldsymbol{g}$ may be a learned parameter or attention scores; see, e.g., [16]. This setup is in line with the recent trend of edge field representations for aggregation of features across neighborhoods.

Like many graph learning methods, these approaches show much better justification and scaling behavior for *sparse* graphs (cf. Section 2.2.3). For denser or even fully connected graphs, the number of edges may quickly become too large to work with. Though the gradient matrix ∇ is by definition highly sparse with exactly two entries per row, storing and operating with higher-dimensional edge features may become impractical.

2.3.6 Motivation via Minimum Cut problems

The PDE point of view gives a good intuition of the concept of flow through the graph. For the clustering properties of the small eigenvalues, a better intuition can

be gained from the relationship to the NCut and RatioCut problems. This section is only to give a short introduction to this angle to lay the groundwork for a better understanding of the upcoming eigenvalue computation methods. For more details especially on non-binary classification problems, we refer to [98, Section 5], the work on which this section is based.

Basic terms and notation. Consider a *connected* weighted undirected graph. A *cut* is a division of the node set $V = \{1, \dots, n\}$ into a subset $A \subset V$ and its complement $\bar{A} = V \setminus A$. For the purpose of this section, both A and $\bar{A}$ must be non-empty. Same as before, let $|A|$ denote the number of elements in A. In addition, let the *volume* of a subset, $\mathrm{vol}(A) = \sum_{i \in A} d_i$, denote the sum of node degrees. For two subsets $A, B \subset V$, we furthermore define the cross weight as $W(A, B) = \sum_{i \in A} \sum_{j \in B} a_{ij}$. The basic binary MinCut problem is to find a cut A with minimum cost $W(A, \bar{A})$.

The binary NCut problem. The Normalized Cut or *NCut* problem is a "balanced" version of the original MinCut problem which penalizes cuts with substantially different sizes $\mathrm{vol}(A) \not\approx \mathrm{vol}(\bar{A})$. This is achieved by minimizing the cost function

$$\mathrm{NCut}(A, \bar{A}) = W(A, \bar{A}) \left(\frac{1}{\mathrm{vol}(A)} + \frac{1}{\mathrm{vol}(\bar{A})} \right). \tag{2.15}$$

Its minimization is equivalent to the NP-hard optimization problem

$$\min_{A \subset V} \quad \boldsymbol{g}^T \mathcal{L} \boldsymbol{g} \quad \text{s.t.} \quad g_i = \begin{cases} \sqrt{\mathrm{vol}(\bar{A}) / \mathrm{vol}(A)} & \text{if } i \in A, \\ -\sqrt{\mathrm{vol}(A) / \mathrm{vol}(\bar{A})} & \text{if } i \notin A, \end{cases}$$

featuring the combinatorial graph Laplacian $\mathcal{L}$. The objective $\boldsymbol{g}^T \mathcal{L} \boldsymbol{g}$ can be seen to be a scaled variant of (2.15) via Proposition 2.9(c), where the terms $g_i - g_j$ are constant depending only on whether nodes i and j belong to the same side of the cut. An approximation to the solution can be obtained from the relaxation where the combinatorial constraint is replaced by two properties that were always satisfied in the original formulation,

$$\min_{\boldsymbol{g} \in \mathbb{R}^n} \quad \boldsymbol{g}^T \mathcal{L} \boldsymbol{g} \quad \text{s.t.} \quad \mathbf{1}^T \mathcal{D}^{1/2} \boldsymbol{g} = 0, \quad \boldsymbol{g}^T \mathcal{D} \boldsymbol{g} = \mathrm{vol}(V).$$

Finally, by substituting $\boldsymbol{f} = \mathcal{D}^{1/2} \boldsymbol{g}$, we obtain

$$\min_{\boldsymbol{f} \in \mathbb{R}^n} \quad \boldsymbol{f}^T \mathcal{L}_{\mathrm{sym}} \boldsymbol{f} \quad \text{s.t.} \quad \mathbf{1}^T \mathcal{D}^{1/2} \boldsymbol{f} = 0, \quad \|\boldsymbol{f}\|_2 = \mathrm{vol}(V). \tag{2.16}$$

Because $\mathcal{D}^{1/2}\mathbf{1}$ is the zero eigenvector, this formulation is equal to the characterization of λ_2 from Proposition 2.15(b). This shows that the Fiedler vector gives a solution to the relaxed binary RatioCut problem. For this reason, the signs of its entries can be used for binary clustering.

The binary RatioCut problem. Similar to NCut, the RatioCut problem is another variant of MinCut, leading to balanced cuts in the sense of $\mathrm{vol}(A) \approx \mathrm{vol}(\bar{A})$ via the objective function

$$\mathrm{RatioCut}(A, \bar{A}) = W(A, \bar{A}) \left(\frac{1}{|A|} + \frac{1}{|\bar{A}|} \right).$$

The equivalent combinatorical optimization problem is

$$\min_{A \subset V} \quad \boldsymbol{f}^T \mathcal{L} \boldsymbol{f} \quad \text{s.t.} \quad f_i = \begin{cases} \sqrt{|\bar{A}|/|A|} & \text{if } i \in A, \\ -\sqrt{|A|/|\bar{A}|} & \text{if } i \notin A, \end{cases}$$

which can be relaxed similarly to obtain

$$\min_{\boldsymbol{f} \in \mathbb{R}^n} \quad \boldsymbol{f}^T \mathcal{L} \boldsymbol{f} \quad \text{s.t.} \quad \mathbf{1}^T \boldsymbol{f} = 0, \quad \|\boldsymbol{f}\|_2 = \sqrt{n}. \tag{2.17}$$

Since the vector of ones is the eigenvector to the eigenvalue 0 of $\mathcal{L}$, this relaxation is solved by the second smallest eigenvector of the combinatorial Laplacian.

Therefore the difference between $\mathcal{L}$ and $\mathcal{L}_{\mathrm{sym}}$ in terms of cuts and classification is that the resulting clusters are balanced in different ways. The combinatorial Laplacian leads to similar numbers of nodes in each cluster, while the normalized Laplacian leads to similar total degrees.

K-ary cut problems. Clustering into $K > 2$ classes can be achieved by partitioning the vertex set into K sets and minimizing a similar NCut or RatioCut objective function. The relaxed problems are solved by the sets of the first K eigenvectors of $\mathcal{L}_{\mathrm{sym}}$ or $\mathcal{L}$, respectively, which is again related to Proposition 2.15 and to Spectral Clustering. The formal steps can again be found in [98].

Cheeger inequalities. The arguments above empirically motivate the use of the Fiedler vector for clustering. For the associated eigenvalue λ_2 of $\mathcal{L}_{\mathrm{sym}}$, on the other hand, we only get the estimate

$$\lambda_2 \leq \min_{A \subset V} \mathrm{NCut}(A, \bar{A}).$$

In a similar setting, this relationship can be studied more extensively by using Cheeger constants. First, we introduce a related cut functional

$$h(A) = \frac{W(A, \bar{A})}{\min(\mathrm{vol}(A), \mathrm{vol}(\bar{A}))} \quad \Rightarrow \quad h(A) \leq \mathrm{NCut}(A, \bar{A}) \leq 2\, h(A)$$

and based on that, the Cheeger constant

$$h_G = \min_{A \subset V} h(A).$$

There are different versions of Cheeger constants and inequalities. We here state the version given and proved in [19] for unweighted graphs:

$$1 - \sqrt{1 - h_G^2} \leq \lambda_2 \leq 2h_G.$$

In other words, as the Cheeger constant approaches 1, λ_2 must also reach 1. The lower bound is achieved by constructing a cut explicitly based on the entries of $\boldsymbol{u}_2$ [19, Corollary 2.4].

2.4 Graph Convolution

The generalization of convolution to graphs has gotten a lot of attention lately due to the success of Graph Convolutional Networks, a class of neural networks that will be the focus of Chapter 4. Here, we will introduce the theoretical background and motivation behind graph Fourier transforms, convolution, and filters, which are central concepts of signal processing on graphs [90].

2.4.1 Background on Fourier transforms and convolution

Fourier analysis is one of the fundamental tools in the arsenal of applied mathematics with numerous applications across all kinds of scientific fields, especially in signal processing and its use cases in data science, engineering, and physics, among others. The ability to decompose spatial and temporal signals into waves of different frequencies enables a multitude of data analysis concepts, such as wavelets and frequency filtering.

For illustration, consider a simple one-dimensional example from classical signal processing. Background information and proofs can be found, e.g., in [79].

Continuous signals. A 1-periodic temporal signal is a univariate function $f \in L_2(\mathbb{T})$, where $\mathbb{T} = \left[-\frac{1}{2}, \frac{1}{2}\right]$ and $L_2(\mathbb{T})$ is the space of scalar-valued Lebesgue-integrable functions with domain $\mathbb{T}$. This function space is spanned by the orthonormal basis $\{u_k\}_{k\in\mathbb{Z}}$ with $u_k(t) = \exp(2\pi \mathrm{i} kt)$. Each basis function is an eigenfunction to the Laplacian operator, as seen via

$$-\Delta u_k(t) = -\frac{\partial^2}{\partial t^2} u_k(t) = 4\pi^2 k^2 \exp(2\pi \mathrm{i} kt).$$

The Fourier transform of the signal f is given by its Fourier coefficients

$$\hat{f}(k) = \langle f, u_k \rangle = \int_{-1/2}^{1/2} f(t) \mathrm{e}^{-2\pi \mathrm{i} kt} \mathrm{d}t \quad \forall\ k \in \mathbb{Z}$$

with the inner product $\langle \cdot, \cdot \rangle$ on $L_2(\mathbb{T})$, $\langle f, g \rangle = \int_{-1/2}^{1/2} f(t)\overline{g(t)}\mathrm{d}t$. The coefficients at k and $-k$ informally describe how much f oscillates with frequency $|k|$. Because the u_k form an orthonormal basis, f can be represented in that basis as

$$f(t) = \sum_{k \in \mathbb{Z}} \hat{f}(k) \mathrm{e}^{2\pi \mathrm{i} k t}.$$

Some important signal manipulations can be implemented by operating on the Fourier coefficients. For example, high-pass or low-pass filters are realized by transforming the signal to the frequency domain, truncating its Fourier coefficients with frequency $|k|$ smaller or larger than some threshold, and then transforming back to the time domain [79].

The *convolution* of two signals f, g is defined as

$$(f * g)(t) = \int_{-1/2}^{1/2} f(s) g(t-s) \mathrm{d}s \quad \forall\, t \in \mathbb{T}, \tag{2.18}$$

where g is assumed to be evaluable outside of $\mathbb{T}$ by 1-periodic continuation. It turns out that the Fourier coefficients of the convolution are equal to the frequency-wise products of those of f and g,

$$\widehat{(f * g)}(k) = \hat{f}(k)\hat{g}(k) \quad \Rightarrow \quad (f * g)(t) = \sum_{k=1}^{\infty} \hat{f}(k)\hat{g}(k) \mathrm{e}^{2\pi \mathrm{i} k t}.$$

Discrete signals on regular grids. In applications, signals are often not known in a continuous form in each arbitrary time step, but only by samples $f_j = f(j/N)$, $j \in \mathbb{Z}$, taken on a discrete grid with width $1/N$. These samples can be given via vectors $\boldsymbol{f} \in \mathbb{R}^N$, which is assumed to be one period of the original N-periodic sequence with indices in $\mathbb{Z}$. The low-frequency Fourier coefficients of the underlying unknown continuous signal can be approximated by

$$\hat{f}_k = \frac{1}{N} \sum_{j=1}^{N} f_j \exp\left(2\pi \mathrm{i} \frac{jk}{N}\right) \quad \forall\, k \in \left\{-\frac{N}{2}, \dots, \frac{N}{2} - 1\right\},$$

which is a version of a discrete Fourier transform (cf. Section 3.1.1). The approximation $\hat{f}_k \approx \hat{f}(k)$ is subject to an aliasing error, cf. [79, Section 3.1.1]. Moreover, the coefficients can be written as inner products $\hat{f}_k = \boldsymbol{u}_k^H \boldsymbol{f}$, where the vectors $\boldsymbol{u}_k = \left(\frac{1}{N} \exp(-2\pi \mathrm{i} jk/N)\right)_{j=1}^{N}$ can easily be verified to be orthogonal eigenvectors of

the discretized Laplacian for periodic signals:

$$L\boldsymbol{u}_k = \left(2 - 2\cos\left(2\pi\frac{k}{N}\right)\right)\boldsymbol{u}_k \quad \text{with} \quad L = \begin{bmatrix} 2 & -1 & 0 & & -1 \\ -1 & 2 & -1 & \ddots & \\ 0 & -1 & \ddots & \ddots & 0 \\ & \ddots & \ddots & 2 & -1 \\ -1 & & 0 & -1 & 2 \end{bmatrix}.$$

The *cyclic convolution* of two such vectors $\boldsymbol{f}$ and $\boldsymbol{g}$ is simply defined as

$$\boldsymbol{f} * \boldsymbol{g} = \boldsymbol{h} = (h_j)_{j=1}^N \quad \text{with} \quad h_j = \sum_{k=1}^N f_k g_{j-k}, \tag{2.19}$$

where again the index of g_{k-j} is understood modulo N to yield the periodic behavior. Similar to the continuous case, we get

$$\hat{h}_k = \frac{1}{N}\hat{f}_k\hat{g}_k \quad \Rightarrow \quad h_j = \frac{1}{N}\sum_{k=1}^N \hat{f}_k\hat{g}_k \exp\left(-2\pi\mathrm{i}\frac{jk}{N}\right).$$

the right hand side equation utilizes an inversion property that we will discuss in more detail in Section 3.1.1.

2.4.2 The graph Fourier transform

Spectral graph theory, the mathematical field that analyzes graphs based on the spectral property of the Laplacian, is well-established and has enjoyed many decades of research [19]. Compared to that, the field of *signal processing* on graphs has only emerged relatively recently. The goal is to use the Laplacians and their spectral properties to analyze not the graph itself, but signals on the graph, in order to solve learning tasks and inverse problems [90].

Both the continuous and regular discrete Fourier transforms are basically basis changes from the canonical representation to an orthonormal basis of eigenfunctions of some Laplacian on that space. Each signal can then be identified as a linear combination of these eigenfunctions and the coefficients are known as Fourier coefficients at frequencies determined by the corresponding eigenvalues. The generalization to graphs follows the exact same pattern.

2.21 Definition. *(The graph Fourier transform)*
On an undirected graph with n nodes, let λ_k, $\boldsymbol{u}_k$, Λ, U denote the eigeninformation of $\mathcal{L}_{\text{sym}}$ as in Definition 2.13. Then $\hat{f}_k = \boldsymbol{f}^T\boldsymbol{u}_k$ is the Fourier coefficient of a signal $\boldsymbol{f} \in \mathbb{R}^n$ at frequency λ_k.
The vector of Fourier coefficients is given by $\hat{\boldsymbol{f}} = (\hat{f}_k)_{k=1}^n = U^T\boldsymbol{f}$. Naturally the

inverse Fourier transform is given by $(U^T)^{-1} = U$, so $\boldsymbol{f}$ can be recovered from its Fourier coefficients $\hat{\boldsymbol{f}}$ via $\boldsymbol{f} = U\hat{\boldsymbol{f}}$.

Similar to the traditional case, the coefficient at frequency 0 can be understood as the signal's mean value, other low frequencies ($0 < \lambda_k < 1$, including the Fiedler value on connected graphs) correspond to the large-scale behavior across the graph, and high frequencies ($1 < \lambda_k < 2$) correspond to rapid oscillations and noise.

The signal space, spatial domain, or graph domain are different names for the set of vectors in which the i-th entry directly refers to node i of the graph. The frequency domain or the spectral domain refers to the set of vectors where the k-th entry is associated with the frequency λ_k.

2.4.3 Convolution and spectral filters on graphs

Convolution as in (2.18) or (2.19) does not have an intuitive equivalent definition on graph domains. This is because the vertex set is not a linear space and hence shifting a signal does not have any meaning.

What can be done, however, is defining convolution as the operation that leads to element-wise multiplication in the spectral space. For two graph signals $\boldsymbol{f}, \boldsymbol{g} \in \mathbb{R}^n$ and $\boldsymbol{g} * \boldsymbol{f} = \boldsymbol{h}$, we should get

$$\hat{h}_k = \boldsymbol{u}_k^T(\boldsymbol{g} * \boldsymbol{f}) = \hat{g}_k\hat{f}_k = (\boldsymbol{u}_k^T\hat{\boldsymbol{g}})(\boldsymbol{u}_k^T\hat{\boldsymbol{f}}) \quad \forall\, k = 1, \dots, n$$

and hence

$$\boldsymbol{g} * \boldsymbol{f} = \boldsymbol{h} = U\hat{\boldsymbol{h}} = \sum_{k=1}^{n} \hat{h}_k \boldsymbol{u}_k = \sum_{k=0}^{n-1} (\boldsymbol{u}_k^T\boldsymbol{f})(\boldsymbol{u}_k^T\boldsymbol{g})\boldsymbol{u}_k.$$

2.22 Definition. *(Graph convolution)*
The convolution of two signals $\boldsymbol{f}, \boldsymbol{g} \in \mathbb{R}^n$ on an undirected graph is defined as

$$\boldsymbol{g} * \boldsymbol{f} = U\left(\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{f}}\right) = U\left((U^T\boldsymbol{g}) \odot (U^T\boldsymbol{f})\right),$$

where $\odot$ denotes element-wise multiplication. If the Fourier coefficients $\hat{g}_k$ of $\boldsymbol{g}$ are known directly, this can be written as

$$\boldsymbol{g} * \boldsymbol{f} = U\,\mathrm{diag}(\hat{g}_1, \dots, \hat{g}_n)U^T\boldsymbol{f}. \tag{2.20}$$

It is worth noting that graph convolution does not have an intuitive spatial interpretation just like traditional convolution, but the operation nevertheless achieves the same purpose.

In signal processing on regular discrete domains, one of the many use cases of convolution is the analysis of *linear time-invariant (LTI) filters*. These are operators

on the signal space with certain properties, which can be shown to be characterizable as convolution with their *impulse response*.

On graphs, this can be generalized in the following way. A linear filter operator $G : \mathbb{R}^n \to \mathbb{R}^n$ with "impulse response" $\boldsymbol{g} \in \mathbb{R}^n$ can be defined via (2.20) as

$$G(\boldsymbol{f}) = \boldsymbol{g} * \boldsymbol{f} = U \operatorname{diag}(\hat{g}_0, \dots, \hat{g}_{n-1}) U^T \boldsymbol{f},$$

so the graph generalization of LTI filters are matrices with the same eigenvectors as $\mathcal{L}_{\text{sym}}$ but arbitrary corresponding eigenvalues.

Since graph convolution is not easily interpretable in the spatial domain, the behavior of a filter can hardly be predicted just from the values of the impulse response. For that reason, instead of specifying $\boldsymbol{g}$, a filter can be given directly by the impulse response's Fourier coefficients $\hat{g}_k$.

In many cases, these coefficients can be given as $\hat{g}_k = \varphi(\lambda_k)$ via a function φ defined on the spectrum of $\mathcal{L}_{\text{sym}}$. Filters can be represented by φ if all eigenvalues have multiplicity 1 or if multiple eigenvalues have the same Fourier coefficient $\hat{g}_k$.

2.23 Definition. *(Spectral filters)*
A spectral filter on an undirected graph is given by a function

$$\varphi : \{\lambda_k \,:\, k = 1, \dots, n\} \to \mathbb{R}$$

associated with the filter matrix obtained by applying φ as a matrix function to $\mathcal{L}_{\text{sym}}$,

$$\varphi(\mathcal{L}_{\text{sym}}) = U \varphi(\Lambda) U^T \quad \text{with} \quad \varphi(\Lambda) = \operatorname{diag}\left(\varphi(\lambda_1), \dots, \varphi(\lambda_n)\right), \tag{2.21}$$

where λ_k, Λ, and U represent the eigeninformation of $\mathcal{L}_{\text{sym}}$ as in Definition 2.13. For simplicity, φ can be defined on the whole interval $[0, 2]$ that is known to contain the spectrum according to Proposition 2.11(c).

We conclude the theoretical basics of graph theory with a statement about how spectral filters affect signals in the Fourier basis.

2.24 Proposition. *(Spectral filters as eigenvector amplifiers)*
For a signal given as a Fourier series, $\boldsymbol{f} = \sum_{k=1}^{n} \hat{f}_k \boldsymbol{u}_k$, the Fourier coefficient $\hat{f}_k$ specifies the component of $\boldsymbol{f}$ in direction of $\boldsymbol{u}_k$.
The image of $\boldsymbol{f}$ under the spectral filter φ has the Fourier coefficients of $\boldsymbol{f}$, but multiplied by the factors $\varphi(\lambda_k)$, as in

$$\varphi(\mathcal{L}_{\text{sym}}) \boldsymbol{f} = \sum_{k=1}^{n} \varphi(\lambda_k) \hat{f}_k \boldsymbol{u}_k.$$

Hence eigenvectors with relatively large $\varphi(\lambda_k)$ are *reinforced* by the filter, while eigenvectors with relatively small $\varphi(\lambda_k)$ are *damped*.

3 Numerical Computation Schemes for Dense Graph Laplacians

When complex mathematical expressions need to be evaluated in practice, there is often a great performance range between naive computations and those that fully exploit the mathematical context. Numerical mathematics deals with creating algorithms for efficient and accurate results.

The two main resources in a computation are the time needed to execute it and the amount of memory needed to store all intermediate and resulting quantities. Both these requirements generally increase with the problem size in some way, and the main goal of algorithm design is typically to reduce the asymptotic growth rate. As computational resources are always limited, this directly determines the maximum problem size for which the method is still applicable before it runs out of memory or the runtime is practically infeasible. Even for problems that can be solved with existing methods on a high-performance system, making the same results obtainable in a more accessible environment is also worthwhile in its own right.

In this chapter, we will discuss multiple types of numerical methods that are relevant in the context of graphs with dense graph Laplacian matrices.

3.1 Approximating kernel-based matrix-vector products

As described in Section 2.2.7, fully connected graphs are closely related to kernel methods in analytical fields and pattern analysis [89, 50, 12, 88]. These kernels have proven to be powerful tools over decades of research.

A few of the most well-known machine learning methods, like Spectral Clustering and Support Vector Machines, rely on kernel matrices. However, in large-scale regimes, the usage of the full $n \times n$ matrices is often hindered by their storage and computational requirements. Quadratic growth is simply unacceptable in many settings. For that reason, when graph learning methods are applied to large-scale non-graph data, the graph is almost always created in a sparse way by k-nearest neighbor or radius graphs, cf. Section 2.2.5. Examples include point clouds [76] as well as high-dimensional tabular data mining and manifold learning, see [25] and the references therein.

However, quadratic behavior can be avoided even for fully connected graphs by approximating matrix-vector products analytically without having to set up the kernel matrix explicitly. The mathematical tools for such an approximation have been

known for a long time, known simply as *fast summation* in general or *fast Gauss transform* for the Gaussian kernel in particular. Still, the technical and implementational challenges of these approximations as well as the availability of sparse neighborhood graphs as a simple alternative have inhibited the exploration of these approximation methods in large-scale machine learning.

Here we will focus on the case of two sets of points in d-dimensional Euclidean space, $\mathcal{X} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_{n_x}\}$ and $\mathcal{Y} = \{\boldsymbol{y}_1, \dots, \boldsymbol{y}_{n_y}\} \subset \mathbb{R}^d$. We assume that the spatial dimension d is small, $d \leq 3$, which is mostly the case if the data points stem from geometric descriptions of real-life objects. The goal is to evaluate matrix-vector products with the *Gram matrix* or *kernel matrix*

$$\mathcal{K}_{\mathcal{Y}\mathcal{X}} = \big(K(\boldsymbol{x}_i - \boldsymbol{y}_j)\big)_{j,i=1}^{n_y,n_x} \in \mathbb{R}^{n_y \times n_x} \tag{3.1}$$

for a given rotationally invariant kernel $K : \mathbb{R}^d \to \mathbb{R}$. This kernel definition differs slightly from the one given in Section 2.2.6. We will further abuse notation and sometimes switch between the notations

$$K(\boldsymbol{x}, \boldsymbol{y}) \cong K(\boldsymbol{x} - \boldsymbol{y}) \cong K(\|\boldsymbol{x} - \boldsymbol{y}\|_2), \tag{3.2}$$

so the argument to K may either be a pair of points in $\mathbb{R}^d$, a difference vector in $\mathbb{R}^d$, or a scalar distance, depending on the setting. For example, this extends the notation of the Gaussian kernel from Example 2.7 to

$$K(\boldsymbol{z}) = \exp\left(-\frac{\|\boldsymbol{z}\|_2^2}{\sigma^2}\right). \tag{3.3}$$

We further assume that the kernel is not well-localized, i.e., it does not decay so fast that $K(\boldsymbol{x} - \boldsymbol{y})$ is negligible outside of a tight neighborhood of $\boldsymbol{x}$, which would make $\mathcal{K}_{\mathcal{Y}\mathcal{X}}$ numerically sparse or at least approximable by a sparse matrix. In our setting, we expect all entries of $\mathcal{K}_{\mathcal{Y}\mathcal{X}}$ to be meaningfully large and encoding similarity even over longer distances. A more detailed comparison to the well-localized setting can be found in Section 3.1.4.

Coming back to fully connected graphs with node feature set $\mathcal{X}$ and loop weight w_{loop}, their adjacency matrix as in (2.2) can be written as

$$\mathcal{A} = \mathcal{K}_{\mathcal{X}\mathcal{X}} + \kappa_0 I \quad \text{with} \quad \kappa_0 = w_{\text{loop}} - K(0). \tag{3.4}$$

Hence fast approximations of (3.1) can also be used for fast application of fully connected adjacency matrices.

The following subsections will give a step-by-step derivation to the tools required for the fast summation of kernel values based on the Non-equispaced Fast Fourier Transform (NFFT).

3.1.1 The Fast Fourier Transform

The first tool that we are going to need is the Fast Fourier Transform, which means the fast computation of the Discrete Fourier Transform. This section will only introduce the basic concept to lay the ground for the following sections. For a more detailed explanation, see, e.g., [79]. We start with a definition of the quantity of interest.

3.1 Definition. *(Discrete Fourier Transform)*
For an even bandwidth $N \in 2\mathbb{N}$, define the one-dimensional equispaced set of frequencies as

$$\mathcal{I}_N = \left\{ -\frac{N}{2}, -\frac{N}{2}+1, \ldots, -1, 0, 1, \ldots, \frac{N}{2}-1 \right\}. \tag{3.5}$$

A d-dimensional frequency multi-index is a vector $\boldsymbol{k} \in \mathcal{I}_N^d$ with entries in $\mathcal{I}_N$.
(a) The d-dimensional forward Discrete Fourier Transform (Forward DFT) of coefficients coefficients $\hat{g}_{\boldsymbol{k}} \in \mathbb{C}$ ($\boldsymbol{k} \in \mathcal{I}_N^d$) is given by

$$g_{\boldsymbol{l}} = \sum_{\boldsymbol{k} \in \mathcal{I}_N^d} \hat{g}_{\boldsymbol{k}} \exp\left(-2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{N} \right) \quad \forall\, \boldsymbol{l} \in \mathcal{I}_N^d. \tag{3.6}$$

(b) The d-dimensional adjoint Discrete Fourier Transform (Adjoint DFT) of coefficients $\hat{\hat{g}}_{\boldsymbol{l}} \in \mathbb{C}$ ($\boldsymbol{l} \in \mathcal{I}_N^d$) is given by

$$\hat{g}_{\boldsymbol{k}} = \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} \hat{\hat{g}}_{\boldsymbol{l}} \exp\left(2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{N} \right) \quad \forall\, \boldsymbol{l} \in \mathcal{I}_N^d. \tag{3.7}$$

(c) Assume all values $g_{\boldsymbol{l}}$, $\hat{g}_{\boldsymbol{k}}$, and $\hat{\hat{g}}_{\boldsymbol{l}}$ from (3.6) and (3.7) are stored in vectors $\boldsymbol{g}, \hat{\boldsymbol{g}}, \hat{\hat{\boldsymbol{g}}} \in \mathbb{C}^{N^d}$. Then the forward and adjoint DFT can be written as

$$\boldsymbol{g} = \mathcal{F} \hat{\boldsymbol{g}} \quad \text{and} \quad \hat{\boldsymbol{g}} = \mathcal{F}^H \hat{\hat{\boldsymbol{g}}}$$

with the Fourier matrix

$$\mathcal{F} \in \mathbb{C}^{N^d \times N^d}, \quad \mathcal{F}_{\boldsymbol{k},\boldsymbol{l}} = \exp\left(-2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{N} \right). \tag{3.8}$$

Note that this definition of the DFT has one simplification over the general setting, as the bandwidth N is the same for each dimension. In general, it is possible to choose a vector of bandwidths $\boldsymbol{N} \in \mathbb{N}^d$, which leads to the index set $\mathcal{I}_{\boldsymbol{N}} = \times_{t=1}^{d} \mathcal{I}_{N_t}$ replacing $\mathcal{I}_N^d$, and the term $\boldsymbol{k}^T \operatorname{diag}(\boldsymbol{N})^{-1} \boldsymbol{l}$ replacing $\frac{\boldsymbol{k}^T \boldsymbol{l}}{N}$ in the exponent. However, for the purpose of this work, we are only going to need the case with uniform bandwidth.

DFTs are the discrete variant of continuous Fourier transforms and appear in many scientific fields. An important property of DFTs that we are going to need is the fact that the forward and adjoint DFT are the inverse operations of one another, up to a constant scaling factor. This fundamental property is formalized in the following proposition. For a proof, see, e.g., [79, Section 3.2].

3.2 Proposition. *(Inversion property of the DFT)*
The Fourier matrix (3.8) satisfies

$$\mathcal{F}^H \mathcal{F} = \mathcal{F}\mathcal{F}^H = N^d I.$$

With the notation of Definition 3.1, this means that (3.6) is equivalent to

$$\hat{g}_{\boldsymbol{k}} = \frac{1}{N^d} \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} g_{\boldsymbol{l}} \exp\left(2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{N}\right) \quad \forall\, \boldsymbol{l} \in \mathcal{I}_N^d$$

and (3.7) is equivalent to

$$g_{\boldsymbol{l}} = \frac{1}{N^d} \sum_{\boldsymbol{k} \in \mathcal{I}_N^d} \hat{g}_{\boldsymbol{k}} \exp\left(-2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{N}\right) \quad \forall\, \boldsymbol{l} \in \mathcal{I}_N^d. \tag{3.9}$$

Hence the inverse forward DFT is a scaled adjoint DFT and vice versa.

Naively, the computational cost of the DFT should be in $\mathcal{O}(N^{2d})$. Due to the special structure of the Fourier matrix, it is possible to compute the exact result with considerably less operations. The different algorithms are known as Fast Fourier Transforms (FFT). The roots of this methods go back to Gauss and the first modern algorithm of this kind, the Cooley-Tukey algorithm, has been known since the 1960s [22]. We will not go into detail here, but the main idea for $d = 1$ is to split the sum in (3.6) into even and odd parts, both of which can again be computed by an FFT, yielding logarithmic behavior.

3.3 Proposition. *(Fast Fourier Transform)*
Both the forward and adjoint d-dimensional DFTs with uniform bandwidth N can be evaluated in $\mathcal{O}(N^d d \log N)$ time complexity using forward and adjoint Fast Fourier Transform (FFT) algorithms, respectively.

3.1.2 The Non-equispaced Fast Fourier Transform

Practical interest in the fast computation of Fourier series with *non-equispaced* points is similarly high as for the DFT [79]. Various methods had been proposed before the scheme popular today was introduced for Fourier analysis in [28], though it is closely

related to *gridding* algorithms known in image processing before that (see [79, 82] and the references therein).

In this context, non-equispaced points mean that the terms $\frac{1}{N}\boldsymbol{k} \in \left[-\frac{1}{2}, \frac{1}{2}\right]^d$ in (3.7) get replaced by arbitrary points $\boldsymbol{x}_j$ from the same cube, which is known as a *torus* in the Fourier community. The following definition introduces the desired quantity. Again, we use the simplified setting where the bandwidth N is the same in each dimension.

3.4 Definition. *(Non-equispaced Discrete Fourier Transform)*
For a spatial dimension $d \in \mathbb{N}$, define the d-dimensional torus as

$$\mathbb{T}^d = \left\{ \boldsymbol{x} = \left(x_t\right)_{t=1}^d \in \mathbb{R}^d : \ -\frac{1}{2} \leq x_t \leq \frac{1}{2} \right\} = \left[-\frac{1}{2}, \frac{1}{2}\right]^d \tag{3.10}$$

Let a set of n points $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \mathbb{T}^d$ and a fixed bandwidth N be given.
(a) The forward Non-equispaced Discrete Fourier Transform (Forward NDFT) is the computation of

$$f_j = \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} \hat{f}_{\boldsymbol{l}} \exp\left(-2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{x}_j\right) \in \mathbb{C} \quad \forall\ j = 1, \ldots, n \tag{3.11}$$

for coefficients $\hat{f}_{\boldsymbol{l}} \in \mathbb{C}$ that are given for each frequency $\boldsymbol{l} \in \mathcal{I}_N^d$.

(b) The adjoint NDFT is the computation of

$$\hat{f}_{\boldsymbol{l}} = \sum_{j=1}^{n} \hat{\hat{f}}_j \exp\left(2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{x}_j\right) \in \mathbb{C} \quad \forall\ \boldsymbol{l} \in \mathcal{I}_N^d \tag{3.12}$$

for coefficients $\hat{\hat{f}}_j \in \mathbb{C}$ that are given for each point index $j = 1, \ldots, n$.

(c) Assume all coefficients f_j, $\hat{f}_{\boldsymbol{l}}$, and $\hat{\hat{f}}_j$ from (3.11) and (3.12) are stored in vectors $\boldsymbol{f}, \hat{\hat{\boldsymbol{f}}} \in \mathbb{R}^n$ and $\hat{\boldsymbol{f}} \in \mathbb{R}^{N^d}$. Then the forward and adjoint NDFT can be written as

$$\boldsymbol{f} = \hat{\mathcal{F}} \hat{\boldsymbol{f}} \quad \text{and} \quad \hat{\boldsymbol{f}} = \hat{\mathcal{F}}^H \hat{\hat{\boldsymbol{f}}}$$

with the non-equispaced Fourier matrix

$$\hat{\mathcal{F}} \in \mathbb{C}^{n \times N^d}, \quad \hat{\mathcal{F}}_{j,\boldsymbol{l}} = \exp\left(2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{x}_j\right). \tag{3.13}$$

Note how the NDFT contains the DFT from Definition 3.1 as a special case by setting the equispaced grid points

$$\mathcal{X} = \left\{ \frac{1}{N}\boldsymbol{k} : \boldsymbol{k} \in \mathcal{I}_N^d \right\} \subset \mathbb{T}^d \quad \text{with} \quad n = |\mathcal{X}| = N^d.$$

However, the NDFT generally does *not* satisfy a similar property to the inversion property of the equispaced DFT, cf. Proposition 3.2, and its *exact* computation can *not* be sped up in a similar way as with the FFT, cf. Proposition 3.3. Hence, while the FFT provides the fast *computation* of the DFT, the NFFT will only provide the fast *approximation* of the NDFT.

The ansatz of the Non-equispaced Fast Fourier Transform (NFFT). The goal of the NFFT is to compute an efficient approximation to the forward NDFT (3.12). This task comes down to approximating the trigonometric polynomial

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} \hat{f}_{\boldsymbol{l}} \exp\left(-2\pi \mathrm{i} \boldsymbol{k}^T \boldsymbol{x}\right) \tag{3.14}$$

in n given points $\boldsymbol{x}_j \in \mathbb{T}^d$. For the approximation, we use the ansatz

$$s(\boldsymbol{x}) = \sum_{\boldsymbol{l} \in \mathcal{I}_M^d} g_{\boldsymbol{l}} \, \tilde{\varphi}\left(\boldsymbol{x} - \frac{1}{M}\boldsymbol{l}\right) \tag{3.15}$$

where $\tilde{\varphi} : \mathbb{R}^d \to \mathbb{R}$ is a 1-periodic window function and $M > N$ is an even oversampled bandwidth parameter.

The window function. Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a basic window function that is even, $\varphi(x) = \varphi(-x)$, and well-localized and quickly decaying, $\varphi(x) \ll 1$ for $|x| \geq \frac{1}{2}$. We construct a multi-dimensional variant of this function that is 1-periodic in each dimension via

$$\tilde{\varphi}(\boldsymbol{x}) = \sum_{\boldsymbol{z} \in \mathbb{Z}^d} \prod_{t=1}^{d} \varphi(x_t + z_t), \tag{3.16}$$

using the convention that x_t and z_t denote the components of $\boldsymbol{x}$ and $\boldsymbol{z}$. We assume that $\tilde{\varphi}$ can be expressed via the uniformly convergent Fourier series

$$\tilde{\varphi}(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in \mathbb{Z}^d} \prod_{t=1}^{d} \hat{\varphi}_{k_t} \exp\left(-2\pi \mathrm{i} k_t x_t\right) = \sum_{\boldsymbol{k} \in \mathbb{Z}^d} \hat{\varphi}_{\boldsymbol{k}} \exp\left(-2\pi \mathrm{i} \boldsymbol{k}^T \boldsymbol{x}\right), \tag{3.17}$$

where $\hat{\varphi}_k \in \mathbb{C}$ are the Fourier coefficients of the original function φ at frequency $k \in \mathbb{Z}$, and $\hat{\varphi}_{\boldsymbol{k}} = \prod_{t=1}^{d} \hat{\varphi}_{k_t}$. An appropriate choice of φ requires that $\hat{\varphi}_k$ can be evaluated efficiently, or at least approximated accurately.

Truncation in the frequency domain. Under an appropriate choice of φ and M, the Fourier coefficients $\hat{\varphi}_{\boldsymbol{k}}$ are decaying quickly and can be assumed to be negligible for $\boldsymbol{k} \notin \mathcal{I}_M^d$. Truncating the sum in (3.17) accordingly and plugging it into (3.15), we obtain the approximation

$$\begin{aligned} s(\boldsymbol{x}) &\approx \sum_{\boldsymbol{l} \in \mathcal{I}_M^d} g_{\boldsymbol{l}} \sum_{\boldsymbol{k} \in I_M^d} \hat{\varphi}_{\boldsymbol{k}} \exp\left(-2\pi \mathrm{i} \boldsymbol{k}^T \left(\boldsymbol{x} - \frac{1}{M}\boldsymbol{l}\right)\right) \\ &= \sum_{\boldsymbol{k} \in I_M^d} \hat{g}_{\boldsymbol{k}} \exp\left(-2\pi \mathrm{i} \boldsymbol{k}^T \boldsymbol{x}\right) \quad \text{with} \quad \hat{g}_{\boldsymbol{k}} = \sum_{\boldsymbol{l} \in \mathcal{I}_M^d} g_{\boldsymbol{l}} \exp\left(2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{M}\right). \end{aligned} \tag{3.18}$$

Note that the $\hat{g}_{\boldsymbol{k}}$ are the result of an adjoint DFT with bandwidth M, cf. Definition 3.1(b).

Now, by comparing (3.18) with (3.14) and noticing that $\mathcal{I}_N$ is a subset of $\mathcal{I}_M$, we see that the choice

$$\hat{g}_{\boldsymbol{k}} = \begin{cases} \hat{f}_{\boldsymbol{k}} / \hat{\varphi}_{\boldsymbol{k}} & \text{if } \boldsymbol{k} \in \mathcal{I}_N, \\ 0 & \text{if } \boldsymbol{k} \in \mathcal{I}_M \setminus \mathcal{I}_N \end{cases} \tag{3.19}$$

leads to a good approximation $s(\boldsymbol{x}) \approx f(\boldsymbol{x})$ that is subject only to an aliasing error. In order to use the form of (3.15), we need to compute the coefficients $g_{\boldsymbol{l}}$ from (3.19) utilizing the inversion property for the DFT, cf. Proposition 3.2. Hence the coefficients can be computed as

$$g_{\boldsymbol{l}} = \frac{1}{M^d} \sum_{\boldsymbol{k} \in \mathcal{I}_M^d} \hat{g}_{\boldsymbol{k}} \exp\left(-2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{M}\right), \tag{3.20}$$

via an adjoint FFT, cf. Proposition 3.3.

Truncation in the spatial domain. If the window function φ decays quickly, i.e., it is well localized in space, its values outside of the interval $[-\frac{m}{M}, \frac{m}{M}]$ can be neglected and replaced by zero. The integer parameter $m \in \mathbb{N}, m < \frac{M}{2}$ controls the window cutoff; smaller values mean less accuracy. The resulting truncated window function is

$$\varphi_m(x) = \varphi(x)\, \chi_{[-\frac{m}{M}, \frac{m}{M}]}(x)$$

with the characteristic function χ_I of an interval I. The periodic multi-dimensional version of φ_m is given by

$$\tilde{\varphi}_m(\boldsymbol{x}) = \sum_{\boldsymbol{z} \in \mathbb{Z}^d} \prod_{t=1}^{d} \varphi_m(x_t + z_t) = \prod_{t=1}^{d} \varphi_m(x_t - \mathrm{round}(x_t)),$$

where $\mathrm{round}(x)$ denotes the closest integer to a real number x. This term gets introduced because $\varphi_m(x_t + z_t)$ can only be nonzero if $|x_t + z_t| \leq \frac{m}{M} < \frac{1}{2}$, so the

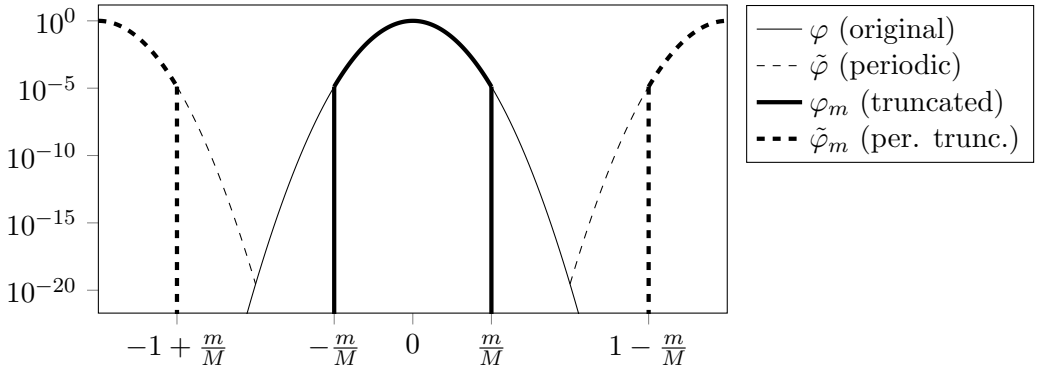


Figure 3.1: Representation of the window function variants for $d = 1$ on the interval $[-1, 1]$, using the example of the Gaussian window function.

only potentially non-vanishing summand is the one where $\boldsymbol{z} \in \mathbb{Z}^d$ has all entries $z_t = -\operatorname{round}(x_t)$. A visualization of the four window function variants for $d = 1$ is given in Figure 3.1.

We now approximate $s(\boldsymbol{x})$ by $s_m(\boldsymbol{x})$ where $\tilde{\varphi}_m$ replaces $\tilde{\varphi}$ in (3.15). There all arguments $\boldsymbol{x} - \frac{1}{M}\boldsymbol{l}$ have entries in $[-1, 1]$, so the approximation comes down to

$$s_m(\boldsymbol{x}) = \sum_{\boldsymbol{l} \in \mathcal{I}_M^d} g_{\boldsymbol{l}} \prod_{t=1}^{d} \begin{cases} \varphi(x_t - \frac{l_t}{M} + 1) & \text{if } x_t - \frac{l_t}{M} \in [-1, -1 + \frac{m}{M}], \\ \varphi(x_t - \frac{l_t}{M}) & \text{if } x_t - \frac{l_t}{M} \in [-\frac{m}{M}, \frac{m}{M}], \\ \varphi(x_t - \frac{l_t}{M} - 1) & \text{if } x_t - \frac{l_t}{M} \in [1 - \frac{m}{M}, 1], \\ 0 & \text{else.} \end{cases}$$

The multi-index set of those $\boldsymbol{l} \in \mathcal{I}_M^d$ for which $\tilde{\varphi}_m(\boldsymbol{x} - \frac{1}{M}\boldsymbol{l})$ does not vanish can be computed as

$$\begin{aligned} \mathcal{I}_{M,m}(\boldsymbol{x}) = \Big\{ \boldsymbol{l} \in \mathbb{Z}^d \mid \forall t = 1, \ldots, d : \; & -\frac{M}{2} \le l_t \le Mx_t - M + m \\ & \vee \; \max\big(Mx_t - m, -\frac{M}{2}\big) \le l_t \le \min\big(Mx_t + m, \frac{M}{2} - 1\big) \\ & \vee \; Mx_t + M - m \le l_t \le \frac{M}{2} - 1 \Big\} \\ = \bigtimes_{t=1}^{d} \Phi_M \left(\{ \lceil Mx_t \rceil - m, \; \ldots, \; \lfloor Mx_t \rfloor + m \} \right) & \end{aligned} \tag{3.21}$$

where $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ denote the floor and ceiling operators, and Φ_M wraps each element to $\{-M/2, \; \ldots, \; M/2 - 1\}$ by adding or subtracting M. Note that $\mathcal{I}_{M,m}$ has at most

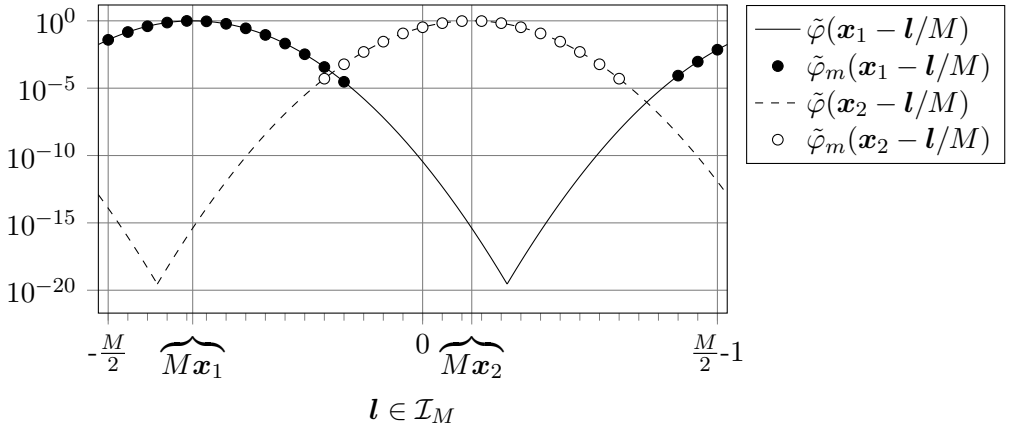


Figure 3.2: Example of the shifted window function values $\varphi(\boldsymbol{x}_j - \boldsymbol{l}/M)$ for $M = 32$, $m = 8$, and two points in dimension $d = 1$, using the Gaussian window function as in Figure 3.1. The marks for $\tilde{\varphi}_m$ show the indices $\boldsymbol{l} \in \mathcal{I}_{M,m}(\boldsymbol{x}_j)$ and their corresponding function values on the right hand side of (3.22).

$(2m+1)^d$ elements. This yields our final approximation of (3.14),

$$f(\boldsymbol{x}) \approx s_m(\boldsymbol{x}) = \sum_{\boldsymbol{l} \in \mathcal{I}_{M,m}(\boldsymbol{x})} g_{\boldsymbol{l}} \prod_{t=1}^{d} \varphi\Big(x_t - \frac{l_t}{M} - \operatorname{round}\big(x_t - \frac{l_t}{M}\big)\Big). \tag{3.22}$$

Full algorithm. All steps of the above process are summarized in Algorithm 3.1. Its complexity and linearity are stated in the next proposition.

Algorithm 3.1 The forward Non-equispaced Fast Fourier Transform (NFFT)

Input: Data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{T}^d$, bandwidth $N \in 2\mathbb{N}$, input coefficients $\hat{f}_{\boldsymbol{l}}\ \forall \boldsymbol{l} \in \mathcal{I}_N^d$, oversampled bandwidth $M \in 2\mathbb{N}$ $(M > N)$, window function φ, cutoff parameter $m \in \mathbb{N}$ $(m < M/2)$

Output: Approximations to f_j as in (3.11) for all $j = 1, \ldots, n$

1: Compute $\hat{g}_{\boldsymbol{k}}$ as in (3.19) for all $\boldsymbol{k} \in \mathcal{I}_M^d$.
2: Compute $g_{\boldsymbol{l}}$ as in (3.20) for all $\boldsymbol{l} \in \mathcal{I}_M^d$ via the adjoint FFT.
3: **for** $j = 1, \ldots, n$ **do**
4: Set up $\mathcal{I}_{M,m}(\boldsymbol{x}_j)$ as in (3.21).
5: Compute the approximation $s_m(\boldsymbol{x}_j) \approx f_j$ as in (3.22).
6: **end for**

3.5 Proposition. *(Properties of the NFFT)*
(a) For fixed spatial dimension d, the forward NDFT (3.11) can be approximated in $\mathcal{O}(M^d \log(M) + m^d n)$ time complexity via Algorithm 3.1. The operation is linear w.r.t. the input $\hat{\boldsymbol{f}}$. In the matrix notation of (3.13), this approximation can be written as

$$\boldsymbol{f} = \hat{\mathcal{F}}\hat{\boldsymbol{f}} \approx \frac{1}{M^d} B \mathcal{F}^H D \hat{\boldsymbol{f}} \tag{3.23}$$

with the equispaced Fourier matrix $\mathcal{F} \in \mathbb{C}^{M^d \times M^d}$ as in (3.8), the rectangular Kronecker product structured matrix

$$D = \bigotimes_{t=1}^{d} \begin{bmatrix} \mathbf{0} \\ \operatorname{diag}\left(\left(\frac{1}{\tilde{\varphi}_{k_t}} \right)_{k_t \in \mathcal{I}_N} \right) \\ \mathbf{0} \end{bmatrix} \in \mathbb{C}^{M^d \times N^d} \quad \text{with} \quad \mathbf{0} = (0) \in \mathbb{R}^{(M-N)/2 \times N},$$

and the truncated window function matrix

$$B = (B_{j,\boldsymbol{l}}) \in \mathbb{R}^{n \times M^d} \quad \text{with} \quad B_{j,\boldsymbol{l}} = \tilde{\varphi}_m \left(\boldsymbol{x}_j - \frac{1}{M} \boldsymbol{l} \right) \quad \forall\, j = 1, \ldots, n,\ \boldsymbol{l} \in \mathcal{I}_M^d.$$

(b) Similarly, the adjoint NDFT (3.12) can be approximated in $\mathcal{O}(M^d \log(M) + m^d n)$ time complexity as

$$\hat{\boldsymbol{f}} = \hat{F}^H \hat{\boldsymbol{f}} \approx \frac{1}{M^d} D^H \mathcal{F} B^T \hat{\boldsymbol{f}}. \tag{3.24}$$

(c) The error of the NFFT can be estimated via

$$\|(\hat{\mathcal{F}} - \frac{1}{M^d} B \mathcal{F}^H D)\hat{\boldsymbol{f}}\|_\infty \leq C \|\hat{\boldsymbol{f}}\|_1, \quad \|(\hat{\mathcal{F}}^H - \frac{1}{M^d} D^H \mathcal{F} B^T)\boldsymbol{f}\|_\infty \leq C \|\boldsymbol{f}\|_1$$

with a constant $C = C(\varphi, \frac{M}{N}, m)$ that is independent of N, n, $\hat{\boldsymbol{f}}$, and $\boldsymbol{f}$ and instead depends only on the window function φ, the oversampling rate M/N, and the cutoff parameter m. For fixed φ and M/N, C decays exponentially for $m \to \infty$.

Proof. The matrix formulation of the forward NFFT follows directly from its derivation: $D\hat{\boldsymbol{f}}$ equals $\hat{\boldsymbol{g}}$ from (3.19), $\frac{1}{M^d}\mathcal{F}^H \hat{\boldsymbol{g}}$ equals $\boldsymbol{g}$ from (3.20), and $B\boldsymbol{g}$ finally equals the approximation $(s_m(\boldsymbol{x}_j))_{j=1}^n$ from (3.22).

The complexity of the algorithm can similarly be derived from the cost of multiplication with these matrices: D is sparse with one entry per column, leading to a multiplication cost of $\mathcal{O}(M^d)$; multiplication with $\hat{\mathcal{F}}^H$ can be evaluated in $\mathcal{O}(M^d \log(M))$ via the adjoint FFT; and B is sparse with a maximum total of $(2m+1)^d$ entries in each of its n rows, leading to a multiplication cost in $\mathcal{O}(m^d n)$. The sum of these costs is in $\mathcal{O}(M^d \log(M) + m^d n)$.

Finally, the error estimate is a standard result found in, e.g., [79, 53]. □

3.6 Remark. *(Implementation details)*
(a) The oversampled bandwidth M should be at least $2N$ to reduce the aliasing error by the frequency truncation. For maximum efficiency of the FFT, M should also be a power of two. In practice, we can always choose M as the smallest power of two that is at least $2N$. Because this approach always leads to $M < 4N$, the complexity of the NFFT can be written as $\mathcal{O}(N^d \log(N) + m^d n)$.

(b) For the window function φ, modern implementations use the Kaiser-Bessel function [53] or the recent "exponential of semicircle" window function [7].

(c) In practice, a window cutoff parameter value of $m = 8$ is often enough to reach double precision accuracy with the Kaiser-Bessel window [53, Section 5.2].

3.1.3 NFFT-based fast summation

Now we have the tools to introduce the final step for fast approximate products with the kernel matrix from (3.1) based on a kernel with notation as in (3.2). The technique we describe here is the NFFT-based *fast summation* scheme proposed in [80, 81]

3.7 Definition. *(Fast summation)*
Let a source point set $\mathcal{X} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_{n_x}\} \subset \mathbb{R}^d$ and target point set $\mathcal{Y} = \{\boldsymbol{y}_1, \dots, \boldsymbol{y}_{n_y}\} \subset \mathbb{R}^d$ be given. For a smooth, rotationally invariant kernel $K : \mathbb{R}^d \to \mathbb{C}$ and a source coefficient vector $\boldsymbol{\alpha} \in \mathbb{C}^{n_x}$ the goal of *fast summation* is the fast approximation of the vector

$$\boldsymbol{f} = \mathcal{K}_{\mathcal{Y}\mathcal{X}}\, \boldsymbol{\alpha} \in \mathbb{C}^{n_y} \quad \text{with entries} \quad f_j = \sum_{i=1}^{n_x} \alpha_i K(\boldsymbol{x}_i - \boldsymbol{y}_j) \tag{3.25}$$

for the kernel matrix $\mathcal{K}_{\mathcal{Y}\mathcal{X}}$ as in (3.1).

We here expect a kernel K that is sufficiently smooth on the whole $\mathbb{R}^d$. For kernels with a singularity at 0, we refer to [81].

In order to approach fast summation based on the NFFT, let us first assume that all $\boldsymbol{x}_i, \boldsymbol{y}_j$ have Euclidean norm $\|\boldsymbol{x}_i\|_2, \|\boldsymbol{y}_j\|_2 \leq \frac{1}{4} - \frac{\varepsilon_B}{2}$, where $0 \leq \varepsilon < \frac{1}{2}$ is a small parameter. Under this assumption, K is only evaluated in points $\boldsymbol{z} = \boldsymbol{x}_i - \boldsymbol{y}_j$ with norm $\|\boldsymbol{z}\|_2 \leq \frac{1}{2} - \varepsilon_B$.

Trigonometric approximation of K. The first step is to approximate K by a trigonometric polynomial,

$$K(\boldsymbol{z}) \approx K_{RF}(\boldsymbol{z}) = \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} \hat{b}_{\boldsymbol{l}} \exp\left(2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{z}\right) \tag{3.26}$$

with $\mathcal{I}_N$ as in (3.5). The approach of [81] is to choose K_{RF} in such a way that it interpolates a yet to be determined regularized kernel $K_R \approx K$ in the grid points $\boldsymbol{z} \in \frac{1}{N}\mathcal{I}_N^d$. This leads to

$$K_{RF}\left(\frac{1}{N}\boldsymbol{k}\right) = \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} \hat{b}_{\boldsymbol{l}} \exp\left(2\pi \mathrm{i} \frac{\boldsymbol{l}^T \boldsymbol{k}}{N}\right) = K_R\left(\frac{1}{N}\boldsymbol{k}\right) \qquad \forall\, \boldsymbol{k} \in \mathcal{I}_N^d,$$

an adjoint DFT. By the inversion property stated in Proposition 3.2, the coefficents $\hat{b}_{\boldsymbol{l}}$ are hence given by

$$\hat{b}_{\boldsymbol{l}} = \frac{1}{N^d} \sum_{\boldsymbol{k} \in \mathcal{I}_N^d} K_R\left(\frac{1}{N}\boldsymbol{k}\right) \exp\left(-2\pi \mathrm{i} \frac{\boldsymbol{k}^T \boldsymbol{l}}{N}\right) \qquad \forall\, \boldsymbol{l} \in \mathcal{I}_N^d \tag{3.27}$$

and can be computed from the function values $K_R\left(\frac{1}{N}\boldsymbol{k}\right)$ in $\mathcal{O}(N^d \log(N))$ time using a forward FFT.

Because K_{RF} by (3.26) is always infinitely smooth and 1-periodic, the error of the interpolation $\|K_{RF} - K_R\|_\infty$ is smaller if K_R is sufficiently smooth and 1-periodic as well. Furthermore, $K_R(\boldsymbol{z})$ should be equal to $K(\boldsymbol{z})$ on each possible point $\boldsymbol{z} = \boldsymbol{x}_i - \boldsymbol{y}_j$ with norm $\|\boldsymbol{z}\|_2 \leq \frac{1}{2} - \varepsilon_B$ in order to avoid an additional approximation error. Finally, the setup and evaluation of K_R should be inexpensive, since N^d of its function values are required for (3.27). The solution of [81] is to define K_R piecewise as

$$K_R(\boldsymbol{z}) = \begin{cases} K(\boldsymbol{z}) & \text{if } \|y\|_2 \leq \frac{1}{2} - \varepsilon_B, \\ T_B(\|\boldsymbol{z}\|_2) & \text{if } \frac{1}{2} - \varepsilon_B \leq \|\boldsymbol{z}\|_2 \leq \frac{1}{2}, \\ T_B(\frac{1}{2}) & \text{else} \end{cases} \quad \forall\, \boldsymbol{z} \in \mathbb{T}^d = [-\frac{1}{2}, \frac{1}{2}]^d \tag{3.28}$$

based on a regularizing function $T_B : \left[\frac{1}{2} - \varepsilon_B,\, \frac{1}{2}\right] \to \mathbb{R}$. Outside of $\mathbb{T}^d$, K_R is continued 1-periodically.

The parameter $0 \leq \varepsilon_B < \frac{1}{2}$ determines the width of the regularization area. If $\varepsilon_B = 0$, we choose the only value of T_B as $T_B(\frac{1}{2}) = K(\frac{1}{2})$ to at least secure C^0-continuity of K_R. To gain C^{p-1} smoothness over the whole $\mathbb{R}^d$, i.e., if K_R is supposed to be $p-1$ times continuously differentiable everywhere, one needs a regularization

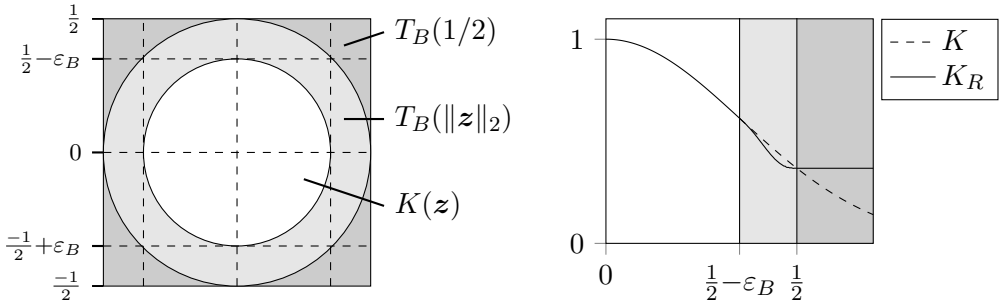


Figure 3.3: The regularized kernel K_R with $\varepsilon_B > 0$ for the Gaussian kernel from (3.3). Left: The definition pieces of $K_R(\boldsymbol{z})$ on $\mathbb{T}^2$ for $d = 2$ (reproduced from [2]). Right: The function values of $K_R(\boldsymbol{z})$ on a 1-dimensional slice through $\mathbb{T}^d$, $r \mapsto K(r\boldsymbol{\eta})$, for arbitrary d and an arbitrary direction with $\|\boldsymbol{\eta}\|_2 = 1$.

width $\varepsilon_B > 0$ and a function T_B that satisfies[1]

$$\begin{aligned} \frac{\partial^s}{\partial r^s} T_B\left(\frac{1}{2} - \varepsilon_B\right) &= \frac{\partial^s}{\partial r^s} K\left(\frac{1}{2} - \varepsilon_B\right) \quad \forall\ s = 0, \dots, p-1, \\ T_B\left(\frac{1}{2}\right) = K\left(\frac{1}{2}\right), \qquad & \frac{\partial^s}{\partial r^s} T_B\left(\frac{1}{2}\right) = 0 \quad \forall\ s = 1, \dots, p-1. \end{aligned}$$

Various examples of how such a function can be determined are listed in [33]. One method is a two-point Taylor interpolation resulting in an algebraic polynomial of degree $2p - 1$. The approximation is visualized in Figure 3.3.

Reformulation as two NDFTs. Now if we approximate f_j from (3.25) by replacing K with K_{RF} as in (3.26), that leads to

$$\begin{aligned} f_j \approx f_j^{RF} &= \sum_{i=1}^{n_x} \alpha_i K_{RF}(\boldsymbol{x}_i - \boldsymbol{y}_j) = \sum_{i=1}^{n_x} \alpha_i \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} \hat{b}_{\boldsymbol{l}} \exp\left(2\pi \mathrm{i} \boldsymbol{l}^T (\boldsymbol{x}_i - \boldsymbol{y}_j)\right) \\ &= \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} \left(\hat{b}_{\boldsymbol{l}} \sum_{i=1}^{n_x} \alpha_i \exp\left(2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{x}_i\right) \right) \exp\left(-2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{y}_j\right). \end{aligned} \tag{3.29}$$

Here we easily see that the inner sum is an adjoint NDFT with points $\boldsymbol{x}_i$ and coefficients α_i, while the outer sum is a forward NDFT with points $\boldsymbol{y}_j$ and coefficients given by the result of the first NDFT scaled by the $b_{\boldsymbol{l}}$.

[1] The constraint $T_B(\frac{1}{2}) = K(\frac{1}{2})$ is not strictly required for the desired smoothness, but it is often included to avoid arbitrary values of T_B.

Fast summation for arbitrary points. Up to now, we always assumed that all points $\boldsymbol{x}_i, \boldsymbol{y}_j$ are contained in a ball around the origin with radius $\frac{1}{4} - \frac{\varepsilon_B}{2}$. In the general case, the point sets are localized in an arbitrary region of $\mathbb{R}^d$. In order to apply the fast summation process, we need to shift and scale the points.

First choose a shift $\boldsymbol{s} \in \mathbb{R}^d$ and scaling factor $\rho > 0$ such that the maximum norm $\tilde{\rho}$ of any shifted point $\|\boldsymbol{x}_i - \boldsymbol{s}\|_2$ or $\|\boldsymbol{y}_j - \boldsymbol{s}\|_2$ is (roughly) minimal. For ease of implementation, $\boldsymbol{s}$ is often just set as the centroid position of all data points. Then set $\rho = (\frac{1}{2} - \varepsilon_B)/\tilde{\rho}$ and the new points as

$$\tilde{\boldsymbol{x}}_i = \rho\,(\boldsymbol{x}_i - \boldsymbol{s}), \qquad \tilde{\boldsymbol{y}}_j = \rho\,(\boldsymbol{y}_j - \boldsymbol{s}). \tag{3.30}$$

The new points satisfy $\tilde{\boldsymbol{x}}_i - \tilde{\boldsymbol{y}}_j = \rho\,(\boldsymbol{x}_i - \boldsymbol{y}_j)$, so in order for the kernel values to stay the same, we need the scaled kernel

$$\tilde{K}(\tilde{\boldsymbol{z}}) = K(\tilde{\boldsymbol{z}}/\rho) \qquad \forall\ \tilde{\boldsymbol{z}} \in \mathbb{R}^d. \tag{3.31}$$

For the example of the Gaussian kernel as in (3.3), this results in

$$\tilde{K}(\tilde{\boldsymbol{z}}) = \exp\left(-\frac{\|\tilde{\boldsymbol{z}}/\rho\|^2}{\sigma^2}\right) = \exp\left(-\frac{\|\tilde{\boldsymbol{z}}\|^2}{(\rho\sigma)^2}\right),$$

a scaled Gaussian with localization parameter $\tilde{\sigma} = \rho\sigma$.

Algorithm 3.2 NFFT-based fast summation

Input: Smooth kernel K, point sets $\mathcal{X}, \mathcal{Y} \subset \mathbb{R}^d$ with $|\mathcal{X}| = n_x$ and $|\mathcal{Y}| = n_y$, input coefficients $\boldsymbol{\alpha} \in \mathbb{C}^{n_x}$, bandwidth N, oversampled bandwidth $M > N$, window cutoff parameter m, regularization width $0 \le \varepsilon_B < \frac{1}{2}$, regularization smoothness $p \in \mathbb{N}$

Output: $\boldsymbol{f} \in \mathbb{C}^{n_y}$ with $\boldsymbol{f} \approx \mathcal{K}_{\mathcal{Y}\mathcal{X}}\boldsymbol{\alpha}$

1: Make sure that all points have norm $< \frac{1}{4} - \frac{\varepsilon_B}{2}$, otherwise replace the points by (3.30) and the kernel by (3.31)
2: Setup the regularized kernel K_R as in (3.28) with the regularization determined by ε_B and p
3: Compute Fourier coefficients $\hat{b}_{\boldsymbol{l}}$ as in (3.27) for all $\boldsymbol{l} \in \mathcal{I}_N^d$
4: Approximate $\hat{g}_{\boldsymbol{l}} = \sum_{k=1}^{n_x} \alpha_k \exp\left(2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{x}_k\right)$ for $\boldsymbol{l} \in \mathcal{I}_N^d$ via an adjoint NFFT with oversampled bandwidth M and window cutoff parameter m
5: Compute the scaled coefficients $g_{\boldsymbol{l}} = b_{\boldsymbol{l}} \hat{g}_{\boldsymbol{l}}$ for $\boldsymbol{l} \in \mathcal{I}_N^d$
6: Approximate $f_j = \sum_{\boldsymbol{l} \in \mathcal{I}_N^d} g_{\boldsymbol{l}} \exp\left(-2\pi \mathrm{i} \boldsymbol{l}^T \boldsymbol{y}_j\right)$ for $j = 1, \ldots, n_y$ via a forward NFFT with oversampled bandwidth M and window cutoff parameter m

The full method is presented in Algorithm 3.2. Note that the first three steps are setup steps that are independent of $\boldsymbol{\alpha}$. Hence, if matrix-vector products are required for multiple input vectors $\boldsymbol{\alpha}$, then the first three steps need only be taken once.

3.8 Proposition. *(Fast summation)*
Let a smooth kernel K, fixed point sets $\mathcal{X}, \mathcal{Y} \subset \mathbb{R}^d$, and parameters N,M,m,ε_B,p as in Algorithm 3.2 be given and fixed.
(a) The operator that maps $\boldsymbol{\alpha} \in \mathbb{C}^{n_x}$ to the output $\boldsymbol{f} \in \mathbb{C}^{n_y}$ of Algorithm 3.2 is linear and can be written as

$$\boldsymbol{f} = \mathcal{K}_{\mathcal{Y}\mathcal{X}}^{\mathrm{FS}} \boldsymbol{\alpha} \quad \text{with} \quad \mathcal{K}_{\mathcal{Y}\mathcal{X}}^{\mathrm{FS}} = \frac{1}{M^{2d}} B_{\mathcal{Y}} \mathcal{F}^H D \hat{B} D^H \mathcal{F} B_{\mathcal{X}}^T, \tag{3.32}$$

where $\mathcal{F}$ is the Fourier matrix from (3.8); $B_{\mathcal{Y}}$, $B_{\mathcal{X}}$, and D are as in Proposition 3.5 for the point sets $\mathcal{X}$ and $\mathcal{Y}$, respectively; and $\hat{B}$ is the diagonal matrix holding the $\hat{b}_{\boldsymbol{l}}$ from (3.27).

(b) The complexity of Algorithm 3.2 is in $\mathcal{O}(M^d \log(M) + m^d n)$. If steps 1–3 are precomputed, their complexity is $\mathcal{O}(N^d \log(N) + nd)$.

(c) For real-valued input $\boldsymbol{\alpha} \in \mathbb{R}^{n_y}$, the output $\boldsymbol{f} \in \mathbb{R}^{n_x}$ of Algorithm 3.2 is real-valued as well. Hence $\mathcal{K}_{\mathcal{Y}\mathcal{X}}^{\mathrm{FS}} \in \mathbb{R}^{n_y \times n_x}$ is real valued and we have

$$\left(\mathcal{K}_{\mathcal{Y}\mathcal{X}}^{\mathrm{FS}}\right)^T = \mathcal{K}_{\mathcal{X}\mathcal{Y}}^{\mathrm{FS}}.$$

For $\mathcal{Y} = \mathcal{X}$, $\mathcal{K}_{\mathcal{X}\mathcal{X}}^{\mathrm{FS}}$ is symmetric.

(d) The error of $\mathcal{K}_{\mathcal{Y}\mathcal{X}}^{\mathrm{FS}}$ can be estimated by

$$\|(\mathcal{K}_{\mathcal{Y}\mathcal{X}}^{\mathrm{FS}} - \mathcal{K}_{\mathcal{Y}\mathcal{X}})\, \boldsymbol{\alpha}\|_\infty \leq C_{\mathrm{FS}} \|\boldsymbol{\alpha}\|_1 \quad \text{with} \quad C_{\mathrm{FS}} = C_{\mathrm{RF}} + \tilde{C}_{\mathrm{NFFT}}.$$

Here the first part of the error constant reflects the kernel interpolation error,

$$C_{\mathrm{RF}} = C_{\mathrm{RF}}(d, K, N, \varepsilon_B, p) = \max_{\|\boldsymbol{z}\|_2 \leq \frac{1}{2} - \varepsilon_B} \left| K_{RF}(\boldsymbol{z}) - K(\boldsymbol{z}) \right|, \tag{3.33}$$

and depends on d and K and the chosen parameters N, ε_B, and p, but is independent of n_x, n_y. Estimates for C_{RF} can be found, e.g., in [81, 33]. The second part $\tilde{C}_{\mathrm{NFFT}} = \tilde{C}_{\mathrm{NFFT}}(d, K, \varphi, N, M, m)$ contains the propagated error of the two NFFTs, cf. Proposition 3.5, which is independent of n_x, n_y and vanishes for large m.

Proof. **(a)** The operator identity follows directly from the derivation above.

(b) Step 1 has complexity $\mathcal{O}(nd)$ because that is the storage size of $\mathcal{X}$. Steps 2 and 3 consist mainly of an FFT with complexity $\mathcal{O}(N^d \log(N))$. Steps 4 and 6 are NFFTs with complexity $\mathcal{O}(M^d \log(M) + m^d n)$ according to Proposition 3.5. The diagonal multiplication in between only has complexity $\mathcal{O}(N^d)$.

(c) It can be easily seen that for two complex vectors connected by the DFT, $\boldsymbol{g} = \mathcal{F}\hat{\boldsymbol{g}}$ and $\hat{\boldsymbol{g}} = \frac{1}{M^d} \mathcal{F}^H \boldsymbol{g}$, it holds that $\hat{\boldsymbol{g}}$ is real-valued if and only if $\boldsymbol{g}$ is "hermitian" in

the sense that $g_{\boldsymbol{l}}$ is the complex conjugate of $g_{(-\boldsymbol{l})}$ for all $\boldsymbol{l} \in \mathcal{I}_M^d$ [79]. Because B is real-valued, this shows that $\mathcal{F}B^T\boldsymbol{\alpha}$ is hermitian and gets multiplied element-wise by the diagonals of D^H, $\hat{B}$, and D, all of which all come from DFTs of real values and are thus also hermitian. Consequently $\mathcal{F}^H(D\hat{B}D^H\mathcal{F}B^T\boldsymbol{\alpha})$ and hence $\boldsymbol{f}$ must be real-valued. The symmetry properties follow trivially from the operator characterization (3.32).

(d) The basic form of the error estimation is a standard result in, e.g., [81], where the NFFT error is mostly ignored for the fast summation error estimation. The addition of $\tilde{C}_{\mathrm{NFFT}}$ can be achieved by a standard error split. Unlike the constant C_{NFFT} from Proposition 3.5, $\tilde{C}_{\mathrm{NFFT}}$ depends on K and N because of the double propagation and the presence of $\hat{B}$.

□

Use for square adjacency matrices. As stated in (3.4), the adjacency matrix of a fully connected graph (2.2) based on the kernel K is the sum of the kernel matrix with $\mathcal{Y} = \mathcal{X}$ and a scaled identity based on the loop weight. By replacing $\mathcal{K}_{\mathcal{X}\mathcal{X}}$ with $\mathcal{K}^{\mathrm{FS}}_{\mathcal{X}\mathcal{X}}$ as in Proposition 3.8, the NFFT-based approximation constitutes an inexact adjacency operator

$$\mathcal{A}^{(E)} = \mathcal{A} + E \quad \text{with} \quad E = \mathcal{K}^{\mathrm{FS}}_{\mathcal{X}\mathcal{X}} - \mathcal{K}_{\mathcal{X}\mathcal{X}}, \tag{3.34}$$

retaining the error approximations from Proposition 3.8 of the form

$$\|E\boldsymbol{\alpha}\|_\infty \leq C_{\mathrm{FS}}\|\boldsymbol{\alpha}\|_1 \leq n\, C_{\mathrm{FS}}\|\boldsymbol{\alpha}\|_\infty.$$

Hence, the operator norm of E w.r.t. to a single norm can *not* be estimated independent of the number of points, but we only get

$$\|E\|_\infty \leq n\, C_{\mathrm{FS}}. \tag{3.35}$$

The full final setup of $\mathcal{A}^{(E)}$ is summarized in Algorithm 3.3.

3.1.4 Comparison to the Fast Gauss Transform

Fast summation with the Gaussian kernel from (3.3) is a very important special case, with applications, e.g., in the fast solution of integral equations. With this kernel, the summation of (3.25) is also known as the Discrete Gauss Transform (DGT) and its fast approximation as the Fast Gauss Transform (FGT).

A number of FGT algorithms have been proposed, but they are commonly only targeted at the univariate case ($d = 1$) or the case where the parameter σ is particularly small that $K(\boldsymbol{x}_i - \boldsymbol{y}_j)$ vanishes for most data point pairs, i.e., $\mathcal{K}_{\mathcal{Y}\mathcal{X}}$ is numerically sparse. The latter case is of special practical relevance in some applications and has

Algorithm 3.3 Setup of approximate matrix-vector products with the adjacency matrix of a kernel-based fully-connected graph

Input: Kernel function K, point set $\mathcal{X}$, loop weight w_{loop}, fast summation parameters N, M, ε_B, p

Output: Linear operator $\mathcal{A}^{(E)}$ that approximates the adjacency matrix of a fully connected graph as in (2.2)

1: Perform steps 1–3 of Algorithm 3.2

2: Setup the operator $\mathcal{A}^{(E)}$ such that $\boldsymbol{y} = \mathcal{A}^{(E)}\boldsymbol{x}$ is evaluated via:

i) Compute $\tilde{\boldsymbol{y}} = \mathcal{K}^{\text{FS}}_{\mathcal{X}\mathcal{X}}\,\boldsymbol{x}$ via steps 4–6 of Algorithm 3.2

ii) Return $\boldsymbol{y} = \tilde{\boldsymbol{y}} + \big(w_{\text{loop}} - K(0)\big)\,\boldsymbol{x}$

hence received increased interest. However, these methods are not suitable for the case where σ is medium to large, since the methods either truncate important entries or lose their complexity benefit over the direct computation of the DGT. One popular class of FGT methods are approaches based on trees, voxelization, or similar data structures [42, 43, 102, 72]. One of these methods is the FIGTree method that we will use as a baseline in Section 3.5.1.

A specialized FGT based on the NFFT has been proposed in [59] for the univariate case, but allowing complex parameters. This method is closely related to the scheme presented in Section 3.1.3, but using the analytical formula for the Fourier coefficients $\hat{b}_{\boldsymbol{l}}$ of the non-regularized kernel. Dropping the boundary regularization is once again targeted at the case of a small parameter σ, when the kernel is close to zero near the boundary anyway and does not require regularization.

3.2 Eigenvalue computation with Krylov methods

Computing eigenvalues and eigenvectors of large matrices is one of the fundamental problems in numerical linear algebra, along with the solution of linear equations [40, 51, 87]. With ever increasing magnitudes in data availability and problem sizes, computing the whole spectrum of a matrix is often infeasible from the start. Instead, sophisticated iterative methods have been developed to efficiently approximate a small portion of the spectrum.

In this section, we will assume that the largest k eigenvalues of a large-scale symmetric linear operator $A \in \mathbb{R}^{n\times n}$ are wanted, where $k \ll n$. In the context of symmetric eigenvalue problems, the term "largest" can either refer to the *largest real* eigenvalues (those closest to positive infinity) or the *largest magnitude* eigenvalues (those with largest absolute value, closest to any infinity, and farthest from zero). For now, we will assume that A is shifted in such a way that both these definitions

are the same, e.g., because A is positive semi-definite.

The usual setting is that A is not explicitly available through its matrix entries, but only via a method to evaluate matrix-vector products. Iterative methods are naturally equipped to deal with this setting as they only need to apply a few matrix-vector products, typically one per iteration. This is important to avoid a runtime and storage complexity of $\mathcal{O}(n^2)$ in special cases where matrix-vector products can be evaluated cheaper than that. The most important such case is that of a sparse matrix, but it crucially also enables the combination with the NFFT-based fast summation from the previous section for fast approximation with kernel matrices.

Krylov subspace methods are arguably the most popular iterative methods as they are extensively studied, easy to implement (in their basic form), versatile and applicable in almost all situations, and give good convergence guarantees for the first eigenvalues.

3.2.1 Basics

First off, we are going to lay the foundation for the upcoming algorithms by introducing some fundamental concepts. For a more thorough introduction to the various techniques used for large-scale eigenvalue computation, see, e.g., [87].

3.9 Definition. *(Rayleigh-Ritz method)*
In the Rayleigh-Ritz procedure, a few eigenpairs of a matrix $A \in \mathbb{R}^{n\times n}$ are approximated by setting up a projection matrix $V \in \mathbb{R}^{n\times m}$ with orthonormal columns, $V^TV = I$, and the projected matrix $B = V^TAV$. Let $\theta_i, \boldsymbol{q}_i$ $(i = 1, \ldots, m)$ be the m eigenpairs of B. Then θ_i is called a *Ritz value* and $\boldsymbol{u}_i = V\boldsymbol{q}_i$ is the corresponding *Ritz vector*. The Ritz pairs give approximations to the eigenpairs of A, $A\boldsymbol{u}_i \approx \theta_i\boldsymbol{u}_i$, in that the residual of that approximation is orthogonal to the columns of V. If A is symmetric, we expect the $\boldsymbol{q}_i$ (and hence the Ritz vectors $\boldsymbol{u}_i$) to be chosen orthonormally.

The Rayleigh-Ritz method itself is a useful starting point for eigenvalue approximation, but it directly poses the question of how to construct the necessary projection matrix. In iterative methods, new columns are added sequentially to slowly expand the subspace of $\mathbb{R}^n$ onto which the problem is projected. The new column must be orthonormal to the existing ones and contribute a new "search direction" pointing where the correct solution is expected. As known from the power method, the true eigenvectors for the largest eigenvalues are reinforced by the higher powers of A. This observation naturally leads to choosing V as a basis matrix of a Krylov space.

3.10 Definition. *(Krylov space)*
For a matrix $A \in \mathbb{R}^{n\times n}$ and a starting vector $\boldsymbol{v} \in \mathbb{R}^n$, the k-dimensional Krylov

space is the linear subspace of $\mathbb{R}^n$ defined by

$$\mathcal{K}_k(A, \boldsymbol{v}) = \operatorname{span}\left\{\boldsymbol{v}, A\boldsymbol{v}, A^2\boldsymbol{v}, \ldots, A^{k-1}\boldsymbol{v}\right\}. \tag{3.36}$$

Krylov methods operate by iteratively constructing an incremental Krylov space. This typically leads to *Arnoldi* or *Lanczos* decompositions, which were generalized in [92] to the following more flexible decomposition.

3.11 Definition. *(Krylov decomposition)*
For a given matrix $A \in \mathbb{R}^{n\times n}$, a Krylov decomposition of order m is given by

$$AV_m = V_m B_m + \boldsymbol{v}_{m+1}\boldsymbol{b}_{m+1}^T = V_{m+1}\begin{bmatrix} B_m \\ \boldsymbol{b}_{m+1}^T \end{bmatrix} \tag{3.37}$$

with $V_m \in \mathbb{R}^{n\times m}$, $\boldsymbol{v}_{m+1} \in \mathbb{R}^n$, $B_m \in \mathbb{R}^{m\times m}$, and $\boldsymbol{b}_{m+1} \in \mathbb{R}^m$ such that $V_{m+1} = \left[V_m, \boldsymbol{v}_{m+1}\right]$ has orthogonal columns, $V_{m+1}^T V_{m+1} = I$.

Note that in any Krylov decomposition, the matrix B_m is determined by V_m via the equation

$$V_m^T A V_m = \underbrace{V_m^T V_m}_{=I} B_m + \underbrace{V_m^T \boldsymbol{v}_{m+1}}_{=0}\boldsymbol{b}_{m+1}^T = B_m.$$

In particular, if A is symmetric, B_m must also be symmetric.

Krylov decompositions can be used either for eigenvalue approximation with the Rayleigh-Ritz method or also for the approximative solution of linear systems of equations with A. Here we will focus on the first case. One of the advantages of these decompositions is that the residual of the Ritz pairs can be easily evaluated, as is stated in the following lemma.

3.12 Lemma. *(Error of the Rayleigh-Ritz procedure for Krylov decompositions)*
In a Krylov decomposition (3.37), let $\theta_i, \boldsymbol{q}_i$ be the eigenpairs of B_m. Then the residual norms of the Ritz pairs $\theta_i, \boldsymbol{u}_i = V_m\boldsymbol{q}_i$ are given in exact arithmetics by

$$\|A\boldsymbol{u}_i - \theta_i\boldsymbol{u}_i\|_2 = \left|\boldsymbol{b}_{m+1}^T\boldsymbol{q}_i\right|. \tag{3.38}$$

Proof. The residual is directly proven by

$$\begin{aligned} AV_m\boldsymbol{q}_i = V_m B_m\boldsymbol{q}_i + \boldsymbol{v}_{m+1}\boldsymbol{b}_{m+1}^T\boldsymbol{q}_i &= V_m\theta_i\boldsymbol{q}_i + (\boldsymbol{b}_{m+1}^T\boldsymbol{q}_i)\boldsymbol{v}_{m+1} \\ \Rightarrow \qquad A\boldsymbol{u}_i - \theta_i\boldsymbol{u}_i &= (\boldsymbol{b}_{m+1}^T\boldsymbol{q}_i)\boldsymbol{v}_{m+1}, \end{aligned}$$

where $\boldsymbol{v}_{m+1}$ has norm 1. □

The general idea of the Krylov-based Rayleigh-Ritz procedure is summarized in Algorithm 3.4.

Algorithm 3.4 Rayleigh-Ritz method based on a Krylov decomposition

Input: Matrix A, number of desired eigenvalues k, subspace dimension $m \geq k$
Output: k Ritz pairs $\theta_i, \boldsymbol{u}_i$ with accuracies ε_i such that $\|A\boldsymbol{u}_i - \theta_i \boldsymbol{u}_i\|_2 \leq \varepsilon_i$ for all $i = 1, \ldots, k$

1: Obtain a Krylov decomposition of order m as in (3.37)
2: Compute the k largest-magnitude eigenpairs $\theta_i, \boldsymbol{q}_i$ of B_m
3: **for** $i = 1, \ldots, k$ **do**
4: Set up the Ritz vector $\boldsymbol{u}_i = V_m \boldsymbol{q}_i$
5: Compute the accuracy $\varepsilon_i = |\boldsymbol{b}_{m+1}^T \boldsymbol{q}_i|$
6: **end for**

3.2.2 The Lanczos process

The most common way to create a Krylov decomposition is the Arnoldi algorithm, which has been introduced in 1951 [6]. It consists of iteratively constructing a sequence of orthonormal vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots$ such that $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ span the Krylov space $\mathcal{K}_k(A, \boldsymbol{v}_1)$. The first vector $\boldsymbol{v}_1$ can be generated randomly and must only have norm 1. In iteration j, the next vector $\boldsymbol{v}_{j+1}$ is computed by applying A to the previous vector $\boldsymbol{v}_j$ and orthogonalizing against all computed vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_j$. This leads to

$$\boldsymbol{v}_{j+1} = \frac{1}{h_{j+1,j}} \left(A\boldsymbol{v}_j - \sum_{i=1}^{j} h_{i,j} \boldsymbol{v}_i \right),$$

where $h_{j+1,j}$ is the normalizing factor to achieve $\|\boldsymbol{v}_{j+1}\|_2 = 1$, and $h_{i,j} = \boldsymbol{v}_i A \boldsymbol{v}_j$ for $i = 1, \ldots, j$. Rearranging and collecting the first m of these equations, we obtain

$$AV_m = V_m H_m + \begin{bmatrix} 0 & \cdots & 0 & h_{m+1,m}\boldsymbol{v}_{m+1} \end{bmatrix}$$

with $V_m = \begin{bmatrix} \boldsymbol{v}_1 & \cdots & \boldsymbol{v}_m \end{bmatrix}$ and the *upper Hessenberg* matrix

$$H_m = \begin{bmatrix} h_{1,1} & \cdots & \cdots & h_{m,m} \\ h_{2,1} & \ddots & & \vdots \\ & \ddots & \ddots & \vdots \\ & & h_{m,m-1} & h_{m,m} \end{bmatrix}.$$

This constitutes a Krylov decomposition as in (3.37) with $B_m = H_m$ and $\boldsymbol{b}_{m+1} = h_{m+1,m}\boldsymbol{e}_m$, where $\boldsymbol{e}_m = \begin{bmatrix} 0 & \cdots & 0 & 1 \end{bmatrix}^T \in \mathbb{R}^m$ is the m-th unit vector.

Though the Lanczos process predates the Arnoldi method, it can be thought of as a theoretical simplification for the case that A is symmetric [60]. Then the projected Hessenberg matrix $H_m = V_m^T A V_m$ is symmetric as well, which makes it a tridiagonal

matrix

$$T_m = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_m \\ & & \beta_m & \alpha_m \end{bmatrix} \quad \text{with} \quad \alpha_j = h_{j,j},\ \beta_j = h_{j,j-1}. \tag{3.39}$$

In particular, this shows that in exact arithmetics $A\boldsymbol{v}_j - \alpha_j \boldsymbol{v}_j - \beta_j \boldsymbol{v}_{j-1}$ must be orthogonal to all $\boldsymbol{v}_i$, $i = 1, \ldots, j$. Hence in the original Lanczos process, there was no orthogonalization against the previous $\boldsymbol{v}_i$. In practice however, rounding errors cause quick loss of this orthogonality and thus high numerical instability. The remedy is reorthogonalization against the previously computed vectors in each iteration [77, 40].

Algorithm 3.5 The Lanczos process for eigenvalue approximation

Input: Symmetric matrix $A \in \mathbb{R}^{n \times n}$, subspace dimension m
Output: Krylov decomposition $AV_m = V_m T_m + \beta_{m+1} \boldsymbol{v}_{m+1} \boldsymbol{e}_m^T$

1: Randomly generate a starting vector $\boldsymbol{v}_1 \in \mathbb{R}^n$ with norm $\|\boldsymbol{v}_1\|_2 = 1$
2: **for** $j = 1, \ldots, m$ **do**
3: $\quad \boldsymbol{w} = A\boldsymbol{v}_j$
4: $\quad \alpha_j = \boldsymbol{v}_j^T \boldsymbol{w}$
5: $\quad \boldsymbol{w} = \boldsymbol{w} - \alpha \boldsymbol{v}_j - \beta_j \boldsymbol{v}_{j-1}$ (last part only if $j > 1$)
6: $\quad$ Reorthogonalization: $\boldsymbol{w} = \boldsymbol{w} - (\boldsymbol{v}_i^T \boldsymbol{w}) \boldsymbol{v}_i$ for all $i = 1, \ldots, j$
7: $\quad \beta_{j+1} = \|\boldsymbol{w}\|_2$
8: $\quad \boldsymbol{v}_{j+1} = \frac{1}{\beta_{j+1}} \boldsymbol{w}$
9: **end for**
10: Set up $V_m = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]$ and T_m as in (3.39)

The full Lanczos process is described in Algorithm 3.5, though it is most commonly combined with the Rayleigh-Ritz method for eigenvalue approximation, cf. Algorithm 3.4. In that case, the eigendecomposition of T_m can be computed efficiently with complexity $\mathcal{O}(m^2)$ using a divide-and-conquer method [40].

Due to the connection of Krylov spaces to matrix polynomials, a sophisticated convergence theory known as the Kaniel-Paige theory is possible, yielding estimates for the distances between Ritz values and exact eigenvalues. We here state the most important theorem, which can be found with a proof, e.g., in [87, Section 6.6].

3.13 Theorem. *(Kaniel-Paige convergence theory)*
Let the exact eigenvalues of a symmetric matrix $A \in \mathbb{R}^{n \times n}$ be given by $\mu_1 \geq \mu_2 \geq \ldots \geq \mu_n$ with corresponding unit eigenvectors $\boldsymbol{w}_i$. Let the sorted Ritz vectors of the Lanczos process of order m be given by $\theta_1^{(m)} \geq \ldots \geq \theta_m^{(m)}$. Then for all $i = 1, \ldots, m$

it holds

$$\mu_i \geq \theta_i^{(m)} \geq \mu_i - (\mu_1 - \mu_n) \, \frac{1 - (\boldsymbol{w}_i^T v_1)^2}{(\boldsymbol{w}_i^T v_1)^2} \, \left(\frac{\kappa_i^{(m)}}{C_{m-i}(1 + 2\rho_i)} \right)^2 \tag{3.40}$$

with the Chebyshev polynomials of the first kind C_s, given by

$$C_0(t) = 1, \qquad C_1(t) = t, \qquad C_s(t) = 2tC_{s-1}(x) - C_{s-2}(t) \quad (s \geq 2)$$

for all $t \in \mathbb{R}$, and the constants

$$\kappa_1^{(m)} = 1, \qquad \kappa_i^{(m)} = \prod_{j=1}^{i-1} \frac{\theta_j^{(m)} - \mu_n}{\theta_j^{(m)} - \mu_i} \quad \forall \; i = 2, \ldots, m \tag{3.41}$$

and

$$\rho_i = \frac{\mu_i - \mu_{i+1}}{\mu_{i+1} - \mu_n} \geq 0. \tag{3.42}$$

3.14 Remark.
It is known that for $t > 1$, the values of the Chebyshev polynomials can be estimated by $C_s(t) \geq \frac{1}{2}\big(t + \sqrt{t^2 - 1}\big)^s$ [87]. Hence, for fixed i with $\mu_i > \mu_{i+1}$ and increasing subspace dimension $m \to \infty$, the factor $1/C_{m-i}(1 + 2\rho_i)$ converges to zero exponentially fast while all other factors are bounded. The exact rate of convergence depends crucially on the ratio ρ_i.

3.2.3 The Krylov-Schur method

In order to use Algorithm 3.5 in practice, the choice of the subspace dimension m is still an open question. Even though the Kaniel-Paige convergence theory ensures asymptotically good quality of the Ritz values for $m \to \infty$, this does not provide an a priori estimate of m in order to achieve a certain accuracy.

In a simple solution, it is possible to use the Lanczos process with a more flexible termination criterion instead of a fixed m. Given a target tolerance ε for the residual norm $\|A\boldsymbol{u}_i - \theta_i \boldsymbol{u}_i\|_2$, repeatedly compute the k largest current Ritz pairs and their residual norms as stated by Lemma 3.12. If all residuals are below ε, the algorithm is finished, otherwise more iterations are needed.

The problem with this approach is that the effort of reorthogonalization scales with m. This makes later iterations much more expensive than the earlier ones.

One way to avoid quadratic behavior is by "restarting" the method once the subspace gets too large, without throwing away the previously computed information. The most famous such method is the Implicitly Restarted Arnoldi method [61], which is implemented in the popular ARPACK library [62] and thereby used in many ap-

plications, e.g., in Python's `scipy.sparse.linalg.eigsh` function.[2] However, these restarts can lead to implementational difficulties [49]. A mathematically equivalent reformulation leads to the *Krylov-Schur* algorithm proposed by Stewart [92], which we will present in this section. It is used for other popular implementations like MATLAB's `eigs` function[3] or the SLEPc library [49].

Assume that the k largest eigenvalues are desired and we have an arbitrary Krylov decomposition as in (3.37), for example by means of the Lanczos method. We start by "locking" the p largest eigenpairs of $B_m = V_m^T A V_m$, again denoted by $\theta_i, \boldsymbol{q}_i$ ($i = 1, \dots, p$). The number $p < m$ can be chosen by computing the residuals of the Ritz pairs as in (3.38), evaluating how many have already converged, and picking p in between the number of converged vectors and m. Then the remaining unwanted $m-p$ eigenpairs can be "purged" by multiplying the decomposition by $Q_p = [\boldsymbol{q}_1, \dots, \boldsymbol{q}_p] \in \mathbb{R}^{m \times p}$, yielding

$$A\tilde{V}_p = \tilde{V}_p \tilde{B}_p + \tilde{\boldsymbol{v}}_{p+1} \tilde{\boldsymbol{b}}_{p+1}^T$$

with $\tilde{V}_p = V_m Q_p$, $\tilde{B}_p = \operatorname{diag}(\theta_1, \dots, \theta_p)$, $\tilde{\boldsymbol{v}}_{p+1} = \boldsymbol{v}_{m+1}$, and $\tilde{\boldsymbol{b}}_{p+1} = Q_p^T \boldsymbol{b}_{m+1}$, which is a smaller Krylov decomposition of order p.

Afterwards, this decomposition can be expanded to order m again by the Lanczos process. Let us now omit the tilde on the variable names. For the first new iteration, we add $\boldsymbol{v}_{p+1}$ to V_p. The new entries in B_{p+1} are determined via $B_{p+1} = V_{p+1}^T A V_{p+1}$ as

$$B_{p+1} = \begin{bmatrix} B_p & \boldsymbol{b}_{p+1} \\ \boldsymbol{b}_{p+1}^T & \alpha_{p+1} \end{bmatrix} \quad \text{with} \quad \alpha_{p+1} = \boldsymbol{v}_{p+1}^T A \boldsymbol{v}_{p+1}.$$

For the next vector, set $\boldsymbol{v}_{p+2}$ to the normalized result of $A\boldsymbol{v}_{p+1}$ after orthogonalization against $\boldsymbol{v}_1, \dots, \boldsymbol{v}_{p+1}$, which should be given in exact arithmetics by

$$\boldsymbol{v}_{p+2} = \frac{1}{\beta_{p+2}} \left(A\boldsymbol{v}_{p+1} - V_p \boldsymbol{b}_{p+1} - \alpha_{p+1} \boldsymbol{v}_{p+1} \right)$$

with the familiar normalizing factor β_{p+2}. Finally we can now perform normal Lanczos steps to increase the subspace dimension to m again. This gives us a Krylov decomposition with

$$B_m = \left[\begin{array}{c|cccc} B_p & \boldsymbol{b}_{p+1} & 0 & \cdots & 0 \\ \hline \boldsymbol{b}_{p+1}^T & \alpha_{p+1} & \beta_{p+2} & & \\ 0 & \beta_{p+2} & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_m \\ 0 & & & \beta_m & \alpha_m \end{array}\right] \tag{3.43}$$

[2] As of SciPy 1.7.0, cf. the online documentation.
[3] As of MATLAB 2021a. The source code is available within the installation.

At this point, we can analyze the spectrum of B_m again to decide whether we need another restart, which can be done just like before. The whole process is summarized in Algorithm 3.6. For the first iteration, we exploit the notation by setting $p = 0$ so that (3.43) only consists of its bottom right tridiagonal block.

With this algorithm, we have a full implementation of a powerful state of the art eigenvalue solver. Open questions are only how to pick the subspace dimension m and the number of kept eigenvalues p. In modern implementations, it is common to choose $m = 2k$ (or larger if k is particularly small), and p as the number of converged eigenvalues plus k, but at most $k + (m - k)/2$. This way, later iterations will require fewer Lanczos steps because more eigenvalues have already converged, but at least $(m - k)/2$ vectors are newly computed each time to prevent stagnation.

3.3 Eigenvalue computation for the normalized Laplacian

Now that we have a way to approximate the fully connected adjacency matrix and to efficiently compute eigenvalues, we would like to apply these methods to compute the smallest eigenvalues of the symmetric normalized Laplacian $\mathcal{L}_{\text{sym}}$. Recall that we still denote its eigenpairs by $\lambda_i, \boldsymbol{u}_i$ for $i = 1, \ldots, n$ as in Definition 2.13, indexed in an *ascending* order,

$$0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n.$$

In the following, when referring to matrices where the *largest* part of the spectrum is of interest, we will denote the eigenvalues by μ_i and index them in descending order, i.e.,

$$\mu_1 \geq \mu_2 \geq \ldots \geq \mu_n.$$

The corresponding eigenvectors will be denoted by $\boldsymbol{w}_i$.

The challenging part of any operations with $\mathcal{L}_{\text{sym}}$ are the matrix-vector products with the normalized adjacency matrix,

$$\mathcal{A}_{\text{sym}} = \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}. \tag{3.44}$$

We have seen that the matrix $\mathcal{A}$ may be sparse (cf. citation graphs in Example 2.18), have low rank (cf. categorial hypergraphs in Example 2.20), or be based on a kernel that can be approximated well with NFFT-based fast summation (cf. 3D point clouds in Example 2.19). In all these cases, matrix-vector products with $\mathcal{A}$ can be evaluated or approximated in $\mathcal{O}(n)$ time.

3.3.1 Matrix shifts and Wielandt deflation

As stated in Section 3.2, Krylov methods by themselves will always converge towards those eigenvalues with largest absolute value. However, we require the eigenvalues of $\mathcal{L}_{\text{sym}}$ that are closest to zero.

Algorithm 3.6 Krylov-Schur method for eigenvalue approximation

Input: Symmetric matrix $A \in \mathbb{R}^{n \times n}$, number of desired eigenvalues k, subspace dimension $m > k$, relative tolerance ε, maximum number of iterations maxiter

Output: Ritz pairs $\theta_i, \boldsymbol{u}_i$ approximating the k largest eigenvalues of A with residuals $\|A\boldsymbol{u}_i - \theta_i \boldsymbol{u}_i\|_2 \leq \varepsilon|\theta_i|$

Generate random starting vector $\boldsymbol{v}_1 \in \mathbb{R}^n$ with $\|\boldsymbol{v}_1\|_2 = 1$
Initialize $p = 0$ for the first iteration
for iter $= 1, \ldots,$ maxiter **do**
    for $j = p+1, \ldots, m$ **do**
        $\boldsymbol{w} = A\boldsymbol{v}_j$
        $\alpha_j = \boldsymbol{v}_j^T \boldsymbol{w}$
        $\boldsymbol{w} = \boldsymbol{w} - \alpha_j \boldsymbol{v}_j$
        if iter > 1 and $j = p+1$ **then**
            $\boldsymbol{w} = \boldsymbol{w} - \left[\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{j-1}\right] \boldsymbol{b}_{p+1}$
        else if $j > 1$ **then**
            $\boldsymbol{w} = \boldsymbol{w} - \beta_j \boldsymbol{v}_{j-1}$
        end if
        Reorthogonalization: $\boldsymbol{w} = \boldsymbol{w} - (\boldsymbol{v}_i^T \boldsymbol{w})\boldsymbol{v}_i$ for all $i = 1, \ldots, j$
        $\beta_{j+1} = \|\boldsymbol{w}\|_2$
        $\boldsymbol{v}_{j+1} = \frac{1}{\beta_{j+1}} \boldsymbol{w}$
    end for
    Set up $V_m = \left[\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m\right]$ and B_m as in (3.43)
    Compute all eigenpairs $\theta_i, \boldsymbol{q}_i$ of B_m, sorted in descending order
    Compute the residuals according to (3.38), where $\boldsymbol{b}_{m+1} = \beta_{m+1}\boldsymbol{e}_m$
    Count how many of the first eigenvalues have residuals below $\varepsilon|\theta_i|$
    if iter = maxiter **or** the largest k eigenvalues have converged **then**
        Exit with the Ritz values θ_i and Ritz vectors $\boldsymbol{u}_i = V_m \boldsymbol{q}_i$
    end if
    Choose $p < m$, at least equal to the number of converged eigenvalues
    $\boldsymbol{v}_j = V_m \boldsymbol{q}_j$ for $j = 1, \ldots, p$
    $\boldsymbol{v}_{p+1} = \boldsymbol{v}_{m+1}$
    $B_p = \operatorname{diag}(\theta_1, \ldots, \theta_p)$
    $\boldsymbol{b}_{p+1} = \left[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_p\right]^T \boldsymbol{b}_{m+1}$
end for

Typically, when eigenvalues of a matrix A close to a shift $\tau \in \mathbb{R}$ are wanted, the approach is to apply Krylov methods to $(A - \tau I)^{-1}$. Of course the inverse is not computed explicitly, but a linear system of equations with $A - \tau I$ is solved whenever a matrix-vector product is required. When the shift τ is chosen dynamically, we speak of a Rational Krylov method [86]. Still, the whole procedure is significantly more expensive than it is for the largest eigenvalues.

In our case, a convenient property is the boundedness of the spectrum. Recall that the *signless* normalized Laplacian from (2.6), $\mathcal{L}_{\text{sym}+} = I + \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}$, has the same eigenvectors as $\mathcal{L}_{\text{sym}}$ but with eigenvalues $\mu_i = 2 - \lambda_i$, also within the range $[0, 2]$. Hence the smallest λ_i can be recovered from the largest eigenvalues of $\mathcal{L}_{\text{sym}+}$.

The signless Laplacian is not the only matrix that comes into question. For any shift $\alpha \in \mathbb{R}$, we can define a shifted signless Laplacian $\mathcal{M}_\alpha = \alpha I - \mathcal{L}_{\text{sym}} = (\alpha - 1)I + \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}$, so $\alpha = 2$ yields the signless Laplacian. Its eigenvalues are obviously $\alpha - \lambda_i$ and as long as the shift is big enough, the smallest k eigenvalues of $\mathcal{L}_{\text{sym}}$ can be recovered from the largest-magnitude eigenvalues of $\mathcal{M}_\alpha$.[4]

If we were to use, e.g., a power method instead of a Krylov subspace method for the eigenvalue computation, the convergence rate would depend on α. Because the power method is best for large ratios between the two largest eigenvalues, it would be beneficial to choose α as small as possible [87].

For Krylov methods, however, there is no such dependence. In the Kaniel-Paige theory of Theorem 3.13, the eigenvalues only enter the estimate in the form of differences $\mu_i - \mu_j$, so convergence is faster if the computed eigenvalues are well separated. Shifting all eigenvalue by the same value α does not influence the convergence rate guarantees at all. In practice, we also did not notice an advantage for choosing α lower than two, so we will simply work with the signless Laplacian.

Wielandt deflation and its implementation in Krylov methods. An interesting, but rarely used technique for accelerated eigenvalue computation is the so-called *Wielandt deflation* [87, Section 4.2.1]. If the largest-magnitude eigenvalue μ_1 of a matrix A and a corresponding unit eigenvector $\boldsymbol{w}_1$ are known a priori, the eigenvalue can be removed from the computed part of the spectrum by instead using the *deflated* matrix $\hat{A} = A - \beta \boldsymbol{w}_1 \boldsymbol{w}_1^T$. If β is large enough, the largest eigenvalue of $\hat{A}$ is now the previous second largest eigenvalue $\boldsymbol{w}_2$. The corresponding eigenvectors have not changed, but $\boldsymbol{w}_1$ is now an eigenvector to the shifted eigenvalue $\mu_1 - \beta$. When using a Krylov method to compute the k largest-magnitude eigenvalues of A, we can compute only the $k - 1$ largest-magnitude eigenvalues of $\hat{A}$ and then add the eigenpair $\mu_1, \boldsymbol{w}_1$ manually.

Problems might arise if β is too small or too large, making $\mu_1 - \beta$ one of these

[4] To be precise, this is true as long as $\alpha > \frac{1}{2}(\lambda_k + \lambda_n)$, because for smaller values of α, $|\alpha - \lambda_n|$ starts to surpass $|\alpha - \lambda_k|$.

$k-1$ computed largest-magnitude eigenvalues. We can avoid this by setting $\beta = \mu_1$. With this choice, $\hat{A}\boldsymbol{x}$ is always orthogonal to $\boldsymbol{w}_1$ in exact arithmetic for all $\boldsymbol{x}$.

However, if we now employ an off-the-shelf Lanczos or Krylov-Schur method to $\hat{A}$ for the eigenvalue computation, we drop important information since the algorithm does not know that all leading eigenvectors are orthogonal to $\boldsymbol{w}_1$. Most importantly, if the starting vector $\boldsymbol{v}_1$ has some component in direction of $\boldsymbol{w}_1$, then the output eigenvector approximations of the Lanczos method will generally not be orthogonal to $\boldsymbol{w}_1$. When we subsequently augment the computed eigenvectors by the known $\boldsymbol{w}_1$, the full eigenvector matrix will violate orthonormality to a degree that surpasses the usual rounding errors. The simple solution is to always choose a starting vector $\boldsymbol{v}_1$ that is orthogonal to $\boldsymbol{w}_1$.

In exact arithmetic, now the following Krylov basis vectors $\boldsymbol{v}_2, \boldsymbol{v}_3, \ldots$ in the Lanczos process will be guaranteed to be orthogonal to $\boldsymbol{w}_1$ as well. However, just like the orthogonality of the $\boldsymbol{v}_i$ against each other, this orthogonality will quickly be lost due to the propagation of rounding errors. Hence $\boldsymbol{w}_1$ should be included in the reorthogonalization step in the Lanczos and Krylov-Schur methods, Algorithms 3.5 and 3.6.

If we adapt the algorithm with these two simple modifications, we don't even need to formally use the matrix $\hat{A} = A - \mu_1 \boldsymbol{w}_1 \boldsymbol{w}_1^T$. It is equivalent to just use the matrix vector product with A and then let the orthogonalization step take care of the deflation.

Wielandt deflation for the signless Laplacian. When computing the eigenvalues of $\mathcal{L}_{\text{sym}}$ via $\mathcal{L}_{\text{sym}+}$, the main condition of Wielandt deflation is met: The largest eigenpair of $\mathcal{L}_{\text{sym}+}$ is known a priori to be given by $\mu_1 = 2$ and $\boldsymbol{w}_1 = \boldsymbol{u}_1$. Recall from Proposition 2.12(b) that any multiple of $\mathcal{D}^{1/2}\mathbf{1}$ is an eigenvector to the eigenvalue $\lambda_1 = 0$ of the symmetric normalized Laplacian.

When the adjacency matrix is given in any form (either by its explicit entries, a sparse format, a low-rank factorization, or Algorithm 3.3), we can compute its degrees via a matrix-vector product with the vector of ones. At the same time, the result can be used to set up the unit eigenvector $\boldsymbol{u}_1$.

Of course, the matrix $\mathcal{L}_{\text{sym}+} = I + \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}$ does not need to be set up explicitly. Matrix vector products with $\mathcal{L}_{\text{sym}+}$ only require two element-wise multiplications with the diagonal of $\mathcal{D}^{-1/2}$, one application of $\mathcal{A}$, and an addition of the input vector. The full procedure with the techniques described above is summarized in Algorithm 3.7.

3.3.2 Effect of inexact degree computation

Algorithm 3.7 is also applicable if the input operator is not the exact adjacency matrix $\mathcal{A}$, but only an approximation $\mathcal{A}^{(E)}$, e.g., stemming from the NFFT-based

Algorithm 3.7 Eigenvalue approximation for $\mathcal{L}_{\text{sym}}$ with Wielandt deflation

Input: Operator providing matrix-vector products with the adjacency matrix $\mathcal{A}$; number of desired eigenvalues k; parameters m, ε, maxiter for Algorithm 3.6

Output: Approximations to the k smallest eigenvalues $\lambda_1, \ldots, \lambda_k$ of $\mathcal{L}_{\text{sym}}$ with corresponding orthonormal eigenvectors $\boldsymbol{u}_i$

1: $\boldsymbol{d}^{1/2} = \sqrt{\mathcal{A}\mathbf{1}}$ (element-wise root)

2: $\boldsymbol{u}_1 = \boldsymbol{d}^{1/2} \,/\, \|\boldsymbol{d}^{1/2}\|_2$

3: $\boldsymbol{d}^{-1/2} = 1 \oslash \boldsymbol{d}^{1/2}$ (element-wise division)

4: Set up the operator $\mathcal{L}_{\text{sym}+}$ such that $\boldsymbol{y} = \mathcal{L}_{\text{sym}+}\boldsymbol{x}$ is evaluated via:

   i) $\tilde{\boldsymbol{y}} = \boldsymbol{d}^{-1/2} \odot \boldsymbol{x}$ (element-wise multiplication)

   ii) $\tilde{\boldsymbol{y}} = \mathcal{A}\tilde{\boldsymbol{y}}$ (depending on the structure of $\mathcal{A}$)

   iii) $\tilde{\boldsymbol{y}} = \boldsymbol{d}^{-1/2} \odot \tilde{\boldsymbol{y}}$ (element-wise multiplication)

   iv) $\boldsymbol{y} = \boldsymbol{x} + \tilde{\boldsymbol{y}}$

5: Run Algorithm 3.6 to compute the $k-1$ largest Ritz values $\theta_1, \ldots, \theta_{k-1}$ and Ritz vectors $\tilde{\boldsymbol{u}}_1, \ldots, \tilde{\boldsymbol{u}}_{k-1}$ of $\mathcal{L}_{\text{sym}+}$ with the following adjustments:

   - Choose the starting vector $\boldsymbol{v}_1$ such that $\boldsymbol{u}_1^T \boldsymbol{v}_1 = 0$
   - During reorthogonalization, also orthogonalize the iterate $\boldsymbol{w}$ against $\boldsymbol{u}_1$

6: Merge the approximate eigenpairs $\lambda_i, \boldsymbol{u}_i$ of $\mathcal{L}_{\text{sym}}$ for $i = 1, \ldots, k$:

$$\lambda_i = \begin{cases} 0 & \text{if } i = 1, \\ 2 - \theta_{i-1} & \text{if } i > 1, \end{cases} \qquad \boldsymbol{u}_i = \begin{cases} \boldsymbol{u}_1 & \text{if } i = 1, \\ \tilde{\boldsymbol{u}}_{i-1} & \text{if } i > 1. \end{cases}$$

Algorithm 3.3.

In our framework, $\mathcal{A}^{(E)}$ is itself a constant and symmetric linear operator. This is different from other applications where inexact matrix-vector products mean that there is an independent error each time $\boldsymbol{y} \approx \mathcal{A}\boldsymbol{x}$ is evaluated, for example due to underlying randomness. Error theory for Krylov methods in such an environment can be found, e.g., in [91].

Here the result of $\mathcal{A}^{(E)}\boldsymbol{x}$ is not only deterministic, but even still linear in $\boldsymbol{x}$. Moreover, $\mathcal{A}^{(E)}$ is a symmetric matrix, which is important because otherwise different eigenvalue computation techniques would be required. Still, it is not necessarily an adjacency matrix of a perturbed graph because it might have negative entries.

Running Algorithm 3.7 with $\mathcal{A}$ replaced by $\mathcal{A}^{(E)}$ will first set up the approximated degree matrix

$$\mathcal{D}^{(E)} = \operatorname{diag}(\mathcal{A}^{(E)}\mathbf{1}) = \mathcal{D} + \operatorname{diag}(E\mathbf{1}). \tag{3.45}$$

and then give approximations of the eigenvalues of the propagated normalized Laplacian

$$\mathcal{L}_{\text{sym}}^{(E)} = I - \mathcal{A}_{\text{sym}}^{(E)} \quad \text{with} \quad \mathcal{A}_{\text{sym}}^{(E)} = (\mathcal{D}^{(E)})^{-1/2}\mathcal{A}^{(E)}(\mathcal{D}^{(E)})^{-1/2}. \tag{3.46}$$

This setup is of course only applicable if $(\mathcal{D}^{(E)})^{-1/2}$ exists. For negative entries in $\mathcal{D}^{(E)}$, taking a complex square root is no remedy because realness is a requirement for subsequent computations, analysis, and interpretation. Hence we will require the approximation to be good enough to make sure that all degrees are positive.

3.15 Lemma. *(Smallest eigenvalue of the approximated normalized Laplacian)*
For a symmetric approximated adjacency matrix $\mathcal{A}^{(E)} \in \mathbb{R}^{n\times n}$ such that all entries of $\mathcal{A}^{(E)}\mathbf{1}$ are positive, let $\mathcal{D}^{(E)}$ and $\mathcal{L}_{\text{sym}}^{(E)}$ be defined as in (3.45) and (3.46). Then $\lambda = 0$ is an eigenvalue of $\mathcal{L}_{\text{sym}}^{(E)}$ with eigenvector $\boldsymbol{u}^{(E)} = (\mathcal{D}^{(E)})^{1/2}\mathbf{1}$.
If in addition all entries of $\mathcal{A}^{(E)}$ are non-negative, then $\lambda = 0$ is the smallest eigenvalue of $\mathcal{L}_{\text{sym}}^{(E)}$, otherwise there might be a negative eigenvalue.

Proof. If $\mathcal{A}^{(E)}$ does not have any negative entries, then $\mathcal{L}_{\text{sym}}^{(E)}$ is the adjacency matrix of a perturbed graph and has all the properties introduced in Section 2.3, including positive semi-definiteness.

Some properties however, including the formula for $\boldsymbol{f}^T\mathcal{L}_{\text{sym}}\boldsymbol{f}$ in (2.9), do not technically require the underlying $\mathcal{A}$ to be an adjacency matrix. Hence they also apply to more general $\mathcal{A}^{(E)}$ as long as $\mathcal{A}_{\text{sym}}^{(E)}$ can be set up. For the eigenvalue $\lambda = 0$, this can be easily seen via

$$\begin{aligned}\mathcal{A}_{\text{sym}}^{(E)}\boldsymbol{u}^{(E)} &= (\mathcal{D}^{(E)})^{-1/2}\mathcal{A}^{(E)}(\mathcal{D}^{(E)})^{-1/2}(\mathcal{D}^{(E)})^{1/2}\mathbf{1} = (\mathcal{D}^{(E)})^{-1/2}\mathcal{A}^{(E)}\mathbf{1} \\ &= (\mathcal{D}^{(E)})^{-1/2}\mathcal{D}^{(E)}\mathbf{1} = (\mathcal{D}^{(E)})^{1/2}\mathbf{1} = \boldsymbol{u}^{(E)}\end{aligned}$$

proving $\mathcal{L}_{\text{sym}}^{(E)}\boldsymbol{u}^{(E)} = 0$. □

In practice, it appears sensible to require the approximation $\mathcal{A}^{(E)}$ to be good enough that $\mathcal{L}_{\text{sym}}^{(E)}$ has no negative eigenvalues. Otherwise, there is simply no point in calculating the spectral information via this approximation.

Evaluation of accuracy. The final goal of Algorithm 3.7 with $\mathcal{A}^{(E)}$ is to obtain "accurate" approximate eigenpairs. Two separate notions of accuracy are important. In the following, let λ_i, $\boldsymbol{u}_i$ be the i-th exact eigenpair of $\mathcal{L}_{\text{sym}}$; let $\lambda_i^{(E)}$, $\boldsymbol{u}_i^{(E)}$ be the i-th exact eigenpair of $\mathcal{L}_{\text{sym}}^{(E)}$; and let $\tilde{\lambda}_i^{(E)}, \tilde{\boldsymbol{u}}_i^{(E)}$ be the i-th approximate eigenpair as computed by Algorithm 3.7.

The *eigenvalue accuracy* is given by

$$|\lambda_i - \tilde{\lambda}_i^{(E)}| \leq |\lambda_i - \lambda_i^{(E)}| + |\lambda_i^{(E)} - \tilde{\lambda}_i^{(E)}|. \tag{3.47}$$

Here the first term can be estimated by the Bauer-Fike theorem (see, e.g., [40]): For any p-norm ($p \in [1, \infty]$), it holds

$$|\lambda_i - \lambda_i^{(E)}| \leq \|U\|_p \|U^{-1}\|_p \|\mathcal{L}_{\text{sym}} - \mathcal{L}_{\text{sym}}^{(E)}\|_p$$

with the exact eigenvector matrix U of $\mathcal{L}_{\text{sym}}$ as in Definition 2.13. Since U is orthogonal, the two first terms are 1 for $p = 2$. Generally, we can obtain error estimates of the form

$$|\lambda_i - \lambda_i^{(E)}| \leq \|\mathcal{L}_{\text{sym}} - \mathcal{L}_{\text{sym}}^{(E)}\|_2 \leq \sqrt{n} \|\mathcal{L}_{\text{sym}} - \mathcal{L}_{\text{sym}}^{(E)}\|_\infty.$$

The second part of (3.47), $|\lambda_i^{(E)} - \tilde{\lambda}_i^{(E)}|$, represents the error of the Krylov method. For the Lanczos method, this part can be estimated using the Kaniel-Paige theory (Theorem 3.13), yielding exponential convergence to zero for increasing subspace dimension. The error of the Krylov-Schur method empirically behaves similarly, as it can be seen as an extension of the Lanczos procedure.

A second notion of accuracy is the *residual norm*

$$\|\mathcal{L}_{\text{sym}} \tilde{\boldsymbol{u}}_i^{(E)} - \tilde{\lambda}_i^{(E)} \tilde{\boldsymbol{u}}_i^{(E)}\|_2 \leq \|(\mathcal{L}_{\text{sym}} - \mathcal{L}_{\text{sym}}^{(E)}) \tilde{\boldsymbol{u}}_i^{(E)}\|_2 + \|\mathcal{L}_{\text{sym}}^{(E)} \tilde{\boldsymbol{u}}_i^{(E)} - \tilde{\lambda}_i^{(E)} \tilde{\boldsymbol{u}}_i^{(E)}\|_2. \tag{3.48}$$

Here the first part is estimated as $\leq \|\mathcal{L}_{\text{sym}} - \mathcal{L}_{\text{sym}}^{(E)}\|_2$ because of $\|\tilde{\boldsymbol{u}}_i^{(E)}\|_2 = 1$. The second part is the residual norm that can be estimated a posteriori in Krylov methods via Equation (3.38), and can be controlled below a given tolerance in the Krylov-Schur method (Algorithms 3.6 or 3.7).

Error propagation onto $\mathcal{L}_{\text{sym}}$. With both accuracy measures, an important contributing factor is the post-normalization matrix distance

$$\|\mathcal{L}_{\text{sym}} - \mathcal{L}_{\text{sym}}^{(E)}\|_2 = \|\mathcal{A}_{\text{sym}} - \mathcal{A}_{\text{sym}}^{(E)}\|_2$$

with $\mathcal{A}_{\text{sym}}$ and $\mathcal{A}_{\text{sym}}^{(E)}$ as in 3.44 and 3.46. The norm depends nonlinearly on the approximation quality $\|E\|$, as it causes errors in $\mathcal{A}^{(E)}$ as well as in $(\mathcal{D}^{(E)})^{-1/2}$. This nonlinear error propagation can be quantified best in the ∞-norm, as seen in the following lemma contributed by the author and published in [2]. For estimates in the 2-norm, the best we can do is use the well-known equivalence estimate $\|M\|_2 \leq \sqrt{n}\|M\|_\infty$ that holds for any matrix $M \in \mathbb{R}^{n\times n}$.

3.16 Lemma. *(Error propagation for the normalized Laplacian)*
Let $\mathcal{A} \in \mathbb{R}^{n\times n}$ be a matrix with non-negative entries and at least one positive entry per row. Given an error matrix $E \in \mathbb{R}^{n\times n}$, we define $\mathcal{A}^{(E)} = \mathcal{A} + E$ and the exact and approximate version of the degrees and normalized adjacency matrix as

$$\begin{aligned} [d_1,\dots,d_n]^T &= \mathcal{A}\mathbf{1}, & [d_1^{(E)},\dots,d_n^{(E)}]^T &= \mathcal{A}^{(E)}\mathbf{1},\\ \mathcal{D} &= \operatorname{diag}(d_1,\dots,d_n), & \mathcal{D}^{(E)} &= \operatorname{diag}(d_1^{(E)},\dots,d_n^{(E)}),\\ \mathcal{A}_{\text{sym}} &= \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}, & \mathcal{A}_{\text{sym}}^{(E)} &= (\mathcal{D}^{(E)})^{-1/2}\mathcal{A}^{(E)}(\mathcal{D}^{(E)})^{-1/2}. \end{aligned}$$

Let $d_{\min} = \min_i d_i > 0$ denote the minimum diagonal entry of $\mathcal{D}$. If $\|E\|_\infty < d_{\min}$, then it holds

$$\|\mathcal{A}_{\text{sym}} - \mathcal{A}_{\text{sym}}^{(E)}\|_\infty \leq \frac{\|E\|_\infty(\|\mathcal{A}\|_\infty + d_{\min})}{d_{\min}(d_{\min} - \|E\|_\infty)}.$$

Proof. Let us set the auxiliary variables

$$\eta := \frac{d_{\min}}{\|\mathcal{A}\|_\infty} \qquad \text{and} \qquad \varepsilon := \frac{\|E\|_\infty}{\|\mathcal{A}\|_\infty}.$$

First, we easily see that the individual degree errors are bounded by $\|E\|_\infty$:

$$|d_i - d_i^{(E)}| \leq \|\mathcal{A}\mathbf{1} - \mathcal{A}^{(E)}\mathbf{1}\|_\infty = \|E\mathbf{1}\|_\infty \leq \|E\|_\infty\|\mathbf{1}\|_\infty = \|E\|_\infty = \varepsilon\|\mathcal{A}\|_\infty.$$

With this, we can write the infinity-norm error of $\mathcal{D}^{-1/2}$ as

$$\begin{aligned} \|\mathcal{D}^{-1/2} - (\mathcal{D}^{(E)})^{-1/2}\|_\infty &= \max_i \left|d_i^{-1/2} - (d_i^{(E)})^{-1/2}\right| \\ &\leq \max_i \max_{-\|E\|_\infty \leq \delta \leq \|E\|_\infty} \left|d_i^{-1/2} - (d_i+\delta)^{-1/2}\right|. \end{aligned}$$

Due to the shape of $x \mapsto x^{-1/2}$, the inner maximum is attained at the left-most interval bound and the outer maximum is attained where d_i is minimal:

$$\begin{aligned} \|\mathcal{D}^{-1/2} - (\mathcal{D}^{(E)})^{-1/2}\|_\infty &\leq \max_i \left|d_i^{-1/2} - (d_i - \|E\|_\infty)^{-1/2}\right| \\ &= \left|d_{\min}^{-1/2} - (d_{\min} - \|E\|_\infty)^{-1/2}\right| \\ &= \left|\eta^{-1/2} - (\eta-\varepsilon)^{-1/2}\right| \, \|\mathcal{A}\|_\infty^{-1/2} \\ &= \Big((\eta-\varepsilon)^{-1/2} - \eta^{-1/2}\Big)\|\mathcal{A}\|_\infty^{-1/2}. \end{aligned}$$

Analogously we obtain

$$\|(\mathcal{D}^{(E)})^{-1/2}\|_\infty \leq (\eta-\varepsilon)^{-1/2}\|\mathcal{A}\|_\infty^{-1/2}.$$

Together with

$$\|\mathcal{D}^{-1/2}\|_\infty = d_{\min}^{-1/2} = \eta^{-1/2}\|\mathcal{A}\|_\infty^{-1/2}$$

and

$$\|\mathcal{A}^{(E)}\|_\infty \leq \|\mathcal{A}\|_\infty + \|E\|_\infty = (1+\varepsilon)\|\mathcal{A}\|_\infty,$$

this yields

$$\begin{aligned}
\|\mathcal{A}_{\text{sym}} - \mathcal{A}_{\text{sym}}^{(E)}\|_\infty &= \Big\| (\mathcal{D}^{-1/2} - (\mathcal{D}^{(E)})^{-1/2})\mathcal{A}\mathcal{D}^{-1/2} \\
&\qquad + (\mathcal{D}^{(E)})^{-1/2}(\mathcal{A}-\mathcal{A}^{(E)})\mathcal{D}^{-1/2} \\
&\qquad\quad + (\mathcal{D}^{(E)})^{-1/2}\mathcal{A}^{(E)}(\mathcal{D}^{-1/2} - (\mathcal{D}^{(E)})^{-1/2})\Big\|_\infty \\
&\leq \Big((\eta-\varepsilon)^{-1/2} - \eta^{-1/2}\Big)\|\mathcal{A}\|_\infty^{-1/2}\ \|\mathcal{A}\|_\infty\ \eta^{-1/2}\|\mathcal{A}\|_\infty^{-1/2} \\
&\qquad + (\eta-\varepsilon)^{-1/2}\|\mathcal{A}\|_\infty^{-1/2}\ \varepsilon\|\mathcal{A}\|_\infty\ \eta^{-1/2}\|\mathcal{A}\|_\infty^{-1/2} \\
&\qquad\quad + \Big((\eta-\varepsilon)^{-1/2}\|\mathcal{A}\|_\infty^{-1/2}\ (1+\varepsilon)\|\mathcal{A}\|_\infty \\
&\qquad\qquad \cdot \Big((\eta-\varepsilon)^{-1/2} - \eta^{-1/2}\Big)\|\mathcal{A}\|_\infty^{-1/2}\Big) \\
&= \Big((\eta-\varepsilon)^{-1/2} - \eta^{-1/2}\Big)\Big(\eta^{-1/2} + (1+\underline{\varepsilon)(\eta-\varepsilon)^{-1/2}}\Big) \\
&\qquad + \underline{(\eta-\varepsilon)^{-1/2}\varepsilon}\eta^{-1/2}.
\end{aligned}$$

Now detach the upper underlined part from its paranthesed expression and combine it with the lower underlined part:

$$\begin{aligned}
&= \Big((\eta-\varepsilon)^{-1/2} - \eta^{-1/2}\Big)\Big((\eta-\varepsilon)^{-1/2} + \eta^{-1/2}\Big) \\
&\qquad + \underline{\varepsilon(\eta-\varepsilon)^{-1/2}}\Big((\eta-\varepsilon)^{-1/2} - \eta^{-1/2} + \eta^{-1/2}\Big)
\end{aligned}$$

Resolve the binomial expression in the first line and simplify the second line:

$$\begin{aligned}
&= (\eta-\varepsilon)^{-1} - \eta^{-1} + \varepsilon(\eta-\varepsilon)^{-1} = \frac{\eta - (\eta-\varepsilon) + \varepsilon\eta}{\eta(\eta-\varepsilon)} \\
&= \frac{\varepsilon(1+\eta)}{\eta(\eta-\varepsilon)} = \frac{\|E\|_\infty(\|\mathcal{A}\|_\infty + d_{\min})}{d_{\min}(d_{\min} - \|E\|_\infty)}.
\end{aligned}$$

This concludes the proof for the desired inequality. □

The requirement $\|E\|_\infty < d_{\min}$ cannot be avoided since otherwise, negative entries in $\mathcal{D}^{(E)}$ could not be ruled out, invalidating $(\mathcal{D}^{(E)})^{-1/2}$ and $\mathcal{A}^{(E)}$ with it. On the other hand, in the regime where $\|E\|_\infty$ is well below $d_{\min}$, the absolute error in $\mathcal{A}$ can be estimated proportional to $\|E\|_\infty$. For example, if $\|E\|_\infty \leq \frac{d_{\min}}{2}$, Lemma 3.16 yields

$$\|\mathcal{A}_{\text{sym}} - \mathcal{A}_{\text{sym}}^{(E)}\|_\infty \leq 2\frac{\|\mathcal{A}\|_\infty + d_{\min}}{d_{\min}^2}\|E\|_\infty.$$

NFFT-based error for scaling data sets. For the NFFT-based approximation, Equation (3.35) shows that the perturbation $\|E\|_\infty$ grows linearly in the size of the data set. If the minimum degree $d_{\min}$ grew less fast, then such an approximation would be practically useless for large n because $\|E\|_\infty$ would eventually supersede $d_{\min}$.

However, if we assume that increasing n means adding more similarly-distributed data points to the data set, the *average* entry in $\mathcal{A}$ does not change and thus all row sums of $\mathcal{A}$ are proportional to n, including $d_{\min}$ and the maximum row sum $\|\mathcal{A}\|_\infty$. In that case, $\|\mathcal{A}_{\text{sym}} - \mathcal{A}_{\text{sym}}^{(E)}\|_\infty$ is bounded independent of n.

Particular problems arise in two cases. First, suppose increasing n also increases the probability for severe outliers, for example due to an underlying normal distribution. Outliers are the points with minimum degrees, so in this case, $d_{\min}$ might not grow with n. Then, for large n there will eventually be outliers with approximated degrees $d_i^{(E)} < 0$, rendering the NFFT-based approximation impractical unless some sort of outlier detection is employed.

Second, if the points are distributed uniformly on a domain that grows with n, then intuitively the node degrees will also stay constant. For the NFFT-based approximation to be applicable, one remedy in practice is to have the localization of the kernel be adjusted to the domain size. For the example of the Gaussian kernel from (3.3), choosing the parameter σ proportional to the data set diameter might be enough to counteract the scaling.

In Figure 3.4, we test how the different error components behave in practice. For our data, we use the fixed Oakland1 data set with $n = 36\,932$ nodes that was introduced in Example 2.19. We evaluate the approximation of the Gaussian adjacency matrix using the NFFT parameter setup #2 from Example 3.18. For the localization parameters σ, we use seven different values between 1 and 100.

The norm $\|\mathcal{A}\|_\infty$, which is equal to the maximum node degree, grows monotonically with σ until it plateaus because it cannot surpass $n - 1$. The minimum node degree $d_{\min}$ behaves similarly, but starting at around 1 for $\sigma = 1$. At $\sigma = 100$, $d_{\min}$ arrives close to $\|\mathcal{A}\|_\infty$, which means that all node degrees are in the same order of magnitude because all edge weights are close to 1.

We furthermore compute the exact error norm $\|E\|_\infty$ by iterating over all standard basis vectors $e_i \in \mathbb{R}^n$, whose only nonzero entry is a 1 at index i. The expression

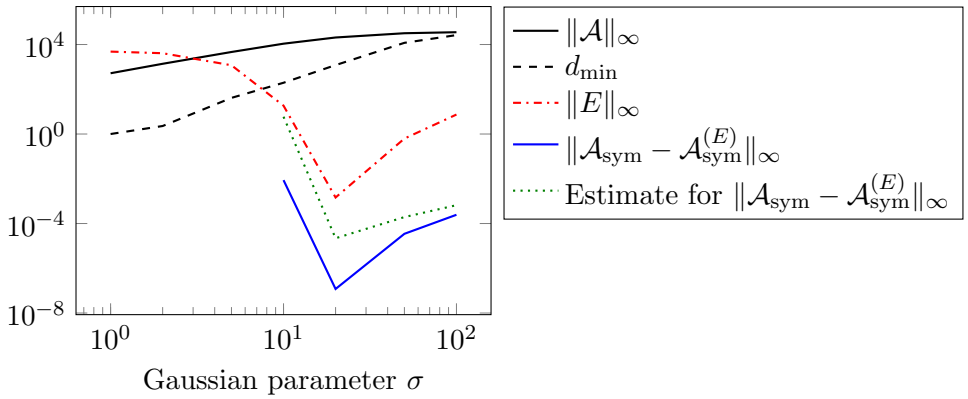


Figure 3.4: Empirical measurements of various error components for the graph Laplacian of the "Oakland1" data set from Example 2.19 for different parameter values in the Gaussian kernel function.

$\|\mathcal{A}e_i - \mathcal{A}^{(E)}e_i\|_1$ equals the absolute i-th row sum, so its maximum over i yields the desired norm as $\mathcal{A}$ and $\mathcal{A}^{(E)}$ are symmetric. We see that $\|E\|_\infty$ starts off comparably large, but then decreases with σ until its minimum at around $\sigma = 20$. For larger σ, the absolute error starts to increase significantly once more. This might be because the kernel is easier to approximate by the trigonometric function for medium σ. Even increasing the regularization width ε_B does not improve the error for large σ.

For $\sigma \geq 10$, our experiments yield $\|E\|_\infty < d_{\min}$ and Lemma 3.16 is applicable. This is identical to the parameter range where all approximate degrees are positive and $\mathcal{A}^{(E)}_{\text{sym}}$ can be evaluated. The results confirm the upper bound for the true error $\|\mathcal{A}_{\text{sym}} - \mathcal{A}^{(E)}_{\text{sym}}\|_\infty$. The tightness of the estimate is quite low for $\sigma = 10$, as the bound overestimates the true error by a factor of almost 10^3. This ratio decreases to around 2.5 at $\sigma = 100$. The final minimum approximation error of ca. 10^{-7} is attained around $\sigma = 20$.

3.3.3 Specialized computations for hypergraphs from categorial data

Throughout Section 3.3, we have so far discussed techniques to compute the spectrum of $\mathcal{L}_{\text{sym}}$ in a way that is agnostic to the particular structure of the adjacency matrix, only assuming that there is some form of structure that allows for efficient matrix-vector products or approximations thereof. However, there is an important special case where the specific structure enables us to compute the eigenvalues in an even more efficient and specialized manner, without requiring the Krylov-Schur method and Algorithm 3.7. It occurs if the adjacency matrix $\mathcal{A}$ has low rank. Here, we discuss the example of hypergraphs with few, large hyperedges as in Example 2.20, though

other instances of low-rank adjacency matrices are conceivable. This subsection is based on [3, Section 3.2].

As stated in Section 2.2.8, a hypergraph is given by its incidence matrix $\mathcal{H} \in \{0,1\}^{n\times E}$ and optionally a diagonal weight degree matrix $\mathcal{W} \in \mathbb{R}^{E\times E}$. Just like there are various ways to analyze a hypergraph for different applications and define the notion of adjacency, there are also different definitions of the hypergraph Laplacian [14], partly even as a nonlinear operator [17].

Here we will use the linear definition introduced by [103] given as

$$\mathcal{L}_{\text{sym}} = I - \mathcal{D}^{-1/2}\mathcal{H}\mathcal{W}\mathcal{B}^{-1}\mathcal{H}^T\mathcal{D}^{-1/2}, \tag{3.49}$$

which is equal to the graph Laplacian of the dense clique expansion graph, cf. Section 2.2.8 and Example 2.20. As a consequence of the density, naive computations with $\mathcal{L}_{\text{sym}}$ may become very expensive.

One important application of hypergraphs is categorial data where each sample is described by a few discrete attributes as in Example 2.20. We can simply construct one hyperedge for each possible value of an attribute, connecting all the samples which share that particular attribute value. This leads to the number of hyperedges being significantly smaller than the number of samples, $|E| \ll n$. For other automatically generated hypergraphs, there is precedence for the benefits of generating fewer, larger hyperedges as well [83].

In this special case, the Laplacian definition directly exhibits a useful structure. The matrix subtracted from the identity in (3.49) has rank $|E|$, so $\mathcal{L}_{\text{sym}}$ is a linear combination of the identity and a low-rank matrix and can be written as

$$\mathcal{L}_{\text{sym}} = I - \tilde{\mathcal{H}}\tilde{\mathcal{H}}^T \quad \text{with} \quad \tilde{\mathcal{H}} = \mathcal{D}^{-1/2}\mathcal{H}\mathcal{W}^{1/2}\mathcal{B}^{-1/2} \in \mathbb{R}^{n\times E}. \tag{3.50}$$

3.17 Proposition. *(Eigenvalue decomposition of hypergraphs with few hyperedges)* For a given hypergraph with $|E| \ll n$, let the *thin* singular value decomposition of the low-rank factor $\tilde{\mathcal{H}}$ from (3.50) be given by

$$\tilde{\mathcal{H}} = \tilde{U}\tilde{\Sigma}\tilde{V}^T, \tag{3.51}$$

where $\tilde{U} \in \mathbb{R}^{n\times|E|}$ and $\tilde{V} \in \mathbb{R}^{n\times|E|}$ have orthogonal columns and $\tilde{\Sigma} \in \mathbb{R}^{|E|\times|E|}$ holds the singular values on its diagonal in decreasing order. If $\tilde{\mathcal{H}}$ does not have full rank, $\tilde{\Sigma}$ contains trailing zeros.
The eigenvalue decomposition of $\mathcal{L}_{\text{sym}}$ in the sense of Definition 2.13 is given by

$$\mathcal{L}_{\text{sym}} = U\Lambda U \quad \text{with} \quad U = \begin{bmatrix}\tilde{U} & \hat{U}\end{bmatrix} \quad \text{and} \quad \Lambda = \begin{bmatrix} I - \tilde{\Sigma}^2 & \\ & I \end{bmatrix},$$

where the columns of $\hat{U} \in \mathbb{R}^{n\times(n-|E|)}$ are chosen arbitrarily to extend the orthonormal columns of $\tilde{U}$ to a full orthonormal basis of $\mathbb{R}^n$.
In other words, $n - |E|$ eigenvalues of $\mathcal{L}_{\text{sym}}$ are 1 and the remaining r eigenvalues are given by $\tilde{\Lambda} = I - \tilde{\Sigma}^2$.

Proof. Because of the Laplacian structure

$$\mathcal{L}_{\text{sym}} = I - \tilde{\mathcal{H}}\tilde{\mathcal{H}}^T = I - \tilde{U}\tilde{\Sigma}\tilde{V}^T\tilde{V}\tilde{\Sigma}\tilde{U}^T = I - \tilde{U}\tilde{\Sigma}^2\tilde{U}^T,$$

the orthogonal matrix U can be identified as the eigenvector matrix via

$$\mathcal{L}_{\text{sym}}U = U - \tilde{U}\tilde{\Sigma}^2\tilde{U}^T\begin{bmatrix}\tilde{U} & \hat{U}\end{bmatrix} = U - \begin{bmatrix}\tilde{U}\tilde{\Sigma}^2 & 0\end{bmatrix} = U - U\begin{bmatrix}\tilde{\Sigma}^2 & 0\\ 0 & 0\end{bmatrix} = U\Lambda. \qquad \square$$

In most cases, $\tilde{\mathcal{H}}$ will have full rank $|E|$, so the multiplicity of the eigenvalue 1 is exactly $n - |E|$ and the remaining eigenvalues are distinctly smaller than 1. The exception is the case of an hyperedge that does not add new information, e.g., that is just the union of other, smaller hyperedges. Then one of the $|E|$ singular values of $\tilde{\mathcal{H}}$ is zero, which leads to an additional eigenvector for the eigenvalue 1.

The SVD of the low-rank factor. The computation of the SVD (3.51) can be tackled in various ways, e.g., via the eigenvalue decomposition of

$$\tilde{\mathcal{H}}\tilde{\mathcal{H}}^T \in \mathbb{R}^{n\times n} \quad \text{or} \quad \begin{bmatrix}0 & \tilde{\mathcal{H}}^T\\ \tilde{\mathcal{H}} & 0\end{bmatrix} \in \mathbb{R}^{(n+|E|)\times(n+|E|)}.$$

(cf. [40]). In our case of $|E| \ll n$, however, the most efficient approach is based on the eigenvalue decomposition

$$\tilde{\mathcal{H}}^T\tilde{\mathcal{H}} = \mathcal{B}^{-1/2}\mathcal{W}^{1/2}\mathcal{H}^T\mathcal{D}^{-1}\mathcal{H}\mathcal{W}^{1/2}\mathcal{B}^{-1/2} = \tilde{V}\tilde{\Sigma}^2\tilde{V}^T \in \mathbb{R}^{|E|\times|E|},$$

which directly reveals $\tilde{\Sigma}$ and the right singular vector matrix $\tilde{V}$ from (3.51). The corresponding left singular vectors can be recovered via $\tilde{U} = \tilde{\mathcal{H}}\tilde{V}\tilde{\Sigma}^{-1}$.

Because the problem size is now only $|E|\times|E|$, this problem can be solved efficiently.

Loop weights. The clique expansion graph of any hypergraph will contain loops whose weights are given by the diagonal entries a_{ii} of $\mathcal{A} = \mathcal{H}\mathcal{W}\mathcal{B}^{-1}\mathcal{H}^T$ via

$$a_{ii} = \sum_{e\in E} h_{ie}^2\frac{|e|}{w_e} = \sum_{e\in E:\ i\in e}\frac{|e|}{w_e}.$$

Sometimes in graph learning it can be beneficial to adjust an adjacency matrix by uniformly adding or removing loops or changing the loop weights, i.e., replacing values

on the diagonal of $\mathcal{A}$. If this is done for hypergraphs, the low-rank structure of (3.50) will be destroyed and the eigenvalues of $\mathcal{L}_{\text{sym}}$ will have no connection to the SVD of any matrix like $\tilde{\mathcal{H}}$. Specifically, there might be n different eigenvalues instead of an eigenvalue 1 with multiplicity $n - |E|$.

Hence there will no be longer any special structure to be exploited and the eigenvalues may need to be computed via the standard Krylov-Schur method. This needs to be kept in mind when discussing loop weights for hypergraphs.

3.4 Alternative eigenvalue algorithm: The Nyström method

As a baseline for the upcoming numerical experiments, we will use the Nyström extension that is used extensively in practice for the speedup of certain kernel methods.[5] This section is taken from [2] with slight adjustments.

Once again, the goal is to approximate kernel-based fully connected adjacency matrices as in (3.4), $\mathcal{A} = \mathcal{K}_{\mathcal{X}\mathcal{X}} + \kappa_0 I$ for a point set $\mathcal{X} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$. Though the method can also be applied to rectangular matrices with different source and target point sets, $\mathcal{Y} \neq \mathcal{X}$, we will only cover the symmetric case here.

Compared to the NFFT approach, the Nyström method will result in much rougher approximations, which will cause issues with the inexact degree computation. Its advantage is that the complexity only scales linearly in d, so it will also be applicable in higher-dimensional feature spaces.

3.4.1 The traditional Nyström extension

The Nyström extension is often used as a method of choice for numerical approximation of kernel-based matrices that are too large to allow for direct eigenvalue computation. Originally introduced in the matrix computations context in [100], further improvements have been suggested in [36] and [26] and its usage for classification problems has been proposed in [10]. See, e.g., [37] and [70] for its applications in different settings.

The Nyström method is based on dividing the data points into a sample set $\mathcal{X}_1$ of m nodes and its complement $\mathcal{X}_2$. After permutation, the adjacency matrix $\mathcal{A}$ can be split into blocks

$$\mathcal{A} = \begin{bmatrix} \mathcal{A}_{11} & \mathcal{A}_{12} \\ \mathcal{A}_{12}^T & \mathcal{A}_{22} \end{bmatrix},$$

where the blocks $\mathcal{A}_{11} \in \mathbb{R}^{m \times m}$ and $\mathcal{A}_{22} \in \mathbb{R}^{(n-m)\times(n-m)}$ are the adjacency matrices of the canonical subgraphs with node sets $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively, and the block $\mathcal{A}_{12} \in \mathbb{R}^{m\times(n-m)}$ contains the similarities between all combinations of nodes from $\mathcal{X}_1$ and $\mathcal{X}_2$.

[5] See, e.g., `https://scikit-learn.org/stable/modules/kernel_approximation.html`.

The basic idea of the Nyström method is to compute only $\mathcal{A}_{11}$ and $\mathcal{A}_{12}$ explicitly, but not the remaining block $\mathcal{A}_{22}$. If $m \ll n$, the approach significantly decreases the required number of data point comparisons. Assuming that $\mathcal{A}_{11}$ is regular, the method approximates $\mathcal{A}$ by

$$\mathcal{A} \approx \mathcal{A}^{(E)} = \begin{bmatrix} \mathcal{A}_{11} \\ \mathcal{A}_{12}^T \end{bmatrix} \mathcal{A}_{11}^{-1} \begin{bmatrix} \mathcal{A}_{11} & \mathcal{A}_{12} \end{bmatrix} = \begin{bmatrix} \mathcal{A}_{11} & \mathcal{A}_{12} \\ \mathcal{A}_{12}^T & \mathcal{A}_{12}^T \mathcal{A}_{11}^{-1} \mathcal{A}_{12} \end{bmatrix}, \tag{3.52}$$

which constitutes a rank-m approximation due to the size and regularity of $\mathcal{A}_{11}$. The complexity is $\mathcal{O}(n\,m\,d)$ for the setup of $\mathcal{A}_{11}$ and $\mathcal{A}_{12}$ and then $\mathcal{O}(n\,m + m^2)$ for each matrix-vector product with $\mathcal{A}^{(E)}$.

This formula is used once in approximating the degree matrix $\mathcal{D}$ by $\mathcal{D}^{(E)} = \operatorname{diag}(\mathcal{A}^{(E)}\mathbf{1})$ and once in approximating the eigenvalues of $\mathcal{A}_{\text{sym}}$ via the rank-m eigenvalue decomposition

$$\mathcal{A}_{\text{sym}}^{(E)} := (\mathcal{D}^{(E)})^{-1/2}\, \mathcal{A}^{(E)}\, (\mathcal{D}^{(E)})^{-1/2} = V_m \Lambda V_m^T.$$

This can be computed without having to set up the full matrix, e.g., by the technique described in [36] made up mainly of two singular value decompositions of $(m \times m)$-sized matrices.

Alternatively, we have achieved better results by computing the QR factorization $\hat{Q}\hat{R} = (\mathcal{D}^{(E)})^{-1/2}[\mathcal{A}_{11}\ \mathcal{A}_{12}]^T$ and the eigenvalue decomposition $U_m \Lambda_m U_m^T = \hat{R}\mathcal{A}_{11}^{-1}\hat{R}^T$, leading to

$$\mathcal{A}_{\text{sym}}^{(E)} = \hat{Q}\,\hat{R}\,\mathcal{A}_{11}^{-1}\,\hat{R}^T\,\hat{Q}^T = \hat{Q}\,U_m\,\Lambda_m\,U_m^T\,\hat{Q}^T = V_m\,\Lambda_m\,V_m^T$$

with the eigenvector matrix $V_m = \hat{Q}U_m$. The arithmetic complexity of these steps is $\mathcal{O}(n\,m^2)$ for the QR factorization and $\mathcal{O}(m^3)$ for the reduced eigenvalue decomposition, yielding $\mathcal{O}(n\,m^2)$ in total.

The eigenvalue accuracy depends strongly on the quality of the approximation

$$\mathcal{A}_{22} \approx \mathcal{A}_{12}^T \mathcal{A}_{11}^{-1} \mathcal{A}_{12}.$$

Since the sample set $\mathcal{X}_1$ is a randomly chosen subset of the indices from $1, \ldots, n$, its size m is the decisive method parameter and its choice is a nontrivial task. On the one hand, m needs to be small for the method to be efficient. On the other hand, too small a choice of m may cause extreme errors, especially because the approximation error in $\mathcal{D}^{(E)}$ propagates to the eigenvalue computation the same way as stated in Lemma 3.16. However, we do not have a meaningful bound on $\|E\|_\infty = \|\mathcal{A}_{22} - \mathcal{A}_{12}^T \mathcal{A}_{11}^{-1} \mathcal{A}_{12}\|_\infty$ that would guarantee favorable error behaviour, even for fairly large m.

3.4.2 NFFT-based accelerated Nyström

An additional contribution of [2] is the development of an improved Nyström method that utilizes the NFFT-based fast summation from Section 3.1. As opposed to the traditional Nyström extension, this will again only be applicable for small d. It is based on a slightly different algorithm that has been recently introduced as a randomized Nyström method, cf. [67] and the references therein. Their basic idea is rewriting the traditional Nyström approximation as

$$\mathcal{A} \approx (\mathcal{A}Q)(Q^T\mathcal{A}Q)^{-1}(\mathcal{A}Q)^T$$

where $Q \in \mathbb{R}^{n\times L}$ is a matrix with orthogonal columns. If Q holds the first m columns of a permutation matrix, one obtains the traditional Nyström method. Inspired by similar randomized linear algebra algorithms such as randomized singular value decomposition, this choice of Q is replaced in [67] by $Q = \mathrm{orth}(\mathcal{A}G)$, where $G \in \mathbb{R}^{n\times m}$ is a random matrix with normally distributed entries and orth denotes column-wise orthonormalization. Unfortunately, this setup requires $2m$ matrix-vector products with the full matrix $\mathcal{A}$.

We now propose accelerating these matrix-vector products by computing $\mathcal{A}G$ and $\mathcal{A}Q$ column-wise via the NFFT-based fast summation Algorithm 3.2 in order to avoid full matrix setup or slow direct matrix-vector products. In addition, we propose replacing the inverse $(Q^T\mathcal{A}Q)^{-1}$ by a low-rank approximation based only on the $p \in \mathbb{N}$ largest eigenvalues of $Q^T\mathcal{A}Q$. This way, a rank-p approximation of $\mathcal{A}$ is produced, where p may be the actual number of required eigenvalues or larger.

The resulting method "Nyström-Gaussian-NFFT" is presented in Algorithm 3.8. Its arithmetic complexity is $\mathcal{O}\big(n\,m^2\big)$. On first glance, this arithmetic complexity seems to be identical to the one of the traditional Nyström method. However, as we observe in the numerical tests in Section 3.5.1, we may choose the parameter m distinctly smaller for the hybrid method, i.e., $m \sim k$, where k is the number of eigenvalues and eigenvectors. Then, the resulting arithmetic complexity is $\mathcal{O}\big(n\,k^2\big)$.

3.5 Numerical evaluations

We now test the practical performance of the presented framework, both by evaluating the standalone eigenvalue accuracy and by applying the results in a few accomplished learning methods. All our experiments are performed using MATLAB implementations based on the NFFT3 library and MATLAB's `eigs` function. A short example code was published with our original paper [2].[6] Recently, we have also released the

[6] `https://www.tu-chemnitz.de/mathematik/wire/people/files_alfke/NFFT-Lanczos-Example-v1.tar.gz`

Algorithm 3.8 NFFT-based accelerated Nyström-Gaussian method for Laplacian eigenvalue approximation

Input: Point set $\mathcal{X} \subset \mathbb{R}^d$ with $|\mathcal{X}| = n$, kernel K, number of eigenvalues k, number of random Gaussian columns $m \geq k$, rank of inversion p with $k \leq p \leq m$

Output: Approximated eigenpairs $\lambda_i, \boldsymbol{u}_i$ with $\mathcal{L}_{\text{sym}}\boldsymbol{u}_i \approx \lambda_i \boldsymbol{u}_i$

1: Choose parameters $N, M, m, \varepsilon_B, p$ for the NFFT-based fast summation, cf. Section 3.1
2: Compute the approximate egree matrix $\mathcal{D} = \operatorname{diag}(\mathcal{A}\mathbf{1})$ using Algorithm 3.2
3: Generate a random matrix $G \in \mathbb{R}^{n \times m}$ with i.i.d. entries from a normal distribution with mean 0 and standard deviation 1
4: Compute $Y = \mathcal{A}G$ column-wise via Algorithm 3.2
5: Compute an orthonormal basis matrix Q from the columns of Y via a QR factorization
6: Compute $B_1 = AQ \in \mathbb{R}^{n \times m}$ column-wise via Algorithm 3.2 and $B_2 = Q^T B_1 \in \mathbb{R}^{m \times m}$
7: Compute the diagonal matrix Σ_p holding the p largest positive eigenvalues of B_2 and the corresponding eigenvector matrix $U_p \in \mathbb{R}^{m \times p}$
8: Compute the QR factorization $\hat{Q}\hat{R} = B_1 U_p$
9: Compute the diagonal matrix Λ_k holding the k largest eigenvalues of $\hat{R}\Sigma_p^{-1}\hat{R}^T$, the corresponding eigenvector matrix $\hat{U}_k$, and $V_k = \hat{Q}\hat{U}_k$

Python package `fastadj`[7] providing a dedicated interface to the `C` implementation of NFFT3.

The results in this section were published in [2], together with additional experiments that showcase the possibility to use the NFFT-based approximation in other setups, e.g., in combination with the Conjugate Gradient method or with different kernels. All experiments are run on a computer with Intel Core i7 CPU 970 (3.20 GHz) using one thread. Although the NFFT3 library supports OpenMP (cf. [97]), our experiments are restricted to a single thread to allow better comparison.

Throughout this section, we will compare different setups of the NFFT-based fast summation method, Algorithm 3.2. The required parameters are the bandwidth N, the window cutoff parameter m, and the regularization width ε_B. If $\varepsilon_B > 0$, the regularization smoothness p is also required. For the oversampled bandwidth, we always use $M = 2N$. If the approximation is used for eigenvalue computation as in Algorithm 3.7, then the Krylov-Schur tolerance is also required and should be in line with the approximation quality of the fast summation (cf. Section 3.3.2). We always use a maximum iteration count of 20 and a subspace dimension of $m = \max(2k, 20)$.

3.18 Example. *(NFFT & fast summation parameters)*
We test the following three parameter sets:

- Setup #1: $N = 16$, $m = 2$, $\varepsilon_B = 0$, Krylov-Schur tolerance $\varepsilon = 10^{-4}$
- Setup #2: $N = 32$, $m = 4$, $\varepsilon_B = 0$, Krylov-Schur tolerance $\varepsilon = 10^{-8}$
- Setup #3: $N = 64$, $m = 7$, $\varepsilon_B = 0$, Krylov-Schur tolerance $\varepsilon = 10^{-14}$

These setups are implemented in our `fastadj` python package under the preset names `'rough'`, `'default'`, and `'fine'`.

In our experiments, we did not see any empirical benefit in utilizing a positive regularization width $\varepsilon_B > 0$. Note that this does not mean that the kernel is not regularized at all; for $\|\boldsymbol{z}\|_2 > \frac{1}{2}$, $K_R(\boldsymbol{z})$ is set to $K(\frac{1}{2})$ (cf. Figure 3.3).

To investigate scalability with increasing n, we use a simple artificial point cloud data set that can be used as an academic example.

3.19 Example. *(Spiral data set)*
To obtain a scalable 3D point cloud with arbitrary node counts, we generate an artificial data set of five blobs situated along a spiral in 3D space.[8] A given number of n points are distributed equally to the five blobs, where they are generated by a normal distribution with mean value set to the blob centroid on the spiral and standard deviation $\sigma_N = 1$.

[7] `https://github.com/dominikalfke/FastAdjacency`

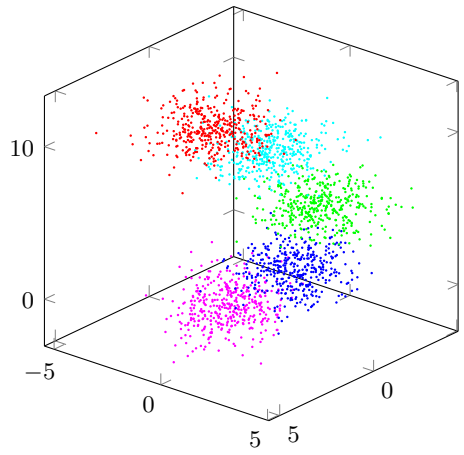


Figure 3.5: Visualization of the spiral data set from Example 3.19 with $n = 2\,000$ points

For classification tasks, each data point can be given the label of its closest centroid index, which is not necessarily the centroid of the normal distribution from which it was generated.
The data set is visualized in Figure 3.5.

3.5.1 Accuracy and runtime of eigenvalue computations

In this section, we evaluate different error measures for the computed eigenvalues of a fully-connected graph on the artificial spiral data set from Example 3.19. Since data set construction involves randomness, all reported measures are averages over 5 different data set instantiations. For the kernel function, we use the Gaussian kernel with $\sigma = 3.5$ (about a third of the largest distance between blob centroids). The goal is to compute the smallest $k = 10$ eigenvalues of the normalized graph Laplacian $\mathcal{L}_{\text{sym}}$.

We compare five different approaches:

- "Full–Krylov": Eigenvalue computation with Algorithm 3.7 for the exact adjacency matrix $\mathcal{A}$. Since $\mathcal{A}$ cannot be stored in memory for large n, we set up its entries sequentially every time a matrix-vector product is required. The tolerance for the Krylov-Schur method is set to 10^{-14}.
- "NFFT–Krylov": Eigenvalue computation with Algorithm 3.7 for $\mathcal{A}^{(E)}$ as approximated by the NFFT-based fast summation, Algorithm 3.3, with the three

[8] The data generation is based on the Matlab function `generateSpiralDataWithLabels.m` from `https://sites.google.com/site/kittipat/matlabtechniques`.

parameter setups from Example 3.18.

- "Nyström": The traditional Nyström extension from Section 3.4.1 with parameter $m \in \{n/10, n/4\}$. Due to high dependence on the random m-element subset $\mathcal{X}_1$, we conduct 10 runs for each of the 5 data set instantiations and again report the average results.

- "NFFT–Nyström": The hybrid method from Algorithm 3.8 with $m \in \{20, 50\}$ random columns, inversion rank $p = k = 10$, and fast summation parameters as in setup #2 from Example 3.18.

- "FIGTree–Krylov": Eigenvalue computation with Algorithm 3.7 with the adjacency matrix approximated by the FIGTree method from [72], a Fast Gauss Transform method (cf. Section 3.1.4). As FIGTree can reach arbitrary accuracy based on the chosen tolerance parameter ϵ, the goal of these experiments is mainly runtime comparison to the NFFT-based method. We choose $\epsilon \in \{5 \cdot 10^{-3}, 2 \cdot 10^{-6}, 10^{-10}\}$ in order to empirically match the residual norms of the three fast summation parameter setups.

We conduct tests for $n \in \{2\,000, 5\,000, 10\,000, 20\,000, 50\,000, 100\,000\}$, except for methods with infeasible runtimes for large n. Full results are given in Figure 3.6. A version of this plot with error bars can be found in [2].

Subfigure 3.6a depicts the eigenvalue accuracy in the sense of (3.47), i.e.,

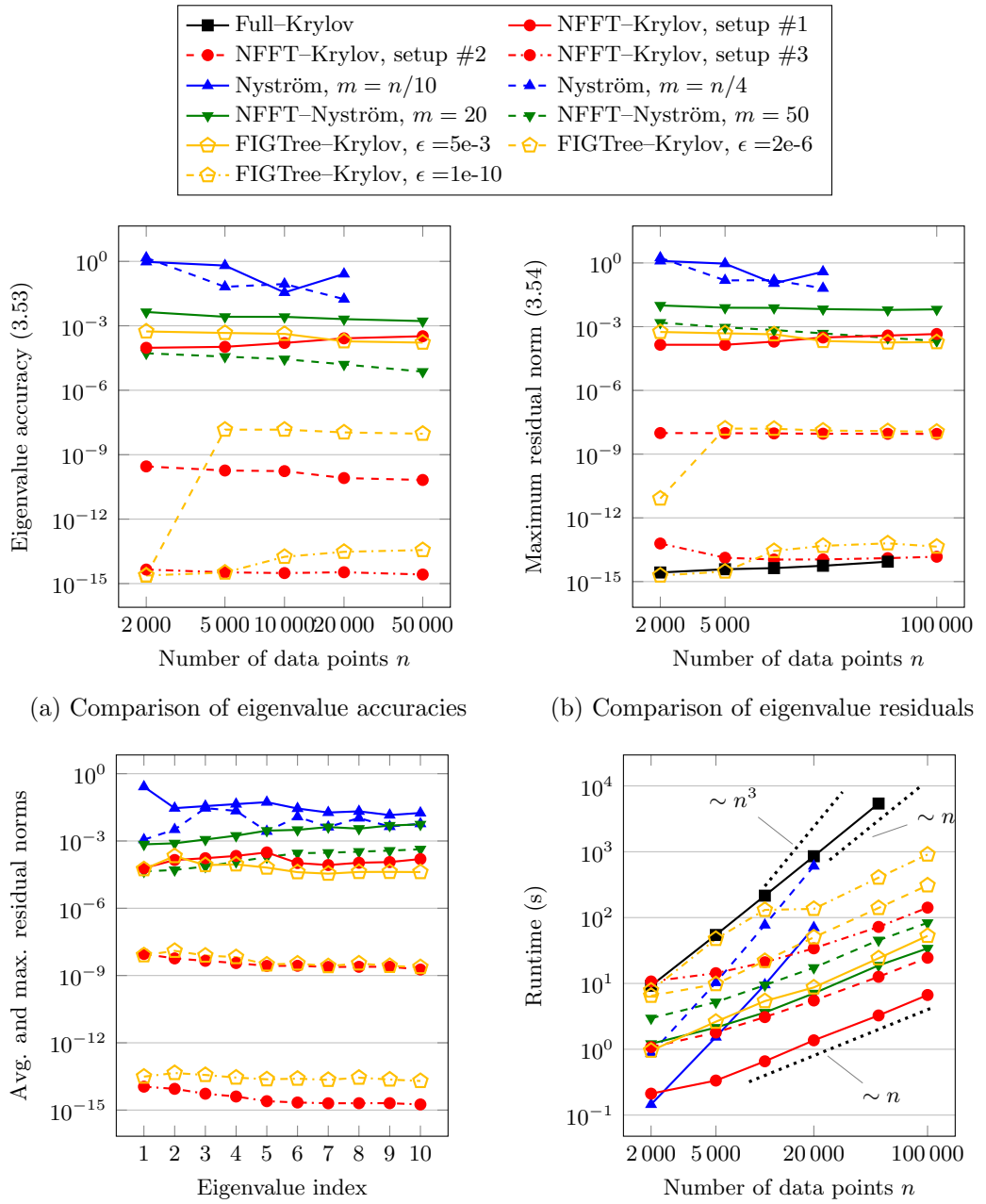
$$\max_{j=1,\ldots,k} \left| \tilde{\lambda}_j^{(E)} - \tilde{\lambda}_j \right| \tag{3.53}$$

as an average over all experiment runs. Using the notation of Section 3.3.2, $\tilde{\lambda}_j^{(E)}$ denotes the j-th eigenvalue computed by the method under consideration and $\tilde{\lambda}_j$ the one computed by the method "Full–Krylov", which we accept as the ground truth. Subfigure 3.6b depicts the maximum residual norms in the sense of (3.48), i.e., an average over the quantity

$$\max_{j=1,\ldots,k} \|\mathcal{L}_{\mathrm{sym}} \tilde{\boldsymbol{u}}_j^{(E)} - \tilde{\lambda}_j^{(E)} \tilde{\boldsymbol{u}}_j^{(E)}\|_2, \tag{3.54}$$

where $\tilde{\lambda}_j^{(E)}$, $\tilde{\boldsymbol{u}}_j^{(E)}$ is the computed eigenpair but $\mathcal{L}_{\mathrm{sym}}$ is the exact Laplacian. Subfigure 3.6c shows the same residuals, but for each individual eigenvalue index j instead of as maximum over j, for a single problem size $n = 20\,000$.

We observe that the three different setups of the NFFT–Krylov method reach accuracies and residuals of around 10^{-4}, 10^{-8}, and 10^{-13}, respectively. In particular, setup #3 yields residual norms close to machine precision and the residuals of the full method. The FIGTree parameter was successfully tuned to match this accuracy.



(a) Comparison of eigenvalue accuracies

(b) Comparison of eigenvalue residuals

(c) Residuals for $n = 20\,000$

(d) Comparison of runtimes

Figure 3.6: Comparison of accuracies and runtimes for Laplacian eigenvalue computation on the spiral data set (Example 3.19) with Gaussian adjacency matrix, $\sigma = 3.5$

The traditional Nyström method fails to achieve satisfactory accuracy, even for $m = n/4$, which is of course much too large in practice. The reason can be found in the unfavorable error propagation due to the inexact degree computation, cf. Lemma 3.16. We needed to choose m this large to make sure that the computed degrees are all positive, but as stated before, these degrees cannot even be close to zero to achieve a good approximation after normalization. The hybrid NFFT–Nyström method does not have this problem, which is why we can choose m much smaller and still yield better accuracy. However, it uses NFFT setup #2 but does not match the accuracy obtained by the combination with Krylov-Schur. Hence if the NFFT-based approximation is available, it should be used with a Krylov method instead of the Nyström-based Algorithm 3.8.

In Figure 3.6c, we investigate the residual norms of the individual eigenvalues, $\|\mathcal{L}_{\text{sym}}\boldsymbol{u}_j - \lambda_j \boldsymbol{u}_j\|_2$, for the first ten eigenpairs in the fixed case of $n = 20\,000$ data points. Here we notice that the observed residual norms do not vary much over the computed part of the spectrum.

Finally, Figure 3.6d shows the average runtimes of the different methods and parameter choices in dependence of the total number of data points n. We observe that the runtimes of the traditional Nyström method has complexity $\mathcal{O}(n^3)$ because we chose m proportional to n, and the complexity of the full method is in $\mathcal{O}(n^2)$. All other methods show complexity in $\mathcal{O}(n)$ but with varying proportionality factors. The NFFT-based fast summation method is faster than its Fast Gauss competitor FIGTree by a factor of approximately 5–10. Moreover, NFFT–Nyström is slower than NFFT–Krylov with the same setup #2, so the Krylov-Schur method adds less overhead than the Nyström based decomposition.

All in all, the combination of NFFT-based fast summation with a state-of-the-art Krylov-Schur method yields by far the best performance. The three parameter choices from Example 3.18 give competitive results for different scenarios and desired accurucies, even with $\varepsilon_B = 0$. Due to the favorable handling of the medium-sized kernel parameter σ, fast summation is considerably faster than the popular Fast Gauss Transform method FIGTree.

In the following, we will showcase the effect of the improved accuracy on popular non-neural learning methods that utilize the graph Laplacian matrix. We will compare how the methods perform if the eigenvectors are computed with the NFFT-based Lanczos method or the traditional Nyström extension.

3.5.2 Application to spectral clustering of images by pixel color

As briefly described in Remark 2.17, spectral clustering is an increasingly popular technique [98] and we briefly illustrate the method proposed in [75]. For clustering into k classes, we compute new node features by computing the truncated eigenvector matrix U_k of the first k eigenvectors and row-normalizing each row such that each

node feature vector is on the unit sphere in $\mathbb{R}^k$. Then the simple clustering algorithm *k-means* is employed to cluster these new features into k classes.

Here, we apply spectral clustering to an image segmentation problem. The original image of size 533×800 is depicted in Figure 3.7a.[9] We construct a graph Laplacian where each pixel corresponds to a node in the graph and the distance measure is the distance between the values in all three color channels, i.e., the feature vectors are $\boldsymbol{x}_j \in \{0, 1, \ldots, 255\}^3$. Correspondingly, the graph Laplacian would be a dense matrix of size $426\,400 \times 426\,400$. We set the scaling parameter to $\sigma = 90$.

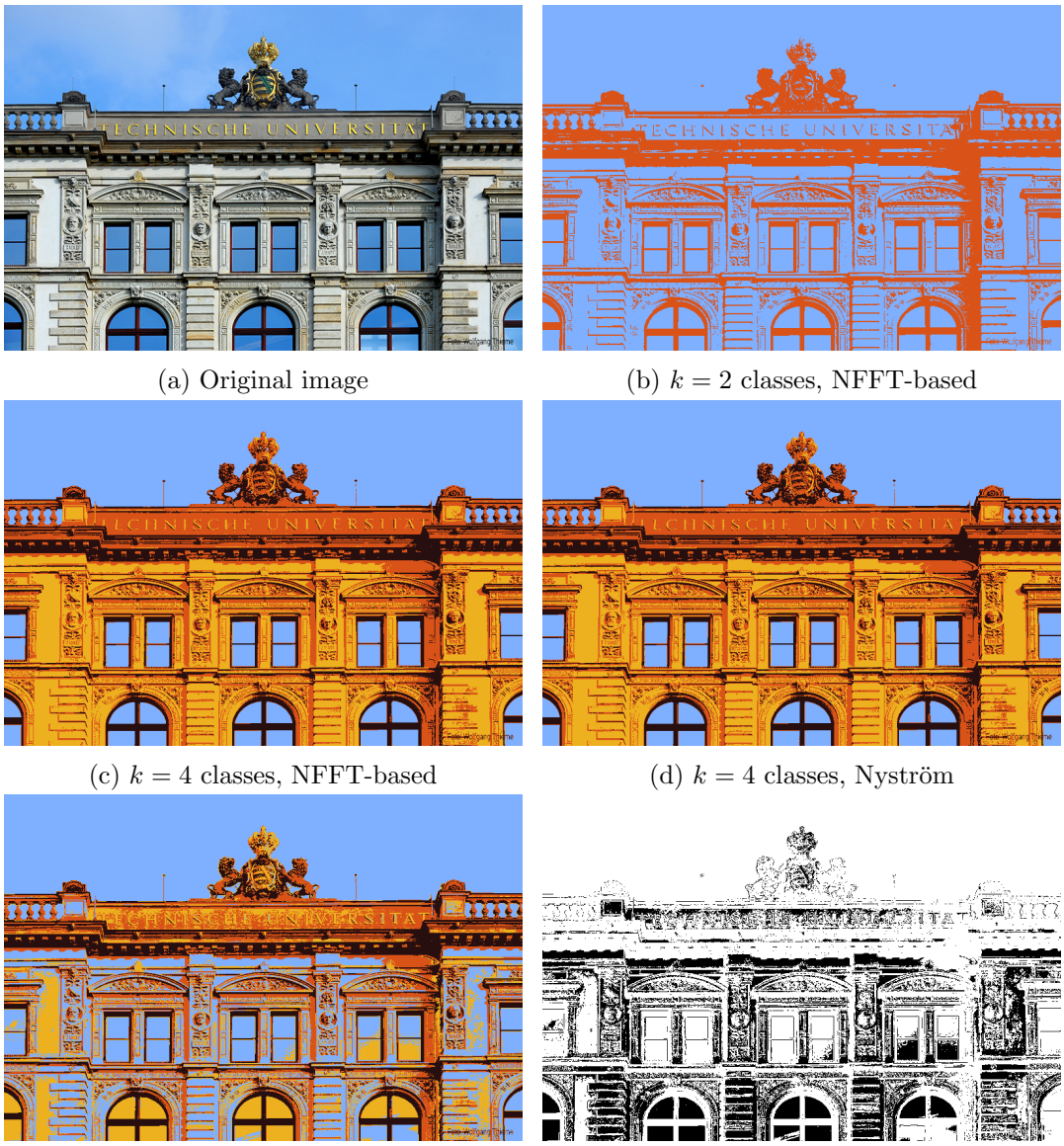
For obtaining reference results, we use the Krylov-Schur algorithm with the exact full matrix $\mathcal{A}$ computing 4 eigenvectors and this required more than 31 hours using up to 32 threads on a computer with Intel Xeon E7-4880 CPUs (2.50 GHz), using more than 500 CPU hours in total. Figure 3.8 shows the first ten eigenvalues of the matrix $\mathcal{L}_{\text{sym}}$. Since there are comparably large gaps after the first two and four eigenvalues, respectively, this can be interpreted as a sign that there are two main clusters which can be further subdivided into four similarly-sized clusters.

Next, we applied the NFFT–Krylov method as described in the previous section with parameters $N = 16$, $m = 2$, $p = 2$ and $\varepsilon_B = 1/8$ for the eigenvector computations. We show the results in Figure 3.7b and 3.7c for $k = 2$ and $k = 4$ classes, respectively. The segmented images look satisfactory in that the main features of the image are preserved and large areas of similar color are correctly assigned to the same cluster, while there are only small "noisy" areas. Compared to the segmented image from the direct computations, we have approximately 0.1 % differences (467 out of 426,400) in the class assignments in the case of $k = 4$ classes. For the runtimes, we measure approximately 25 seconds for the NFFT-based Lanczos method and 18 seconds for the k-means algorithm on a computer with Intel Core i7 CPU 970 (3.20 GHz) using one thread.

Additionally, we ran the Nyström method 100 times with parameter $m = 250$. Here the runtimes were approximately 60 seconds on average without the runtime for the clustering. We applied the k-means algorithm for $k = 4$ classes, which required approximately 22 seconds on average. We observed that in 79 of the 100 test runs, the images appear to be very close to the baseline reference, i.e., the differences are less than 2 %. In Figure 3.7d, we visualize the results of a corresponding test run. However, in 13 of the 100 test runs, the Nyström method returned eigenvectors which caused segmentation differences of more than 20 %, leading to such "noisy" images that we consider these as "failed" runs. See Figure 3.7e for one example with approximately 25 % differences. The differences between Figure 3.7c and 3.7e are shown as a black and white picture in Figure 3.7f.

Moreover, we tested increasing the parameter m to 500. Then, the runtimes increased to approximately 152 seconds on average. When applying the k-means algo-

[9] Image source: TU Chemnitz/Wolfgang Thieme



(a) Original image

(b) $k = 2$ classes, NFFT-based

(c) $k = 4$ classes, NFFT-based

(d) $k = 4$ classes, Nyström

(e) $k = 4$ classes, Nyström ("failed" run)

(f) Differences between (c) and (e)

Figure 3.7: Results of color-based image segmentation ($533 \times 800 = 426\,400$ pixels) via spectral clustering and k-means using the NFFT-based Krylov-Schur method from Algorithm 3.7 and the Nyström method from Section 3.4.1. "Failed run" in Subfigure (e) means segmentation differences of more than 20 % compared to the results obtained with the exact full adjacency matrix.

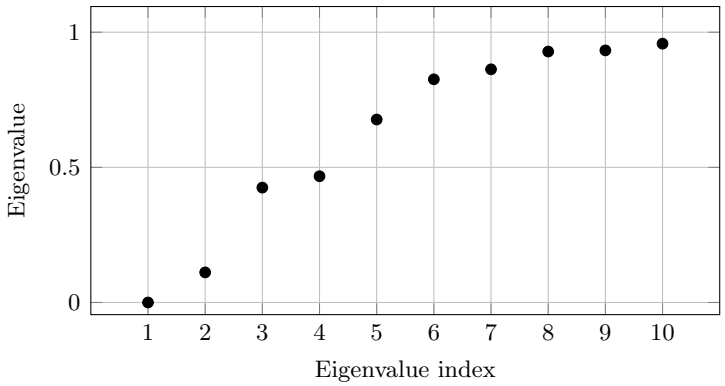


Figure 3.8: The first ten eigenvalues of $\mathcal{L}_{\text{sym}}$ for the fully-connected graph with Gaussian weights and scaling parameter $\sigma = 90$ for Figure 3.7

rithm to the obtained eigenvectors, the results improved. The differences compared to the reference image segmentation are less than 2 % in 85 of the 100 test runs and larger than 20 % in 9 test runs.

3.5.3 Semi-supervised learning by a phase field method

We here want to state a method that relies heavily on a number of eigenvectors of the graph Laplacian. It was proposed by Bertozzi and Flenner [10] for the setting of semi-supervised learning (SSL).

Suppose we have a graph-based data set as before where each vertex is assigned to one of C classes. A training set of s random sample vertices from each class is set up. For the case of $C = 2$ classes, a training vector $\boldsymbol{f} \in \mathbb{R}^n$ is set up with entries -1 for training nodes from one class, 1 for training nodes from the other class, and 0 for nodes that do not belong to the training set. The goal is to use $\boldsymbol{f}$ to find a classification vector $\boldsymbol{g} \in \mathbb{R}^n$. The sign of its entries is then used to predict each node's assigned class.

One successful approach computes $\boldsymbol{g}$ as the end point of the trajectory described by the Allen–Cahn equation

$$\boldsymbol{g} : [0, \infty) \to \mathbb{R}^n, \qquad \frac{\partial}{\partial t}\boldsymbol{g} = -\varepsilon\, \mathcal{L}_{\text{sym}}\, \boldsymbol{g} - \frac{1}{\varepsilon}\psi'(\boldsymbol{g}) + \Omega\,(\boldsymbol{f} - \boldsymbol{g}), \qquad \boldsymbol{g}(0) = \boldsymbol{f}$$

(see [95, 66] for details). Here $\psi(u) = (u^2 - 1)^2$ is the double-well potential, which we understand to be applied component-wise, and Ω denotes a diagonal matrix with entries $\Omega_{ii} = \omega_0 > 0$ if vertex i belongs to the training data and $\Omega_{ii} = 0$ otherwise. To discretize this ODE we will not introduce an index for the temporal discretization but rather assume that all values $\boldsymbol{g}$ are evaluated at the new time-point whereas $\bar{\boldsymbol{g}}$

indicates the previous time-point. We then obtain

$$\frac{\boldsymbol{g}-\bar{\boldsymbol{g}}}{\tau} + \varepsilon\,\mathcal{L}_{\text{sym}}\,\boldsymbol{g} + c\,\boldsymbol{g} = -\frac{1}{\varepsilon}\psi'(\bar{\boldsymbol{g}}) + c\,\bar{\boldsymbol{g}} + \Omega\,(\boldsymbol{f}-\bar{\boldsymbol{g}}),$$

where $c > 0$ is a positive parameter for the convexity splitting technique [10]. For a more detailed discussion of how to set these parameters we refer to [10, 13].

We now use the k computed eigenvalues and eigenvectors $\lambda_j, \boldsymbol{u}_j$ of $\mathcal{L}_{\text{sym}}$ such that we can write $\boldsymbol{g} = \sum_{j=1}^{k} g_j \boldsymbol{u}_j$ and from this we get

$$\frac{g_j-\bar{g}_j}{\tau} + \varepsilon\,\lambda_j\,g_j + c\,g_j = -\frac{1}{\varepsilon}\boldsymbol{u}_j^T\psi'(\bar{\boldsymbol{g}}) + c\,\bar{g}_j + \boldsymbol{u}_j^T\Omega\,(\boldsymbol{f}-\bar{\boldsymbol{g}}) \qquad \forall\; j = 1,\ldots,k.$$

This equation can be solved to obtain the new coefficients g_j from the old coefficients $\bar{g}_j$. After a sufficient number of time steps, $\boldsymbol{g}$ will converge against a stable solution and the signs of its entries can be used for classification.

We apply this method to the spiral data set from Example 3.19 for $n = 100\,000$, again with $\sigma = 3.5$. Since this is a non-binary classification task with five classes, we perform "one-vs-rest" classification, i.e., we learn one classifier $\boldsymbol{g}^{(j)}$ for each of the five classes, predicting whether a node belongs to class $\#j$ or to any of the remaining four. We finally set the predicted class of each node to the index j for which $\boldsymbol{g}^{(j)}$ is largest.

We computed the eigenvectors to the $k = 5$ smallest eigenvalues of the Laplacian; once by the NFFT–Krylov method with $N = 32$, $m = 4$, and $\varepsilon_B = 0$ (setup #2 from Example 3.18), and once with the traditional Nyström method with $m = p = 1\,000$ and $k = 5$. We then applied the described method with $\tau = 0.1$, $\varepsilon = 10$, $\omega_0 = 10\,000$, and $c = \frac{2}{\varepsilon} + \omega_0$. The iteration terminated if the squared relative change in $\boldsymbol{g}$ was less than 10^{-10}. We repeat this process for 50 instances of the spiral data set and sample sizes $s \in \{1, 2, 3, 4, 5, 7, 10\}$. Note that this constitutes a maximum of 50 training nodes in total, yielding a ratio of labeled nodes of between 0.005% and 0.05%.

Figure 3.9 depicts the average accuracy results. We conclude that in this example, the increased eigenvector quality achieved by the NFFT-based method yields an average accuracy boost of approximately 1 percentage point. On a computer with Intel Core i7 CPU 4770 (3.40 GHz), the runtimes were approximately 8 seconds for the NFFT-based Lanczos method, 27 seconds for the Nyström method, and less than a second for the solution of the Allen–Cahn equation, which almost always converged after only three time steps.

3.5.4 Conclusion

In this chapter, we have successfully applied the computational power of NFFT-based fast summation to core tools of data science. This was possible due to the nature of the fully connected graph Laplacian and the fact that many algorithms – most notably

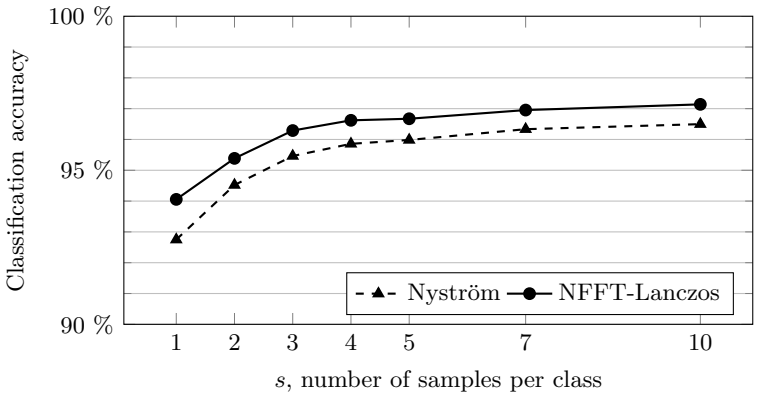


Figure 3.9: Comparison of average classification rates with the phase field method on relabeled spiral data sets

the Lanczos method for eigenvalue computation – only require matrix-vector products with the Laplacian matrix. By using Fourier coefficients to approximate the Gaussian kernel, we use Algorithm 3.3 to compute strong approximations of the matrix-vector product in $\mathcal{O}(n)$ complexity without storing or setting up the full matrix, as opposed to the full matrix's $\mathcal{O}(n^2)$ storage, setup, and application complexity. The constant in the $\mathcal{O}(n)$ notation grows exponentially with the feature space dimension d, which makes this method only applicable to the case of $d \leq 4$. In particular, the fully connected graphs for 3D point clouds as in Example 2.19 can be analyzed with these tools.

For eigenvalue and eigenvector computations, we have discussed the current alternative method of choice in the Nyström extension and developed a hybrid method that allows the basic Nyström idea to benefit from NFFT-based fast matrix-vector products. In our numerical experiments, we found that the NFFT-Nyström method achieved much better eigenvalue accuracy than the traditional Nyström extension even for a significantly smaller parameter m, but was in turn outperformed by the NFFT-Krylov method.

In strongly eigenvector-dependent applications like in Section 3.5.3, the higher accuracy of the NFFT-Krylov method directly leads to better classification results. In some other applications, however, it is hard to predict if better eigenvector accuracy distinctly improves the results. For instance in Section 3.5.2, the traditional Nyström extension still achieved good image clusterings on average with small parameter m despite its rather inaccurate eigenvectors. Here, the NFFT-Krylov method still has very good selling points in its greatly improved runtime as well as its consistency, while the traditional Nyström tends to "fail" in some test runs.

For more extensive results in other settings, we again refer to [2].

4 Graph Convolutional Networks for fully connected graphs

This chapter presents the second main contribution of our research, published in [3]. A few parts of the descriptions are taken from that work unchanged.

Graph Convolutional Networks (GCNs) are a special type of neural network that utilize the graph structure by performing graph convolution as in Section 2.4. Like most GNNs, its most popular form [56] is designed towards sparse graphs.

For the alternative case of dense graphs, we here present the computational and methodological challenges for GCNs and motivate steps we can undertake to avoid these disadvantages. We finally arrive at our proposed architecture, the Pseudoinverse GCN, which utilizes low-rank techniques from numerical linear algebra to achieve efficient extraction of clustering information from the dense Laplacian. The feature maps are based on a novel three-part filter space motivated by an approximation of the Laplacian pseudoinverse. In the end we showcase our method's performance regarding runtime and accuracy in various experiments with real-world data sets.

4.1 Background of Graph Convolutional Networks

4.1.1 Neural Networks

In the last decade, the terms *artificial intelligence*, *machine learning*, and *neural networks* have almost interchangeably become synonyms with progress in research for data science problems. Neural networks are the driving force behind many impressive novelties in our digital world. Computers, phones and other "smart" devices use these learning methods to emulate cognitive capabilities that are hard to obtain with traditional programming.

The general concept of neural networks has been around for a long time, but their disruptive emergence as the go-to solution for learning tasks has only come up in this millennium. This is due to a sharp increase in its two main ingredients: computational power and data availability.

It is crucial to stress that despite their recent omnipresence in the advertising of industrial innovations, neural networks by themselves cannot guarantee quality results. The networks need to be constructed and trained carefully and are highly dependent on the training data. For many applications of machine learning, the goal is only user assistance instead of imperative problem solution. Hence absolute

reliability without exception is often not the main requirement. For these tasks, neural networks are perfectly apt.

Ethical concerns arise in sensitive areas where miscalculations may have catastrophic consequences, and any method that is anything less than absolutely failproof will not be accepted. Automated driving is currently the most highly debated such instance. Adversarial attacks on neural networks are able to disturb such methods to the point of causing physical danger [31]. There is still much research necessary before neural networks will be able to satisfy absolute performance guarantees, if that is ever possible. However, that is not within the scope of this thesis.

In this section, we define the underlying concepts of neural networks to the degree that we need it as the background for the upcoming derivations.

General concepts. At their core, neural networks are highly over-parameterized nonlinear functions that map input feature vectors $\boldsymbol{x} \in \mathbb{R}^{N_{\text{in}}}$ to new output feature vectors $\boldsymbol{y} \in \mathbb{R}^{N_{\text{out}}}$. For notational purposes, the parameters can be formally collected in a parameter vector $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$. The neural network can be described as a function

$$\Phi_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^m, \qquad \Phi_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{y}. \tag{4.1}$$

Each input feature vector $\boldsymbol{x}$ is thought to be a fixed d-dimensional representation of one *sample*. The parameters $\boldsymbol{\theta}$ should be chosen in such a way that the output $\boldsymbol{y} = \Phi_{\boldsymbol{\theta}}(\boldsymbol{x})$ can be used as a more informative description of the sample based on which some underlying data science task can be solved. The nature and structure of the samples as well as the desired output properties depend on the application.

In order to find appropriate parameter values, neural networks are *trained*. This refers to the optimization of a loss function on some training data. Usually a large number of samples is repeatedly passed to the network and the discrepancy between the output and desired output is used to update the parameters. However, since this training loss cannot quantify how well the network generalizes to new unlabeled data, choosing $\boldsymbol{\theta}$ as the exact optimal solution would often lead to subpar predictions due to *overfitting*. This key concept refers to the adaptation of the network to statistical particularities in the training set that cannot be expected to be found in future data. Hence a rough approach to the optimization is sufficient and even desired.

Perceptrons. The fundamental building blocks of many neural network models are simple mappings known as perceptrons or neurons. They are essentially compositions of an affine linear function with a nonlinear element-wise activation function.

Perceptrons have originally been developed as the simplified mathematical description of neurons in brain cells [5]. The activation value of a neuron y_j is obtained by taking a bunch of input neurons x_i, forming the weighted sum of their values, adding a scalar *bias*, and applying an activation function $\sigma : \mathbb{R} \to \mathbb{R}$. The whole operation is

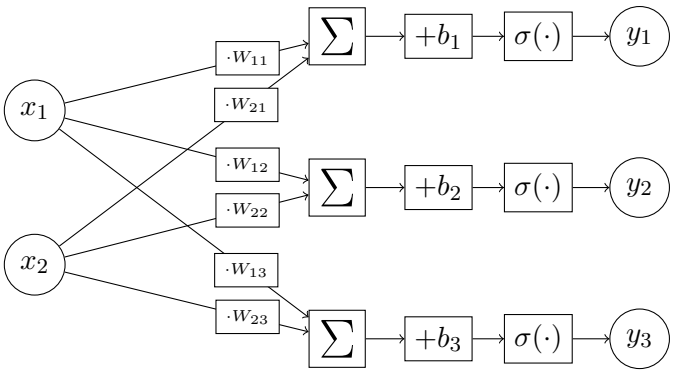


Figure 4.1: Visualization of a perceptron with two input neurons and three output neurons

visualized in Figure 4.1 and can be written as $y_j = \sigma\big(\sum_i W_{ij} x_i + b_j\big)$. Multiple such neurons y_j form the output of the perceptron.

By collecting the N_{in} input neurons and N_{out} output neurons in vectors or matrices, we obtain a neural network operation in the sense of (4.1) as

$$\boldsymbol{y} = \Phi_{W,\boldsymbol{b}}(\boldsymbol{x}) = \sigma(W^T \boldsymbol{x} + \boldsymbol{b}) \tag{4.2}$$

The weight matrix $W \in \mathbb{R}^{N_{\text{in}} \times N_{\text{out}}}$ and the bias vector $\boldsymbol{b} \in \mathbb{R}^{N_{\text{out}}}$ are the parameters of the module.

Modular formulation. More complex neural networks can be constructed by taking multiple perceptrons or similar modules and composing them by passing one module's output to another module as input. The result is a chain of modules that can be called recusively.

This is called a *feed-forward* architecture. The opposing concept is that of *recurrent* neural networks, where the modules may be arranged in a circular fashion without strict chronological ordering from input to outputs [5].

4.1 Definition. *(Modular Feed-Forward Neural Networks)*
A feed-forward neural network is a sequence of L neural modules

$$\Phi^{(l)}_{\boldsymbol{\theta}^{(l)}} : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l} \qquad \forall\, l = 1, \dots, L,$$

called *layers*. The number of modules L is the *depth* and the matching input and output feature dimensions $N_0, \dots, N_L$ are the *widths* of the network. Specifically, $N_0 = d$ is the input width, $N_L = m$ is the output width, and $N_1, \dots, N_{L-1}$ are the *hidden* widths.

The network operation is the composition of the layers,

$$\Phi_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^m, \quad \boldsymbol{x} \mapsto \boldsymbol{y} = \Phi_{\boldsymbol{\theta}}(\boldsymbol{x}) = \big(\Phi^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ \cdots \circ \Phi^{(1)}_{\boldsymbol{\theta}^{(1)}}\big)(\boldsymbol{x}) \tag{4.3}$$

and the network's parameter vector $\boldsymbol{\theta}$ is the concatenation of all layer parameters $\boldsymbol{\theta}^{(l)}$. The evaluation of $\Phi(\boldsymbol{x})$ is called *propagation* of $\boldsymbol{x}$ through the network and can be written recursively as

$$\boldsymbol{x}^{(0)} = \boldsymbol{x} \in \mathbb{R}^{N_0}, \qquad \boldsymbol{x}^{(l)} = \Phi^{(l)}_{\boldsymbol{\theta}^{(l)}}(\boldsymbol{x}^{(l-1)}) \in \mathbb{R}^{N_l} \quad \forall\, l = 1, \ldots, L, \qquad \boldsymbol{y} = \boldsymbol{x}^{(L)}.$$

The intermediate quantities $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(L-1)}$ are called *hidden features* of the propagated sample.

The most basic example of feed-forward neural network is a Multi-Layer Perceptron (MLP) where each layer module is a perceptron as in (4.2). The activation functions σ are typically the same for all layers except for the last one, which does not use any activation function (i.e., σ is the identity). The full propagation scheme $\boldsymbol{y} = \Phi_{\boldsymbol{\theta}}(\boldsymbol{x})$ can be written as

$$\begin{aligned}
&\text{Input:} && \boldsymbol{x}^{(0)} = \boldsymbol{x} && \in \mathbb{R}^{N_0} \text{ with } N_0 = d,\\
&\text{Hidden layers:} && \boldsymbol{x}^{(l)} = \sigma\big(W^{(l)T}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)}\big) && \in \mathbb{R}^{N_l} \text{ for } l = 1, \ldots, L-1,\\
&\text{Output:} && \boldsymbol{y} = \boldsymbol{x}^{(L)} = \quad W^{(L)T}\boldsymbol{x}^{(L-1)} + \boldsymbol{b}^{(L)} && \in \mathbb{R}^{N_L} \text{ with } N_L = m.
\end{aligned}$$

All weight matrices $W^{(l)}$ and bias vectors $\boldsymbol{b}^{(l)}$ are contained in the global parameter vector $\boldsymbol{\theta}$.

Generally, the layers $\Phi^{(l)}$ may be more complex constructs than just perceptrons, e.g., composite structures of submodules that are themselves modular neural networks.

4.1.2 Supervised classification

The typical domain of neural networks is *supervised learning*. In this setting, we assume that we have a data set of independent training samples $\boldsymbol{x}$ for which the desired output is known, and the goal is to find a model to *infer* predictions for new data points that were not known at training time.

A neural network neural network Φ can be used for this task by learning its parameters in such a way that some loss function on the training set is minimized. Afterward, a prediction for any new sample $\boldsymbol{x}$ can be made based on $\Phi(\boldsymbol{x})$. The success of neural networks is due to the empirical observation that many architectures generalize well from the training data to previously unseen, but similarly distributed data.

4.2 Definition. *(Supervised sample classification)*
Consider a training data set of n_{train} samples described by a feature vector $\boldsymbol{x}_i^{\text{train}} \in \mathbb{R}^d$ and assigned a true class label $c_i^{\text{train,true}} \in \{1, \dots, m\}$ $(i = 1, \dots, n_{\text{train}})$. The goal of *supervised sample classification* is to find a classifier model

$$F : \mathbb{R}^d \to \{1, \dots, m\}$$

such that $F(\boldsymbol{x}_i^{\text{train}}) = c_i^{\text{train,true}}$ for most of the training samples and that F generalizes well onto similar non-training nodes $\boldsymbol{x}$.

There are a vast number of methods for defining a classifier model. A fundamental challenge for computational methods is that the prediction must be discrete. Simply taking a continuous variable and rounding it to $\{1, \dots, m\}$ is of course misguided unless the classes are specifically ordered in some way. Otherwise, the order of classes is generally arbitrary and two "adjacent" classes are not more similar than any other pair.

For binary classification, it is often the most simple approach to use a model that returns a single number, $f(\boldsymbol{x}) \in \mathbb{R}$, and then set $F(\boldsymbol{x})$ based on the sign of that number. Some simple examples were given in Section 3.5. For multi-class classification, such methods can still be used by training multiple "one-vs-rest" classifiers and picking the class with the highest score, cf. Section 3.5.3.

Neural networks can be trained for classification by using a simple framework of logits, softmax, and the cross-entropy loss that is consensually regarded as the standard way to approach these tasks [5].

4.3 Definition. *(Neural network for sample classification)*
Consider a supervised sample classification task as in Definition 4.2 with feature dimension d and m classes.
(a) A neural network $\Phi_{\boldsymbol{\theta}}$ with input width $N_0 = d$ and output width $N_L = m$ can be used for this task by propagating any sample feature vector $\boldsymbol{x} \in \mathbb{R}^d$ through the network and interpreting the output as *logits* of the predicted likelihood for the sample to belong each of the m classes. The larger the j-th logit, the more likely sample $\boldsymbol{x}$ is considered to belong to class j, and the class with maximum logit is set as the predicted label. This leads to the classifier function

$$F_{\boldsymbol{\theta}}(\boldsymbol{x}) = \operatorname*{arg\,max}_{c=1,\dots,m} y_c \quad \text{with} \quad (y_c)_{c=1}^m = \Phi_{\boldsymbol{\theta}}(\boldsymbol{x}). \tag{4.4}$$

For better quantification, each vector of logits y_j can be turned into proper proba-

bilities by means of the *softmax* function,

$$\mathbb{P}_{\boldsymbol{\theta}}^{\text{pred}}(c \,|\, \boldsymbol{x}) = \frac{\exp(y_c)}{\sum_{j=1}^{m} \exp(y_j)} \in (0,1) \qquad \forall\, c = 1, \ldots, m. \tag{4.5}$$

With this, all probabilities are non-negative and sum up to 1.

(b) For the training data pairs $\boldsymbol{x}_i^{\text{train}}, c_i^{\text{train,true}}$ $(i = 1, \ldots, n_{\text{train}})$, we can set the true probabilities as

$$\mathbb{P}^{\text{true}}(c \,|\, \boldsymbol{x}_i^{\text{train}}) = \begin{cases} 1 & \text{if } c = c_i^{\text{train,true}}, \\ 0 & \text{else,} \end{cases}$$

which is also called the *one-hot encoding* of the true labels. For each individual training sample, the *cross-entropy* or *negative log-likelihood* loss is defined as

$$\begin{aligned} L(\boldsymbol{x}_i^{\text{train}}, c_i^{\text{train,true}}; \boldsymbol{\theta}) &= \sum_{c=1}^{m} \mathbb{P}^{\text{true}}(c \,|\, \boldsymbol{x}_i^{\text{train}}) \, \log\left(\mathbb{P}_{\boldsymbol{\theta}}^{\text{pred}}(c \,|\, \boldsymbol{x}_i^{\text{train}})\right) \\ &= \log\left(\mathbb{P}_{\boldsymbol{\theta}}^{\text{pred}}(c_i^{\text{train,true}} \,|\, \boldsymbol{x}_i^{\text{train}})\right) \\ &= y_{c_i^{\text{train,true}}} - \log\left(\sum_{j=1}^{m} \exp(y_j)\right) \;\text{ with } (y_c)_{c=1}^{m} = \Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_i^{\text{train}}). \end{aligned}$$

Then the network can be trained by minimizing the average loss on the whole training data set

$$\min_{\boldsymbol{\theta}} \quad L(\boldsymbol{\theta}) = \frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} L(\boldsymbol{x}_i^{\text{train}}, c_i^{\text{train,true}}; \boldsymbol{\theta})$$

or some statistical approximation thereof.

The probability notation $\mathbb{P}(c|\boldsymbol{x})$ in (4.5) is common in machine learning research [74] and refers to the probability that the label is c under the condition that the corresponding feature vector is $\boldsymbol{x}$. It is borrowed from statistical learning theory, where the class label c and the feature vector $\boldsymbol{x}$ are considered correlated random variables.

4.1.3 Graph Neural Networks for Transductive Learning

As opposed to the task of supervised learning described above, *semi-supervised learning* already provides unlabeled data during training time. Methods need to extract knowledge both from the available labeleled data as well as from hidden structures within the large amount of unlabeled data.

There are two types of semi-supervised learning paradigms. *Inductive* learning is more similar to supervised learning in that it aims to create a model in the form of a mapping from inputs to outputs. In this thesis, we will focus on *transductive learning*. The premise is that the entire set of samples is known at training time, including *all* unlabeled samples for that we ever need a prediction. Hence the goal is to come up with a finite set of predictions instead of a predictor function.

In the following definition, we introduce the transductive versions of Definitions 4.2 and 4.3.

4.4 Definition. *(Transductive semi-supervised sample classification)*
(a) Assume a data set of n samples is given, each described by a feature vector $\boldsymbol{x}_i \in \mathbb{R}^d$, $i = 1, \dots, n$. For a subset of *training samples*, a true class label

$$c_i^{\text{true}} \in \{1, \dots, m\} \quad \forall\, i \in \mathcal{I}_{\text{train}} \subset \{1, \dots, n\}$$

is known, where m is the number of classes. Generally there are much fewer labeled samples than unlabeled ones, $|\mathcal{I}_{\text{train}}| = n_{\text{train}} \ll n$. The goal of *transductive sample classification* is to find predicted classes

$$c_i^{\text{pred}} \in \{1, \dots, m\} \quad \forall\, i \notin \mathcal{I}_{\text{train}}$$

for the remaining $n - n_{\text{train}}$ unlabeled samples, such that samples with similar features are likely to belong to the same class.

(b) A neural network model $\Phi_{\boldsymbol{\theta}}$ with input width $N_0 = d$ and output width $N_L = m$ can be used for transductive sample classification by assembling *all* sample features in a matrix $X = \begin{bmatrix}\boldsymbol{x}_1 \cdots \boldsymbol{x}_n\end{bmatrix}^T$ and writing the propagation through the neural network in matrix-valued notation,

$$\mathbb{R}^{n\times d} \ni X \mapsto Y = \Phi_{\boldsymbol{\theta}}(X) = \begin{bmatrix}\Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_1)^T \\ \vdots \\ \Phi_{\boldsymbol{\theta}}(\boldsymbol{x}_n)^T\end{bmatrix} \in \mathbb{R}^{n\times m}. \tag{4.6}$$

The entries of the i-th row of Y can be used as the logits for each class, yielding the class prediction

$$c_i^{\text{pred}} = \operatorname*{arg\,max}_{j=1,\dots,m} Y_{ij}. \tag{4.7}$$

The network can be trained by minimizing the average cross-entropy loss on the training subset,

$$\min_{\boldsymbol{\theta}}\ L(\boldsymbol{\theta}) = \frac{1}{n_{\text{train}}} \sum_{i\in\mathcal{I}_{\text{train}}} \left(Y_{i\, c_i^{\text{true}}} - \log\left(\sum_{k=1}^{m} \exp(Y_{ik}) \right) \right). \tag{4.8}$$

The propagation of multiple samples through the network as in (4.6) is a common concept in practical implementations of supervised learning as well. Any number of B samples can be bundled together as a *batch* and propagated through the network simultaneously to enable better parallelization. The network can hence be understood as a mapping $\mathbb{R}^{B\times d} \to \mathbb{R}^{B\times m}$ where $\Phi_{\boldsymbol{\theta}}$ is applied row-wise.

The advantage of the transductive setting is that the network can always expect to receive the full data set as one batch, i.e., the batch size is always $B = n$. Hence the model does not need to behave in the same way whether a single sample or a large batch is given, which makes it possible for i-th output logits to depend on the features of other samples than itself. This behavior is crucial for transductive learning to enable the extraction of information from the presence of unlabeled nodes.

However, note that this makes the network structure less flexible and key concepts from supervised learning cannot be utilized, including the training strategy of *mini batches* used in Stochastic Gradient Descent (SGD) and its variants [5].

As an example, the matrix-valued propagation scheme of a single perceptron layer as in (4.2) can be written as

$$Y = \begin{bmatrix} \sigma(W^T\boldsymbol{x}_1 + \boldsymbol{b})^T \\ \vdots \\ \sigma(W^T\boldsymbol{x}_n + \boldsymbol{b})^T \end{bmatrix} = \sigma\big(XW + \mathbf{1}\boldsymbol{b}^T\big) \quad \text{with} \quad X = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_n^T \end{bmatrix}. \tag{4.9}$$

The addition of the bias term is sometimes only written as $+\boldsymbol{b}$ instead of $+\mathbf{1}\boldsymbol{b}^T$.

Transductive learning on graphs. Transductive learning is found in settings where there is a fixed set of sample entities and a notion of similarity between samples. The overlap with graph applications is evident. Hence many examples for transductive tasks are found in graph learning.

4.5 Definition. *(Transductive node classification on graphs)*
Assume a graph with n nodes, each equipped with a node feature vector $\boldsymbol{x}_i \in \mathbb{R}^d$ is given. Transductive *node classification* or *graph segmentation* is the task of Definition 4.4(a) where the samples are the nodes of the graph. The goal is to assign class labels to each node in the graph based on a small subset of labeled training nodes, such that adjacent nodes are likely to belong to the same class.
Neural networks that solve this task in the sense of Definition 4.4(b) while utilizing the graph structure are called Graph Neural Networks.

Learning on graphs is often described as a generalization of learning on audio signals or images, because the regular pixel grid in an image defines a more strictly structured notion of adjacency. However, the equivalent task of node classification in the sense of Definition 4.5 would be the classification of pixels in a *single* image based on a few labeled pixels, without knowledge on similar images. This task is relevant in some

applications [9] but the prevalent setting in computer vision is the supervised transfer of segmentation information from training images to future test images.

4.1.4 Convolution and spectral filters as learned parameters

Combining neural networks with graph convolution was first proposed by Bruna et al. [15]. It offers a unique interpretation of its action in terms of graph signals as in Section 2.4.

Let us focus on the multiplication with the weight matrix in a single perceptron as in (4.9),

$$X^{\text{out}} = X^{\text{in}}\, W \quad \text{with} \quad X^{\text{in}} \in \mathbb{R}^{n \times N_{\text{in}}},\ X^{\text{out}} \in \mathbb{R}^{n \times N_{\text{out}}},\ W \in \mathbb{R}^{N_{\text{in}} \times N_{\text{out}}}.$$

Usually we interpret these feature matrices as collections of n feature vectors in their rows, directly corresponding to the n nodes of the underlying graph. Alternatively we can interpret the *columns* of X^{in} and X^{out} as signals on the graph (cf. Definition 2.6). Let us here use the notation

$$\begin{aligned} X^{\text{in}} &= \begin{bmatrix} - \ \boldsymbol{h}_1^{\text{in}\,T} \ - \\ \vdots \\ - \ \boldsymbol{h}_n^{\text{in}\,T} \ - \end{bmatrix} = \begin{bmatrix} | & & | \\ \boldsymbol{f}_1^{\text{in}} & \cdots & \boldsymbol{f}_{N_{\text{in}}}^{\text{in}} \\ | & & | \end{bmatrix} \\ \text{and} \qquad X^{\text{out}} &= \begin{bmatrix} - \ \boldsymbol{h}_1^{\text{out}\,T} \ - \\ \vdots \\ - \ \boldsymbol{h}_n^{\text{out}\,T} \ - \end{bmatrix} = \begin{bmatrix} | & & | \\ \boldsymbol{f}_1^{\text{out}} & \cdots & \boldsymbol{f}_{N_{\text{out}}}^{\text{out}} \\ | & & | \end{bmatrix}, \end{aligned} \tag{4.10}$$

i.e., $\boldsymbol{h}_i$ refer to the node features and $\boldsymbol{f}_i$ to the graph signals. The multiplication with the weight matrix can be understood as the creation of N_{out} new output signals as weighted sums of the N_{in} input signals,

$$\boldsymbol{f}_j^{\text{out}} = \sum_{i=1}^{N_{\text{in}}} W_{ij}\, \boldsymbol{f}_i^{\text{in}}. \tag{4.11}$$

The coefficients W_{ij} of these linear combinations are learned. The basic idea of Graph Convolutional Networks is to replace the simple weighted sums in perceptrons by convolutions with different filters as in Definition 2.22, i.e.,

$$\boldsymbol{f}_j^{\text{out}} = \sum_{i=1}^{N_{\text{in}}} \boldsymbol{g}_{ij} * \boldsymbol{f}_i^{\text{in}}.$$

Instead of learning the signals $\boldsymbol{g}_{ij} \in \mathbb{R}^n$ in the spatial domain, a better interpretation is gained by learning their Fourier coefficients $\hat{\boldsymbol{g}}_{ij} \in \mathbb{R}^n$. This corresponds to learning

spectral filters φ_{ij} in the sense of Definition 2.23 and leads to the formula

$$\boldsymbol{f}_j^{\text{out}} = \sum_{i=1}^{N_{\text{in}}} \varphi_{ij}(\mathcal{L}_{\text{sym}})\boldsymbol{f}_i^{\text{in}} = \sum_{i=1}^{N_{\text{in}}} U\varphi_{ij}(\Lambda)U^T\boldsymbol{f}_i^{\text{in}}$$

with the notation from Definition 2.13 and Equation (2.21).

Each φ_{ij} must be a function defined on the spectrum of $\mathcal{L}_{\text{sym}}$, $\varphi_{ij} : \{\lambda_k\}_{k=0}^{n-1} \to \mathbb{R}$. They can be defined on the whole interval $[0, 2]$ for simplicity but only their values on the eigenvalues matter. Hence each filter has n degrees of freedom instead of the one degree of freedom in (4.11), drastically increasing the complexity of the model.

However, Proposition 2.24 describes that not all spectral filters are suited for classification tasks. Based on that, assume that we can design a K-dimensional function space of spectral filters with desirable properties

$$\mathcal{P}_K = \text{span}\{\varphi^{(1)}, \dots, \varphi^{(K)}\}$$

with basis functions $\varphi^{(k)}$. Restricting all learned filters to this space, $\varphi_{ij} \in \mathcal{P}_K$, can be achieved by claiming

$$\varphi_{ij}(\lambda) = \sum_{k=1}^{K} W_{ij}^{(k)}\varphi^{(k)}(\lambda). \tag{4.12}$$

and learning the coefficients $W_{ij}^{(k)}$. This way, the output signal and full output matrix can be written as

$$\boldsymbol{f}_j^{\text{out}} = \sum_{k=1}^{K}\sum_{i=1}^{N_{\text{in}}} W_{ij}^{(k)}\varphi^{(k)}(\mathcal{L}_{\text{sym}})\boldsymbol{f}_i^{\text{in}} \quad \text{and} \quad X^{\text{out}} = \sum_{k=1}^{K} \varphi^{(k)}(\mathcal{L}_{\text{sym}})X^{\text{in}}W^{(k)}.$$

When using this formula to replace the matrix multiplication in the perceptron module from (4.9), we get the following propagation rule.

4.6 Definition. *(Graph Convolutional Layer)*
Consider a graph with n nodes and K real-valued basis filter functions $\varphi_1, \dots, \varphi_K$ whose domains are the spectrum of $\mathcal{L}_{\text{sym}}$. A *graph convolutional layer* with activation function σ and input and output widths N_{in}, N_{out} is parameterized by K weight matrices $W^{(k)} \in \mathbb{R}^{N_{\text{in}} \times N_{\text{out}}}$ for $k = 1, \dots, K$ and the bias vector $\boldsymbol{b} \in \mathbb{R}^{N_{\text{out}}}$. The layer constitutes the mapping

$$X^{\text{in}} \mapsto X^{\text{out}} = \sigma\left(\sum_{k=1}^{K} \varphi^{(k)}(\mathcal{L}_{\text{sym}})\, X^{\text{in}}\, W^{(k)} + \mathbf{1}\boldsymbol{b}^T\right). \tag{4.13}$$

Consider the special case of $K = 1$. Then all learned filters φ_{ij} are differently scaled multiples of a single basis function φ_1 and the convolution result is just

$\varphi^{(1)}(\mathcal{L}_{\text{sym}})X^{\text{in}}W^{(1)}$. In a graph convolutional layer as in (4.13), this is equivalent to convolving each column of X^{in} with $\varphi^{(1)}$ and afterwards applying a standard perceptron with weight matrix $W^{(1)}$.

Just like an MLP is made up of multiple perceptron layers, we can define a graph convolutional network as the composition of multiple graph convolutional layers. This architecture and its utilization for transductive learning is based on [56].

4.7 Definition. *(Graph convolutional network)*
Consider a graph with n nodes. A graph convolutional network (GCN) for transductive node classification consists of L graph convolutional layers that all share the same filter space $\mathcal{P}_K = \text{span}\{\varphi^{(1)}, \dots, \varphi^{(K)}\}$. All layers use the same activation function σ except for the last one, which uses the identity. The input and output widths of each layer are given by two consecutive numbers in $N_0, \dots, N_L$, where the network input width $N_0 = d$ is the input feature dimension and the network output width $N_L = m$ is the number of classes.
The network can be written as $Y = \Phi_{\boldsymbol{\theta}}(X)$ with the propagation scheme

$$
\begin{aligned}
X^{(0)} &= X && \in \mathbb{R}^{n\times N_0} \text{ with } N_0 = d,\\
X^{(l)} &= \sigma\Big(\sum_{k=1}^{K} \varphi^{(k)}(\mathcal{L}_{\text{sym}})\, X^{(l-1)}\, W^{(l,k)} + \mathbf{1}\boldsymbol{b}^{(l)T}\Big) && \in \mathbb{R}^{n\times N_l} \text{ for } l = 1, \dots, L-1,\\
X^{(L)} &= \sum_{k=1}^{K} \varphi^{(k)}(\mathcal{L}_{\text{sym}})\, X^{(L-1)}\, W^{(L,k)} + \mathbf{1}\boldsymbol{b}^{(L)T} && \in \mathbb{R}^{n\times N_L} \text{ with } N_L = m.
\end{aligned}
$$

The decisive metaparameter of a GCN architecture is the choice of the filter space $\mathcal{P}_K$. Its design occupies a large portion of research for the development of new methods.

4.1.5 General design principles for filter functions

In this subsection, we are going to summarize a few desirable properties of the filter basis functions that lead to favorable behavior of the neural network.

Localization property. First, we name the polynomial filter design of [24] which gives a direct connection between the spectral filters and spatial localization.

If the spectral filter φ is a polynomial in λ of degree p,

$$\varphi(\lambda) = \sum_{j=0}^{p} a_j \lambda^j \quad \text{with} \quad a_0, \dots, a_p \in \mathbb{R},$$

then the corresponding convolutional matrix can be set up as

$$\varphi(\mathcal{L}_{\text{sym}}) = \sum_{j=0}^{p} a_j \mathcal{L}_{\text{sym}}^j$$

without requiring the eigenvalue decomposition of $\mathcal{L}_{\text{sym}}$. This way, $\varphi(\mathcal{L}_{\text{sym}})$ is connected to walks of length p. For instance, the i-j-th entry of $\varphi(\mathcal{L}_{\text{sym}})$ is zero if there is no path of length at most p between nodes i and j.

4.8 Remark. *(Spatially localized filters)*
If all filter basis functions $\varphi^{(k)}$ are polynomials in λ with maximum degree p, then all convolutional matrices $\varphi^{(k)}(\mathcal{L}_{\text{sym}})$ can be set up without computing any eigeninformation of $\mathcal{L}_{\text{sym}}$. Matrix-vector products with $\varphi^{(k)}$ can be evaluated using at most p products with the adjacency matrix $\mathcal{A}$. The filter space is *spatially localized* in the sense that the output features of node i after a single layer only depend on input features of nodes j that are at most p steps away from node i.

Note that this does not mean that all filter functions must be polynomials. However, if we come to the conclusion that non-polynomial filters have merit, then their setup will require additional care.

Smoothing filter spaces. Next, we notice that the learned filters can directly influence the smoothness of the output. As seen in Proposition 2.15, the low-frequency eigenvectors of $\mathcal{L}_{\text{sym}}$ carry clustering information. In order for a node classification method to follow the graph's intrinsic clustering structure, it appears desirable to reinforce this kind of clustering information in the output logits (cf. Definition 4.4(b)).

Proposition 2.24 states that convolution with a filter φ_{ij} pushes the result more in direction of those eigenvectors for which $|\varphi_{ij}(\lambda)|$ is large. Hence empirically, a better classification result will be reached if all $|\varphi_{ij}(\lambda)|$ are larger at the lower frequencies and close to zero for the higher frequencies. Then convolution with φ_{ij} will *smooth* the learned signals over the graph and dampen noise.

This behavior can be enforced by function spaces designed in the following way.

4.9 Remark. *(Smoothing filter spaces)*
If all basis functions $\varphi^{(k)}$ are *smoothing* in the sense that $|\varphi^{(k)}(\lambda)|$ is larger at the lower frequencies and smaller at the higher frequencies, then all learned filters φ_{ij} reinforce clustering information and the output of each graph convolutional layer will be at least as smooth as the input in the sense of Section 2.3.3.

The effect of filter scaling on the training procedure. The spanned function space of course does not depend on the scale of the basis functions. Replacing any $\varphi^{(k)}$ by $\alpha\varphi^{(k)}$, where α is a scalar factor, does not change the set of achievable filters and thus the set of functions representable by the neural network. The scaling can be counteracted by multiplying all corresponding weights $W^{(l,k)}$ with $1/\alpha$. Hence the optimal solution of (4.8) is not affected, up to a constant factor for a subset of parameters.

In practice, however, the exact global optimum solution is never reached during training. The sequence of parameter iterates, especially during the larger steps in the first few epochs when gradients are still steeper, does depend heavily on the initialization of the parameter vector. Common initialization strategies set up all weight matrix entries randomly according to a uniform or normal distribution with mean zero, and the variance is usually chosen carefully and comparably small depending on the size of the weight matrix. The most popular and successful example is Glorot initialization [39]. Too large entries in the initial parameter values can cause large gradients in unfavorable directions, and the damage caused may be hard to repair in later epochs.

Replacing $\varphi^{(k)}$ by $\alpha\varphi^{(k)}$ corresponds to scaling the initial guess for all entries of $W^{(l,k)}$ by factor α, which may cause problems for the training process. In order to utilize the established empirical results about initializations that work well in practice, we should make sure that the graph convolutional layer operation from (4.13) has a similar effect on the magnitude of the output as a simple perceptron. To quantify this, multiplication with the convolutional matrix should not change the 2-norm of a signal drastically, i.e.,[1]

$$\|\varphi^{(k)}(\mathcal{L}_{\text{sym}})\boldsymbol{f}\|_2 \approx \|\boldsymbol{f}\|_2 \quad \Leftrightarrow \quad \|\varphi^{(k)}(\Lambda)\hat{\boldsymbol{f}}\|_2 \approx \|\hat{\boldsymbol{f}}\|_2 \quad \forall\, \boldsymbol{f}, \hat{\boldsymbol{f}} \in \mathbb{R}^n.$$

Of course, if $\boldsymbol{f}$ is a high-frequency eigenvector of $\mathcal{L}_{\text{sym}}$ or a similarly noisy signal in the sense of Section 2.3.3, then Remark 4.9 states that $\|\varphi^{(k)}(\mathcal{L}_{\text{sym}})\boldsymbol{f}\|_2$ should indeed be much smaller than $\|\boldsymbol{f}\|_2$ because such signals need to be damped. Hence the desired property of $\varphi^{(k)}$ should be that no signal is overly amplified and that there are signals that are *not* damped. This observation leads to the following choice of scaling.

4.10 Remark. *(Scale of the basis filter functions)*
The basis functions $\varphi^{(k)}$ of the filter space ($k = 1, \dots, K$) should be chosen in such a way that the 2-norm of the convolutional matrix is 1, i.e.,

$$\|\varphi^{(k)}(\mathcal{L}_{\text{sym}})\|_2 = \|\varphi^{(k)}(\Lambda)\|_2 = \max_{j=1,\dots,n} |\varphi^{(k)}(\lambda_j)| = 1. \tag{4.14}$$

The role of the filter value at frequency zero. Remark 4.9 essentially states that $|\varphi^{(k)}(\lambda_i)|$ should be proportional to how desirable the eigenvector $\boldsymbol{u}_i$ is in the output of the neural network. A special case is the smallest eigenvalue $\lambda_1 = 0$. Let us here assume that the graph is connected, i.e., the multiplicity of this eigenvalue is one. The corresponding eigenvector is given by $\boldsymbol{u}_1 = \mathcal{D}^{1/2}\mathbf{1} \,/\, \|\mathcal{D}^{1/2}\mathbf{1}\|_2$, where $\|\mathcal{D}^{1/2}\mathbf{1}\|_2 = \sqrt{\text{vol}(G)}$ is the square root of the sum of all node degrees.

[1] The equivalence is due to the fact that $\varphi_k(\mathcal{L}_{\text{sym}}) = U\varphi_k(\Lambda)U^T$, where the orthogonal matrix U does not change the 2-norm of any signal.

If an output signal has a large component in the direction of $\boldsymbol{u}_1$, that means that the signal has similar values in similarly *central* nodes. This means that all outliers with low degrees have similar values as well as all nodes in high-density regions with high degrees, without any influence whether these nodes are connected. This behaviour *might* be valuable for clustering if the different classes vary in node density and the degree is actually an indicator for the class affiliation.

Typically, however, the eigenvector $\boldsymbol{u}_1$ does not contribute explicit clustering information. Then the contribution of the filter value at zero acts like an additional bias term. To illustrate this, we split the filter basis functions φ_k into a zero and nonzero part,

$$\tilde{\varphi}^{(k)}(\lambda) = \begin{cases} 0 & \text{if } \lambda = 0, \\ \varphi^{(k)}(\lambda) & \text{else} \end{cases} \quad \Rightarrow \quad \varphi^{(k)}(\mathcal{L}_{\text{sym}}) = \tilde{\varphi}^{(k)}(\mathcal{L}_{\text{sym}}) + \varphi^{(k)}(0)\boldsymbol{u}_1\boldsymbol{u}_1^T. \tag{4.15}$$

Then the convolutional layer operation can be written as

$$X^{(l)} = \sigma\Bigg(\sum_{k=1}^{K} \tilde{\varphi}^{(k)}(\mathcal{L}_{\text{sym}}) X^{(l-1)} W^{(l,k)} + \boldsymbol{u}_1 \underbrace{\boldsymbol{u}_1^T X^{(l-1)} \sum_{k=1}^{K} \varphi^{(k)}(0) W^{(l,k)}}_{=\boldsymbol{w}^T} + \mathbf{1}\boldsymbol{b}^{(l)T} \Bigg). \tag{4.16}$$

We see that the vector $\boldsymbol{w}$ is added to each node feature fector almost like the bias vector $\boldsymbol{b}^{(l)}$, only scaled by the entries of $\boldsymbol{u}_1$. The vector depends on $X^{(l-1)}$ and all of the layer's weight matrices. However, for a fully trained network in the transductive setting of Definition 4.4(b), both these terms are constant.

In the special case that the graph is regular, i.e., all node degrees are equal, then $\boldsymbol{u}_1$ is a multiple of $\mathbf{1}$ and hence the term $\boldsymbol{u}_1\boldsymbol{w}^T$ can be absorbed by the bias term $\mathbf{1}\boldsymbol{b}^{(l)T}$. In other words, the representable feature maps are the same with the basis functions $\tilde{\varphi}^{(k)}$ as with $\varphi^{(k)}$, because having a nonzero filter value at $\lambda = 0$ only affects the term with $\boldsymbol{w}$. The same effect can be learned by the bias term instead.

4.11 Lemma. *(The zero filter value for regular graphs)*
On a regular undirected graph with constant node degrees, assume a trained graph convolutional network Φ is given with arbitrary basis filter functions $\varphi^{(k)}$ ($k = 1, \dots, K$) and with a fixed input matrix X in the transductive setting of Definition 4.4(b). Then there exists a second graph convolutional network $\tilde{\Phi}$ with the same architecture and weight matrices $W^{(l,k)}$, but filter basis functions $\tilde{\varphi}^{(k)}$ as in (4.15) and updated bias vector $\tilde{\boldsymbol{b}}^{(l)}$, such that the output of the two networks is the same, $\Phi(X) = \tilde{\Phi}(X)$.

Proof. On a regular graph, the node degrees and thus the entries of $\boldsymbol{u}_1$ are constant, $\boldsymbol{u}_1 = \frac{1}{\sqrt{n}}\mathbf{1}$. Hence, for the constant input matrix $X^{(0)} = X$ and its propagation

results $X^{(l)}$, we can define

$$\tilde{\boldsymbol{b}}^{(l)T} = \boldsymbol{b}^{(l)T} + \frac{1}{n}\mathbf{1}^T X^{(l-1)} \sum_{k=1}^{K} \varphi^{(k)}(0) W^{(l,k)}.$$

With this choice, (4.16) yields

$$X^{(l)} = \sigma\Big(\sum_{k=1}^{K} \tilde{\varphi}^{(k)}(\mathcal{L}_{\text{sym}}) X^{(l-1)} W^{(l,k)} + \mathbf{1}\tilde{\boldsymbol{b}}^{(l)T} \Big),$$

which is a graph convolutional layer with updated filters and bias yielding the same output as the original one. □

The lemma means that for regular graphs, the learnable feature maps of a graph convolutional network are independent of the filter value $\varphi^{(k)}(0)$. If this filter value can be chosen freely, one could be tempted to simply determine $\varphi^{(k)}(0) = 0$ because the effect of a nonzero value can simply be learned by the bias term. However, note that the lemma does *not* state that there is no merit at all to having a nonzero filter value $\varphi^{(k)}(0) \neq 0$. Similar to the remarks above about filter scaling, theoretical equivalence of two filter spaces does not imply identical behavior in practice. In this case, the gradient for the implicit bias $\tilde{\boldsymbol{b}}^{(l)}$ is the sum of the gradients for $\boldsymbol{b}^{(l)}$ and all $W^{(l,k)}$. During training, the bias is hence affected by updates for the weight matrices as well as parameters in the previous layers influencing $X^{(l-1)}$. Whether or not this "composite bias" is beneficial or detrimental is hard to predict a priori, but as we'll see in Section 4.4, empirical evidence suggests that some presence of $\boldsymbol{u}_1$ may in fact improve the performance.

Of course, most graphs in practice are not regular. Still, the eigenvector $\boldsymbol{u}_1$ fulfills a similar role as it may be close to constant, but it also carries information about node centrality. All in all, we generally suggest picking a filter space such that filters with $\varphi_{ij}(0) \neq 0$ can be learned.

4.1.6 Brief history of Graph Convolutional Network architectures

To illustrate how the GCN setting is used in practice, we give a few examples of architectures that followed this approach. This section is not intended to give a full overview over all GCN variants. For that we refer to dedicated survey articles [105].

Early iterations of the usage of graph convolution in neural networks were still designed for supervised classification in the spirit of a direct generalization of Convolutional Neural Networks for image classification or segmentation.

4.12 Example. *(Spline-based Spectral Networks on Graphs)*
The first example of a modern Graph Convolutional Network was proposed in [15]. It is based on the combinatorial Laplacian $\mathcal{L}$ and the choice of $\mathcal{P}_K$ as a spline space defined on the whole or partial spectrum of $\mathcal{L}$. The function space dimension K was much larger than is common today, e.g., $K = 64$ in the numerical experiments.

4.13 Example. *(ChebNet)*
The architecture proposed in [24] utilizes a polynomial space with shifted Chebyshev basis polynomials, $\varphi^{(k)}(\lambda) = C_{k-1}(2\lambda/\lambda_{\max} - 1)$ up to degree $p = K - 1$. The authors propose the avoidance of eigenvalue computations as in Remark 4.8 as well as a recursive formula to obtain all $\varphi^{(k)}(\mathcal{L})$ for $k = 1, \dots, K$ with $\mathcal{O}(K)$ applications of the adjacency matrix $\mathcal{A}$. The number of basis functions was still comparably high at $K = 25$.

The true most potent setting for graph convolution in neural networks became apparent when the following method known simply as the Graph Convolutional Network was proposed. We will also refer to this architecture as the "traditional" GCN to distinguish from the general GCN framework of Definition 4.7.

4.14 Example. *(Graph Convolutional Network)*
Under the name Graph Convolutional Network, Kipf et al. [56] proposed the first specific GCN architecture for the use in transductive learning on graphs. It utilizes a one-dimensional filter space spanned by the single basis function $\varphi(\lambda) = 1 - \lambda$ (i.e., $K = 1$). The authors also proposed the addition of loops with weight 1. The resulting convolutional matrix is given by $\varphi^{(1)}(\hat{\mathcal{L}}_{\text{sym}}) = (\mathcal{D} + I)^{-1/2}(\mathcal{A} + I)(\mathcal{D} + I)^{-1/2}$, where $\hat{\mathcal{L}}_{\text{sym}}$ is the Laplacian of the new graph with loops.
In the notation of (4.10), the resulting feature map can be written as

$$\boldsymbol{h}_j^{\text{out}} = \sum_{i \in \mathcal{N}_j \cup \{j\}} \frac{\hat{a}_{ji}}{\sqrt{\hat{d}_i \hat{d}_j}} W^{(l)T} \boldsymbol{h}_i^{\text{in}} = W^{(l)T} \sum_{i \in \mathcal{N}_j \cup \{j\}} \frac{\hat{a}_{ji}}{\sqrt{\hat{d}_i \hat{d}_j}} \boldsymbol{h}_i^{\text{in}}, \tag{4.17}$$

where $\hat{a}_{ji}$ and $\hat{d}_i$,$\hat{d}_j$ are the adjacency weights and node degrees of the graph with loops. We see that each node's new features are a linear combination of the old feature vectors of the node itself and its neighbors. This can be viewed as a special case of the polynomial filters described in Remark 4.8. The localization is restricted to edges already present in the graph data and loops.

Since this breakthrough work and similarly fundamental non-convolutional GNNs like GraphSAGE [45], a big portion of machine learning research has been focused on the advancement, expressivity, and theoretical understanding of neural networks for graph learning.

4.1.7 The role of loops

In the traditional GCN method of Example 4.14 and many related GNN methods [105], the input graph is augmented by loops with a uniform weight of $\hat{a}_{ii} = 1$. In the formulation of (4.17), we see that otherwise the old feature vector $\boldsymbol{h}_i^{\text{out}}$ would not contribute at all to the new feature vector $\boldsymbol{h}_i^{\text{in}}$.

Originally, this has been motivated as a "re-normalization trick" applied to the convolutional matrix $I + \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}$ based on the filter $\varphi(\lambda) = 2 - \lambda$. Later the addition of loops was alternatively interpreted as a smoothing operation [64]. To develop an understanding of this, recall that the entries of the non-normalized Laplacian $\mathcal{L}$ are independent of the loops, but the node degrees with loops, $\hat{d}_i$, are uniformly increased by one, $\hat{d}_i = d_i + 1$. Hence all entries of $\mathcal{L}_{\text{sym}}$ will be slightly closer to zero, which scales down the eigenvalues. In addition, the ratio of the largest node degree over the smallest degree will be closer to one, so relatively speaking, all node degrees will be closer to each other, and thus the eigenvector $\boldsymbol{u}_1$ will be closer to $\mathbf{1}$. For all these reasons, the addition of loops does indeed have a smoothing effect.

In this work we will think of the loops as a part of the graph setup instead of the GNN, so if we are using artificial loops, we expect them to be already contained in the adjacency matrix $\mathcal{A}$. This way, the convolutional matrix of the traditional GCN is just $\varphi(\mathcal{L}_{\text{sym}}) = \mathcal{A}_{\text{sym}}$.

4.2 The Pseudoinverse Graph Convolutional Network architecture

In this section, we motivate, derive, and define our own proposed method based on the GCN architecture explained above.

Throughout this section, we assume that the underlying graph is connected, i.e., the multiplicity of the eigenvalue $\lambda_1 = 0$ of $\mathcal{L}_{\text{sym}}$ is one. This is needed so that $\boldsymbol{u}_1$ spans the nullspace of $\mathcal{L}_{\text{sym}}$. However, we note that all of the concepts in this section can be easily generalized to disconnected graphs by replacing $\boldsymbol{u}_1$ with an orthonormal basis matrix $U_0 \in \mathbb{R}^{n \times K}$ of the nullspace of $\mathcal{L}_{\text{sym}}$, where K is the number of connected components.

4.2.1 Challenges of dense graphs

Our motivation stems from the observation that most common graph neural networks are only targeted at sparse graphs. Recall that *dense* constructed graphs in the sense of Section 2.2.4 have special spectral properties, cf. Section 2.3.4. In this subsection we once more summarize the results from previous sections in light of their influence on the performance of GNNs.

Graph learning has traditionally been targeted at purely data-inherent graphs where not only the nodes describe real-world entities, but also every single edge represents a real-world connection. As stated in Section 2.2.3, such types of graphs usually have *sparse* adjacency matrices and lower algebraic connectivities, cf. 2.16. This is the setting for which GNNs are typically designed [65]. Even in methods like DenseGCN [63], the density only refers to the *computational* graph instead of the data graph, i.e., there are skip connections between all layers.

If we apply these methods to dense graphs anyway, the complexity of the computational cost and storage requirements associated with the adjacency matrix increases from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$, which might be expected to be a decisive issue. That is however often not the case, as the special structure of the constructed adjacency matrices can be exploited to speed up the relevant algorithms.

The true problem is posed by the intrinsically different spectral properties of the dense graph Laplacian. Here the algebraic connectivity or eigengap is typically much larger and almost all eigenvalues are clustered close to 1, as seen in Section 2.3.4 (also cf. [8]).

Most common GNNs reinforce the clustering information contained in the first nonzero eigenvectors (cf. Proposition 2.15) in their feature maps [105]. In *spectral* approaches, this is achieved explicitly via convolution with a spectral filter designed specifically for that purpose. If the eigengap is larger, the informative eigenvectors are considerably harder to extract by existing filters as well as spatial feature maps. Common GNN architectures hence tend to underperform on dense graphs.

Many popular machine learning methods involve repeated multiplication of a feature matrix with some kind of adjacency matrix [65]. This includes the standard GCN from Example 4.14 with the filter function

$$\varphi(\lambda) = 1 - \lambda \quad \Rightarrow \quad \varphi(\mathcal{L}_{\text{sym}}) = \mathcal{A}_{\text{sym}} = \mathcal{D}^{-1/2}\mathcal{A}\mathcal{D}^{-1/2}. \tag{4.18}$$

If all non-zero eigenvalues are close to 1, then $\varphi(\lambda)$ is close to zero for all eigenvalues except the first one. Hence $\mathcal{A}_{\text{sym}}\boldsymbol{f} = U\varphi(\Lambda)U^T\boldsymbol{f}$ will be dominated by the first eigenvector $\boldsymbol{u}_1$ for most $\boldsymbol{f}$, i.e., $\mathcal{A}_{\text{sym}}\boldsymbol{f}$ will almost be a multiple of $\boldsymbol{u}_1$ and close to orthogonal to all other $\boldsymbol{u}_i$ $(i > 1)$. This can be seen via the inner product

$$\boldsymbol{u}_i^T\mathcal{A}_{\text{sym}}\boldsymbol{f} = \varphi(\lambda_i)\boldsymbol{u}_i^T\boldsymbol{f} \begin{cases} = \boldsymbol{u}_1^T\boldsymbol{f} & \text{if } i = 1, \\ \approx 0 & \text{else.} \end{cases}$$

Since all entries of $\boldsymbol{u}_1$ have the same sign, this makes it hard for the network output to contain meaningful clustering information.

As described in Section 4.1.7, many methods utilize loops to reduce the eigenvalues of sparse graphs even further and make the graph even "sparser" from a spectral point of view [64]. For dense graphs, the effect is reduced because the node degrees are

already much larger, and a small reduction of the eigenvalues does not change the spectral properties to any significant extent.

The method that we will introduce now is designed towards dense graphs and large eigengaps, and hence we will not need to add loops.

4.2.2 Pseudoinverse spectral filters

In order to overcome the issues of multiplying with the adjacency matrix, we would like to introduce the *pseudoinverse* of the graph Laplacian, denoted by $\mathcal{L}_{\text{sym}}^{\dagger}$. In the machine learning context, the pseudoinverse has been used, e.g., for online learning [48].

Though there are different definition of pseudoinverses as generalizations of the matrix inverse, we will always refer to the Moore-Penrose inverse.

4.15 Definition. *(Moore-Penrose inverse)*
For any rectangular matrix $A \in \mathbb{R}^{m \times n}$, the Moore-Penrose inverse is defined as the unique matrix $A^{\dagger} \in \mathbb{R}^{n \times m}$ with the properties

- $AA^{\dagger}A = A$,
- $A^{\dagger}AA^{\dagger} = A^{\dagger}$,
- $(AA^{\dagger})^T = AA^{\dagger}$,
- $(A^{\dagger}A)^T = (A^{\dagger}A)$.

A well-known characterization of the pseudoinverse can be obtained from the singular value decomposition (SVD) of the matrix A. This connection can be found, e.g., in [40], and it can be easily proven by verifying that the matrix below satisfies all four Moore-Penrose properties.

4.16 Lemma. *(Connection between the Moore-Penrose inverse and the SVD)*
Let the singular value decomposition of the matrix $A \in \mathbb{R}^{m \times n}$ with rank $r \leq \min\{m, n\}$ be given by $A = U\Sigma V^T$ with the diagonal positive definite matrix $\Sigma \in \mathbb{R}^{r \times r}$ and the matrices with orthonormal columns $U \in \mathbb{R}^{m \times r}$ and $V \in \mathbb{R}^{n \times r}$, $U^TU = V^TV = I$. The Moore-Penrose inverse of A is given by

$$A^{\dagger} = V\Sigma^{-1}U^T.$$

A consequence of this lemma is that the pseudoinverse of the graph Laplacian $\mathcal{L}_{\text{sym}}$ is also the convolutional matrix of a special spectral filter function.

4.17 Lemma. *(Pseudoinverse of the graph Laplacian matrix)*
The pseudoinverse of the graph Laplacian operator $\mathcal{L}_{\text{sym}}$ with eigenvalue decompo-

sition $U\Lambda U^T$ as in Definition 2.13 is equal to the convolutional matrix

$$\mathcal{L}_{\text{sym}}^{\dagger} = \varphi^{\dagger}(\mathcal{L}_{\text{sym}}) = U\varphi^{\dagger}(\Lambda)U^T \quad \text{with} \quad \varphi^{\dagger}(\lambda) = \begin{cases} 0 & \text{if } \lambda = 0, \\ \frac{1}{\lambda} & \text{if } \lambda > 0. \end{cases} \tag{4.19}$$

Proof. By splitting the eigenvalue decomposition into a zero and a positive part,

$$\mathcal{L}_{\text{sym}} = U\Lambda U^T = \begin{bmatrix} \boldsymbol{u}_1 & U_+ \end{bmatrix} \begin{bmatrix} 0 & \\ & \Lambda_+ \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1^T \\ U_+^T \end{bmatrix} = U_+\Lambda_+U_+^T,$$

we obtain the SVD of $\mathcal{L}_{\text{sym}}$ with rank $n-1$ (recall that we assume the graph to be connected). Inserting this SVD into the characterization of Lemma 4.16 yields

$$\mathcal{L}_{\text{sym}}^{\dagger} = U_+\Lambda_+^{-1}U_+^T = U \begin{bmatrix} 0 & \\ & \Lambda_+^{-1} \end{bmatrix} U^T = U \operatorname{diag}\left(0, \frac{1}{\lambda_1}, \ldots, \frac{1}{\lambda_{n-1}}\right) U^T.$$

The proof is concluded by realizing that the right-hand side diagonal matrix is the pseudoinverse of Λ and is holding the filter values of $\varphi^{\dagger}$, $\Lambda^{\dagger} = \varphi^{\dagger}(\Lambda)$. □

This function is decreasing on the nonzero eigenvalues, so low-frequency signals are reinforced and high-frequency signals are damped, following Remark 4.9. Because of $\varphi^{\dagger}(0) = 0$, the first eigenvector $\boldsymbol{u}_1$ is completely removed from the output of the convolution $\mathcal{L}_{\text{sym}}^{\dagger}\boldsymbol{f}$, i.e., that output will always be orthogonal to $\boldsymbol{u}_1$. The problematic behaviour described in Section 4.2.1 is hence completely avoided by the pseudoinverse filter. Note that the decay on the positive eigenvalues can also be achieved by different filters that might warrant investigation in the future.

However, this is a double-edged sword, since often some limited presence of $\boldsymbol{u}_1$ may be beneficial, as stated in Section 4.1.5. Hence we propose custom filters that allow for the combination of the pseudoinverse approach with added $\boldsymbol{u}_1$. This can be achieved by filter functions of the form

$$\varphi_{\alpha,\beta}(\lambda) = \begin{cases} \alpha & \text{if } \lambda = 0, \\ \frac{\beta}{\lambda} & \text{if } \lambda > 0, \end{cases} \tag{4.20}$$

where the parameters α, β can either be assigned manually or learned. The corresponding convolutional matrix is

$$\varphi_{\alpha,\beta}(\mathcal{L}_{\text{sym}}) = U\varphi_{\alpha,\beta}(\Lambda)U^T = \alpha\boldsymbol{u}_1\boldsymbol{u}_1^T + \beta\mathcal{L}_{\text{sym}}^{\dagger}. \tag{4.21}$$

4.2.3 Low-rank approach

In most scenarios we cannot compute the full eigenvalue decomposition of $\mathcal{L}_{\text{sym}}$. However, we have seen in Chapter 3 that there are efficient ways to compute the

smallest few eigenvalues of $\mathcal{L}_{\text{sym}}$. These can be used to approximate the pseudoinverse $\mathcal{L}_{\text{sym}}^{\dagger}$ efficiently.

For the derivation, we first cite a general result from linear algebra. A proof can be found, e.g., in [51, Section 7.4.2].

4.18 Lemma. *(Best low-rank approximation)*
Let $A \in \mathbb{R}^{m\times n}$ be given with the SVD $A = U\Sigma V^T$ as in Lemma 4.16. For a given target rank r, let $U_r \in \mathbb{R}^{m\times r}$, $\Sigma_r \in \mathbb{R}^{r\times r}$, and $V_r \in \mathbb{R}^{n\times r}$ denote the leading submatrices of U, Σ, and V corresponding to the r largest singular values. Then the best rank-r approximation to A in terms of the 2-norm deviation is given by the truncated SVD

$$A \approx A_r = U_r \Sigma_r V_r^T = \operatorname*{arg\,min}_{\substack{B\in\mathbb{R}^{m\times n}\\ \operatorname{rank}(B)\leq r}} \|A - B\|_2.$$

Now, we can use this technique to characterize approximations of spectral filters and their convolutional matrices.

4.19 Lemma. *(Best approximation of spectral filters)*
(a) For a given spectral filter φ, suppose the r eigenvalues $\lambda_{k_1}, \ldots, \lambda_{k_r}$ with largest absolute filter value are known,

$$\big|\varphi(\lambda_{k_1})\big| \geq \ldots \geq \big|\varphi(\lambda_{k_r})\big| > \max_{\substack{k=1,\ldots,n\\ k\notin\{k_1,\ldots,k_r\}}} |\varphi(\lambda_k)|.$$

Then the best rank-r approximation to the convolutional matrix $\varphi(\mathcal{L}_{\text{sym}})$ is given by the low-rank convolutional matrix

$$\varphi(\mathcal{L}_{\text{sym}}) \approx \varphi_r(\mathcal{L}_{\text{sym}}) \quad \text{with} \quad \varphi_r(\lambda) = \begin{cases} \varphi(\lambda) & \text{if } \lambda \in \{\lambda_{k_1}, \ldots, \lambda_{k_r}\}, \\ 0 & \text{else.} \end{cases}$$

(b) For the eigenvalue decomposition of the graph Laplacian, $\mathcal{L}_{\text{sym}} = U\Lambda U^T$, we refer to the submatrices corresponding to the r smallest nonzero eigenvalues as

$$\Lambda_r = \operatorname{diag}(\lambda_2, \ldots, \lambda_{r+1}) \in \mathbb{R}^{r\times r} \quad \text{and} \quad U_r = \big[\boldsymbol{u}_2, \ldots, \boldsymbol{u}_{r+1}\big] \in \mathbb{R}^{n\times r}. \tag{4.22}$$

Then the best rank-r approximation to the pseudoinverse $\mathcal{L}_{\text{sym}}^{\dagger}$ is given by

$$\mathcal{L}_{\text{sym}}^{\dagger} \approx U_r \Lambda_r^{-1} U_r. \tag{4.23}$$

Proof. **(a)** The SVD of $\varphi(\Lambda)$ can be easily verified to be

$$\varphi(\mathcal{L}_{\text{sym}}) = U\varphi(\Lambda)U^T = \begin{bmatrix}\boldsymbol{u}_1 & \cdots & \boldsymbol{u}_n\end{bmatrix} \begin{bmatrix}|\varphi(\lambda_1)| & & \\ & \ddots & \\ & & |\varphi(\lambda_n)|\end{bmatrix} \begin{bmatrix}\pm\boldsymbol{u}_1^T \\ \vdots \\ \pm\boldsymbol{u}_n^T\end{bmatrix},$$

where the signs in the right-hand matrix are set to match the signs of the $\varphi(\lambda_i)$. The largest r singular values of $\varphi(\mathcal{L}_{\text{sym}})$ are hence the $|\varphi(\lambda_{k_i})|$. Using the formula from Lemma 4.18 results in the best rank-r approximation to the convolutional matrix being given by

$$\begin{aligned}\varphi(\mathcal{L}_{\text{sym}}) &\approx \begin{bmatrix}\boldsymbol{u}_{k_1} & \cdots & \boldsymbol{u}_{k_r}\end{bmatrix} \begin{bmatrix}|\varphi(\lambda_{k_1})| & & \\ & \ddots & \\ & & |\varphi(\lambda_{k_r})|\end{bmatrix} \begin{bmatrix}\pm\boldsymbol{u}_{k_1}^T \\ \vdots \\ \pm\boldsymbol{u}_{k_r}^T\end{bmatrix} \\ &= \begin{bmatrix}\boldsymbol{u}_{k_1} & \cdots & \boldsymbol{u}_{k_r}\end{bmatrix} \begin{bmatrix}\varphi(\lambda_{k_1}) & & \\ & \ddots & \\ & & \varphi(\lambda_{k_r})\end{bmatrix} \begin{bmatrix}\boldsymbol{u}_{k_1}^T \\ \vdots \\ \boldsymbol{u}_{k_r}^T\end{bmatrix} = \varphi_r(\mathcal{L}_{\text{sym}})\end{aligned}$$

with the specified low-rank filter φ_r.

(b) Following Lemma 4.16 and its proof, the largest singular values of $\mathcal{L}_{\text{sym}}^\dagger$ are given by the inverse smallest nonzero eigenvalues of $\mathcal{L}_{\text{sym}}$. In combination with Lemma 4.18, this directly shows the desired characterization. Alternatively, part (a) identifies the convolutional matrix for the filter

$$\varphi_r^\dagger(\lambda) = \begin{cases}\frac{1}{\lambda} & \text{if } \lambda_2 \leq \lambda \leq \lambda_{r+1}, \\ 0 & \text{else.}\end{cases} \tag{4.24}$$

as the best rank-r approximation of $\mathcal{L}_{\text{sym}}^\dagger$. □

Equation (4.24) shows that the low-rank approximation of $\mathcal{L}_{\text{sym}}$ is equivalent to a filter cut-off in the frequency domain.

If we replace $\mathcal{L}_{\text{sym}}^\dagger$ with $U_r\Lambda_r^{-1}U_r^T$ in the convolutional matrix for $\varphi_{\alpha,\beta}$ as in (4.21), we obtain

$$\varphi_{\alpha,\beta}(\mathcal{L}_{\text{sym}}) = \alpha\boldsymbol{u}_1\boldsymbol{u}_1^T + \beta\mathcal{L}_{\text{sym}}^\dagger \approx \alpha\boldsymbol{u}_1\boldsymbol{u}_1^T + \beta U_r\Lambda_r^{-1}U_r^T.$$

Similar to Lemma b, the right-hand side approximation is the convolutional matrix of the approximated filter

$$\varphi_{\alpha,\beta}(\mathcal{L}_{\text{sym}}) \approx \varphi_{\alpha,\beta,r}(\mathcal{L}_{\text{sym}}) \quad \text{with} \quad \varphi_{\alpha,\beta,r}(\lambda) = \begin{cases}\alpha & \text{if } \lambda = 0, \\ \frac{\beta}{\lambda} & \text{if } \lambda_1 \leq \lambda \leq \lambda_r, \\ 0 & \text{else.}\end{cases} \tag{4.25}$$

On the one hand, this low-rank approach is necessary to avoid computing the full eigenvalue decomposition. On the other hand, it also has spectral benefits. Following Remark 4.9, the filter basis functions of a GCN should be small in the higher-frequencies. The low-rank filter above accomplishes this in a particularly blunt way. If the rank r is chosen roughly as the number of informative eigenvalues, then all non-informative signals are completely removed from the output. In particular, any high-frequency oscillations are prevented by design.

However, similar to how a positive value at frequency zero may have benefits despite not being necessary from a spectral point of view, the same observation holds for the upper end of the spectrum. It will be beneficial to damp high-frequency signals, but completely projecting them to zero may overdo it. The important part is that we still avoid computing more than the smallest r nonzero eigenvalues.

As a middle ground, we propose to replace β/λ in (4.20) with an additional small constant γ for all $\lambda > \lambda_{r+1}$ instead of setting the filter value to zero. This leads to filters of the form

$$\varphi_{\alpha,\beta,\gamma,r}(\lambda) = \begin{cases} \alpha & \text{if } \lambda = 0, \\ \frac{\beta}{\lambda} & \text{if } \lambda_2 \leq \lambda \leq \lambda_{r+1}, \\ \gamma & \text{if } \lambda > \lambda_{r+1} \end{cases} \tag{4.26}$$

and the convolutional matrix

$$\varphi_{\alpha,\beta,\gamma,r}(\mathcal{L}_{\text{sym}}) = (\alpha - \gamma)\boldsymbol{u}_1\boldsymbol{u}_1^T + U_r(\beta\Lambda_r^{-1} - \gamma I)U_r^T + \gamma I \tag{4.27}$$

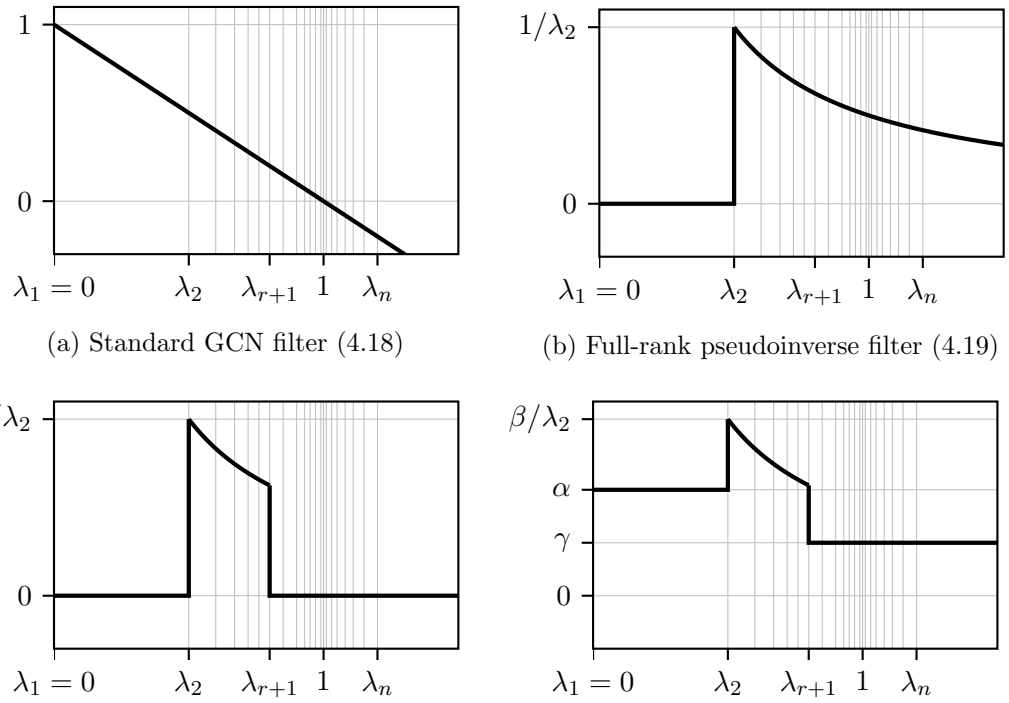
with Λ_r, U_r as in (4.22). Specifically, this setup still only requires the eigenpairs up to λ_{r+1}.

If r is chosen appropriately, this low-rank filter can be argued to perform pseudoinverse convolution on the informative part of the spectrum, while the non-informative eigenvectors – especially noise – are damped uniformly. Hence, the low-rank approximation may very well have positive effects on the accuracy. Since there is no hard boundary between informative and non-informative eigenvalues, there is a wide range of ranks that can yield these benefits.

Figure 4.2 depicts the different filter mentioned throughout this section.

4.2.4 Pseudoinverse filters in Graph Convolutional Networks

We now incorporate our proposed pseudoinverse filters into the GCN framework by picking a convolutional filter space $\mathcal{P}_K$ that contains functions of the form (4.26). One possibility is to choose the parameters α, β, γ, r manually and set the resulting $\varphi_{\alpha,\beta,\gamma,r}$ as the only basis function for a one-dimensional filter space. That requires a-priori identification of the parameter impact on desirable behaviour, which differs from data set to data set. For that reason, this approach bears the risk of lost potential through an adverse parameter choice.



(a) Standard GCN filter (4.18)

(b) Full-rank pseudoinverse filter (4.19)

(c) Low-rank pseudoinverse filter (4.24)

(d) Instance of the three-part filter (4.26)

Figure 4.2: Visualization of selected convolutional filter functions of our proposed form (4.26). The thin vertical lines depict the spectrum of $\mathcal{L}_{\text{sym}}$ for a fictitious example graph.

Instead, we only fix the rank parameter r and have the other parameters learned in training. This can be achieved by the filter basis functions

$$\varphi^{(1)}(\lambda) = \begin{cases} 1 & \text{if } \lambda = 0, \\ 0 & \text{else,} \end{cases}$$
$$\varphi^{(2)}(\lambda) = \begin{cases} \frac{1}{\lambda} & \text{if } \lambda_2 \leq \lambda \leq \lambda_{r+1}, \\ 0 & \text{else,} \end{cases}$$
$$\varphi^{(3)}(\lambda) = \begin{cases} 1 & \text{if } \lambda > \lambda_{r+1}, \\ 0 & \text{else.} \end{cases}$$

Then the individual learned filter functions φ_{ij} as in (4.12) are equal to

$$\varphi_{ij} = \sum_{k=1}^{K} W_{ij}^{(k)} \varphi^{(k)} = \varphi_{\alpha,\beta,\gamma,r} \quad \text{with} \quad \alpha = W_{ij}^{(1)},\ \beta = W_{ij}^{(2)},\ \gamma = W_{ij}^{(3)}. \tag{4.28}$$

The three basis functions can be interpreted as a *zero-impulse filter*, a *low-rank pseudoinverse filter* as in (4.24), and a *high-pass filter*. However, in the current formulation, the scaling of the basis function does not follow Remark 4.10. Recall that we usually postulate that the maximum filter value in any eigenvalue is one, i.e., $\max_i \varphi^{(k)}(\lambda_i) = 1$. The scale of the pseudoinverse part can be fixed by multiplication with the eigengap λ_2. For the high-pass part, we stress once more that the noise eigenvectors should be damped, so the parameter γ in (4.26) should be smaller than the others. We reinforce this behavior by deliberately decreasing the nonzero part of $\varphi^{(3)}$ to a value below one. To align this filter with the pseudoinverse part, we pick the constant equal to the projected value of $\varphi^{(2)}$ in $\lambda = 1$, which is simply the eigengap λ_2.

4.20 Definition. *(Proposed basis functions for the pseudoinverse filter space)*
For the graph convolutional layer from Definition 4.6, the spectral filter space of our proposed Pseudoinverse GCN method is spanned by the three basis functions

$$\varphi^{(1)}(\lambda) = \begin{cases} 1 & \text{if } \lambda = 0, \\ 0 & \text{else,} \end{cases} \qquad \text{(zero-impulse part)} \tag{4.29}$$
$$\varphi^{(2)}(\lambda) = \begin{cases} \frac{\lambda_2}{\lambda} & \text{if } \lambda_2 \leq \lambda \leq \lambda_{r+1}, \\ 0 & \text{else,} \end{cases} \qquad \text{(low-rank pseudoinverse part)} \tag{4.30}$$
$$\varphi^{(3)}(\lambda) = \begin{cases} \lambda_2 & \text{if } \lambda > \lambda_{r+1}, \\ 0 & \text{else.} \end{cases} \qquad \text{(high-pass part)} \tag{4.31}$$

The corresponding convolutional matrices can be set up as

$$\mathcal{K}^{(1)} = \boldsymbol{u}_1\boldsymbol{u}_1^T, \quad \mathcal{K}^{(2)} = \lambda_2 U_r \Lambda_r^{-1} U_r^T, \quad \mathcal{K}^{(3)} = \lambda_2 \left(I - \boldsymbol{u}_1\boldsymbol{u}_1^T - U_r U_r^T\right). \tag{4.32}$$

This feature map can now be embedded in a classical GCN architecture.

4.2.5 Two-layer architecture

We follow the traditional GCN setup for semi-supervised classification in that we use $L = 2$ layers each consisting of the convolutional feature map with an added bias and ReLU activation between the layers. The numbers of channels before and after each layer are given by the feature dimension $N_0 = d$, the *hidden width* hyperparameter $N_1 = h$, and the number of classes $N_2 = m$.

Let $X^{(0)} \in \mathbb{R}^{n\times d}$ be the input feature matrix. Extending the notation by an index for the layer, we get the propagation scheme

$$\begin{aligned} X^{(1)} &= \sigma\Big(\sum_{k=1}^{3} \mathcal{K}^{(k)} X^{(0)} W^{(1,k)} + \mathbf{1} b^{(1)T}\Big), \\ X^{(2)} &= \quad \sum_{k=1}^{3} \mathcal{K}^{(k)} X^{(1)} W^{(2,k)} + \mathbf{1} b^{(2)T}. \end{aligned} \tag{4.33}$$

Here σ is the ReLU function applied element-wise and $W^{(1,k)} \in \mathbb{R}^{d\times h}$, $b^{(1)} \in \mathbb{R}^h$, $W^{(2,k)} \in \mathbb{R}^{h\times m}$, and $b^{(2)} \in \mathbb{R}^m$ are the trainable network parameters ($k = 1, \dots, K$).

Asymptotic cost of layer operations. Similar to $\mathcal{L}_{\text{sym}}^{\dagger}$, the convolutional matrices $\mathcal{K}^{(k)}$ from (4.32) will of course not be sparse. However, we do not have to store and apply the full matrices, but rather exploit their low rank by keeping them in their factorized form and computing the feature map (4.13) via

$$\begin{aligned} Y = \boldsymbol{u}_1 \Big((\boldsymbol{u}_1^T X) W^{(1)} - \lambda_2 \boldsymbol{u}_1^T (X W^{(3)})\Big) \\ + \ \lambda_2 U_r \Big(\Lambda_r^{-1} (U_r^T X) W^{(2)} - U_r^T (X W^{(3)})\Big) \ + \ \lambda_2 X W^{(3)}. \end{aligned} \tag{4.34}$$

This way, the asymptotic cost of a single feature map is $\mathcal{O}(N_{\text{in}} n r)$, where N_{in} is the number of input features. Note that in general, the cost of multiplications with the dense adjacency matrix is already in $\mathcal{O}(N_{\text{in}} n^2)$.

Choice of rank. As stated in Section 4.2.3, the target rank r should be chosen roughly as the number of informative eigenvalues. However, higher ranks may have additional benefits. Typically, the rank choice for low-rank approximations comes down to a trade-off between accuracy and runtime. An intuition about that number

can sometimes be obtained from the presence of a gap in the eigenvalue distribution as in Remark 2.17(d), cf. [98]. An alternative is to view r as a meta-parameter to be determined via cross validation. We investigate its influence in Section 4.4.6. Note that it is very common for GNNs to depend on a few parameters that control some level of approximation.

4.3 Techniques for special cases

The proposed architecture of Section 4.2 is not targeted at any specific Laplacian structure, except for being tailored towards the spectral properties of dense graphs. From a computational point of view though, our approach is agnostic to the underlying structure and method of eigenvalue computation. The only requirement is that we have any method to compute the $r+1$ smallest eigenvalues of the graph Laplacian efficiently.

For the 3D point cloud data sets of Example 2.19, we of course propose to use the Krylov-Schur method in combination with the NFFT-based fast summation scheme as described in Chapter 3. For the citation graphs of Example 2.18, matrix-vector products with the sparse adjacency matrix are equally fast, which also enables the Krylov-Schur method.

4.3.1 Hypergraphs from categorial data

As described in Section 2.2.8 and Example 2.20, categorial attributes can be described by the hyperedges of a hypergraph. This type of data generally leads to small hyperedge sets, $|E| \ll n$. Despite that, the adjacency matrix of the corresponding clique expansion graph is typically dense, which is also reflected in the spectral properties.

In Section 3.3.3, we have seen that the resulting Laplacian has a specific spectral structure where the largest eigenvalue is 1 with multiplicity of at least $n - |E|$, and the remaining eigenvalues can be retrieved from the SVD of the weighted incidence matrix $\tilde{\mathcal{H}} = \mathcal{D}^{-1/2}\mathcal{H}\mathcal{W}^{1/2}\mathcal{B}^{-1/2}$.

If that structure is not exploited, naive computations with the adjacency matrix can become quite expensive. This problem also affects Hypergraph Neural Networks [32], which essentially apply the standard GCN architecture of Example 4.14 to the clique expansion graph. In particular, the added artificial self loops destroy the low-rank structure as shown in Section 3.3.3.

For that reason, we now derive structure-exploiting formulas for the pseudoinverse according to Lemma 4.17.

4.21 Lemma. *(Pseudoinverse of the hypergraph Laplacian)*
Consider a hypergraph with $|E| \ll n$ such that the clique expansion graph is con-

nected. Let the low-rank factor $\tilde{\mathcal{H}}$ of $\mathcal{L}_{\text{sym}}$ be decomposed as in Proposition 3.17, given by

$$\tilde{\mathcal{H}} = \mathcal{D}^{-1/2}\mathcal{H}\mathcal{W}^{1/2}\mathcal{B}^{-1/2} = \tilde{U}\tilde{\Sigma}\tilde{V}^T,$$

so the Laplacian eigenvalue decomposition is given by

$$\mathcal{L}_{\text{sym}} = I - \tilde{\mathcal{H}}\tilde{\mathcal{H}}^T = U\Lambda U \quad \text{with} \quad \Lambda = \begin{bmatrix} \tilde{\Lambda} & \\ & I \end{bmatrix}, \tilde{\Lambda} = I - \tilde{\Sigma}^2.$$

Then the full-rank pseudoinverse of $\mathcal{L}_{\text{sym}}$ is given by

$$\mathcal{L}_{\text{sym}}^{\dagger} = I + \tilde{U}(\tilde{\Lambda}^{\dagger} - I)\tilde{U}^T \quad \text{with} \quad \tilde{\Lambda}^{\dagger} = \operatorname{diag}\left(0, \frac{1}{\lambda_2}, \ldots, \frac{1}{\lambda_{|E|}}\right). \tag{4.35}$$

Proof. With the notation of Proposition 3.17 and Lemma 4.17, we can write the pseudoinverse as

$$\mathcal{L}_{\text{sym}}^{\dagger} = U\varphi^{\dagger}(\Lambda)U^T = U\begin{bmatrix} \varphi^{\dagger}(\tilde{\Lambda}) & \\ & \varphi^{\dagger}(I) \end{bmatrix}U^T.$$

Here $\varphi^{\dagger}(I)$ is the identity due to $\varphi^{\dagger}(1) = 1$, and the pseudoinverse filter values on the remaining eigenvalues are given by

$$\varphi^{\dagger}(\tilde{\Lambda}) = \tilde{\Lambda}^{\dagger} = \operatorname{diag}\left(\varphi^{\dagger}(\lambda_1), \ldots, \varphi^{\dagger}(\lambda_{|E|})\right) = \operatorname{diag}\left(0, \frac{1}{\lambda_2}, \ldots, \frac{1}{\lambda_{|E|}}\right).$$

Put together, we obtain the desired expression

$$\mathcal{L}_{\text{sym}}^{\dagger} = U\left(I + \begin{bmatrix} \tilde{\Lambda}^{\dagger} - I & \\ & 0 \end{bmatrix}\right)U^T = I + \tilde{U}\big(\tilde{\Lambda}^{\dagger} - I\big)\tilde{U}^T. \qquad \square$$

For our low-rank approach, this means that we should not choose $r > |E| - 1$ because that would require singling out a few eigenvectors to the eigenvalue 1, which are indistinguishable. However, computing the first $r + 1 \leq |E|$ eigenvalues of the hypergraph Laplacian becomes much cheaper, since we only need the singular value decomposition of $\tilde{\mathcal{H}}$ or equivalently the eigenvalue decomposition of the $|E| \times |E|$ matrix $\tilde{\mathcal{H}}^T\tilde{\mathcal{H}}$, as described in Section 3.3.3. Thus the setup cost is significantly reduced.

Furthermore, we can recreate the full-rank filter (4.20) within the low-rank setting of (4.26) by fixing $r = |E| - 1$ and $\gamma = \beta$. In theory, this is identical to choosing $r = n - 1$ and thus not using any high-pass part, but it still yields the exploitable low-rank structure. Similar to Definition 4.20, the parameters α and $\beta = \gamma$ can be learned by a GCN architecture with the following two-dimensional filter space.

4.22 Definition. *(Proposed basis functions without separate high-pass part)*
On hypergraphs with $|E| \ll n$, filters of the form (4.20) can be learned in a graph convolutional layer from Definition 4.6 by using the filter space spanned by

$$\varphi^{(1)}(\lambda) = \begin{cases} 1 & \text{if } \lambda = 0, \\ 0 & \text{else,} \end{cases} \qquad \text{(zero-impulse part)}$$

$$\varphi^{(2)}(\lambda) = \begin{cases} \frac{\lambda_2}{\lambda} & \text{if } \lambda \geq \lambda_2, \\ 0 & \text{else.} \end{cases} \qquad \text{(full-rank pseudoinverse part)}$$

The corresponding convolutional matrices can be set up in an efficient manner as

$$\mathcal{K}^{(1)} = \boldsymbol{u}_1 \boldsymbol{u}_1^T, \qquad \mathcal{K}^{(2)} = \lambda_2 \big(I + \tilde{U}(\tilde{\Lambda}^\dagger - I)\tilde{U}^T\big)$$

using the notation of Lemma 4.21.

4.3.2 Hypergraph regularization of sparse graphs

The citation graphs from Example 2.18 are examples of highly sparse graphs with small eigengaps. Hence our method will not be optimally suited to data of this kind.

In all three examples, the node features are binary and sparse, though this property is typically not exploited [56]. However, the features are very similar to the categorial features that form the basis of our hypergraph data sets, cf. Example 2.20. This observation gives rise to the idea to analyze the citation graphs with hypergraph methods.

Other neural network methods for hypergraph are usually tested on the graph benchmarks by converting the citation graphs into so-called *centroid* hypergraphs [32, 101]. This is achieved by adding one hyperedge for each node, containing the node itself and all its neighbors. Under the assumption that the original graph was unweighted and loop-free, this leads to the incidence matrix equal to the original adjacency matrix including loops, $|E| = n$ and $\mathcal{H} = \mathcal{A} + I$. All hyperedges are assigned a hyperedge weight of 1, i.e., $\mathcal{W} = I$.

In this setting, we have a hypergraph describing the citation graph, but not using the categorial node features at all. In the new clique expansion graph, two nodes are now connected if they share membership in at least one hyperedge, which means that the nodes must be connected by a path of length ≤ 2. Hence the clique expansion adjacency matrix is $(\mathcal{A}+I)\mathcal{B}^{-1}(\mathcal{A}+I)$, where $\mathcal{A}$ is the original adjacency matrix. In particular, its sparsity pattern is identical to that of $\mathcal{A}^2 + \mathcal{A} + I$. Since $\mathcal{A}$ is highly sparse, the new clique expansion graph will be a little bit denser but still sparse to an exploitable degree. Each node has now more neighbors but with lower edge weights, in such a way that the new degrees are equal to the original degrees plus one. The number of connected components will be the same. The spectral properties will be

insignificantly more favorable for our pseudoinverse method.

Our proposed technique is to use the binary node feature matrix $X \in \{0,1\}^{n \times d}$ for "categorial regularization". For each of the d categorial attributes, we add one new hyperedge connecting all the nodes for which the feature value is 1. The new hyperedges are assigned a uniform hyperedge weight of $\beta > 0$, which acts like a regularization parameter.

The regularized hypergraph is described by the incidence matrix $\hat{\mathcal{H}}$ and the hyperedge weight matrix $\hat{\mathcal{W}}$, which are set up based on their centroid hypergraph equivalents $\mathcal{H}$ and $\mathcal{W}$ via

$$\hat{\mathcal{H}} = \begin{bmatrix} \mathcal{H} & X \end{bmatrix} = \begin{bmatrix} \mathcal{A} + I & X \end{bmatrix} \quad \text{and} \quad \hat{\mathcal{W}}_E = \begin{bmatrix} \mathcal{W} & \\ & \beta I \end{bmatrix} = \begin{bmatrix} I & \\ & \beta I \end{bmatrix}.$$

Two nodes are now adjacent if they were originally connected by a path of length ≤ 2 or if there is at least one attribute that is true for both nodes.

The new number of hyperedges is $n + d$. In particular, we have $|E| > n$ and the efficient eigenvalue computation of Section 3.3.3 is not feasible. Instead, we need to use the Krylov-Schur from Algorithm 3.7, where matrix-vector products with $\hat{\mathcal{A}} = \hat{\mathcal{H}}\hat{\mathcal{W}}\hat{\mathcal{B}}^{-1}\hat{\mathcal{H}}$ may be implemented in a way that exploits the sparsity of $\hat{\mathcal{H}}$ if $\hat{\mathcal{A}}$ cannot be stored.

Table 4.1 shows how the categorial regularization influences certain properties. We see that the Laplacian becomes much denser both from a spatial and spectral point of view.

4.4 Experiments

We test our proposed method in a series of settings in order to showcase its benefits on dense graphs. The code for our experiments is available online,[2] implemented in Python using PyTorch and PyTorch Geometric [34].

4.4.1 Implementational details

The input features of each data set are propagated through the network architecture (4.33) with the hidden width set to $h = 32$ in all experiments. In the first layer, the products $\mathcal{K}^{(k)}X^{(0)}$ are precomputed as proposed in [18], and in the second layer we employ the efficient scheme from (4.34). Afterwards, the rows of the output $X^{(2)}$ are transformed via the softmax function to gain the predicted class probabilities for each sample. The parameters are trained using the average cross entropy loss of the predicted probabilities for the true class of the training samples. We use the Adam optimizer [55] with learning rate 0.01. During training, we use a dropout rate of

[2] `https://github.com/dominikalfke/PinvGCN`

Table 4.1: Effect of regularization on the citation graphs from Example 2.18. The symbols denote: β – regularization parameter, $|E|$ – number of edges or hyperedges, Dens. – density (ratio of nonzero elements) of $\mathcal{L}_{\text{sym}}$, d_{avg} – average node degree, λ_2 – eigengap of $\mathcal{L}_{\text{sym}}$.

Data set	Graph type	β	$\lvert E\rvert$	Dens.	d_{avg}	λ_2
Citeseer, $n = 3\,327$	Original graph		4 552	0.11%	2.74	0
	Centroid hypergraph		3 327	0.45%	3.74	0
	Regularized hypergraph	0.1	7 030	71.13%	6.90	0.26
		1			35.35	0.51
		10			319.83	0.55
Cora, $n = 2\,708$	Original graph		5 278	0.18%	3.90	0
	Centroid hypergraph		2 708	1.36%	4.90	0
	Regularized hypergraph	0.1	4 141	60.8%	6.72	0.08
		1			23.1	0.44
		10			186.6	0.67
Pubmed, $n = 19\,717$	Original graph		44 324	0.03%	4.50	0.0095
	Centroid hypergraph		19 717	0.30%	5.50	0.019
	Regularized hypergraph	0.1	20 217	99.26%	10.51	0.26
		1			55.61	0.62
		10			506.60	0.74

0.5 between the layers. For the weight matrices $W^{(l,k)}$, we use Glorot initialization [39] and a weight decay factor of $5 \cdot 10^{-4}$, while for the bias vectors $b^{(l)}$, we use zero initialization and no weight decay.

For each run, we set a fixed seed for random number generation, build the model, run 500 training epochs, and finally evaluate the classification accuracy on the non-training samples. We generally perform 100 of these runs for each experimental setting and report averages and standard deviations. Only in a few slow baseline experiments did we reduce the number of runs to 10. All experiments are run on a laptop with an Intel Core i7 processor and an NVIDIA GeForce RTX 2060.

4.4.2 Baseline methods

We refer to our own method as PinvGCN in plots and tables, where we compare it against the following methods on graphs:

- GCN [56], using the implementation from PyTorch Geometric.
- GraphSAGE [45], using the implementation from PyTorch Geometric with mean aggregation. We do not use neighbor sampling since that is designed for node batches in large data sets and it did not improve the results in any of our tests.
- GDC [58], using the implementation from PyTorch Geometric with $\alpha = 0.05$ and top-64 sparsification. Since the data sets where too large for exact matrix inversion, we needed to use the "inexact" version, which is not supported in the original code published with the paper.[3]
- ARMA [11], using the implementation from PyTorch Geometric with parameters $K = 3$ and $T = 2$.

On hypergraphs we additionally tested the following two methods:

- HGNN* [32], which we mark with an asterisk because we use our own implementation that does *not* add loops to the graph (cf. Section 3.3.3) and exploits the low-rank structure of the Laplacian (3.50) in a similar way to our own method, significantly speeding up the runtime.
- HyperGCN [101], using the code published with the paper.[4] Since the fast and non-fast variant yield the same accuracy in our experiments, we only employ "FastHyperGCN with mediators".

[3] `https://github.com/klicperajo/gdc`
[4] `https://github.com/malllabiisc/HyperGCN`

4.4.3 Semi-supervised point classification in 3D point clouds

In order to illustrate our method's superior performance on very dense graphs, we employ the two 3D point clouds from Example 2.19. This data is usually used for 3D point segmentation, where the task is to transfer knowledge from one cloud to other clouds [76]. Instead, we look at the two clouds independently and perform 5-class semi-supervised point classification on each of them by splitting each cloud into its own sets of training points and test points. We use 100 random training points from each of the five classes. For the input feature matrix $X^{(0)}$, we use the original 3D coordinates.

For eigenvalue computation, we use the `fastadj` Python implementation[5] of the NFFT-based fast summation scheme described in Algorithm 3.3 with setup #2 from Example 3.18, combined with the Krylov-Schur eigenvalue computation described in Algorithm 3.7 with tolerance 10^{-3}. These settings are chosen to give fast, rough approximations of the eigenvalues because we found that higher quality did not have any influence on the PinvGCN results.

We conduct experiments for $\sigma \in \{10, 100\}$ in the Gaussian similarity function (3.3), where the larger value amounts to a stronger inclusion of non-local information. Our experimental results are shown in Table 4.2. Since the baselines are not designed for such dense graphs and would have exploding time and memory requirements on the fully connected graph, we only employ them on a k-NN subgraph of the k nearest neighbors, where k is either 10 or 100. Note that it would be possible to employ the GCN baseline on the fully connected graph by utilizing the same NFFT-based fast summation, but providing such an implementation was beyond the scope of our experiments.

To summarize the results, our method produces accurate predictions in reasonable time, comfortably outperforming all baselines. The fact that our accuracy is substantially better for $\sigma = 100$ shows that our method greatly profits from non-local information in non-sparse Laplacians. As we have seen in Table 2.1, adding non-local information via a larger value of σ results in increasing λ_2, confirming the connection between spatial and spectral properties. Especially in the second point cloud, the strengthening of more remote connections appears to be crucial because otherwise λ_2 is too small for our method.

4.4.4 Hypergraph data sets from categorical attributes

We finally employ our method on the three hypergraphs described in Example 2.20, based on the *Mushroom* and *Covertype* data sets from the UCI machine learning repository [27]. These are common benchmarks for semi-supervised classification on

[5] `https://github.com/dominikalfke/FastAdjacency`

Table 4.2: Average classification results on the Oakland data sets from Example 2.19 with 100 training samples per class

	Method	σ	Oakland 1		Oakland 2	
			Time	Accuracy (± SD)	Time	Accuracy (± SD)
10-NN	GCN	10	4.23 s	58.16 % (± 4.39)	5.93 s	68.70 % (± 18.28)
		100	4.24 s	58.42 % (± 4.10)	5.91 s	68.64 % (± 3.28)
	GraphSAGE	–	2.61 s	56.22 % (± 5.57)	3.50 s	71.32 % (± 4.26)
	GDC	–	112.08 s	48.61 % (± 8.50)	1160.33 s	63.25 % (± 25.15)
	ARMA	100	29.23 s	49.63 % (± 2.55)	451.49 s	64.29 % (± 4.41)
100-NN	GCN	10	17.99 s	35.44 % (± 11.87)	44.04 s	66.47 % (± 20.54)
		100	17.67 s	54.17 % (± 6.49)	43.66 s	70.89 % (± 3.27)
	GraphSAGE	–	6.63 s	59.34 % (± 4.94)	13.76 s	70.55 % (± 3.98)
	GDC	–	3446.95 s	37.84 % (± 5.09)	Out of memory	
	ARMA	100	76.02 s	56.04 % (± 4.51)		
PinvGCN, $r = 10$		10	6.70 s	84.64 % (± 4.56)	14.86 s	74.26 % (± 6.39)
		100	5.79 s	92.58 % (± 1.42)	10.82 s	93.25 % (± 1.02)
PinvGCN, $r = 30$		10	11.02 s	81.58 % (± 4.64)	24.62 s	74.18 % (± 6.61)
		100	11.08 s	93.13 % (± 1.68)	24.03 s	94.91 % (± 0.93)
PinvGCN, $r = 100$		10	29.80 s	81.70 % (± 5.02)	69.54 s	73.29 % (± 6.18)
		100	28.93 s	**93.50 %** (± 1.83)	67.70 s	**95.38 %** (± 0.90)

hypergraphs [47, 101]. As in [101], we use the hypergraph incidence matrix as input features for all three data sets, $X^{(0)} = \mathcal{H}$.

Classification results on all three data sets are listed in Table 4.3. We only employ our Pseudoinverse GCN with the maximum rank $r = |E| - 1$, as will be supported in Section 4.4.6. For the graph-based methods, we use k-NN sparsification of the clique expansion graph, so each node is connected to the $k = 10$ other nodes with highest shared hyperedge membership. On the Mushroom hypergraph, our results are in the same order of magnitude as the graph baselines, which is better than the other hypergraph methods, but the ARMA network gives superior results. The graph methods perform remarkably well considering the fact that this process of clique expansion sparsification has, to our knowledge, never been discussed in the hypergraph literature. On both Covertype data sets, however, our Pseudoinverse GCN yields the best accuracies.

Choosing the maximum possible rank, $r = |E| - 1$ as in Lemma 4.21, gives slightly worse results than $r = 20$ on the Mushroom hypergraph but slightly better results on the Covertype hypergraphs.

In addition, we also emulated the full-rank pseudoinverse filter as described in Definition 4.22. This method is named *PinvGCN without high-pass* in Table 4.3 and it produces comparable results. This shows that in the case that all informative eigenvalues can be computed, the high-pass part is no longer obligatory. However, manually removing this part also does not increase accuracy, so we recommend always using all three filter basis functions.

For further investigation, Figure 4.3 presents the dependency of the misclassification rate (i.e., the complement of the accuracy, here plotted logarithmically) on the number of random training samples. The plots confirm the results of Table 4.3 in principle. We remark that our method has a significantly better convergence rate on the Covertype45 hypergraph.

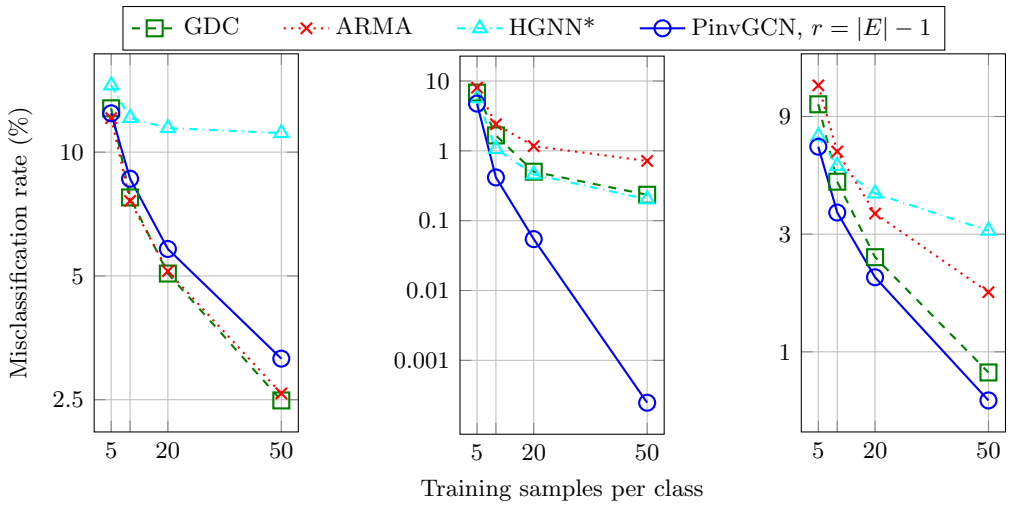
4.4.5 Limitations: Sparse graph data sets

For comparison and transparency, we also apply our method to the standard citation graphs Cora, Citeseer, and Pubmed, introduced in Example 2.18. These graph data sets are typical benchmarks for GNNs. However, they all are examples of sparse graphs that are exactly the opposite of what our method is designed to handle. The main challenge of these examples for our method are the very small eigengaps. Moreover, for two of these graphs, the eigenvalue zero has multiplicity $K > 1$. To make our method applicable nonetheless, we replace the eigenvector $\boldsymbol{u}_1$ with the orthogonal basis matrix $U_0 \in \mathbb{R}^{n \times K}$ of the eigenspace for $\lambda = 0$, and the eigengap is now λ_{K+1} instead of λ_2. This setup requires computing the connected components of the graph or the $r + K$ smallest eigenvalues of $\mathcal{L}_{\text{sym}}$.

Full results are given in Table 4.4. Unsurprisingly, our method by itself produces

Table 4.3: Average classification results on the UCI categorical data sets from Example 2.20 with 10 training samples per class

Data	Method		Runtime	Accuracy (± SD)
Mushroom	Sparsified graph	GCN	2.93 s	91.46 % (± 2.99)
		GraphSAGE	2.61 s	92.23 % (± 3.71)
		GDC	31.98 s	92.24 % (± 3.17)
		ARMA	8.16 s	**92.36 %** (± 3.19)
	HGNN*		0.61 s	83.38 % (± 11.23)
	FastHyperGCN		2.93 s	76.13 % (± 16.38)
	PinvGCN, $r = 20$		3.10 s	91.48 % (± 4.14)
	PinvGCN, $r = \|E\| - 1$		3.22 s	91.38 % (± 3.92)
	PinvGCN without high-pass		2.77 s	91.58 % (± 3.33)
Covertype45	Sparsified graph	GCN	1.49 s	96.68 % (± 2.43)
		GraphSAGE	1.30 s	97.80 % (± 1.76)
		GDC	138.49 s	98.16 % (± 1.76)
		ARMA	9.79 s	96.96 % (± 3.39)
	HGNN*		0.62 s	94.83 % (± 15.75)
	FastHyperGCN		3.51 s	89.81 % (± 8.72)
	PinvGCN, $r = 20$		3.08 s	99.35 % (± 1.01)
	PinvGCN, $r = \|E\| - 1$		3.33 s	**99.58 %** (± 0.71)
	PinvGCN without high-pass		2.71 s	99.56 % (± 0.81)
Covertype67	Sparsified graph	GCN	2.80 s	90.01 % (± 2.63)
		GraphSAGE	3.14 s	92.91 % (± 2.33)
		GDC	417.18 s	94.20 % (± 2.72)
		ARMA	30.39 s	93.53 % (± 2.56)
	HGNN*		1.09 s	91.07 % (± 10.78)
	FastHyperGCN		9.00 s	80.57 % (± 13.30)
	PinvGCN, $r = 20$		3.34 s	96.08 % (± 1.81)
	PinvGCN, $r = \|E\| - 1$		3.55 s	**96.33 %** (± 1.41)
	PinvGCN without high-pass		3.02 s	96.27 % (± 1.43)



(a) Mushroom (b) Covertype45 (c) Covertype67

Figure 4.3: Misclassification rate development over different split sizes on the UCI categorical data sets from Example 2.20

poor results on these data sets and it does not come close to the performance of a traditional GCN [56].

However, if we apply the categorial regularization technique described in Section 4.3.2, the graphs are turned into hypergraphs with much more favorable properties for our method. For each data set, the results for various values of the regularization parameter β are visualized in Figure 4.4 and the best obtained accuracy is listed in Table 4.4. The regularization indeed improves the performance by a large margin and helps our method catch up with state of the art. In practice though, this approach presents its own challenges in the high sensitivity and lack of a heuristic regarding β. It appears that for these citation graphs, the sparse representation contains more useful clustering information than the node features. Our method is better suited to intrinsically dense data instead of relying on "densification".

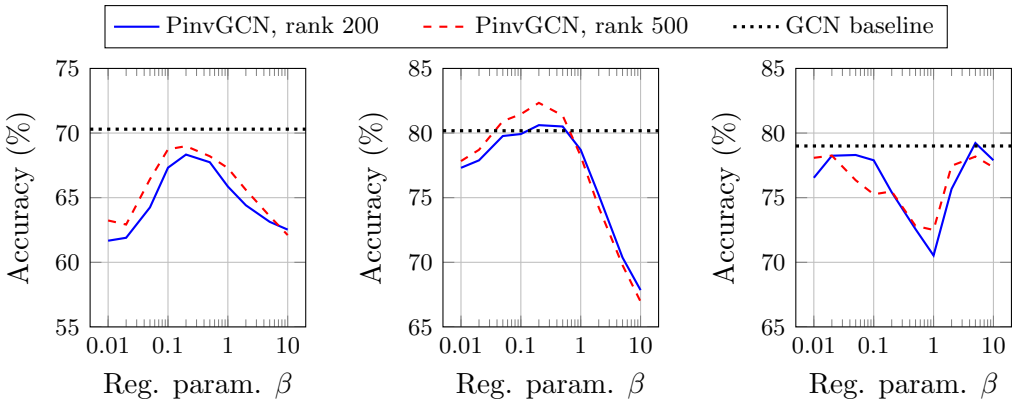
4.4.6 Rank dependency

Since the target rank r is the only hyperparameter of our method, it is important to discuss its impact. As is typical for low-rank approximations, the choice of rank is subject to a trade-off between runtime and accuracy. Figure 4.5 depicts the development of the misclassification rate (plotted logarithmically) and runtime.

We note that the accuracy depends nontrivially on the rank. For almost all data

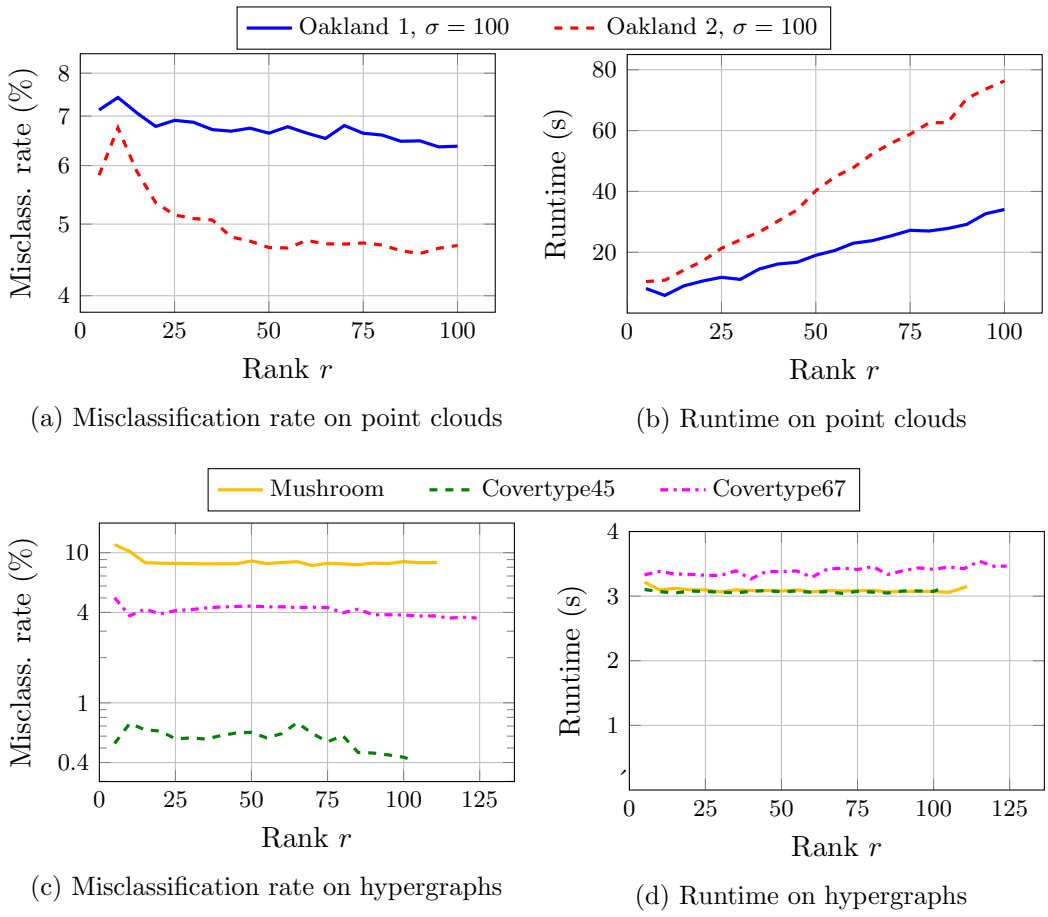
Table 4.4: Average classification results for the sparse citation graphs from Example 2.18, using the standard training splits of each data set

Method	Citeseer	Cora	Pubmed
GCN (as reported in [56])	**70.3 %**	81.5 %	**79.0 %**
PinvGCN, rank 50	61.14 %	70.79 %	70.76 %
PinvGCN, rank 200	60.90 %	71.15 %	71.53 %
PinvGCN, rank 500	61.62 %	71.41 %	71.58 %
PinvGCN, rank 500, best regularization	68.33 %	**82.34 %**	78.26 %



(a) Citeseer (b) Cora (c) Pubmed

Figure 4.4: Average classification results for the sparse citation graphs as in Table 4.4, transformed into centroid hypergraphs with categorical regularization.



(a) Misclassification rate on point clouds

(b) Runtime on point clouds

(c) Misclassification rate on hypergraphs

(d) Runtime on hypergraphs

Figure 4.5: Misclassification rate and runtime development over different ranks r. Top row: Point cloud data sets from Example 2.19 with maximum rank 100. Bottom row: Categorial hypergraph data sets from Example 2.20 with maximum rank $|E|-1$, depending on the data set. Left column: Misclassification rate (complement of the accuracy), plotted logarithmically. Right column: Total setup and training runtime.

sets, the best performance is achieved with the highest tested rank, but the dependency is not monotonic. Especially the hypergraphs show behavior where increasing the rank over a short range slightly deteriorates the accuracy. The point clouds are closer to monotony in this regard. However, the method appears to be very robust with respect to the choice of r, as long as it is not too small ($r < 20$).

The runtime development, on the other hand, depends on the exploited structure of the adjacency matrix. The total effort is governed by the setup cost, while the cost of layer operations does not seem to have a big impact. For point clouds, the number of computed eigenvalues influences the number of Krylov-Schur iterations, which leads to an almost linear dependency on r. For hypergraphs, the SVD of the normalized adjacency matrix is cheap enough to avoid any increase in runtime, leading to almost constant times.

For the best performance, we recommend choosing the maximum $r = |E| - 1$ on hypergraphs without worrying about this parameter. For point clouds, we encourage choosing r as large as possible while keeping a practically feasible computational cost.

4.4.7 Analysis of learned weight entries

As described in Section 4.2.4, our neural network learns the parameters of the filter functions (4.26) in training. The individual learned filters (4.28) may put varying focus on each of the three parts (4.29)–(4.31) as determined by the magnitudes of $W_{ij}^{(1,l)}$, $W_{ij}^{(2,l)}$, $W_{ij}^{(3,l)}$. By forming the averages of the absolute weight entries, we can quantify the importance of each filter part for the trained network. To account for the different weight matrix sizes in each layer, we use the formula

$$\mu_k = \frac{1}{2}\left(\frac{1}{dh}\sum_{i=1}^{d}\sum_{j=1}^{h}\left|W_{ij}^{(k,1)}\right| + \frac{1}{hm}\sum_{i=1}^{h}\sum_{j=1}^{m}\left|W_{ij}^{(k,2)}\right|\right) \quad (k = 1, 2, 3). \tag{4.36}$$

These numbers are furthermore averaged over all 100 runs. Table 4.5 lists these values for multiple PinvGCN instances. We clearly see that the pseudoinverse part consistently is the most important filter basis function, which supports the notion that these eigenvectors carry the most clustering information. The weights of the other two parts are smaller, but not by a large margin, which supports the intuition that the other eigenvectors are still beneficial for the classification result. At the same time, we observe that the entry ratios differ quite a bit between data sets, which implies that is indeed hard to manually choose parameters for one suitable filter function as in (4.26) a priori.

4.4.8 Conclusion

The experimental results confirm the aptitude of our proposed architecture, the Pseudoinverse GCN, for transductive classification tasks on dense graphs where popular

Table 4.5: Average absolute weight matrix entries, μ_k as in (4.36), for the three separate filter parts of (4.26)

Data set	Rank	Part 1 (zero-impulse)	Part 2 (pseudoinverse)	Part 3 (high-pass)
Oakland 1, $\sigma = 100$	100	0.085	0.488	0.193
Oakland 2, $\sigma = 100$		0.247	0.464	0.191
Mushroom	$\lvert E\rvert - 1$	0.075	0.137	0.046
Covertype45		0.053	0.112	0.013
Covertype67		0.034	0.114	0.009

GNNs struggle. In particular, we see that the choice of the three-part filter space improves performance. The network is indeed able to learn different emphasis on each of the three parts as required by the specific data set. A further advantage of our method is the robustness with respect to its only parameter.

5 Dense adjacency matrices in PointNet variants

In addition to the main contributions of the previous chapters, we conclude this thesis with a short proof of concept that showcases that dense adjacency matrices have merit in other applications as well. Contrary to the previous semi-supervised setting, we will now focus on *supervised* learning on 3D point clouds.

In this setting, there are two main tasks: point cloud *classification* and *segmentation*. The former means the assignment of one class label to each point cloud as a whole. For the latter task, each point cloud needs to be *segmented* by assigning a class label to each point in each point cloud [76]. In this chapter, we only focus on the classification task, but the observations and proposed techniques are equally valid for the segmentation task.

The purpose of this chapter is not to invent a novel neural network model that surpasses the state of the art in every single aspect. Instead, we will take a popular baseline methods, point out which parts are computationally expensive, and develop ideas on how they can be accelerated. As is often case, we expect this speedup to come at a price of reduced accuracy. We are hopeful that with further research, our approach will give rise to a full-fledged model that challenges the state of the art.

5.1 Original architectures for point cloud classification

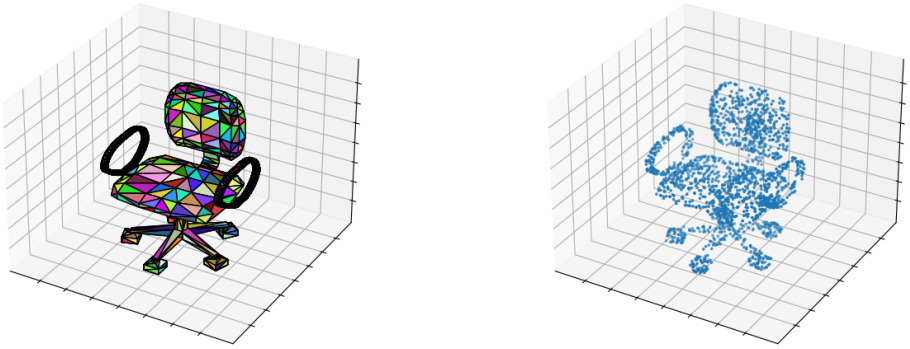
In order to establish the basis for our approach, we first formally introduce the problem formulation of point cloud classification as well as the popular PointNet and PointNet++ methods [84, 85].

5.1.1 Problem setting

In this chapter, we work in the setting of supervised learning on point clouds. As stated in Section 4.1.2, this means that we aim to find a classifier function F that maps each sample to its predicted class.

In our context, the input samples are independent 3D point clouds of a uniform size n.[1] Hence, each sample is described by a matrix $P \in \mathbb{R}^{n\times 3}$ whose rows are the point coordinates in 3D space. We denote the individual points in the rows of P by

[1] The term *sample* here refers to a single instance in the data set. At the same time, the individual points that make up a point cloud sample are sometimes also thought of as samples of the underlying geometry, and their construction will be called *sampling*. The context will clarify the intended meaning.



(a) Original CAD mesh (b) Sampled point cloud

Figure 5.1: Visualization of one exemplary point cloud from the "chair" category in the ModelNet40 data set

$\boldsymbol{p}_i \in \mathbb{R}^3$. As opposed to semi-supervised learning (cf. Section 4.1.3), the input point matrix P is *not* fixed, but only offers a description of one specific sample.

One possible source of such data are the same LiDAR scans that also form the foundation of Example 2.19. For benchmarking purposes, however, the common data sets are constructed from sets of Computer-Aided Design (CAD) models of small everyday objects. Each of these models consists of a set of connected triangles and quadrangles in 3D space which describe the object's surface. The model can be turned into a point cloud by randomly sampling a fixed total number of n points on the faces.

5.1 Example. *(ModelNet data set)*
The ModelNet40 data set is a collection of CAD models of everyday objects from 40 classes. The data set is split into 9 843 training samples and 2 468 test samples. The *supervised classification* task is to detect the category based on a given number of $n = 2\,048$ 3D points randomly sampled on the model surface.
As a smaller benchmark, the ModelNet10 data set refers to a subset of only 10 classes, containing 3 991 training models and 908 test models.

Figure 5.1 depicts a single sample from this data set, belonging to the "chair" category. The original mesh consists of triangles of strongly varying sizes. The sampling process takes this into account to make sure that the points are uniformly distributed over the whole surface, independent of the exact triangulation.

In addition to the point data, it is common to have auxiliary features for each point to give a more detailed description of the geometry. In our setting, these features may be the normal vector of the surface on which the point is located.

5.1.2 The PointNet scheme

Neural networks can be used for the task of point cloud classification as described in Definition 4.3. The specific architecture needs to satisfy a few requirements, e.g., being independent of the ordering of the points. Therefore the design of neural network modules and architectures is a complex task.

We base our investigations on the PointNet architecture and its successor PointNet++. The first version was proposed in [84]. Prior to that, most neural network methods for point cloud learning were based on a shift in data representation onto more regular domains; see the references in [84].

The core idea of PointNet is a combination of two novel neural network modules. First, the so-called "T-net" module maps the input features to an orthogonal rotation matrix, which is then used to transform the features. The goal is to normalize the rotation of all input point clouds. This module was later dropped from PointNet++, so we will not offer a more detailed description.

The second module is a layer that computes new point features $\boldsymbol{y}_j \in \mathbb{R}^{N_{\text{out}}}$ based on a set of old features $\boldsymbol{x}_i \in \mathbb{R}^{N_{\text{in}}}$, where N_{in} and N_{out} are the old and new feature dimensions, respectively. The layer is built around a submodule

$$\Phi^{\text{local}} : \mathbb{R}^{N_{\text{in}}+3} \to \mathbb{R}^{N_{\text{out}}} \tag{5.1}$$

which can be a neural networks like an MLP or just the identity. The input dimension $N_{\text{in}}+3$ stems from a concatenation of the input features with a 3D coordinate vector. For simplicity, we will refer to Φ^{local} as the local neural network (NN) of the layer.

The propagation scheme consists of four steps that are performed for each point $\boldsymbol{p}_j$:

- Find the neighborhood $\mathcal{N}_j$ of the k nearest neighbors of point $\boldsymbol{p}_j$, or alternatively the set of all points within radius ρ.
- For each neighbor $i \in \mathcal{N}_j$, concatenate the features with its relative position w.r.t. $\boldsymbol{p}_j$, given by $\boldsymbol{p}_i - \boldsymbol{p}_j$.
- Pass each concatenated feature vector through the local NN Φ^{local} to obtain new local features.
- Compute the element-wise maximum of the local feature vectors of all neighbors to get the new feature vector of point j.

Note that this method constructs a neighborhood graph in the sense of Section 2.2.5. The choice of the type of neighborhood as well as k or ρ are hyperparameters. Unlike in previous chapters, when using the nearest neighbor technique, there is no need to make sure that the graph is undirected.

The PointNet layer can be summarized as

$$\boldsymbol{y}_j = \max_{i \in \mathcal{N}_j} \Phi^{\text{local}} \left(\begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{p}_i - \boldsymbol{p}_j \end{bmatrix} \right) \in \mathbb{R}^{N_{\text{out}}} \qquad \forall\, j = 1, \dots, n, \tag{5.2}$$

where the maximum is again evaluated element-wise. This can be viewed as a special case for $n' = n$ of the advanced PointNet++ set abstraction layer that will be described in Algorithm 5.1.

5.2 Definition. *(PointNet for ModelNet classification)*
The classical PointNet architecture for classification tasks consists of a sequence of PointNet layers to compute a high-dimensional feature vector $\boldsymbol{x}_i \in \mathbb{R}^N$ for each point $i = 1, \dots, n$, followed by a global maximum pooling layer to compute a single feature vector for the whole point cloud,

$$\boldsymbol{x}_{\max} = \max_{i=1,\dots,n} \boldsymbol{x}_i \in \mathbb{R}^N \qquad \text{(entry-wise maximum)}.$$

That vector is then passed through a final classifier MLP to compute the logit vector $\boldsymbol{y} \in \mathbb{R}^m$, where m is the number of classes.

When a point cloud is passed through this network, we first need to compute the neighborhoods $\mathcal{N}_j$ and can then use this information for all layers.

5.1.3 PointNet++ with Set Abstraction Modules

Only shortly after PointNet [84], the authors published a further advancement based on the same architecture and named the new network PointNet++ [85].

The core methodological innovation is the introduction of *downsampling* steps in each layer. Instead of using the full point set in each PointNet layer and then applying global maximum pooling once in the end, now each layer only computes its output features for a subset of its input points, called *centroids*. This leads to a decreasing sequence of point sets until the final PointNet layer again employs global pooling. Otherwise, the scheme of Equation (5.2) remains the same, i.e., the only difference is that these features are not computed for all j but just a subset.

The centroid subset is chosen via Farthest Point Sampling (FPS), which is argued to lead to a better representation of the whole point cloud when compared to random sampling. The FPS method iteratively picks its centroids as the point with the maximum distance from the current centroid set. In practice, the number of centroids is usually picked comparably large at around half of the input points, which makes the sampling operation quite costly, even with dedicated efficient implementations like the so-called Fast Marching FPS algorithm [71].

The neighborhoods $\mathcal{N}_j$ still need to be set up for each centroid, but not for the remaining points. This process is called *grouping*. Due to the asymmetry, we no longer

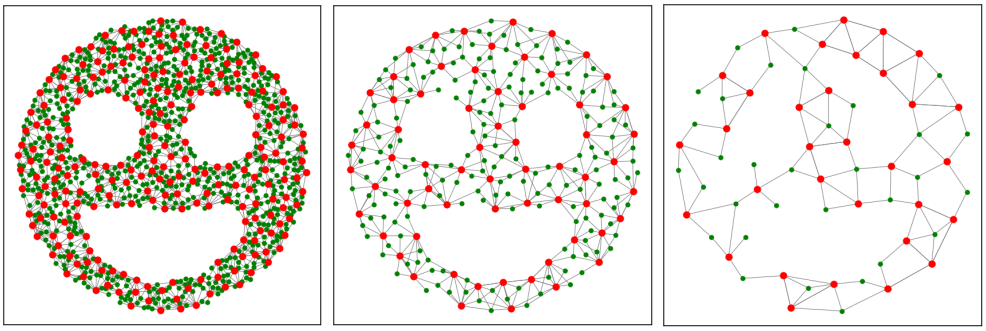


Figure 5.2: Visualization of the point sequence in three consecutive SA modules for a 2D point cloud. Green and red points depict all input points of the layer, the larger red points are chosen as centroids via FPS. The gray edges depict the connections to the k nearest neighbors of each centroid. Left: $n = 1024$, $n' = 256$, $k = 12$. Middle: $n = 256$, $n' = 64$, $k = 8$. Right: $n = 64$, $n' = 32$, $k = 4$.

use the complete neighborhood graph, but only a directed subgraph containing the connections from the neighbors to the centroids.

The combination of sampling, grouping, and the original PointNet operation are called a *set abstraction* (SA) layer. The full propagation scheme for a single such layer is summarized in Algorithm 5.1.

Figure 5.2 depicts the sequence of points and centroids in a chain of three SA modules for a simple 2D point cloud example. The shrinking behavior means that in each module, the adjacency information needs to be computed afresh and cannot be shared between layers. This makes the propagation more expensive than for the original PointNet.

In order to describe a full PointNet architecture, we introduce the following notation for the layers.

5.3 Definition. *(Architecture notation)*
(a) In the following, we will write

$$\mathrm{MLP}(N_0 \to N_1 \to \cdots \to N_L \text{ features})$$

to denote an L-layer sequence of perceptrons as introduced in Section 4.1.1. The input and output width of the l-th layer are given by N_{l-1} and N_l. Hence

$$\mathrm{MLP}(N_0 \to N_1 \text{ features})$$

just denotes a single-layer preceptron. The MLP constitutes a mapping from $\mathbb{R}^{N_0}$

Algorithm 5.1 PointNet++ Set Abstraction (SA) layer

Input: Local NN $\Phi^{\text{local}} : \mathbb{R}^{N_{\text{in}}+3} \to \mathbb{R}^{N_{\text{out}}}$; input points $\boldsymbol{p}_i$ and input features $\boldsymbol{x}_i \in \mathbb{R}^{N_{\text{in}}}$ for all $i = 1, \ldots, n$; number of centroids n'; neighborhood size k or ρ

Output: Subset of points $\mathcal{C} \subseteq \{1, \ldots, n\}$ with $|\mathcal{C}| = n'$, new centroid features $\boldsymbol{y}_j \in \mathbb{R}^{N_{\text{out}}}$ for all $j \in \mathcal{C}$

$\mathcal{C} = \emptyset$

while $|\mathcal{C}| < n'$ **do**

  Find the point that is farthest from any centroid:

$$i = \operatorname*{arg\,max}_{i=1,\ldots,n,\, i \notin \mathcal{C}} \; \min_{j \in \mathcal{C}} \|\boldsymbol{p}_i - \boldsymbol{p}_j\|_2$$

  Add point i as the next centroid: $\mathcal{C} = \mathcal{C} \cup \{i\}$

end while

for each centroid $j \in \mathcal{C}$ **do**

  Collect the neighborhood $\mathcal{N}_j$ of $\boldsymbol{p}_j$ consisting of the k nearest points or all points within radius ρ

  for each neighbor $i \in \mathcal{N}_j$ **do**

    Concatenate the neighbor's features with its relative position w.r.t. $\boldsymbol{p}_j$:

$$\boldsymbol{x}_i^{(j)} = \begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{p}_i - \boldsymbol{p}_j \end{bmatrix} \in \mathbb{R}^{N_{\text{in}}+3}$$

    Pass the concatenated features through the local NN:

$$\boldsymbol{y}_i^{(j)} = \Phi^{\text{local}}(\boldsymbol{x}_i^{(j)}) \in \mathbb{R}^{N_{\text{out}}}$$

  end for

  Perform maximum pooling over the centroid's neighborhood:

$$\boldsymbol{y}_j = \max_{i \in \mathcal{N}_j} \boldsymbol{y}_i^{(j)} \in \mathbb{R}^{N_{\text{out}}} \qquad \text{(entry-wise maximum)}$$

end for

to $\mathbb{R}^{N_L}$. Each layer is equipped with the same activation function σ, including the last layer, unless stated otherwise.

(b) In order to denote an SA layer as in Algorithm 5.1, we write

$$\text{SAModule}(n \to n' \text{ points}, N_0 \to N_1 \to \cdots \to N_L \text{ features}, k \text{ or } \rho),$$

which means that the layer expects n input points and returns n' centroids, while the local NN is given as

$$\Phi^{\text{local}} = \text{MLP}(N_0 \to N_1 \to \cdots \to N_L \text{ features}).$$

We provide both k and ρ to describe the architecture, but only one of these parameters will be used depending on how the adjacency information is set up.

(c) The special case of an SA layer with $n' = 1$ is denoted by

$$\text{GlobalSAModule}(N_0 \to N_1 \to \cdots \to N_L \text{ features}).$$

Here each input point is considered adjacent to the single centroid, which is why k and ρ are not required. The global SA module can be viewed as the combination of an SA layer with the global maximum pooling operation of Definition 5.2.

5.4 Example. *(PointNet++ architecture)*
Consider the supervised classification task for pointclouds with $n = 2048$ input points and m classes. The basic PointNet++ architecture as proposed in [85] consists of the following sequence of modules:

- SAModule($2048 \to 512$ points, $3 + 3 \to 64 \to 64 \to 128$ features, $k = 32$ or $\rho = 0.2$)
- SAModule($512 \to 128$ points, $128 + 3 \to 128 \to 128 \to 256$ features, $k = 32$ or $\rho = 0.4$)
- GlobalSAModule($256 + 3 \to 256 \to 512 \to 1024$ features)
- MLP($1024 \to 512 \to 256 \to m$ features) (no activation after final layer)

The input feature matrix X for the first layer contains the normal vectors of each point. The output is a single vector containing the m logits of the probabilities for each class.
Depending on m, this architecture has a total of $1\,464\,064 + 257m$ parameters in its weight matrices and biases.

5.2 Proposed modifications

Depending on the implementation, the computationally expensive part of PointNet is the setup of the sparse neighborhoods of the k nearest neighbors. Though in theory, the tight localization leads to lower complexity, many practical implementations like PyTorch and their GPU backends are much more optimized for dense matrix multiplications than for index-based computations [54]. This is due to better block-wise storage exploitation on the hardware level.

In order to speed up the implementation, we propose a combination of three modifications: 1. change the architecture such that Φ^{local} is only a single-layer perceptron, 2. replace the local maximum by an addition-based pooling operation, and 3. possibly replace the adjacency matrix by a dense kernel matrix.

5.2.1 Single-layer perceptrons as local components

The behaviour of the PointNet++ layer operation from Algorithm 5.1 is shaped in part by the design of the local NN Φ^{local} from (5.1). Now, we would like to take a look at the special case that Φ^{local} is only a single-layer perceptron, i.e.,

$$\boldsymbol{y}^{(j)} = \Phi^{\text{local}}(\boldsymbol{x}_i^{(j)}) = \sigma\big(W^T \boldsymbol{x}_i^{(j)} + \boldsymbol{b}\big). \tag{5.3}$$

To simplify the effect of Algorithm 5.1, we collect the input points $\boldsymbol{p}_i$ and features $\boldsymbol{x}_i$ in matrices $P \in \mathbb{R}^{n\times 3}$ and $X \in \mathbb{R}^{n\times N_{\text{in}}}$. For a fixed centroid $j \in \mathcal{C}$, we use set indexing to denote the submatrices of neighbor positions $P_{\mathcal{N}_j} \in \mathbb{R}^{\mathcal{N}_j\times 3}$ and neighbor features $X_{\mathcal{N}_j} \in \mathbb{R}^{\mathcal{N}_j\times N_{\text{in}}}$. This way, we can write line 9 of Algorithm 5.1 in matrix notation as

$$X^{(j)} = \big[\boldsymbol{x}_i^{(j)}\big]_{i\in\mathcal{N}_j} = \big[X_{\mathcal{N}_j},\quad P_{\mathcal{N}_j} - \mathbf{1}\boldsymbol{p}_j^T\big] \in \mathbb{R}^{\mathcal{N}_j\times(N_{\text{in}}+3)}.$$

In line 10, Φ^{local} is applied to each $\boldsymbol{x}_i^{(j)}$. With the shallow structure from (5.3), this line can be written similarly to (4.9) as

$$Y^{(j)} = \sigma\Big(X^{(j)}W + \mathbf{1}\boldsymbol{b}^T\Big).$$

By splitting the weight matrix $\mathbb{R}^{(N_{\text{in}}+3)\times N_{\text{out}}}$ into two parts,

$$W = \begin{bmatrix} W^{(X)} \\ W^{(P)} \end{bmatrix} \quad \text{with} \quad W^{(X)} \in \mathbb{R}^{N_{\text{in}}\times N_{\text{out}}} \quad \text{and} \quad W^{(P)} \in \mathbb{R}^{3\times N_{\text{out}}},$$

the above formulas can be combined into

$$\begin{aligned} Y^{(j)} &= \Phi^{\text{local}}\Big(\big[X_{\mathcal{N}_j},\ P_{\mathcal{N}_j} - \mathbf{1}\boldsymbol{p}_j^T\big]\Big) \\ &= \sigma\Big(\big[X_{\mathcal{N}_j},\ P_{\mathcal{N}_j} - \mathbf{1}\boldsymbol{p}_j^T\big]W + \mathbf{1}\boldsymbol{b}^T\Big) \\ &= \sigma\Big(X_{\mathcal{N}_j}W^{(X)} + \big(P_{\mathcal{N}_j} - \mathbf{1}\boldsymbol{p}_j^T\big)W^{(P)} + \mathbf{1}\boldsymbol{b}^T\Big) \\ &= \sigma\Big(\big[X_{\mathcal{N}_j},\ P_{\mathcal{N}_j}\big]\,W + \mathbf{1}\big(-\boldsymbol{p}_j^TW^{(P)} + \boldsymbol{b}^T\big)\Big). \end{aligned}$$

This way, we can detach the centroid position from the neighbor-specific data.

Now, in line 12, the row vector $\boldsymbol{y}_j^T$ can be thought of as the column-wise maximum of the matrix $Y^{(j)}$. We denote this step by $\boldsymbol{y}_j^T = \operatorname{col\,max}\big(Y^{(j)}\big)$. Under the assumption that the activation function σ is monotonically increasing (which is true for all popular choices like the (Leaky) ReLU function, the sigmoid function, and the hyperbolic tangent), the order of pooling and activation can be interchanged and we obtain

$$\begin{aligned} \boldsymbol{y}_j^T &= \operatorname{col\,max}\sigma\Big(\big[X_{\mathcal{N}_j},\ P_{\mathcal{N}_j}\big]\,W + \mathbf{1}\big(-\boldsymbol{p}_j^TW^{(P)} + \boldsymbol{b}^T\big)\Big) \\ &= \sigma\Big(\operatorname{col\,max}\Big(\big[X_{\mathcal{N}_j}, P_{\mathcal{N}_j}\big]\,W + \mathbf{1}\big(-\boldsymbol{p}_j^TW^{(P)} + \boldsymbol{b}^T\big)\Big)\Big). \end{aligned}$$

Moreover, we notice that the row-wise addition of $\big(-\boldsymbol{p}_j^TW^{(P)}+\boldsymbol{b}^T\big)$ does not influence where the column-wise maximum is attained. This means that the latter part can be extracted from the maximization and simply added afterwards, yielding

$$\boldsymbol{y}_j^T = \sigma\Big(\operatorname{col\,max}\big(\big[X_{\mathcal{N}_j}, P_{\mathcal{N}_j}\big]\,W\big) - \boldsymbol{p}_j^TW^{(P)} + \boldsymbol{b}^T\Big) \tag{5.4}$$

This reformulation of the lines 9–12 in Algorithm 5.1 separates the centroid and neighbor parts before the application of W. In other words, different centroids now *share* rows in $\big[X_{\mathcal{N}_j}, P_{\mathcal{N}_j}\big]\,W$, which do not have to be computed multiple times. In order to fully exploit this, we will now move away from maximum pooling.

5.2.2 Add pooling and adjacency matrices

Based on the simplified formula (5.4), we can study the effect of replacing maximum pooling with other pooling operations.

Note that we do *not* propose using a different pooling operation from the start, because then the derivation of Section 5.2.1 is no longer feasible. Specifically, switching pooling and activation as well as extracting the constant row vector is only possible for max pooling.

Instead, we propose to use max pooling to obtain (5.4), and then *approximate* the maximum locally by a different pooling operation.

We first introduce the idea by using *add pooling*, which is of course a bad approximation of max pooling but showcases the benefits of addition-based pooling techniques. With add pooling, output features $\boldsymbol{y}_j$ are computed as the sum of all $\boldsymbol{y}_i^{(j)}$ over all $i \in \mathcal{N}_j$ instead of the entry-wise maximum. This results in the formula

$$\boldsymbol{y}_j^T = \sigma\Big(\mathbf{1}^T \big[X_{\mathcal{N}_j}, P_{\mathcal{N}_j}\big]\, W - \boldsymbol{p}_j^T W^{(P)} + \boldsymbol{b}^T\Big).$$

Now we can utilize that $X_{\mathcal{N}_j}$ is a submatrix of X. Hence, its column sums $\mathbf{1}^T X_{\mathcal{N}_j}$ are masked column sums of X, which can be expressed through the characteristic vector of $\mathcal{N}_j$ via

$$\mathbf{1}^T X_{\mathcal{N}_j} = \boldsymbol{\chi}_{\mathcal{N}_j}^T X \quad \text{with} \quad \boldsymbol{\chi}_{\mathcal{N}_j} \in \mathbb{R}^n, \quad \big(\boldsymbol{\chi}_{\mathcal{N}_j}\big)_i = \begin{cases} 1 & \text{if } i \in \mathcal{N}_j, \\ 0 & \text{else.} \end{cases}$$

Because the same is true for $\mathbf{1}^T P_{\mathcal{N}_j}$, we can write the result as

$$\boldsymbol{y}_j^T = \sigma\Big(\boldsymbol{\chi}_{\mathcal{N}_j}^T \big[X, P\big]\, W - \boldsymbol{p}_j^T W^{(P)} + \boldsymbol{b}^T\Big). \tag{5.5}$$

Here we realize that the centroid index j only appears in the row vectors that are multiplied from the left. This observation reveals that by collecting the vectors $\boldsymbol{y}_j^T$, $\boldsymbol{\chi}_{\mathcal{N}_j}^T$, and $\boldsymbol{p}_j^T$ in the rows of the larger matrices

$$Y = \big[\boldsymbol{y}_j^T\big]_{j\in\mathcal{C}} \in \mathbb{R}^{\mathcal{C}\times N_{\text{out}}}, \quad \mathcal{A} = \big[\boldsymbol{\chi}_{\mathcal{N}_j}^T\big]_{j\in\mathcal{C}} \in \mathbb{R}^{\mathcal{C}\times n}, \quad P_{\mathcal{C}} = \big[\boldsymbol{p}_j^T\big]_{j\in\mathcal{C}} \in \mathbb{R}^{\mathcal{C}\times 3},$$

we obtain the matrix-valued formulation

$$Y = \sigma\Big(\mathcal{A}\, \big[X, P\big]\, W - P_{\mathcal{C}} W^{(P)} + \mathbf{1}\boldsymbol{b}^T\Big) \in \mathbb{R}^{\mathcal{C}\times N_{\text{out}}}. \tag{5.6}$$

The matrix $\mathcal{A}$ can be understood as a rectangular adjacency matrix. We will elaborate on that in Section 5.2.4.

The advantage of this setup is that each row of $[X, P]\, W$ can be computed once and then shared between all centroids, which decreases the number of matrix-vector products with W to n. In the original setup, that number was kn', because for each centroid we needed to transform the concatenated features of its k neighbors. Typical values for these hyperparameters are $n' \in \{n/4, n/2\}$ and $k \in \{16, 32\}$, which means that the shared application of W may reduce the cost by a significant factor.

5.2.3 Other pooling operations

Unfortunately, add pooling is not the best choice for PointNet++. The original work identified max pooling as the key ingredient that enables the method's good results as well as a universal approximation result [84]. As we will see in our experiments, simply using add pooling leads to a significant drop in the prediction quality. Fortunately, there are other addition-based pooling operations that give better approximations of max pooling while allowing us to use the same reformulation as above.

Mean pooling. The first well-known pooling operation that comes to mind is mean or average pooling, where the centroid features are set to the average values of their neighbors' features. For the k-nearest neighbor approach, this is equivalent to add pooling with additional division by k. In general, the number of neighbors can be retrieved from the row sums of the adjacency matrix $\mathcal{A}$, which leads to the degree matrix $\mathcal{D} = \operatorname{diag}(\mathcal{A}\mathbf{1}) \in \mathbb{R}^{\mathcal{C}\times\mathcal{C}}$ and the feature map

$$Y = \sigma\Big(\mathcal{D}^{-1}\mathcal{A}\big[X, P\big]W - P_{\mathcal{C}}W^{(P)} + \mathbf{1}\boldsymbol{b}^T\Big).$$

However, mean pooling does not give significantly better results than add pooling in our case.

Log-Sum-Exp pooling. A well-known trick in computational analysis is the approximation of a maximum by the log-sum-exp (LSE) construct, which makes the result differentiable in all the elements. In general, for a single vector $\boldsymbol{z} = (z_i)_{i=1}^n$, the approximation is defined as

$$z_{\max} = \max_i z_i \approx \operatorname{LSE}_\gamma(\boldsymbol{z}) = \gamma \log\Big(\textstyle\sum_i \exp\big(\frac{1}{\gamma}z_i\big)\Big) = \gamma \log\Big(\mathbf{1}^T \exp\big(\frac{1}{\gamma}\boldsymbol{z}\big)\Big), \quad (5.7)$$

where $\gamma > 0$ is a scalar parameter and log and exp are applied element-wise.

For time series, LSE is used to define SoftDTW, a differentiable approximation to the Dynamic Time Warping (DTW) distance measure [23]. In the machine learning setting, LSE was proposed as a pooling operation in [78]. If we use it to replace col max in (5.4) and use the same tricks as in Section 5.2.2, we obtain the formula

$$Y = \sigma\left(\gamma \log\Big(\mathcal{A}\exp\big(\frac{1}{\gamma}\big[X, P\big]W\big)\Big) - P_{\mathcal{C}}W^{(P)} + \mathbf{1}\boldsymbol{b}^T\right). \quad (5.8)$$

Note that again, the adjacency matrix $\mathcal{A}$ takes the place of $\mathbf{1}^T$ in (5.7).

In exact arithmetic, the LSE term converges against the maximum for $\gamma \to 0$. In finite precision, however, overflow becomes a decisive issue, as $\exp(\frac{1}{\gamma}z_i)$ eventually grows too large to represent. A common trick[2] to improve (5.7) is to shift the data by the maximum,

$$\operatorname{LSE}_\gamma(\boldsymbol{z}) = z_{\max} + \gamma \log\Big(\textstyle\sum_i \exp\big(\frac{1}{\gamma}(z_i - z_{\max})\big)\Big). \quad (5.9)$$

This does not change the result in exact arithmetic while preventing overflow. Underflow, on the other hand, is unproblematic because even if some exponential terms are numerically zero, there is always at least one term where $z_i = z_{\max}$ and thus the summand is one. Hence the argument of the logarithm is always at least one.

[2] cf. the blog post `https://nhigham.com/2021/01/05/what-is-the-log-sum-exp-function/` by Nick Higham.

Utilizing this idea in (5.8) means having a different shift for each column $\boldsymbol{z}$ of $[X, P]\, W$. Apart from that, the shift can be chosen in two ways.

The first option is to use an individual shift for each centroid, accounting for the maximum feature entry in that centroid's neighborhood. However, that invalidates the sharing of features between all centroids because again each centroid needs its own shifted feature matrix, leading to increased computational cost.

The alternative is to shift each column by its global maximum entry. However, for some centroids, the maximum will be attained outside of the neighborhood, so the exponents in (5.9) might be extremely negative. This is not a problem until in the worst case, *all* these exponents cause underflow, causing the logarithm argument to be zero.

Our solution in practice is to choose the shift and the parameter γ dynamically. For each column $\boldsymbol{z}$ of $[X, P]\, W$, we pick an individual value of γ and μ such that the range of $\frac{1}{\gamma}(z_i - \mu)$ is mapped exactly to the interval $[-\eta, \eta]$, where the constant value η is chosen in such a way that $\exp(-\eta)$ and $\exp(\eta)$ can be represented without underflow or overflow. In single precision, for example, we can still represent $\exp(88.5)$ but $\exp(89)$ causes overflow.

L_p-norm pooling. The next pooling operation we want to discuss is based on the L_p-norm, $\|\boldsymbol{z}\|_p = \left(\sum_i z_i^p\right)^{1/p}$, where $p \geq 1$ is a scalar hyperparameter. It is well known that for $p \to \infty$, the norm converges against the maximum absolute entry. Its usage as a pooling operation was proposed in [44], including a shift

$$L_p(\boldsymbol{z}) = z_{\min} + \left(\sum_i (z_i - z_{\min})^p\right)^{1/p}$$

in order to maintain monotony.

Because polynomial growth is less fast than exponential growth, this pooling does not cause the same overflow and underflow issues in practice as LSE pooling. Hence we can simply compute the shift for each column of $[X, P]W$ individually without worrying about the different neighborhoods. The resulting operation is

$$Y = \sigma\left(\mathbf{1}\boldsymbol{z}_{\min}^T + \left(\mathcal{A}\big([X, P]W - \mathbf{1}\boldsymbol{z}_{\min}^T\big)^p\right)^{1/p} - P_{\mathcal{C}} W^{(P)} + \mathbf{1}\boldsymbol{b}^T\right) \tag{5.10}$$

with $\boldsymbol{z}_{\min}^T = \operatorname{col}\min\big[X, P\big]\, W$ and the exponentiation is of course applied element-wise.

Polynomial pooling. The final pooling operation we will study was proposed in [99], also as a smooth approximation of max pooling. It is based on the rational polynomial expression

$$\mathcal{P}_p(\boldsymbol{z}) = z_{\min} + \frac{\sum_i (z_i - z_{\min})^{p+1}}{\sum_i (z_i - z_{\min})^p}$$

utilizing a similar shift as the previous L_p pooling. In the proposing paper, it is argued to lead to a better smooth approximation of max pooling especially in the backwards process.

The resulting feature map is given by

$$Y = \sigma\left(\mathbf{1}\boldsymbol{z}_{\min}^T + \frac{\mathcal{A}\left([X,P]\,W - \mathbf{1}\boldsymbol{z}_{\min}^T\right)^{p+1}}{\mathcal{A}\left([X,P]\,W - \mathbf{1}\boldsymbol{z}_{\min}^T\right)^{p}} - P_{\mathcal{C}}W^{(P)} + \mathbf{1}\boldsymbol{b}^T\right), \tag{5.11}$$

with $\boldsymbol{z}_{\min}$ as in (5.10). Again, exponentiation and division are treated element-wise.

5.2.4 Nonlocal pooling and fully connected adjacency matrices

As stated briefly in Section 5.2.2, the matrix $\mathcal{A}$ with rows $\chi_{\mathcal{N}_j}^T$ from (5.6) can be understood as a rectangular adjacency matrix with entries

$$\mathcal{A}_{ji} = \begin{cases} 1 & \text{if } i \in \mathcal{N}_j \\ 0 & \text{else} \end{cases} \qquad \forall\, j \in \mathcal{C},\, i = 1, \ldots, n. \tag{5.12}$$

For the k-nearest neighbor approach, this neighborhood setup is actually to some degree expensive to compute. Obtaining the distances between centroid $\boldsymbol{p}_j$ and all other points $\boldsymbol{p}_i$ is still comparably cheap in this medium-range size of $n = 2048$. Comparing them all with each other to get the k smallest distances is significantly more expensive, depending on the implementation.

For the ρ-radius neighborhood, the adjacency matrix has the form

$$\mathcal{A}_{ji} = \begin{cases} 1 & \text{if } \|\boldsymbol{p}_i - \boldsymbol{p}_j\|_2 \le \rho, \\ 0 & \text{else.} \end{cases} \qquad \forall\, j \in \mathcal{C},\, i = 1, \ldots, n. \tag{5.13}$$

This is equivalent to the kernel matrix as in (3.1) for the binary indicator kernel with $K(r) = 1$ if $r \le \rho$ and $K(r) = 0$ otherwise.

From there, it is only a small step to allow arbitrary kernel matrices, like, e.g., the Gaussian kernel (3.3), resulting in the adjacency matrix

$$\mathcal{A}_{ji} = \exp\left(-\frac{1}{\sigma^2}\|\boldsymbol{p}_i - \boldsymbol{p}_j\|_2^2\right) \qquad \forall\, j \in \mathcal{C},\, i = 1, \ldots, n. \tag{5.14}$$

Now the adjacency matrix is no longer binary but contains *edge weights* for each connection between a centroid and a "neighbor". This behavior is enabled only by the switch from max pooling to addition-based pooling.

Once again the question is how to choose the Gaussian parameter σ. Recall that a small σ makes the adjacency matrix numerically sparse, while a large σ promotes nonlocal influences over larger distances. We here propose to set σ equal to the radius

parameter ρ. This way, neighbors within distance ρ will be assigned significant weight of order around 1, while the weights for more distant points quickly decay. As stated in Section 2.2.7, points with distance $\gtrsim 4\rho$ will not give any contribution in single precision.

Hence such adjacency matrices will be much denser than the sparse original adjacency matrix to the extent that a sparse storage format does not make sense, similar to fully connected graphs. However, as we will see, the computational cost of multiplication with a dense matrix is not prohibitive for the typical matrix sizes that we encounter in practice. In fact, the cost is to some extent compensated for by the lower setup cost compared to the index setup required for the sparse matrix.

Combination with NFFT-based fast summation. If $\mathcal{A}$ is a Gaussian kernel matrix, the matrix products for (5.6) can of course be approximated by the numerical scheme described in Section 3.1. In our current setting, such an approximation is not practical because $\mathcal{A}$ is just not large enough and the NFFT would not yet yield a time benefit. For future research, it will be interesting to see how supervised learning on larger point clouds can be improved by the fast summation scheme.

5.2.5 SoftPointNet method

As stated before, our approximation of the non-differentiable maximum pooling by means of other, smoother pooling operations is similar to the SoftDTW method as a smooth variant of Dynamic Time Warping [23]. For that reason, we call our approach SoftPointNet. Note that this method is not a full-fledged neural network method but only a tool to showcase the possibilities of utilizing dense adjacency matrices in this specific context.

In order to enable the reformulation of Section 5.2.1, we need to construct a similar architecture to Example 5.4 but each SAModule can only have a single-layer perceptron for its feature propagation. In order to account for that, we use one additional SA layer and interweave the layers with MLPs to allow for more sophisticated feature updates between the pooling operations. We choose the following architecture.

5.5 Example. *(SoftPointNet architecture)*
For our experiments with different pooling operations and adjacency matrices, we use the composition of the following layers:

- SAModule($2048 \to 1024$ points, $3 + 3 \to 64$ features, $k = 32$ or $\rho = 0.2$)
- MLP($64 \to 128$ features)
- SAModule($1024 \to 256$ points, $128 + 3 \to 256$ features, $k = 24$ or $\rho = 0.3$)

- MLP($256 \to 256 \to 256$ features)
- SAModule($256 \to 64$ points, $256 + 3 \to 512$ features, $k = 16$ or $\rho = 0.4$)
- MLP($512 \to 512 \to 512$ features)
- GlobalSAModule($512 + 3 \to 1024$ features)
- MLP($1024 \to 512 \to 256 \to m$ features) (no activation after final layer)

In this architecture, the choice of pooling and adjacency only affects the SAModule layers. The GlobalSAModule, on the other hand, always uses global max pooling in all settings.
Depending on m, this architecture has a total of $2\,032\,896 + 257m$ parameters in its weight matrices and biases.

5.3 Experimental results

5.3.1 Training process & implementation details

Due to the intrinsically different environment when compared to the transductive setting of Chapter 4, the neural network training process needs to be adjusted in a few ways. We follow the setup and hyperparameters of [85] and its implementation.[3] Our code is written in PyTorch with PyTorch Geometric [34].

Since the network input is no longer constant like in Chapter 4, one epoch no longer consists of a single propagation through the network, but of a sequence of training steps where a batch of 32 training point cloud samples are passed through the network and contribute to the loss. The gradients of that loss w.r.t. the network parameters are obtained to perform one step of the Adam optimizer [55]. One epoch consists of as many steps that each training point cloud is passed through the network exactly once. We perform 250 such epochs.

During training, we augment the training set by randomly rotating and "jittering" the input point clouds in the same way as PointNet++. This is achieved by adding a small i.i.d. Gaussian noise to each individual point coordinate, performing a random small rotation of up to $\pm 10°$ around the x and z axes, respectively, and performing one complete random rotation of up to $\pm 180°$ around the y axis.

In order to remove any dependence on the point cloud location and scaling, we prepend a normalization step to each architecture, i.e., all point clouds are moved to the origin and scaled to radius 1 before being propagated through the neural network.

After each epoch, we evaluate the current test performance as the average prediction accuracy on the test set of point clouds. In the end, we report the best

[3] `https://github.com/charlesq34/pointnet2`

accuracy in any epoch. This method will overestimate the network's performance, as in practice the true accuracy is not available and we need some heuristic termination criterion. However, for better comparability to existing methods, we choose the same evaluation procedure as PointNet++.

We start with a learning rate of 10^{-3} and apply an exponential "staircase" decay, i.e., after each 20 epochs, we reduce the learning rate by the factor 0.7 until a minimum value of 10^{-5} is reached. The other parameters of Adam are left at their default values. We furthermore do not use weight decay, but batch normalization after each perceptron application and dropout in the final classifier MLP.

Just like in in Chapter 4, the weight matrices of all perceptrons are initialized by the Glorot method [39] and the biases are initialized as zero.

5.3.2 Experiments on the ModelNet data sets

We now conduct experiments with the ModelNet data from Example 5.1. We use a uniform point cloud size of $n = 2048$ and include the surface normal vectors as point features as described in Section 5.1.1. We don't perform the random point generation ourselves but use the presampled data published along with the official PointNet++ implementation.[4]

We test the architecture described in Example 5.5 with three different notions of adjacency for the matrix $\mathcal{A}$: The k-nearest neighbor approach from (5.12), the ρ-radius approach from (5.13), and the Gaussian adjacency matrix from (5.14). We combine each adjacency method with the pooling operations introduced in Sections 5.2.2 and 5.2.3.

In addition, we also test a baseline implementation of the PointNet method provided by PyTorch Geometric.[5] With this implementation, we test both the original architecture from Example 5.4 and our new proposed SoftPointNet architecture from Example 5.5.

We run our tests in two different environments: For the smaller ModelNet10 subset, we run our tests on a laptop computer with an Intel Core i7 processor and an NVIDIA GeForce RTX 2060. For the full ModelNet40 data set, we run our code on the high-performance cluster of the ZIH at TU Dresden.[6] All reported numbers are averages of five independent runs.

[4] `https://shapenet.cs.stanford.edu/media/modelnet40_normal_resampled.zip`

[5] Based on `https://github.com/rusty1s/pytorch_geometric/blob/master/examples/pointnet2_classification.py` with slight adjustments. Not to be confused with the TensorFlow implementation provided by the authors of the original paper [85], though that implementation uses basically the same techniques, cf. `https://github.com/charlesq34/pointnet2`

[6] The full computational specifications are available at `https://tu-dresden.de/zih/hochleistungsrechnen/hpc`

Table 5.1: Average classification results for the ModelNet10 data set from Example 5.1 with point cloud size $n = 2048$ and normal vector features, obtained on a laptop computer with an Intel Core i7 processor and an NVIDIA GeForce RTX 2060. The best accuracy for each graph type is marked in bold.

Type of adjacency	Pooling	Time per epoch	Test acc.
k-nearest neighbor	max	59.32 s	93.33 %
	add	46.08 s	91.85 %
	mean	47.68 s	91.96 %
	LSE	53.12 s	**93.34 %**
	L_{10}	50.77 s	93.14 %
	$\mathcal{P}_5$	58.58 s	93.61 %
radius ρ	max	26.49 s	**92.95 %**
	add	13.20 s	92.18 %
	mean	13.74 s	92.73 %
	LSE	18.61 s	92.86 %
	L_5	15.76 s	92.51 %
	L_{10}	17.10 s	90.14 %
	$\mathcal{P}_5$	19.26 s	92.73 %
	$\mathcal{P}_{10}$	19.78 s	88.46 %
Gaussian, $\sigma = \rho$	add	17.66 s	91.19 %
	mean	17.81 s	91.30 %
	LSE	19.08 s	92.60 %
	L_5	19.88 s	93.47 %
	L_{10}	20.03 s	92.57 %
	$\mathcal{P}_5$	19.78 s	93.47 %
	$\mathcal{P}_{10}$	18.99 s	92.36 %
Original impl. & architecture (Example 5.4)		178.23 s	93.34 %
Original impl., SoftPointNet architecture		188.66 s	93.01 %

ModelNet10. The results for ModelNet10 are listed in Tables 5.1. Let us first discuss the runtimes, which are the primary motivation of our modifications. The original implementation takes around three minutes per epoch, resulting in a total training time of approximately twelve hours. Switching to the new architecture from Example 5.5 slightly increased the runtime because the number of parameters is larger and the implementation does not exploit the shallow structure as in Section 5.2.1.

The first row in the k-NN section refers to the exact same architecture, but with our own implementation, which reduces the runtime to about a third. The other addition-based pooling operations are able to exploit the structure even better, as described in Sections 5.2.2 and 5.2.3, further reducing the computational cost.

By replacing the k-NN with the ρ-radius approach, the runtimes are further reduced significantly. This is due to the specific implementation in PyTorch, provided by the `torch_cluster` package.[7] The `knn` function returns the exact k nearest points for each centroid, which requires a lot of comparisons between distances. The `radius` function on the other hand also returns at most k points, but not necessarily the k closest ones. It only checks for each point individually if the distance is less than ρ. If more than k points satisfy this, k points are chosen randomly. This way, the matrix $\mathcal{A}$ is at least as sparse as with the k-NN approach, but its setup is much faster. The downside is that the true closest neighbors might not be included in the centroid's computed neighborhood, which affects the prediction performance and stability.

With the fully connected Gaussian adjacency matrix, the runtimes increase slightly again, but not to the level of the k-NN graph. This shows that the sparse neighborhood computation is more expensive than the setup and multiplication with the dense matrix.

The different pooling operations differ slightly in their runtimes. Add and mean pooling are faster because they don't require the additional nonlinear polynomial, exponential, or logarithmic evaluations.

Concerning the prediction accuracy, the best result is obtained by the original architecture. Among the results for the SoftPointNet architecture, the add and mean pooling operations are generally performing worse than mean or LSE pooling, as predicted in Section 5.2.3.

The L_p and $\mathcal{P}_p$ poolings are capable of yielding competitive accuracies but tend to fail in some instances, especially for larger p. This may indicate that the training process hyperparameters need to be adjusted, which should be investigated in future research. This failure is exclusive to the ρ-radius graph, likely due to the additional instability in the neighbor selection process as described above.

The best accuracy in all our experiments is yielded by the Gaussian graph with $\mathcal{P}_5$ pooling, even surpassing the original architecture by a slight margin.

[7] `https://github.com/rusty1s/pytorch_cluster`

Table 5.2: Average classification results for the ModelNet40 data set from Example 5.1 with point cloud size $n = 2048$ and normal vector features, obtained at the ZIH Dresden high-performance cluster. The best accuracy for each graph type is marked in bold.

Type of adjacency	Pooling	Time per epoch	Test acc.
k-nearest neighbor	max	67.09 s	90.84 %
	add	62.71 s	88.53 %
	mean	62.74 s	89.51 %
	LSE	69.78 s	**90.88 %**
	L_{10}	66.31 s	90.64 %
	$\mathcal{P}_5$	67.06 s	90.87 %
radius ρ	max	48.16 s	90.60 %
	add	43.43 s	87.76 %
	mean	43.45 s	87.20 %
	LSE	50.70 s	**90.75 %**
	L_5	46.65 s	13.09 %
	L_{10}	45.86 s	15.07 %
	$\mathcal{P}_5$	47.04 s	8.02 %
	$\mathcal{P}_{10}$	47.27 s	13.44 %
Gaussian, $\sigma = \rho$	add	40.05 s	86.10 %
	mean	40.18 s	86.99 %
	LSE	47.01 s	89.87 %
	L_5	43.84 s	89.26 %
	L_{10}	43.47 s	89.81 %
	$\mathcal{P}_5$	41.51 s	**90.11 %**
	$\mathcal{P}_{10}$	42.09 s	89.99 %
Gaussian, $\sigma = \rho/2$	LSE	47.19 s	90.47 %
	L_5	43.83 s	90.49 %
	L_{10}	43.96 s	**90.63 %**
	$\mathcal{P}_5$	41.99 s	88.14 %
	$\mathcal{P}_{10}$	41.92 s	88.38 %
Original impl. & architecture (Example 5.4)		118.05 s	91.23 %
Original impl., SoftPointNet architecture		129.05 s	90.60 %

ModelNet40. In Table 5.2, we list the results for ModelNet40. As described above, the experiments were run on a high-performance cluster. The results are qualitatively the same as for ModelNet10, except for a few small differences. The relative runtime reduction is not as severe since the slow parts of PointNet++ benefit more from the high performance framework. Still, a total reduction by around 65 % is achieved. One more interesting observation is that the Gaussian graph is now even faster than the ρ-radius graph.

Moreover, the overall accuracy with the Gaussian graph drops a bit compared to the other adjacency models. Unlike on ModelNet10, it is no longer able to match the performance of the k-NN graph, though the differences are small. One possible interpretation is that with the greater number of classes, more detailed local features need to be emphasized, which is harder with the Gaussian graph. For that reason, we also test the scaled Gaussian graph with half the localization parameter, $\sigma = \rho/2$. Indeed, this choice leads to better results that are again almost on par with the k-NN graph, except for $\mathcal{P}_p$ pooling which appears to become more unstable.

The accuracy of the original architecture can however not be reproduced within our SoftPointNet architecture for this data set.

5.3.3 Interpretation and conclusion

The primary focus of our approach is runtime reduction. We have shown that with the PyTorch implementation and the specific point cloud size, multiplication with the fully connected adjacency matrix is not much slower than with a sparse one. On the contrary, the sparse setup via the exact k nearest neighbors is much more expensive than the full setup.

Our experiments make it clear that the original architecture enables slightly better predictions than our own architecture with shallow Φ^{local}, at least on the larger data set ModelNet40. This is not surprising since the PointNet++ architecture was picked as the best empirical choice from a series of architectures in [85], whereas SoftPointNet poses an additional constraint, and the exact sequence of widths in Example 5.5 is not yet optimized. However, it is very well possible that future research reveals a more competitive design that fully matches the capabilities of PointNet++.

Regarding the localization parameter of the Gaussian kernel, we saw first hints that the larger data set ModelNet40 requires a smaller σ, though an extensive investigation is beyond the scope of this first comparison.

All in all, we have shown that our SoftPointNet approach is able to reach competitive prediction results at a reduced cost. A very promising starting point for future research is the possibility of approximating the Gaussian $\mathcal{A}$ with the NFFT-based fast summation algorithm from Section 3.1 in the case of larger point cloud samples. This will once and for all make the Gaussian approach much more efficient than the ρ-radius approach, as that choice of $\mathcal{A}$ will always have multiplication complexity in

$\mathcal{O}(n\,n')$, whereas the NFFT enables $\mathcal{O}(n)$. Better scalability properties are highly sought after in machine learning research.

6 Summary & Theses

The following statements were developed in this thesis and during the preceding years of research.

1. If a data set with low-dimensional feature vectors are described by a fully-connected graph with kernel-based edge weights, matrix-vector products with the adjacency matrix can be approximated efficiently using NFFT-based fast summation. For a fixed accuracy setup, the computational complexity is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. The proportionality factor grows exponentially with the feature dimension.

2. Some data science applications benefit from the non-local information encoded in fully-connected graphs with kernel-based edge weights, especially if the kernel parameters are chosen in such a way that the edge weight decay slowly and the adjacency matrix is numerically dense.

3. If all entries of the adjacency matrix are nonzero and around the same order of magnitude, the eigengap of the normalized graph Laplacian matrix is empirically large and almost all eigenvalues are clustered around 1. Still, the eigenvectors of the smallest nonzero eigenvalues contain useful clustering information just like for sparse graphs.

4. The smallest eigenvalues of the normalized graph Laplacian $\mathcal{L}_{\text{sym}}$ can be computed via the largest eigenvalues of the signless normalized graph Laplacian $\mathcal{L}_{\text{sym}+}$. As the eigenvector $\boldsymbol{u}_1$ for the original eigenvalue 0 is known a priori, the technique of Wielandt deflation can improve the computational scheme. For Krylov subspace methods, this only means additional orthogonalization of all iterates (especially the first one) against $\boldsymbol{u}_1$.

5. If the adjacency matrix of a graph is approximated by another linear operator and this approximation is used to compute the node degrees and subsequently the normalized Laplacian matrix, then the propagated error can be estimated as described in Lemma 3.16.

6. Data sets where each sample is described by a small number of categorial features can be encoded by a hypergraph whose adjacency matrix has low rank. This special structure can be exploited to compute the eigenvalue decomposition of the hypergraph Laplacian in an efficient manner.

7. The Nyström method is a widespread approximation scheme for kernel matrices, but if employed for the normalized Laplacian, the method suffers from disadvantageous error propagation. In case of a low feature dimensions, the generalized *randomized* Nyström method can be combined with NFFT-based fast summation, but this offers no benefits over the combination of NFFT and Krylov methods.

8. For many spectral data science methods that rely on Laplacian eigenvalues and eigenvectors, rough approximations with residual norms of around 10^{-3} to 10^{-2} appear to be sufficient. Greater accuracy often does not improve the final results empirically. However, even less accurate spectral computations may cause failed predictions that deviate significantly.

9. Common Graph Convolutional Network architectures are better suited for sparse graphs than for dense ones. The main reason can be found in the differing spectral properties, as the convolutional basis functions are designed to amplify small eigenvalues that do no exist in dense graphs.

10. Graph Convolutional Networks benefit from an independent "zero-impulse" basis filter function that only handles propagation of signals in the direction of the eigenvector $\boldsymbol{u}_1$. Especially in dense graphs, this allows the network to learn the appropriate level of contribution for this eigenvector without overshadowing the clustering information provided by the next eigenvectors.

11. Desirable non-polynomial convolutional filters can be made feasible in Graph Convolutional Networks by employing a low-rank approximation for the convolutional matrices, can be interpreted as a cutoff of the spectral filters. On top of the computational benefits, this truncation can be motivated as a strict damping of noisy, high-frequency signals(cf. Section 4.2.3). The remaining eigenvectors can still be preserved by using an efficient "high-pass" filter so that the network can separately learn to what degree signals are important.

12. A particularly beneficial non-polynomial convolutional matrix is the pseudoinverse of $\mathcal{L}_{\text{sym}}$, which can be obtained by the filter with reciprocal decay on the non-zero eigenvalues. The resulting PinvGCN architecture yields strong node classification results on dense graphs and hypergraphs.

13. Sparse graphs with additional categorial node features can be "regularized" by transformation into a hypergraph with a dense adjacency matrix and accordingly adjusted spectral properties (cf. Section 4.3.2). However, its aptitude for classification tasks is limited. Often the most amount of clustering information can be extracted from the low number of natural edges instead of the additional artificial edges, at least in common benchmark data sets.

14. In the popular PointNet++ architecture, modifying the local modules to be single-layer perceptrons opens the door for more efficient implementations where part of the neighbor features are shared between centroids (cf. Section 5.2.1), leading to significantly lower runtimes. Though we only tested an unoptimized architecture that led to a slight reduction in accuracy, future research might reveal specific setups that manage to match the original performance at a reduced cost.

15. Based on the previous modification of PointNet++, we can further avoid the expensive computation of the exact k-nearest neighborhoods by using addition-based pooling operations combined with other sparse or dense adjacency matrices. In the typical setting, the point clouds are small enough that fully-connected kernel matrices are cheaper than the k-NN approach. The exact choice of setup still needs tuning.

Bibliography

Own publications

[1] Dominik Alfke, Miriam Gondos, Lucile Peroche, and Martin Stoll. An empirical study of graph-based approaches for semi-supervised time series classification, 2021. Preprint available at arXiv:2104.08153.

[2] Dominik Alfke, Daniel Potts, Martin Stoll, and Toni Volkmer. NFFT meets Krylov methods: Fast matrix-vector products for the graph Laplacian of fully connected networks. *Frontiers in Applied Mathematics and Statistics*, 4, 2018.

[3] Dominik Alfke and Martin Stoll. Pseudoinverse graph convolutional networks: Fast filters tailored for large eigengaps of dense graphs. *Data Mining and Knowledge Discovery*, 35:1318–1341, 2021.

Other references

[4] Robert A. Adams and John J. F. Fournier. *Sobolev Spaces*, volume 140 of *Pure and Applied Mathematics*. Elsevier, 2nd edition, 2003.

[5] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 1st edition, 2018.

[6] Walter E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9:17–29, 1951.

[7] Alex H. Barnett. Aliasing error of the $\exp(\beta\sqrt{1-z^2})$ kernel in the nonuniform fast Fourier transform. *Applied and Computational Harmonical Analysis*, 51:1–16, 2021.

[8] Frank Bauer and Jürgen Jost. Bipartite and neighborhood graphs and the spectrum of the normalized graph Laplacian. *Communications in Analysis and Geometry*, 21(4), 2009.

[9] Kai Bergermann, Martin Stoll, and Toni Volkmer. Semi-supervised learning for aggregated multilayer graphs using diffuse interface methods and fast matrix vector products. *SIAM Journal on Mathematics of Data Science*, 3(2):758–785, 2021.

[10] Andrea L. Bertozzi and Arjuna Flenner. Diffuse interface models on graphs for classification of high dimensional data. *Multiscale Modeling & Simulation*, 10(3):1090–1118, 2012.

[11] Filippo M. Bianchi, Daniele Grattarola, Cesare Alippi, and Lorenzo Livi. Graph Neural Networks with convolutional ARMA filters, 2019. Preprint available at arXiv:1901.01343.

[12] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[13] Jessica Bosch, Steffen Klamt, and Martin Stoll. Generalizing diffuse interface methods on graphs: non-smooth potentials and hypergraphs. *SIAM Journal on Applied Mathematics*, 78(3):1350–1377, 2018.

[14] Alain Bretto. *Hypergraph Theory*. Mathematical Engineering. Springer, 1st edition, 2013.

[15] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *Proceedings of the 2nd International Conference on Learning Representations*, ICLR '14, 2014.

[16] Ben Chamberlain, James Rowbottom, Maria I. Gorinova, Stefan Webb, Emanuele Rossi, and Michael Bronstein. Grand: Graph neural diffusion. In *Proceedings of the 38th International Conference on Machine learning*, ICML '21, 2021.

[17] T.-H. Hubert Chan and Zhibin Liang. Generalizing the hypergraph Laplacian via a diffusion process with mediators. *Theoretical Computer Science*, 806:416–428, 2020.

[18] Jian Jhen Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *Proceedings of the 2nd International Conference on Learning Representations*, ICLR '14, 2018.

[19] Fan R. K. Chung. *Spectral Graph Theory*, volume 92 of *CBMS Regional Conference Series*. AMS Bookstore, 1997.

[20] Fan R. K. Chung. The heat kernel as the pagerank of a graph. *Proceedings of the National Academy of Sciences of the United States of America*, 104(50):19735–19740, 2001.

[21] Bartomeu Coll and Jean-Michel Morel. A non-local algorithm for image denoising. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, volume 2 of *CVPR '05*, pages 60– 65, 07 2005.

[22] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[23] Marco Cuturi and Mathieu Blondel. Soft-DTW: a differentiable loss function for time-series. In *Proceedings of the 34th International Conference on Machine Learning*, ICML '17, pages 894–903, 2017.

[24] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems 29*, NIPS '16, 2016.

[25] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World Wide Web*, WWW '11, pages 577–586. Association for Computing Machinery, 2011.

[26] Petros Drineas and Michael W Mahoney. On the Nyström method for approximating a Gram matrix for improved kernel-based learning. *Journal of Machine Learning Research*, 6:2153–2175, 2005.

[27] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. `http://archive.ics.uci.edu/ml`.

[28] A. Dutt and V. Rokhlin. Fast fourier transforms for nonequispaced data. *SIAM Journal on Scientific Computing*, 14(6):1368–1393, 1992.

[29] Ernesto Estrada and Philip A. Knight. *A First Course in Network Theory*. Oxford University Press, 2015.

[30] Lawrence C. Evans. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 2010.

[31] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '18, 2018.

[32] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yueming Gao. Hypergraph neural networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, AAAI '19, 2019.

[33] Markus Fenn and Gabriele Steidl. Fast NFFT-based summation of radial functions. *Sampling Theory in Signal and Image Processing*, 3:5–28, 2004.

[34] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *International Conference on Learning Representations – Workshop on Representation Learning on Graphs and Manifolds*, ICLR '19, 2019.

[35] Bengt Fornberg and Natasha Flyer. Solving PDEs with radial basis functions. *Acta Numerica*, 24:215–258, 2015.

[36] Charless Fowlkes, Serge Belongie, Fan R. K. Chung, and Jitendra Malik. Spectral grouping using the Nyström method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2):214–225, 2004.

[37] Cristina Garcia-Cardona, Ekaterina Merkurjev, Andrea L. Bertozzi, Arjuna Flenner, and Allon G. Percus. Multiclass data segmentation using diffuse interface methods on graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(8), 2014.

[38] Guy Gilboa and Stanley Osher. Nonlocal operators with applications to image processing. *Multiscale Modeling & Simulation*, 7(3):1005–1028, 2008.

[39] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, AISTATS '10, 2010.

[40] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 3rd edition, 1996.

[41] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[42] Leslie Greenberg and John Strain. The fast Gauss transform. *SIAM Journal of Scientific and Statistical Computations*, 12:79–94, 1991.

[43] Leslie Greenberg and Xiaobai Sun. A new version of the fast Gauss transform. *Documenta Mathematica*, 3:575–584, 1998.

[44] Caglar Gulcehre, Kyunghyun Cho, Razvan Pascanu, and Yoshua Bengio. Learned-norm pooling for deep feedforward and recurrent neural networks. In *Machine Learning and Knowledge Discovery in Databases*, ECML PKDD '14. Springer, 2014.

[45] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30*, NIPS '17, 2017.

[46] Matthias Hein. Uniform convergence of adaptive graph-based regularization. In *Proceedings of the 19th Annual Conference on Learning Theory*, COLT '06, pages 50–64. Springer, 2006.

[47] Matthias Hein, Simon Setzer, Leonardo Jost, and Syama Sundar Rangapuram. The Total Variation on hypergraphs – Learning on hypergraphs revisited. In *Advances in Neural Information Processing Systems 26*, NIPS '13, pages 2427–2435, 2013.

[48] Mark Herbster, Massimiliano Pontil, and Lisa Wainer. Online learning over graphs. In *Proceedings of the 22nd International Conference on Machine learning*, ICML '05, pages 305–312, New York, US, 01 2005.

[49] V. Hernández, J. E. Román, A. Tomás, and V. Vidal. Krylov-Schur methods in SLEPc. Technical report, Universitat Politècnica de Valencia, 2007.

[50] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. *The Annals of Statistics*, 36(3):1171 – 1220, 2008.

[51] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 2nd edition, 1990.

[52] Armin Iske, Sabine Le Borne, and Michael Wende. Hierarchical matrix approximation for kernel-based scattered data interpolation. *SIAM Journal on Scientific Computing*, 39(5):A2287–A2316, 2017.

[53] Jens Keiner, Stefan Kunis, and Daniel Potts. Using NFFT 3 – a software library for various nonequispaced fast Fourier transforms. *ACM Transactions on Mathematical Software*, 36:1–30, 2009.

[54] Volodymyr Kindratenko, editor. *Numerical Computations with GPUs*. Springer, 2014.

[55] Diederik Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, ICLR '15, 2015.

[56] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, 2017.

[57] Steffen Klamt, Utz-Uwe Haus, and Fabian Theis. Hypergraphs and cellular networks. *PLoS Computational Biology*, 5:e1000385–e1000385, 2009.

[58] Johannes Klicpera, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. In *Advances in Neural Information Processing Systems 32*, NIPS '19, 2019.

[59] Stefan Kunis, Daniel Potts, and Gabriele Steidl. Fast Gauss transform with complex parameters using NFFTs. *Journal of Numerical Mathematics*, 14:295–303, 2006.

[60] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255–282, 1950.

[61] Richard B. Lehoucq and Danny C. Sorensen. Deflation techniques for an implicitly restarted Arnoldi iteration. *SIAM Journal on Matrix Analysis and Applications*, 17(4):789–821, 1996.

[62] Richard B. Lehoucq, Danny C. Sorensen, and Chao Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. Software, Environments and Tools. SIAM, 1998.

[63] Guohao Li, Matthias Müller, Ali Thabet, and Bernard Ghanem. DeepGCNs: Can GCNs go as deep as CNNs? In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, ICCV '19, 2019.

[64] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *The Thirty-Second AAAI Conference on Artificial Intelligence*, AAAI '18, 2018.

[65] Zhiyuan Liu and Jie Zhou. *Introduction to Graph Neural Networks*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool, 2020.

[66] Xiyang Luo and Andrea L. Bertozzi. Convergence of the graph Allen–Cahn scheme. *Journal of Statistical Physics*, 167(3):934–958, May 2017.

[67] Per-Gunnar Martinsson. Randomized methods for matrix computations, 2018. Preprint available at arXiv:1607.01649.

[68] Miller McPherson, Lynn Smith-Lovin, and James M. Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 27(1):415–444, 2001.

[69] James Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society A*, 209:441–458, 1909.

[70] Ekaterina Merkurjev, Tijana Kostic, and Andrea L. Bertozzi. An MBO scheme on graphs for classification and image processing. *SIAM Journal on Imaging Sciences*, 6(4), 2013.

[71] Carsten Moenning and Neil A. Dodgson. Fast marching farthest point sampling for implicit surfaces and point clouds. Technical report, University of Cambridge Computer Laboratory, 2003.

[72] Vlad I. Morariu, Balaji V. Srinivasan, Vikas C. Raykar, Ramani Duraiswami, and Larry S. Davis. Automatic online tuning for fast Gaussian summation. In *Advances in Neural Information Processing Systems 21*, NIPS '09, pages 1113–1120, 2009.

[73] Daniel Munoz, J. Andrew Bagnell, Nicolas Vandapel, and Hebert Martial. Contextual classification with functional max-margin Markov networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '09, 2009.

[74] Kevin Patrick Murphy. *Machine Learning*. MIT Press, 2012.

[75] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems 14*, NIPS '01, pages 849–856, 2001.

[76] Anh Nguyen and Bac Le. 3D point cloud segmentation: A survey. In *Proceedings of the 6th IEEE Conference on Robotics, Automation and Mechatronics*, RAM '13, pages 225–230, 2013.

[77] I. U. Ojalvo and M. Newman. Vibration modes of large structures by an automatic matrix-reduction method. *AIAA Journal*, 8(7):1234–1239, 1970.

[78] Pedro O. Pinheiro and Ronan Collobert. From image-level to pixel-level labeling with convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '15, June 2015.

[79] Gerlind Plonka, Daniel Potts, Gabriele Steidl, and Manfred Tasche. *Numerical Fourier Analysis*. Applied and Numerical Harmonic Analysis. Birkhäuser, 2018.

[80] Daniel Potts and Gabriele Steidl. Fast summation at nonequispaced knots by NFFTs. *SIAM Journal on Scientific Computing*, 24:2013–2037, 2003.

[81] Daniel Potts, Gabriele Steidl, and Arthur Nieslony. Fast convolution with radial kernels at nonequispaced knots. *Numerical Mathematics*, 98:329–351, 2004.

[82] Daniel Potts, Gabriele Steidl, and Manfred Tasche. Fast Fourier transforms for nonequispaced data: A tutorial. In *Modern Sampling Theory: Mathematics and Applications*, chapter 12, pages 249–274. Birkhäuser, 2001.

[83] Pulak Purkait, Tat-Jun Chin, Alireza Sadri, and David Suter. Clustering with hypergraphs: The case for large hyperedges. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39:1697–1711, 2017.

[84] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '17, July 2017.

[85] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in Neural Information Processing Systems*, NIPS '17, 2017.

[86] Axel Ruhe. Rational Krylov sequence methods for eigenvalue computation. *Linear Algebra and its Applications*, 58:391–405, 1984.

[87] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, 2nd edition, 2011.

[88] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive Computation and Machine Learning. MIT Press, 2018.

[89] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 07 2004.

[90] David Shuman, Sunil Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs. *IEEE Signal Processing Magazine*, 30:83–98, 2013.

[91] Valeria Simoncini and Daniel B Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM Journal on Scientific Computing*, 25(2):454–477, 2003.

[92] G. W. Stewart. A Krylov-Schur algorithm for large eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, 23(3):601–614, 2001.

[93] Jiliang Tang, Yi Chang, Charu Aggarwal, and Huan Liu. A survey of signed network mining in social media. *ACM Comput. Surv.*, 49(3), 2016.

[94] Yunzhe Tao, Qi Sun, Qiang Du, and Wei Liu. Nonlocal neural networks, nonlocal diffusion and nonlocal modeling. In *Advances in Neural Information Processing Systems 31*, NIPS '18, pages 496–506, 2018.

[95] Yves van Gennip, Nestor Guillen, Braxton Osting, and Andrea L. Bertozzi. Mean curvature, threshold dynamics, and phase field theory on finite graphs. *Milan Journal of Mathematics*, 82(1):3–65, 2014.

[96] Ragav Venkatesan and Baoxin Li. *Convolutional Neural Networks in Visual Computing*. CRC press, 1st edition, 2017.

[97] Toni Volkmer. OpenMP parallelization in the NFFT software library. Technical report, Faculty of Mathematics, Technische Universität Chemnitz, 2012.

[98] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4), 2007.

[99] Zhen Wei, Jingyi Zhang, Li Liu, Fan Zhu, Fumin Shen, Yi Zhou, Si Liu, Yao Sun, and Ling Shao. Building detail-sensitive semantic segmentation networks with polynomial pooling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '19, June 2019.

[100] Christopher Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.

[101] Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha Talukdar. HyperGCN: A new method for training graph convolutional networks on hypergraphs. In *Advances in Neural Information Processing Systems 32*, NIPS '19, 2019.

[102] Changjiang Yang, Ramani Duraiswami, Nail A Gumerov, and Larry Davis. Improved fast Gauss transform and efficient kernel density estimation. In *Proceedings of the 9th IEEE International Conference on Computer Vision*, ICCV '03, page 464. IEEE, 2003.

[103] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: clustering, classification, and embedding. In *Advances in Neural Information Processing Systems 19*, NIPS '06, 2006.

[104] Denny Zhou, Olivier Bousquet, Thomas N. Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. In *Advances in Neural Information Processing Systems 16*, NIPS '03, 2003.

[105] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.