Differentiable Algorithms with Data-driven Parameterization in 3D Vision

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Jan Eric Lenssen

Dortmund

2022

Tag der mündlichen Prüfung:	01. Februar 2022
Dekan:	Prof. Dr. Gernot A. Fink
Gutachter:	Prof. Dr. Heinrich Müller
	Prof. Dr. Kristian Kersting

Acknowledgments

I would like to express gratitude to the following people, who played a crucial role in my work that ultimately led to this thesis. I thank my supervisor Prof. Dr. Heinrich Müller for the support over the recent years, the academic guidance, as well as the scientific freedom he allowed me to have. His supervision enabled me to explore novel research areas, while at the same time assisted me in keeping focused and provided the help when needed. Without his support this thesis would not have been possible. My gratitude also goes to the other members of the committee, Prof. Dr. Kristian Kersting, Prof. Dr. Emmanuel Müller, and Jun. Prof. Dr. Thomas Liebig, for letting my defense be a great moment with interesting discussions.

I also thank all my colleagues from the Computer Graphics Group at TU Dortmund for many fruitful and inspiring discussions. A special thanks goes to Matthias Fey for many successful collaborations over the last four years, which I greatly enjoyed being a part of. His drive, passion, and attention for detail in research and developing is truly inspiring. I thank Christopher Rest and Dr. Denis Fisseler for helpful discussions and for proofreading this thesis, PD Dr. Frank Weichert for always providing the necessary infrastructure and funding for impactful research in our group, and Dr. Pascal Libuschewski for collaboration and guidance in the first years of my PhD studies. I also thank the Collaborative Research Center 876 at TU Dortmund, which partially funded my research.

Further, I would like to express gratitude to all my research collaborators around the globe, who enabled me to perform research at the forefront of Deep Learning. Thanks goes to my collaborators from Nnaisense in Lugano, Facebook Reality Labs in Redmond, the Stanford Geometric Computation group, the Stanford AI Lab, the Mila Institute in Quebec, and Google Zürich.

Last but not least, I deeply thank my friends and family for the support and necessary distractions over the last several years, which enabled me to successfully conclude this chapter of my life. My special gratitude goes to Leo, to Lars, and to my parents Brigitte and Roland, who provided the necessary opportunities, encouraged me to pursue my own path, and created the emotional foundation of my journey.

Abstract

The field of computer vision has recently been dominated by the fast advancements made in the area of differentiable programming. Convolutional Neural Networks (CNNs) are omnipresent in several image analysis tasks, providing expressive features that are useful for a large set of down-stream tasks. Deep Learning (DL) in general, enabled through the vast processing power and parallel nature of current-gen Graphics Processing Units (GPUs), allows us to learn relations between extremely high-dimensional data and high-level features, which often are too complex for the human mind to grasp and to describe manually. Instead of designing specific algorithms down to the last detail, we now define the rules and data flows of how information is processed, taking control at a higher level of abstraction.

In contrast to data in the field of 2D computer vision, which is available in the form of grid-based images in vast quantity, 3D data representations are much more diverse and usually need to be chosen for each task individually, mainly because the cubic complexity of 3D grids does out-scale our current capabilities for storing and processing data. Thus, data types like point clouds, meshes, and more abstract 3D representations like geometric graphs are widely used concepts to solve tasks in 3-dimensional domains. The beneficial properties of DL are not trivially transferred to these irregularly structured representations, as they often come with their own types of inherent symmetries and are not mapped to the GPU as easily as images and CNNs. Additionally, processing these types of data with pure end-to-end methods, thus simply feeding the vectorized data into Deep Neural Networks, does usually not lead to efficient, high quality, or interpretable methods.

This thesis is concerned with designing and analyzing efficient differentiable data flows for representations in the field of 3D vision and applying it to different 3D vision tasks. To this end, the topic is looked upon from the perspective of differentiable algorithms, a more general variant of DL, utilizing the recently emerged tools in the field of differentiable programming. Contributions are made in the sub-fields of Graph Neural Networks (GNNs), differentiable matrix decompositions and Implicit Neural Functions, which serve as important building blocks for differentiable algorithms in 3D vision. The contributions include SplineCNN, a neural network consisting of operators for continuous convolution on irregularly structured data, Local Spatial Graph Transformers (LSGTs), a GNN to infer local surface orientations on point clouds, and a parallel GPU solver for Eigendecomposition (ED) on a large number of symmetric matrices. For all methods, efficient forward and backward GPU implementations are provided. Consequently, two differentiable algorithms are introduced, composed of building blocks from these concept areas. The concepts and algorithms are analyzed with respect to criteria of quality, efficiency, and interpretability.

The first algorithm, Differentiable Iterative Surface Normal Estimation (DISNE), is an iterative algorithm for surface normal estimation on unstructured point clouds. DISNE is an Iterative Re-weighted Least Squares (IRLS) algorithm, solving a large set of weighted least squares problems in parallel on a point graph, for which after each iteration the point pairs are re-weighted using a GNN. It is shown that the algorithm is orders of magnitude faster than previous DL approaches while also improving on their result quality. Additionally, it is easier to interpret than full end-to-end approaches, as intermediate representations can be visualized and analyzed. The second algorithm, Group Equivariant Capsule Networks, is a version of capsule networks grounded in group theory for unsupervised pose estimation and, in general, for inferring disentangled representations from 2D and 3D data. It is provably able to infer poses that are equivariant with respect to transformations of a chosen group. It is shown that the algorithm is able to succeed in the tasks of rotation invariant object classification and unsupervised pose estimation, while providing interpretable representations.

All in all, this thesis concludes that a favorable trade-off in the metrics of efficiency, quality and interpretability can be found by combining prior geometric knowledge about algorithms and data types with the representational power of DL.

Contents

Chapter 1

Introduction

The advent of Deep Learning (DL) marks one of the most important milestones in recent computer science history. The important difference to previous learning methods and also early neural network research is that it enables us to learn complex, non-linear distributions and coherences just from a large amount of gathered data points. The question arises, what made it possible for DL to rise in the recent years, starting with the well-known work about classifying natural images from the ImageNet database in 2012 [KSH12]. The basic methods of todays DL successes were already present very early. The method for iteratively backpropagating errors through arbitrarily connected systems by applying the chain-rule, the product rule and linearity of differentiation was already thought of in 1970 by Linnainmaa [Lin70] and applied to neural networks from 1981 on [Wer81; RHW86]. The same method is still the core of today's neural network training and is an important foundation of this thesis. Similarly, common neural network models were already introduced very early, such as Perceptrons in 1958 by Rosenblatt [Ros58] (as a machine instead of an algorithm at first), the first adaptive Multi-Layer Perceptrons (MLPs) in 1965 [IL65], and Convolutional Neural Networks (CNNs) for image classification in 1989 [LBD+89]. Even more complex architectures with huge practical success today were already introduced in the 1980s and 90s, such as autoencoders in 1986 [RHW86; Bal87] and Long Short-Term Memory (LSTM) for Recurrent Neural Networks (RNNs) in 1997 [HS97]. The history of DL goes back many years and took several interesting turns, as the reader may discover in the exhaustive overview by Schmidhuber [Sch15].

The main difference of today's research to the peak of DL research in the 1980s is the capability of capturing and processing the sheer amount of data that is used to train neural networks, allowing for deeper and semantically richer models. The parallel processing power of modern Graphics Processing Units (GPUs) enables most of these novel and practically relevant applications we see today. Simple data vectors $\mathbf{x} \in \mathbb{R}^n$ are naturally processed using the well-known MLPs, which can be efficiently brought to the GPU as matrix multiplication. The same goes for two-dimensional images with multiple channels $\mathbf{X} \in \mathbb{R}^{x \times y \times c}$ for which CNNs impose a nearly optimal inductive bias for most real world applications, which is efficiently parallelized on the GPU over the image pixels. Through this extremely efficient processing of large real-world datasets, we are able to find relations in data that were unaccessible before. Evidently, DL enabled us to classify general images [KSH12], to model and sample from complex real-world image distributions [GPM+14], to

represent whole 3D scenes as parameters of one MLP [MST+20], to model natural languages [BMR+20], to model physical processes [SCW+15], and much more. Usually, the complexity of the distributions and relations we can model through DL even surpasses what we can understand, making it hard or impossible to interpret what deep neural network models actually do internally. Therefore, the research goals naturally shifted over the past few years. When we formerly tried to understand the problems we want to solve in as much detail as possible, we are now more interested in designing data flows that are able to capture and solve the problem for us, given large datasets with training examples.

This thesis is built upon two premises, from which the first is a consequence of this observation. It states *designing data flows for forward and backward operations, thus inventing ways to represent and process different data types efficiently on GPUs or similar parallel processing hardware, is one of the most if not the most important task when aiming to advance the state of the art in DL.* In the field of 3D vision, which is the focus of this thesis, we often are confronted with more irregular types of data, such as point clouds, sparse depths maps, meshes or graphs. In addition to the challenge of processing them using a parallel data flow model, they usually exhibit different natural symmetries which are way more complex than the often occurring translative symmetries in the image domain, for which CNNs are a natural fit. Therefore, this thesis takes a step towards optimal processing methods for data types in 3D vision, accounting for the different types of representation and developing appropriate inductive biases.

Years of research in 3D vision led to a vast amount of specialized algorithms and data structures to process, analyze, and store data, accounting for symmetries in the data and incorporating problem-specific knowledge [HZ03]. Taking the simple task of surface normal estimation on point clouds as an example, we know very well that we can formulate the task as a local least squares plane fitting problem and solve it using Eigendecomposition (ED) or Singular Value Decomposition (SVD) (cf. Chapter 5), which already accounts for the underlying symmetry of 180° rotations. In contrast, a full end-to-end deep learning approach might apply a stronger data prior to solve this task but always has to find such well-understood symmetries itself, increasing the required amount of training data and required network capacity. Usually, the fixed function algorithms in the field of computer vision, such as the one for surface normal estimation, come with a set of free parameters, which are chosen manually or using heuristics based on data. In many cases, this part of the algorithm is heavily dependent on prior information about the given data. Consequently, this is the part were applying a DL solution might provide the optimal benefit, which leads us to the second premise of this thesis. *Instead of either solving a problem completely end-to-end with deep universal neural networks, ignoring all problem-specific knowledge, or using traditional algorithms with manually found parameters, we assume that the best solutions can be found by combining the best of both worlds: finding algorithms that utilize existing knowledge about the underlying problems while parameterizing them using data-driven DL models.* The goal is to keep advantages

from both domains. We aim to be more *efficient* in resource consumption than pure end-to-end deep learning approaches, increasing practicality of the methods through faster computation, less required training data, and less memory consumption. The algorithms should be more *interpretable* than current DL methods, i.e. it should be possible to manually analyze and understand some intermediate representations, allowing for better understanding of the whole algorithm. At the same time, we still want to be able to capture and utilize the *complex, non-linear relations* that exist in real world data and that are otherwise hard to describe, keeping the favorable properties that led to the success of many DL methods.

This thesis explores two directions to work towards those goals. The first is to start with existing algorithms, or variants of those, replace certain parts with neural networks, and make the whole approach differentiable. Secondly, we can look at the problem from the DL perspective and design inductive biases that are more appropriate for the given task. This thesis gives examples for both of those approaches and analyzes them with respect to efficiency, interpretability, and quality.

1.1 Contribution of this Work

The structure of contributions in the chapters of this thesis is summarized in Figure 1.1. The two main contributions are two differentiable algorithms with data-driven parameterization for different applications in computer vision. First, the Differentiable Iterative Surface Normal Estimation (DISNE) method, an algorithm for surface normal estimation on point clouds that combines the strengths of deep learning and traditional Iterative Re-weighted Least Squares (IRLS). The method includes several novelties: a differentiable variant of local plane fitting, which is efficiently parallelized on the GPU using novel forward and backward algorithms, a novel graph neural network architecture, Local Spatial Graph Transformers (LSGTs), which enables the application of an equivariant implicit kernel function to re-weight the input points for IRLS. Second, Group Equivariant Capsule Networks (GECNs) are presented, a sophisticated version of capsule networks, which mathematically guarantee interpretable representations that disentangle object pose from identity. The method is able to infer object poses that are equivariant to input rotation and respective invariant activations. The work lays the theoretical groundwork of equivariant capsule networks, presenting an equivariant dynamic routing by agreement, which shows under which conditions provably equivariant poses can be computed for different groups. The first presented algorithm considers the 2D case. The second version utilizes a differentiable IRLS algorithm for 3D pose estimation to route quaternion poses through a capsule network.

The introduced 3D vision DL algorithms are analyzed from the perspective of differentiable algorithms, a paradigm to describe sophisticated algorithms that involve deep neural networks and provide stronger inductive biases and more problem-

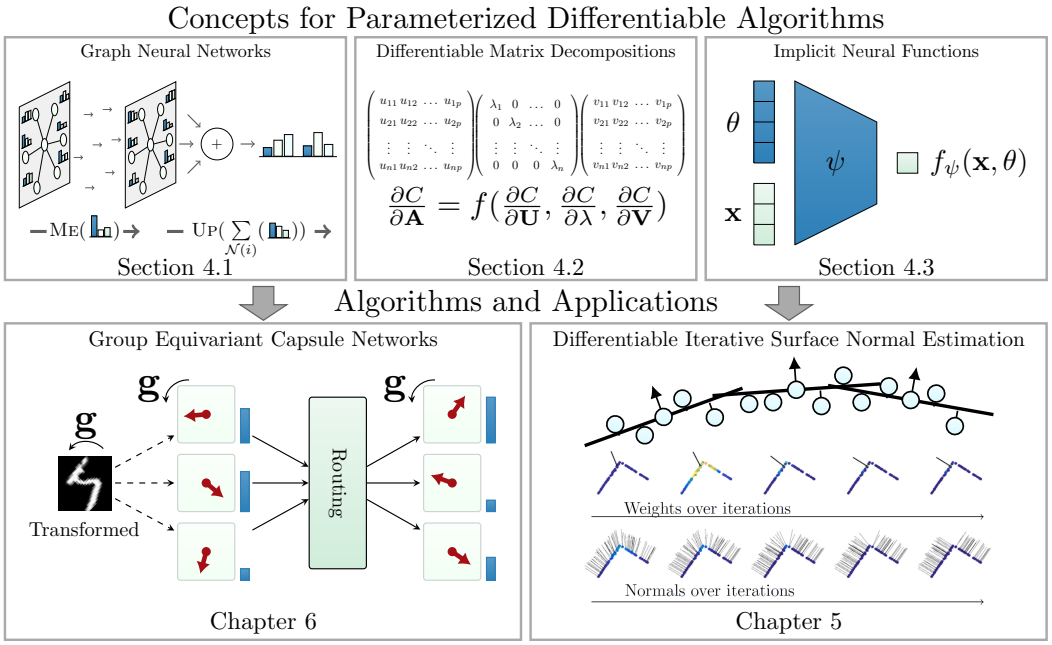


Figure 1.1: Structure and contributions of this thesis. In Chapter 4, the concepts of Graph Neural Networks (GNNs), differentiable matrix decomposition, and Implicit Neural Functions (INFs), including methods that are used in the applications of this thesis. Then, the main contributions of this thesis are presented: DISNE in Chapter 5 and GECNs in Chapter 6.

specific knowledge than pure end-to-end approaches. The paradigm of differentiable algorithms is introduced as a natural extension to the successes that were achieved in DL in the previous years. The presented algorithms are analyzed with respect to qualitative and quantitative metrics for *efficiency*, *interpretability* and *quality* of results.

Further, this thesis describes important building blocks for differentiable algorithms with data-driven parameterization, namely Message Passing Graph Neural Networks (MP-GNNs), differentiable matrix decomposition, and INFs. The first concept, MP-GNNs, is important to process irregularly structured data. In addition to expressing related work in the MP-GNN framework, the SplineCNN [FLW+18] method, a GNN operator for continuous convolution on manifolds, point clouds and graphs is presented. Also the LSGT [LOM20] architecture is introduced, a graph neural network to achieve equivariant operators in local neighborhood of geometric graphs. Both of those methods were novel contributions to the field at the time of publication. The second important concept is differentiable matrix decomposition, which plays an important part in 3D vision DL methods. In addition to summarizing the related literature, this work describes and discusses the applications in state-of-the-art 3D vision algorithms and how to compute them efficiently on the GPU. As third concept, INFs are discussed, which are an efficient method to learn

functions from implicitly given data points. The contributions in this part include an implicit kernel function for surface normal estimation and Deep Local Shapes (DeepLS), a method for practical 3D reconstruction from range scans with DL.

1.2 Organization of the Thesis

The thesis begins with a general introduction to differentiable algorithms in Chapter 2. The chapter defines the general paradigm and introduces the necessary background about Automatic Differentiation (AD) and the backpropagation algorithm, which is of utmost importance to the presented methods. Next, background and the used notations about DL, groups and equivariance, as well as IRLS are introduced in Chapter 3. It follows the first main chapter, Chapter 4 presenting important concepts for differentiable algorithms, including GNNs in Section 4.1, differentiable matrix decompositions in Section 4.2, and INFs in Section 4.3. The subsequent Chapter 5 introduces and evaluates the first algorithm, the DISNE method, a differentiable algorithm for surface normal estimation on point clouds. Then, GECNs are introduced in Chapter 6, providing a theoretical framework for equivariant routing by agreement to obtain capsule networks that disentangle object identity from pose. Lastly, Chapter 7 gives a summary of the results and discusses future research opportunities in the field of differentiable algorithms.

1.3 Author's Publications and Contributions

The research presented in this thesis was initially published by the author in a set of peer-reviewed publications. This section details which methods were originally presented in which publications and outlines the authors contribution to the respective works. The first of the two subjects of this work, the DISNE method presented in Chapter 5, was originally published at CVPR in 2020 (accepted for oral presentation) by the author of this thesis [LOM20]. The author contributed the majority of the work with respect to idea, conceptualization, implementation, and evaluation. The second main topic, GECNs as presented in Chapter 6, was published at NeuRIPS in 2018 [LFL18]. Similarly, the author contributed the majority of the idea, theoretical results, the realization and the evaluation. The extension to 3D data and groups, as presented in Section 6.6, was achieved in joint work with Stanford University and Google Zürich and was presented at ECCV in 2020 [ZBL+20]. The author contributed to concept, realization and writing. The SplineCNN method, as presented in Section 4.1.4, was published at CVPR in 2018 [FLW+18]. Here, the first two authors made equal contributions to all parts of the work. The DeepLS method, as discussed in Section 4.3.4, was created as a collaborative work while participating in an internship at Facebook Reality Labs and was published at ECCV in 2020 [CLI+20]. The author participated in publishing four more important works in the field of GNNs [MRF+19; FL19; FLM+20; FLW+21] at top tier venues of

ICLR, ICML and AAAI, as are listed below. Since GNNs are only a subtopic of this work, these works are only briefly covered in this thesis.

It follows a chronological list of publications, which were created by the author during his PhD studies and which led to this thesis. For each publication, the author's contribution and the appearance in this work are outlined.

[FLW+18] Matthias Fey*, Jan E. Lenssen*, Frank Weichert, Heinrich Müller: *SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels*, CVPR 2018 (*equal contribution). **Contribution:** The author contributed significantly to the idea, realization, evaluation and writing. **Subject of:** Section 4.1.4.

[LFL18] Jan E. Lenssen, Matthias Fey, Pascal Libuschewski: *Group Equivariant Capsule Networks*, NeurIPS 2018. **Contribution:** The author contributed the majority of the work with respect to idea, proofs, realization, evaluation and writing. **Subject of:** Chapter 6, Section 4.2.3.

[MRF+19] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan E. Lenssen, Gaurav Rattan, Martin Grohe: *Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks*, AAAI 2019. **Contribution:** The author assisted in experiments, evaluation and writing. **Mentioned:** Section 4.1.3.

[FL19] Matthias Fey, Jan E. Lenssen: *Fast Graph Representation Learning with PyTorch Geometric*, ICLR 2019 Workshop. **Contribution:** The author contributed to idea and concept and assisted in realization and writing. **Mentioned:** Section 4.1.2.

[LOM20] Jan E. Lenssen, Christian Osendorfer, Jonathan Masci: *Deep Iterative Surface Normal Estimation*, CVPR 2020: **Contribution:** The author contributed the majority of the work with respect to the ideas, realization, evaluation and writing. **Subject of:** Chapter 5, Section 4.1.5, Section 4.2.3, Section 4.3.3.

[FLM+20] Matthias Fey, Jan E. Lenssen, Christopher Morris, Jonathan Masci, Nils M. Kriege: *Deep Graph Matching Consensus*, ICLR 2020: **Contribution:** The Author contributed to idea, concept and writing. **Mentioned:** Section 4.1.3.

[ZBL+20] Yongheng Zhao, Tolga Birdal, Jan E. Lenssen, Emmanuele Menegatti, Leonidas J. Guibas, Federico Tombari: *Quaternion Equivariant Capsule Networks for 3D Point Clouds*, ECCV 2020: **Contribution:** The Author contributed to idea, concept and writing. **Subject of:** Chapter 6, Section 4.2.3.

[CLI+20] Rohan Chabra, Jan E. Lenssen, Julian Straub, Tanner W. Schmidt, Eddy Ilg, Steven Lovegrove, Richard Newcombe: *Deep Local Shapes: Learning Local SDF Priors for Detailed 3D Reconstruction* ECCV 2020: **Contribution:** The Author contributed to realization, evaluation and writing. **Subject of:** Section 4.3.4.

[FLW+21] Matthias Fey, Jan E. Lenssen, Frank Weichert, Jure Leskovec: *GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings* ICML 2021: **Contribution:** The Author contributed to concept and writing. **Mentioned:** Section 4.1.3.

Chapter 2

Differentiable Algorithms with Data-driven Parameterization

The field of Automatic Differentiation (AD) has been an active research area for several years, much longer than the existence of the Deep Learning (DL) concept. AD is a method to obtain derivatives of functions, using neither manual, symbolic or numerical differentiation methods. Instead, it tracks the graph of operations performed to compute a function and utilizes this sequence to iteratively compute the derivative one operation at a time, applying the chain-rule of differentiation. Reverse mode AD, one of the main modes of AD, is a generalized formulation of the backpropagation algorithm, the standard method to obtain sensitivities for training current Deep Neural Network (DNN) architectures. It opened up the field of *differentiable programming*, which serves as underlying technology and motivation for the *differentiable algorithms* of this thesis.

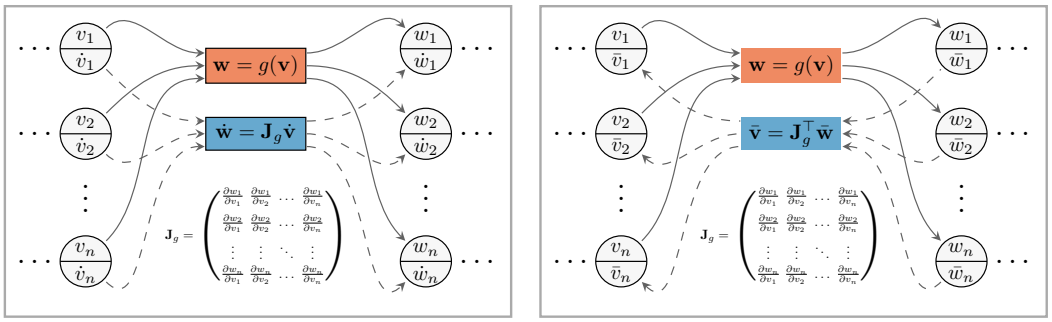
Within and out of the context of AD, several optimization methods exist that utilize higher-order derivative information having, amongst other things, better convergence properties. However, in the current state of DL, those methods did not find relevant practical application, mostly for efficiency reasons. Since the topic of this work are efficient methods for computer vision tasks, where we need to process vast amounts of high-dimensional data, this work focuses on computing first-order derivatives for gradient descent methods.

This chapter will first provide the basic AD background in Section 2.1 on a higher level, motivate the use of the reverse mode AD framework in differentiable algorithms in Section 2.2. Lastly, the framework for differentiable algorithms as used in this thesis is introduced in Section 2.3.

2.1 Automatic Differentiation Modes

Automatic Differentiation (AD) stands for a set of dynamic programming algorithms over a computation graph that iteratively compute partial derivatives of or with respect to intermediate variables of the graph. AD has two main modes [BPR+17]: forward mode AD and reverse mode AD.

Computation Graph A computation graph is a directed graph $\mathcal{C} = (\mathcal{O}, \mathcal{V}, \mathcal{E})$, containing a set of nodes $o \in \mathcal{O}$, which represent operations, a set of nodes $v \in \mathcal{V}$, which represent intermediate values, and a set of edges $e \in \mathcal{E}$, which describe the



(a) Forward Mode AD **(b)** Reverse Mode AD

Figure 2.1: The two main modes of AD. Figure **(a)** shows forward accumulation, where partial derivatives $\dot{v}$ are computed along with the forward pass through the computation graph. Figure **(b)** shows the reverse accumulation, i.e. backpropagation, where the sensitivities $\bar{v}$ are computed in a second, reverse pass through the graph. Both modes are a direct application of the generalized chain-rule.

data flow between operations and intermediate values. Each operation $o \in \mathcal{O}$ has implicit value nodes for output and input values, which often are omitted in the graph figures. If an edge e exists between two nodes o_i and o_j, the output value of o_i has to be equal to the input value of o_j. We make no restriction to the types of operators in $\mathcal{O}$. They can be elementary operations on scalars/vectors but also more sophisticated operators containing sequences of operations. Thus, a given algorithm can be represented by multiple computation graphs, in different levels of detail.

Forward Mode AD Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function with $f(\mathbf{x}) = \mathbf{y}$, for which the derivative should be computed at a point $\mathbf{x} \in \mathbb{R}^n$. For each value node $v_i \in \mathcal{V}$, forward mode AD keeps track of the variable $\dot{v}_i = \frac{\partial v_i}{\partial x_j}$, the partial derivative of value node v_i with respect to changes in one element x_j of the input $\mathbf{x}$. For one element x_j of the input, these values can be computed by forward accumulation, a process that runs in parallel to the normal execution of the computation graph. The process starts with initializing $\dot{\mathbf{x}} = \mathbf{e}_j$, the unit vector with entry 1 at j. Then, it computes the $\dot{v}_i$ for all $v_i \in V$ iteratively, along with computing the v_i, until it reaches all $\dot{y}_i = \frac{\partial y_i}{\partial x_j}$.

We take a look at one operation $g \in \mathcal{O}$ within the computation graph of f, mapping a vector of intermediate values $\mathbf{v}$ to a vector of intermediate values $\mathbf{w}$, as shown in Figure 2.1a. We can propagate the partial derivatives forward using the generalized chain-rule

$$\dot{\mathbf{w}} = \frac{\partial \mathbf{w}}{\partial x_j} = \mathbf{J}_g \frac{\partial \mathbf{v}}{\partial x_j} = \mathbf{J}_g \dot{\mathbf{v}}, \tag{2.1}$$

where $\mathbf{J}_g = (\frac{\partial w_l}{\partial v_k})_{l,k}$ is the Jacobian matrix of the operation g. If we break down the computation graph of f to elementary operations g on scalar inputs and outputs,

these Jacobians reduce to elementary differentiation rules like the product rule, linearity of differentiation, and known derivatives for trigonometric functions. Overall, one evaluation of the full forward accumulation for a function f with initialization $\dot{\mathbf{x}} = \mathbf{e}_j$ amounts to computing the Jacobian vector product $\mathbf{J}_f \mathbf{e}_j$, thus, computing one column of the Jacobian $\mathbf{J}_f \in \mathbb{R}^{m \times n}$ of f and all the intermediate partial derivatives of the v_i's with respect to x_j. For the full evaluation of $\mathbf{J}_f$, n evaluations, one for each element of $\mathbf{x} \in \mathbb{R}^n$, are required.

Reverse Mode AD The second main mode is reverse mode AD. In contrast to forward mode, it keeps track of a different set of intermediate variables. Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with $f(\mathbf{x}) = \mathbf{y}$, it keeps track of $\bar{v}_i = \frac{\partial y_j}{\partial v_i}$ for each value node $v \in \mathcal{V}$, the partial derivatives of one output value y_j, with respect to changes in the intermediate variables v_i. These values $\bar{v}_i$ are also called *sensitivities* (of y_j) with respect to changes in v_i. Computing the $\bar{v}_i$ differs from computing the $\dot{v}_i$ in forward mode. Here, the $\bar{v}_i$ are computed in a second pass through the computation graph in reverse direction, which is called reverse mode accumulation. After the output $\mathbf{y}$ is computed in the forward pass, we initialize $\bar{\mathbf{y}} = \mathbf{e}_j$ for one j at a time and compute the $\bar{v}_i$ using reverse accumulation as a function of succeeding $\bar{v}_j$'s and v_j's, until all $\bar{x}_i = \frac{\partial y_j}{\partial x_i}$ are reached. In general, the derivatives are propagated backwards using the transposed Jacobians. For one operation $g \in \mathcal{O}$ within the computation graph of f, mapping a vector of intermediate values $\mathbf{v}$ to a vector of intermediate values $\mathbf{w}$, as shown in Figure 2.1b, the reverse propagation is defined as

$$\bar{\mathbf{v}} = \frac{\partial y_j}{\partial \mathbf{v}} = \mathbf{J}_g^\top \frac{\partial y_j}{\partial \mathbf{w}} = \mathbf{J}_g^\top \bar{\mathbf{w}}, \tag{2.2}$$

where $\mathbf{J}_g^\top = (\frac{\partial w_l}{\partial v_k})_{l,k}$ is the transposed Jacobian matrix of the operation g. Similar as in the forward mode, the Jacobians are derived from analytic derivatives for elementary operations g. One evaluation of the full reverse mode accumulation for a function f with initialization $\bar{\mathbf{y}} = \mathbf{e}_j$ amounts to the Jacobian vector product $\mathbf{J}_f^\top \mathbf{e}_j$, thus computing one row of the Jacobian $\mathbf{J}_f \in \mathbb{R}^{m \times n}$ of f and all the intermediate sensitivities of y_j with respect to the v_i's. For the full Jacobian, we would need m evaluations, one for each element of $\mathbf{y} \in \mathbb{R}^m$. The downside of reverse mode accumulation is the high memory consumption. Since the derivatives are computed in a second pass, and not alongside the forward computation as with forward accumulation, the values of most $v_i \in \mathcal{V}$ have to be kept in memory for reverse mode accumulation, since they usually are required to compute the Jacobians $\mathbf{J}_g$ for sub-operations g.

2.2 Backpropagation and Reverse Mode AD

In Deep Learning (DL) we are usually concerned with computing the derivative of functions $f : \mathbb{R}^n \to \mathbb{R}$ that have high input dimensionality n and a single output

dimension, the learning objective. As a consequence, the backpropagation algorithm, which is one evaluation of reverse mode AD, naturally became the de facto standard to compute the partial derivatives in DNNs, as the derivatives can be found efficiently using a single evaluation of reverse accumulation. For a single output objective L, backpropagation computes $\bar{v} = \frac{\partial L}{\partial v}$ for each variable v in the computation graph. Thus, the vector containing the $\bar{v}$ of all source variables in the graph is the *gradient* of L. Therefore, in this special case, the terms *gradient*, *Jacobian* or *sensitivities* are often used interchangeably in related literature.

However, applying backpropagation efficiently for large computation graphs is usually not possible by simply executing a chain of Jacobian vector multiplications. For most computation graphs and operations, it requires a large amount of individual engineering for each operation. In the following, a summary of reappearing concepts is presented.

Grouping of operators If a function f is composed of differentiable functions, it is always possible to compute the partial derivatives of f by reverse mode accumulation through all elementary operations in the computation graph, thus breaking it down into the operations sum, multiplication and trigonometric functions. However, there are many situations where this is not the most efficient solution. Examples of those are iterative operations, where we actually know the analytic derivative in closed-form, like it is the case for some matrix decompositions (cf. Section 4.2.1), or cases where storing all intermediate variables consumes too much memory (cf. Section 4.1.4). Also, breaking down the whole graph in its most atomic elements makes it hard to fully utilize the parallel processing power of GPUs. In those cases, it can be heavily beneficial to group sets of operations together as one operation in the computation graph, coming with its own error backpropagation method, a backward algorithm, which is then used in reverse mode accumulation. However, this approach also comes with a disadvantage. While it can make the execution of certain parts of the computation graph more efficient, we can no longer access the sensitivities with respect to intermediate variables within these grouped operations. Thus, variables for which the sensitivities are needed for gradient descent optimization need to be retained in the graph.

Deriving sparse Jacobians In contrast, computing the full Jacobian of an operation g in the computation graph can become extremely inefficient as well, so that more optimized implementations are needed. A common example is the backward algorithm of large matrix multiplications $\mathbf{Y} = \mathbf{W}\mathbf{X}$, as used in MLPs on mini-batches. The naive reverse accumulation steps to compute sensitivities with respect to inputs $\mathbf{X}$ and $\mathbf{W}$ are $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial \mathbf{Y}}{\partial \mathbf{X}}^\top \frac{\partial L}{\partial \mathbf{Y}}$ and $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial \mathbf{Y}}{\partial \mathbf{W}}^\top \frac{\partial L}{\partial \mathbf{Y}}$. For a batch-size of N, I input neurons and O output neurons, thus $\mathbf{W} \in \mathbb{R}^{O \times I}$, $\mathbf{X} \in \mathbb{R}^{I \times N}$ and $\mathbf{Y} \in \mathbb{R}^{O \times N}$, the full Jacobian $(\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}, \frac{\partial \mathbf{Y}}{\partial \mathbf{W}})$ has $I^2 \cdot O^2 \cdot N^2$ elements, which easily explodes to untractable sizes for common, real-world MLPs. For most functions, however, it turns out that the Jacobian is actually very sparsely populated since changes in an

input variable only lead to changes in very small subset of output values. Therefore, in the matrix multiplication case, the sensitivities $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial \mathbf{Y}}{\partial \mathbf{X}} \frac{\partial L}{\partial \mathbf{Y}}$ and $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial \mathbf{Y}}{\partial \mathbf{W}} \frac{\partial L}{\partial \mathbf{Y}}$ with respect to the input variables $\mathbf{X}$ and $\mathbf{W}$ can be expressed in a much more efficient way, that is $\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{W}^\top$ and $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^\top \frac{\partial L}{\partial \mathbf{Y}}$. For a full derivation, the reader is referred to common DL literature [GBC16]. Similar optimized expressions exist for other DNN operators, such as CNNs [LBB+98b] (cf. Section 3.1), and often have to be found for novel operators to make them applicable to large input datasets. A general way of finding sparsity in the Jacobian of a function f is to disassemble f into sub-functions and work on a computation graph with higher level of detail.

Customly defining derivatives Functions which are expressed in computation graphs do not have to be differentiable everywhere in order to compute partial derivatives using AD. On the contrary, a large number of current DL architectures consist of non-differentiable functions, such as the max function or the Rectified Linear Unit (ReLU). Without further adjustments, the AD modes would produce undefined results if for some function g in the computation graph, which is not differentiable at $\mathbf{x}$, the Jacobian $\mathbf{J}_g(\mathbf{x})$ is needed in forward or reverse accumulation. However, in practice that issue is easily circumvented by simply arbitrarily defining derivatives for those vectors. As an example, the derivative at $\mathrm{ReLU}(x) = \max(x, 0)$ can be simply set to 0 and the derivative of $max(x, y)$ at points with $x = y$ is simply the derivative of x. In practice, there is an infinitesimal small chance to hit those points exactly so the arbitrary choice usually does not affect the optimization behavior in a significant way. This principle of customly defining Jacobians can also be applied in more advanced ways. We can also use it to define *surrogate gradients*. Given a function $f(\mathbf{x})$ that is either not differentiable or computing the Jacobian $\mathbf{J}_f$ is too inefficient or numerical unstable, we can define a custom backward function that differs from the correct one. This can either be done directly, by defining a function for reverse mode, or by defining a *surrogate loss*, which is minimized instead of the actual one. If the *surrogate* Jacobian has certain properties, such as pointing to the same local minima as $\mathbf{J}_f$, this technique might enable backpropagation through f. Similarly, if the *surrogate loss* can be shown to be an upper bound of the actual loss, it can be used to minimize the original function using different sensitivities. An example of this, a surrogate loss for backpropagating through ED in a special case, is described in related literature [DYH+20], which is outlined Section 4.2.2 of this thesis.

Implementations of AD, providing the modular architecture and the possibility to implement and utilize all those techniques, became extremely popular in the recent years and are the foundations of many DL frameworks, such as PyTorch [PGM+19], TensorFlow [MAP+15], MXNet [CLL+15] and JAX [BFH+18]. As a consequence, such functionality has also been added to other parallel processing frameworks for computer vision or machine learning like Halide [LGA+18] and Julia [BEK+17]. The rise of these computational frameworks shaped the term *differentiable programming* [Ola15; LeC18; WDW+18] as a new programming paradigm that allows the

definition of programs whose execution automatically come with derivatives that can be used to find program instances in a space of programs. Naturally, standard DL architectures like MLPs, CNNs, RNNs are easily implemented using this paradigm. However, the possibilities do not stop there, since much more complex program spaces with trainable and fixed-function parts can be designed.

2.3 Differentiable Algorithms

With differentiable programming providing such vast possibilities for designing data flows, the question is which principles to apply when creating algorithms for a specific problem. To answer this question, we structure algorithms obtained by differentiable programming under the term *differentiable algorithms with data-driven parameterization* and state hypotheses about design philosophies.

Definition In general, we define a *differentiable algorithm* as a directed graph $A = (\mathcal{O}_T, \mathcal{O}_F, \mathcal{V}, \mathcal{E})$, consisting of trainable operator nodes $\mathcal{O}_T$, fixed operator nodes $\mathcal{O}_F$, value nodes $\mathcal{V}$ and edges $\mathcal{E}$ describing the data flow between values and operators. Thus, a differentiable algorithm is a computation graph with further distinction between trainable and fixed operators. Similar to computation graphs, each operator has implicit input and output nodes, which can be omitted in figures. The algorithm computes a function $F_\Theta(\mathcal{X}) = \mathcal{Y}$, where input $\mathcal{X} \subset \mathcal{V}$ and output $\mathcal{Y} \subset \mathcal{V}$ are modeled as (multiple) value nodes representing vectors. For all operations, which stand between trainable parameters Θ and the output $\mathcal{Y}$, functions to compute $\frac{\partial L}{\partial \mathcal{X}}$ from $\frac{\partial L}{\partial \mathcal{Y}}$ for a scalar criterion $L(\mathcal{Y})$ need to be available for reverse mode accumulation. The operations in $\mathcal{O}_T$ and $\mathcal{O}_F$ are not restricted to be actual differentiable functions but only require defined derivatives, which can also be customly defined or surrogates (cf. Section 2.2). Through its trainable parameter vectors $\Theta = \{\theta_1, ..., \theta_k\}$, a differentiable algorithm A spans a space of program instantiations from which one program is chosen by selecting the parameters $\hat{\Theta}$ that minimize the scalar loss $L(\mathcal{Y})$ on the output of the algorithm.

The paradigm of differentiable programming lets the programmer make several choices to frame this space of programs. Clearly, end-to-end DL models are differentiable algorithms. An MLP, for example, can be expressed as a differentiable algorithm consisting of only one trainable function node, the MLP itself. Since MLPs are universal approximators [HSW89], the actual program for a problem then needs to be found in the space of all possible functions expressible by the network. This allows to learn and detect complex patterns in data, to model and sample from the data distribution and to infer semantic information. It might be the best solution if we have very little information about the problem and how it should be solved. However, due to the large amount of free parameters that is needed to solve complex problems, finding a program in such a large program space also comes with several downsides: it requires a vast amount of training data, might be inefficient

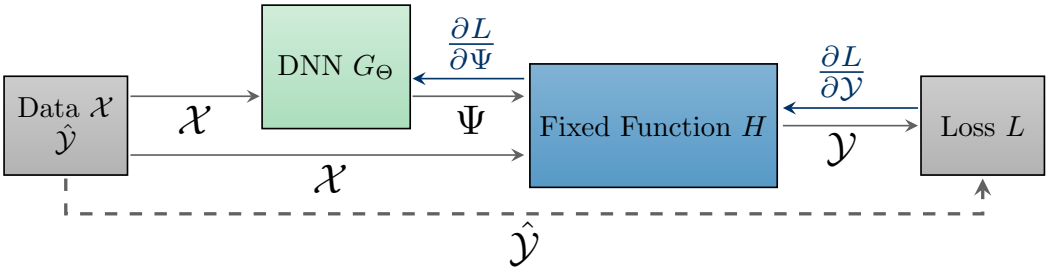


Figure 2.2: A simple, generic example for a differentiable algorithm with data-driven parameterization. The algorithm consists of a ■ fixed function part $H \in \mathcal{O}_F$ with parameterization Ψ and a ■ trainable part (a DNN) $G_\Theta \in \mathcal{O}_T$ with trainable parameters Θ. Parameters Ψ are inferred from data $\mathcal{X}$ using the DNN and the algorithm parameters Θ are trained using backpropagation and gradient descent.

to compute, and it is a black-box solution, making it hard to understand how the problem is solved internally. These downsides may hinder practical application of these DL methods in several areas.

In well-understood problems, there often are sub-problems for which programs are known that already solve them perfectly, given the correct parameterization. The set of parameters might span a much smaller space of programs, without excluding relevant solutions from the space. The goal of this thesis is to explore if the mentioned issues can be resolved by designing algorithms, which solve as much problem complexity as possible in the fixed function nodes $\mathcal{O}_F$ instead of the trainable function nodes $\mathcal{O}_T$, without excluding correct programs for relevant scenarios. This approach amounts to tightening the program space as much as possible by reducing complexity of functions in $\mathcal{O}_T$ and reducing the number of trainable parameters in Θ. Moving complexity from operators $\mathcal{O}_T$ to differentiable, fixed function operators $\mathcal{O}_F$ can also be understood as introducing a strong *inductive bias* that does not reduce the quality of solution. An example scheme for such a differentiable algorithm with data-driven parameterization is given in Figure 2.2, where instead of solving the whole problem with a single DNN, a fixed function algorithm is employed that is merely parameterized by a trainable DNN.

In general, the goal of introducing an appropriate *inductive bias* has been present in machine learning for several years, as a consequence of the no-free-lunch theorem [Mur13]. Usually, the most problem-specific methods provide better results than general ones. However, in DL, we actually saw the opposite trend for many years. Often, problems are tackled using the most expressive and deepest architectures in an effort to learn everything from scratch, given vast amounts of data, ignoring all existing knowledge about the underlying problem. This might be due to several reasons: (1) Larger models usually fit well to large sets of real-world data when trained using a gradient descent variant and backpropagation, (2) designing and implementing more problem specific algorithms and efficient backward steps can be

challenging, and (3) it might not be clear how to process different kinds of input data appropriately, that is, to know which inductive bias to use for a specific data representation.

To overcome these issues, Chapter 4 will present and discuss several novel and existing concepts that can be used as building blocks in differentiable algorithms to introduce the appropriate inductive bias, which takes into account the input data representation and problem-specific knowledge. Further, applications of differential algorithms for problems in the field of 3D vision are presented in Chapter 5 and Chapter 6. By expressing as much complexity in fixed-function parts $\mathcal{O}_F$ as possible, the goal is to improve the following algorithm characteristics:

Efficiency There are different degrees of algorithm efficiency that can be evaluated. Mainly, we are interested in execution time in the inference stage and the amount of required training data. Required training data is directly related to the number of trainable parameters that describe the complexity of the model. Thus, we measure efficiency by comparing inference time and number of trainable parameters of our algorithms to those of existing DL methods solving the same task.

Interpretability Due to the more sophisticated structure of the differentiable algorithm, we can hope to obtain intermediate representations that are interpretable. Naturally, since all operations from $\mathcal{O}_F$ are fixed, they have to have well-defined input and output representations, lying in a known space. As a consequence, we reduce the black-box part of the algorithm, keeping more control over representations within the algorithm. Interpretability is measured qualitatively by visualizing and interpreting intermediate representations in the differentiable algorithm.

Quality of results The main advantage of full end-to-end DL methods is their ability to detect and utilize arbitrary, complex patterns in real-world data in a tractable way. The goal of the differentiable algorithms discussed in this thesis is to keep this ability, which is also the difference to previous methods that use automatic parameterization of algorithms. To this end, data processing techniques with appropriate inductive biases for specific data representations, such as GNNs for geometric data or INFs for volumetric data and kernel functions, are important concepts in this thesis, which find their application in capturing relevant information from those representations and in producing appropriate parameterizations. Quality of results is measured using accuracy and error metrics and is compared to the quality of existing end-to-end DL counterparts.

Chapter 3

Background and Notation

Before diving into the concepts for differentiable algorithms, this section introduces the necessary background and notations for the individual topics. Section 3.1 introduces common DNN architectures and the tensor notation used to describe algorithms in this thesis. Further, geometric groups and the concepts of equi- and invariance are introduced in Section 3.2. Lastly, the well-known concept of Iterative Re-weighted Least Squares (IRLS) is summarized in Section 3.3, as some algorithms in this thesis built upon it.

3.1 General Notation, Tensors and Neural Networks

This section gives an overview about the general notation for differentiable algorithms using vectors, matrices and tensors, as well as basic deep neural network operators and their backward computation for reverse accumulation. In general, vectors are denoted in bold lower case, e.g. $\mathbf{x} \in \mathbb{R}^d$, matrices and tensors in bold upper case, e.g. $\mathbf{A} \in \mathbb{R}^{m \times n}$, scalars in italic lower or upper case, e.g. $s, S \in \mathbb{R}$, and sets in calligraphic upper case, e.g. $\mathcal{P} \subseteq \mathbb{R}^d$. For algorithms presented in this thesis, the tensor notation is used, in accordance to common differentiable programming libraries such as PyTorch [PGM+19]. In this context, a tensor is a k-dimensional matrix (or array, in implementation) $\mathbf{A} \in \mathbb{R}^{d_1 \times \ldots \times d_k}$, which can be indexed using braces (beginning with index 0), e.g. if $k = 3$, $\mathbf{A}[i, j, l]$ describes the element at position (i, j, l) in the tensor. We further use the :-Symbol to index slices of the tensor. Using the symbol instead of a variable selects all indices in that dimension. As an example, $\mathbf{B}[2, :]$ selects the third row of a 2-dimensional tensor $\mathbf{B}$. The element-wise product (Hadamard product) is denoted by $\odot$, while the standard multiplication symbol $\cdot$ (or no symbol at all) describes matrix multiplication, matrix-vector multiplication, the inner/outer product between vectors, or vector scaling. The $\circ$-Symbol is used for group law and group actions, as further detailed in Section 3.2.

Multi-Layer Perceptrons (MLPs) A Multi-Layer Perceptron (MLP) is the composition of L functions $f^i_{\mathbf{W}_i, \mathbf{b}_i} : \mathbb{R}^{n_i} \to \mathbb{R}^{m_i}$, $1 \leq i \leq L$, with $m_i = n_{i+1}$, and

$$f^i_{\mathbf{W}_i, \mathbf{b}_i}(\mathbf{x}) = \sigma(\mathbf{b}_i + \mathbf{W}_i \cdot \mathbf{x}). \tag{3.1}$$

Here, σ is an arbitrary element-wise non-linearity, such as the ReLU function $\mathrm{ReLU}(x) = \max(0, x)$ or the sigmoid function $S(x) = \frac{1}{1+e^{-x}}$, and $\mathbf{W}_i, \mathbf{b}_i$ are trainable

parameters for each function. Let $\mathbf{z}_i = \mathbf{b}_i + \mathbf{W}_i \cdot \mathbf{x}$ be the intermediate result of a layer i before application of the non-linearity, then the sensitivities with respect to parameters $\mathbf{W}_i, \mathbf{b}_i$ and input $\mathbf{x}_i$ are computed as

$$\frac{\partial L}{\partial \mathbf{W}_i} = \frac{\partial L}{\partial \mathbf{z}_i}\mathbf{x}_i^\top, \quad \frac{\partial L}{\partial \mathbf{b}_i} = \frac{\partial L}{\partial \mathbf{z}_i} \quad \text{and} \quad \frac{\partial L}{\partial \mathbf{x}_i} = \mathbf{W}_i^\top \frac{\partial L}{\partial \mathbf{z}_i}, \tag{3.2}$$

given the sensitivities $\frac{\partial L}{\partial \mathbf{z}_i}$ with respect to $\mathbf{z}_i$, which can be computed from the sensitivities $\frac{\partial L}{\partial \mathbf{x}_{i+1}}$ of layer $i+1$ as

$$\frac{\partial L}{\partial \mathbf{z}_i} = \sigma'(\mathbf{z}_i) \odot \frac{\partial L}{\partial \mathbf{x}_{i+1}} \tag{3.3}$$

depending on the used non-linearity σ. Several variants and modifications exist for MLPs, such as different regularization techniques, loss functions and non-linearities, which also modify reverse mode computations [GBC16].

Convolutional Neural Networks (CNNs) A Convolutional Neural Network (CNN) is a composition of L functions $f^i_{\mathbf{K}_i,\mathbf{b}_i} : \mathbb{R}^{N_i \times Y_i \times X_i} \to \mathbb{R}^{N_{i+1} \times Y_{i+1} \times X_{i+1}}$, $1 \le i \le L$, mapping 3-dimensional tensors to 3-dimensional tensors that usually represent image and feature maps. The functions are usually of one of two types: convolutional layers and pooling layers. A convolutional layer with kernel size K, a trainable kernel $\mathbf{K}_i \in \mathbb{R}^{N_{i+1} \times N_i \times K \times K}$, and a trainable bias vector $\mathbf{b}_i \in \mathbb{R}^{N_{i+1}}$ is defined as

$$f^i_{\mathbf{K}_i,\mathbf{b}_i}(\mathbf{X})[:, y, x] = \sigma\Big(\mathbf{b}_i + \sum_{j=0}^{K-1}\sum_{l=0}^{K-1} \mathbf{K}_i[:, :, j, l] \cdot \mathbf{X}\Big[:, y + j - \lfloor\frac{K}{2}\rfloor, x + l - \lfloor\frac{K}{2}\rfloor\Big]\Big), \tag{3.4}$$

with σ being an arbitrary non-linearity, similar to the MLP case. To obtain values of $\mathbf{X}$ that lie out of bounds, different padding strategies, such as padding with zero, are used. It can be observed that the sizes of both trainable parameters do not depend on input dimensions X_i and Y_i, and that they are shared across all positions of the input map. This weight sharing is the most important feature of CNNs, leading to the useful inductive bias of translation equivariance for images, as the same features are detected at multiple spatial positions. Another consequence is that a trained layer can be applied to differently sized inputs later, without modification. Let $\mathbf{Z}_i$ be the resulting tensor of the operation inside the non-linearity σ and $\frac{\partial L}{\partial \mathbf{Z}_i}$ the respective sensitivities, which can be obtained as shown in Equation (3.3). Then, the sensitivities with respect to $\mathbf{K}_i$ $\mathbf{b}_i$ and $\mathbf{X}_i$ can be obtained by

$$\frac{\partial L}{\partial \mathbf{K}_i}[:, :, j, l] = \sum_{y=0}^{Y_i-1}\sum_{x=0}^{X_i-1} \frac{\partial L}{\partial \mathbf{Z}_i}[:, y, x] \cdot \mathbf{X}\Big[:, y - j - \lfloor\frac{K}{2}\rfloor, x + l - \lfloor\frac{K}{2}\rfloor\Big]^\top, \tag{3.5}$$

$$\frac{\partial L}{\partial \mathbf{b}_i} = \sum_{y=0}^{Y_i-1}\sum_{x=0}^{X_i-1} \frac{\partial L}{\partial \mathbf{Z}_i}[:, y, x], \text{ and} \tag{3.6}$$

$$\frac{\partial L}{\partial \mathbf{X}_i}[:, y, x] = \sum_{j=0}^{K-1} \sum_{l=0}^{K-1} \mathbf{K}_i[:, :, l, j] \cdot \frac{\partial L}{\partial \mathbf{Z}_i}\left[:, y + j - \lfloor \frac{K}{2} \rfloor, x + l - \lfloor \frac{K}{2} \rfloor\right]. \tag{3.7}$$

By describing the computation as sums of tensor slices, an analogy to MLPs can be observed. For each combination of filter and input spatial position, the operations are local variants of the MLP matrix-vector multiplication. In the case of $K = 1$, the CNN collapses to an MLP, which is shared over all spatial positions of the input feature map, also called 1×1-convolutional operator. Also, the calculation of $\frac{\partial L}{\partial \mathbf{X}_i}$ can be seen as convolution with a kernel transposed in the last two dimensions. In practice, there exist optimized implementations for several sub-cases of the convolution operator, such as implementations based on the Fast Fourier Transform [MHL14] or the Winograd transformation for convolution with a 3×3 kernel [LG16]. Pooling layers down-scale the input feature maps $\mathbf{X}_i$ in X_i and Y_i dimensions by using an aggregation function like max() or avg() in a local kernel window. The implementation is similar to that of convolution without the use of trainable parameters.

3.2 Groups and Equivariance

Several important inductive biases present in DL are based on equivariant or invariant operators, such as those present in CNNs and GNNs, utilizing symmetries in the input data. In order to represent these properties appropriately, this section summarizes some basic definitions from (Lie) group theory, on top of which they can be formalized.

3.2.1 Groups, Lie Groups, and Matrix Groups

A *group* is specified by a tuple $(\mathcal{G}, \circ)$ containing a set $\mathcal{G}$ and an operation $\circ$ called the group law, which has the following properties:

1. *Closeness:* For all $\mathbf{g}_1, \mathbf{g}_2 \in \mathcal{G}$ holds $\mathbf{g}_1 \circ \mathbf{g}_2 \in \mathcal{G}$.
2. *Associativity:* For all $\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3 \in \mathcal{G}$ holds $(\mathbf{g}_1 \circ \mathbf{g}_2) \circ \mathbf{g}_3 = \mathbf{g}_1 \circ (\mathbf{g}_2 \circ \mathbf{g}_3)$.
3. *Neutral element:* There exists $\mathbf{e} \in \mathcal{G}$ so that for all $\mathbf{g} \in \mathcal{G}$ holds $\mathbf{g} \circ \mathbf{e} = \mathbf{e} \circ \mathbf{g} = \mathbf{g}$.
4. *Inverse:* For all $\mathbf{g} \in \mathcal{G}$ there exists $\mathbf{g}^{-1} \in \mathcal{G}$ with $\mathbf{g} \circ \mathbf{g}^{-1} = \mathbf{g}^{-1} \circ \mathbf{g} = \mathbf{e}$.

A *Lie group* $(\mathcal{G}, \circ)$ is a group that has the following additional properties:

1. *Manifold:* $\mathcal{G}$ is a *manifold*, a topological space that at each element is locally homeomorphic to an open ball in $\mathbb{R}^n$.
2. *Smoothness:* The group law $\circ$ and taking the inverse $\mathbf{g}^{-1}$ are differentiable operations.

A *matrix group* is a group $(\mathcal{G}, \circ)$, where $\mathcal{G}$ is a set of invertible $n \times n$ matrices and $\circ$ is matrix multiplication. In this thesis, we consider either finite permutation groups or matrix Lie groups, specifically, subgroups of the *general linear group* $GL(n, \mathbb{R})$, containing all real, invertible $n \times n$ matrices with matrix multiplication as group law.

3.2.2 Group Representations

Given two groups $(\mathcal{G}_1, \circ_1)$ and $(\mathcal{G}_2, \circ_2)$, a *group homomorphism* from G_1 to G_2 is a function $f : \mathcal{G}_1 \to \mathcal{G}_2$ with $f(\mathbf{g} \circ_1 \mathbf{h}) = f(\mathbf{g}) \circ_2 f(\mathbf{h})$ for all $\mathbf{g}, \mathbf{h} \in \mathcal{G}_1$. If f is bijective, we call it a *group isomorphism* and say $(\mathcal{G}_1, \circ_1)$ and $(\mathcal{G}_2, \circ_2)$ are *equivalent*.

A *group representation* of a group $(\mathcal{G}, \circ)$ is a group homomorphism $\phi : \mathcal{G} \to GL(V)$ mapping elements of $\mathcal{G}$ to elements of the general linear group $GL(V)$ over a vector space V, usually to real or complex matrices that mirror the group structure with matrix multiplication. A representation is called *faithful* if ϕ is injective, i.e. a one-to-one map.

3.2.3 Group Action, Equivariance and Symmetry

We say a group $(\mathcal{G}, \circ)$ can *act on* elements of a space $\mathcal{X}$ from the left if there exist a faithful representation ϕ of $(\mathcal{G}, \circ)$ satisfying

1. $\phi(\mathbf{e})\mathbf{x} = \mathbf{x}$ for all $\mathbf{x} \in \mathcal{X}$ and neutral element $\mathbf{e} \in \mathcal{G}$, and
2. $\phi(\mathbf{g})(\phi(\mathbf{h})\mathbf{x}) = \phi(\mathbf{g} \circ \mathbf{h})\mathbf{x}$ for all $\mathbf{g}, \mathbf{h} \in \mathcal{G}$ and $\mathbf{x} \in \mathcal{X}$.

In case such an operation exists, $\mathcal{X}$ together with the action is called a left $\mathcal{G}$*-set*. In our context, $\mathcal{X}$ usually is $\mathbb{R}^n$, function spaces containing vector fields, or groups. We say a group $(\mathcal{G}, \circ)$ *acts on* another group $(\mathcal{H}, \circ_{\mathcal{H}})$, when additionally to the above

$$\phi(\mathbf{g})(\mathbf{x} \circ_{\mathcal{H}} \mathbf{y}) = \phi(\mathbf{g})\mathbf{x} \circ_{\mathcal{H}} \phi(\mathbf{g})\mathbf{y} \tag{3.8}$$

holds for all $\mathbf{g} \in \mathcal{G}$ and $\mathbf{x}, \mathbf{y} \in \mathcal{H}$.

An operator $\Psi : \mathcal{X} \to \mathcal{Y}$ that maps elements from a left $\mathcal{G}$-set $\mathcal{X}$ to a left $\mathcal{G}$-set $\mathcal{Y}$ is called *left-equivariant* with respect to a group $\mathcal{G}$ if it commutes with the group action:

$$\Psi(\phi(\mathbf{g})\mathbf{x}) = \phi(\mathbf{g})\Psi(\mathbf{x}), \text{ for all } \mathbf{g} \in \mathcal{G}, \mathbf{x} \in \mathcal{X}. \tag{3.9}$$

Analogously, we can define *right action*, *right* $\mathcal{G}$*-set*, and *right-equivariance*. In the further course of this thesis, we also use the group law symbol $\circ$ for describing the group action. Thus, we simply write $\mathbf{g} \circ \mathbf{x}$ instead of $\phi(\mathbf{g})\mathbf{x}$. Then, equivariance can simply be expressed as

$$\Psi(\mathbf{g} \circ \mathbf{x}) = \mathbf{g} \circ \Psi(\mathbf{x}), \text{ for all } \mathbf{g} \in \mathcal{G}, \mathbf{x} \in \mathcal{X}. \tag{3.10}$$

We further say that an operator $\Psi : \mathcal{X} \to \mathcal{Y}$ is *invariant* with respect to $\mathcal{G}$ if no action $\mathbf{g}$ on the input changes the output of Ψ:

$$\Psi(\mathbf{g} \circ \mathbf{x}) = \Psi(\mathbf{x}), \text{ for all } \mathbf{g} \in \mathcal{G}, \mathbf{x} \in \mathcal{X}. \tag{3.11}$$

Lastly, we say that the *symmetries* of an element $\mathbf{x} \in \mathcal{X}$ are actions $\mathcal{S}_{\mathbf{x}} \subseteq \mathcal{G}$ under which $\mathbf{x}$ is invariant, formally

$$\mathcal{S}_{\mathbf{x}} = \{\mathbf{g} \in \mathcal{G} \mid \mathbf{g} \circ \mathbf{x} = \mathbf{x}\}. \tag{3.12}$$

3.2.4 Group Products

Two groups $(\mathcal{G}, \circ_\mathcal{G})$ and $(\mathcal{H}, \circ_\mathcal{H})$ can be combined to define the *direct product group* $(\mathcal{G}, \circ_\mathcal{G}) \times (\mathcal{H}, \circ_\mathcal{H}) = (\mathcal{G} \times \mathcal{H}, \circ)$, with disentangled group law

$$(\mathbf{g}_1, \mathbf{h}_1) \circ (\mathbf{g}_2, \mathbf{h}_2) = (\mathbf{g}_1 \circ_G \mathbf{g}_2, \mathbf{h}_1 \circ_H \mathbf{h}_2). \tag{3.13}$$

If a group $(\mathcal{G}, \circ_\mathcal{G})$ acts on a group $(\mathcal{H}, \circ_\mathcal{H})$ with a function ϕ, we can define the *semidirect product group* $(\mathcal{G}, \circ_\mathcal{G}) \ltimes (\mathcal{H}, \circ_\mathcal{H}) = (\mathcal{G} \ltimes \mathcal{H}, \circ)$ with entangled group law

$$(\mathbf{g}_1, \mathbf{h}_1) \circ (\mathbf{g}_2, \mathbf{h}_2) = (\mathbf{g}_1 \circ_G \mathbf{g}_2, \mathbf{h}_1 \circ_H \phi(\mathbf{g}_2)\mathbf{h}_2). \tag{3.14}$$

3.2.5 Groups in this Thesis

The following groups are relevant in this thesis in Chapters 4, 5, and 6.

Symmetric Groups The first set of groups are the *symmetric groups* S_n, which contain all possible permutations of n elements. We choose permutation matrices $\mathbf{P} \in [0,1]^{n \times n}$ as faithful representation of S_n, which have a single 1 in each row and column. The neutral element is represented as the identity matrix and the inverse of an element $\mathbf{P}$ can be found as $\mathbf{P}^\top$ since permutation matrices are orthonormal. Symmetric groups play a role in the field of GNNs (cf. Section 4.1.2), as the set of graph node features needs to be implemented as ordered tensor and, therefore, GNNs apply specific mechanisms to ensure permutation equivariance.

Translation Groups The translation groups $(\mathbb{R}^n, \circ)$ and $(\mathbb{Z}^n, \circ)$ contain real and integer vectors together with vector addition. The group $(\mathbb{Z}^2, \circ)$ can act on the input of CNNs by translating the vector field. We can express the translation equivariance of a convolutional layer f on a vector field $X : \mathbb{Z}^2 \to \mathbb{R}^d$ as

$$f_{\mathbf{K},\mathbf{b}}((\mathbf{z} \circ X))(\mathbf{x}) = f_{\mathbf{K},\mathbf{b}}(X)(\mathbf{z}^{-1} \circ \mathbf{x}) = (\mathbf{z} \circ f_{\mathbf{K},\mathbf{b}})(X) \tag{3.15}$$

for all $\mathbf{x}, \mathbf{z} \in \mathbb{Z}^2$. Vector fields will be formally introduced in Section 6.4.

The 2D Rotation Group The group $SO(2)$ contains all rotations in two-dimensional space. A common representation, which is also used in this thesis for designing 2D capsule network architectures in Section 6.5, are 2D rotation matrices, thus all orthonormal 2×2 matrices with determinant 1, with which the group is able to act on $\mathbb{R}^2$ and $SE(2)$.

The 3D Rotation Group and the Unit Quaternion Group The group $SO(3)$ contains all rotations in three dimensional space. One often used representation are the orthonormal 3×3 matrices with determinant 1 (3D rotation matrices), with which $SO(3)$ can act on $\mathbb{R}^3$ and $SE(3)$. Further, we use the group $\mathbb{H}^\star$ of unit quaternions

together with quaternion multiplication, which is a double cover of $SO(3)$ and can serve as representation for $SO(3)$, which allows more efficient element generation and admits itself to an equivariant average operator. Both groups play important roles in Section 4.1.5, Section 5.3.1, and Section 6.6.

The Roto-Translation Group $SE(n)$ The special Euclidean groups $SE(n)$ describe the combined affine transformations of rotation and translation. $SE(n)$ is equivalent to the semidirect product group $SO(n) \ltimes (\mathbb{R}^n, +)$ between n-dimensional rotation and translation. It is important to note that $SO(3)$ acts on $(\mathbb{R}^n, +)$ in the law of the semidirect product, thus translation is not disentangled from rotation. A canonical matrix representation is the set of homogeneous transformation matrices with which $SE(n)$ is are able to act on $\mathbb{R}^n$ by matrix multiplication. The groups $SE(n) = SO(n) \ltimes (\mathbb{R}^n, +)$ are used to design equivariant capsule architectures on vector fields, as described in Section 6.4.1.

3.3 Iterative Re-weighted Least Squares

The well-known Iterative Re-weighted Least Squares (IRLS) method is applied and build upon in several parts of this thesis, e.g. it is used to create a method for differentiable robust surface normal estimation in Chapter 5 and related to iterative routing by agreement in capsule networks in Section 6.6. Therefore, the background of IRLS is outlined in the following.

3.3.1 Least Squares Approximation

Least squares approximation is useful to fit a parameterized linear model to given data points by minimizing the squared distance of points to the model. Formally, a set of linear equations of the form $\mathbf{Ax} = \mathbf{b}$ is given, with $\mathbf{A}$ being an $m \times n$ matrix with $m \geq n$ and $\text{rank}(A) = n$, containing m data points $\mathbf{a} \in \mathbb{R}^n$ in its rows. We consider overdetermined systems with $m >> n$, so there is no unique solution. Then, the vector

$$\hat{\mathbf{x}} = \underset{\mathbf{x}}{\text{argmin}} \, \|\mathbf{Ax} - \mathbf{b}\|_2^2 , \tag{3.16}$$

that minimizes the Euclidean vector norm of $\mathbf{e} = \mathbf{Ax} - \mathbf{b}$ is called a least squares approximation to the system of equations [HZ03]. A solution to this problem can be found using ED or SVD, as described in Section 4.2, or by computing the pseudo-inverse of $\mathbf{A}^\top\mathbf{A}$ and find the solution as $\hat{\mathbf{x}} = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b}$ (in the overdetermined case) in closed-form, afterwards.

A generalization of least squares approximation is weighted least squares approximation, which minimizes the weighted norm $\|\text{diag}(\mathbf{w})\mathbf{e}\|_2^2$ (with $\text{diag}(\mathbf{w})$ denoting the square matrix with $\mathbf{w}$ in its diagonal and 0 elsewhere), effectively weighting the

error, also called residual, of each data point by a scalar weight w_i. We search for the solution of

$$\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \|\operatorname{diag}(\mathbf{w})(\mathbf{A}\mathbf{x} - \mathbf{b})\|_2^2, \tag{3.17}$$

which again can be found as $\hat{\mathbf{x}} = (\mathbf{A}^\top \operatorname{diag}(\mathbf{w})^\top \operatorname{diag}(\mathbf{w})\mathbf{A})^{-1}\mathbf{A}^\top \operatorname{diag}(\mathbf{w})^\top \operatorname{diag}(\mathbf{w})\mathbf{b}$ with (pseudo-)inverse computation, or using SVD/ED.

3.3.2 Robustness Through Minimizing l_p-Norms

(Weighted) least squares optimization minimizes the l_2 norm of the errors $\mathbf{e}$. However, this is not always the best way to fit a model, e.g. when outliers are present in the data that heavily influence the resulting model. Finding a model that minimizes a more general l_p error, that is finding the appropriate vector $\mathbf{x}$ for a specific p can lead to more robust solutions in certain situations, i.e. solutions that only weakly consider outliers. For a vector $\mathbf{e} \in \mathbb{R}^n$, the l_p norm (or quasi-norm, as the triangle inequality is not fulfilled for $0 < p < 1$) is defined as

$$\|\mathbf{e}\|_p = \left(\sum_{i=1}^{n} |e_i|^p \right)^{\frac{1}{p}}. \tag{3.18}$$

It has been shown that for each p there exist weights $\mathbf{w}$ so that minimizing $\|\mathbf{e}\|_p$ is equivalent to solving a weighted least squares problem with weights $\mathbf{w}$. Using this definition, we can define the process of Iterative Re-weighted Least Squares (IRLS), which finds the solution to the system of equations that minimizes the l_p norm for specific p. IRLS is a sequence of weighted least squares optimization steps, where the weights of an iteration i depend on the errors of iteration i, starting with uniform weights in iteration zero. Formally, the process in iteration $i+1$ can be described as finding

$$\hat{\mathbf{x}}^{i+1} = \underset{\mathbf{x}}{\operatorname{argmin}} \left\|\operatorname{diag}(f(\mathbf{e}^i))(\mathbf{A}\mathbf{x} - \mathbf{b})\right\|_2^2, \tag{3.19}$$

where the weights are a function $f : \mathbb{R}^n \to \mathbb{R}^n$ of the previous errors $\mathbf{e}^i = \mathbf{A}\hat{\mathbf{x}}^i - \mathbf{b}$. To find the solution for a specific l_p-norm, the weights can be chosen as

$$w_j = f(\mathbf{e})_j = e_j^{\frac{p-2}{2}}, \tag{3.20}$$

what has been shown to converge to the correct solution for at least $1.5 < p < 3$ [Bur14]. For other p several other weighting functions exist, which converge under certain conditions. For an overview of the vast amount of variants and methods in the IRLS domain, the reader is referred to review literature [HW77; Bur14].

In this thesis, methods inspired by IRLS are presented, which replace the fixed weighting function with a DNN, estimating optimal weights for the next information based on the previous error and additional feature information. While we no longer can guarantee that the approach converges to the solution for a specific l_p norm, we

gain other advantages, like the network being much less dependent on the specific, a priori choice of a norm. It is able to learn its own type of error metric to best fit the ground truth data, which can be a specific l_p solution but also more sophisticated, data-dependent variants.

Chapter 4

Concepts for Differentiable Algorithms

Differentiable algorithms and complex neural network architectures are often put together from building blocks that follow the same repeating concepts. This chapter tackles three common concepts that are of large importance to the applications of this thesis and to which contributions have been made in several ways: Graph Neural Networks (GNNs) in Section 4.1, Differentiable Matrix Decompositions in Section 4.2, and Implicit Neural Functions and Representations in Section 4.3. All three techniques are versatile and are used in many state-of-the-art algorithms in the field of 3D vision. Each section will first formally introduce the concept and related work, before presenting the methods that are applied in this thesis.

4.1 Graph Neural Networks

This section will introduce Graph Neural Networks (GNNs), a useful tool for modeling the data flow of deep neural networks on irregularly structured data. Examples for irregularly structured data are point clouds and meshes, however, in every scenario in which we want to obtain feature representations on a structure that can be modeled as a graph, GNNs can be applied. In this work, we will focus on Message Passing Graph Neural Networks (MP-GNNs), a quite universal framework that covers most of the existing GNN operators and allows to naturally describe the application of GNNs in other algorithms. MP-GNNs are introduced in Section 4.1.2, along with the necessary background and details. In Sections 4.1.4 and 4.1.5 SplineCNN and Local Spatial Graph Transformers are described in detail, two operators, which were introduced by the author of this thesis. In the last years, a vast amount of such GNN operators were presented, an overview of which is given in Section 4.1.3. The tools described in this section find their application in Chapters 5 and 6, tackling surface normal estimation and capsule networks.

4.1.1 Geometric Graph Data

We begin by introducing the necessary notation for graphs. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a *directed graph* with adjacency $\mathbf{A} \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$. Further, let $\mathbf{x}_v$ be node feature vectors for all $v \in \mathcal{V}$, $\mathbf{e}_{v,w}$ be edge feature vectors for edges $(v,w) \in \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, and $\mathcal{N}(v) = \{w \in \mathcal{V} \mid (w,v) \in \mathcal{E}\}$ denote the set of incoming neighbors of a node v. A

directed graph in this general formulation, can model large variety of different data types. In addition to classical graph data, we can for example model point clouds, meshes, discrete manifolds or geometric scene graphs.

Point Clouds We model point clouds, by choosing the set of nodes as points of the point cloud $\mathcal{V} = \{\mathbf{p}_1, ..., \mathbf{p}_N\}$, while obtaining the edges either using a radius threshold, $\mathcal{E} = \left\{(\mathbf{p}_i, \mathbf{p}_j) \in \mathcal{V} \times \mathcal{V} \mid \|\mathbf{p}_j - \mathbf{p}_i\|_2 < r\right\}$, or as a k-nearest neighbor graph, $\mathcal{E} = \left\{(\mathbf{p}_i, \mathbf{p}_j) \in \mathcal{V} \times \mathcal{V} \mid \mathbf{p}_j \in \mathcal{N}^{\leq k}(\mathbf{p}_i)\right\}$, with $\mathcal{N}^{\leq k}(\mathbf{p}_i)$ containing the k nearest points of $\mathbf{p}_i$. If available, information about point $\mathbf{p}_i$ (absolute point position, additional information from the scanner, other arbitrary features) can be modeled as node feature $\mathbf{x}_i$ while the edge features contain the relative Cartesian position: $\mathbf{e}_{i,j} = \mathbf{p}_j - \mathbf{p}_i$.

Meshes Meshes are naturally modeled as a graph by taking the vertices as the set of nodes $\mathcal{V} = \{\mathbf{p}_1, ..., \mathbf{p}_N\}$ and the mesh edges as graph edges $\mathcal{E}$. Since we model directed graphs, we store each mesh edge twice, once for each direction. Additionally, the relative Cartesian positions can be used as edge features $\mathbf{e}_{i,j} = \mathbf{p}_j - \mathbf{p}_i$.

Discrete Manifolds Discrete manifolds can be modeled by using a collection of points sampled from the manifold $\mathcal{V} = \{\mathbf{p}_1, ..., \mathbf{p}_N\}$. We further can store *geodesic distances* $d_{i,j}^{\text{geo}}$ in edge features $\mathbf{e}_{i,j}$ for each pair of sampled points. In most situations, it is more efficient to only store the distances between neighboring points, i.e. by only using edges to k nearest neighbors.

Geometric Scene Graphs A further category that deserves mentioning are geometric scene graphs. Those are general graphs in which each node $v \in \mathcal{V}$ represents an entity of the scene and edges $\mathcal{E}$ describe the geometric (or also semantic) relation between those entities. On the nodes, we can use the node features $\mathbf{x}_v$ to store information about the entity while the edge features $\mathbf{e}_{v,w}$ encode the geometric or semantic relations. Graphs consisting of keypoints of an image can be considered as a variant of a geometric scene graph.

4.1.2 Message Passing Graph Neural Networks

We now introduce the framework of Message Passing Graph Neural Networks (MP-GNNs), a general way to describe deep neural networks on all the different kinds of graph data. MP-GNNs operate on graph-structured data $\mathcal{G}$ by following a *message passing scheme* [GSR+17; FL19; Ham20], which computes new node representations in each iteration. A new node representation is obtained by aggregating incoming messages, which are computed for each directed edge in the graph, based on the previous node features and edge features (messages).

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with initial node features $(\mathbf{x}_v^0)_{v\in\mathcal{V}}$ and initial edge features $(\mathbf{e}_{w,v}^0)_{(w,v)\in\mathcal{E}}$, a layer ℓ of an MP-GNN computes new node representations $(\mathbf{x}_v^\ell)_{v\in\mathcal{V}}$ and edge representations $(\mathbf{e}_{w,v}^\ell)_{(w,v)\in\mathcal{E}}$ formally as

$$\begin{aligned} \mathbf{e}_{w,v}^\ell &= \textsc{Message}_{\boldsymbol{\Theta}_\mathbf{M}}^\ell\left(\mathbf{x}_w^{\ell-1}, \mathbf{x}_v^{\ell-1}, \mathbf{e}_{w,v}^{\ell-1}\right), \\ \mathbf{x}_v^\ell &= \textsc{Update}_{\boldsymbol{\Theta}_\mathbf{U}}^\ell\left(\mathbf{x}_v^{\ell-1}, \square_{w\in\mathcal{N}(v)} \mathbf{e}_{w,v}^\ell\right), \end{aligned} \tag{4.1}$$

where $\square$ is a permutation-invariant aggregation function, like sum, mean, or maximum. $\textsc{Message}_{\boldsymbol{\Theta}_\mathbf{M}}^{(\ell)}$ and $\textsc{Update}_{\boldsymbol{\Theta}_\mathbf{U}}^{(\ell)}$ (in the following denoted as $\textsc{Me}_{\boldsymbol{\Theta}_\mathbf{M}}$ and $\textsc{Up}_{\boldsymbol{\Theta}_\mathbf{U}}$) are arbitrary differentiable functions with trainable parameter sets Θ_M and Θ_U, which may or may not be shared over the layers l. In following occurrences, we omit the layer index l for simplicity.

Scatter and Gather on GPUs

The MP-GNN framework directly represents a convenient computation scheme for the GPU, which is one of its main advantages and of importance for deep learning in general. There are two dimensions for parallelization: the edge parallel space for computing the Message function and the node parallel space to compute the Update function. We can switch between those spaces via efficient scatter and gather operations on GPU memory, which allow pseudo-parallel memory access.

A *gather operation* is a parallel read access of E processing cores on the GPU, each working on one graph edge. Given an index tensor $\mathbf{I} \in \mathbb{R}^E$ and a node feature tensor $\mathbf{X} \in \mathbb{R}^{n\times d}$ as input, a processing core j reads the feature vector $\mathbf{X}[\mathbf{I}[j], :]$ from memory, moving the features from node parallel space to edge parallel space.

Similarly, a *scatter operation* is a parallel atomic operation of E processing cores with a predefined aggregation, such as sum, or maximum. Given an index tensor $\mathbf{I} \in \mathbb{R}^E$ and a node feature tensor $\mathbf{X} \in \mathbb{R}^{n\times d}$ as input, a processing core j combines its computed result $\mathbf{m}_j \in \mathbb{R}^d$ (here, the message) with the value at $\mathbf{X}[\mathbf{I}[j], :]$. Since multiple cores can access the same index, the aggregation is performed using atomic operations that can be considered to have $O(1)$ time complexity on modern GPUs. We say *scatter-add* or *scatter-max* to describe scatter operations with sum or maximum aggregation, respectively. Performing scatter with average aggregation is a combination of two scatter-add operations, one adding the values and the second counting the elements, followed by normalization in node space. Overall, we obtain an $O(1)$ runtime in graph size with $O(|\mathcal{E}|)$ processors for the whole message passing procedure, including Message, Update functions and parallel gather/scatter.

For training, the framework also provides an efficient reverse accumulation scheme. Intuitively, the reverse operation of scatter is gather and the reverse operation of gather is scatter. In more detail, the backward operation of gathering node features to edge space is a scatter-add operation, adding the sensitivities of Message function inputs to sensitivities of node features in node space. The backward function of scattering messages from edge to node space using the aggregation function $\square$

depends on the aggregation type. For sum aggregation, the backward function simply gathers sensitivities into edge space, using the same indices as for the forward scatter-add. For maximum aggregation, a masked gather can be used, only keeping sensitivities of features that were the maximum during the forward pass.

The efficient implementation of the MP-GNN framework based on scatter and gather operations is the core of our Pytorch Geometric (PyG) library [FL19], an extension for Pytorch [PGM+19], allowing to easily customize individual MP-GNN models for the given task and train them automatically. Since creation, the library was developed further and includes several optimizations and helpful tools for learning on graph structured data.

MP-GNN Properties

In the following, further useful properties of the MP-GNN framework further are detailed.

Permutation Equivariance The neighborhood aggregation operator is permutation invariant, that is invariant to the order of neighboring graph nodes. The only other function over nodes is the update function, which is shared for all nodes in the graph. Therefore, MP-GNN functions built with the given framework are equivariant with respect to the symmetric group S_n. Formally, for an MP-GNN f, consisting of multiple message passing layers, that maps input node features $\mathbf{X} = (\mathbf{x}_v)_{v \in V}^\top$ with adjacency $\mathbf{A}$ to output node features $\mathbf{Y} = (\mathbf{y}_v)_{v \in V}^\top$ with $\mathbf{Y} = f(\mathbf{X}, \mathbf{A})$, permuting the input features and adjacency matrix $\mathbf{A}$ with a permutation matrix $\mathbf{P}$ is equal to permuting the output:

$$f(\mathbf{P} \cdot \mathbf{X}, \mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}^\top) = \mathbf{P} \cdot f(\mathbf{X}, \mathbf{A}). \tag{4.2}$$

This is not only useful for graphs but also for all the geometric data types mentioned in Section 4.1.1, since in point clouds, meshes, manifolds and other geometric graphs, the order of nodes, points, or entities does not carry any information either. Thus, the MP-GNN framework respects the natural symmetries of the input data types by introducing a fitting inductive bias.

Varying neighborhood sizes Since message functions are shared over edges, node functions are shared over nodes and the aggregation is agnostic of number of neighbors, varying neighborhood sizes in a graph, or between training and test graphs, can be processed by design without any additional computation overhead. This is especially useful for point clouds, since it allows the usage of radius instead of nearest neighbor graphs, where the number of neighbors within a radius r is different for each neighborhood.

Locality Operators built with the MP-GNN framework are local operators, where locality is induced by the adjacency of the underlying graphs. If the graph as a

geometric embedding and the adjacency is defined by distance or neighbor relations, the operators are local in the underlying space. Due to locality, MP-GNN operators can be applied to partial graphs, even if they were trained on larger training data.

CNNs as Message Passing It should be noted that classical CNNs can be interpreted as message passing on a grid graph. Although the dedicated, highly optimized implementations of the CNN convolution operator are certainly more efficient, viewing it as message passing might help to advance the intuitions for the operators presented in the following sections. Let $G = (\mathcal{V}, \mathcal{E})$ be a grid graph with $\mathcal{V} = \{\mathbf{p}_1, ...\mathbf{p}_n\}$ containing one node for each pixel of the input image and $\mathcal{E} = \{\mathbf{e}_1, ...\mathbf{e}_n\}$ containing directed edges between a pixel and its eight neighbors and self-edges. Let further $\mathbf{K} \in \mathbb{R}^{M^\ell \times M^{\ell-1} \times 3 \times 3}$ be the 4-dimensional filter tensor of a 3×3-convolutional layer with $M^{\ell-1}$ input and M^ℓ output feature maps and $\mathbf{b}$ be the trainable bias vector. Then, the standard convolution operator for two-dimensional images can be described in the MP-GNN framework as

$$\mathbf{m}_{w,v} := \textsc{Me}_{\mathbf{K}}(\mathbf{x}_w, \mathbf{x}_v, \mathbf{e}_{w,v}) \quad = \mathbf{K}[:, :, y(\mathbf{e}), x(\mathbf{e})] \cdot \mathbf{x}_w, \tag{4.3}$$

$$\mathbf{x}_v^{(\ell)} := \textsc{Up}_{\mathbf{b}}(\mathbf{x}_v^{(\ell-1)}, \square_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}) = \sigma(\mathbf{b} + \sum_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}), \tag{4.4}$$

with $y(\mathbf{e})$ and $x(\mathbf{e})$ being functions mapping an edge to one of the 3×3 indices which determines its orientation (note that there are only 9 different orientation of edges in the grid graph). Essentially, this formulation is obtained by swapping sum operators in the original convolution expression. Instead of first multiplying each input feature map with the kernel window and then summing over the input feature maps, here, the pixel fibres $\mathbf{x}_w$ are multiplied by an $M^\ell \times M^{\ell-1}$ slice of the kernel first, before summing over the 9 pixel neighborhood.

Mini-batch handling. Efficient Deep Learning architectures heavily profit from the ability to be applied in a batch-wise fashion, that is utilizing parallelization over multiple input objects. It is not immediately clear how to apply the message passing scheme to multiple inputs in parallel without introducing significant overhead through differently sized graphs. However, it can be achieved without overhead by considering all input graphs of a batch as one single graph. By simply concatenating the feature matrices containing the $\mathbf{x}_v$, $\mathbf{e}_{v,w}$ and concatenating the list of edges (or by creating a block-diagonal matrix out of all adjacency matrices), a batch graph is obtained. It can be seen that message and update functions can still be applied in edge parallel or node parallel space, respectively. Even with graphs that have different topology in a single batch, no additional computation is required.

While there are several applications of MP-GNNs in the general graph domain, this thesis focuses on geometric data. When applied to 3D data, a MP-GNN operator consisting of Message and Update functions can have additional properties:

Extrinsic vs. Intrinsic An operator which is *extrinsic in rotation* computes the message function based on extrinsically defined relative Cartesian coordinates $\mathbf{e}_{i,j} = \mathbf{p}_j - \mathbf{p}_i$. The output of the operator changes if the input structure is rotated, i.e. the operator is not invariant to rotation. An operator can also be *extrinsic in translation* by utilizing extrinsic node positions as input features: $\mathbf{x}_i = \mathbf{p}_i$. The result is an operator which produces different results for different translations of the input structure. In contrast, an *intrinsic* operator neither depends on the extrinsically defined coordinate systems and thus is invariant to rotation and translation of the input structure. They can however utilize intrinsic coordinate frames that change together with global input transformation. For simplicity, operators defined *extrinsic* or *intrinsic* coordinate systems are referenced as being *extrinsic* or *intrinsic*, respectively, for the remainder of this thesis.

Anisotropic vs. Isotropic An *anisotropic* operator distinguishes the neighbors of a node by utilizing multi-dimensional relation information $\mathbf{e}_{i,j}$ (e.g. $\mathbf{e}_{i,j} = \mathbf{p}_j - \mathbf{p}_i$ on point clouds or meshes). It computes different messages for different edges in the same neighborhood and thus is usually more expressive than isotropic operators. In contrast, *isotropic* operators compute the message function purely based on the node input features or one-dimensional edge features, e.g. distances. It should be noted that *(an-)isotropy* can also be used to describe properties of pure graph operators, although in a more abstract fashion, where $\mathbf{e}_{i,j}$ are not necessarily spatial coordinates.

As an example, a geometrically correct and maximally expressive mesh convolution formulated as MP-GNN should be *intrinsic* and *anisotropic*. It should consider the full positional relation between between neighbors in the message function but should not rely on extrinsically defined coordinate systems. Instead, it needs to use an intrinsic frame of reference for each point, which is, however, often hard to obtain in practice. For a more detailed discussion about mesh convolution, see Section 4.1.3. In the following, two examples of MP-GNNs for 3D applications are discussed, which were published by the author of this thesis.

4.1.3 Related Work

This section gives an overview about related operators in the GNN literature. In the original works, the operators are mostly framed in their own notations, which is why this work will provide important milestones in the unified MP-GNN framework, which was first mentioned in a similar form by Gilmer et al. [GSR+17], was then formulated as computational framework in our Pytorch Geometric library [FL19] and adopted by the wider GNN literature [Ham20]. The section is divided into four parts: operators for general graphs, those specialized on point clouds, mesh operators, and manifold convolutions.

Graph Operators

Although it is not the main focus of this work, this section gives a short overview about milestones in graph neural networks for general graphs. The first versions of spatial and spectral GNNs were presented by Bruna et al. [BZS+14]. The authors describe spectral and spatial versions of GNNs that do not utilize weight-sharing and are therefore restricted to fixed topology. Their operator is anisotropic though, meaning that different edges in a neighborhood receive different weights. In order to introduce weight-sharing and domain independence, the property of anisotropy was sacrificed by the next iteration of GNN operators, ChebNet [DBV16] and GCN [KW17], which are spatial low-rank approximations of the spectral GNNs described by Bruna et al. [BZS+14]. GCN can be described in the MP-GNN framework as

$$\mathbf{m}_{w,v} \coloneqq \mathrm{ME}\left(\mathbf{x}_w, \mathbf{x}_v, \mathbf{e}_{w,v}\right) = \frac{\mathbf{x}_w}{\sqrt{\deg(v) \cdot \deg(w)}}, \tag{4.5}$$

with $\deg(v)$ denoting the node degree of v. The update function, given a non-linear, element-wise activation function σ, computes the node features $\mathbf{x}_v$ as

$$\mathbf{x}_v^{(\ell)} = \mathrm{UP}_{\mathbf{W}}(\mathbf{x}_v^{(\ell-1)}, \square_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}) = \sigma(\mathbf{W} \cdot \sum_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}), \tag{4.6}$$

with trainable weight matrix $\mathbf{W}$, which are shared over all nodes. It is easy to see that the message function is non-trainable and not dependent on additional edge features, making this approach isotropic.

The next milestone for spatial GNNs was the introduction of attention mechanisms. A prominent example are Graph Attention Networks (GATs) [VCC+18], which utilize self-attention, that is computing edge features based on the features of the two adjacent nodes. This self-attention mechanism introduces a form of artificial anisotropy that can be used in case there are no edge features given in the data. GAT can be expressed in the MP-GNN framework by

$$\mathbf{m}_{w,v} \coloneqq \mathrm{ME}_{\mathbf{W},\mathbf{a}}\left(\mathbf{x}_w, \mathbf{x}_v\right) = \frac{\exp(\mathrm{LReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{x}_v \| \mathbf{W}\mathbf{x}_w]))}{\sum_{u \in \mathcal{N}(v)} \exp(\mathrm{LReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{x}_v \| \mathbf{W}\mathbf{x}_k]))} \mathbf{W}\mathbf{x}_w, \tag{4.7}$$

with LReLU denoting the leaky ReLU activation function, $\|$ denoting vector concatenation and $\mathbf{W}$, $\mathbf{a}$ are trainable parameters. In this formulation, the first term computes normalized edge attention scores based on node features, which weight each message before aggregation. The update function, given a non-linear, element-wise activation function σ, computes the node features $\mathbf{x}_v$ as

$$\mathbf{x}_v^{(\ell)} = \mathrm{UP}(\mathbf{x}_v^{(\ell-1)}, \square_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}) = \sigma(\sum_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}). \tag{4.8}$$

For graphs, the following research has gone in several different directions. Important areas include the analysis of operator expressiveness and the introduction of diffusion processes in GNNs. Regarding the former, several works exist that analyze the

power of GNNs to distinguish non-isomorphic subgraphs. Research has shown that several state of the art graph operators have the same distinguishing power as the well-known Weisfeiler-Lehman (WL) algorithm [MRF+19; XHL+19; WL68]. As a result, several higher-order operators have been proposed to provide even higher expressiveness [MRF+19; MSR+19; MBS+19; BFZ+20]. For the latter, APPNP [KBG19] is a prominent example for diffusion processes in GNNs. It is interesting to see that fixed or trainable diffusion is easily expressed in the MP-GNN framework and can therefore be integrated in GNN architectures without additional overhead. This can also be utilized for operators on manifolds and meshes, as described in the next sections. Since pure graph applications are not the focus of this work, the reader is referred to the exhaustive review literature [HYL17; Ham20; WPC+21] to get a more complete list of advancements in the field of GNNs for general graphs.

Point Cloud Operators

GNNs on point clouds were initially described by Qi et al. in their works about PointNet [QSK+17] and PointNet++[QYS+17]. PointNet is a permutation-invariant network that aggregates a set of points and computes a feature vector, containing informations about point patterns in the input. PointNet++ builds on top of PointNet and creates a hierarchical network consisting of PointNet Operators, sampling and grouping layers. The sampling step selects a subset of well-distributed points using *iterative farthest point sampling*, followed by grouping of k-NEAREST-NEIGHBOR (k-NN) sets around each of those points. Although initially not described as GNNs, PointNet and PointNet++ can be naturally expressed in the MP-GNN framework, where the graph is computed by sampling and grouping, that is the graph edges point from grouped points to their sampled centers. In contrast to the general graph case, we have a different set of edges for each network layer to describe the hierarchy. With $\mathbf{e}_{j,i} = \mathbf{p}_j - \mathbf{p}_i$, the message function is given as

$$\mathbf{m}_{j,i} := \text{ME}_{\Theta_1}(\mathbf{x}_j, \mathbf{x}_i, \mathbf{e}_{j,i}) = h_{\Theta_1}(\mathbf{x}_j \| \mathbf{e}_{j,i}), \tag{4.9}$$

where h_{Θ_1} is an MLP, Θ_1 the trainable parameters of h_{Θ_1}, and $\|$ denotes feature concatenation. The update function is given as

$$\mathbf{x}_i^{(\ell)} = \text{UP}_{\Theta_2}(\mathbf{x}_i^{(\ell-1)}, \square_{j \in \mathcal{N}(i)} \mathbf{m}_{j,i}) = \gamma_{\Theta_2}(\max_{j \in \mathcal{N}(i)} \mathbf{m}_{j,i}), \tag{4.10}$$

with another MLP γ_{Θ_2}, its trainable parameters Θ_2 and maximum aggregation. In contrast to the original formulation, the execution in the MP-GNN frameworks allows using radius-based grouping instead of k-NN sets without introducing computational overhead when the differences in neighborhood size is large.

Wang et al. [WSL+19] follow up on PointNet++ with their EdgeConv approach and allow the MLP to be conditioned on both point feature vectors (including global point positions) $\mathbf{x}_j, \mathbf{x}_i$, i.e.

$$\mathbf{m}_{j,i} := \text{ME}_{\Theta}(\mathbf{x}_j, \mathbf{x}_i, \mathbf{e}_{j,i}) = h_{\Theta_1}(\mathbf{x}_i \| \mathbf{x}_j \| \mathbf{e}_{j,i}), \tag{4.11}$$

making the network more expressive. If extrinsic point positions are included in features $\mathbf{x}_i, \mathbf{x}_j$, the whole operator becomes extrinsic in translation and is no longer invariant with respect to global translation.

After PointNet++, the field evolved into generalizing point cloud operators in the framework of continuous convolution [SK17; FLW+18; XFX+18; WDW+18; WSL+19; TQD+19; LFX+19]. These works have in common, that they define a continuous filter kernel for convolution, from which weights can be sampled for each point in a neighborhood. In the MP-GNN framework, this can be described by the message function

$$\mathbf{m}_{j,i} := \text{ME}_{\mathbf{K}}\left(\mathbf{x}_j, \mathbf{x}_i, \mathbf{e}_{j,i}\right) = \mathbf{K}(\mathbf{e}_{j,i}) \cdot \mathbf{x}_j, \tag{4.12}$$

where the filter matrix $\mathbf{K}$ can be conditioned on $\mathbf{e}_{j,i}$ in several different ways: Our SplineCNN [FLW+18], for instance, parameterizes $\mathbf{K}$ using B-spline surfaces, which will be the main topic of the following Section 4.1.4. In SpiderCNN by Xu et al. [XFX+18] the kernels are constructed as a combination of Taylor polynomials. In FeaStNet by Verma et al. [VBV18] the two adjacent node features are mapped to soft assignments, which build a linear combination of a fixed number of kernel matrices. Similarly, KPConv by Thomas et al. [TQD+19] constructs its kernels as a linear combination of matrices lying on regularly spaced points in the kernel domain, where the weights are chosen based on distance. They also propose a deformable version, in which the network learns to adequately shift the point positions. The biggest category, Simonovsky et al. [SK17], Wang et al. [WDW+18] and Liu et al. [LFX+19], defines the filter as an MLP, mapping the edge features $\mathbf{e}_{j,i}$ to filter matrix $\mathbf{K}$.

Another noteworthy operator specifically for point clouds is PointCNN by Li et al. [LGA+18], which sacrifices permutation invariance in local neighborhood in favor of a learned permutation of points. First, the relative point positions are mapped to a permutation matrix using an MLP, before the resulting matrix is applied to permute the feature vectors in the neighborhood. The approach follows a similar idea as our LSGTs (c.f. Section 4.1.5), producing elements of the point permutation group instead of rotation group.

Mesh and Manifold Operators

Operators on meshes or manifolds differ from those on point clouds by assuming and utilizing the existence of a surface. The core research question is how to choose local coordinate systems on the surface to make an operator truly intrinsic, without sacrificing anisotropy and expressiveness. From the perspective of MP-GNNs, the solutions can be divided into two categories: fixed topology approaches that assume a given, fixed mesh topology and topology agnostic approaches, which can handle varying topologies in training and validation data. At its core, the difference comes down to having unique vertex and edge identity. If the model is always trained on and applied to data lying on the same topology, we can give each node and/or each edge individual trainable parameters. Doing so, we usually obtain more expressive

operators in exchange for topology independence. Fixed topology approaches usually come into play when the task is to regress a function on an existing and fixed mesh, e.g. when animating or fitting an existing parameterizable 3D model. Topology agnostic methods are usually more useful for analyzing captured data, as we can not assume that the data follows a specific topology and because training and application topology can differ. They can be easily expressed in the MP-GNN framework, as they make use of the same weight sharing principle and invariances. Fixed topology approaches can also be formulated and executed in the framework, however, they (partially) sacrifice weight sharing across edges or nodes and are therefore more related to the original spatial GNNs of Bruna et al. [BZS+14].

The first GNN mesh operators for arbitrary topology where Geodesic CNNs (GCNNs) [MBB+15], Anisotropic CNNs (ACNNs) [BMR+16] and MoNet [MBM+17], introducing the term *Geometric Deep Learning* [BBL+17], which also sparked the creation of our extension library *Pytorch Geometric* [FL19]. The key difference to point cloud and graph operators is that they made an effort to be *intrinsic*. They construct surface patches in local coordinate systems, which are used to apply the filters, making the application invariant to global transformation. GCNNs [MBB+15] resolve the ambiguity in patch rotation by sacrificing anisotropy, as each filter is applied for a set of equidistant rotations before aggregating the results using angular pooling. ACNNs [BMR+16] are built on GCNNs and bring back anisotropy, by using the principal curvature of the surface as indicator for filter orientation and anisotropic heat kernels as filters. MoNet [MBM+17] expands on filter expressiveness and constructs the continuous kernel as a Gaussian mixture model. From a computation perspective, MoNet can be described by the message function

$$\mathrm{ME}_{\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k,\mathbf{g}}(\mathbf{x}_w,\mathbf{x}_v,\mathbf{e}_{w,v}) = \left(\sum_{k=1}^{K} g_k \exp\left(-\frac{1}{2}(\mathbf{e}_{\mathbf{w},\mathbf{v}}-\boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\mathbf{e}_{\mathbf{w},\mathbf{v}}-\boldsymbol{\mu}_k)\right)\right)\cdot\mathbf{x}_w, \quad (4.13)$$

as a part of continuous convolution using K Gaussian models as kernel function, with trainable parameters $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, for each $1 \leq k \leq K$ and $\mathbf{g}$. The update function is simply the sum aggregation over all neighbors, as usually seen in continuous convolution. Note that the formulation of convolution is independent of the chosen coordinate system for $\mathbf{e}_{w,v}$. If the intrinsic local coordinate system from ACNNs is used, all continuous convolution operators with anisotropic kernel functions (MoNet, SplineCNN, EdgeConv, and so on) can become intrinsic and anisotropic operators.

However, these local coordinate systems, with [MBB+15; SRC+20] or without angular pooling [BMR+16], suffer from the flaw that if a filter is transported along the surface, it arbitrarily changes orientation based on the circumstances at each vertex, be it maximum curvature or maximum angular activation. Therefore, there are multiple operators refining and extending the ideas of GCNNs and ACNNs. Harmonic Surface Networks (HSNs) [WEH20] and Gauge Equivariant Mesh CNNs (GEM-CNNs) [HWC+21] provide solutions to this issue by computing equivariant vector features in the tangent spaces of each vertex. Those features can be rotated based on local coordinate systems when parallel transported between neighbors,

making the whole approach invariant to local coordinate space conventions. The message function of HSNs and GEM-CNNs can be abstractly described in the MP-GNN framework as

$$\mathrm{ME}_\Theta\left(\mathbf{x}_w, \mathbf{x}_v, \mathbf{e}_{w,v}\right) = \mathbf{K}_\Theta(\alpha_{w,v}, r_{w,v}) \cdot p(w \to v) \cdot \mathbf{x}_w, \tag{4.14}$$

where $p(w \to v)$ parallel transports the complex-valued features $\mathbf{x}_w$ from the tangent space around w to the tangent space around v (considering the difference in frame of reference) and $\mathbf{K}_\Theta(\alpha_{w,v}, r_{w,v})$ is an equivariant filter set with trainable parameters Θ on the polar coordinates $\mathbf{e}_{w,v} = (\alpha_{w,v}, r_{w,v})$ of w in the tangent space of v. The update function utilizes sum aggregation and specific equivariance-preserving non-linearities (c.f. the original literature [WEH20; HWC+21] for the individual implementations).

Lastly, there are fixed topology approaches, most prominently SpiralNet [BBP+19; LDC+18] and SpiralNet++ [GCB+19]. While the original formulation of SpiralNet by Lim et al. [LDC+18] did not restrict itself to fixed topology scenarios, the specific characteristics of the operator makes it most useful in this domain [BBP+19; GCB+19]. All three works have in common that they sequentialize local mesh patches into spiraling sequences of vertices to introduce an order for designing a filter. The spirals for each vertex are not unique, which is why arbitrary choices have to be made when constructing them. Lim et al. [LDC+18] randomized those choices, Bouritsas et al. [BBP+19] designed a criterion based on geodesic distances, and Gong et al. [GCB+19] makes arbitrary decisions, as the spirals are only computed once for a mesh. Since all choices are not optimal when weights are shared over the neighborhoods, this line of methods found its strength in fixed topology applications, where weights can be learned for each spiral individually. In this domain, all issues that arise with arbitrary topology (finding local coordinate systems, parallel transporting filters and features) are circumvented by hard-coding the domain into the network architecture.

4.1.4 SplineCNN

In this section, SplineCNN is described, an anisotropic GNN operator for continuous convolution on 3D data. It can be either extrinsic or intrinsic, depending on the chosen coordinates. The content is adapted from the original publication [FLW+18], brought into the context of MP-GNNs. The success of CNNs in the 2D image domain heavily relies on the translation equivariance of the discrete 2D convolution operator. The goal of SplineCNN is to transfer this feature to irregularly structured data, such as point clouds, meshes and manifolds, in the 3D domain. One arising challenge when designing similar operators for those domains is to handle the fact that the data does not lie on a fixed grid. Therefore, the convolution has to be continuous, that is we need to convolve the input data with a kernel that is defined on a continuous range. SplineConv, which is defined in the following section, was

one of the first in a long list of operators that perform such continuous convolutions (cf. Section 4.1.3 for an overview of existing operators).

In the following, the SplineConv operator is described as MP-GNN operator. It expects the input to be a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{v_1, \dots, v_n\}$ being the set of nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ the set of edges, D-dimensional edge input features $(\mathbf{e}_{w,v} \in [0,1]^D)_{(w,v)\in\mathcal{E}}$ for each directed edge $(w, v) \in \mathcal{E}$, and node input features $(\mathbf{x}_v^0 \in \mathbb{R}^{M^0})_{v\in\mathcal{V}}$ for each node $v \in \mathcal{V}$. This formulation allows to apply the operator to all data types listed in Section 4.1.1, namely point clouds, meshes, manifolds and other types of geometric graphs.

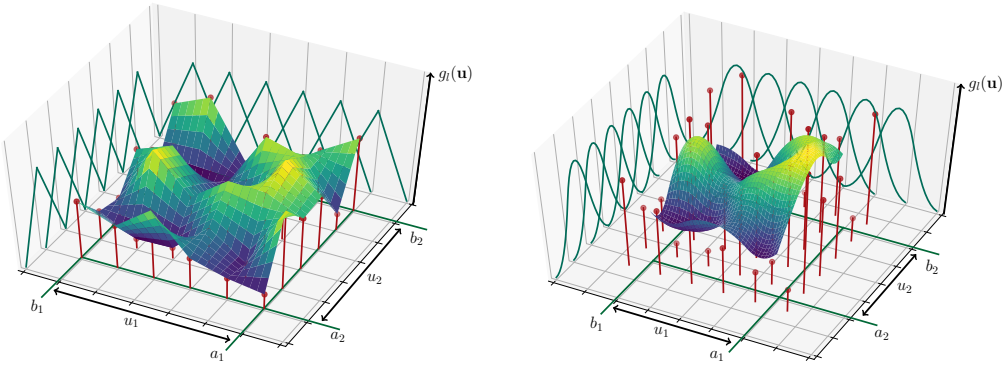
Edge input features For SplineConv, the edge input features $\mathbf{e}_{w,v}$ are of utmost importance, since they are used to sample the continuous kernel, which is defined over the range of those features. For geometric input data, such as point clouds and meshes, the kernel can therefore be defined over local Cartesian or polar spaces around input points. For more general graph data, the kernel may be defined over arbitrary feature spaces. Other than being element of a fixed interval range, we state no further restrictions to the contents of edge features $\mathbf{e}_{i,j}$. Note that, depending on the chosen coordinate systems for the edge features, the whole operator becomes intrinsic or extrinsic. If an intrinsic operator is required, additional effort has to be put in computing intrinsic coordinate frames for each point in a preprocessing step. The edge features of all edges are described by a tensor $\mathbf{E} = (\mathbf{e}_{w,v} \in [0,1]^D)_{(w,v)\in\mathcal{E}} \in \mathbb{R}^{|\mathcal{E}|\times D}$.

Node features We consider each node v to have an input feature vector $\mathbf{x}_v^0$, which describes a discrete sample of the M^0-dimensional input function. Layer ℓ of a SplineCNN computes an M^ℓ-dimensional node feature vector $\mathbf{x}_v^\ell$ for each node. The features of all nodes after layer ℓ can be described as tensor $\mathbf{X}^\ell = (\mathbf{x}_v^\ell)_{v\in\mathcal{V}} \in \mathbb{R}^{|\mathcal{V}|\times M^\ell}$. In analogy to CNNs, a slice $\mathbf{X}^\ell[:, i]$ of that tensor is also called a *feature map*.

B-Splines We shortly define B-spline bases as preliminary for SplineCNN (cf. Piegl et al. [PT97]). Let $((N_{1,k_1}^m)_{1\le k_1\le K_1}, \dots, (N_{D,k_D}^m)_{1\le k_D\le K_D})$ be D open B-spline bases of degree m, based on uniform, i.e. equidistant, knot vectors, with $\mathbf{K} = (K_1, \dots, K_D)$ defining our D-dimensional kernel size.

SplineConv Operator

To express the SplineConv operator in the MP-GNN framework, we need to define message and update functions. In general, the message function is responsible for two operations: Firstly, given a set of $M^\ell \times M^{\ell-1}$ kernel functions, it evaluates the kernels at the position given by the edge input feature $\mathbf{e}_{w,v}$. Then, the resulting matrix of kernel values is multiplied with the feature vector $\mathbf{x}_w$ of the neighboring node, forming the message. The messages coming to one node are aggregated using the sum or mean aggregation function, before passing through the update function, which adds a bias vector and applies a fixed, non-linear activation σ. It can be seen



(a) Linear B-spline basis functions **(b)** Quadratic B-spline basis functions

Figure 4.1: Examples of the continuous convolution kernel for B-spline basis degrees (a) $m = 1$ and (b) $m = 2$ and kernel dimensionality $D = 2$. The heights of the red dots are the trainable parameters for a single input feature map. They are multiplied by the elements of the B-spline tensor product basis (as seen on the sides) to form the kernel surface height. The resulting kernel surface is used for continuous convolution with the input feature maps [FLW+18].

that the whole process can be interpreted as convolution of input node features with a set of continuous kernels, if the edge features represent local Cartesian or polar coordinates. In the following, the process is described formally.

A single spline kernel is defined on a D-dimensional grid of size $K_1 \times \ldots \times K_D$ that spans the definition range of edge features $\mathbf{e} \in [0,1]^D$. A cell $(k_1, \ldots, k_D)$ is associated to the element $(N^m_{1,k_1}, \ldots, N^m_{D,k_D})$ of the Cartesian product of B-spline bases $(N^m_{1,k_1})_{1 \le k_1 \le K_1} \times \ldots \times (N^m_{D,k_D})_{1 \le k_D \le K_D}$. For presentation purposes, we introduce a linear index $k = (k_1 - 1) \cdot \prod_{d=2}^{D} K_d + (k_2 - 1) \cdot \prod_{d=3}^{D} K_d + \ldots + (k_D - 1)$ to uniquely identify grid cells with a single index, and denote the total number of cells as $K = K_1 \cdot \ldots \cdot K_D$. Further, each cell is associated with trainable parameters. Thus, for $M^\ell \times M^{\ell-1}$ kernels (one for each combination of input and output feature map), the trainable parameters are given as a tensor $\mathbf{W} \in \mathbb{R}^{K \times M^\ell \times M^{\ell-1}}$. Then, one evaluation of the B-spline kernel functions for an edge feature $\mathbf{e}$ is defined as

$$\mathbf{G}(\mathbf{e})_{i,j} = \sum_{k=0}^{K-1} \mathbf{W}[k,i,j] \cdot B_k(\mathbf{e}), \tag{4.15}$$

where B_k is the product of the spline basis functions that are associated to cell k:

$$B_k(\mathbf{e}) = \prod_{h=1}^{D} N^m_{h,k_h}(\mathbf{e}[h]). \tag{4.16}$$

In total, we obtain a tensor $\mathbf{G}(\mathbf{e}) \in \mathbb{R}^{M^\ell \times M^{\ell-1}}$ that contains the values of the $M^\ell \times M^{\ell-1}$ kernel functions at positions $\mathbf{e}_{w,v}$. One of such kernel function is shown in Figure 4.1 for different B-spline degrees and $D = 2$. The bases of different

degree are shown on the sides. They span a grid of trainable parameters, which are indicated by the height of the red points. The trainable parameters $\mathbf{W}[p, i, j]$ serve as local control values for the height of the resulting B-spline surfaces, which represents the kernel function. In total, for the $M^{\ell} \times M^{\ell-1}$ kernels in one layer, we have $P = M^{\ell} \cdot M^{\ell-1} \cdot K$ trainable parameters. The B-spline bases have local support [PT97], meaning that each trainable parameter only influences a limited part of the kernel surface, since the basis functions $B_k(\mathbf{e})$ are 0 outside of a small interval. The degree of locality depends on the B-spline degree m. In practice, this means that most summands in Equation (4.15) are zero and do not need to be computed in the first place. The indices of required summands can be determined from the spline degree m in a preprocessing step. Overall, the local support property provides two advantages: (1) in practice, only a small amount of basis functions need to be evaluated for each sample, improving efficiency in the forward step, and (2) since the output values only depend on a smaller subset of trainable parameters, the Jacobian of one evaluation is very sparse, leading to more efficient reverse mode accumulation (cf. Section 4.1.4 and Section 2.1).

Given the kernel function in Equation (4.15), we now introduce the MP-GNN formulation of SplineConv. The message function is defined as:

$$\mathbf{m}_{w,v} := \textsc{Message}_{\mathbf{W}}\left(\mathbf{x}_w^{\ell-1}, \mathbf{x}_v^{\ell-1}, \mathbf{e}_{w,v}\right) = \mathbf{G}(\mathbf{e}_{w,v}) \cdot \mathbf{x}_w^{\ell-1}. \tag{4.17}$$

The update function adds a trainable bias vector $\mathbf{b}$ and applies a non-linear, element-wise activation function. Thus, the final node features $\mathbf{x}_v^{\ell}$ are given as

$$\mathbf{x}_v^{\ell} = \textsc{Update}_{\mathbf{b}}(\mathbf{x}_v^{\ell-1}, \square_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}) = \sigma(\mathbf{b} + \sum_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}). \tag{4.18}$$

with sum (as given here) or mean aggregation. Recalling the message passing formulation for CNNs in Equation (4.3), an analogy between both formulations can be observed. Specifically, instead of having fixed weights $\mathbf{K}$ for each discrete edge case for CNNs, the weights $\mathbf{G}$ are computed continuously from the edge feature, motivating the term *continuous convolution*. What follows is that SplineConv is in fact a direct generalization of the discrete convolution on a grid, which can be applied to irregular domains.

Periodic edge features. One feature of the presented continuous B-spline kernels is that they are able handle periodic edge features $\mathbf{e}$. An obvious example of such a situation is, when the $\mathbf{e}$ represent point pair relations in polar coordinates. In these situations, single dimensions of $\mathbf{e}$ might contain angles for which it might be desired that values at both ends of the represented interval (angles 0 and 2π) evaluate to the same kernel value. In this case, closed B-spline approximation can be used to naturally achieve this goal. In practice this is done by introducing many-to-one mappings of border grid cells to a single parameter.

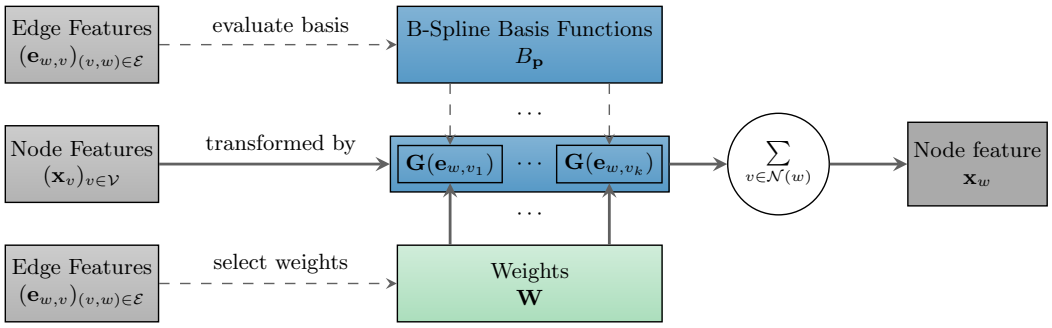


Figure 4.2: Forward computation scheme of the proposed convolution operation. During reverse mode, the sensitivities are computed along the inverted solid arrows, reaching trainable weights $\mathbf{W}$ and the input node features $\mathbf{x}_v$. Sensitivities for edge features $\mathbf{e}_{w,v}$ can theoretically be computed through the upper part of basis evaluation. However, this is not required for most applications, where $\mathbf{e}_{w,v}$ describes relative coordinates, which are fixed on the input domain.

Efficiency through local support The B-spline bases have local support [PT97], meaning that each trainable parameter only influences a limited part of the kernel surface, since $B_k = 0$ outside of a small interval. In detail, only $S := (m+1)^D$ of the K different cells are required to compute one kernel value. The required indices and values of trainable parameters can be found in constant time, which leads to a very efficient computation scheme to compute the matrix $\mathbf{G}(\mathbf{e})$ for each edge, as shown in Figure 4.2. The given computation scheme leads to the efficient GPU implementation of the MESSAGE function, which is explained in the next section.

GPU algorithm

As described in Section 4.1.2, we can obtain the input features $\mathbf{x}_v$ in edge parallel space efficiently by using the parallel gather operation. After computing the messages $\mathbf{m}_{v,w}$ in edge space we can efficiently aggregate them back to node space by using a parallel scatter operation. Both of those operations, as well as the UPDATE function have simple forward and backward functions (cf. Section 4.1.2). This section will therefore tackle the crucial part between those two operations, namely sampling of $\mathbf{G}(\mathbf{e})$ from the B-spline kernel functions and computing the messages in parallel over all edges in the graph. Since the whole approach needs to be differentiable, the backward function for gradient computation is of equal interest as the forward computation. The forward and backward message computations are shown in Algorithm 1 and Algorithm 2, respectively.

Computing the forward message function consists of two successive steps, where each run is parallelized over all edges. First, the identities of the S weights and basis functions that influence the kernel function at position $\mathbf{e}$ are computed. To this end, a function WEIGHT_INDICES is applied that finds those indices using a combination of scaling and rounding to the previous and next integers in each

Algorithm 1 SplineConv message function forward algorithm.

Input:
$M^{\ell-1}$: Number of input features per node
M^{ℓ}: Number of output features per node
$S = (m+1)^D$: Number of non-zero B_k for one edge
$\mathbf{W} \in \mathbb{R}^{K \times M^{\ell-1} \times M^{\ell}}$: Trainable weights
$(N^m_{1,k_1})_{1 \le k_1 \le K_1}, \cdots, (N^m_{D,k_D})_{1 \le k_D \le K_D}$: The D B-spline bases of degree m.
$\mathbf{E} \in \mathbb{R}^{E \times D}$: Input edge features for each edge
$\mathbf{X} \in \mathbb{R}^{E \times M^{\ell-1}}$: Gathered input node features for each edge
Output:
$\mathbf{M} \in \mathbb{R}^{E \times M^{\ell}}$: Messages for each edge

---

Parallelize over $e \in \{1, \dots, E\}$, $s \in \{1, \dots, S\}$:
  $\mathbf{B}[e, s] \leftarrow 1$
  $\mathbf{P}[e, s] \leftarrow$ WEIGHT_INDICES($\mathbf{E}[e, :]$)
  for each $d \in \{1, \dots, D\}$ **do**
    $\mathbf{B}[e, s] \leftarrow \mathbf{B}[e, s] \cdot N_{d,\text{BASIS\_INDEX}(s,d)}(\mathbf{E}[e, d])$
  end for
Parallelize over $e \in \{1, \dots, E\}$, $o \in \{1, \dots, M^{\ell}\}$:
  $r \leftarrow 0$
  for each $i \in \{1, \dots, M^{\ell-1}\}$ **do**
    for each $p \in \{1, \dots, S\}$ **do**
      $w \leftarrow \mathbf{W}[\mathbf{P}[e, p], i, o]$
      $r \leftarrow r + (\mathbf{X}[e, i] \cdot w \cdot \mathbf{B}[e, p])$
    end for
  end for
  $\mathbf{M}[e, o] \leftarrow r$
Return $\mathbf{M}$

dimension. Those indices allow us to only consider weights which have non-zero basis elements at position **e**. Then, for each weight, the product of basis functions over all d dimensions of the kernel is computed. To evaluate the basis functions, explicit polynomial representations are used, depending on the spline degree m. The second step gathers the S weights from $\mathbf{W}$ using indices $\mathbf{P}$, multiplies them with the basis elements stored in $\mathbf{B}$, the node input features $\mathbf{X}$ and sums them up. After adding the values for all input features, the resulting messages are stored in $\mathbf{M}$.

The algorithm has a parallel time complexity of $\mathcal{O}(S \cdot M^{\ell-1})$, with small S, using $\mathcal{O}(E \cdot M^{\ell})$ processors, thus a constant runtime in input graph size with a linear amount of processors in number of graph edges.

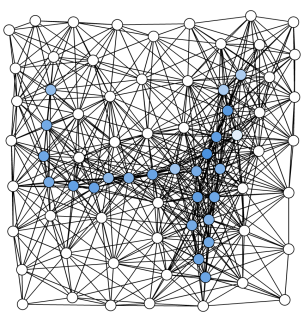
The backward computation in Algorithm 2 requires slightly more operations than the forward part. It is described how to obtain the partial derivatives with respect to all three inputs of the operation, namely the derivatives with respect

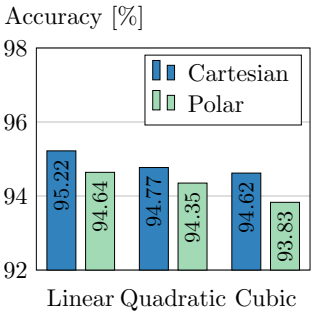
Algorithm 2 SplineConv message function backward algorithm.

Input:
E: Number of edges, $M^{\ell-1}$, M^{ℓ}: Number of input/output feature maps
$S = (m+1)^D$: Number of non-zero B_k for one edge
$\tilde{\mathbf{M}} \in \mathbb{R}^{E \times M^{\ell}}$: Partial derivatives w.r.t. $\mathbf{M}$
$(N^m_{1,k_1})_{1 \le k_1 \le K_1}, \ldots, (N^m_{D,k_D})_{1 \le k_D \le K_D}$: The D B-spline bases of degree m.
$(\tilde{N}^m_{1,k_1})_{1 \le k_1 \le K_1}, \ldots, (\tilde{N}^m_{D,k_D})_{1 \le k_D \le K_D}$: The first derivatives of the D B-spline bases.
Kept from forward:
$\mathbf{B} \in \mathbb{R}^{E \times S}$: Basis products of s weights for each edge
$\mathbf{P} \in \mathbb{N}^{E \times S}$: Indices of s weights in $\mathbf{W}$ for each edge
$\mathbf{W} \in \mathbb{R}^{K \times M^{\ell-1} \times M^{\ell}}$: Trainable weights
$\mathbf{X} \in \mathbb{R}^{E_{\text{in}}}$: Gathered node input features for each edge
Output:
$\tilde{\mathbf{W}} \in \mathbb{R}^{K \times M^{\ell-1} \times M^{\ell}}$, $\tilde{\mathbf{X}} \in \mathbb{R}^{N \times M^{\ell-1}}$, $\tilde{\mathbf{E}} \in \mathbb{R}^{K \times M^{\ell-1} \times M^{\ell}}$: Part. deriv. w.r.t. $\mathbf{W}$, $\mathbf{X}$, $\mathbf{E}$

Parallelize over $e \in \{1, ..., E\}$ and $o \in \{1, ..., M^{\ell}\}$:
  for each $i \in \{1, ..., M^{\ell-1}\}$ **do**
    for each $p \in \{1, ..., S\}$ **do**
      $\tilde{\mathbf{B}}[e,p] \leftarrow \tilde{\mathbf{B}}[e,p] + \tilde{\mathbf{M}}[e,o] \cdot \mathbf{W}[\mathbf{P}[e,p], i, o] \cdot \mathbf{X}[e,o]$
      $\tilde{\mathbf{X}}[e,i] \leftarrow \tilde{\mathbf{F}}[\mathbf{e}, \mathbf{i}] + \tilde{\mathbf{M}}[e,o] \cdot \mathbf{W}[\mathbf{P}[e,p], i, o] \cdot \mathbf{B}[e,p]$
      $\tilde{\mathbf{W}}[\mathbf{P}[e,p], i, o] \leftarrow \tilde{\mathbf{W}}[\mathbf{P}[e,p], i, o] + \tilde{\mathbf{M}}[e,o] \cdot \mathbf{X}[e,i] \cdot \mathbf{B}[e,p]$
    end for
  end for
Parallelize over $e \in \{1, ..., E\}$ and $\hat{d} \in \{1, ..., d\}$:
  $\tilde{\mathbf{E}}[e, \hat{d}] \leftarrow 0$
  for each $s \in \{1, \ldots, S\}$ **do**
    $g \leftarrow \tilde{\mathbf{B}}[e,s] \cdot \tilde{N}_{\hat{d}, \text{Basis\_Index}(s,\hat{d})}(\mathbf{E}[e, \hat{d}])$
    for each $\tilde{d} \in \{1, \ldots, d\} \backslash \{\hat{d}\}$ **do**
      $g \leftarrow g \cdot N_{d, \text{Basis\_Index}(s,d)}(\mathbf{E}[e,d])$
    end for
    $\tilde{\mathbf{E}}[e, \hat{d}] \leftarrow \tilde{\mathbf{E}}[e, \tilde{d}] + g$
  end for
Return $\tilde{\mathbf{X}}$, $\tilde{\mathbf{W}}$, $\tilde{\mathbf{E}}$

to input node features $\tilde{\mathbf{X}}$, trainable weights $\tilde{\mathbf{W}}$, and input edge features $\tilde{\mathbf{E}}$. The first two are required in most operations, since gradient information needs to be propagated to earlier operations through $\tilde{\mathbf{X}}$ and $\tilde{\mathbf{W}}$ is needed to update the weights. The third part is only required for specific operations where we need gradient information with respect to the coordinates stored at edges. This might be the case for applications in which we are interested to either learn them directly, or learn a preceding algorithm that predicts them. In most cases this is not the case





Dataset	Accuracy
MoNet	91.11%
SplineCNN	**95.22%**

(a) MNIST superpixels **(b)** Comparison **(c)** Classification accuracy

Figure 4.3: MNIST 75 superpixels (a) example and (b) classification accuracy of SplineCNN using varying pseudo-coordinates and B-spline base degrees. (c) Classification accuracy on different representations of the MNIST dataset (grid and superpixel) for a classical CNN (LeNet5), MoNet and our SplineCNN approach [FLW+18].

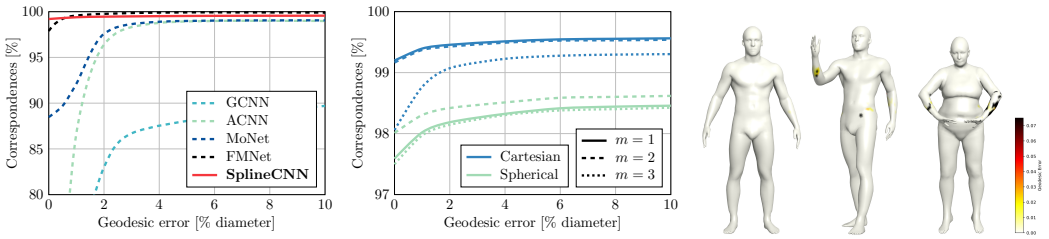
and we can omit the second parallel loop and computation of $\tilde{\mathbf{B}}[e,p]$ completely. The algorithm computes the individual backward steps of all the operations in the forward step, utilizing stored indices and values. Further, it requires the first derivatives $(\tilde{N}^m_{1,h})_{1\le h\le k_1},\cdots,(\tilde{N}^m_{d,h})_{1\le h\le k_D}$ of the B-spline basis functions, which are explicitly hard-coded in the kernel for different spline degrees. It should be noted that in case of degree $m = 1$, the partial derivatives $\tilde{\mathbf{B}}[e,p]$ are not well-defined at the grid centers. However, this is not an issue during computation as we just chose a surrogate derivative from two nearby basis functions, similar to how we deal with ReLU activation (cf. surrogate gradients in Section 2.2).

Under the realistic assumption $D < M_{\text{in}}$, the backward algorithm has a runtime complexity of $\mathcal{O}(S \cdot M^{\ell-1})$ using $\mathcal{O}(E \cdot M^{\ell})$ processors. Similar to the forward computation, it keeps constant runtime in input graph size with a linear amount of processors in number of graph edges.

In the following, we present two simple example architectures utilizing SplineConv. An additional application can be found in Chapter 6, where SplineCNN is used as part of a capsule network architecture. We denote a SplineConv layer with 2-dimensional kernels using the pattern $\text{SConv}((K_1,K_2),M^{\ell-1},M^{\ell})$, where K_1,K_2 denote the kernel size (number of trainable control points in the two dimensions), $M^{\ell-1}$ denotes the number of input feature maps and M^{ℓ} the number of output feature maps.

Example Architecture: Superpixel Graphs

Analogously to CNNs on images, we can use SplineCNN as convolution operator on superpixel graphs obtained from images, as shown as an example in Figure 4.3a for the MNIST dataset [LCB10]. This experiment was published in our original



(a) Results of SplineCNN and other methods

(b) Results for different SplineCNNs

(c) Geodesic error of test examples

Figure 4.4: Geodesic error plots of the shape correspondence experiments with (a) SplineCNN and related approaches and (b) different SplineCNN experiments. The horizontal axis displays the geodesic distance in % of diameter and the vertical axis the percentage of correspondences that lie within a given geodesic radius around the correct node. SplineCNN achieves the highest accuracy for low geodesic error and significantly outperforms other general approaches like MoNet, GCNN and ACNN. In Figure (c), three examples of the FAUST test dataset with geodesic errors of SplineCNN predictions for each node are presented. We show the best (left), the median (middle) and worst (right) test example, sorted by average geodesic error [FLW+18].

SplineCNN publication [FLW+18]. In order to imitate a CNN, operators for pooling on graphs are needed. Several graph pooling operators exist since it is still an area of active research (see [Ham20] for an overview). For this application, we chose the Graclus method for graph coarsening [DGK07; DBV16] and pool the feature values from the original nodes to their respective nodes in the coarsened graph. A pooling layer is denoted by MaxP(C), where C is the cluster size. Then, the architecture to classify superpixel graphs is of the format SConv($(K_1, K_2), 1, 32$) → MaxP(4) → SConv($(K_1, K_2), 32, 64$) → MaxP(4) → AvgP → Lin(128) → Lin(10), where AvgP denotes global averade pooling over all remaining nodes and Lin(O) are fully-connected linear layers with O output neurons. As non-linear activation after each SplineConv layer, the Exponential Linear Unit (ELU) is used. The results of the method are presented in Figure 4.3b and Table 4.3c. On the MNIST superpixel dataset [MBM+17], the presented architecture achieves 95.22% test accuracy and beats the previous state of the art by 4.11 percentage points. Further, the comparison shows that a spline degree of one together with Cartesian coordinates already suffices to produce the best accuracy.

Example Architecture: Shape Correspondence

The second architecture example was created to solve the problem of finding shape correspondence on the Faust mesh dataset [BRL+14] via mesh node classification and was published in our original SplineCNN publication [FLW+18]. The dataset consists of a set of 100 three-dimensional fixed-topology meshes with 6890

nodes each. The 100 meshes are obtained by fitting a template mesh to 10 different persons with 10 different poses each. The goal of the benchmark task [MBM+17; BMR+16; MBB+15; LRR+17] is to obtain the node id for each node only from the mesh data and thus finding the correspondences to the template mesh. The first 80 subjects are used for training while the other 20 are used for validation. We applied a 6-layer SplineCNN architecture with 3-dimensional kernels of the form $\text{SConv}((K_1, K_2, K_3), 1, 32) \rightarrow \text{SConv}((K_1, K_2, K_3), 32, 64) \rightarrow 4\times \text{SConv}((K_1, K_2, K_3), 64, 64) \rightarrow \text{Lin}(256) \rightarrow \text{Lin}(6890)$ directly on the input mesh, mapping to a distribution over 6890 for each vertex. At time of publication, SplineCNN achieved state of the art accuracy on this task, with 99.2% of found correspondences having zero geodesic error in the Princeton benchmark protocol [KLF11; FLW+18], as shown in Figure 4.4.

4.1.5 Local Spatial Graph Transformers

This section introduces Local Spatial Graph Transformers (LSGTs), a MP-GNN operator that was developed to serve as a spatial transformer in local 3D neighborhoods of point clouds. It was initially presented to re-weight the inputs to an IRLS problem [LOM20] for surface normal estimation. However, the concept behind LSGTs is generally applicable and therefore given here on its own. As described in Section 4.1.2, MP-GNN operators for geometric data can make use of *extrinsic* or *intrinsic* coordinate systems. If we want to design filters on surfaces, e.g. meshes or points assumed to be sampled from a manifold, they should be invariant to global object rotation and translation, which prohibits the use of extrinsic coordinate systems. At the same time, they should be as expressive as possible, retaining anisotropy. An intrinsic, anisotropic operator on a 3D geometric graph needs to rely on local frames of reference that are defined for each node individually and translate/rotate with global object transformation, that is, behaving equivariant to global rotation and translation (cf. Section 3.2 for a formal background). There are multiple ways to compute such reference frames $\mathbf{R}_i$ using non-trainable algorithms, such as gathering local point/surface statistics and curvature or by moving a reference frame along the surface using parallel transport. A review of those operators is given in Section 4.1.3.

LSGTs, which are presented in this section, are trainable GNNs that receive rotationally extrinsic coordinates from local neighborhoods and predict local reference frames for each point in the geometric graph. They are inspired by the spatial transformer architectures in the image domain [JSZ+15], which initially showed that DNNs are able to succeed in estimating pose and use that pose to canonicalize the input to another DNN, trained end-to-end. Especially on point clouds, where parallel transport and curvature statistics are hard to compute and rely on several assumptions, LSGTs provide an efficient alternative to estimate the local frames of reference. It should be noted that LSGTs do not formally guarantee to produce equivariant frames but that they are merely a network that is trained to produce

an estimation of equivariant frames. However, as shown in the surface normal application in this thesis (cf. Chapter 5), they provide good results while being efficient and without relying on further assumptions.

LSGTs can be described in the MP-GNN framework, where the relevant computations happen in the last update function. For illustrative purposes we first chose an arbitrary MP-GNN scheme fit for point cloud processing (here, a variant of PointNet++ [QYS+17], cf. Section 4.1.3). Given a geometric graph with points $\mathcal{P}$ and edge input features $\mathbf{e}_{w,v} = \mathbf{p}_v - \mathbf{p}_w$, the message function is formulated as

$$\mathbf{m}_{w,v} := \mathrm{ME}_{\Theta_1}(\mathbf{x}_w, \mathbf{x}_v, \mathbf{e}_{w,v}) = h_{\Theta_1}\big(\mathbf{x}_w \,\|\, \mathbf{e}_{w,v}\big), \tag{4.19}$$

where h_{Θ_1} is an MLP with parameters Θ_1. Further, the update function is given by another MLP γ_{Θ_2}, with parameters Θ_2, after mean aggregation of messages:

$$\mathbf{x}_v^\ell := \mathrm{UP}_{\Theta_2}(\mathbf{x}_v^{\ell-1}, \square_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}) = \gamma_{\Theta_2}\Big(\frac{1}{|\mathcal{N}(v)|} \sum_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}\Big). \tag{4.20}$$

The goal is that the architecture outputs local reference frames after L layers. Therefore, we need to parameterize the group of 3D rotations $SO(3)$, using vector space elements $\mathbf{x}_v^L \in \mathbb{R}^d$. There are several different options to parameterize $SO(3)$. These include using Euler angles, axis-angle representations or unit quaternions, each of which have individual up- and downsides. In this case, we want the parameterization to be continuously differentiable, which is why we chose the double cover parameterization of unit quaternions (cf. Section 3.2). To this end, we choose $d = 4$ and the output vectors $\mathbf{x}_v^L \in \mathbb{R}^4$ are normalized to unit length in order to obtain the unit quaternion: $\mathbf{q}_v = \mathbf{x}_v^L / \|\mathbf{x}_v^L\|$. Then, we apply a differentiable quaternion to rotation matrix map. Forward and backward computations as well as vector normalization are performed in parallel over the nodes, and thus we can extend the last MP-GNN update function to produce a rotation matrix $\mathbf{R}$:

$$\mathbf{R} := \mathrm{UP}^L_{\Theta_2}(\mathbf{x}_v^{L-1}, \square_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}) = \rho_{SO(3)}\left(\frac{\gamma_{\Theta_2}\big(\frac{1}{|\mathcal{N}(v)|}\sum_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}\big)}{\Big\|\gamma_{\Theta_2}\big(\frac{1}{|\mathcal{N}(v)|}\sum_{w \in \mathcal{N}(v)} \mathbf{m}_{w,v}\big)\Big\|}\right), \tag{4.21}$$

where $\rho_{SO(3)}$ is the representation of unit quaternions in the $SO(3)$ rotation group. The forward and backward algorithms for the last update function $\mathrm{UP}^L_{\Theta_2}$ are shown in Algorithm 3 and 4. The forward algorithm uses the standard map from quaternions to rotation matrices in parallel over all nodes of the input graph. The backward algorithm applies the Jacobians of the individual forward operations. It computes the sensitivities $\tilde{\mathbf{X}} = \frac{dL}{d\mathbf{X}}$, given sensitivities $\tilde{\mathbf{R}} = \frac{dL}{d\mathbf{R}}$, utilizing stored intermediate results for input vectors $\mathbf{X}$ and unit quaternions $\mathbf{Q}$ from the forward step.

After computing rotation matrices for each point, the network can be trained either directly or by feeding the matrices into a down-stream task and optimizing the weights to solve this task well. Given row-wise input points $\mathbf{P} \in \mathbb{R}^{n \times 3}$, edges

Algorithm 3 Parallel forward parameterization of rotation matrices.

Input:
N: Number of nodes/input quaternions
$\mathbf{X} \in \mathbb{R}^{N\times 4}$: Node Features for each node
Output:
$\mathbf{R} \in \mathbb{R}^{N\times 4\times 4}$: Rotation matrices for each node

---

Parallelize over $n \in \{1,\dots,N\}$:
$\mathbf{Q}[n,:] \leftarrow \frac{\mathbf{X}[n,:]}{\|\mathbf{X}[n,:]\|_2}$
$\mathbf{R}[e,0,0] \leftarrow 1 - 2\cdot(\mathbf{Q}[n,2]\cdot\mathbf{Q}[n,2] + \mathbf{Q}[n,3]\cdot\mathbf{Q}[n,3])$
$\mathbf{R}[e,0,1] \leftarrow 2\cdot(\mathbf{Q}[n,1]\cdot\mathbf{Q}[n,2] + \mathbf{Q}[n,0]\cdot\mathbf{Q}[n,3])$
$\mathbf{R}[e,0,2] \leftarrow 2\cdot(\mathbf{Q}[n,1]\cdot\mathbf{Q}[n,3] + \mathbf{Q}[n,0]\cdot\mathbf{Q}[n,2])$
$\mathbf{R}[e,1,0] \leftarrow 2\cdot(\mathbf{Q}[n,1]\cdot\mathbf{Q}[n,2] + \mathbf{Q}[n,0]\cdot\mathbf{Q}[n,3])$
$\mathbf{R}[e,1,1] \leftarrow 1 - 2\cdot(\mathbf{Q}[n,1]\cdot\mathbf{Q}[n,1] + \mathbf{Q}[n,3]\cdot\mathbf{Q}[n,3])$
$\mathbf{R}[e,1,2] \leftarrow 2\cdot(\mathbf{Q}[n,2]\cdot\mathbf{Q}[n,3] + \mathbf{Q}[n,0]\cdot\mathbf{Q}[n,1])$
$\mathbf{R}[e,2,0] \leftarrow 2\cdot(\mathbf{Q}[n,1]\cdot\mathbf{Q}[n,3] + \mathbf{Q}[n,0]\cdot\mathbf{Q}[n,2])$
$\mathbf{R}[e,2,1] \leftarrow 2\cdot(\mathbf{Q}[n,2]\cdot\mathbf{Q}[n,3] + \mathbf{Q}[n,0]\cdot\mathbf{Q}[n,1])$
$\mathbf{R}[e,2,2] \leftarrow 1 - 2\cdot(\mathbf{Q}[n,1]\cdot\mathbf{Q}[n,1] + \mathbf{Q}[n,2]\cdot\mathbf{Q}[n,2])$
Return $\mathbf{R}$

$\mathcal{E} \subseteq \mathcal{P}\times\mathcal{P}$ and row-wise, rotationally extrinsic relative coordinates $\mathbf{E} \in \mathbb{R}^{|\mathcal{E}|\times 3}$, an example for a task-agnostic training loss is

$$L = \sum_{i=1}^{n} \left\| \textsc{lsgt}(\mathbf{P},\mathcal{E},\mathbf{E})_i^\top \cdot \mathbf{R}^\top \cdot \textsc{lsgt}(\mathbf{R}\cdot\mathbf{P},\mathcal{E},\mathbf{R}\cdot\mathbf{E})_i - \mathbf{I} \right\|_F, \tag{4.22}$$

with $\mathbf{R}$ being uniformly sampled from $SO(3)$ in each training step and $\mathbf{I}$ being the identity. The network is directly trained to behave equivariant with respect to $SO(3)$, that is, to provide outputs that rotate by the same amount as the input was rotated (cf. Section 3.2). Note that this does not define an extrinsic rotation space, as the absolute outputs of $\textsc{lsgt}(\mathbf{P},\mathcal{E},\mathbf{E})$ are not constrained. It only trains the network to produce rotations that are correct in relation to rotations of the input. If training is done through a down-stream task, a straight-forward example would be to use the frame of reference for continuous convolution: Instead of sampling from a kernel function using extrinsically defined $\mathbf{e}_{j,i} = \mathbf{p}_i - \mathbf{p}_j$, an arbitrary kernel function $\mathbf{K}$ can be sampled using the estimated intrinsic coordinates $\mathbf{e}_{j,i} = \mathbf{R}_i(\mathbf{p}_i - \mathbf{p}_j)$ and used in continuous convolution in the MP-GNN framework as

$$\mathbf{m}_{j,i} := \textsc{Me}_{\mathbf{K}}(\mathbf{x}_j,\mathbf{x}_i,\mathbf{e}_{j,i}) = \mathbf{K}(\mathbf{R}_i(\mathbf{p}_i - \mathbf{p}_j)\cdot\mathbf{x}_j), \tag{4.23}$$

before trained against a down-stream loss after further processing.

Symmetries An additional advantage of local frames estimated with LSGTs is that they can deal with symmetries in the input naturally, depending on the given

Algorithm 4 Parallel backward algorithm of rotation parameterization.

Input:
N: Number of nodes/input quaternions
$\tilde{\mathbf{R}} \in \mathbb{R}^{N\times 4\times 4}$: Partial derivatives w.r.t. $\mathbf{R}$
Kept from forward:
$\mathbf{X} \in \mathbb{R}^{N}$: Node feature vectors.
$\mathbf{Q} \in \mathbb{R}^{N\times 4}$: Unit quaternions for each node
Output:
$\tilde{\mathbf{X}} \in \mathbb{R}^{N\times 4}$: Partial derivatives w.r.t. $\mathbf{X}$

Parallelize over $n \in \{1, ..., N\}$:

$$\begin{aligned}
\tilde{\mathbf{Q}}[n,0] \leftarrow\ & 2\cdot(-\mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,0,1] + \mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,0,2] + \mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,1,0] \\
& -\mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,1,2] - \mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,2,0] + \mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,2,1]) \\
\tilde{\mathbf{Q}}[n,1] \leftarrow\ & 2\cdot(\mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,0,1] + \mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,0,2] + \mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,1,0] \\
& -2\cdot\mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,1,1] - \mathbf{Q}[n,0]\cdot\tilde{\mathbf{R}}[n,1,2] + \mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,2,0] \\
& +\mathbf{Q}[n,0]\cdot\tilde{\mathbf{R}}[n,2,1] + 2\cdot\mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,2,2]) \\
\tilde{\mathbf{Q}}[n,2] \leftarrow\ & 2\cdot(-2\cdot\mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,0,0] + \mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,0,1] + \mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,0,2] \\
& +\mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,1,0] - \mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,1,2] - \mathbf{Q}[n,0]\cdot\tilde{\mathbf{R}}[n,2,0] \\
& +\mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,2,1] - 2\cdot\mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,2,2]) \\
\tilde{\mathbf{Q}}[n,3] \leftarrow\ & 2\cdot(-2\cdot\mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,0,0] - \mathbf{Q}[n,0]\cdot\tilde{\mathbf{R}}[n,0,1] + \mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,0,2] \\
& +2\cdot\mathbf{Q}[n,0]\cdot\tilde{\mathbf{R}}[n,1,0] - 2\cdot\mathbf{Q}[n,3]\cdot\tilde{\mathbf{R}}[n,1,1] + \mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,1,2] \\
& +\mathbf{Q}[n,1]\cdot\tilde{\mathbf{R}}[n,2,1] + \mathbf{Q}[n,2]\cdot\tilde{\mathbf{R}}[n,2,2]) \\
\tilde{\mathbf{X}}[n,:] \leftarrow\ & \frac{\tilde{\mathbf{Q}}[n,:]}{\|\mathbf{X}[n,:]\|_2} - \frac{\tilde{\mathbf{Q}}[n,:]\odot\mathbf{X}[n,:]\odot\mathbf{X}[n,:]}{\|\mathbf{X}[n,:]\|_2^3}
\end{aligned}$$

Return $\tilde{\mathbf{X}}$

down-stream task. If there are rotational symmetries in the input (as they occur quite often on local point cloud patches), the quaternions essentially become a d-cover (with $d > 2$) instead of a 2-cover of the underlying symmetry group. If the following down-stream task is invariant to those symmetries, the loss surface on $SO(3)$ will have d local minima, one for each of the d correct quaternions representing the input orientation. Kernel functions, the usual operator following an LSGT, are invariant to symmetries in the input since they are functions.

PRFNet Architecture: Deep Equivariant Re-weighting for IRLS

The leading application of LSGTs in this thesis is the re-weighting for an IRLS surface normal estimation algorithm, which is the main topic of Chapter 5. In the given setting, the input is a point cloud with a least squares plane fitting problem given for each node, where each input point for each least squares problem is denoted by an incoming edge. Since the task is to re-weight all inputs for all least squares problems in the point cloud, weights for the next iteration and residuals of the previous solutions lie exactly on the edges of the graph. PRFNet

aims to perform this re-weighting using an MP-GNN, mapping edge input features to edge output features, utilizing LSGTs and an implicit parameterizable kernel function, as described in Section 4.3. PRFNet profits from the MP-GNN properties of permutation invariance, locality and the ability to process varying neighborhood sizes. Due to the permutation invariant aggregation function, the order of points has no influence on the result, as desired. Further, each neighborhood can have a different number of points, without introducing additional computational overhead. It is possible for training and testing data to have differently sized neighborhoods. The property of locality allows the network to be applied on partial point clouds, achieving equal results, even if it was trained on complete ones.

4.2 Differentiable Matrix Decompositions

After introducing GNNs and the GNN building blocks that are used in this thesis, we turn to the next concept, which are differentiable matrix decompositions. Matrix decompositions in general are a crucial part of many 3D vision algorithms. For instance, they are used to find solutions of least squares problems for, e.g. normal vector or fundamental matrix estimation [HZ03], to obtain spectral basis vectors of the Laplace-Beltrami Operator (LBO) on surfaces for spectral shape analysis [Lev06], and for dimensionality reduction using Principal Component Analysis (PCA) [Pea01]. The process of most matrix decompositions is differentiable, that is, we are able to compute the sensitivity with respect to the input matrix given sensitivities with respect to decomposed matrices, which will be the topic of this section. The field of differentiable matrix decomposition for deep learning is relatively new and contains several obstacles, which is why up till now the process is only applied in special cases. The section begins with introducing the necessary background in Section 4.2.1 and giving an overview about existing work in Section 4.2.2. Two special cases for which we can derive practical forward and backward steps are presented in Section 4.2.2, which find their application in differentiable IRLS approaches later in this thesis in Chapters 5 and 6.

4.2.1 Matrix Decompositions

This section covers the topics of SVD and ED for the special case of real matrices. We begin by introducing both concepts, before putting them into perspective of each other and describe the challenges that arise when computing sensitivities.

Singular Value Decomposition Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a matrix with real entries and $m \geq n$. Then, basic linear algebra states that the SVD of $\mathbf{A}$ given by

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^\top = \sum_{i=1}^{n} d_i \mathbf{U}_i \mathbf{V}_i^T \tag{4.24}$$

always exists, where $\mathbf{U} \in \mathbb{R}^{m \times n}$ contains n orthogonal *left singular vectors* $\mathbf{U}_i$ as columns, $\mathbf{D} \in \mathbb{R}^{n \times n}$ is a diagonal matrix that contains the n *singular values* d_i, and $\mathbf{V} \in \mathbb{R}^{n \times n}$ contains n orthogonal *right singular vectors* $\mathbf{V}_i$. As the sum formulation suggests, the SVD is invariant to permutation of singular values and their corresponding left and right singular vectors. Therefore, we assume that they are always sorted by descending magnitude of the singular values d_i.

In a differentiable computation graph, we usually map the matrix $\mathbf{A}$ (which is obtained by the preceding part of the algorithm, e.g. a DNN) to its decomposition $\mathbf{A} \mapsto (\mathbf{U}, \mathbf{D}, \mathbf{V})$, by computing the SVD. Thus, for a given scalar loss L as a function of $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$, the reverse accumulation step amounts to

$$\left(\frac{\partial L}{\partial \mathbf{U}}, \frac{\partial L}{\partial \mathbf{D}}, \frac{\partial L}{\partial \mathbf{V}}\right) \mapsto \frac{\partial L}{\partial \mathbf{A}}. \tag{4.25}$$

The backward computation can be performed independently from the chosen forward implementation. Specifically, even when using iterative schemes for performing the SVD, the sensitivities can still be obtained in closed-form using

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{A}} = &\left[\mathbf{U}\left(K \odot \left(\mathbf{U}^\top \frac{\partial L}{\partial \mathbf{U}} - \left(\frac{\partial L}{\partial \mathbf{U}}\right)^\top \mathbf{U}\right)\right)\mathbf{D} + \left(\mathbf{I}_m - \mathbf{U}\mathbf{U}^\top\right)\frac{\partial L}{\partial \mathbf{U}}\mathbf{D}^{-1}\right]\mathbf{V}^T \\ &+\mathbf{U}\left(\mathbf{I}_n \odot \frac{\partial L}{\partial \mathbf{D}}\right)\mathbf{V}^\top \\ &+\mathbf{U}\left[\mathbf{D}\left(\mathbf{K} \odot \left(\mathbf{V}^\top \frac{\partial L}{\partial \mathbf{V}} - \left(\frac{\partial L}{\partial \mathbf{V}}\right)^\top \mathbf{V}\right)\right)\mathbf{V} + \mathbf{D}^{-1}\left(\frac{\partial L}{\partial \mathbf{V}}\right)^\top \left(\mathbf{I}_n - \mathbf{V}\mathbf{V}^\top\right)\right],\end{aligned} \tag{4.26}$$

with $\mathbf{I}_n$ being the identity of size $n \times n$ and the matrix $\mathbf{K}$ containing the inversed, pair-wise differences of squared singular values with zero diagonal:

$$\mathbf{K}_{i,j} = \begin{cases} \frac{1}{d_j^2 - d_i^2}, & \text{if } i \neq j, \\ 0, & \text{if } i = j. \end{cases} \tag{4.27}$$

It should be noted that, usually, we do not have to compute the full Equation (4.26) but only the subset of terms with non-zero input sensitivities. It can be seen that if $\frac{\partial L}{\partial \mathbf{U}} = 0$, the first term becomes zero, if $\frac{\partial L}{\partial \mathbf{D}} = 0$, the second term becomes zero, and if $\frac{\partial L}{\partial \mathbf{V}} = 0$, the third term becomes zero. In many common computation graphs, the computations following an SVD only depend on $\mathbf{U}$ and, therefore, only $\frac{\partial L}{\partial \mathbf{U}}$ is non-zero, in which case only the first term needs to be computed.

Eigendecomposition Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be a matrix with real entries. In this thesis, we restrict ourself to the special case of positive semi-definite, symmetric matrices to ensure the existence of positive real eigenvalues, thus there exists a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ with $\mathbf{A} = \mathbf{X}^\top \mathbf{X}$. Then, a decomposition

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{U}^\top \iff \mathbf{A}\mathbf{U} = \mathbf{U}\mathbf{D} \tag{4.28}$$

exists, with $\mathbf{U} \in \mathbb{R}^{n \times n}$ containing n orthogonal *eigenvectors* of $\mathbf{A}$ and a diagonal matrix $\mathbf{D}$ containing n non-negative eigenvalues d_i of $\mathbf{A}$ in its diagonal. Similar to

SVD, the ED is usually applied in computation graphs by mapping a matrix $\mathbf{A}$ to its decomposition, the matrices $\mathbf{U}$ and $\mathbf{D}$. Thus the backward computation needs to be of the form $(\frac{\partial L}{\partial \mathbf{U}}, \frac{\partial L}{\partial \mathbf{D}}) \mapsto \frac{\partial L}{\partial \mathbf{A}}$ and can be computed in closed form [Gil08] by

$$\frac{\partial L}{\partial \mathbf{A}} = \mathbf{U}\left[\frac{\partial L}{\partial \mathbf{D}} + \mathbf{K} \odot \left(\mathbf{U}^\top \frac{\partial L}{\partial \mathbf{U}}\right)\right]\mathbf{U}^\top, \tag{4.29}$$

with $\mathbf{K}$ containing the inversed, pair-wise differences of eigenvalues with zero diagonal:

$$\mathbf{K}_{i,j} = \begin{cases} \frac{1}{d_j - d_i}, & \text{if } i \neq j, \\ 0, & \text{if } i = j. \end{cases} \tag{4.30}$$

Relation between SVD and ED Eigenvalues and singular vectors are not the same but they relate in a specific way [HZ03]: The product of SVDs of a matrix $\mathbf{A}$ and its transpose $\mathbf{A}^\top$, $\mathbf{A}^\top\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{U}^\top\mathbf{U}\mathbf{D}\mathbf{V}^\top = \mathbf{V}\mathbf{D}^2\mathbf{V}^\top$, amount to the ED of $\mathbf{A}^\top\mathbf{A}$. Therefore, it can be observed that the eigenvalues of matrix $\mathbf{A}^\top\mathbf{A}$ are the squared singular values of the matrix $\mathbf{A}$ and that the left singular vectors $\mathbf{U}$ of $\mathbf{A}$ are the eigenvectors of $\mathbf{A}^\top\mathbf{A}$. It should be noted that $\mathbf{A}^\top\mathbf{A}$ is symmetric and positive semi-definite, leading to real, non-negative eigenvalues. This relation between SVD and ED might help when designing differentiable algorithms. If we can find a solution to a least squares problem by computing a left singular vector of a matrix $\mathbf{A}$, we can alternatively compute the ED of $\mathbf{A}^\top\mathbf{A}$ instead. Vice versa, the desired solution is often produced by an ED of a covariance matrix $\mathbf{C}$, which can be per definition decomposed into $\mathbf{C} = \mathbf{A}^\top\mathbf{A}$, allowing to compute an SVD of $\mathbf{A}$ instead. Since forward and backward computations of SVD and ED have slightly different properties, it might be advantageous to favor one over the other, depending on the task at hand. For SVD, unstable behaviour can be avoided in some cases (cf. Section 4.2.2). ED on the other hand, might be more efficient to compute, since the matrix to decompose is of fixed size, which is independent from the number of input points to a least squares problem.

Challenges when differentiating SVD and ED When computing the backward algorithms of SVD and ED as described above, challenges arise. The sensitivities $\frac{\partial L}{\partial \mathbf{A}}$ for ED and SVD both face a discontinuity when two ore more singular/eigenvalues are equal. In Equation (4.26) and (4.29), this surfaces as ill-defined entries of the matrix $\mathbf{K}$, since they contain inverse differences $\frac{1}{d_j - d_i}$ or $\frac{1}{d_j^2 - d_i^2}$, respectively. The discontinuity leads to two different problems in practice. First, the gradient becomes very unstable near such discontinuities, even if the exact point of equal singular/eigenvalues can be practically avoided. Second, if the sensitivities with respect to the input matrix are computed for a single output singular/eigenvector, e.g. the one with smallest singular/eigenvalue, the vectors flip if the magnitude order changes. Therefore, the backward signal abruptly reaches different parameters. Both effects lead to unstable training in practice.

4.2.2 Related Work

In general, the closed-form computation of SVD and ED was derived by several works [PL00; Gil08; Tow16]. For SVD, Papadopoulo et al. [PL00] provide a solution for the issue of undefined sensitivities, by finding the minimum norm Jacobian for degenerate cases with $d_j = d_i$. However, this does only solve the issue of not defined sensitivities and does not prevent unstable behavior in border cases. The first attempt to utilize differentiable matrix decompositions in deep architectures was introduced by Ionescu et al. [IVS15], who presented neural network layers for SVD and ED, and apply it to compute the matrix logarithm and normalized cuts. This section will summarize subsequent work of differentiable matrix decompositions in the context of deep learning. It will cover two different fields of application and their individual techniques to make the differentiation work in practice. The first topic is the application to find solutions of least squares problems in a differentiable manner, which occur very often in (3D) computer vision. Secondly, the first steps in the direction of differentiable spectral analysis are tackled, utilizing eigenvectors of Laplacian-like operators as basis for spectral domains, while enabling the learning of the basis generating operators.

Differentiable Least Squares Given a set of linear equations of the form $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{A}$ being an $m \times n$ matrix with $m \geq n$ and $\text{rank}(\mathbf{A}) = n$, thus being potentially over-determined. Then, the vector

$$\hat{\mathbf{x}} = \underset{\mathbf{x}}{\text{argmin}} \, \|\mathbf{Ax} - \mathbf{b}\|^2 \tag{4.31}$$

that minimizes the vector norm error is called a least squares solution to the system of equations [HZ03]. It is well established that such a solution can be found using SVD or ED as:

$$\hat{\mathbf{x}} = \mathbf{V}(\mathbf{U}^\top \mathbf{b} \odot \mathbf{d}^{-1}) \tag{4.32}$$

with $\mathbf{A} = \mathbf{UDV}^\top$ being the SVD of $\mathbf{A}$ and $\mathbf{d}^{-1}$ containing the inverse singular values (cf. Hartley and Zisserman [HZ03] for a proof). An important special case of the above is the homogeneous system of equations $\mathbf{Ax} = \mathbf{0}$ (sizes and rank as above) with the additional constraint of $\|\mathbf{x}\| = 1$. In this case, the solution is obtained even simpler by just taking the right singular vector of $\mathbf{A}$ that corresponds to the smallest singular value. Alternatively, the solution can be obtained by taking the eigenvector of $\mathbf{A}^\top\mathbf{A}$, corresponding to the eigenvalue with smallest magnitude (which always is the smallest eigenvalue because $\mathbf{A}^\top\mathbf{A}$ is positive semi-definite).

There are several works solving (homogeneous) least squares problems in a differentiable algorithm to solve important computer vision tasks [SST+18; YTO+18; RK18; LOM20]. Suwajanakorn et al. [SST+18] perform differentiable Procrustes alignment by finding the optimal rotation between two sets of keypoints as

$$\hat{\mathbf{R}} = \underset{\det(\mathbf{R})=0}{\text{argmin}} \, \|\mathbf{RA} - \mathbf{B}\|_F^2 \tag{4.33}$$

and solve the least squares problem using differentiable SVD. The parameters of the full algorithm are optimized to produce keypoints $\mathbf{A}$ and $\mathbf{B}$ (under further constraints) that lead to a low angle error of the alignment. To fight problems with undefined and flipping SVD sensitivities, they add noise to the keypoints before computing the SVD.

Ranftl and Koltun [RK18] and Yi et al. [YTO+18] find the fundamental or essential matrix between to images by first computing the solution of a weighted, differentiable homogeneous least squares problem (a weighted version of the 8-point algorithm [HZ03]) as

$$\hat{\mathbf{e}} = \underset{\|\mathbf{e}\|=1}{\operatorname{argmin}} \left\| \mathbf{X}^T \operatorname{diag}(\mathbf{w}) \mathbf{X} \mathbf{e} \right\|^2, \tag{4.34}$$

with $\mathbf{X}$ containing preprocessed correspondences. In a second step, they enforce the rank-deficiency constraint of the fundamental/essential matrix by finding an optimal solution to the problem

$$\tilde{\mathbf{E}} = \underset{\det(\mathbf{E})=0}{\operatorname{argmin}} \left\| \mathbf{E} - \hat{\mathbf{E}} \right\|_F^2, \tag{4.35}$$

where $\hat{\mathbf{E}}$ is the matrix form of $\hat{\mathbf{e}}$. This is achieved by applying an additional SVD $\hat{\mathbf{E}} = \mathbf{U} \cdot \operatorname{diag}((d_1, d_2, d_3)^\top) \cdot \mathbf{V}^\top$ and computing $\tilde{\mathbf{E}}$ by setting the smallest singular value to zero: $\hat{\mathbf{E}} = \mathbf{U} \cdot \operatorname{diag}((d_1, d_2, 0)^\top) \cdot \mathbf{V}^\top$. Both works employ different normalization and pre-processing steps before solving the actual problems, which are left out for the sake of a simple overview. In case of Yi et al. [YTO+18], the solution to the first problem is obtained by computing the differentiable ED of $\mathbf{X}^T \operatorname{diag}(\mathbf{w}) \mathbf{X}$. Ranftl and Koltun [RK18], in contrast, use SVD for both steps. In both works, the parameters of the whole algorithm are that of a deep neural network producing the weights $\mathbf{w}$ to optimally weight the given correspondences for robust estimation (c.f IRLS, Section 3.3).

In our work about deep differentiable surface normal estimation [LOM20], which will be the topic of Chapter 5 in this thesis, we go a similar route to compute the normal vectors of point clouds by solving a large number of parallel least squares problems in an iterative, re-weighted fashion. The employed ED algorithm is outlined in the following Section 4.2.3.

Differentiable Spectral Bases A second prominent application of ED in deep learning related research areas is that of Laplacian eigenbases for spectral analysis of functions lying on irregularly structured domains. Given a Laplacian operator (e.g. the Laplacian matrix for a graph, the Laplace-Beltrami Operator (LBO) for manifolds, or approximated variants of the LBO for meshes and point clouds), the eigenfunctions of it yield a spectral basis for the given domain. A classic example for regular domains is the Fourier basis. Applying the same principle to other domains (e.g. graphs or manifolds) we can generalize the concept of Fourier analysis to those domains, including applying filters by element-wise multiplication in the frequency domain.

Formally in a discrete setting, given a Laplacian matrix $\mathbf{L}$, which can be the graph Laplacian or an approximated LBO on a mesh or point cloud graph, we can find the functions ψ, which adhere to

$$\mathbf{L}\psi = \lambda\psi, \tag{4.36}$$

as eigenvectors of $\mathbf{L}$ by eigendecomposition. Then, the set of eigenfunctions $\mathbf{\Psi} = (\psi_i)_{1\leq i\leq n}$ forms an orthonormal basis of the function space on the original graph, mesh or point cloud domain. There are several applications of this technique in deep learning. It motivated the spectral construction of the first graph neural networks [BZS+14] and networks on manifolds and graphs [BBL+17], as outlined in Section 4.1.3. Further, the functional maps framework [OBS+12; LRR+17], was heavily adopted in deep learning, solving correspondence tasks between manifolds.

The Laplacian operator describes how a quantity diffuses on a specific domain over time. We can multiply the Laplacian with a function on the domain to simulate this diffusion. In the spatial domain, computing this diffusion for a specific time t would require an iterative process. The Laplacian eigenbasis contains stable modes of this diffusion, i.e. functions, which do only vary in magnitude when diffused further (since $\mathbf{L}\psi = \lambda\psi$ per definition). Using the basis, the diffusion process for a specific time t can be approximated in closed form by mapping a function to its spectral representation, exponential scaling based on t in the spectral domain, and mapping it back to the spatial domain. The resulting diffusion process has differing degree of locality, depending on the chosen spectral filter, which was recently utilized by Sharp et al. in their work about diffusing GNNs on meshes and point clouds [SAC+20]. Instead of transferring information only locally, as GNNs usually do, integrating such a procedure allows for domain dependent global information transfer, which may supplement the standard properties of GNNs in certain scenarios.

Alternative Differentiation Schemes Even if the analytical gradients of ED exists and are computable in closed form, they have negative properties in some cases as outlined above. Therefore, a current trend in research is to find alternative ways to backpropagate through ED. One example is the work by Wei et al. [WDH+19], which proposes an alternative backward computation based on the backward function of power iterations, which is an iterative method for ED that comes with its own derivative. Through assuming initialization with the actual eigenvector computed in the forward step, they change the iterative power iteration backward algorithm to a non-iterative computation, improving efficiency and stability.

Another attractive differentiation scheme for ED is the adjoint method, as outlined by Xie et al. [XLW20]. As Equation (4.29) shows, the standard analytic gradient of ED depends on the full set of eigenvectors $\mathbf{U}$ and eigenvalues $\mathbf{D}$. In cases where we only need a small subset of eigenpairs for the subsequent part, we need to compute all pairs anyway, just for the backward algorithm. The adjoint scheme formulates a different version of analytic gradients, which is especially useful when only a small number of eigenpairs is computed in the forward step and only

the sensitivities to those eigenpairs are non-zero. Formally, in case of a symmetric matrix $\mathbf{A}$ with k eigenpairs $(\mathbf{v}_i, d_i)_{1\leq i\leq k}$ having non-zero sensitivities $(\frac{\partial L}{\partial \mathbf{v}_i}, \frac{\partial L}{\partial d_i})$, the sensitivity $\frac{\partial L}{\partial \mathbf{A}}$ can alternatively expressed as

$$\frac{\partial L}{\partial \mathbf{A}} = \sum_{i=1}^{k} \left(\frac{\partial L}{\partial d_i} \mathbf{v_i} - \boldsymbol{\xi}_i \right) \mathbf{v}^\top, \tag{4.37}$$

where the $\boldsymbol{\xi}_i$ are the solutions to the linear systems

$$(\mathbf{A} - d_i \mathbf{I})\boldsymbol{\xi} = (1 - \mathbf{v}\mathbf{v}^\top) \frac{\partial L}{\partial \mathbf{v}_i} \tag{4.38}$$

which can be solved efficiently using iterative procedures leveraging Krylov subspaces [XLW20]. It is clear that this approach is only feasible if k is small. For the most common application with $k = 1$, the approach becomes very efficient when combined with e.g. power iterations to only compute the dominant eigenpair in the forward step.

Another important work was presented recently by Dang et al. [DYH+20] and introduces a way to completely circumvent the problems with backpropagating through ED and SVD under certain conditions. The idea follows the concept of surrogate gradients (cf. Section 2.2), which aims to find alternative ways to produce learning signals that lead to the same minimum but do not require the actual backward computation. The method is restricted to cases where the output of the algorithm is the zero magnitude eigenvector $\mathbf{e}_1$ of a matrix $\mathbf{A}$, for which a direct ground truth $\tilde{\mathbf{e}}$ is given. Since the implication

$$\mathbf{A}\mathbf{e}_1 = 0 \implies \mathbf{e}_1 \mathbf{A}^\top \mathbf{A} \mathbf{e}_1 = 0 \tag{4.39}$$

holds for this eigenvector, we can aim to find the the matrix $\mathbf{A}$ that minimizes the surrogate loss

$$L(\theta) = \tilde{\mathbf{e}} \mathbf{A}^\top \mathbf{A} \tilde{\mathbf{e}}, \tag{4.40}$$

for the ground truth vector $\tilde{\mathbf{e}}$, i.e. aiming to find a matrix that has $\tilde{\mathbf{e}}$ as its zero magnitude eigenvector. Since a trivial minimum of this loss is $\mathbf{A} = 0$, the authors propose to additionally maximize the norm of projections of data vectors that are orthogonal to $\tilde{\mathbf{e}}$, by adding a loss term:

$$L(\theta) = \tilde{\mathbf{e}} \mathbf{A}^\top \mathbf{A} \tilde{\mathbf{e}} + \alpha \exp(-\beta tr(\bar{\mathbf{A}}^\top \bar{\mathbf{A}})), \tag{4.41}$$

with $\bar{\mathbf{A}} = \mathbf{A}(\mathbf{I} - \tilde{\mathbf{e}}\tilde{\mathbf{e}}^\top)$. For a more detailed derivation the reader is referred to the original work [DYH+20].

4.2.3 Differentiable Solvers in this Thesis

This section will describe the differentiable least squares solver applied in this thesis. It mainly covers the parallel differentiable least squares solver for surface normal estimation, which is the main topic of Chapter 5. Also, the application in 3D capsule networks, as further described in Chapter 6, is introduced.

Parallel Least Squares Solver

The problems tackled in Chapter 5 of this thesis include a set of weighted least-squares problems for plane fitting to estimate surface normals $\mathbf{n}$:

$$\mathbf{n}_i^* = \underset{\mathbf{n}:|\mathbf{n}|=1}{\operatorname{argmin}} ||\mathbf{P}(i)\mathbf{n}||^2 = \underset{\mathbf{n}:|\mathbf{n}|=1}{\operatorname{argmin}} \sum_{j \in \mathcal{N}(i)} ||\mathbf{p}_j \cdot w_{i,j} \cdot \mathbf{n}||^2, \tag{4.42}$$

where $\mathbf{P}(i)$ contains the weighted points $\mathbf{p}_j \cdot w_{i,j}$ in the neighborhood of size k_i around point i. Each problem lies on a node of a graph $G = (\mathcal{V}, \mathcal{E})$, where the graph edges define neighborhood relations. The goal is to provide forward and backward algorithms that work in parallel over the nodes of the graph, in order to integrate the algorithms into a MP-GNN formulation.

According to Section 4.2.1, we have two straight-forward options to find the solution of the least squares problems: SVD and ED. For the former, we would apply the SVD to the $k_i \times 3$ matrix $\mathbf{P}(i)$ directly, which becomes difficult to parallelize over differently sized neighborhoods in the graph, since we need to solve n SVDs and each one has an input matrix of different size. Instead, we chose to perform ED on the weighted covariance matrices $\mathbf{C}_i = \mathbf{P}(i)^\top \text{diag}(\mathbf{w}_i)\mathbf{P}(i)$, with weights $\mathbf{w}_i \in \mathbb{R}^{k_i}$, which is symmetric, positive-semidefinite and always of size 3×3. ED on symmetric, positive-semidefinite, 3×3 matrices is known to have a closed form solution that does not require an iterative approach [GV89]. Also, the operations can be easily parallelized over all graph nodes, since the structure of input, output and required operations is the same for all n solvers.

The full forward and backward algorithms, from a point cloud graph with weighted edges to normal vectors and vice-versa for sensitivities, are shown in Algorithm 5 and Algorithm 6, respectively. The forward algorithm starts with computing the weighted covariance matrix $\mathbf{C}_i = \mathbf{P}(i)^\top \text{diag}(\mathbf{w}_i)\mathbf{P}(i)$ for each node i, using an MP-GNN formulation. We can rewrite $\mathbf{C}$ as

$$\mathbf{C}_i = \mathbf{P}(i)^\top \text{diag}(\mathbf{w}_i)\mathbf{P}(i) = \sum_{j \in \mathcal{N}(i)} \mathbf{P}(i)_j^\top w_{i,j} \mathbf{P}(i)_j, \tag{4.43}$$

where we first compute the contribution to the $\mathbf{C}_i$'s for each edge in the graph, before adding them up over the neighborhoods of nodes. In practice, we can achieve parallelization over edges when computing the individual summands. Then, we use a scatter-add operation to bring the data to node space, over which we can parallelize for computing the individual EDs. What follows in node parallel space is the well-known closed form computation of ED for symmetric 3×3 matrices [GV89]. First, the eigenvalues of the input $\mathbf{C}$ are computed by finding the roots of the characteristic cubic polynomial $\det(\beta\mathbf{I} - \mathbf{B})$ for a surrogate matrix $\mathbf{B}$ in closed form. The eigenvalues of $\mathbf{C}$ can be recovered later from those of $\mathbf{B}$ by a linear transformation. Then, given the computed eigenvalues $\mathbf{d}$, we compute the eigenvectors of $\mathbf{C}$ by choosing the largest magnitude pair-wise cross product of $\mathbf{C} - d_i\mathbf{I}$ for each eigenvector d_i. For an in-depth description and analysis of the method, the reader is referred to Golub and Van Loan [GV89].

Algorithm 5 Parallel, weighted least squares plane fitting: forward algorithm.

Input:
N: Number of nodes
E: Number of edges
$\mathbf{P} \in \mathbb{R}^{E\times 3}$: Relative point coordinates for each edge
$\mathbf{w} \in \mathbb{R}^{E}$: Weights for each edge
Output:
$\mathbf{N} \in \mathbb{R}^{N\times 3}$: Normal vectors for each node

---

Parallelize over $e \in \{1, \dots, E\}$:
  $\mathbf{C}_E[e,:,:] \leftarrow \mathbf{P}[e,:]^\intercal \cdot \mathbf{w}[e] \cdot \mathbf{P}[e,:]$
  $\mathbf{C}[n_{\text{sink}}(e),:,:] \leftarrow \text{ScatterAdd}(\mathbf{C}_E[e,:,:])$ to node space
Parallelize over $n \in \{1, \dots, N\}$:
  $q \leftarrow tr(\mathbf{C}[n,:,:])/3$
  $p \leftarrow \sqrt{tr((\mathbf{C}[n,:,:] - qI)^2)/6}$
  if $p = 0$ **then**
    $\mathbf{D}[n,:] \leftarrow (\mathbf{C}[n,0,0] || \mathbf{C}[n,1,1] || \mathbf{C}[n,2,2])$
  else
    $\mathbf{B} \leftarrow (\mathbf{C}[n,:,:] - q\mathbf{I})/p$
    $r \leftarrow \det(\mathbf{B})/2$
    if $r \le -1$ **then**
      $\phi \leftarrow \pi/3$
    else if $r \ge 1$ **then**
      $\phi \leftarrow 0$
    else
      $\phi \leftarrow \text{acos}(r)/3$
    end if
    $\mathbf{D}[n, 0:2] \leftarrow (q + 2 \cdot p \cdot \cos(\phi), q + 2 \cdot p \cdot \cos(\phi + 2\pi/3))$
    $\mathbf{D}[n,2] \leftarrow 3q - \mathbf{D}[n,0] - \mathbf{D}[n,1]$
  end if
  for each $x \in \{1,2,3\}$ **do**
    $\mathbf{B}[n,:,:] \leftarrow \mathbf{C}[n,:,:] - \mathbf{D}[n,x]\mathbf{I}$
    $\mathbf{r}_1 \leftarrow \mathbf{B}[n,0,:] \times \mathbf{B}[n,1,:]$
    $\mathbf{r}_2 \leftarrow \mathbf{B}[n,0,:] \times \mathbf{B}[n,2,:]$
    $\mathbf{r}_3 \leftarrow \mathbf{B}[n,1,:] \times \mathbf{B}[n,2,:]$
    $i \leftarrow \text{argmax}\{\|\mathbf{r}_1\|_2, \|\mathbf{r}_2\|_2, \|\mathbf{r}_3\|_2\}$
    $\mathbf{U}[n,x,:] \leftarrow \mathbf{r}_i / \|\mathbf{r}_i\|_2$
  end for
  $i \leftarrow \text{argmin}\{\|\mathbf{D}[n,0], \mathbf{D}[n,1], \mathbf{D}[n,2]\|\}$
  $\mathbf{N}[n,:] \leftarrow \mathbf{U}[n,i,:]$
Return $\mathbf{R}$

Algorithm 6 Parallel, weighted least squares plane fitting: backward algorithm.

Input:
N: Number of nodes
E: Number of edges
$\tilde{\mathbf{N}} \in \mathbb{R}^{N\times 3}$: Partial derivatives w.r.t. $\mathbf{N}$
Kept from forward:
$\mathbf{U} \in \mathbb{R}^{N\times 3\times 3}$: Eigenvectors for each node.
$\mathbf{D} \in \mathbb{R}^{N\times 3}$: Eigenvalues for each node
Output:
$\tilde{\mathbf{w}} \in \mathbb{R}^{E}$: Partial derivatives w.r.t. $\mathbf{w}$

Parallelize over $n \in \{1, ..., N\}$:
$i \leftarrow \text{argmin}\{\|\mathbf{D}[n,0], \mathbf{D}[n,1], \mathbf{D}[n,2]\|\}$
$\tilde{\mathbf{U}}[n,:,:] \leftarrow \mathbf{0}$
$\tilde{\mathbf{U}}[n,i,:] \leftarrow \tilde{\mathbf{N}}[n,:]$
$\mathbf{K}[i,j] \leftarrow (\mathbf{D}[n,j] - \mathbf{D}[n,i])^{-1}$, $\forall i,j \in \{1,2,3\}$ with $i \neq j$, 0 else.
$\tilde{\mathbf{C}}[n,:,:] \leftarrow \mathbf{U}[n,:,:]\left[\mathbf{K} \odot \left(\mathbf{U}[n,:,:]^\top \tilde{\mathbf{U}}[n,:,:]\right)\right]\mathbf{U}[n,:,:]^\top$
Parallelize over $e \in \{1, ..., E\}$:
$\tilde{\mathbf{C}}_E[e,:,:] \leftarrow \text{Gather}(\tilde{\mathbf{C}}[n_{\text{sink}}(e),:,:])$
$\tilde{\mathbf{w}}[e] \leftarrow \sum_{i,j}(\mathbf{P}[e,:]^\top \mathbf{P}[e,:] \odot \tilde{\mathbf{C}}_E[e,:,:])[i,j]$
Return $\tilde{\mathbf{w}}$

The backward computation in Algorithm 6 utilizes the analytical gradient of ED given in Equation (4.29) in node parallel space, before gathering the result into edge parallel space. There, we compute the sensitivity with respect to the weights for each edge.

Least Squares $SO(3)$ Averaging

The second application in this thesis that utilizes differentiable matrix decompositions are 3D quaternion capsule networks as presented in Chapter 6. Here, the goal is to find the weighted geometric mean of a set of $SO(3)$ elements. Specifically, given a set of $SO(3)$ elements $\mathcal{M} = \{\mathbf{g}_1, ...\mathbf{g}_n\}$ and a geodesic distance measure $d_{\text{geo}} : SO(3) \times SO(3) \rightarrow \mathbb{R}$ on $SO(3)$, we aim to find the $SO(3)$ element that minimizes the distances to elements of $\mathbf{M}$:

$$\hat{\mathbf{g}} = \underset{\mathbf{h}\in SO(3)}{\text{argmin}} \sum_{\mathbf{g}\in\mathcal{M}} d_{\text{geo}}(\mathbf{h}, \mathbf{g}). \tag{4.44}$$

In practice, unit quaternions are used to represent the group elements because they enable us to find the mean efficiently and have additional advantageous properties (cf. Section 3.2). A differentiable IRLS algorithm is used that iteratively encloses on the desired mean.

Given unit quaternions $\mathcal{Q} = \{\mathbf{q}_1, ...\mathbf{q}_n\}$ with $\mathbf{q}_i \in \mathbb{S}^3$, representing the group elements, weights $\mathbf{w} \in \mathbb{R}^n$, and the quaternion distance $d_{\text{quat}}(\mathbf{q}_1, \mathbf{q}_2) = 2\cos^{-1}(\mathbf{q}_1^\top \mathbf{q}_2)$, which equals the angle between the two represented $SO(3)$ rotations, one step of the iterative algorithm amounts to finding the solution of [MCC+07]

$$\hat{\mathbf{q}} = \underset{\mathbf{q}\in\mathbb{S}^3}{\operatorname{argmax}}\, \mathbf{q}^\top \mathbf{M} \mathbf{q}, \tag{4.45}$$

with $\mathbf{M}$ being the weighted covariance matrix of the quaternions

$$\mathbf{M} = \sum_{i=1}^{n} w_i \mathbf{q}_i \mathbf{q}_i^\top. \tag{4.46}$$

After each step, new weights are chosen as a function of the distance $d_{\text{quat}}(\hat{\mathbf{q}}, \mathbf{q}_i)$ between the input quaternions $\mathbf{q}_i$ and the intermediate solution $\hat{\mathbf{q}}$ (cf. Section 3.3). The solution to the problem in Equation (6.35) is given by the largest magnitude eigenvector of the real, symmetric, 4×4 matrix $\mathbf{M}$ [MCC+07; ZBL+20]. In the quaternion capsule networks, the forward algorithm uses the iterative Lanczos solver implemented in Pytorch [PGM+19]. The backward algorithm implements the closed form, analytic gradient given in Equation (4.26). For the whole method, the reader is referred to Chapter 6.

4.3 Implicit Neural Functions and Representations

This section will describe Implicit Neural Functions (INFs), a simple but effective concept to learn functions and representations given data points as implicit function descriptions, which gained a lot of interest through successful applications in the recent years. The concepts described in this section are applied in the joint work of Deep Local Shapes (DeepLS) and in surface normal estimation, as implicit kernel function for re-weighting. This section will apply INFs to summarize a deep and non-linear variant of Compressed Sensing (CS) in Section 4.3.1 and gives an overview about applications in recent literature in Section 4.3.2. Then, the two applications in this thesis are described, an INF as parameterized kernel function in Section 4.3.3 and the application for DeepLS in Section 4.3.4.

Concept In 2- or 3-dimensional domains, we usually implement deep functions between these domains as CNN, mapping from and to dense data given on grids like images or voxel grids, or as GNN, mapping from and to irregular, sparse data points, such as point clouds or meshes. In contrast, an INF is defined continuously over the domain $\mathbb{R}^d$.

We define a family of INFs as functions $f_\psi(\mathbf{x}, \theta) : \mathbb{R}^d \times \mathbb{R}^h \to \mathbb{R}^n$, where f_ψ is realized as an MLP with parameters ψ. Here, MLPs are a natural choice to implement those functions, due to their ability to approximate any function [HSW89]. The additional parameter vector θ decides on a specific function from this family, which

maps a coordinate vector $\mathbf{x} \in \mathbb{R}^d$ to a value $\mathbf{y} \in \mathbb{R}^n$. In practice, the parameter vector θ and the coordinate vector $\mathbf{x}$ are concatenated and fed as single vector to the MLP input neurons. The two sets of parameters allow for a two level training procedure. First, by optimizing ψ we can find a space of functions, which f_ψ is able to compute. Second, by optimizing θ we can find individual functions within this space that best approximate functions implicitly defined by data points.

There are different ways to realize the given framework. An INF can be integrated in a differentiable algorithm, which is trained in an end-to-end fashion, with parameters ψ being directly optimized and parameters θ being the output of a preceding part of the algorithm, e.g. another neural network. The kernel function for IRLS re-weighting described in Section 4.3.3 and Chapter 5 is an example for this type of application. A more sophisticated way to utilize INFs is to use them for a differentiable, non-linear variant of Compressed Sensing (CS), which is described in the following section.

4.3.1 Deep, non-linear Compressed Sensing

Traditionally, Compressed Sensing (CS) is a method to measure and reconstruct a function from fewer samples than required by the sampling theorem. We aim to find the signal that best explains a set of samples, given sparsity constraints. Those sparsity constraints are enforced by compressing the signal in a low dimensional space using a sparse set of linear independent vectors, which we call the CS compression matrix. Therefore, we first have to find vectors that span a space in which unique identification of a signal purely based on the given observation is possible.

DNNs are known for their ability to heavily compress complex signals in low-dimensional representations. In contrast to the CS compression matrix, they transform the signal using non-linear functions. Even if we loose most of the theoretical results from the field of CS (since they rely on the linearity of the compression matrix), we can generalize the concept to non-linear domains using INFs. We solve both steps, finding the optimal compression function and finding the best representation for signal reconstruction by reverse mode accumulation. The compression space is learned from a data set of functions, each sampled with high frequency. Then, future signals following the same characteristics can be reconstructed from very sparse sets of samples. The concept was originally mentioned for shallow networks in early neural network works [TM95] and was applied to signal error detection [RM98] and process monitoring [BH12]. Later, a similar concept was used for deep matrix completion [FC18] and Generative Adversarial Networks (GANs) [BJL+18], before it was brought to the field of 3D vision with deep networks by Groueix et al. [GFK+18], Mescheder et al. [MON+19], Chen et al. [CZ19] and Park et al. [PFS+19] for representing surfaces, occupancy, indicator functions and Signed Distance Functions (SDFs), respectively. In the following, it is described in general as deep, non-linear CS for arbitrary functions.

Let $g_1, ..., g_m : \mathbb{R}^d \to \mathbb{R}^n$ be M functions, which are described by sets of implicit function samples $\mathcal{S}_i = \{(\mathbf{x}_1^i, \mathbf{y}_1^i), ..., (\mathbf{x}_{K_i}^i, \mathbf{y}_{K_i}^i)\}$, so that each function g_i obeys to $g_i(\mathbf{x}^i) = \mathbf{y}^i$ for all $(\mathbf{x}^i, \mathbf{y}^i) \in \mathcal{S}_i$. We denote the set of all those sample sets as $\mathcal{D} = \{\mathcal{S}_1, ..., \mathcal{S}_n\}$, which will be the training set for our deep non-linear CS model. Given an INF $f_\psi(\mathbf{x}, \theta) : \mathbb{R}^d \times \mathbb{R}^k \to \mathbb{R}^n$, we jointly obtain a compression space and representations for training functions by finding the optimal parameters $\hat{\psi}$ and $\{\hat{\theta}_i\}_{i=1}^M$ as

$$(\hat{\psi}, \{\hat{\theta}_i\}) = \underset{\psi, \{\theta_i\}}{\operatorname{argmin}} \sum_{m=1}^{M} \left(\sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{S}_m} \left\| f_\psi(\mathbf{x}, \theta_m) - \mathbf{y} \right\| + \lambda \left\| \theta_m \right\|_2^2 \right), \tag{4.47}$$

where $\lambda \left\| \theta_m \right\|_2^2$ is an L_2 regularization term on the function representations θ_i and $\|\cdot\|$ can be chosen depending on the application. Given the trained INF f_ψ, we can then encode a new observation $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), ..., (\mathbf{x}_K, \mathbf{y}_K)\}$ by finding the representation θ that optimally fits $\mathcal{S}$, using fixed parameters ψ:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{S}} \left\| f_\psi(\mathbf{x}, \theta) - \mathbf{y} \right\| + \lambda \left\| \theta \right\|_2^2 . \tag{4.48}$$

The fixed network f_ψ was trained to reconstruct functions from a specific subspace. Thus, we optimize for the most fitting function representation lying within this subspace. Having a trained network f_ψ and a representation θ, the function can be sampled at arbitrary $\mathbf{x}$ in continuous space by evaluating $f_\psi(\mathbf{x}, \theta)$. If the whole function is needed, it can be efficiently evaluated densely by utilizing batch-wise matrix multiplication implementations on the GPU, which are highly optimized due to their use in MLP training.

4.3.2 Related Work

This section will cover the existing work utilizing INFs in computer vision, which can be mainly found in the field of 3D reconstruction and auto-encoding of shapes. The idea appeared first in the work of Groueix et al. [GFK+18], describing a method called AtlasNet, which parameterizes local 2-manifolds using INFs by mapping chart coordinates $\mathbf{x} \in]0,1[^2$ and surface representation vector $\theta \in \mathbb{R}^h$ to surface patches in $\mathbb{R}^3$ using an MLP $f :]0,1[^2 \times \mathbb{R}^h \to \mathbb{R}^3$. The main advantage of describing surfaces as a set of charts, i.e. atlases, is that the mapping can also be used to parameterize textures and other surface properties along with the geometry. However, in practice, the approach faces challenges when it comes to stitching the 2-manifolds together, as they appear discontinuous at borders.

The next iteration of representing shapes with INFs was introduced simultaneously by Park et al. [PFS+19], Chen et al. [CZ19], and Mescheder et al. [MON+19]. Park et al. [PFS+19] represent the signed distance (negative values for points inside and positive values for points outside of the object) to the surface of an object as INF $f_\psi(\mathbf{x}, \theta) = s$ and define the surface as the zero level set of it:

$$\mathcal{S} = \{x \in \mathbb{R}^3 \mid f_\psi(\mathbf{x}, \theta) = 0\}, \tag{4.49}$$

where the network is trained using densely sampled Signed Distance Functions (SDFs) of different objects. Similarly, Chen et al. [CZ19] and Mescheder et al. [MON+19] use INFs to represent indicator functions for occupancy, with value 0 for points inside the object and value 1 for points outside.

Several works build upon the idea of representing SDFs with INFs. Genova et al. presented Structured Implicit Functions [GCV+19], showing that objects can be composed as a sum of INFs with limited support. In joint work, we presented Deep Local Shapes (DeepLS) [CLI+20], creating a deep analogue of the widely used SDF fusion approaches for 3D reconstruction, allowing the INFs to be applied to real scans of large real-world scenes. DeepLS is the topic of the following Section 4.3.4. Several works combined INFs with differentiable renderers, showing that they can be trained using image-based loss functions [SZW19; NMO+20; YKM+20]. Additionally, they show that INFs can also regress color [SZW19], can infer depth [NMO+20], and can refine the camera pose [YKM+20]. Chibane et al. [CAP20] presented Implicit Feature Networks, a hierarchical extension to the single per object representation used in previous work. Instead of a single representation θ, they extract a set of representations $F_1(\mathbf{x}), ..., F_n(\mathbf{x})$ for a coordinate $\mathbf{x}$ from a feature hierarchy and predict occupancy through the INF $f(F_1(\mathbf{x}), ..., F_n(\mathbf{x}))$. Chibane et al. [CMP20] present Unsigned Distance Function Networks, allowing to model surfaces that are not closed, together with a rendering technique based on sphere tracing. CvxNet, as proposed by Deng et al. [DGY+20] uses a differentiable method to decompose objects into convex parts, before regressing those parts using a network predicting hyperplanes. The more interpretable representation allows for direct transformation to polygon meshes. Similarly, Chen et al. [CWH+20] propose BSPNet, which organizes the convex parts in a binary space partitioning tree. Sitzmann et al. [SMB+20] present Siren, an INF with periodic activation functions which enables to learn functions from implicit second and third order information. Through this formulation, they are able to find solutions to partial differential equations by neural network training.

The appearance of INFs sparked the field of Neural Volume Rendering, which dominated 3D vision and rendering in the last two years. The field was initiated by the works of Lombardi et al. [LSS+19] about Neural Volumes, and Mildenhall et al. [MST+20], presenting Neural Radiance Fields (NeRF). Instead of just regressing an SDF or occupancy, they directly use an INF to regress color values and opacity for each point in a volume. In case of NeRF, the view angle is additionally encoded in the domain of the function, leading to a view-dependent representation. The scene is rendered by a ray-casting procedure, producing the color value by integration over a ray through the INF domain. Several optimizations and extensions to NeRF have been proposed. These include Neural Sparse Voxel Fields [LGZ+20], which organize the scene in an octree containing representations, increasing rendering efficiency, and Decomposed Radiance Fields (DeRF) [RJY+20], which divide the scene into soft Voronoi cells, each having their own representation to increase detail and efficiency. Zhang et al. [ZRS+20] improve on NeRF with their NeRF++ by modeling the background individually and analyze how NeRF handles the

shape-radiance ambiguity. Finally, Lindell et al. present AutoInt [LMW21], which introduces a very general new technique to compute integrals over one-dimensional subspaces of INFs. Specifically, by applying the chain-rule to an MLP $f_\theta : \mathbb{R}^d \to \mathbb{R}^3$, they obtain a new network $f'_\theta : \mathbb{R}^d \to \mathbb{R}^3$, which computes the derivative of f and has the same parameters. Then, f' can be trained as in INF and f can be evaluated, using the shared parameters, to obtain integrals over f' by evaluating the anti-derivative with only two network evaluations. Several additional works improve on certain aspects of NeRF and Neural Volumes, e.g. by introducing local reflectance models [SDZ+21; BXS+20], by conditioning on shape representations [SLN+20], composing whole scenes of object level representations [GFW+20; OMT+21], and extensions to handle dynamic scenes [PSB+20; LNS+21]. Since the field of research is very young, no exhaustive review literature exists. However, initial attempts to summarize the literature have been made in a non peer-reviewed manner [DL21].

4.3.3 Parameterized Kernel Functions

A straight-forward application of INFs in this thesis is their appearance as kernel function in DISNE, which is the main topic of Chapter 5. The method requires a differentiable function, from which weights can be sampled in a continuous domain and which can be conditioned on a feature vector θ. INFs are a natural fit for these requirements. Specifically, in the DISNE method, the weights for point pairs $(\mathbf{p}_i, \mathbf{p}_j)$ in a neighborhood around point $\mathbf{p}_i$ are obtained as

$$w_{i,j} = f_\psi(\mathbf{R}_i(\mathbf{p}_j - \mathbf{p}_i) \| \theta_i), \tag{4.50}$$

with f_ψ being an INF realized as an MLP with parameters ψ and θ_i is a feature vector for the neighborhood around point $\mathbf{p}_i$ obtained from a preceding graph neural network. Additionally, the kernel function can be rotated by inversely rotating the input coordinates $(\mathbf{p}_j - \mathbf{p}_i)$ using a rotation matrix $\mathbf{R}_i$. In the DISNE application, this rotation is obtained by applying a Local Spatial Graph Transformer (LSGT) (cf. Section 4.1.5). The whole algorithm, including the INF for weight prediction, the LSGT for inferring local rotations and the GNN to obtain kernel parameterization, is trained end-to-end to achieve the task of surface normal estimation. For a detailed description and evaluation of the method, the reader is referred to Chapter 5.

4.3.4 Deep Local Shapes

This section will discuss DeepLS [CLI+20], joint work to extend DeepSDF [PFS+19] to represent larger scenes based on real scanned data. We briefly summarize DeepLS, its application, and its results.

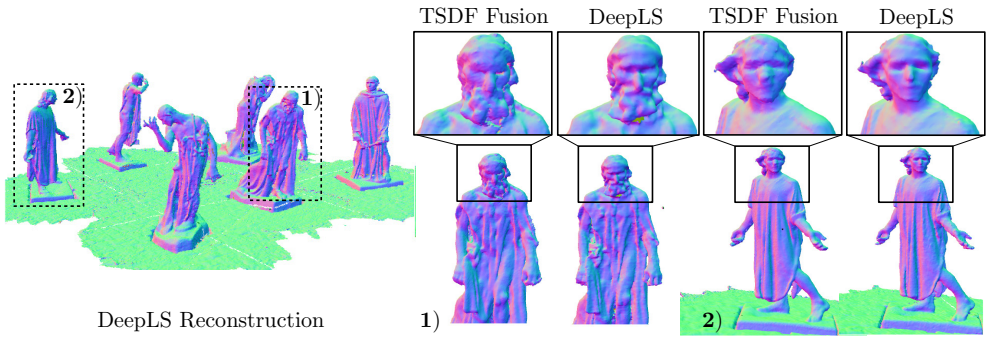
Representing Local Shapes DeepSDF [PFS+19] uses the two step training formulation described in Section 4.3.1 to first jointly learn a space of SDFs and representations for a specific object class, before being able to infer representations

for new (partial) observations of objects from the same class. The main advantage of this method is the strong integration of prior class information, e.g. restricting the function space to model for example the SDFs of *airplane*-like objects. In consequence, the model is able to reconstruct objects from very sparse observations, given only a few sample points from one side of the object. In practice, however, the method is hindered by the vast amount of required training data. For each object class that should be represented, we need to have a wide variety of training examples, making representing whole, real world scenes nearly impossible. In contrast, current scanning technology allows for much denser observation than what DeepSDF actually requires. Therefore, DeepLS [CLI+20] moves a few steps back from the full, global deep learning solution to apply the same technique only on a local level, aiming to keep some of the prior incorporating capabilities while being applicable in practice on real world reconstruction tasks.

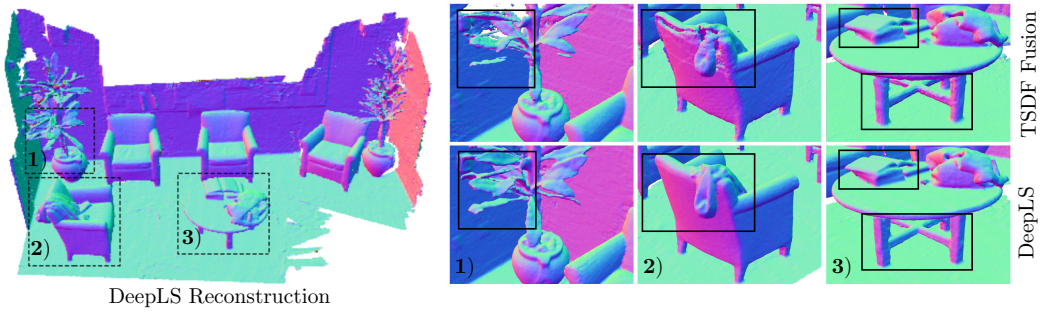
With DeepLS [CLI+20], a scene is divided into a voxel grid of variable cell size, where each cell C_i stores an implicit representation θ_i for the SDF function it contains. Given indicator functions $\mathbb{1}_{C_i} : \mathbb{R}^3 \to \{0, 1\}$ for each cell C_i with $\mathbb{1}_{C_i}(\mathbf{x}) = 1$ for points $\mathbf{x}$ within C_i and 0 otherwise, the surface for the whole scene is modeled as a joint zero level set over all voxel cells:

$$\mathcal{S} = \{\mathbf{x} \in \mathbb{R}^3 \mid \sum_i \mathbb{1}_{C_i}(\mathbf{x}) f_\psi(T_i(\mathbf{x}), \theta_i) = 0\}, \tag{4.51}$$

with $T_i(\mathbf{x}) = \mathbf{x} - \mathbf{x}_i$ transforming $\mathbf{x}$ into the local coordinate space of cell C_i. It is important to note that the INF f_ψ used to decode representations θ_i into SDF values is shared over all cells, thus utilizing the same function space for reconstruction. DeepLS is using the training and inference formulations given in Equation (4.47) and Equation (4.48), respectively. For rendering, we can densely sample the volume and apply Marching Cubes [LC87] to obtain the iso-surface. An important difference to DeepSDF is the required training data. Instead requiring whole objects or scenes, the DeepLS INF can be trained using only local synthetic shape parts. For the experiments it was trained on synthetically generated primitives and parts of the ShapeNet dataset [CFG+15], which are synthetic objects as well. Thus, the learned function space consists of local surfaces in different forms and orientations, which can be utilized to complete and reconstruct scanned data on a local level. The trained model can still be applied to reconstruct scenes from real scans by applying techniques from traditional SDF fusion. Given depth maps of a scene, we generate two SDF value pairs $(\mathbf{x}, s)$ for each depth point $\mathbf{y}$ in 3D scene space by moving the depth point along the surface normal $\mathbf{n}$ at point $\mathbf{y}$ in positive $(\mathbf{x}_1, s_1) = (\mathbf{y} + d \cdot \mathbf{n}, \|d \cdot \mathbf{n}\|_2)$ and negative direction $(\mathbf{x}_2, s_2) = (\mathbf{y} - d \cdot \mathbf{n}, -\|d \cdot \mathbf{n}\|_2)$ for a small value d. Additionally, we generate free space samples along the observation rays to ensure to not generate floating surfaces. For rendering, Marching Cubes is only applied in a narrow band around depth observation points.



(a) Burghers of Calais



(b) Lounge

Figure 4.5: Qualitative comparison between reconstruction results of DeepLS and TSDF Fusion [CL96] on two scenes from the 3D Scene Dataset [ZK13] (from original publication [CLI+20]). It can be seen that the local prior information incorporated by DeepLS is able to fill spots of missing data. Local surfaces need to be drawn from the INF function spaces, which have been trained on synthetic primitive objects.

Results and Discussion Along with general qualitative and quantitative evaluation of DeepLS, for which the reader is referred to the original publication [CLI+20], we performed experiments to evaluate the trade-offs between usage of prior information and practicality. The trade-offs can be controlled by varying the size of local shape cells. For larger cells, our INFs learn to encode more semantically rich prior information and are able to perform better in local completion of geometry. However, we could show that these advantages come at the cost of less accurate reconstruction and slower inference of representations. Also, since more complex objects need to be represented, we need more sophisticated training data. By choosing the correct cell size, is is possible to utilize the prior information from synthetic local surface training examples to fill unobserved areas and compensate for errors and noise in the depth data to a certain degree. This can be observed in the comparison in Figure 4.5, which shows that DeepLS succeeds in reconstructing fine-grained objects in real scene scans. In addition, we found that inference and rendering is much faster than for DeepSDFs object level representations. In total, DeepLS bridges the

gap between traditional SDF fusion and implicit representations, allowing to utilize data-driven parameterization for practical reconstruction of scanned 3D scenes.

Chapter 5

Differentiable Iterative Surface Normal Estimation

This chapter will introduce and analyze Differentiable Iterative Surface Normal Estimation (DISNE), a differentiable algorithm for surface normal estimation on unstructured point clouds, which was originally published in 2020 [LOM20]. The task of surface normal estimation is an important research topic that has been studied for several years, with early publications dating far back to early computer vision research [HDD+92], since surface normal vectors are heavily used as local surface descriptors in several computer vision algorithms, such as surface reconstruction [KBH06], registration [PCS15] and segmentation [GMR17]. While there is a large amount of fixed function algorithms available, data-driven methods for surface normal estimation, utilizing deep learning, only emerged recently [BLF18; BM16; GKO+18]. Those method use large DNNs that are trained in an end-to-end fashion. It was shown that, given the required amount of training data, those methods are able to produce better results than existing fixed-function approaches, which can be contributed to their ability of adapting to the given data characteristics. However, they also have several disadvantages. They need large, annotated training datasets to begin with, are much slower than fixed function approaches, and are not interpretable. Further, they often ignore existing knowledge about the intrinsic problem structure so that the programs instantiations need to be selected from the full space of all continuous functions, mapping from points to unit vectors.

An important insight into the problem of unsigned surface normal estimation is that the 3-dimensional unit sphere is a double cover of the space of solutions, as the normal vectors $\mathbf{n}$ and $-\mathbf{n}$ describe the same surface plane. Thus, the problem can naturally be formulated as a least squares plane fitting problem. DISNE utilizes this problem-specific knowledge by building a differentiable algorithm around a fixed function IRLS scheme (cf. Section 3.3) for robust surface normal estimation. Instead of re-weighting the input points using a fixed, concave function, as it is traditionally done in IRLS, the re-weighting is done based on data, by inferring the weights for each iteration using a graph neural network on residual features and input points. Therefore, instead of spanning a differentiable program space of all functions from points to unit vectors, we only span a space for the subproblem of assigning weights to input points. Then, computing the normals is just a matter of plane fitting in local neighborhoods of individually weighted points.

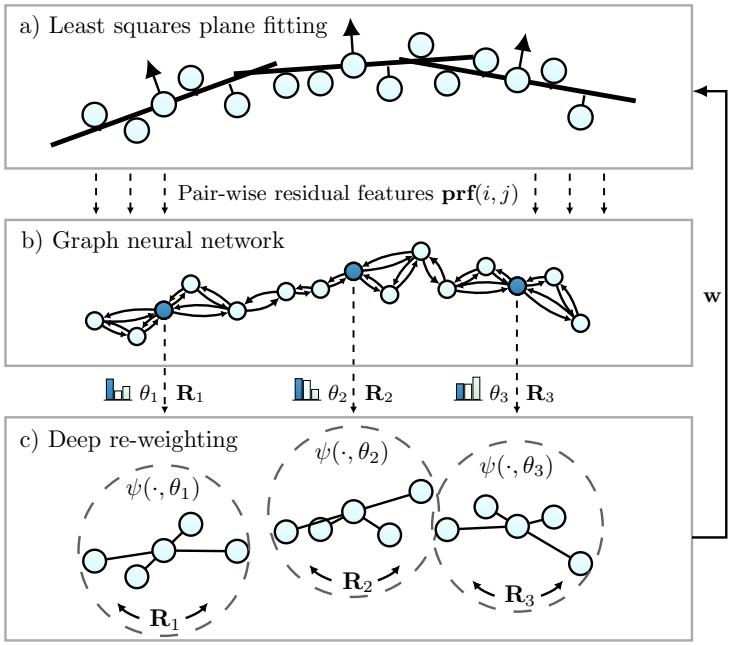


Figure 5.1: Overview of the DISNE method [LOM20]. The method consists of three steps, illustrated here on three different point neighborhoods. (a) First, normal vectors are obtained by optimizing weighted least squares. (b) Kernel parameters and local orientations are obtained using an LSGT, which receives pair-wise residual features from least squares as input descriptors. (c) Point pairs are re-weighted using a kernel INF, utilizing the parameters and rotations obtained from the GNN.

The differentiable algorithm is summarized in Figure 5.1. It consists of three steps, which are iteratively repeated multiple times, during training and inference. First, least squares plane fitting with uniform weights is applied to local neighborhoods of the input point clouds to obtain normal vectors. Then, the plane fitting error is used to compute pairwise residual features $\mathbf{prf}_{i,j}$, which are used as input to a GNN. The GNN takes the residual features $\mathbf{prf}_{i,j}$ and the input points and parameterizes an INF kernel ψ with parameters θ_i and provides approximately equivariant local reference frames $\mathbf{R}_i$ through the application of LSGTs (cf. Section 4.1.5). Lastly, the parameterized kernel INF ψ is used to assign weights $\mathbf{w}$ to points in local neighborhoods, which are used in the next iteration of weighted plane fitting. All in all, the following geometric characteristics are incorporated into the algorithm as inductive bias:

- rotation equivariance of normal vectors,
- point permutation invariance, and
- the double cover property of normal vectors on the sphere.

We show that the algorithm is able to slightly improve on the results of full DL approaches, while at the same time being orders of magnitude more efficient in terms

of parameters and execution time. Additionally, the approach is indeed easier to interpret, as we can visualize the weights and normal vectors within the algorithm.

The chapter will first discuss related work in Section 5.1, before Section 5.2 introduces the necessary background and formal introduction to the task of surface normal estimation. The method is then detailed in Section 5.3 and evaluated in Section 5.4.

5.1 Related Work

We first discuss pre-DL methods for surface normal estimation, which have been a subject to extensive research for many years. Traditionally, surface normal estimation is solved by fitting planes to manually chosen neighborhoods using specific, fixed kernel functions. The algorithms to solve the respective least squares problems usually involve unweighted PCA [HDD+92] solved with Eigendecomposition (ED) or plane fitting with Singular Value Decomposition (SVD) [HJ87; HM01]. For an overview of these approaches, the reader is referred to review literature [KAW+09]. These optimization-based approaches share the downside of depending on data-specific hyperparameters, such as neighborhood size or weighting function, which usually have to be chosen manually and adjusted for changing data characteristics like noise or varying point density. As a consequence, several heuristics for hyperparameter selection have been proposed, such as methods that automatically find appropriate neighborhood sizes for plane fitting [MNG04]. A common limitation of plane fitting methods is that they tend to smoothen sharp details, in fact, they can be seen as isotropic low-pass filters. Therefore, approaches that preserve sharp features have been proposed, such as estimating normals from Voronoi cells [AB98; MOG11] and combining that with PCA [ACT+07]. Alternative approaches include edge-aware sampling [HWG+13] or normal vector estimation in Hough space [BM12]. In addition, several methods arise from more complex surface reconstruction techniques, e.g. moving least squares (MLS) [Lev98], spherical fitting [GG07], jet fitting [CP03] and multi-scale kernel methods [ASL+17].

Deep learning methods. Deep learning based approaches also found their way into surface normal estimation with the recent success of deep learning in a wide range of domains. These approaches can be divided into two groups, depending on the actual type of input data they use. The first group aims at normal estimation from single images [BRG16; EF15; FGH13; LZP14; LSD+15; QLL+18; WFG15] and has received a lot of interest over the last few years due to the well understood properties of CNNs for grid-structured data.

The second line of research directly uses unstructured point clouds and emerged only recently, partially due to the advent of graph neural networks and geometric deep learning [BBL+17]. Boulch et al. [BM16] proposed to use a CNN on Hough transformed point clouds in order to find surface planes of the point cloud in Hough

space. Based on the widely used point processing network, PointNet [QSK+17; QYS+17], Guerrero et al. [GKO+18] proposed a deep multi-scale architecture for surface normal estimation. Later, Ben-Shabat et al. [BLF18] improved on those results using 3D point cloud fisher vectors as input features and a three-dimensional CNN architecture consisting of multiple expert networks.

5.2 Problem and Background

Let $\mathcal{S}$ be a manifold in $\mathbb{R}^3$, $\mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_N\}$ a finite set of sampled and possibly distorted points from that manifold and $\hat{\mathbf{N}} = \{\hat{\mathbf{n}}_1, ..., \hat{\mathbf{n}}_N\}$ the tangent plane normal vectors at sample points $\mathbf{p}_i$. *Surface normal estimation* for the point cloud $\mathcal{P}$ can be described as the problem of estimating a set of normal vectors $\mathbf{N} = \{\mathbf{n}_1, ..., \mathbf{n}_N\}$ given $\mathcal{P}$, whose directions match those of the actual surface normals $\hat{\mathbf{n}}_i$ as close as possible. We consider the problem of unoriented normal estimation, determining the normal vectors up to a sign flip. Estimating the correct sign can be done in a post-processing step, depending on the task at hand, and is explicitly tackled by several works [MGD+10; HLZ+19; WHG+15].

A standard approach to determine unoriented surface normals is fitting planes to the local neighborhood of every point $\mathbf{p}_i$ [Lev98]. Given a radius r or a neighborhood size k, we model the input as a nearest neighbor graph $G = (\mathcal{P}, \mathcal{E})$, where we have a directed edge $(i, j) \in \mathcal{E}$ if and only if $\|\mathbf{p}_j - \mathbf{p}_j\|_2 < r$ or if $\mathbf{p}_j$ is one of the k nearest neighbors of $\mathbf{p}_i$, respectively. Let $\mathcal{N}(i)$ denote the local neighborhood of $\mathbf{p}_i$, with $k_i = |\mathcal{N}(i)|$, containing all $\mathbf{p}_j$ with $(i, j) \in \mathcal{E}$. Furthermore, let $\mathbf{P}(i) \in \mathbb{R}^{k_i \times 3}$ be the matrix of centered coordinates of the points from this neighborhood, that is

$$\mathbf{P}(i)_j = \mathbf{p}_j^\top - \frac{1}{k_i} \sum_{m \in \mathcal{N}(i)} \mathbf{p}_m^\top, \qquad \mathbf{p}_j \in \mathcal{N}(i). \tag{5.1}$$

Fitting a plane to this neighborhood is then described as finding the least squares solution of a homogeneous system of linear equations:

$$\mathbf{n}_i^* = \underset{\mathbf{n}:\|\mathbf{n}\|_2=1}{\operatorname{argmin}} \|\mathbf{P}(i) \cdot \mathbf{n}\|_2^2 = \underset{\mathbf{n}:\|\mathbf{n}\|_2=1}{\operatorname{argmin}} \sum_{j \in \mathcal{N}(i)} \|\mathbf{P}(i)_j \cdot \mathbf{n}\|_2^2, \tag{5.2}$$

which can be solved in practice using ED or SVD, as described in Section 4.2.1. The simple plane fitting of Equation (5.2) is not robust and does not result in high-quality normal vectors. It produces accurate results only if there are no outliers in the data, which is never the case in practice. Additionally, this approach eliminates sharp details because it acts as a low-pass filter on the point cloud. Even when an isotropic radial kernel function $\theta(\|\mathbf{P}(i)_j\|)$ is used to weight points according to their distance to the local mean, fine details cannot be preserved.

Both issues can be resolved through integrating weighting functions into Equation (5.2). *Sharp features can be preserved* with an *anisotropic* kernel that infers weights of point pairs based on their relative positions, i.e.

$$\mathbf{n}_i^* = \underset{\mathbf{n}:|\mathbf{n}|=1}{\operatorname{argmin}} \sum_{j \in \mathcal{N}(i)} \psi(\mathbf{p}_j - \mathbf{p}_i) \cdot \|\mathbf{P}(i)_j \cdot \mathbf{n}\|_2^2, \tag{5.3}$$

where $\psi(\cdot)$ is an anisotropic kernel, considering the full Cartesian relationship between neighboring points, instead of only their distance. However, an anisotropic kernel is no longer rotation invariant, so that *equivariance* of output normals needs to be ensured additionally. *Robustness* to outliers can be achieved by another kernel that weights points according to an inlier score $s_{i,j}$. More specifically, Equation (5.2) is changed to

$$\mathbf{n}_i^* = \underset{\mathbf{n}:|\mathbf{n}|=1}{\operatorname{argmin}} \sum_{j \in \mathcal{N}(i)} s_{i,j} \cdot \|\mathbf{P}(i)_j \cdot \mathbf{n}\|_2^2, \tag{5.4}$$

where $s_{i,j}$ weights outliers with a low and inliers with a high score. However, in order to infer information about the outlier status of points an initial model estimation is necessary. A standard solution to this circular dependency is to formulate the problem as a sequence of weighted least squares problems [HW77; RK18] (cf. Section 3.3). Given the residuals $\mathbf{r}^l$ of the least squares solution from iteration l, the solution for iteration $l+1$ is computed as

$$\mathbf{n}_i^{l+1} = \underset{\mathbf{n}:|\mathbf{n}|=1}{\operatorname{argmin}} \sum_{j \in \mathcal{N}(i)} s(\mathbf{r}_{i,j}^l) \cdot \|\mathbf{P}(i)_j \cdot \mathbf{n}\|_2^2. \tag{5.5}$$

That is, the inlier score and the estimated model are refined in an alternating fashion.

5.3 Deep Iterative Surface Normal Estimation

In this section, the DISNE method is presented, which combines the described properties of robustness, anisotropy and equivariance with the deep learning property of adaptation to large data set statistics. An anisotropic, iterative weighting function is able to span the full space of possible solutions for normal vectors and includes all relevant data-dependent degrees of freedom. Therefore, in contrast to existing deep learning methods [BLF18; GKO+18], we do not directly regress normal vectors from point features but build a differentiable algorithm that uses DL only to infer weights for a single least squares optimization step, utilizing the problem specific knowledge outlined above.

The core of the algorithm is a trainable kernel INF $\psi : \mathbb{R}^3 \times \mathbb{R}^d \to \mathbb{R}$, which infers weights as

$$w_{i,j} = \psi(\mathbf{R}_i(\mathbf{p}_j - \mathbf{p}_i), \theta_i), \tag{5.6}$$

where θ_i are kernel parameters and $\mathbf{R}_i$ is a rotation matrix. The kernel is shared by all local neighborhoods of the point graph while θ_i and $\mathbf{R}_i$ are individual for each node. Because there is no a priori information about the structure of the input data, a reasonable approach is to model ψ as an MLP and to find kernel parameters through supervised learning from data. To this end, parameters θ_i and poses $\mathbf{R}_i$ for each neighborhood are jointly regressed by a graph neural network, specifically, a Local Spatial Graph Transformer (LSGT) (cf. Section 4.1.5), on the point neighborhood graph. Then, the kernel function ψ regresses anisotropic,

Algorithm 7 Differentiable iterative normal estimation

Input:
$\mathcal{P}$: Point cloud
L: Number of iterations
k or r: Neighborhood size (num. neighbors or radius)
Output:
$\mathbf{N}$: Normal vector estimations

$(\mathcal{P}, \mathcal{E}) \leftarrow$ Neighborhood graph from $\mathcal{P}$ and k / r
$\mathbf{C} \leftarrow \text{CovMatrices}(\mathcal{P}, \mathcal{E})$
$\mathbf{U}, \mathbf{\Sigma} \leftarrow \text{ParallelEig}(\mathbf{C})$ (cf. Section 4.2.3)
$\mathbf{N}^0 \leftarrow$ Extract Solutions from $\mathbf{U}$
for each $l \in \{1, ..., L\}$ **do**
  $(\Theta, \mathbf{Q}) \leftarrow \text{GNN}(\mathcal{P}, \mathcal{E}, \mathbf{N}^{l-1})$ (cf. Section 4.1.2)
  $\mathbf{R} \leftarrow \text{QuatsToMats}(\mathbf{Q})$ (cf. Section 4.1.5)
  $\mathbf{W} \leftarrow \text{ApplyKernel } \psi(\mathbf{R}, \mathcal{P}, \Theta, \mathcal{E})$ (cf. Section 4.3.3)
  $\mathbf{C} \leftarrow \text{WeightedCovMatrices}(\mathcal{P}, \mathbf{W}, \mathcal{E})$
  $\mathbf{U}, \mathbf{\Sigma} \leftarrow \text{ParallelEig}(\mathbf{C})$ (cf. Section 4.2.3) (cf. Section 4.2.3)
  $\mathbf{N}^l \leftarrow$ Extract Solutions from $\mathbf{U}$
end for
Return $\mathbf{N}^L$

approximately equivariant weights $w_{i,j}$ for each edge in the graph, which are used to find the normal vectors using traditional weighted least squares optimization

$$\mathbf{n}_i = \underset{\mathbf{n}:|\mathbf{n}|=1}{\text{argmin}} \sum_{j \in \mathcal{N}(i)} \underset{j \in \mathcal{N}(i)}{\text{softmax}}(w_{i,j}) \, \|\mathbf{P}(i)_j \cdot \mathbf{n}\|_2^2 , \tag{5.7}$$

in parallel for all $\mathbf{p}_i \in \mathcal{P}$, using softmax for normalization of weights in each neighborhood. Similar to iterative re-weighting least squares (c.f. Section 3.3), we apply the method in an iterative fashion to achieve robustness and provide the residuals of the previous solution as input to the graph neural network.

The core algorithm is formulated as pseudo code in Algorithm 7. The initial weighting of the points in a neighborhood is chosen to be uniform, which results in unweighted least squares plane fitting in the initial iteration. In the following, we present the graph neural network, the local quaternion rotation and our differentiable least square solver in more detail.

5.3.1 PRFNet for Kernel Parameterization

For regressing parameters θ_i and rotations $\mathbf{R}_i$ for the whole point cloud, Message Passing Graph Neural Networks (MP-GNNs) (cf. Section 4.1) are a natural fit because the network must be invariant to the order of points in a neighborhood and it must be able to allow weight sharing over neighborhoods with varying cardinality.

Network	Architecture		
h_1	$L(32)$,	$ReLU$,	$L(16)$
γ_1	$L(32)$,	$ReLU$,	$L(8)$
h_2	$L(32)$,	$ReLU$,	$L(16)$
γ_2	$L(32)$,	$ReLU$,	$L(8)$
h_3	$L(32)$,	$ReLU$,	$L(16)$
γ_3	$L(32)$,	$ReLU$,	$L(12)$
ψ	$L(64)$,	$ReLU$,	$L(1)$

Table 5.1: Details of the PRFNet architecture for iterative re-weighting. $L(x)$ stands for a linear layer with x output neurons.

As detailed in Section 4.1, MP-GNNs offer both of those properties. Therefore, an LSGT called PRFNet was designed to regress θ_i and $\mathbf{R}_i$ for each point $\mathbf{p}_i$, which is described in the following.

The PRFNet architecture consists of three consecutive neighborhood aggregation steps. Given MLPs h and γ, the neighborhood aggregation scheme is given by the message function

$$\mathbf{m}_{j,i} := \textsc{Me}_{\Theta_1}\left(\mathbf{x}_j^{\ell-1}, \mathbf{x}_i^{\ell-1}, \mathbf{e}_{j,i}\right) = h_{\Theta_1}\left(\mathbf{x}_j^{\ell-1} \,\|\, \mathbf{d}_{j,i} \,\|\, \mathbf{prf}_{j,i}\right), \tag{5.8}$$

with $\mathbf{e}_{i,j} = (\mathbf{d}_{i,j} \,\|\, \mathbf{prf}_{j,i})$ and the node update function

$$\mathbf{x}_i^{\ell} := \textsc{Up}_{\Theta_2}(\mathbf{x}_i^{\ell-1}, \square_{j \in \mathcal{N}(i)} \mathbf{m}_{j,i}) = \gamma_{\Theta_2}\Big(\frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \mathbf{m}_{j,i}\Big), \tag{5.9}$$

with $\|$ denoting feature concatenation. Using this scheme, we alternate between computing messages $\mathbf{m}_{j,i}$ and new node features $\mathbf{x}_i^{\ell}$. In addition to the Cartesian relation vector $\mathbf{d}_{j,i} = (\mathbf{p}_j - \mathbf{p}_i)$, pair-wise residual features, a modified version of *Point Pair Features (PPF)* [DBI18a; DBI18b], are provided as edge input features:

$$\mathbf{prf}_{j,i} = (|\mathbf{n}_i \cdot \mathbf{d}_{j,i}|, |\mathbf{n}_j \cdot \mathbf{d}_{j,i}|, |\mathbf{n}_i \cdot \mathbf{n}_j|, \|\mathbf{d}_{j,i}\|_2^2). \tag{5.10}$$

They are computed directly from the last set of least squares solutions $\mathbf{n}_i$ and contain the residuals as point-plane distances $|\mathbf{n}_i \cdot \mathbf{d}_{j,i}|$.

The GNN consists of three message passing layers, containing 6 MLPs in total, denoted as h_ℓ and γ_ℓ for $\ell \in \{1, 2, 3\}$ in the following Table 5.1, which lists the architectures of those MLPs and the kernel network ψ. The h and ψ networks are shared over all edges in the neighborhood graph while the γ are shared over all points. Additionally, all MLPs are shared over the iterations of the algorithm. Each MLP consists of two linear layers, separated by a ReLU non-linearity. Layer sizes are given in Table 5.1. All in all, the networks contain 7981 parameters.

After applying the message passing scheme, the output node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times (d+4)}$ is interpreted as a tuple $(\Theta \in \mathbb{R}^{N \times d}, \mathbf{Q} \in \mathbb{R}^{N \times 4})$, containing kernel and

rotation parameters for all nodes. We use the row-normalized elements of $\mathbf{Q}$ as unit quaternions to efficiently parameterize the rotation group $SO(3)$, as described in Section 4.1.5. By applying the differentiable map from quaternion space to the space of rotation matrices given in Algorithms 3 and 4, the local rotation matrices $\mathbf{R}_i$ for all point neighborhoods are efficiently computed in parallel. The full algorithm fulfills the following properties.

Permutation Equivariance Neighborhood aggregation is performed using an average operator, which is invariant regarding the order of points. Since there are no other functions over sets of points, the resulting network is equivariant to point permutation, as described in Section 4.1.2.

Varying neighborhood sizes For the cases in which we decide to use a radius graph instead of a k-NN graph, the network allows differently sized neighborhoods in one graph, since all parameters are shared over edge or nodes and the only operation over the whole neighborhood, the average, is agnostic to the neighborhood size.

Locality Due to using only local operators, the algorithm does not rely on global point cloud statistics or features. Therefore, it can be applied on partial point clouds and range scans, which is of importance for many practical applications.

5.3.2 Parallel Differentiable Least Squares

In every iteration of the presented algorithm, the plane fitting problem of Equation (5.7) needs to be solved. As outlined in Section 3.3, a standard approach to solve this problem is to utilize the Singular Value Decomposition of the weighted matrix $\text{diag}(\sqrt{\mathbf{w}_i^l})\mathcal{P}(i)$: Let $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathbf{T}$ be its decomposition, then the column vector of $\mathbf{V}$ corresponding to the smallest singular value is the optimal solution for the given least squares problem [HZ03; RK18]. However, N SVDs (for potentially varying matrix sizes) need to be solved in our scenario, one for each neighborhood, which makes this approach prohibitive. A much more efficient approach in this case is to consider the eigendecomposition of the weighted 3×3 covariance matrix $\mathbf{C}(i) = \mathcal{P}(i)^\top \text{diag}(\mathbf{w}_i^l)\mathcal{P}(i)$ which has the columns of $\mathbf{V}$ as its eigenvectors [HZ03]. The solution for Equation (5.7) is then the eigenvector associated with the smallest eigenvalue. The computational complexity for the eigendecomposition of this 3×3 matrix is $O(1)$ and hence for one overall iteration $O(N)$.

Our algorithm is trained end-to-end by minimizing the distance between ground truth normals and the least squares solution, requiring backpropagation through the eigendecomposition. The reader is referred to Section 4.2 for a discussion about differentiable ED and SVD. Here, given partial derivatives $\partial L/\partial \mathbf{U}$ and $\partial L/\partial \mathbf{\Sigma}$ for

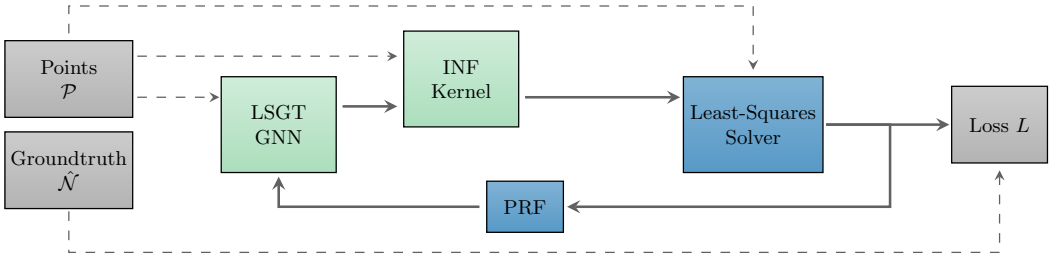


Figure 5.2: Data flow graph for the DISNE algorithm, consisting of ■ fixed function parts and the ■ trainable LSGT and INF networks. Solid arrows indicate data that need gradient information in reverse mode accumulation in order to train the algorithm in an end-to-end fashion. To obtain gradients for the network parameters, the fixed function realizations of the least squares solver and the pair-wise residual features need to be differentiable.

eigenvectors and eigenvalues, respectively, we compute the partial derivatives for a real symmetric 3×3 covariance matrix $\mathbf{C}$ as

$$\frac{\partial L}{\partial \mathbf{C}} = \mathbf{U}\Big((\frac{\partial L}{\partial \mathbf{\Sigma}})_{diag} + \mathbf{F} \circ \mathbf{U}^\top \frac{\partial L}{\partial \mathbf{U}}\Big)\mathbf{U}^\top, \tag{5.11}$$

where $\mathbf{F}_{i,j} = (\lambda_j - \lambda_i)^{-1}$ contains inverse eigenvalue differences. We implemented forward and backward steps for eigendecomposition of a large number of symmetric 3×3 matrices, where we parallelize over graph nodes, leading to an $O(1)$ implementation (using $O(N)$ processors) of parallel least squares solvers.

Handling numerical instability Backpropagation through the eigendecomposition can lead to numerical instabilities due to at least two reasons: 1) Low-rank input matrices with two or more zero eigenvalues. 2) Exploding gradients when two eigenvalues are very close to each other and values of $\mathbf{F}$ go to infinity. In DISNE, these problems are avoided by applying two practical techniques. First, a small amount of noise is added to the diagonal elements of all covariance matrices, making them full-rank. Second, gradients are clipped after the backward step on very large values, to tackle the cases of almost identical eigenvalues.

5.3.3 Training

Training is performed by minimizing the Euclidean distance between estimated normals $\mathbf{N}$ and ground truth normals $\hat{\mathbf{N}}$, averaged over all normal vectors in the training set:

$$L(\hat{\mathbf{N}}, \mathbf{N}) = \frac{1}{N}\sum_{i=1}^{N} \min(\|\hat{\mathbf{n}}_i - \mathbf{n}_i\|_2, \|\hat{\mathbf{n}}_i + \mathbf{n}_i\|_2), \tag{5.12}$$

where the minimum of the distances to the flipped or non-flipped ground truth vectors is used. While we also experimented with different angular losses, we found

that the Euclidean distance loss still provides the best result and the most stable training. A loss is computed after each least squares step and the network is trained iteratively by performing a gradient descent step after each iteration of the algorithm. This fights vanishing gradients that occur due to the normalization of vectors in quaternion and eigenvector computations. The weights of our network are shared over algorithm iterations, allowing generalization to further iterations.

5.3.4 Differentiability

The computation graph of the DISNE method, including trainable and fixed-function parts, is shown in Figure 5.2. It can be seen that, in order to train the kernel INF and GNN, the gradients only need to be backpropagated through the fixed-function ED for least squares optimization.

The implementation of the computation graph is based on the *Pytorch Geometric* library [FL19] and uses the provided scheme consisting of scattering and gathering between node and edge feature space, as described in Section 4.1. Therefore, varying neighborhood sizes (e.g. varying node degree) can still be handled in parallel on the GPU by parallelization in graph edge space.

For ED of a large number of symmetric 3×3 matrices and for the parallel quaternion to rotation matrix map custom modules were implemented as described in Section 4.2.1 and 4.1.5. The modules include efficient forward and backward steps on GPU and CPU, which can be parallelized over the whole point cloud.

5.4 Experiments

The following experiments compare the proposed DISNE algorithm with state-of-the-art methods both quantitatively, measuring normal estimation accuracy and model complexity, and qualitatively, on a Poisson reconstruction and on a transfer learning task. Further, we show improved interpretability by quantitatively analyzing normals and weights over the course of iterations. Section 5.4.1 introduces the dataset used to train our model and the protocol followed in our experiments. Then, qualitative (Section 5.4.2) and quantitative (Section 5.4.6) results are presented and an analysis of complexity and execution time (Section 5.4.3) is given.

5.4.1 PCPNet Dataset and Experimental Setup

The DISNE method is trained and validated quantitatively on the PCPNet dataset as provided by Guerrero et al. [GKO+18]. It consists of a mixture of high-resolution scans, point clouds sampled from handmade mesh surfaces and differentiable surfaces. Each point cloud consists of 100k points. We reproduce the experimental setup of [BLF18; GKO+18], training on the provided split containing 32 point clouds affected by different levels of noise. The test set consists of six categories, containing

		Noise [σ]			Varying Density		
	No noise	0.00125	0.006	0.012	Stripes	Grad.	Avg.
PCA	12.29	12.87	18.38	27.5	13.66	12.81	16.25
Jet [CP03]	12.23	12.84	18.33	27.68	13.39	13.13	16.29
HoughCNN [BM16]	10.23	11.62	22.66	33.39	12.47	11.02	16.9
PCPNet [GKO+18]	9.68	11.46	18.26	22.8	11.74	13.42	14.56
Nesti-Net [BLF18]	6.99	10.11	17.63	22.28	8.47	9.00	12.41
Ours ($k = 64$, $L = 4$)	**6.72**	**9.95**	**17.18**	**21.96**	**7.73**	**7.51**	**11.84**

Table 5.2: Results for unoriented normal estimation. Shown are normal estimation errors in angle RMSE. For PCA and Jet, optimal neighborhood size for average error is chosen. For our approach, we display results for a balanced neighborhood size $k = 64$, which improves on the state of the art for all noise levels. Results for different k are shown in Table 5.3.

four sets with different levels of noise (no noise, $\sigma = 0.00125$, $\sigma = 0.0065$ and $\sigma = 0.012$) and two sets with different sampling density (striped pattern and gradient pattern).

In the following experiments, we evaluate unoriented normal estimation, without considering the sign of the normals. The Root Mean Squared Error (RMSE) on the provided 5k points subset is used as performance metric following the protocol of related work, where the RMSE is first computed for each test point cloud before the results are averaged over all point clouds in one category. Model selection was performed manually using the provided validation set. Despite inheriting the neighborhood size parameter from traditional PCA, it is possible for a network trained on a specific neighborhood size k to be applied for other k as well. This is because all networks can be shared across an arbitrary number of points and the softmax function normalizes weights for neighborhoods of varying sizes. We observed that generalization across different k only leads to a very small increase in average error. However, to fairly evaluate our method for different k, a network is trained for each $k \in \{32, 48, 64, 96, 128\}$. Training consists of 300 epochs using the RMSProp optimization method [TH12]. All reported test results are given after 4 re-weighting iterations of our algorithm. Iterating longer does not show significant improvements. In addition to RMSE results on the given test set, we evaluate quantitative results over different numbers of iterations, results for extrapolation over iterations and generalization between different k. For further realization details, the reader is referred to the implementation[1].

5.4.2 Quantitative Evaluation

First, RMSE results for the DISNE method (with $k = 64$) and related works on the PCPNet test set are shown in Table 5.2. DISNE improves on the state of the art

[1] https://github.com/nnaisense/deep-iterative-surface-normal-estimation

			Noise [σ]			Varying Density		
	Neigh. Size k	No noise	0.00125	0.006	0.012	Stripes	Grad.	Avg.
PCA	k = 32	9.10	11.22	28.41	45.35	10.48	9.96	19.09
	k = 48	9.94	11.56	23.00	38.48	11.40	10.74	17.52
	k = 64	10.68	12.08	20.68	33.67	12.07	11.35	16.75
	k = 96	11.93	12.71	18.81	28.81	13.18	12.36	16.30
	k = 128	12.54	12.97	18.12	26.67	14.07	13.21	16.26
Ours L = 4	k = 32	**6.09**	10.22	18.17	25.17	**7.22**	**6.84**	12.28
	k = 48	6.63	**9.63**	17.36	22.40	7.63	7.19	**11.81**
	k = 64	6.72	9.95	17.18	21.96	7.73	7.51	11.84
	k = 96	6.82	10.45	17.03	**21.80**	7.87	7.69	11.94
	k = 128	7.35	9.64	**16.90**	22.13	8.67	8.49	12.20

Table 5.3: Comparison of unoriented normal estimation RMSE between the proposed method and PCA for different neighborhood sizes k. It can be seen that our method consistently provides lower errors while being significantly more robust to changes of that parameter, compared to PCA.

on all noise levels and varying densities. While the improvement is only small, it should be noted that we reach it while being orders of magnitude faster and more parameter efficient (c.f. Section 5.4.3), which is of importance for many applications in resource constrained environments. For the non deep learning approaches, PCA and Jet, results for medium neighborhood sizes are displayed. Further, results for different k are provided in Table 5.3 and compared to errors obtained by PCA with the same respective neighborhood size. Our method performs better than the PCA baseline in all scenarios. As expected, varying k leads to a behavior similar to that of PCA, with large k's performing better on more noisy data. However, it can be observed that our approach is more robust against changes of k: Even for small neighborhood sizes, high noise is handled significantly better than by PCA and large neighborhoods still produce satisfactory results for low noise data. It should be noted that for all evaluated k we improve on the state of the art with respect to average error.

While the RMSE error metric is well suited for a general comparison, it is not a good proxy to estimate the ability of recovering sharp features since it does not take the error distribution over angles into account. Therefore, as an additional metric, Figure 5.3 presents the percentage of angle errors falling below different angle thresholds. The results confirm that our approach is better at preserving details and sharp edges, especially for low noise point clouds and varying density, where it outperforms other approaches. For higher noise, the achieved results are similar to those of Nesti-Net.

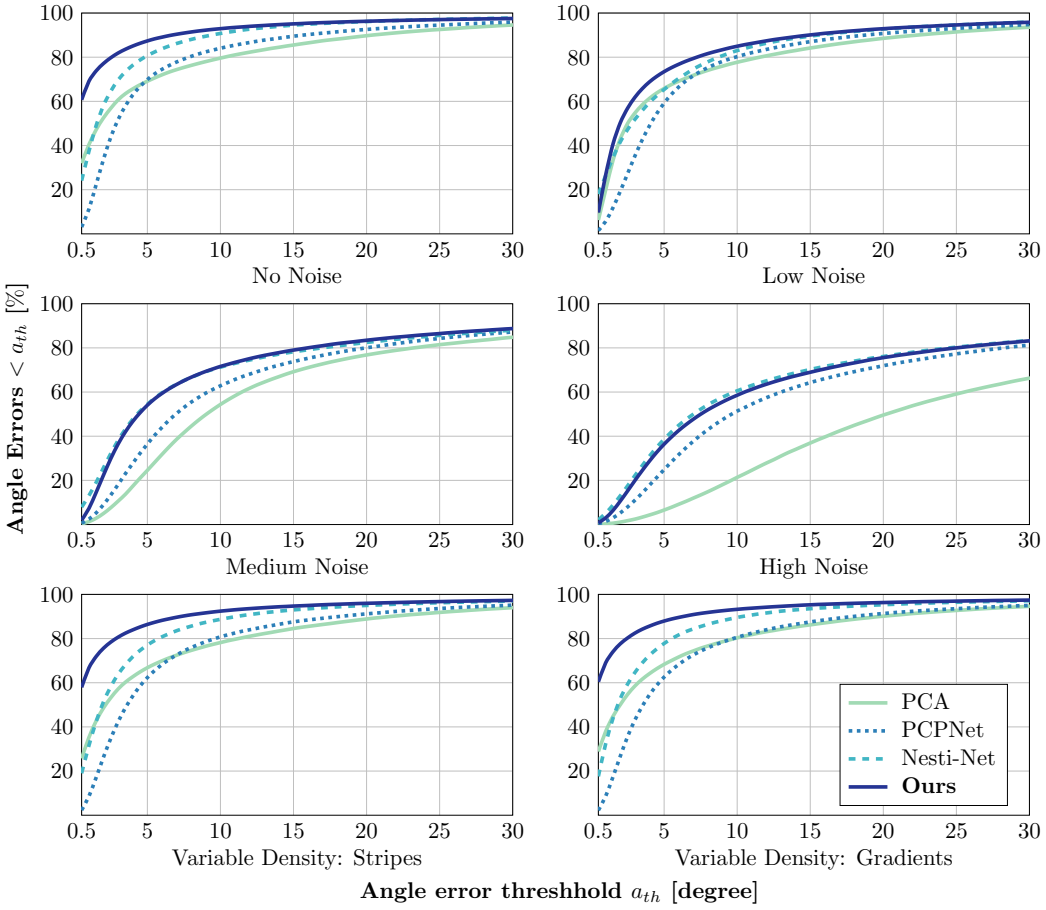


Figure 5.3: Comparison for varying angle error threshold [LOM20]. For error thresholds on the horizontal axis, the vertical axis shows the percentage of normals which have an error lower than that threshold. Our method and PCA use neighborhood size $k = 64$. For low noise settings and varying density, our method succeeds in recovering sharp features, as shown by the higher accuracies for low angle thresholds.

5.4.3 Efficiency

The DISNE model is small, consisting of only 7981 trainable parameters, shared over iterations and spatial locations. On a single Nvidia Titan Xp, a point cloud with $100k$ points is processed in 5.67 seconds (0.0567 ms per point). A large part of this execution time is consumed by the kd-tree used to compute the nearest neighbor graph, which takes 2.1 seconds of the 5.67 seconds. It is run on the CPU and could be further sped-up by utilizing GPUs.

In Table 5.4, the practical complexity of the DISNE approach is compared against the related deep learning approaches Nesti-Net and PCPNet. DISNE is orders of magnitude (378× and 131×) faster than the related approaches. The comparison was made as fair as possible by excluding nearest neighbor queries (note that this

	Ours	Nesti-Net [BLF18]	PCPNet [GKO+18]
Num. parameters	**7981**	179M	22M
Exec. time, 100k p.	**3.57 s**	1350 s	470 s
Relative exec. time	**1**×	**378**×	**131**×

Table 5.4: Comparison of efficiency between the approaches using deep learning. We list number of model parameters as well as average execution times for estimating normals on a point cloud with 100k points.

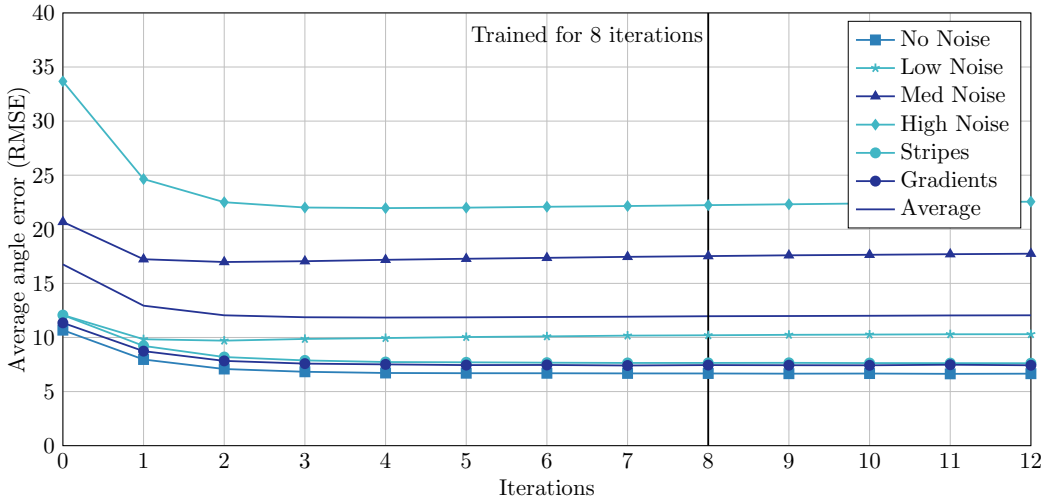


Figure 5.4: Test errors (RMSE) over iterations of the proposed algorithm [LOM20]. Iteration 0 shows results for unweighted PCA only. The network was trained on the training set for 8 iterations. For evaluation, we perform four additional iterations to evaluate stability.

favors the other approaches since they need larger neighborhoods) and the original implementations. The speedup of the DISNE method can be contributed to the much smaller network size and the parallel design of the GNN and least squares optimization steps.

5.4.4 Behaviour Over Iterations

The algorithm is trained for $L = 8$ (performing 8 iterations of re-weighting), where we compute a loss and perform an optimization step after each iteration. It produces normal vector estimations after each iteration, which can be analyzed quantitatively. The RMSE results for the PCPNet test set over algorithm iterations are shown in Figure 5.4. It can be seen that after iteration 4, further iterations do not lead to significant improvements. Also, the algorithm behaves reasonable stable, not diverging immediately after we pass the iterations for which the network was trained. However, we observe a small drift in favor of low-noise datasets over the iterations.

Errors for the test sets with no noise or variable density still decrease further while errors for data with higher noise levels slightly increase. Meanwhile, the average error stays nearly constant.

			Noise [σ]			Varying Density		
	k^{test}	No noise	0.00125	0.006	0.012	Stripes	Grad.	Avg.
$k^{\text{train}} = 32$	$k^{\text{test}} = 32$	6.09	10.22	18.17	25.17	7.22	6.84	12.28
	$k^{\text{test}} = 48$	6.96	10.01	17.44	22.97	7.92	7.46	12.12
	$k^{\text{test}} = 64$	7.43	10.09	17.22	22.33	8.51	8.06	12.27
	$k^{\text{test}} = 96$	8.25	10.37	17.08	21.91	9.43	8.80	12.64
	$k^{\text{test}} = 128$	8.77	10.62	17.05	7.21	9.90	9.21	12.89
$k^{\text{train}} = 64$	$k^{\text{test}} = 32$	6.13	10.19	18.28	25.20	7.21	6.89	12.31
	$k^{\text{test}} = 48$	6.47	9.93	17.43	22.53	7.55	7.17	11.85
	$k^{\text{test}} = 64$	6.72	9.95	17.18	21.96	7.73	7.51	11.84
	$k^{\text{test}} = 96$	7.10	10.18	17.01	21.69	8.16	8.04	12.00
	$k^{\text{test}} = 128$	7.27	10.35	16.94	21.67	8.03	8.49	12.10
$k^{\text{train}} = 128$	$k^{\text{test}} = 32$	6.66	9.89	20.98	30.99	7.80	7.48	13.97
	$k^{\text{test}} = 48$	7.01	9.57	18.40	24.94	8.14	7.75	12.63
	$k^{\text{test}} = 64$	7.24	9.50	17.63	23.20	8.37	8.11	12.34
	$k^{\text{test}} = 96$	7.29	9.50	17.07	22.34	8.61	8.39	12.20
	$k^{\text{test}} = 128$	7.35	9.64	16.90	22.13	8.67	8.49	12.20

Table 5.5: Results for transferring models between different neighborhood sizes k. Shown are RMSE values for models trained with $k^{\text{train}} \in \{32, 64, 128\}$, each tested with $k^{\text{test}} \in \{32, 48, 64, 96, 128\}$.

5.4.5 Transfer Between Neighborhood Sizes

The proposed algorithm generalizes reasonably well between neighborhood sizes, meaning that a model trained using neighborhood size k^{train} can be applied using a different neighborhood size k^{test} while producing good results. For verification, RSME errors for different combinations of k^{train} and k^{test} are reported in Table 5.5. It can be seen that, if the difference in neighborhood size is not too big, transferred models often only perform slightly worse than models trained directly for the appropriate k. However, transferring over a very large difference, like from 128 to 32 or the other way around, leads to a significant decrease in performance. The model trained on the balanced $k = 64$ performs very well on all other neighborhood sizes.

Additionally, Table 5.6 provides results for applying the model on even smaller neighborhood sizes, to evaluate the minimum k before the method breaks down. We found that when using a $k^{\text{train}} <\approx 30$, the training becomes unstable, which is why we transfer the model from $k^{\text{train}} = 32$ to smaller k^{test}. Results show that the

			Noise [σ]			Varying Density		
	k^{test}	No noise	0.00125	0.006	0.012	Stripes	Grad.	Avg.
$k^{\text{train}} = 32$	$k^{\text{test}} = 2$	17.26	54.02	61.08	61.29	19.50	22.89	39.34
	$k^{\text{test}} = 4$	7.23	49.66	60.91	61.26	8.14	8.44	32.59
	$k^{\text{test}} = 8$	5.63	33.65	55.32	58.89	6.53	6.51	27.75
	$k^{\text{test}} = 16$	5.36	13.80	28.17	41.37	6.36	6.23	16.88
	$k^{\text{test}} = 24$	5.77	10.74	19.78	28.99	6.71	6.57	13.09
	$k^{\text{test}} = 32$	6.09	10.22	18.17	25.17	7.22	6.84	12.28

Table 5.6: Results for transferring the model trained on $k^{\text{train}} = 32$ to even smaller $k^{\text{test}} \in \{2, 4, 8, 16, 24, 32\}$ until the method breaks down. Note that $k^{\text{test}} = 2$ means 2 neighbors, excluding point i, so there are still 3 points in total for each neighborhood, avoiding underdefined plane fitting problems.

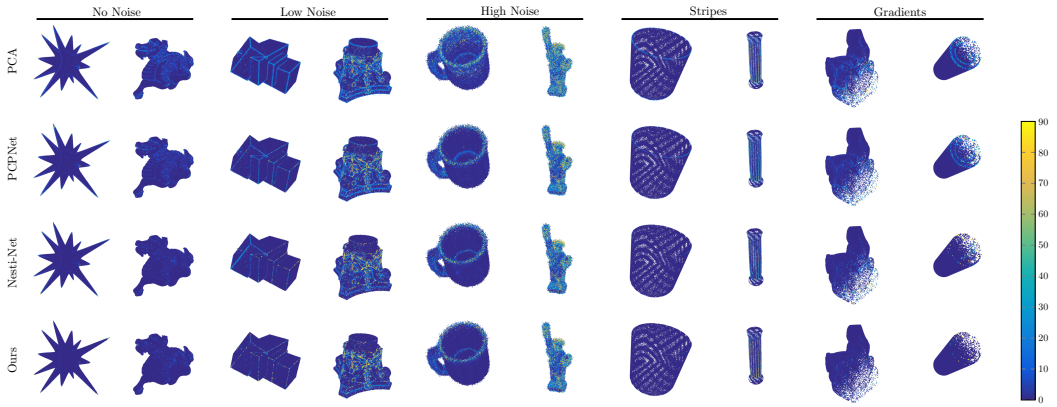


Figure 5.5: Qualitative comparison between our method ($k = 64$, $L = 4$) and related work [LOM20]. We show diverse examples from the test set, sampled from different categories, noise levels and density variations. The color encodes the angle error of estimated normals in degrees. Best viewed in the digital version.

algorithm provides good results for noise-free data down to $k = 4$. For noisy data, the approach breaks down quite fast when lowering k, as expected: At least $k = 24$ is required to provide reliable results. For lower k, the results approach the accuracy of random normals.

5.4.6 Qualitative Evaluation

This section visually presents surface normal errors for various elements of the PCPNet test set in Figure 5.5 and compares them against results from the PCA baseline and related deep learning approaches. It can be seen that the biggest improvements are obtained for low noise scenarios and varying density, where the DISNE method is able to preserve sharp features of objects better than the other

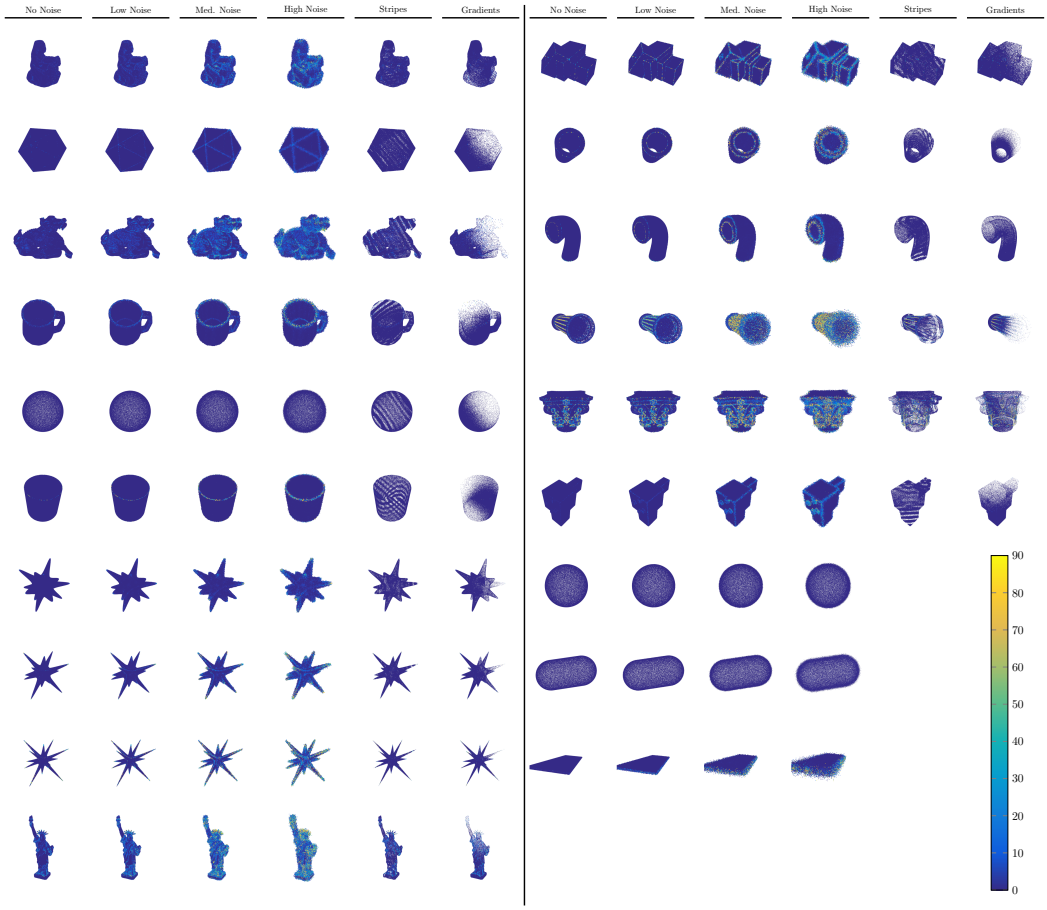
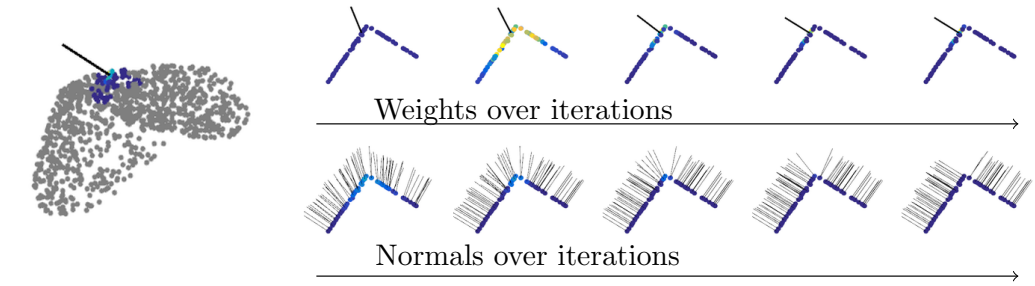


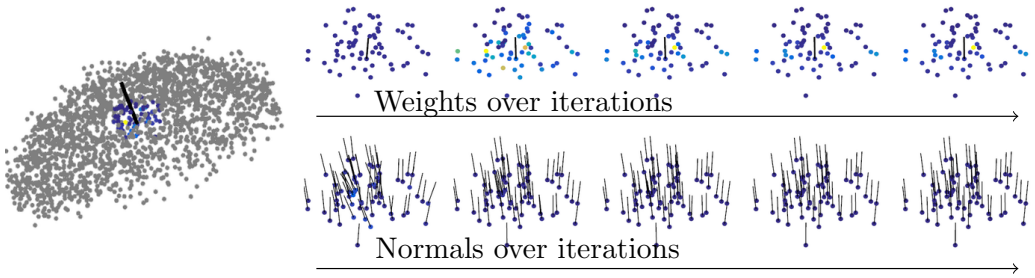
Figure 5.6: Qualitative results for all examples of the test set [LOM20]. Colors encode the RMSE in degree for each point. Best viewed in the digital version.

methods. In general it can be observed that our approach tends to provide sharp, discriminative normals for points on edges instead of smooth averages. In rare cases, this can lead to a false assignment of points to planes, as we can see in the example in column 8. It can be observed that, in contrast to Nesti-Net, our approach behaves equivariant to input rotation as is seen clearly on the diagonal edge of the box example in column 3. Sharp edges are kept also in uncommon rotations, which we can attribute to the applied LSGT network.

Lastly, we provide qualitative results for the whole PCPNet test set in Figure 5.6. For point clouds with varying density, the point size is reduced in order to better visualize the densities. Similar to the selected results in Figure 5.5, it can be seen that the method produces very sharp normal vectors, which usually resemble the plane normal of one of the plausible planes in the neighborhood. Objects consisting of primitives are good examples to show equivariance, as all edges show similar errors, independent of orientation. Sometimes, points are assigned to a false plane,



(a) Sharp edge



(b) Noisy surface

Figure 5.7: Local behaviour of our method over several iterations for (a) a sharp edge and (b) a noisy surface [LOM20]. The partial point clouds where sampled from the PCPNet test dataset. The colors in the first rows show the weights from the kernel network for one normal in the neighborhood while the colors in the second row show the angle error of all neighborhood normals.

leading to high error normal vectors. Compared to other approaches, we do not observe heavy smoothing around edges.

Interpretability. In order to interprete the results of our method, Figure 5.7 shows a detailed view of local neighborhoods over several iterations of our algorithm. An example for a sharp edge is shown in Figure 5.7a and a high noise surface in Figure 5.7b. Both sets of points were sampled from the real test data. For the sharp edge, the algorithm initially fits a plane with uniform weights, leading to smoothed normals. Over the iterations, high weights concentrate on the more plausible plane, leading to recovering of the sharp edge. In the noisy example, we can see that outliers are iteratively receiving lower weights, leading to more stable estimates.

Surface reconstruction. To further evaluate the quality of the produced normals when used as input to other computer vision pipelines, Figure 5.8 shows the results for Poisson surface reconstruction. Since the methods in this comparison all perform unoriented normal estimation (Guerrero et al. [GKO+18] evaluates both, unoriented and oriented, where we chose the unoriented version for a fair comparison), we

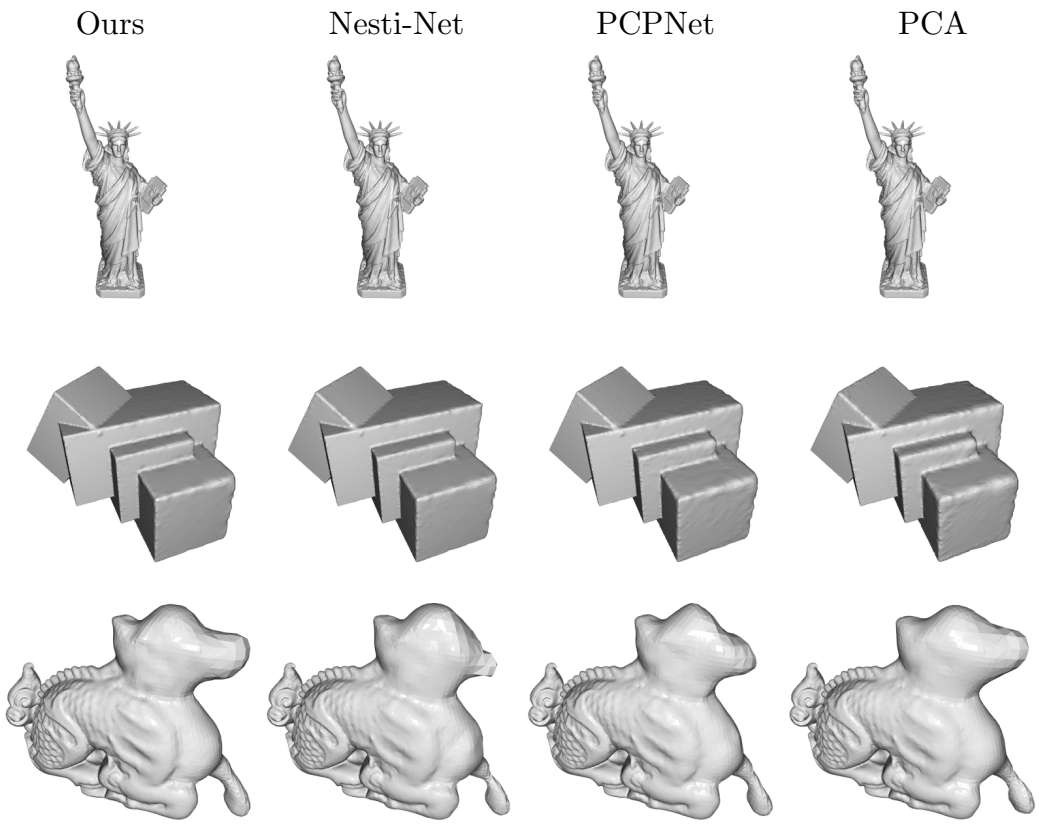


Figure 5.8: Selected results after applying Poisson surface reconstruction using the estimated normal vectors [LOM20]. In most cases, differences between the methods are very small. Examples 2 and 3 show reconstructions from point clouds with varying density, which show the largest differences.

determine the signs of the output normals from all four methods using the ground truth normals. Most of the reconstructions show only small differences, with our approach and Nesti-Net retaining slightly more details than the others. Significant differences can be observed for point clouds with varying density, displayed in rows 2 and 3. Here, our approach successfully retains the original structure of the object while still providing sharp edges.

Transfer to NYU depth dataset. In order to show generality of the DISNE approach, the models which were trained on the PCPNet dataset are validated on the NYU depth v2 dataset [NF12], a common benchmark dataset in the field of estimating normals from single images. It contains 1449 aligned and preprocessed RGBD frames, which are transformed to a point cloud before applying our method. After performing unoriented estimation, the normals are flipped towards the camera position. Evaluation is done qualitatively, since the dataset does not contain ground truth normal vectors. Results for three different neighborhood sizes in comparison

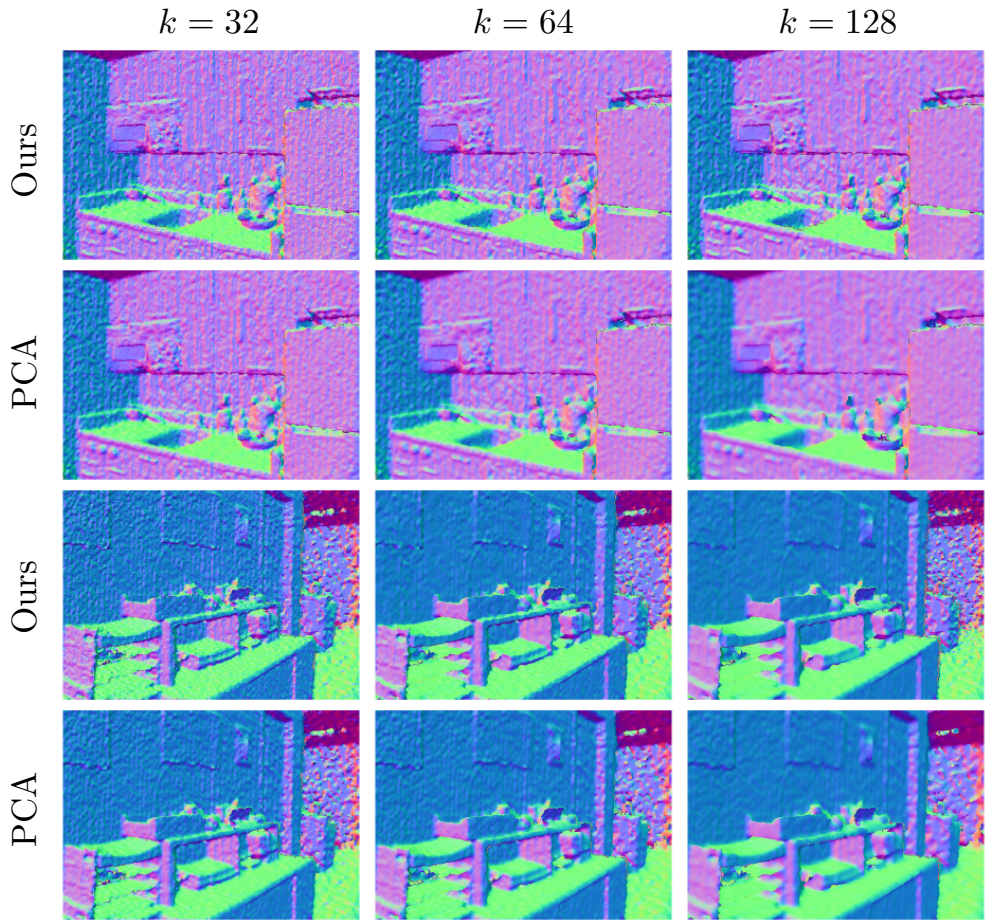


Figure 5.9: Examples for normal estimation on scanned data from the NYU depth v2 dataset [LOM20]. Colors encode the orientation of normals. Our model generalizes to this dataset while being able to retain more details and sharper edges than PCA. However, scanning artifacts are also kept and visible. Best viewed in the digital version of the thesis.

to PCA are shown in Figure 5.9. The DISNE approach behaves as expected, as it is able to infer plausible normals for the given scenes. For all k, our approach is able to preserve sharp features while PCA produces very blurry results. However, this also leads to the sharp extraction of scanning artifacts, which can be seen on the walls of the scanned room.

5.5 Discussion

This chapter presented the DISNE method, a novel differentiable algorithm for deep surface normal estimation on unstructured point clouds, consisting of parallel, differentiable least squares optimization and deep re-weighting. In each iteration,

the weights are computed using a kernel function that is individually parameterized and rotated for each neighborhood by a task-specific graph neural network.

It could be shown that DISNE is able to reach the goals of a differentiable algorithm. It is much more efficient in number of parameters and execution time than previous, full end-to-end deep learning methods while slightly improving on state-of-the-art accuracy on the given task. In addition, it has favorable properties like approximate equivariance, robustness to noise and higher interpretability than pure DL approaches. All those features are achieved by using problem-specific knowledge to design appropriate inductive biases.

A limitation of the DISNE method might be the limited receptive field considered as input for each normal vector. The applied GNN does not use global information about the object but purely regresses the normals using inputs from a local patch. In a hypothetical scenario in which much more annotated training data is available, which would allow to infer more object level information, an architecture which considers a wider receptive field might provide a stronger parameterization for normal estimation on individual neighborhoods. However, on the currently available datasets, the experiments show that more global approaches as Nesti-Net [BLF18] and PCPNet [GKO+18] are not able to make use of this information.

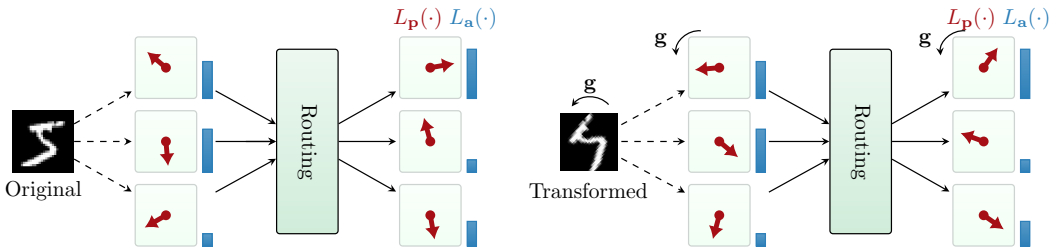
Chapter 6

Group Capsule Networks for Orientation Estimation

This chapter introduces Group Equivariant Capsule Networks (GECNs), a differentiable algorithm for processing 2D and 3D data, which comes with provable equivariance and invariance properties for certain Lie group transformations. They were originally presented in 2018 [LFL18], from which this chapter is adapted. The 3D version as discussed in Section 6.6 was published in 2020 [ZBL+20]. Equivariance and invariance are an important concept in deep learning architectures. Convolutional Neural Networks (CNNs), for example, heavily rely on equivariance of the convolution operator under translation. Weights are shared between different spatial positions, which reduces the number of parameters in a model and pairs well with translational symmetries and hierarchical object relations in image data. It naturally follows that a large amount of research is done to exploit other underlying transformations and symmetries and provide deep neural network models with equivariance or invariance with respect to those transformations as well. Further, equivariance and invariance are useful properties when aiming to produce data representations that disentangle factors of variation: when transforming a given input example by varying one factor, we usually aim for equivariance in one part of the representation and invariance in another. An important example for such a transformation is rotation, for which an example for factor disentanglement is given in Figure 6.1 for the 2D image domain. When rotating the input image of an architecture, we would like one part of the output representation, that which describes the image content, to not change (invariance) and another part of the representation to change according to input rotation (equivariance), encoding the pose of the image content. One recent line of methods that aim to provide a relaxed version of such a setting are *capsule networks* [HKW11; SFH17]. Instead of extracting scalar features from input data, they compute feature tuples, containing a pose and an activation. Capsule networks will be described in more detail in Section 6.1.

GECNs, the topic of this chapter, are a formalized version of capsule networks that guarantees the properties of equivariance and invariance with respect to certain transformations. Further, they bring together capsule networks with *group convolutions* [CW16], which also provide provable equivariance properties under the transformations of a group.

The chapter is structured in five parts. First, it introduces capsule networks as proposed by Hinton et al. [HKW11] and Sabour et al. [SFH17] in Section 6.1, before



Equivariance of pose vectors: $L_{\mathbf{p}}(\mathbf{g} \circ \mathbf{P}, \mathbf{a}) = \mathbf{g} \circ L_{\mathbf{p}}(\mathbf{P}, \mathbf{a})$; Invariance of agreements: $L_{\mathbf{a}}(\mathbf{g} \circ \mathbf{P}, \mathbf{a}) = L_{\mathbf{a}}(\mathbf{P}, \mathbf{a})$

Figure 6.1: One layer of dynamic routing in a capsule network, resulting in equivariant pose vectors $L_p^j(\mathbf{P}, \mathbf{a})$ and invariant agreements $L_a^j(\mathbf{P}, \mathbf{a})$ [LFL18]. Layers with those properties can be used to build viewpoint invariant architectures, which disentangle factors of variation.

introducing the formal framework of GECNs in Section 6.3, following the derivations of the original publication [LFL18]. Then, GECNs are formally introduced on vector fields in Section 6.4, as a more detailed extension of the original publication. Further, two instantiations of the method on 2D and 3D data are described in Section 6.5 and Section 6.6, which were first published in 2018 [LFL18] and 2020 [ZBL+20], respectively.

6.1 Capsule Networks

Capsule networks [HKW11] and their main feature, the routing by agreement algorithm [SFH17], represent a novel paradigm for deep neural networks for vision tasks, which extends the basic ideas of CNNs. They aim to hard-wire the ability to disentangle the pose of an object in an image from the evidence of its existence, also called *viewpoint equi- and invariance* in the context of vision tasks. Similar to CNNs they detect linear, hierarchical relationships occurring in the data, where semantically richer objects are composed of several parts that lie lower in the hierarchy. As input, they can receive images or irregularly structured data, depending on the architecture. As output, they produce capsules, which are described in the following. The output of one capsule network layer is a set of tuples $(\mathbf{p}, a)$, containing a pose vector $\mathbf{p}$ and an activation a, instead of just an activation as in regular CNNs. Sabour et al. [SFH17] encode a as length of the pose vector $\mathbf{p}$ and Hinton et al. [HSF18] use matrix poses. Here, we summarize the general concept by talking about generic pose vectors and activations, where a pose vector can be any representation that describes poses. Sabour et al. [SFH17] describe the dynamic routing by agreement algorithm that iteratively computes how to route capsule data from one layer to the next. The process of dynamic routing receives n capsules $(\mathbf{p}_i, a_i)_{1 \leq i \leq n}$, containing activations $a_i \in \mathbb{R}$ and pose vectors $\mathbf{p}_i \in \mathbb{R}^d$ and produces m output capsules $(\mathbf{p}_j, a_j)_{1 \leq j \leq m}$. First,

the input poses are transformed by trainable linear transformations $\mathbf{T}_{i,j} \in \mathbb{R}^{d \times d}$ to cast $n \cdot m$ votes, n votes for each of the m output capsules, respectively:

$$\mathbf{v}_{i,j} = \mathbf{T}_{i,j} \cdot \mathbf{p}_i.$$

For the j'th output pose, a weighted average $\hat{\mathbf{p}}_j$ of the n votes $\mathbf{v}_{\cdot,j}$ is calculated to receive a first proposal for the output capsule pose. Then, the weights are iteratively adjusted by choosing them in proportion to the inverse distances between votes $\mathbf{v}_{\cdot,j}$ and the average $\hat{\mathbf{p}}_j$, which amounts to executing an IRLS scheme (cf. Section 3.3) that iteratively refines the output poses. Lastly, the agreement values a_j are computed to encode how strong the votes agree on the output pose.

Intuitively, one can understand routing by agreement as a detector of input pose combinations. If the n input poses $\mathbf{p}_i$ match the inverse of the trained matrices $\mathbf{T}_{i,j}$ well, and thus the resulting votes $\mathbf{v}_{\cdot,j}$ agree well on a single pose, the resulting high activation indicates that the specific pose pattern in $\mathbf{T}_{i,j}$ exists in the input poses. Each output capsule checks for one of those patterns. The detection is robust in that it filters outliers over iterations so that a small number of not matching input capsules does not distort the output pose significantly.

Capsule networks using the described routing by agreement algorithm do not come with guaranteed equivariance or invariance, which are, however, essential to guarantee disentangled representations and viewpoint invariance. During the work on GECNs, we identified two issues that prevent exact equivariance in previous capsule architectures: First, the averaging of votes takes place in a vector space, while the underlying space of poses is a manifold. The vote averaging of vector space representations does not produce equivariant mean estimates on the manifold. Second, capsule layers use trainable transformation kernels defined over a local receptive field in the spatial vector field domain, where the receptive field coordinates are agnostic to the pose of the receptive field, instead of considering the induced action of semi-direct group products (cf. Section 3.2). Both of these issues lead to non-equivariant votes and consequently, non-equivariant output poses. The GECN method represents possible solutions for these issues.

GECNs differs from the original capsule networks in three important parts. First, they have group capsules as intermediate feature representations, a specialized kind of capsules where pose vectors are elements of a group $(G, \circ)$ (cf. Section 3.2). Given this special representation, GECNs define a general scheme of dynamic routing by agreement algorithms for which can be shown that, under certain conditions, equivariance and invariance properties under transformations from $\mathcal{G}$ are mathematically guaranteed. Second, they propose a way to aggregate over local receptive fields on vector field inputs with changing poses, without losing the guaranteed properties (cf. Section 6.4). Third, they combine capsule networks with group convolutions and leverage the group capsule information to obtain convolutional neural networks that inherit the guaranteed equi- and invariances, as well as producing disentangled representations (cf. Section 6.4.2).

6.2 Related Work

Different ways to provide deep neural networks with specific equivariance properties have been introduced. One way is to share weights over differently rotated filters or augment the input by transformations [YQQ+17; WHS18]. A related but more general set of methods are the group convolutional networks [CW16; DDK16] and its applications like Spherical CNNs in $SO(3)$ [CGK+18; EAM+18] and Steerable CNNs in $SO(2)$ [CW17], which both result in special convolution realizations. The difference between group convolution and capsule networks is the data representation. Group convolutions produce feature maps that are defined for each element (or a dense sampling) of the group while GECNs produce a set of explicit group elements, describing object poses.

Capsule networks were introduced by [HKW11]. Lately, dynamic routing algorithms for capsule networks have been proposed [SFH17; HSF18]. Our work builds upon their methods and vision for capsule networks, as well as connect them to group convolutional networks.

Further methods include harmonic networks [WGT+17], which use circular harmonics as a basis for filter sets, and vector field networks [MVK+17]. These methods focus on two-dimensional rotational equivariance. While we chose an experiment which is similar to their approaches, our work aims to build a more general framework for different groups and disentangled representations.

Concurrent to or after the publication of GECNs, several related method have been proposed. The line of group equivariant networks was extended to general surfaces by methods that achieve gauge equivariance [CWK+19]. Recently, it was brought together with GNNs [WEH20; HWC+21], providing equivariant kernels on geometric graphs, utilizing parallel transport (cf. Section 4.1.3). The topic of equivariant convolutions is tackled by a vast amount of research for which the reader is referred to exhaustive review literature [WFV+21].

Capsule network research has developed in different directions. They were extended to process 3D points [ZBD+19], extended to full autoencoders for geometric objects [KST+19], and several variations or alternatives of the routing algorithm were proposed [RLK20; PKK19; JLK19; DWG+20; MSC21]. Those advances follow the original route of capsule networks and do not come with guaranteed equivariance but aim to learn it from data.

6.3 Group Equivariant Capsule Networks

We begin with essential definitions for group capsule layers and the properties we aim to guarantee. Given a Lie group $(\mathcal{G}, \circ)$ (cf. Section 3.2), we formally describe a group capsule layer with m output capsules by a set of function tuples

$$\{(L_p^j(\mathbf{P}, \mathbf{a}), L_a^j(\mathbf{P}, \mathbf{a})) \mid j \in \{1, \dots, m\}\}. \tag{6.1}$$

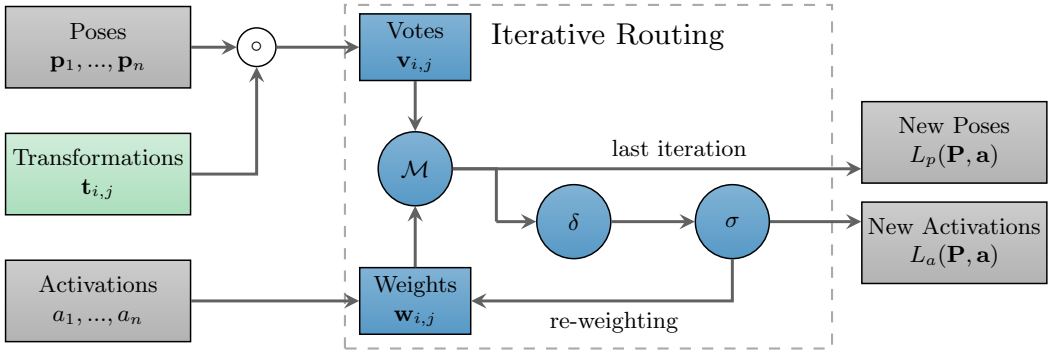


Figure 6.2: Data flow graph for the equivariant routing algorithm, the main building block of GECNs, consisting of ■ fixed function parts and the ■ trainable transformations $\mathbf{t}_{i,j}$. The gradient information is backpropagated along the inverse solid arrows in order to train an architecture consisting of multiple layers in an end-to-end fashion. To obtain gradients for poses $\mathbf{p}_1, ..., \mathbf{p}_n$, activations $a_1, ..., a_n$ and transformations $\mathbf{t}_{i,j}$, the fixed function realizations of $\mathcal{M}$, δ and σ need to be differentiable.

Here, the functions $L_p^j : \mathcal{G}^n \times \mathbb{R}^n \to \mathcal{G}$ compute the output pose vectors while functions $L_a^j : \mathcal{G}^n \times \mathbb{R}^n \to \mathbb{R}$ compute output activations, given input pose vectors $\mathbf{P} = (\mathbf{p}_1, ..., \mathbf{p}_n) \in \mathcal{G}^n$ and input activations $\mathbf{a} \in \mathbb{R}^n$. We denote the output poses and activations for all $1 \leq j \leq m$ as $L_p(\mathbf{P}, \mathbf{a})$ and $L_a(\mathbf{P}, \mathbf{a})$, respectively, by omitting the index j. The goal is to achieve invariance of activations and equivariance of pose vectors under actions of the group. Thus, we define those two properties for one single group capsule layer (cf. Figure 6.1). First, the function computing the output pose vectors of one layer is *equivariant* with respect to actions of the group if

$$L_p(\mathbf{g} \circ \mathbf{P}, \mathbf{a}) = \mathbf{g} \circ L_p(\mathbf{P}, \mathbf{a}), \quad \forall \mathbf{g} \in \mathcal{G}, \tag{6.2}$$

where $\mathbf{g} \circ \mathbf{P}$ denotes the element-wise group law application on all elements of $\mathbf{P}$. Second, the function computing activations of one layer is *invariant* with respect to actions of the group if

$$L_a(\mathbf{g} \circ \mathbf{P}, \mathbf{a}) = L_a(\mathbf{P}, \mathbf{a}), \quad \forall \mathbf{g} \in \mathcal{G}. \tag{6.3}$$

Since equivariance is transitive, it can be deducted that stacking layers that fulfill these properties preserves both properties for the combined operation. Therefore, if we apply a transformation from $\mathcal{G}$ on the input of a sequence of those layers (e.g. a whole deep network), the output activations remain the same and we produce output pose vectors which are transformed by the same transformation.

6.3.1 Group Capsule Layer

We define the group capsule layer functions as the output of an iterative routing by agreement, that follows an iterative scheme on elements of $\mathcal{G}$. The whole algorithm

Algorithm 8 Group capsule layer

Input: poses $\mathbf{P} = (\mathbf{p}_1, \ldots, \mathbf{p}_n) \in \mathcal{G}^n$, activations $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$
Trainable parameters: transformations $\mathbf{t}_{i,j}$
Output: poses $\hat{\mathbf{P}} = (\hat{\mathbf{p}}_1, \ldots, \hat{\mathbf{p}}_m) \in \mathcal{G}^m$, activations $\hat{\mathbf{a}} = (\hat{a}_1, \ldots, \hat{a}_m) \in \mathbb{R}^m$

$\mathbf{v}_{i,j} \leftarrow \mathbf{p}_i \circ \mathbf{t}_{i,j}$ for all input capsules i and output capsules j
$\hat{\mathbf{p}}_j \leftarrow \mathcal{M}((\mathbf{v}_{1,j}, \ldots, \mathbf{v}_{n,j}), \mathbf{a})$ $\forall j$
for r iterations **do**
  $w_{i,j} \leftarrow \sigma(-\delta(\hat{\mathbf{p}}_j, \mathbf{v}_{i,j})) \cdot a_i$ $\forall i, j$
  $\hat{\mathbf{p}}_j \leftarrow \mathcal{M}((\mathbf{v}_{1,j}, \ldots, \mathbf{v}_{n,j}), \mathbf{w}_{:,j})$ $\forall j$
end for
$\hat{a}_j \leftarrow \sigma(-\frac{1}{n} \sum_{i=1}^{n} \delta(\hat{\mathbf{p}}_j, \mathbf{v}_{i,j}))$ $\forall j$
Return $\hat{\mathbf{p}}_1, \ldots, \hat{\mathbf{p}}_m$, $\hat{\mathbf{a}}$

for one capsule layer, given a generic *weighted average operation* $\mathcal{M}$ and a *distance measure* δ, is shown in Algorithm 8. Additionally, Figure 6.2 shows the data flow.

Generally, votes are cast by applying trainable group elements $\mathbf{t}_{i,j}$ to the input pose vectors $\mathbf{p}_i$ (using the group law $\circ$), where i and j are the indices for input and output capsules, respectively. Then, the agreement is iteratively computed: First, new pose candidates $\hat{\mathbf{p}}_j$ are obtained by using the weighted average operator $\mathcal{M}$. Second, the negative, shifted δ-distance between votes $\mathbf{v}_{:,j}$ pose candidates $\hat{\mathbf{p}}_j$ are used for the weight update. After iteratively refining the poses and weights, the output agreement is computed by averaging negative distances between votes $\mathbf{v}_{:,j}$ and the new pose $\hat{\mathbf{p}}_j$. The functions σ can be chosen to be some scaling and shifting non-linearity, for example $\sigma(x) = \texttt{sigmoid}(\alpha \cdot x + \beta)$ with trainable α and β, as done by Hinton et al. [HSF18], or as softmax over the output capsule dimension, as done by Sabour et al. [SFH17].

Properties of $\mathcal{M}$ and δ For the following theorems we need to define specific properties of $\mathcal{M}$ and δ. The average operation $\mathcal{M} : \mathcal{G}^n \times \mathbb{R}^n \to \mathcal{G}$ should map n elements of the group $(\mathcal{G}, \circ)$, weighted by values $\mathbf{x} = (x_1, ..., x_n) \in \mathbb{R}^n$, to some kind of weighted average of those values in $\mathcal{G}$. Besides the closure, $\mathcal{M}$ should be *equivariant* under the group law, formally

$$\mathcal{M}(\mathbf{g} \circ \mathbf{P}, \mathbf{x}) = \mathbf{g} \circ \mathcal{M}(\mathbf{P}, \mathbf{x}), \quad \forall \mathbf{g} \in \mathcal{G}, \tag{6.4}$$

as well as invariant under permutations of the inputs. Further, the distance measure δ needs to be chosen so that transformations $\mathbf{g} \in \mathcal{G}$ are δ-distance preserving:

$$\delta(\mathbf{g} \circ \mathbf{g}_1, \mathbf{g} \circ \mathbf{g}_2) = \delta(\mathbf{g}_1, \mathbf{g}_2), \quad \forall \mathbf{g} \in \mathcal{G}. \tag{6.5}$$

Given these preliminaries, we can formulate the following two theorems.

Theorem 1. Let $\mathcal{M}$ be a weighted averaging operation that is equivariant under left-applications of $\mathbf{g} \in \mathcal{G}$ and let $\mathcal{G}$ be closed under applications of $\mathcal{M}$. Further, let δ be chosen so that all $\mathbf{g} \in \mathcal{G}$ are δ-distance preserving. Then, the function $L_p(\mathbf{P}, \mathbf{a}) = (\hat{\mathbf{p}}_1, \ldots, \hat{\mathbf{p}}_m)$, defined by Algorithm 8, is equivariant under left-applications of $\mathbf{g} \in \mathcal{G}$ on input pose vectors $\mathbf{P} \in \mathcal{G}^n$:

$$L_p(\mathbf{g} \circ \mathbf{P}, \mathbf{a}) = \mathbf{g} \circ L_p(\mathbf{P}, \mathbf{a}), \qquad \forall \mathbf{g} \in \mathcal{G}. \tag{6.6}$$

Proof. The theorem is shown by induction over the inner loop of the algorithm, using the equivariance of $\mathcal{M}$, preservation of δ and group properties. The initial step is to show equivariance of the pose vectors before the loop. After that we show that, given equivariant first pose vectors we receive invariant routing weights $\mathbf{w}$, which again leads to equivariant pose vectors in the next iteration.

Induction Basis. Let $\hat{\mathbf{p}}^0$, $\hat{\mathbf{p}}^0_{\mathbf{g}}$ be the first computed pose vectors (before the loop) for non-transformed and transformed inputs, respectively. The equivariance of those poses can be shown given associativity of group law, the equivariance of $\mathcal{M}$ and the invariance of activations coming from a previous layer (input activations $\mathbf{a}$ are equal for transformed and non transformed inputs) by

$$\begin{aligned}
\hat{\mathbf{p}}^0_{\mathbf{g}} &= \mathcal{M}(((\mathbf{g} \circ \mathbf{p}_1) \circ \mathbf{t}_1, \ldots, (\mathbf{g} \circ \mathbf{p}_n) \circ \mathbf{t}_n), \mathbf{a}^{\mathbf{g}}) \\
&= \mathcal{M}((\mathbf{g} \circ (\mathbf{p}_1 \circ \mathbf{t}_1)), \ldots, \mathbf{g} \circ (\mathbf{p}_n \circ \mathbf{t}_n)), \mathbf{a}) \\
&= \mathbf{g} \circ \mathcal{M}((\mathbf{p}_1 \circ \mathbf{t}_1, \ldots, \mathbf{p}_n \circ \mathbf{t}_n), \mathbf{a}) \\
&= \mathbf{g} \circ \hat{\mathbf{p}}^0.
\end{aligned}$$

Note that we show the result for one output capsule. Therefore, index j is constant and omitted. In addition, it can be seen that the computed votes also are equivariant.

Induction Step. Assuming equivariance of old pose vectors ($\mathbf{g} \circ \hat{\mathbf{p}}^m = \hat{\mathbf{p}}^m_{\mathbf{g}}$), we show equivariance of new pose vectors ($\mathbf{g} \circ \hat{\mathbf{p}}^{m+1} = \hat{\mathbf{p}}^{m+1}_{\mathbf{g}}$) after the next routing iteration. First we show that calculated weights $\mathbf{w}$ behave again invariant under input transformation $\mathbf{g}$. This follows directly from the induction assumption, δ-distance preservation, equivariance of the votes and the invariance of $\mathbf{a}$:

$$\begin{aligned}
w_i^{\mathbf{g}} &= \sigma(-\delta(\hat{\mathbf{p}}^m_{\mathbf{g}}, \mathbf{v}_i^{\mathbf{g}})) \cdot a_i \\
&= \sigma(-\delta(\mathbf{g} \circ \hat{\mathbf{p}}^m, \mathbf{g} \circ \mathbf{v}_i)) \cdot a_i \\
&= \sigma(-\delta(\hat{\mathbf{p}}^m, \mathbf{v}_i)) \cdot a_i \\
&= w_i.
\end{aligned}$$

Now we show equivariance of $\hat{\mathbf{p}}^{m+1}$, similarly to the induction basis, but using invariance of $\mathbf{w}$:

$$\begin{aligned}
\hat{\mathbf{p}}^{m+1}_{\mathbf{g}} &= \mathcal{M}(((\mathbf{g} \circ \mathbf{p}_1) \circ \mathbf{t}_1, \ldots, (\mathbf{g} \circ \mathbf{p}_n) \circ \mathbf{t}_n), \mathbf{w}^{\mathbf{g}}) \\
&= \mathcal{M}((\mathbf{g} \circ (\mathbf{p}_1 \circ \mathbf{t}_1)), \ldots, \mathbf{g} \circ (\mathbf{p}_n \circ \mathbf{t}_n)), \mathbf{w}) \\
&= \mathbf{g} \circ \mathcal{M}((\mathbf{p}_1 \circ \mathbf{t}_1, \ldots, \mathbf{p}_n \circ \mathbf{t}_n), \mathbf{w}) \\
&= \mathbf{g} \circ \hat{\mathbf{p}}^{m+1}.
\end{aligned}$$

□

Theorem 2. Given the same conditions as in Theorem 1. Then, the function $L_a(\mathbf{P}, \mathbf{a}) = (\hat{a}_1, \ldots, \hat{a}_m)$ defined by Algorithm 8 is invariant under joint left-applications of $\mathbf{g} \in \mathcal{G}$ on input pose vectors $\mathbf{P} \in \mathcal{G}^n$:

$$L_a(\mathbf{g} \circ \mathbf{P}, \mathbf{a}) = L_a(\mathbf{P}, \mathbf{a}), \quad \forall \mathbf{g} \in \mathcal{G}. \tag{6.7}$$

Proof. The result follows by applying Theorem 1 and the δ-distance preservation. Equality of a and $a_{\mathbf{g}}$ is shown using Theorem 1 and the δ-distance preservation of $\mathcal{G}$.

$$\begin{aligned} a_{\mathbf{g}} &= \sigma\left(-\frac{1}{2}\sum_{i=1}^{n} \delta(L_p(\mathbf{g}_l \circ \mathbf{P}, \mathbf{a}), \mathbf{g}_l \circ \mathbf{p}_i \circ \mathbf{g}_i)\right) \\ &= \sigma\left(-\frac{1}{2}\sum_{i=1}^{n} \delta(\mathbf{g}_l \circ L_p(\mathbf{P}, \mathbf{a}), \mathbf{g}_l \circ \mathbf{p}_i \circ \mathbf{g}_i)\right) \\ &= \sigma\left(-\frac{1}{2}\sum_{i=1}^{n} \delta(L_p(\mathbf{P}, \mathbf{a}), \mathbf{p}_i \circ \mathbf{g}_i)\right) \\ &= a. \end{aligned}$$

□

Again, the result is shown for one output capsule. Given these two theorems, we are able to build a deep group capsule network by composition of those layers, which guarantee global invariance in output activations and equivariance in pose vectors.

6.3.2 Examples for Applicable Groups

Given the proposed algorithm, $\mathcal{M}$ and δ have to be chosen based on the chosen group and element representations. In the following, different Lie groups that provide useful equivariances and can potentially be used in the proposed framework are discussed.

The two-dimensional rotation group $SO(2)$ The canonical application of the proposed framework on images is achieved by using the two-dimensional rotation group $SO(2)$, as discussed in Section 6.5. We chose to represent the elements of $\mathcal{G}$ as two-dimensional unit vectors, chose $\mathcal{M}$ as the renormalized Euclidean weighted mean and $\delta(\mathbf{p}_1, \mathbf{p}_2) = \frac{1}{2}(-\mathbf{p}_1^\top \mathbf{p}_2 + 1)$ as distance (cf. Section 6.5 for details). Then, δ is distance preserving and $\mathcal{M}$ is left-equivariant, assuming given poses do not add up to zero, which can be guaranteed through practical measures.

The three-dimensional rotation group $SO(3)$ Similarly, the framework can be applied to achieve invariance and equivariance with respect to the 3D rotation group $SO(3)$. The main challenge is to obtain an equivariant average operator $\mathcal{M}$ on the $SO(3)$ manifold, which can be subject to backpropagation. The realization using an IRLS Weiszfeld algorithm to compute $\mathcal{M}$ and the geodesic distance as δ is discussed in Section 6.6.

Translation group $(\mathbb{R}^n, +)$ An potentially interesting application of group capsules are translation groups. Essentially, a layer in the network is no longer evaluated for each spatial position, but rather predict which positions will be of special interest and may sparsely evaluate a feature map at those points. Therefore, the number of evaluations is heavily reduced, from number of output capsules times number of pixels in the feature map to only the number of output capsules. However, in our current architectures we would not expect that this construction would work, because the capsule network would not be able to receive gradients which point in the direction of good transformations $\mathbf{t}$. It would rather be a random search, until good translational dependencies between hierarchical parts of objects are found. Also, due to usually local filters in convolutions and sparse evaluations, the outputs would often be zero at points of interest. Choosing $\mathcal{M}$ and δ however is straight-forward: the Euclidean weighted average and the $l2$-distance fulfill all requirements.

Group products It should be noted that using the direct product of groups makes it possible to apply the framework on combination of groups. This is discussed in Section 6.4.4.

6.4 Vector Fields of Group Capsules

In the previous sections, the considered inputs and outputs of a capsule layer were simply sets of capsules, thus sets of tuples containing pose vectors and activation. In practice, however, the goal is to obtain an operator that receives and produces vector fields of capsules, usually defined over the input domain $\mathbb{R}^n$. To transfer the previously obtained results to that scenario, we first define these fields in general.

Given two groups $(\mathcal{H}, \circ_H)$, $(\mathcal{G}, \circ_G)$, a *capsule vector field* is a function $f : \mathcal{H} \to \mathcal{G}$, that maps group elements $\mathbf{h} \in \mathcal{H}$ to group elements $\mathbf{g} \in \mathcal{G}$. We say f is a $\mathcal{G}$-vector field over $\mathcal{H}$. We denote the space of all $\mathcal{G}$-vector fields over $\mathcal{H}$ as $\mathcal{F}(\mathcal{H}, \mathcal{G})$. We further assume that $(\mathcal{G}, \circ_G)$ acts on $(\mathcal{H}, \circ_H)$ with $\circ_G$. Then, we can define the action of $(\mathcal{G}, \circ_G)$ on a $\mathcal{G}$-vector field f over $\mathcal{H}$ as

$$(\mathbf{g} \circ_G f)(\mathbf{h}) = \mathbf{g} \circ_G f(\mathbf{g}^{-1} \circ_G \mathbf{h}). \tag{6.8}$$

Additionally, we define the action of $\mathbf{G}$ on an $\mathbb{R}$-vector field over $\mathcal{H}$ as

$$(\mathbf{g} \circ_G f)(\mathbf{h}) = f(\mathbf{g}^{-1} \circ_G \mathbf{h}), \tag{6.9}$$

as we consider the activations itself to be invariant to applications of $\mathcal{G}$. In practice, we will consider discrete vector fields instead of continuous ones. However, we do not restrict ourself to vector grids but assume our vector fields are defined for a finite set of elements $\mathbf{x} \in \mathcal{H}$.

Example Assuming we are only interested in rotation and translation, the input is an $SO(n)$-vector field over $\mathbb{R}^n$. If a rotation is applied to the input, not only the

input poses are rotated but also the position of the poses. This is described by the group $SO(n)$ acting on $SE(n)$, the semidirect product $SO(n) \ltimes (\mathbb{R}^n, +)$ between rotation and translation in n dimensions (cf. Section 3.2), in which $SO(n)$ acts on $\mathbb{R}^n$ as well. For elements $\mathbf{R} \in SO(n)$ and $(\mathbf{R}', \mathbf{x}') \in SE(n)$ (in appropriate group representations), where $\mathbf{x}'$ represents the translational and $\mathbf{R}'$ the rotational part, the action of $SO(n)$ on $SE(n)$ can be described as

$$\mathbf{R} \circ (\mathbf{R}', \mathbf{x}') = (\mathbf{R} \cdot \mathbf{R}', \mathbf{R}\mathbf{x}'), \tag{6.10}$$

according to the action of semidirect products (cf. Section 3.2). Thus, when the input of an $SO(n)$ capsule layer, an $SO(n)$-vector field over $\mathbb{R}^n$, is transformed by rotation $\mathbf{g} \in SO(n)$, not only the deeper $SO(n)$ pose vectors change accordingly but the rotations also induce changes in positions of those pose vectors in $\mathbb{R}^n$.

Vector Field Capsules Layer, Part I We define a *$\mathcal{G}$-vector field capsule layer* over a continuous domain $\mathcal{H}$ as a function $C : \mathcal{F}(\mathcal{H}, \mathcal{G}) \times \mathcal{F}(\mathcal{H}, \mathbb{R}) \to \mathcal{F}(\mathcal{H}, \mathcal{G}) \times \mathcal{F}(\mathcal{H}, \mathbb{R})$, mapping from and to $\mathcal{G}$-vector fields over $\mathcal{H}$ (pose vector field) and $\mathbb{R}$-vector fields over $\mathcal{H}$ (activation field). We use C_p to denote the first element of C, the pose vector field, and C_a to denote the second element, the activation field. Similarly, the inputs to C are denoted f_p and f_a for $\mathcal{G}$-vector fields and $\mathbb{R}$-vector fields, respectively.

Since $\mathcal{G}$ acts on $\mathcal{H}$ we cannot simply consider equivariance and invariance for each element of $\mathcal{H}$ individually but need to extend the definition. Instead, we want the pose vector field C_p to be *equivariant* with respect to actions $\mathbf{g} \in \mathcal{G}$, meaning

$$C_p(\mathbf{g} \circ_G f_p, \mathbf{g} \circ_G f_a) = \mathbf{g} \circ_G C_p(f_p, f_a). \tag{6.11}$$

Note that the action of $\mathbf{g}$ on C_p and f_p is defined as given in Equation (6.8), as action on the output and inverse action on the input, while the action of $\mathbf{g}$ on f_a is defined as in Equation (6.9), as only the reverse action on the input. Further, we want the activation field function C_p to be *equivariant* with respect to action of $\mathbf{g} \in \mathcal{G}$, meaning

$$C_a(\mathbf{g} \circ_G f_p, \mathbf{g} \circ_G f_a) = \mathbf{g} \circ_G C_a(f_p, f_a). \tag{6.12}$$

Due to the transfer to vector fields, we also require equivariance in activation fields, instead of simple invariance in activations. Here, with respect to a transformation $\mathbf{g} \in \mathcal{G}$, the function needs to be equivariant in $\mathcal{H}$ and invariant in $\mathbb{R}$ due to $\mathbf{g}$ not acting on activations.

Before defining how C_p and C_a are computed and showing that the properties of Equation (6.11) and (6.12) are fulfilled, we take a closer look at how capsule networks aggregate poses over spatial receptive fields.

6.4.1 Spatial Aggregation with Group Capsules Fields

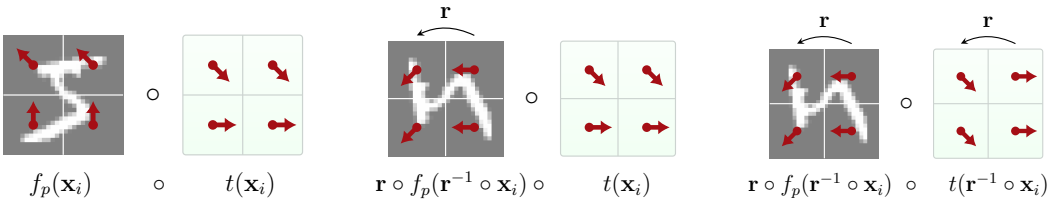
We point out an issue regarding equivariance in the original capsule formulation by Sabour et al. [SFH17], before proposing a solution to make our vector field capsule network equivariant.

In these considerations, we only take a closer look at one receptive field, aggregating capsules lying at different positions $\mathbf{x}$ within the receptive field into capsules on a single output position. In the original capsule networks by Sabour et al. [SFH17], the trainable transformations $\mathbf{T}$ lie in a fixed kernel window, where entries are defined for fixed positions of the local receptive field over $\mathbb{R}^n$. We bring them to our vector field framework by expressing them as a $GL(\mathbb{R}^n)$-vector field $t : \mathbb{R}^n \to GL(\mathbb{R}^n)$ over $\mathbb{R}^n$. Further, we consider the vector field of votes, which are computed at each input position $\mathbf{x} \in \mathbb{R}^n$ of the local receptive field as $V(f_p)(\mathbf{x}) = f_p(\mathbf{x}) \circ t(\mathbf{x})$. It can be seen that in order for the full operator to be equivariant with respect to an input transformation $\mathbf{R} \in GL(\mathbb{R}^n)$, the computation of vote vector fields has to be equivariant. However, we can derive that V is not equivariant with respect to transforming the input vector field f_p:

$$\begin{aligned} V(\mathbf{R} \circ f_p)(\mathbf{x}) &= \mathbf{R} \circ f_p(\mathbf{R}^{-1}\mathbf{x}) \circ t(\mathbf{x}) \\ &\neq \mathbf{R} \circ f_p(\mathbf{R}^{-1}\mathbf{x}) \circ t(\mathbf{R}^{-1}\mathbf{x}) \\ &= \mathbf{R} \circ V(f_p)(\mathbf{x}). \end{aligned} \tag{6.13}$$

Therefore, reasonably assuming that the transformation group of output capsules acts on $\mathbb{R}^n$, the composition of pose vectors and trainable transformations to compute the votes changes with input transformation. As a result, the votes (and following pose averages over the local receptive field) are no longer equivariant and the computed activations are no longer invariant. A visual example of the described issue (and a counterexample for equivariance) for an aggregation over a 2×2 block and $\mathcal{G} = SO(2)$ can be found in Figures 6.3a and 6.3b.

Pose-aligning transformation kernels We can derive from Equation (6.13) that to achieve equivariant computation of votes over vector fields, we need to counteract the induced action of $\mathcal{G}$ on the vector space group $\mathcal{H}$ in the t-kernel. This is not trivially done as the transformations $\mathbf{R} \in GL(\mathbb{R}^n)$ from Equation (6.13), which we will call $\mathbf{g} \in \mathcal{G}$ now as we move back to our $\mathcal{G}$-capsules, are not known, since we only see the already transformed input. However, in GECNs we can utilize that we already have an estimate of input pose at each position in the vector field, given by the previous layers output capsules. Thus, those poses can be utilized to canonicalize the field t. According to our group action, we can compute $\bar{\mathbf{g}} \circ t(\mathbf{x}) = t(\bar{\mathbf{g}}^{-1} \circ \mathbf{x})$, given a pose $\bar{\mathbf{g}}$ from the input vector field. In practice, this means we need to switch from a discrete to a continuous convolution kernel $t(\cdot)$ (cf. Section 4.1.3), which can be sampled at arbitrary points $\mathbf{x}$.



(a) Non-rotated input and poses

(b) Rotated input, false matching

(c) Pose-aligned t-kernels

Figure 6.3: Example for the spatial aggregation of a 2×2 block of $SO(2)$ capsules [LFL18]. Figure (a) shows the behavior for non-rotated inputs. The resulting votes have full agreement, pointing to the top. Figure (b) shows the behavior when rotating the input by $\pi/2$, where we obtain a different element-wise matching of pose vectors $f_p(\cdot)$ and transformations $t(\cdot)$, depending on the input rotation. Figure (c) shows the behavior with the proposed kernel alignment. It can be seen that f_p and t match again and the result is the same full pose agreement as in (a) with equivariant mean pose, pointing to the left.

There are multiple candidates for the source pose that is used for canonicalization. If there is a single pose in the center of the current receptive field, it can be taken directly. If there is no central one, we can compute $\bar{\mathbf{g}} = \mathcal{M}(\mathbf{p}_1, \dots, \mathbf{p}_n, \mathbf{1})$ or $\bar{\mathbf{g}} = \mathcal{M}(\mathbf{p}_1, \dots, \mathbf{p}_n, \mathbf{a})$, a mean pose vector for the current receptive field, given local pose vectors $\mathbf{p}_1, \dots, \mathbf{p}_n$ and optionally activations $\mathbf{a}$. The appropriate average operator $\mathcal{M}$ is already required for equivariant routing anyway, so we can assume to have access to one. The estimated poses $\bar{\mathbf{g}}$ of non-transformed inputs and inputs transformed by $\mathbf{g} \in \mathcal{G}$ differ exactly by $\mathbf{g}$, thus $\bar{\mathbf{g}} = \mathbf{g} \circ \mathbf{p}$ for a fixed $\mathbf{p} \in \mathcal{G}$. This follows from equivariance of $\mathcal{M}$, invariance of $\mathcal{M}$ under permutation, and from the equivariance property of previous layers, meaning that the rotation applied to the input directly translates to the pose vectors in deeper layers, making these pose estimations equivariant, too.

Thus, we can apply the inverse pose $\bar{\mathbf{g}}^{-1} = \mathbf{p}^{-1} \circ \mathbf{g}^{-1}$ to the input positions $\mathbf{x}$ of t and calculate the votes in one receptive field as $V(f_p)(\mathbf{x}) = p(\mathbf{x}) \circ t(\bar{\mathbf{g}}^{-1} \circ \mathbf{x})$. We can see that now, if a transformation $\mathbf{g}$ acts on the input f_p, equivariance holds:

$$\begin{aligned} V(\mathbf{g} \circ f_p)(\mathbf{x}) &= \mathbf{g} \circ f_p(\mathbf{g}^{-1} \circ \mathbf{x}) \circ t(\bar{\mathbf{g}}^{-1} \circ \mathbf{x}) \\ &= \mathbf{g} \circ f_p(\mathbf{g}^{-1} \circ \mathbf{x}) \circ t(\mathbf{p}^{-1} \circ \mathbf{g}^{-1} \circ \mathbf{x}) \\ &= \mathbf{g} \circ V(f_p)(\mathbf{x}), \end{aligned} \tag{6.14}$$

as also illustrated in the example in Figure 6.3c. Since $f_p(\cdot)$ and $t(\cdot)$ are combined in a way that is independent from $\mathbf{g}$, applying a transformation $\mathbf{g}$ to the input results in a set of votes that is the same as without input transformation, just rotated by $\mathbf{g}$. Note that $\bar{\mathbf{p}}^{-1} \in \mathcal{G}$ is constant for all input transformations and we have no notion of what a default pose is anyway. Therefore, this does not lead to further issues and $\mathbf{p}^{-1}$ can just be considered as part of t. In practice, we use a two-layer MLP to calculate

$t(\cdot)$, which maps the a local receptive field position $\mathbf{x}$ to $n \cdot m$ transformations (for n input capsules per position and m output capsules). The proposed method can be understood as pose-aligning a trainable, continuous kernel window, which generates transformations from $\mathcal{G}$. It is similar to continuous convolution techniques applied for data aggregation in irregular domains using GNNs, as discussed in Section 4.1.3.

Vector Field Capsule Layer, Part II What is left is to define the computation of the $\mathcal{G}$-vector field capsule layer that produces whole vector fields defined on positions $\mathbf{x} \in \mathbb{R}^n$, given input capsules $(f_p(\mathbf{y}), f_a(\mathbf{y}))$ at positions $\mathbf{y} \in \mathbb{R}^n$. For a given output position $\mathbf{x}$, we denote its receptive field as $\mathcal{N}(\mathbf{x}) \subset \mathbb{R}^n$, containing all input positions $\mathbf{y} \in \mathbb{R}^n$, that lie within a predefined radius. Further, let DR_p and DR_a denote the dynamic routing after computing votes (that is, lines 2 – 8 of Algorithm 8 with sets of tuples, containing votes and activations, as input). Then, we define the functions C_p and C_a, computing a whole vector field of capsules, as

$$C_{p/a}(f_p, f_a)(\mathbf{x}) = \mathrm{DR}_{p/a}\Big(\big\{(V(f_p)(\mathbf{x}, \mathbf{y}), f_a(\mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\big\}\Big), \tag{6.15}$$

with

$$V(f_p)(\mathbf{x}, \mathbf{y}) = f_p(\mathbf{y}) \circ t(f_p(\mathbf{y})^{-1} \circ (\mathbf{y} - \mathbf{x})) \tag{6.16}$$

computing the votes as described above, for each pair of input $\mathbf{y}$ and output $\mathbf{x}$ with $\mathbf{y} \in \mathcal{N}(\mathbf{x})$. Thus, the output capsules at a position $\mathbf{x}$ are computed by gathering the equivariant votes and activations over the receptive field around $\mathbf{x}$ and apply dynamic routing to them. In case $f_p(\mathbf{y})$ in Equation (6.16) is not defined, we use the equivariant mean over the whole receptive field, as described.

The proposed operator fulfills the wanted properties of equivariance as defined in Equations (6.11) and (6.12) up to an error introduced by sampling of the vector fields, which is shown by proving the following theorem.

Theorem 3. Given pose and activation vector fields f_p and f_a for group $\mathcal{G}$ as input. We assume that the weighted average of votes in different samplings of the same receptive fields is approximately equal, which means that

$$\begin{aligned} & \mathcal{M}\Big(\big\{(v(\mathbf{x}, \mathbf{y}), f_a(\mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{g}^{-1} \circ \mathbf{x})\big\}\Big) \\ \approx \quad & \mathcal{M}\Big(\big\{(v(\mathbf{g}^{-1} \circ \mathbf{x}, \mathbf{g}^{-1} \circ \mathbf{y}), f_a(\mathbf{g}^{-1} \circ \mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\big\}\Big) \end{aligned} \tag{6.17}$$

holds. Then, the vector field capsule layer $C_{p/a}(f_p, f_a)$ is approximately equivariant in poses and activations:

$$C_p(\mathbf{g} \circ_G f_p, \mathbf{g} \circ_G f_a) \approx \mathbf{g} \circ_G C_p(f_p, f_a), \text{ and} \tag{6.18}$$

$$C_a(\mathbf{g} \circ_G f_p, \mathbf{g} \circ_G f_a) \approx \mathbf{g} \circ_G C_a(f_p, f_a). \tag{6.19}$$

Proof. We first show that the computation of votes behaves equivariant, that is

$$V(\mathbf{g} \circ f_p)(\mathbf{x}, \mathbf{y}) = (\mathbf{g} \circ V)(f_p)(\mathbf{x}, \mathbf{y}) = \mathbf{g} \circ V(f_p)(\mathbf{g}^{-1} \circ \mathbf{x}, \mathbf{g}^{-1} \circ \mathbf{y}), \tag{6.20}$$

given the construction showed in this section:

$$\begin{aligned} V(\mathbf{g} \circ f_p)(\mathbf{x}, \mathbf{y}) &= \mathbf{g} \circ f_p(\mathbf{g} \circ^{-1} \mathbf{y}) \circ t((\mathbf{g} \circ f_p(\mathbf{g}^{-1} \circ \mathbf{y}))^{-1} \circ (\mathbf{y} - \mathbf{x})) \\ &= \mathbf{g} \circ f_p((\mathbf{g}^{-1} \circ \mathbf{y}) \circ t(f_p(\mathbf{g}^{-1} \circ \mathbf{y})^{-1} \circ \mathbf{g}^{-1} \circ (\mathbf{y} - \mathbf{x})) \\ &= \mathbf{g} \circ f_p((\mathbf{g}^{-1} \circ \mathbf{y}) \circ t(f_p(\mathbf{g}^{-1} \circ \mathbf{y})^{-1} \circ ((\mathbf{g}^{-1} \circ \mathbf{y}) - (\mathbf{g}^{-1} \circ \mathbf{x}))) \\ &= \mathbf{g} \circ V(f_p)(\mathbf{g}^{-1} \circ \mathbf{x}, \mathbf{g}^{-1} \circ \mathbf{y}). \end{aligned}$$

The individual steps follow from applying the definitions of actions on vector fields and the property of group actions on groups as given in Equation (3.8) in Section 3.2.

Given that, we show equivariance of C_p and C_a. From Theorem 1 and 2 we already know that the dynamic routing DR is equivariant/invariant, given equivariant votes as input. Thus we can follow approximate equivariance (depending on sampling quality) with

$$\begin{aligned} & C_p(\mathbf{g} \circ f_p, \mathbf{g} \circ f_a)(\mathbf{x}) \\ &= \mathrm{DR}_{p/a}\Big(\Big\{(V(\mathbf{g} \circ f_p)(\mathbf{x}, \mathbf{y}), f_a(\mathbf{g}^{-1} \circ \mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\Big\}\Big) && \text{(Def.)} \\ &= \mathrm{DR}_{p/a}\Big(\Big\{(\mathbf{g} \circ V(f_p)(\mathbf{g}^{-1} \circ \mathbf{x}, \mathbf{g}^{-1} \circ \mathbf{y}), f_a(\mathbf{g}^{-1} \circ \mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\Big\}\Big) && \text{(V equiv.)} \\ &= \mathbf{g} \circ \mathrm{DR}_{p/a}\Big(\Big\{(V(f_p)(\mathbf{g}^{-1} \circ \mathbf{x}, \mathbf{g}^{-1} \circ \mathbf{y}), f_a(\mathbf{g}^{-1} \circ \mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\Big\}\Big) && \text{(DR equiv.)} \\ &\approx \mathbf{g} \circ \mathrm{DR}_{p/a}\Big(\Big\{(V(f_p)(\mathbf{x}, \mathbf{y}), f_a(\mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{g}^{-1} \circ \mathbf{x})\Big\}\Big) && \text{(Assumption)} \\ &= \mathbf{g} \circ C_p(f_p, f_a)(\mathbf{g}^{-1} \circ \mathbf{x}) && \text{(Def.)} \\ &= (\mathbf{g} \circ C_p(f_p, f_a))(\mathbf{x}). \end{aligned}$$

Similarly, we can follow the equivariance of the activation map.

$$\begin{aligned} & C_a(\mathbf{g} \circ f_p, \mathbf{g} \circ f_a)(\mathbf{x}) \\ &= \mathrm{DR}_{p/a}\Big(\Big\{(V(\mathbf{g} \circ f_p)(\mathbf{x}, \mathbf{y}), f_a(\mathbf{g}^{-1} \circ \mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\Big\}\Big) && \text{(Def.)} \\ &= \mathrm{DR}_{p/a}\Big(\Big\{(\mathbf{g} \circ V(f_p)(\mathbf{g}^{-1} \circ \mathbf{x}, \mathbf{g}^{-1} \circ \mathbf{y}), f_a(\mathbf{g}^{-1} \circ \mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\Big\}\Big) && \text{(V equiv.)} \\ &= \mathrm{DR}_{p/a}\Big(\Big\{(V(f_p)(\mathbf{g}^{-1} \circ \mathbf{x}, \mathbf{g}^{-1} \circ \mathbf{y}), f_a(\mathbf{g}^{-1} \circ \mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{x})\Big\}\Big) && \text{(DR inv.)} \\ &\approx \mathrm{DR}_{p/a}\Big(\Big\{(V(f_p)(\mathbf{x}, \mathbf{y}), f_a(\mathbf{y})) \mid \mathbf{y} \in \mathcal{N}(\mathbf{g}^{-1} \circ \mathbf{x})\Big\}\Big) && \text{(Assumption)} \\ &= C_a(f_p, f_a)(\mathbf{g}^{-1} \circ \mathbf{x}) && \text{(Def.)} \\ &= (\mathbf{g} \circ C_a(f_p, f_a))(\mathbf{x}). \end{aligned}$$

□

The results from Theorem 3 complete our equivariant vector field capsule layer and we can use it to build approximately equivariant architectures on vector field inputs. It should be noted that if we down-sample the fields in each layer, consequently reaching a vector field with one sample point in the end, the equivariance

property of C_a again reduces to invariance with respect to input transformation. The effects of re-sampling errors in practice will be detailed for both full architecture realizations in Sections 6.5 and Section 6.6, when analyzing the pose vector quality.

6.4.2 Group Capsules and Group Convolutions

This section will discuss the combination of the previously described equivariant capsule networks with group convolutional networks. A variant of group convolutional networks over multiple groups is proposed, where only one group is densely sampled and the other is sparsely evaluated using the group capsules. It is shown that the architecture inherits the vector field equivariance under the group law from the capsule part of the network.

In comparison to group capsule networks alone, which have limited expressibility due to representations being restricted to weighted elements of a group, we gain expressiveness through the use of arbitrary feature maps and anisotropic filters. Thus, the extension described in this section improves the qualitative performance of our capsule networks and is still able to provide disentangled information. In the following, group convolutions are shortly introduced before the combined method is detailed.

Group Convolution

Group convolutions are a generalized convolution operator defined for elements of a group. For a Lie group $(\mathcal{G}, \circ)$ the *group convolution* is defined as

$$[f \star \psi](\mathbf{g}) = \int_{\hat{\mathbf{g}} \in \mathcal{G}} f(\hat{\mathbf{g}}) \psi(\mathbf{g}^{-1} \circ \hat{\mathbf{g}}) \, d\hat{\mathbf{g}}, \tag{6.21}$$

which behaves equivariant under applications of the group law $\circ$ [CW16; CGK+18]. The authors showed that they can be used to build group equivariant convolutional networks that apply a stack of those layers to obtain an equivariant architecture. However, compared to capsule networks, they do not directly compute disentangled representations, which we aim to achieve through the combination with capsule networks.

Additionally, we consider the convolution over two groups $(\mathcal{H}, \circ_H)$ and $(\mathcal{G}, \circ_G)$, where $(\mathcal{G}, \circ_G)$ acts on $(\mathcal{H}, \circ_H)$ according to the action of the semi-direct product (cf. Section 3.2 and Section 6.4). The convolution is given as

$$[f \star \psi](\mathbf{h}, \mathbf{g}) = \int_{\hat{\mathbf{h}} \in \mathcal{H}} \int_{\hat{\mathbf{g}} \in \mathcal{G}} f(\hat{\mathbf{h}}, \hat{\mathbf{g}}) \psi((\mathbf{g}^{-1} \circ_{\mathcal{G}} \mathbf{h}^{-1}) \circ_{\mathcal{H}} (\mathbf{g}^{-1} \circ_{\mathcal{G}} \hat{\mathbf{h}}), \mathbf{g}^{-1} \circ_{\mathcal{G}} \hat{\mathbf{g}}) \, d\hat{\mathbf{g}} d\hat{\mathbf{h}}, \tag{6.22}$$

where feature maps and filters are defined densely over both groups. In the methods presented later, $(\mathcal{H}, \circ_{\mathcal{H}})$ will be a translation group over $\mathbb{R}^n$ and $(\mathcal{G}, \circ_{\mathcal{G}})$ will be a rotation group $SO(n)$.

Sparse Group Convolution

In this section, the sparse group convolution will be presented, which combines group capsule networks with group convolution. An intuition for the proposed method is to interpret the pose vector fields throughout a capsule network as a sparse tree representation of a group convolution in indices $\mathbf{g} \in \mathcal{G}$. The output vector field of a group convolution layer $[f \star \psi](\mathbf{h}, \mathbf{g})$ over groups $\mathcal{H}$ and $\mathcal{G}$, as given in Equation (6.22), is defined densely for each element $\mathbf{g} \in \mathcal{G}$. In contrast, the output of the group capsule layer is a set of tuples $(\mathbf{g}, a)$ with group element $\mathbf{g}$ (pose vector), which can be interpreted as sparse indices for the output of a group convolution layer in dimension $\mathcal{G}$. Instead of evaluating the convolution densely for $\mathcal{H}$ and $\mathcal{G}$, we only evaluate it densely for $\mathcal{H}$ and for each $\mathbf{g} \in \mathcal{G}$ given to us by the capsule network. In this context, the pose $\mathbf{g}$, computed using routing by agreement from poses of layer l, serves as the hypothesis for the relevance of the group convolution feature map content of layer l at position $\mathbf{g}$.

In the following, an additional vector field is introduced, a feature map $f : \mathcal{H} \to \mathbb{R}^d$, which is acted upon by $\mathbf{g} \in \mathcal{G}$ according to

$$\mathbf{g} \circ f = f(\mathbf{g}^{-1} \circ \mathbf{x}). \tag{6.23}$$

Then, given pose vector field f_p from a $\mathcal{G}$-vector field capsule layer, a feature vector field $f : \mathcal{H} \to \mathbb{R}$, and filter $\psi : \mathcal{H} \to \mathbb{R}$ defined on a group $\mathcal{H}$, which is a $\mathcal{G}$-set (thus, there exists an action $\circ_{\mathcal{G}}$ of $\mathcal{G}$ on $\mathcal{H}$), we define the *sparse group convolution operator* $[f \star (f_p \circ_{\mathcal{G}} \psi)]$ as

$$[f \star (f_p \circ_{\mathcal{G}} \psi)](\mathbf{x}) = \int_{\mathbf{h} \in \mathcal{H}} f(\mathbf{h}) \psi((f_p(\mathbf{x})^{-1} \circ_{\mathcal{G}} \mathbf{x}^{-1}) \circ_{\mathcal{H}} (f_p(\mathbf{x})^{-1} \circ_{\mathcal{G}} \mathbf{h}))\, d\mathbf{h}. \tag{6.24}$$

The operator uses poses obtained by the capsule network to index the group convolution over $\mathcal{H}$ with a sparse set of elements $\mathbf{g} \in \mathcal{G}$. Note that this definition omits indices and sums for multiple filters, feature maps and pose maps, showing only one pose map, one feature map, and one filter. In practice, the equation within the integral is evaluated for larger stacks of poses, filters and features, and summed over them. The specific dimensions are treated as hyper-parameters. Specifically, we also sum over multiple poses $\mathbf{g} \in \mathcal{G}$ given at a position $\mathbf{h} \in \mathcal{H}$, which is the replacement for the integral over group $\mathcal{G}$. We now show that the operator fulfills the vector field equivariance property defined in Equation (6.12). In practice, this theorem will allow us to sample from spatial convolution over a vector field defined in $\mathcal{H} = \mathbb{R}^n$ by letting the capsule poses from a different group $\mathcal{G} = SO(n)$ act on the filter, and thus keeping invariance with respect to actions of group $SO(n)$ on $\mathbb{R}^n$.

Theorem 4. Given pose vector field f_p as output of a group capsule layer for group $\mathcal{G}$, a feature vector field $f : \mathcal{H} \to \mathbb{R}$, and filter $\psi : \mathcal{H} \to \mathbb{R}$ defined on a group $\mathcal{H}$, which is a $\mathcal{G}$-set. Then, the group convolution $[f \star (f_p \circ_{\mathcal{G}} \psi)]$ is equivariant under joint left-action of $\mathbf{g} \in \mathcal{G}$ on pose vector field f_p and feature field f:

$$[(\mathbf{g} \circ_{\mathcal{G}} f) \star ((\mathbf{g} \circ_{\mathcal{G}} f_p) \circ_{\mathcal{G}} \psi)](\mathbf{x}) = \mathbf{g} \circ_{\mathcal{G}} [f \star (f_p \circ_{\mathcal{G}} \psi)](\mathbf{x}) = [f \star (f_p \circ_{\mathcal{G}} \psi)](\mathbf{g}^{-1} \circ_{\mathcal{G}} \mathbf{x}). \tag{6.25}$$

Proof. In order to prove the result we first apply the definitions of group action on the feature map $(\mathbf{g} \circ_G f)(\mathbf{h}) = f(\mathbf{g}^{-1} \circ \mathbf{h})$, filter $(\mathbf{g} \circ_G \psi)(\mathbf{h}) = \psi(\mathbf{g}^{-1} \circ_G \mathbf{h})$, and pose field $(\mathbf{g} \circ_G f_p)(\mathbf{h}) = \mathbf{g} \circ_G f_p(\mathbf{g}^{-1} \circ_G \mathbf{h})$. Then, the group property $(\mathbf{g}_1 \circ_G \mathbf{g}_2)^{-1} = \mathbf{g}_2^{-1} \circ_G \mathbf{g}_1^{-1}$ (using existence of inverse and neutral element properties of groups) and a substitution $\mathbf{h} \to \mathbf{g} \circ_{\mathcal{G}} \mathbf{h}$ are used, to obtain the result. We use $\circ = \circ_{\mathcal{G}}$.

$$
\begin{aligned}
&[(\mathbf{g} \circ f) \star ((\mathbf{g} \circ f_p) \circ \psi)](\mathbf{x}) \\
&= \int_{\mathbf{h} \in \mathcal{H}} f(\mathbf{g}^{-1} \circ \mathbf{h}) \psi(((\mathbf{g} \circ f_p)(\mathbf{x})^{-1} \circ \mathbf{x}^{-1}) \circ_{\mathcal{H}} ((\mathbf{g} \circ f_p)(\mathbf{x})^{-1} \circ \mathbf{h}))\, d\mathbf{h} \\
&= \int_{\mathbf{h} \in \mathcal{H}} f(\mathbf{g}^{-1} \circ \mathbf{h}) \psi(((\mathbf{g} \circ f_p(\mathbf{g}^{-1} \circ \mathbf{x}))^{-1} \circ \mathbf{x}^{-1}) \circ_{\mathcal{H}} ((\mathbf{g} \circ f_p(\mathbf{g}^{-1} \circ \mathbf{x}))^{-1} \circ \mathbf{h}))\, d\mathbf{h} \\
&= \int_{\mathbf{h} \in \mathcal{H}} f(\mathbf{g}^{-1} \circ \mathbf{h}) \psi((f_p(\mathbf{g}^{-1} \circ \mathbf{x})^{-1} \circ \mathbf{g}^{-1} \circ \mathbf{x}^{-1}) \circ_{\mathcal{H}} (f_p(\mathbf{g}^{-1} \circ \mathbf{x})^{-1} \circ \mathbf{g}^{-1} \circ \mathbf{h}))\, d\mathbf{h} \\
&= \int_{\mathbf{h} \in \mathcal{H}} f(\mathbf{h}) \psi((f_p(\mathbf{g}^{-1} \circ \mathbf{x})^{-1} \circ (\mathbf{g}^{-1} \circ \mathbf{x})^{-1}) \circ_{\mathcal{H}} (f_p(\mathbf{g}^{-1} \circ \mathbf{x})^{-1} \circ \mathbf{h}))\, d\mathbf{h} \\
&= [f \star (f_p \circ \psi)](\mathbf{g}^{-1} \circ \mathbf{x})
\end{aligned}
$$

The key insight is that if f_p contains the correct pose for each $\mathbf{x}$, each filter ψ can be oriented correctly. Note that in practice, if there are re-sampling errors and the results from Theorem 3 only hold approximately, the action $\mathbf{g}$ on f_p is only realized approximately as well, that is $(\mathbf{g} \circ_G f_p)(\mathbf{h}) \approx \mathbf{g} \circ_G f_p(\mathbf{g}^{-1} \circ_G \mathbf{h})$. In this case, the equality between line 2 and line 3 is weakened. □

Theorem 4 tells us that it is possible to compute convolution with feature maps over a translation group $\mathcal{H}$ using a sparse set of anisotropic, continuous convolution operators ψ that are parameterized by elements $\mathbf{g} \in \mathcal{G}$ and obtain features, which are invariant to actions of $\mathcal{G}$. The process can be performed over a sparse set of group elements without eliminating invariance. Additionally, the agreement values from capsules can be used to dampen or amplify the resulting feature map contents, bringing pose covariances, which have been captured by the capsule network, into consideration. Figure 6.4 shows a scheme of the resulting full architecture.

In practice, ψ is realized as localized continuous convolution kernel, for which we use a SplineConv filter (cf. Section 4.1.4). Other operators for continuous convolution or grid-warp approaches [HV17] can be used as well. The kernel is zero outside of a given interval around $\mathbf{x}$ and always centered around $\mathbf{x}$, the current point of evaluation. Thus, in Equation (6.24), the $\mathbf{x}$ in ψ is always zero and the first part of the index vanishes so that the capsule poses only need to be applied to $\mathbf{h}$. According to Equation (6.24), calculation of the convolutions can be performed by applying the inverse transformation to the local input coordinates using the capsule's pose vector, as it is pictured in Figure 6.5.

Further, we can use the iteratively computed activation fields f_a from the routing algorithm to perform *pooling by agreement* on the feature maps: instead of using max or uniform average operators for spatial aggregation, the feature map content can be aggregated as weighted average, using the weights obtained in dynamic routing.

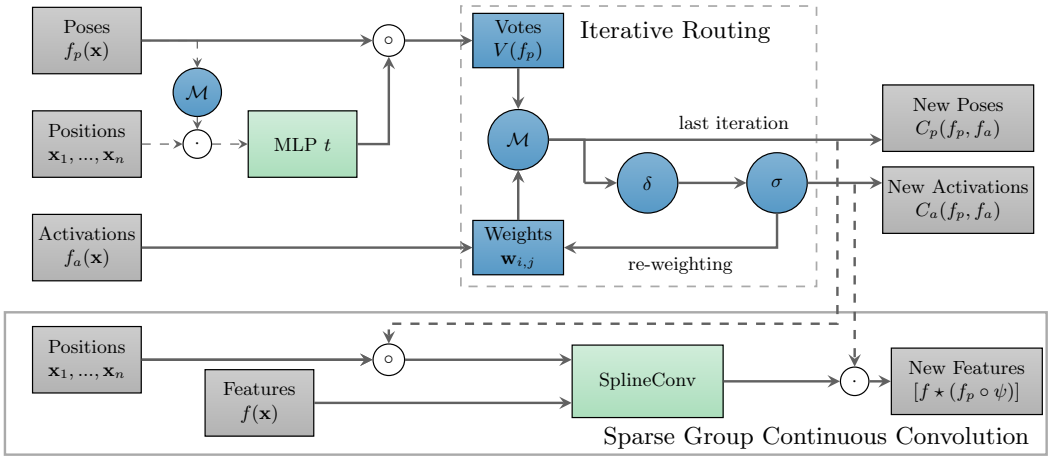


Figure 6.4: Data flow graph for one layer of the described GECN algorithm, containing the capsule part on the top and the sparsely evaluated convolution on the bottom. The building blocks include ■ fixed function parts and the ■ trainable MLP and SplineConv operators. Solid arrows indicate data that need gradient information in reverse mode accumulation in order to train an architecture consisting of multiple layers in an end-to-end fashion. To obtain gradients for poses $\mathbf{p}_1, ..., \mathbf{p}_n$, activations $a_1, ..., a_n$ and MLP parameters Θ, the fixed function realizations of $\mathcal{M}$, δ and σ need to be differentiable.

6.4.3 Full Algorithm and Reverse Mode

A scheme for a full layer of a GECN, including the trainable transformation MLP t and the SplineConv operator indexed by group elements, is shown in Figure 6.4. The iterative part of the routing algorithm is fixed-function and parameterized by the learned votes. Those parts of the data flow that need inverse gradient flow in reverse mode is indicated by solid arrows, while dashed arrows show data flow that does not need gradient information. Since we aim to parameterize the whole capsule part of the algorithm by training the transformation kernels t in all layers to optimally solve a down-stream task, all operators that lie between those kernels and the output need to be differentiable.

It can be observed that the crucial parts in the computation graphs are the operators $\mathcal{M}$ and δ, as for all other operations, the backward function is known and, since we restrict the approach to Lie groups $\mathcal{G}$, we obtain differentiability (almost everywhere at least) in the group representation by definition. Thus, when designing architecture instances for different groups, the differentiability of $\mathcal{M}$, δ and σ needs to be ensured. The explicit reverse mode computations are discussed in Sections 6.5 and 6.6, where the individual applications are presented.

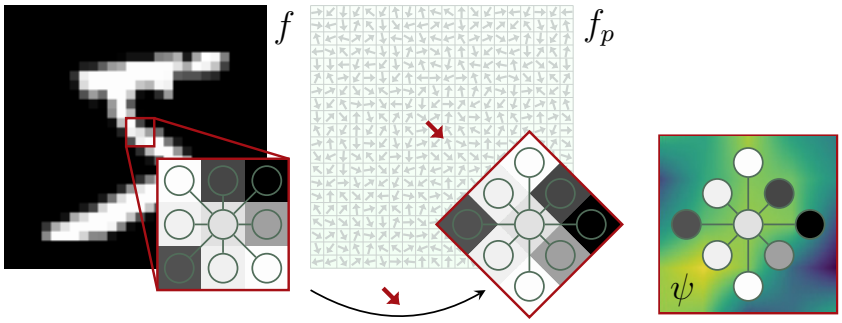


Figure 6.5: Realization of the sparse group convolution on vector field representations [LFL18] with indexing using group elements from dynamic routing. The local receptive fields are transformed using the calculated pose field f_p before aggregated using a continuous SplineConv kernel function ψ.

6.4.4 Product Group Convolutions

In this section, the potential application of GECNs to product groups is discussed, which can be a recipe for keeping the disentanglement between poses of a larger set of groups. The results of this section are not realized in the GECN applications presented later in this chapter but can be considered a outlook to potential future work. Given two groups $(\mathcal{G}, \circ_\mathcal{G})$ and $(\mathcal{H}, \circ_\mathcal{H})$, we can construct the direct product group $(\mathcal{G}, \circ_\mathcal{G}) \times (\mathcal{H}, \circ_\mathcal{H}) = (\mathcal{G} \times \mathcal{H}, \circ)$, with $(\mathbf{g}_1, \mathbf{h}_1) \circ (\mathbf{g}_2, \mathbf{h}_2) = (\mathbf{g}_1 \circ_G \mathbf{g}_2, \mathbf{h}_1 \circ_H \mathbf{h}_2)$. Therefore, Theorem 1 and 2 also apply for those combinations. As a result, the pose vectors contain independent poses for each group, keeping information disentangled between the individual ones.

Acknowledging that, we can go further and only use capsule routing for a subset of groups in the product: Given a product group $(\mathcal{G}, \circ) = (\mathcal{G}_1, \circ_1) \times (\mathcal{G}_2, \circ_2)$ (note that both can again be product groups), we can use routing by agreement with sparse convolution evaluation over the group $(\mathcal{G}_1, \circ_1)$ and dense convolution evaluation without routing over the group $(\mathcal{G}_2, \circ_2)$. This is similar to the sparse convolution described in Section 6.4.2 and can be general recipe for sets of groups that do not act on each other. Evaluation of the convolution operator changes to

$$[f \star \psi](\mathbf{x}_1, \mathbf{x}_2) = \int_{(\mathbf{g}_1, \mathbf{g}_2) \in G} f(\mathbf{g}_1, \mathbf{g}_2) \psi(\mathbf{x}_1^{-1} \circ_1 \mathbf{g}_1, \mathbf{x}_2^{-1} \circ_2 \mathbf{g}_2) \, d\mathbf{g}_1 d\mathbf{g}_2. \tag{6.26}$$

We preserve equi- and invariance results for the group with routing and equivariance for the one without, which we show by proving the following theorem. It leads to evaluating the feature maps densely for the second group while sparsely evaluating different elements of the first group at each element of the second group and routing between them from a layer l to layer $l+1$. Activations would still be invariant under application of the group that is indexed by the capsule poses.

Theorem 5. Let $(\mathcal{G}, \circ) = (\mathcal{R}, \circ_1) \times (\mathcal{T}, \circ_2)$ be a direct product group and $\mathcal{M}$ and δ be given like in Theorem 1. Further, let $\mathbf{e}$ be the neutral element of group T. Then,

the group convolution $[f \star \psi]$ is invariant under joint left-applications of $\mathbf{r} \in R$ on capsule input pose vectors $\mathbf{P} \in \mathcal{R}^n$ and vector field $f : \mathcal{G} \to \mathbb{R}$, for all $\mathbf{t} \in T$:

$$[((\mathbf{r}, \mathbf{e}) \circ f) \star \psi] (L_p(\mathbf{r} \circ \mathbf{P}, \mathbf{a}), \mathbf{t}) = [f \star \psi] (L_p(\mathbf{P}, \mathbf{a}), \mathbf{t}). \tag{6.27}$$

Proof. The proof is given for one output capsule j and one input feature map i (omitting the sum in the process). We show the equality analogously to Theorem 4 by applying the result of Theorem 1, the definition of the direct product group action on the feature map $(\mathbf{g}_1, \mathbf{g}_2) \circ f(\mathbf{h}_1, \mathbf{h}_2) = f(\mathbf{g}_1^{-1} \circ \mathbf{h}_1, \mathbf{g}_1^{-1} \circ \mathbf{h}_2)$, a substitution $\mathbf{g}_1 \to \mathbf{r} \circ \mathbf{g}_1$ and the group property $(\mathbf{g}_1 \circ \mathbf{g}_2)^{-1} = \mathbf{g}_2^{-1} \circ \mathbf{g}_1^{-1}$, using the existence of inverse and neutral element properties of groups:

$$\begin{aligned}
&[(\mathbf{r}, \mathbf{e}) \circ f \star \psi] (L_p(\mathbf{r} \circ \mathbf{P}, \mathbf{a}), \mathbf{t}) \\
&= [(\mathbf{r}, \mathbf{e}) \circ f \star \psi] (\mathbf{r} \circ L_p(\mathbf{P}, \mathbf{a}), \mathbf{t}) \\
&= \int_{(\mathbf{g}_1, \mathbf{g}_2) \in G} f(\mathbf{r}^{-1} \circ \mathbf{g}_1, \mathbf{e} \circ \mathbf{g}_2) \psi((\mathbf{r} \circ L_p(\mathbf{P}, \mathbf{a}))^{-1} \circ \mathbf{g}_1, \mathbf{t}^{-1} \circ \mathbf{g}_2) \; d\mathbf{g}_1 d\mathbf{g}_2 \\
&= \int_{(\mathbf{g}_1, \mathbf{g}_2) \in G} f(\mathbf{g}_1, \mathbf{g}_2) \psi((\mathbf{r} \circ L_p(\mathbf{P}, \mathbf{a}))^{-1} \circ \mathbf{r} \circ \mathbf{g}_1, \mathbf{t}^{-1} \circ \mathbf{g}_2) \; d\mathbf{g}_1 d\mathbf{g}_2 \\
&= \int_{(\mathbf{g}_1, \mathbf{g}_2) \in G} f(\mathbf{g}_1, \mathbf{g}_2) \psi((L_p(\mathbf{P}, \mathbf{a})_1^{-1} \circ \mathbf{r}^{-1} \circ \mathbf{r} \circ \mathbf{g}_1, \mathbf{t}^{-1} \circ \mathbf{g}_2) \; d\mathbf{g}_1 d\mathbf{g}_2 \\
&= \int_{(\mathbf{g}_1, \mathbf{g}_2) \in G} f(\mathbf{g}_1, \mathbf{g}_2) \psi((L_p(\mathbf{P}, \mathbf{a})_1^{-1} \circ \mathbf{g}_1, \mathbf{t}^{-1} \circ \mathbf{g}_2) \; d\mathbf{g}_1 d\mathbf{g}_2 \\
&= [f \star \psi] (L_p(\mathbf{P}, \mathbf{a}), \mathbf{t}).
\end{aligned}$$

□

The theorem allows us to create capsule modules which precisely allow to choose equivariances and invariances over a set of groups. It should be noted that it can only be applied to direct products of groups, as it depends on the disentangled group action. For semidirect products like $SE(n)$, the action of one group on the other needs to be eliminated, for example by the method described before in Section 6.4.2.

6.5 Group Capsule Networks on 2D Image Data

In this section, the instantiation of GECNs on 2D images with $SO(2)$ group capsules is presented, discussed and evaluated. It was originally published in 2018 [LFL18]. First, the specific design choices and the reasons for them are detailed, before turning to the evaluation on different MNIST datasets.

6.5.1 $SO(2)$ Group Capsule Networks

In order to derive a practical instantiation of the theoretical, group level considerations in Section 6.3 and Section 6.4, we need to decide on specific realizations of the group representation, the equivariant mean operator $\mathcal{M}$, the distance measure δ, and the network architecture.

Group Representation For $SO(2)$, a two-dimensional vector representation is chosen, the set of all two-dimensional unit vectors $\mathcal{P} = \{\mathbf{p} \in \mathbb{R}^2 \mid \|x\|_2 = 1\}$. Even if this representation is not minimal (since it is possible to parametrize $SO(2)$ in 1D), it provides the important advantage of being a continuous representation, in contrast to angle representations, which have at least one discontinuity in $SO(2)$. This makes it easier to define the following operators so that they obey to the required properties of equivariance, closeness, distance-preservation (cf. Theorem 1), and differentiability. To apply the group law, the vectors need to be brought into the form of 2D rotation matrices. For two poses $\mathbf{p}_1, \mathbf{p}_2 \in \mathcal{P}$, the group law $\mathbf{p} \circ \mathbf{p}'$ is computed as:

$$\mathbf{p} \circ \mathbf{p}' = \left(\begin{bmatrix} p_1 & -p_2 \\ p_2 & p_1 \end{bmatrix} \cdot \begin{bmatrix} p_1' & -p_2' \\ p_2' & p_1' \end{bmatrix} \right)_{\cdot,1}, \tag{6.28}$$

where the first column of the resulting matrix is taken as result vector. In practice, we can directly compute the operation as matrix-vector multiplication, omitting the second column of the second matrix.

Distance Measure For two poses $\mathbf{p}_1, \mathbf{p}_2 \in SO(2)$, we define the distance measure as

$$\delta(\mathbf{p}_1, \mathbf{p}_2) = \frac{1}{2}(-\mathbf{p}_1^\top \mathbf{p}_2 + 1). \tag{6.29}$$

The distance δ only consists of elementary operations and thus is easily differentiable in reverse mode. Note that we are not using the true geodesic distance of the 1-sphere, which would require correct scaling by applying the $\arccos(\cdot)$ to the scalar product. For better efficiency in forward and backward computations, we omit the $\arccos(\cdot)$ and use a linear approximation for scaling, as we found that it does not have an effect on the algorithm results. The distance is preserved by application of the group law, as the scalar product is preserved by simultaneous vector rotation. Thus, the necessary requirement for application in the group capsule network is fulfilled.

Weighted Average Operator The problem of finding the average of our $SO(2)$ poses with representations in $\mathcal{P}$ is equivalent to finding the centroid of a set of points on the 1-sphere. That is, for n poses $\mathbf{p}_1, ..., \mathbf{p}_n \in \mathcal{P}$ with weights $\mathbf{w} = (w_1, ..., w_n) \in \mathbb{R}^n$ the goal is to find the point on the sphere that minimizes the weighted distance to all given points:

$$\bar{\mathbf{p}} = \underset{\mathbf{p} \in \mathcal{P}}{\operatorname{argmin}} \sum_{i=1}^{n} w_i \delta(\mathbf{p}, \mathbf{p}_i). \tag{6.30}$$

Finding the correct solution to this problem usually requires an iterative approach, depending on the distance δ. To avoid that, we again use an approximation and define the weighted average operator $\mathcal{M} : \mathcal{P}^n \times \mathbb{R}^n \to \mathcal{P}$ as re-normalized Euclidean distance:

$$\mathcal{M}(\mathbf{p}_1, ..., \mathbf{p}_n, \mathbf{w}) = \frac{\sum_{i=1}^n w_i \mathbf{p}_i}{\left\| \sum_{i=1}^n w_i \mathbf{p}_i \right\|_2}. \tag{6.31}$$

The downside of this method is that it is not defined for all possible inputs, as the vectors may amount to zero. However, in practice this can be easily avoided in the few rare cases it might happen. Since we expect our capsule poses to be more relevant if the votes align well (in which case this mean also is more accurate), we can expect this operator to produce sufficiently good results in important cases.

Verifying the required properties, it can be seen that the operator is easily differentiable in reverse mode (if the points do not amount to zero) by calculating the elementary derivatives and that it is left-equivariant:

$$\begin{aligned}\mathcal{M}(\mathbf{p}\circ\mathbf{p}_1,...,\mathbf{p}\circ\mathbf{p}_n,\mathbf{w}) &= \frac{\sum_{i=1}^n w_i(\mathbf{p}\circ\mathbf{p}_i)}{\|\sum_{i=1}^n w_i(\mathbf{p}\circ\mathbf{p}_i)\|_2}\\ &= \frac{\mathbf{p}\circ\sum_{i=1}^n w_i\mathbf{p}_i}{\|\mathbf{p}\circ\sum_{i=1}^n w_i\mathbf{p}_i\|_2}\\ &= \mathbf{p}\circ\frac{\sum_{i=1}^n w_i\mathbf{p}_i}{\|\sum_{i=1}^n w_i\mathbf{p}_i\|_2}\\ &= \mathbf{p}\circ\mathcal{M}(\mathbf{p}_1,...,\mathbf{p}_n,\mathbf{w})\end{aligned}$$

,

utilizing that vector rotation is norm-preserving, that scaling and rotation are commutative, and the distributive law of matrix multiplication.

Initial pose extraction An important subject which needs to be tackled is the first pose extraction of a group capsule network. We need to extract pose vectors $f_p^0 : \mathbb{R}^2 \to SO(2)$ with activations $f_a^0 : \mathbb{R}^2 \to \mathbb{R}$ from the raw input of the network without eliminating the equi- and invariance properties of Equations (6.11) and (6.12). The chosen solution for images is to compute local gradients using a Sobel operator and taking the length of the gradient as activation. Naturally, a zero gradient (and thus a zero activation) does lead to an undefined pose. Since the pose activation is used as weight in the weighted average, those poses have no influence on the result. One thing that needs to be ensured manually is that capsules with only zero inputs also produce a zero agreement and an undefined pose vector.

Convolution operator As convolution implementation for the group convolution part of the architecture, we chose SplineCNN [FLW+18], the continuous convolution operator described in Section 4.1.4. Although the discrete two- or three-dimensional convolution operator is also applicable, this variant allows us to omit the resampling of grids after applying group transformations on the input image. The reason for this is the continuous definition range of the B-spline kernel functions. We represent images as grid graphs and rotate the kernels by inversely rotating the relative positions given on the edges, as described in Section 6.4.2 and Section 4.1.4.

Dynamic routing In contrast to the method from Sabour et al. [SFH17], we do not use softmax over the output capsule dimension but the sigmoid function for

each weight individually. The sigmoid function makes it possible for the network to route information to more than one output capsule as well as to no output capsule at all. Further, we use two iterations of computing pose proposals.

Architecture and parameters The evaluated architecture consists of five $SO(2)$-vector field capsule layers where each layer aggregates capsules from 2×2 spatial blocks with stride 2. Thus, which each layer, we reduce the number of spatial sampling points by a factor of 2, until we reach only one spatial position in the last layer. The learned transformations t are shared over the spatial positions. We also pair each layer with a sparse group convolution layer, pose-indexed by the output of the vector field capsule layer, as described in Section 6.4.2, with ReLU non-linearities after each layer. The numbers of output capsules per position are 16, 32, 32, 64, and 10 for each of the five capsule layers, respectively. In total, the architecture contains 235k trainable parameters (145k for the capsules and 90k for the CNN). The architecture results in two sets of classification outputs: the agreement values of the last capsule layer, which has one output capsule for each class, as well as the softmax outputs from the sparse group convolution part. We use the spread loss as proposed by [HSF18] for the capsule part and standard cross entropy loss for the convolutional part and add them up. We trained our models for 45 epochs. For further details, the reader is referred to the implementation, which is available on Github[1].

6.5.2 $SO(2)$ Capsule Results

We provide proof of concept experiments to verify and visualize the theoretic properties shown in the previous sections. As an instance of the GECN algorithm, we first apply the previously described architecture to the task of $SO(2)$ equivariant classification on different MNIST datasets [LBB+98a]. Further, the resulting capsule poses are analyzed qualitatively and quantitatively.

Equivariance properties and accuracy We confirm equivariance and invariance properties of the algorithm by training the network on non-rotated MNIST images and test it on images, which are randomly rotated by multiples of $\pi/2$. We can confirm that we achieve exactly the same accuracy, as if we evaluate on the non-rotated test set, which is 99.02%. We also obtain the same output activations and equivariant pose vectors with occasional small numerical errors < 0.0001, which confirms perfect equi- and invariance in case of $\pi/2$ rotations, which do not introduce sampling errors. This is true for the capsule output as well as the output of the paired CNN. When we consider arbitrary rotations for testing, and thus having sampling errors, the accuracy of a network trained on non-rotated images is 89.12%, which is a decent generalization result, compared to standard CNNs, which achieve

[1] Implementation at: https://github.com/mrjel/group_equivariant_capsules_pytorch

	MNIST rot. (50k)	AffNist	MNIST rot. (10k)
CNN(*)	92.30%	81.64%	90.19%
Capsules	94.68%	71.86%	91.87%
Whole	**98.42%**	**89.10%**	**97.40%**

(a) Ablation experiment results

	Average pose error [degree]
Naive average poses	70.92
Capsules without reconstruction loss	28.32
Capsules with reconstruction loss	**16.21**

(b) Average pose errors for different configurations

Table 6.1: (a) Ablation experiments for the individual parts of our architecture including the CNN without induced pose vectors, the equivariant capsule network and the combined architecture [LFL18]. All MNIST experiments are conducted using randomly rotated training and testing data. (b) Average pose extraction error for three scenarios: simple averaging of initial pose vectors as baseline, our capsule architecture without reconstruction loss, and the same model with reconstruction loss.

classification results of approximately 60% in this scenario, depending on the chosen architecture.

For fully randomly rotated training and test sets we performed an ablation study using three datasets. Those include standard MNIST dataset with 50k training examples and the dedicated MNIST-rot dataset with the native 10k/50k train/test split [LEC+07]. In addition, we replicated the experiment of [SFH17] on the affNIST dataset[2], a modification of MNIST, where small, random affine transformations are applied to the images. We trained on padded and translated (not rotated) MNIST and tested on affNIST. All results are shown in Table 6.1a. We chose our CNN architecture without information from the capsule part as our baseline (*). Without the induced poses, the network is equivalent to a traditional CNN, similar to the grid experiment presented by [FLW+18]. When trained on a non-rotated MNIST, it achieves 99.13% test accuracy and generalizes weakly to a rotated test set with only 58.79% test accuracy. For training on rotated data, results are summarized in the table. The results show that combining capsules with convolutions significantly outperforms both parts alone. The pose vectors provided by the capsule network guide the CNN, which significantly boosts the

[2]affNIST: http://www.cs.toronto.edu/~tijmen/affNIST/

CNN for rotation invariant classification. However, the state-of-the-art of 99.29% in rotated MNIST classification obtained by [WHS18] is not reached. In the affNIST experiment we surpass the result of 79% from [SFH17] with much less parameters (235k vs. 6.8M) by a large margin.

Representations We provide a quantitative and a qualitative analysis of generated representations of our MNIST trained model in Table 6.1b and Figure 6.7, respectively. We measured the average pose error by rotating each MNIST test example by a random angle and calculated the distance between the predicted and expected poses. The results of our capsule networks with and without a reconstruction loss (cf. next paragraph) are compared to the naive approach of hierarchically averaging local pose vectors. The capsule poses are far more accurate, since they do not depend equally on all local poses but mostly on those which can be explained by the existence of the detected object. It should be noted that the pose extraction was not directly supervised—the networks were trained using discriminative class annotations (and reconstruction loss) only. Similar to [SFH17], we observe that using an additional reconstruction loss improves the extracted representations. In Figure 6.7a we show output poses for eleven random test samples, each rotated in $\pi/4$ steps. It can be seen that equivariant output poses are produced in most cases. The bottom row shows an error case, where an ambiguous pattern creates false poses. Figure 6.7b shows poses after the first (top) and the second (bottom) capsule layer.

We further plotted this error for each MNIST class individually in Figure 6.6. It can be seen that, for all classes, far away predictions are rarer than those near the correct pose. We can also observe variances between the classes. The classes with the largest errors are 1, 4 and 8 while pose vectors from classes 3, 6 and 9 are most accurate. We suspect that inherent symmetries of the symbols cause a larger pose error.

Reconstruction For further verification of disentanglement, we also replicated the autoencoder experiment of [SFH17] by appending a three-layer MLP to convolution outputs, agreement outputs, and poses and train it to reconstruct the input image. Example reconstructions can be seen in Figure 6.7c. To verify the disentanglement of rotation, we provide reconstructions of the images after we applied $\pi/4$ rotations to the output pose vectors. It can be seen that we have fine-grained control over the orientation of the resulting image. However, not all representations were reconstructed correctly. Visually correct ones were chosen for display.

6.6 Group Capsule Networks on 3D Point Clouds

In this section, the 3D application of GECNs on 3D point clouds is described in more detail. It was originally published as a joint work in 2020 [ZBL+20], from

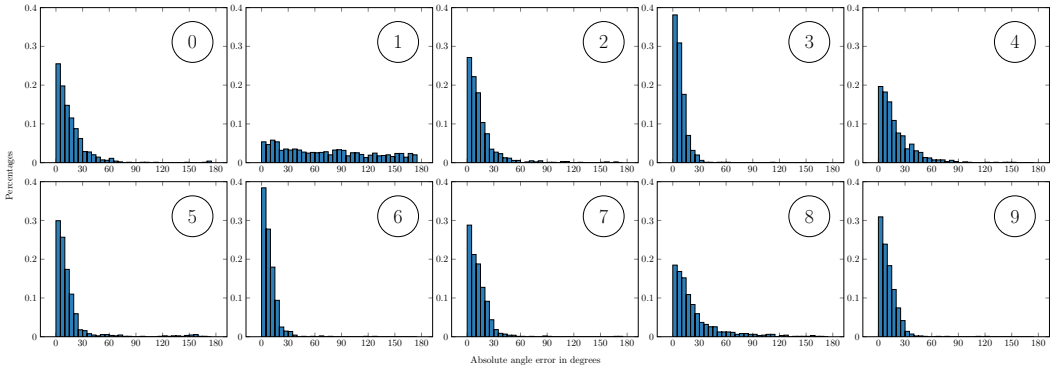


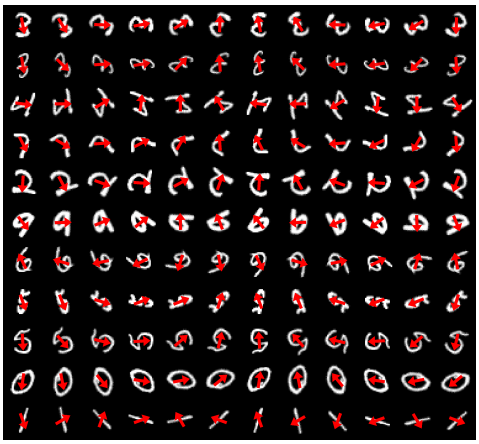
Figure 6.6: Angle error histograms for rotated inputs that require resampling [LFL18]. The plots are given for each MNIST class individually. The x axis shows bins for the angle errors in degree. The y axis represents the fraction of test examples falling in each bin.

which is section is partially adapted. We begin by detailing the design choices for the 3D capsule representation, operators and architecture, before summarizing results obtained on 3D point cloud datasets.

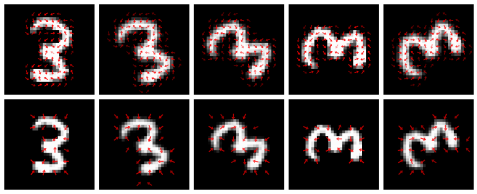
6.6.1 $SO(3)$ Group Capsule Networks

In the following, the chosen representation, distance operator and average operator of the 3D capsule instantiation is described in detail. Then, the architecture for point cloud rotation estimation and classification is presented, which is later evaluated in Section 6.6.2.

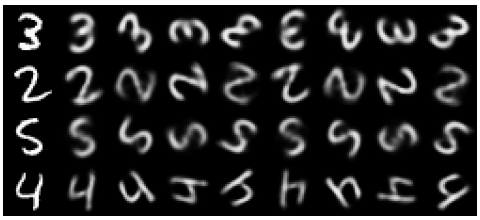
Group Representation There are several options for representing the $SO(3)$ rotation group, such as different axis-angle representations, rotation matrices, and quaternions. They all provide different advantages and disadvantages, which may or may not fit the given task. For $SO(3)$ capsules, the goal is to have a representation that (1) continuously represents $SO(3)$, (2) has an easy to compute distance measure, and (3) lends itself to an efficient, equivariant, differentiable average operation. Similar to angle representations for $SO(2)$, simple axis-angle representations for $SO(3)$ usually contain discontinuities at one or multiple points in the group. In addition to problems with differentiation, those discontinuities pose challenges for defining an equivariant average, which is why we refrain from choosing those representations. Rotation matrices, as such, are over-parameterized and averaging them trivially requires an iterative re-orthogonalization procedure. Considering all criteria, we chose unit quaternions as the best fitting $SO(3)$ representation for GECNs. Thus, the pose vectors $\mathbf{p}$ lie in the set of all four-dimensional unit vectors $\mathcal{P} = \{\mathbf{p} \in \mathbb{R}^4 \mid \|x\|_2 = 1\}$. One property that needs to be considered is that unit quaternions pose a double cover of $SO(3)$, in that the quaternions described by $\mathbf{p}$



(a) Output pose vectors for rotated inputs



(b) Poses after first and second capsule layer



(c) Reconstruction with transformed poses

Figure 6.7: Visualization of output poses (a), internal poses (b), and reconstructions (c) [LFL18]. (a) It can be seen that the network produces equivariant output pose vectors. The bottom row shows a rare error case, where symmetries lead to false poses. (b) Internal poses behave nearly equivariant, we can see differences due to changing discretization and image resampling. (c) The original test sample is on the left. Then, reconstructions after rotating the representation pose vector are shown. For the reconstruction, we selected visually correct reconstructed samples, which was not always the case.

and $-\mathbf{p}$ always describe the same rotation. However, it turns out that this does not pose a problem for defining the required operators.

Given quaternions as $SO(3)$ representation, the application of the group law can be done efficiently by quaternion multiplication

$$\mathbf{p} \circ \mathbf{p}' = \mathbf{p}\mathbf{p}' = (p_1 + p_2\mathbf{i} + p_3\mathbf{j} + p_4\mathbf{k})(p_1' + p_2'\mathbf{i} + p_3'\mathbf{j} + p_4'\mathbf{k}), \tag{6.32}$$

with subsequent re-normalization to account for numerical errors, which is efficient in forward and backward computation. It is also possible to convert the quaternion to a rotation matrix first (cf. Section 4.1.5 for forward and backward operations for this conversion) but converting and multiplying the matrices is less efficient than simply performing the multiplication in quaternion space.

Distance Measure Measuring the distances between unit quaternions as representation of $SO(3)$ is a well-understood task. The geodesic distance between quaternion poses $\mathbf{p}, \mathbf{q} \in \mathcal{P}$ is defined as

$$\delta(\mathbf{p}, \mathbf{q}) = 2 \cdot \arccos(|\langle \mathbf{p}, \mathbf{q} \rangle|), \tag{6.33}$$

which considers the double cover property and can be directly computed in forward and backward algorithms. It also is preserved by application of the group law:

$$\begin{aligned}\delta(\mathbf{g}\circ\mathbf{p},\mathbf{g}\circ\mathbf{q}) &= 2\cdot\arccos(|\langle\mathbf{g}\circ\mathbf{p},\mathbf{g}\circ\mathbf{q}\rangle|)\\ &= 2\cdot\arccos(|(\mathbf{gp})^\top(\mathbf{gq})|)\\ &= 2\cdot\arccos(|\mathbf{p}^\top\mathbf{g}^\top\mathbf{gq}|)\\ &= 2\cdot\arccos(|\mathbf{p}^\top\mathbf{eq}|)\\ &= 2\cdot\arccos(|\langle\mathbf{p},\mathbf{q}\rangle|)\\ &= 2\cdot\arccos(|\langle\mathbf{p},\mathbf{q}\rangle|)\\ &= \delta(\mathbf{p},\mathbf{q})\end{aligned}$$

which follows from the orthonormality of the transformations.

Weighted Average Operator Given unit quaternions $\mathbf{p}_1,\ldots\mathbf{p}_n$ and weights $\mathbf{w}\in\mathbb{R}^n$, the goal of the weighted average operation is to find the solution of

$$\hat{\mathbf{g}} = \operatorname*{argmin}_{\mathbf{h}\in SO(3)}\sum_{i=1}^{n}\delta(\mathbf{h},\mathbf{p}_i). \tag{6.34}$$

which usually requires an iterative algorithm to solve.

Markley et al. [MCC+07] showed that finding the average of a set of quaternions can be formulated as a least squares problem, by interpreting the quaternions as normal vectors of four-dimensional planes and finding the average plane of these given normal vectors. Fortunately, this can be solved using Eigendecomposition (ED), for which we have a tractable backward operation (cf. Section 4.2). The problem amounts to finding

$$\hat{\mathbf{q}} = \operatorname*{argmax}_{\mathbf{q}\in\mathcal{P}}\mathbf{q}^\top\mathbf{Mq}, \tag{6.35}$$

with $\mathbf{M}$ being the weighted covariance matrix of the quaternions

$$\mathbf{M} = \sum_{i=1}^{n} w_i\mathbf{q}_i\mathbf{q}_i^\top, \tag{6.36}$$

for which the largest magnitude eigenvector needs to be found. The operator is equivariant as required for application in GECNs:

$$\begin{aligned}\operatorname*{argmax}_{\mathbf{q}\in\mathcal{P}}\mathbf{q}^\top(\sum_{i=1}^{n}w_i(\mathbf{g}\circ\mathbf{q}_i)(\mathbf{g}\circ\mathbf{q}_i)^\top)\mathbf{q} &= \operatorname*{argmax}_{\mathbf{q}\in\mathcal{P}}\mathbf{q}^\top(\sum_{i=1}^{n}w_i\mathbf{g}\mathbf{q}_i\mathbf{q}_i^\top\mathbf{g}^\top)\mathbf{q}\\ &= \operatorname*{argmax}_{\mathbf{q}\in\mathcal{P}}\mathbf{q}^\top\mathbf{g}(\sum_{i=1}^{n}w_i\mathbf{q}_i\mathbf{q}_i^\top)\mathbf{g}^\top\mathbf{q}\\ &= \operatorname*{argmax}_{\mathbf{q}\in\mathcal{P}}\mathbf{q}^\top\mathbf{g}\mathbf{M}\mathbf{g}^\top\mathbf{q}\\ &= \mathbf{g}\circ\operatorname*{argmax}_{\mathbf{p}\in\mathcal{P}}\mathbf{p}^\top\mathbf{Mp},\end{aligned}$$

utilizing the orthonormality of transformations and a substitution $\mathbf{p} = \mathbf{g}^\top \mathbf{q}$.

It is interesting to see that one average computation using the presented scheme is equivalent to one Weiszfeld iteration, a well-known IRLS scheme for robustly computing the L_q geometric median. Hence, the resulting routing by agreement procedure for quaternion GECNs can be understood as a trainable variant of the Weiszfeld algorithm and was called Weiszfeld dynamic routing [ZBL+20].

Initial pose extraction Similar to the 2D version, GECNs on 3D point clouds need initial $SO(3)$ poses as input, which need to behave equivariant to global rotation of the point cloud. A typical descriptor for such poses on 3D point clouds are estimated Local Reference Frames (LRFs), consisting of three orthogonal vectors, indicating local surface orientation. We use the FLARE method [PD12] to compute one for each input capsule locally on the input point cloud. As first vector, a locally fitted surface normal is taken. The second axis is fixed as the direction to the most distant point from the tangent plane, projected onto that plane. Without resampling, the operator is equivariant with respect to input rotation. Similar to the 2D case, if a different point sampling of the same object is used, the LRFs might differ due to different sampling.

Dynamic routing The Weiszfeld dynamic routing takes place as detailed in Algorithm 8, where pose vectors $\mathbf{p}$ are given as unit quaternions and the transformations $\mathbf{t}$ are the output of an continuous kernel on the point positions, as described in Section 6.4. As weight scaling function σ, the sigmoid functions is used. The number of iterations r is set to 3.

Architecture and parameters For the experiments on $SO(3)$ capsule networks, two different types of architectures were used [ZBL+20]. One standard classification architecture, mapping input point clouds to one capsule per class, similar to the 2D MNIST experiment described in Section 6.5, and one siamese architecture for pose estimation. The siamese architecture maps an object in two different orientations to the respective pose vectors and computes the relative rotation between the two inputs. It consists of two instantiations of the classification architecture, sharing weights.

The classification network consists of two hierarchically stacked capsule layers, the first receiving 64 input point cloud patches, for each of which an LRF is computed as input pose. The centers of those patches, called pooling centers, are computed via uniform farthest point sampling [BI17]. Then, two $SO(3)$-vector field capsule layers are applied. In the first layer, the capsules of each patch are aggregated with the capsules of their 9 nearest neighbors, resulting in 64 output capsules for each of the 64 points. The second layer takes all intermediate capsules as input and aggregates them into one spatial point with 40 output capsules, one for each class of the ModelNet40 dataset, used for evaluation.

	NR/NR	NR/AR	Num. Params.
PointNet [QSK+17]	88.45	12.47	3.5M
PointNet++ [QYS+17]	89.82	21.35	1.5M
DGCNN [WSL+19]	**92.90**	29.74	2.8M
KDTreeNet [KL17]	86.20	8.49	3.6M
Point2Seq [LHL+19]	92.60	10.53	1.8M
Spherical CNNs [CGK+18]	-	43.92	0.5M
PRIN [YLL+20]	80.13	68.85	1.5M
PPF-FoldNet [DBI18a]	70.16	70.16	3.5M
$SO(3)$ **GECN**	74.43	**74.07**	**0.4M**

Table 6.2: Results of the $SO(3)$ GECN for ModelNet40 point cloud classification in comparison to related methods [ZBL+20]. For both evaluation scenarios (*NR/NR* and *NR/AR*), the classification accuracy is reported in percent. In the rightmost column, the number of model parameters is compared.

6.6.2 $SO(3)$ Capsule Results

In this section, results for the $SO(3)$ capsule architecture are presented. For the full evaluation of the method, the reader is referred to the original publication [ZBL+20]. The evaluation is split into two different parts, evaluating classification accuracy of the proposed architecture and analyzing the quality of output pose vectors for object alignment and interpretation.

ModelNet40 Dataset The ModelNet40 dataset [WHG+15] contains a large set of CAD models represented as meshes, which are divided between 40 different object classes. In the recent years, it was often used for evaluating deep learning methods for point cloud processing [QSK+17; QYS+17], by sampling point clouds from the given meshes. For evaluation of the capsule network, the official split with 9,843 objects for training and 2,468 objects for testing has been used. For each object, a point cloud with 10k points has been randomly sampled from the mesh surface.

Classification Similar to the 2D experiments on MNIST, we evaluated two different scenarios and compared them against results of related methods. The goal is to evaluate the network capability of generalizing to different input poses, that is correctly classifying objects that are given in orientations that were never observed during training. Thus, for both scenarios, the network was trained on aligned objects, which are given in canonical orientation for each class. Then, the trained networks were evaluated on (1, *NR/NR*) an aligned test set and (2, *NR/AR*) a test set, where each test object is given in five different random $SO(3)$ poses.

The results are shown in Table 6.2 as average classification accuracy in percent over the whole test set. While the network does not reach the state of the art results in classification of aligned objects, it outperforms all other networks when

Method	Avg. All	NoSym.	Chair	Bed	Sofa	Toilet	Monitor
Mean LRF	0.41	0.35	0.32	0.36	0.34	0.41	0.34
PCA-S	0.40	0.42	0.60	0.53	0.46	0.32	0.12
PCA-SR	0.67	0.67	0.69	0.70	0.67	0.68	0.61
PointNetLK [AGS+19]	0.37	0.38	0.43	0.31	0.40	0.40	0.31
IT-Net [YHM+18]	0.27	0.19	0.10	0.22	0.17	0.20	0.28
Ours (Siamese)	**0.20**	**0.09**	**0.08**	**0.10**	**0.08**	**0.11**	**0.08**

(a) Classes without rotational symmetries

Method	Table	Desk	Dresser	NS	Bathtub
Mean LRF	0.45	0.60	0.50	0.46	0.32
PCA-S	0.47	**0.23**	**0.33**	0.43	0.55
PCA-SR	0.67	0.67	0.67	0.66	0.70
PointNetLK [AGS+19]	0.40	0.33	0.39	0.38	0.34
IT-Net [YHM+18]	**0.31**	0.41	0.44	0.40	0.39
Ours (Siamese)	0.40	0.35	0.34	**0.32**	**0.30**

(b) Classes with rotational symmetries

Table 6.3: Relative angular error (RAE) of rotation estimation of ModelNet10 categories with (b) and without (a) rotational symmetries [ZBL+20]. The results of the siamese $SO(3)$ GECN architecture are compared against naive baselines (Mean LRF, PCA-S, PCA-SR) and previous deep learning methods for alignment of point clouds (PointNetLK, IT-Net).

classifying objects in arbitrary rotation. The results match those obtained with the 2D version of GECNs. Notably, it can be seen that it produces better results than other equivariant or invariant architectures, such as Spherical CNNs, PRIN, and PPF-FoldNet, while using less parameters. The second best method for this task, PPF-FoldNet [DBI18a], uses rotation invariant input features, which makes it completely invariant to input rotation, losing expressiveness by discarding anisotropic patterns. In contrast, GECNs consider anisotropic features while still being able to classify objects in arbitrary rotations. Further analysis shows that most of the classification errors made by GECNs are made between object categories that have rotational symmetries [ZBL+20].

Pose Quality In addition to classification accuracy on ModelNet40, we analyze the quality of pose vectors produced by $SO(3)$ GECNs on ModelNet10 [WHG+15], a subset of ModelNet40. The Siamese architecture, which is described in Section 6.6, is trained on the ModelNet10 train set to infer the relative rotation between two randomly rotated input objects of the same type, without using any absolute pose supervision. The results are shown in Table 6.3, split between classes with

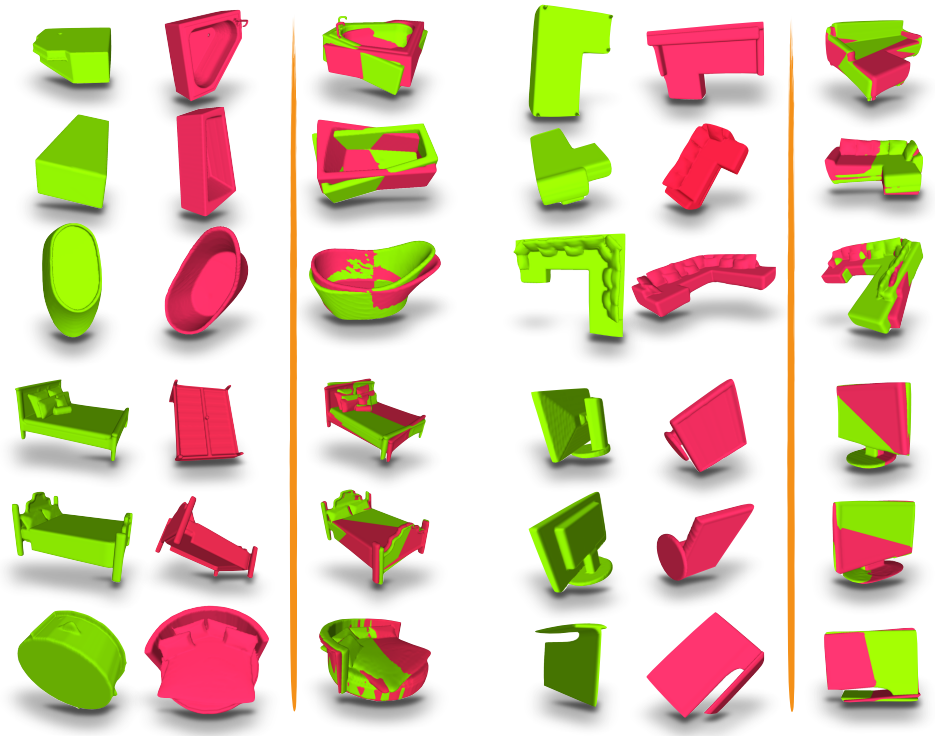


Figure 6.8: Qualitative results of object alignment with the Siamese $SO(3)$ GECN architecture (originally published in [ZBL+20]). In each of the two columns, the two leftmost examples show the input objects while the alignment result is shown on the right. From both input meshes, point clouds are sampled, which serve as input the network. The output are two object poses, which can directly be used to align the inputs.

inherent rotational symmetries and those without, and compared against three different baselines (Mean LRF, PCA-S, PCA-SR) and two previous state-of-the-art deep learning methods for point cloud alignment (PointNetLK [AGS+19], IT-Net [YHM+18]). Here, Mean LRF is simple averaging of all initial LRFs computed on the point cloud, PCA-S refers to principal axis alignment between two samplings of the same object, and PCA-SR refers to principal axis alignment between differently rotated and re-sampled versions of the same object. It can be seen that the naive approaches produce very bad results, some failing completely. PointNetLK and IT-Net are iterative DL architectures that can be understood as differentiable versions of the iterative closest point (ICP) algorithm [BM92]. $SO(3)$ GECNs produce much better alignment results than the related methods. This is best seen on objects without rotational symmetries, where the method is able to align the objects up to a very small error. On objects with symmetries, the error becomes larger, as GECNs do not have a mechanism for symmetry handling. However, on average, the errors are still lower than those obtained by previous work. Figure 6.8 shows

qualitative results for object alignment. The figure shows different examples of object pairs that have been aligned using a single evaluation of the Siamese $SO(3)$ GECN architecture. It should be noted that the method receives point cloud samples of the objects as input.

6.7 Discussion and Limitations

In this chapter, Group Equivariant Capsule Networks (GECNs) have been detailed that provide provable equivariance and invariance properties for transformations in certain Lie groups. They include a scheme for differentiable, iterative routing by agreement algorithms, a spatial aggregation method, and the ability to integrate group convolutions. The proven properties of equivariance and invariance were confirmed through experiments on 2D images and 3D point clouds for the Lie groups $SO(2)$ and $SO(3)$, by showing that the architecture provides informative, disentangled pose vectors which can be interpreted by humans. On certain tasks, for which invariance or equivariance is advantageous, the presented architectures improved on previous state-of-the-art results while requiring fewer trainable parameters.

Limitations of the presented method arise from the restriction of capsule poses to be elements of a group for which we can define proper $\mathcal{M}$ and δ. Therefore, in contrast to the original, less restricted capsule networks, arbitrary pose vectors can no longer be extracted. Through product groups though, it is possible to combine several groups and achieve more general pose vectors with internally disentangled information if we can find $\mathcal{M}$ and δ for this group. As for $SO(2)$ and $SO(3)$, an implementation of an equivariant center of mass $\mathcal{M}$ could be found for more Lie groups. However, for each group there is a different number of possible realizations from which only few are applicable in a deep neural network architecture, since it still needs to be efficiently differentiable. Iterative solutions that need a long time to converge (or maybe do not) may not be sufficient, since the mean is applied several times within one model.

The most notable theoretical limitation is the lack of symmetry handling in GECN. The method imposes the assumption that each capsule represents an object including its unique pose. However, in reality, several objects have inherent symmetries in the capsule group. For example, in the 2D version, the digit 1 may look exactly the same for two different rotations, offset by 180°, depending on the drawing style. Similarly, a 3D representation of a table might have several poses in which it looks exactly like the non-rotated version. Thus, the full Lie group is not always the best representation to describe poses of real world objects, leading to degraded performance of GECNs in those scenarios. For future work, there are potential solutions for this issue (cf. Section 7.3).

Chapter 7

Conclusion and Future Work

This thesis started with two premises, describing a hypothesis of the current state of DL, in Section 1. The first premise stated the importance of designing parallel data flow and data representations to efficiently process the vast amount of available data types we have in practice. The second premise states the hypothesis that in the near future, the most successful methods in the near future will combine the interpretability and efficiency of fixed algorithms incorporating problem-specific knowledge with a strong data-driven parameterization using DL methods. In the following, the contributions of this thesis are summarized in Section 7.1 and their results summarized and discussed in Section 7.2. Lastly, an outline of potential future work is given in Section 7.3.

7.1 Summary

The thesis followed the two premises by presenting methods for representing and processing different types of data and by designing sophisticated differentiable algorithms to solve tasks in 3D vision. Chapter 4 introduced important building blocks for designing differentiable algorithms. Contributions were made in the field of GNNs, including SplineCNN (cf. Section 4.1.4), an operator for differentiable continuous convolution on geometric data, and LSGTs, a network that infers local surface poses for equivariant data processing. For both methods, efficient GPU implementations for forward and backward application were described, allowing them to be used to capture information from large 3D data sets. For ED on a large number of symmetric 3×3 matrices, efficient forward and backward algorithms have been proposed in Section 4.2.1, allowing to integrate that operator in large scale, parallel DL architectures. Additionally, Section 4.3.4 outlined a method for 3D surface reconstruction, which can be applied to large real world datasets by applying recent successes in the field of INFs on a local level.

The first main contribution of this thesis, the DISNE method, was presented in Chapter 5, a differentiable algorithm for surface normal estimation on large point clouds, which combines traditional IRLS for plane fitting with a data-driven DL parameterization. The algorithm utilized building blocks introduced in Chapter 4, namely the LSGT method, the presented ED solver, and a kernel INF for re-weighting. The second main contribution, GECNs, were presented in Chapter 6, a novel take on capsule networks, which introduces strong, appropriate inductive biases into capsules, allowing to achieve provable equivariance properties with respect to the

group action. Two potential applications in 2D and 3D have been described, utilizing presented building blocks from the fields of GNNs and differentiable ED.

7.2 Discussion of Results

The two presented algorithms were analyzed with respect to *interpretability*, *efficiency* and *quality of results* in order to confirm the second premise of the introduction. The main hypothesis was that using fixed function parts to introduce stronger inductive bias leads to a better trade-off in these three categories than pure end-to-end DL approaches or pure hand-designed algorithms.

Differentiable Iterative Surface Normal Estimation (DISNE) After analyzing the results, we can say that the DISNE method clearly provides a better trade-off in the given metrics than previous methods. The quality of results is only slightly improved by lowering the average normal error by 4.6% in comparison to the second best method. However, the true gains are achieved in efficiency and interpretability. DISNE achieves speed ups of orders of magnitude, being 378× and 131× faster than previous DL approaches, which achieve second and third best quality of results. Additionally, DISNE reduces the required number of trainable parameters by more than 99%. DISNE is also easier to interpret: intermediate results of inferred point weights and normal vectors can be visualized and analyzed in behavior over iterations, showing how the method converges to the most plausible plane. Since the obtained normals are always a result of weighted plane fitting, we can narrow down the space of possible solutions. Most of the gains can be explained by the extend of the learning task that is solved. Instead of learning the whole relation between point sets and normal vectors, we only solve a sub-task of normal estimation using deep learning and incorporate well-known geometric relationships in the fixed function least squares solver.

Group Equivariant Capsule Networks (GECNs) When analyzing the results of the GECN method, the results are mixed, obtaining improvements in some categories while making sacrifices others. Improvements with respect to traditional capsule networks could be found in number of required trainable parameters, where a reduction of 97% was obtained. However, in case of GECNs, this does not lead to an improved runtime, as the fixed-function group operations are computation-heavy. The quality of results in generalization to novel test poses was improved by 12% ($SO(2)$ poses) and 5.7% ($SO(3)$ poses) but it could be observed that the model suffered in capability of memorization when trained on fully augmented training sets, which can be contributed to the less number of parameters. The largest improvements can be seen in interpretability. The capsule pose vectors are guaranteed to behave equivariant (except for re-sampling artifacts) and can directly be interpreted as elements of a pre-defined transformation group, making it possible

to directly use the method for unsupervised pose estimation. Also, it was shown that when using the capsules as hidden representation, we have fine-grained control over the pose of reconstructions obtained from them. These are novel achievements in the area of equivariant networks. All in all, through incorporating stronger inductive biases, we obtained a method that is easier to interpret and better in tasks for which it was specifically designed. However, sacrifices in efficiency and memorization had to be made.

We can conclude that designing sophisticated data flow for differentiable algorithms has the potential to obtain solutions that are more practical than pure end-to-end approaches while keeping their data-dependency and resulting quality. It was not possible to extract a general recipe for creating such methods. In order for them to succeed, they have to be build upon deeper understanding of the individual task, shifting away complexity from the learned function to fixed-function parts. However, certain concepts and building blocks, such as those presented in Chapter 4, are quite general and can be successfully applied in different problem domains.

7.3 Future Work

The area of differentiable algorithms with data-driven parameterization is very broad, leaving potential for designing methods to solve a large variety of different tasks. This section will outline three potential areas, which might be advanced by creating novel differentiable algorithms using the tools described in this thesis.

Consider Symmetries in Group Capsules One important limitation of the current GECNs as described in Chapter 6 is their inability to correctly detect and consider symmetries in the detected objects. Lets consider a 3D object like a table, which is symmetric with respect to 180° rotation. Then, the network should have the possibility to express two potential poses, instead of only one. There are two potential ways to solve this problem. One of that is to represent capsules as a distribution over the group, instead of a single group element, allowing for multiple peaks in the distributions. Instead of individual poses, we would propagate those distributions, or patterns, through the network in an equivariant fashion. Another approach could be to introduce different symmetry groups instead of using one Lie group, e.g. point groups for rotation, which describe the operations under which the object behaves invariant. Both approaches would change the method significantly and would require to recreate all involved operators to correctly process distributions or multiple symmetry groups in an equivariant way. Since GECNs already are computation heavy, those approaches would also require significant additional investigation to create tractable, efficient implementations.

Differentiable Laplacian Eigenbases In Section 4.2.2 it was described how to use Laplacian eigenbases in differentiable algorithms. The Laplacian operator $\mathbf{L}$

describes how a quantity diffuses over time over a given domain: we can multiply the Laplacian with a function on the domain to simulate this diffusion. The eigenbasis ψ of an operator contains stable modes of this diffusion, i.e. functions, which do only vary in magnitude when diffused further (since $\mathbf{L}\psi = \lambda\psi$ per definition). Thus, the mapping from an operator to its eigenbasis can be seen as a global diffusion process until convergence.

In most existing works (spectral GNNs [BZS+14; BBL+17], deep functional maps [OBS+12; LRR+17], diffusion networks [SAC+20]), the process of estimating the eigenbasis of the Laplacian is considered a fixed-function preprocessing step, using manually designed Laplacian operators $\mathbf{L}$ that adhere to certain properties. Then, the learning takes place either in spatial or spectral domain, using the eigenbasis of $\mathbf{L}$ as generalized Fourier transform between these two domains. However, in future work, we might be able utilize differentiable ED to learn information about the domain instead of just learning functions on a fixed domain, thus, to backpropagate through global domain diffusion of a trainable operator. If we solve a supervised down-stream task using the eigenbasis $\mathbf{\Psi}$ (or a function of it), we can either directly learn an optimal operator $\mathbf{L}$ as appropriate description of the underlying domain or train a network to produce an optimal operator based on observed data. First steps in this research direction have been taken by in recent work [SS21]. However, the issues with differentiating through ED, stability and complexity of the operation for large matrices $\mathbf{L}$, currently hinder the construction of advanced methods in this area. Thus, developing such methods requires more investigation in the area of efficient, differentiable ED or appropriate surrogates.

Equivariant Operators in INF Reconstruction Volumetric representations, such as DeepSDF [PFS+19], DeepLS (cf. Section 4.3.4) [CLI+20], and other variants that express volumes as a set of INFs suffer from the limitation of being restricted to a single orientation. Thus, an INF that learned to represent a certain object category, is only able to represent that object in a single orientation. Therefore, if new data should be fused that comes in a slightly different orientation, the applied models easily fail. In current local methods, such as DeepLS, this problem is circumvented by training the local INFs on objects with randomly augmented orientations. However, in the future, we can work towards a more sophisticated approach using equivariant transformations on INF input coordinates. Given a partial scene, we could estimate local orientations $\mathbf{R}$, e.g. through the application of LSGT (cf. Section 4.1.5) or GECNs (cf. Section 6.6) and canonicalize the input domain of an INF by canonicalization: $f_\psi(\mathbf{R}^{-1}\mathbf{x}, \theta)$. Since local INFs would no longer need to learn to represent objects in all possible orientations, this change has the potential to heavily improve on efficiency. Further, it might be possible to attach such equivariant INFs to a graph, further increasing the sparsity of local INF representations.

Acronyms

k-NN	k-nearest-neighbor 30, 72
AD	Automatic Differentiation 5, 7–11
CNN	Convolutional Neural Network iii, 1, 2, 11, 12, 16, 17, 19, 27, 33, 34, 36, 40, 41, 56, 87, 88
CS	Compressed Sensing 56–58
DeepLS	Deep Local Shapes 5, 56, 59–62
DISNE	Differentiable Iterative Surface Normal Estimation iv, 3–5, 60, 65, 66, 69, 73–75, 77, 78, 80, 83–85, 121, 122
DL	Deep Learning iii, iv, 1–5, 7, 9, 11–14, 17, 66, 67, 69, 85, 121, 122
DNN	Deep Neural Network iii, 7, 10, 11, 13, 15, 21, 42, 47, 57, 65
ED	Eigendecomposition iii, 2, 11, 20, 21, 46, 48–53, 55, 67, 68, 72, 74, 114, 121, 122, 124
GAN	Generative Adversarial Network 57
GECN	Group Equivariant Capsule Network iv, 3–5, 87–91, 97, 104–106, 109, 111, 112, 114–119, 121–124
GNN	Graph Neural Network iii, iv, 4–6, 14, 17, 19, 23, 28–30, 32, 33, 42, 46, 51, 56, 60, 66, 71, 74, 85, 90, 99, 121, 122, 124
GPU	Graphics Processing Unit iii, 1–4, 10, 25, 121
INF	Implicit Neural Function iii, 4, 5, 14, 56–62, 66, 69, 73, 74, 121, 124
IRLS	Iterative Re-weighted Least Squares iv, 3, 5, 15, 20, 21, 42, 45, 46, 50, 55, 57, 65, 89, 94, 115, 121
LBO	Laplace-Beltrami Operator 46, 50, 51

LSGT	Local Spatial Graph Transformer iii, 3, 4, 31, 42–46, 60, 66, 69, 71, 73, 81, 121, 124
LSTM	Long Short-Term Memory 1
MLP	Multi-Layer Perceptron 1, 2, 10, 12, 15–17, 30, 31, 43, 56–58, 60, 71, 104
MP-GNN	Message Passing Graph Neural Network 4, 23–34, 36, 42–44, 46, 53, 70, 71
PCA	Principal Component Analysis 46, 67
RNN	Recurrent Neural Network 1, 12
SDF	Signed Distance Function 57, 59–61
SVD	Singular Value Decomposition 2, 20, 21, 46–50, 52, 53, 67, 68, 72

Bibliography

[AB98] N. Amenta and M. Bern. "Surface Reconstruction by Voronoi Filtering". In: *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*. SCG '98. 1998, pp. 39–48 (Cited on page 67).

[ACT+07] P. Alliez, D. Cohen-Steiner, Y. Tong, and M. Desbrun. "Voronoi-based Variational Reconstruction of Unoriented Point Sets". In: *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*. SGP '07. 2007, pp. 39–48 (Cited on page 67).

[AGS+19] Y. Aoki, H. Goforth, R. A. Srivatsan, and S. Lucey. "PointNetLK: Robust & Efficient Point Cloud Registration Using PointNet". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (Cited on pages 117 sq.).

[ASL+17] S. Aroudj, P. Seemann, F. Langguth, S. Guthe, and M. Goesele. "Visibility-consistent Thin Surface Reconstruction Using Multi-scale Kernels". In: *ACM Transactions on Graphics (ToG)* 36.6 (2017), 187:1–187:13 (Cited on page 67).

[Bal87] D. H. Ballard. "Modular Learning in Neural Networks". In: *The AAAI Conference on Artificial Intelligence*. AAAI Press, 1987, pp. 279–284 (Cited on page 1).

[BBL+17] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. "Geometric Deep Learning: Going beyond Euclidean data". In: *IEEE Signal Processing Magazine* (2017), pp. 18–42 (Cited on pages 32, 51, 67, 124).

[BBP+19] G. Bouritsas, S. Bokhnyak, S. Ploumpis, M. Bronstein, and S. Zafeiriou. "Neural 3D Morphable Models: Spiral Convolutional Networks for 3D Shape Representation Learning and Generation". In: *The IEEE International Conference on Computer Vision (ICCV)*. 2019 (Cited on page 33).

[BEK+17] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98 (Cited on page 11).

[BFH+18] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: `http://github.com/google/jax` (Cited on page 11).

[BFZ+20] G. Bouritsas, F. Frasca, S. Zafeiriou, and M. M. Bronstein. "Improving Graph Neural Network Expressivity via Subgraph Isomorphism Counting". In: vol. abs/2006.09252. 2020. arXiv: `2006.09252` (Cited on page 30).

[BH12] M. Bouakkaz and M.-F. Harkat. "Combined input training and radial basis function neural networks based nonlinear principal components analysis model applied for process monitoring". In: *IJCCI 2012 - Proceedings of the 4th International Joint Conference on Computational Intelligence* (01/2012), pp. 483–492 (Cited on page 57).

[BI17] T. Birdal and S. Ilic. "A point sampling algorithm for 3D matching of irregular geometries". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*. IEEE, 2017, pp. 6871–6878 (Cited on page 115).

[BJL+18] P. Bojanowski, A. Joulin, D. Lopez-Pas, and A. Szlam. "Optimizing the Latent Space of Generative Networks". In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 10–15 Jul/2018, pp. 600–609 (Cited on page 57).

[BLF18] Y. Ben-Shabat, M. Lindenbaum, and A. Fischer. "Nesti-Net: Normal Estimation for Unstructured 3D Point Clouds using Convolutional Neural Networks". In: *CoRR* abs/1812.00709 (2018) (Cited on pages 65, 68 sq., 74 sq., 78, 85).

[BM12] A. Boulch and R. Marlet. "Fast and Robust Normal Estimation for Point Clouds with Sharp Features". In: *Computer Graphics Forum* (2012) (Cited on page 67).

[BM16] A. Boulch and R. Marlet. "Deep Learning for Robust Normal Estimation in Unstructured Point Clouds". In: *Computer Graphics Forum* (2016) (Cited on pages 65, 67, 75).

[BM92] P. J. Besl and N. D. McKay. "A Method for Registration of 3-D Shapes." In: *IEEE Trans. Pattern Anal. Mach. Intell.* 14.2 (1992), pp. 239–256 (Cited on page 118).

[BMR+16] D. Boscaini, J. Masci, E. Rodolà, and M. Bronstein. "Learning Shape Correspondence with Anisotropic Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems (NIPS)*. 2016, pp. 3189–3197 (Cited on pages 32, 42).

[BMR+20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901 (Cited on page 2).

[BPR+17] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. "Automatic Differentiation in Machine Learning: A Survey". In: *J. Mach. Learn. Res.* 18.1 (2017), pp. 5595–5637 (Cited on page 7).

[BRG16] A. Bansal, B. C. Russell, and A. Gupta. "Marr Revisited: 2D-3D Alignment via Surface Normal Prediction". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 5965–5974 (Cited on page 67).

[BRL+14] F. Bogo, J. Romero, M. Loper, and M. J. Black. "FAUST: Dataset and Evaluation for 3D Mesh Registration". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2014, pp. 3794–3801 (Cited on page 41).

[Bur14] C. S. Burrus. *Iterative Reweighted Least Squares*. 2014 (Cited on page 21).

[BXS+20] S. Bi, Z. Xu, P. P. Srinivasan, B. Mildenhall, K. Sunkavalli, M. Havsan, Y. Hold-Geoffroy, D. Kriegman, and R. Ramamoorthi. "Neural Reflectance Fields for Appearance Acquisition". In: *CoRR* abs/2008.03824 (2020). arXiv: 2008.03824 (Cited on page 60).

[BZS+14] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. "Spectral Networks and Locally Connected Networks on Graphs". In: *International Conference on Learning Representations (ICLR)*. 2014 (Cited on pages 29, 32, 51, 124).

[CAP20] J. Chibane, T. Alldieck, and G. Pons-Moll. "Implicit Functions in Feature Space for 3D Shape Reconstruction and Completion". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 06/2020 (Cited on page 59).

[CFG+15] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. *ShapeNet: An Information-Rich 3D Model Repository*. Tech. rep. 2015. arXiv: 1512.03012 (Cited on page 61).

[CGK+18] T. S. Cohen, M. Geiger, J. Köhler, and M. Welling. "Spherical CNNs". In: *International Conference on Learning Representations (ICLR)*. 2018 (Cited on pages 90, 101, 116).

[CL96] B. Curless and M. Levoy. "A Volumetric Method for Building Complex Models from Range Images". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. 1996, pp. 303–312 (Cited on page 62).

[CLI+20] R. Chabra, J. E. Lenssen, E. Ilg, T. Schmidt, J. Straub, S. Lovegrove, and R. A. Newcombe. "Deep Local Shapes: Learning Local SDF Priors for Detailed 3D Reconstruction". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Vol. 12374. Springer, 2020, pp. 608–625 (Cited on pages 5 sq., 59–62, 124).

[CLL+15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *CoRR* abs/1512.01274 (2015). arXiv: 1512.01274 (Cited on page 11).

[CMP20] J. Chibane, A. Mir, and G. Pons-Moll. "Neural Unsigned Distance Fields for Implicit Function Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 12/2020 (Cited on page 59).

[CP03] F. Cazals and M. Pouget. "Estimating Differential Quantities Using Polynomial Fitting of Osculating Jets". In: *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. 2003, pp. 177–187 (Cited on pages 67, 75).

[CW16] T. S. Cohen and M. Welling. "Group Equivariant Convolutional Networks". In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2016, pp. 2990–2999 (Cited on pages 87, 90, 101).

[CW17] T. S. Cohen and M. Welling. "Steerable CNNs". In: *International Conference on Learning Representations (ICLR)*. 2017 (Cited on page 90).

[CWH+20] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. "Simple and Deep Graph Convolutional Networks". In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2020, pp. 1725–1735 (Cited on page 59).

[CWK+19] T. Cohen, M. Weiler, B. Kicanaoglu, and M. Welling. "Gauge Equivariant Convolutional Networks and the Icosahedral CNN". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 09–15 Jun/2019, pp. 1321–1330 (Cited on page 90).

[CZ19] Z. Chen and H. Zhang. "Learning Implicit Fields for Generative Shape Modeling". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (Cited on pages 57 sqq.).

[DBI18a] H. Deng, T. Birdal, and S. Ilic. "PPF-FoldNet: Unsupervised Learning of Rotation Invariant 3D Local Descriptors". In: *European Conference on Computer Vision (ECCV)*. 2018, pp. 620–638 (Cited on pages 71, 116 sq.).

[DBI18b] H. Deng, T. Birdal, and S. Ilic. "PPFNet: Global Context Aware Local Features for Robust 3D Point Matching". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 195–205 (Cited on page 71).

[DBV16] M. Defferrard, X. Bresson, and P. Vandergheynst. "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: *Advances in Neural Information Processing Systems (NIPS)*. 2016, pp. 3837–3845 (Cited on pages 29, 41).

[DDK16] S. Dieleman, J. De Fauw, and K. Kavukcuoglu. "Exploiting Cyclic Symmetry in Convolutional Neural Networks". In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2016, pp. 1889–1898 (Cited on page 90).

[DGK07] I. S. Dhillon, Y. Guan, and B. Kulis. "Weighted Graph Cuts without Eigenvectors: A Multilevel Approach". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2007), pp. 1944–1957 (Cited on page 41).

[DGY+20] B. Deng, K. Genova, S. Yazdani, S. Bouaziz, G. Hinton, and A. Tagliasacchi. "CvxNet: Learnable Convex Decomposition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020 (Cited on page 59).

[DL21] F. Dellaert and Y.-C. Lin. "Neural Volume Rendering: NeRF And Beyond". In: *CoRR* abs/2101.05204 (2021). arXiv: 2101.05204 (Cited on page 60).

[DWG+20] X. Ding, N. Wang, X. Gao, J. Li, X. Wang, and T. Liu. "Group Feedback Capsule Network". In: *IEEE Transactions on Image Processing* 29 (2020), pp. 6789–6799 (Cited on page 90).

[DYH+20] Z. Dang, M. K. Yi, Y. Hu, F. Wang, P. Fua, and M. Salzmann. "Eigendecomposition-Free Training of Deep Networks for Linear Least-Square Problems". In: *IEEE Trans. Pattern Anal. Mach. Intell.* (2020) (Cited on pages 11, 52).

[EAM+18] C. Esteves, C. Allen-Blanchette, A. Makadia, and K. Daniilidis. "Learning SO(3) Equivariant Representations with Spherical CNNs". In: *European Conference on Computer Vision (ECCV)*. 2018 (Cited on page 90).

[EF15] D. Eigen and R. Fergus. "Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-scale Convolutional Architecture". In: *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 2650–2658 (Cited on page 67).

[FC18] J. Fan and J. Cheng. "Matrix completion by deep matrix factorization". In: *Neural Networks* 98 (2018), pp. 34–41 (Cited on page 57).

[FGH13] D. F. Fouhey, A. Gupta, and M. Hebert. "Data-Driven 3D Primitives for Single Image Understanding". In: *International Conference on Computer Vision (ICCV)*. 2013 (Cited on page 67).

[FL19] M. Fey and J. E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019 (Cited on pages 5 sq., 24, 26, 28, 32, 74).

[FLM+20] M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege. "Deep Graph Matching Consensus". In: *International Conference on Learning Representations (ICLR)*. 2020 (Cited on pages 5 sq.).

[FLW+18] M. Fey, J. E. Lenssen, F. Weichert, and H. Müller. "SplineCNN: Fast Geometric Deep Learning With Continuous B-Spline Kernels". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 869–877 (Cited on pages 4 sqq., 31, 33, 35, 40 sqq., 108, 110).

[FLW+21] M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec. "GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings". In: *Proceedings of the International Conference on Machine Learning (ICML)*. 2021 (Cited on pages 5 sq.).

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (Cited on pages 11, 16).

[GCB+19] S. Gong, L. Chen, M. Bronstein, and S. Zafeiriou. "SpiralNet++: A Fast and Highly Efficient Mesh Convolution Operator". In: *International Conference on Computer Vision (ICCV) Workshops*. 2019, pp. 4141–4148 (Cited on page 33).

[GCV+19] K. Genova, F. Cole, D. Vlasic, A. Sarna, W. T. Freeman, and T. Funkhouser. "Learning Shape Templates With Structured Implicit Functions". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 10/2019 (Cited on page 59).

[GFK+18] T. Groueix, M. Fisher, V. G. Kim, B. C. Russell, and M. Aubry. "A Papier-Maché Approach to Learning 3D Surface Generation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018 (Cited on pages 57 sq.).

[GFW+20] M. Guo, A. Fathi, J. Wu, and T. A. Funkhouser. "Object-Centric Neural Scene Rendering". In: *CoRR* abs/2012.08503 (2020). arXiv: 2012.08503 (Cited on page 60).

[GG07] G. Guennebaud and M. Gross. "Algebraic Point Set Surfaces". In: *ACM Transactions on Graphics (ToG)* 26.3 (2007) (Cited on page 67).

[Gil08] M. B. Giles. "Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation". In: *Advances in Automatic Differentiation*. Ed. by C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke. Springer Berlin Heidelberg, 2008, pp. 35–44 (Cited on pages 48 sq.).

[GKO+18] P. Guerrero, Y. Kleiman, M. Ovsjanikov, and N. J. Mitra. "PCPNet Learning Local Shape Properties from Raw Point Clouds". In: *Computer Graphics Forum* 37.2 (2018), pp. 75–85 (Cited on pages 65, 68 sq., 74 sq., 78, 82, 85).

[GMR17] E. Grilli, F. Menna, and F. Remondino. "A Review of Point Cloud Segmentation and Classification Algorithms". In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLII-2/W3 (02/2017), pp. 339–344 (Cited on page 65).

[GPM+14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger. Vol. 27. Curran Associates, Inc., 2014 (Cited on page 1).

[GSR+17] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2017, pp. 1263–1272 (Cited on pages 24, 28).

[GV89] G. Golub and C. Van Loan. *Matrix Computations*. 2nd. Baltimore: Johns Hopkins University Press, 1989 (Cited on page 53).

[Ham20] W. L. Hamilton. "Graph Representation Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3 (2020), pp. 1–159 (Cited on pages 24, 28, 30, 41).

[HDD+92] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. "Surface Reconstruction from Unorganized Points". In: *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '92. 1992, pp. 71–78 (Cited on pages 65, 67).

[HJ87] R. Hoffman and A. K. Jain. "Segmentation and Classification of Range Images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-9.5 (1987), pp. 608–620 (Cited on page 67).

[HKW11] G. E. Hinton, A. Krizhevsky, and S. D. Wang. "Transforming Auto-Encoders". In: *Artificial Neural Networks and Machine Learning - 21st International Conference on Artificial Neural Networks (ICANN)*. 2011, pp. 44–51 (Cited on pages 87 sq., 90).

[HLZ+19] X. Huang, Z. Liang, X. Zhou, Y. Xie, L. J. Guibas, and Q. Huang. "Learning Transformation Synchronization". In: *CoRR* abs/1901.09458 (2019) (Cited on page 68).

[HM01] J. Huang and C.-H. Menq. "Automatic data segmentation for geometric feature extraction from unorganized 3-D coordinate points". In: *IEEE Transactions on Robotics and Automation* 17.3 (2001), pp. 268–279 (Cited on page 67).

[HS97] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780 (Cited on page 1).

[HSF18] G. E. Hinton, S. Sabour, and N. Frosst. "Matrix Capsules with EM Routing". In: *International Conference on Learning Representations (ICLR)*. 2018 (Cited on pages 88, 90, 92, 109).

[HSW89] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366 (Cited on pages 12, 56).

[HV17] J. F. Henriques and A. Vedaldi. "Warped Convolutions: Efficient Invariance to Spatial Transformations". In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2017, pp. 1461–1469 (Cited on page 103).

[HW77] P. W. Holland and R. E. Welsch. "Robust regression using iteratively reweighted least-squares". In: *Communications in Statistics - Theory and Methods* 6.9 (1977), pp. 813–827. DOI: 10.1080/03610927708827533 (Cited on pages 21, 69).

[HWC+21] P. de Haan, M. Weiler, T. Cohen, and M. Welling. "Gauge Equivariant Mesh CNNs: Anisotropic convolutions on geometric graphs". In: *International Conference on Learning Representations (ICLR)*. 2021 (Cited on pages 32 sq., 90).

[HWG+13] H. Huang, S. Wu, M. Gong, D. Cohen-Or, U. Ascher, and H. (Zhang. "Edge-aware Point Set Resampling". In: *ACM Transactions on Graphics (ToG)* 32.1 (02/2013), 9:1–9:12 (Cited on page 67).

[HYL17] W. L. Hamilton, R. Ying, and J. Leskovec. "Representation Learning on Graphs: Methods and Applications". In: *IEEE Data Eng. Bull.* 40.3 (2017), pp. 52–74 (Cited on page 30).

[HZ03] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. New York, NY, USA: Cambridge University Press, 2003. ISBN: 0521540518 (Cited on pages 2, 20, 46, 48 sqq., 72).

[IL65] A. Ivakhnenko and V. Lapa. *Cybernetic Predicting Devices*. Jprs report. CCM Information Corporation, 1965 (Cited on page 1).

[IVS15] C. Ionescu, O. Vantzos, and C. Sminchisescu. "Matrix Backpropagation for Deep Networks with Structured Layers". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)* (2015), pp. 2965–2973 (Cited on page 49).

[JLK19] T. Jeong, Y. Lee, and H. Kim. "Ladder Capsule Network". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 09–15 Jun/2019, pp. 3071–3079 (Cited on page 90).

[JSZ+15] M. Jaderberg, K. Simonyan, A. Zisserman, and k. kavukcuoglu koray. "Spatial Transformer Networks". In: *Advances in Neural Information Processing Systems*. Vol. 28. 2015 (Cited on page 42).

[KAW+09] K. Klasing, D. Althoff, D. Wollherr, and M. Buss. "Comparison of surface normal estimation methods for range sensing applications". In: *2009 IEEE International Conference on Robotics and Automation*. 05/2009, pp. 3206–3211 (Cited on page 67).

[KBG19] J. Klicpera, A. Bojchevski, and S. Günnemann. "Predict then Propagate: Graph Neural Networks meet Personalized PageRank". In: *International Conference on Learning Representations (ICLR)*. 2019 (Cited on page 30).

[KBH06] M. Kazhdan, M. Bolitho, and H. Hoppe. "Poisson Surface Reconstruction". In: *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*. SGP '06. 2006, pp. 61–70 (Cited on page 65).

[KL17] R. Klokov and V. Lempitsky. "Escape From Cells: Deep Kd-Networks for the Recognition of 3D Point Cloud Models". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017 (Cited on page 116).

[KLF11] V. G. Kim, Y. Lipman, and T. Funkhouser. "Blended Intrinsic Maps". In: *ACM Transactions on Graphics (ToG)* 30.4 (07/2011), 79:1–79:12 (Cited on page 42).

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012 (Cited on page 1).

[KST+19] A. Kosiorek, S. Sabour, Y. W. Teh, and G. E. Hinton. "Stacked Capsule Autoencoders". In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019 (Cited on page 90).

[KW17] T. N. Kipf and M. Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations (ICLR)*. 2017 (Cited on page 29).

[LBB+98a] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324 (Cited on page 109).

[LBB+98b] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324 (Cited on page 11).

[LBD+89] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1 (1989), pp. 541–551 (Cited on page 1).

[LC87] W. E. Lorensen and H. E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm." In: *SIGGRAPH*. Ed. by M. C. Stone. ACM, 1987, pp. 163–169 (Cited on page 61).

[LCB10] Y. LeCun, C. Cortes, and C. Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010) (Cited on page 40).

[LDC+18] I. Lim, A. Dielen, M. Campen, and L. Kobbelt. "A Simple Approach to Intrinsic Correspondence Learning on Unstructured 3D Meshes". In: *European Conference on Computer Vision (ECCV) Workshops*. 2018 (Cited on page 33).

[LEC+07] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. "An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation". In: *Proceedings of the 24th International Conference on Machine Learning*. 2007, pp. 473–480 (Cited on page 110).

[LeC18] Y. LeCun. *Deep Learning est mort. Vive Differentiable Programming!* 2018. URL: https://www.facebook.com/yann.lecun/posts/10155003011462143 (Cited on page 11).

[Lev06] B. Levy. “Laplace-Beltrami Eigenfunctions Towards an Algorithm That "Understands" Geometry”. In: *IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)*. 2006, pp. 13–13. DOI: 10.1109/SMI.2006.21 (Cited on page 46).

[Lev98] D. Levin. “The Approximation Power of Moving Least-squares”. In: *Math. Comput.* 67.224 (10/1998), pp. 1517–1531 (Cited on pages 67 sq.).

[LFL18] J. E. Lenssen, M. Fey, and P. Libuschewski. “Group Equivariant Capsule Networks”. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 8844–8853 (Cited on pages 5 sq., 87 sq., 98, 105 sq., 110, 112 sq.).

[LFX+19] Y. Liu, B. Fan, S. Xiang, and C. Pan. “Relation-Shape Convolutional Neural Network for Point Cloud Analysis”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (Cited on page 31).

[LG16] A. Lavin and S. Gray. “Fast Algorithms for Convolutional Neural Networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 (Cited on page 17).

[LGA+18] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley. “Differentiable programming for image processing and deep learning in Halide”. In: *ACM Transactions on Graphics (ToG)* 37.4 (2018), 139:1–139:13 (Cited on pages 11, 31).

[LGZ+20] L. Liu, J. Gu, K. Zaw Lin, T.-S. Chua, and C. Theobalt. “Neural Sparse Voxel Fields”. In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 15651–15663 (Cited on page 59).

[LHL+19] X. Liu, Z. Han, Y.-S. Liu, and M. Zwicker. “Point2Sequence: Learning the Shape Representation of 3D Point Clouds with an Attention-Based Sequence to Sequence Network”. In: *The AAAI Conference on Artificial Intelligence*. AAAI Press, 2019, pp. 8778–8785 (Cited on page 116).

[Lin70] S. Linnainmaa. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors.” In Finnish. MA thesis. University of Helsinki, 1970 (Cited on page 1).

[LMW21] D. B. Lindell, J. N. P. Martel, and G. Wetzstein. “AutoInt: Automatic Integration for Fast Neural Volume Rendering”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021 (Cited on page 60).

[LNS+21] Z. Li, S. Niklaus, N. Snavely, and O. Wang. “Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021 (Cited on page 60).

[LOM20] J. E. Lenssen, C. Osendorfer, and J. Masci. “Deep Iterative Surface Normal Estimation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020 (Cited on pages 4 sqq., 42, 49 sq., 65 sq., 77 sq., 80–84).

[LRR+17] O. Litany, T. Remez, E. Rodolà, A. M. Bronstein, and M. M. Bronstein. "Deep Functional Maps: Structured Prediction for Dense Shape Correspondence". In: *IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 5660–5668 (Cited on pages 42, 51, 124).

[LSD+15] B. Li, C. Shen, Y. Dai, A. van den Hengel, and M. He. "Depth and surface normal estimation from monocular images using regression on deep features and hierarchical CRFs". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1119–1127 (Cited on page 67).

[LSS+19] S. Lombardi, T. Simon, J. Saragih, G. Schwartz, A. Lehrmann, and Y. Sheikh. "Neural Volumes: Learning Dynamic Renderable Volumes from Images". In: *ACM Transactions on Graphics (ToG)* 38.4 (07/2019), 65:1–65:14 (Cited on page 59).

[LZP14] L. Ladicky, B. Zeisl, and M. Pollefeys. "Discriminatively Trained Dense Surface Normal Estimation". In: *European Conference on Computer Vision (ECCV)*. 2014 (Cited on page 67).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: `https://www.tensorflow.org/` (Cited on page 11).

[MBB+15] J. Masci, D. Boscaini, M. M. Bronstein, and P. Vandergheynst. "Geodesic Convolutional Neural Networks on Riemannian Manifolds". In: *IEEE International Conference on Computer Vision Workshop (ICCV)*. 2015, pp. 832–840 (Cited on pages 32, 42).

[MBM+17] F. Monti, D. Boscaini, J. Masci, E. Rodolà, J. Svoboda, and M. M. Bronstein. "Geometric Deep Learning on Graphs and Manifolds Using Mixture Model CNNs". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 5425–5434 (Cited on pages 32, 41 sq.).

[MBS+19] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. "Provably Powerful Graph Networks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019 (Cited on page 30).

[MCC+07] F. L. Markley, Y. Cheng, J. L. Crassidis, and Y. Oshman. "Averaging quaternions". In: *Journal of Guidance, Control, and Dynamics* 30.4 (2007), pp. 1193–1197 (Cited on pages 56, 114).

[MGD+10] P. Mullen, F. de Goes, M. Desbrun, D. Cohen-Steiner, and P. Alliez. "Signing the Unsigned: Robust Surface Reconstruction from Raw Pointsets". In: *Computer Graphics Forum* 29.5 (2010), pp. 1733–1741 (Cited on page 68).

[MHL14] M. Mathieu, M. Henaff, and Y. LeCun. "Fast Training of Convolutional Networks through FFTs". In: *International Conference on Learning Representations (ICLR)*. Ed. by Y. Bengio and Y. LeCun. 2014 (Cited on page 17).

[MNG04] N. J. Mitra, A. Nguyen, and L. Guibas. "Estimating Surface Normals in Noisy Point Cloud Data". In: *International Journal of Computational Geometry and Applications*. Vol. 14. 4–5. 2004, pp. 261–276 (Cited on page 67).

[MOG11] Q. Merigot, M. Ovsjanikov, and L. J. Guibas. "Voronoi-Based Curvature and Feature Estimation from Point Clouds". In: *IEEE Transactions on Visualization and Computer Graphics* 17.6 (2011), pp. 743–756 (Cited on page 67).

[MON+19] L. Mescheder, M. Oechsle, M. Niemeyer, S. Nowozin, and A. Geiger. "Occupancy Networks: Learning 3D Reconstruction in Function Space". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (Cited on pages 57 sqq.).

[MRF+19] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. "Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks". In: *The AAAI Conference on Artificial Intelligence*. AAAI Press, 2019, pp. 4602–4609 (Cited on pages 5 sq., 30).

[MSC21] V. Mazzia, F. Salvetti, and M. Chiaberge. "Efficient-CapsNet: capsule network with self-attention routing". In: *Scientific reports* 11 (2021) (Cited on page 90).

[MSR+19] R. L. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro. "Relational Pooling for Graph Representations". In: *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2019, pp. 4663–4673 (Cited on page 30).

[MST+20] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Vol. 12346. Lecture Notes in Computer Science. Springer, 2020, pp. 405–421 (Cited on pages 2, 59).

[Mur13] K. P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, 2013 (Cited on page 13).

[MVK+17] D. Marcos, M. Volpi, N. Komodakis, and D. Tuia. "Rotation Equivariant Vector Field Networks". In: *IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 5058–5067 (Cited on page 90).

[NF12] P. K. Nathan Silberman Derek Hoiem and R. Fergus. "Indoor Segmentation and Support Inference from RGBD Images". In: *European Conference on Computer Vision (ECCV)*. 2012 (Cited on page 83).

[NMO+20] M. Niemeyer, L. Mescheder, M. Oechsle, and A. Geiger. "Differentiable Volumetric Rendering: Learning Implicit 3D Representations without 3D Supervision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020 (Cited on page 59).

[OBS+12] M. Ovsjanikov, M. Ben-Chen, J. Solomon, A. Butscher, and L. Guibas. "Functional Maps: A Flexible Representation of Maps between Shapes". In: *ACM Transactions on Graphics (ToG)* 31.4 (2012) (Cited on pages 51, 124).

[Ola15] C. Olah. *Neural Networks, Types, and Functional Programming*. 2015. URL: http://colah.github.io/posts/2015-09-NN-Types-FP/ (Cited on page 11).

[OMT+21] J. Ost, F. Mannan, N. Thuerey, J. Knodt, and F. Heide. "Neural Scene Graphs for Dynamic Scenes". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021 (Cited on page 60).

[PCS15] F. Pomerleau, F. Colas, and R. Siegwart. "A Review of Point Cloud Registration Algorithms for Mobile Robotics". In: *Found. Trends Robot* 4.1 (05/2015), pp. 1–104 (Cited on page 65).

[PD12] A. Petrelli and L. Di Stefano. "A Repeatable and Efficient Canonical Reference for Surface Matching". In: *Second International Conference on 3D Imaging, Modeling, Processing, Visualization Transmission*. 2012, pp. 403–410 (Cited on page 115).

[Pea01] K. Pearson. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572 (Cited on page 46).

[PFS+19] J. J. Park, P. Florence, J. Straub, R. A. Newcombe, and S. Lovegrove. "DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 165–174 (Cited on pages 57 sq., 60, 124).

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019 (Cited on pages 11, 15, 26, 56).

[PKK19] I. Paik, T. Kwak, and I. Kim. "Capsule Networks Need an Improved Routing Algorithm". In: *CoRR* abs/1907.13327 (2019). arXiv: 1907.13327 (Cited on page 90).

[PL00] T. Papadopoulo and M. I. A. Lourakis. "Estimating the Jacobian of the Singular Value Decomposition: Theory and Applications". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer, 2000, pp. 554–570 (Cited on page 49).

[PSB+20] K. Park, U. Sinha, J. T. Barron, S. Bouaziz, D. B. Goldman, S. M. Seitz, and R. Martin-Brualla. "Deformable Neural Radiance Fields". In: *CoRR* abs/2011.12948 (2020). arXiv: 2011.12948 (Cited on page 60).

[PT97] L. Piegl and W. Tiller. *The NURBS Book*. Springer-Verlag New York, Inc, 1997 (Cited on pages 34, 36 sq.).

[QLL+18] X. Qi, R. Liao, Z. Liu, R. Urtasun, and J. Jia. "GeoNet: Geometric Neural Network for Joint Depth and Surface Normal Estimation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 283–291 (Cited on page 67).

[QSK+17] C. R. Qi, H. Su, M. Kaichun, and L. 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)*. 2017, pp. 77–85 (Cited on pages 30, 68, 116).

[QYS+17] C. R. Qi, L. Yi, H. Su, and L. J. Guibas. "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space". In: *Advances in Neural Information Processing Systems (NIPS)*. 2017 (Cited on pages 30, 43, 68, 116).

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Ed. by D. E. Rumelhart and J. L. Mcclelland. Cambridge, MA: MIT Press, 1986, pp. 318–362 (Cited on page 1).

[RJY+20] D. Rebain, W. Jiang, S. Yazdani, K. Li, K. M. Yi, and A. Tagliasacchi. "DeRF: Decomposed Radiance Fields". In: *CoRR* abs/2011.12490 (2020). arXiv: 2011.12490 (Cited on page 59).

[RK18] R. Ranftl and V. Koltun. "Deep Fundamental Matrix Estimation". In: *European Conference on Computer Vision (ECCV)*. 2018 (Cited on pages 49 sq., 69, 72).

[RLK20] F. D. S. Ribeiro, G. Leontidis, and S. D. Kollias. "Capsule Routing via Variational Bayes." In: *The AAAI Conference on Artificial Intelligence*. 2020, pp. 3749–3756 (Cited on page 90).

[RM98] V. Reddy and M. Mavrovouniotis. "An Input-Training Neural Network Approach for Gross Error Detection and Sensor Replacement". In: *Chemical Engineering Research and Design* 76.4 (1998). Process Operations and Control, pp. 478–489 (Cited on page 57).

[Ros58] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65.6 (1958), pp. 386–408 (Cited on page 1).

[SAC+20] N. Sharp, S. Attaiki, K. Crane, and M. Ovsjanikov. "Diffusion is All You Need for Learning on Surfaces". In: *CoRR* abs/2012.00888 (2020) (Cited on pages 51, 124).

[Sch15] J. Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural Networks* 61 (2015), pp. 85–117 (Cited on page 1).

[SCW+15] X. SHI, Z. Chen, H. Wang, D.-Y. Yeung, W.-k. Wong, and W.-c. WOO. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015 (Cited on page 2).

[SDZ+21] P. P. Srinivasan, B. Deng, X. Zhang, M. Tancik, B. Mildenhall, and J. T. Barron. "NeRV: Neural Reflectance and Visibility Fields for Relighting and View Synthesis". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021 (Cited on page 60).

[SFH17] S. Sabour, N. Frosst, and G. E. Hinton. "Dynamic Routing Between Capsules". In: *Advances in Neural Information Processing Systems (NIPS)*. 2017, pp. 3859–3869 (Cited on pages 87 sq., 90, 92, 97, 108, 110 sq.).

[SK17] M. Simonovsky and N. Komodakis. “Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 29–38 (Cited on page 31).

[SLN+20] K. Schwarz, Y. Liao, M. Niemeyer, and A. Geiger. “GRAF: Generative Radiance Fields for 3D-Aware Image Synthesis”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020 (Cited on page 60).

[SMB+20] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein. “Implicit Neural Representations with Periodic Activation Functions”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 7462–7473 (Cited on page 59).

[SRC+20] Z. Sun, E. Rooke, J. Charton, Y. He, J. Lu, and S. Baek. “ZerNet: Convolutional Neural Networks on Arbitrary Surfaces Via Zernike Local Tangent Space Estimation”. In: *Computer Graphics Forum* 39.6 (2020), pp. 204–216 (Cited on page 32).

[SS21] D. Smirnov and J. M. Solomon. “HodgeNet: Learning Spectral Geometry on Triangle Meshes”. In: *CoRR* abs/2104.12826 (2021). arXiv: 2104.12826 (Cited on page 124).

[SST+18] S. Suwajanakorn, N. Snavely, J. J. Tompson, and M. Norouzi. “Discovery of Latent 3D Keypoints via End-to-end Geometric Reasoning”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2018, pp. 2063–2074 (Cited on page 49).

[SZW19] V. Sitzmann, M. Zollhoefer, and G. Wetzstein. “Scene Representation Networks: Continuous 3D-Structure-Aware Neural Scene Representations”. In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019 (Cited on page 59).

[TH12] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012 (Cited on page 75).

[TM95] S. Tan and M. L. Mayrovouniotis. “Reducing data dimensionality through optimizing neural network inputs”. In: *AIChE Journal* 41.6 (1995), pp. 1471–1480 (Cited on page 57).

[Tow16] J. Townsend. *Differentiating the Singular Value Decomposition*. 2016. URL: https://j-towns.github.io/papers/svd-derivative.pdf (Cited on page 49).

[TQD+19] H. Thomas, C. R. Qi, J.-E. Deschaud, B. Marcotegui, F. Goulette, and L. J. Guibas. “KPConv: Flexible and Deformable Convolution for Point Clouds”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 10/2019 (Cited on page 31).

[VBV18] N. Verma, E. Boyer, and J. Verbeek. “FeaStNet: Feature-Steered Graph Convolutions for 3D Shape Analysis”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018 (Cited on page 31).

[VCC+18] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. “Graph Attention Networks”. In: *International Conference on Learning Representations (ICLR)*. 2018 (Cited on page 29).

[WDH+19] W. Wang, Z. Dang, Y. Hu, P. Fua, and M. Salzmann. "Backpropagation-Friendly Eigendecomposition". In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019 (Cited on page 51).

[WDW+18] F. Wang, J. Decker, X. Wu, G. Essertel, and T. Rompf. "Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming". In: *Advances in Neural Information Processing Systems*. Vol. 31. 2018 (Cited on pages 11, 31).

[WEH20] R. Wiersma, E. Eisemann, and K. Hildebrandt. "CNNs on Surfaces Using Rotation-Equivariant Features". In: *ACM Transactions on Graphics (ToG)* 39.4 (2020) (Cited on pages 32 sq., 90).

[Wer81] P. J. Werbos. "Applications of Advances in Nonlinear Sensitivity Analysis". In: *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*. 1981, pp. 762–770 (Cited on page 1).

[WFG15] X. Wang, D. F. Fouhey, and A. Gupta. "Designing deep networks for surface normal estimation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 539–547 (Cited on page 67).

[WFV+21] M. Weiler, P. Forr'e, E. Verlinde, and M. Welling. "Coordinate Independent Convolutional Networks - Isometry and Gauge Equivariant Convolutions on Riemannian Manifolds". In: *CoRR* abs/2106.06020 (2021). arXiv: 2106.06020 (Cited on page 90).

[WGT+17] D. E. Worrall, S. J. Garbin, D. Turmukhambetov, and G. J. Brostow. "Harmonic Networks: Deep Translation and Rotation Equivariance". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017 (Cited on page 90).

[WHG+15] S. Wu, H. Huang, M. Gong, M. Zwicker, and D. Cohen-Or. "Deep Points Consolidation". In: *ACM Transactions on Graphics (ToG)* 34.6 (2015), 176:1–176:13 (Cited on pages 68, 116 sq.).

[WHS18] M. Weiler, F. A. Hamprecht, and M. Storath. "Learning Steerable Filters for Rotation Equivariant CNNs". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018 (Cited on pages 90, 111).

[WL68] B. Weisfeiler and A. A. Lehman. "A Reduction of a Graph to a Canonical Form and an Algebra Arising During this Reduction". In: *Nauchno-Technicheskaya Informatsia* 2.9 (1968) (Cited on page 30).

[WPC+21] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), pp. 4–24 (Cited on page 30).

[WSL+19] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. "Dynamic Graph CNN for Learning on Point Clouds". In: *ACM Transactions on Graphics (TOG)* (2019) (Cited on pages 30 sq., 116).

[XFX+18] Y. Xu, T. Fan, M. Xu, L. Zeng, and Y. Qiao. "SpiderCNN: Deep Learning on Point Sets with Parameterized Convolutional Filters". In: *European Conference on Computer Vision (ECCV)*. 2018 (Cited on page 31).

[XHL+19] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. “How Powerful are Graph Neural Networks?” In: *International Conference on Learning Representations (ICLR)*. 2019 (Cited on page 30).

[XLW20] H. Xie, J.-G. Liu, and L. Wang. “Automatic differentiation of dominant eigensolver and its applications in quantum physics”. In: *Phys. Rev. B* 101 (24 06/2020), p. 245139 (Cited on pages 51 sq.).

[YHM+18] W. Yuan, D. Held, C. Mertz, and M. Hebert. “Iterative Transformer Network for 3D Point Cloud”. In: *CoRR* abs/1811.11209 (2018). arXiv: 1811.11209 (Cited on pages 117 sq.).

[YKM+20] L. Yariv, Y. Kasten, D. Moran, M. Galun, M. Atzmon, B. Ronen, and Y. Lipman. “Multiview Neural Surface Reconstruction by Disentangling Geometry and Appearance”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 2492–2502 (Cited on page 59).

[YLL+20] Y. You, Y. Lou, Q. Liu, Y.-W. Tai, L. Ma, C. Lu, and W. Wang. “Pointwise Rotation-Invariant Network with Adaptive Sampling and 3D Spherical Voxel Convolution”. In: *The AAAI Conference on Artificial Intelligence*. AAAI Press, 2020, pp. 12717–12724 (Cited on page 116).

[YQQ+17] Z. Yanzhao, Y. Qixiang, Q. Qiang, and J. Jianbin. “Oriented Response Networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 4961–4970 (Cited on page 90).

[YTO+18] K. M. Yi, E. Trulls, Y. Ono, V. Lepetit, M. Salzmann, and P. Fua. “Learning to Find Good Correspondences”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018 (Cited on pages 49 sq.).

[ZBD+19] Y. Zhao, T. Birdal, H. Deng, and F. Tombari. “3D Point Capsule Networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (Cited on page 90).

[ZBL+20] Y. Zhao, T. Birdal, J. E. Lenssen, E. Menegatti, L. J. Guibas, and F. Tombari. “Quaternion Equivariant Capsule Networks for 3D Point Clouds”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Vol. 12346. Springer, 2020, pp. 1–19 (Cited on pages 5 sq., 56, 87 sq., 111, 115–118).

[ZK13] Q.-Y. Zhou and V. Koltun. “Dense scene reconstruction with points of interest”. In: *ACM Transactions on Graphics (ToG)* 32.4 (2013), pp. 1–8 (Cited on page 62).

[ZRS+20] K. Zhang, G. Riegler, N. Snavely, and V. Koltun. “NERF++: Analyzing and Improving Neural Radiance Fields”. In: *CoRR* abs/2010.07492 (2020). arXiv: 2010.07492 (Cited on page 59).