

Machine Learning for Assembly Modeling in Computer-Aided Design

Dissertation
Zur Erlangung des Doktorgrades Dr. rer. nat.

Institut für Software & Systems Engineering
Fakultät für Angewandte Informatik
Universität Augsburg

Carola Anna Lenzen



Machine Learning for Assembly Modeling in Computer-Aided Design

Reviewers: Prof. Dr. Wolfgang Reif
Prof. Dr. Bernhard Bauer
Prof. Dr. Alexander Schiendorfer

Day of Defense: December 09, 2024

Abstract

In the domain of assembly modeling, design engineers utilize computer-aided design (CAD) software to develop complex assemblies. However, the increasing volume and size of non-standardized part catalogs from various manufacturers pose significant challenges in the selection of suitable parts from the vast number of options. This thesis addresses these challenges by proposing a novel methodology that leverages machine learning techniques to support inexperienced designers in the assembly design process by extracting expert knowledge from previous assemblies. Specifically, it investigates the use cases of global part recommendation, localized part recommendation and handling anomalies in assemblies.

The first contribution is a generic, data-driven approach to extract patterns of proven part combinations across multiple real-world assemblies. The core methodology utilizes a graph-based representation of assemblies, wherein parts are represented as nodes and their connections as edges. Leveraging this representation, the methodology provides means and guidelines for modeling arbitrary part recommendation tasks by applying graph machine learning.

As second contribution, we developed an embedding technique to learn the similarity of parts in terms of their usage across multiple assemblies. This technique adapts a method from the field of natural language processing to general graph structures. The resulting embeddings provide features for the parts in all learning tasks to enhance the models' generalization capabilities.

The third contribution comprises an automated approach to generate learning instances for self-supervised machine learning from assembly data which can be tailored to a specific learning task on assemblies. This method addresses the scarcity of labeled data in real-world assembly datasets without involving domain experts. Moreover, this thesis introduces an algorithm for generating synthetic anomalous assemblies by extracting regular part combinations from a dataset of assemblies.

Finally, this thesis is the first to address part recommendation during assembly modeling by analyzing previous assemblies through machine learning. It proposes a general framework for a recommendation task, modeled as a classification problem to generate a fixed number of recommendations. The experimental results across all use cases demonstrate that machine learning-based methods can greatly enhance the efficiency of assembly modeling, improve knowledge transfer among engineers, and reduce time for perusing extensive part catalogs.

Acknowledgments

First, I would like to express my gratitude to my doctoral supervisor Prof. Dr. Wolfgang Reif for his guidance and support throughout the years, the freedom he gave me in shaping my research topic and the excellent technical equipment he provided. I am also grateful for the critical questions in our discussions and the different perspectives he pointed out. In addition, I would like to thank Prof. Dr. Bernhard Bauer and Prof. Dr. Alexander Schiendorfer for their work as reviewers of this thesis.

I am very grateful to both Prof. Dr. Alexander Knapp and Dr. Gerhard Schellhorn for their contributions to the quality assurance of this thesis. I would like to thank Dr. Stefan Bodenmüller in particular for his persistent proofreading work. Additional thanks go to Dr. Constantin Wanninger for sharing his expertise in CAD design and his support in revising the figures of this thesis.

Many thanks to all former and current colleagues at the Institute for Software & Systems Engineering who have created a workplace where I have enjoyed working. The inspiring discussions and amusing conversations during the lunch breaks enriched my working days. My special thanks go to Prof. Dr. Alexander Schiendorfer, who guided me during the first years of my doctoral studies.

I am especially grateful to the students that have contributed to the research project KOGNIA and this thesis. In particular, I would like to thank Maximilian Jannack for his work on programming the Part Embedding Refiner, Michael Huber for his contribution to the pretraining of part recommendation models, and Leon Toplak for his preliminary work on exploring strategies for localized part recommendation.

My special thanks to my family and friends for continuously supporting me, be it through motivating conversations or the necessary distraction in challenging situations. Finally, I would like to express my deepest gratitude to my loving husband Philip who motivated, supported and believed in me throughout the entire period. His support and suggestions have contributed significantly to the success of this thesis.

Carola Lenzen

Contents

1	Introduction	1
1.1	Challenges in Assembly Modeling	2
1.2	Goals of this Thesis	3
1.3	Research Project KOGNIA	5
1.4	Scientific Contribution	5
1.5	Thesis Outline	7
2	Computer-Aided Design Foundations	9
2.1	Part Modeling	11
2.2	Assembly Modeling	12
2.3	Enhancing Reusability in Assembly Modeling via Part Catalogs	14
3	Data and Machine Learning Foundations	19
3.1	Basic Concepts of Machine Learning	20
3.1.1	Artificial Neural Networks	24
3.1.2	Deep Learning	27
3.2	The Graph Data Structure	28
3.3	Graph Machine Learning	32
3.3.1	Categorization of Learning Tasks on Graphs	33
3.3.2	Graph Neural Networks	35
3.4	Embedding Discrete Objects	44
3.4.1	Learning Embeddings	45
3.4.2	Word2Vec	45
4	Assembly Data	51
4.1	Representing CAD Assemblies as Graphs	52
4.2	Assembly Datasets	54
4.3	Unsupervised Pretraining of Part Embeddings	56
4.3.1	Experimental Results	59
4.3.2	Further Uses for Trained Embeddings	60
4.3.3	Related Work	64

5	Part Recommendation in Assembly Modeling	67
5.1	Generating Recommendation Instances from Assemblies	70
5.2	Baselines and Upper Bound	75
5.3	Experimental Setup	76
5.4	Experimental Results	78
5.5	Digressions	88
5.5.1	Pretraining of Part Recommendation Models	88
5.5.2	Graph Transformers for Part Recommendation	89
5.5.3	Context-Specific Number of Recommendations	90
5.6	Related Work	92
6	Localized Recommendation By Targeted Part Placement	95
6.1	Generating Instances From Assemblies	98
6.2	Variant I: Recommending Part Types for User-Given Extension Point . . .	100
6.2.1	Experimental Setup	101
6.2.2	Experimental Results	102
6.3	Variant II: Recommending Extension Point and Part Type	105
6.3.1	Modeling Approach (1): First Predicting Part Type, Afterward Ex- tension Point	106
6.3.2	Modeling Approach (2): First Predicting Extension Point, Afterward Part Type	109
6.3.3	Experimental Setup	110
6.3.4	Experimental Results	112
6.4	Related Work	116
7	Handling Anomalies in Assemblies	119
7.1	Generating Synthetic Anomalous Assemblies	121
7.2	Detecting Anomalous Parts in Assemblies	126
7.2.1	Experimental Setup	127
7.2.2	Experimental Results	128
7.3	Recommending Alternative Parts	130
7.3.1	Experimental Setup	133
7.3.2	Experimental Results	133
7.4	Related Work	136
8	Summary and Outlook	139
8.1	Summary	139
8.2	Lessons Learned	143
8.3	Outlook	144
	Bibliography	147

Introduction

Summary This chapter sets the stage for this thesis by outlining the challenges in assembly modeling, particularly in selecting appropriate parts from extensive part catalogs. It presents the goals and use cases addressed by the research, which include leveraging machine learning techniques to extract design knowledge from past assembly data to improve part selection processes by recommending parts next needed or analyzing assemblies in terms of anomaly during the design process. The chapter also introduces the research project KOGNIA which provided the inspiration for the problems addressed in this thesis. Finally, it details the scientific contributions that this thesis offers and concludes with an outline of its contents.

Contents

1.1	Challenges in Assembly Modeling	2
1.2	Goals of this Thesis	3
1.3	Research Project KOGNIA	5
1.4	Scientific Contribution	5
1.5	Thesis Outline	7

Engineering drawings of parts and assemblies have been an integral component of the manufacturing industry since the First Industrial Revolution [107]. By the Third Industrial Revolution, however, the drawings had become so complex that they were error-prone and time-consuming to create, both of which impacting on revenue. Consequently, companies started to leverage software for *computer-aided design (CAD)* to reduce costs and creation time for their designs [10]: Sketchpad, an early design system for technical drawings with a graphical user interface, allowed to draft 2D designs on a computer screen just like with pencil and paper. Additionally, designers were already able to move and modify objects on the screen, which allowed to refine and alter details without the need for mechanical erasers. Once CAD software became affordable (e.g., ADAM, 1971, was designed to work on all mainframe computers) and introduced solid modeling capabilities in 3D (Synthavision, 1972), it revolutionized the way engineering designs were created: CAD systems allow to easily create, copy, or modify parts of the designs, to store them for later use in other models, and to scale them automatically without dimensional error. This not only improves quality of the design, but also speeds the design process up by ten times compared to technical drawing by hand [107]. Latest with the introduction of CAD systems for common

Personal Computers (AutoCAD, 1982), they became the standard way of creating technical designs across numerous industries, such as automotive, shipbuilding, or aerospace [119].

As the complexity of product designs was – and still is [113] – continuously increasing, it eventually became more cost and time efficient to create designs based on the same generic parts [4]. On the one hand, manufacturing departments could optimize tooling and equipment specifically for producing and assembling those parts. On the other hand, design engineers could focus on designing the assembly instead of creating customized part designs from scratch – a shift from part modeling to *assembly modeling*. This eventually led to the rise of highly specialized part manufacturers offering mass-produced standard parts and providing CAD models of these parts to their customers [4]. In turn, industrial companies refrained more and more from manufacturing customized parts in-house and instead purchased such standard parts selected from *part catalogs* of the part manufacturers. As the mass-produced parts could be manufactured both at fewer costs and with better quality assurance, this in turn reduced the time-to-market and overall costs of products: The production effort shrank to mainly assembling the final product from standard parts instead of manufacturing all parts of the product entirely in-house.

Nowadays, numerous manufacturers produce parts in standardized sizes and specifications that can be used across multiple applications, such as screws, bearings and motors [50]. Each of them maintains their very own part catalogs, and the number as well as the volume of those is still increasing [113]. This poses a challenge for design engineers [87, 35]: They have to select the one part which fits best for their assembly out of thousands of more or less similar parts. And while they search for possible candidates, they typically have to peruse multiple part catalogs from different manufacturers which are difficult to compare: As there is no standardized format for the catalogs, part manufacturers use proprietary formats which often differ in detail and presentation of the parts.

1.1 Challenges in Assembly Modeling

At the time when individual parts for each product or machine were still custom-designed and manufactured, it was neither necessary nor possible to transfer knowledge about compatible combinations of parts to the next product. However, this changed with the rise of available parts catalogs, as the same parts can be reused in several designs since then. Nowadays, there are multiple suppliers of parts catalogs and their number and volume are constantly increasing [113]. Consequently, designers have to go through many options when searching for a suitable part during assembly modeling, which is very time-consuming and a source of error for inexperienced designers in particular. According to a survey of design engineers from multiple companies [44], they spend almost an hour a day searching for suitable parts in the catalogs available to them and the same amount of time for modeling the required part themselves if their search was unsuccessful.

Furthermore, knowledge of compatible part combinations is an essential factor in selecting the best-suited part from all candidates for a new assembly. Once a suitable combination of parts has been found, a designer can transfer this knowledge to assemblies for comparable application problems which require a similar design. For example, in contract manufacturing or special machine construction, customers typically commission slight product adaptations to solve comparable yet slightly different problems, which allows experienced designers to adapt solutions they have found for similar problems in the past [81, 74]. However, although experienced designers have gained knowledge of proven

part combinations over the course of their careers, this knowledge must be preserved and made accessible to other designers, especially inexperienced ones; otherwise, they are forced to reinvent the wheel once the experts leave the company.

Current numbers prove that the challenge of knowledge transfer between design engineers becomes increasingly important: First, the number of experienced engineers is declining due to the retirement of baby boomer engineers [32]. In addition, temporary workers hired to cover high order volumes may have design experience but are not familiar with the specific parts used in the current company and therefore have little advantage over inexperienced designers in finding suitable parts. Furthermore, knowledge transfer often faces significant challenges, for instance because the transfer is deliberately withheld, or the knowledge is hard to formalize [149]. For example, by hoarding knowledge, individuals may seek to make themselves indispensable to the company. However, even if there is a willingness to share expertise, it cannot always be clearly communicated, even if it stems from years of experience. Only a limited part of human knowledge can be consciously retrieved and actively communicated [142] as it is deeply embedded.

There have already been efforts to support a designer in finding a part effectively to overcome the challenge posed by the enormous variety of available parts [91, 45]. This includes the classic search for impact terms or designations, which becomes more challenging due to inconsistent designations and heterogeneous organization of the parts across multiple catalogs. Other techniques focus more on the geometry of the parts so that, for example, parts of similar shape can be searched for by providing a reference part [13] or a drawn 2D or 3D sketch of the envisioned part [45]. However, the engineer must have the right part in mind or have the right idea in order to come up with the right part.

Unfortunately, retrieving this particular knowledge, i.e., formalizing expert knowledge about part combinations and transferring it to inexperienced design engineers, turned out to be a time-consuming and error-prone task if carried out manually [9]. Instead, it would be beneficial to have a generic approach that is applicable to any collection of assemblies and parts, and automatically extracts expert knowledge from existing designs, i.e., without the necessity of manually modeling design knowledge. A support system built on this approach could educate inexperienced designers regarding suitable combinations and eventually transfer the expert knowledge. Ideally, such a system also reduces the search space for parts and thus solves the challenge arising from the variety of available parts, too.

1.2 Goals of this Thesis

We now translate the challenges outlined above into specific problems that designers encounter within the area of assembly modeling, which we will address as *use cases* in the context of this work. The main challenge we focus on is the difficulty in easily or automatically sharing design knowledge about good part combinations from previous designs between designers. We therefore aim at educating inexperienced designers by providing the design knowledge of experts with an assistance system. Precisely, we want to provide recommendations for next required parts for the current state of the assembly. For example, the recommendations can comprise a list of suitable parts for the current assembly, from which the designer can select the desired one. To obtain them, however, we need to derive knowledge about suitable parts from part combinations which appear in similar assemblies. Finally, the assistant system should also assess the quality of the current assembly and indicate unusual part combinations. As they rarely appear or not at all in

existing designs, they are more likely to be inappropriate for the intended application. Note that such a system primarily supports novice designers and can thus shorten their learning phase, but also facilitates knowledge transfer among experienced designers, enabling them to benefit from the insights of their peers. Decisive factors for the realization of such an assistant system are the decreasing number of designers, which limits the pool of potential contributors, as well as the desire to minimize additional work for the remaining designers.

In summary, we primarily aim to support inexperienced designers in assembly modeling by extracting implicit design knowledge from existing assemblies through following an automated approach that minimizes the involvement of domain experts. In order to tackle the challenges in assembly modeling presented above, we identified three main use cases to support engineers during assembly design:

- The first use case comprises *global part recommendation* during design, i.e., recommendation of next, directly attachable parts to the current assembly. Such a recommendation system based on experienced design knowledge would already utilize design knowledge from earlier assemblies.
- However, especially inexperienced designers would benefit from additional support in locating the recommended parts within the current design, i.e., where to attach the recommended parts, which we refer to as *localized part recommendation*. For example, either the assistance system could recommend both a next part and the part from the assembly it could be attached to, or design engineers could select an existing part to receive targeted suggestions of suitable parts for it.
- Finally, the third use case deals with quality assessment of assemblies. Unusual part uses or combinations may result from a lack of knowledge about or unsuccessful searches for suitable parts. Therefore, an assistant system should be capable of *handling anomalies in assemblies*, i.e., it should identify anomalous parts and support in correcting them by suggesting alternatives.

In general, we assume that the experienced designers' knowledge is contained in their designs as these involve information on both the parts used and the combinations of parts to solve a specific application problem. Therefore, we have to extract this implicit knowledge in the form of recurring patterns from these designs and to make it available to other designers after processing. To make design knowledge explicit, several methods are available. For instance, in his thesis [9], Stefan Bartsch developed a part recommendation system that assists designers using explicitly modeled design knowledge. The work revealed that precise recommendations, even for a few considered parts, can be realized only with high manual effort on modeling the design knowledge. This approach is therefore unsuitable for handling large quantities of parts and contradicts our goal to minimize additional designers' work. Further, due to the heterogeneity of information in different parts catalogs and CAD systems, our approach should only rely on universally available information.

Instead of a specialized, manual approach, we therefore aim at developing a *generic, data-driven approach* leveraging artificial intelligence, which applies to any assembly data. This aligns with the ongoing trend of evolving computer-aided processes like designing, engineering and manufacturing (referred to CAx) into artificial intelligence-aided processes (AIAx) [28, 57, 160, 167]. In particular, by providing a data-driven support for designers during assembly modeling, the assistance system belongs to the category of AI-aided design (AIAD).

Concretely, we aim at extracting universally available information inherently embedded within the designs themselves: the parts used and their combination within assemblies. To do so, we leverage machine learning techniques. This sub-area of artificial intelligence deals with the extraction of patterns from data. In our case, we extract recurring patterns representing design knowledge from existing assemblies. As assemblies can be naturally represented as graphs, we employ *graph neural networks*, a specific form of neural networks designed to process graph structures. To automate the approach, we use techniques of automatic generation of labeled data – called *self-supervision* – which obviates manual labeling of assembly data by designers.

1.3 Research Project KOGNIA

The inspiration for addressing the challenges in assembly modeling described above originates from the research project KOGNIA¹ [46, 66]. It aims to support CAD designers in the design process, particularly in assembly modeling. CADENAS GmbH, the project’s industrial partner, specializes in developing software solutions for engineering, including a part management system that assists engineers manage and find parts, such as by searching for geometrically similar parts based on a sketch [45].

As a first simple attempt to the project, the above-mentioned thesis [9] addressed the development of an assistance system for designers based on modeled knowledge. It revealed that high prediction accuracy requires a great deal of effort for only a few parts. Consequently, the project’s focus was set on a generic, data-driven approach using machine learning.

The assembly data examined in this thesis were provided by KOGNIA, and the majority of the research results on part recommendation were achieved within the scope of this initiative, including the development of a demonstrator presented in Chapter 5. A press release [135] summarizes the most significant results achieved upon the project’s completion.

1.4 Scientific Contribution

This thesis contributes to the field of artificial intelligence-aided design and focuses on the application of graph machine learning to assembly modeling. The relevance of its contributions has been recognized by the scientific community, as they already serve as the basis for further research in this area (e.g., [89]). In particular, this thesis makes the following main contributions:

Methodology to Apply Graph Machine Learning to Assembly Modeling This thesis provides an automated, generic approach to extract design knowledge from assemblies. Knowledge about which parts have to be combined in order to achieve the overall purpose of an assembly is implicitly encoded by the used parts as well as the connections between them. Consequently, the assemblies are transformed into a graph representation based on the connections between the assembled parts. The methodology provides means

¹Project KOGNIA (*Konstruktionsunterstützung durch künstliche Intelligenz und automatisiertes maschinelles Lernen*) has been funded by the Bavarian Ministry of Economic Affairs, Regional Development and Energy (StMWi).

and guidelines for extracting implicit design knowledge from those graphs, and for modeling a machine learning task leveraging this knowledge based on arbitrary graph neural network architectures.

Embedding Representation of Assembly Parts An important aspect for generalization capabilities of machine learning models in the domain of assembly modeling is the similarity of parts in terms of their usage. Thus, this thesis gives an embedding technique for assembly graphs inspired by natural language processing in order to grasp part similarity based on frequent combinations with other parts. As parts occurring infrequently in a given assembly dataset may obtain a poor embedding, an interactive editor to refine embeddings based on user-given similarity constraints is provided.

Data Augmentation Approach for Instance Generation from Assemblies Real-world assembly datasets typically comprise only final assemblies without labels, and usually not enough to serve as training instances. This thesis contributes an automated approach to generate instances for self-supervised machine learning from assembly datasets. It increases the number of instances through data augmentation and is customizable with regard to the shape of instances required for a given learning task.

Models for Part Recommendation Tasks This thesis is the first to employ machine learning techniques for recommending parts based on extracted design knowledge from previous assemblies. It provides reference models for part recommendation tasks in which new parts are to be added to an assembly or existing parts are to be replaced. Notably, a subset of the models is capable of incorporating a given location within the assembly, others can recommend a suitable location for a part. To assess the performance of part recommendation models, novel baseline and upper bound models are given. In addition, a reference model was also applied to detect anomalous parts in assemblies. The proposed models were carefully evaluated on real-world assemblies and consistently achieved excellent results in all use cases.

Publications The following publications were created in the context of the work presented in this thesis.

1. Carola Lenzen, Alexander Schiendorfer, and Wolfgang Reif. Graph Machine Learning for Assembly Modeling. In *Learning on Graphs Conference (LoG)*, 2022
2. Carola Gajek, Alexander Schiendorfer, and Wolfgang Reif. A Recommendation System for CAD Assembly Modeling based on Graph Neural Networks. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2022
3. Carola Lenzen and Wolfgang Reif. Localized Recommendation in Assembly Modeling: Employing GNNs for Targeted Part Placement. In *International Conference on Machine Learning and Applications (ICMLA)*, 2024
4. Carola Lenzen, Vinzenz Löffel, and Wolfgang Reif. Handling Anomalies in CAD Assemblies: Detecting Anomalous and Suggesting Alternative Parts. In *International Conference on Computational Science and Computational Intelligence (CSCI)*, 2024

1.5 Thesis Outline

The remainder of this thesis is structured as follows:

The subsequent two chapters lay the necessary foundations for this thesis: Chapter 2 provides an overview of the application domain of this thesis, namely computer-aided design (CAD), and covers various types of modeling, including part modeling and assembly modeling. The chapter emphasizes the impact of standardized, mass-produced parts on designing and their integration into CAD systems organized in so-called part catalogs to streamline the assembly design process.

Chapter 3 introduces the reader to fundamental machine learning concepts relevant to this thesis. It covers the definition of graph structures and machine learning techniques on graphs. Thereafter, it focuses on graph neural networks (GNNs), a special form of neural networks designed for processing graphs, and highlights four architectures in detail. The chapter concludes with a consideration of methods for embedding discrete objects, such as parts in assemblies, and presents word2vec [96] as a learning technique from natural language processing, which is not based on specifics of natural language.

Chapters 4 to 7 constitute the main contributions of this work.

Chapter 4 introduces the representation of CAD assemblies as undirected graphs over parts and presents the three assembly datasets examined in this thesis. These are real-world datasets, as the designs originate from orders submitted to specific configurators for assemblies. Further, it presents a new unsupervised pretraining technique called *part2vec* to learn low-dimensional embeddings of parts which are later employed as features for the parts. This technique is a generalization of word2vec [96] from the field of natural language processing to arbitrary graph structures. Moreover, the chapter highlights additional applications of the trained embeddings for designers and their companies and concludes with the presentation of a developed embedding refiner, which allows a user to define similarity constraints for particular parts in order to enhance the embedding of infrequent parts in the assembly dataset.

The subsequent three chapters present solutions for the considered use cases of assembly modeling using graph machine learning. They each present the proposed methodology for addressing the respective problem and present the experimental results on the three assembly datasets.

Chapter 5 presents an interactive recommendation model that suggests a fixed number of part types for a given assembly during design – referred to as global part recommendation. It describes the procedure for generating corresponding recommendation instances from the final assemblies in a self-supervised manner and introduces various comparison models for assessing the recommendation results. This includes a self-developed frequency-based baseline inspired by market basket analysis and an upper bound estimation of the maximal achievable performance. Potential techniques for improvement are then examined, such as pretraining of the recommendation models and employing Graph Transformers. Experiments and evaluation results from a user study confirm that the global part recommendation has been successfully implemented.

Chapter 6 addresses the use case of localized part recommendation in assembly modeling through two use case variants. The first variant involves recommending part types for a designer-given extension point in the design, while the second variant focuses on recommending both an extension point and a part type. The extension point refers to a part already included in the current assembly. The chapter further explains the necessary

adjustments for instance generation in order to integrate the information of the extension point. For both variants, different modeling approaches are examined and compared to baseline models on the assembly datasets.

Chapter 7 deals with anomaly detection in assemblies, specifically the identification of anomalous parts originating from the previous selection of an unsuitable part type. Since the assemblies examined in this thesis do not exhibit any anomalies, it presents an algorithm to generate synthetic anomalous assemblies by leveraging an extracted set of proven part combinations extracted from the assembly datasets. Furthermore, it leverages concepts of the first localization variant of Chapter 6 in order to recommend alternative parts for the anomalous one while not including their type information in the recommendation model. Experimental results on the assembly datasets show the successful modeling of both tasks.

Finally, Chapter 8 summarizes the content and findings of this thesis, provides lessons learned about the presented methodologies throughout this thesis, and presents an outlook on future work.

Computer-Aided Design Foundations

Summary This chapter provides a foundational overview of computer-aided design (CAD) with an emphasis on part and assembly modeling. While parts constitute atomic components, an assembly is composed of multiple parts with pairwise constraints defined relative to the parts' topology – so-called mating conditions. The chapter further highlights the shift in design practices due to the availability of standardized, mass-produced parts in part catalogs, which allow designers to select from pre-existing parts rather than having to create new ones from scratch for modeling new products. Lastly, it discusses the integration of these parts into CAD systems to streamline the assembly design process.

Contents

2.1	Part Modeling	11
2.2	Assembly Modeling	12
2.3	Enhancing Reusability in Assembly Modeling via Part Catalogs . . .	14

For a deeper insight into the application domain in which this thesis is located, this chapter provides an overview of different modeling variants of computer-aided design, in particular the design of individual parts and their assemblies. Finally, we discuss the setting examined in this thesis, in which designers create assemblies consisting of parts stemming from catalogs of manufacturers. Our explanations on part and assembly modeling follow fundamental literature [124, 105].

Computer-aided design (CAD) refers to the use of computer systems for assisting in the creation, modification, analysis and optimization of an engineering design [105]. This area is currently experiencing great popularity due to the increasing use of 3D printing in many branches such as automotive, aviation and healthcare, but also in the private sector [65]. Before the use of computers, engineering drafts were created on paper to document the designs. Depending on the information to be depicted, the drawings took various forms, from small sketches to detailed drawings on large sheets of paper using drafting machines. This is why CAD is also referred to as computer-aided drafting. In the course of the twentieth century, CAD became the standard method of creating engineering drawings [71]. This led to an increase in both the productivity of the designer and the quality of the designs.

Modern CAD systems are user-oriented systems based on interactive computer graphics, in which designers create and manipulate design data based on geometric operations. The designs should be more thought of as vector graphics rather than bitmaps indicating colored dots on a monitor: The central item within a CAD system that designers create and interact with is a *model*, more specifically a mathematical model of the designed object. The name is derived from the fact that the object's geometry is defined by a combination of geometric figures (such as points, lines, circles and so on) and their relationship to each other, which can both be described by mathematical equations, for example dimensions or parallelism. CAD systems are therefore equipped with an engine that generates the geometric object based on its model.

CAD models can be created in both two and three dimensions. Typically, CAD systems support a combination of different modeling types. The first step in 3D modeling is often based on 2D modeling, for example by first creating a sketch of the geometry on a plane in 2D, which is then expanded to include the third dimension.

As another categorization, a CAD model can represent a single, individual *part* or a whole collection of multiple connected parts, so-called *assemblies*. A *part model* is self-contained: it can be stored independently within the CAD system and the mathematical equations defining it are entirely sufficient to regenerate and display the part on its own. Part models are composed of basic geometric shapes using geometric operations. By contrast, an *assembly model* is made up of a set of part models. To be precise, the assembly model does not contain them, instead it *references* the individual part models and defines their position, either globally in a given coordinate system of the assembly or locally, i.e., in relation to each other. By adding parts to an assembly, their model remains unaffected; however, due to referencing, changes to the part models are reflected in the assembly. There are typically separate workspaces in CAD systems for part modeling and assembly modeling.

Part models can be further divided into *surface models* and *solid models*. The former have no volume and consist only of surfaces, which could be imagined as being made of a wafer-thin material. Solid models, on the other hand, do have volume, i.e., all faces of the model are connected to other faces to form, for instance, a cuboid. For the purposes of this dissertation, we will only consider solid part models.

Over time, a large variety of different CAD systems have been developed. Common CAD systems include for instance AutoCAD, CATIA, SolidWorks or FreeCAD [134]. Unfortunately, no standard CAD format has been established, so each CAD system vendor developed their own proprietary file formats which complicates data exchange between different systems. However, there exist few neutral file formats that every CAD system can read, for example DXF, IGES and STEP. Neutral formats typically contain less information, e.g., only edge, surface and volume models [124]. Furthermore, CAD models can be exported to various formats, such as polygon meshes, point clouds or solid objects consisting of geometric elements.

In the following, we will have a closer look at the design variants part and assembly modeling. In our explanations, we refer to the example of a bicycle as an assembly from [124].

2.1 Part Modeling

The focus of this thesis is on assembly modeling, so we will only briefly outline the creation of part models to convey an overall picture of CAD. Parts are atomic production elements that cannot be disassembled into several parts and are typically manufactured from a single material [26], such as a screw. All the information required to represent a part can be taken from the associated model file. In some CAD systems, they are also referred to as components; however, this term also refers to more complex elements consisting of several elements similar to assemblies, so we will keep to the term part for atomic elements.

In many CAD systems, part modeling is an iterative three-step process in which geometric shapes are repeatedly sketched in 2D and then extended by the third dimension [124]. This is also known as a feature-based process, whereby a *feature* is a self-contained segment of a part model, e.g., a protrusion or a cut-out. The first step involves selecting a two-dimensional plane in which a two-dimensional geometry is created in the subsequent step. Initially, this can be a plane of the initial coordinate system or a custom-added plane; if geometric elements already exist, their faces can also be chosen. A two-dimensional shape is then created in the selected plane, for example a circle if a cylinder is to be modeled. The 2D shape can also be transformed into more complex shapes using geometric operations such as union or intersection, e.g., to form the basic shape of a bicycle pedal as depicted in Figure 2.1. In the third step, the two-dimensional shape is expanded to 3D, giving the model its three-dimensional features. The designer selects which segment of the 2D geometry is to be used to create the feature and which method is to be used; typical options are *extruding*, *revolving* and *sweeping*. Extruding expands a 2D geometry along a single direction, transforming a circle into a cylinder; revolving creates surfaces that revolve about an axis, turning a circle into a sphere or ring; and sweeping expands a shape along an arbitrarily smooth path, which is used to model tubes, for instance.

The right-hand side of Figure 2.1 shows the 3D shape of a bicycle pedal after extrusion. Most 3D part models are complex and thus require new features to be added onto the base feature. These three steps are then repeated, for example by adding notches (by negative extrusion) and spikes to the pedal to model its profile.

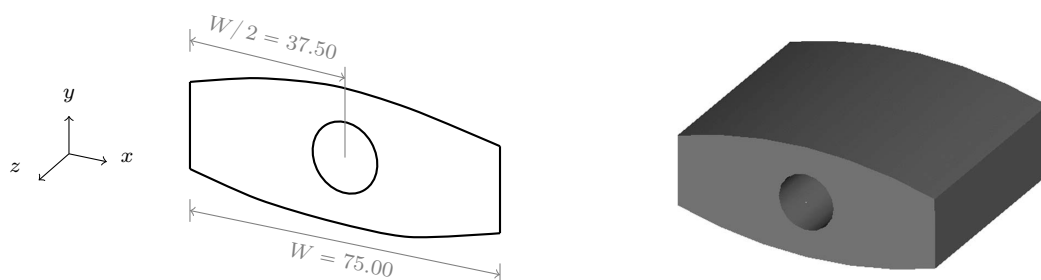


Figure 2.1: Part modeling illustrated by the basic shape of a bicycle pedal. The basic shape is sketched in 2D on the left, showing the result after the first two part modeling steps. A parametric constraint positions the hole horizontally in the center of the pedal; other constraints are omitted. The resulting 3D shape after creating the feature, i.e., extrusion along the normal axis z , is shown on the right. Adjustment of [124, Figures 8.1 and 8.2].

As mentioned, a (part) model consists of mathematical equations that determine its shape and position. So-called *constraints* can be used to specify dimensions or define the position of geometric elements in relation to each other, for example by parallelism, perpendicularity or coincidence. Some constraints are automatically inferred by the CAD system, for instance if a dimension between two lines is given, the system sets them as parallel. The dimensional constraints can be hard-coded with fixed values or specified *parametrically* in relation to certain dimensions. This can be used, for example, to ensure that the pedal's hole for connection to the crank is always positioned in the center of the pedal. To do this, the designer sets the dimension between the center of the hole and the sides to half the width or height of the pedal. If a designer then needs to change the size of the model, they only need to adjust a few dimensions which can save a lot of time in a later design project. Constraining can also be seen as removing degrees of freedom. Two-dimensional objects have three degrees of freedom as they can be moved in the x and y directions as well as rotated around the z-axis. In the case of an underconstrained setting, i.e., if not all degrees of freedom have been removed, CAD systems would have multiple possibilities for representing the respective element (since there are several solutions of the mathematical model) and would not necessarily display the part model in the same way when the model file is opened. For example, if the position of a hole is not fully constrained, it could be placed at any position along the remaining degrees of freedom.

The core idea of parametric constraints has furthermore been extended to so-called *part families* or *family tables* [124]. These part models are essentially very similar, only deviating slightly in a few aspects like a certain dimension or detail features. Instead of creating separate models for slightly different variants of a part, a designer specifies a set of permitted values for the corresponding parameters. For example, a designer can define a single model for a pedal and limit its width and length to specific values (e.g., 8, 10 and 12 cm, respectively, which results in nine different specific parts).

Modern CAD systems also offer functionality for analyzing and validating the created models, for example if some shapes are intersecting, i.e., are sharing the same volume of space, or to compute the distance between two faces, which is relevant to prevent manufacturing problems.

2.2 Assembly Modeling

In assembly modeling, separately saved models of individual parts are combined within a CAD system to form a holistic model, the assembly. Instead of actually containing the parts and their geometry, an assembly only references the associated part models and stores information about their relationship. The parts can be joined together in various ways, for example welded, fastened together or simply form a working unit in a more complex product like a motor [124]. It allows engineers to define how different parts fit together to form a complex design and view it from different perspectives.

This modeling distinguishes between two types of assemblies: A *bottom-up assembly* relies on previously designed parts, which are brought together in the assembly to create a complex product. This method is comparable to building blocks, whereby each block represents an individual part and the overall construction represents the entire assembly. The *top-down approach*, however, starts with designing the final assembly in a holistic sense, followed by breaking it down into individual parts. This procedure is suitable if the individual parts are manufactured in-house – which is also the case in 3D printing – as it

allows all individual parts to be designed relative to other parts, ensuring perfect assembly alignment. In contrast to this variant, the bottom-up approach is supported by all CAD systems [124]. This thesis also concerns itself with the modeling of bottom-up assemblies.

An assembly exhibits three essential characteristics, which we will discuss in detail below: instantiation of parts, positioning of parts and assembly structure [124].

Instantiation of Parts In many assemblies, the same parts occur several times, for instance the struts on a bicycle wheel. Therefore, the parts of an assembly are actually *instances* of the part models. This concept is the same as in software development, where multiple objects can be created as instances of a class. This means that a particular part only needs to be modeled once in order to be added to an assembly multiple times. Changes to this part model are then reflected in all of its instances. This allows the assembly to store instance-based information, for example about its respective position, which leads us to the second characteristic of assemblies.

Positioning of Parts within an Assembly A central aspect of combining a collection of parts into a complex product is to specify the *position of the individual part instances* with respect to the 3D modeling space. This is a prerequisite for deciding whether parts collide or interfere with each other, which requires readjustments to be made on the assembly. Similar to part modeling, *constraints* can also be applied to assemblies in order to determine the location and orientation of the part instances within an assembly model. In the context of assembly modeling, they are also termed *mating conditions* or simply *mates*. There are six degrees of freedom in the 3D space. The part's position can in principle be specified by coordinates in a global coordinate system, but is typically carried out using local relations between the parts. For example, dimensions can be defined between geometric entities such as planes, edges and points; but parallelism, perpendicularity or contact can also be specified. These can again be defined as parametric constraints with reference to specific dimensions. The available constraint types typically differ among the CAD systems [68]. The final position of all parts based on these mates is calculated by the CAD system's geometry constraint engine. As opposed to part modeling, in the context of assemblies it is common to remove only some degrees of freedom – previously referred to as underconstraining – in order to explicitly model certain directions of movement of the parts, for example the rotation of a gear or allowed translation of a bolt in a certain direction. Assembly constraints are meant to represent the designers' intention, but are sometimes misused to simplify the modeling workflow.

Assembly Structure Complex assembly designs such as manufacturing machines contain a huge amount of parts. In order to get a better overview and maintenance of individual components, the parts are often arranged *hierarchically* in so-called *subassemblies* instead of a flat list. This hierarchical structure is also known as *assembly tree*. The subassemblies are basically virtual containers for grouping specific part instances or other subassemblies. When employing an assembly hierarchy, it is best to define the mating conditions based on the same structure as well: Every assembly tracks positional information of its direct parts and of the origin coordinate system of its direct subassemblies. The top-level assembly relies on all the positional information and composes the coordinate systems of the subassemblies to display all the parts regardless of their assembly level in the defined position.

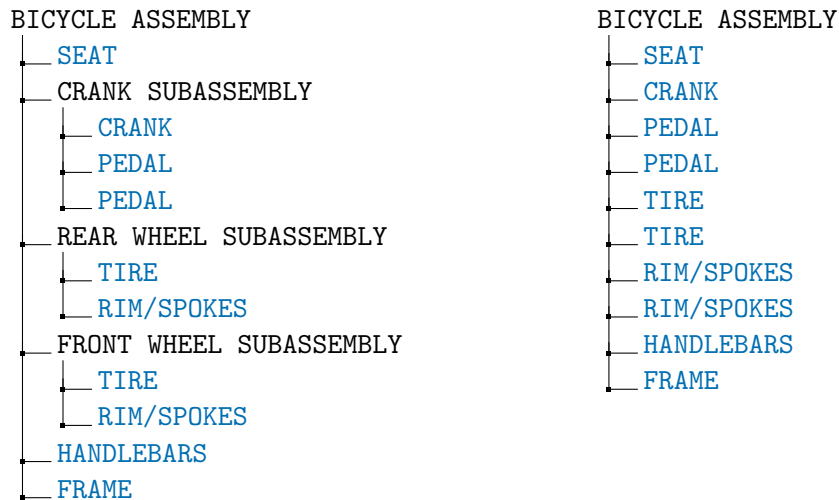


Figure 2.2: Two variants of the assembly structure of a bicycle: hierarchically with sub-assemblies (left) or as a flat list of parts (right). Parts are colored blue and assemblies are colored black. Adapted from [124].

The same assembly can basically be represented by several hierarchies. Of course, not every arbitrary segmentation into subassemblies is meaningful, as this is typically related to the spatial proximity of parts. However, the exact structuring into subassemblies depends on the preferences of the respective designer. It is therefore likely that two different designers will create two different assembly hierarchies for the same assembly. Figure 2.2 shows two possible assembly hierarchies for structuring a bicycle assembly.

2.3 Enhancing Reusability in Assembly Modeling via Part Catalogs

In the early days of product manufacturing, the individual parts of a machine were custom-designed in CAD systems and manufactured specifically for each product. However, creating custom parts was costly due to the extensive design and production time involved, and it eventually became more practical to draw on existing parts rather than designing new ones from scratch [4]. This led to the rise of part manufacturers offering mass-produced parts at lower costs, which are often more economical and quality-assured compared to custom-made parts. These manufacturers produce parts in standardized sizes and specifications that can be used across multiple applications, such as screws, bearings and motors [50]. As a result, companies purchase these parts instead of manufacturing them in-house, leading to a shift from part modeling to assembly modeling.

Nowadays, there are numerous part manufacturers which are offering a wide range of parts [113, 50]. The manufacturers provide their customers with product catalogs called *part catalogs*, which include detailed descriptions and technical specifications of the parts. As an example, we have a look at the *Product Overview 2023* part catalog of the Festo SE & Co. KG [72]. The company offers a wide range of products, such as pneumatic components, motors, vacuum technology, grippers, and valves, among others. As an example, Figure 2.3

Pneumatic shut-off valves >

Shut-off valves and ball valves





	 Hand slide valves VBOH	 Shut-off valves HE	 Ball valves QH-QS, QHS-QS	 Ball valves QH
Valve function	3/2 double solenoid	2/2 double solenoid, 3/2 double solenoid	2/2 double solenoid	2/2 double solenoid
Pneumatic connection 1	G1/2, G1/4, G1/8, G3/4, G3/8, M5	QS-10, QS-12, QS-6, QS-8, R1/2, R1/4, R1/8, R3/8	QS-4, QS-6, R1/8	G1, G1 1/2, G1/2, G1/4, G3/4, G3/8
Standard nominal flow rate	236 ... 7691 l/min	256.5 ... 834.3 l/min	148 ... 560 l/min	3400 ... 84000 l/min
Operating pressure [MPa]	-0.095 ... 1.2 MPa	-0.095 ... 1 MPa	-0.1 ... 1 MPa	
Operating pressure	-0.95 ... 12 bar	-0.95 ... 10 bar	-1 ... 10 bar	
Description	<ul style="list-style-type: none"> Used as a shut-off function for pressurising and exhausting compressed air systems, for example upstream of service units, for air guns and also for exhausting pneumatic cylinders Non-overlapping, so no pressure losses when switching Minimal installation effort 	<ul style="list-style-type: none"> Shut-off valve, manually operated Connection: thread at both ends, push-in connector at both ends, thread/push-in connector Different mounting options 	<ul style="list-style-type: none"> Shut-off valve, manually operated In-line installation, can be screwed in, bulkhead fitting Variants: thread at both ends, push-in connector at both ends, thread/push-in connector 	<ul style="list-style-type: none"> Shut-off valve, manually operated In-line installation Female thread at both ends With hand lever Pipe thread to ISO 2281
online: →	vboh	he	qh	qh

Figure 2.3: Excerpt from the part catalog of Festo SE & Co. KG showing several shut-off valves and ball valves. [72, p. 120]

shows an excerpt of the chapter about valves: For several shut-off valves and ball valves, detailed technical specifications such as standard nominal flow rate or operating pressure are given, together with a description about usage, installation, and notable aspects such as configuration variants or special characteristics.

In addition to the print version of their part catalogs, manufacturers started to provide *digital* versions tailored for the use in CAD systems. As these digital catalogs contain the 3D CAD models of the parts, CAD software vendors can integrate the digital catalogs into their systems in order to make them available to designers. For example, the company CADENAS GmbH offers an online library of part catalogs called *3Dfindit*¹: At the time of writing, the library contained over 6,800 catalogs from various manufacturers, and the company provided plugins for 35 CAD systems to access the catalogs directly in the tools.

Typically, customers tend to peruse catalogs from multiple manufacturers and compare the offered parts before selecting the one which fits best for their assembly. However, keeping track of and selecting the right parts from these extensive catalogs can be challenging due to the large volume of options [87, 35]: The digital version of the Festo SE & Co. KG part catalog contained over 50,800 parts in total at the time of writing this thesis. Fortunately, some manufacturers (including Festo SE & Co. KG) applied a similar hierarchical structure as in the print version to the digital catalogs, such as an increasingly refined categorization of part types. This enables designers to search for a specific part at least the same way as if they searched in the print version: For instance, to find the ball valve with type code *QH-QS* from the example above, a designer would choose the *Festo* catalog in *3Dfindit*, select *Valves*, refine the category by *Shut-off valves*, and finally find the part under *Ball valves and on-off valves* (cf. Figure 2.4).

¹<https://www.3dfindit.com/en/cad-bim-library>

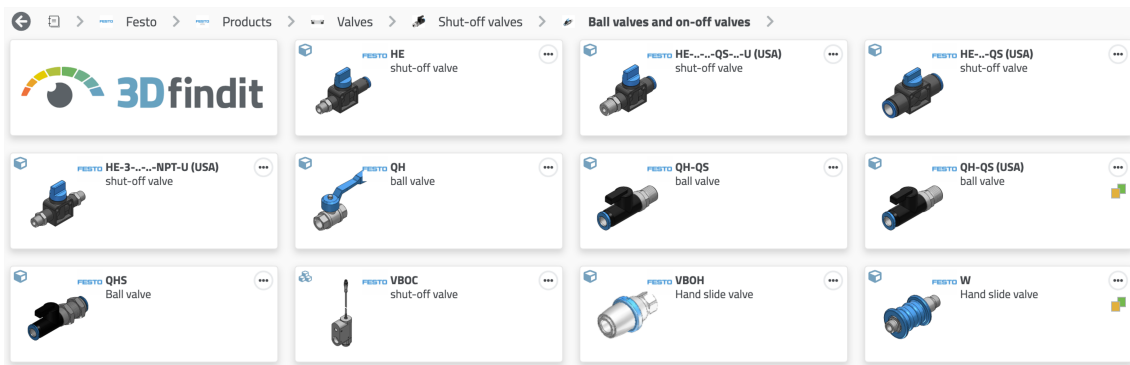


Figure 2.4: Search path and results in 3Dfindit [49] for ball valves and on-off valves in the digital catalog of Festo SE & Co. KG.

Unfortunately, the digital part catalogs are not standardized, and each manufacturer can organize its catalog in a custom way (or not at all and just provide a flat list of parts in arbitrary order). This makes searching across catalogs and comparing parts from different manufacturers a manual and time-consuming task, as existing software solutions struggle with defining a generalized categorization due to the proprietary formats. For example, 3Dfindit provides a single category filter value *Fluid power* for the *Ball valves and on-off valves* category of the *Festo* catalog – however, applying this filter on top-level resulted in over 39,800 parts. At least the classification filter allows for more fine-grained filtering, but the result set still consisted of over 3,600 parts at the time of writing. Still, the filter’s values depend on semantic annotations which differ for similar parts from different manufacturers, forcing the designer to validate the results by manually pursuing the catalog when comparing parts.

The comparison of parts from different manufacturers itself, however, is another difficult and error-prone task for designers due to the lack of standardization among digital part catalogs. While the print versions usually provide information about technical specifications or special characteristics of a part, this is not always the case for their digital pendants. Figure 2.5 shows the detailed properties of two ball valves in 3Dfindit: The first table contains technical specifications for the variant *QH-QS-6* of the Festo valve from the previous example, the second for the ball valve *KH1/2NPT71X* of the manufacturer *Parker* (confer their print catalog [27] for more details). Obviously, they both contain geometric properties of the part, but use different labels for the respective fields. Further, the Festo valve lacks essential information required to assess the suitability of the part for use in an assembly: Its operating pressure range is not provided, whereas the table for the Parker valve not only lists its nominal pressure, but also the material of the valve.

To support designers in comparing and finding similar parts from different catalogs, 3Dfindit offers a feature called *similarity search* [47, 48]. A designer can select a reference model from a part catalog, upload a 3D model, or create a 2D sketch, and the tool searches for parts with similar geometric and structural features. However, for a detailed comparison, design engineers still has to manually compare the specific properties of the parts themselves to decide between them. Since in the worst case the property names are completely different, this poses a risk that parts unsuitable for the intended use will be selected, even if they look similar to the desired part.

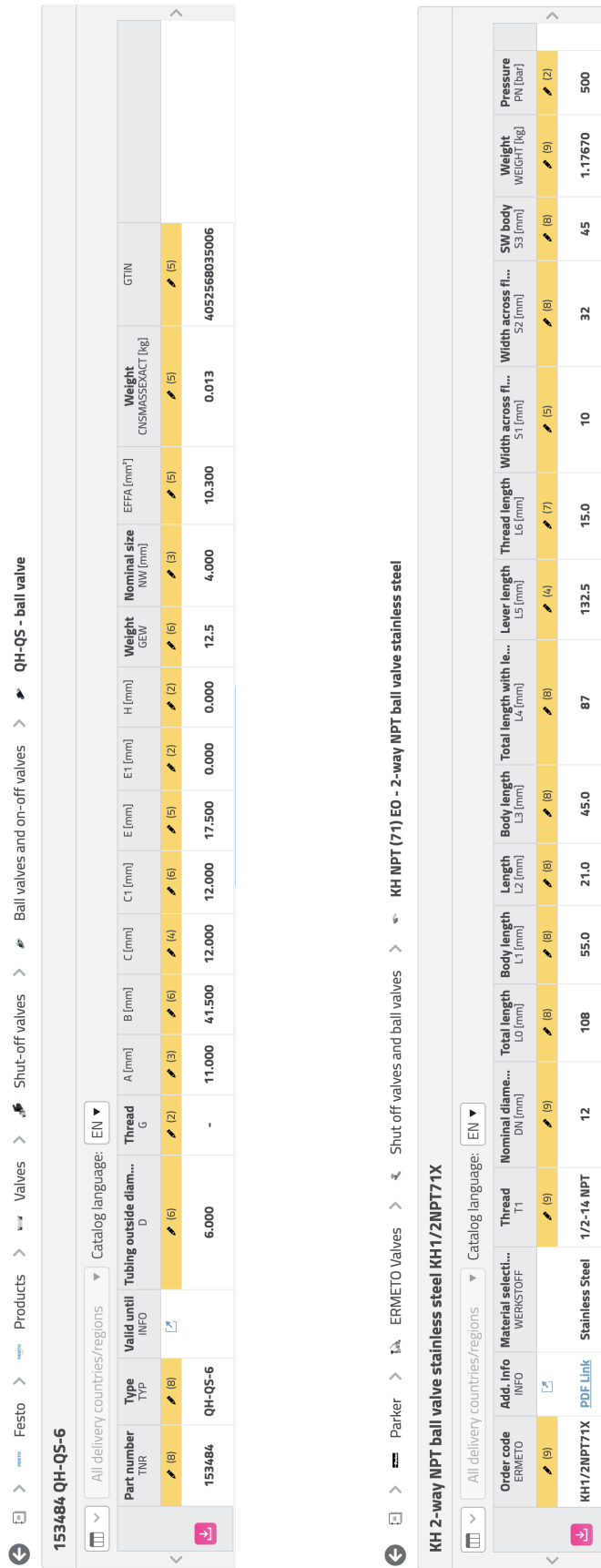


Figure 2.5: Comparison of available information on two ball valves from digital catalogs of different part manufacturers in 3Dfindit [49]: The upper table shows information on the ball valve of type *QH-QS-6* from the *Festo* catalog, whereas the lower table shows information on the valve *KH1/2NPT71X* of the manufacturer *Parker*.

Data and Machine Learning Foundations

Summary This chapter introduces the reader to the fundamental concepts and established methodology in machine learning required for the following sections. After a general introduction, the chapter focuses on the data structure of graphs and machine learning methods for graphs. This includes the necessary definitions and notations in order to represent the assemblies as graphs. We present the principles of *graph neural networks* (GNN), a form of neural network that was specifically designed for processing graphs, and outline four specific architectures, all of which have promising properties for processing assemblies for our use cases: Graph Convolutional Networks (GCN) [79], Graph Attention Networks (GAT) [144], Graph Isomorphism Networks (GIN) [157] and GraphSAGE [58]. Finally, we discuss techniques for embedding discrete objects, such as parts in an assembly. In particular, the technique word2vec [96] is presented, which originates from the field of natural language processing (NLP), but is not specific to natural language and can thus be transferred to our domain of CAD assemblies.

Contents

3.1	Basic Concepts of Machine Learning	20
3.1.1	Artificial Neural Networks	24
3.1.2	Deep Learning	27
3.2	The Graph Data Structure	28
3.3	Graph Machine Learning	32
3.3.1	Categorization of Learning Tasks on Graphs	33
3.3.2	Graph Neural Networks	35
3.4	Embedding Discrete Objects	44
3.4.1	Learning Embeddings	45
3.4.2	Word2Vec	45

Machine learning (ML) is a sub-area of artificial intelligence (AI) which aims at automatically extracting knowledge and patterns from *data*. ML thus refers to a subset of AI *techniques*. In contrast to traditional approaches in software engineering where a system's behavior is explicitly programmed by hard-coded rules, ML aims to extract (i.e., learn) these rules from data. The foundation of ML is built upon multiple disciplines such as mathematics, statistics, information theory and computer science, from which various learning algorithms emerged. Due to increasing data volumes and ever more efficient com-

puting systems, ML achieved ever faster and better results and thus enjoyed increasing popularity. ML is used in countless application areas, such as computer vision, natural language processing (NLP), recommendation systems, medical diagnosis aid and industrial applications, as it is also the case for this thesis. AI methods outside of ML are nowadays referred to as classical AI.

At the beginning, we want to provide an overview of the mathematical notations used in this thesis which is presented in Table 3.1.

Table 3.1: Overview of the most common mathematical notations used in this thesis.

Notation	Explanation
\mathbf{M}	An uppercase, bold and italic symbol denotes a matrix.
\mathbf{x}	A lowercase, bold and italic symbol denotes a vector.
c	A lowercase and italic symbol denotes a scalar.
x_i	Denotes the (i) -th scalar entry of a vector \mathbf{x}
m_{ij}	Denotes the (i, j) -th scalar entry of a matrix \mathbf{M}
$\mathbf{M}^\top, \mathbf{x}^\top$	Transpose of matrix or vector
\mathbb{R}^n	The n -dimensional real space, also called Euclidean space
\mathbb{B}^n	The n -dimensional binary space
\mathbb{N}_0	The natural numbers including zero, i.e., $\{0, 1, 2, \dots\}$
\mathcal{S}	Calligraphic symbols denote more complex structures like sets and tuples.
$ \mathcal{S} $	Denotes the cardinality of a set \mathcal{S} , i.e., the number of its elements
$\{a, b, c\}$	Denotes the <i>set</i> containing the elements a , b and c
$\{a, b, b\}$	Denotes the <i>multiset</i> containing once the element a and twice the element b
$\langle a, b, c \rangle$	Denotes the <i>tuple</i> consisting of the elements a , b and c . Notation of instances.

3.1 Basic Concepts of Machine Learning

The techniques used in this thesis to address the use cases presented are mainly from the field of machine learning, which is why we will briefly outline the most important basic concepts and established methodology from foundational literature [51, 118, 15].

Datasets and Instances Since ML aims to extract patterns and rules from data, the applications are based on so-called *datasets*. A dataset usually consists of multiple different *instances* which serve as examples or sampling points of the function to be learned. An instance consists of an input, which is fed into the ML algorithm, and optionally a desired output, also referred to as *target* or *label*. The typically multidimensional input consists of multiple discrete or continuous *features* representing properties of the respective instance. The target values can also have several dimensions and can take on both discrete and continuous values. Since machine learning methods are based on mathematical methods, discrete features (and targets) must be translated into numerical values. A common approach is *one-hot encoding* where a new binary feature is created for each discrete value. If a certain discrete value is present, the corresponding new binary feature is set to 1, while all others are set to 0.

Categorization of ML Tasks Machine learning can be broadly divided into *supervised* and *unsupervised learning*. In the first case, the data consists of input and corresponding target. The goal is to learn a function mapping the input to the target which can be afterward evaluated on unseen inputs. Supervised learning can be further divided into *classification* and *regression*. In classification, the target function takes on discrete values representing possible categories of the instances, whereas in regression, inputs are mapped to numerical, continuous values. In contrast to supervised learning, the data in unsupervised learning consists of input values only. The goal of this learning method is to analyze patterns within the data. Instead of evaluating them through an output value, the aim is to discover similarities in the data that provide insights into their structure or finding a more condensed representation. For both forms, the achievable quality of the learning technique highly depends on the quality and quantity of the data. Since the collection of targets proves to be very expensive in some domains, hybrid forms such as *semi-supervised* learning have emerged, in which only a (small) part of the data is labeled. *Self-supervision* refers to training models using the data itself to generate target values instead of relying on external labels provided by humans.

A central challenge in machine learning is *generalization* which refers to the ability of a model to produce adequate predictions for unseen data. For good generalization, the model must extract the general concept from the data without learning too many specifics that do not apply to the unseen data. Learning methods can also be differentiated according to the type of generalization: *Instance-based* methods memorize the training instances and refer to identical or similar seen training data during inference. This requires a similarity measure on the input. In contrast, in *model-based* learning, a mapping is built from the given data which is afterward used to infer predictions for unseen data instances. These models are characterized by model *parameters*, which are adjusted during training to fit the given training data. Often, models or the training process as a whole also have *hyperparameters* that determine its functioning. Hyperparameters are used for many purposes, for instance to specify how the data is normalized, or to restrict the complexity of the model function. They need to be set prior to training as they cannot be learned. To find a suitable hyperparameter setting for the current problem, we need to train the model on various combinations of hyperparameters in order to find a suitable set. This procedure is called hyperparameter *optimization* or *tuning*.

Data Split For learning from data, the overall dataset is divided into three disjoint sets: the training data is used to adjust the model parameters, while validation data is used to determine the hyperparameters, compare different models and model architectures. Finally, the test data is used to estimate the *generalization error* on unseen data after selecting a final model. The prerequisite is that the assumption of *independently, identically distributed* (i.i.d.) [51] data applies, i.e., the instances of each data set are independently distributed, the distribution of all sets does not fluctuate, and all instances are drawn from the same probability distribution. Violating this assumption can lead to considerable performance degradation and poor generalization.

Training and Evaluation To evaluate the quality of a learning model, the difference between the model outputs and the target values is determined using a so-called *loss function*. In the case of unsupervised learning, the instances do not comprise target values, so the loss functions evaluate properties of the data instead, for example, measure the

reconstruction error from a compressed representation or evaluate other task-specific features [51]. The loss function is also used to optimize the model parameters during training: *Gradient Descent* [22] measures the local gradient of the loss function with regard to the current model parameters and adjusts them in the direction of descending gradient in order to minimize the loss. This requires the loss function to be differentiable. Formally, a parameter θ_i is updated according to

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial L}{\partial \theta_i} \quad (3.1)$$

with learning rate $\alpha > 0$ and loss function L . The learning rate controls the intensity of the parameter adjustment and is a hyperparameter. Instead of adjusting the parameters after each individual instance, with *Mini-Batch Gradient Descent* [14] the loss gradients of all instances of a small batch are summed up before the model parameters are adjusted. Typically, the training data set is iterated multiple times; each round is called an *epoch*. When a model fits too closely or even exact to its training data and thus learns too specific concepts that do not generally apply, *overfitting* occurs [118]. This leads to poor generalization and should therefore be avoided. During training, overfitting can be detected by a strong discrepancy between training and validation loss. Overfitting can result from too long training, for example.

In contrast to loss functions, *performance measures* or *performance metrics* are used to evaluate the quality of a model after training has been completed. These functions measure the quality of the actual learning task, for example the accuracy of image classification. In classification problems, the loss function and performance measure typically differ for the same task, as performance measures are usually unsuitable for parameter adjustment due to their non-differentiability. Furthermore, the value range of loss functions often depends on the specific modeling of the problem (e.g., output dimension), so different modelings should always be compared using performance measures. The loss function is chosen so that its minimization leads to models that achieve high values of the performance metric.

Loss functions and performance metrics relevant to this thesis are presented below.

- **(Categorical) Cross-Entropy** Cross-entropy measures the quality of a binary or multi-class classification model. Specifically, this loss function measures the difference between the one-hot encoded true distribution p of the labels and the output distribution q for all instances of a given instance set \mathcal{D} by

$$H_b(p, q) = - \sum_{\mathbf{x} \in \mathcal{D}} p(\mathbf{x}) \log_b q(\mathbf{x}) \quad (3.2)$$

where $\log_b z$ denotes the logarithm of z to base b .

- **Top-k Rate** This performance metric computes the ratio that the label lies within the top k predictions of a model. Its value monotonically increases with the number of predictions k .
- **Shannon Entropy** The (Shannon) entropy stems from information theory and measures the disorder of a distribution. It corresponds to the cross-entropy measuring the difference between the same distribution, formally defined as

$$H_b(p) = H_b(p, p) = - \sum_{\mathbf{x} \in \mathcal{D}} p(\mathbf{x}) \log_b p(\mathbf{x}) \quad (3.3)$$

- **Perplexity** Perplexity measures the uncertainty of a model in a certain multi-class prediction. It is based on the Shannon entropy. The larger the value, the less certain the model is about the prediction. It can be calculated by

$$PP(p) = 2^{-\sum_{\mathbf{x} \in \mathcal{D}} p(\mathbf{x}) \log_2 p(\mathbf{x})} = 2^{H_2(p)} \quad (3.4)$$

for an output distribution p over all classes.

- **Precision and Recall** These performance metrics evaluate the predictive quality of a model in a binary classification problem. Precision is the fraction of predicted instances that are indeed true, i.e., the fraction of correct predicted among all predicted instances:

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \quad (3.5)$$

Recall on the other side is the fraction of positive targets that were indeed predicted by the model:

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \quad (3.6)$$

- **F-score** In many application areas, both good precision and recall should be achieved, which is why both metrics are often combined into one value as the F-score given by:

$$\text{F-score} = \frac{2pr}{p + r} \quad (3.7)$$

Here, p denotes precision and r denotes recall.

Hyperparameter Tuning Finding a suitable set of hyperparameters for the tackled task can be carried out fully manually or by an automatic selection strategy. *Grid Search* and *Random Search* are the most common automatic strategies [118]. At the beginning, one must specify a set of promising values per hyperparameter, typically selected in a logarithmic way. Grid Search involves systematically exploring every joint of the Cartesian product of the pre-selected hyperparameter values. Random Search requires probability distributions for all hyperparameters, according to which a set of values is drawn in every step. For each hyperparameter set, the model is trained on the training set and evaluated on the validation set. Finally, the hyperparameter set that leads to the best generalization error on the validation data is selected. The same procedure can be repeated iteratively on a finer grid for promising regions of the hyperparameters found. More advanced hyperparameter optimization techniques like *Bayesian Optimization* internally build a probabilistic model that maps hyperparameter sets to the loss evaluated on the validation set in order to target the selection of next hyperparameter values. Regions in the hyperparameter space that led to good generalization results are explored more deeply than regions with poorer results.

Early stopping has been established for determining the training duration: Both the training and validation loss are monitored over the training epochs. The training stops if the loss did not decrease on the validation set within a certain epoch span, also referred to as *patience*. Early stopping can thus prevent a model from overfitting to the training data.

3.1.1 Artificial Neural Networks

Today, ML is dominated by (*artificial*) *neural networks*, which were designed as a simplified model of the human brain. They aim to learn complicated concepts by constructing them from simpler ones in a hierarchical manner. Their prediction function is basically a mathematical function over the input. The basic computational unit of a neural network is the *neuron*. Neurons are connected by edges that are assigned real-valued *weights*. A neuron takes in inputs from previous neurons weighted by the corresponding edge weight, sums all these products and applies a transformation function called *activation function* before returning a real-valued output. The weighted sum before applying the activation function is called *activation*. The structure of an artificial neuron is shown in Figure 3.1. The weights are adaptable (i.e., parameters) and control the influence of every input of a neuron. To approximate a specific function, the weights must be set properly.

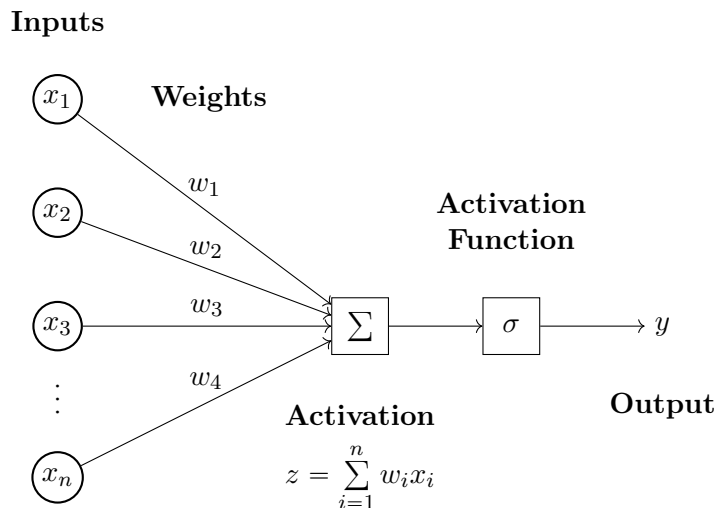


Figure 3.1: Mathematical model of a neuron [118].

In order to represent complex functions, neurons are arranged in so-called *layers* that are stacked as modules in a sequential manner in order to form the network. For example, we might have a neural network consisting of three sequential layers l_1 , l_2 and l_3 . The function of the overall network is given as the composition $f(\mathbf{x}) = l_3(l_2(l_1(\mathbf{x})))$. There are different types of layers, which differ in the number and structure of connections with the neurons of the previous layer. Neural networks are often termed according to the most common or specific type of layer they are composed of. The simplest and most common layer is *feed-forward* or *fully-connected*, which performs a linear transformation including a bias vector \mathbf{b} on the vector of all inputs \mathbf{x} computing $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$. The entries w_{ij} of the weight matrix \mathbf{W} correspond to the edges weights of neuron i of the previous layer to neuron j of the current layer. A neural network consisting only of feed-forward layers is also referred to as a multilayer perceptron (MLP) [117]. Typically, all neurons in a layer have similar connection scheme to the previous layer, however, differing in terms of the learned parameters.

For complexity reasons, neural network layers often represent linear functions. Since the concatenation of linear functions (due to multiple sequential layers) is still linear, a non-linear component – also referred to as *nonlinearity* – is required in between layers

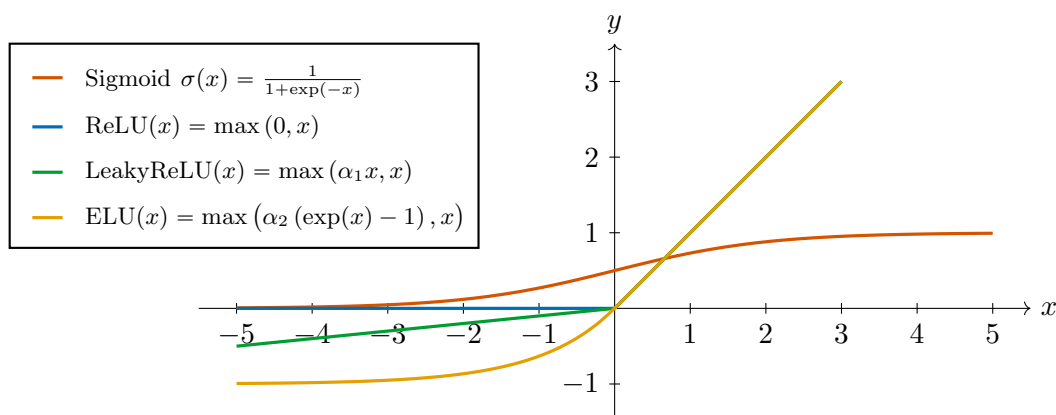


Figure 3.2: Overview of most common activation functions [118]. ReLU, LeakyReLU and ELU represent the identity function for positive inputs, thus their graphs fall together. LeakyReLU is illustrated with default value $\alpha_1 = -0.1$ for the gradient of the negative slope, ELU with $\alpha_2 = 1$.

to build more complex functions, which is solved by the activation function: A neuron’s activation is passed into a non-linear activation function in order to produce its output. Figure 3.2 displays the most common activation functions developed over time that we will encounter in the experiments in this thesis. Only the Softmax activation function is not shown because, unlike the other activation functions, it does not operate on a single neuron but on all neurons in a layer together. Therefore, it is also referred to as Softmax layer. Specifically, it converts a real-valued vector $z \in \mathbb{R}^m$ into a probability distribution of same size, formally: $\text{Softmax} : \mathbb{R}^m \rightarrow (0, 1)^m$. It is commonly applied in classification tasks in order to transform the neuron activations of the final layer into a probability distribution of the problem classes. Its formula is given by

$$\text{Softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^m \exp(z_j)} \quad (3.8)$$

where $\exp()$ denotes the exponential function.

In course of time, different types of neural network layers have been developed that are tailored to specific tasks and data types. In the context of this thesis, we are mainly working with *graph neural networks*, which are discussed in detail in Section 3.3.2. These generalize the convolution operation, which is used in the context of image processing by *convolutional neural networks*, to graphs. Furthermore, we employ or refer to other architectural concepts and layer types in some passages, thus we briefly outline them below.

Convolutional Neural Network (CNN) Yann LeCun is considered the founder of CNNs [83], which were designed to process and analyze *grid-like data structures*, such as images. The core component of a CNN layer is the *convolution*, a mathematical operation which can be visualized as sliding a *filter (matrix)* over the input grid while multiplying the filter and grid values. The filters aim to detect local patterns, such as edges or textures, and represent the learnable parameters of the corresponding convolutional layer. Note that the

same filter, i.e., the same collection of parameters, is employed for the whole image within a layer, which is referred to as *parameter sharing*. Through multiple layers, the initially extracted simple concepts can be progressively combined into more complex features of the data so that the neural network is finally able to distinguish, e.g., cats from dogs.

Recurrent Neural Network (RNN) RNNs [120] are specialized for processing *sequential data* such as natural language text, audio and video streams or time series data. Besides sequential connections, they also use loops – so-called recurrent connections – within their architecture to maintain a memory of previous inputs, allowing them to capture temporal patterns and dependencies. In particular, they process each element of the input using the same weight matrix and thereby perform parameter sharing. However, because of this repeated multiplication, standard RNNs struggle with learning long-range dependencies due to issues like vanishing or exploding gradients during training [51]. Gated RNNs address this problem by incorporating a more complex memory cell structure with *gates* that control the information flow, enabling the network to learn both short-term and long-term dependencies effectively. The *long short-term memory* (LSTM) [122] model introduced by Hochreiter and Schmidhuber is such a gated RNN architecture that uses four different gates that control which information is added to, retained or discarded from the cell state and eventually output by the cell.

Autoencoder The autoencoder architecture [118] was designed to learn efficient representations of input data by encoding them into a lower-dimensional latent space and then reconstructing the original input from this compressed representation. The size of the latent space is to be chosen as a hyperparameter. The network consists of two main parts: an *encoder* $enc()$ that transforms an input into a compact representation and a *decoder* $dec()$ that reconstructs the input from this encoded form. Formally speaking, an autoencoder is trained to satisfy the equation $dec(enc(x)) = x$ for any input x . By training the autoencoder to minimize the reconstruction error, i.e., the difference of target x and the model's output $dec(enc(x))$, the model learns to capture the most important features of the data in the compressed representation. The specific layer types of an autoencoder are chosen depending on the data type to learn the representation for, e.g., in the context of images, convolutional layers are typically utilized.

Attention Mechanisms The *attention mechanism* is a technique to improve a model's prediction performance by focusing on relevant aspects of the input. It allows the model to assign different *attention scores* or *attention coefficients* to different parts of the input and thus to prioritize relevant information over less important details for the prediction. The scores are determined by a *learnable function* over the current representation of the input and afterward normalized by Softmax to a distribution of *attention weights*. Finally, the input is multiplied by the respective attention weights before computing the model's output. The attention mechanism has been presented in 2014 by Bahdanau et al. [6] which led to a significant improvement in neural machine translation, i.e., translating natural language text utilizing neural networks. It is nowadays employed in numerous applications, even outside pure NLP, for example in the generation of image captions [156]. Furthermore, it laid the foundational groundwork for the development of the *Transformer* architecture [141] by Vaswani et al. in 2017 which once more revolutionized neural machine translation and even became the state-of-the-art technique in most applications of

computer vision and natural language processing [118]. The architecture generalizes the concept of attention to *self-attention*: While in classical attention, the input is attended when processing the output, in self-attention each element of the input attends *each other element* in the input and thus captures their relationship. The use of self-attention replaced recurrent layers to encode a sequential input, so that the Transformer architecture is based purely on feed-forward layers (in addition to techniques for training stability such as regularization).

3.1.2 Deep Learning

Deep learning is formally described as learning with neural networks composed of at least three successive processing layers. Deep networks can approximate more complex functions which allows processing raw input data instead of handcrafted features. This eliminates manually driven feature extraction and transfers this responsibility to the models. The model should extract important features for the respective task itself. There are numerous examples where this approach has yielded better results, a popular one is the drastic improvement in the image classification of the ImageNet dataset¹ [60]. However, larger models have more parameters, which means they require more data to adequately adjust them. This phenomenon is called the *curse of dimensionality*: The number of instances needed to approximate a function increases exponentially with its dimensionality. Growing computational power and the availability of large datasets have allowed to train deep models successfully. This has led to performance breakthroughs in numerous domains, such as computer vision or NLP.

Some applications require complex and therefore deep networks, but only a limited amount of data is available. *Transfer learning* is a popular strategy to overcome the curse of dimensionality in these cases. It is motivated by the observation that many features learned by deep neural networks, especially in the early layers, are often general and can be useful across different tasks or domains. Therefore, a model that has been trained on another task is used as a basis for a different but related task. In particular, the task-specific posterior layers of this pretrained model are replaced by new, untrained ones and the whole model is trained – also called *fine-tuned* – on the small amount of data from the new task. Thus, transfer learning significantly reduces computational resources and training time, which makes it a popular approach even when the data is not limited.

The parameter adjustment of neural networks during training is performed by Gradient Descent. For the last layer, it is straightforward how to determine the gradient of the loss function with respect to the weights. However, no intermediate target representation is known for the previous layers, thus no intermediate loss and loss gradient can be computed. This was a long-standing problem with neural networks, which was finally solved by *Backpropagation* [148]. This method allows determining loss gradients for parameters in all layers of a neural network by propagating the loss backwards layer by layer through the network using the chain rule of calculus. Successive layers of a neural network basically form a concatenation of functions, e.g., $f(\mathbf{x}) = f_2(f_1(\mathbf{x}))$ for a network composed of two layers representing functions f_1 and f_2 , respectively. The influence of these functions and their associated parameters, i.e., the weights of the layers, can be determined by deriving the concatenated function f using the chain rule.

¹The ImageNet project has created a large image database which is used as a benchmark for various computer vision tasks. The dataset can be accessed here: <https://www.image-net.org/>

In standard Gradient Descent, local minima and in particular saddle point plateaus of the loss landscape cause difficulties, which is why more complex *optimizers* were developed. At the time of writing this thesis, ADAM (adaptive momentum) [77] is the most widely used algorithm, which combines adaptive adjustment of the loss landscape with summarizing previous gradients (momentum). It has been shown to work effectively on large datasets.

As already mentioned, deep networks can approximate increasingly complex functions. This raises the risk of overfitting, i.e., the model adapting too much to the training data and thus representing a too complex function. Therefore, methods have been developed to restrict the complexity of a neural network and give the model a preference for a specific set of model functions, so-called *regularization techniques*. Highly variable functions typically have high weight values, which is why the first presented technique, *weight decay*, limits the range of the model weights. For this purpose, an additional summand is introduced into the loss function, which penalizes high weight values, e.g., the Euclidean norm of the weight matrices. *Dropout* [133] deactivates certain neurons based on a given probability (hyperparameter), so that several neurons of the same layer are forced to learn similar concepts. In this way, the neurons should learn several concepts at the same time, i.e., learn a more general concept overall, which should counteract overfitting. Dropout is commonly employed in computer vision for regularizing CNNs.

3.2 The Graph Data Structure

The following formal descriptions of graphs depend mainly on [59, 16], unless otherwise stated.

In its simplest form, a graph² $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined as a pair consisting of a finite set of nodes or vertices \mathcal{V} and a finite set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ representing pairwise relations between nodes. Both the nodes and the edges are unordered. If we want to clearly indicate which graph \mathcal{G} the sets belong to, this is specified in the subscript of the sets ($\mathcal{V}_{\mathcal{G}}$ and $\mathcal{E}_{\mathcal{G}}$). An edge $e = (u, v) \in \mathcal{E}$ denotes that there is a connection from node u to node v , also referred to u being *adjacent* to v . The *order* of a graph is the number of its nodes $|\mathcal{V}|$. A graph is called *undirected* if all edges are bidirectional, i.e., $(u, v) \in \mathcal{E}$ implies $(v, u) \in \mathcal{E}$. If this property does not hold, the graph is called directed. In *simple* graphs, there can be at most one edge from one node to another, whereby edges from a node to itself (so-called *self-loops*) are not permitted. In some applications, it is necessary to distinguish between different types of edges and nodes: A *multi-relational* graph can have several edge types to express different relations between the nodes. For this purpose, we extend the notation of the edges by the relation type r , for example $(u, r, v) \in \mathcal{E}$. Similarly, the nodes can be assigned to different types $\mathcal{T} = \{\tau_1, \dots, \tau_k\}$ so that we divide the set of nodes into disjoint sets $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \dots \cup \mathcal{V}_k$ where $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ holds for $i \neq j$. The graph definition can be extended by a type function $T : \mathcal{V} \rightarrow \mathcal{T}$ which assigns each node its type. Node types often exhibit certain attributes, which is why this is also referred to as *attributed nodes*. In *heterogeneous* graphs, both nodes and edges are annotated with types. Their edges are often constrained according to the node types so that certain edges can only connect nodes of certain types.

²In this thesis, the term “graph” is used to describe the abstract data structure. Other resources in the field of data mining may interchangeably utilize the term “network” to refer to the same type of data. However, we may use the latter to describe specific, real-world instantiations of this data structure, e.g., social networks, or to refer to artificial neural networks.

In literature, there are many more special forms of graphs; explanations of these are omitted in this thesis. As the assemblies studied in this thesis are represented by undirected, simple graphs with additional attributes, we confine the subsequent definitions to apply solely to this form.

The *neighbors* $\mathcal{N}(v) \subseteq \mathcal{V}$ of a node v are given by the set of nodes directly connected to it, i.e., $\mathcal{N}(v) = \{u \in \mathcal{V} \mid (v, u) \in \mathcal{E}\} = \{u \in \mathcal{V} \mid (u, v) \in \mathcal{E}\}$. The degree $\text{deg}(v)$ of a node v indicates the number of undirected edges it is involved in, which corresponds to the number its neighbors: $\text{deg}(v) = |\mathcal{N}(v)|$. In addition to the direct relationships represented by edges, the indirect relationships in a graph incorporating multiple nodes are also of interest: A *path* is a sequence of edges such that the start node of an edge is the end node of the preceding edge. Its length is defined as the number of its edges. If two nodes $u, v \in \mathcal{V}$ are connected by a path, v is *reachable* from u and vice versa. An undirected graph is called *connected* if there is a path between any two nodes. If the start node and end node of a non-empty path are the same, it is called a *cycle*. A graph without cycles is called *acyclic*, a connected graph without cycles is called a *tree*. Following the terminology used for trees, in general graph structures we also refer to a node v with exactly one neighbor ($|\mathcal{N}(v)| = 1$) as a *leaf node*. For a path of length n , i.e., n consecutive edges within the sequence, the start and end nodes are referred to as *n -hop distant*. Based on this, the definition of neighbors introduced above, more precisely to be referred to as 1-hop neighbors, can be extended to n -hop neighbors $\mathcal{N}_n(v)$ which includes all nodes *up to* n hops distant. The graph *diameter* is the length of the longest shortest path between any two graph nodes, i.e., the length of the shortest path between the most distant nodes. A graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ is called a *subgraph* of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$ holds. Thus, each graph is a subgraph of itself. The largest connected subgraphs are then the *connected components* of the graph. Isolated nodes of a graph are trivial connected components. Note that a graph that has exactly one connected component is connected. Given a connected subgraph \mathcal{G}' of \mathcal{G} , if there exists another connected subgraph \mathcal{G}'' of \mathcal{G} which \mathcal{G}' is also a subgraph of, then \mathcal{G}'' *subsumes* \mathcal{G}' . Otherwise, \mathcal{G}' is a connected component of the graph.

Constructive and Destructive Operations The process of constructing new graphs from existing ones often involves the addition or removal of nodes and edges. We consider the operations for simple graphs without any node or edge information. If v is a node of graph \mathcal{G} , we define the removal $\mathcal{G} \setminus \{v\}$ as the subgraph that results from removing v from \mathcal{V} and, consequently, removing all edges involving v from \mathcal{E} , formally defined as $\mathcal{G} \setminus \{v\} = (\mathcal{V} \setminus \{v\}, \mathcal{E} \setminus \{(u, u') \mid v = u \vee v = u'\})$. We call a node of a connected graph *cohesive*³ if its removal would result in a graph composed of multiple connected components. Adding new nodes to a graph simply extends its node set. When adding new edges, the corresponding nodes must either already belong to the graph or its node set must be extended accordingly.

Representing Graphs A convenient representation of graphs is through an *adjacency matrix* $\mathbf{A} \in \mathbb{B}^{|\mathcal{V}| \times |\mathcal{V}|}$. Its entries specify the number of edges between the nodes. As we do not consider multiple edges between the same pair of nodes, the entries can take binary values: either 1 indicating the presence of an edge or 0 otherwise. Therefore, we need to put the nodes in an arbitrary order so that every node indexes a particular row and column in the adjacency matrix. This ordering is necessary for its representation with algebraic objects like matrices, but has no bearing on the structure of the underlying graph itself.

³In literature, such a node is also referred to as a “strong articulation point” in directed graphs [67].

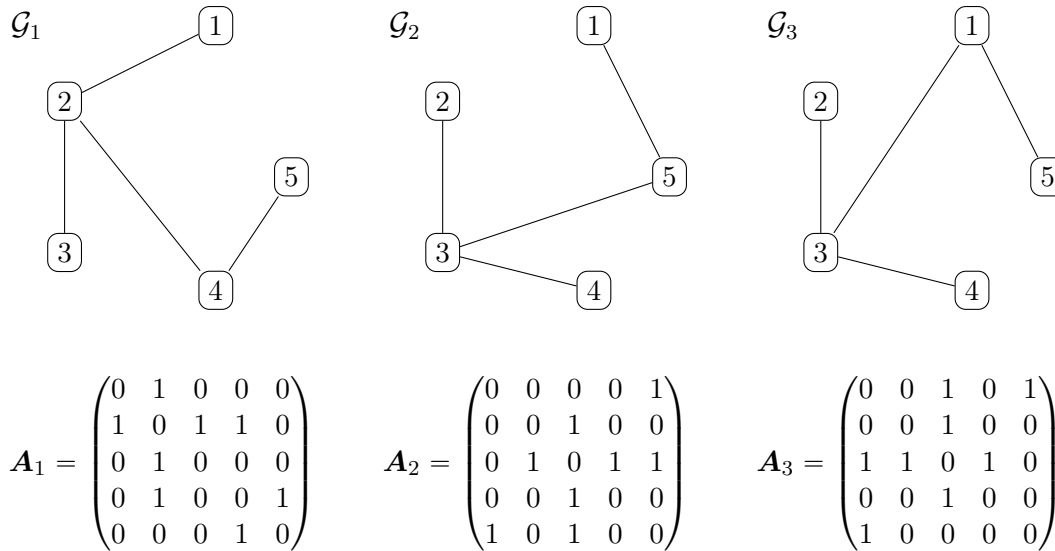


Figure 3.3: Three isomorphic simple graphs \mathcal{G}_i , $i = 1, 2, 3$, with different node orderings, their corresponding adjacency matrices \mathbf{A}_i are displayed below. The nodes are each annotated with their row/column index in the corresponding adjacency matrix. Example taken from [138, Figure 1.3].

Figure 3.3 shows three exemplary graphs and their corresponding adjacency matrices. If the graph is undirected, the adjacency matrix will be symmetrical, i.e., $\mathbf{A} = \mathbf{A}^\top$. Based on this node ordering, the node degrees can also be consolidated in a so-called *degree matrix* $\mathbf{D} \in \mathbb{N}_0^{|\mathcal{V}| \times |\mathcal{V}|}$. For undirected graphs, this is a diagonal matrix, where the degrees of the nodes can be found on the main diagonal while all other entries are zero: $d_{ii} = \sum_{j=1}^{|\mathcal{V}|} a_{ij}$.

For graphs with attributed nodes, the attributes are typically described by feature vectors which can be stacked to a matrix as well: node-level attributes $\mathbf{x}_v \in \mathbb{R}^d$ consisting of d scalar features for each node $v \in \mathcal{V}$ form a node feature matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d}$. For reasons of simplicity, we will limit ourselves here to real-valued features. We assume the ordering of the nodes to be consistent to the one in the adjacency matrix. For heterogeneous graphs, each different type of node has its own type of attributes, represented in separate feature matrices for each type. Attributes can also be associated with the graph itself, typically serving as labels for a classification or regression graph-level task, e.g., predicting properties of molecules.

Graph Isomorphism Finally, we would like to address the comparison of graphs. The transition from unordered node and edge sets to the representation with algebraic objects like an adjacency matrix using a random but fixed order of the nodes complicates the comparison of graphs. If two graphs differ at most in the order of their nodes, but not in their graph structure and attributes, the two graphs are called *isomorphic*. Formally defined, if we have two graphs with adjacency matrices \mathbf{A}_i and node feature matrices \mathbf{X}_i for $i = 1, 2$ respectively, the two graphs are isomorphic if and only if there exists a *permutation matrix* $\mathbf{P} \in \mathbb{B}^{|\mathcal{V}| \times |\mathcal{V}|}$ such that $\mathbf{P}\mathbf{A}_1\mathbf{P}^\top = \mathbf{A}_2$ and $\mathbf{P}\mathbf{X}_1 = \mathbf{X}_2$ hold. A permutation matrix is a square binary matrix in which exactly one entry in each row and column is one and all other entries are zero. Figure 3.3 shows three isomorphic graphs with different node

orderings and their corresponding adjacency matrices, whereby the nodes in the graphs are annotated with their row/column index, respectively. The graphs do not have any node or edge features. When looking at the adjacency matrices, it is hard to decide whether the graphs are isomorphic. The middle graph \mathcal{G}_2 can be easily transformed to the right one \mathcal{G}_3 by swapping nodes 1 and 5.

The Weisfeiler-Lehman Graph Isomorphism Test Even though it has a straightforward definition, assessing graph isomorphism presents an inherently challenging task. A naive approach would involve searching over the full set of permutation matrices to evaluate whether there exists a single one that leads to an equivalence between the two graphs, leading to computational complexity $\mathcal{O}(|\mathcal{V}|!)$. The graph isomorphism problem is in the complexity class NP, however it does not belong to its two subclasses: The problem is known to be not NP-complete, but also no polynomial time algorithm exists that correctly tests isomorphism for general graphs; thus the problem is referred to as *NP-indeterminate* or *NP-intermediate* (NPI). At the same time, isomorphism for many special classes of graphs can be solved in polynomial time, and in practice graph isomorphism can often be solved efficiently. The Weisfeiler-Lehman algorithm [147] is known to solve the isomorphism problem for a broad set of graphs [128, 5]. We will sketch the procedure in its simplest form, commonly referred to as “1-WL”. It is based on the idea of iterative neighborhood aggregation: The approach extracts node-level features that contain more information than simply the node neighborhood and aggregates these richer features into a graph-level representation. In a first step, an initial label $l_v^{(0)}$ is assigned to each node $v \in \mathcal{V}$, e.g., the node degree or given node features. After that, the algorithm iteratively assigns a new label to each node by hashing the multiset (denoted as $\{\cdot\}$) of current labels of the node’s neighborhood including the node itself: $l_v^{(i)} = \text{hash}(\{l_u^{(i-1)} \mid u \in \mathcal{N}(v) \cup \{v\}\})$. After running a given number of κ re-labeling iterations, a label $l_v^{(\kappa)}$ summarizes the structure of its κ -hop neighborhood. Two graphs are declared to be isomorphic if and only if their graph-level multisets $L_{\mathcal{G}} = \{l_v^{(i)} \mid v \in \mathcal{V}, i = 0, \dots, \kappa\}$ are identical. Comparing all computed labels after κ iterations of the Weisfeiler-Lehman algorithm is a popular approach to approximate graph isomorphism. It is able to tell if two graphs are non-isomorphic, but it cannot guarantee that they are isomorphic.

Permutation Equivariance and Invariance The representation of graphs with adjacency and feature matrices is useful due to the efficient interoperability with existing machine learning frameworks, but it enforces the nodes to be ordered. Algorithms and models that process graphs should ideally calculate node and graph representations regardless of the order of the nodes in the matrix representations by design. If they do not, every single model would have to learn that the order is irrelevant, in addition to the actual learning task. In short, the processing of two isomorphic graphs should internally lead to the same representation. This property is called permutation invariance: A function f is called *permutation invariant* if permuting the input, i.e., multiplying with a permutation matrix, does not affect the output, i.e., $f(\mathbf{P}\mathbf{X}) = f(\mathbf{X})$. In contrast, the property of the function f that permuting the output of f is the same as applying f to the permuted input is called *permutation equivariance*, formally defined as $f(\mathbf{P}\mathbf{X}) = \mathbf{P}f(\mathbf{X})$. Informally, permutation invariance means that the function’s output does not depend on the arbitrary ordering of the rows and columns in the input, while permutation equivariance means that the ordering of the function’s output is consistent to the one in the input. In the context

of graphs, these properties are often expressed using the adjacency matrix. As this takes into account the node order in both the rows and columns, the permutation matrix must be multiplied to the adjacency matrix both from the left and transposed from the right, i.e., $f(\mathbf{PAP}^\top)$.

Machine learning models processing graphs consist of a composition of permutation invariant and equivariant functions (i.e., layers), depending on the use case, since concatenation of permutation invariant or equivariant functions preserves the property. For graph-level tasks such as graph classification, the last layers are typically permutation invariant, since the classification is independent of the order of the nodes. For tasks at node or edge level, however, the order is indeed relevant for the output, which is why permutation equivariant functions are used. Details on how to design a permutation equivariant or invariant neural network layer can be found in Section 3.3.2.

3.3 Graph Machine Learning

The following explanations are mainly based on [59], unless otherwise stated.

Graph structures occur in many domains, for example as molecules in chemistry, meshed surfaces in computer graphics, sensor networks or social networks in information theory, traffic networks or knowledge graphs, to name a few. Hence, they were studied early on; the first work on graph theory by Leonhard Euler dates back to the 18th century. Due to the ever-increasing scale and complexity of graph datasets, machine learning techniques were used to model, analyze and understand graph data. The field of analyzing graph-structured real-world data without machine learning is nowadays referred to as network analysis [59]. Early work on processing graph data with machine learning techniques was based on extracting hand-crafted statistical features that served as input to standard models like logistic regression. As is well known, hand-crafted features are inflexible as they cannot be adapted through learning and are typically application-specific and time-consuming to build. Thus, the approaches proceeded to *learning* the node and graph representations, which includes the extraction of structural information about the graph.

The difficulty in processing graphs is their non-Euclidean nature. The Euclidean spaces are defined by \mathbb{R}^n for some finite dimension n ; thus data is called Euclidean, if they can be modelled in Euclidean space. For a long time, machine learning was applied on Euclidean data such as tabular data, images or sequences. For images for example, each pixel can be represented in a 3-dimensional space where x and y specify its coordinate and the z -axis represents the color or intensity. This spatial representation instead of using a vector induces so-called *geometric prior* [17], i.e., assumptions on spatial properties of the data space. The term “prior” goes back to the mathematical concept of the prior probability distribution in probability theory and expresses initial knowledge or belief about the data space before observing specific instances [51]. In the field of computer vision, for example, such a prior could be the belief that detecting edges on the image could be helpful for object recognition.

Having priors (especially in models) is important as they restrict the set of function fitting the data to those adhering to them. Non-Euclidean data, such as graphs and manifolds, do not naturally fit into \mathbb{R}^n . Let us consider a graph without any node or edge features; there is no natural translation of a set of nodes into the Euclidean space. If we map the nodes to certain points in this space, we might establish a certain prior that does not actually exist between the nodes as in Euclidean space, nearby points are similar

to each other. The distance measure in graphs is also completely different: the distance between nodes is defined by the length of the shortest path, as opposed to the length of the straight line (Euclidean distance) between the points. A correct mapping establishing no false priors of the graph structure might still be possible for small graphs, but fails for more complex ones. Thus, using a space with a more suitable prior is preferable.

Geometric Deep Learning In mathematics, many alternative spaces have been defined over time based on other geometries deviating from Euclidean laws. In 1872, a German mathematician, Felix Klein, presented a framework to unify all these geometries: he proposed to treat geometry as the study of invariances and symmetries, defining a geometry by specifying its domain and what symmetries the data must adhere to. This concept shift was adopted to machine learning in 2017 by Bronstein et al. who introduced the term *Geometric Deep Learning* [17, 18]. This group-theoretical framework attempts to generalize deep neural networks to non-Euclidean domains. The approach is driven by the idea to pick suitable model architectures according to the prevalent symmetries and invariances of the domain underlying data. In terms of graphs, the symmetry is node ordering as the nodes are defined as a set. Thus, we define equivariance or invariance to node permutation and aim for neural network layers with these properties.

3.3.1 Categorization of Learning Tasks on Graphs

Machine learning on graphs – also referred to as *graph machine learning* or short *GraphML* – can also be categorized into supervised and unsupervised tasks. However, there are more appropriate categorizations of graph learning tasks. First, the methods can be differentiated according to whether they are suitable for a single (usually large) graph like social networks or many (small) graphs like molecules – some algorithms fall in both categories. The second categorization of learning tasks refers to the component of the graph that is to be reasoned about, i.e., nodes, edges, subgraphs or the entire graph itself.

Node Classification The goal in this type of learning problem is to assign a class (represented by label y_v) to each node $v \in \mathcal{V}$ of an input graph, thus also called a *node-level* task. For example, for each user of a social network, we want to decide whether it is a bot. Other popular examples include classifying the topic of a document based on their citation linkage [79] or the category of a product based on the products co-purchase graph [94].

Node classification seems very similar to typical classification in supervised learning (each input is mapped to a label), but there are important differences. The *i.i.d. assumption*, i.e., the central assumption for generalizability from training to unseen instances in machine learning, does not hold for the nodes: Because of their connections, nodes are just not statistically independent of all others. Successful methods for node classification leverage these connections. Some of them exploit *homophily*, which describes the property of nodes to have similar attributes as their neighboring nodes. For instance, people typically form friendships with others who share similar interests or demographic characteristics. Following that property would lead algorithms to assign similar labels to neighboring nodes in a graph, i.e., propagating labels to neighboring nodes. In contrast, *heterophily* describes the property that nodes are preferentially connected to dissimilar nodes, i.e., with different attributes or labels. Node classification methods should leverage these concepts instead of simply treating each node as separate instance.

In the context of node classification, often label information is only available for a small subset of nodes in a graph, e.g., classifying bots in a social network from few manually labeled examples. Consequently, a new type of learning task was introduced: *transductive learning* as a differentiation from *inductive learning*. Inductive learning aims to infer *general rules* from (labeled) training data, in our context labeled nodes, which are then evaluated on unseen test data. For evaluation, unseen test nodes could be (incrementally) added to the current graph, or the trained model can be used to process completely unseen test graphs. Transductive learning however attempts to recognize patterns from all – both labeled training and unlabeled test – data in order to transfer them from the training instances to the test instances. This setup may be seen as propagating labels from training nodes to unlabeled nodes, typically used in homophilous graphs [143]. By their nature, transductive approaches cannot generalize to nodes outside the seen nodes during training, e.g., new users in a social network. Some studies also refer to the transductive approach as *semi-supervised*, since both labeled (training) and unlabeled (test) data are used. However, standard formulations of semi-supervised learning usually require compliance with the i.i.d. assumption, which, as described above, does not apply here.

Graph Classification and Regression This type of graph application is focused on predictions $y_{\mathcal{G}}$ for entire graphs \mathcal{G} (i.e., a *graph-level* tasks), instead of its individual components (i.e., edges or nodes). For example, one would like to predict chemical properties such as hydrophobicity (classification) or solubility (regression) for molecules. The datasets typically consist of several different graphs, where predictions are to be made specific to each graph. This makes this graph machine learning category probably the most similar to classical supervised learning. Typical approaches include component-wise predictions, which are then aggregated appropriately at the graph level. However, the challenge in these graph-level tasks lies in defining useful features that account for the relational structure within each instance. The task of graph clustering is quite related, where we aim to learn a similarity measure of graph pairs, which is analogous to clustering in unsupervised learning. Furthermore, clustering is also investigated on subgraphs, which is also referred to as community detection. This involves identifying groups in graphs according to their structural properties as nodes within these communities are much more likely to form edges than with nodes outside their community.

Edge/Link Prediction This class of machine learning applications to graph data deals with predicting whether two nodes should be connected by an edge, e.g., whether two people in a social network should be recommended to connect or if a specific content should be recommended to a user. More complex graph datasets like multi-relational biomedical knowledge-graphs require advanced reasoning and inference strategies [106]. Depending on the respective application and domain, this is also referred to as relation prediction, graph completion or relational inferring. They all have in common that the training data represents an incomplete set of edges $\mathcal{E}_{\text{train}} \subsetneq \mathcal{E}$ over a fixed set of nodes \mathcal{V} , while the goal is to predict the missing edges of $\mathcal{E} \setminus \mathcal{E}_{\text{train}}$. This includes datasets with single graphs as well as settings in which relations between several disjoint graphs are to be predicted. Like node classification, edge prediction also blurs the boundaries between traditional machine learning categories, thus being referred to as both a supervised and unsupervised problem.

3.3.2 Graph Neural Networks

A very simple approach to processing graph-structured data would be to neglect the edges and only process the set of nodes, for example every node independently with an MLP or Deep Sets [163]. In fact, in many ML applications, individual instances are in certain relations (for example, images of the same object from different angles), but these are typically disregarded and the instances are processed in isolation [144]. However, the domain data is typically structured as graphs for a reason: In the applications, it is assumed that the connections contain important information for the respective task and should therefore be taken into account. However, previous neural network architectures are unable to process graph structures, especially graphs of arbitrary size, which is why new models had to be developed specifically for this data structure.

Graph neural networks (GNNs) are a class of neural networks designed specifically for processing graphs. Numerous works have attempted to extend neural networks to handle arbitrary graphs. The first GNN model is typically attributed to Gori et al. [52], who introduced the term in 2005. They used random walks to extract sequences from graph structures, which were then processed using recursive neural networks [37, 131]. Their approach was restricted to directed acyclic graphs. Scarselli et al. [121] extended their formulation in 2009 by incorporating edge features. This was followed by the development of many other variants. Some work attempted to generalize the variety of emergent forms under a unified scheme, such as the *message passing neural network framework* by Gilmer et al. [43].

The basic GNNs can be motivated in different ways, we stick to the generalization of convolutions to non-Euclidean data first formulated by Bruna et al. in [19], considering graphs with attributed nodes without edge information. The main idea is to propagate information along the edges and iteratively update the node states.

The convolutional operation has some desirable properties that motivated the generalization from grid-like data such as images (or sequences in 2D) to general graph structures, confer Section 3.1. Due to the constant number of parameters in the convolutional filters, the operation is independent of the size of the input and thus both computationally and memory efficient. The geometric prior thus helps to avoid the curse of dimensionality. Furthermore, convolution works on the local neighborhood of a node, taking into account the structural information [143]. In the case of images, the neighborhood of a pixel considered during convolution is always the same size and these neighbors have a fixed order – such structural rigidity does not hold for general graph structures. Therefore, the convolution option must be generalized from an ordered collection of fixed size to an unordered set of arbitrary size as depicted in Figure 3.4. This was achieved by the *message passing paradigm* that GNN layers adhere to. Vector messages are exchanged between nodes and transformed along the edges using neural networks. Following a neighborhood aggregation scheme, a node’s representation is computed by iteratively aggregating its neighbors’ transformed messages. Thus, message passing represents a permutation equivariant function to update a graph’s node representations. Message passing only changes the node information, but not the structure of the graph.

Like all neural networks, GNNs consist of several stacked layers, cf. Figure 3.5. From the perspective of a node v , the general update in a message-passing layer l is given by:

$$\mathbf{h}_v^{(l+1)} = \phi^{(l)} \left(\mathbf{h}_v^{(l)}, \psi^{(l)} \left(\{ \mathbf{h}_u^{(l)} \mid u \in \mathcal{N}(v) \} \right) \right) \quad (3.9)$$

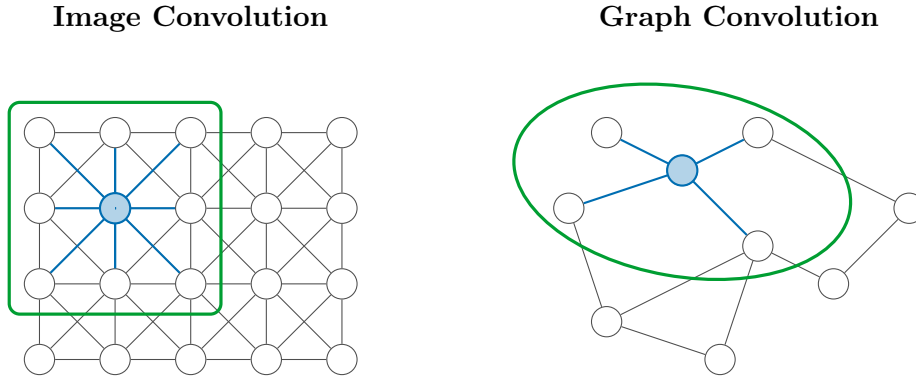


Figure 3.4: Comparison of image convolution (left) and graph convolution (right). The direct neighbors of the highlighted center node are framed in green, respectively. For image convolution, a (3×3) -kernel matrix takes into account a center pixel and its eight neighbors. While the neighbors of a node in an image are ordered and of same size, a graph node's neighbors $\mathcal{N}(v)$ are an unordered set of arbitrary size. Remake of [155, Figure 1].

where $\mathbf{h}_v^{(l)}$ denotes the representation of v in the l -th layer. Furthermore, $\phi^{(l)}$ and $\psi^{(l)}$ are parameterized (i.e., trainable) mappings, e.g., fully connected layers followed by a nonlinearity [155]. While $\psi^{(l)}$ transforms and aggregates the multiset of neighbor features, $\phi^{(l)}$ updates a node's representation.

Permutation invariance is achieved by $\psi^{(l)}$ accepting an arbitrary number of inputs for the aggregation. Simple examples for $\psi^{(l)}$ would be the sum, average or maximum. For notational simplicity, we stick to the curly bracket representation for multisets, too, also in the formulas of the following GNN architectures. The same mappings are used within a layer (parameter sharing). The initial node representation $\mathbf{h}_v^{(0)}$ are set to the input features for all nodes. Since the aggregation can be performed with any number of elements, GNNs can handle graphs with different structures and even different sizes. The node representations $\mathbf{h}_v^{(l)}$ have the same dimension for all nodes $v \in \mathcal{V}$ for a specific layer l , denoted as d_l . In each message passing layer, the information of the direct neighbors is aggregated. This means that at higher layers $l \geq 2$, however, node information of more distant nodes, precisely the l -hop neighbors, is also indirectly included as they have been already propagated along l edges.

In message passing layers, the representation of individual nodes is updated; but for graph-level tasks we need a representation for the entire graph. For this purpose, so-called *readout* or *pooling* layers are used, which aggregate the node representations appropriately into a global, graph-wide, representation:

$$\mathbf{h}_{\mathcal{G}} = \text{readout} \left(\{ \mathbf{h}_v^{(l)} \mid v \in \mathcal{V}_{\mathcal{G}} \} \right) \quad (3.10)$$

The readout function should also be permutation invariant, which can already be achieved by simple aggregators like sum or mean, for example. More sophisticated readout architectures are useful for large graphs; the survey [155] provides an overview of current readout approaches. Hence, with a subsequent fully connected layer and Softmax activation, a graph classification can be performed.

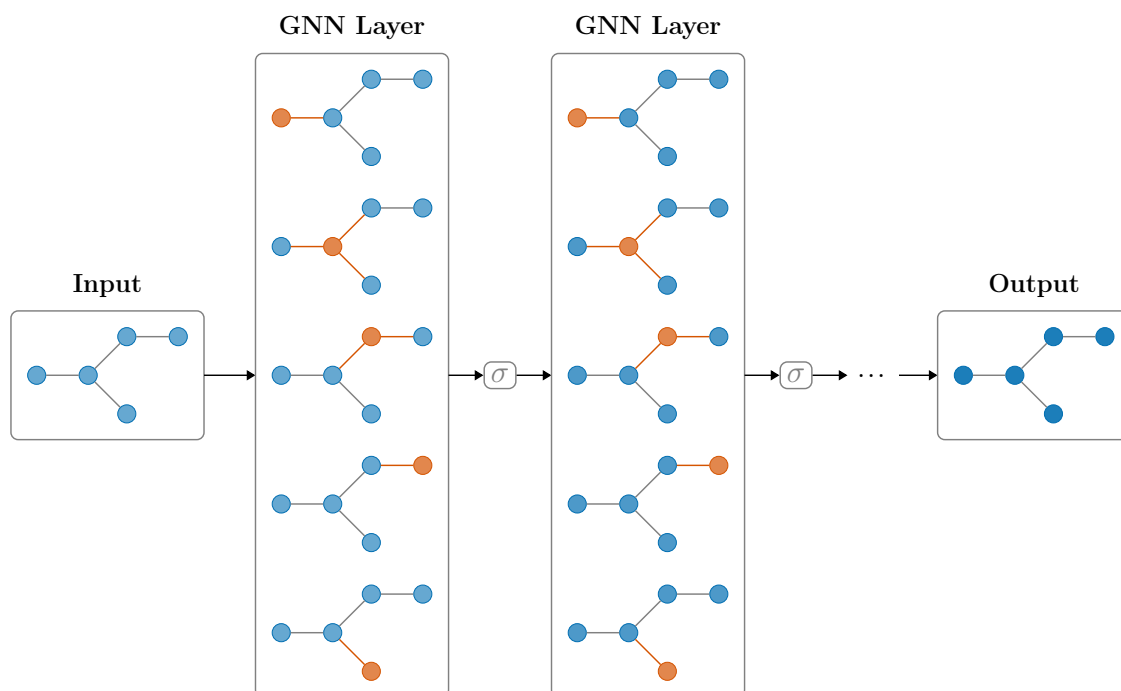


Figure 3.5: Visualization of a GNN “rolled out”: The input graph is processed by stacked GNN layers, each followed by nonlinearity σ . In a GNN layer, every node aggregates the features of its neighbor nodes and thus updates its own state. This is shown vertically per layer for every node of the input graph, where the corresponding center node and its edges are highlighted. Presentation taken from [69].

Relation to the Weisfeiler-Lehman Test The attentive reader may have noticed a certain similarity between the functioning of message passing in GNNs and the Weisfeiler-Lehman algorithm (Section 3.2), especially in terms of information aggregation. However, different goals are pursued with iterative aggregation, respectively: While with GNNs similar graphs should be embedded close to each other so that it is easy to generalize for inference, in isomorphism testing we want to obtain very *different* representations for similar but different graphs (which in fact is a property of hash functions). Despite similarity, the isomorphism test is not suitable for machine learning on graphs. However, it is used to assess the expressive power of a GNN and serves as inspiration for some architectures. A major finding in GraphML is that GNNs are no more powerful than the Weisfeiler-Lehman algorithm [101, 157]. Specifically, if the 1-WL isomorphism test assigns the same label to two nodes, then any message-passing GNN will do the same. Consequently, if the isomorphism test cannot distinguish two graphs, a message-passing GNN can neither.

Various forms of GNN architectures have been developed according to the message passing paradigm emerging from diverse application areas of graph data. Applications originate from different fields that usually have few points of contact, so publications tend to use different terminology and notation. For the sake of comparability, we have aimed for a uniform notation and terminology for the explanations of the following architectures. In

accordance with the *no free lunch theorem*⁴ [152], none of the architectures is consistently better – even though some address issues of former ones –, which is why we will present a selection of suitable architectures and examine them for our use cases. We would like to point out that all the following GNN architectures use linear transformations with feed-forward layers. For reasons of readability, however, the bias term is always omitted in the update rule equations. Since nonlinearity plays a crucial role in the expressive power of neural networks, linear transformations are always used with a nonlinear activation function that can be set as a hyperparameter. The dimension of the weight matrix controls the size of the resulting hidden representations as it maps vectors $\mathbf{z}^{(l)} \in \mathbb{R}^{d_l}$ to vectors $\mathbf{z}^{(l+1)} \in \mathbb{R}^{d_{l+1}}$.

Graph Convolutional Network (GCN)

Graph convolutional networks (GCNs) [79] have been introduced by Kipf and Welling for semi-supervised node classification in 2017. They outperformed former established approaches and were also more computationally efficient.

Let us derive and motivate the formula for a GCN layer in graph-level matrix notation step by step [144]. According to the initial features \mathbf{X} , the node representation after each layer l are stored row-wise in a graph-level node representation matrix $\mathbf{H}^{(l)} \in \mathbb{R}^{|\mathcal{V}| \times d_l}$. Transforming and aggregating the node representation can be expressed by applying a learnable linear transformation $\mathbf{W}^{(l)}$ (feed-forward layer) and multiplying with the adjacency matrix \mathbf{A} , followed by a nonlinearity σ such as ReLU:

$$\mathbf{H}^{(l+1)} = \sigma \left(\mathbf{A} \mathbf{H}^{(l)} \mathbf{W}^{(l)\top} \right) \quad (3.11)$$

The first dimension of the weight matrix serves as hyperparameter and determines the dimension of the resulting node representations. This equation illustrates that only the information of the neighboring nodes is used to update the node information, i.e., the information of the node itself is discarded, which can lead to severe information loss. Therefore, a small correction is performed by implicitly inserting self-loops so that the node itself counts as a neighbor of itself. The adjusted adjacency matrix $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_{|\mathcal{V}|}$ results from the sum of the original adjacency matrix and the identity matrix of same size. However, this adjustment modifies the scale of the resulting features, and therefore normalization is needed. The GCN update rule uses symmetric normalization:

$$\mathbf{H}^{(l+1)} = \sigma \left(\tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \mathbf{H}^{(l)} \mathbf{W}^{(l)\top} \right) \quad (3.12)$$

where $\tilde{\mathbf{D}}$ refers to the degree matrix of $\tilde{\mathbf{A}}$, i.e., $\tilde{d}_{ii} = \sum_j \tilde{a}_{ij}$. According to the authors, these modifications resulted in a positive impact on training performance. The symmetric normalization follows the formulation of generalized convolutions to graphs by [19].

For comparison with the other GNN architectures, we also introduce the formula from the perspective of a single node. Note that the inclusion of the node itself for the update is not visible in the node-wise formula, as this is achieved by previously adapting the graph

⁴The no free lunch theorem [152] states, that no machine learning algorithm is universally any better than any other. Specifically, averaged over *all* possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved inputs [51].

structure. The node-wise update rule of a GCN layer is given by

$$\mathbf{h}_v^{(l+1)} = \sigma \left(\sum_{u \in \mathcal{N}(v)} \frac{1}{c_{vu}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l)} \right) \quad (3.13)$$

with normalization constant $c_{vu} = \sqrt{|\mathcal{N}(v)| \cdot |\mathcal{N}(u)|}$. The constant heavily depends on the graph structure, not incorporating the node features, which is a limitation of this architecture. Kipf himself stated, that adaptable assigning weights to neighbors depending on their importance might be beneficial [80]. Some architectures, among them the next one, have therefore replaced the constant with a learnable function of the neighboring node features.

Graph Attention Network (GAT)

Graph attention networks (GAT) [144] address the limitation of fixed neighbor weighting as in GCNs by generalizing the self-attention operator (cf. Section 3.1.1) introduced in the Transformer architecture [141] to the graph domain. The key idea is to assign attention weights to neighbor nodes, allowing the network to focus on the most relevant ones for the specific task. Again, we consider the calculation rules for one GAT layer. First, the node features are transformed: As for GCNs, at least one linear transformation, parameterized by a weight matrix \mathbf{W} , of the node features is required. Afterward, self-attention is performed on the nodes by an attention mechanism $a : \mathbb{R}^{d_l} \times \mathbb{R}^{d_l} \rightarrow \mathbb{R}$ on pairwise node representations, resulting in real-valued attention coefficients

$$e_{vu} = a(\mathbf{h}_v, \mathbf{h}_u) \quad (3.14)$$

indicating the importance of the features of node u to node v . A single feed-forward linear layer together with a nonlinear activation function σ represents the attention mechanism. In their paper [144], they employ LeakyReLU with a negative input slope of 0.2 as activation function. As a scalar output is desired, the corresponding weight matrix actually is a weight vector $\mathbf{a} \in \mathbb{R}^{2d_l}$:

$$a(\mathbf{h}_v, \mathbf{h}_u) = \sigma(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_v \parallel \mathbf{W}\mathbf{h}_u]) \quad (3.15)$$

where \parallel represents vertical vector concatenation. Both the linear transformation and attention mechanism are shared within a GAT layer. The most general formulation allows attending every node to every other node, neglecting all structural information. By performing *masked* attention, i.e., only computing the coefficients for adjacent nodes (or any other desired neighborhood definition), the model takes the structural information into account. Similar to GCNs, the node itself is included in its neighborhood, so that attention coefficients are computed for every node and direct neighbor as well as itself. To ensure that the coefficients are easily comparable across different nodes, they are normalized across all choices of neighbors using the Softmax function to attention *weights*, written out as

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{w \in \mathcal{N}(v)} \exp(e_{vw})} \quad (3.16)$$

Combining all computational steps, the node-wise update rule of a GAT layer l is given by

$$\mathbf{h}_v^{(l+1)} = \sigma \left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(l)} \mathbf{W}^{(l)} \mathbf{h}_u^{(l)} \right) \quad (3.17)$$

including a nonlinearity σ and with attention weights

$$\alpha_{vu}^{(l)} = \frac{\exp \left(\sigma \left(\mathbf{a}_{(l)}^\top \left[\mathbf{W}^{(l)} \mathbf{h}_v^{(l)} \parallel \mathbf{W}^{(l)} \mathbf{h}_u^{(l)} \right] \right) \right)}{\sum_{w \in \mathcal{N}(v)} \exp \left(\sigma \left(\mathbf{a}_{(l)}^\top \left[\mathbf{W}^{(l)} \mathbf{h}_v^{(l)} \parallel \mathbf{W}^{(l)} \mathbf{h}_w^{(l)} \right] \right) \right)} \quad (3.18)$$

Again, the dimension of the resulting node representations $\mathbf{h}_v^{(l+1)}$ is a hyperparameter determined by the number of rows of the weight matrix.

Compared to the node-wise update rule for GCN in Equation (3.13), only the calculation of the weights of the neighboring nodes has changed. Whereas with GCNs this depended solely on the node degrees of the neighbors, with GATs they are calculated depending on the node features. Thus, a GAT allows for assigning different weights (importance) to neighbors.

Following the findings of [141] that employing *multiple* independent attention mechanisms per layer stabilizes the learning process of self-attention, *multi-head* attention is included. This means, that $\kappa \in \mathbb{N}$ independent mechanisms execute the transformations of Equation (3.17). Their outputs must then be suitably aggregated into one; Veličković et al. suggest concatenation for this. Consequently, the dimension of $\mathbf{h}_v^{(l+1)}$ per node equals $\kappa \cdot d_{l+1}$ instead of d_{l+1} . The number of heads is a second hyperparameter of a GAT layer. For the final layer however, typically averaging over all attention heads is employed. By using multi-head self-attention, the node-wise update rule changes to

$$\mathbf{h}_v^{(l+1)} = \left\| \prod_{k=1}^{\kappa} \sigma \left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(l,k)} \mathbf{W}^{(l,k)} \mathbf{h}_u^{(l)} \right) \right. \quad (3.19)$$

and for the final layer using average to

$$\mathbf{h}_v^{(l+1)} = \sigma \left(\frac{1}{\kappa} \sum_{k=1}^{\kappa} \sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(l,k)} \mathbf{W}^{(l,k)} \mathbf{h}_u^{(l)} \right) \quad (3.20)$$

The second element of the superscript (l, k) of the attention weights and weight matrix denotes the index of the corresponding attention mechanism. As the computations of the individual heads are completely independent, they can be parallelized, thus making GAT layers highly efficient. Although more computations were included by the attention mechanism, the computational complexity of a GAT layer is at the same level as a GCN layer.

GraphSAGE

GraphSAGE was introduced by Hamilton et al. in 2017 [58] as framework for learning representations especially for large graphs like social networks. The architecture was designed in order to tackle the problem of transductive node representation methods to not efficiently generalize to unseen nodes. The inductive single-graph datasets analyzed in this paper were a citation network consisting of 300 thousand nodes with average node degree of 9, and a network of Reddit posts consisting of 230 thousand nodes with average degree of nearly 500. The difficulty in processing such large graphs is the number of nodes (and thus the computational operations), but also the connection intensity, for example with hub nodes like celebrities on social media.

As a form of GNNs, the node information is updated by exchanging messages with neighboring nodes – but with a slight modification. The main novelty of this architecture is a neighbor sampling step which results in the benefit of scalability to graphs with high node degrees. Instead of taking all direct neighbors $\mathcal{N}(v)$ of every node v into account, only a random sample $\mathcal{N}^*(v) \subseteq \mathcal{N}(v)$ of them is considered. In every layer and for every processing of a graph, a new sample is chosen randomly. The specific sampling strategy is a hyperparameter that needs to be selected. Still, every node is represented by some aggregation of its neighbors, which works even for new unseen nodes. In the GCN and GAT architectures discussed so far, only direct neighbors are considered, although an adaptation of the neighborhood definition to neighborhood sampling or also integrating more distant nodes would definitely be possible. The downside of sampling is potential information loss as well as the higher variance of the gradient due to randomization during training which could make the model harder to train.

The second important modification in GraphSAGE is that the function for aggregating the neighbors' representation is a *learnable* function. Formally, the intermediate node representation after aggregating the sampled neighbors' representation can be written as

$$\tilde{\mathbf{h}}_v^{(l)} = \text{aggregate} \left(\{ \mathbf{h}_u^{(l)} \mid u \in \mathcal{N}^*(v) \} \right) \quad (3.21)$$

The authors propose three aggregator functions (hyperparameter) in their work, in order to make their approach applicable to many applications: mean, pooling and LSTM. (i) For the mean operator, simply the element-wise mean of the sampled neighbor's representation is taken. This aggregator is quite similar to GCNs, however does not incorporate the normalization based on the node degrees. (ii) For the pooling aggregator approach, the nodes representations are independently transformed by a shared MLP and element-wise max-pooling is applied to the resulting vectors. Both of these operators are permutation invariant as opposed to the last operator. (iii) LSTMs process inputs in a sequential manner, thus they actually seem to be an unsuitable choice. This problem was addressed by applying a random permutation to the nodes to force the model to only focus on the features itself rather than on their order.

The aggregated neighbor information $\tilde{\mathbf{h}}_v^{(l)}$ is used together with the representation of the node itself to update its state. As usual, a linear transformation $\mathbf{W}^{(l)}$ and a nonlinearity σ are used. In summary, this results in the node-wise update rule of a GraphSAGE layer by

$$\mathbf{h}_v^{(l+1)} = \sigma \left(\mathbf{W}^{(l)} \left[\mathbf{h}_v^{(l)} \parallel \tilde{\mathbf{h}}_v^{(l)} \right] \right) \quad (3.22)$$

The focus of GraphSAGE is primarily on scalability. In contrast to GAT, the neighbors cannot be assigned different importance in the neighborhood aggregation.

Graph Isomorphism Network (GIN)

The last presented variant of GNNs, Graph Isomorphism Networks (GIN), were introduced in 2018 by Xu et al. in their paper [157]. They found out that some GNN architectures like GCN and GraphSAGE cannot distinguish some graph structures by design, i.e., they represent these similar but different graphs the same way. Therefore, the authors designed the GIN architecture in order to create a more representational powerful GNN. The goal was to be able to distinguish the same graph structures as the Weisfeiler-Lehman isomorphism test [147] can.

Figure 3.6 illustrates some examples of graph structures that cannot be distinguished by GCN and GraphSAGE. Their main design problem is using mean or max-pooling as aggregation operation. Let's consider an illustrative example when applying mean as aggregation operator (cf. right example in the figure): assume we are having two graphs; in the first the node v has the two neighbors a and b , while in the second graph the two nodes each occur twice in the neighborhood of v . When performing neighbor aggregation, node v is going to obtain the same representation in both graphs because $\text{mean}(\mathbf{x}_a, \mathbf{x}_b) = \text{mean}(\mathbf{x}_a, \mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_b)$ holds, where \mathbf{x}_i represents a node i 's representation. For this example, also max-pooling fails to produce different representations.

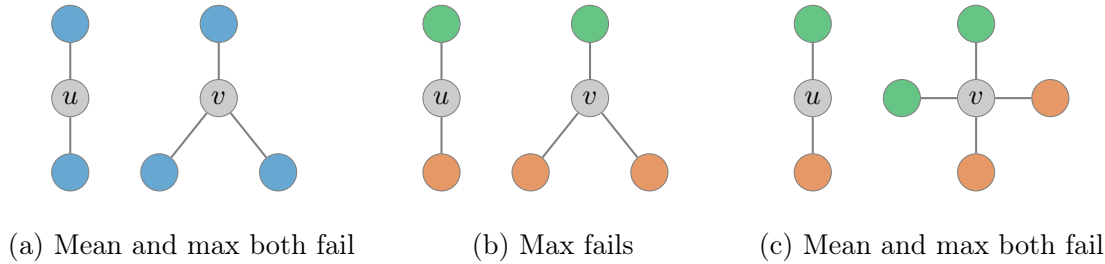


Figure 3.6: Three examples of graph pairs that GCN and GraphSAGE models cannot distinguish due to their aggregator operators mean and max. Unlabeled nodes of the same color have the same representation vector. Between the two graphs, nodes u and v get the same updated representation even though their corresponding graph structures differ. The subtitles indicate which aggregation operators fail when differentiating between the two graphs. Remake of [157, Figure 3].

The problem described above occurs to a much lesser extent when using the sum as an aggregation operation, which is why it is used for GINs. However, further architectural differences to the models considered so far can be found. The GIN architecture is based on the finding, that GNNs can have as large discriminative power as the Weisfeiler-Lehman test, if their aggregation scheme is also an *injective* function. An injective function f maps two different elements to two different images, formally $x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$. In the context of GINs, the elements are multisets of node vectors.

The authors have shown that an update function g of a GNN layer l for a multiset \mathcal{S} (i.e., the node neighbors) and a single element c (i.e., the node itself) of the form

$$g^{(l)}(c, \mathcal{S}) = \phi^{(l)} \left(\left(1 + \epsilon^{(l)}\right) \cdot f^{(l)}(c) + \sum_{s \in \mathcal{S}} f^{(l)}(s) \right) \quad (3.23)$$

for any $\epsilon^{(l)} \in \mathbb{R}$ fulfills these properties for suitable functions $f^{(l)}$ and $\phi^{(l)}$. We generously omit the mathematical details here and refer the interested reader to [157] for the derivation. Due to the universal approximation theorem [61], these functions can be learned by MLPs; by clever rewriting to function concatenation $f^{(l+1)} \circ \phi^{(l)}$ over consecutive layers, this can even be solved with a single MLP. We would like to explicitly point out that the MLP must have at least two layers so that the universal approximation theorem applies. The number of layers and their respective output dimensions are hyperparameters. Summarizing, the node-wise update rule of a GIN layer then takes the following form:

$$\mathbf{h}_v^{(l+1)} = \text{MLP}^{(l)} \left(\left(1 + \epsilon^{(l)} \right) \cdot \mathbf{h}_v^{(l)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(l)} \right) \quad (3.24)$$

The scalar $\epsilon^{(l)}$ can be a learnable parameter or pre-set to a fixed value (hyperparameter).

Finally, we would like to provide an important note: Although the GIN architecture is in theory superior to other GNN architectures, this does not always translate to real-world problems. In theory, an MLP is able to map the required functions, but it is not given that this will always be achieved. Therefore, despite the theoretical disadvantages, GCNs, for instance, can yield better results in practical tests.

Issues of Graph Neural Networks Various problems can occur when using GNNs, the two most important of which are *over-squashing* and *over-smoothing* [59]. Over-squashing refers to the phenomenon of information loss, when too much information or information over a too long dependency needs to be compressed into a fixed-size vector. This can occur, if the learning task requires long-range dependencies for far away nodes or the graph structure exhibits bottlenecks, e.g., connecting two strongly connected subgraphs. The problem is therefore more likely to occur with larger graphs. It can be alleviated by adapting the graph structure, also referred to “rewiring” [136]. In contrast, in the case of over-smoothing, the node information is distributed too much so that all node representations become very similar because the node-specific information becomes washed out after several iterations. This phenomenon can be observed in cases where the node neighbors dominate the node update. It becomes problematic when the learning task requires a deep architecture, but the expressive power decreases as the depth increases. A natural way to address this issue is to employ so-called *skip connections*, i.e., additional connections of a layer to layers further up in the network, in order to preserve information from previous layers [59].

Transformers are Graph Neural Networks The invention of the Transformer [141] architecture in 2017, initially developed for machine translation, has led to a major breakthrough in the field of NLP where nowadays nearly every use case is dominated by Transformers [151]. In contrast to RNNs, which were used in the past, Transformers do not process the input sequentially but calculate the importance of all input elements among each other based on their features (self-attention mechanism) and from this an updated internal representation of all elements. In other words, each element’s representation is an aggregation of all other elements’ transformed features. This bears some similarities

to the message passing paradigm of GNNs: Recapitulating, GNNs learn node representations by iteratively aggregating its neighbors’ transformed features. The main difference is that Transformers use the features of *all* elements to update an element’s features whereas GNNs employ restrictions to only use the representation of the *direct neighborhood* for a node update [103]. Let’s consider a sentence as fully connected graph where every word is connected to every other word, as displayed in Figure 3.7, to clarify the connection. Now, a GNN can be used to build features for each word and thus perform NLP tasks. Neglecting the attention mechanism, GNNs are basically Transformers which operate on arbitrary graph structures, not restricted to fully connected graphs. Since Transformers make minimal assumptions about the structure of their input (a set of elements with optional positional information), they usually require more training data to grasp the concept compared to architectures with stronger inductive biases, like GNNs [103].

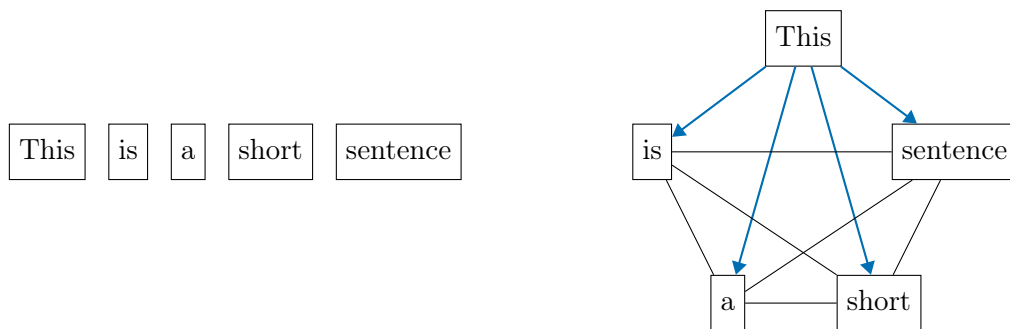


Figure 3.7: A sentence represented as fully connected graph. Directed edges starting from first word “This” are highlighted.

The relationship between GNNs and Transformers as well as the great success of Transformers led to the desire to apply them for arbitrary graph structures as well. This gave rise to different architectural variants of so-called *Graph Transformers*, which still utilize the attention mechanism introduced for standard Transformers [141], but handle the unique challenges posed by graph-structured data in different ways.

3.4 Embedding Discrete Objects

Datasets can have both numerical continuous and discrete (also called categorical) features, and the labels of the instances can also appear in both forms. Obvious examples would be natural language text composed of words or image classification. While the numerical features and targets can be used directly for processing in ML models (after some preprocessing steps, if necessary), categorical features still need to be converted beforehand into a suitable numerical representation. It may seem natural to count the number of disjoint values k of a categorical attribute and assign a numerical value between 0 and $k - 1$ to each value. However, numerical algorithms are based on the assumption that numbers that are closer together are more closely related than values that are further apart – this ties in with the rules in Euclidean space. Consequently, categories 0 and 1 need to be more similar than e.g., category 0 and $k - 1$ (for $k > 2$). In most cases, this order can only be found with a great deal of domain knowledge and therefore manual effort, or not at all. Alternatively, a generally valid approach is one-hot encoding the attribute, i.e., creating a binary column for each categorical value. However, this encoding also has drawbacks:

The dimension of the one-hot encoding corresponds to the number of attribute values, which means that a high number of attribute values may slow down training and degrade performance due to the curse of dimensionality. Furthermore, this representation does not express any similarity, as the one-hot vectors are orthogonal to each other⁵. Instead of high-dimensional and sparse representations, low-dimensional and dense representations are more suitable for learning.

3.4.1 Learning Embeddings

In the past, *feature engineering* involved the selection or construction of appropriate features for a learning problem, primarily relying on mathematical methods. However, a more contemporary approach is feature learning or *representation learning*, where the dense representations themselves are learned by neural networks, either by an upstream model or during training the task model. Specifically, this refers to learning a mapping from one discrete object (type) to a point in the vector space – commonly referred to *embeddings* [51]. The dimension of the embedding is a hyperparameter that needs to be tuned. The positioning of the objects in the vector space is intended to provide added value for subsequent algorithms, typically so that similar objects should be closer in the vector space. Learning-based embedding methods are currently state-of-the-art because of their broad applicability and performance. A major advantage is that these embeddings often can be reused successfully for other similar tasks (by applying fine-tuning or transfer learning), e.g., in the case of word embeddings. Individual axes in the learned space could, for example, be “brightness” or “weight”, which are used to arrange words in the embedding space. There, the word “sun” would be assigned a significantly higher value on both axes than, for instance, the word “lamp”, because the sun is both brighter and heavier than a lamp. Therefore, embeddings are also referred to as distributed representations, since the meaning and other properties of a word are distributed across different dimensions. In practice, though, very few of the resulting features are interpretable to humans. However, the coded similarity of the vectors enables the models to better understand inputs that were seen similarly during training: By having similar vectors for similar words, a model can more easily recognize the similarity of sentences, even if the exact words used are slightly different, resulting in better generalizing models. For example, in a trained word embedding, all animals should be arranged in one region, although pets such as cats and dogs should be much closer together (as they are often mentioned in the same context) than other animals such as elephants or whales.

3.4.2 Word2Vec

Representing discrete objects as dense vectors is at the core of deep learning’s successes in NLP. In 2000, Yoshua Bengio et al. provided a major contribution to natural language processing through their publication “A Neural Probabilistic Language Model” [12], in which they introduced high-dimensional word embeddings gained from neural networks capturing semantic relationships between words and thus representing word meaning. In 2013,

⁵Similarity of vectors is often measured by the *cosine distance*, which is the dot product of the vectors normalized by the product of their magnitudes: $S_C(\mathbf{x}, \mathbf{y}) = \cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|}$, where θ denoted the angle between the vectors. For two different one-hot vectors, their dot product and thus the cosine distance equals 0.

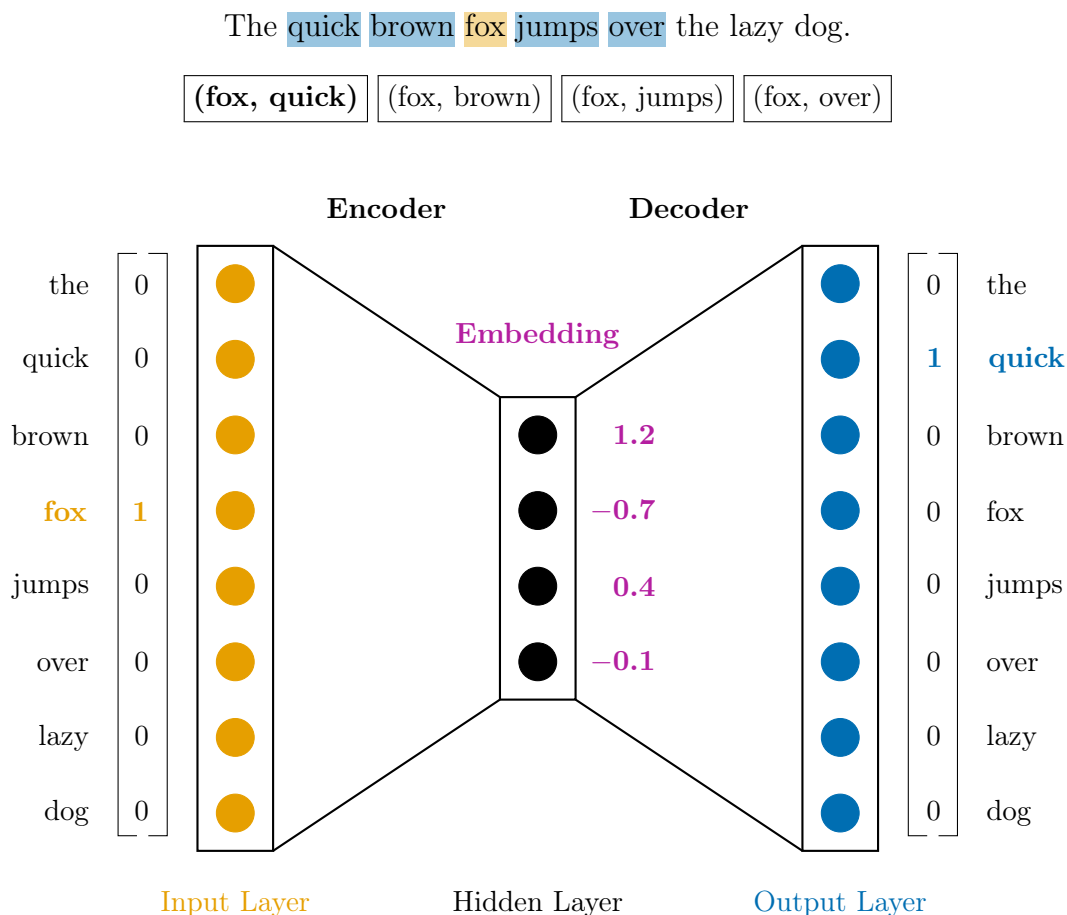


Figure 3.8: Illustration of the architecture of word2vec, which consists of an encoder and decoder similar to an autoencoder, in the variant Skip-gram. The sentence shown above has been transformed into Skip-gram instances, with showing the processing of the first instance (fox, quick) in the model. To represent the words in the input and output, the bag-of-words representation, i.e., one-hot encoding, is used. Example embedding vector for the input word “fox” can be seen next to the hidden layer.

Mikolov et al. presented an efficient technique called *word2vec* [96] to learn word embeddings using neural networks on a large corpus of text. Modern large language models such as GloVe [111], GPT [114] or BERT [31] – to name just a few – are further developments and provide word embeddings trained on large datasets, but have particular modifications specific to natural language properties built-in. In contrast, word2vec is not specialized in natural language and can therefore be easily used and adapted for other application areas – such as assembly modeling – which is why we present this technique in detail in the following.

For word2vec [96], only plain text without annotations or manual labels is needed for training. The result is an embedding vector for each word of the corpus. The basic idea is that the meaning of a word is defined by its context words. So if two words occur in the same context, it is assumed that they have something in common, i.e., that they are similar in some way. Architecturally, the model is structured like an autoencoder

and consists of two processing feed-forward layers, in particular a hidden layer (encoder) followed by a nonlinearity and an output layer (decoder), cf. Figure 3.8. As it is the case for all neural networks, the input layer only represents the input and does not perform any computations. Instead of trying to reproduce the input as in a real autoencoder, successive words are mapped to each other, while cross-entropy serves as loss function. There are two variants of word2vec: for *Skip-gram*, a center word is mapped to its context, while for *CBOW* (Continuous Bag of Words), context words are mapped to a center word, cf. Figure 3.9. Context words are words within a certain range before and after the center word, defined by a hyperparameter `window_size`. Experiments show that the first variant works well with small datasets and can better represent rare words, whereas the latter can be trained faster, but focuses more on good representations for frequent words [96]. Since the two variants of word2vec basically just swap input and target of the instances, but the way the work is the same, we confine our following explanations to only one variant, namely Skip-gram.

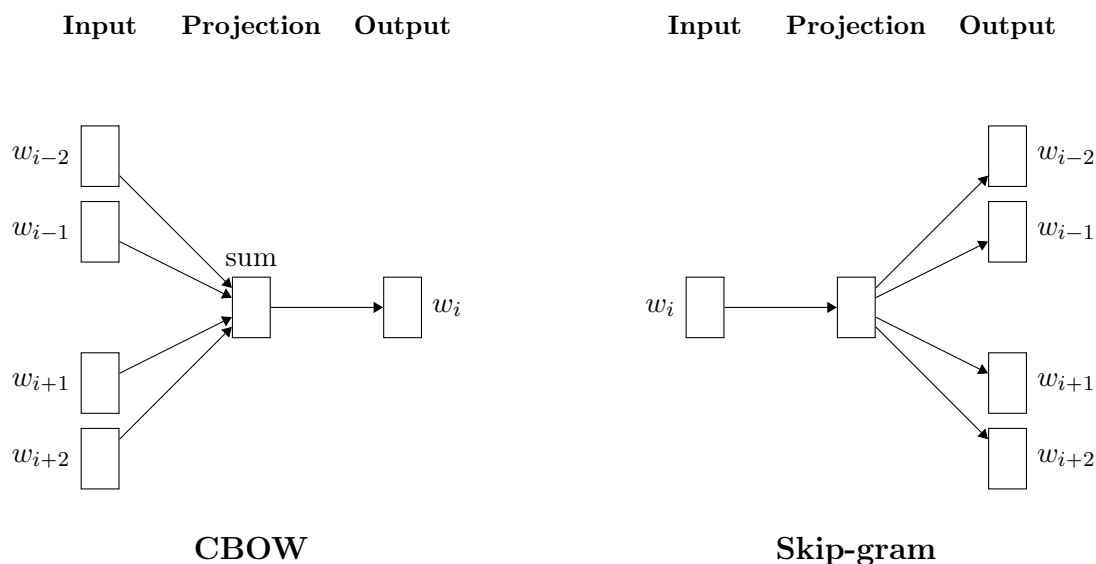


Figure 3.9: The two model architectures of word2vec: CBOW and Skip-gram. The CBOW architecture predicts the center word w_i based on the context words ($w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}$) for `window_size = 2`, while the Skip-gram architecture predicts context words given the center word. The Index indicates the position of a word in the considered sentence. Adoption of [96, Figure 1].

Self-Supervised Instance Generation Word2vec falls into the category of self-supervised methods, as labeled (supervised) instances are extracted from unlabeled plain text. The given training sentences are iterated word by word, creating multiple instances for each current word, one for each pair of center word and context word. In other words, the text is iterated in $2 \cdot \text{window_size} + 1$ consecutive words, while pairs consisting of the middle word and every other within-window neighbor are built, thus skipping the middle word. Therefore, the resulting instances are called *Skip-gram instances*. However, the instances are not created beyond the limits of a sentence: So in most cases, exactly $2 \cdot \text{window_size}$

Source Text	Skip-Gram Instances from Source Text
The quick brown fox jumps over the lazy dog.	(The, quick) (The, brown)
The quick brown fox jumps over the lazy dog.	(quick, The) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog.	(brown, The) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog.	(fox, quick) (fox, brown) (fox, jumps) (fox, over)
The quick brown fox jumps over the lazy dog.	(jumps, brown) (jumps, fox) (jumps, over) (jumps, the)
The quick brown fox jumps over the lazy dog.	(over, fox) (over, jumps) (over, the) (over, lazy)
The quick brown fox jumps over the lazy dog.	(the, jumps) (the, over) (the, lazy) (the, dog)
The quick brown fox jumps over the lazy dog.	(lazy, over) (lazy, the) (lazy, dog)
The quick brown fox jumps over the lazy dog.	(dog, the) (dog, lazy)

Figure 3.10: Example for creation of Skip-gram instances of the form (center word, context word) for a given sentence with `window_size = 2`. Every word serves as center word (highlighted in yellow), the two preceding and two subsequent words form the respective context words, which are highlighted in blue.

instances are generated, but if the center word is too close to the beginning or end of the sentence, a reduced number of instances is extracted. An example is shown in Figure 3.10. Consequently, there are several targets (context words) for one input (center word). For processing in the word2vec model, the words are represented by one-hot vectors – both the input and the target.

The size of the model’s hidden layer defines the resulting embedding dimension as hyperparameter; the dimensions of the input and output are determined by the size of the vocabulary, i.e., the unique words of the text corpus. The hidden layer maps a given word encoded as one-hot to a lower dimensional embedding vector, which is mapped back to the dimension of the words by the decoder layer as visualized in Figure 3.8. The latter one is only used in training and truncated for inference as the embeddings are calculated in the hidden layer. The weight matrix of the encoder serves as lookup table for word embeddings as the i -th row corresponds to the embedding of i -th word in the vocabulary. Such embedding techniques are referred to as *shallow embedding approaches* [59]. The activations of the output layer represent a distribution of scores over the entire vocabulary. After applying the Softmax activation function, these can be interpreted as probabilities over the vocabulary. The highest and therefore most probable value of the prediction indicates the predicted word. Mikolov et al. [97] also present some techniques to increase efficiency and improve computational complexity, such as negative sampling that aims for balancing the number of positive and negative rewarded tokens.

Embedding Evaluation The problem of finding embeddings for words or categorical objects in general is basically an unsupervised learning problem since we do not know the target vectors. However, the evaluation of unsupervised models is very difficult due to the lack of ground truth [104]. Word2vec solves this problem with a kind of trick: Due to the architecture of mapping center words to context words (Skip-gram), a supervised

learning problem is solved during training. The model is based on the assumption that if the supervised learning problem is well solved, the unsupervised learning problem for predicting embedding vectors is also well solved. This means that a supervised learning problem indirectly solves an unsupervised learning problem. Modern embedding methods also solve the unsupervised problem with an auxiliary task, such as predicting the next word or a missing word.

Existing methods for evaluating embeddings can be divided into extrinsic and intrinsic methods. The survey [7] provides an extensive overview of recent advantages and widely-used evaluation approaches; we will only give a short summary here. Extrinsic methods evaluate embeddings on a supervised downstream task, typically the actual target learning task. The idea is that meaningful features should lead to good task performance, perhaps even with only a few instances [104]. These tasks, however, are typically elaborate and slow to compute. A shortcoming of this approach is that it is difficult to identify the reason for poor performance; it could be the representation, the downstream model or their interaction. With intrinsic methods, specific intermediate subtasks are first defined based on domain knowledge and the trained embeddings are then evaluated on them. These subtasks are typically simple and fast to compute – but need to be hand-crafted in the beginning. These include, for example, synonym detection (selecting the synonym for a given word from a set of words), outlier word detection (identifying the outlier from a set of words) or analogy completion. In the latter task, for given three words w_1 , w_2 and w_3 , the matching fourth word is to be provided, which answers the following question: “What is the word that is similar to w_3 in the same sense as w_2 is similar to w_1 ?”.

The analyses of the learned embeddings showed that the word analogies searched for in analogy completion tasks – whether based on semantic or syntactic rules – can simply be represented as arithmetic operations in the vector space. For instance, to find a word that is similar to “swimming” in the same sense as “did” is similar to “doing”, we can compute $\mathbf{x} = \vec{\text{did}} - \vec{\text{doing}} + \vec{\text{swimming}}$. The closest word to point \mathbf{x} measured by cosine distance serves as answer. For well-trained embeddings, this word corresponds to the correct answer “swam”. This means that there is a vector that maps the verb forms from progressive to past, which is very similar to the vector from “doing” to “did”. Analogously, you can find vectors for diminutive, assignment of capital cities to countries or male-to-female transformations, to name just a few examples. Two examples are visualized in Figure 3.11.

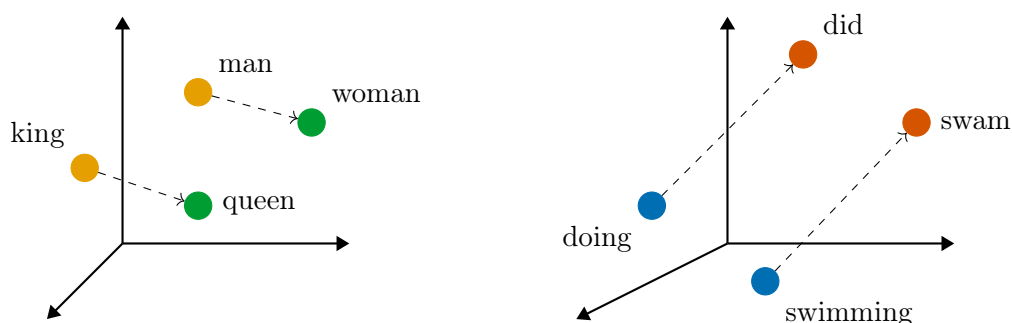


Figure 3.11: Visualization of exemplary semantic (left) and syntactic (right) vectors that can be found in a learned word2vec embedding space. The semantic vector translates male word forms into female word forms whereas the syntactic vector performs a verb tense transformation.

Intuition We would like to briefly provide an intuition why this architecture provides embeddings that contain both semantic and syntactic similarity of words. The goal of this model is to place similar words close to each other in the vector space. Let's imagine we have two sentences, "The quick brown fox jumps over the lazy dog" and a second one where the word "fox" has been replaced by the word "dog". If we mask the word "fox", we as humans realize that not all kinds of words can be placed here, but only those that make sense in the context, i.e., things or animals that can jump over something. Assuming we now have these two very similar sentences and consider the instances that each have "fox" and "dog" as the center word, we would like to map the input for "dog" and "fox" both to the same context words in the output. In order for two different inputs to produce approximately the same output, the activation of the hidden layer must be similar. The activation of the hidden layer corresponds to the embedding, cf. Figure 3.8.

Chapter 4

Assembly Data

Summary This chapter presents the real-world assembly datasets used in the thesis which originate from customer orders in assembly configurators and provides an approach to represent CAD assemblies as undirected graphs over parts. Further, it covers the characteristics of the three datasets, which differ in number of part types and graph characteristics, thus making them suitable for investigating our generic approaches for the three use cases in assembly modeling. We developed an unsupervised pretraining technique named *part2vec* to generate low-dimensional embeddings for part types using a context-based approach as a generalization of word2vec [96] from sequential data to general graph structures. In a preliminary study, suitable sizes of embeddings were examined. The resulting representations of the parts serve as initial features for all use cases examined in this work.

Contents

4.1	Representing CAD Assemblies as Graphs	52
4.2	Assembly Datasets	54
4.3	Unsupervised Pretraining of Part Embeddings	56
4.3.1	Experimental Results	59
4.3.2	Further Uses for Trained Embeddings	60
4.3.3	Related Work	64

Publications The representation of assemblies as graphs has been introduced in [39, 86] and the datasets have been initially published in [39]. Furthermore, the results of the preliminary study on the part type embeddings are published in [39].

The methodology presented in this thesis is based on representing assemblies as undirected graphs over individual parts. However, the task of converting specific CAD models into these graph representations is beyond the scope of this thesis. For the datasets examined in this work, this conversion was carried out by the industrial partner of project KOGNIA. Consequently, we will present a problem formulation for the representation and only basic description for the corresponding conversion in the following.

4.1 Representing CAD Assemblies as Graphs

Due to the heterogeneity of available attributes of parts¹ in part catalogs, our approach uses the only information that is always available in a catalog: a unique part identifier. We assume a set of pre-existing part types² \mathcal{T} which serves as the elementary building blocks upon which a dataset of n assemblies $\{\mathcal{A}_i\}_{i=1}^n$ is built. In conformity with the field of natural language processing, in which sentences are put together from a collection of tokens, we also use the term *vocabulary* to refer to \mathcal{T} . For instance, a part type might refer to a particular kind of hinge or gear. The parts present within an assembly are *instantiations* of these defined part types, comparable to software objects as instantiations of classes in software systems. The type of part p is denoted as $T(p) \in \mathcal{T}$ using a type function T .

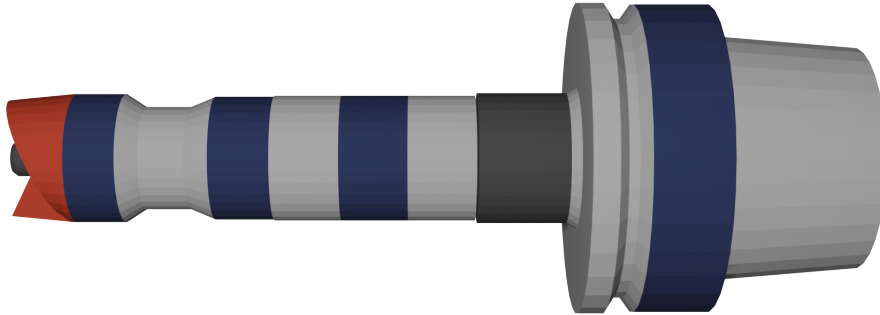
We assume furthermore that the considered assemblies are connected. If a CAD file contains two (spatially) separate assemblies, for example two variants of a machine, these are processed separately. Since the organization of an assembly in an assembly tree depends entirely on the preferences of the designer (see Section 2.2), we must first carry out a preprocessing step to neglect this hierarchy: expanding all subassemblies and summarizing the parts within the leaf nodes into a flat list while keeping defined mates. As an example, the right part of Figure 2.2 shows the resulting structure, a flat list of parts without subassemblies, after flattening the assembly hierarchy on the left.

As a collection of parts that are connected to each other, flattened assemblies can be naturally represented as graphs over parts: An assembly \mathcal{A}_i specifies its containing parts as nodes $\mathcal{V}_{\mathcal{A}_i}$, and information about connected parts as edges $\mathcal{E}_{\mathcal{A}_i}$. The part types of the nodes can be obtained from the type function T . An assembly may contain several instances of the same part type. As the available mating types differ in the CAD systems, we neglect this distinction and only use one type of edge (“connected”). The edges are conceptually symmetrical (part p_1 is connected to part p_2), i.e., it only matters *which* parts are connected, not the edge’s direction. Therefore, all edges are bidirectional and unweighted. Furthermore, there are neither multiple connections between two nodes nor self-connections, resulting in simple graphs (cf. Section 3.2). Edges can result from specified mating conditions in CAD systems or can be read off by geometric proximity, which resembles a physical connection of the parts in the later construction. The nodes, on the other hand, are attributed and thus heterogeneous, as there is exactly one node type (namely the type *part*) but they can take on different values with the part types. The available information about parts varies greatly in the different part catalogs. As we aim for a generic approach that should be suitable for all assembly data, we only extract the information that is always available and omit heterogeneous metadata such as the type of mate, material properties or geometric features of the parts. Companies with a long design history often have designs in various formats from different CAD programs; a uniform representation is essential here. Nevertheless, this information can be incorporated, for example as additional geometric or problem-specific features of the parts. Figure 4.1 displays an example assembly and its corresponding graph representation that is going to be examined throughout this thesis.

¹We use the term “part” over “component” to indicate that we deal with atomic parts, that cannot be decomposed. This would also include a motor, for example, if it is an elementary part in a part model.

²The parts can be purchased or in-house parts.

Assembly Model



Extracted Graph

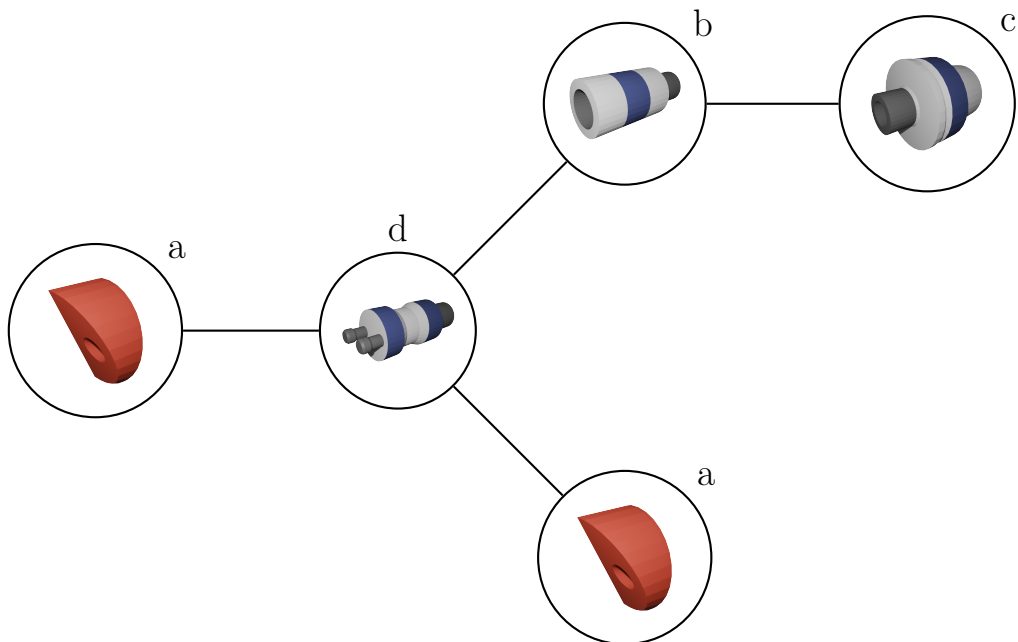


Figure 4.1: Exemplary assembly of an attachment to robots for machining. The assembly consists of five parts of four different part types $\mathcal{T} = \{a, b, c, d\}$, i.e., it contains two part instances of type *a*. The representation of the assembly as an undirected graph is shown below, with the respective part type annotated at the part node. Part *c* is a basic holder that is connected to a boring tool (*d*) via an extension (*b*). Two insert holders (*a*) are further attached to the boring tool. They can be fitted with cutting inserts for milling which are not part of this assembly.

Ambiguous Edges Following the classification given in [169], which differentiates between structural scenarios where graph structure is explicit (such as in molecule generation) and non-structural scenarios where graphs are implicit, e.g., inferred from text or images, the applications proposed herein can be categorized as “semi-structural”. The extraction of the graph structure is an ambiguous matter, as only some mating relations may be given and even only a portion of them can be extracted canonically as some may serve other purposes than denoting meaningful connections, e.g., to support the designer’s workflow. In this case, some edges may need to be extracted from geometric proximity of parts where also some difficulties arise: Parts that should touch each other do not necessarily have zero distance – either the designer did not model it accurately enough or the different CAD systems are based on different thresholds. Sometimes it is difficult to decide whether a gap is intended; for example, a small air gap between two sheets may be intended in order to ensure cooling. Due to these difficulties, the use of the distance between parts as an edge weight is not advisable.

Permutation Invariance and Heterophily The assembly graphs do not contain any temporal information about their creation: Although every assembly has been created by a sequence of part insertion and connection operations, the resulting design only represents its final state without storing its creation history. This means that the graphs are statically attributed. At first glance, the design process might appear to be sequential, but this is not the case: an assembly can be designed in different ways – again depending on the designer’s preference – making no sequence correct or preferable. This *permutation invariance* is an important property of the assembly data, making graph machine learning particularly suitable for application. However, the assembly datasets exhibit a specific property that is not present in most graph-based datasets: *heterophily*. Two parts serving different purposes and thus being dissimilar are likely to be connected in mechanical designs. The connection of identical or similar parts may occur (for example in the case of extensions), but this is the exception rather than the rule. The majority of graph datasets and benchmarks make tactic assumptions of homophily, thus many algorithms are based on the assumption. According to findings of [170], many of these models show poor performance for heterophilous datasets.

4.2 Assembly Datasets

Three independent real-world assembly datasets based on three distinct sets of part types are available for this thesis³. The datasets were provided by project KOGNIA and originate from so-called assembly configurators. There, users can put together assemblies online, which are assembled and sent to the customer after ordering. The resulting assemblies are acyclic, i.e., tree structures, which simplifies some preprocessing steps⁴.

Each dataset is composed of about 12,000 mechanical assemblies. The datasets differ in the size of the vocabulary, while the assemblies differ in the number of parts per assembly and graph diameters, as shown in Table 4.1. Due to their different characteristics, the

³The datasets were initially published in [39] and can be accessed via https://figshare.com/articles/dataset/ECML22_GRAPE_Data/20239767. The assemblies are given as pseudonymized graphs, i.e., the part types were pseudonymized to IDs.

⁴The methodology in this thesis is constantly described for arbitrary undirected graphs; simplifications or modifications arising for trees are explicitly pointed out in the respective passages.

datasets are well suited to investigate the applicability of our approaches. Since the datasets are based on independent vocabularies and thus no synergy effects are to be expected from their combination, we investigate each dataset separately. In the course of the thesis, we will consider different modifications of the assembly data to implement the use case of (localized) part recommendation and anomaly detection. In order to distinguish them notationally, we refer to an existing assembly in our database as \mathcal{A}^* .

Table 4.1: Key facts of the three assembly datasets used. For each metric column, the first line indicates the range of values and the second line the corresponding average value.

Dataset	# Graphs	# Part Types	Node Degrees	# Nodes	# Edges	Graph Diameter
A	11,826	1,930	1 – 9 $\bar{\varnothing}$ 1.7	4 – 33 $\bar{\varnothing}$ 6.12	3 – 32 $\bar{\varnothing}$ 5.1	2 – 32 $\bar{\varnothing}$ 4.45
B	11,895	3,099	1 – 13 $\bar{\varnothing}$ 1.9	4 – 69 $\bar{\varnothing}$ 18.18	3 – 68 $\bar{\varnothing}$ 17.2	2 – 38 $\bar{\varnothing}$ 10.06
C	11,943	1,924	1 – 16 $\bar{\varnothing}$ 1.7	4 – 20 $\bar{\varnothing}$ 6.74	3 – 19 $\bar{\varnothing}$ 5.7	2 – 6 $\bar{\varnothing}$ 2.94

We deliberately use real-world datasets that have not been pre-curated to ensure a balance in graph size, labels, etc. As the assemblies are based on customer orders, the same graphs and subgraphs may occur several times. These duplicates were deliberately not deleted as they reflect the real-world bias. Accordingly, the sizes of the designs are not equally distributed and the parts also occur with different frequencies. The uneven distribution reflects the real-world bias of the problem and should therefore be maintained. This is also common practice in other application domains such as NLP, as there is no equal distribution across the words or letters.

Besides the described information, the datasets do not contain any further information, in particular no labels describing the quality of an assembly. This corresponds to our requirement to develop a procedure that is as automated as possible and ideally requires no manual labeling or maintenance of the data by domain experts. The existing datasets of assemblies have been ordered by customers in exactly this way, so we assume that they are of good (enough) quality and therefore form a suitable knowledge base. Therefore, we want to extract the implicit design knowledge contained therein, analogous to natural language processing as stated in [114]: Even if some assemblies may not be perfect, this is averaged out over the mass of data. Good designs prevail over mediocre ones due to their frequency, so it can be assumed that good design knowledge is learned overall.

CAD Datasets in Literature Recently, interest in data-driven approaches in CAD has increased, leading to the publication of several datasets, for instance the ABC dataset [82] by Koch et al., the AutoMate dataset introduced in [68], the ShapeNet repository [24], the DMU-Net dataset [30], or PartNet [100]. These datasets and applications are centered around the part modeling side of the CAD workflow in contrast to assembly modeling. Although segments of the 3D models (similar to parts of an assembly) are annotated in some datasets, e.g., the seat, backrest and feet of an office chair, the same submodels (parts) are not found in other objects. The designs are therefore not composed of a vocabulary of part types, which is a prerequisite for our approach. However, some approaches share representing designs as graphs with ours, although surfaces and edges become nodes there opposed to parts in our approach.

Preparations for Machine Learning

For all use cases considered, the assembly datasets were split 60:20:20 into training, validation, and test graphs⁵, confer Table 4.2, following a common data split strategy [42]. This division is fixed for all experiments, even for the pretraining of part embedding. When performing the data split, we ensured that all part types occur in the training graphs so that no parts unknown in the training occur in the validation or test data.

Table 4.2: Data split of the three assembly datasets into training, validation and test assemblies.

Dataset	# Part Types	# Training Assemblies	# Validation Assemblies	# Test Assemblies
A	1,930	7,095	2,365	2,366
B	3,099	7,137	2,379	2,379
C	1,924	7,165	2,389	2,389

To process the assembly graphs with neural networks, their graph structure is represented using an adjacency matrix. Strategies from Section 3.4 can be used to represent the different part types of the nodes, for example by one-hot encoding this discrete feature or learning an embedding for it. We thus replace the type function $T : \mathcal{V} \rightarrow \mathcal{T}$ with the feature matrix \mathbf{X} composed of the respective representations of the part types. In some machine learning applications such as NLP, representing discrete objects by continuous vector embeddings instead of one-hot vectors has proven to be advantageous. Since assembly modeling has many parallels to NLP, we investigate whether using part embeddings as node features contributes to the performance of our models.

In addition to considering the relationships between parts in an assembly, it seems natural to include geometric information about the parts in their features as well. In a preliminary investigation [125], we examined algorithms that describe various geometric properties of CAD models in order to find parts similar to a given reference part, including some algorithms used for searching geometrically similar parts in [45]. We evaluated these algorithms on the publicly available DMU-Net dataset [30], which contains models of engineering parts from various categories, such as bearings, gears and wing nuts. Our experiments revealed that across the different part categories, the set of features required to successfully identify similar parts varied significantly. In summary, the overall task of finding similar models across all categories could only be solved with moderate success. As a consequence, we focus exclusively on the relationships within assemblies to extract meaningful features for parts. We believe that this information is more conducive to our use cases, because on the one hand, geometrically similar objects do not necessarily have to serve the same function, and on the other hand, in assembly modeling, parts are selected based on their function within the overall assembly rather than their shape.

4.3 Unsupervised Pretraining of Part Embeddings

Learning word embeddings was a milestone in the field of NLP. Nowadays, there are numerous available embeddings trained by language models such as GloVe [111] or BERT [31]

⁵In this thesis, we use the validation set for finding suitable hyperparameters and comparing different models for the same task in order to find the best one whereas the test set is used to measure the generalization error. There is some ML literature that uses both terms in reverse.

on a huge text corpus, which are available for custom language applications. In contrast to words, there are unfortunately no available embeddings for mechanical parts that we could use and fine-tune for our tasks, which is why we need to create them ourselves.

With regard to our investigated data, we are aiming for a representation for the part types \mathcal{T} , as opposed to the nodes of the graphs. In this way, we want to ensure that the initial representation of identical parts (i.e., parts of the same type) is identical, instead of being assembly and neighboring part specific. With this, we can directly transform the formulation for graphs with attributed nodes $\mathcal{G} = (\mathcal{V}, \mathcal{E}, T : \mathcal{V} \rightarrow \mathcal{T})$ to the machine learning centered representation $\tilde{\mathcal{G}} = (\mathbf{A}, \mathbf{X})$ based on the adjacency matrix \mathbf{A} and feature matrix \mathbf{X} . We want to investigate whether using part embeddings as node features is also beneficial outside the domain of NLP, and in particular whether it can contribute to the performance of our application models.

Similarities of Assemblies to Natural Language Assembly modeling has many parallels to NLP: Just as documents are composed of words, assemblies consist of parts. Moreover, neighboring words (predecessor and successor) correspond to the possibly larger set of adjacent nodes in a graph. Therefore, it is natural to investigate NLP techniques for assemblies. Modern word embedding techniques are based on complex, multi-parameter architectures (mostly Transformers) that require a large amount of training data and often have language-specific modifications, making them unsuitable for the datasets considered in this work. Word2vec (cf. Section 3.4.2), on the other hand, is based on a simpler architecture that can determine embeddings very efficiently for large data sets, but has a comparably small number of parameters and can therefore be trained with significantly fewer data. In addition, this technique allows the text to be processed in its plain form without domain experts having to annotate the sentence’s structure or individual words. This meets our requirement to develop an automated system that requires as little human input as possible. This technique has already been applied successfully outside of NLP, e.g., in recommendation systems [53, 54, 140].

Part2Vec Word2vec has been designed for processing text, i.e., sequential data. Since sequences are basically a special type of graph, we want to generalize this technique to general graph structures. Following the naming scheme, we call this method *part2vec*⁶, as we want to assign a vector to each part type. Since in word2vec the meaning of a word is derived from its context, in part2vec the purpose or function of a part is inferred from its connected parts. The embedding therefore describes similarity of parts in terms of usage. The implicit relationships learned between the objects are also very promising for the use cases under consideration: If a designer is to redesign an assembly in a different size or with different materials in order to adapt it for other conditions, learned part relations such as “aluminum to steel” or “9mm screw to 14mm screw” would be helpful, as counterparts to the semantic and syntactic vectors for words (cf. Figure 3.11).

In experiments, the Skip-gram architecture (cf. Figure 3.9) has been shown to be suitable for representing rarely occurring tokens [96], so we use this variant for part2vec. For text, the architecture trains a mapping from words to their context words within a defined context window. The equivalent to n -distant words of a center word are n -hop distant

⁶In the original publication [39], we introduced the model as “comp2vec” as abbreviation for component. Since we restricted ourselves to the term “part” in this thesis, we adjusted the model name for consistency reasons.

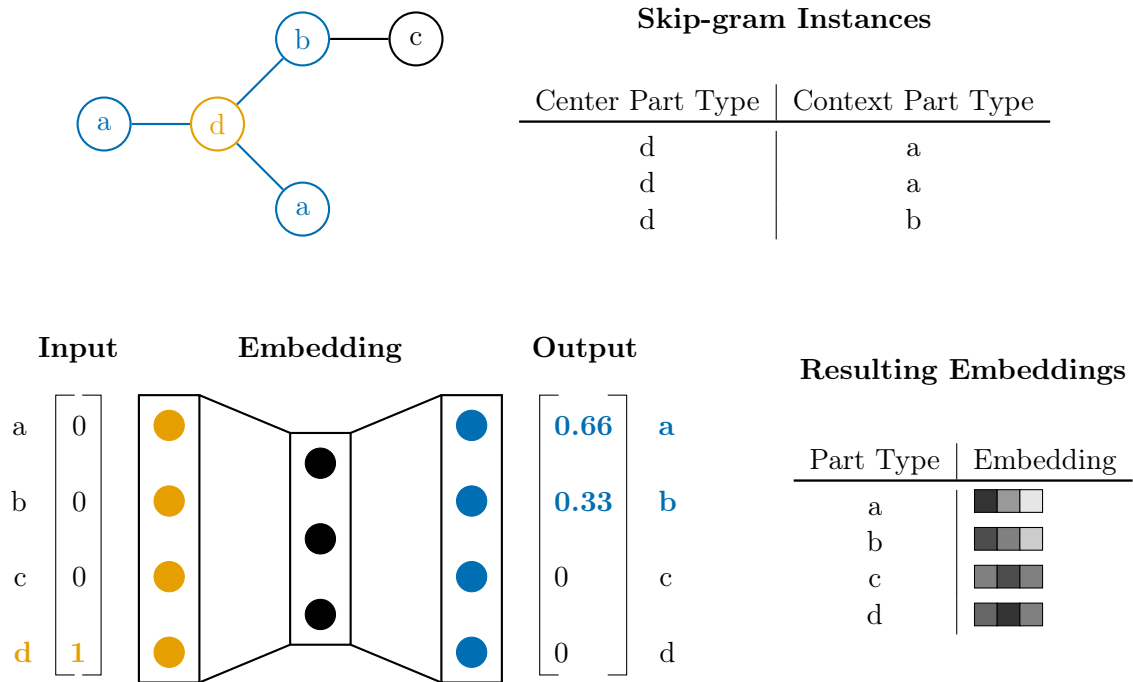


Figure 4.2: Pretraining of part embeddings: For a given assembly graph (top left), Skip-gram instances are created with `window_size = 1` which map center part types to context part types. The table on the top right displays the resulting instances with part type `d` as center node. Training with the Skip-gram instances results in embeddings per part type which are displayed as grayscale vectors on the bottom right. Since part types `a` and `b`, as well as `c` and `d`, are connected to a similar set of parts, respectively, their resulting embeddings are similar.

nodes of a center node in a graph. For a graph, this approach may result in more generated instances. The upper part of Figure 4.2 shows how Skip-gram instances are obtained from an example graph. Despite adapting the instance extraction, the training process itself remains the same as in `word2vec`. In particular, we also use cross-entropy as a loss function here. After training, the embeddings can be obtained from the weight matrix of the first layer as each row corresponds to the embedding of a part type in the vocabulary.

Because representation learning is a subdomain of unsupervised learning, the difficulty is that the model cannot be evaluated directly due to the lack of labels. This is why intrinsic (using specific intermediate subtasks) or extrinsic (on the real task) evaluation is proposed for word embeddings in the literature [7]. The former is, in principle, transferable to assembly modeling, but requires high manual effort from design experts and consequently contradicts our goal of automatic learning. Extrinsic methods, on the other hand, are typically elaborate and slow to compute, therefore also undesirable. Fortunately, the embeddings are learned by solving a supervised task (predicting context tokens from center tokens), which allows to directly evaluate by loss. The model is based on the assumption that the unsupervised task is well solved if the corresponding supervised task is well solved, which is why we optimize for loss. As a consequence, different embedding sizes can be compared, since the output dimension is defined by the task (vocabulary size) and

thus does not depend on hyperparameters.

4.3.1 Experimental Results

In a preliminary study, we investigated suitable embedding sizes for the three assembly datasets. To do this, we used the assemblies divided into training, validation and test sets, united the first two to form a new training set for creating embeddings and used the remaining set as validation set. Since this model is used for data representation, we do not need a test set to measure the generalization error. The new assembly sets were separately transformed into sets of Skip-gram instances with `window_size = 2` as described above. The resulting number of training and validation instances is shown in Table 4.3. Since all part types occur in the training assemblies, we can create representations for all of them so that we can present vectors for all parts in the validation or test assemblies in later downstream tasks.

Table 4.3: Number of training and validation Skip-gram instances for part2vec of all three assembly datasets.

Dataset	# Part Types	Size Training Set	Size Validation Set
A	1,930	147,598	36,932
B	3,099	793,250	199,552
C	1,924	185,244	46,630

To rate the individual models (i.e., the trained embeddings), we computed the sum of training loss and gap between training and validation loss, following the description given in [51, p. 418]. In comparison, different embedding sizes behaved the same for each dataset: Sizes between 20 and 90 as well as from 100 upwards each led to a similar error level, respectively. Consequently, we chose the minimum per range, i.e., 20 and 100 respectively, as value for the embedding size hyperparameter for each dataset. The error value of the 100-dimensional embedding was consistently below the 20-dimensional one, consequently it should provide a better representation of the part types.

We identified these two embedding sizes as promising for all datasets when optimizing according to the loss function of the supervised task (mapping center part types to context part types). The selection of the embedding dimension was based on the assumption that good part type embeddings can be found by optimizing the supervised task. To test this assumption, we employ extrinsic evaluation as discussed above by using both embeddings to represent the part types in our assembly datasets and measuring the achieved performance on an actual task. This is done in the experiments of the first use case of this thesis, namely global part recommendation (cf. Section 5.4). We expect that the recommendation model can achieve better results with the 100-dimensional embedding as it was superior on the supervised embedding task.

Embedding Visualization and Insights

High-dimensional spaces are hard to visualize, so they are typically transformed into two dimensions, using suitable dimension reduction techniques such as t-SNE [139] or UMAP [95] that try to preserve the relations in the high-dimensional space in the low-dimensional space as well. Figure 4.3 displays a section of the trained part embedding for dataset A

transformed to 2D using t-SNE. By reducing many dimensions to two, some information is obviously lost, but we can still recognize some relationships, especially that similar parts, i.e., parts from the same category, are grouped close to each other. Clusters are clearly visible in the entire projection space. Some parts categories have several clusters, such as the blue category for cutting inserts. As cutting insert come in different sizes and thicknesses to suit the material being processed, it is natural that several clusters arise. Note that the clusters are typically very dense, even if they partially overlap with other clusters of other component categories due to the 2D projection.

4.3.2 Further Uses for Trained Embeddings

The trained embeddings are used as an initial representation of the parts for all considered use cases, but can also serve other purposes: the learned relationships between the parts can be made available to the designers through part clustering or suitable visualization, e.g., Figure 4.4. The figure shows the nearest parts in the embedding space for two selected parts, where on the left side the learning of similar parts was very successful, as the nearest parts are also basic holders. In the right part, however, the next neighbors consist of more different parts, ranging from extensions to serrated tools.

This can reveal implicit relationships in the assembly data that were previously unknown and give designers an insight into black box neural networks. If the relationships shown are plausible, the designers' confidence in the assistance systems can be strengthened. In addition, the learned embeddings can become the basis for the company's part management. The availability of well-curated, hierarchical taxonomies of part types depends on the level of maturity of a company and traditionally requires significant manual effort which is why maintaining a knowledge basis is often neglected. A data-driven solution that exploits usage patterns in assembly models could (pre-)organize a company's frequently used part types. Due to the preparatory work of the embedding model, less manual effort of design experts is required.

Embedding Refinement by User Interaction

A high quality of embeddings is desirable: The better the representation of the parts, the better the performance of the downstream models can be expected. The investigations of the learned embeddings show that valuable relationships between the parts could be extracted, but that there is still room for improvement in some areas (cf. Figure 4.4). It would be helpful to have more assembly data, so that in particular the parts that have rarely been used so far would occur more often, and thus a better representation could be learned from more examples. Unfortunately, we are limited to the designs available. If designers examine visualizations of the embeddings and discover incorrectly learned correlations, these can be adjusted using manual feedback. Ideally, the model itself should recognize for which examples it is quite uncertain and report this, which is what the research area of active learning is concerned with. During the training, the model can interactively query a teacher (human user or other information sources) to label new data points with the desired outputs. Since reviewing the embedding is very time-consuming for numerous part types, this is not a realistic strategy in practice.

Instead, we want to identify a subset of parts whose representation should be improved through user interaction. To do this, we use three methods to identify potentially problematic parts: First, embeddings are usually inaccurate for rarely occurring parts, as the

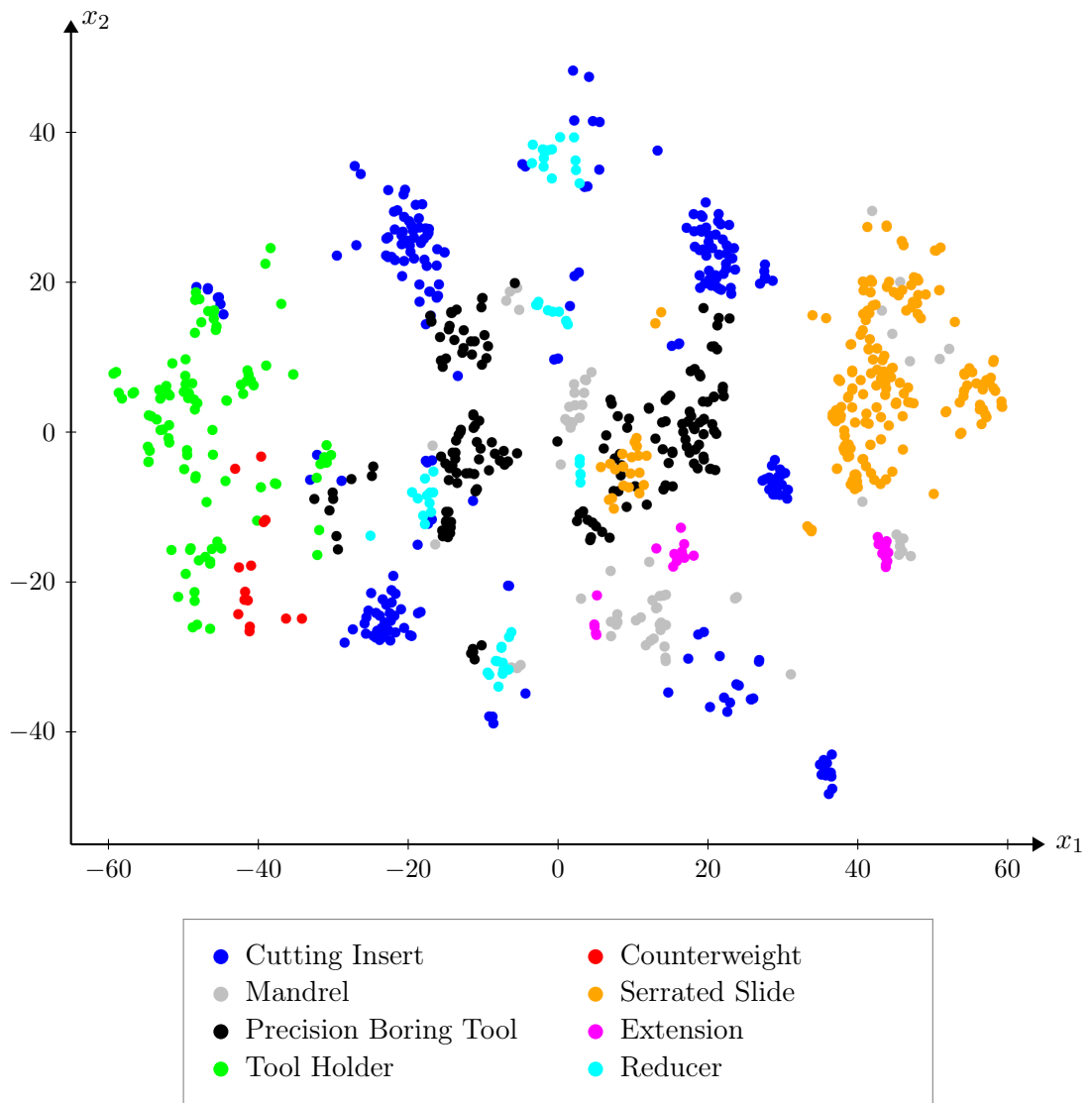


Figure 4.3: Inspection of the 2D projection of a section of the learned part2vec embedding for dataset A. Dots of the same color represent different part types of the same category, such as tool holders.



Figure 4.4: Inspection of eight nearest parts to given part (shown centered in first row) in the embedding space – good and bad example for learning similar representations of similar parts shown on the left and right, respectively.

model had few different examples to infer the purpose or function of the part from its combination with other parts. Rarely occurring parts can be easily identified during data analysis, which is typically performed before using machine learning anyway.

Moreover, following the idea of active learning, we want to identify those parts that the model is uncertain about. The uncertainty of a model can be measured for individual instances by examining the probability distribution in the output: The measure *perplexity* is based on entropy and therefore measures the disorder in a distribution. If many parts are predicted with a low probability, i.e., the distribution is similar to a uniform distribution the perplexity is very high. If, on the other hand, only a few parts are assigned positive probability values, the perplexity takes on a low value. Consequently, input parts that lead to high perplexity are potentially problematic.

Lastly, in companies with a large collection of parts, it can be assumed that there are several parts that fulfill the same or a similar purpose. This means that there should always be several parts nearby in the embedding space. If we now divide the parts in the embedding space into clusters and some individual parts remain isolated, these are also candidates for problematic parts. We employed HDBSCAN (hierarchical density-based spatial clustering of applications with noise) [21], a cluster algorithm that outputs outliers found in addition to the clusters formed.

For all potentially problematic parts found in prior analysis, their representation is to be improved through user feedback if necessary. The fundamental property of the embedding space is to place similar objects close to each other and thus place dissimilar objects far away. Our goal is to adjust the positions of the potentially problematic parts to reinforce this property.

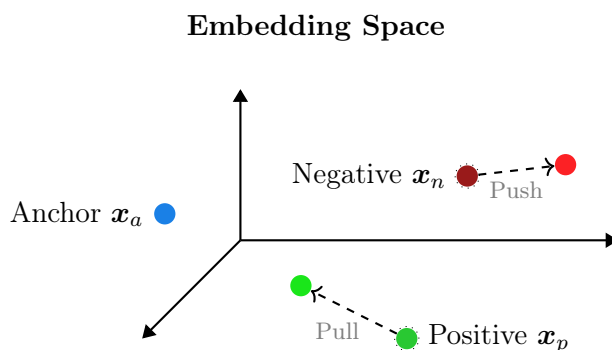


Figure 4.5: Visual impression of the effect of triplet loss for one given triplet: increasing distance between anchor and negative, i.e., push negative away from anchor, while decreasing distance between anchor and positive, i.e., pull positive towards anchor. Based on [76, Figure 2].

In order to influence the embedding vectors during training, either additional training instances can be created or *constraints* can be included as additional optimization criteria in the loss function. The so-called *triplet loss*⁷ compares a reference input \mathbf{x}_a called anchor to a similar input \mathbf{x}_p (called positive) and a dissimilar input \mathbf{x}_n (called negative). By optimizing the following objective, it enforces the embedding model f to reduce the distance between similar objects while increasing the distance for dissimilar objects, cf. Figure 4.5:

$$\min (|f(\mathbf{x}_a) - f(\mathbf{x}_p)| - |f(\mathbf{x}_a) - f(\mathbf{x}_n)|)$$

The distance metric $|\cdot|$ could be Euclidean distance, for example.

We developed a GUI application called *Part Embedding Refiner*, in which a designer can define *similarity constraints* on parts that are transformed into triplets, see Figure 4.6: All parts identified as potentially problematic are listed on the left, the positive and negative parts are to be selected in the right-hand section. For this purpose, current neighbors of the considered anchor are displayed, which can be marked as positive (similar) or negative (dissimilar) by dragging it into the corresponding list. If no decision is made regarding a part’s similarity to the anchor, it can be left in the original list, thus marked as neutral, and excluded from the triplet analysis. Optionally, parts can be searched for in order to mark them as positive or negative. The triplets created from the defined similarity constraints in the application are taken into account when training the embedding. When adjusting (i.e., fine-tuning) the learned embeddings for only a small set of parts, belatedly moving individual parts in the embedding space could lead to the violation of globally prevailing rules (e.g., semantic or syntactic vectors). Therefore, we do not simply perform fine-tuning with the triplets on an earlier trained embedding, but extend the loss function by a term with the triplet loss for training. Therewith, the model simultaneously attempts to fulfill the original supervised task (map center parts to context parts) as well as the newly introduced triplet constraints. A weighting factor (hyperparameter) determines the influence of the constraints on the model.

⁷The triplet-loss was originally proposed in [126], whereas [127] introduced an earlier equivalent formulation over ten years before.

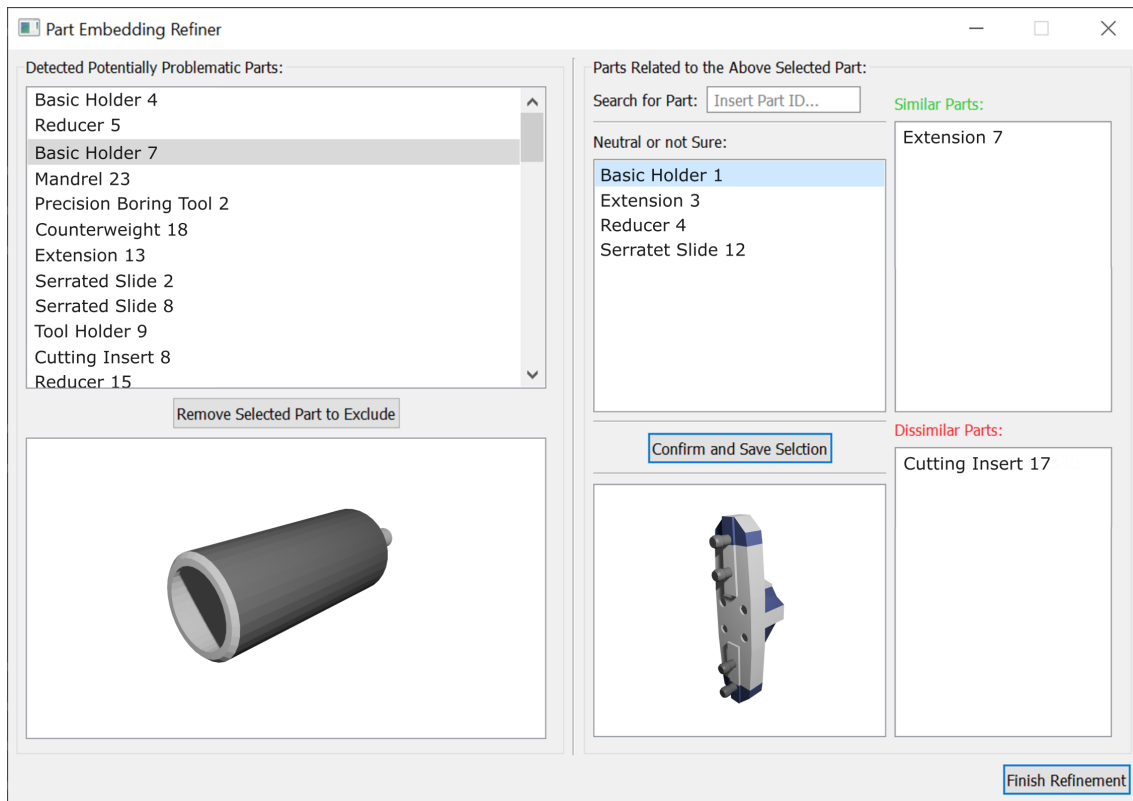


Figure 4.6: User interface for refining a learned part embedding: Found problematic parts are displayed on the left. Current neighbors in the embedding space are listed on the right in order for a user to assign them into three categories: (i) similar, (ii) dissimilar or (iii) neutral or not sure. Categorized parts are transformed into triplet instances, on which the embedding is fine-tuned.

The need for improvements in the embeddings of rarely occurring parts was also noted in the course of a user study on global part recommendation that was carried out as part of the research project KOGNIA (see Section 5.4). Due to the end of the project, it was unfortunately no longer possible to conduct a user study of the embedding refiner with design experts.

4.3.3 Related Work

A central criterion for the selection of a technique to create part type embeddings in this thesis was the identified similarity of the assemblies to natural language. A similar comparison between CAD and natural language has already been made by Ganin et al. in their paper “Computer-Aided Design as Language” [40]. They have created a tool to support part modeling, in particular the creation of two-dimensional sketches, which provide the basis for 3D part and employ NLP techniques to suggest work steps to create a 2D sketch.

Embedding methods for graphs are fundamentally differentiated with regard to the component to be embedded: nodes, edges, subgraphs and entire graphs. An overview of existing methods is provided in [20], and in the following, we focus on related work on the category which matches our problem setting: Since the nodes in our graphs represent the

parts (of a particular part type) of the assemblies, our desired representation for part types is most similar to representations for nodes.

In particular, our goal is to get a representation for the part types so that each instantiation of this type gets the same feature vector. However, node embedding techniques aim to map each *node* to an embedding, thus often including the node’s position in the graph and structural properties [59]. Therefore, two nodes corresponding to two parts of the same type could have different representations. Our focus is primarily on the function (purpose) of the part type, incorporating the structural assembly information and interaction with the connected parts is achieved by processing with message passing in GNNs. These techniques aim to embed nodes that are “close” to each other to similar vector representations. The definition of closeness between two nodes is majorly differentiated between first-order (edge weight) and second-order proximity (similarity of neighborhood); for our setting, the second variant is more appealing (as we do not have edge features).

Classical node embedding techniques focus on matrix factorization techniques, i.e., decomposing a matrix into the product of multiple smaller matrices, typically applied to graph adjacency matrices. The goal is to capture the relationships between the nodes in the graph in the lower-dimensional representation. However, many matrix factorization techniques suffer from high computational costs due to their mathematical nature which is challenging for large graphs. Moreover, they were found to capture more simple and static relationships between the nodes compared to learning-based methods [59].

A notable subset of node embedding techniques are based on random walks: they extract node sequences from graphs to process them with embedding methods for sequences successfully applied in the NLP domain, such as word2vec. It was found that a naive evaluation of the optimization function can be computationally intensive for a large vocabulary, so DeepWalk [112] uses hierarchical Softmax for approximation and node2vec [55] uses a noise contrastive approach based on negative sampling. The key difference to our proposed approach is that the generalization of word2vec to graphs is solved differently: While DeepWalk and node2vec sample individual paths from the graphs, which are then used as sentences for instance generation, we have generalized the definition of Skip-gram instances from sequences to graphs (n -hop neighbors rather than n -distant words). Because our approach does not involve sampling, it should produce more accurate embeddings, but may be too computationally intensive for larger graphs. For our small graph datasets, this is not an issue.

Finally, one could even use a GNN (or other machine learning methods applicable to graphs such as Graph Transformers [98]) and extract internal node representations from it: When a GNN processes a graph, the node representations get updated with information about their neighboring nodes. These representations (e.g., taken from the last hidden layer) can also be used as embeddings; however then these are graph-specific node embeddings instead of part type embeddings. This means that even two parts of the same type within a graph typically have different vectors.

Part Recommendation in Assembly Modeling

Summary This chapter presents a methodology for generating part recommendations in assembly modeling, based on analyzing past assemblies to predict the next needed part for the whole assembly. For the representation of the parts in the assemblies, we use the part embeddings previously trained with part2vec. We have developed an algorithm for recursively decomposing assembly graphs, which forms the basis for generating instances for part recommendation in a self-supervised manner. The proposed recommendation system supports designers by suggesting relevant parts, reducing the need for manual searches in part catalogs. The chapter includes experimental results and discusses the performance of different GNN architectures, including the impact of pretrained embeddings and effects of the properties of the real-world datasets. The GNN models achieve significantly better results than simple recommendation strategies and a classical recommendation model. When recommending ten parts, the expected part was included in between 82.1% and 92.8% of cases. The results thus demonstrate that our system can successfully suggest a designer’s next required parts and thus significantly reduce the search time.

Contents

5.1	Generating Recommendation Instances from Assemblies	70
5.2	Baselines and Upper Bound	75
5.3	Experimental Setup	76
5.4	Experimental Results	78
5.5	Digressions	88
5.5.1	Pretraining of Part Recommendation Models	88
5.5.2	Graph Transformers for Part Recommendation	89
5.5.3	Context-Specific Number of Recommendations	90
5.6	Related Work	92

Publications The algorithm for generating recommendation instances as well as the methodology of the GNN-based recommendation system is published in [39]. This also includes the baseline models and upper bound as well as their experimental comparison with recommendation models based on GCNs and GATs.

In assembly modeling, design engineers can choose from a variety of existing part types stemming from various catalogs of purchased, standard or in-house parts. Consequently, selecting the right part types is a cumbersome task as they have to pick from many possibilities. Past assemblies contain information on both the collection of used parts and their combination to solve a specific task. A governing assumption of our approach is that parts that are *used together frequently* have a causal relationship that is captured in the data. Consider, as a simplified example, a cabinet that consists of five plates, a hinge, a handle, screws, etc. [39]. A heavy hinge might often be combined with a heavy door and similarly for lighter parts, whereas combinations of a heavy door with fragile hinges are unlikely. We propose to analyze a dataset of past assemblies composed of parts from a given set of part types in order to suggest the next needed part. That way, experienced designers' implicit knowledge could be extracted from existing assemblies and speed up the design process. This can foster the transfer of knowledge between designers, so that inexperienced designers in particular benefit and their learning phase is shortened.

In this problem setting, we assume that designers in principle already know which parts they want to connect and in which way to form the desired assembly. Hence, we aim for a recommendation system that supports them in supplying required parts so that they do not have to search for them manually in the available part catalogs. Our system addresses the prediction of part types, leaving the appropriate placement of the part to the designer.

For the part recommendation, there are some requirements that arise either from the use case itself or from the general requirements of this thesis:

1. First, we assume that the designer continues to build incrementally on the assembly, i.e., always builds a *connected* assembly. We exclude scenarios in which two separate components are designed in parallel, which are joined after a certain time. Therefore, the part recommendations should support the designer *directly*, i.e., the parts should be directly attachable to the current assembly as opposed to needing some parts in between in order to connect the current assembly with the recommended part type.
2. In line with our goal of an automated approach, we strive for a data-driven solution that adapts to the sets of assemblies on which it is trained, rather than having design engineers manually maintain domain-specific rules for building assemblies.
3. Furthermore, we aim for an *interactive recommendation system* that provides *single-step* auto-complete-like suggestions rather than automatically completing or generating a design of an assembly from scratch. The design engineers should be in full control of the design process at all times, being free to choose whether to select from the suggestions or search the catalogs themselves. In terms of (graph) machine learning, models generating a graph incrementally (e.g., node-by-node) as opposed to all-at-once are called *autoregressive* [59]. They operate under the premise that past states have an effect on current states and consequently predict future values based on past values. This holds for part recommendation.

We decided to create a recommendation system with a *fixed number of suggestions* ordered by relevance, even if that entails imperfect recommendations in some cases. Indeed, we also investigated a context-specific number of recommendations also ordered by relevance (cf. Section 5.5.3). For very specific assemblies, there may only be a few suitable parts, i.e., fewer than the number of recommendations, so that the recommendations are filled with other parts. Since we are aiming for a recommendation system to support

designers instead of automatic assembly generation, this is acceptable. In addition, there may also be numerous suitable parts, so that only some of them are suggested initially. Recommendations listed at a later position can also correspond to the part currently being searched for if this combination has rarely occurred in the existing designs. Therefore, further suggestions may be requested later. The task of part recommendation is inherently ambiguous as it depends on the designer’s design intent, who may be even connecting two parts for the first time when designing a new innovative product. However, our focus is on extracting and generalizing existing relationships between parts instead of inventing creative combinations of parts. In this way, design knowledge can be transferred to similar assemblies instead of simply reconstructing them.

We would like to emphasize that the type of recommendation in this use case differs from standard recommendation systems like collaborative or content-based filtering [1]. A good part recommendation does not depend on subjective preferences of the individual designer – in the sense of “liking” of specific parts – but on the intended design. Although the design sequence depends on the designer’s preferences, this does not apply to the selection of the part types themselves. We assume that the graph structure is relevant for recognizing the intention. Depending on *which* parts were connected, different designs can emerge.

In summary, the recommendation model should suggest a fixed number of k part types $\tau_1, \dots, \tau_k \in \mathcal{T}$ for a given assembly \mathcal{A} that can be added directly to it. If the designer selects such a suggestion τ , they can add the instantiated part and connect it to the current assembly. This is described by the constructive operator $\mathcal{A} \triangleleft \tau$. The assemblies are represented as undirected graphs over parts and exhibit symmetry properties that makes them suitable for graph machine learning, cf. Chapter 4. We formulate the part recommendation task as a *graph classification problem* where each class corresponds to a part type $\tau \in \mathcal{T}$ and the recommendation model is trained to predict the next required part type, see Figure 5.1. Formally, we aim at learning a discriminative model $P(\mathcal{T} | \mathcal{A})$, where \mathcal{A} refers to an assembly that is to be extended. This model is part of an autoregressive model which, if unrolled, would lead to a generative process [59]. By using `Softmax` as activation function in a final fully connected layer, we get normalized scores over all part types that can be interpreted as a ranking of the recommendations. To evaluate the models, the top- k rate is used as a performance measure, referring to the percentage of the target part type being in the top- k predictions of a model. A comparison of the performance values for different values of k helps to select the right number of recommendations. In the context of the project KOGNIA, we were especially interested in $k = 10$, as this number of recommendations can be well integrated into a CAD system and offers a wide choice of parts for designers.

The pretrained part embedding is used to represent the part types in the input. Intuitively, one may be inclined to reuse the same representation of parts in the output, i.e., to predict part embeddings instead of a one-hot encoding. Although it would significantly reduce the model’s output dimension, this modeling also has severe disadvantages and was therefore discarded:

- a) For part recommendations, the predicted embedding must always be assigned to a part. However, it is very likely that no part maps to the predicted embedding exactly. In addition, it may even happen that there is no embedded part type in the proximity of the prediction in the embedding space because none satisfies the desired properties. It is unclear which part should be taken in this case.

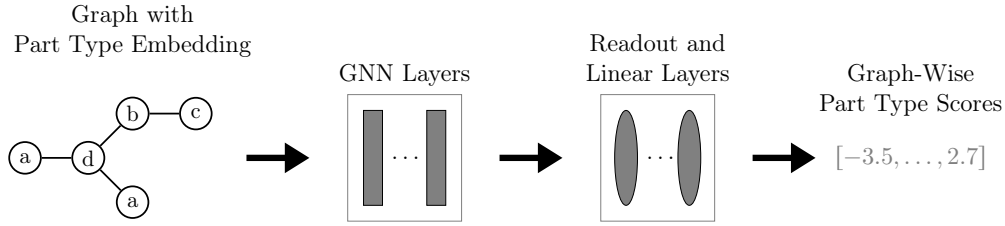


Figure 5.1: Model architecture for part recommendation: The task of part recommendation is modeled as a graph classification task. The given assembly graph is processed with message passing in GNN layers followed by a readout and linear classification layer. The classes correspond to all part types of the vocabulary \mathcal{T} . The resulting scores per part type are displayed as a list in gray. For training, these part type scores are converted into probabilities using a Softmax function. The top k part types with the highest scores are selected as the recommended part types. In this specific example, the last type would be placed before the first in the ranking due to its higher score. Regularization techniques applied in this process are not illustrated.

- b) The number of desired recommendations determines the architecture of the model, i.e., for k recommendations the model would be constituted of k times the embedding dimension output neurons. So, if more recommendations are desired, the model must be re-built and re-trained, or at least fine-tuned. In our modeling, we can simply pick the desired number of most likely elements from the distribution over all part types.

Therefore, the recommendation model maps assemblies to part types (classification) instead of part embeddings (regression).

Since we use pretrained embeddings to represent the parts, the question arises as to why we need another model for processing the partial assemblies. The part2vec embeddings are so-called *shallow embeddings*, as their representation is pre-calculated and can therefore be easily looked up [59]. That is because the vectors were individually optimized in advance using the Skip-gram instances. In contrast, models taking into account the node features and local graph structure, e.g., the neighboring nodes, go beyond the shallow approach. GNNs following the message passing paradigm are in fact such models. Since we assume that not only the individual parts but also their interaction is relevant for part recommendation, we process the graph context with GNNs.

As part of the research project KOGNIA, we developed a prototypical recommendation system as a demonstrator in which a model for global part recommendation was integrated. Figure 5.2 shows the demonstrator to illustrate how we envisioned the part recommendation during assembly modeling in a CAD system. Suitable part types for the current assembly are displayed on the right side, which are sorted according to the ranking predicted by the recommendation model.

5.1 Generating Recommendation Instances from Assemblies

In order to train an ML model for part recommendation, we need supervised instances consisting of “unfinished” assemblies, hereafter referred to as *partial assembly* or *partial graph*, and directly attachable parts. However, we are only equipped with complete assemblies purchased by customers, so we have to artificially generate the intermediate states during

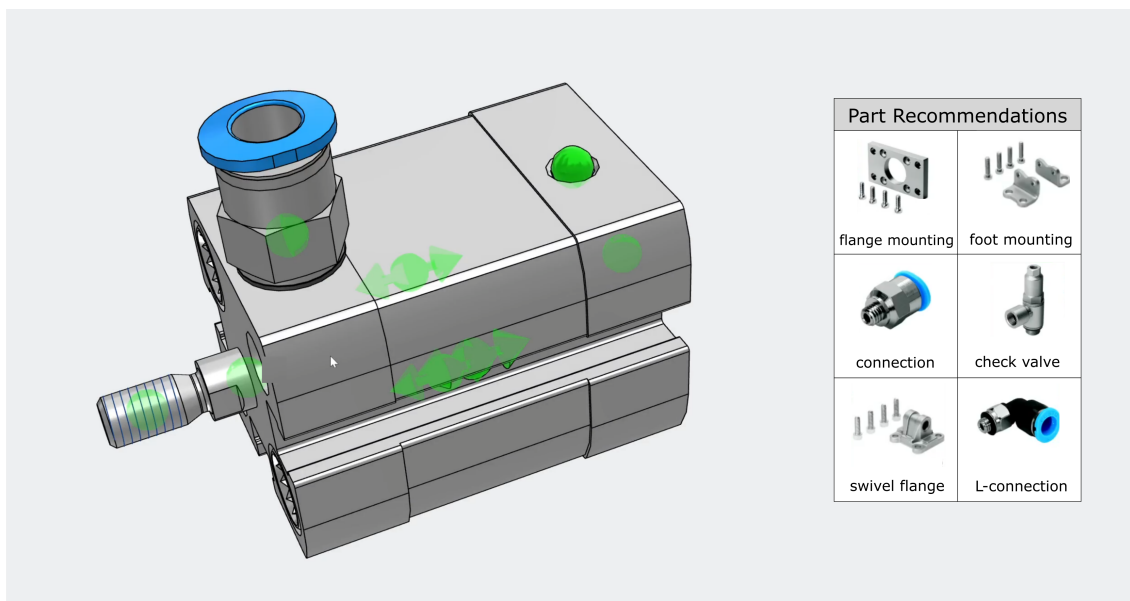


Figure 5.2: Demonstrator for global part recommendation developed as part of the project KOGNIA: Customers can combine pneumatic cylinders with accessory parts. Displayed on the right side are suitable parts for the current assembly which are sorted according to the ranking predicted by our recommendation model. (The part designations were translated into English for this figure.)

the design process. As already discussed, we do not know the original creation sequence that led to the final design and different creation sequences can lead to the same assembly. This means that even if we knew the original sequence, it would still be beneficial to consider other alternatives, as another designer may have used a different approach. Of course, in practice, not every creation history is equally likely: alternate adding parts on opposite sides of a design may be less common. In particular, when a section of the assembly is finished, it is quite conceivable that the designer will switch to another side to continue. As we follow an incremental approach, we only consider the addition of a single part in a single time step. Consequently, we do not make any prior assumptions about the design history and therefore consider all possible ones, taking into account that the assembly is always connected.

We recursively decompose the assembly graphs by means of “cutting off” nodes, resulting in pairs $\langle \mathcal{A}, \tau \in \mathcal{T} \rangle$ consisting of the remaining partial assembly and the part type of the cut-off node, as described by Algorithm 1. As the targets are extracted using the assemblies themselves rather than needing external labels provided by humans, this data generation follows the paradigm of self-supervision, cf. Section 3.1. Starting from an assembly graph \mathcal{A}^* in our database, we iteratively cut off non-cohesive nodes until the remaining graph contains a minimum number of nodes `min_size`. The definitions of node removal and cohesiveness have been provided in Section 3.2. The instances resulting from the decomposition are stored in the (initially empty) instance set \mathcal{D} , whereby all duplicates are eliminated. Finally, in order to process the recommendation instances with machine learning models, part embeddings are used as node features of the graphs and the target part type is replaced by its one-hot encoding.

Algorithm 1 Decomposition of Assemblies into Recommendation Instances

```

1: procedure DECOMPOSEGRAPH( $\mathcal{A} = (\mathcal{V}, \mathcal{E}, T)$ ,  $\mathcal{D}$ )
2:    $seenGraphs \leftarrow \{\mathcal{A}\}$ 
3:   if  $|\mathcal{V}| > \text{min\_size}$  then ▷ domain-specific hyperparameter
4:     for every non-cohesive part  $p \in \mathcal{V}$  do
5:        $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle \mathcal{A} \setminus \{p\}, T(p) \rangle\}$ 
6:       if  $\mathcal{A} \setminus \{p\} \notin seenGraphs$  then ▷ isomorphism test
7:         DECOMPOSEGRAPH( $\mathcal{A} \setminus \{p\}$ ,  $\mathcal{D}$ )
8:          $seenGraphs \leftarrow seenGraphs \cup \{\mathcal{A} \setminus \{p\}\}$ 
9:   return  $\mathcal{D}$ 

```

By tracking in *seenGraphs* which partial assemblies have already been decomposed, we ensure not to change the bias of the data: Let us imagine that we remove two parts p_1 and p_2 from an assembly; in one case p_1 first, in the other case p_2 first. The two subgraphs resulting after both removals are isomorphic. If we would further decompose them, the resulting decomposed instances would occur twice as often. If we consider the removal permutations of three parts, the factor increases to six. However, we only want to examine a certain intermediate assembly once, regardless of how we arrived at this state, which is why we also use permutation invariant models for processing. Figure 5.3 illustrates the decomposition procedure for our example assembly (Figure 4.1) up to a minimum size of two nodes. Specifically, it displays the situation of duplicate recommendation instances being discarded as well as already seen partial assemblies not being further decomposed. Figure 5.4 shows the resulting instances for the given assembly after removing duplicates. For the comparison of the reduced graph with the set of all seen graphs in line 6 of Algorithm 1, we do not always have to perform a complete isomorphism test as two assembly graphs having a different number of nodes or a different multiset of part types can never be isomorphic.

For an entire assembly dataset $\{\mathcal{A}_i^*\}_{i=1}^n$ consisting of n assemblies, their corresponding instance sets \mathcal{D}_i are unified into a *multiset* of instances, $\mathcal{D} = \uplus_{i=1}^n \mathcal{D}_i$, thus allowing for duplicates. The reason for the multiset is again the bias of the real-world: If a certain partial assembly occurs multiple times across all assemblies, it appears to be a common subassembly and should therefore occur more frequently in the total instances of the entire dataset.

The described procedure for generating recommendation instances follows a top-down approach in which nodes are successively cut off. Since we only decompose each subgraph once for an assembly \mathcal{A}^* (controlled by *seenGraphs*), it results in the same instances as a bottom-up approach, in which a partial assembly is iteratively extended by adding nodes to result in a recorded assembly \mathcal{A}^* .

Regarding complexity, the decomposition produces $\mathcal{O}(|\mathcal{V}|!)$ instances – depending on the graph structure – which is prohibitive for large assemblies. The designs we considered in our experiments (composed of up to 70 parts, cf. Table 4.1) were highly sequential, i.e., each individual (partial) assembly contained sufficiently few leaf nodes, so that this was not an issue. To scale up the approach for larger graphs, we would suggest a sampling-based approach that only performs a subset of the removals, or a random walk-based approach as used for large network graphs [58], to work well.

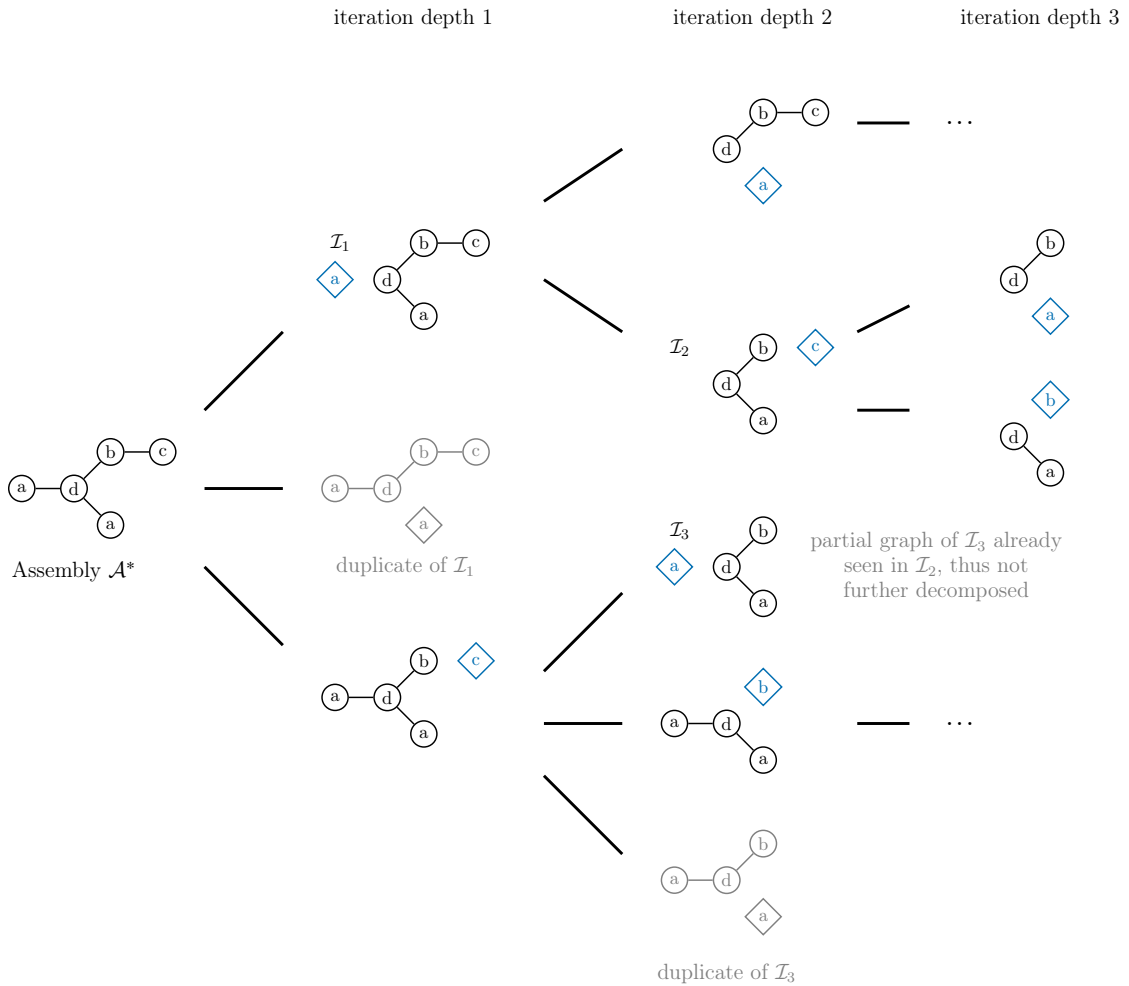


Figure 5.3: Tree of recommendation instances resulting from decomposition of assembly graph up to a `min_size` of 2. The labels are depicted as blue rhombus. Some branches are omitted (...) in order to highlight interesting situations: (1) In the first iteration, the first two instances are duplicates (\mathcal{I}_1), which is why this instance is only included once in the resulting instance set. In addition, the associated partial graph of the second instance is not decomposed any further. (2) In the second iteration, the partial graph of the third instance \mathcal{I}_3 is isomorphic to the partial graph of the previous instance and is therefore not decomposed any further. However, as their labels are different, both instances \mathcal{I}_2 and \mathcal{I}_3 are added to the instance set.

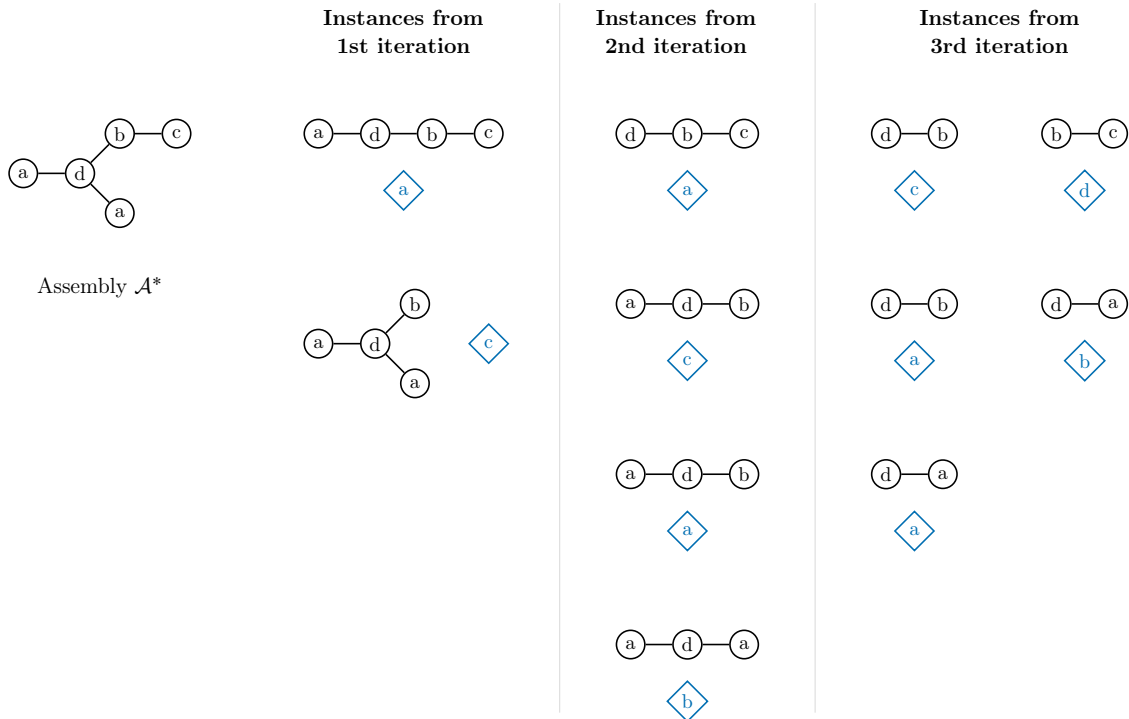


Figure 5.4: Visualization of resulting recommendation instances for deconstruction of example assembly graph until `min_size` 2. The instances are composed of partial assembly graph and part type as label (illustrated as blue rhombus).

Due to the different graph structures, in particular graph order and node degrees (cf. Table 4.1), of the respective datasets, between 220 thousand and 2.5 million recommendation instances arise from each of the twelve thousand assemblies, as Table 5.1 shows. The execution time for the decomposition of all assembly graphs in a dataset was between one and a half and 16 hours when using multiprocessing with eight parallel processes on an Intel Core i7-8700K processor with 3.70GHz. As the instances have to be computed once only, the calculation time is acceptable.

Table 5.1: Number of recommendation instances for all three datasets.

Dataset	# Part Types	Size Training Set	Size Validation Set	Size Test Set	Total
A	1,930	133,271	42,531	43,420	219,222
B	3,099	1,531,257	474,233	491,492	2,496,982
C	1,924	1,050,993	378,712	399,763	1,829,468

If we look at the generated instances, we see that there are several possible target part types for a specific partial assembly due to multiple different parts that can be assembled to the partial assembly, so the mapping from input to target is not unique. If we collect all possible target part types for an input graph, we combine several single-label target instances into a multi-label target instance. The multi-label target are sets, i.e., every part type occurs at most once in a multi-label target. This corresponds to our understanding that several parts fit to one partial assembly which is why we recommend several parts.

For training our recommendation models, we stick to single-label instances, as the more frequent occurrence of an instance reflects the real-world bias and leads to higher ranking of the corresponding part in the predictions. However, the aggregated multi-label instances are suitable for further evaluation, and are thus used in the experiments.

In literature, there are various ways of representing instances for a multi-label task, including breaking them down into multiple single-label instances as we do, referred to as problem transformation method PT5 in [137]. Other presented methods randomly discard all but one label per instance (PT1) or discard all instances with more than one label (PT2), which would result in significant information loss. Method PT3 transfers every different set of labels into a new label leading to datasets with a huge number of new classes but few examples per class, which is undesirable for machine learning. Lastly, PT4 suggests learning a binary classifier for every different label. This problem transformation is common practice in machine learning, but undesirable for our use case: as the labels are evaluated independently of each other, it is difficult to sort the predicted labels by relevance. If we had perfectly trained binary classifiers, they would all predict 100% probability for the corresponding target part, thus all predicted parts would be placed in first position.

We would like to emphasize that the instances were extracted from real-world data and therefore show unequal distributions of parts in the assemblies, assembly sizes and especially target part types. This distinguishes our datasets from well-curated ones, but is common in the field of NLP, for instance.

5.2 Baselines and Upper Bound

The task of part recommendation in assembly modeling as well as our assembly datasets have not been explored before, neither with machine learning nor classical AI approaches. In order to better assess the results of our models, we want to investigate our datasets to estimate the performance from below and above, as well as to apply classical AI techniques as baseline. The baseline model actually solves the task of part recommendation, whereas the bounds are only provided for better understanding the task and the results. Intuitively, the upper bound considers the whole context of the assembly and memorizes the mapping from input graph to part type, whereas the simple baseline does not take any context into account to generate predictions.

Frequency-Based Baseline Model The task of recommending parts for assembly modeling is similar to market basket analysis, as both aim to suggest additional candidates for a given collection of elements. Most market basket analysis methods identify frequent itemsets and associated rules through counting. However, unlike market basket items, parts in an assembly are explicitly connected [1]. To our knowledge, there is no established technique in the literature that considers these connections and thus the graph structure. Therefore, we developed an instance-based model inspired by market basket analysis as a baseline. This model stores a relative frequency distribution of assembled part types for each partial assembly in the training set. During inference, for a given query graph, it identifies which part types were most frequently attached to it. If the graph has not been seen in the training data, it searches for previously seen subgraphs that together form the query graph. These subgraphs should be as large as possible to cover the assembly context extensively and provide accurate part recommendations. To avoid redundant subgraphs, the model identifies the minimal set of its largest observed subgraphs that subsume the

query graph. Then, it aggregates the relative frequency distributions of part types for these subgraphs into an overall distribution by taking the part-wise maximum. Finally, it identifies the k most frequent part types from the distribution. The component of graph subsumption is essential, since in our experiments only a portion of the partial assemblies of the test sets were included in the training data, cf. Table 5.2. This is basically a kind of generalization to similar seen graphs like in neural networks.

Table 5.2: Number of test instances whose partial assemblies occur in the training instances per dataset.

Dataset	Number of Filtered Test Instances	Proportion of Whole Test Instances
A	33,586	77.35%
B	229,947	46.79%
C	240,857	60.25%

Upper Bound As already discussed, there are typically several target part types for one partial graph in our recommendation instances. This implies that for a given partial assembly, if the number of possible recommendations is less than the number of suitable parts, a model can never list all types, i.e., it cannot achieve 100% performance. Therefore, for each dataset we determine the *upper bound* of the top- k rate a perfect, but non-oracle, model could reach for k suggestions. This is an estimate of the performance that can be achieved with the best recommendation strategy for the selected test set. To do this, the test set is analyzed to determine which recommendation order produces the most hits: We analyze the distribution of the targets on the test set and sort the part types in descending order based on their occurrence. This is a deterministic procedure in order to be correct for most single-label instances.

The Evergreen Model There is also the question of the difficulty of the recommendation task, for example whether the same parts are always requested or whether the target parts actually depend on the input graph. To answer these questions, we apply a simple model called Evergreen: This model predicts the k most common labels seen during training (based on a frequency distribution of the labels from the training set), independent of the specific input, representing an unconditional model $P(\mathcal{T})$. The comparison with this model is to show whether the GNNs are capable of processing contextual information from the input graph and to prove that the prediction task is indeed non-trivial. The basic idea of this model is common for highly imbalanced datasets [51] to see if the learned models are better estimators than a simple model that only outputs the most frequent label.

5.3 Experimental Setup

To test the applicability of our approach, we perform experiments on each of the three assembly dataset individually. The assemblies divided into training, validation and test were separately transformed into recommendation instances (cf. Section 5.1). As the task is modeled as a (graph) classification problem, we apply cross-entropy as loss function during training.

Representing Part Types In a preliminary study (Section 4.3), we trained low-dimensional, continuous embeddings for the part types to serve as features for the respective parts of the assembly graphs. Embeddings of dimension 20 and 100 yielded the best results, whereby the latter consistently outperformed the 20-dimensional embedding. In the context of part recommendation, we want to investigate if pretraining part type embeddings is beneficial over starting from a one-hot encoding. Therefore, we additionally generated a one-hot encoding for part types. In order to evaluate the conduciveness of the pretrained part type embedding on the task, we compare the performance of the GNN models operating on the assembly graphs where the part types are represented by the one-hot encoding with those processing assembly graphs with one of the embeddings from the preliminary study serving as node representation, respectively.

GNN Models We investigate the four GNN architectures GAT, GCN, GIN and GraphSAGE presented in the foundations (Section 3.3.2) for part recommendation, as they belong to the state-of-the-art GNN architectures [103]. Beyond GNNs, we also investigated Graph Transformer models for part recommendations. However, they did not achieve a better performance nor could we derive a generic model architecture for all datasets, so this is presented as a digression in Section 5.5.2. All the GNN architectures have the following hyperparameters in common: number of hidden layers, dimension of hidden layers, and activation function. GCN is one of the earlier developed and rather simpler architectures, nevertheless has achieved competitive results to more advanced architectures, e.g., [70]. With GATs, the neighboring nodes can be individually weighted by an attention mechanism before aggregation, which may prove advantageous for our task as some parts of the current assembly may be more important for recommending next parts than others. The number of attention mechanisms (heads) is a hyperparameter. Due to their isomorphism property, GINs should be able to decide whether the connection possibilities of a part are already exhausted and therefore if suitable parts for it should be proposed. To exploit their theoretical advantages, the processing MLP from Equation (3.23) must consist of at least two processing layers; we have set this hyperparameter to exactly the value 2. The interesting aspect of the GraphSAGE architecture for our task is above all the learnable aggregation function, which is to be selected as a hyperparameter. In a preliminary study, we investigated various sampling strategies for GraphSAGE, but these consistently led to (slightly) poorer performance values. The reason for this is possibly the very small graph size of our considered assemblies, so that crucial information is lost through sampling. The sampling step was originally included to the architecture in order to keep the computational costs similar and small when processing large graphs with high node degrees. As the number of neighbors (i.e., node degrees) are ranging from 1 to 18 in the original assembly graphs (cf. Table 4.1), which is absolutely computationally acceptable, we use this architecture without neighborhood sampling on our data. For the aggregation step, we investigated mean and pooling as aggregator functions. Furthermore, we employ dropout as regularization technique in every but the first processing layer, where the dropout rate is also a hyperparameter.

When implementing the GNN models, a deep learning framework for graphs was needed, which supports automatic differentiation and thus Backpropagation of errors and already provides GPU support for training the models on such hardware. At the time of writing this thesis, three frameworks were available: PyTorch Geometric [36], PyTorch3D [116], and Deep Graph Library (DGL) [146]. PyTorch3D deals with the pro-

cessing of three-dimensional data such as meshes or point clouds, and would therefore be more suitable for processing the 3D information of the parts in part modeling, but not for processing our assembly graphs over parts. We chose to take DGL due to their better benchmarking results [146] over PyTorch Geometric and because the framework was more mature when addressing our first GNN use cases of this work. The framework supports implementations for all common GNN layers, including our four architectures, which were used to implement our GNN models in PyTorch [110].

Each of the four GNN architectures were trained based on the three types of part type representation: pretrained 20- and 100-dimensional embedding and one-hot encoding. Including the two baseline models (simple Evergreen model and frequency-based baseline model) and the upper bound used to frame the part recommendation problem, we examine a total of 15 models per dataset.

Training and Evaluation Following the standard methodology of training machine learning models, the models were trained on the training set while monitoring their generalization on the validation set. After each batch of 1024 instances, the model’s weights were updated using the optimization algorithm ADAM [77]. We apply the method of early stopping to determine when to stop training. To rate the models, we calculate the sum of the training loss and the difference between the training and validation losses, as suggested in [51, p. 418]. If the summed loss has not decreased over a patience of 20 epochs, training is stopped and the model with fewest combined loss is selected. The precise hyperparameters per architecture were determined using automatic hyperparameter search with the framework Optuna [2]. Different hyperparameter settings and model architectures are compared based on their achieved validation loss. The final models are evaluated on the test set. As the task of part recommendation is modeled as a graph classification problem where each class corresponds to a part type, the top- k rate is used as performance metric to rate multiple recommendations. The metric incorporated both the correctness of the recommendations and their order.

5.4 Experimental Results

Table 5.3 presents an overview of the performance of the models as well as the baselines and upper bound for the top- k rate¹. Additionally, Figure 5.5 visualizes the performance of the best models and bounds per dataset. The bounds span a wide corridor, so that the Evergreen model is between 28 and 89 percentage points below the upper bound. This proves that the task of part recommendation is not trivial, as it is not solved satisfactorily by suggesting the k most frequent target parts of the training set by the Evergreen model. The Evergreen model illustrates the uneven distribution of the target values: it shows that the ten most frequent target part types were sought in between 15% and 35% of cases (as opposed to the range between 0.03% and 0.05% when evenly distributed), depending on the dataset. For datasets with such few part types, even such a simple model could provide support. However, the frequent parts are usually known so that they can be found quickly. Both the baseline and the GNN models are clearly superior to the Evergreen model for each

¹The results presented in this thesis of the comparison models Evergreen, frequency-based baseline and upper bound model differ from the original publication [39], as those have been subject to an off-by-one error. This thesis presents the corrected values.

Table 5.3: Summary of results for part recommendation per dataset based on the top- k rate evaluated on the test set for $k = 1, \dots, 20$ recommendations. The performance of the upper bound, baseline and Evergreen model is given to contextualize the results. All values are given in percent. The identifier following the GNN model architecture indicates the employed part type representation: 100-, 20-dimensional embedding or one-hot encoding (1H). The overall winner per dataset and k is highlighted as bold; the winner per representation is annotated with a suffix asterisk (*).

Dataset	Model \ k	1	2	3	5	10	15	20
A	Upper Bound	60.0	88.8	96.2	98.6	99.7	99.9	100.0
	Baseline	37.2	59.7	68.6	75.2	80.8	83.0	84.0
	Evergreen	2.3	4.5	6.3	9.7	15.0	19.0	22.2
	GAT-100	46.7	71.4*	79.7*	85.2*	90.0*	92.2*	93.3*
	GCN-100	47.4*	71.0	79.3	84.9	89.6	91.5	92.7
	GIN-100	47.0	70.6	78.7	83.8	88.2	90.0	91.3
	GraphSAGE-100	46.4	70.6	79.3	85.1	89.8	91.6	92.7
	GAT-20	47.9*	72.2*	80.4*	85.5*	89.8*	91.5*	92.5*
	GCN-20	46.6	70.9	79.5	84.8	89.4	91.3	92.4
	GIN-20	44.3	66.5	75.9	82.4	88.3	90.7	92.1
	GraphSAGE-20	44.5	69.2	78.2	84.2	89.0	91.0	92.1
	GAT-1H	46.0*	70.3*	79.0*	84.6*	89.6*	91.5*	92.6*
	GCN-1H	45.4	69.3	78.1	83.8	88.6	90.7	92.0
	GIN-1H	35.7	51.1	60.3	69.3	78.4	82.4	84.7
	GraphSAGE-1H	44.4	68.8	78.0	84.0	88.9	91.0	92.2
	B	Upper Bound	44.3	71.4	88.2	98.0	99.7	99.9
Baseline		18.1	31.9	42.7	55.5	68.0	72.7	74.7
Evergreen		9.4	13.3	15.5	20.3	28.4	35.3	41.7
GAT-100		30.8*	50.6*	63.1*	73.8*	82.1*	86.1*	88.4*
GCN-100		30.5	49.2	61.2	72.4	81.8	85.9	88.0
GIN-100		30.1	49.2	61.4	72.5	80.8	84.5	86.5
GraphSAGE-100		29.2	48.0	60.8	72.9	82.0	86.0	88.4
GAT-20		30.0*	50.1*	62.6*	73.4*	82.0*	85.9	88.0
GCN-20		28.0	46.4	58.8	71.4	81.3	85.6	88.1
GIN-20		29.1	46.9	58.7	70.5	80.3	84.6	87.3
GraphSAGE-20		26.0	44.4	57.9	72.2	81.8	86.0*	88.4*
GAT-1H		30.2*	49.5*	61.8*	72.5*	80.7*	84.6*	86.6
GCN-1H		27.8	45.1	56.8	69.2	79.8	84.3	86.7*
GIN-1H		28.0	44.8	56.0	67.5	77.1	81.5	84.3
GraphSAGE-1H		29.1	47.7	60.0	71.7	80.6	84.4	86.6
C		Upper Bound	36.7	61.5	79.5	96.5	99.9	100.0
	Baseline	17.3	31.4	43.3	59.5	71.0	73.0	73.8
	Evergreen	7.8	14.4	19.9	25.5	35.7	43.7	48.1
	GAT-100	28.5*	49.3*	65.0*	81.8*	92.8*	95.8*	96.9
	GCN-100	27.9	48.3	63.8	80.5	92.0	95.5	97.1
	GIN-100	25.3	44.7	59.8	77.7	91.5	95.6	97.3*
	GraphSAGE-100	27.8	48.6	64.4	81.0	91.5	94.9	96.1
	GAT-20	27.9*	48.5*	64.0	80.7	91.9	95.1	96.3
	GCN-20	27.5	48.4	64.2*	81.2*	92.5*	95.8*	97.1
	GIN-20	27.2	47.3	62.5	79.9	91.9	95.9	97.3*
	GraphSAGE-20	24.3	43.7	59.4	80.0	90.8	95.2	97.0
	GAT-1H	27.5*	48.4*	64.0*	80.6*	91.2*	94.0	95.2
	GCN-1H	27.0	47.1	62.4	79.2	90.8	94.2*	95.7
	GIN-1H	22.5	39.5	52.6	70.7	86.9	91.2	93.5
	GraphSAGE-1H	25.2	45.2	61.1	79.5	90.7	94.2*	96.0*

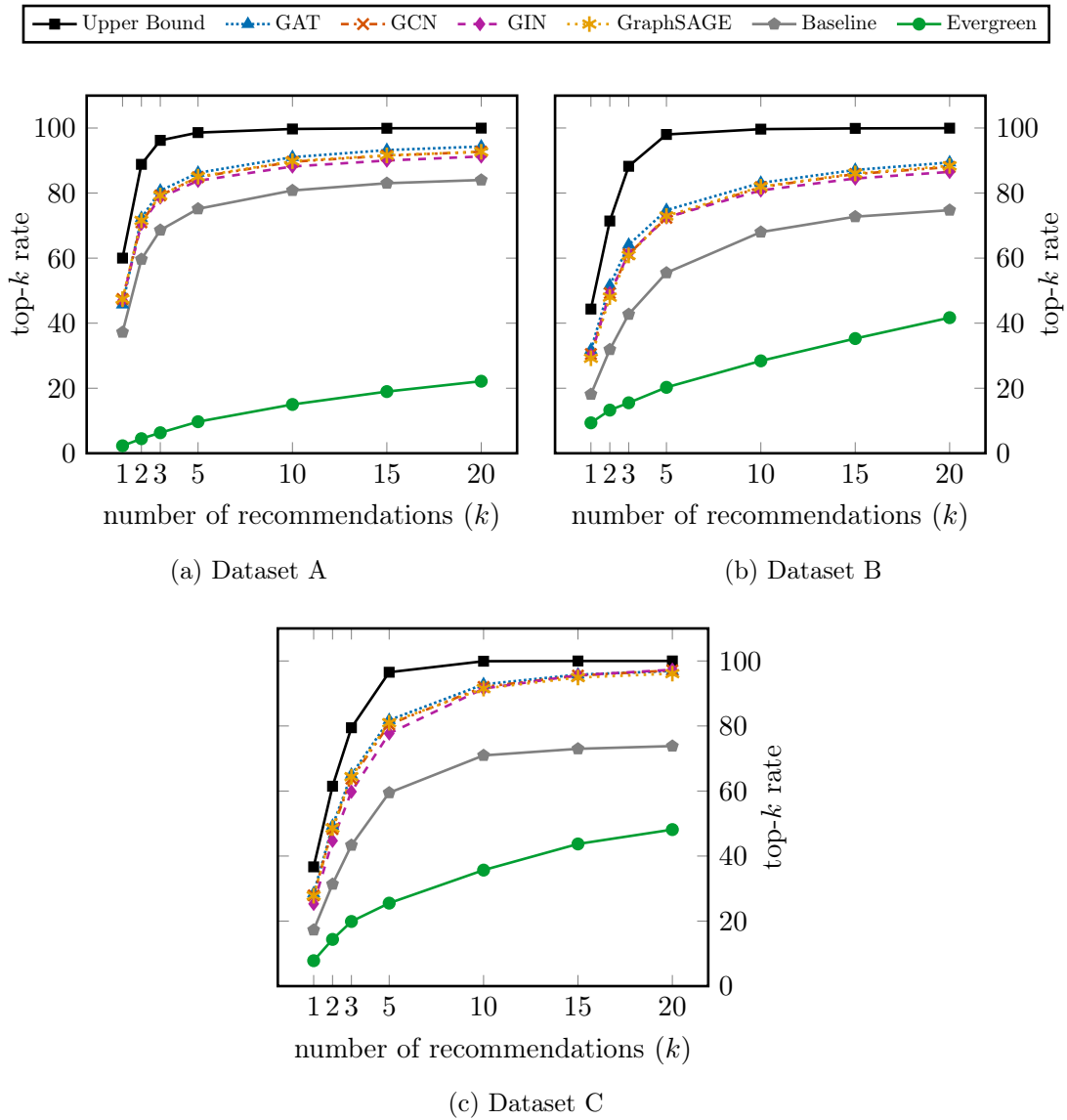


Figure 5.5: Visual comparison of part prediction models: For all GNN architectures, the best models (embedding size 100) were used for each dataset.

dataset, with the latter performing considerably better. For every number of recommendations, the GNN models outperform the baseline by a large margin, demonstrating that they are capable of generalizing beyond exact subgraph pattern-matching. When assessing the baseline model, a direct lookup of the input graph could only be performed for some of the instances. Consequently, subgraphs typically had to be identified and combined to form the input graph. Unlike GNNs, the baseline model does not scale well with the size of the datasets or assemblies, making GNN models better suited for larger datasets.

Influence of the Part Type Representation

In our experiments, we used three different variants to represent part types: two embeddings of different size and one-hot encoding for the part types. On the one hand, we want to investigate whether the assumption of correlating supervised and unsupervised task for the selection of the two embeddings holds in our setting and, on the other hand, whether pretraining of part type embeddings is advantageous compared to one-hot encoding. As already discussed, finding representations is an unsupervised learning problem that cannot be evaluated directly due to the lack of labels. In accordance with our goal of automatic learning, we use extrinsic evaluation with the help of a downstream task, namely the part recommendation: Given a good data representation, the part recommendation models can achieve good performance.

The respective embeddings of dimension 20 and 100 stem from a preliminary study of part2vec (cf. Section 4.3.1) in which these dimensions achieved the lowest error values from a range of embedding sizes. Part2vec is a generalization of word2vec to graph structures and is therefore also based on the assumption that the unsupervised task of getting embeddings is well solved if the corresponding supervised task (mapping center nodes to context nodes) is well solved. For this reason, we evaluated the quality of the learned embedding by the loss of the supervised model. The preliminary study showed that embeddings of 100 dimensions or more resulted in lower loss values compared to those with only 20 dimensions.

We now want to investigate whether this assumption actually holds, i.e., whether models based on the higher dimensional embedding perform better than those using the lower dimensional embedding. Figure 5.6 visualizes the performance of the various GNN architectures depending on the part type representation for dataset B. GNNs using 100-dimensional embeddings slightly outperform those with 20-dimensional embeddings. This improvement with increasing embedding dimensions is consistent for all model architectures, whereby there is an outlier with GINs for $k = 20$ recommendations. This suggests that using even higher embedding dimensions might further enhance performance, but probably only marginally. However, our preliminary study did not show any improvement in the pretraining task with higher embedding dimensions. Therefore, we chose not to investigate them further for the part prediction task as bigger embeddings would result in more computational costs.

Both preliminary study and extrinsic evaluation showed the superiority of the higher dimensional embedding, thus indicating that the assumption of correlation between the supervised and unsupervised tasks in part2vec holds. The comparison with one-hot encoding shows clearer differences in performance: Models based on embeddings consistently outperform those starting with one-hot encoding for the part types, however, only by few percentage points. In conclusion, pretraining proved to be advantageous for the recommendation task, although one-hot-based GNNs also performed well by consistently outperforming the baseline. This could be relevant for practical purposes, such as when there

is insufficient time to set up a pretraining pipeline. Given the superior results of the 100-dimensional embeddings, we use them as default part type representation for the remainder of the thesis.

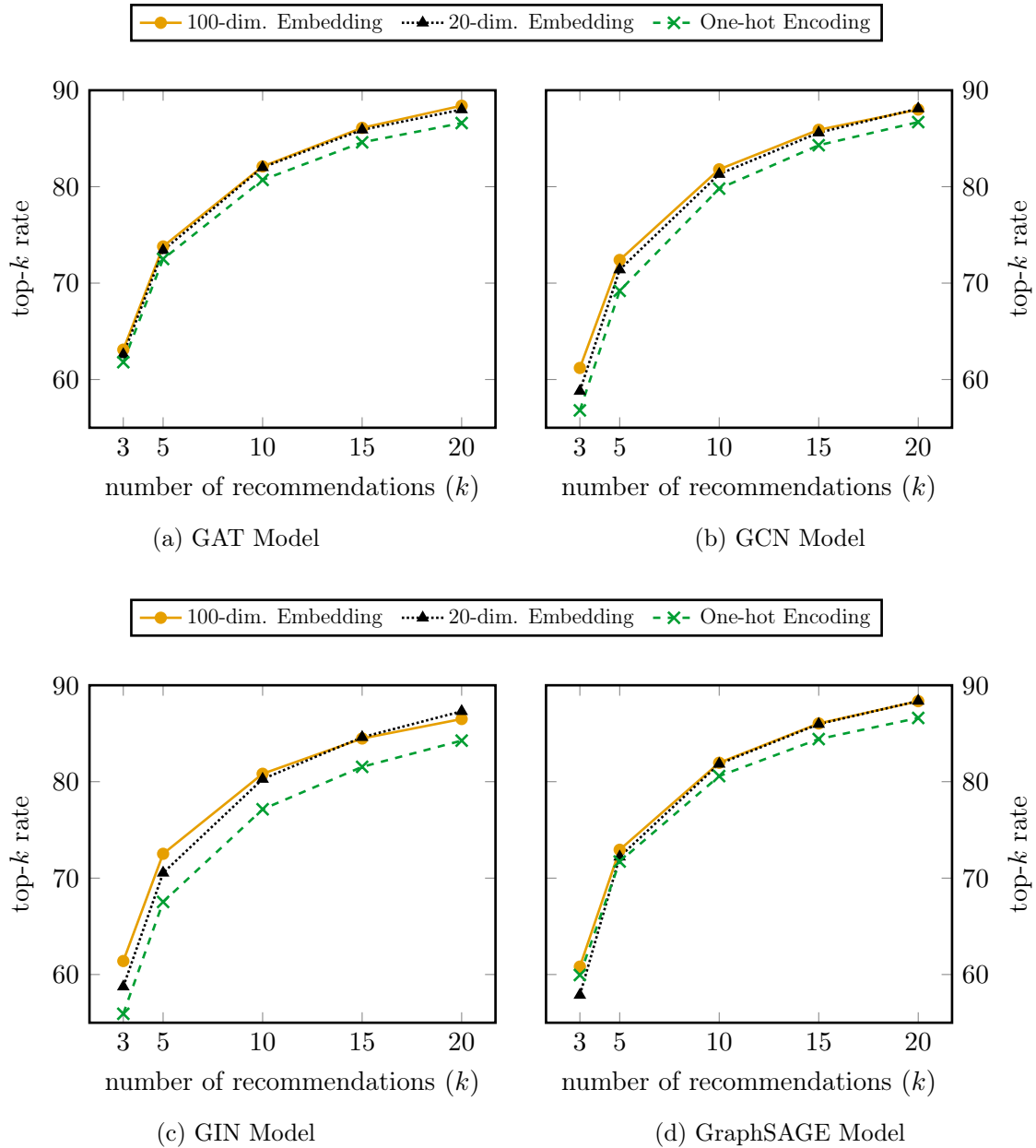


Figure 5.6: Effects of the three part representations on the performance (top- k rate) of the four GNN models for dataset B. $k = 1, 2$ omitted due to scaling.

Performance of GNN Models

The performance of the GNN models increases significantly as the number of recommendations k increases. In terms of top-10 rate, the best performing model (GAT-100) achieved over 90% for datasets A and C and still 82.1% for dataset B, although this dataset is more ambiguous than the other two as it includes more parts – which is also reflected in the Evergreen and upper bound models. This shows that the GNN models can reliably reduce the candidates for required parts from 1,930 (dataset A), 3,099 (dataset B) and 1,924 (dataset C) to 10, which is a useful preselection for designers.

The upper bound indicates the maximum performance that can be achieved by a non-oracle model for a certain number of part type recommendations for the respective test set. For example, with $k = 1$ recommendations on dataset B, GAT-100 achieved 30.8% of a maximum of 44.3%, i.e., almost 70% of the maximum performance. For $k = 10$ recommendations, the upper bound for each dataset is almost 100%, such that the performance values of the recommendation models are basically already normalized to the maximum value.

In a direct comparison of the GNN architectures, the GAT models achieve slightly better results with the same embedding consistently on all datasets, as illustrated in Figure 5.7 as an example for dataset A. This makes the GAT with an embedding size of 100 the overall winner. Interestingly, GCNs outperform GINs in the vast majority of cases, although the latter architecture should be superior by design as it can distinguish more graph structures than GCN (cf. Section 3.3.2). This shows that theoretical superiority does not always translate into practice. There is no discernible pattern for the other architectures; all models yield quite similar performance values. However, it is noticeable that the performance of the GIN model drops significantly when using one-hot encoding for datasets A and C.

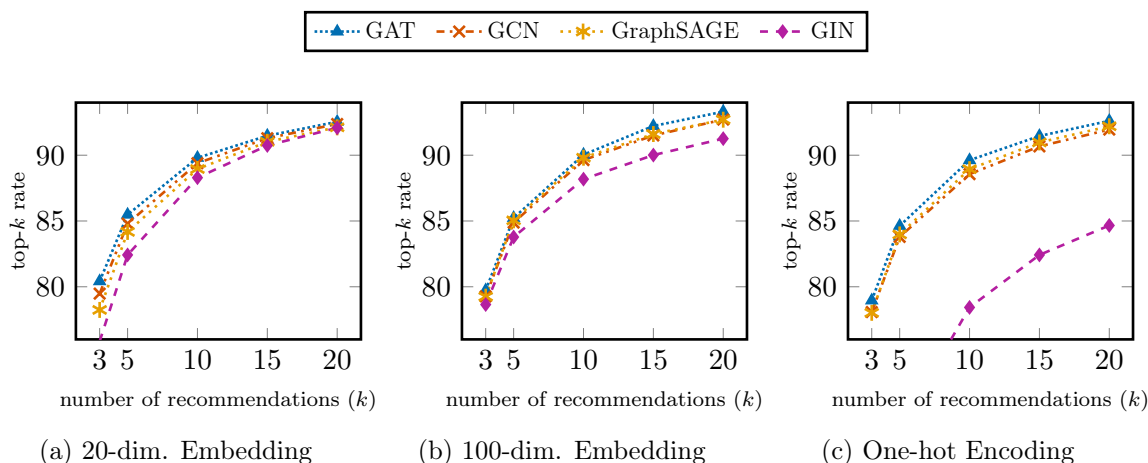


Figure 5.7: Comparison of the performance of GNN models trained on the same part representation for dataset A. Visualization of $k = 1, 2$ omitted due to scaling.

The update rule of GAT and GCN is quite similar, the key difference between them lies in how the models weight the neighbor nodes: in GCN, all neighbors are weighted with the same constant factor based on the graph structure (cf. Equation (3.13)), whereas in GAT, an individual weight for each neighbor node is determined based on the node features, specifically the attention scores (cf. Equation (3.17)). The ability to individually

weight neighbor nodes must be a decisive factor for our problem, as GATs consistently achieve better results. To investigate whether a GAT indeed weights certain neighbor nodes higher (utilizing the attention mechanism), one can compare the attention distribution to the uniform distribution, as suggested in [165]. For each graph instance and each of its nodes, the attention distribution over the neighbor nodes and the corresponding uniform distribution are determined. Then, in order to compare the respective distributions, we determine the Shannon entropy of each distribution, a scalar value that describes the disorder of a distribution (cf. Section 3.1). Figure 5.8 shows a histogram of the calculated entropy values (attention distribution and uniform distribution), excluding nodes with exactly one neighbor (i.e., leaf nodes), as for these nodes the learned attention distribution always matches the uniform distribution. Since the characteristics were similar for all GAT models, the figure only visualizes one particular model (GAT-100 for dataset B). The high prevalence of entropy 0 scores indicates that in many neighborhoods within the assemblies, the attention scores focus almost entirely on a single (or very few) neighbors, resulting in a distribution that is very different from the uniform one.

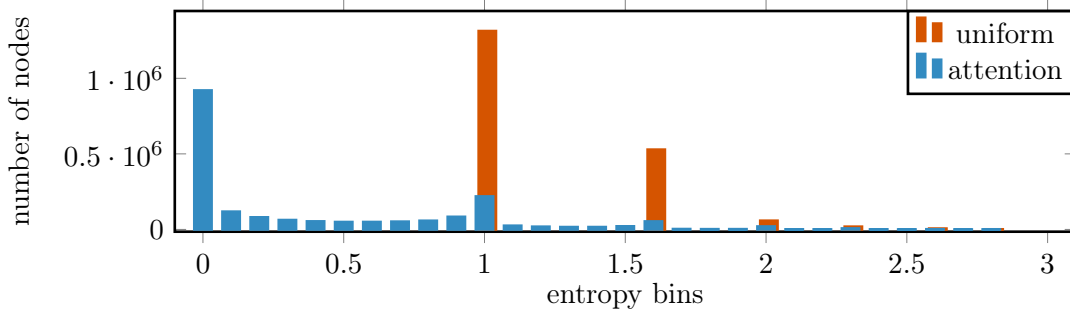


Figure 5.8: Histogram of calculated Shannon entropy of the attention distribution and the corresponding uniform distribution evaluated on the test set. The attention values originate from an arbitrarily selected head of the last GAT layer from GAT-100 trained for dataset B. For visual differentiation, the bars for the uniform distribution are slightly shifted. Nodes with only one neighbor were excluded, since their attention distribution equals the uniform distribution.

Finally, we would like to discuss the training and evaluation time of the GNN models. All models were trained with an Nvidia DGX-1 station² composed of eight Tesla V100 GPUs, where one GPU was sufficient for training. We consider the execution times for dataset B because it contains both the most instances and those with the largest partial graphs, confer Tables 4.1 and 5.1. The times barely differ for the different GNN architectures. The GAT-100 model took on average 6.4 minutes for one training epoch (including the calculation of the training and validation loss), with all models being trained in less than 10 epochs. This means that the training of a model was completed in under an hour. When evaluating the trained model on the test data, about 1000 instances could be evaluated within one minute. The inference time is therefore negligible. For the other datasets, both training and evaluation times are lower due to the fewer instances to process.

²The station features 40 CPUs, in particular dual-socket Intel Xeon CPUs E5-2698 v4 with 2.20GHz. The GPUs are interconnected through Nvidia NVLink.

Difference in Training and Test Labels

The assembly graphs examined in this thesis originate from real-world orders by customers. They thus show which parts were combined in order to create the respective required assembly. However, they do not provide a list of all possible assemblies over the part types. Since we extract the recommendation instances from these sample assemblies, the instances (represented as multi-label) also contain only a subset of all possible matching parts for a given partial assembly. In short, we do not know the exhaustive list of parts that would really fit to a partial assembly, i.e., the exhaustive list of labels for an input. Creating the list of all possible part connections would require intensive manual effort by design experts. It can therefore happen that a model suggests a suitable part by generalizing from similar assemblies, but we do not count it as correct in the evaluation because it is not among the added parts in the evaluation set. Consequently, the displayed performance hit rates are a lower estimate and could be even higher in application.

Furthermore, it is therefore also possible that the labels for a particular partial graph in the training data differ from the corresponding in the test data and during inference. While a model has been trained to output certain part types in training, these might not be counted anymore as correct in the evaluation. Also, a new part type could be required that was not among the labels for this partial graph in the training. Let’s consider this situation in a simple example: assume that for a particular partial assembly \mathcal{A}_i , part type τ_1 appears twice and part type τ_2 once as targets in the training instances. We would expect the model to output τ_1 first and τ_2 second when performing predictions for \mathcal{A}_i , i.e., the output is ranked according to relevance. Assume that in the test instances, however, for the same partial assembly \mathcal{A}_i , the corresponding test targets consist of twice part type τ_1 and once part type τ_3 , because these part types have been added to the partial assembly in the test assemblies. Consequently, predicting τ_2 for \mathcal{A}_i is not counted as correct in test, and the model is moreover expected to predict part type τ_3 as well – GNNs could achieve this through generalization. For $k = 2$ recommendations (τ_1 and τ_2), the model would yield 66.6% hit rate on these test instances.

Table 5.4: Estimate of the impact of the difference in training and test labels for part recommendation on all datasets. The data set used to learn the upper bound model is indicated in the second column. Both models were evaluated on test instances, whose partial graph also occurred in the training instances, noted as “filtered test” set. Top- k rate is given in percent.

Dataset	“Train” Set \ k	1	2	3	5	10	15	20
A	Train	54.10	79.93	87.62	91.38	93.48	94.04	94.22
	Filtered Test	58.63	87.25	95.15	98.18	99.62	99.91	99.96
B	Train	38.33	61.36	75.55	85.17	88.40	89.26	89.53
	Filtered Test	42.25	68.33	84.86	96.10	99.25	99.75	99.89
C	Train	32.24	54.43	70.74	86.43	90.61	90.88	90.91
	Filtered Test	35.81	60.21	78.10	95.77	99.87	99.99	100.00

While we cannot approach the first problem automatically without domain knowledge to compile the complete list of suitable part for a partial assembly, it is possible to estimate the impact of the difference between the labels in training and test for the part recommendation task: To do this, we first filter all test instances to those whose partial assembly also

occurs in training, since we can only measure the difference for such instances. This results in 33,586 instances (77%) for dataset A, 229,947 (47%) for dataset B, and 240,857 (60%) filtered test instances for dataset C, respectively. Then we can use two instantiations of the upper bound model to memorize the training data in the first model, and the test data in the second one. In particular, this model represents a mapping from partial assembly to list of part types, which are sorted according to their occurrence as label. Both models are evaluated on the reduced test set for part recommendation based on the top- k rate; their performance is displayed in Table 5.4. As expected, the second model, which has learned the test data, performs better. However, the difference to the model based on the training data is acceptable, it is in a similar range for all datasets: about four percentage points difference with only one prediction; nine percentage points difference with ten predictions for datasets B and C and six percentage points for dataset A. The effect of the label difference is noticeable, but remains within acceptable limits. However, this may be responsible for the gap between the GNN models and the upper bound in the part recommendation (cf. Table 5.3).

Are the recommendations indeed reasonable?

Part recommendation is motivated by the tedious, time-consuming search for the right parts during assembly modeling. Due to the heuristic instance creation, some instances are created where the target part type (the suggestion) is already present in the input graph. For these instances, the task is not very challenging as the model only needs to “look up” the part type in the input graph. From a designer’s perspective, a part type that is already included in the design does not require further searching as the designer can simply copy the existing part. We want to investigate whether the recommendation models can come up with parts that are not yet included in the current assembly graph. Therefore, we filtered the (test) instances so that we only consider those where the target is not yet included in the input graph, in order to assess if the recommendation model is indeed capable to predict parts not already included and thus if its predictions are reasonable. As for these instances, the set of all part types of the assembly $\{T(p) \mid p \in \mathcal{V}_{\mathcal{A}}\}$ and the (single-element set of) target part types is disjoint, we also denote these filtered instances as *disjoint*.

Table 5.5: Number of recommendation instances before and after filtering for disjoint instances per data set of dataset C.

Data Set	Original	Disjoint	Reduction by Filtering
Training	1,050,993	704,712	32.9%
Validation	378,712	251,113	33.7%
Test	399,763	270,385	32.4%

Table 5.5 exemplifies the reduction of each data set through filtering for disjoint instances for dataset C, where filtering reduces each data set by about one-third. For datasets A and B, the reduction was about 20% and 40%, respectively, and the observations were analogous. The best models trained for recommending ten parts (GAT-100 and GCN-100) were evaluated on the disjoint test set, with the results shown in Table 5.6. Evaluation of the original models on the disjoint test instances naturally delivers poorer results, as the models have been trained to also suggest parts that are already included. When evaluating

on the disjoint test set, these are no longer counted as correct, so the models’ performance is lower than on the original test set. However, this effect averages out for an increasing number of recommendations, reaching a similar performance level latest at $k = 10$. If the filtering is also applied to the training and validation set in order to train new models on them, these models are trained to suggest only parts that are not already included in the partial assembly, which is why they perform better on the disjoint test set and thus outperform the original models for small values of k .

Table 5.6: Performance of trained part recommendation models given as top- k rate in percent evaluated on original and filtered test set for dataset C. Includes performance values of original GNNs from Table 5.3 for comparison.

Train/Val. Set	Test Set	Model \ k	1	2	3	5	10	15	20
Original	Disjoint	GAT-100	21.9	42.7	59.9	79.2	92.0	95.4	96.6
		GCN-100	21.4	41.6	58.2	76.9	89.8	94.1	96.2
Disjoint	Disjoint	GAT-100	31.6	54.4	69.5	83.7	93.3	95.8	96.8
		GCN-100	31.1	53.3	68.3	83.0	93.0	95.7	96.8
Original	Original	GAT-100	28.5	49.3	65.0	81.8	92.8	95.8	96.9
		GCN-100	27.9	48.3	63.8	80.5	92.0	95.5	97.1

We would like to make clear, that a performance comparison is only proper, when the models are evaluated on the same data set (and ideally also trained on the same training set). Consequently, the performance evaluations on the original and disjoint test set cannot be directly compared with each other. However, these analyses demonstrate that the part recommendations not only reflect parts that are already included in the partial assembly, but actually support the design process by reducing the search for new required parts.

Are the recommendations indeed beneficial?

In addition to the quantitative analyses of the trained recommendation models on the test instances presented above, we were also able to evaluate the benefits of the recommendations as part of a user study in the project KOGNIA. For this purpose, the project partner CADENAS GmbH integrated a recommendation model prototypically into the CAD system Autodesk Inventor [64].

The study was carried out on a smaller assembly dataset, which only included 27 training assemblies consisting of parts from a vocabulary of over 2,000 part types. However, the assemblies were noticeably larger with an average of 428 parts. These assemblies were transformed into recommendation instances according to Algorithm 1 and used to train a GAT recommendation model.

In the study, six engineers assessed the quality of all ten part recommendations in the system using a star rating scheme: The full number of five stars was assigned to recommendations that corresponded to the desired target part; if the recommendation could not yet be directly attached to the assembly, but was needed in a subsequent step, four stars were awarded. If a part of the correct part family (cf. Section 2.1) was recommended, but the specific recommended variant was unsuitable for the current assembly, it resulted in another point deduction. Finally, unsuitable variants of parts that could only be used later were given two stars, and unusable recommendations were given one star.

Table 5.7 shows how often part type recommendations were rated with four or five stars in the user study. These two ratings are of particular interest to us, as they effectively support the designer in assembly modeling by providing suitable parts in the current or a subsequent step. With $k = 10$ part recommendations, the study participants received effective support in 76.3% of the recommendations. This means that in over three quarters of the cases, they did not have to search for the required part in the catalogs themselves, as it was suggested to them by our recommendation system.

Table 5.7: Results of the user study on part recommendation: Proportion of four and five star ratings across all ratings of $k = 1, 3, 5, 10$ part recommendations, respectively. Five stars corresponds to the highest possible ranking.

Rating	1	3	5	10
★★★★★	11.0%	24.5%	39.7%	55.6%
★★★★	7.7%	12.6%	16.8%	22.7%

Furthermore, the feedback from the design engineers was consistently positive, describing the system as both convenient to use and an effective aid to reducing the time required for assembly modeling. The user study further revealed that some of the suggestions also involved parts that were previously unknown to the designers. When searching for parts manually, the fact that the envisioned part is not even known to the designer places a huge hurdle. By suggesting these parts, our system enabled the designers to integrate these parts into their assemblies. In summary, the user study proved the effectiveness of our approach and that the experimental results are reliable.

5.5 Digressions

In addition to the methodology and examinations presented so far, we also investigated other established improvement techniques for the use case of part recommendation, but these did not lead to any significant changes. Our approach and findings for the pretraining of the recommendation models and the use of Graph Transformer models are outlined below. Furthermore, we have considered the necessary modifications to our approach for recommending a context-specific number of parts instead of a fixed recommendation number. This subsection ends with the models investigated for this task and their results.

5.5.1 Pretraining of Part Recommendation Models

Experience has shown that deep, regularized neural networks achieve better results than more shallow unregularized ones [51]. These models often have millions of parameters and therefore require a vast amount of data for training. However, obtaining labels for this data is usually time-consuming and expensive. In the fields of image and natural language processing, pretraining models has proven to be an effective strategy to address this issue [51], leading to its establishment in other areas as well. In this approach, models are initially trained either on unlabeled data using an unsupervised method or via an auxiliary task where labels are easier to obtain e.g., by self-supervision. After pretraining the final layers for the auxiliary task, they are replaced by layers for the actual task – similar to transfer learning. The whole model (pretrained first layers and added new prediction

layers) are afterward fine-tuned for the actual task. The idea is to use the auxiliary task to gain a better understanding of the data and thus improve task performance.

Since our datasets contain only 12,000 assemblies, we investigated the usefulness of pre-training for the recommendation models. We chose a pretraining strategy that is applicable for all GNN architectures and operates on the graph structure itself, as we have no further information on the assembly graphs: generative pretraining of GNNs (GPT-GNN) [62]. This technique involves two self-supervised tasks which are assessed by two separate loss terms: predicting node attributes and reconstructing masked edges.

We examined the effect of pretraining based on the number of available assemblies (500, 1,000, 2,000, 4,000, all training assemblies). As expected, pretrained models started with higher performance and reached their peak performance faster. However, the more assemblies were available, the quicker the non-pretrained model caught up, reducing the overall training time required. Unfortunately, in terms of performance, pretraining was not found to be beneficial: pretraining showed only minimal improvements (less than one percentage point) and sometimes even resulted in worse final performance. In summary, pretraining could be useful when there are very few assemblies or limited computational resources; otherwise, it is not particularly beneficial.

5.5.2 Graph Transformers for Part Recommendation

In the fields of computer vision and NLP, central application fields of machine learning, Transformers [141] have established themselves as state-of-the-art technology in almost all applications. As outlined in the foundations, Transformers can be seen as GNNs applying an attention mechanism and operating on fully connected graphs. This naturally led to the development of architectural variants, designed for processing general graph structures, known as *Graph Transformers*. They generally utilize the attention mechanism introduced by Vaswani et al. [141], but differ in how they integrate graph information. For a detailed overview of existing architectural designs, we refer the reader to the comprehensive surveys [98, 102].

Due to the great success of the Transformer architecture, we investigated Graph Transformers as alternative models for part recommendation. Specifically, we investigated the two architectures *GT* [34] and *GPS* [115]. Dwivedi and Bresson presented GT as a natural and simple generalization of the Transformer for graphs. In their experiments on a graph regression and two node classification tasks, they outperformed GCNs and GATs. However, GPS is a highly modular framework presented by Rampášek et al., capable of incorporating various positional or structural encodings, which combines message-passing GNNs with global attention mechanisms. The second architecture was particularly interesting because of its modularity to incorporate structural information of the graph.

In his master thesis [123], Ludwig Schneider experimentally compared the Graph Transformers to the best GNN architecture for part recommendation, namely GATs. All variants of Graph Transformers achieved very similar results per dataset for part recommendation, despite using different attention mechanisms and taking various structural information into account. For each of the three assembly datasets examined, a different Graph Transformer architecture variant emerged as the best model, respectively. Compared to GATs, Graph Transformers were occasionally able to outperform GATs for a large number of recommendations, but only by 1-2 percentage points. We were able to find some tendencies that GAT models performed better on smaller graphs, whereas Graph Transformers excelled on larger graphs.

Interestingly, the Graph Transformer models have significantly fewer parameters than the GAT models, while achieving comparable hit rates. Models with significantly more parameters performed worse, as they showed poorer generalization ability. These results suggest that the available assembly datasets were not large enough to train complex Graph Transformer models with many parameters. We hypothesize that these models would perform significantly better on larger and more complex datasets, as it is the case for their basis architecture Transformers. However, since we do not have any further assembly graphs, this remains to be investigated in future work on larger assembly datasets.

5.5.3 Context-Specific Number of Recommendations

For the part recommendation system, we decided to present the user with a fixed number of recommendations for every assembly, for example ten suggestions. As a result, for certain assemblies there may be considerably more suitable parts so that only a subset of them is listed as recommendations due to the fixed number. Therefore, more recommendations can be requested from the system at any time. However, for other assemblies, fewer suggestions than the fixed number may be indeed suitable, so that unsuitable parts were added in order to fill up to the desired number of suggestions. Since we are aiming for an assistance system under the control of the designer instead of automatic completion of assemblies, imperfect suggestions are acceptable. However, recommending unsuitable parts could diminish the designers' confidence in the assistance system. For this reason, we aimed for adopting the recommendation system to output a self-determined number of part recommendations specific for the context of the current assembly, in short a *context-specific number* of part recommendations. This means that the recommendation model should decide *both how many and which* directly attachable parts should be suggested for the current assembly. The suggestions should again be sorted by relevance so that frequently used parts in this context are ranked higher. However, this implementation has its own advantages and disadvantages: If only in fact suitable parts are displayed, this reduces the number of parts that a designer has to look through when searching for a suitable one. A major disadvantage, however, is that requesting further parts is typically difficult to realize with such systems. So if the desired part is not listed, the designer will probably have to search through the variety of catalogs again.

Providing a context-specific number of recommendations can already be achieved by slightly modifying our existing approach based on a fixed number of recommendations, even without having to train new models. To do this, we simply modify the way of determining recommendations based on the part type scores of the recommendation model's output layer. We refer to the resulting models as baselines in the following. The first possibility is to determine an experimental threshold (based on the validation data) for the normalized part type scores after applying Softmax, i.e., only part types with normalized scores above the threshold are suggested. As before, the part types can be sorted according to their scores to build ranked suggestions. In this modeling, all part scores are correlated, so that scores of unsuitable parts influence the normalized values of suitable parts, which makes the search for a universal threshold difficult. The second possibility is to consider the problem as a multi-class classification, meaning that a set of part types can be assigned to an assembly. This corresponds to problem transformation method PT4 given in [137], as discussed in Section 5.1. Architecturally, this can be achieved by using Sigmoid instead of Softmax as activation function in the output layer. In this way, the model uses a binary classification to decide individually for each part type whether it should be predicted. In

principle, the same sorting approach can also be carried out here, but the assumption that parts with higher scores are more relevant does not necessarily apply: In fact, a perfectly trained model would predict 100% probability for each matching part, so the parts could not be sorted appropriately. Therefore, this approach provides a set of part recommendations rather than a ranking of them. Due to the weaknesses of the simple modifications of the previous approach, we also investigated new architectures independent of the original system.

Basically, a ranked list can be considered as a sequence, i.e., a collection of elements with spatial order. Hence, the desired recommendation output can be understood as a sequence of part types of a context-specific length. Sequential outputs often occur in the field of natural language processing, for instance, when a suitable caption, i.e., a sequence of words, is to be generated for an image. Adhering to typically employed models for sequence generation [118], we investigated RNNs and Transformers [141] for our task. Since the standard architecture of RNNs exhibits gradient problems [51], we used LSTMs [122], a variant that solves this problem. In detail, the sequential-based recommendation models are based on an encoder-decoder architecture, with GNNs processing the assembly graphs as encoders. The two sequential models serve as subsequent decoders to produce a sequence of suitable part types. In order to achieve a context-specific output length, we follow the standard approach of extending the vocabulary by a so-called *end of sequence token* (EOS) [51]. As soon as the model predicts this token, no further part type predictions are retrieved. To train these two sequential model variants, the recommendation instances had to consist of partial assembly and target sequence of matching part types. Within each data set, all recommendation instances with the same partial graphs are first grouped so that a multiset of part types is available. The multiset is then converted into duplicate-free sequences: The part types are sorted in descending order based on their frequency in the respective multiset.

In his thesis, Michael Huber investigated and compared the four presented approaches. Here, we provide a brief overview of the results and findings, for a detailed overview of the results and findings, we refer the interested reader to [63]. The performance measures precision and recall are used to evaluate the task, as well as the F-score as a combination of both values. The hyperparameter selection was based on the F-score. In contrast to classic sequence generation such as with texts, the generated part type sequence should contain each type at most once. To address this particularity, we have applied a small model adjustment for inference to ensure duplicate avoidance: Inspired by the concept of masking employed in the Transformer architecture, we are setting the logits in the output layer of part types already predicted to minus infinity. Since the Transformer architecture is designed to process large amounts of data, the experiments are limited to dataset B, which comprises the largest and most complex assemblies and thus produced the most instances, see Table 5.1.

In the experiments, all models except the first baseline achieved a very similar F-score of about 70%. The first baseline scored almost ten percentage points lower, which is mainly due to its very low precision. For the second baseline and the sequential architecture based on a Transformer, both precision and recall are around 70%. This means that on average across all test instances, 70% of all predicted part types were correct and 70% of all targets were indeed predicted. The sequential architecture based on LSTMs achieves slightly better precision, but at the same time slightly poorer recall.

With regard to correct ordering the predicted part types according to their relevance, the sequential models emerged as the winners. This finding is in line with expectations, as these models – in contrast to the baselines – were explicitly trained for a suitable order. Interestingly, both baselines perform equally well in terms of order, with the second even performing slightly better for few number of recommendations. This is surprising, as the second baseline should be less suitable for ordering the suggestions by design.

5.6 Related Work

Supporting CAD Designers The cognitive demands placed on design engineers using a complex CAD system have already been recognized [87]. The authors employ collaborative filtering to recommend relevant software commands to streamline the workflow. Our method shares this motivation but shifts the focus towards identifying parts that are likely to be added to the current assembly rather than suggesting commands.

Utilizing search engine and information retrieval concepts to assist CAD design engineers is referred to as assembly retrieval [91]. The search can be performed based on keywords, shape similarity [38] or a combination of search criteria [25]. Assembly retrieval involves various similarity criteria (e.g., usage patterns and part overlaps) to facilitate searches for similar designs, whereas our method emphasizes suggesting potential extensions for the assembly while a designer does not have to specify any requirements or queries.

Other search methods focus on the geometry of parts, allowing users to search for similar-shaped parts by using a reference part [13] or by providing a 2D or 3D sketch of the desired part [45]. However, these approaches require the engineer to already have a specific part in mind in order to successfully end up with the correct part.

GNNs in Recommendation Systems The use of graph neural networks (GNNs) in recommendation systems has also been recognized [154], essentially because many recommendation tasks (such as recommending movies to users or products to customers) can be represented as a (hyper)graph, where users and items are nodes of different types, and edges connect users to the items they like or have purchased. Such classical recommendation tasks aim at predicting new connections, i.e., recommending items to users, rather than introducing new nodes to the graph. GNNs are well-suited for these task because they can directly process the graph composed of users, items, and their associated information to generate accurate recommendations. Notable examples of GNN-based recommendation systems have been developed for big social media platforms like Twitter [3], Pinterest [159], or for the e-commerce platform Alibaba [171].

Liang et al. [89] have built upon our approach for part recommendation by investigating a new neural architecture search framework named CusGNN to automatically customize data-specific GNN models on our datasets. Each layer within the recommendation model can have a different architecture, aggregation function and possibly readout function, among others. Through extensive dataset-specific hyperparameter search, specialized models could be developed, which mostly achieve similar performance as the models presented in this thesis, but are up to 1.5 percentage points ahead in some cases. In contrast, we pursue a generic approach in order to find a modeling and model architecture that is adequate for any kind of assembly dataset instead of fine-tuning individually for each dataset. It remains to be evaluated whether the difference in performance measured on the test data has a noticeable impact on the design experience for users of the application.

Generative Tasks on Graphs The task of predicting the next part in an assembly can be seen as a step-by-step generative model for graphs. However, existing approaches to generative deep graph models aim to learn the probability distribution of the entire graph at once, a process referred to as “one-shot generating” in [56]. These methods commonly use techniques such as variational autoencoders [78], generative adversarial networks [145], or normalizing flows. Furthermore, the authors of [88] propose learning a sequence of node and edge insertions, which they termed structure building actions. Their approach, particularly the neural network architecture used to map an intermediate graph to the next insertion action, shares some similarities with our method. However, in their approach, edges can be added in any order, potentially resulting in disconnected graphs – something we explicitly avoid. Our method focuses on inserting individual parts at each step, eliminating the need for the recurrent structures employed in [88]. Additionally, training general generative graph models is reported to be more challenging in terms of balance, which is why we focus on the discriminative task of predicting the probability $P(\mathcal{T} \mid \mathcal{A})$. We handle the generation of suitable data through self-supervision outside the training loop.

Alternatively, graph generation can also be performed using *autoregressive* models, which predict single or multiple nodes or edges step by step, based on the current state of the graph. Given our goal of assisting design engineers by offering suggestions rather than automating the entire process, we opt for this second approach. During CAD modeling, we aim to allow designers to make changes to the partial assembly as needed. Consequently, our model must be able to generate graphs of arbitrary size, which is often not feasible with non-incremental models.

Other methods generate graphs with similar structural characteristics as the training data [90, 161]. These approaches assess only the final graph structure, focusing on aggregated graph statistics like degree distributions, without considering intermediate states. This allows for canonical node numbering (through breadth-first or depth-first traversals) during the generation of training instances, keeping the number of instances manageable since node permutations don’t need to be considered. However, in assembly modeling, engineers may begin with any part or subgraph, and the sequence of part additions can vary according to their preferences. Therefore, we generate training instances for every possible creation sequence of an assembly by iteratively removing nodes which then serve as labels for the resulting partial assemblies. Both [90] and [161] employ RNNs and thus do not incorporate permutation invariance by design, placing them at a disadvantage compared to GNNs. Unlike [88], where nodes can be added without necessarily connecting to the existing graph structure, our approach ensures that newly added parts are always connected to the prior structure, which is essential for maintaining coherence during the design process.

In the context of molecule graph generation, valid molecules are to be generated according to given desired chemical properties. Methods typically incorporate different types of nodes and even of edges, corresponding to different atoms and bond types. Ensuring the validity of the generated graph, e.g., verifying the chemical structure of molecules [129], may seem important for assembly modeling as well. Unfortunately, this validation process is less straightforward for assemblies, as the designer does not have to exhaust all connection points a part offers. However, especially in terms of a recommendation system, designers decide themselves anyway which recommended parts are indeed valid for the current design.

In [153], Transformer-based generative models have been applied to CAD models by generating sequences of CAD-typical geometrical operations such as “sketching”, “extruding”, “boolean subtracting”. While this work addresses part modeling tasks, our method focuses on usage similarity of parts reoccurring in assemblies, abstracts away from geometrical features, and – most notably – does not require a linear order of operations. Instead of geometric operations, we focus on predicting the next required parts in the assembly.

Localized Recommendation By Targeted Part Placement

Summary This chapter focuses on the problem of localized part recommendation in assembly modeling, presenting two primary variants of this use case. The first variant involves recommending appropriate part types for a designer-selected extension point within an existing design. The second variant aims at recommending both an extension point and a corresponding part type for the current assembly. The chapter details the adjustments needed for instance generation to incorporate information about the extension points effectively. Various modeling strategies are explored and compared to baseline models to evaluate their performance in both variants. For the localized part recommendations for a designer-selected extension node, the system was able to provide the expected part in 87.5% to 97.5% of the cases when suggesting ten parts. In the case of the more complex localization task, both components (a next part and its localization) were still correct in 67.2% to 84.9% of cases for ten recommendations. Thus, in both cases, our approaches could significantly reduce the search time for designers.

Contents

6.1	Generating Instances From Assemblies	98
6.2	Variant I: Recommending Part Types for User-Given Extension Point	100
6.2.1	Experimental Setup	101
6.2.2	Experimental Results	102
6.3	Variant II: Recommending Extension Point and Part Type	105
6.3.1	Modeling Approach (1): First Predicting Part Type, Afterward Extension Point	106
6.3.2	Modeling Approach (2): First Predicting Extension Point, Afterward Part Type	109
6.3.3	Experimental Setup	110
6.3.4	Experimental Results	112
6.4	Related Work	116

Publications The use case of localized part recommendation has initially been published in [86]; the specifics on instance generation and the two localization variants including methodology and experiments are published in [85].

For the use case addressed in Chapter 5, part types are recommended on a *global* level for the whole assembly, i.e., without locating which part of the current assembly they should (or could) be connected to. Thus, the part placement must be fully carried out by the user. The focus of the previous use case was on *what* to directly connect to the current assembly, without considering where. We will therefore refer to the first use case as global or “non-localized” part recommendation in the following. Global part recommendation was the central use case investigated within the research project KOGNIA. After its completion, we concentrated on more targeted part recommendation during the design process in order to support designers even more effectively: Generating global recommendations may be acceptable for small assemblies, but becomes unwieldy for large assemblies with plenty of possible connection options. We therefore also want to provide support for determining *where* to connect a new part. In addition, localization can also be seen as a kind of filtering of the suggestions, as non-localized suggestions apply to the whole graph: If some suggested parts would fit to several existing parts, they would appear higher up in the ranking and thus displace suitable parts for fewer locations. If a designer wants to continue building on a specific part, the current suggestions may not be suitable for connecting to it. Clearly, they can request more recommendations until the desired part is listed, but this is not efficient.

This use case is not only aimed at reducing the time needed to search for required parts, but also at providing support for the whole designing task, i.e., selecting *and* connecting parts. However, we want to explicitly point out that localization in our use case refers to an existing part of the current assembly, as opposed to the specific geometric relation in a CAD system. This means that only parts suitable for the selected existing part are proposed, but the designer still has to align and rotate the proposed part to be attachable to the selected part of the assembly. The existing part of an assembly to which a recommended part is to be attached is in the following referred to as the *extension point* or *extension node*¹ of the assembly. An inexperienced designer could use this recommendation system to view suitable parts for the extension node and thus indirectly learn design rules. This system can be the foundation for developing a training system for novice designers.

The localized part recommendation addressed in this chapter aims to support designers not only in the selection but also in the placement of the selected parts. This leads to an extended operator $\mathcal{A}_{n \triangleleft} \tau$ for adding a part of type $\tau \in \mathcal{T}$ to the extension node $n \in \mathcal{V}_{\mathcal{A}}$ of assembly \mathcal{A} . However, the exact geometric positioning and alignment of the recommended part at the extension point in a CAD system is not part of this approach. This aspect is dealt with by works such as [68] and [150], that position chosen pairs of pre-selected parts, which can basically follow on from our work. Due to the two shortcomings of non-localized part recommendation mentioned above, we investigate two distinct variants of localized part recommendation:

VARIANT I: RECOMMENDING A NEW PART FOR A DESIGNER-SELECTED EXTENSION POINT

In a CAD system, this can be envisioned as follows: a designer selects (e.g., clicks on) an existing part of the assembly to receive recommendations for new parts to attach to it. By default, the most recently added part could be pre-selected as desired extension point. This variant corresponds to “filtering” part recommendations to a specified extension point. Both the current partial assembly and its extension node are provided as input to the conditional recommendation model learning $P(\mathcal{T} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$.

¹Since several instances of the same part type can occur in an assembly, we try to avoid using the term “extension part” in order to create clarity regarding the localization.

Variation II: Recommending Both a New Part and Its Extension Point The second localization variant extends the former recommendation by additionally recommending the extension point. For a given partial assembly, both a new part type and its extension point are recommended, given by $P(\mathcal{T}, \mathcal{V}_{\mathcal{A}} \mid \mathcal{A})$. In a CAD system, when a designer hovers over a recommendation, the extension nodes could be highlighted. In addition to the selection of the part type, this would also reduce the cognitive burden on designers of selecting the extension point.

The requirements for this use case essentially correspond to the previous one and are therefore briefly summarized: We again assume that the designer incrementally adds to the assembly, ensuring it remains *connected* throughout the process. Therefore, the extension point – whether as input in variant I or as additional target value in variant II – always equals to a part of the existing assembly \mathcal{A} , i.e., $n \in \mathcal{V}_{\mathcal{A}}$. We likewise want to pursue an *automated, data-driven* approach and develop an *interactive recommendation system* where designers can either accept automated suggestions (in whole or in part) or input their own choices, enhancing flexibility and user control. This form of recommendation is similar to the word suggestions on a smartphone when typing a text: The user always gets suggestions for next words (i.e., the previous word serves as extension point). The suggestions can speed up the design process for experienced designers and also indirectly reveal implicit design knowledge to young designers.

The problem of localized part recommendation differs from traditional graph machine learning tasks such as link prediction and purely generative tasks. Compared to the former, that deals with predicting whether two existing nodes should be connected by an edge, we here need to predict a new node in addition to an edge. Graph generation mainly focuses on macro-level statistical properties such as connectivity. In our variants, however, the focus is on the individual intermediate steps in the creation of an assembly rather than on a similar graph structure. Consequently, a novel approach is required that integrates the prediction of part types along with their connection to the existing graph structure – at a specific node. Distinction from other graph learning problems will be discussed in section Section 6.4.

Both localization variants are novel, therefore we frame the problem by adaptations of upper bound and Evergreen model from Section 5.2. However, we do not consider the frequency-based baseline model further because of its consistent underperformance in comparison to all GNNs and computational complexity due to the graph comparisons and subsumption.

The attentive reader may have noticed that both variants of localized part recommendation subsume global part recommendation, which was discussed in Chapter 5. In order to receive part recommendations for an entire assembly ($P(\mathcal{T} \mid \mathcal{A})$), one could determine localized recommendations according to $P(\mathcal{T} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$ for each part n of the assembly and finally aggregate them into recommendations on a global level. To do so, a suitable aggregation strategy is needed which combines node-wise scores for each part type of the vocabulary into graph-wise scores for each part type. Several options are available for aggregation, from simple element-wise operations such as mean or maximum to parameterized aggregation functions as employed in GraphSAGE. The best strategy can be determined experimentally. Similarly, global part recommendation could also be solved by the second variant of localized recommendation: the recommendation tuples consisting of next part and extension node are predicted, whereby the localization information is simply

neglected. However, these investigations are beyond the scope of this thesis, and we will evaluate the models designed for both localization variants on the respective localization task only.

6.1 Generating Instances From Assemblies

For both localization variants, we stick to instances with single-label targets, meaning that the target is a *single* possible target (part type or tuple consisting of extension point and part type) instead of a set or list of all possible targets to a query input. In contrast, the predictions of the models are ranked lists. This representation meets the requirement of ranked recommendations according to which more frequently seen combinations should appear higher up due to higher scores. In an initial step, triple-instances consisting of partial assembly, extension point and part type are created, which are transformed depending on the use case variant: for the first variant, the assembly and extension point form the input, while in the second variant, the extension point and part type are grouped together for the target.

The triple-instances can be obtained by slightly adapting the generation procedure of Algorithm 1: An assembly \mathcal{A}^* is recursively decomposed into triple-instances consisting of a partial graph, an extension point, and an expected part type, i.e., $\langle \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}}, \tau \in \mathcal{T} \rangle$. When a non-cohesive node m is removed from the graph, extension points originate from its connected nodes $\mathcal{N}(m) = \{n \in \mathcal{V}_{\mathcal{A}} \mid (n, m) \in \mathcal{E}_{\mathcal{A}}\}$. As all assemblies in our datasets are acyclic (cf. Section 4.2), all non-cohesive nodes are leaf nodes. These nodes have exactly one neighbor, so we get exactly one triple-instance for each removed node m . In cyclic graphs, however, a removed non-cohesive node can have multiple neighbors which results in multiple instances being created when removing m , one for each neighboring node. The target part type τ is always the part type of the cut-off node m , i.e., $\tau = T(m)$. Duplicates of the triple instances that originate from the same assembly are discarded. The adopted instance generation algorithm is described by Algorithm 2, while Figure 6.1 illustrates the resulting localization triple-instances resulting from an example assembly. In order to obtain triple-instances for an entire assembly dataset $\{\mathcal{A}_i^*\}_{i=1}^n$ consisting of n assemblies, the multiset $\mathcal{D} = \biguplus_{i=1}^n \mathcal{D}_i$ represents the union of all assembly-specific datasets \mathcal{D}_i , while allowing for duplicates.

Algorithm 2 Assembly Decomposition for Localized Recommendation Instances

```

1: procedure DECOMPOSEGRAPHLOC( $\mathcal{A} = (\mathcal{V}, \mathcal{E}, T)$ ,  $\mathcal{D}$ )
2:    $seenGraphs \leftarrow \{\mathcal{A}\}$ 
3:   if  $|\mathcal{V}| > \text{min\_size}$  then ▷ domain-specific hyperparameter
4:     for every non-cohesive node  $m \in \mathcal{V}$  do
5:       for every neighbor  $n \in \mathcal{N}(m)$  do
6:          $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle \mathcal{A} \setminus \{m\}, n, T(m) \rangle\}$  ▷ localization included in instance
7:         if  $\mathcal{A} \setminus \{m\} \notin seenGraphs$  then ▷ isomorphism test
8:           DECOMPOSEGRAPHLOC( $\mathcal{A} \setminus \{m\}$ ,  $\mathcal{D}$ )
9:            $seenGraphs \leftarrow seenGraphs \cup \{\mathcal{A} \setminus \{m\}\}$ 
10:  return  $\mathcal{D}$ 

```

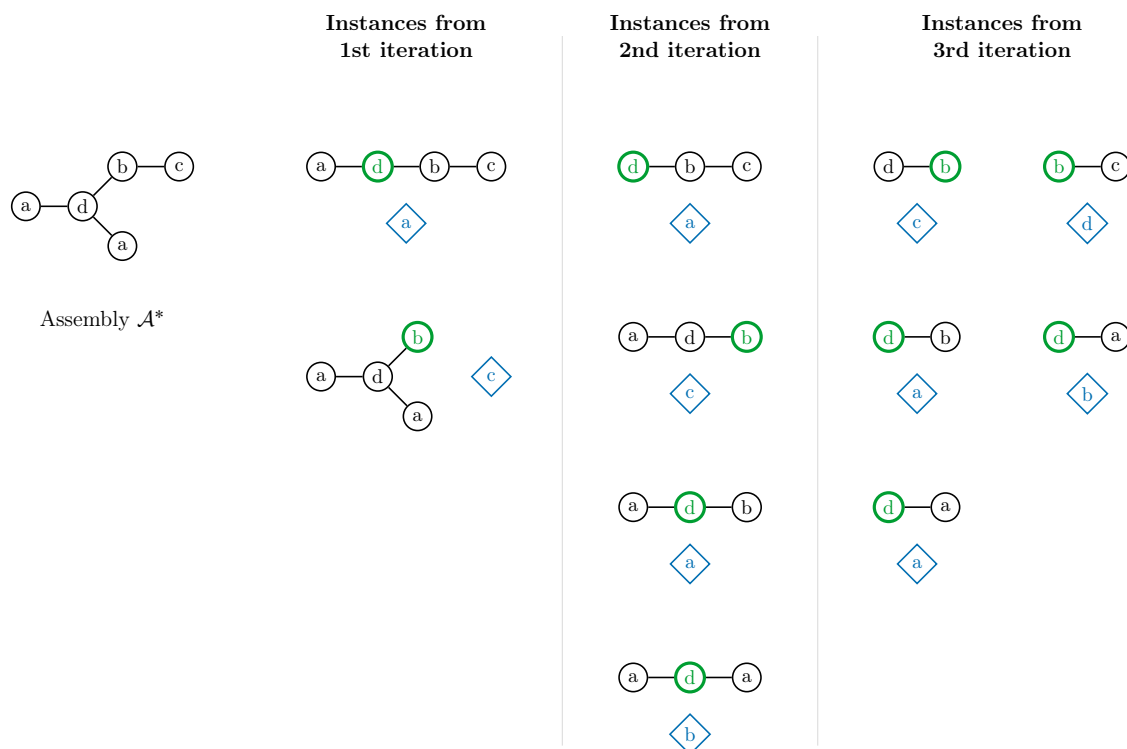


Figure 6.1: Visualization of resulting localized recommendation instances for deconstruction of example assembly graph until graph `min_size 2`. Duplicate instances are already discarded. The instances are composed of partial assembly graph, extension node (highlighted in green with thick border) and part type (illustrated as blue rhombus).

The assemblies per dataset divided into fixed training, validation and test sets (cf. Section 4.2) were converted into localization instances according to the procedure explained above. Since only additional information was added to the localization instances in comparison to the former recommendation instances, the number of instances per data set remained almost the same as for global part recommendation displayed in Table 5.1 and are therefore not explicitly listed once again.

Analogous to global part recommendation (cf. Section 5.4), the performance values of the localization models in the quantitative evaluation on the test instances may be below their actual performance in practice since the knowledge of suitable part types for a given assembly (and extension node) is incomplete and the labels of the same inputs can differ from training to test instances. Since we specifically replay the assemblies from the test data, it can happen that part types for which the model was trained and actually suitable part types that are suggested due to generalization, are not counted as correct in the test. These are common issues with a real-world dataset. The effects of this are not attempted to be estimated in the experiments.

6.2 Variant I: Recommending Part Types for User-Given Extension Point

First, we will address the first localization variant, in which we want to receive part recommendations for a given assembly and extension point, which can be added directly to the extension point. Compared to the former recommendation problem addressed in Chapter 5, the extension point is now additionally included in the input. Thus, we aim to train a joint discriminative model $P(\mathcal{T} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$. Since the predictions should refer to a specific node instead of the entire graph, we approach the learning task as a *node classification* problem, where the objective is to predict next part types for each node, with each part type representing a distinct class. To get recommendations for a specified extension point during inference, we extract the scores of the part types associated with the corresponding graph node, as depicted in Figure 6.2. In other words, we filter the node-wise part type predictions on the given extension node. The corresponding part types are then ranked based on their scores, from which we choose the desired number of suggestions.

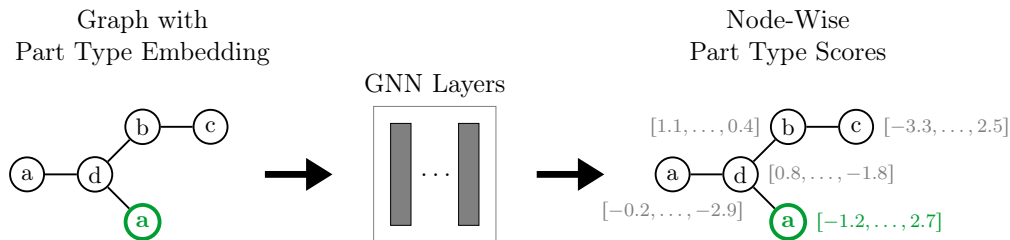


Figure 6.2: Model architecture for localized part prediction, variant I: The task is modelled as a node classification task where the input consists of an assembly and desired extension point (highlighted in green with thick border). The prediction model performs node-wise predictions of suitable part types by assigning a score to each part type $\tau \in \mathcal{T}$ (as displayed in gray). The unnormalized distribution for the given extension node is selected, from which the desired number k of part type recommendations according to the highest scores is retrieved. For the given extension node, for example, type **a** is assigned a score of -1.2 while type **d** is assigned a higher score of 2.7 for the next part. As a result, **d** would be suggested in the first position. Regularization techniques applied are not illustrated.

Both localization variants are novel, therefore we frame the problems by adaptations of upper bound and Evergreen (Section 5.2).

Upper Bound The purpose of this model is to give an estimate from above of the possible performance of a deterministic model for a given test set. Since the input-target mapping is not unique, a model that provides fewer recommendations than the number of different target part types for a graph and extension point can never achieve 100% accuracy. The upper bound model outputs the best possible ranking of part type suggestions for the given input based on their frequency as targets of the input in the evaluation data – the most frequently occurring target type for an input tuple becomes the first suggestion, followed by the second most frequently occurring type, and so on. The sequence is deterministic for an input, i.e., the model does not behave like an oracle that always puts the true target type first.

Evergreen4EP The simple model should give insight in the difficulty of the learning problem as well as help to assess whether the target part types indeed depend on the current assembly. Therefore, we consider a tailored variant of the Evergreen model from Section 5.2 called “Evergreen4EP”, which also ignores the graph structure, but additionally takes the *extension point* $n \in \mathcal{V}_{\mathcal{A}}$ into account. Formally, it represents the conditional model $P(\mathcal{T} \mid n \in \mathcal{V}_{\mathcal{A}}) = P(\mathcal{T} \mid T(n))$, predicting the most frequent part types assembled to the part corresponding to the extension node.

6.2.1 Experimental Setup

For the experiment setup, we mainly stick to that of the previous use case. To test the applicability of our approach, we evaluate it on the three assembly datasets. The assemblies separated in training, validation and test assemblies were recursively decomposed into localization instances following Algorithm 2, resulting in training, validation and test instances. We adhere to findings of the previous experiments and use the pretrained 100-dimensional part type embedding as this greatly reduces the features and thus number of parameters of the neural networks. Consequently, we omit the designation of the embedding in the model name. Cross-entropy was again used as loss function for multi-class classification.

GNN Architectures We again use the four flavors of GNNs (GAT, GCN, GIN and GraphSAGE) with same hyperparameters, although the latter three were inferior to GAT in the global part recommendation use case for all datasets. The current task is markedly different, so we want to examine all architectures. The GIN architecture is designed to better distinguish certain graph structures from each other, which is all the more important for localized part recommendation: Consider the introductory assembly example from Figure 4.1: A total of two insert holders can be attached to the boring tool. If only one is attached so far, a second one should be suggested; however, if both are already attached, the predictions should rather focus on other parts, such as useful extensions. GINs can by design distinguish the case of one and two insert holders in the assembly. Again, we skip the neighborhood sampling step of the GraphSAGE architecture due to our small assembly graphs. Preliminary experiments for global part recommendation revealed that sampling leads to poorer performance, probably because crucial information is lost through sampling.

Early Stopping in Training For training, we utilized early stopping as the criterion to halt training. By monitoring the loss values and performance metrics for both the training and validation sets over time, we observed a notable phenomenon: even as the validation loss began to increase (indicating overfitting), the *performance* on both the validation and training sets continued to improve. This discrepancy appears to stem from the imperfect correlation between the loss and the performance measure. Specifically, while cross-entropy measures the difference between the real-valued softmaxed score $\hat{y} \in [0, 1]$ and the binary target value $y \in \{0, 1\}$, the hit rate compares the binarized prediction $\tilde{y} \in \{0, 1\}$ with the binary target value y . Furthermore, the logarithmic nature of cross-entropy results in heavier weighting of incorrect “probability values” for classes. Consequently, we decided to apply early stopping based on the hit rates to get a more exact stopping criterion. The evaluation of performance metrics typically is more computationally intensive, as

they are computed for single instances instead of instance batches. Thus, this decision increased training time, specifically, evaluation time of training and validation error for early stopping. However, this was still entirely acceptable for this variant.

Performance Metrics The primary evaluation metric remains the top- k hit rate, which measures the percentage of instances the target appears within a model’s top- k predictions, to evaluate the part recommendations for given assembly and extension nodes. It assesses the accuracy of the prediction as well as the effectiveness of the ranking. Continuing with the idea of predicting a context-specific number of recommendations as explored in Section 5.5.3, we measure as a second evaluation metric the overlap of the target parts for a partial assembly graph with the same number of recommendations of the respective model. To do this, we aggregate our instances into multi-label instances, i.e., a partial assembly is mapped to all labels in the respective data set. As we have discussed in that section, the calculation of precision is equivalent to that of recall, since the size of labels and outputs is equal. We therefore limit our discussion to precision. It is only for the Evergreen4P model that equality of size cannot be guaranteed, as there may be fewer labels for a partial assembly in the training data than the number of labels in the test data. However, the values between precision and recall differ only minimally, so we will limit ourselves to precision. The aggregation of the precision values is carried out per instance.

6.2.2 Experimental Results

Compared to the initial graph classification problem addressed in Chapter 5, we have half as many instances with the same input, resulting in fewer correct target part types. This implies that higher hit rates can be expected for a smaller number of recommendations. We compare the proposed GNN-based approach with the upper bound of the best achievable performance and our simple model, Evergreen4EP. Table 6.1 presents an overview of the models’ performance as well as the comparison models for the top- k rate, which are also visualized in Figure 6.3. In addition, the simple Evergreen model is listed, which disregards the entire input (both graph and extension nodes) and thus represents a frequency distribution of the target part types. The very strong performance difference to the Evergreen4EP model – which maps the given the extension point to a distribution over the part types – across all datasets clearly shows that the connection of parts in assemblies is subject to certain rules and that not all parts can be meaningfully connected to each other.

Datasets A and B show a different picture than dataset C in several aspects: While the bound models for the first two datasets open up a fairly wide corridor that gradually narrows as the number of recommendations increases, this corridor is significantly narrower for the third dataset, with a margin of 8 to 28 percentage points. For this reason, the GNNs and lower bound are also closer to each other. The high performance values of Evergreen4EP (almost 70% with $k = 3$ recommendations) suggest that only a few meaningful part combinations are possible with the third dataset. Furthermore, for the first two datasets, there is a significant performance disparity between GATs and GCNs. Specifically, the top- k rate for a single suggestion ($k = 1$) shows an eight percentage point difference, respectively, which diminishes progressively with larger numbers of recommendations. This substantial performance difference is likely attributable to the attention mechanism in GATs, which allows for more precise weighting of individual existing parts

Table 6.1: Summary of results for localization variant I: part recommendation for given assembly graph and extension point. Evaluation of the top- k rate for $k = 1, \dots, 15$ recommendations and the recommendation precision on the test set. All values are given in percent. Best model per dataset and number of recommendations (k) is highlighted.

Dataset	Model \ k	1	2	3	5	10	15	Precision
A	Upper Bound	89.5	97.3	98.8	99.6	99.9	100.0	100.0
	GraphSAGE	70.1	82.1	86.3	90.2	93.8	95.1	75.7
	GIN	68.2	79.7	83.6	87.3	90.7	92.2	73.8
	GAT	58.0	74.0	80.2	85.7	90.9	93.2	63.4
	GCN	50.3	67.9	75.0	82.0	88.2	90.5	54.3
	Evergreen4EP	36.2	49.9	58.7	69.3	83.2	87.7	37.2
	Evergreen	2.3	4.5	6.3	9.7	15.0	19.0	4.6
B	Upper Bound	77.8	95.4	98.3	99.4	99.9	100.0	100.0
	GraphSAGE	53.2	69.6	75.6	81.6	87.5	90.1	64.2
	GIN	52.2	68.1	73.8	79.4	85.6	88.4	62.4
	GAT	50.7	66.7	72.9	79.4	86.2	89.2	61.0
	GCN	42.0	61.1	68.4	75.8	83.5	86.7	49.4
	Evergreen4EP	33.1	48.9	56.9	66.0	77.1	81.4	38.4
	Evergreen	9.4	13.3	15.5	20.3	28.4	35.3	15.7
C	Upper Bound	64.0	84.9	94.2	99.3	99.9	100.0	100.0
	GraphSAGE	50.0	70.5	82.4	92.0	97.5	98.4	79.0
	GIN	54.5	73.8	83.3	90.9	96.2	97.6	82.0
	GAT	42.0	63.5	75.5	86.6	94.8	96.9	62.0
	GCN	41.3	63.3	75.1	85.5	94.3	96.6	59.1
	Evergreen4EP	36.7	56.9	69.4	81.7	90.9	92.3	54.2
	Evergreen	7.8	14.4	19.9	25.5	35.7	43.7	18.4

in the assembly relevant to predicting subsequent parts at the extension point. In contrast, for global contextual predictions, all nodes must be considered to generate suggestions for the entire graph. Notably, only in the case of dataset C do the two models exhibit similar performance levels.

As our investigated GNN models outperform the Evergreen4EP for all numbers of recommendations and datasets, they demonstrate that they are capable of incorporating the whole graph context and that this is beneficial for the task. The most notable difference compared to the non-localized part recommendation is the clear dominance of the GIN and GraphSAGE models over GAT and GCN. For datasets A and B, GraphSAGE consistently outperforms all other GNN models across all selected recommendation counts, while for the third dataset, the GIN is ahead by up to 3 recommendations. The best architecture (GraphSAGE) achieves 81–92% accuracy when suggesting five recommendations for part types, depending on the chosen dataset.

Comparing these results with the achieved performance for the first use case shows that they are at the same level: As already mentioned, we have half as many targets for a specific input for this task as for the first use case. If we now compare the performance achieved in the first use case of between 82% and 92% for $k = 10$ predictions with the hit rates achieved in this task with half as many recommendations (i.e., $k = 5$ recommendations), which is the range of 81% to 92%, we can see that we were able to solve both problems with (almost) the same accuracy. Consequently, the inclusion of localization information did not lead to any loss in performance.

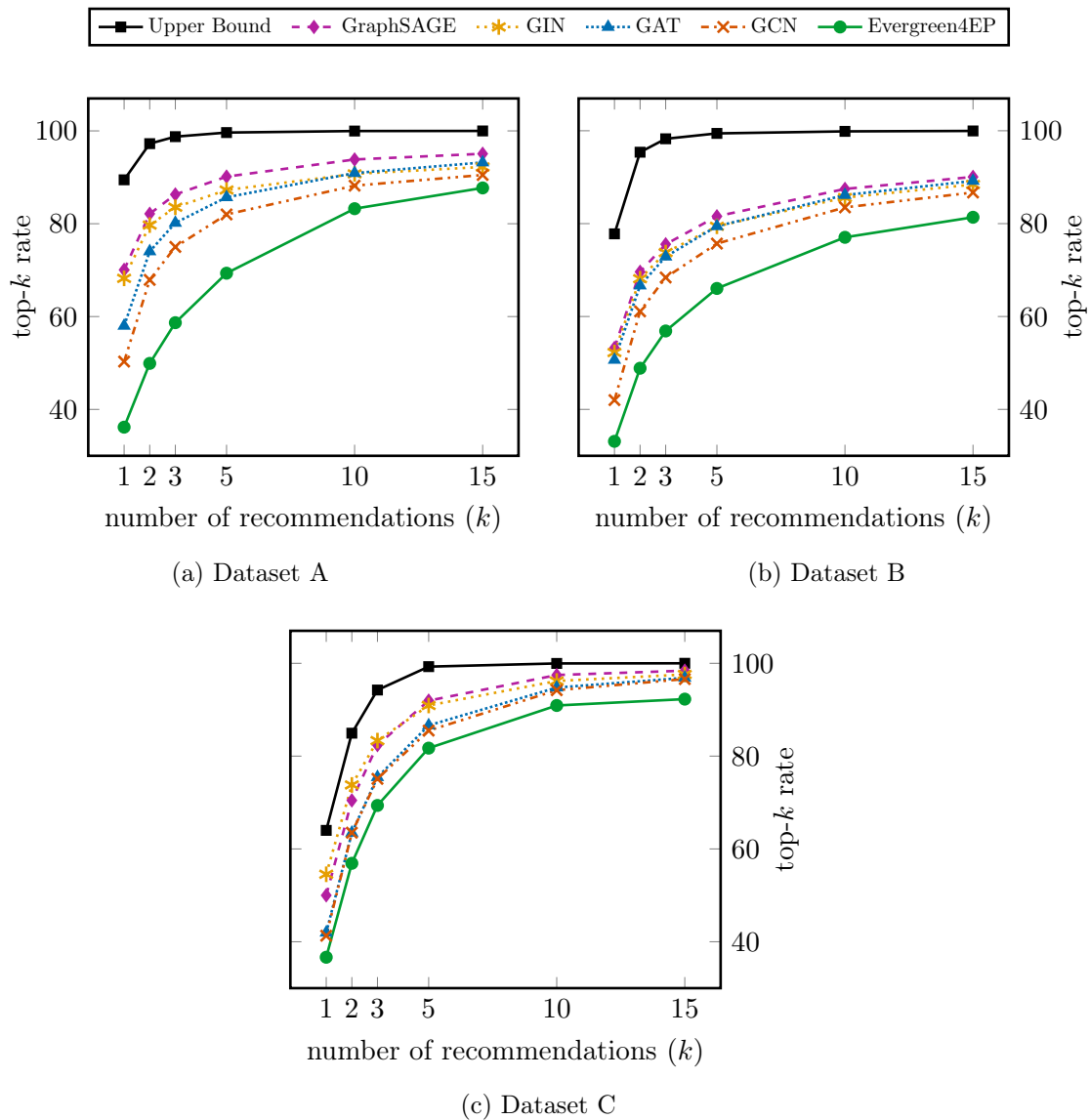


Figure 6.3: Visual comparison of the part prediction models for given localization information (variant I). Evergreen is omitted due to scaling.

The results of our second performance metric, precision, show a similar picture: GraphSAGE is ahead for the first two datasets, while GIN performs best for dataset C. The precision of the best models is 75% for dataset A and 82% for dataset C. The slightly lower precision of the second dataset of 64% is attributable to its higher ambiguity due to the larger number of part types (3099 as opposed to 1930 (A) and 1924 (C)).

A main difference in the experimental setup compared to global part recommendation is that the termination condition of early stopping is based on the performance measure instead of the loss values. Since the evaluation of the top- k rate is more computationally expensive than the cross-entropy loss, the training time for one epoch on dataset B increased to about 22 minutes on an Nvidia DGX-1. The majority of this time was needed to calculate the performance on the training and validation time. Furthermore, almost all

models could be trained in a maximum of 20 epochs. The best GraphSAGE model for dataset B reached the optimal parameters after only four epochs, hence the training was completed after one and a half hours. The inference time is identical to the global recommendation models on the same dataset; more than 1000 test instances could be evaluated on the top- k rate within one second, the pure inference time is therefore even lower.

6.3 Variant II: Recommending Extension Point and Part Type

The second variant of localized part recommendation for assembly modeling aims at giving recommendations of part types next needed together with their placement in the current assembly. Therefore, we need a *multi-task model*. However, the two tasks must be modeled differently: While the part type prediction can be modeled as a classification of a fixed size depending on the number of part types in the vocabulary and completely independent of the graph structure, the prediction of the extension point depends on the assembly graph, precisely on the number of its nodes. Consequently, part type prediction is a graph classification task, while extension point prediction is a node-based task.

For modeling the prediction of the extension node, it may also seem natural to carry out a graph-based classification of fixed dimension, with the possible classes arising from the graph sizes that occur. Then the prediction of class i would mean that the i th node of the graph is the predicted extension node. However, this modeling has some disadvantages and causes problems in handling:

1. The class for the first extension node occurs most frequently, since every non-empty graph has at least one node. Accordingly, higher node indices occur less frequently, as there are fewer graphs with this minimum size. The larger the minimum size of the graphs, the fewer matching instances there are. This causes bias to small node indices and keeps the model from predicting higher indices.
2. It could be the case that the predicted node index is higher than the assembly graph's size which needs to be treated as a specific issue.
3. One would need a fixed, deterministic numbering of the nodes so that the same graph (isomorphic) and the same extension point are always represented in the same way, i.e., the target is then the same. However, the problem is that there is no canonical order of nodes and edges in graphs, especially when multiple instances of the same type may occur. The model would then also have to learn this numbering.

For these reasons, we instead predict a real-valued “extension score” per node where the node with the highest score is chosen as extension node. As our considered assemblies are acyclic graphs, every cut-off non-cohesive node has exactly one neighbor and thus one extension node. We investigate two opposing two-step modeling approaches to address this variant which differ in the order of subtasks dealt with:

- (1) In the first modeling approach of variant II, a first model predicts part types for the given assembly representing $P(\mathcal{T} \mid \mathcal{A})$, then a second model suggests a placement conditioned on this part type following $P(\mathcal{V}_{\mathcal{A}} \mid \mathcal{A}, \tau \in \mathcal{T})$.
- (2) The second modeling approach follows the opposite approach: Initially, a promising extension point is predicted based on $P(\mathcal{V}_{\mathcal{A}} \mid \mathcal{A})$. Then, specific part types are suggested for the extension point, modeled by $P(\mathcal{T} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$.

For each modeling approach, we combine two separate GNNs in different ways, each of which takes care of one aspect of the recommendation. When comparing these two approaches, we are going to investigate the overall model predicting a new part along with its extension point as well as the two submodels individually. The first models process the actual input and can thus be evaluated independently of previous models, i.e., *in isolation*. To independently assess the subsequent models, we use the target values instead of the outputs of the previous models as inputs for the subsequent models. This means that the subsequent models' predictions are determined *under the condition* that the preceding model's prediction is correct.

Adaptions of Upper Bound and Simple Baseline

Again we aim to frame the performance of the overall models as well as the respective submodels for the second localization variant. Therefore, we adjust the upper bound and simple baseline Evergreen originally introduced in Section 5.2. The models only need to be adapted slightly for the overall task: Instead of outputting one part type at a time, the target tuple consisting of part type and extension node must now be returned. The basic procedure however remains the same for each model.

In the case of the submodels predicting the part types, however, we must differentiate based on the specific task of the respective modeling approach: In the first approach, a prediction for part types is generated from the assembly only ($P(\mathcal{T} | \mathcal{A})$). The attentive reader may already have noticed, that this is essentially the setting of global part recommendation. Consequently, we employ the original Evergreen model as simple baseline and the original upper bound model of Section 5.2 for this subtask. For the second approach, however, the part type prediction model is given both an assembly and an extension point. Since this task is essentially the problem addressed in the first localization variant, we apply the modified framing models, Evergreen4EP and the adjusted upper bound incorporating the extension node that were presented in Section 6.2.

In the following, we are going into the specific modeling of the two approaches in detail.

6.3.1 Modeling Approach (1): First Predicting Part Type, Afterward Extension Point

In this modeling approach, at first a part type is predicted for a partial assembly, which is then located on the assembly in a second step. The first task essentially is global part recommendation addressed in Chapter 5. Consequently, we employ the same model architecture for the first submodel. The second submodel, on the other side, handles the placement of the predicted part. As the information of the specific type is relevant for the placement of the part on the graph, we need to include it into the graph somehow. Instead of predicting new connections *within* the existing graph, as in conventional link prediction, the goal is to predict the connection of the new part *to* the existing graph. To achieve this, a new node representing the previously predicted part is inserted, along with edges from this node to all existing nodes in the graph. For each newly inserted edge, a shared GNN-based model called *edge score predictor* is trained to predict a scalar score based on the features of the two corresponding nodes. This allows us to determine the most promising edge from the new node to the existing graph and thus the most promising extension node.

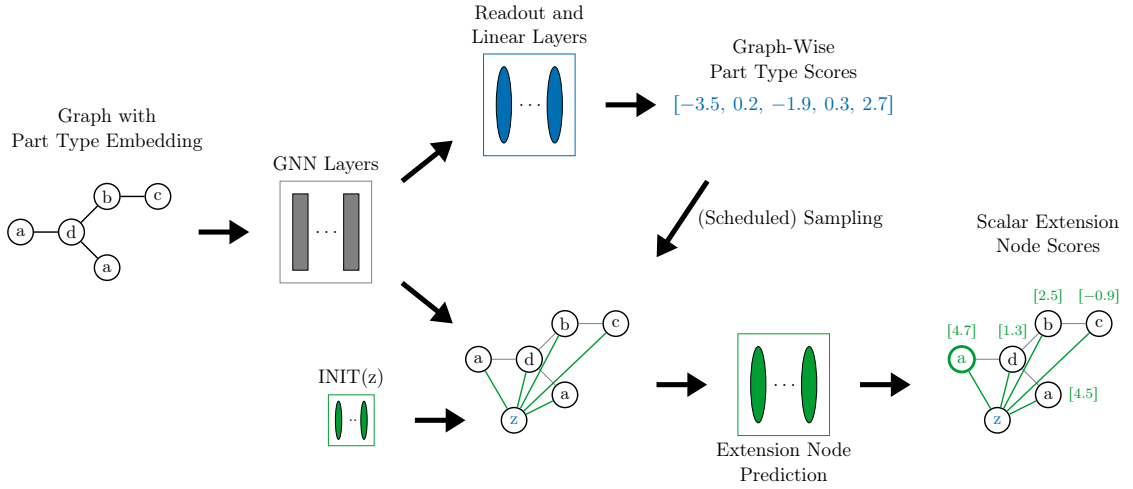


Figure 6.4: High-level view of the model architecture for modeling approach (1): An assembly graph over part types $\mathcal{T} = \{a, b, c, d, z\}$ is initially enriched with part type embeddings. A first model processes the graph context and predicts next parts to add to the assembly (represented by the blue model). In the example shown, part type z achieves the highest score with a value of 2.7. A second model (green) suggests the placement conditioned on the predicted part type. It operates on the shared hidden representation of the first model (depicted in gray) for the pre-existing parts and a suitable initialization as representation for the sampled part type based on its embedding. The initialization model (INIT) is also a trainable MLP. Overall, the new part of type z is connected to the highlighted node with the highest extension score of 4.7.

In our setting, both the individual parts and their combination within the assembly (i.e., the overall context) are relevant for determining the extension point. Therefore, the overall context should be considered in the edge scoring of the second model. The graph classification model used to predict the part type for a partial assembly, a GNN, has already captured the context of the assembly through message passing. It seems that both task models share common ground in processing the assembly context, so it is reasonable that they could profit from each other’s insights by sharing an initial feature extractor. Consequently, the edge scores are computed based on the nodes’ hidden states in the GNN prior to the readout and classification layers, rather than the initial node features, i.e., the embeddings. In other words, we have an initial shared GNN for processing the graph context, like a shared feature extractor. Its resulting hidden representations are then fed into the two subsequent task-specific models.

However, since the new part was not included in the graph in the first step, it lacks a hidden representation. Therefore, an initialization MLP learns a corresponding representation from the given part type embedding. Together, the edge score predictor and the initialization model form the second submodel. The overall model architecture for modeling approach (1) is depicted in Figure 6.4.

Training Adjustments

The two submodels work together, with the second model processing the part proposed by the first model as input and both using a shared hidden representation. Initially, during

training, the first model may perform poorly, resulting in the subsequent model receiving incorrect predictions for the part types. Thus, the second model must initially process incorrect or highly fluctuating inputs. This problem is similar to those encountered in sequence processing models, such as RNNs, where the prediction of a token is used as input for the next token. A well-known solution is *teacher forcing*, a training technique where the output of the previous model is replaced with the actual target value during training before it is given to the next model [51]. However, if the model is always given the teacher-provided inputs during training, it may struggle during inference without access to the true inputs as it must fully rely on the outputs of the previous model – a phenomenon called *exposure bias* [51]. *Scheduled sampling* [11] was developed to mitigate the exposure bias introduced by teacher forcing: The model is gradually given more and more predictions of the previous model during training, in order to make it more robust to imperfect inputs. We employ this technique for the second model, where the actual part type is fed into the second model instead of the prediction of the first model with a decreasing probability over time. We want to point out that scheduled sampling is only used during the training aiming to make the second submodel more robust, but not during the inference for the evaluation or in later use.

Alternatively, the submodels can be trained independently. In this case, they cannot share the initial feature extractor model, as only one model could adapt its weights during training. Thus, each model must first process the graph context before predicting part types or calculating edge scores, respectively, which requires more parameters to learn. This approach makes training more manageable, but the overall model often suffers from exposure bias during inference: if the second model was always fed with correct inputs during training, it has not learned to deal with any imperfect predictions made by the previous model. Both approaches are examined in the experiments.

Recommendations during Inference

The generation of recommendations during inference for the second localization variant is more complex due to the two subtasks. We aim to determine a probability distribution over tuples of part type and extension node by multiplying the probabilities of the individual elements from the two submodels based on the definition of conditional probability [51]:

$$P(\mathcal{T}, \mathcal{V}_A | \mathcal{A}) = P(\mathcal{T} | \mathcal{A}) \cdot P(\mathcal{V}_A | \mathcal{A}, \tau \in \mathcal{T}) \quad (6.1)$$

In the case of the first modeling approach, part types are first proposed for a given assembly and their scores are transformed into a probability distribution $P(\mathcal{T} | \mathcal{A})$ over all part types using Softmax. The top part types are selected based on this distribution. For each part type, its corresponding embedding is converted into a suitable representation for the extension node predictor using the initialization MLP. An extension score is then determined for each part type and existing part in the assembly graph. The extension scores of all nodes for a part type τ are also converted into a probability distribution $P(\mathcal{V}_A | \mathcal{A}, \tau \in \mathcal{T})$ across all nodes. By multiplying the two probability distributions, we obtain a joint distribution over the desired tuples of part type and extension nodes, from which the desired number k of top tuples can finally be selected. In particular, as the prediction of the extension node is computationally intensive when performed for all part types of the vocabulary, we restrict the prediction to only the top $2k$ part types of the part type recommendation model, when k recommendation tuples (consisting of part type and extension node) are desired.

6.3.2 Modeling Approach (2): First Predicting Extension Point, Afterward Part Type

The second modeling approach proceeds in the opposite way: The first model predicts promising extension points according to $P(\mathcal{V}_A | \mathcal{A})$ before specific part types are proposed for it in the second step, following $P(\mathcal{T} | \mathcal{A}, n \in \mathcal{V}_A)$. The second task aligns with the first localization variant, so we build on that model: the graph structure is processed with message passing, afterward a score for each part type $\tau \in \mathcal{T}$ is determined for each node (cf. Figure 6.2). The prediction of an extension score per node is achieved by predicting a real-valued scalar in the last GNN layer. As discussed for the first modeling approach, predicting new parts also depends on the combination of existing parts rather than considering them in isolation, so we again use an initial shared feature extractor and feed its common hidden representations into the subsequent task-specific submodels, as depicted in Figure 6.5. The overall model consists of shared GNN layers and two distinct output components responsible for node scoring and part prediction (which is analogous to variant I), respectively.

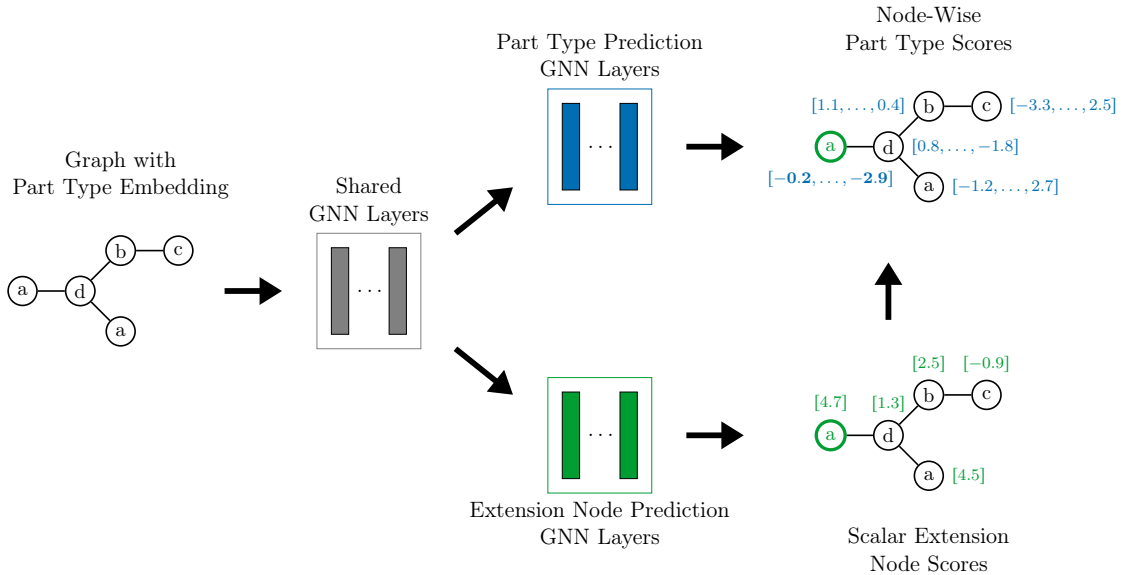


Figure 6.5: High-level view of the model architecture for the second modeling approach: An assembly graph over part types $\mathcal{T} = \{a, b, c, d, z\}$ is initially enriched with part type embeddings. A shared model processes the graph context before two output models (colored in green and blue, respectively) predict extension scores and part type score for each node, respectively. In the example shown, the leftmost node achieves the highest extension score of 4.7; for the same node, the first part type (a) achieves the highest score of -0.2 for the part type recommendation. The combination of both outputs forms the recommendation.

We want to point out that the second submodel outputs part type predictions for all nodes of the assembly graph and that we select the part type scores for the extension node predicted by the first submodel in order to get a ranking of part types. Consequently, the second submodel does not process the information of the extension node predicted by the first submodel. In other words, the choice of the extension node does not affect

the node-wise part type predictions. Therefore, there is no need to make this submodel robust against incorrect inputs from the previous submodel, which we addressed in the first modeling approach with scheduled sampling. Thus, we only examine training the overall model as a whole.

Recommendations during Inference Analogous to the first modeling approach, we determine recommendations for tuples consisting of part type and extension nodes for the second modeling approach by taking the top k elements from a joint probability distribution. According to the definition of conditional probability, the joint probability distribution is the product of the distributions of the subtasks:

$$P(\mathcal{T}, \mathcal{V}_A | \mathcal{A}) = P(\mathcal{V}_A | \mathcal{A}) \cdot P(\mathcal{T} | \mathcal{A}, n \in \mathcal{V}_A) \quad (6.2)$$

The first submodel provides extension scores per node, which can be converted into a probability distribution over all nodes using Softmax, i.e., $P(\mathcal{V}_A | \mathcal{A})$. Then, the second submodel provides a distribution over all part types per node, which is normalized to a probability distribution $P(\mathcal{T} | \mathcal{A}, n \in \mathcal{V}_A)$.

6.3.3 Experimental Setup

The experiment setup is mainly identical to the first localization variant described in Section 6.2.1. In terms of GNN architectures, we do no longer investigate GCNs for the experiments of this variant due to their inferiority for the simpler localization variant (cf. Table 6.1). Although GATs were also consistently inferior to the other two architectures, they achieved the best results for the global part predictions, which is the first submodel in modeling approach (1) of the current localization variant. For simplicity, we use the same GNN architecture for both submodels for every modeling approach, respectively. We search for the standard hyperparameters of a neural network (number of layers, dimension of the layers, activation function) separately for the respective submodels.

Loss Function The prediction of part types is, as known, a classification task, where the number of part types corresponds to the number of classes. For the extension node prediction, all scalar node scores can be summarized to a list of scores, which corresponds to a node-wise classification problem. Here, the number of classes is graph-specific, as it corresponds to the number of nodes. Summarizing, both subtasks are treated as classification problems. Therefore, we again use categorical (i.e., multi-class) cross-entropy as loss function for both subtasks. However, in order to be able to train both subtasks together, as we aim to do in the first modeling approach with teacher forcing and also the second modeling approach, we have to combine the loss functions into one for the overall task. The common solution strategy for this is a combined loss function as weighted sum $L = L_{PT} + \lambda \cdot L_{EP}$ of both loss components (L_{PT} for part type prediction and L_{EP} for extension node prediction), similar to adding regularization terms to a loss function [51]. The loss weight λ is a hyperparameter that needs to be set in advance. Depending on which subtask is to be assigned more weight, this weight can be selected accordingly. We decided to weight both tasks equally, so that we chose λ as a multiple of 10, so that both loss components were in a similar value range. In the experiments, this value was set to 10^{-3} or 10^{-4} , depending on the dataset.

Performance Metrics In the second localization variant, we are addressing a multi-task problem, so we are examining the performance of the overall model (predicting a tuple of part type and extension node) as well as the respective submodels individually. The top- k hit rate remains the main evaluation metric, used to assess the performance of the part type prediction submodels as well as the overall model. It evaluates both the correctness and ranking of the prediction. Assessing the performance of extension node prediction is more complex due to its dependence on the graph size. Identifying the correct extension point in a large graph is significantly more challenging than in a small graph, as there are more options to choose from. Therefore, successfully predicting the correct node in a larger graph is more impactful and should be rewarded more. Our goal is to evaluate the ranking relatively: the first position with three nodes should be rated the same as the first five position with 15 nodes. To achieve this, we include the graph size in the reciprocal rank metric [38], which leads to the *mean weighted reciprocal rank (MWRR)* averaged over all triple-instances in a data set \mathcal{D} as:

$$\text{MWRR} = \frac{1}{|\mathcal{D}|} \sum_{\langle \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}}, \tau \in \mathcal{T} \rangle \in \mathcal{D}} \frac{|\mathcal{V}_{\mathcal{A}}|}{\text{rank}(n)} \quad (6.3)$$

Due to the adjustment of the numerator, the maximum value of this metric is no longer 1, but the graph size $|\mathcal{V}_{\mathcal{A}}|$.

Due to the discrepancy between loss function (cross-entropy) and performance measure (top- k rate) observed in the first localization variant, we apply early stopping on the performance measures. In this variant, the resulting increase in training time is much more noticeable. The reason for this is the sequential nature of the modeling so that, for example, the first modeling approach requires extension node predictions to be generated for several initially proposed part types. In order to keep the training time manageable, the metrics were only evaluated on 20% of the instances (both training and validation) during training. Furthermore, we decreased the early stopping patience to 3 epochs, as the learning curves were very smooth.

Modeling Approach (1) In the first modeling approach, which initially predicts a part type and subsequently its placement, we examine the performance of the models with both independently trained submodels and jointly trained models using scheduled sampling. Exponential decay [11] was implemented as the decay schedule for the teacher-forced sampling: Starting from an initial probability $0 < p < 1$, the teacher forcing probability ϵ_i in a training epoch i is given as $\epsilon_i = p^i$. Over the epochs, the extension node predictor is thus exposed to more and more predictions of the part type predictor instead of the actual target values. The initial probability is selected so that the teacher forcing probability is close to zero when training of the overall model is stopped by early stopping, which was achieved by $p = 0.95$ and $p = 0.97$ in our case. Note that scheduled sampling was only employed during training, but not when evaluating.

Since the first subtask corresponds to the global part recommendation and the associated training instances differed only slightly, we used the corresponding trained model of the first use case as the first submodel. The second submodel for the extension node prediction was then trained in isolation. When using scheduled sampling, however, both models were trained from scratch.

Modeling Approach (2) The second modeling approach simultaneously predicts a suitable extension point and the suitable part types for each existing part of the assembly. Both predictions are combined to form the overall prediction via a tuple of part type and extension point. The two task models share a common feature extractor, on whose hidden representation the two prediction models are then based. In contrast to the first modeling approach, there are no further hyperparameters with regard to the training process. In order to train the model as a whole, a loss weight controls the weighting of the two models during training.

6.3.4 Experimental Results

Table 6.2 presents the evaluation results for the overall task of the second localization variant for all three datasets. Additionally, Figure 6.6 displays the upper bound and simple baseline as well as GraphSAGE and GIN models for the first modeling approach; other models have been omitted for reasons of clarity. We compare the proposed GNN-based approaches with the upper bound of the optimal achievable performance on the test set and our adapted baseline model, Evergreen. The upper bound and baseline span a corridor in which most of the GNN models lie in the middle or in the upper half. As with the first localization variant, there is a clear difference in performance between the trained models and the Evergreen baseline. This means that these models are again able to process the context of the assembly appropriately in order to generate suitable extension node and part type suggestions, also on this more complex multi-task.

Using scheduled sampling to train the first modeling approach did not pay off: It usually only makes a small difference on performance, but the models trained with scheduled sampling are less successful on the overall task, consistently on all datasets. The training process was made more complex by handling two models simultaneously and harder to manage by this method, but this did not result in improvement. Nonetheless, a reduction in the required training time could be observed when applying scheduled sampling. Table 6.3 shows the performance of the individual submodels in isolation exemplary for dataset A. The second sub-model of modeling approach (1), i.e., the prediction of the extension node for a given part type, was also unable to achieve any improvement through the training adaptation. Compared to the part recommendation problem addressed in Chapter 5, which models the same problem $P(\mathcal{T} | \mathcal{A})$, the respective submodel trained independently achieves a similar performance level, confer Table 5.3. Furthermore, independently training the two submodels also outperforms the second proposed modeling approach.

The loss factor in the second modeling approach was optimized with regard to the performance of the overall task. Unfortunately, the results of this modeling approach lie behind the first modeling approach for all datasets. For the isolated evaluation given in Table 6.3, the respective second submodel was given the respective target value as input instead of the prediction of the previous submodel. The same task was addressed in the first localization variant, of which the results are given in Table 6.1. Compared to that models, the part recommendation of the second modeling approach performs worse, about 3 percentage points for every number of recommendations. Shifting the emphasis more to that prediction task by adjusting the loss weight led to a significant performance reduction in the other submodel, so that also the overall performance decreased. When comparing the two options for training both prediction models together, i.e., modeling approach (1) with scheduled sampling and modeling approach (2), the latter usually emerges as the winner. This could be attributable to the representation to be learned by the initialization

Table 6.2: Evaluation results for recommending new parts along with their placement for given assembly graph (variant II): Evaluation of the top- k rate for $k = 1, \dots, 10$ recommendations given in percent. Best model is highlighted per dataset. The same GNN architecture is used for both submodels for every modeling approach, respectively.

Dataset	Modeling Approach	Model \ k	1	2	3	5	10	
A		Upper Bound	59.4	88.4	96.1	98.3	99.3	
		Evergreen	12.7	20.7	27.0	37.3	52.9	
	(1), Independent Training	GraphSAGE	44.9	69.5	78.8	84.9	90.0	
		GIN	44.5	68.4	77.2	82.6	87.1	
		GAT	37.5	59.4	70.1	79.8	86.6	
	(1), Scheduled Sampling	GraphSAGE	42.1	62.0	71.0	79.3	86.2	
		GIN	40.2	58.3	68.4	77.4	85.1	
		GAT	35.4	49.7	60.2	72.6	82.8	
	(2)	GraphSAGE	42.0	65.9	75.5	82.0	87.6	
		GIN	37.6	58.0	67.9	76.3	84.0	
		GAT	14.1	29.0	38.5	53.8	71.1	
	B		Upper Bound	42.6	69.4	86.8	97.7	99.2
			Evergreen	12.7	21.7	29.4	40.1	54.9
		(1), Independent Training	GraphSAGE	23.5	41.1	53.9	67.2	76.9
			GIN	25.2	42.7	54.6	66.6	75.8
GAT			24.4	40.6	51.5	63.4	73.8	
(1), Scheduled Sampling		GraphSAGE	24.2	40.5	51.6	63.4	73.3	
		GIN	24.3	40.0	50.5	61.7	72.0	
		GAT	21.7	36.3	46.3	58.1	70.0	
(2)		GraphSAGE	24.9	42.2	53.9	66.3	75.7	
		GIN	24.9	41.9	53.3	64.9	74.0	
		GAT	19.4	32.9	43.2	55.7	68.4	
C			Upper Bound	35.6	60.1	78.0	95.6	99.5
			Evergreen	7.6	13.8	20.5	36.4	70.0
		(1), Independent Training	GraphSAGE	27.3	47.0	62.3	80.1	92.0
			GIN	27.1	46.8	62.1	79.9	91.8
	GAT		25.4	42.6	54.7	69.4	84.8	
	(1), Scheduled Sampling	GraphSAGE	25.5	45.0	60.2	77.3	87.3	
		GIN	24.3	42.7	56.7	73.7	85.6	
		GAT	24.6	41.6	53.8	68.2	82.4	
	(2)	GraphSAGE	18.4	33.4	46.2	65.2	85.9	
		GIN	24.8	42.9	57.7	76.2	89.4	
		GAT	23.5	39.3	50.4	63.9	79.4	

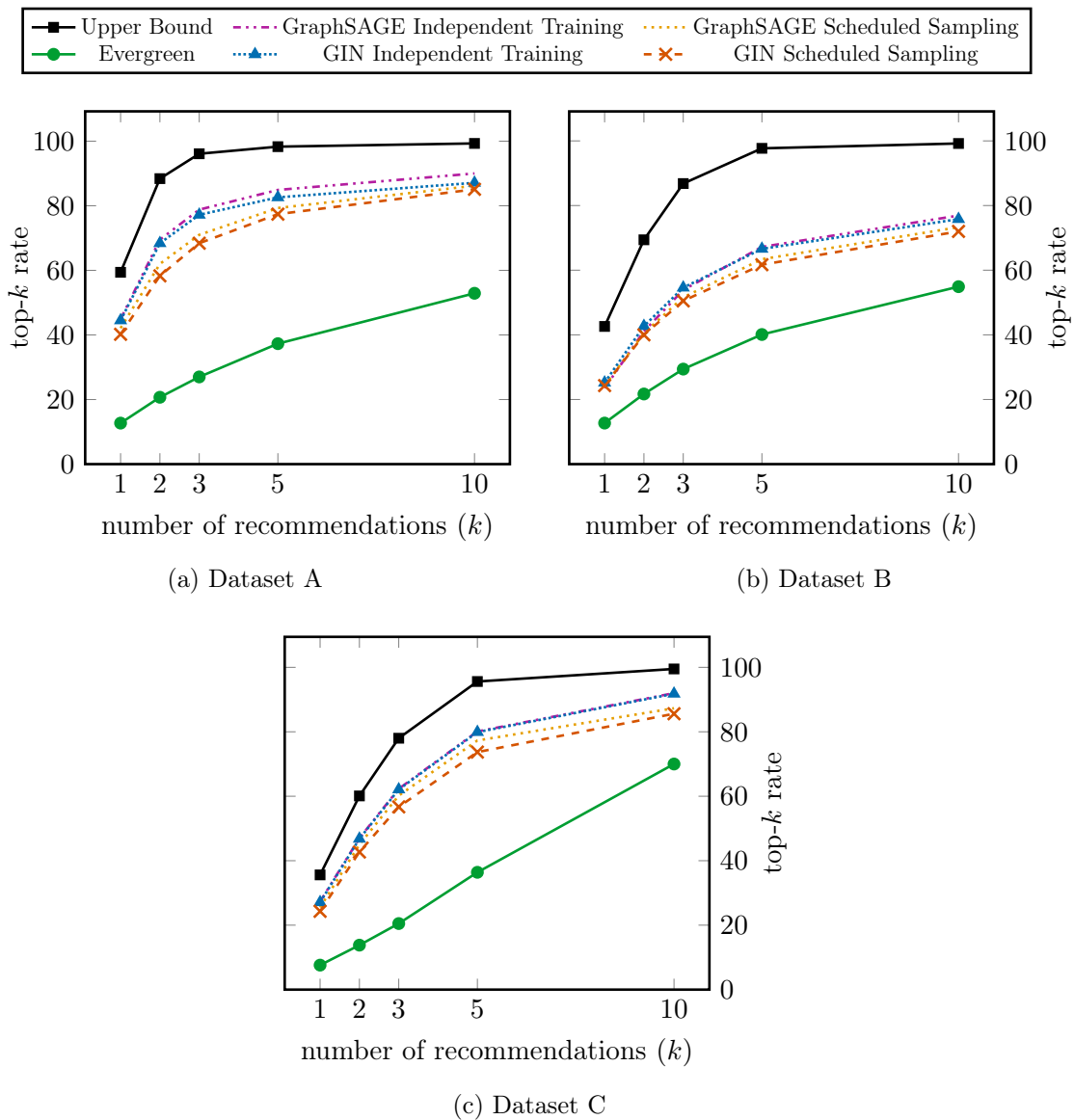


Figure 6.6: Visual comparison of the results for localized part recommendation where both a new part type and its localization to the current assembly is to be predicted (variant II). Only GraphSAGE and GIN models according to the first modeling approach are displayed.

MLP for the newly inserted part type, which has to adapt to the constantly changing hidden representations of the previous shared GNN during the training process.

For all extension node prediction models, the GIN and GraphSAGE models achieve very similar MWRR performance values, which are significantly better than those of the GATs. This phenomenon can be observed across all modeling approaches. However, the pattern is not so clear for the part recommendation as the GINs fall behind the GATs for both training types of modeling approach (1). Nevertheless, the localization submodel can compensate for this for the overall task, so that overall GIN is ahead of GAT again (see Table 6.2). For both subtasks, GraphSAGE models perform best, which supports the choice of the same GNN architecture for both submodels.

Table 6.3: Evaluation of prediction submodels of second localization variant in isolation for dataset A. The part type prediction submodels are evaluated based on the top- k rate given in percent for $k = 1, \dots, 10$ recommendations, whereas the performance of the extension node predictors is assessed by the mean weighted reciprocal rank (MWRR). As the two tasks are approached in a different order in the modeling approaches, the corresponding discriminative model is specified.

Modeling Approach	Model \ k	Part Type					Extension
		1	2	3	5	10	MWRR
		$P(\mathcal{T} \mathcal{A})$					$P(\mathcal{V}_A \mathcal{A}, \tau \in \mathcal{T})$
(1), Independent Training	GraphSAGE	47.4	71.6	80.2	86.1	90.8	4.31
	GIN	47.0	70.6	78.6	83.8	88.2	4.30
	GAT	46.7	71.4	79.7	85.2	90.0	3.97
(1), Scheduled Sampling	GraphSAGE	44.6	67.7	77.3	84.1	90.1	4.29
	GIN	43.2	66.2	76.0	83.4	89.2	4.23
	GAT	45.7	69.5	78.2	84.2	89.3	3.79
		$P(\mathcal{T} \mathcal{A}, n \in \mathcal{V}_A)$					$P(\mathcal{V}_A \mathcal{A})$
(2)	GraphSAGE	67.6	79.5	83.9	87.9	91.6	3.48
	GIN	59.3	72.7	78.1	84.0	89.1	3.48
	GAT	47.0	65.4	73.4	81.8	89.1	2.22

In summary, GraphSAGE emerges as the most successful architecture for the second localization variant, which is only slightly outperformed by other models in individual cases. This finding is consistent to the first localization variant. Furthermore, training the submodels independently for both subtasks of modeling approach (1) turned out to be the best modeling strategy across all datasets. The best models were able to achieve the multi-task prediction of both an extension node in the assembly and a matching new part type at $k = 5$ predictions for dataset A with an accuracy of 84.9%, for dataset B with 67.2% and for dataset C with 80.1%. For $k = 10$ predictions, the performance increases to 90.0%, 76.9% and 92.0%, respectively. The lower performance for dataset B is attributable to the larger number of part types (3,000 instead of 2,000) and the more complex graph structures, confer Table 4.1.

The higher complexity of the multi-task model in comparison to the first variant of localized part recommendation also affects the training and inference time of the models: For the combined trained models (i.e., modeling approach (1) with scheduled sampling and approach (2)), a training epoch for GraphSAGE increases to one and a half in the first case and even three hours in the second case. This is caused by the more complex model structure and more complex evaluation strategy of the submodels with the respective performance measures top- k rate and MWRR. However, the overall models can still be trained on a single GPU of an Nvidia DGX-1 station and optimal parameter settings were consistently found in under 15 training epochs; the GraphSAGE model for dataset B achieved this after only 9 epochs. Of course, the complexity of the task also affects the evaluation time on the test set. Nevertheless, depending on the modeling approach, between 50 and 100 instances can still be evaluated within one second. This means that the pure inference of recommendation tuples can still be carried out in a negligible amount of time.

6.4 Related Work

For the use case of localized part recommendation, we address two problems: the addition of new parts to an assembly and their connection to existing parts. Classical approaches for supporting a designer in searching parts during modeling (i.e., assembly retrieval) as well as the similarity of recommending parts in order to extend an assembly to graph generation have already been discussed in the related work for global part recommendation, confer Section 5.6.

The most similar approach to localized part recommendation in literature is [88], which presents a generative model that learns a sequence of node and edge insertions based on recurrent models. Sequencing graphs introduces the challenge of assigning node orders, which is not necessary in our GNN-based approach. In fact, it contradicts our understanding that many different creation sequences can lead to the same assembly, and all are equally valid. Additionally, while their work evaluates only the final graph, our approach emphasizes meaningful extensions at each step of the graph-building process.

Link Prediction Opposed to classical link or relation prediction, where a given set of nodes and an incomplete set of edges is given in order to predict missing or new promising edges for these nodes, we aim for connecting a new part to the assembly, i.e., adding a new node and a new edge to a graph. We adapt the idea of determining scores for pairs of unconnected nodes and calculate scores for potential edges that would connect the new part to the existing parts in order to find the most promising connection. This procedure is similar to determining the link nodes of the existing graph in [90].

Predicting Part Locations and Relations Assembly-based modeling represents a similar problem domain to ours: rather than assembling individual parts to form an assembly, this approach involves amalgamating multiple geometric 3D models to generate a novel entity [38]. For instance, consider the scenario where two distinct chairs serve as the basis: the leg of the first chair is paired with the seat and backrest of the second chair, resulting in a unique chair design. While localization in this setting refers to the geometric alignment of the individual models to each other, we extend an assembly with a new part at a specific existing part (referred to as extension node in our approach).

In addition to our work, other research also deals with the positioning of parts within an assembly, albeit for different use cases: For instance, in AutoMate [68], Jones et al. focus on predicting one of eight possible mate types (e.g., Fastened or Revolute) for the connection between two parts. Their approach assumes that the complete set of parts to be connected to an assembly is already available, with the emphasis placed on solving the interconnection between these parts. Instead of recommending parts to be connected, AutoMate relies on the designer selecting two parts for mating, as well as specifying the faces of the parts that the mate will constrain. In our use case, by contrast, a designer can voluntarily select a single part of an assembly to receive targeted part recommendations. In the second variant of the use case, the designer does not have to pre-select any parts at all, as we additionally predict the most suitable extension node. Similar to our approach, their interactive recommendation system uses GNNs, but it centers on the geometric features of parts. The graphs they process are based on the geometric faces and edges of 3D parts, rather than treating assemblies as a graph of connected parts.

Similarly, JoinABLE [150] predicts the relative geometric positions of two parts in a

CAD system (referred to as “joints” rather than “mates”) also employing GNNs. However, this approach is fully automated and does not involve user interaction. While it can predict the position between two parts without additional geometric input from the user, it is limited to two parts at a time and does not involve a retrieval processes. Our use case, by contrast, primarily deals with part recommendations, either for a given localization or the models additionally predict the location.

Neither of these approaches fully aligns with the specific needs of our use case, but they could be integrated with our system as complementary steps to determine mate types or precise positioning *after* localized part recommendations are made in order to further support a designer.

Handling Anomalies in Assemblies

Summary This chapter addresses the task of detecting anomalous parts within assemblies to identify where and how it is deviating from standard design practices. It presents an algorithm for generating synthetic anomalous assemblies by extracting regular part combinations from a dataset of regular assemblies. The chapter also explores the recommendation of alternative parts to replace identified anomalies, ensuring that assemblies meet quality standards. Experiments show that our models correctly classify over 97% of all parts as regular and anomalous and that alternative suggestions consisting of ten part type suggestions successfully correct the anomaly in between 89.7% and 93.8% of cases. This shows that a designer can rely completely on the detection of anomalous parts, and in nine out of ten cases a suitable alternative part is suggested that can be applied directly to correct the anomaly.

Contents

7.1	Generating Synthetic Anomalous Assemblies	121
7.2	Detecting Anomalous Parts in Assemblies	126
7.2.1	Experimental Setup	127
7.2.2	Experimental Results	128
7.3	Recommending Alternative Parts	130
7.3.1	Experimental Setup	133
7.3.2	Experimental Results	133
7.4	Related Work	136

Publications The use case on detecting anomalies within assemblies has initially been outlined in [86]. The specifics on the generation of anomalous assemblies as well as the methodology of the two tasks of anomaly detection and alternative suggestions together with their experimental evaluation are published in [84].

Instead of recommending parts next needed during the design, the third use case primarily focuses on the quality assessment of an assembly by identifying unusual parts or combinations of parts that suggest an unconventional solution to a design problem. The large selection of parts from various part catalogs can easily lead to unfavorable part selections. There are a number of reasons why this can occur in assembly modeling. Let us consider the example where a robust variant of a control cabinet consisting of robust side panels and heavy doors is to be designed. In this case, the designer may only be familiar

with lightweight parts, so that they use fragile hinges instead of robust ones to stabilize a heavy door. The combination of robust, heavy doors with fragile hinges should be identified as unusual, but is not necessarily wrong or faulty. In fact, this could be intentional when fragile hinges are cheaper than robust ones, even when they need to be replaced more frequently due to excessive stress. This makes it apparent that the definition and therefore the detection of anomalies is difficult, as the boundary between regular and anomalous data is not always clear. A second cause for anomalous assemblies is when a designer, due to a lack of experience, would insert a robust hinge from a manufacturer outside from the desired pool of parts. From a business perspective, companies might limit their procurement to a set of well-known part types (due to better contracts with manufacturers, higher reliability during the product lifecycle, etc.) within their strategic part management. As a third case, when customizing a product in contract manufacturing or special machine construction, for example, a designer does not necessarily start from scratch, but typically takes an existing assembly as a basis and replaces parts. It is conceivable that the initial assembly stems from a much earlier product with some part types having become obsolete since then.

Completely new part combinations within a similar group of designs indicate that certain parts have not been used in accordance with the guidelines, which is why these parts should be marked as an anomaly. In these cases, we want to make knowledge implicitly anchored in newer designs (such as the omission of certain parts) explicit. While in the latter two cases described above, a part is anomalous solely from the use of the corresponding part type alone, the parts in the first scenario deviate due to their unusual *combination* with other parts, in other words due to their unusual use in a certain context. Anomalies corresponding to the second and third cases can be easily identified by tracking the part types used over time and therefore neither require a learning component nor the incorporation of the graph structure. Therefore, in this use case, we focus on the first case described above, namely the identification of anomalous parts based on their *unusual usage* within an assembly. Our aim is to identify *anomalous* parts of assemblies, in the sense of them being different from standard or unusual – but not necessarily in a negative way (referred to as abnormal). Indeed, a certain part combination can represent a new, sensible way that just has not been used or needed before. The designer can then check these parts once again and replace them with more suitable parts if necessary. Assemblies with recurring patterns of parts and connections, on the other hand, are more likely to be classified as fully *regular*.

Anomaly detection problems on graphs are categorized according to whether the anomaly relates to a graph element (node, edge or subgraph) or the entire graph [92]. To effectively support a designer, we want to analyze not only *whether* an assembly deviates in any way (i.e., graph detection), but also *where* and *how*. When employing edge detection, we would assume that the part itself belongs to the assembly, but has been connected to the wrong parts. In addition, if an anomalous edge has been detected, it is still undecided which of the two parts was used incorrectly. Instead, we address the scenario where the cause of the anomaly is that an unsuitable part was selected for the current assembly, so that there would have been a more suitable variant. We therefore assume that, in principle, a part can be connected to the current neighboring nodes, but that an unusual part type was selected. This problem therefore relates to anomalous *node detection* as opposed to edge detection. In subgraph detection, scenarios are investigated in which subgraphs exhibit irregularities due to their *internal structure* or size, as is the case, for example, with a

botnet within a social network [92]. This means that the individual nodes of the subgraph would not be anomalous in themselves. Our focus, by contrast, is on unusual node types rather than unusual numbers of connections as we do not want to mark connected parts as anomalous because they have an unusually large number of connections with other parts, for instance.

Analogous to the first two use cases, we aim for an *interactive support system* that *highlights unusual parts* in the current assembly to the designers. A cause for using unusual parts is that inexperienced designers did not know or find the parts they actually needed. It therefore seems natural to additionally provide the designers with *alternative suggestions* for the anomalous parts on request, which is the second goal of the current use case. This has some similarities to the first variant of localized part recommendation addressed in Section 6.2 in the sense that we aim to recommend suitable part types for a specific node of an assembly graph, i.e., $P(\mathcal{T} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$. For localized recommendations, this involved *adding* a new part of the predicted type to the existing assembly graph at a specific node. However, with alternative suggestions, we intend to suggest part types for an existing node as we want to *replace* the corresponding part. Section 7.3 illustrates how the procedure for part type recommendation must be adapted in this respect.

In contrast to anomaly detection, alternative suggestions for anomalous parts should be explicitly requested by the user; we are explicitly not pursuing an automatic correction model for anomalous parts as we want to support design engineers in creating high-quality assemblies instead of making automated adjustments without consent. Again, we initially provide a fixed number of part types, although more recommendations can be requested. The designer can select an alternative part from the suggestions if desired or can search the catalogs themselves for a different part type. In principle, this system can also be used for alternative suggestions for regular parts.

From a graph machine learning perspective, both anomalous node detection and recommending alternative parts are node classification problems. For the first task, a binary decision must be made per node between the two classes regular and anomalous. For the second task, alternative part types are to be suggested for a specific existing node of the assembly. Analogous to part type recommendation in previous use cases, we predict a score for each part type in the vocabulary which serve as classes of this classification task.

We want to point out that the graph structure plays a central role also in this use case: Considering the design of heavily loaded machines, so-called predetermined breaking points are often integrated, which are intended to break in case of overload for safety reasons. These parts are only useful at specific locations within an assembly. If the graph structure was neglected and only the set of parts is considered, the localization information would be lost, and the anomalous use of such parts in incorrect locations could not be detected. Therefore, we operate on assemblies represented as undirected graphs over parts and process them with GNNs that follow the message passing paradigm – according to our approach for the previous use cases.

7.1 Generating Synthetic Anomalous Assemblies

A known challenge in anomaly detection problems is the lack of labeled data [23]. This is also the case for our assembly datasets as they originate from orders through assembly configurators. Since they were indeed purchased in this configuration, we assume their quality to be good enough to be considered as regular. Therefore, we need to artificially

create anomalous assemblies which is a typical first step in many anomaly detection problems [23]. We again want to follow our goal of an automated approach which does not involve design engineers having to manually create and label anomalous assemblies. Since anomalies rarely occur in many application areas, researchers have developed techniques to create synthetic anomalies. In the context of graphs, they can be roughly categorized as follows according to [92]:

- (i) Creating synthetic graphs with injected anomalies: new graphs are created (mainly manually) into which anomalies are planted afterward.
- (ii) Injecting anomalies in real-world instances: existing real-world graphs are modified in structure or attributes in order to form a transformed, anomalous graph.
- (iii) Downsampling of graph classification datasets: one particular class is chosen to represent regular graphs while all graphs of other categories form the anomalies. To obtain class balance, excess anomalous graphs are discarded (downsampling).

The last category is not suitable as we have no class assignment to our assemblies and all our assemblies are fully regular. Following the requirement of an automated approach for creating anomalous assemblies, only strategies of the second category are applicable. We therefore augment the existing assemblies in order to get synthetic anomalous assemblies. In the following, an anomalous assembly is denoted as $\bar{\mathcal{A}}$. The difficulty with the augmentation is to ensure that the anomalies are genuine as we do not have information on all allowed part combinations, only about those present in our existing assemblies.

We consider the connections in the dataset as regular and make modifications to the assemblies that do not comply with them. To do so, we represent all regular connections between two part types in a relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$. The pairs of the relation are obtained from the assemblies of *all* data sets (training, validation and test) in order to avoid creating inconsistent instances. If they were only extracted from the training (and possibly also validation) assemblies, it could happen that a modified and therefore anomalous graph according to the training set would also appear as a regular assembly in the test set. In this case, the trained model would predict the supposedly anomalous node as such during the evaluation, but this prediction would be assessed as incorrect, as the whole assembly is considered as fully regular in the test set. Moreover, the model could even learn incorrect correlations from the training data due to this instance, which would impair its generalization ability. Consequently, such inconsistent targets must be avoided in any case.

In order to create synthetic anomalous assemblies, we first need to extract all regular, allowed connections from the existing assemblies so that we can replace parts with unsuitable ones which do not comply with them. For a given set of assemblies $\{\mathcal{A}_i = (\mathcal{V}_i, \mathcal{E}_i, T)\}_{i=1}^n$, the regular connections are given as $\mathcal{R} = \{(T(u), T(v)) \mid \exists \mathcal{A}_i : (u, v) \in \mathcal{E}_i\}$. Since they are extracted from undirected graphs (i.e., $(u, v) \in \mathcal{E}_i \implies (v, u) \in \mathcal{E}_i$), the relation is symmetric¹. For a specific part type $\tau \in \mathcal{T}$, $\mathcal{R}(\tau) = \{\tau' \in \mathcal{T} \mid (\tau, \tau') \in \mathcal{R}\}$ gives the set of all connected part types to τ . Figure 7.1 shows the extracted pairs of regular connections for an example assembly.

¹A relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is called symmetric if: $\forall a, b \in \mathcal{S} : (a, b) \in \mathcal{R} \implies (b, a) \in \mathcal{R}$

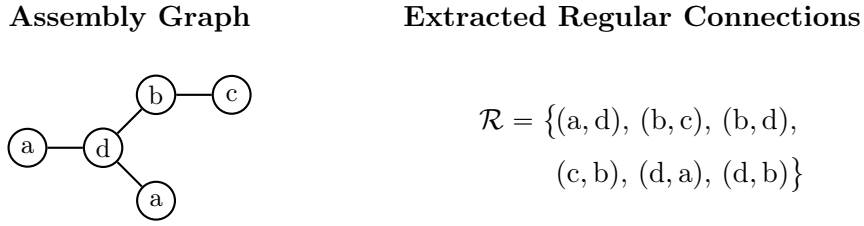


Figure 7.1: Extracted regular connections between part types for a given assembly graph represented in a symmetric relation \mathcal{R} .

Afterward, we can create anomalous assembly instances, which will be used for both anomaly detection and alternative part type recommendation. Due to the small size of the assemblies (on average six parts for datasets A and C) and small number of neighbors (on average less than two neighbors on all datasets), confer Table 4.1, we limit ourselves to a single anomalous part per assembly. The higher the proportion of anomalous parts in an assembly, the more difficult it is to distinguish which of them are anomalous. At the latest when the anomalous nodes make up the majority, even the few remaining regular nodes would be recognized as anomalous. From a domain perspective, it can also be assumed that inappropriately selected parts occur rather rarely, so that a ratio of one anomalous part out of six appears to be sufficient.

Consequently, for a given regular assembly \mathcal{A} , one arbitrary selected part is replaced by an unsuitable part and thus marked as anomalous; all unchanged parts are accordingly marked as regular. Unsuitable parts are selected to be irregular to all neighboring nodes according to \mathcal{R} . The original type of the replaced part serves as the correct part type $l_\tau \in \mathcal{T}$ for the proposal of a more suitable alternative part. Algorithm 3 summarizes the procedure of generating synthetic anomalous assemblies in pseudocode. For anomaly detection, we derive a binary vector $\mathbf{l}_{\mathcal{V}} \in \mathbb{B}^{|\mathcal{V}|}$ as the target, which indicates anomaly or regularity for each node. As there is exactly one anomalous part per assembly, this is a one-hot vector.

Algorithm 3 Augmenting Assemblies to Anomalies

- 1: **procedure** AUGMENTGRAPH($\mathcal{A} = (\mathcal{V}, \mathcal{E}, T), \mathcal{R}$)
 - 2: select arbitrary node $v \in \mathcal{V}$
 - 3: $l_\tau \leftarrow T(v)$ ▷ save original part type
 - 4: $\mathcal{R}_{\mathcal{N}(v)} \leftarrow \bigcup_{n \in \mathcal{N}(v)} \mathcal{R}(T(n))$ ▷ regular part types for neighbors
 - 5: select arbitrary part type $\tau' \in \mathcal{T} \setminus \mathcal{R}_{\mathcal{N}(v)}$
 - 6: $T(v) \leftarrow \tau'$ ▷ update part type of v
 - 7: **return** $\langle \mathcal{A}, v, l_\tau \rangle$
-

When selecting the anomalous type, it must necessarily be irregular to all neighboring nodes as the detection of the anomalous part becomes ambiguous if the connection to some of the neighbors remains regular, as illustrated by an example in Figure 7.2: In the middle assembly, the new part type **c** was chosen to be irregular to all neighboring nodes, while the new type **a** in the right assembly is only irregular to the right neighbor node. Since the connection between types **a** and **d** (which is the type of the other neighbor of the modified node) is regular (cf. Figure 7.1), the last assembly could also have been created by anomalizing the rightmost node resulting in type **c**. To avoid ambiguity, the new part type is chosen irregularly to all neighboring nodes in our approach.

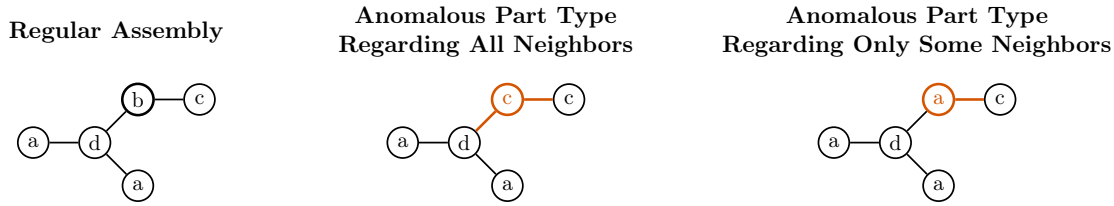


Figure 7.2: Comparison of two strategies for creating a synthetic anomalous assembly. A regular assembly (depicted on the left) is anomalized at a randomly selected node v (highlighted with a thick border) based on the regular connections \mathcal{R} in Figure 7.1. Anomalous parts and edges (i.e., the corresponding connection is not conforming to the relation of regular connections) are highlighted in red. We assume a given part type set $\mathcal{T} = \{a, b, c, d\}$. In the middle assembly, the new part type is irregular to all its neighbors (i.e., chosen from $\mathcal{T} \setminus \mathcal{R}_{\mathcal{N}(v)} = \{c, d\}$), whereas for the right assembly, the new part type is irregular only to the right neighbor node, i.e., chosen from $\mathcal{T} \setminus \mathcal{R}(c) = \{a, c, d\}$.

The described procedure for selecting an anomalous part type may result in obvious anomalies by choosing inappropriate types, such as a new type τ' that significantly differs in its purpose from the original regular type τ . In the introductory example of the control cabinet, a hinge could be replaced by, e.g., a drawer pull-out. To create more realistic anomalies, we restrict the possible replacement type τ' (selected in line 5 of Algorithm 3) to the same *part family* [124] as the original type. As introduced in Section 2.1, slight variations of a part (for example regarding dimensions or position of holes) are handled by parameterization of the part model instead of creating a separate model for each specific part. These variations are summarized in a part family. This would be equivalent to the designer having selected a part that is in principle compatible, but the exact specification is unsuitable for the specific assembly – e.g., a fragile hinge instead of the needed robust hinge. The limitation to the same part family greatly reduces the number of remaining part types: Without restriction, $\mathcal{T} \setminus \mathcal{R}_{\mathcal{N}(v)}$ still includes at least 1,727 out of 1,930 types for dataset A, at least 2,087 out of 3,099 for dataset B and at least 1,202 out of 1,924 for dataset C over all nodes. If the restriction is applied, on average 39.3 types remain for dataset A, 24.5 for dataset B and 51.9 for dataset C. However, it also happened that there were no remaining part types from the same family as the original part. On the three assembly datasets, this was the case in 4.7% of all partial assemblies created for A, 13.3% for B and 3.8% for C, respectively. In these cases, we repeal the restriction to the same family and select an arbitrary remaining type instead.

As previously discussed in this thesis, we do not know the exhaustive list of parts that would fit to each other (cf. Section 5.4). We are only provided with a limited set of assemblies from which we extract regular part connections. Consequently, the extracted set of regular connections is also not exhaustive. This means that if we draw a part type outside the regular connections as an anomalous part, there is a chance that we will select a part type that can actually be used regularly in this position. We would then have created a regular assembly (not mirrored in our dataset), but marked it as an anomaly. To avoid this situation as much as possible, we created the regular connections from instances of all data sets as discussed above. The restriction to the same part family even increases this risk. We expect that only certain part types can be meaningfully connected to each other, i.e., only a small fraction of the two to three thousand part types will fit together.

The numbers of remaining alternative types indicates that there are still enough types left to choose from, so that the probability of selecting an unknown matching type is quite low. However, how many of the remaining part types would actually be regular cannot be assessed without domain experts. We consider the risk also to be low when restricting to the same part family: Given that a part type represents a key specification of its part family, we can reasonably assume that these specifications are essential when using it in a particular context, and there are no arbitrary “matches” when selecting a different type outside \mathcal{R} from the same family. For example, drill holes specify the required length and thickness of screws very precisely such that a screw of the same family with different specifications is therefore obviously not suitable.

To maintain balance between regular and anomalous assemblies, we create an anomalous assembly for each regular assembly. This would give us a total of about 24,000 assemblies per dataset (that need to be split in training, validation, and test sets), which is a very small number of instances. Therefore, we apply the common trick of *data augmentation* to get more instances [42]. In the previous use cases, we generated all possible subgraphs of an assembly by iteratively truncating non-cohesive nodes (cf. Algorithm 1 and 2) in order to create self-supervised recommendation instances. Here, we follow the same procedure to generate all unique subgraphs of an assembly, but discard the cut-off target part and add the resulting regular partial graph only to our instances. To be specific, we generate these subgraphs for all assemblies within a data set, i.e., individually for the initially separated training, validation and test assemblies. An additional reason for using partial assemblies to create anomalous assemblies is that we aim to recognize anomalies not only in final assemblies but primarily *during the design process*. Therefore, our instances should contain partial assemblies with anomalies.

The required set of regular connections does not change by this augmentation step and can thus initially be extracted on the original assemblies. As they represent regular partial assemblies, we set the binary vector $\mathbf{l}_V \in \mathbb{B}^{|V|}$ to regular for every node. Finally, we create an anomalous assembly for each partial assembly following Algorithm 3. Table 7.1 shows the number of resulting partial assemblies per data set and dataset. Due to the varying graph structures, particularly in terms of graph order and node degrees across the datasets, decomposition of the about 12 thousand assemblies results in between 123 thousand and 740 thousand subgraphs and thus both anomalous and regular instances. Table 7.2 provides an overview of the distribution of anomalous and regular nodes in the anomalized assemblies.

Table 7.1: Number of regular subgraph assemblies used for anomaly detection for all three datasets. As each regular assembly is converted into an anomalous one, this results in the same number of anomalous assemblies per data set.

Dataset	# Part Types	Size Training Set	Size Validation Set	Size Test Set	Total
A	1,930	74,638	24,151	24,587	123,376
B	3,099	453,477	140,906	145,288	739,671
C	1,924	247,575	85,512	90,060	423,147

Table 7.2: Distribution of regular and anomalous nodes in anomalous assemblies for each dataset. Since exactly one part per assembly is replaced by an anomalous type, the number of anomalous nodes corresponds to the number of original partial assemblies in Table 7.1.

Dataset	Training		Validation		Test	
	Regular	Anomalous	Regular	Anomalous	Regular	Anomalous
A	271,308	74,638	86,937	24,151	88,368	24,587
B	2,889,401	45,347	892,232	140,906	933,117	145,288
C	1,287,238	247,575	457,283	85,512	483,826	90,060

Depending on the specific task, the respective instances consist of the required assemblies as well as the relevant constituents from Algorithm 3: Both regular and anomalous assemblies are used for the anomaly detection task. By processing both types of assemblies, the model can develop an understanding of both concepts and thus better distinguish between them. The goal of anomaly detection is to predict for each node whether it is used regularly or anomalously in the assembly. Therefore, we need the information about which node belongs to which class as target values, which is given in the vector \mathbf{l}_ν . An instance for anomaly detection thus consists of an (anomalous) assembly as input and the stacked binary vector indicating anomaly for each node of the assembly as target. For the recommendation of alternative parts, however, we only use instances composed of anomalous assemblies together with the corresponding original part types l_τ as targets. Since we only present alternative suggestions for anomalous parts, regular assemblies are not considered. Consequently, we have twice as many instances for the detection task. The vector \mathbf{l}_ν is used as input to indicate the anomalous node in the assembly and thus for which of the nodes we aim to get suggestions of alternative part types.

7.2 Detecting Anomalous Parts in Assemblies

Our first goal in this use case is to detect anomalous parts within an assembly. Specifically, for a connected assembly, we want to decide for each part whether it is regular or anomalous, i.e., determining $P(\mathbb{B} \mid \mathcal{A}, v \in \mathcal{V}_{\mathcal{A}})$ for each part v . Since we focus on anomalies that have emerged due to inexperience or an unsuccessful search for the right part types, the anomaly arises due to the use of an unusual part type in a certain context, i.e., it was combined with improper part types. Consequently, the graph structure, especially the (direct) neighbors, is crucial to decide on anomaly. GNNs are therefore particularly suitable for this problem. From a graph machine learning perspective, we model this problem as a binary node classification problem on assembly graphs. The model architecture therefore corresponds to the node classification task for the first variant of localized part recommendation shown in Figure 6.2, whereby binary instead of multi-class classification is performed.

In our synthetically generated anomalous assemblies, we limited ourselves to one anomalous part; however, the same procedure of anomaly detection could be applied to assemblies with multiple anomalous parts – basically even when anomalous parts are adjacent and thus form a connected subgraph of anomalous parts. Note that detecting anomalous subgraphs is more challenging, as the anomalous parts could be regular to each other, but the subgraph as a whole does not fit the rest of the assembly [75]. This situation could occur when incompatible subassemblies from different previous products are combined.

Baseline Model Our anomaly detectors are trained on an equal number of regular and anomalous graphs. However, since only one node of an anomalous assembly is anomalous, there is a strong imbalance between regular and anomalous nodes at node level. The majority of all nodes belongs to the class of regular nodes. Obviously, the minority class is harder to predict by machine learning algorithms because there are few examples of this class and the dataset encodes a bias towards the majority class. This problem occurs in many anomaly applications because anomalies are naturally rare and often overshadowed by the majority of regular instances. To assess the quality of an anomaly detector in such imbalanced settings, its performance is typically compared with that of a model that predicts the majority class (in our case, regular) for each node as this model is very often correct. This allows us to evaluate whether the anomaly detection model goes beyond the prediction of the majority class and actually makes node-specific decisions between anomalous and regular. We also use such a model as a baseline model. Since our main class is regular, we call the associated model *regular predictor*. This model basically corresponds to the Evergreen model (Section 5.2), which we have used as a simple baseline for part recommendation, if it is restricted to output exactly one class, thus the majority class of the targets. In this thesis, however, we confine ourselves to a single anomalous part.

7.2.1 Experimental Setup

We performed experiments for anomalous node detection on each of the three assembly datasets individually. Initially, the assemblies divided into training, validation and test sets were separately decomposed into subgraphs following Algorithm 1. Afterward, each regular partial assembly was transformed into an anomalous assembly by replacing an arbitrarily selected part according to Algorithm 3 as described in Section 7.1. Both the regular and anomalous partial assemblies form the instances for this anomaly detection task, where the binary node-wise indication of anomaly, i.e., the vector \mathbf{l}_V , serves as label. Therefore, we still have the same number of anomalous nodes among all instances as given in Table 7.2, but additionally the number of regular and anomalous nodes as regular nodes due to the additional regular assemblies of the same structure. For example, the partial assemblies in the test set of dataset A still have 24,587 anomalous nodes and a total of 201,323 regular nodes. We adhere to the findings of the first use case and employ the pretrained 100-dimensional part type embedding as representational vectors for the parts. It was trained such that parts with similar usage or purpose are close to each other in the embedding space, cf. Section 4.3.

Investigated Models In the experiments on the previous two use cases, the GIN and GraphSAGE architectures achieved the best results, which is why we now employ these two for anomaly detection and neglect the other two GNN architectures. The particular model architecture is given by specifying the usual hyperparameters: number of layers, dimension of the layers and activation function. For GraphSAGE, we again examine the most suitable aggregation function. Binary classification can be implemented using both a one-dimensional output with a sigmoid activation function and a two-dimensional output with a Softmax activation function [51]. Both options are very similar: While in the first variant, only one score is calculated for the first class and it is compared with the score of the second class, which is fixed to the value 0 by definition, in the second modeling, scores are calculated for both classes. In both cases, the higher score determines which class was predicted. Controlled by an additional hyperparameter, we try out both variants in the

experiments. The GNNs are regularized by dropout from the second layer onwards so that no information from the input assembly is dropped. Due to the problem of class imbalance (the majority of nodes is regular) we also apply the regular predictor as baseline model. This allows us to evaluate whether the detector actually decides contextually with respect to anomaly instead of just outputting the majority class.

Training The training details are basically the same as for the first use case: As the task is modeled as (binary) node classification task, we again apply cross-entropy as loss function for training the detector models. Furthermore, the end of training is determined by early stopping based on the sum of the training loss and the difference between the training and validation losses. We apply automatic hyperparameter search supported by the framework Optuna [2].

Performance Metrics As performance metrics we use the standard metrics for binary classification, namely precision and recall. While precision measures the fraction of correctly predicted among all positively predicted instances, recall states the fraction of positively predicted instances among all positive instances. In this task, there is no preference to which type of misclassification (anomalous parts are recognized as regular or regular parts are recognized as anomalous) should be avoided. Consequently, we measure precision and recall with respect to both classes, i.e., in one case regular is chosen as the positive class, in the other case anomalous is the positive class. Both metrics are basically designed for single instances, so there are different ways to aggregate their values for a set of instances, namely the whole test set [130]. First, the respective metric is determined individually for each class. *Macro*-averaging then takes the arithmetic mean across all classes which weights each class equally, regardless of the number of instances. However, *weighted* averaging takes the balance of the classes into account by weighting the class-based metrics by the number of instances for each class – referred to as class size.

7.2.2 Experimental Results

Table 7.3 displays the performance of the GIN and GraphSAGE detectors as well as of the regular predictor for as baseline detecting anomalous nodes on the test set for every dataset. The performance metrics, precision and recall, are shown for each class individually as well as macro-averaged and weighted by the respective class size.

First, we would like to briefly discuss certain performance values of the regular predictor that result by its definition: The model achieves 100% recall for the regular class, since all actual regular nodes are predicted as such. Since it never predicts a node to be anomalous, both precision and recall for the anomalous class have the value 0. The arithmetic mean for macro-averaging of recall is therefore 50%.

Except for recall for the regular class, both GNN models consistently outperform the baseline in all performance tests across all datasets. This shows that they are indeed able to contextually decide whether a part is used in an unusual way within an assembly. With regard to the regular class, the GNN models achieve both precision and recall of at least 98% for all datasets. In general, the values for recall are slightly below precision for the regular class, but only 0.3–1.5 percentage points. This is consistent with the fact that recall achieves higher values than precision for the anomalous class. That shows that more regular nodes were detected as anomalous than the other way round, which can also be

Table 7.3: Summary of experimental results for anomalous node detection on the test set for all three assembly datasets. The performance of the GNN models is compared with the regular predictor as baseline in terms of precision and recall, determined individually for each class as well as macro-averaged and weighted by class size. All values are given in percent.

Dataset	Model	Regular		Anomalous		Macro		Weighted	
		Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
A	GraphSAGE	99.6	98.1	86.2	96.6	92.9	97.3	98.1	98.0
	GIN	99.2	98.1	85.5	93.4	92.4	95.8	97.7	97.6
	Baseline	89.1	100.0	0.0	0.0	44.6	50.0	79.4	89.1
B	GraphSAGE	99.8	98.9	86.9	97.9	93.4	98.4	99.0	98.9
	GIN	99.7	99.1	88.0	95.9	93.9	97.5	98.9	98.8
	Baseline	93.3	100.0	0.0	0.0	46.6	50.0	87.0	93.3
C	GraphSAGE	99.8	99.5	94.6	98.1	97.2	98.8	99.4	99.4
	GIN	99.8	99.3	92.3	97.4	96.0	98.4	99.2	99.2
	Baseline	92.2	100.0	0.0	0.0	46.1	50.0	84.9	92.2

Table 7.4: Experimental results for anomalous node detection of the GraphSAGE models of all datasets in a binary confusion matrix, respectively. Each row represents the fraction of instances in a target class, while each column represents the fraction of instances in a predicted class. The diagonal entries represent the instances that are correctly predicted. All values are given in percent.

Dataset A		Predictions		Dataset B		Predictions	
		Regular	Anomalous			Regular	Anomalous
Targets	Regular	87.44	1.68	Targets	Regular	92.27	0.99
	Anomalous	0.37	10.51		Anomalous	0.14	6.60

Dataset C		Predictions	
		Regular	Anomalous
Targets	Regular	91.72	0.44
	Anomalous	0.15	7.69

seen in Table 7.4 showing the actual and predicted classes of all nodes for the GraphSAGE models. The reason for this could be that some part connections appear very rarely or even not at all in the training data, so that these connections of similar parts are marked as irregular: Since the datasets contain real-world assemblies, the distribution of parts and connections is not uniform, i.e., some parts occur extremely rarely. Thus, this corresponds to acceptable behavior of an anomaly detector, since rare connections are unusual.

The performance of the GNN detectors drops significantly from the regular to the anomalous class, which was to be expected due to the clear class imbalance. For datasets A and C, the GraphSAGE architecture outperforms the GIN models on nearly all metrics, while performing equally well for the recall of the regular class. Dataset B, by contrast, shows that the GIN model focuses on the anomalous class, which is reflected by higher precision on this class and lower recall for regular nodes. This means that this model flags regular nodes more often as anomalies, thus increasing the effort for designers as they

would have to check all predicted anomalies. Nevertheless, we would opt for the GIN for this dataset, as we would accept increased effort for the sake of quality of the assemblies if necessary. It achieves a precision of 93.9% and recall of 97.5% at macro level. The best models achieve a recall of between 96.6% (dataset A) and 98.1% (dataset C) for anomalous parts; for precision, the values are only range from 86.2% to 94.6% for the same datasets.

The aggregated values after weighted-averaging best reflect the performance of the detector in application, as anomalies also occur less frequently there. Due to the weighting based on the class sizes, the values of both metrics are almost identical. Overall, anomalous parts can be detected with a precision (and recall) of 98.0% to 99.4%, which is a very good performance. Surprisingly, the results for the first dataset are the poorest, whereas this was always the easiest to master throughout all experiments of the previous use cases. As outlined above, we assume that certain combinations from the test set barely occurred in the training data. Therefore, the model classified them as anomalous, even though they were regular in the test assemblies.

Table 7.4 shows the percentages for comparing the prediction with the actual classes for the GraphSAGE models. Misclassifications were made on dataset A in 2.05%, on dataset B in 1.13% and dataset C in only 0.59% of cases. Accordingly, for at least 97.9% of the nodes, it was possible to correctly predicted whether it was an anomalous or regular part in the assembly, which is an outstanding result of the anomaly detectors.

Finally, we would like to discuss the training and evaluation time of the GNN models for anomaly detection on our Nvidia DGX-1 station. Once more, we consider dataset B to do so, as it consists of the largest assemblies and therefore the most instances across all datasets. However, compared to the previous two use cases of global and localized part recommendation, each data set for anomaly detection contains about 40% less instances, respectively, which is reflected in the training and evaluation time. Further, in contrast to the experiments for localized recommendations, early stopping monitors the loss values of the training and validation sets in this task, which can be determined faster than performance measures. This resulted in an average epoch training time of 5.5 minutes for the GraphSAGE model. In general, the training completed within 25 epochs for all models. The evaluation time is even lower than for global part recommendation as around 1100 instances per second can be evaluated by the performance measures precision and recall.

7.3 Recommending Alternative Parts

Subsequent to the detection of anomalous parts within an assembly, we want to propose alternative types for the detected anomalous parts of a given assembly. Specifically, for a given anomalous assembly $\bar{\mathcal{A}}$ and anomalous part $\bar{n} \in \mathcal{V}_{\bar{\mathcal{A}}}$, we aim for a fixed number of part types $\tau_1, \dots, \tau_k \in \mathcal{T}$ as suggestions alternative to the anomalous part in order to correct the anomaly. Formally speaking, we have examined a very similar problem in the first variant of localized part recommendation, since both can be modeled by $P(\mathcal{T} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$. However, both problems are fundamentally different: In the localized part recommendation, we follow a *constructive* approach as we add a new part to the assembly at the extension point whose type corresponds to the recommended part type. After the operation $\mathcal{A}_n \triangleleft \tau$, the assembly therefore contains a new part of type τ connected to part n . In the case of recommending alternative part types, however, we want to *replace* the anomalous part by an alternative, better suitable type that was recommended by our model. The number of parts therefore

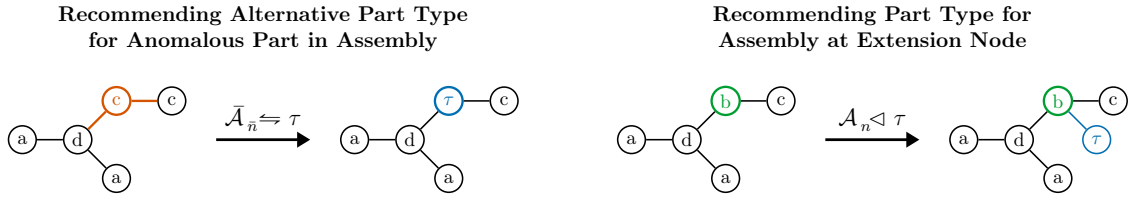


Figure 7.3: Comparison of recommending alternative part types in the context of anomaly detection (left) with recommending new part types for the first localization variant (right). Both tasks can be modeled similarly as conditional models $P(\mathcal{T} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$. For the alternative suggestion, we use the notations $\bar{\mathcal{A}}$ and \bar{n} for the anomalous assembly and node, respectively. The respective operator is given on the arrow, with the resulting assembly after its application displayed on the right. The selected part type $\tau \in \mathcal{T}$ is highlighted in blue, respectively.

remains the same when applying the operator $\bar{\mathcal{A}}_{\bar{n}} \Leftrightarrow \tau$, only the type of the anomalous part \bar{n} has been replaced by the recommended type τ . Both approaches are shown in Figure 7.3.

We assume that the recommendations for alternative parts are independent of the specific anomalous type. For instance, in the introductory cabinet example, it should not matter which variant of lightweight hinges were used, the more suitable robust hinges should be selected due to the heavy doors and robust side panels used. Consequently, the alternative recommendations should be based on all parts except the anomalous one, so it appears reasonable to remove the anomalous part from the assembly to exclude that information. If we additionally select its direct neighbor(s) as extension node(s), it seems that we can use the constructive operator and thus the same procedure as for the first localization variant for the alternative suggestions, too. The example in Figure 7.3 clearly shows the problem here: The anomalous part can be a cohesive node, so that the assembly would break down into several connected components after removal. However, all of our approaches are designed for connected assemblies, as message passing only propagates information along the edges, meaning that in the case of multiple connected components, information is only processed within the individual components, but not across them. As a result, the context of the entire assembly could not be captured, but only of the individual isolated connected components, which would lead to a poor recommendation performance.

This issue did not occur in localization because the instance generation procedure Algorithm 2 only removes non-cohesive nodes. There are several possible solutions for this issue: If we continued the approach of node removal, we could ensure the information flow in the entire assembly by, e.g., *rewiring* the graph [136] by adapting its edges, or inserting an artificial node connected to all existing parts, referred to as *global node* in literature [158], to get a connected graph in both cases. However, graph rewiring is a computationally intensive solution and thus discarded. Therefore, instead of removing the entire node, we follow an approach that disregards the information of the anomalous part *type* represented as node features and leave the graph structure the same. Indeed, this strategy is similar to using a global node, but we do not add edges to all other nodes of the graph, but stick to the existing connections of the anomalous node. We examine two variants for handling anomalous parts for alternative type recommendation, which are explained in detail hereinafter.

Learning a Default Representation for Anomalous Parts If we delete the type information of the anomalous node from the assembly, we need an alternative vector of suitable dimension as node features. This could be done manually by selecting a vector, but again we follow an automated learning approach. We intend to learn a shared default representation for anomalous parts to replace their original embedding with. Intuitively, the representation should not distinguish different types of anomalous parts but only express that it *is* anomalous – hence, one representation for all anomalous parts. The recommendation model itself should find a suitable representation in the embedding space in order to best recommend alternative parts. At the beginning of the training of the recommendation models, this vector is initialized randomly just like the parameters of the GNN layers. The error gradient is backpropagated to the vector and its entries adjusted accordingly during the parameter update. After training is completed, this fixed default representation is used for inference to represent anomalous parts.

Adoption of Message Passing by Masking of the Anomalous Part Instead of replacing the node features of all anomalous parts with a learned representation, the second variant adapts the functionality of message passing in the first layer to not incorporate the anomalous part’s features. In other words, the corresponding node is *masked* in the first layer; a similar idea was introduced by [99] with the aim of improving the generalizability of GNNs. Many GNN architectures include both the neighboring nodes and the respective node itself when updating the node features (Equation (3.9)), as described in Section 3.3.2. Masking leads to the following changes for message passing: When the anomalous node is updated, its own information is masked so that its representation for the subsequent layer is only calculated from its neighboring nodes (unless these are also anomalous). Its information is also not taken into account when updating the anomalous node’s neighbors. However, the update rule remains the same for all other nodes. From the second GNN layer onwards, standard message passing is performed.

To summarize, the problem of recommending alternative parts for anomalous assemblies is modeled as a multi-class node classification $P(\mathcal{T} \mid \bar{\mathcal{A}}, \bar{n} \in \mathcal{V}_{\mathcal{A}})$ so that the model architecture basically corresponds to that of the first localization variant shown in Figure 6.2, whereby in the first layer one of the described anomaly handling strategies is used and $\mathbf{l}_{\mathcal{V}}$ indicates the anomalous node for which alternative part type suggestions are desired (which corresponds to the extension node in the localization variant).

Framing the Task by Simple Baseline and Upper Bound As usual, we want to be able to assess the results of our proposed methodology and use comparative models for this purpose. Since the alternative suggestion model basically outputs suggestions of part types, we can build on the simple baseline and the estimation of the upper bound from Section 5.2. The simple baseline Evergreen does not require any modification, as it only creates a frequency distribution of all labels of the training data and outputs the desired k most frequent part types, thus representing $P(\mathcal{T})$.

The upper bound model memorizes the entire test set, i.e., the mapping from anomalous graph and anomalous node to original part type for all test instances, in order to give an estimate from above of the best performance to be achieved by a non-oracle model. As the recommendation of alternative suitable types for the anomalous node should be independent of the actual anomalous type, the information of the anomalous type must

first be removed from the assembly. We therefore replace the anomalous part at node \bar{n} with a dummy part of a type $\tau_0 \notin \mathcal{T}$ that is not included in the part type vocabulary, resulting in an adjusted assembly $\tilde{\mathcal{A}} = \bar{\mathcal{A}}_{\bar{n} \Leftarrow \tau_0}$. As our anomalous assemblies only contain one anomalous node and thus after the replacement only one part of the dummy type, the adjusted assembly automatically includes the localization for which node in the assembly alternative types are to be suggested. This means that, for our anomalous assemblies, it is sufficient for the upper bound model to remember the mapping from the adapted assembly $\tilde{\mathcal{A}}$ to the original part type.

However, if there were several anomalous nodes within an assembly, the input must be expanded to a tuple of adopted assembly and specific node – analogous to the first localization variant (cf. Section 6.2). Nevertheless, all anomalous parts first need to be replaced by a dummy part.

7.3.1 Experimental Setup

Finally, we conduct experiments to recommend alternative types for anomalous parts in an assembly. Anomalies were created from the regular partial assemblies by replacing one part with an anomalous one, as already described for anomaly detection in Section 7.2.1. In contrast to anomaly detection, we only use the anomalous assembly instances to reconstruct the original part type of the anomalous part from the remaining assembly. As with the previous recommendation models for global and localized part recommendation (Chapters 5 and 6), the instances have single-label targets.

The overall experimental setup is the same as for the first variant of localized part recommendation described in Section 6.2.1. With regard to the GNN architectures examined for the recommendation models, we restricted ourselves to GIN and GraphSAGE, as these consistently delivered the best results in the second use case, which this task is most similar to. We employ early stopping that monitors the top-5 rate on training and validation sets to determine when to stop the training in order to prevent overfitting and ensure good generalization on the test set. Furthermore, the best fitting hyperparameters are determined using automatic hyperparameter search with the framework Optuna [2]. We frame the results of the GNNs by the simple baseline Evergreen, which outputs a frequency distribution over the target part types, and the upper bound. Since several alternative parts could be suitable for an anomalous part of a certain assembly, the upper bound indicates the best possible performance on the respective test set.

In order not to consider the information of the anomalous part type for the recommendation, we investigate two strategies: Simultaneous to the training of the recommendation models, a shared default representation for anomalous parts is learned. The second strategy masks the associated node in the assembly graph, so that its features are not considered for updating either its own representation or that of its neighboring nodes in the first layer.

7.3.2 Experimental Results

Table 7.5 summarizes the results of the GNN architectures using both anomaly handling strategies and the comparison models for all three assembly datasets. Additionally, Figure 7.4 visualizes the performance of all GNN models and the upper bound per dataset. The Evergreen model and the upper bound define a broad corridor, similar to the first examined use case in Chapter 5.

Table 7.5: Summary of results for recommending alternative types for anomalous parts in assemblies on the test set for all three assembly datasets. Displayed is the top- k rate in percent for $k = 1, \dots, 15$ part type recommendations for the GNN models as well as the upper bound and simple baseline Evergreen, in order to contextualize the GNNs’ performance. Best model is highlighted per dataset.

Dataset	Model	Strategy \ k	1	2	3	5	10	15
A	Upper Bound		91.4	97.4	98.8	99.6	100.0	100.0
	GraphSAGE	Default Rep.	64.6	78.0	83.1	87.8	92.5	94.3
		Masking	64.1	77.2	82.6	87.2	92.0	93.8
	GIN	Default Rep.	62.7	75.2	80.1	85.4	90.2	92.2
		Masking	60.5	74.0	79.5	85.2	90.7	92.7
Evergreen		3.2	5.2	7.2	9.2	13.2	16.8	
B	Upper Bound		92.2	97.3	98.5	99.3	99.8	99.9
	GraphSAGE	Default Rep.	60.6	74.5	79.8	84.8	89.7	91.7
		Masking	61.1	74.8	79.8	84.8	89.7	91.9
	GIN	Default Rep.	60.2	73.7	78.8	83.8	88.6	90.8
		Masking	57.3	70.8	76.1	81.8	87.9	90.5
Evergreen		4.2	8.2	11.0	18.0	30.1	37.0	
C	Upper Bound		84.3	95.0	97.8	99.2	99.8	99.9
	GraphSAGE	Default Rep.	53.8	71.0	79.8	87.9	93.8	95.5
		Masking	53.1	70.2	79.3	87.7	93.7	95.5
	GIN	Default Rep.	53.2	70.5	79.4	87.5	93.3	95.0
		Masking	53.0	70.1	78.9	86.8	93.1	95.0
Evergreen		5.8	11.2	15.8	21.0	27.9	34.2	

Across all datasets, the GraphSAGE architecture consistently outperforms GIN. However, when it comes to the anomaly handling strategy, the results are less clear-cut. For datasets A and C, regardless of the GNN architecture, learning a default representation proves to be more successful than masking. Only for dataset B does masking emerge as the more promising strategy, although the default representation achieves very similar, if not identical, performance levels for the majority of recommendation numbers. Notably, on dataset C, all four model variants perform very closely, with a maximum difference of only 1.3 percentage points. The difference in performance between the two anomaly handling strategies is possibly caused by the small size of the assemblies and number of neighbors, which is given in Table 4.1 for the original assemblies. With masking, the representation of the anomalous part is determined from the neighboring nodes in the first layer; if there are few neighbors, the determined representation is not expressive. The reason for the comparable performance of the masking strategy on datasets B and C could be that they contain the largest graphs with the largest node degrees so that there is a sufficient number of neighboring nodes to determine an expressive representation for the anomalous node with masking. For $k = 5$ recommendations, the best modeling variant on each dataset is using the GraphSAGE architecture with default representation strategy. The model correctly identifies the appropriate part in 84.8% of cases for dataset A and 87.9% for datasets B and C, respectively.

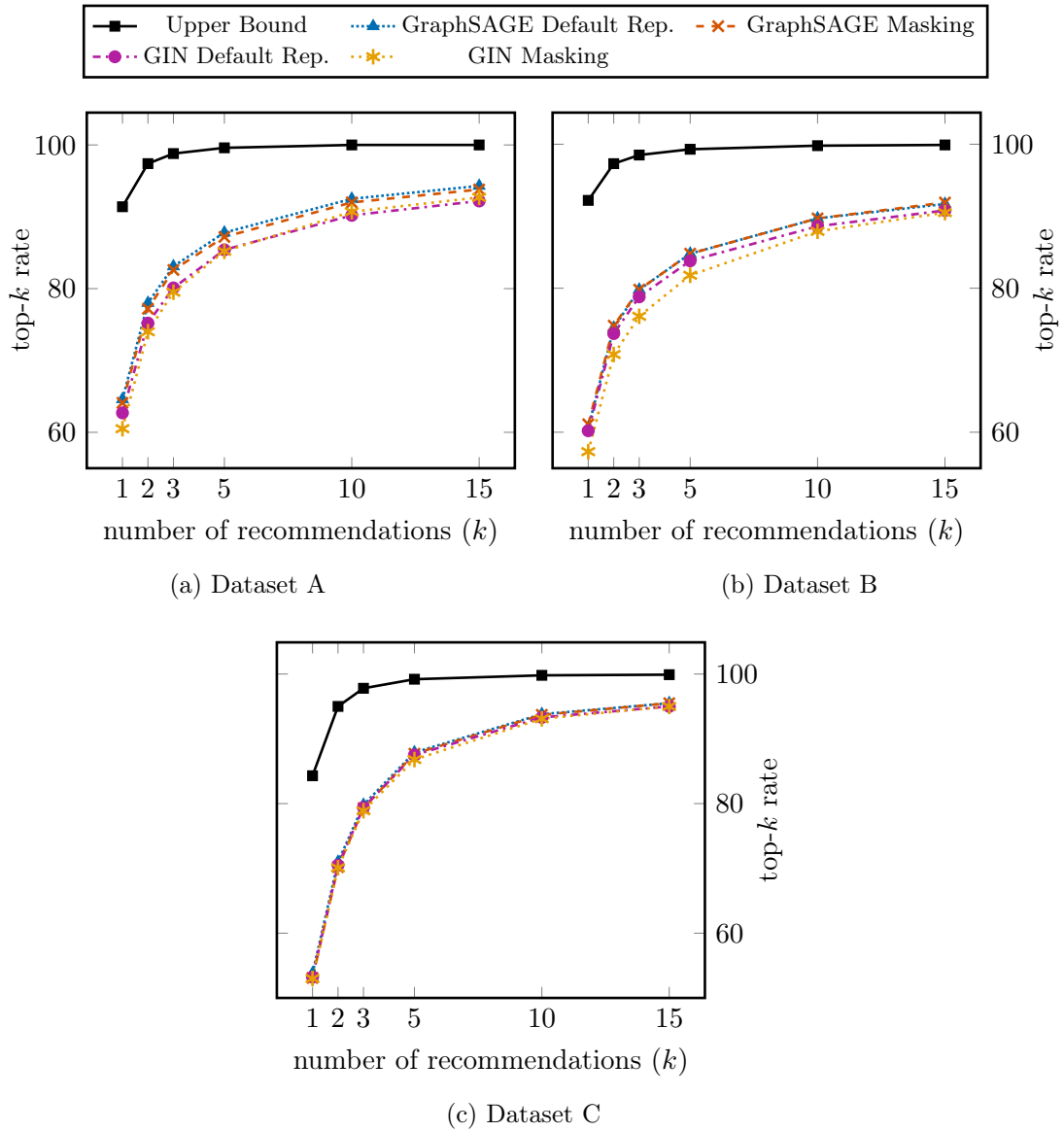


Figure 7.4: Visual comparison of alternative recommendation models on top- k rate. Evergreen omitted due to scaling.

In the setup for this task, the termination of the training is determined by monitoring the performance measure top-5 rate through early stopping. However, since we only have corrected part types for anomalous assemblies, each data set for this task contains about 30% of instances compared to global and localized part recommendation. Training of the GraphSAGE models took about 8.6 minutes per epoch on dataset B and an Nvidia DGX-1. The training mostly had to run between 20 and 30 epochs before the optimal parameters were found, however, it was still completed in under five hours. No significant time difference was found for the two anomaly handling strategies investigated (default representation and masking). The reason for this is that the small number of additional parameters to be learned for the default embedding is negligible compared to the number of parameters of the overall model and therefore has no time impact. The evaluation time of the fully trained model on the training data is very similar to anomaly detection: over 1000 instances could be evaluated per second.

7.4 Related Work

Anomaly detection is a well-established and widely studied problem, although it has rarely been explored in the context of CAD assemblies. In graph-based anomaly detection, anomalies can either refer to the graph as a whole or to specific graph elements such as nodes, edges, or subgraphs [92]. To effectively assist a designer in a targeted manner, we not only aim to assess whether an entire assembly is anomalous (i.e., graph-level detection) but also pinpoint the specific location of the anomaly, in particular, by identifying the anomalous part (i.e., node-level detection).

Early approaches to graph anomaly detection have mainly relied on domain knowledge and statistical methods, where features for detecting anomalies have been manually crafted, which is very time-consuming and labor-intensive. To overcome the limitations of the early work, considerable attention has recently been paid to deep learning approaches when detecting anomalies in graphs. Graph neural networks (GNNs) have also been used for anomaly detection for some time, for example [168, 33]. The surveys [92, 75] provide an overview of anomaly detection techniques with deep learning and GNNs. In deep learning approaches, nodes are considered anomalous if their features differ strongly from those of their neighbors or if they have different connection patterns – or a combination of both [92]. However, anomaly of parts in assemblies goes beyond this definition in our use case as parts are anomalous if they do not fit to their connected parts. Therefore, it is crucial for our use case to capture the interaction of the parts, which is why we employ GNNs. Existing node classification methods are often specialized for a specific GNN architecture, however, we want to provide a generic approach based on an arbitrary GNN architecture.

In addition, most techniques rely on a combination of GNNs and autoencoders, e.g., [8, 164]. In particular, autoencoders reconstruct an anomalous graph and compare the output with the original input. Deviations in node information or edges indicate anomalies. However, for our domain, this method can lead to not only anomalous parts being reconstructed differently but also regular parts being replaced by other suitable alternatives. This makes it difficult to distinguish the detection of actual anomalies from parts for which there would be several suitable alternatives. Therefore, our approach is purely based on GNNs.

Since anomalies are rare in many real-world applications, multiple methods have been developed to generate synthetic anomalies. According to the categorization provided in [23], our approach involves modifying real-world graphs to create anomalous ones.

When generating anomalous assemblies, we have opted for strictly relying on valid part type combinations observed in assemblies in order to allow precise control over the anomalies produced. Alternatively, the anomalous assemblies could also be generated using a learning-based method such as the automated approach presented in [109], which uses diffusion models to generate anomalous data. While this method can produce a broader and more realistic range of anomalies beyond replacing a single node within the graph, the resulting anomalies are harder to control and their evaluation (i.e., determining whether they truly represent anomalies in assemblies) is more difficult. Before these instances are used to learn an anomaly detector, they should be validated by domain experts to avoid learning a wrong concept of anomalous assemblies. As no domain experts were available for this thesis (and we even aim for no involvement of experts in our solutions), we chose to employ an automatic approach only depending on existing assemblies instead.

The field of recommending solutions for detected anomalies in graphs with the intent to correct them has received little attention so far, and the same also applies to the combined task of detecting and correcting anomalies which we refer to as anomaly handling. To the best of our knowledge, this problem has only been addressed by Srinivas et al. in the context of time-series data on hypergraphs from large-scale industrial units [132]. They leverage genetic algorithms to recommend causes of action to remedy the anomalies. Modeling the recommendation of alternative parts as node classification as in our approach is not unusual in the field of GNN-based recommendation systems [41], although link prediction is generally more prevalent. However, we are interested in recognizing an unsuitable part which was added to the assembly, rather than detecting that a part that is actually suitable has just been incorrectly connected. In combining an anomaly detection task with a generic recommendation task, the GraphRfi study [166] comes close to our overall anomaly handling system. However, their recommendation system is not aimed at mitigating the anomaly itself. Instead, the authors use the product recommendations for a user to determine whether the user is a fraudster who is deliberately submitting false reviews to affect the product recommendation model. Similar to our approach, they employ GCNs for the recommendation, however on an edge-level task (i.e., assigning a score to each edge from the user to a product) instead of on node-level.

Summary and Outlook

Summary This chapter summarizes the key contributions of this dissertation, followed by lessons learned that could help in transferring the developed methodologies to other application areas. Finally, it concludes with the major avenues that future work could pursue.

Contents

8.1	Summary	139
8.2	Lessons Learned	143
8.3	Outlook	144

8.1 Summary

This thesis contributes to the field of *artificial intelligence-aided design* (AIAD) and is the first to provide methodologies leveraging machine learning techniques to support inexperienced design engineers in assembly modeling based on expert knowledge extracted from previous assemblies. Given the assumption that knowledge about *how* to solve an application problem is implicitly included in the form of used parts and their combinations, such expert knowledge is encoded as recurring patterns of part combinations over multiple assemblies. On this basis, we presented a generic, data-driven approach which does not require the involvement of any domain experts: We successfully applied *graph neural network* (GNN) architectures from the field of graph machine learning in order to extract recurring patterns from assemblies and to support engineers in three different use cases while they design an assembly:

1. *Global part recommendation* of suitable next parts attachable to the current assembly
2. *Localized part recommendation* of next parts for the current assembly, either for a location selected by the engineer or together with the part of the assembly it can be attached to
3. *Handling anomalies in assemblies* such that anomalous parts are identified and suitable alternative parts are suggested

Representing Assemblies as Graphs over Part Types In order to process heterogeneous assemblies from different part catalogs in a uniform manner, we defined a representation of assemblies as undirected graphs based on the only information available across those catalogs (Chapter 4): We assume a set \mathcal{T} of *part types*, and define each part in an assembly as instantiation of a given type. Consequently, each part of the assembly becomes a node $n \in \mathcal{V}$ of type $\tau \in \mathcal{T}$ in the resulting graph $\mathcal{A} = (\mathcal{V}, \mathcal{E})$. For each two connected parts in the assembly, we add an undirected edge between the corresponding nodes to \mathcal{E} .

We applied the translation to three real-world datasets provided within the project KOGNIA. Each dataset contains around 12,000 mechanical assemblies consisting of parts from a part catalog used exclusively in this dataset (each catalog comprises around 2,000 to 3,000 part types). The assemblies were designed by customers of the part manufacturers providing those part catalogs.

Training Embeddings of Part Types To grasp the knowledge behind recurring patterns of part combinations, it is not sufficient to only memorize previously connected parts, even when considering neighborhoods beyond direct neighbors: If an inexperienced designer needs to connect a valid part to their assembly which has never occurred in this specific combination before (e.g., to design a new product variant), we would never suggest this part and even mark it as anomalous. We rather want to capture the similarity of parts (and thus their adequacy) in terms of their intended functional usage.

Consequently, we developed an embedding technique termed *part2vec* which calculates low-dimensional embeddings encoding the similarity of part types (Section 4.3). As a generalization of the embedding technique *word2vec* [96] from the domain of natural language processing (NLP), *part2vec* is capable of processing graphs of parts instead of only sequential data (such as sentences in the case of *word2vec*).

Intuitively, if multiple parts of different types have been combined with the same set of parts in previous assemblies, these differing part types are considered to be similar to each other as they have been used in similar contexts. As a result, they obtain a similar embedding by *part2vec*.

Refining Part Type Embeddings The fact that some part types may appear infrequently across the assemblies leads to poor embeddings for these types. To improve their representation, we built an *interactive editor* to define further *similarity constraints* for particular part types (Section 4.3.2). These constraints are incorporated into the training routine of *part2vec* to enforce similarity between the part type embeddings. Although this seemingly contradicts the idea of an automated approach, our methodology does not require this manual refinement step. However, in case of poor performance of downstream models, the editor offers a user-friendly way to effectively improve part type representations and thus to increase the prediction quality.

Decomposition of Assemblies for Self-Supervised Learning Training of machine learning models requires training instances representing input and expected target values of the model. However, as our datasets comprise final assemblies, we had to generate such instances ourselves. To avoid manual labeling by design experts, we developed a *data augmentation algorithm* to decompose an assembly graph into single-label instances for the

respective use cases (Algorithms 1 and 2). We also leveraged the same algorithm for generating *synthetic anomalous assemblies* (Algorithm 3), as it not only increases the number of instances, but also allows detecting anomalies during the design process instead of only in the final assembly. Applying the data augmentation algorithm resulted in between 123 thousand and 2.5 million instances, depending on the dataset and use case.

Baseline and Upper Bound Models As recommending parts for and handling anomalies in assemblies pose new challenges in the domain of ML, we established *baseline and upper bound models* to assess the experimental results per use case (Section 5.2).

The baseline for each recommendation use case is a deterministic algorithm which approximates the unconditional probability distribution over all part types $P(\mathcal{T})$ for a given dataset. Intuitively, it computes a usage ranking of part types over all training assemblies. Thus, the model ignores the entire context given by the current partial assembly and always returns the part types occurring most frequently as targets in the training instances.

In contrast, the upper bound is determined by a deterministic algorithm simulating the conditional probability distribution over all part types given a partial assembly $P(\mathcal{T} | \mathcal{A})$, i.e., it computes the best possible recommendations for a given assembly \mathcal{A} from a specific dataset. However, since multiple parts of different type could be attached to the same assembly, the number of recommendations is crucial for the maximum achievable performance: If we recommend fewer parts than those which are attached to \mathcal{A} in the test set, we can obviously never achieve 100% performance during evaluation. Consequently, the algorithm memorizes the *test* instances and returns the k part types occurring most frequently in them for the given assembly \mathcal{A} , where k denotes the number of recommendations.

Although designed for the evaluation of global part recommendation, both the baseline and upper bound models also served as a realistic frame for the other use cases while requiring only minor, if any, adjustments.

Modeling of Recommendation Tasks In general, we modeled all tasks arising from the three use cases as multi-class classification problems, except for anomaly detection (which is a binary classification problem). However, the subject to be classified varies across the different tasks: While global part recommendation is modeled as a graph classification problem, we modeled the other tasks as node classification problems. We employ graph neural networks (GNN) to solve these problems, as GNNs learn representations of each graph node by aggregating the representations of its direct neighbors and of the node itself.

Global Part Recommendation We understand global part recommendation (Chapter 5) as the task to compute a conditional probability distribution $P(\mathcal{T} | \mathcal{A})$ over all part types \mathcal{T} for a given assembly \mathcal{A} , where most suitable part types receive the highest probabilities. We modeled this task as a graph classification problem, since we want to recommend parts for the assembly \mathcal{A} as a whole rather than for only a particular extension node.

Localized Part Recommendation In contrast to global part recommendation, localized part recommendation for a location selected by the engineer (Section 6.2) is modeled as a node classification problem: It refers to the task of computing a conditional probability distribution $P(\mathcal{T} | \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$ over all part types \mathcal{T} for a given assembly \mathcal{A} and a given extension node n out of all nodes $\mathcal{V}_{\mathcal{A}}$ in the assembly, where the most suitable part types receive the highest probabilities.

Recommending both a suitable part and a corresponding extension node (Section 6.3) describes the task of computing a conditional probability distribution $P(\mathcal{T}, \mathcal{V}_{\mathcal{A}} \mid \mathcal{A})$ over all part types \mathcal{T} as well as nodes $\mathcal{V}_{\mathcal{A}}$ of a given assembly \mathcal{A} , where the most suitable part types for the nodes most frequently extended receive the highest probability. Intuitively, a model should recommend the most suitable parts for the extension points it is most certain about first. We modeled the task as a two-task classification problem: One model predicts the most suitable part type, and the other computes extension scores for each node. Both subtasks can be chained in any order, however, the subsequent one must consume the output of the first task as input. We therefore presented modeling approaches for both possible permutations. Depending on the order, the split into two tasks allowed us to leverage the task modelings for global part recommendation or for the first variant of localized part recommendation, respectively, to solve the part recommendation subtask.

Handling Anomalies in Assemblies Handling of anomalies consists of two tasks: Detecting anomalies in assemblies (Section 7.2) refers to computing the conditional probability distribution $P(\mathbb{B} \mid \mathcal{A}, n \in \mathcal{V}_{\mathcal{A}})$ over the decision if a node n in the given assembly \mathcal{A} is anomalous or regular. As this is obviously a binary classification problem on node-level, we applied the multi-class node classification model developed for the localized part recommendation given an extension node selected by an engineer for two classes.

Suggesting alternatives for anomalous parts (Section 7.3) can be understood as the task of computing a conditional probability distribution $P(\mathcal{T} \mid \bar{\mathcal{A}}, \bar{n} \in \mathcal{V}_{\bar{\mathcal{A}}})$ over all part types \mathcal{T} given an anomalous node \bar{n} from an anomalous assembly $\bar{\mathcal{A}}$, where the part types most suitable for replacing the anomalous node receive the highest probabilities. As this formulation is similar to the task of localized part recommendation for given extension node, we modeled it as a multi-class node classification problem in the same way as the localization task. However, instead of predicting part types to extend a given assembly at the given node, the model predicts an alternative type for the given *existing* node within the anomalous assembly.

Experimental Evaluation on Different GNN Architectures We instantiated four state-of-the-art graph neural network (GNN) architectures for our use cases in order to investigate the applicability of our modeling across different architectures. Further, we evaluated their performance on the three assembly datasets provided as part of the project KOGNIA to prove the generic nature of our approach. The experiments are reproducible, as the datasets have been published.

In all three use cases, the GNN models consistently outperformed the respective rule-based baseline models. While the latter were only capable of exactly memorizing existing assemblies, the graph neural networks successfully generalized knowledge about suitable part combinations to unseen yet similar situations. However, the choice of the specific GNN architecture usually had little influence on the prediction accuracy. In summary, all tasks were solved successfully by GNNs, and our experiments indicate that our approach can drastically reduce the search time spent by designers: As the wanted part was always one of ten recommended parts in more than 76% of the cases, the designer would not have to peruse the part catalogs for at least three out of four needed parts. Depending on the use case and dataset, our models even recommended the correct part in 97.5%, and detected anomalies correctly in 97.3% of the cases.

Tool Support Besides the interactive editor to define similarity constraints mentioned above as a standalone tool, we implemented a prototypical demonstrator for global part recommendation (Chapter 5) in the Festo Design Tool 3D [73] within the research project KOGNIA. Moreover, we integrated a similar model into the CAD system Autodesk Inventor [64] to conduct a user study as part of KOGNIA: Although trained on very few assemblies which were based on a similar number of part types, the top ten recommendations were still rated as beneficial in 78.3% of the cases. The design engineers described the system as convenient and effective, since the time-consuming search for needed parts is no longer necessary. Further, they noted that even suitable parts not previously known to the engineer were recommended, enabling the designer to integrate them into the assembly. The user study proves the effectiveness of our approach and that the experimental results can be trusted.

8.2 Lessons Learned

In this section, we aim to share our lessons learned and findings on good practices in the application of the presented methodology gained throughout this thesis as they could benefit other assembly use cases and datasets.

First, we would like to emphasize the necessary prerequisites and requirements for the application of the developed methods to custom datasets: The foundation of all approaches relies on the assumption that the assemblies are composed of a consistent set of part types. This means that the same types of parts must be identifiable across different assemblies. When applying our methodology to a custom assembly dataset, the set of part types will usually differ from ours, so they will not be represented by our embeddings. Consequently, the embeddings as well as the models have to be trained from scratch for differing assembly data. Furthermore, we operate on assemblies represented as undirected graphs, where nodes represent parts and edges represent connections between these parts as introduced in Section 4.1. Although assemblies can be naturally represented as graphs, the extraction into this format has to be implemented explicitly, as there is no matching standardized CAD format.

Superiority of ML Models In our first addressed use case, we compared our proposed ML-based approach with a frequency-based model leveraging classical AI techniques, which fell significantly short of the ML approach. Due to their ability of generalizing beyond exact subgraph pattern-matching, we recommend the use of ML models over classical approaches.

Part Embeddings Serve Multiple Purposes Any task in the context of assembly modeling needs an adequate representation for the different part types. Although there are various possibilities, we chose to solely rely on the part type itself since the provided information about parts is often varying across multiple part catalogs. In this setting, the part types can either be represented by a simple one-hot encoding or an embedding that can be learned based on their use in the assemblies, as presented in Section 4.3. Our approach learns the intended purpose of a part from its combinations with other parts in the assemblies. Pretraining low-dimensional embedding vectors has been beneficial for the actual tasks in our experiments, and they may offer further advantages: they can provide insights into the assembly data and can be used as a basis for a part taxonomy in order to improve a company’s part management. From an ML perspective, the reduced number

of model parameters is also desirable. Insofar it is feasible in terms of computational effort and time, we therefore advise pretraining of the part type embeddings. If features representing certain metadata or geometry are consistently available for the part types, they can be used in addition to part embeddings.

Additional Instances through Subgraphs In cases of a limited number of assemblies (potentially with numerous parts), it can be beneficial to augment the dataset in order to create more instances. All instance generation methods of the three use cases discussed have this data augmentation aspect in common: We extract all connected subgraphs of a given assembly graph (e.g., in Algorithm 1), which increased the number of partial assemblies by a factor between 10 and 200. As this proved to be an effective method to increase the performance of ML models, we recommend its application.

Start with the GraphSAGE Architecture Across all considered use cases, the four different GNN architectures achieved quite similar results, with GAT performing best for global part recommendation and GraphSAGE for localized recommendation and handling anomalies. Therefore, we would recommend to start experimenting with GraphSAGE, and afterward perform task-specific methodology adaptations. In case the results are not promising, other GNN architectures can be evaluated instead. Based on our data representation, other tasks in the context of assembly modeling can also be addressed using our methodology. We presented a modeling approach for learning tasks on graph-, node- as well as on edge-level, confer Chapter 5, Section 6.2 and Section 6.3, respectively, which can be used as a starting point for other tasks.

8.3 Outlook

Although we built a demonstrator for illustrating global part recommendation in the context of the project KOGNIA, the full integration including other use cases into existing CAD systems is still to be realized. For this reason, user studies to evaluate the usefulness of localized part recommendation as well as handling anomalies have yet to be carried out. As the experimental results for both of these use cases were similar or even better than for global part recommendation, we expect even better user feedback for those more specialized (because localized) tasks. However, the integration into a CAD system places further requirements on the recommendation and detection systems.

One aspect is that the part catalogs change over time, e.g., new parts are added. Part types are represented by embeddings pretrained according to their occurrence in assemblies. Consequently, no representation exists if a new part appears for the first time in an assembly, as it was not known during pretraining. In order to keep up part recommendations or anomaly detection, unknown part types included in the current assembly must be handled separately, e.g., by employing a (learned) default embedding for them.

Furthermore, the models in use must be kept up-to-date. In order to further improve recommendation performance, newly created assemblies as well as recent part catalog updates should be taken into account. This poses the need to retrain the models with the new data in the sense of online learning. Updating ML models is not trivial and therefore addressed by a whole research area: Practices for continuous software development (such as DevOps [29]) have been expanded by actions specific to machine learning which are generally referred to as *machine learning operations* (MLOps) [93].

To minimize additional workload on design experts, we focused on developing automated approaches without the need for human interaction like data labeling. However, involving experts can be very beneficial: First, data can be recorded when the systems are used by domain experts, which can then be used to further fine-tune the system. For instance, in the case of anomaly detection, a designer could mark parts recognized as anomalous as false positives. Furthermore, domain experts could improve the instance generation both for the recommendation and anomaly detection problems. They could create more realistic anomalies in assemblies – even going beyond single anomalous part types – and label them accordingly, resulting in potentially more accurate and useful detectors. Lastly, expert feedback can be used for training in a human-in-the-loop fashion which was also used for the currently very popular chatbot ChatGPT [108]: Two ML models were trained in an interplay, the actual chatbot and a model assessing the quality of the chatbot’s answers to a given prompt. In order to build the assessment model, humans ranked multiple answers to the same prompt, which served as training instances. Transferred to our setting, designers could evaluate part suggestions, i.e., specifically rank the individual suggestions. An assessment model trained on this data can then be used to continuously improve the recommendation system. In contrast to natural language, however, the number of experts for assembly modeling is significantly smaller, and realization would presumably be more difficult.

In the context of this thesis, only a limited number of assemblies were available, so that the trained part type embeddings can only represent a portion of their usage. However, if the embeddings were trained on a large number of designs, perhaps stemming from several companies, standard part manufacturers could even extend their catalogs with embeddings of their parts. In natural language processing, it is already common to share (word) embeddings trained by language models on a large corpus [51].

Finally, in terms of employing our assistance system to educate inexperienced designers, the recommendations should be taken with a grain of salt. An inherent property of machine learning is that one cannot rely on predictions being always correct – neither for training instances nor for unseen inputs. As this lowers trust in the validity of learned lessons, further steps have to be taken to address this problem: For example, the neural networks could be extended by a *policy layer*, i.e., a new final layer to restrict the outputs to a predefined rule set. The hard-coded rules in turn have to frame what validity of a part recommendation for a given assembly should mean in the application domain. Alternatively, one could integrate means from the research area for *explainable AI* (XAI) [118], which focuses on making decisions and behavior of AI models transparent and understandable to humans. In the context of part recommendations, this would allow explaining *why* a certain part type was predicted given the current assembly. Unfortunately, GNNs have received little attention from the XAI community so far. Nevertheless, there is promising work which could be applied to our models: For instance, SubgraphX [162] tries to explain which subgraph influenced the overall prediction the most. In our case, this maps directly to determining which existing parts (and combinations) from the current assembly led to a specific part type recommendation. This would increase comprehensibility of the recommendation and therefore the trust in the system, and would further support designers in understanding the correlation between parts.

Bibliography

- [1] Charu Aggarwal. *Recommender Systems: The Textbook*. Springer Cham, 2016.
- [2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, et al. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.
- [3] Marco Arazzi, Marco Cotogni, Antonino Nocera, and Luca Virgili. Predicting Tweet Engagement with Graph Neural Networks. In *Proceedings of the 2023 ACM International Conference on Multimedia Retrieval (ICMR)*, pages 172–180, 2023.
- [4] Volker Arnold, Hendrik Dettmering, Torsten Engel, and Andreas Karcher. *Product Lifecycle Management beherrschen: Ein Anwenderhandbuch für den Mittelstand*. Springer Berlin, Heidelberg, 2005.
- [5] László Babai and Ludik Kučera. Canonical Labelling of Graphs in Linear Average Time. In *20th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 39–46, 1979.
- [6] Dzmitry Bahdanau. Neural Machine Translation by Jointly Learning to Align and Translate. 2014. arXiv preprint. [arXiv:1409.0473](https://arxiv.org/abs/1409.0473).
- [7] Amir Bakarov. A Survey of Word Embeddings Evaluation Methods. 2018. arXiv preprint. [arXiv:1801.09536v1](https://arxiv.org/abs/1801.09536v1).
- [8] Sambaran Bandyopadhyay, Lokesh Nagalapatti, Saley Vishal Vivek, and Narasimha Murty. Outlier Resistant Unsupervised Deep Architectures for Attributed Network Embedding. In *Proceedings of the 13th International Conference on Web Search and Data Mining (WSDM)*, pages 25–33, 2020.
- [9] Stefan Bartsch. Konzepte und Strategien zur Erweiterung eines bestehenden Normteil- und Bauteilkataloges mit Wissensbasierten Werkzeugen zur Unterstützung des Konstrukteurs. Bachelor’s thesis, University of Applied Sciences Augsburg, Germany, 2017.
- [10] CADENAS GmbH: Adam Beck. 60 Years of CAD Infographic: The History of CAD since 1957. <https://partsolutions.com/60-years-of-cad-infographic-the-history-of-cad-since-1957/>, 2017. Accessed: 09/02/2024.

- [11] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28, 2015.
- [12] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A Neural Probabilistic Language Model. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 13, 2000.
- [13] Ruslan Bernijazov. KI-Marktplatz: Die digitale Plattform für KI im Engineering. <https://its-owl.de/projekte/ki-marktplatz/>. Accessed: 08/29/2024.
- [14] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Using PHiPAC to Speed Error Back-propagation Learning. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 4153–4156, 1997.
- [15] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, 2009.
- [16] Béla Bollobás. *Modern Graph Theory*. Graduate Texts in Mathematics 184. Springer-Verlag New York, 1 edition, 1998.
- [17] Michael Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. 2021. arXiv preprint. [arXiv:2104.13478](https://arxiv.org/abs/2104.13478).
- [18] Michael Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, et al. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.
- [19] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral Networks and Locally Connected Networks on Graphs. In *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*, 2014.
- [20] Hongyun Cai, Vincent Zheng, and Kevin Chen-chuan Chang. A Comprehensive Survey of Graph Embedding : Problems, Techniques and Applications. *IEEE transactions on knowledge and data engineering*, 30(9):1616–1637, 2018.
- [21] Ricardo Campello, Davoud Moulavi, and Joerg Sander. Density-Based Clustering Based on Hierarchical Density Estimates. In *Advances in Knowledge Discovery and Data Mining*, pages 160–172, 2013.
- [22] Augustin Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes rendus de l'Académie des Sciences*, 25(1847):536–538, 1847.
- [23] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM computing surveys (CSUR)*, 41(3), 2009.
- [24] Angel Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, et al. ShapeNet: An Information-Rich 3D Model Repository. 2015. arXiv preprint. [arXiv:1512.03012](https://arxiv.org/abs/1512.03012).
- [25] Xiang Chen, Shuming Gao, Song Guo, and Jing Bai. A Flexible Assembly Retrieval Approach for Model Reuse. *Computer-Aided Design*, 44(6):554–574, 2012.

- [26] Kathy Cheng and Alison Olechowski. Some (Team) Assembly Required: An Analysis of Collaborative Computer-Aided Design Assembly. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2021.
- [27] Parker Hannifin Corporation. Industrial Tube Fittings, Adapters and Equipment: Catalog 4300 PDF Version. Issue 2023/09, 2023.
- [28] James Cunningham, Timothy Simpson, and Conrad Tucker. An Investigation of Surrogate Models for Efficient Performance-Based Decoding of 3D Point Clouds. *Journal of Mechanical Design*, 141(12), 2019.
- [29] Patrick Debois, Jez Humble, Joanne Molesky, Eric Shamow, et al. Devops: A Software Revolution in the Making. *Journal of Information Technology Management*, 24(8):3–39, 2011.
- [30] Jonathan Dekhtiar, Alexandre Durupt, Matthieu Bricogne, Benoit Eynard, et al. Deep learning for big data applications in cad and plm – research review, opportunities and case study. *Computers in Industry*, 100:227 – 243, 2018.
- [31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2019.
- [32] Genevieve Diesing. How to Attract the Next Generation of Engineers. <https://www.qualitymag.com/articles/96163-how-to-attract-the-next-generation-of-engineers>, 2020. Accessed: 11/01/2020.
- [33] Yingtong Dou, Kai Shu, Congying Xia, Philip Yu, et al. User Preference-Aware Fake News Detection. In *Proceedings of the 44th international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2051–2055, 2021.
- [34] Vijay Prakash Dwivedi and Xavier Bresson. A Generalization of Transformer Networks to Graphs. 2020. arXiv preprint. [arXiv:2012.09699](https://arxiv.org/abs/2012.09699).
- [35] Ann-Christine Falck and Mikael Rosenqvist. What are the obstacles and needs of proactive ergonomics measures at early product development stages? – An interview study in five Swedish companies. *International Journal of Industrial Ergonomics*, 42(5):406–415, 2012.
- [36] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. 2019. arXiv preprint. [arXiv:1903.02428](https://arxiv.org/abs/1903.02428).
- [37] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A General Framework for Adaptive Processing of Data Structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, 1998.
- [38] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, et al. Modeling by Example. *ACM Transactions on Graphics (TOG)*, 23(3):652–663, 2004.
- [39] Carola Gajek, Alexander Schiendorfer, and Wolfgang Reif. A Recommendation System for CAD Assembly Modeling based on Graph Neural Networks. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2022.

- [40] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, et al. Computer-Aided Design as Language. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, pages 5885–5897, 2021.
- [41] Chen Gao, Yu Zheng, Nian Li, Yinfeng Li, et al. A Survey of Graph Neural Networks for Recommender Systems: Challenges, Methods, and Directions. *ACM Transactions on Recommender Systems*, 1(1), 2023.
- [42] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and Tensor-Flow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O’Reilly Media, Inc., 2nd edition, 2019.
- [43] Justin Gilmer, Samuel Schoenholz, Patrick Riley, Oriol Vinyals, et al. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 1263–1272, 2017.
- [44] CADENAS GmbH. 3D Supplier Parts: Each Engineer Wastes 59 working Days Per Year Searching for and Recreating Supplier Parts. <https://partsolutions.com/3d-supplier-parts-each-engineer-wastes-59-working-days-per-year-searching-for-and-recreating-supplier-parts/>. Accessed: 02/07/2020.
- [45] CADENAS GmbH. GeoSearch: Finding Instead of Searching. https://www.cadenas.de/tl_files/cadenas/Downloads/PDF/Produktflyer/EN/CADENAS_GEOsearch_Brochure_EN.pdf, 2013.
- [46] CADENAS GmbH. Research Project KOGNIA: Design Aided by AI. <https://www.cadenas.de/en/news/kognia-design-ai>, 2023. Accessed: 04/20/2023.
- [47] CADENAS GmbH. 3Dfindit: A Comparison of the Visual Search Options. <https://www.3dfindit.com/en/engiclopedia/the-possibilities-of-visual-search-in-comparison>, 2024. Accessed: 06/03/2024.
- [48] CADENAS GmbH. 3Dfindit: Find with a Combination of Search Methods. <https://www.3dfindit.com/en/enterprise/find-with-a-combination-of-search-methods>, 2024. Accessed: 08/26/2024.
- [49] CADENAS GmbH. 3Dfindit: The Central Platform for Engineers, Architects, Buyers & Creative Minds. <https://www.3dfindit.com/en/>, 2024. Accessed: 06/03/2024.
- [50] CADENAS GmbH. Manufacturer certified catalogs with Intelligent Engineering Data. <https://www.cadenas.de/en/products/partsolutions/intelligent-standards-supplier-part-catalog/available-manufacturer-catalogs>, 2024. Accessed: 08/21/2024.
- [51] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [52] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A New Model for Learning in Graph Domains. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 729–734, 2005.

- [53] Mihajlo Grbovic and Haibin Cheng. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 311–320, 2018.
- [54] Mihajlo Grbovic, Vladan Radosavljevic, Nemanja Djuric, Narayan Bhamidipati, et al. E-commerce in Your Inbox: Product Recommendations at Scale. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1809–1818, 2015.
- [55] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 855–864, 2016.
- [56] Xiaojie Guo and Liang Zhao. A Systematic Survey on Deep Generative Models for Graph Generation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(5):5370–5390, 2022.
- [57] Xiaoxiao Guo, Wei Li, and Francesco Iorio. Convolutional Neural Networks for Steady Flow Approximation. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 481–490, 2016.
- [58] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.
- [59] William Hamilton. *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer Cham, 2020.
- [60] Mark Harris. The top-5 error rate in the ImageNet Large Scale Visual Recognition Challenge has been rapidly reducing since the introduction of deep neural networks in 2012. <https://developer.nvidia.com/blog/mocha-jl-deep-learning-julia/image1/>, 2015. Accessed: 11/01/2020.
- [61] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer Feedforward Networks are Universal Approximators. *Neural networks*, 2(5):359–366, 1989.
- [62] Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, et al. GPT-GNN: Generative Pre-Training of Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1857–1867, 2020.
- [63] Michael Huber. Erweiterung eines Bauteilempfehlungssystems für CAD-Konstruktionen: kontextbasierte Anpassung der Empfehlungsanzahl mit RNNs und Transformer Modellen. Bachelor’s thesis, University of Augsburg, Germany, 2024.
- [64] Autodesk Incorporation. Autodesk Inventor: 3D Modeling Software for Designers and Engineers. <https://www.autodesk.com/products/inventor/overview>, 2024. Accessed: 06/01/2022.
- [65] Fortune Business Insights. 3D-Druck-Marktgröße, Wachstum, Anteil | Globaler Bericht. <https://www.fortunebusinessinsights.com/de/industrie-berichte/markt-f-r-3d-druck-101902>, 2024. Accessed: 08/20/2024.

- [66] Institute for Software & Systems Engineering, University of Augsburg. KOG-NIA: Machine-learning-based Recommender System for Mechanical Design. <https://www.uni-augsburg.de/en/fakultaet/fai/isse/projects/kognia/>, 2019. Accessed: 06/30/2022.
- [67] Giuseppe Italiano, Luigi Laura, and Federico Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. *Theoretical Computer Science*, 447:74–84, 2012.
- [68] Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, et al. AutoMate: A Dataset and Learning Approach for Automatic Mating of CAD Assemblies. *ACM Transactions on Graphics (TOG)*, 40(6), 2021.
- [69] Sergios Karagiannakos. Graph Neural Networks - An overview. https://theaisummer.com/Graph_Neural_Networks/, 2020. Accessed: 05/07/2022.
- [70] Andrea Karnyoto, Chengjie Sun, Bingquan Liu, and Xiaolong Wang. Augmentation and Heterogeneous Graph Neural Network for AAAI2021-COVID-19 Fake News Detection. *International Journal of Machine Learning and Cybernetics*, 13, 2022.
- [71] David Kasik, William Buxton, and David Ferguson. Ten CAD Challenges. *IEEE Computer Graphics and Applications*, 25(2):81–92, 2005.
- [72] Festo SE & Co. KG. Product Overview 2023. Issue 2023/07, 2023.
- [73] Festo Vertrieb GmbH & Co. KG. CAD configuration software Festo Design Tool 3D. https://www.festo.com/de/en/e/solutions/digital-transformation/digital-engineering-tools/festo-design-tool-3d-id_330026/, 2024. Accessed: 06/01/2022.
- [74] Schneider Technologies GmbH + Co. KG. Special Machine Construction. <https://www.schneider-technologies.eu/en/mechanical-engineering/special-machine-construction/>, 2024. Accessed: 09/01/2022.
- [75] Hwan Kim, Byung Suk Lee, Won-Yong Shin, and Sungsu Lim. Graph Anomaly Detection with Graph Neural Networks: Current Status and Challenges. *IEEE Access*, 10:111820–111829, 2022.
- [76] Hyeonwoo Kim, Hyungjoon Kim, Bumyeon Ko, Jonghwa Shim, et al. Two-stage Person Re-identification Scheme Using Cross-Input Neighborhood Differences. *The Journal of Supercomputing*, 78, 2022.
- [77] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014. arXiv preprint. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
- [78] Thomas Kipf and Max Welling. Variational Graph Auto-Encoders. 2016.
- [79] Thomas Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.
- [80] Thomas Norbert Kipf. *Deep Learning with Graph-Structured Representations*. PhD thesis, University of Amsterdam, Netherlands, 2020.

- [81] Jenny Knowdell. The Benefits and Disadvantages of Contract Manufacturing. *IQS Newsroom. Industrial Quick Search, Inc*, 2010.
- [82] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, et al. ABC: A Big CAD Model Dataset For Geometric Deep Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9601–9611, 2019.
- [83] Yann LeCun. Generalization and Network Design Strategies. *Connections in Perspective*, 1989.
- [84] Carola Lenzen, Vinzenz Löffel, and Wolfgang Reif. Handling Anomalies in CAD Assemblies: Detecting Anomalous and Suggesting Alternative Parts. In *International Conference on Computational Science and Computational Intelligence (CSCI)*, 2024.
- [85] Carola Lenzen and Wolfgang Reif. Localized Recommendation in Assembly Modeling: Employing GNNs for Targeted Part Placement. In *International Conference on Machine Learning and Applications (ICMLA)*, 2024.
- [86] Carola Lenzen, Alexander Schiendorfer, and Wolfgang Reif. Graph Machine Learning for Assembly Modeling. In *Learning on Graphs Conference (LoG)*, 2022.
- [87] Wei Li, Justin Matejka, Tovi Grossman, Joseph Konstan, et al. Design and Evaluation of a Command Recommendation System for Software Applications. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 18(2), 2011.
- [88] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, et al. Learning Deep Generative Models of Graphs. 2018. arXiv preprint. [arXiv:1803.03324](https://arxiv.org/abs/1803.03324).
- [89] Fengqi Liang, Huan Zhao, Yuhan Quan, Wei Fang, et al. Customizing Graph Neural Network for CAD Assembly Recommendation. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1746–1757, 2024.
- [90] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, et al. Efficient Graph Generation with Graph Recurrent Attention Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.
- [91] Katia Lupinetti, Jean-Philippe Pernot, Marina Monti, and Franca Giannini. Content-based CAD Assembly Model Retrieval: Survey and Future Challenges. *Computer-Aided Design*, 113:62–81, 2019.
- [92] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, et al. A Comprehensive Survey on Graph Anomaly Detection With Deep Learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(12):12012–12038, 2021.
- [93] Sasu Mäkinen, Henrik Skogström, Eero Laaksonen, and Tommi Mikkonen. Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help? In *IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, pages 109–112, 2021.

- [94] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. Image-Based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–52, 2015.
- [95] Leland McInnes, John Healy, and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. 2020. arXiv preprint. [arXiv:1802.03426](https://arxiv.org/abs/1802.03426).
- [96] Tomáš Mikolov, Greg Corrado, Kai Chen, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. 2013. arXiv preprint. [arXiv:1301.3781v3](https://arxiv.org/abs/1301.3781v3).
- [97] Tomáš Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, et al. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 26, 2013.
- [98] Erxue Min, Runfa Chen, Yatao Bian, Tingyang Xu, et al. Transformer for Graphs: An Overview from Architecture Perspective. 2022. arXiv preprint. [arXiv:2202.08455](https://arxiv.org/abs/2202.08455).
- [99] Pushkar Mishra, Aleksandra Piktus, Gerard Goossen, and Fabrizio Silvestri. Node Masking: Making Graph Neural Networks Generalize and Scale Better. 2020. arXiv preprint. [arXiv:2001.07524](https://arxiv.org/abs/2001.07524).
- [100] Kaichun Mo, Shilin Zhu, Angel Chang, Li Yi, et al. PartNet: A Large-scale Benchmark for Fine-grained and Hierarchical Part-level 3D Object Understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 909–918, 2019.
- [101] Christopher Morris, Martin Ritzert, Matthias Fey, William Hamilton, et al. Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks. In *Proceedings of the 33rd Conference on Artificial Intelligence (AAAI)*, pages 4602–4609, 2019.
- [102] Luis Müller, Mikhail Galkin, Christopher Morris, and Ladislav Rampásek. Attending to Graph Transformers. 2023. arXiv preprint. [arXiv:2302.04181](https://arxiv.org/abs/2302.04181).
- [103] Nicola Müller, Pablo Sánchez, Jörg Hoffmann, Verena Wolf, et al. Comparing State-of-the-art Graph Neural Networks and Transformers for General Policy Learning. 2024.
- [104] Kevin Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.
- [105] Lalit Narayan, Mallikarjuna Rao, and Mohammed Sarcar. *Computer Aided Design and Manufacturing*. PHI Learning Pvt. Ltd., 2008.
- [106] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.
- [107] Wilson Nyemba. *Computer Aided Design: Engineering Design and Modeling using AutoCAD*. CRC Press, 2022.

- [108] OpenAI. Introducing ChatGPT. <https://openai.com/index/chatgpt/>, 2022. Accessed: 04/28/2023.
- [109] Shikang Pang, Chunjing Xiao, Wenxin Tai, Zhangtao Cheng, et al. Graph Anomaly Detection with Diffusion Model-Based Graph Enhancement. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(21):23610–23612, 2024.
- [110] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.
- [111] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.
- [112] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 701–710, 2014.
- [113] Jiantao Pu, Yagnanarayanan Kalyanaraman, Subramaniam Jayanti, Karthik Ramanani, et al. Navigation and Discovery in 3D CAD Repositories. *IEEE Computer Graphics and Applications*, 27(4):38–47, 2007.
- [114] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving Language Understanding by Generative Pre-Training, 2018.
- [115] Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, et al. Recipe for a General, Powerful, Scalable Graph Transformer. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 14501–14515, 2022.
- [116] Nikhila Ravi, Jeremy Reizenstein, David Novotný, Taylor Gordon, et al. Accelerating 3D Deep Learning with PyTorch3D. 2020. arXiv preprint. [arXiv:2007.08501](https://arxiv.org/abs/2007.08501).
- [117] Frank Rosenblatt. *Perceptions and the Theory of Brain Mechanisms*. Spartan Books, 1961.
- [118] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2022.
- [119] Ideen Sadrehaghighi. *Computer Aided Design (CAD)*, 2022.
- [120] Fathi Salem. *Recurrent Neural Networks: From Simple to Gated Architectures*. Springer Cham, 2022.
- [121] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, et al. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.
- [122] Jürgen Schmidhuber and Sepp Hochreiter. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.
- [123] Ludwig Schneider. A Recommendation System for CAD Assembly Modeling Based on Graph Transformers. Master’s thesis, University of Augsburg, Germany, 2024.

- [124] Stephen Schoonmaker. *The CAD Guidebook: A Basic Manual for Understanding and Improving Computer-Aided Design*. CRC Press, 2002.
- [125] Dominik Schott. Intelligente Ähnlichkeitsanalyse von CAD Modellen. Master’s thesis, University of Augsburg, Germany, 2020.
- [126] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A Unified Embedding for Face Recognition and Clustering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 815–823, 2015.
- [127] Matthew Schultz and Thorsten Joachims. Learning a Distance Metric from Relative Comparisons. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 16, 2003.
- [128] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, et al. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 12(9):2539–2561, 2011.
- [129] Chence Shi, Minkai Xu, Zhaocheng Zhu, Weinan Zhang, et al. GraphAF: A Flow-based Autoregressive Model for Molecular Graph Generation. 2020. arXiv preprint. [arXiv:2001.09382](https://arxiv.org/abs/2001.09382).
- [130] Marina Sokolova and Guy Lapalme. A Systematic Analysis of Performance Measures for Classification Tasks. *Information Processing & Management*, 45(4):427–437, 2009.
- [131] Alessandro Sperduti and Antonina Starita. Supervised Neural Networks for the Classification of Structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.
- [132] Sakhinana Sagar Srinivas, Rajat Kumar Sarkar, and Venkataramana Runkana. Hypergraph Learning Based Recommender System for Anomaly Detection, Control and Optimization. In *IEEE International Conference on Big Data*, 2022.
- [133] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [134] Yunlei Sun, Kangping Liu, Yucong Li, and Dalin Zhang. Towards an Open-Source Industry CAD: A Review of System Development Methods. *Tehnički vjesnik*, 29(6):2127–2136, 2022.
- [135] University of Augsburg Teresa Grunwald. Mit KI schneller zum fertigen Produkt. <https://www.uni-augsburg.de/de/campusleben/neuigkeiten/2023/01/31/mit-ki-schneller-zum-fertigen-produkt/>, 2023. Accessed: 02/01/2023.
- [136] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, et al. Understanding Over-Squashing and Bottlenecks on Graphs via Curvature. 2021. arXiv preprint. [arXiv:2111.14522](https://arxiv.org/abs/2111.14522).
- [137] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3), 2007.

- [138] André Kruschke und Helge Röpcke. *Graphen und Netzwerktheorie*. Hanser Fachbuchverlag, 2015.
- [139] Laurens Van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(11), 2008.
- [140] Flavian Vasile, Elena Smirnova, and Alexis Conneau. Meta-Prod2Vec - Product Embeddings Using Side-Information for Recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 225–232, 2016.
- [141] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, et al. Attention is All you Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.
- [142] Verein Deutscher Ingenieure (VDI). *Knowledge Management for Engineering - Fundamentals, Concepts, Approach*. Engl. VDI-Gesellschaft Produkt- und Prozessgestaltung, 2009.
- [143] Petar Veličković. *The Resurgence of Structure in Deep Neural Networks*. PhD thesis, University of Cambridge, England, 2019.
- [144] Petar Veličković, Arantxa Casanova, Pietro Liò, Guillem Cucurull, et al. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.
- [145] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, et al. GraphGAN: Graph Representation Learning With Generative Adversarial Nets. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.
- [146] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, et al. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [147] Boris Weisfeiler and Andrei Leman. The Reduction of a Graph to Canonical Form and the Algebra Which Appears Therein. *nti, Series*, 2(9):12–16, 1968.
- [148] Paul Werbos. Applications of Advances in Nonlinear Sensitivity Analysis. In *System Modeling and Optimization*. Springer, 1982.
- [149] Sigurd Wichter. *Wissenstransfer zwischen Experten und Laien: Umriss einer Transferwissenschaft*. Lang, 2001.
- [150] Karl Willis, Yewen Pu, Jieliang Luo, Hang Chu, et al. Fusion 360 Gallery: A Dataset and Environment for Programmatic CAD Construction from Human Design Sequences. *ACM Transactions on Graphics (TOG)*, 40(4), 2021.
- [151] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, et al. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 38–45, 2020.
- [152] David Wolpert. The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

- [153] Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6772–6782, 2021.
- [154] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, et al. Graph Neural Networks in Recommender Systems: A Survey. *ACM Computing Surveys*, 55(5), 2022.
- [155] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, et al. A Comprehensive Survey on Graph Neural Networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.
- [156] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, et al. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 2048–2057, 2015.
- [157] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? 2018. arXiv preprint. arXiv:1810.00826.
- [158] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, et al. Do Transformers Really Perform Badly for Graph Representation? In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, pages 28877–28888, 2021.
- [159] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 974–983, 2018.
- [160] Soyoung Yoo, Sunghee Lee, Seongsin Kim, Kwang Hyeon Hwang, et al. Integrating Deep Learning into CAD/CAE System: Generative Design and Evaluation of 3D Conceptual Wheel. *Structural and Multidisciplinary Optimization*, 64(4):2725–2747, 2021.
- [161] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, et al. GraphRNN : Generating Realistic Graphs with Deep Auto-regressive Models. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 5708–5717. PMLR, 2018.
- [162] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, et al. On Explainability of Graph Neural Networks via Subgraph Explorations. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pages 12241–12252, 2021.
- [163] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, et al. Deep Sets. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.
- [164] Fengbin Zhang, Haoyi Fan, Ruidong Wang, Zuoyong Li, et al. Deep Dual Support Vector Data Description for Anomaly Detection on Attributed Networks. *International Journal of Intelligent Systems*, 37(2):1509–1528, 2022.
- [165] Hao Zhang, Mufei Li, Minjie Wang, and Zheng Zhang. Understand Graph Attention Network. <https://www.dgl.ai/blog/2019/02/17/gat.html>, 2022. Accessed: 04/06/2022.

- [166] Shijie Zhang, Hongzhi Yin, Tong Chen, Quoc Viet Nguyen Hung, et al. GCN-Based User Representation Learning for Unifying Robust Recommendation and Fraudster Detection. In *Proceedings of the 43rd international ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 689–698, 2020.
- [167] Zhibo Zhang, Prakhar Jaiswal, and Rahul Rai. FeatureNet: Machining Feature Recognition based on 3D Convolution Neural Network. *Computer-Aided Design*, 101:12–22, 2018.
- [168] Lingxiao Zhao and Leman Akoglu. On Using Classification Datasets to Evaluate Graph Outlier Detection: Peculiar Observations and New Insights. *Big Data*, 11(3):151–180, 2023.
- [169] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, et al. Graph Neural Networks: A Review of Methods and Applications. *AI open*, 1:57–81, 2020.
- [170] Jiong Zhu, Yujun Yan, Lingxiao Zhao, Mark Heimann, and others. Beyond Homophily in Graph Neural Networks: Current Limitations and Effective Designs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 28877–28888, 2020.
- [171] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, et al. AliGraph: A Comprehensive Graph Neural Network Platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.