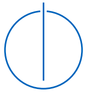


Fakultät für Informatik
Technische Universität München

TUM

Relational Representation Learning Beyond Simple Graphs

Frederik Gerzer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende:
Prof. Dr. Anne Brüggemann-Klein

Prüfende der Dissertation:
1. Prof. Dr.-Ing. habil. Alois Knoll
2. Assistant Prof. David Rolnick, Ph. D.

Die Dissertation wurde am 21.06.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.10.2021 angenommen.

Abstract

Graph Neural Network (GNN) models are usually applied to simple graphs, i.e. graphs with only node features and a fixed graph topology. In this thesis, we argue for the need to expand them to more complex graph types. It is organized in four main topics:

First, we benchmark common GNN architectures on simple graphs. From this, we draw three conclusions: GNN models show a large variability in their result depending on the random initialization of their parameters. Graph Convolutional Network (GCN) layers show good performance on a large number of datasets, which makes them a good starting point for constructing an initial model. Lastly, GNN models do not yet profit from a large depth.

Next, we introduce a model modifying input graph topology. EdgePool is a local and hard pooling layer. We show that it generally outperforms other pooling methods and can be easily integrated in both graph- and node-level prediction problems. We also show that it carries a 50% computational time penalty, but that it improves memory usage on larger graphs.

As our first application, we explore the task of vehicle prediction on the highway. This introduces edge features to the graph. We show how the changes we introduce into two GNN models improve prediction quality, with the best model reducing prediction error by 30%. We also evaluate construction methods for the scene representation, and show how to make the model more interpretable.

Our second application is on high-voltage power grid control. By modelling a power grid as a graph—and thereby departing from the simple graph assumption by having both complex edge features and heterogeneous nodes—we are able to predict the output of a classical optimization algorithm. This model produces results four orders of magnitude faster than the original solver. By using the model's output to warm-start the optimization algorithm, we improved the latter's runtime by a factor of 3.8× while keeping the optimizer's guarantees for a feasible solution. This is crucial for deploying such techniques to real-life power grids.

Zusammenfassung

GNN-Modelle werden üblicherweise entwickelt um sie auf einfache Graphen, d.h. solche nur mit Knoteneigenschaften und einer konstanten Topologie, anzuwenden. In dieser Arbeit argumentieren wir, dass die Verarbeitung komplexerer Graphen notwendig ist. Sie ist in vier Themen organisiert:

Zuerst evaluieren wir häufig verwendete GNN-Architekturen auf einfachen Graphen. Daraus ziehen wir drei Schlüsse: Die Ergebnisse von GNN-Modellen variieren sehr stark, je nach der zufälligen Initialisierung ihrer Parameter. GCN-Layer erreichen gute Ergebnisse über viele Datensätze, was sie zu einem guten Startpunkt für die Architektursuche macht. Und GNN-Modelle profitieren noch nicht stark von einer größeren Tiefe.

Im nächsten Abschnitt entwickeln wir ein Modell zur Modifikation von Graphtopologie. EdgePool ist eine lokale und harte Pooling-Methode. Wir zeigen, dass EdgePool bessere Ergebnisse erreicht als andere Pooling-Methoden und dass es einfach in existierende Modelle zur Knoten- und Graphprädiktionen integriert werden kann. Wir zeigen, dass EdgePool etwa 50% mehr Laufzeit benötigt, aber dass es die Speichereffizienz von GNN-Modellen verbessert.

Unsere erste Anwendung ist die Prädiktion der Fahrzeugbewegungen auf der Autobahn. Für diese Anwendung führen wir Kanteneigenschaften ein. Wir zeigen, dass GNN-Modelle mit von uns eingeführten Anpassungen den Prädiktionsfehler verglichen mit Modellen ohne Interaktionen um bis zu 30% reduzieren. Wir evaluieren außerdem Konstruktionsmethoden für die Repräsentation und zeigen, wie man das Modell interpretierbarer machen kann.

Unsere zweite Anwendung ist die Kontrolle von Hochspannungsnetzen. Indem wir ein Hochspannungsnetz als Graph modellieren, und damit sowohl komplexere Kanteneigenschaften als auch unterschiedliche Knotentypen einführen, können wir ein GNN-Modell verwenden um das Ergebnis eines klassischen Optimierers hervorzusagen. Unser GNN-Modell berechnet Ergebnisse vier Größenordnungen schneller als der Optimierer. Indem wir die Ergebnisse des Modells als Startpunkte für den Optimierer verwenden können wir die Laufzeit um einen Faktor von 3.8× verbessern während wir gleichzeitig die Legalität des Kontrollsignals garantieren. Dies ist absolut notwendig, um solche Techniken auf echten Hochspannungsnetzen einzusetzen.

Contents

Abstract iii

Zusammenfassung v

Contents vii

1 Introduction 1
1.1 Neural Networks on Fixed Data Structures . . . 2
1.2 Graph Neural Networks . . . 3
1.3 Beyond Simple Graphs . . . 3
1.4 Thesis Outline and Contributions . . . 4

2 Background 7
2.1 Definitions . . . 7
2.2 Representing a Graph . . . 9
2.3 Spectral Graph Theory . . . 9
2.3.1 Scalar and Tangent Vector Fields on Graphs . . . 10
2.3.2 Differential and Divergence . . . 10
2.3.3 Laplacian Eigenfunctions . . . 11
2.3.4 Fourier Analysis of the Graph Laplacian . . . 12
2.3.5 Convolutions on Graphs . . . 12
2.4 Learning Tasks on Graphs . . . 13
2.4.1 Node-Level Tasks . . . 14
2.4.2 Edge-Level Tasks . . . 14
2.4.3 Graph-Level Tasks . . . 15
2.4.4 Hybrid and Other Cases . . . 15
2.5 Graph Neural Networks Paradigms . . . 15
2.5.1 Recurrent Graph Neural Networks . . . 16
2.5.1.1 Recurrent Graph MLP . . . 16
2.5.1.2 Gated Graph Sequence Neural Networks . . . 16
2.5.1.3 Stochastic Steady-State Embedding . . . 17
2.5.1.4 Conclusion . . . 17
2.5.2 Spectral Graph Neural Networks . . . 17
2.5.2.1 Spectral Networks . . . 17
2.5.2.2 ChebNet . . . 18
2.5.2.3 CayleyNet . . . 19
2.5.2.4 Adaptive Graph Convolutional Networks . . . 20
2.5.2.5 Conclusion . . . 20

2.5.3 Spatial Graph Neural Networks . . . 20
2.5.3.1 Graph Convolutional Networks (GCNs) . . . 21
2.5.3.2 Message Passing Neural Networks (MPNNs) . . . 22
2.5.3.3 GraphSAGE . . . 22
2.5.3.4 Graph Attention Network (GAT) . . . 23
2.5.3.5 Graph Networks . . . 23
2.5.3.6 WL-Graph-Isomorphy and the GIN Model . . . 24
2.5.3.7 Conclusion . . . 25
2.5.4 Computational Implementation . . . 25
2.6 Conclusion . . . 26

3 Simple Graphs **27**
3.1 The Datasets . . . 27
3.1.1 Semi-Supervised Node Classification Datasets . . . 28
3.1.2 Supervised Graph Classification Datasets . . . 28
3.1.2.1 The *proteins* Dataset . . . 28
3.1.2.2 The *nci1* Dataset . . . 29
3.1.2.3 The *imdb* Dataset . . . 29
3.1.2.4 The *rdt-b* and *rdt-12k* Datasets . . . 29
3.2 GNN Architecture . . . 29
3.2.1 Building Blocks . . . 30
3.2.2 Tweaks . . . 31
3.2.3 Baseline Model . . . 32
3.3 Experiments and Discussion . . . 33
3.3.1 **Q1**: How Useful is the Inclusion of Graph Information? . . . 34
3.3.2 **Q2**: How Reproducible Are GNN Models? . . . 35
3.3.3 **Q3**: How Do Different GNN Layers Perform? . . . 37
3.3.4 **Q4**: How Do We Encode Node Features? . . . 38
3.3.5 **Q5**: How Do We Global-Pool Graphs? . . . 39
3.3.6 **Q6**: How Do We Construct the Final Graph Output? . . . 40
3.3.7 **Q7**: How Deep should GNN models be? . . . 41
3.3.8 **Q8**: Which Tweaks Improve GNN Performance? . . . 42
3.3.9 **Q9**: Is it Helpful to Separate Processing and Propagation? . . . 45
3.4 Conclusion . . . 46

4 Modifying Graph Topology **49**
4.1 Motivation . . . 49
4.2 Other Pooling Methods . . . 50
4.2.1 DiffPool . . . 50
4.2.2 TopKPool . . . 51
4.2.3 SAGPool . . . 51
4.3 EdgePool . . . 52
4.3.1 Choosing Edges . . . 52
4.3.2 Computing New Node Features . . . 53

4.3.3 Integrating Edge Features . . . 53
4.3.4 Unpooling EdgePool . . . 55
4.3.5 Computational Performance . . . 55
4.4 Experiments and Discussion . . . 55
4.4.1 General Setup and Training . . . 55
4.4.2 **Q1**: Does EdgePool Outperform Alternative Pooling Approaches? 57
4.4.3 **Q2**: Can EdgePool Be Integrated into Existing Architectures? . . 58
4.4.4 **Q3**: Can EdgePool be Used For Node Classification? . . . 59
4.4.5 **Q4**: How Does EdgePool Impact Performance? . . . 62
4.5 Conclusion . . . 63

5 Edge Features 65
5.1 Motivation . . . 65
5.2 Traffic Participant Prediction from a Graph . . . 67
5.2.1 Adapting Graph Convolutional Networks . . . 67
5.2.2 Adapting Graph Attention Networks . . . 67
5.2.3 Graph and Feature Construction . . . 68
5.3 Experiments . . . 69
5.3.1 Datasets . . . 69
5.3.2 Baselines . . . 70
5.3.3 Model Configuration . . . 70
5.3.4 Performance Measure . . . 71
5.3.5 Experimental Procedure . . . 71
5.4 Results and Discussion . . . 71
5.4.1 **Q1**: Which of Our Adaptations to GNNs Are Necessary? . . . 72
5.4.2 **Q2**: How do We Construct an Interaction Graph? . . . 73
5.4.3 **Q3**: Does a Graph Model Increase Prediction Quality? . . . 73
5.4.4 Conclusion . . . 75
5.5 Inspecting GNNs for Traffic Prediction . . . 75
5.5.1 Creating Saliency Graphs . . . 76
5.5.1.1 Saliency Maps . . . 76
5.5.1.2 Computing Gradients . . . 77
5.5.1.3 Summarizing Feature-Wise Gradient . . . 77
5.5.1.4 Plotting the Saliency Graph . . . 77
5.5.2 Results and Discussion . . . 78
5.5.2.1 Influence of the Ego Vehicle . . . 78
5.5.2.2 The Effect of Edge Features . . . 79
5.5.2.3 Analyzing the Influence of Neighbours . . . 79
5.5.2.4 Saliencies for Specific Scenarios . . . 81
5.5.2.5 Interpreting a Complete Traffic Scene . . . 81
5.5.3 Conclusion . . . 82
5.6 Conclusion . . . 82

6 Heterogeneous Nodes **83**
6.1 Introduction . . . 83
6.1.1 Motivation . . . 83
6.1.2 Approach . . . 84
6.2 Simulating and Controlling Power Grids . . . 85
6.2.1 Power Grid Equations . . . 85
6.2.2 Power Flow . . . 86
6.2.3 Optimal Power Flow . . . 87
6.3 Methodology . . . 88
6.3.1 Modelling the Power Grid . . . 88
6.3.2 GNN Models for Power Grids . . . 88
6.3.2.1 Independent Model . . . 91
6.3.2.2 Heterogeneous Model . . . 91
6.3.2.3 Summarized Features Model . . . 91
6.3.2.4 Summarized Embeddings Model . . . 92
6.3.2.5 Separate Components Model . . . 92
6.3.3 Ensuring Feasibility . . . 93
6.4 Experiments . . . 93
6.4.1 Datasets and Experimental Setup . . . 93
6.4.2 **Q1**: Which of the Adapted GNN Models Performs Best? . . . 96
6.4.2.1 Experimental Setup . . . 96
6.4.2.2 Results and Discussion . . . 96
6.4.2.3 Conclusion . . . 97
6.4.3 **Q2**: Can an ACOPF Solver be Warm-Started by a GNN Model? . . . 97
6.4.3.1 Experimental Setup . . . 97
6.4.3.2 Results and Discussion . . . 97
6.4.3.3 Conclusion . . . 99
6.5 Conclusion . . . 99

7 Conclusion & Outlook **101**
7.1 Summary . . . 101
7.2 Conclusion . . . 102
7.2.1 Modelling Tasks as Graphs is Powerful . . . 103
7.2.2 Recommendations for Exploring New Problems . . . 103
7.2.3 Building ML Models for Critical Infrastructure . . . 104
7.3 Outlook . . . 104

Publications **105**

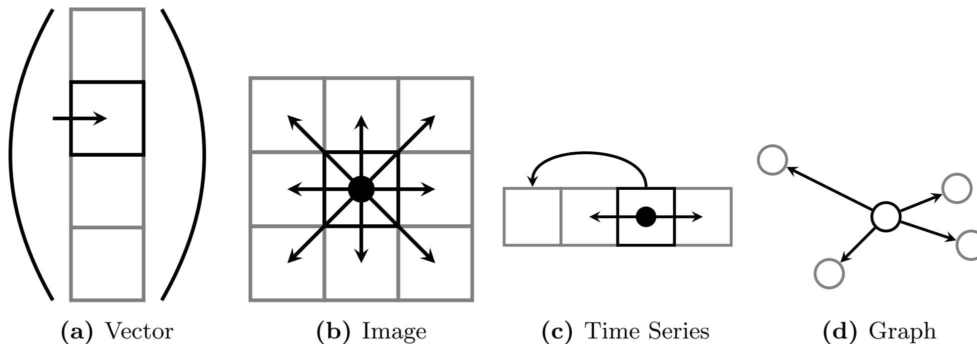
Bibliography **107**

1 Introduction

Machine learning (ML) methods and particularly deep learning methods have become incredibly successful in the last decade. Deep learning techniques have been deployed in many applications, powering among others search, translation, image recognition, game playing, and speech recognition technologies. All of these have a regular structure in common—in vectors, time series, and images, there is a clear notion of position and relative position, allowing the concept of ”preceding” time step for time series or relative pixel positions for images (see also Fig. 1.1).

And while many different inputs such as images or time series can be easily modelled using these regular structures, many more cannot: The world is filled with objects defined not by their position on a grid but by their relationship to other objects. Each object might be related to many other objects or to none, something that is exceedingly difficult to model using regular structures. Yet without modelling these objects and their relationships, we cannot learn on them.

For ML models, just like with many algorithmic problems, “the key [. . .] is to think of them in terms of graphs.” (Skiena 2008, p. 146). Graphs provide a natural way of modelling objects and their relationships as nodes and edges but require new deep learning techniques to learn on them. Graph Neural Networks (GNNs) are a category of models that allow us to learn on graphs, taking into account not just node features but also graph topology.



(a) Vector **(b)** Image **(c)** Time Series **(d)** Graph

Fig. 1.1: Neighbour information in different data types. (a) In vectors, each entry represents a fixed feature. (b) In images, spatial relations uniquely identify the eight neighbouring pixels and can be chained to cover all pixels. (c) In time series, temporal relations uniquely identify the preceding and following timestep and can be chained to identify all timesteps. (d) In graphs, neighbouring nodes cannot be uniquely identified by topology.

Yet, most GNN models and benchmark datasets are restricted to what we call simple graphs, featuring nodes features and a fixed graph topology. While these can model many problems, even more remain out of reach. In this thesis, we aim to break through these limitations by modifying graph topology and by introducing tasks that require reasoning both on edge features and on heterogeneous nodes.

This chapter first introduces neural networks as applied to fixed data structures (Section 1.1), followed by a short overview over GNN models (Section 1.2). We then introduce in more detail our distinction between simple and complex graphs (Section 1.3). Finally, we outline the remainder of the thesis (Section 1.4).

1.1 Neural Networks on Fixed Data Structures

The great successes of modern deep learning have been achieved on data with a fixed and known structure such as images or sequences. In these, there is a clear notion of position—an entry in a vector always represents the same feature, the character in a specific relative position is always the one preceding the current character, and the pixel in a specific relative position is always one pixel left of the current one (see Fig. 1.1). For many applications, the samples already exist in such forms:

Vectors have fixed entries, and each entry always represents the same feature.

Time Series have a fixed order, and one can build relative coordinates by chaining references to the previous and next timestep.

Images are best represented in matrix form, in which there is a clear notion of relative position. Each of the eight neighbouring pixels uniquely identified by relative coordinates.

Videos and other higher-dimensional data can be addressed similarly, referencing to previous and next relative positions for both spatial and temporal dimensions.

This fixed order allows us to easily build algorithms that operate on these data structures. For vectors, we can use Multi-Layer Perceptrons (MLPs), which successively and non-linearly transform input features into output features. MLPs are less useful when operating on data with a more regular structure, and so both Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) models are used on these. CNN layers convolve the same operation—usually a matrix multiplication with the weights followed by a non-linearity—on each sample entry. RNN models keep a hidden state representing information gathered from preceding timesteps and use this to produce the current timestep's output.

But while these methods are unquestionably successful, many more applications feature samples that do not have a fixed structure. If our data consists only of objects and their relationships (see Fig. 1.1d), none of these methods work: MLPs require a fixed-size input, while we may have varying numbers of objects and relationships. RNN models require a clear notion of previous and next timestep, while we have only an unordered

set of relationships for each object. And CNN models require relative positions and a grid-like structure, while we have only the existence of relationships.

Instead, it would fall to developers to manually construct a representation of the data such that a fixed structure for ML methods can be created. Depending on the task, this might be straightforward—or we might suffer from significant information loss.

1.2 Graph Neural Networks

Many of the object-relationship tasks are not easily condensed into a fixed representation such as required by standard ML models. However, graphs are a natural model for these tasks, modelling objects as nodes and the existence of relationships as edges. These might be scientists and coauthorships, traffic participants and interactions between them, atoms and chemical bonds, or power grid junctions and connecting power lines. Like images or text, it is clear that each of the objects should be treated identically and we should therefore apply the same model to each object. Like with CNN or RNN models, this should lead to better generalization, better accuracy, and more flexible models. Yet unlike images or text, any such object might have variable numbers of interactions. A hydrogen atom forms only one bond; a carbon atom four. A grid junction might connect a single generator to a single power line; it might also be an electrical substation connecting a dozen lines.

The subfield of graph representation learning promises models that can act and learn on graph-structured data. Generally, they update each node's features using the same transformation function. Depending on the model, these transformation functions might take only the node itself, or the node and its neighbours, or the complete graph topology into account. We introduce a number of these methods in Section 2.5 and benchmark several in Chapter 3.

And yet, while GNN models solve the problem modelling objects that have a variable numbers of unordered relationships with other objects, most GNN methods remain inherently limited: They only process node features and the existence of edges between them. There are no edge features, graph topology does not change, and nodes are generally assumed to be of the same type.

1.3 Beyond Simple Graphs

Motivated by these shortcomings, we aim to move beyond simple graphs, i.e. those with a fixed graph topology and only node features (Fig. 1.2a). We consider three specific departures from this simple graph model:

- We develop a method for modifying graph topology (see Fig. 1.2b and Chapter 4). This learns how to pool nodes together, combining both their features and edges.
- We apply GNN models to a traffic prediction problem, in which edge features play a crucial role (see Fig. 1.2c and Chapter 5). We also introduce a simple way to inspect node importance in GNN models.

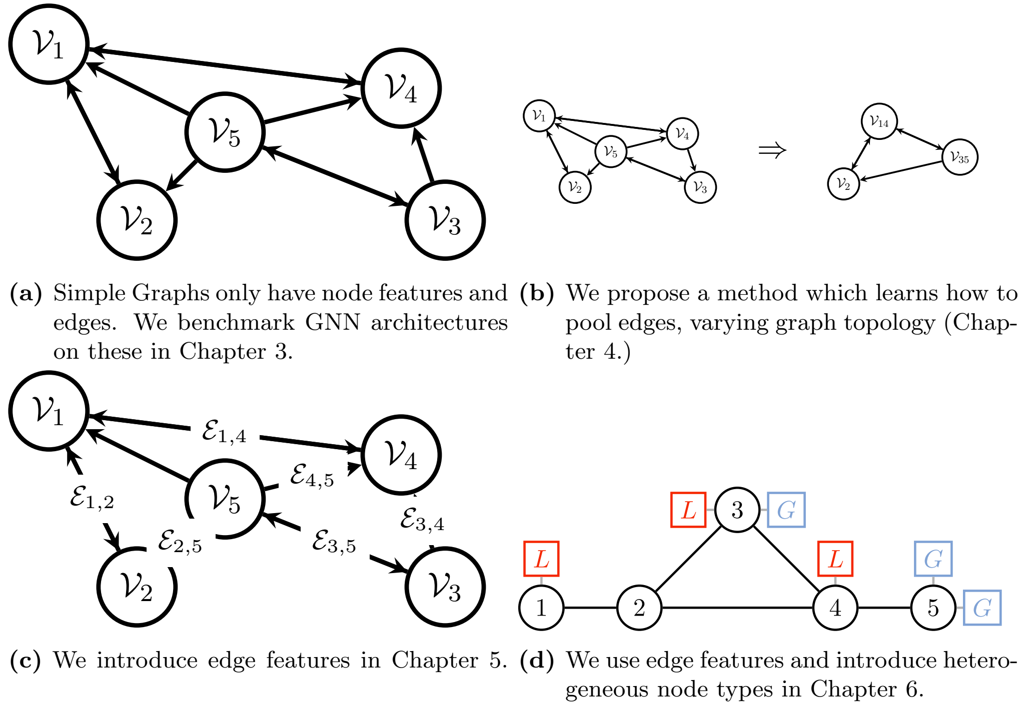
- On the task of power grid control, we not only need to use edge features but also to find a way for treating heterogeneous types of nodes (see Fig. 1.2d and Chapter 6). We also integrate GNN models with classical solvers to guarantee solution feasibility.

In summary, we introduce three departures from the more commonly-used model of simple graphs towards real-world applicability of GNN models.

1.4 Thesis Outline and Contributions

This thesis consists of seven chapters:

- **Chapter 1** has outlined both the need for GNN models and the shortcomings of current models.
- **Chapter 2** provides the reader with the necessary background. This includes how to represent graphs (Section 2.2), the necessary spectral graph theory which many methods rely on to apply the concept of convolution to graphs (Section 2.3), how to formulate graph problems (Section 2.4), and finally a number of GNN methods (Section 2.5). The latter include recurrent graph networks, and spectral and spatial graph networks. We also introduce theoretical analysis of GNN



(a) Simple Graphs only have node features and edges. We benchmark GNN architectures on these in Chapter 3.

(b) We propose a method which learns how to pool edges, varying graph topology (Chapter 4.)

(c) We introduce edge features in Chapter 5.

(d) We use edge features and introduce heterogeneous node types in Chapter 6.

Fig. 1.2: Graph types used in this dissertation.

models (Section 2.5.3.6) and special computational considerations necessary for implementing GNN models (Section 2.5.4).

- **Chapter 3** compares a large number of the GNN models from Section 2.5 and their tweaks on simple graphs, i.e. those with only node features and feature-less connections between nodes. We do this on three semi-supervised node classification tasks and five graph classification tasks. We are particularly interested in how to build GNN models and explore among others depth, layer types, and encoder and decoder types. We also explore the usage of graph features and reproducibility of GNN training.

- **Chapter 4** repeals the first restriction of simple graphs: We introduce a pooling layer which modifies graph topology, allowing further layers to act on groups of nodes instead of single nodes. EdgePool is a local and hard pooling layer, allowing simple disaggregation and unpooling for node prediction tasks. We explore EdgePool in comparison to other pooling methods, plug it into both node-level and graph-level prediction, and measure its performance impact.

- **Chapter 5** then repeals another of the restrictions of simple graphs and introduces simple edge features. On the task of traffic prediction, we show that representing a traffic scene as a graph and applying GNN models allows for powerful and expressive models to be applied while keeping model complexity limited. We are primarily interested in the usefulness of our adaptations, how to construct the interaction graph, and whether using a graph to model interactions improves prediction quality.

- **Chapter 6** applies GNN models to the far more complex problem of power grid optimization. We model a power grid, featuring heterogeneous components as nodes and diverse and complex edge features. We also combine these models with a classical optimization algorithms with the aim to improve the latter's runtime while still guaranteeing feasible control inputs. Since infeasible control inputs for power grids can result in brownouts or blackouts, guaranteeing feasibility is crucial to deploying such algorithms in the real world. We are primarily interested in how to construct the model architecture and in the prediction quality of our method.

- Lastly, in **Chapter 7**, we summarize this thesis, draw the main conclusions from our experiments, and close with an outlook based on averting the simple graph paradigm.

2 Background

In this chapter, we go over the necessary background for the remainder of the thesis. We first introduce the notation used (Section 2.1), followed by how to represent a graph in a computer (Section 2.2). In Section 2.3, we introduce spectral graph theory with an eye to adapting the convolution theorem to graphs. This allows us to extend the notion of convolution to graphs, which is used by several GNN models. Section 2.4 introduces different types of tasks based on graphs. Lastly, Section 2.5 goes over different GNN models, presenting recurrent graph networks, spectral and spatial GNN models, theoretical analysis of the latter, and several tweaks to GNN models.

2.1 Definitions

A graph G is a tuple $G = (V, E)$, with V being the set of nodes and E being the set of edges. We denote a single node as $v \in V$, using subscripts to distinguish between nodes when necessary. We generally assume a single fixed but arbitrary node order. Edges are pair-wise connections between nodes; we refer to an edge from node v_i to v_j as $e_{ij} = (v_i, v_j) \in E$. Nodes that are connected by an edge are also called adjacent; a node is incident to any edge that connects it. We allow self-loops (i.e. e_{ii}) to exist, but do not allow multigraphs (i.e. with multiple edges e_{ij}). GNN methods generally treat undirected graphs (i.e. $\forall e_{ij} \in E : \exists e_{ij} \Leftrightarrow \exists e_{ji}$) as directed graphs.

We refer to the degree of a node v as $\deg(v)$; it is the number of adjacent nodes. In matrix-notation, we write might use the degree matrix $\mathbf{D}$, which is the diagonal matrix containing the node degrees. $|V|$ and $|E|$ are the number of nodes and edges in a graph respectively.

A walk is an alternating sequence of nodes and edges $v_0, e_1, v_1, \ldots v_{n-1}, e_n, v_n$, such that each edge e_n is incident to v_{n-1} and v_n. The walk's length is the number of edges it contains. If all vertices and edges of a walk are distinct, it is called a path.

Two nodes are connected if a path exists between them[1]; a graph is connected if all node pairs are connected.

The neighbourhood of a node is the set of all adjacent nodes. We also use the n-hop neighbourhood $\mathcal{N}_n(v)$ of a node v, which is the set of all nodes connected to v by a path of at most length n. More constrained, the strict n-hop neighbourhood $\hat{\mathcal{N}_n}(v)$ of said node is the set of all nodes connected by a path of exactly length n. Less constrained, $\mathcal{N}^+(v)$ refers to the neighbourhood of node v and the node itself.

[1] In a directed graphs, both directions of a path have to exist.

Table 2.1: Symbols used in this thesis.

	Symbol	Meaning
Graph	v_i, V	Node i, all nodes
	e_{ij}, E	Edge from node v_i to v_j, all edges
	G	Graph
	$\mathcal{N}(v)$, $\mathcal{N}_n(v_i)$	Single-hop and n-hop neighbourhood of node v_i
	$\hat{\mathcal{N}}_n(v_i)$	Strict n-hop neighbourhood
	$\mathcal{N}^+(v_i)$	neighbourhood plus node v_i
	$\mathbf{A}$	Adjacency matrix
	$\lvert V\rvert$ $\lvert E\rvert$	Number of nodes and edges
	$\deg(v_i)$, $\mathbf{D}$	Degree of node v_i; degree matrix
	$\odot$	Elementwise multiplication
	$\tilde{\square}$, $\square^*$	Complex number, complex conjugate
Spectral	$\mathcal{L}$, $\mathcal{L}_{\text{sym}}$	Graph Laplacian, normalized symmetric graph Laplacian
	E_{Dir}	Dirichlet energy
	$\widehat{\square}$, $f \star g$	Fourier transform, convolution of f and g
GNN	$\mathcal{V}_i$, $\mathcal{E}_{ij}$, $\mathcal{G}$	Features of node v_i, e_{ij}, and the graph
	$\lvert\mathcal{V}\rvert$, $\lvert\mathcal{E}\rvert$, $\lvert\mathcal{G}\rvert$	Dimensionality of node, edge, and global features
	$\square'$	Feature $\square$ after transformation
	$\bigoplus$	Aggregation function

Graphs can have different sparsities, a measure for the connectedness of a graph. A graph is called sparse when nodes are connected to only a few other nodes, and dense when nodes are connected to many. This can be measured using the graph density

$$D = \frac{|E|}{|V|\,|V|}. \tag{2.1}$$

Since we assume graphs to be directed and include self-loops, the maximum number of edges is $|V| \times |V|$.

Depending on the task, a graph might provide features for different components: Node features $\mathcal{V}_i$ for node v_i, edge features $\mathcal{E}_{ij}$ for the edge e_{ij}, or global features $\mathcal{G}$ per graph. We also use $|\mathcal{V}|$, $|\mathcal{E}|$, and $|\mathcal{G}|$ to refer to their size. Generally, we use $\Box'$ to refer to the transformed features produced by a layer.

2.2 Representing a Graph

Most commonly, a graph is represented in a computer by either an adjacency matrix or an adjacency list. Both rely on defining a fixed but arbitrary node order.

Adjacency Matrix An adjacency matrix $\mathbf{A}$ is a boolean matrix of shape $|V| \times |V|$, with entry $\mathbf{A}_{i,j}$ indicating whether an edge between node i and j exists. Advantages are a constant-time lookup of edge existence and edge manipulation. Its main disadvantage is a quadratic space requirement in the number of nodes, which is particularly problematic for sparse graphs.

When representing edge features, an adjacency matrix can be supplemented by an edge feature matrix of shape $|V| \times |V| \times |\mathcal{E}|$. Alternatively, we can add an edge feature list and store the index of the corresponding edge features in the adjacency matrix.

Adjacency List An adjacency list is a list of length $|E|$, containing an element (i, j) if an edge between the ith and jth node exists. Primary advantage is in storage requirements, which grow linearly with the number of edges. Its main disadvantage is that checking for the existence of an edge requires searching through the whole list.

Edge features can be easily represented by using a separate edge feature matrix of shape $|E| \times |\mathcal{E}|$, whose ith entry corresponds to the ith entry in the adjacency list.

2.3 Spectral Graph Theory

From the graph definition above, one can draw connections between graphs and manifolds. Informally, by treating each node as a point and using the graph neighbourhoods, a graph can be interpreted as a topological space. Treating each neighbour as equidistant, a locally-Euclidian interpretation allows us to interpret the graph as a manifold. Interpreting a graph as a manifold, we are now motivated to find metrics for graph topology. The following draws from Bronstein et al. (2017), to which we refer the reader for more details.

2.3.1 Scalar and Tangent Vector Fields on Graphs

One such metric on manifolds is the Laplacian, the divergence of the gradient. The rough analogy to scalar fields and tangent vector fields are functions defined on nodes ($f : V \to \mathbb{R}$) and edges ($F : E \to \mathbb{R}$). With a weight b for each node and a weight w for each edge, we can use the inner products of

$$\langle f, g \rangle_{H(V)} = \sum_{v \in V} b_v f_v g_v \text{ and} \tag{2.2}$$

$$\langle F, G \rangle_{H(E)} = \sum_{e \in E} w_e F_e G_e \tag{2.3}$$

to define Hilbert spaces on these functions. Intuitively, f associates each node (the equivalent of a point on manifolds) with a value, while F can be interpreted as producing a linear combination of the edge vectors, i.e. a single vector, for each node.

2.3.2 Differential and Divergence

Now, we can define a differential operator on such functions (Lim 2020). $\nabla : L^2(V) \to L^2(E)$ is the graph gradient operating on the Hilbert spaces induced on the node and edge functions respectively, and defined as

$$(\nabla f)_{ij} = f_i - f_j. \tag{2.4}$$

Assuming, as Bronstein et al. (2017) do, that F is alternating, this satisfies

$$(\nabla f)_{ij} = -(\nabla f)_{ji} \tag{2.5}$$

and the gradient is therefore symmetric. This allows us to define the graph divergence $\text{div} : H(E) \to H(V)$ as

$$(\text{div} F)_i = \frac{1}{b_i} \sum_{e_{ij} \in E} w_{ij} F_{ij}. \tag{2.6}$$

Graph Laplacian Now that we have defined both graph divergence and a graph differential operator, we can define the graph Laplacian $\Delta : H(V) \to H(V)$, which like the normal Laplacian is simply $\Delta = -\text{div}\nabla$, as

$$(\Delta f)_i = \frac{1}{b_i} \sum_{(i,j) \in E} w_{ij} (f_i - f_j). \tag{2.7}$$

Stacking edge weights into the $n \times n$-matrix $\mathbf{W} = (w_{ij})$ of edge weights, node weights into the n-dimensional vector $\mathbf{B}$, and degrees into the diagonal degree matrix $\mathbf{D}$ we can rewrite Eq. (2.7) into the more familiar matrix-vector form of the graph Laplacian

$$\mathcal{L}\mathbf{f} = \mathbf{B}^{-1}(\mathbf{D} - \mathbf{W})\mathbf{f}. \tag{2.8}$$

Assuming unweighted nodes and edges, this is equivalent to the simple Laplacian form $\mathcal{L}$, with

$$\mathcal{L} = \mathbf{D} - \mathbf{A} = \begin{cases} \deg(v_i) & i = j \\ -1 & i \neq j; (i,j) \in E \\ 0 & \text{otherwise} \end{cases}. \tag{2.9}$$

Intuitively, the graph Laplacian of a node captures the difference of a function applied to a node and the average of the function applied to its neighbours, just like the normal Laplacian is capturing the difference between a function applied to a point and the local average of the function around that point.

2.3.3 Laplacian Eigenfunctions

The eigenfunctions of a function f on a certain domain $\mathcal{X}$ are the set of orthogonal functions that minimize the Dirichlet energy

$$E_{\text{Dir}}(f) = \int_{\mathcal{X}} \|\nabla f(x)\|^2_{T_x X}\, dx. \tag{2.10}$$

The eigenfunctions are the solution of the optimization problem

$$\min_{\phi_i} E_{\text{Dir}}(\phi_i) \text{ s.t. } \|\phi_i\| = 0 \text{ and } \phi_i \perp \phi_j \ \forall\, 0 \leq j < i. \tag{2.11}$$

Since a graph domain is necessarily a discrete setting, we can simplify Eq. (2.11) to

$$\min_{\mathbf{\Phi}_k \in \mathcal{R}^{n\times k}} \operatorname{Tr}\left(\mathbf{\Phi}_k^\top \mathcal{L} \mathbf{\Phi}_k\right) \text{ s.t. } \mathbf{\Phi}_k^\top \mathbf{\Phi}_k = \mathbf{I}, \tag{2.12}$$

with $\mathbf{\Phi}_k$ being the $n \times k$ matrix of the first k stacked Laplacian eigenvectors. The solution to this equation is given by the first k eigenvectors of $\mathcal{L}$, i.e. satisfying

$$\mathcal{L}\mathbf{\Phi}_k = \mathbf{\Phi}_k \mathbf{\Lambda}_k, \tag{2.13}$$

with $\mathbf{\Lambda}_k$ being the diagonal matrix of the corresponding eigenvalues. Using the definition of $\mathcal{L}$ from Eq. (2.8), we can rewrite this as

$$(\mathbf{D} - \mathbf{W})\,\mathbf{\Phi}_k = \mathbf{B}\mathbf{\Phi}_k \mathbf{\Lambda}_k \tag{2.14}$$

or, introducing $\mathbf{\Psi}_k = \mathbf{B}^{-\frac{1}{2}} \mathbf{\Phi}_k$, as

$$\mathbf{B}^{-\frac{1}{2}} (\mathbf{D} - \mathbf{W})\, \mathbf{B}^{-\frac{1}{2}} \mathbf{\Psi}_k = \mathbf{\Psi}_k \mathbf{\Lambda}_k. \tag{2.15}$$

In the case of using the node degrees as node weights (i.e. $\mathbf{B} = \mathbf{D}$) and identical edge weights (i.e. $\mathbf{W} = \mathbf{A}$), this simplifies to

$$\underbrace{\mathbf{D}^{-\frac{1}{2}} (\mathbf{D} - \mathbf{A})\, \mathbf{D}^{-\frac{1}{2}}}_{=:\mathcal{L}_{\text{sym}}} \mathbf{\Psi}_k = \mathbf{\Psi}_k \mathbf{\Lambda}_k. \tag{2.16}$$

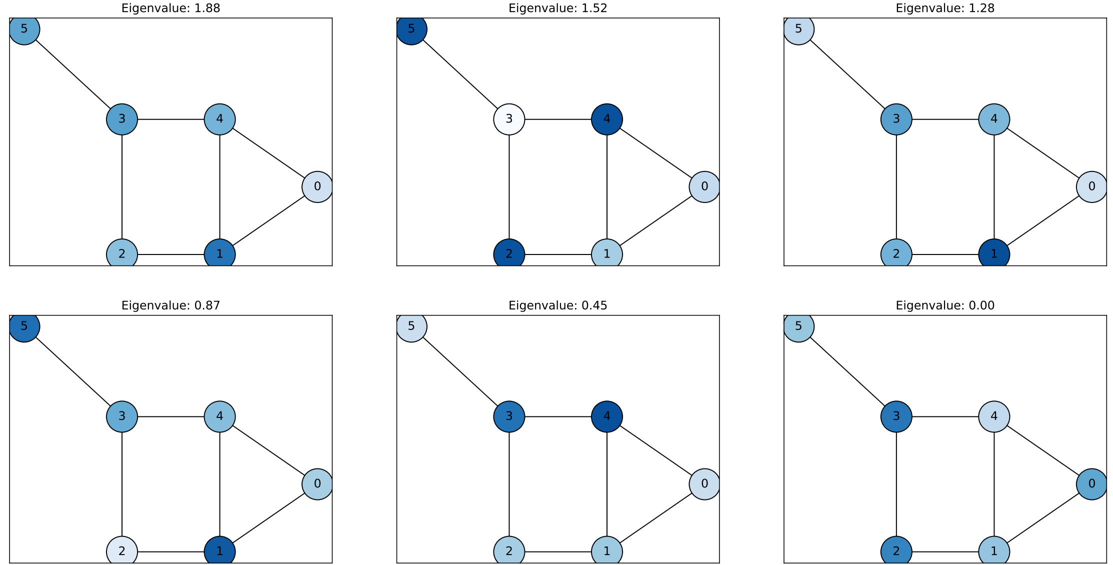


Fig. 2.1: Fourier decompositions of a graph, ordered by the magnitude of the eigenvalue.

$\mathcal{L}_{\text{sym}}$ is called the normalized symmetric Laplacian, and is also often written as $\mathcal{L}_{\text{sym}} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$.

2.3.4 Fourier Analysis of the Graph Laplacian

For any existing graph, Eq. (2.8) allows us to compute the $n \times n$ matrix Δf describing the graph. Often, we are interested in a description capturing both node and topology information. Spectral graph theory allows us to create such a description by applying a Fourier transform to the graph Laplacian. This creates a set of eigenfunctions $\phi_0, \phi_1, \dots$ and ordered eigenvalues $0 = \lambda_0 \leq \lambda_1 \leq \dots$ which together form the *spectrum* of the graph Laplacian.

Given this, we can now decompose any function f on the graph into a Fourier series

$$f(x) = \sum_{i \geq 0} \langle f, \phi_i \rangle_{H(V)} \phi_i(x), \tag{2.17}$$

where $\langle f, \phi_i \rangle_{H(V)}$ forms the components of the Fourier series. This is shown in Fig. 2.1.

2.3.5 Convolutions on Graphs

Transforming graphs into function space allows us to apply the convolution theorem. The convolution theorem states that

$$\widehat{f \star g} = \widehat{f} \times \widehat{g} \text{ and } \widehat{f \times g} = \widehat{f} \star \widehat{g}, \tag{2.18}$$

i.e. the Fourier transform $\widehat{\square}$ of a convolution $f \star g$ of two functions f and g is equivalent to a multiplication of their Fourier transforms (and vice versa). The immediate formulation of the convolution can be written as

$$(\widehat{f \star g})(\omega) = \int_{-\infty}^{\infty} f(x)e^{-i\omega x}dx \int_{-\infty}^{\infty} g(x)e^{-i\omega x}dx. \tag{2.19}$$

We can use this equivalency as a definition, thereby sidestepping the issue of being unable to define convolutions directly on the graph.

Doing so, we arrive at

$$(f \star g)(x) = \sum_{i \geq 0} \langle f, \phi_i \rangle_{H(V)} \langle g, \phi_i \rangle_{H(V)} \phi_i(x). \tag{2.20}$$

For graphs, we use the inner product from Eq. (2.2). Integrating this into Eq. (2.20), we get

$$(f \star g)(x) = \sum_{i \geq 0} \left(\sum_{v \in V} b_v f_v \phi_i(x) \sum_{v \in V} b_v g_v \phi_i(x) \right) \phi_i(x) \tag{2.21}$$

or, in matrix-vector notation,

$$(f \star g) = \mathbf{\Phi} \mathrm{diag}\left(\widehat{\mathbf{g}}\right) \mathbf{\Phi}^{\mathsf{T}} \mathbf{f}, \tag{2.22}$$

where $\mathbf{\Phi}$ are the stacked Laplacian eigenvectors and $\widehat{\mathbf{g}}$ is the spectral representation of the filter.

2.4 Learning Tasks on Graphs

The tasks discussed in this thesis follow a similar structure: Each sample is a single graph (the exception are semi-supervised node-level prediction tasks, see Section 2.4.1). We store its topology information in an adjacency list and, depending on the tasks, might store node features $\mathcal{V}$, edge features $\mathcal{E}$, and global features $\mathcal{G}$. Not all tasks provide all features.

The goal in graph-based machine learning tasks can be sorted into one of three categories, depending on which parts of a graph they are interested in: Node-level tasks produce outputs for each node, edge-level tasks for each edge, and graph-level tasks one set of outputs per graph.

While machine learning models generally distinguish between categories such as classification or regression, supervised, unsupervised, semi-supervised and reinforcement learning tasks, this has little influence in the design of GNN layers, which primarily form the feature extraction backbone out of which we build the final model.

2.4.1 Node-Level Tasks

Node-level prediction produces outputs per node. Trained on a set of labelled nodes, the goal is to predict the output values for each node on unseen graphs.

In image processing, the analogy is pixel-wise semantic segmentation, where each pixel is assigned a class.

Examples for this task are recommender systems[2], prediction of particle motion in physics simulations (Sanchez-Gonzalez et al. 2020), or vehicle motion prediction (see Chapter 5).

Semi-Supervised Node Tasks Semi-supervised node-level prediction tasks are a special case of node-level tasks. They remain important benchmark tasks in the GNN literature, following their use by Kipf and Welling (2016). The three main datasets are citation tasks, where papers are modelled as nodes and their citation links as edges (see Section 3.1.1 for more details).

Generally, the semi-supervised node-level prediction task operates on a single graph[3] for which a small percentage of nodes (usually less than 5%) are labelled. The goal is then to predict the unlabelled nodes on the same graph as used during training.

Outside of citation networks, this type of task occurs wherever there is one single large graph. Main example are large social graphs (Ying, He, et al. 2018), road networks, or product graphs in recommender systems.

2.4.2 Edge-Level Tasks

For some tasks, the goal is not to predict node features but instead edge features. These usually come in two flavours: Either predicting features for existing edges, or predicting whether an edge exists (link prediction).

Edge Feature Prediction Edge feature prediction is used whenever we are interested in the attributes of a relation. This could be classifying a type of relationship in a social graph (friendship, colleagues, or family), attributes of streets in a street network, or power line attributes in a power grid.

Link Prediction In contrast, link prediction is only concerned with whether a certain edge exists. While predicting the complete structure of a graph is usually infeasible[4], predictions for single connections are very useful. Link prediction can be conceptually simple to implement: One can add a virtual edge and directly predict existence (Zhang and Chen 2018), compute features for both adjacent nodes and create a model predicting link existence from them, or compare the output of a graph both with and without the link.

[2] When not represented as a single big graph, in which case this is rather a semi-supervised node-level prediction task.

[3] In principle, it is also possible to have several semi-labelled graphs.

[4] Computational complexity to handle all possible connections obviously grows quadratically in the number of nodes.

2.4.3 Graph-Level Tasks

In graph-level prediction tasks, we are interested in a single output per graph. In the simple case of a fixed graph size, any machine learning model can easily produce this output. Once we treat graphs of different sizes though, a graph-aware model to compute component features followed by a global pooling method to combine these features into a single output becomes more sensible.

The classical example for this task are molecule property prediction, where a molecule is modelled as a graph with atoms as nodes and chemical bonds as edges and one has to predict attributes such as toxicity and solubility (see Section 3.1.2).

2.4.4 Hybrid and Other Cases

The tasks mentioned above are not exclusive: A task might require predicting any combination of them. For example, a model controlling a power grid might be tasked to output both signals to control power generation of generators (modelled as nodes and a node-level tasks) and signals to control the power lines between generators (modelled as edges and an edge-level task).

Similarly, we are not restricted only to supervised or semi-supervised prediction tasks: We can also use graph models for unsupervised tasks or reinforcement learning. In the latter case, we might output global actions and global q-values (for example when modelling a robot's environment as a graph), local node or edge actions and global q-values (for a cooperative multi-agent scenario), or both local actions and local q-values (for a non-cooperative multi-agent scenario).

2.5 Graph Neural Networks Paradigms

In the following section, we give an overview over the development of GNNs. We divide GNNs into three different categories: Recurrent graph neural networks repeatedly apply the same function to the graph representation until it converges. Both spectral and spatial GNNs are closer to the non-graph deep learning paradigm in that they consist of a series of separately-parametrized layers which successively transform graph features. They can be distinguished based on the technique they use for that transform: Spectral GNNs leverage the convolution definition from spectral graph theory (see Section 2.3.5) to apply a mathematically rigorous convolution operation to the graph. In contrast, spatial GNNs compute new features based neighbouring nodes' features, avoiding the inefficiencies inherent in spectral GNNs. We note, though, that there is no hard line distinguishing spectral and spatial methods.

2.5.1 Recurrent Graph Neural Networks

Recurrent Graph Neural Networks (RGNNs)[5] were the initial form of neural networks applied to graphs. They usually apply the same transformation function to node features until a certain convergence criterium is met, then produce an output.

2.5.1.1 Recurrent Graph MLP

First introduced by Gori et al. (2005) and later expanded by Scarselli et al. (2009), they can produce both graph- and node-level outputs.

Their model consists of an iterative two-stage process. Given a set of initial node labels $\mathcal{V}^{(0)}$, a set of edge labels $\mathcal{E}^{(0)}$ and a set of node states[6] $\mathcal{V}$ (in modern parlance, these are the initial node and edge features and the transformed node features respectively), a parametric transition function $\mathcal{T}$ transforming node states is applied to each node

$$\mathcal{V}_i' = \sum_{v_j \in \mathcal{N}(v_i)} \mathcal{T}\left(\mathcal{V}_i^{(0)}, \mathcal{E}_{ij}^{(0)}, \mathcal{V}_j, \mathcal{V}_j^{(0)}\right). \tag{2.23}$$

This transfer function is applied iteratively until convergence, producing final node states $\hat{h}_i$. To guarantee convergence, the transfer function is required to be contractive, and Scarselli et al. (2009) construct a transfer function based on a fully-connected MLP that fulfils this requirement.

After convergence, a readout function takes the converged node state $\hat{\mathcal{V}}_i$ and the initial node features for each node to produce its final output:

$$o_i = \rho\left(\hat{\mathcal{V}}_i, \mathcal{V}_i^{(0)}\right). \tag{2.24}$$

They again use a fully-connected MLP to learn the output function ρ.

Intuitively, this can be seen as an iteratively-applied convolutional layer on graphs, with its receptive field increasing with each application. This is primarily based on the intuition of nodes as entities and edges as their relationship; this way, multiple applications of the transfer function multi-hop relationships.

2.5.1.2 Gated Graph Sequence Neural Networks

Li, Tarlow, et al. (2017) modify the original Recurrent Graph MLP (Scarselli et al. 2009) to use Gated Recurrent Units (GRUs) (Cho et al. 2014) with a fixed number of iterations. They also use a soft attention mechanism in their output model. Their main contribution is in extending this to sequential outputs, demonstrated on both reasoning and a program verification task.

[5] Scarselli et al. (2009) refer to this method as Graph Neural Network and abbreviate it as GNN. To avoid confusion with the much broader term of GNN that is in use today, we follow Wu et al. (2019) in using the term Recurrent Graph Neural Network (RGNN) instead.

[6] Scarselli et al. (2009) do not concern themselves with how to initialize the node state since states converge to the same final state regardless of input.

2.5.1.3 Stochastic Steady-State Embedding

Dai et al. (2018) observed that solving many graph algorithms, such as component detection, PageRank scores, or mean field inference, can be modelled by similar equations as used by the Recurrent Graph MLP model. They iteratively update each node representation until convergence, using

$$h'_i = \mathcal{T}\left(\{h_j\}_{j \in \mathcal{N}(v_i)}\right) \tag{2.25}$$

with a constant initial $h_{v_i}^{(0)}$. As Scarselli et al. (2009), they produce a final output using a readout function

$$o_i = \rho\left(\hat{h}_i\right). \tag{2.26}$$

Dai et al. (2018) learn both the transfer function $\mathcal{T}$ and the readout function ρ. They show promising results on learning several graph algorithms and show their methodology being applicable to node classification.

2.5.1.4 Conclusion

We have seen several models that follow the scheme of repeatedly applying a transformation. Initially, methods applied these until convergence, but the general movement has been towards a fixed number of iterations to avoid huge computational costs. Interestingly, this idea of applying a function until convergence has also been used to expand the depth of CNN models (Chen et al. 2019) by modelling the CNN model as a differential equation.

The field of RGNN models is the smallest and least active of our subdivisions.

2.5.2 Spectral Graph Neural Networks

Spectral Graph Neural Networks (SGNNs) use spectral graph theory (in particular spectral convolutions on graphs, see Section 2.3.5) to define convolutional layers that work on graphs. Applying several of these successively transforms node features to a final representation. Contrary to RGNNs, each layer forms a separate transformation with its own parameters and each is applied once in order. This is far closer conceptually to MLP or CNN models used on non-graph data.

2.5.2.1 Spectral Networks

Bruna et al. (2013) first introduced the notion of a GNN layer based on spectral convolutions. They compute new node features layer-wise using

$$\mathcal{V}' = \mathbf{\Phi}^\mathsf{T}\left(\mathbf{\Phi}\mathcal{V} \odot \mathbf{\Phi}\mathbf{\Theta}\right) \tag{2.27}$$

where $\mathcal{V}$ is the layer's feature representation of all nodes, transformed into the layer's output features $\mathcal{V}'$. $\mathbf{\Phi}$ is the ordered matrix of eigenvectors of the graph Laplacian. In

practice, they often reduce this to only the first k eigenvectors. $\mathbf{\Theta}$ is a diagonal weight matrix (filter) which has one parameter per eigenvector it acts on, and accordingly requires the training of $|\mathcal{V}| \times |\mathcal{V}'| \times d$ parameters. While they argue it should be possible to reduce the number of necessary parameters, they do not provide any constructive method beyond a 1-dimensional ordering of eigenvectors. They demonstrate their method on two MNIST-derived datasets: One subsampled to an irregular grid of 400 coordinates, the other projected onto a 3D sphere.

Henaff et al. (2015) provide an algorithm to achieve the parameter reduction mentioned above, building a smooth interpolation of the weights while learning the weights themselves. More interestingly, they also propose a graph topology estimation procedure based on estimating the distance between two samples and building a Gaussian diffusion Kernel. They demonstrate both methodologies on a news corpus, a computational biology task, and show similar performance to a very simple CNN on ImageNet with a known graph structure.

Unfortunately, both approaches require computation of the eigen-decomposition of the graph, which has a complexity of $\mathcal{O}\left(|V|^3\right)$, clearly infeasible for large graphs. Even with a precomputed eigen-decomposition, $\mathbf{\Phi}$ is a dense matrix of shape $|V| \times |V|$, meaning computational complexity also scales with $\mathcal{O}\left(|V|\right)^3$. Many of the following methods concentrated on finding approximations to reduce this computational complexity.

2.5.2.2 ChebNet

Aiming to correct the computational issue inherent in SGNNs with convolutions expressed in the Fourier domain, Defferrard et al. (2016) introduced a simplified version: They replace the multiplication by the diagonal weight matrix $\mathbf{\Theta}$ from Eq. (2.27) with a polynomial function. For this, they use the Chebyshev expansion. A Chebyshev polynomial $T_k(x)$ is computed using the stable recurrence

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x), \tag{2.28}$$

with $T_0 = 1$ and $T_1 = x$. Replacing the Fourier transformation and convolution in Eq. (2.27) with a Chebyshev polynomial of order K then becomes

$$\mathcal{V}' = \sum_{k=0}^{K-1} \theta_k T_k(\hat{\mathcal{L}})\mathcal{V}, \tag{2.29}$$

with a scaled graph Laplacian $\hat{\mathcal{L}} = \frac{2\mathcal{L}}{\lambda_{\max} - \mathbf{I}}$. Since this is a Kth order polynomial in the Laplacian, output node features only depend on nodes in their K-neighbourhood. By intelligent computation, runtime scales with $\mathcal{O}\left(K\,|E|\right)$, significant savings particularly on sparse graphs. They also show, experimentally, large speedups compared to the original formulation.

Taking things one step further, Kipf and Welling (2016) (which we will revisit in Section 2.5.3 from a spatial convolution perspective) limit K to 1 to receive a function that is linear with respect to the graph Laplacian. They argue that, similar to MLPs and

CNNs, complex filters can still be recovered by stacking multiple such layers combined with a non-linear transformation function. They also assume that $\lambda_{\max}$ is approximately 2, relying on the neural network to adapt to this. Using this, Eq. (2.29) simplifies to

$$\mathcal{V}' = \theta_0 + \theta_1 (\mathcal{L} - \mathbf{I}) \mathcal{V} = \theta_0 - \theta_1 \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} \mathcal{V}. \tag{2.30}$$

Further simplifying by using a single parameter $\theta = \theta_0 = -\theta_1$ results in

$$\mathcal{V}' = \theta \left(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} \right) \mathcal{V}. \tag{2.31}$$

Lastly, to alleviate stability issues, they renormalize the formula to

$$\mathcal{V}' = \theta \left(\bar{\mathbf{D}}^{-\frac{1}{2}} \bar{\mathbf{A}} \bar{\mathbf{D}}^{-\frac{1}{2}} \right) \mathcal{V}, \tag{2.32}$$

with $\bar{\mathbf{A}} = \mathbf{D} + \mathbf{I}$ being the adjacency matrix with added self-connections, and $\bar{\mathbf{D}}$ the degree matrix with added self-connections. Formulated in matrix-vector form, this becomes

$$\mathcal{V}' = \bar{\mathbf{D}}^{-\frac{1}{2}} \bar{\mathbf{A}} \bar{\mathbf{D}}^{-\frac{1}{2}} \mathbf{\Theta} \mathcal{V}, \tag{2.33}$$

with a learned weight matrix $\mathbf{\Theta}$ of size $|\mathcal{V}'| \times |\mathcal{V}|$.

2.5.2.3 CayleyNet

Chebyshev filters, while being far more efficient to compute than the original proposed method by Bruna et al. (2013), suffer whenever narrow-band filters are required. Levie et al. (2017) show that the number of Chebyshev coefficients K must grow inversely proportional to the size of an eigenvalue cluster to be able to distinguish between them. Since ChebNet's computational complexity grows according to $\mathcal{O}(K\,|E|)$, this quickly results in large computational requirements.

This motivates Levie et al. (2017) to introduce Cayley filters, based on the eponymous Cayley transform. They introduce the Cayley filter of order K as

$$\mathcal{V}' = \theta_0 \mathcal{V} + 2\mathrm{Re} \left(\sum_{k=1}^{K} \theta_k \left(h\mathcal{L} - i\mathbf{I} \right)^k \left(h\mathcal{L} + i\mathbf{I} \right)^{-k} \mathcal{V} \right), \tag{2.34}$$

with a learnable parameter vector θ of one real and K complex coefficients and a learnable scalar parameter h, the spectral zoom parameter. h in particular, they argue, allows a better spreading of either high frequencies (for smaller h) or low frequencies (for larger h). Intuitively, Eq. (2.34) projects the real node features onto the complex unit half-circle and extracts only the real component for each.

They demonstrate their method on a citation dataset and a recommender system.

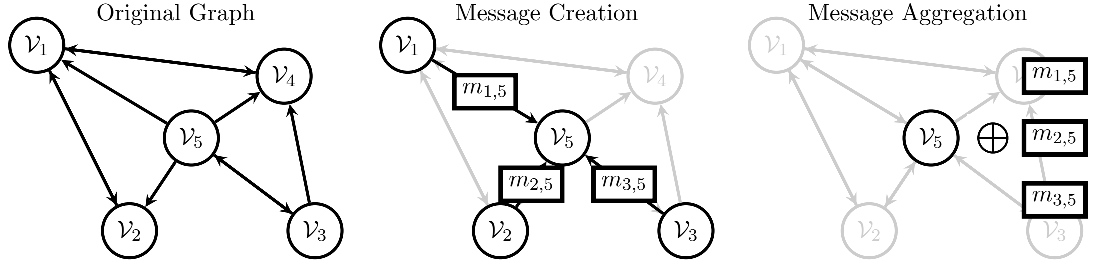


Fig. 2.2: The message passing paradigm, shown for node v_5. For an original graph (left), messages are computed for each edge using the incident edges (middle). These are then aggregated for each node (right), producing new node features.

2.5.2.4 Adaptive Graph Convolutional Networks

Li, Wang, et al. (2018) propose a variation in which the graph Laplacian itself is learned. Specifically, they learn the parameter $W \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$, which they then use to form the covariance matrix S of a generalized Mahalanobis distance measure as $S^{-1} = WW^{\mathsf{T}}$:

$$\mathbb{D}(v_i, v_j) = \sqrt{(\mathcal{V}_i - \mathcal{V}_j)^{\mathsf{T}} S^{-1} (\mathcal{V}_i - \mathcal{V}_j)}. \tag{2.35}$$

Given this distance measure, they use the Gaussian kernel

$$\mathbb{G}(v_i, v_j) = \exp\left(-\frac{\mathbb{D}(v_i, v_j)}{2\sigma^2}\right) \tag{2.36}$$

which, after normalization, produces a dense modified adjacency matrix $\hat{\mathbf{A}}$ or, if an adjacency matrix is already defined by the data, a residual graph Laplacian matrix $\mathcal{L}_{\text{res}}$ using $\hat{\mathcal{L}} = \mathcal{L} + \alpha \mathcal{L}_{\text{res}}$ with a learned parameter α.

While the model shows improved performance compared to models using a given graph formulation, they pay for this with large computational cost. In particular, computing the kernels $\mathbb{G}$ requires $\mathcal{O}\left(|V|^2\right)$ computations and results in a dense adjacency matrix, invalidating sparse computation tricks to keep computation requirements low.

2.5.2.5 Conclusion

SGNNs models are mathematically elegant, using spectral graph theory to rigorously define convolution operations on graphs. Unfortunately, they usually suffer from computational issues, with complexity growing with $\mathcal{O}\left(|V|^2\right)$ or even $\mathcal{O}\left(|V|^3\right)$ and necessitating dense graph Laplacian matrices. Several approaches aim to reduce the required computation, but there has not yet been a single convincing method to do so.

2.5.3 Spatial Graph Neural Networks

Contrary to SGNNs, spatial GNN models operate on the notion of neighbourhood and do not take the whole graph into account. Its main implementations are Message Passing

Neural Network (MPNN) models, which use only the immediate neighbourhood of a node to compute that node's new features. Generally, a message function M creates messages m_{ij} for node v_i based on its neighbouring nodes v_j and the edge e_{ij}

$$m_{ij} = M\left(v_i, v_j, e_{ij}\right). \tag{2.37}$$

Each node's new features are then computed by a node update function $\mathcal{U}_v$

$$\mathcal{V}_i' = \mathcal{U}_v\left(\mathcal{V}_i, \bigoplus_{v_j \in \mathcal{N}(v_i)} m_{ij}\right) \tag{2.38}$$

using the messages m and the previous layer's node features. Messages are collated using a collation function $\bigoplus$, which has to be transitive and associative in order to avoid issues with node order. Usually, the sum is used, with a maximum and mean operation as alternatives. See Fig. 2.2 for a graphical overview.

From the description above, it is clear that computational complexity is better than using SGNNs, scaling in $\mathcal{O}\left(|E|\right)$ without requiring polynomial approximations. However, global graph structure cannot be modelled well using this system (see Section 2.5.3.6).

2.5.3.1 Graph Convolutional Networks (GCNs)

GCNs have already been introduced in Section 2.5.2.2 as an extreme form of approximating the spectral convolution using only a first-order Chebyshev polynomial approximation. However, while Kipf and Welling (2016) do not formulate their method as a message passing approach, it nonetheless can easily be expressed as that. Setting

$$M\left(v_i, v_j, e_{ij}\right) = \left(\deg\left(v_i\right)\deg\left(v_j\right)\right)^{-1/2} \mathbf{A}_{ij}\mathcal{V}_j, \tag{2.39}$$

i.e. depending only on the degrees of the corresponding nodes and the hidden features of the neighbours[7], setting $\bigoplus = \sum$, and

$$\mathcal{U}_v\left(v_i\right) = \mathbf{\Theta} \sum_{v_j \in \mathcal{N}^+(v_i)} m_{ij}, \tag{2.40}$$

we can reformulate the original GCN as a MPNN[8]. This also shows that a GCN layer is based on what is essentially a weighted sum of neighbouring node features, which are then transformed by a learned weight matrix.

This formulation makes the sparsity of the update procedure immediately apparent: Each node's update depends only on the immediate neighbours. Using efficient implementations[9], computation time scales in $\mathcal{O}\left(|E|\right)$.

[7] The GCN model adds self-connections to the graph, which allows an elegant formulation by adding the node v itself to its neighbourhood in Eq. (2.39). Alternatively, one can modify Eq. (2.40) to integrate the node features before applying the weight.

[8] Kipf and Welling formulate their layer definition as including the activation function σ, which non-linearly transforms the final output. We use the definition without an activation function for consistency.

[9] Implementing these is non-trivial: GPUs are dominate deep learning due to their capability of fast parallelized matrix multiplications, but the usecase for which the low-level libraries are implemented do

2.5.3.2 Message Passing Neural Networks (MPNNs)

Aside from introducing the message passing formulation used above, Gilmer et al. (2017) introduced the MPNN model, which also takes edges into account. Compared to the GCN model, the message function is modified to

$$M\left(v_i, v_j, e_{ij}\right) = \mathrm{MLP}\left(\mathcal{V}_i \| \mathcal{V}_j \| \mathcal{E}_{ij}\right) \tag{2.41}$$

to take edge information into account by applying an MLP to them. The node update function takes the original node features separately into account[10]:

$$\mathcal{U}_v\left(v_i\right) = \boldsymbol{\Theta}\mathcal{V}_i + \sum_{v_j \in \mathcal{N}^+(v_i)i} m_{ij}, \tag{2.42}$$

The MPNN model is conceptually much more powerful than the GCN model, since it allows for both node-dependent updates and includes edge attributes. It is, however, restricted to transforming node features[11]. Edge features are used as an input for each layer, but remain themselves untransformed. The authors do propose the use of global features by introducing a virtual *master node*, connected to every other node. This could also be used to produce graph-level outputs.

They demonstrate MPNNs on the task of predicting chemical properties for molecules.

2.5.3.3 GraphSAGE

GraphSAGE (Hamilton et al. 2017) is based on a localized sampling of the neighbourhood. Each node is updated based on the k-neighbourhood[12]:

$$\mathcal{V}_i' = \boldsymbol{\Theta}\left(\mathcal{V}_i \| \bigoplus_{v_j \in \mathcal{N}_k(node_i)} \left(\mathcal{V}_j\right)\right). \tag{2.43}$$

In practice, their experiments show a two-hop neighbourhood to slightly increase prediction performance compared to the immediate neighbourhood. Larger neighbourhoods greatly increased computation time for marginal prediction increases.

Hamilton et al. (2017) explore three different aggregation functions in their work: One takes the elementwise mean of the vectors, one applies learned Long Short-Term Memory (LSTM) units on a random permutation of the neighbours, and one applies max pooling on independently transformed node features. On their example tasks of citation and protein classification, they found the latter two to generally perform best, but found the LSTM-based approach to require roughly double the computation time.

---

not include the large number of scattering operations that GNNs require. However, custom kernels (such as *pytorch-geometric* (Fey and Lenssen 2019)) can alleviate this concern and are necessary for large-scale experiments.

[10] In particular, this means it does not use the same self-connection trick as the GCN model.

[11] Gilmer et al. are primarily concerned with predicting values or classifying for the complete graph, and do not formulate their model for a node-level prediction task. It is trivial to use the model for this, though.

[12] We again do not include the activation function in our formulation.

2.5.3.4 Graph Attention Network (GAT)

Whereas GCNs treat all neighbours identically and produce a final output based on an equal mix of their features, Graph Attention Networks (GATs) (Veličković et al. 2017) produce attention weights for each neighbouring node based on their features. That is, the original node update function from Eq. (2.40) is augmented with an edge-specific attention weight α_{ij} for edge e_{ij}

$$\mathcal{U}_v(v_i) = \mathbf{\Theta} \sum_{v_j \in \mathcal{N}^+(v_i)} \alpha_{ij} m_{ij}. \tag{2.44}$$

α is computed using the agreement between transformed source and target node features, normalized using the softmax operator:

$$\alpha_{ij} = \mathrm{softmax}_{\mathcal{N}^+(v_i)} \left(\sigma \left(\mathbf{\Theta} \mathcal{V}_i \| \mathbf{\Theta} \mathcal{V}_j \right) \right). \tag{2.45}$$

Usually, they use multi-head attention. They demonstrate their method on three citation datasets and a protein dataset.

2.5.3.5 Graph Networks

Expanding on the notion of message passing, Battaglia et al. (2018) introduced a generalization they call a graph network. It is a very general framework, defining three update rules for edge features, node features, and global features. These are updated sequentially.

Edges are updated first, and transformed using their current features, the node features of source and target node, and the global features:

$$\mathcal{E}'_{ij} = \mathcal{U}_e \left(\mathcal{E}_{ij}, \mathcal{V}_i, \mathcal{V}_j, \mathcal{G} \right), \tag{2.46}$$

where $\mathcal{U}_e$ is an arbitrary learnable edge update function.

Afterwards, the nodes are updated according to

$$\mathcal{V}'_i = \mathcal{U}_v \left(\bigoplus_{\forall e_{ij} \in E}^{E \to V} \left(\mathcal{E}_{ij} \right), \mathcal{V}_i, \mathcal{G} \right), \tag{2.47}$$

where $\bigoplus^{E \to V}$ is an aggregation function aggregating edges and $\mathcal{U}_v$ is, again, an arbitrary learnable node update function.

Lastly, the global state is updated using

$$\mathcal{G}' = \mathcal{U}_{\mathcal{G}} \left(\bigoplus_{\forall e_{ij} \in E}^{E \to G} \left(\mathcal{E}_{ij} \right), \bigoplus_{\forall v_i \in V}^{V \to G} \left(\mathcal{V}_i \right), \mathcal{G} \right), \tag{2.48}$$

with two aggregate functions $\bigoplus^{E \to G}$ and $\bigoplus^{V \to G}$ aggregating all of the graph's edges and nodes respectively. These are then used by the global update function $\mathcal{U}_{\mathcal{G}}$, again an arbitrary learnable function, to produce the new global state.

As has become apparent from the preceding equations, the Graph Network paradigm is extremely powerful and flexible. However, it is also so broadly defined that it offers no real guidance on the choice and tradeoffs of either the aggregation or the update functions, though Battaglia et al. (2018) use MLPs for their demonstrations on finding the shortest path, simulating mass-spring systems, and sorting numbers.

2.5.3.6 WL-Graph-Isomorphy and the GIN Model

The Weisfeiler-Leman graph isomorphy test (Weisfeiler and Leman 1968) is used to test two graphs for isomorphy. It is a powerful heuristic, despite not being able to distinguish all non-isomorphic graphs. Morris et al. (2018) showed how MPNN models are equivalent to the 1-WL algorithm, using this to construct more powerful k-GNN models that operate on sets of nodes. Xu, Hu, et al. (2018) also prove that equivalency and introduce a GNN layer that is as powerful as possible within the 1-WL framework.

The WL Algorithm The WL algorithm depends on iteratively computing node colourings. We assign each node v_i an initial unique node colouring $c_i^{(0)}$

$$c_i^{(0)} = i. \tag{2.49}$$

Afterwards, we iteratively compute new colourings

$$c_i^{(t+1)} = \text{HASH}\left(c_i^{(t)}, \{c_j^{(t)} | v_j \in \mathcal{N}(v_i)\}\right) \tag{2.50}$$

using some bijective hash function HASH that maps each input pair onto a unique value. That is, we compute the new colour for each node from a combination of its old label and its neighbours' labels. We repeat this until convergence, that is until the number of colours does not change between iterations. This happens after at most $|V|$ iterations.

When testing for graph isomorphy, we run this algorithm on both graphs in parallel. When the two graphs are mapped to a different final colouring, they are not isomorphic.

The WL algorithm can be expanded to the k-WL algorithm by using the k-hop neighbourhood $\mathcal{N}_k(v_i)$, though this increases computational complexity significantly.

Equivalence to GNN models Morris et al. (2018) show how MPNN models are equivalent to the 1-WL algorithm. We only sketch the proofs here, and refer readers to the appendix of Morris et al. (2018) for the full proof.

They first prove how MPNN models are no more powerful than the 1-WL, i.e. that the new node features $\mathcal{V}'_i$ and $\mathcal{V}'_j$ of nodes v_i and v_j produced by a MPNN layer must be equal if the new colourings c'_i and c'_j are equal. If the colourings are equal, both nodes' previous colourings and the colourings of their neighbourhoods must have been identical. Since the node features are computed from exactly these same nodes, they must also be equal.

Afterwards, they prove that, for certain choices of weights, the 1-WL is no more powerful than MPNN models. They construct a weight matrix for a MPNN model such that it achieves the same mapping as the 1-WL algorithm.

Accordingly, GNN models operating on the neighbourhood of graphs by using a simple aggregation function is as powerful as the 1-WL algorithm.

k-GNN Models Discontent with the shortcomings of neighbourhood-operating GNN models (which they call 1-GNN models), Morris et al. (2018) instead propose the generalization of k-GNN models, based on the k-WL algorithm. They consider the k-element subsets of V^k instead of the nodes V, i.e. they operate on neighbouring node sets.

They show advantages compared to standard GNN models on both small benchmark datasets and the larger *QM9* dataset.

Graph Isomorphism Network (GIN) Models Concurrently to Morris et al. (2018), Xu, Hu, et al. (2018) also proved the equivalency of GNN models and the 1-WL algorithm. They formulated two conditions (both update and aggregation function have to be injective and the graph-level readout function has to be injective), and decide to use MLPs to model these functions for maximum flexibility. That is, they introduce a parameter ϵ (either learned or fixed) and an MLP model into the node aggregation function Eq. (2.38) such that

$$\mathcal{V}_i' = \mathrm{MLP}\left((1+\epsilon)\mathcal{V}_i + \sum_{v_j \in \mathcal{N}(v_i)} m_{ij}\right). \tag{2.51}$$

They also find failure cases for commonly-used global pooling strategies and choose to use the summation of the node features from all layers.

2.5.3.7 Conclusion

In contrast to SGNNs models, spatial GNN models are far better scaleable. This has made them the de facto standard GNN type in use today. Of particular interest are the theoretical insights gained into GNN models and the resulting models.

2.5.4 Computational Implementation

The implementation of GNN layers is difficult: Whereas standard neural networks rely only on matrix multiplication, addition, and convolution and are therefore fairly simple to implement and optimize, GNN models operate on sparse data structures and often require the scattering of values from a vector onto another matrix based on an index. Standard GPU implementations like CUDA do not support these functions well and specialized implementations such as *pytorch-geometric* (Fey and Lenssen 2019) are needed.

2.6 Conclusion

We have introduced the necessary background for this thesis. We have started by introducing the notation and definitions (Section 2.1) and how to represent a graph in a computer (Section 2.2). In Section 2.3, we then introduced spectral graph theory upon which graph convolutions are often based. Section 2.4 then showed a number of different problem types that might occur on graphs. In Section 2.5, we then introduced recurrent graph networks, and spectral and spatial GNN models as well as theoretical analysis of the latter.

3 Simple Graphs

In its simplest form, a graph consists only of nodes (each with some features) and featureless edges between them, modelling relations (see Fig. 3.1). This class of problems has long been used for benchmarks. This remains an issue because these datasets only represent a small subset of possible tasks: They lack edge features and their nodes are usually homogeneous, always representing the same type of object.

These datasets are nonetheless extremely diverse, containing data such as academic citation graphs, proteins, online forum interactions, or co-purchased products. These task classes have driven the development of models since the beginning of research into modern GNN models: Kipf and Welling (2016) evaluated their GCN model on three citation network tasks and one knowledge graph dataset preprocessed to follow the same format. Since then, the majority of GNN development has been evaluated—most of them exclusively—on either these citation datasets or a subset of the datasets curated by Kersting et al. (2020).

In this chapter, we use these datasets to evaluate how to build GNN models, evaluating among others encoder types, layer types, depth, and global pooling methods. Section 3.1 introduces the datasets we use to evaluate the models. Section 3.2 showcases the base GNN model and the relevant parameters one can optimize. Based on these building blocks, Section 3.3 introduces the research questions used to guide our investigation and their results. Lastly, we summarize our results in Section 3.4.

3.1 The Datasets

GNN models have been evaluated on a large number of benchmark datasets. The most widely used graph datasets are a set of three citation graphs (*citeseer*, *cora*, and *pubmed*), which are used for semi-supervised node classification, and graphs from several different

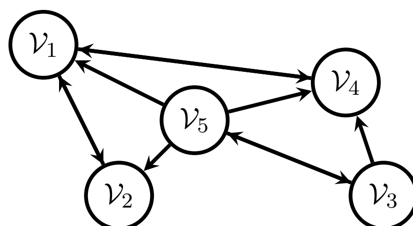


Fig. 3.1: A simple graph: A total of five nodes with different features. There is a graph topology, but edges do not have separate features.

Table 3.1: Attributes of the three citation datasets. As can be seen, these differ in both size and the number of targets. However, they are all similarly sparse.

Dataset	Nodes	Edges	Avg. Degree	Features	Classes	Labeled Nodes	Density
citeseer	3 327	4 552	1.37	3 703	6	120 (3.64 %)	0.008 21
cora	2 708	5 278	1.95	1 433	7	140 (5.17 %)	0.001 44
pubmed	19 717	44 324	2.25	500	3	60 (0.30 %)	0.000 23

domains, collected by Kersting et al. (2020). The latter are usually supervised graph classification tasks.

3.1.1 Semi-Supervised Node Classification Datasets

All three citation datasets were introduced as a benchmark for graph tasks by Sen et al. (2008)[1]. The datasets model scientific papers as nodes, including bag-of-word features as node features, and their citations as edges. Details on the datasets can be found in Table 3.1.

The generally agreed-upon task is a semi-supervised node classification: Given 20 labelled nodes per class and the graph structure, we aim to classify the unlabeled nodes into one of 3 to 7 subfields. Since semi-supervised node classification acts on only one single graph, graph topology is identical between training and evaluation. Only the labelled nodes differ.

3.1.2 Supervised Graph Classification Datasets

Most datasets curated by Kersting et al. (2020) are supervised graph classification datasets. Each sample is a graph, either labelled or unlabelled, and the model is trained to predict a single label per graph. We concentrate on five of the collected 130 tasks: *proteins*, *nci1*, *imdb*, *rdt-b*, and *rdt-12k*, which represent a large cross-section of dataset and individual graph size. Their statistics are shown in Table 3.2.

3.1.2.1 The proteins Dataset

In the *proteins* dataset (Borgwardt et al. 2005; Dobson and Doig 2003), each graph represents one protein. Nodes are secondary structure elements like helices, sheets, and turns. Nodes are connected if they are neighbours in the amino acid sequence, and to their three spatially-closest neighbours. Models are trained to predict whether a protein is an enzyme.

[1] *citeseer* (Giles et al. 1998) and *cora* (McCallum et al. 2000) are adaptations of previously published datasets. *pubmed* was released later by Sen et al. (2008) and is not mentioned in the paper; nevertheless, they ask for that paper to be cited.

Table 3.2: Attributes of selected graph classification datasets. As can be seen, these vary in both size (1 000–12 000 graphs), number of targets (2–11 classes), graph size (20–430 nodes per graph) and density (0.005 3–0.518 5).

Dataset	Graphs	Avg. Nodes	Avg. Edges	Avg. Degree	Features	Classes	Density
proteins	1 113	39.1	72.8	1.9	3	2	0.097 7
nci1	4 110	29.9	32.3	1.1	37	2	0.074 8
imdb	1 000	19.8	96.5	4.9	136	2	0.518 5
rdt-b	2 000	429.6	497.8	1.2	1	2	0.005 3
rdt-12k	11 929	391.4	456.9	1.2	1	11	0.005 9

3.1.2.2 The nci1 Dataset

Wale and Karypis (2006) introduced the *nci1* dataset based on data from the PubChem database (Kim et al. 2021)[2]. It describes the inhibition on a number of chemical compounds by the NCI-H23 human Non-Small Cell Lung tumour cell line, by whose name it is also sometimes referred to. Vertices represent the chemical compounds' atoms and edges represent their bonds.

3.1.2.3 The imdb Dataset

In the *imdb* dataset (Yanardag and Vishwanathan 2015), each node is an actress or actor. These are connected if they have appeared together in a movie. The goal is to classify movies into the Romance or the Action genre. Movies appearing in both genres are removed.

3.1.2.4 The rdt-b and rdt-12k Datasets

Both *reddit* datasets (Yanardag and Vishwanathan 2015) are constructed from discussions on the reddit online platform, with each graph modelling one discussion. Nodes represent users, which are connected if they interact with each other in that discussion. The goal for *rdt-b* is to distinguish between question/answer-based subreddits and discussion-based subreddits. For *rdt-12k*, the goal is to learn which of 11 different subreddits a discussion belongs to.

3.2 GNN Architecture

In their architecture, GNN models share similarities to the standardization of CNN architectures, and most GNN models share the same building blocks. We also introduce tweaks commonly used to improve performance, followed by our baseline model configuration.

[2]We follow the wishes of the authors and cite the newest update paper of the PubChem database.

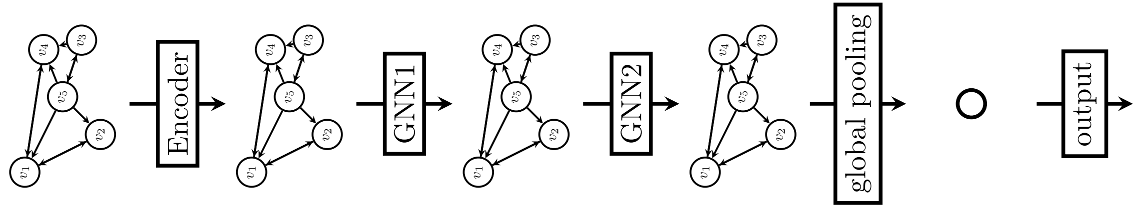


Fig. 3.2: Schematic of the generally-used GNN architecture, showcasing (a) the initial feature encoder, (b) the GNN layers, (c) the global pooling layer, and (d) the output layer. This example is the architecture for graph-level tasks; a node-level task will not use the global pooling layer.

3.2.1 Building Blocks

The most commonly used scheme is depicted in Fig. 3.2, and consists of the following building blocks:

1. The **encoding layer** preprocesses each node's features independently.
2. A number of **GNN layers** then take graph structure and node features into account to successively transform node features.
3. For graph-level prediction, a single output vector is created by **globally pooling** the node features.
4. An **output layer** produces the final output, either per node or per graph.

Encoding Layer The initial encoding layer mainly serves to encode the node features into a more expressive subspace. On the citation datasets, for example, this can compress bag-of-word features of several hundred dimensions into significantly smaller representations. Alternatively, it might process one-hot-encoded features into denser feature representations. We explore these in Section 3.3.4.

GNN Layers Most research has concentrated on proposing new GNN layers (see Section 2.5 for an overview). These share the same principles, however: Given graph topology and node features, they transform node features into a new feature space. In general, none of these layers transform the graph structure itself. We explore these in Section 3.3.3.

Global Pooling For graph prediction tasks, it is crucial to produce a single vector output for the final classification. This pooling must be associative and commutative to be order-invariant. Usually, this is either produced by adding or averaging all node features (global sum pooling, global average pooling), or by taking the component-wise maximum of the node features (global max pooling). We explore these in Section 3.3.5.

Output Layer The final layer produces either logits (for a classification task) or the predicted value (for a regression task). Usually, these are simple linear layers; however, it is also possible to introduce more extensive transformations on the summarized graph features. We explore these in Section 3.3.6.

3.2.2 Tweaks

A number both of proposed GNN methods and of GNN implementations include several tweaks. We explore these in Section 3.3.8.

Dropout Introduced by Srivastava et al. (2014), dropout aims to avoid overfitting by randomly masking features for each layer. This avoids over-adaptation of single features in the neural network. Classically, this zeroes out fifty percent of the features in each hidden layer, and twenty percent of the input features.

In GNNs, dropout can also be applied node- and edge-wise in addition to feature-wise.

Batch Normalization Introduced by Ioffe and Szegedy (2015), batch normalization aims to enforce layer outputs that follow a standard normal distribution. Adapting the original equations for graph processing, we compute mini-batch mean and variance during training as

$$\mu_{\mathcal{V}} = \frac{1}{|V|}\sum_{i=1}^{|V|}\mathcal{V}_i, \qquad \sigma_{\mathcal{V}}^2 = \frac{1}{|V|}\sum_{i=1}^{|V|}\left(\mathcal{V}_i - \mu_{\mathcal{V}}\right)^2, \tag{3.1}$$

i.e. we treat each node as a separate sample and compute mean and variance over all nodes in our mini-batch. We then apply the normalization as

$$\tilde{\mathcal{V}}_i = \frac{\mathcal{V}_i - \mu_{\mathcal{V}}}{\sqrt{(\sigma_{\mathcal{V}}^2) + \epsilon}} \tag{3.2}$$

with a small ϵ to ensure numerical stability. After this, we apply a scale and shift operation

$$\mathcal{V}_i' = \gamma\tilde{\mathcal{V}}_i + \beta \tag{3.3}$$

with learned feature vectors γ and β. During training, this is achieved by using batch statistics to normalize features for each layer; during evaluation, population statistics collected during training are used instead.

We found that using batch normalization as usually implemented is not compatible with semi-supervised node classification on one graph. Accordingly, when used for semi-supervised node classification, we do not use running statistics.

Table 3.3: Accuracy and 95%-confidence bound (in percent) of the baseline model on the eight simple graph datasets.

	citeseer	***cora***	***pubmed***	***proteins***	***nci1***	***imdb***	***rdt-b***	***rdt-12k***
Baseline	54.7±1.2	70.5±1.0	68.3±1.7	71.7±1.9	75.8±2.7	73.9±3.1	91.6±0.9	48.9±0.5

Jumping Knowledge Introduced by Xu, Li, et al. (2018), the jumping knowledge (JK) formulation concatenates all previous GNN layer outputs to produce the final feature representation. This allows the final classification layer to adaptively choose the relevant neighbourhood size of a node by ignoring larger or smaller neighbourhood sizes.

Separate Node Processing Most GNN models (see Section 2.5.3.1, for example the GCN formula) aggregate features from neighbours and ego node equally. Introducing a separate processing step for the ego node increases the computational possibilities. For a GCN model, this changes Eq. (2.40) to

$$\mathcal{U}_v\left(v_i\right) = \mathbf{\Theta}\left(\mathcal{V}_i \| \sum_{v_j \in \mathcal{N}(v_i)} m_{ij}\right). \tag{3.4}$$

3.2.3 Baseline Model

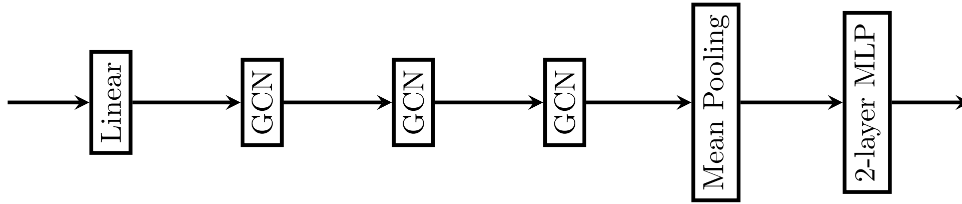
In the following experiments, we use a single configuration as a baseline, whose details we vary depending on the research question. This is depicted in Fig. 3.3, both for graph-level and node-level prediction tasks. The baseline uses three GCN layers of 128 channels each, preceded by a linear feature transformation. To produce a final prediction, the outputs of the GCN layers are concatenated (JK), and the final output produced by a two-layer MLP. For graph-level prediction tasks, we use global mean pooling. We use BatchNorm after each GNN layer[3].

We train the models with a batch size of 128 for a maximum of 500 epochs, using the Adam optimizer (Kingma and Ba 2014) and an initial learning rate of 10^{-3}. On reaching a plateau (no improvement of the validation loss for 25 epochs), we halve the learning rate. If no improvement occurs after two consecutive halvings of the learning rate (50 epochs), we stop the training early. We always load the model with the best validation loss.

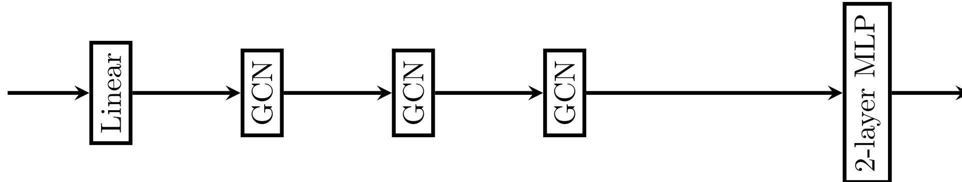
All models are evaluated using ten-fold cross-validation. On the five graph classification datasets, this is simply ten-fold stratified cross-validation. On the citation datasets, we pick a different random combination of twenty labeled nodes per class for each run.

Baseline performance can be found in Table 3.3. In this and in following tables, we show results as mean and the boundaries of the 95% confidence bound. We mark the best method and comparable methods within the confidence bounds.

[3]Note that we modify BatchNorm (see Section 3.2.2) for its use in the semi-supervised node prediction tasks.



(a) Baseline for graph-level tasks.



(b) Baseline for node-level tasks.

Fig. 3.3: Schematic of the baseline architecture, showing both the configuration used for graph-level and for node-level prediction tasks.

3.3 Experiments and Discussion

In the following section, we present our experimental setup and discuss our results. We are primarily interested in two different basic questions: What influence does graph information have on neural network models (**Q1** and **Q2**), and how to build GNN models (**Q3**–**Q9**). We therefore answer the following research questions:

Q1: How useful is the inclusion of graph information?

Q2: How reproducible are GNN models?

Q3: How do different GNN layers perform?

Q4: How do we encode node features?

Q5: How do we global-pool graphs?

Q6: How do we construct the final graph output?

Q7: How deep should GNN models be?

Q8: Which tweaks improve GNN performance?

Q9: Is it helpful to separate processing and propagation?

Table 3.4: Accuracy and 95%-confidence bound (in percent), comparing graph information. Our baseline (*) uses both the original node features and the edges.

	citeseer	*cora*	*pubmed*	*proteins*	*nci1*	*imdb*	*rdt-b*	*rdt-12k*	Rel. Perf.	Mean Rank
Baseline (*)	56.0±1.6	70.0±1.2	68.3±1.2	73.3±2.2	76.4±1.5	73.2±2.3	92.4±1.0	48.8±0.5	1.00	1.125
No Features	19.8±0.6	17.8±1.7	37.9±1.4	59.9±3.1	58.6±4.3	53.3±3.8	86.9±4.8	49.6±1.5	0.68	4.125
Topology Features Only	19.0±1.1	13.1±2.0	34.8±1.1	66.0±2.7	60.5±1.4	70.3±2.3	87.1±1.1	45.3±0.7	0.69	4.25
Topology and Node Features	23.0±2.4	18.7±2.8	40.4±2.1	72.1±1.9	72.2±1.9	71.8±3.0	87.3±1.1	44.8±0.7	0.75	2.75
No Topology	34.0±1.1	32.8±1.1	51.7±0.9	70.4±1.9	64.1±2.2	70.9±2.7	87.2±2.2	45.0±0.7	0.81	2.75

3.3.1 Q1: How Useful is the Inclusion of Graph Information?

The advantage of GNN models is that they can take graph topology into account, thereby gaining access to additional information. However, the added complexity—in particular the increase in computational runtime due to inefficient data formats (see Section 2.5.4)—might not make this a good idea even in tasks where graph topology helps.

Experiment

To evaluate the usefulness of graph information, we conduct a series of experiments which remove certain sources of information.

Baseline Our baseline uses both the original node features and graph topology, i.e. edge features. It uses standard GCN layers.

No Features This experiments removes all node features, and therefore relies only on graph topology. It uses standard GCN layers.

Topology Features Only This experiment removes both edges and node features. Instead, each nodes' features are replaced with its degree. We expect this to perform very badly. It uses an MLP.

Topology Features and Node Features This model removes edges. Instead, the degree is added as additional node feature. It uses an MLP.

No Topology This approach ignores edges, and does not add any additional node features. This is equivalent to a non-GNN approach, and uses an MLP to process nodes.

Results and Conclusion

Table 3.4 shows the results of these experiments. We can make several observations:

- On most datasets, node features are important. Ignoring node features results in a performance little better than random on the citation datasets. Only on *rdt-12k* does the model without node features perform comparably. This is to be expected, since *rdt-12k* only contains an artificial normalized degree feature.

- Artificial topology features are no replacement for the actual edges. The only exceptions are the *proteins* and the *imdb* datasets, where they perform comparable to GNN models.
- The actual topology information and the communication between nodes from including these, increases performance in all cases.

These observations answer **Q1**: Including graph data is necessary for a good performance on most datasets, and they cannot be replaced with just using artificial topology features.

3.3.2 Q2: How Reproducible Are GNN Models?

In many areas of deep learning, results are hugely variable and may depend more on a lucky initial initialization of the the neural network parameters than on the hyperparameters or methods (see Lucic et al. (2018) and Pineau (2018) for examples). For GNNs, Shchur et al. (2018) made similar observations: Depending on the dataset, models with the same hyperparameters might vary by ten or more percentage points (pp), far more than the performance gain model optimization could produce.

Experiment

We examine this phenomenon by repeatedly training our baseline model with different initializations and compare its results on different datasets. We train each model a total of 100 times.

Results and Conclusion

Fig. 3.4 shows the results. We can make several observations:

- Variability of results depends strongly on the dataset. Even the smallest extent lies at about 10 pp. For the more varied datasets—*proteins* and *imdb*—it reaches 20 pp.
- All show a similar pattern: Fairly close performances, with outliers trailing the other performances. This is particularly pronounced for the more varied datasets—*proteins*, *imdb*, *nci1*—in which the outliers trail mean performance by 10 or more pp.
- This pattern holds over problem type (node-level and graph-level predictions), and dataset size.

This answers **Q2**: GNN models on simple graphs are not particularly reproducible, and show large variability of performance. This suggests we might not initialize or train GNNs efficiently yet. Training a GNN multiple times is a stopgap measure.

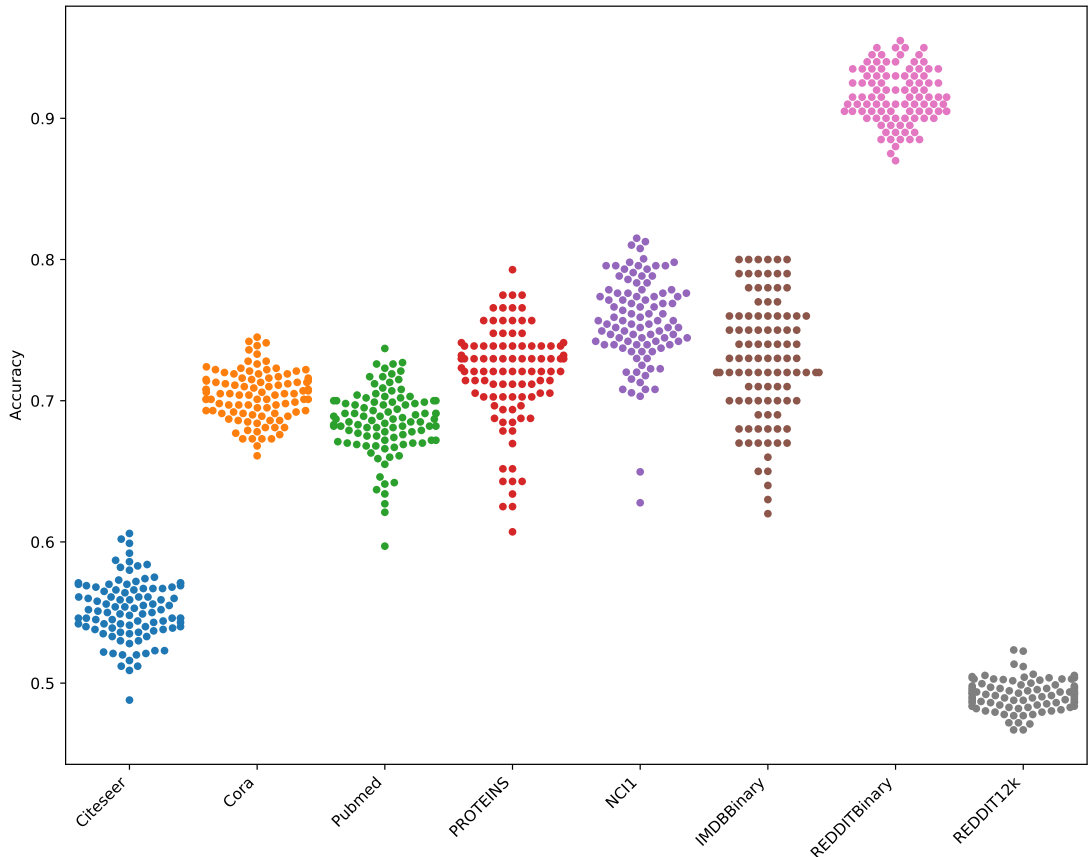


Fig. 3.4: Influence of different random initializations on the final result. We train the same model 100 times and plot the final test accuracy. Each dot represents the final result of one run.

Table 3.5: Accuracy and 95%-confidence bound (in percent), comparing different proposed GNN layers to the baseline (*).

	citeseer	*cora*	*pubmed*	*proteins*	*nci1*	*imdb*	*rdt-b*	*rdt-12k*	**Rel. Perf.**	**Mean Rank**
GCNConv (*)	55.3±1.4	70.7±0.8	67.0±1.6	72.0±2.7	74.5±3.3	71.9±2.8	92.0±1.3	49.1±0.8	0.99	1.75
GATConv	54.3±1.2	69.7±1.0	65.2±1.4	71.8±2.3	76.2±2.2	71.0±2.9	87.8±2.5	39.1±5.0	0.95	3.375
GINConv	45.6±1.2	58.2±1.7	63.4±2.0	70.1±2.3	79.0±1.5	74.6±2.9	90.8±1.1	47.1±1.1	0.94	2.75
SAGEConv	56.3±1.3	71.1±0.8	66.9±1.0	71.8±2.7	75.8±2.6	69.7±4.2	90.4±1.1	44.2±0.9	0.97	2.625
MLPConv	32.9±1.6	32.8±2.2	53.8±2.2	69.9±1.9	63.6±2.0	71.4±3.1	87.1±1.7	45.0±0.6	0.81	4.5

3.3.3 Q3: How Do Different GNN Layers Perform?

A large number of GNN layers have been proposed in the past (see Section 2.5 for an overview). As always, each paper claims theoretical or empirical advantages over other methods. Previous work by Shchur et al. (2018) has shown that most methods apparently do not outperform GCN layers, one of the first and arguably one of the simplest formulations of a spatial GNN layer, on all datasets.

Experiment

To compare different proposed layers, we use the same experimental setup as the baseline (which uses GCN layers), and only vary the type of GNN layer. We evaluate five different layers:

GCN GCN layers (see Section 2.5.3) form our baseline.

GAT GAT layers (see Section 2.5.3.4) promise to better filter irrelevant neighbouring nodes through an attention mechanism.

Graph Isomorphism Network (GIN) GIN layers (see Section 2.5.3.6) avoid theoretical weaknesses of standard GNN models. We use a two-layer MLP with the same number of channels for the GIN layer, and learn ϵ.

GraphSAGE GraphSAGE layers (see Section 2.5.3.3) aggregate only parts of the neighbourhood, allowing the application to larger graphs.

MLP The MLP model does not use edge information and is a simple baseline.

Results and Discussion

Results are shown in Table 3.5. We can draw several conclusions:

- There is no single best GNN layer.
- Over all datasets, GCNs layers perform very well. They form an excellent starting point for constructing GNN architectures. This mirrors the findings of Shchur et al. (2018).
- Despite their theoretical power, GIN layers fail on node prediction tasks. However, they perform well on graph prediction datasets, particularly the *nci1* dataset.
- On *proteins* and *imdb*, the MLP model performs on par with the worst GNN models despite not using any graph information. This suggests that graph topology is not as important for these datasets (see also Section 3.3.1).
- However, the best GNN models still profit from graph topology.

This answers **Q3**: There is no single best GNN layer, but the simple GCN layer seems to provide a useful starting point for any task.

Table 3.6: Accuracy and 95%-confidence bound (in percent), comparing different encoders. Our baseline is a linear encoder (*).

	citeseer	*cora*	*pubmed*	*mutag*	*proteins*	*imdb*	*rdt-b*	*rdt-12k*	Rel. Perf.	Mean Rank
None	61.0±1.5	75.7±1.7	72.6±1.6	75.1±4.3	71.7±2.5	74.0±2.6	89.8±1.6	48.1±0.8	0.99	2.375
Linear (*)	53.8±3.4	70.4±2.5	68.0±3.6	74.5±4.1	71.8±2.7	72.9±3.4	91.0±1.2	48.8±0.5	0.96	2.625
2 layers	44.8±5.5	59.0±8.8	63.9±3.1	73.9±5.5	72.3±2.9	72.7±2.9	92.1±0.9	49.2±0.9	0.91	2.875
3 layers	43.1±5.0	46.4±6.2	56.2±2.1	72.9±3.2	71.6±2.2	71.2±3.3	92.2±1.0	49.7±0.3	0.87	3.75
4 layers	33.2±3.2	38.6±5.8	55.0±1.6	71.8±6.1	73.4±1.8	71.6±3.3	92.9±1.3	50.3±0.7	0.84	3.375

3.3.4 Q4: How Do We Encode Node Features?

The input features in many problems are of a different type than those transformed by neural network layers. For example, the citation datasets provide bag-of-word features, i.e. relatively sparse features. In contrast, neural network layers produce dense and often non-linearly interlaced feature spaces. Depending on the task faced, we might therefore decide to split the initial feature encoding from feature propagation.

Experiment

We compare five different encoders:

None uses no encoder. Instead, the original features are fed to the GNN.

Linear uses a learnable linear transformation.

2, 3, and 4 layers use MLPs to transform the features. These are 2–4 layers, using the ReLU activation function.

Results and Discussion

Table 3.6 shows the results. There are three takeaway messages:

- None of the citation datasets profits from the increase in model capacity a more powerful encoder provides. For these datasets, directly feeding the node features to the first GNN layer performs best.
- For graph-level predictions, a different image emerges. On *mutag*, *proteins*, and *imdb*, i.e. the graph-level tasks with smaller graphs, smaller encoders perform well. For the larger *rdt-b* and *rdt-12k* tasks, deeper encoders significantly outperform shallower encoders.
- Encoding node features with a learned linear representation seems to provide a good tradeoff, with failing on no dataset.

This answers **Q4**: There is again no clear best-performing encoder. However, a linear encoding performs well over all datasets.

Table 3.7: Accuracy and 95%-confidence bound (in percent), comparing different global pooling methods to the mean pooling baseline (*).

	mutag	***proteins***	***imdb***	***rdt-b***	***rdt-12k***	**Rel. Perf.**	**Mean Rank**
Mean (*)	75.6±5.0	72.0±2.7	73.0±2.8	90.7±1.2	48.4±0.8	0.97	3.4
Sum	74.1±5.5	75.0±1.7	72.9±3.0	91.6±1.0	49.7±0.6	0.98	2.6
Max	73.5±5.5	74.9±2.7	73.3±2.5	89.5±1.0	47.5±0.8	0.97	3.8
Mean, Sum, Max	79.2±5.3	75.1±2.6	73.7±2.5	92.4±0.7	49.9±1.0	1.00	1.0
Attention	73.9±4.1	73.3±2.8	72.5±2.1	88.0±1.7	48.8±0.6	0.96	4.2
Sort	70.3±5.7	69.4±2.9	71.9±4.2	76.7±3.5	35.8±1.8	0.87	6.0

3.3.5 Q5: How Do We Global-Pool Graphs?

In graph prediction tasks, the node features have to be pooled into a graph-level representation, conceptually the equivalent to a global pooling for CNNs. We are interested in which of the proposed pooling methods shows performance advantages.

Experiment

Any order-invariant function can be used for global-pooling a graph. Traditionally, a simple mean over all features is used, but several other methods are common:

Mean Pooling Mean pooling computes final graph-level features by computing the mean over all node features: $\mathcal{G} = \frac{1}{|V|} \sum_{v_i \in V} v_i$. Accordingly, it is insensitive to the number of nodes.

Sum Pooling Sum pooling computes graph-level features by computing the feature-wise sum over all nodes: $\mathcal{G} = \sum_{v_i \in V} v_i$. It is sensitive to the number of nodes.

Max Pooling Max pooling computes graph-level features by taking the feature-wise maximum over all nodes: $\mathcal{G}^k = \max_{v_i \in V} v_i^k$ for the kth feature channel. It is also insensitive to the number of nodes.

Combined Pooling Concatenating the previous three methods increases final graph feature size, but allows more expressive graph features, which can combine both count-sensitive and count-insensitive features.

Attention Pooling Li, Tarlow, et al. (2017) introduced attention pooling, which computes a weighted sum of node features by computing node attention weights using an MLP: $\alpha_i = \text{softmax}_V \text{MLP}_{\text{att}} (\mathcal{V}_i)$. This is then used to weight the final feature representations using $\mathcal{G} = \sum_{v_i \in V} \alpha_i \odot v_i$.

Sort Pooling Sort pooling (Zhang, Cui, et al. 2018) sorts node outputs according to their last channel and concatenates the first k nodes' features.

Results and Discussion

As Table 3.7 shows, there are large performance differences between the global pooling methods. There are three main takeaway messages:

- Attention and sort pooling introduce complexity into the model, requiring to either compute attention or even to sort the nodes. The latter increases runtime requirements superlinearly. However, they generally perform worse than the simple alternatives of summing up node features, averaging them, or computing the maximum.
- While mean, sum, and max pooling methods perform well, combining all of them increases performance significantly. The increase in performance compared to each of the submethods suggests that each of them extracts features the others are missing. Combining these features apparently allows more performance gains.
- If one cannot combine all simple pooling methods, sum pooling appears to perform best. We assume this is because it can distinguish between the numbers of nodes, similar to the theoretical advantages of GIN layers (see Section 2.5.3.6).

This answers **Q5**: Concatenating mean, sum, and max pooling methods is to be preferred.

3.3.6 Q6: How Do We Construct the Final Graph Output?

While our default baseline uses a simple two-layer MLP to output the final prediction, other configurations are possible. Specifically, we can either directly use the output of the final GNN layer—modified by global pooling if we are interested in graph-level instead of node-level predictions—or we can increase the complexity of the output model. This output model is applied after all graph-related steps have been executed, and—if necessary— the global pooling has happened.

Experiment

We compare a total of five different configurations. These mirror the configuration for the input encoder (see Section 3.3.4):

None uses no output model. Instead, the features produced by the last GNN layer are used as a prediction.

Linear uses a learnable linear transformation.

2, 3, and 4 layers use MLPs to transform the last GNN layer's features. These are 2–4 layers, using the ReLU activation function.

Table 3.8: Accuracy and 95%-confidence bound (in percent), comparing different output models to the two-layer MLP (*) used as a baseline.

	citeseer	*cora*	*pubmed*	*proteins*	*nci1*	*imdb*	*rdt-b*	*rdt-12k*	Rel. Perf.	Mean Rank
None	5.1±0.7	12.6±1.4	3.0±0.7	63.5±2.4	58.3±2.8	71.0±4.0	69.9±2.3	31.9±0.5	0.54	5.0
Linear	58.0±1.1	72.0±1.2	70.0±1.1	68.8±2.9	73.8±3.7	72.7±3.0	89.6±1.4	47.4±0.7	0.98	2.375
2 layers (*)	54.6±1.0	69.7±0.7	68.0±1.0	70.5±2.3	75.9±1.7	72.5±2.5	91.8±1.1	49.2±0.5	0.98	1.875
3 layers	54.7±1.9	69.2±1.0	68.0±1.4	72.2±2.9	73.9±1.7	71.5±2.8	91.1±1.1	48.3±0.7	0.98	2.25
4 layers	51.2±1.7	68.3±1.5	67.1±1.4	70.3±2.2	71.9±2.6	71.5±3.5	90.4±1.2	47.8±0.5	0.96	3.5

Results and Discussion

The results are listed in Table 3.8. There are three main takeaway messages:

- There is a clear and significant disadvantage in using the GNN layers' output directly, particularly on node-level prediction tasks. On these, the models fail to learn anything at all, and produce close to random output. Only on *imdb* does it perform close to the other models. This contrasts with the results from Section 3.3.4, where using no encoder performed well on these tasks.

- For node-level prediction tasks, large output models seem unnecessary and even counter-productive, with a linear output projection performing best.

- For graph-level prediction tasks, the sweet spot for depth is larger than in node-level prediction tasks, at about 2–3 layers. We assume this is because the aggregated node information in a graph prediction task has to be aggregated and then transformed, for which a certain minimum expressive power of the output model is necessary. Greater depth is, however, still counter-productive.

This answers **Q6**: Either a linear projection or a two-layer MLP suffice as an output model, with the former preferable for node-level and the latter for graph-level tasks.

3.3.7 Q7: How Deep should GNN models be?

Especially since the introduction of residual connections (He et al. 2015), CNNs have grown ever deeper. Empirically, these deeper models generally outperform shallower models assuming sufficient regularization. Theoretically, removing a layer might need exponentially more nodes in the existing layers for the same representational power (Telgarsky 2016). On the other hand, most proposed GNN models use three or fewer layers of GNNs. As with CNNs, processing capability and receptive field of GNNs are closely linked: Each additional layer increases the receptive field by one hop and adds an additional processing step. NT and Maehara (2019) claim that simple GNN formulations essentially act as a low-pass filter, smoothing node features and therefore actually removing discriminative ability.

Experiment

We are therefore interested in the performance of deeper GNNs models. We evaluate between one and sixteen GNN layers.

We are also interested in the performance impact of JK connections (see Section 3.2.2): Do they allow us to stack GNN layers arbitrarily deep by allowing the network to concentrate on earlier layers if necessary?

Results and Discussion

Table 3.9 shows performance values over all datasets. Fig. 3.5 shows performance visually and includes JK connections. We can draw several conclusions:

- Node-level tasks suffer greatly from deeper GNN models. *citeseer* is particularly vulnerable to this, with performance declining by more than 10 pp from one to six layers. However, two or three GNN layers work well even here.
- Performance on graph-level tasks remains roughly constant with deeper models.
- JK connections do not alleviate performance differences with deeper networks. As Fig. 3.5 shows, performance and particularly the performance disadvantage of deeper networks on node-level predictions remain constant with or without JK connections. This suggests that the claimed advantage of being able to only consider greater neighbourhoods when necessary and ignore them otherwise does not materialize.
- Two to four GNN layers appear to be a good general-purpose value.

This answers **Q7**: Deeper networks hurt especially on node-level tasks and do not provide much benefit on graph-level tasks. Two to four GNN layers are a good starting point.

3.3.8 Q8: Which Tweaks Improve GNN Performance?

Machine learning models often depend on tweaks to maximize performance. We are interested in which of the often-used tweaks from Section 3.2.2 improve performance.

Experiment

We evaluate the influence on the performance of the following tweaks:

Dropout uses a dropout layer after each GNN layer. This randomly zeros features with a chance of 50% to combat overfitting.

No JK The baseline model uses JK connections, i.e. the final GNN output is the concatenated output of all previous GNN layers. This should allow the model to

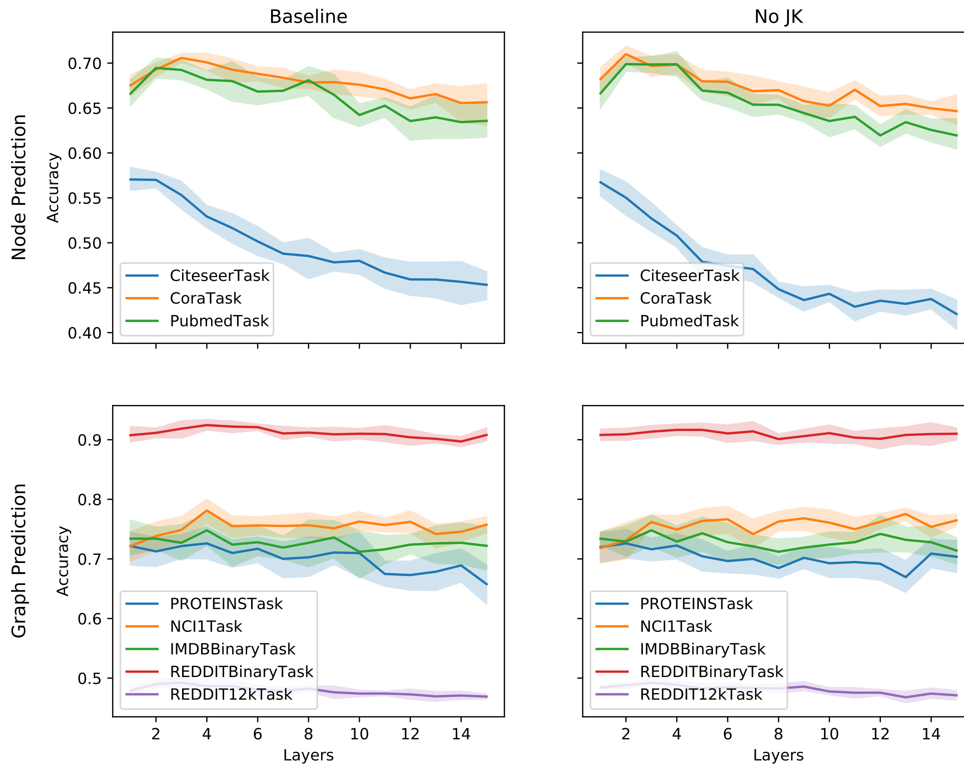


Fig. 3.5: Performance on the simple graph datasets dependent on the depth of the network. We compare both the baseline (left) to the baseline without JK connections (right) on both graph and node prediction tasks.

Table 3.9: Accuracy and 95%-confidence bound (in percent), comparing the performance of different depths. The baseline (*) uses 3 layers.

	citeseer	*cora*	*pubmed*	*proteins*	*nci1*	*imdb*	*rdt-b*	*rdt-12k*	Rel. Perf.	Mean Rank
1	57.1±1.4	67.5±1.2	66.6±1.6	72.2±3.1	72.1±2.7	73.4±2.9	90.8±1.3	47.9±0.7	0.97	7.375
2	57.0±0.9	69.3±0.7	69.5±1.2	71.3±2.6	73.8±2.5	73.4±1.9	91.1±0.8	49.0±0.7	0.98	4.5
3 (*)	55.3±1.6	70.6±0.5	69.2±1.1	72.2±2.1	74.9±2.4	72.7±3.2	91.9±1.5	49.2±0.9	0.99	4.0
4	52.9±1.3	70.1±1.0	68.1±1.0	72.6±2.9	78.1±2.0	74.8±2.9	92.5±0.9	48.6±0.8	0.99	2.0
5	51.6±1.8	69.3±1.4	68.0±2.3	71.0±2.4	75.5±2.0	72.4±3.7	92.2±1.0	48.6±1.0	0.97	5.75
6	50.1±1.7	68.8±0.8	66.8±1.7	71.7±2.2	75.6±1.5	72.8±2.5	92.1±0.5	48.0±0.8	0.96	5.375
7	48.8±1.1	68.4±1.1	66.9±1.1	70.0±3.2	75.5±2.0	71.9±2.1	91.1±1.2	47.9±0.5	0.95	8.125
8	48.5±2.3	67.8±0.7	68.1±1.6	70.3±3.7	75.6±2.2	72.7±4.0	91.2±0.7	48.2±0.5	0.96	6.5
9	47.8±1.1	67.9±1.4	66.5±2.4	71.1±2.3	75.1±2.0	73.6±3.4	90.9±1.2	47.6±1.1	0.95	7.875
10	48.0±1.4	67.6±1.3	64.2±1.3	71.0±4.1	76.3±1.6	71.2±4.3	91.0±0.9	47.4±0.6	0.95	9.125
11	46.7±1.7	67.1±1.1	65.3±1.1	67.5±2.3	75.7±1.2	71.6±2.6	91.0±1.4	47.4±0.4	0.94	10.375
12	45.9±1.9	66.1±1.2	63.6±1.9	67.3±2.7	76.2±1.9	72.4±1.7	90.4±1.4	47.3±0.9	0.93	11.5
13	45.9±2.1	66.6±1.0	64.0±2.5	67.8±3.2	74.2±2.1	72.6±3.1	90.2±0.8	46.9±0.9	0.93	12.375
14	45.7±2.5	65.6±1.9	63.4±2.1	68.9±2.9	74.5±1.7	72.7±3.7	89.7±0.9	47.1±0.7	0.93	12.75
15	45.3±1.7	65.6±2.5	63.6±2.1	65.8±3.4	75.7±1.4	72.2±3.8	90.8±1.1	46.9±0.4	0.93	12.375

concentrate on only a specific neighbourhood while keeping the ability for long-range reasoning intact. However Section 3.3.7 has shown that this might not work. We evaluate the models without JK.

BatchNorm is often used in CNNs to speed up convergence. Since our baseline model includes BatchNorm, we evaluate a model that does not include it.

Results and Discussion

Table 3.10 shows the results of varying the tweaks. We can make several interesting observations:

- Removing BatchNorm from the baseline results in decreased performance (about 7 pp compared to the baseline), primarily on the node prediction and larger graph prediction datasets.
- As already observed in Section 3.3.7, JK does not show large improvements, and therefore removing it changes does not change the performance much.
- Dropout shows some small gains in performance on the node prediction tasks, but also performs extremely similar to the baseline. Together with the performance of very deep models observed in Section 3.3.7, we can hypothesize that GCN layers have a very powerful in-built regularizer. This meshes with the low-pass filter argument of NT and Maehara (2019).

This answers **Q8**: The only tweak that shows clear performance advantages is the inclusion of BatchNorm.

Table 3.10: Accuracy and 95%-confidence bound (in percent), comparing tweaks made to the baseline (*).

	citeseer	*cora*	*pubmed*	*proteins*	*nci1*	*imdb*	*rdt-b*	*rdt-12k*	**Rel. Perf.**	**Mean Rank**
Baseline	63.1±1.2	75.6±1.0	74.5±1.0	74.4±2.2	74.8±3.8	73.2±3.5	91.7±1.3	48.8±0.7	0.99	1.75
No BatchNorm	49.3±3.7	65.4±2.6	65.3±4.4	74.4±3.0	73.6±1.8	73.5±3.1	87.7±1.6	47.2±0.8	0.92	3.50
Dropout	66.4±1.1	75.4±1.1	72.8±1.1	74.0±2.5	73.7±3.8	72.3±2.3	91.4±1.1	48.5±0.9	0.99	2.75
No JK	60.9±1.5	74.5±0.9	73.6±1.3	74.5±2.1	77.1±1.4	73.9±2.1	91.7±1.5	49.3±0.6	0.99	2

3.3.9 Q9: Is it Helpful to Separate Processing and Propagation?

One potential disadvantage of GNN layers is that they link processing power with receptive field: Every time a GCN performs a single non-linear transformation, it also increases its receptive field by one hop. While this is similar to most CNNs with kernel sizes greater than 1×1, these usually have much larger inputs: VGG-16, for example, uses image inputs of 224×224 pixels. In contrast, the graph datasets used in this chapter (see Tables 3.1 and 3.2) have an average size of less than 400 nodes. This leads to an average shortest path length between nodes in these graphs of less than 7, compared to an average shortest path length of about 150 for a VGG-16-sized image.

Accordingly, while the centre feature representation of a CNN will need about 75 stacked 3×3 kernels for its receptive field to cover most of the image, a GNN will already cover most of the graph at seven layers.

Experiment

Similar to a CNN's 1×1 convolutions, we can introduce multiple processing layers by including node-wise MLPs into the GNN. This processes node features without taking neighbours into account and should increase the representation capabilities of the model while keeping the respective receptive fields constant. We evaluate one to four GNN layers, which are augmented by up to three MLP layers each. Each MLP layer uses batch normalization and dropout.

Results and Discussion

Table 3.11 shows the results. We can make several observations:

- Starting with three GNN layers, the additional representational capability of processing layers is always counter-productive.
- For smaller networks, processing layers only sometimes add small performance increases.
- Introducing processing layers generally appears to be counter-productive.

This answers **Q9**: Stacking GNN layers seems to suffice for these problems. Adding separate processing layers is unnecessary.

Table 3.11: Accuracy and 95%-confidence bound (in percent), comparing different combinations of GNN layers with MLP processing layers. The baseline (*) uses three GNN layers.

	citeseer	*cora*	*pubmed*	*proteins*	*nci1*	*imdb*	*rdt-b*	*rdt-12k*	Rel. Perf.	Mean Rank
1GNN, 0MLP	55.9±1.7	67.1±1.5	66.4±1.4	71.0±3.2	72.0±1.6	73.8±2.4	90.5±1.6	48.2±0.4	0.97	11.5
1GNN, 1MLP	53.6±1.3	64.9±0.9	63.6±1.5	71.4±3.4	74.0±2.3	73.2±3.1	91.8±0.8	47.8±0.9	0.96	10.875
1GNN, 2MLP	49.9±1.3	61.3±1.2	59.8±1.8	73.1±3.0	73.5±2.8	74.7±2.7	91.4±0.9	48.4±0.8	0.94	9.625
1GNN, 3MLP	47.4±2.0	56.7±1.9	57.0±2.0	72.4±2.3	74.6±2.1	72.9±2.6	91.6±1.3	48.2±0.8	0.92	12.75
2GNN, 0MLP	56.4±1.5	71.3±1.0	69.3±1.3	72.3±2.1	72.7±2.3	71.7±3.1	90.8±1.4	49.1±0.3	0.98	9.75
2GNN, 1MLP	52.1±1.3	65.3±0.9	64.5±1.5	70.5±3.3	78.1±2.2	74.1±2.6	90.2±1.2	49.1±0.7	0.96	8.125
2GNN, 2MLP	48.1±1.2	59.8±0.5	59.6±1.7	70.3±2.8	77.4±2.1	74.4±1.8	90.2±1.7	47.9±0.4	0.93	12.5
2GNN, 3MLP	40.3±2.5	53.1±1.3	55.0±1.4	70.5±2.6	78.0±1.6	72.1±3.1	92.2±0.9	48.6±0.8	0.89	11.75
3GNN, 0MLP (*)	55.9±1.1	71.1±0.7	68.1±0.9	72.3±2.9	76.3±2.2	73.4±4.0	92.1±1.2	48.8±0.6	0.99	5.937 5
3GNN, 1MLP	47.9±1.8	63.7±0.8	63.0±0.6	68.5±3.4	77.4±1.6	73.8±2.6	91.7±0.9	49.2±0.6	0.94	8.625
3GNN, 2MLP	41.2±1.9	56.0±1.4	56.3±2.1	70.0±2.8	76.8±1.7	74.3±2.9	91.8±1.2	49.0±0.6	0.91	10.25
3GNN, 3MLP	35.4±2.3	49.6±2.1	52.1±2.4	70.3±2.3	78.4±1.2	72.3±3.3	91.9±0.9	48.2±0.8	0.87	13.812 5
4GNN, 0MLP	53.0±1.6	70.3±1.2	67.7±1.9	71.0±2.6	75.8±2.4	72.4±2.4	91.5±0.6	49.1±0.5	0.97	8.875
4GNN, 1MLP	45.2±1.6	63.2±1.1	62.9±2.0	68.1±2.8	76.9±1.4	71.8±4.3	91.9±0.8	48.6±0.7	0.93	11.937 5
4GNN, 2MLP	39.9±2.1	55.2±1.9	55.4±2.5	69.4±2.7	76.1±2.1	73.1±2.7	91.6±1.6	48.1±0.8	0.89	15.0
4GNN, 3MLP	33.5±2.4	46.3±2.1	51.3±3.0	67.1±3.0	77.5±1.7	73.9±2.8	90.8±1.1	48.0±0.8	0.85	17.187 5

3.4 Conclusion

In this chapter, we have conducted nine experiments with the goal of evaluating GNN architecture choices. We have evaluated both basic attributes of GNN layers (their stability with respect to initialization and whether graph information is useful), architecture choices (encoder and output layer choices, global pooling layers and tweaks) and looked into edge-cases of GNN models (making them deeper and using separate processing layers).

These experiments have primarily gained us three valuable insights into constructing GNN models:

- Using GCN layers is a very good initial approach (see Section 3.3.3). While other layer types might perform better on single problems, models using GCN layers consistently perform well.
- GNN model training is unstable, and some training runs result in significantly worse performance depending on the random initialization (see Section 3.3.2). This suggests we might either not have a useful method to initialize GNN models or that our current models are generally unstable. Both suggest future research directions.
- Contrary to other deep learning models, GNN models do not profit from large network depths (see Section 3.3.7). This situation is somewhat similar to CNNs before the introduction of residual connections. However, simple residual connections to not alleviate the problem for GNN models, and neither do JK connections despite their promise.

In summary, the field is in a state of large and fast growth. There are many publications, but introduced layers do not generally show better performances in a fair

comparison. We hope for a more unified field, with a number of standard approaches either arrived at by empirical performance advantages or theoretical analysis.

4 Modifying Graph Topology

The GNN models evaluated in the previous chapters have concentrated on updating features of a graph, as have most published models. However, modifying the graph structure itself holds significant promise: Pooling multiple nodes together might both identify clusters (feature- or structure-based) and reduce computational requirements by reducing the number of nodes. Together, these promise to abstract from flat nodes to hierarchical sets of nodes (see Fig. 4.1).

This chapter is structured as follows: We first motivate the desire to modify graph topology (Section 4.1) and introduce other pooling methods (Section 4.2), showcasing why these do not fulfil our requirements. EdgePool, our proposed method, is introduced in Section 4.3. Section 4.4 introduces our experiments, particularly concentrating on comparisons to the other pooling methods, integrating EdgePool in standard models for graph-level and node-level classifications, and evaluating the impact on computational performance.

4.1 Motivation

Intelligent node pooling operations are a stepping stone towards the goal of allowing GNNs to modify graph *topology* instead of only node features.

We propose a new pooling layer based on edge contractions[1]. EdgePool learns a localized and sparse hard pooling transform. We do this by viewing the task not as

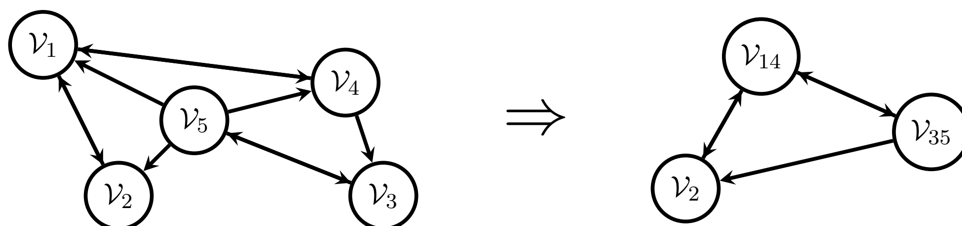


Fig. 4.1: EdgePool in action: The layer takes an input graph (left) and produces a smaller output graph with fewer nodes (right). In this example, nodes v_1 and v_4 are pooled into $v_{(1,4)}$ while nodes v_3 and v_5 are pooled into node $v_{(3,5)}$. Node v_2 is not pooled.

[1]Parts of this chapter have been previously published (Diehl, Brunner, et al. 2019b; Diehl 2019b).

choosing nodes but as choosing edges and pooling the connected nodes. This immediately and naturally takes the graph structure into account and ensures that we never drop nodes completely.

The main advantages of our proposed EdgePool layer are:

- EdgePool outperforms other pooling methods.
- EdgePool can be integrated in existing graph classification architectures.
- EdgePool can be used for node classification and improves performance.
- EdgePool improves memory efficiency even compared to sparse baseline GNN models.

4.2 Other Pooling Methods

Graph pooling strategies can be divided into two types: We can either use *fixed* pooling methods, usually based on graph topology, or use *learned* pooling methods. We concentrate on comparisons with learned pooling methods, since these appear to outperform fixed pooling methods.

4.2.1 DiffPool

Ying, You, et al. (2018) were the first to propose a learned pooling layer. DiffPool learns to soft-assign each node to a fixed number of clusters based on their features. For a graph with $|V|$, the DiffPool layer learns a cluster assignment matrix $\mathbf{S} \in \mathbb{R}^{|V| \times |V'|}$ which maps the nodes to a fixed number $|V'|$ of new cluster nodes. This cluster assignment matrix is created through a separate GNN layer,

$$\mathbf{S} = \mathrm{softmax}_{\mathrm{clusters}} \left(\mathrm{GNN} \left(\mathbf{A}, \mathcal{V} \right) \right), \tag{4.1}$$

with each node being soft-assigned to each output cluster. The cluster assignment matrix is then used to compute both new node features and a new adjacency matrix using

$$\mathcal{V}' = \mathbf{S}^T \mathcal{V} \qquad \text{and} \tag{4.2}$$

$$\mathbf{A}' = \mathbf{S}^T \mathbf{A} \mathbf{S} \tag{4.3}$$

respectively.

DiffPool works well, but suffers from three disadvantages:

- The number of clusters has to be chosen in advance, which might cause performance issues when used on datasets with different graph sizes.

- Cluster assignment is based only on node features. Nodes are assigned to the same cluster based only on their features, ignoring graph topology. Accordingly, very distant nodes might be assigned to the same cluster.
- The method soft-assigns clusters. Therefore, the cluster assignment matrix a dense $\mathbb{R}^{|V|\times|V'|}$. Since $|V'|$ is usually chosen according to the total number of nodes, the cluster assignment matrix scales quadratically with the number of nodes. Even worse, the adjacency matrix following the pooling is dense, scaling in $\mathcal{O}\left(|V'|^2\right)$. This both means memory usage scales very fast and makes the usual GNN implementations useless.

DiffPool also requires several auxiliary objectives (link prediction, node feature ℓ_2 regularization, cluster assignment entropy regularization) to make training work well.

4.2.2 TopKPool

The Graph U-Net model, introduced by Gao and Ji (2019), uses a simple top-k choice of nodes for its gPool layer, learning a node score and dropping all but the top nodes. For this, they train a linear projection vector p, which takes each node's features $\mathcal{V}_i$ and projects them into a single score y_i

$$y_i = \frac{\mathcal{V}_i p}{\|p\|}. \tag{4.4}$$

They then choose those nodes with the largest scores, usually the top 80%. They preserve edges and features of these nodes and remove all other edges and features. Cangea et al. (2018) later applied this to graph classification.

While this approach is both sparse and variable in graph size, its node choice is dependent on global state. This introduces two new issues: (a) Adding nodes to a graph can change the pooling result of the whole graph. (b) Whole areas of a graph might see no node chosen, which causes loss of information.

4.2.3 SAGPool

Lee et al. (2019) introduced Self-Attention Graph Pooling (SAGPool). A variant of TopKPool, SAGPool replaces the linear projection of Eq. (4.4) with a GNN-computed self-attention score:

$$y_i = \sigma\left(\mathrm{GNN}\left(\mathbf{A}, \mathcal{V}\right)\right). \tag{4.5}$$

This no longer uses only node features to compute node scores but uses graph convolutions to take neighbouring node features into account Based on y_i, they also choose a fraction of the nodes and their incident edges to keep.

While their method improves TopKPool qualitatively (see Section 4.4.2), the disadvantages remain.

4.3 EdgePool

We base our pooling operation on edge contractions. Contracting the edge $e_{ij} = (v_i, v_j)$ introduces the new node $v_{(ij)}$ and new edges such that $v_{(ij)}$ is adjacent to all nodes either v_i or v_j have been adjacent to. v_i, v_j, and all their incident edges are then deleted from the graph. Since edge contractions are commutative, we can also define an edge set contraction. By constructing the set such that no two edges are incident to the same node, we can simply apply the naive notion of single-edge contraction multiple times.

Intuitively, we choose a single edge to contract by merging its nodes. This new node is then connected to all nodes the merged nodes had been connected to. We repeat this procedure multiple times, taking care not to include a newly-merged node into it. See Fig. 4.2 for an example on a graph from the *proteins* dataset.

4.3.1 Choosing Edges

Given the conditions mentioned above, we greedily choose edges by computing a score for each edge, then iteratively contract the highest-scoring edge which does not have a newly-merged node incident.

In our procedure, we compute raw scores from the concatenated node features. For an edge e_{ij} from node v_i to node v_j, we compute the raw edge score r_{ij} as

$$r_{ij} = \text{MLP}\left(\mathcal{V}_i \| \mathcal{V}_j\right), \tag{4.6}$$

concatenating the raw node features $\mathcal{V}_i$ and $\mathcal{V}_j$.

To compute the final node score s_{ij} for an edge, we employ a local softmax normalization over all edges of a node[2].

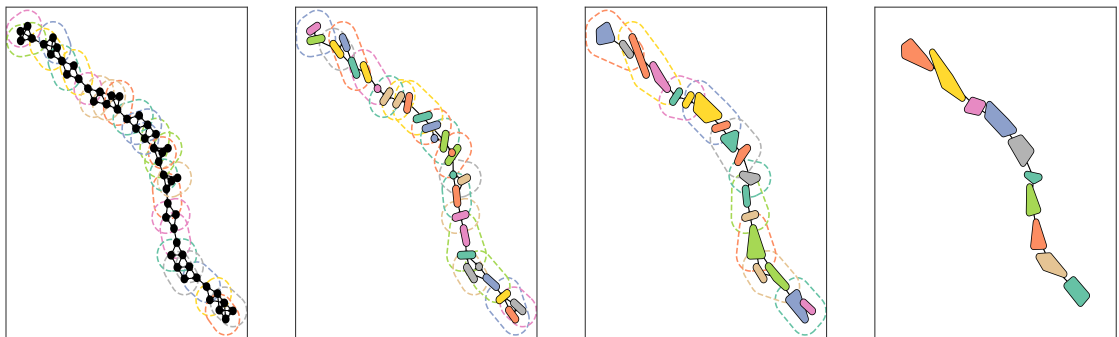


Fig. 4.2: Edge pooling in action on a graph from the *proteins* dataset. The original graph (leftmost) is pooled three times and results in the graph depicted to the right. In each step, nodes that will be merged are surrounded by a dashed line of a random colour. In the next step, the nodes are drawn as their convex hull, filled with the same colour. Notice how the the pooled graph keeps the mostly-linear structure of the original graph.

[2]We experimented with a simple tanh gating function, but found softmax normalization to perform better.

We also modify the final score such that the mean of the score range lies at 1. Later on, this enables us to include the score in the unpooling procedure without numerical stability issues. We also found this to lead to better performance in the graph classification task, which we believe is due to better gradient flow. The final score then becomes:

$$s_{ij} = 0.5 + \mathrm{softmax}_{r_{\star j}}(r_{ij}). \tag{4.7}$$

Given the edge scores, we now iteratively contract edges according to the scores, ignoring those which have a newly-merged node incident. An illustration of the process is depicted in Fig. 4.3.

Note that this will always pool roughly 50% of the total nodes. Contrary to DiffPool and TopKPool, this ratio cannot be changed.

4.3.2 Computing New Node Features

There are many strategies for combining the features of pairs of nodes. In particular, we are not restricted to symmetric functions since the edges chosen have a specific direction. Nonetheless, we found that taking the sum of the node features works well.

We use gating and multiply the combined node features by the edge score:

$$\hat{\mathcal{V}}_{(ij)} = s_{ij}\left(\mathcal{V}_i + \mathcal{V}_j\right). \tag{4.8}$$

This allows the gradient to flow through the scores and thereby enables learning the score function weights.

4.3.3 Integrating Edge Features

EdgePool can be updated to take edge features $\mathcal{E}_{ij}$ of edge e_{ij} into account. To do so, we have to include them in the raw score computation. The simplest approach is to concatenate them, changing Eq. (4.6) to

$$r_{ij} = \mathrm{MLP}\left(\mathcal{V}_i \| \mathcal{V}_j \| \mathcal{E}_{ij}\right). \tag{4.9}$$

Additionally, we will likely have to change the procedure to compute new node features; we propose using an MLP that operates on the concatenation of both nodes' features, the features of the chosen edge, and the features of the reverse edge if it exists. This changes Eq. (4.8) to

$$\mathcal{V}_{(ij)} = s_{ij}\mathrm{MLP}\left(\mathcal{V}_i \| \mathcal{V}_j \| \mathcal{E}_{ij} \| \mathcal{E}_{ji}\right). \tag{4.10}$$

Lastly, we need a procedure to combine the edge features of edges that ended at both merged nodes and will therefore be merged. We believe a simple sum should work well here, too. However, we have not conducted experiments on this.

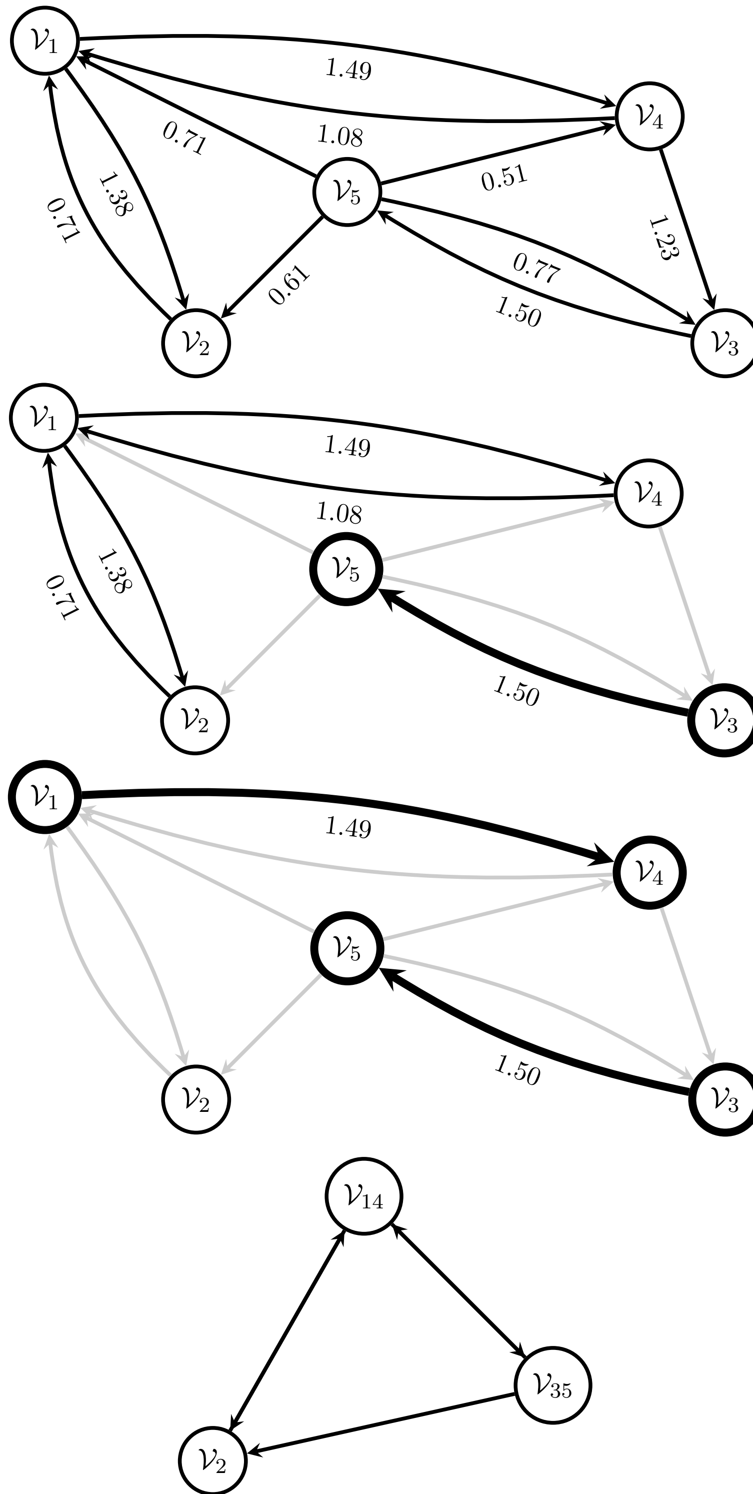


Fig. 4.3: Edge score computation. We compute the edge scores, then greedily choose edges. We first merge nodes v_3 and v_5. Then, we merge nodes v_1 and v_4, which is the edge with the highest score that is not adjacent to a previously chosen node. Node v_2 is left unmerged.

4.3.4 Unpooling EdgePool

To use pooling in the context of node-level prediction, an unpooling operation is necessary. To do so, each EdgePool layer also emits the mapping of each of the previous graph's nodes to the newly-pooled graph's nodes. When unpooling, we then create an inverse mapping of pooled nodes to unpooled nodes. Since we assign each node to exactly one merged node, this mapping can be chained through many pooling layers. We compute features for the unpooled nodes by dividing by the corresponding edge scores:

$$\mathcal{V}'_i = \mathcal{V}'_j = \frac{\mathcal{V}_{(ij)}}{s_{ij}}. \tag{4.11}$$

Integrating the edge score s_{ij} here again allows for better gradient flow.

4.3.5 Computational Performance

Given the EdgePool procedure, we immediately see that EdgePool can operate on sparse representations. When doing so, both runtime and memory scales linearly in the number of edges. This particularly avoids the scaling issues of DiffPool's cluster assignment matrix.

Additionally, EdgePool is locally independent: As long as the node scores of two nodes v and w and of their neighbours do not change (by changing nodes within the receptive fields), the choice of edge will not change. Accordingly, EdgePool does not have to be computed for the whole graph at once. If the graph changes, only the pooling local to the changed areas needs to be updated.

4.4 Experiments and Discussion

We design our experiments to answer four questions:

Q1: Does EdgePool outperform alternative pooling methods?

Q2: Can EdgePool be used as a plug-and-play addition for any GNN?

Q3: Can EdgePool be used for node classification?

Q4: How does EdgePool impact performance?

4.4.1 General Setup and Training

We evaluate our models on multiple graph and node classification datasets, and share most of the training procedures between all models. We conduct 10-fold cross-validation for all datasets and report mean and 95% confidence bounds. We choose all folds at random, eschewing the default planetoid split.

We implemented the models using PyTorch (Paszke et al. 2017) and in particular the *pytorch-geometric* library (Fey and Lenssen 2019). Experiments were conducted on several Geforce 1080Ti GPUs in parallel, leveraging Singularity containers (Kurtzer et al. 2017) for reproducibility.

Table 4.1: Attributes of the graph classification datasets. Note how *collab* is significantly more dense than the other datasets.

Dataset	Graphs	Avg. Nodes	Avg. Edges	Avg. Degree	Features	Classes	Density
proteins	1 113	39.1	72.8	1.9	3	2	0.097 7
rdt-b	2 000	429.6	497.8	1.2	1	2	0.005 3
rdt-12k	11 929	391.4	456.9	1.2	1	11	0.005 9
collab	5 000	74.5	2 457.8	33.0	1	3	0.442 9

Table 4.2: Attributes of the five node classification datasets. As can be seen, these differ in both size and the number of targets. However, they are all similarly sparse.

Dataset	Nodes	Edges	Avg. Degree	Features	Classes	Labeled Nodes	Density
citeseer	3 327	4 552	1.4	3 703	6	120 (3.64 %)	0.008 21
cora	2 708	5 278	1.9	1 433	7	140 (5.17 %)	0.001 44
pubmed	19 717	44 324	2.2	500	3	60 (0.30 %)	0.000 23
photo	7 487	119 043	15.9	745	8	160 (2.14 %)	0.002 12
computer	13 381	245 778	18.4	767	10	200 (1.49 %)	0.001 37

Graph classification datasets For graph classification, we evaluate on three of the datasets introduced in Section 3.1.2: *proteins*, *rdt-b*, and *rdt-12k*. We additionally use the *collab* dataset (Yanardag and Vishwanathan 2015). Each graph in the *collab* dataset models scientific collaborations of one researcher. The task is to classify which of three fields the researcher belongs to. Table 4.1 shows dataset details.

Node classification datasets We also evaluate EdgePool on five semi-supervised node classification datasets (see Table 4.2 for dataset details): The three citation datasets introduced in Section 3.1.1 (*cora*, *citeseer*, and *pubmed*) and the *photo* and *computer* datasets (Shchur et al. 2018). The latter are part of the Amazon co-purchasing graph. Nodes are products and edges model co-purchases between products. The goal is to predict the product category.

Each of these datasets has bag-of-word features as node features. We use 20 nodes per class as training data and 30 nodes per class as test data. Every other node is unlabelled.

Training For our training procedure, we follow Ying, You, et al. (2018) for greater comparability. Each model is trained for a total of 200 epochs using the Adam optimizer (Kingma and Ba 2014) with a learning rate of 10^{-3}, which is halved every 50 epochs. 128 graphs are batched together at each step by treating them as a single unconnected graph. We use 128 channels except for *proteins* and the node classification datasets, where we used 64. We vary other model configuration parameters depending on the experiment we conduct.

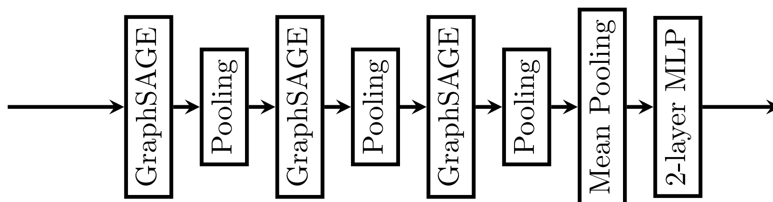


Fig. 4.4: Schematic of the architecture used for comparing pooling layer performances.

All models use both dropout and batch normalization (Ioffe and Szegedy 2015). As noted in Section 3.2.2, we found batch normalization to suffer greatly when evaluated using population statistics and instead use mini-batch statistics even during testing.

We also found using an edge score dropout significantly increased EdgePool's performance, and during training set every edge score to 0 with a chance of 0.2.

4.4.2 Q1: Does EdgePool Outperform Alternative Pooling Approaches?

Other graph pooling methods introduced in Section 4.2 have theoretical disadvantages compared to EdgePool. However, we are interested in the performance of EdgePool in practice.

Experiment

To evaluate this, we use the same architecture as used by Ying, You, et al. (2018) for DiffPool (see Fig. 4.4): The model has three SAGEConv blocks (Hamilton et al. 2017) whose outputs are globally mean-pooled and concatenated. Final classification occurs after two fully-connected layers. The base model does not pool nodes, every other model pools after every block. Note that DiffPool uses a siamese architecture, using separate SAGEConv blocks to compute cluster assignments. We restrict DiffPool to a maximum of 750 nodes per graph and set the pool ratio for both TopKPool and SAGPool to 0.5 to remain comparable to EdgePool.

Additionally, we only use the cross-entropy loss to train the model. To ensure a fair comparison, we also do this for DiffPool, which originally used three additional auxiliary losses and tasks to stabilize training and precomputed additional features.

Results and Discussion

Table 4.3 shows mean accuracy and 95% confidence bound for graph classification tasks. As can be seen, EdgePool consistently improves performance over the non-pooling models and TopKPool. We draw several conclusions:

- Performance on the *proteins* dataset, as already noted in Section 3.3.3, is very close and though EdgePool outperforms other methods, it does not do so within significance bounds.

Table 4.3: Comparing pooling strategies: Accuracy (in percent) and 95% confidence bound on benchmark datasets. [*] Ying, You, et al. (2018) use several additional techniques and auxiliary losses to stabilize training, and also include additional computed features. We report results without these.

	proteins	***rdt-b***	***rdt-12k***	***collab***
Base Model	74.1±2.0	69.9±2.3	35.1±1.0	65.4±0.9
DiffPool [*]	72.3±3.6	82.9±2.1	34.8±1.2	70.1±0.9
TopKPool	70.6±3.0	68.9±2.0	28.7±1.1	64.6±1.3
SAGPool	71.8±3.7	84.7±2.7	41.9±2.0	63.9±1.5
EdgePool	72.5±2.0	87.3±2.5	45.6±1.1	67.1±1.7

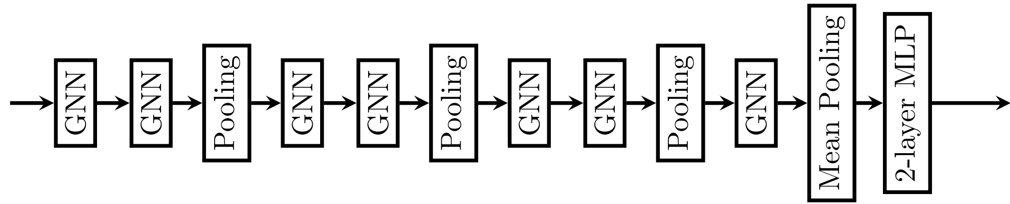


Fig. 4.5: Schematic of the architecture used to evaluate EdgePool impact on graph-level prediction performance.

- DiffPool performs slightly better on the *collab* dataset. However, it performs significantly worse on both the *rdt-b* and *rdt-12k* dataset. We assume this to be a consequence of the *collab* dataset's far higher density.

- Except for the *collab* case noted above, EdgePool outperforms all other pooling methods. On the *collab* dataset, it still outperforms TopKPool and SAGPool.

- On all datasets but *proteins*, EdgePool outperforms the baseline significantly. On *rdt-b* and *rdt-12k*, it does so by 17 pp and 10 pp respectively.

This answer **Q1**: EdgePool consistently outperforms all pooling methods but DiffPool. While DiffPool might perform better on some graphs, EdgePool scales far better and can be used on sparse and large graphs.

4.4.3 Q2: Can EdgePool Be Integrated into Existing Architectures?

It is valuable to have pooling methods that can be easily integrated into pre-existing architectures. DiffPool, for example, cannot be efficiently integrated since it transforms a spare graph into a dense graph, making standard GNN models useless.

Experiment

To evaluate whether EdgePool can be integrated into pre-existing architectures, we follow the model configuration from *pytorch-geometric*'s benchmarks (Fey and Lenssen 2019). Specifically, we use a total of seven convolutional layers, followed by a global pooling layer and two fully-connected layers. If pooling is used, it is added after every second convolutional layer (i.e. there are three pooling layers). This is shown in Fig. 4.5.

The convolutional layers we evaluate this on are GCN (Kipf and Welling 2016), GIN and GIN0 (Xu, Hu, et al. 2018), and GraphSAGE (Hamilton et al. 2017) both with and without accumulating intermediate results (SAGE nacc). Additionally, we construct a model using node-independent MLPs, in which only pooling might lead to communication between nodes.

Results and Discussion

Table 4.4 shows comparative results for different benchmark models with and without EdgePool. We make several observations:

- On a large majority of GNN/dataset combinations, EdgePool increases performance, by an average of almost 2 pp. GIN and GIN0 profit the least (mean improvement of 0.3 pp), while GraphSAGE profits the most (5.5 pp).
- Only GIN and GIN0 show cases with significant decreases in performance when using EdgePool, again on the *collab* dataset. On all other datasets, adding EdgePool either increases or does not significantly decrease performance.
- Interestingly, we can see that EdgePool allows even the MLP model to perform fairly well. This model can only rely on pooling to gain information on the neighbourhood. Nonetheless, it performs competitively with all GNN models on *proteins* and *collab*, and performs competitively to both GraphSAGE model variants on all datasets.

This allows us to answer **Q2**: It is easily possible to integrate EdgePool in existing architectures. Doing so will lead to an estimated improvement of about 2 pp, but might for some combinations of model and dataset decrease performance.

4.4.4 Q3: Can EdgePool be Used For Node Classification?

Pooling methods are well-suited for graph-level tasks, since the final result is far coarser than the initial graph anyway. For node-level tasks, every pooling has to be reverted. We are interested in the performance of our unpooling procedure (Section 4.3.4).

Experiment

On node classification tasks, we evaluate a simple architecture, varying the GNN layer type. We evaluate GCN, GIN and GIN0, and GAT layers. Again, we also evaluate an

Table 4.4: Integrating EdgePool into existing architectures for graph-level prediction: Accuracy (in percent) and 95% confidence bound of benchmark models with and without EdgePool. SAGE is short for GraphSAGE; nacc means without accumulating results.

proteins	GCN	GIN	GIN0	SAGE	SAGE nacc	MLP
No Pooling	71.4±3.1	70.4±1.7	70.9±2.3	71.7±2.2	73.0±3.0	71.8±2.6
EdgePool	73.1±2.9	72.9±2.2	71.7±2.2	73.5±2.2	69.9±3.0	73.1±2.9
RDT-B						
No Pooling	87.1±1.7	91.9±1.1	92.3±0.9	62.5±3.0	50.3±5.2	51.0±2.7
EdgePool	87.8±1.9	92.1±1.4	93.0±1.1	68.0±3.3	64.5±2.9	69.9±1.7
RDT-12K						
No Pooling	47.6±0.4	49.5±0.7	50.0±0.8	22.9±1.4	24.4±0.9	21.9±0.9
EdgePool	47.4±1.3	49.3±0.7	49.6±0.7	36.9±1.3	37.8±1.2	34.6±0.8
COLLAB						
No Pooling	67.0±1.4	74.2±1.1	74.1±1.0	63.6±1.5	64.1±1.3	52.0±1.6
EdgePool	71.5±1.2	73.0±1.3	72.2±1.0	64.3±1.2	64.1±1.4	67.8±3.2

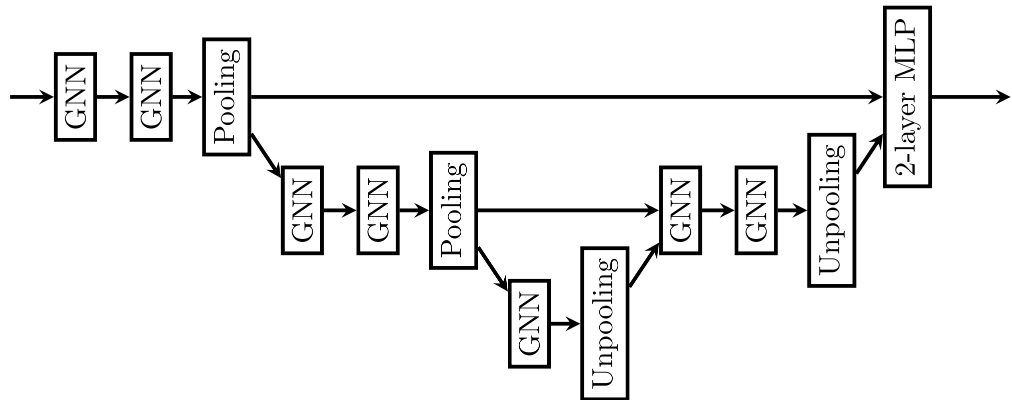


Fig. 4.6: Schematic of the architecture used to evaluate EdgePool impact on node-level prediction performance. Each pooling's features are both used for the next layer (at a decreased resolution) and concatenated with the corresponding unpooling layer's output.

MLP layer. As with Q2, we use seven convolutional layers. We pool after the second and fourth and unpool after the fifth and seventh layer, with shortcuts between the poolings. The concatenated features are then used by a two-layer MLP to predict each node's class. This is depicted in Fig. 4.6.

Results and Discussion

As Table 4.5 shows, GNN layers using EdgePool can be integrated in node classification architectures and improve performance for 21 of 25 dataset/model combinations. We make the following observations:

- Only on the *photo* and *computer* datasets using a GCN model does performance decrease significantly.
- Again, using EdgePool in combination with an MLP model improves performance greatly. On several datasets, this combination performs at least as good as the worst GNN model.
- For GNN algorithms, EdgePool improves performance by an average of 3.5 pp, performing worst on *pubmed* (no improvement on average) and for GCNs (decrease by 0.1 pp). It performs best for GIN and GIN0, at 5.8 pp and 6.6 pp improvements respectively.

This answers **Q3**: EdgePool will, in most cases, improve performance for node-level prediction. The expected improvement is an average of 3.5 pp.

Table 4.5: Using EdgePool for node-level prediction. Accuracy (in percent) and 95% confidence bound of benchmark models with and without EdgePool.

cora	GCN	GIN	GIN0	GAT	MLP
No Pooling	71.8±2.1	52.1±2.9	55.9±2.7	68.0±2.8	35.6±1.6
EdgePool	72.8±1.2	63.0±3.4	61.3±2.4	70.3±2.1	58.3±2.2
citeseer					
No Pooling	62.9±1.8	40.9±2.9	41.4±2.4	58.9±1.7	35.5±2.0
EdgePool	65.3±1.7	50.6±2.4	49.9±3.5	61.0±2.1	50.0±2.3
pubmed					
No Pooling	74.2±1.1	60.8±4.2	61.0±2.7	73.0±1.2	62.4±2.5
EdgePool	74.1±1.3	61.0±4.0	61.9±3.0	72.0±2.9	64.8±2.0
photo					
No Pooling	88.4±1.4	69.9±2.0	71.9±2.5	78.5±2.8	59.6±3.0
EdgePool	86.5±0.5	77.1±1.1	78.1±0.9	81.0±2.6	81.4±1.4
computer					
No Pooling	80.0±1.6	53.1±3.4	52.4±2.2	60.6±7.7	43.0±4.2
EdgePool	77.9±1.4	58.1±3.0	60.4±2.7	62.5±8.1	69.4±1.4

4.4.5 Q4: How Does EdgePool Impact Performance?

EdgePool introduces additional parameters and computation when added to a GNN model. We are interested in how memory usage and runtime are impacted by it.

Experiment

To evaluate memory usage, we follow the evaluation of Cangea et al. (2018): We construct Erdős-Rényi-Graphs with $|E| \approx 2\,|V|$. We use the same model as in Section 4.4.2, i.e. a three-layer GraphSAGE model with 128 random node features, and compute one forward and one backward pass. We evaluate this on a 1080 Ti GPU with 11GB memory.

Results and Discussion

As Fig. 4.7 shows, both the sparse base model and EdgePool scale linearly in the number of nodes while DiffPool scales quadratically. Accordingly, DiffPool cannot process more than about 18 k nodes. The sparse base model's 250 k node maximum is increased by EdgePool to 300 k nodes since it offsets its increase in parameters with drastically reduced graph sizes after several poolings.

At the same time, runtime is consistently worse than the non-pooled sparse baseline, with being about 50% slower, since EdgePool adds a whole layer of additional computa-

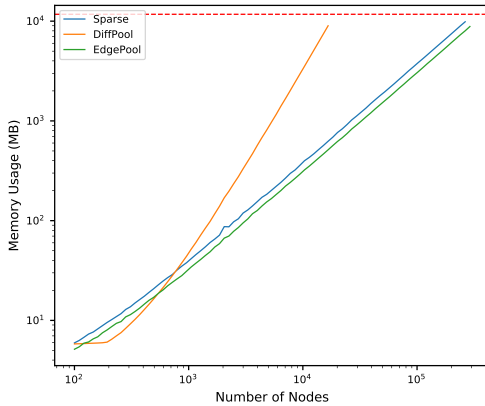


Fig. 4.7: Memory requirements for different pooling algorithms. Note the log-log axis. The dashed red line depicts available GPU memory (11GB).

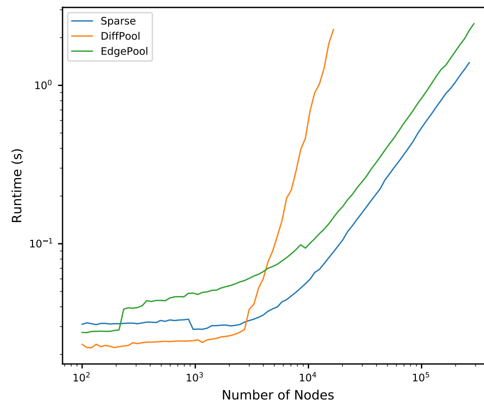


Fig. 4.8: Computational runtime for one forward and one backward pass for different pooling algorithms. Note the log-log axis. Runtimes are plotted until reaching the memory limit.

tion that needs to be executed. However, it still runs consistently faster than DiffPool beginning at graph sizes upwards of 4 000 nodes.

4.5 Conclusion

We have proposed EdgePool, a local and hard pooling method for Graph Neural Networks, based on edge contraction. This pooling is both localized (and therefore independent of non-local graph changes) and sparse (and therefore computationally efficient even on large graphs).

EdgePool outperforms all other sparse pooling methods. Except for a single dataset, it also outperforms DiffPool, which due to its construction cannot be integrated easily in standard GNN models and is computationally inefficient.

We also showed that EdgePool can be integrated into a large number of GNN architectures and usually improves performance on graph classification tasks without any adaptations to training or architecture. We also proposed an unpooling method which allows it to be applied to node-level prediction tasks. Here, too, it usually improves performance.

We have evaluated the computational performance of EdgePool: While it introduces a 50% increase in runtime, it significantly decreases memory usage compared to a standard, sparse, GNN model, increasing possible graph size by 20%. Compared to DiffPool, it allows processing of graphs with 16× as many nodes and achieves the same runtime on graphs of 300 k nodes as DiffPool does on graphs with 14 k nodes.

Besides the obvious use of EdgePool in improving existing GNN architectures, we hope it will serve as a stepping stone towards methods that learn how to modify graph

structures. We believe this will lead towards methods that no longer operate on nodes but on abstracted groups of nodes, which would be another step away from the simple graphs dominating GNN research.

5 Edge Features

Whereas previous chapters showed the performance of GNN models on simple graphs, i.e. those with only graph topology and node features, many real-world problems feature more complex structures. This chapter introduces the problem of traffic prediction as such a task and shows that GNN models can be used to naturally model this situation[1].

In this chapter, we first introduce the task and its importance (Section 5.1). Afterwards, we introduce the techniques used to build a graph from a traffic scene and the adaptations to our GNN models necessary for this task (Section 5.2). We then introduce the setup used to evaluate these models (Section 5.3), particularly the two datasets, both learned-model baselines and fixed baselines, and the experimental procedures. This is followed by Section 5.4, which introduces both our research questions and the observations we made in our experiments. Since GNN models are difficult to interpret, we then introduce an adaptation of saliency maps to graphs, which allow us to inspect the traffic situation and find important vehicles that influence our ego vehicle (Section 5.5).

5.1 Motivation

Accurate short-term behaviour prediction of traffic participants is important for applications such as automated driving or infrastructure-assisted human driving (Krämmer et al. 2019; Hinz, Buechel, et al. 2017). A major open research question is how to model *interaction* between traffic participants. In the past, interactions have been modelled by either creating a representation of one or several traffic participants (Treiber 2000; Lenz et al. 2017a) or by using a fixed environment representation such as a simulated LIDAR sensor (Kuefler et al. 2017).

However, these methods impose certain disadvantages: A fixed environment representation poses a much harder problem to learn, since we cannot use data we might

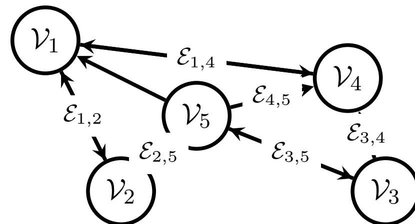


Fig. 5.1: The graph type used in this chapter: It expands upon the simple graphs from Chapter 3 by introducing edge attributes.

[1]Parts of this chapter have been previously published (Diehl, Brunner, et al. 2019a).

have extracted previously. Traffic participant representations, on the other hand, scale computationally with the amount of possible interactions, require a human to decide on a useful representation, and underspecify the problem one should learn.

By modelling each vehicle as a node and possible interactions between vehicles as edges (see Fig. 5.2 for a visualization), we gain a sparse and high-level representation of a traffic scene as a graph. At the same time, it has been shown (Morton et al. 2016; Kuefler et al. 2017; Lenz et al. 2017a) that machine learning models and particularly (deep) neural networks perform well on this problem.

Marrying the representation of a traffic situation as a graph with the modelling capabilities of GNN models promises a clear method to take interactions between traffic participants into account, good predictive performance, and efficient computation.

To evaluate this, we conduct traffic participant prediction on two real-world datasets, evaluating their predictive performance and comparing them to three baseline models. We show that prediction error decreases by 30% compared to our baseline when interaction is plentiful and performs no worse when little interaction occurs. At the same time, computational complexity remains reasonable and scales linearly in the number of interactions.

This suggests a graph interpretation of interacting traffic participants is a worthwhile addition to traffic prediction systems.

Our main contributions are:

- We show that representing interactions as graphs leads to better performance.
- We introduce several adaptations into the GNN models to take edge features into account. We show that using edge features is necessary for a good performance.
- We study both the results of different graph construction techniques and our introduced adaptations on two different datasets.

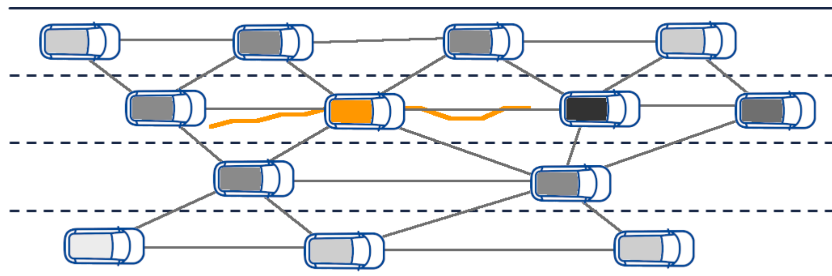


Fig. 5.2: Interaction graph of a traffic situation. Interactions are assumed to occur between the ego vehicle (orange) and its up to eight neighbours, assigned by current lane (dashed lines are lane dividers). These interactions are modelled as edges between the each vehicle and its neighbouring vehicles. Vehicles are coloured according to their influence on the ego vehicle. As can be seen, this representation is sparse and models the whole traffic situation at once.

5.2 Traffic Participant Prediction from a Graph

We concentrate on two different graph models: GCN and GAT. As we have shown in Section 3.3.3, GCN models generally perform very well while remaining a simple model. GAT models, on the other hand, allows us to easily incorporate edge information into the model.

5.2.1 Adapting Graph Convolutional Networks

We originally applied the unmodified GCN layer as described in Section 2.5.3.1. However, we found two changes to be crucial:

Weighting by Distance Kipf and Welling (2016) note that the adjacency matrix can be binary or weighted. We evaluate weighting edges by the inverse distance, with self-loops set to 1. This changes the message function from Eq. (2.39) to

$$M\left(v_i, v_j, e_{ij}\right) = \frac{1}{d}\left(\deg\left(v_i\right)\deg\left(v_j\right)\right)^{-1/2}\mathbf{A}_{ij}\mathcal{V}_j, \tag{5.1}$$

where d is the distance between the two nodes

Residual Weights The default GCN formula treats both ego node and neighbours identically, aggregating all of them. However, this means a GCN model cannot treat the ego node's own features differently from any of its neighbours. In the prediction task, this appears to be a significant obstacle to good performance. Accordingly, we change the node update function of Eq. (2.40) to

$$\mathcal{U}_v\left(v_i\right) = \mathbf{\Theta}_r\mathcal{V}_i + \mathbf{\Theta}_n \sum_{v_j \in \mathcal{N}(v_i)} m_{ij}, \tag{5.2}$$

separating the layer's weights into a neighbour weight matrix $\mathbf{\Theta}_n$ and a residual connection weight matrix $\mathbf{\Theta}_r$.

5.2.2 Adapting Graph Attention Networks

As with GCN layers, GAT layers require two adaptations to perform well on this task.

Edge Attributes The original GAT formulation depends only on the features of the two nodes. However, the relative positions of two nodes are additional data available to us in this scenario. Accordingly, we augment the attention computation from Eq. (2.45) by including edge features, such that

$$\alpha_{ij} = \mathrm{softmax}_{\mathcal{N}(v_i)}\left(\sigma\left(\mathbf{\Theta}\left(\mathcal{V}_i \| \mathcal{V}_j \| \mathcal{E}_{ij}\right)\right)\right). \tag{5.3}$$

i.e. we modify the attention mechanism to work on the concatenated node features and the edge features. We do not transform the edge features themselves, and they remain constant over all layers.

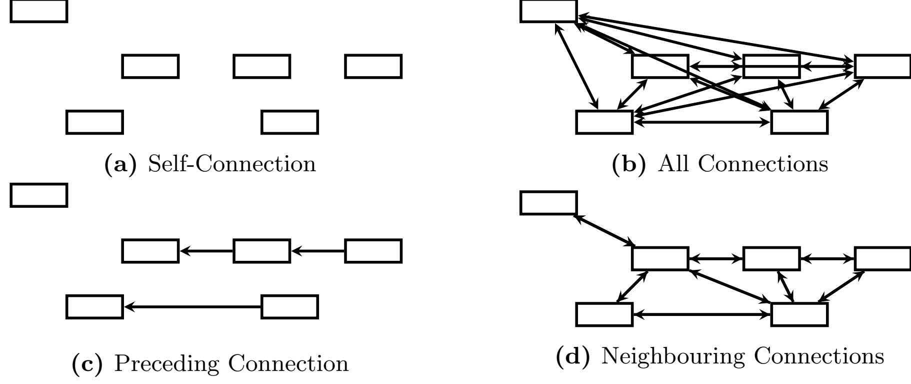


Fig. 5.3: Schematics of the connection strategies on the same scene. Cars are moving to the right.

Residual Weights As with GCN layers, we introduce a residual weight, treating the ego node differently from its neighbours. This changes Eq. (2.44) to

$$\mathcal{U}_v(v_i) = \mathbf{\Theta}_r \mathcal{V}_i + \mathbf{\Theta}_n \sum_{v_j \in \mathcal{N}(v_i)} \alpha_{ij} m_{ij} \tag{5.4}$$

also separating the layer's weights into a neighbour weight matrix $\mathbf{\Theta}_n$ and a residual connection weight matrix $\mathbf{\Theta}_r$.

5.2.3 Graph and Feature Construction

Formulating the prediction problem as a graph still leaves open the task of how we construct said graph and the node features. While there is an obvious strategy to construct node features—namely to use the corresponding car features like position or velocity—the same does not apply to constructing a graph topology. Here, no single strategy is immediately superior to others. However, four basic strategies are possible, and are depicted in Fig. 5.3:

Self-Connection This only adds self-loops to the graph. It ignores all interaction performance and should perform identically to a simple MLP model operating on the ego vehicle data only. It is depicted in Fig. 5.3a.

All Connections Connecting all vehicles ensures that no interactions are ignored. However, this ignores previous knowledge on spatial position and interaction and scales quadratically in the number of vehicles. It is depicted in Fig. 5.3b.

Preceding Connection Arguably the most important interaction is with the vehicle immediately in front of us. We can therefore construct interactions only between the current vehicle and its predecessor. It is depicted in Fig. 5.3c.

Neighbouring Connections We argue that the main interactions are with the vehicles in an ego vehicle's direct environment, which are at most eight vehicles located to the front, rear, and sides of the ego vehicle. This construction is similar to the approach by Lenz et al. (2017a) and Morton et al. (2016). It is depicted in Fig. 5.3d.

While we would prefer to learn these connecting strategies, this is a very difficult open problem and scales quadratically with the number of considered vehicles. We therefore only evaluate the fixed strategies.

5.3 Experiments

In order to evaluate the newly proposed models, we conduct a prediction experiment on real-world traffic data. We purposely keep baselines and models simple to demonstrate whether the graph interpretation is beneficial without introducing a multitude of confounding factors. We therefore do not include RNN architectures, simulation steps, or imitation or reinforcement learning.

5.3.1 Datasets

We conduct our experiment on two different datasets: The *NGSIM* I-80 dataset (Federal Highway Administration (FHWA) 2005) and the *HighD* dataset (Krajewski et al. 2018).

NGSIM The NGSIM project's I-80 dataset contains trajectory data for vehicles in a highway merge scenario for three 15-minute timespans. These are tracked using a fixed camera system. As Thiemann et al. (2008) show, position, velocity, and acceleration data contain unrealistic values. We therefore smooth the positions using double-sided exponential smoothing with a span of 0.5 s and compute velocities from these.

We use two of the recordings as training set and split the last one equally into validation and test set. We subsample the trajectory data to 1 frame per second (FPS) and extract trajectories consisting of a total of ten seconds of length. The goal of the model is to predict the second half of the trajectory given the first five seconds.

HighD Since the *NGSIM* dataset still contains many artifacts (errors in bounding boxes, undetected cars, complete non-overlap of bounding box and true vehicle), we additionally conduct experiments on the new *HighD* dataset (Krajewski et al. 2018), which is a series of drone recordings and extracted vehicle features from about 400 meters each from several locations on the German Autobahn. A total of 16.5 hours of data is available, containing 110 000 vehicles with a total driving distance of 45 000 km. However, since the dataset consists mainly of roads without on- or off-ramps and without traffic jams, interaction seems limited: Only about 5% of the cars experience a lane change.

To avoid information leakage, we split the dataset by recording. The last 10 % of the recordings are used as test set, the 10 % before that as validation set. Trajectory construction is then identical to the *NGSIM* dataset.

5.3.2 Baselines

We compare our approach to two different model-based static approaches, and one learned approach.

Constant Velocity Model The Constant Velocity Model (CVM) considers each car to continue moving at the same velocity (both laterally and longitudinally) as in the last frame it was observed in. It is a simple model, which nonetheless has been shown to perform surprisingly well (Schöller et al. 2020).

Intelligent Driver Model The Intelligent Driver Model (IDM) (Treiber 2000) is a commonly-used driver model for microscopic traffic simulation since it is interpretable and collision-free. We use this to predict the changes in longitudinal velocity and keep the lateral position constant.

The IDM's acceleration is computed from both a *free road* and an *interaction* term. The free road acceleration is computed as

$$a_{free} = a_{max}\left(1 - \left(\frac{v}{v_0}\right)^{\delta}\right),$$

using the current velocity v and three tunable parameters: Maximum acceleration a_{max}, the acceleration exponent δ, and the desired velocity v_0. The interaction term is defined as

$$a_{int} = -a_{max}\left(\frac{s_0 + v * \tau}{s} + \frac{v\Delta_v}{2s\sqrt{a_{max}b}}\right),$$

where the minimum distance to the front vehicle s_0, the time gap τ, and the maximum deceleration b are tunable parameters. v is again the vehicle's speed and Δ_v the closing speed to its predecessor. The total acceleration is the sum of both the free road and the interaction acceleration. Since the IDM only outputs a longitudinal acceleration, we assume no lateral motion when using the IDM

We take the IDM parameters for the *NGSIM* dataset from Morton et al. (2016). For the *HighD* dataset, we tune the IDM's parameters using guided random search with a total of 20 000 samples. Both values are listed in Table 5.1.

Feed-Forward Model In addition to the models taking interaction into account, we also add a simple feed-forward MLP, predicting the trajectory from only the ego vehicle's past data. We use this baseline model to measure the improvement we gain from including interaction into our models.

5.3.3 Model Configuration

Each model uses a similar configuration: Two GNN layers producing a 256-dimensional feature representation, followed by a feed-forward layer which produces the final output, the displacement in x and y direction. All models use the ReLU nonlinearity. The GAT

Table 5.1: Optimized parameters of the IDM. IDM parameters for *NGSIM* are from Morton et al. (2016); for *HighD* they are from guided random search.

Parameter			*HighD*	*NGSIM*
Desired velocity	v_0	$\left[\frac{m}{s}\right]$	58.87	17.8
Maximum acceleration	a_{max}	$\left[\frac{m}{s^2}\right]$	0.14	0.76
Time gap	τ	$[s]$	0.12	0.92
Comfortable deceleration	b	$\left[\frac{m}{s^2}\right]$	12.17	3.81
Minimum distance	s	$[m]$	14.46	5.249

employs four attention heads (and therefore 64-dimensional feature representations per head).

Since the GNN models use two layers, their effective receptive field is the two-hop neighbourhood from the ego vehicle.

All models receive inputs and produce outputs in fixed-length timesteps without recurrence. They are trained to predict displacement relative to the last position and receive position and velocity for each past timestep. They train to minimize the mean squared error over all outputs. All models are implemented in pytorch (Paszke et al. 2017) using and expanding upon the *pytorch-geometric* library (Fey and Lenssen 2019).

5.3.4 Performance Measure

We report performances of the model by measuring the error in position between ground truth and prediction. We both report mean displacement error over five seconds, weighting each timestep identically, and final displacement error after five seconds.

5.3.5 Experimental Procedure

Our choice of experiments is guided by the three main questions introduced in Section 5.4, which we answer in Sections 5.4.1 to 5.4.3. To ensure meaningful results, we repeat each evaluation a total of ten times using different random seeds. In tables, we report all results as mean ± 95% confidence bounds. Figures are violin plots, showing both individual results and the total result distribution.

We optimize both network adaptations and graph construction strategies on the *NGSIM* I-80 dataset since it is both smaller and contains more interactions. We then use these insights to pick the best-performing models and evaluate them on both the *NGSIM* I-80 and the *HighD* dataset.

5.4 Results and Discussion

We structure our evaluation according to three research questions:

Q1: Which of our adaptations to GNNs are necessary?

Table 5.2: Results for our GNN ablation study on the NGSIM I-80 dataset. We evaluate our adaptations for the GCN and GAT models and our connection strategies. The latter are evaluated using the default GAT model.
($\star$) Uses 3 instead of 10 evaluations.

		Mean Displ. [m]	Displ. @5s [m]
GCN	Default	2.50±0.04	4.68±0.09
	no MLP output	2.52±0.04	4.66±0.07
	with weighted edges	2.60±0.05	4.91±0.09
	no residuals & with weighted edges	2.87±0.02	5.19±0.05
	no residuals	3.74±0.07	6.42±0.06
GAT	Default	1.92±0.02	3.45±0.05
	no MLP output	1.93±0.04	3.38±0.10
	no residuals	2.32±0.01	3.96±0.02
	no edge features	2.40±0.03	4.48±0.06
Connections	Self-Connection	2.68±0.03	5.08±0.05
	Preceding Connection	2.70±0.02	5.11±0.04
	Neighbouring Connections	1.93±0.05	3.47±0.08
	All Connections ($\star$)	2.41±0.02	4.42±0.03

Q2: How do we construct an interaction graph?

Q3: Does a graph model increase prediction quality?

5.4.1 Q1: Which of Our Adaptations to GNNs Are Necessary?

In Section 5.2.3, we proposed several changes to the GCN and GAT architectures. To answer which of these changes are beneficial, we conduct an ablation study. From the results in Table 5.2, we can make several observations:

- Removing the residual connections increases prediction error by at least 20%. This is likely because there is a clear difference between a neighbouring and the ego node in this task.
- Using an MLP to produce the final output does not increase prediction performance significantly. However, it does seem to stabilize training.
- Introducing relative positions as edge features into the GAT model is a clear success, reducing the final displacement by about a meter. This is the most significant improvement of any single component.
- Contrary to that, edge weights for the GCN model slightly decrease performance, especially when omitting residual weights. We believe that the main contribution

of edge weights for the GCN model in our scenario is to discern between the ego and surrounding vehicles, which is already more effectively modelled through residual weights.

We therefore evaluate the graph construction using our baseline GAT model with MLP output, residuals, and edge features.

5.4.2 Q2: How do We Construct an Interaction Graph?

In Section 5.2.3, we proposed four construction strategies for the interaction graph. We evaluate the quality of predictions with each of these strategies using the GAT model. We note that in practical scenarios, a tradeoff might be necessary between prediction quality and computational complexity. Table 5.2 shows results, from which we can make several observations:

- As expected, the Self-Connection strategy performs identically to the MLP baseline model. Somewhat surprisingly, the Preceding Connection strategy performs no better.
- The All Connections strategy improves prediction quality over both of these. However, it imposes significant computational disadvantages with quadratic instead of linear runtime and, in our experiments, a slowdown of about 50×.
- The Neighbouring Connections graph construction method clearly performs best, improving prediction by almost a metre after five seconds compared to the second-best connection strategy and by 1.5 m compared a model without interactions. We argue this is due to the structural bias imposed by the model, which strongly observes the neighbouring vehicles only.

We therefore use the Neighbouring Connections graph construction strategy for our evaluation.

5.4.3 Q3: Does a Graph Model Increase Prediction Quality?

The motivation of our work is to evaluate whether it is beneficial to model interaction between traffic participants and whether this can be modelled in a graph construction. To answer this question, we compare GNN models to a model without interactions (FF). We also include a comparison with two classical models (CVM and IDM).

We choose the GAT model as best-performing GNN. We also include a GAT model without edge features (called GAT NEF in our figures and tables) for a fair comparison with the GCN model.

NGSIM Fig. 5.4 and Table 5.3 show the performance of both our non-learning models, the MLP model, and three GNN models. We can make several observations:

Table 5.3: Performance comparison on the NGSIM dataset.

	Displ. $[m]$	Displ. @5s $[m]$
GAT	1.89±0.01	3.40±0.02
GAT NEF	2.39±0.02	4.46±0.04
GCN	2.51±0.04	4.69±0.07
FF	2.78±0.05	5.22±0.09
CVM	2.58±0.01	5.00±0.01
IDM	3.10±0.01	6.60±0.01

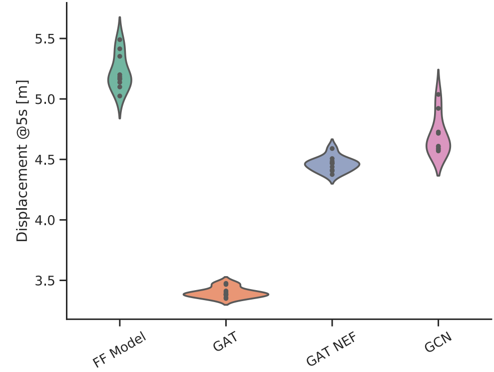


Fig. 5.4: Performance on the NGSIM Dataset. As can be seen, all three GNN models perform better than the baseline which does not take interaction into account.

- The non-learned baselines perform significantly worse on longer time-scales than any learned model.
- The CVM outperforming IDM on shorter timescales is consistent with previous work (Lenz et al. 2017a) and it is likely to achieve better performances in a closed- or open-loop simulation.
- Every GNN model outperforms the baseline, reducing mean error by 0.5–1.8 m.
- Our GAT adaptations, particularly the inclusion of edge features, increases performance significantly compared to other GNN models. This supports our belief that edge features are crucial for this task.
- The adapted GAT model reduces prediction error by 30% compared to the FF baseline.

We can answer **Q3**: On the NGSIM dataset, the adapted GAT model significantly increases prediction quality.

HighD Fig. 5.5 and Table 5.4 show the performances on the HighD dataset. We can make several observations:

- The non-learned baselines still perform significantly worse than any learned model. However, there is no significant difference between IDM and CVM performance.
- Measured by mean displacement over all timesteps, all learned models show similar performance. This is independent of their use of interactions.

Table 5.4: Performance comparison on the HighD dataset.

	Displ. $[m]$	Displ. @5s $[m]$
GAT	$0.47_{\pm 0.02}$	$1.04_{\pm 0.04}$
GAT NEF	$0.49_{\pm 0.04}$	$1.06_{\pm 0.05}$
GCN	$0.47_{\pm 0.05}$	$1.10_{\pm 0.09}$
FF	$0.45_{\pm 0.04}$	$1.09_{\pm 0.06}$
CVM	$1.09_{\pm 0.01}$	$2.66_{\pm 0.01}$
IDM	$1.12_{\pm 0.01}$	$2.66_{\pm 0.01}$

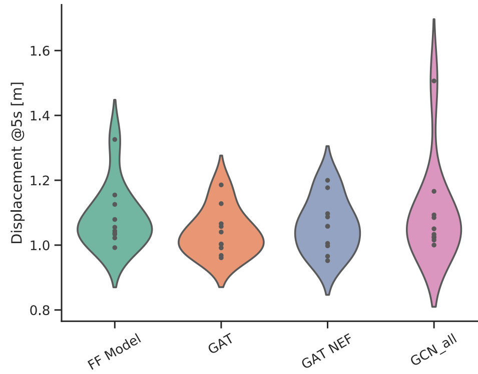


Fig. 5.5: Performance on the HighD Dataset. As can be seen, the algorithms perform similar. We believe this to be due to little interaction occurring in the dataset.

- On long-term prediction, the GAT and GAT NEF models slightly outperform the GCN model and the FF baseline.

Answering **Q3** on the HighD dataset, there is no large performance increase for using interaction models. We believe this to be a consequence of little interaction between the cars, which makes all learned models degenerate to the non-interacting case and makes the interaction term of the IDM model irrelevant. However, the results show including interaction representations into our model does slightly increase performance even when there is little interaction in the dataset.

5.4.4 Conclusion

In summary, we show that (**Q1**) several of our changes result in better performance, (**Q2**) as does a good interaction graph construction strategy. (**Q3**) In total, our model retains performance on a dataset with little interaction and greatly improves it on a dataset with plentiful interaction.

5.5 Inspecting GNNs for Traffic Prediction

GNN models share a disadvantage with other neural-network based methods: They are difficult to interpret. This places engineers in a bind: They can either accept the lower performance of interpretable models or sacrifice interpretability for performance.

Instead, we present a technique by which representing a traffic scene as a graph improves interpretability. To do so, we adapt the concept of saliency maps (Simonyan et al. 2013), the primary technique used to visualize relevant image regions for CNNs,

to graphs. Fig. 5.2 shows how these saliency graphs allow engineers to visually inspect and interpret the model outputs.

While our presented task is supervised vehicle behavior prediction, it can be applied to any task using a graph representation. Particularly in reinforcement learning applications, this technique can be used to find reasons for the ego vehicle's motion. This could then be displayed for the passengers to justify the ego vehicle's behavior.

Aside from such real-time justification to users, engineers can use this interpretability to guide model architecture choices. Our inspection of the models from Section 5.2 shows that the naive GNN models do not prioritize relevant vehicles, but our specific adaptations do, particularly when including edge features. This knowledge allows engineers to justify their model choices on better grounds than just a performance difference and to better decide which algorithm to deploy to an autonomous car.

5.5.1 Creating Saliency Graphs

We create saliency graphs by adapting the technique of saliency maps to the graph context.

5.5.1.1 Saliency Maps

Saliency maps, first introduced in the context of weakly-supervised object detection by Simonyan et al. (2013), are a technique to find the features responsible for a classification decision.

To do so, they first compute the derivative of the classification score $S_c(I)$ with respect to an image I for a specific image I_0

$$w = \left. \frac{\delta S_c}{\delta I} \right|_{I_0} . \tag{5.5}$$

When operating on grayscale images, the final saliency map is constructed by rearranging the values from w such that it is positioned where the corresponding pixel was. For colour images, this is preceded by extracting a single value for each pixel through taking the maximum.

There has been much work in producing better-looking saliency maps for images. The saliency construction detailed above produces very noisy images, which makes interpretation difficult. A large number of methods (for example Smilkov et al. (2017), Sundararajan et al. (2017), and Selvaraju et al. (2016)) have been proposed to ensure good-looking results, and further work, for example by Olah et al. (2018), include and combine these with other techniques.

However, Adebayo et al. (2018) have called several of the saliency construction methods into question. They showed that some methods' saliency maps look similar for both a trained and untrained model. The standard gradient model explained above is resistant to this, but the saliency maps it produces look noisy. Because nodes in a grapk have a much lower density than pixels in an image and each node is a meaningful unit,

this does not pose a problem for saliency graphs. Hence, we can use the simple gradient construction method for saliency graphs.

5.5.1.2 Computing Gradients

First, we compute the derivative of the prediction output p with respect to a graph input G for our specific graph G_0:

$$w = \left.\frac{\delta p}{\delta G}\right|_{G_0}. \tag{5.6}$$

This is equivalent to Eq. (5.5) but for graphs. Note that we are usually interested only in the influence on a single vehicle, which we call the ego vehicle. We therefore compute the gradient w only with respect to the prediction of the ego vehicle's motion p_e, which is a subset of the graph prediction:

$$w_e = \left.\frac{\delta p_e}{\delta G}\right|_{G_0}. \tag{5.7}$$

5.5.1.3 Summarizing Feature-Wise Gradient

Given the procedure from last section, we end up with one gradient value per feature per vehicle. The problem of summarizing them is similar to the one faced by Simonyan et al. (2013) when operating on coloured images. We evaluated both taking the mean of each vehicle's features and their solution of taking the maximum. We found that they resulted in somewhat similar feature maps and therefore chose taking the maximum value.

That is, the influence of the ith vehicle on the ego vehicle is then

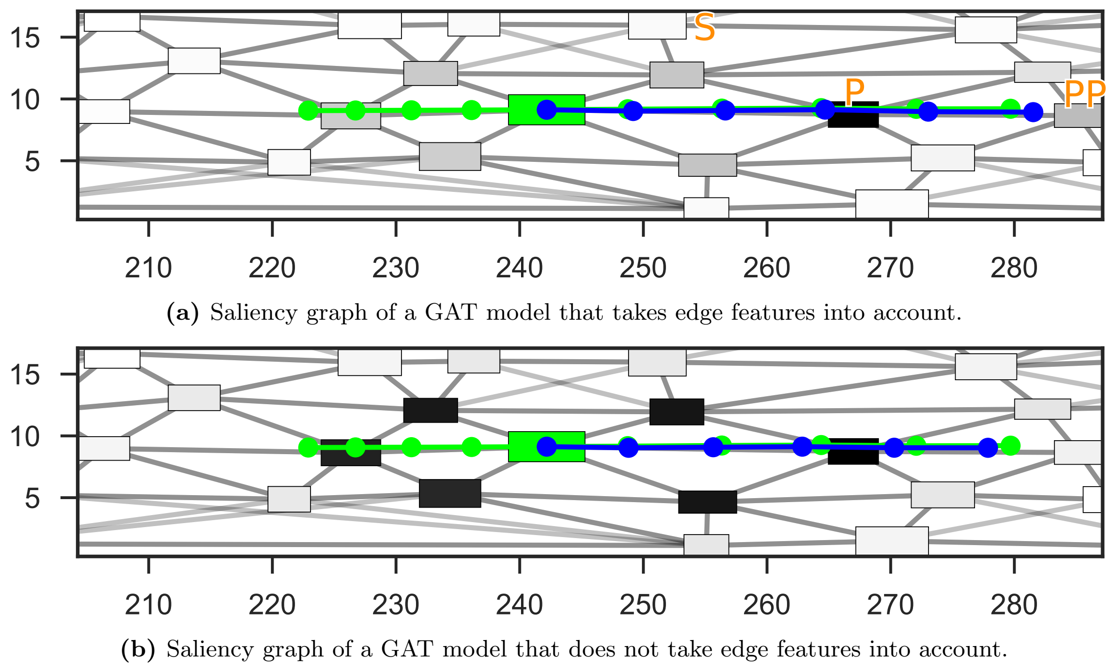
$$w_i = \max_{f_i^k \in f_i} \left.\frac{\delta p_e}{\delta f_i^k}\right|_{G_0}, \tag{5.8}$$

where f_i^k is the kth feature of the ith vehicle.

5.5.1.4 Plotting the Saliency Graph

This procedure gives us saliency values for each node (vehicle) of the graph. Inspecting these manually immediately reveals that the main influence stems from the ego vehicle's features themselves. While this makes sense (see Section 5.5.2.1), it complicates interpretation of the neighbours' influences. We therefore do not plot ego vehicle influence in our saliency graphs. We also normalize node importance for each graph, meaning that all maximum values have the same colour in each of our saliency graphs. While this means we cannot compare two different saliency graphs, it gives us the maximum range of values to plot and therefore makes each saliency graph more expressive.

Note that this procedure also gives us influences of the edge features if they exist. Experimentally, however, these are one to two orders of magnitude smaller than the node features and we therefore do not plot them.



(a) Saliency graph of a GAT model that takes edge features into account.

(b) Saliency graph of a GAT model that does not take edge features into account.

Fig. 5.6: Comparison of the saliency maps of GAT models with and without edge features. Cars are depicted as boxes, with their connections shown. The ego vehicle and its real trajectory is shown in green, with the predicted trajectory in blue. Other cars are shaded by influence. As can be clearly seen, the GAT model with edge features primarily takes the direct predecessor (P) into account and - to a lesser extent - the cars on its side and the car preceding its predecessor (PP). Using the GAT model without edge features, however, all cars in the one-hop neighbourhood share the same influence.

5.5.2 Results and Discussion

Using the ability to construct saliency graphs, we now analyze influences on the prediction of a ego vehicle. For this, we first compare the influence of the ego vehicle's features to the influence of the neighbouring vehicles. Afterwards, we evaluate whether the inclusion of relative positions into a model improves which vehicles are seen as relevant. Lastly, we take the created saliency graph and interpret model performance using it.

5.5.2.1 Influence of the Ego Vehicle

As noted in Section 5.5.1.4, the influence of the ego vehicle usually dominates. We experimentally find it varies between $1-4\times$ the influence of the most influential neighbouring car.

Intuitively, this makes sense: The main feature responsible for predicting the future displacement are the ego vehicle's current features. The features of other vehicles merely act as an influence on this, changing the final motion but not completely controlling it.

Since we are primarily interested in the influence of neighbouring vehicles, we do not plot the saliency of the ego vehicle.

5.5.2.2 The Effect of Edge Features

As noted in Section 5.4.1, including the relative position as edge features in the graph led to better results. With the saliency graph, we can now inspect the actual predictions and influences thereon in far greater detail than a single numeric performance metric.

One example is shown in Fig. 5.6. As can be seen, the GAT model without edge features (Fig. 5.6b) is influenced equally by every car in its one-hop neighbourhood. This makes sense, since the default GAT model can only distinguish between neighbours and non-neighbours. No distinction is possible by position or similar measures. It therefore can only compute its result as some aggregation of the set of nodes from the one-hop and two-hop neighbourhood. While not immediately comparable, this meshes with theoretical analysis by Xu, Hu, et al. (2018), who show that many classes of GNN cannot distinguish isomorphic graphs, particularly multisets.

In contrast, the GAT model with edge features (Fig. 5.6a) clearly distinguishes between cars based on their position: The direct predecessor has the greatest influence on the ego vehicle's predicted performance. The inclusion of these edges has the advantages of both including physical information in a useful format and of the theoretical ability to distinguish between multisets of identical nodes.

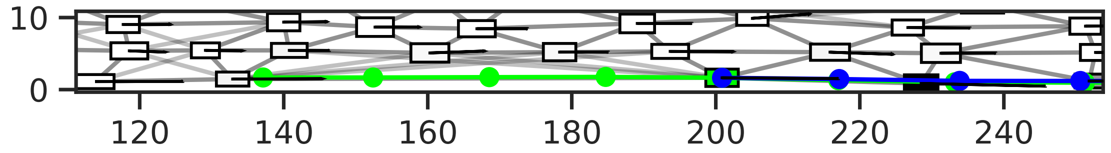
5.5.2.3 Analyzing the Influence of Neighbours

Using the constructed saliency graph, we can now analyze the influence of vehicles on the ego vehicle. Inspecting Fig. 5.6a, we see that the car directly in front (marked P) has the highest influence, followed by the neighbouring and following cars. However, the car immediately preceding the preceding car (marked PP) has a similar influence. This is despite there being no direct edge between them and the ego vehicle, clearly showing that the propagation of node features.

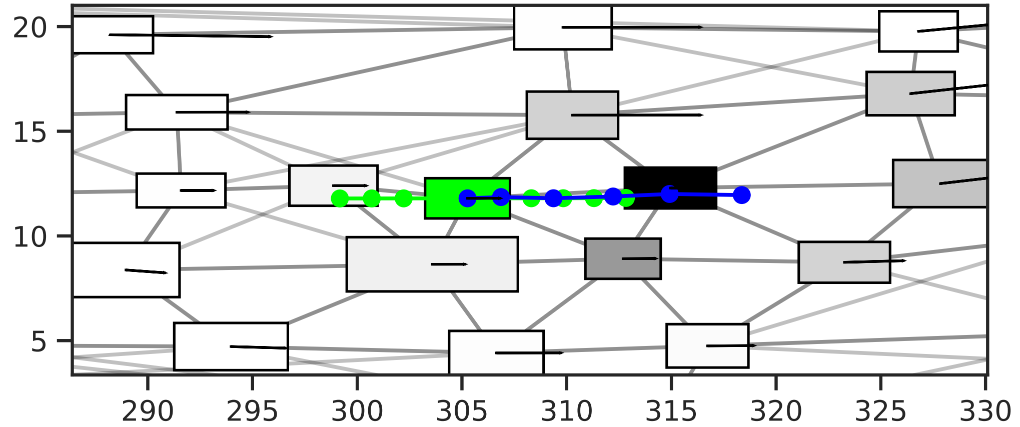
At the same time, this shows that the prediction takes similar priorities as humans do: The mainly relevant cars are directly in front, and cars preceding them have a slightly lower influence. Observe, for example, the difference in magnitude between the car twice preceding the ego vehicle (PP) and the one two lanes up from the ego vehicle (S): Despite both being two edges distant from the ego vehicle, their importance differs greatly.

Compare this with Fig. 5.6b: Here, all direct neighbours of the ego vehicle share the same influence, as do all two-hop neighbours. On visual inspection, this allows us to immediately disqualify the model, since we expect the vehicles in front to have a higher influence on the prediction.

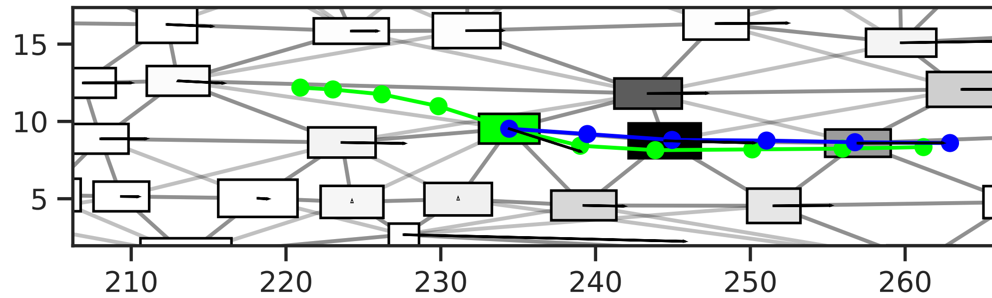
In summary, the modified GAT model primarily looks "forward" of the ego vehicle and even takes cars driving in front of these into account. This only occurs when using edge features, suggesting that the modification using relative positions as edge weights not only improves performance but also lets the model prioritize between cars similarly to humans.



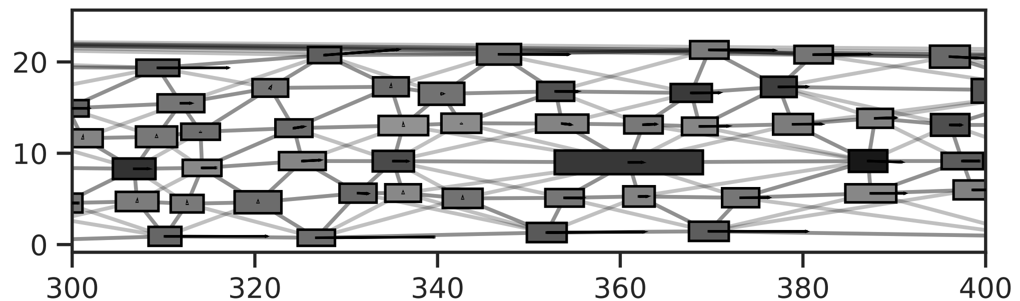
(a) Saliency graph of a free-flow scenario. Best viewed zoomed-in. Note the velocity vector of the plotted vehicles (black arrows): This is a fast traffic situation.



(b) Saliency graph of a dense-traffic scenario. Note again the velocity vector, which shows that the vehicles are very slow.



(c) Saliency graph of a lane change.



(d) Saliency graph for a full traffic scene.

Fig. 5.7: Saliency graphs for three different traffic scenarios (a-c) and for a full traffic scene (d). These also plot velocity vectors (black arrows) for each car to show differences in speed in the different scenarios. These show the displacement after one second.

5.5.2.4 Saliencies for Specific Scenarios

Several saliency graphs for specific traffic scenarios are depicted in Fig. 5.7. These are a lane change, a free-flow scenario, and a dense traffic scenario.

Free-Flow Fig. 5.7a shows a traffic scenario with free traffic flow. The ego vehicle is moving with a velocity of ca. 16 m/s (60 km/h), as do the surrounding vehicles. As can be seen, despite all the plotted vehicles having a small influence on the prediction of the ego vehicle, the two main vehicles to look for are the vehicle immediately preceding the ego vehicle, and the vehicle preceding that.

This makes sense: When moving at such speeds in a free-flow situation, the mainly relevant vehicles are the vehicle in front, which must be monitored for braking. Since this is mainly influenced by the vehicle to its front, its relevance also makes sense.

Contrary to this, the neighbouring vehicles would only matter if they change lanes, which none of these do.

Dense Traffic Fig. 5.7b shows a dense and very slow traffic scenario. The ego vehicle is moving with a velocity of ca. 1.5 m/s (5 km/h), as do the surrounding vehicles. Note how, contrary to Fig. 5.7a, surrounding vehicles now have a significantly higher influence on the ego vehicle's prediction.

This makes sense: At slow speeds, lane changes are more likely to happen, and any surrounding car might either allow a lane change or change lanes themselves. Note also that the predicted trajectory is of a fairly low quality, overestimating the ego vehicle's progress by 7 m after five seconds. This is more difficult to predict than a free-flow scenario, and will require the addition of an open-loop simulation (Lenz et al. 2017a).

Lane Change Fig. 5.7c shows the ego vehicle changing lanes. As can be seen, the greatest importance is assigned to the immediately preceding car. However, the preceding car of the previous lane still has a large influence on the prediction. This is despite the model not having any explicit representation of lanes; it only reacts to relative positions and velocities of vehicles.

5.5.2.5 Interpreting a Complete Traffic Scene

Previously, we only restricted ourselves to the influence vehicles have on the single ego vehicle. We can also use this technique to interpret a whole traffic scene by considering the influence the participating vehicles have on each other.

One such scene is depicted in Fig. 5.7d. Of particular note is the truck (long box) depicted in the centre of the image. It shares the greatest influence on the scene with the vehicle directly in front of it. In comparison, the vehicles on the lowest lane—which move at a significantly higher speed than the dense traffic in the middle lanes—have little influence on the whole scene.

5.5.3 Conclusion

We have shown how saliency graphs can be used to interpret the behaviour influences on the ego vehicle. We have used this to show how our adaptations to the GAT model allow the model to concentrate on more relevant vehicles. We have also shown how to use this technique to find the most influential vehicle on a traffic scene as a whole.

5.6 Conclusion

We have proposed modelling a traffic scene as a graph of interacting vehicles. Through this interpretation, we gain a flexible and abstract model for interactions. We evaluated two computationally efficient GNNs and proposed several adaptations for our scenario.

In particular, we moved away from simple graphs by introducing edge features, and have shown how this is necessary for our task. We have introduced several adaptations to GCN and GAT layers to enable processing edge features. We conducted an ablation study on these adaptations and found them to improve prediction quality.

On the *NGSIM* dataset, a traffic dataset with plentiful interaction, modelling interactions decreases prediction error by over 30% compared to the best baseline model. At the same time, we saw no decrease in prediction quality on the *HighD* dataset, a dataset with little interaction.

We also showed that GNN-based prediction of traffic participants can be made human-interpretable. We used this to gain insight in the performance differences of our models. In particular, we found that a better-performing variant of the GAT model also differentiates between cars based on their relative position and concentrates more on cars also relevant to a human in that situation. This shows that the graph interpretation not only performs well but also allows humans to inspect and interpret reasons for a prediction in single scenes.

While we have improved prediction quality, much work remains to be done: This work is only a proof-of-concept that modelling interactions as a graph is worthwhile and should thus be seen as only one technique for one aspect of traffic prediction. Integrating this model into existing state-of-the-art methodology, particularly RNNs and simulations, remains an open task.

And yet, it is a potent argument in favour of moving away from simple graphs: The inclusion of edge features decreases prediction error by 25% compared to the same model without edge features.

6 Heterogeneous Nodes

In Chapter 5, we have moved away from the simple graph model by introducing edge features. Yet many real-world tasks not only feature attributes for the relationships between objects but also different types of object (Fig. 6.1). This chapter introduces high-voltage power transmission grids as such a task[1] and shows how modelling these as graphs comes naturally. We also show how to apply GNN models on such a highly critical infrastructure task by combining the learned and fast GNN model with a provably correct solver to produce a solution faster than the solver alone could manage.

We first motivate why tackling this problem is important (Section 6.1). Section 6.2 then introduces the problems of Power Flow (PF) and Optimal Power Flow (OPF) and the equations and physical laws governing these. Section 6.3 introduces our methodology: Both how to model the power grid, how to build GNN models to operate on this problem, and how we ensure feasibility of the solution. Our experiments in Section 6.4 then aim to answer two primary questions: Which of our possible architectures performs best, and whether our methodology is able to tackle this problem.

6.1 Introduction

We first introduce the problem context and why it is important to solve. Afterwards, we introduce our approach.

6.1.1 Motivation

Electricity and heat production make up 25% of the world-wide yearly emissions of roughly 50 $GtCO_2$-eq. (IPCC 2014, p. 46f). Accordingly, any increase in efficiency has the potential of a large impact. At the same time, energy supply is an extremely critical and sensitive system, in which blackouts and brownouts are unacceptable. Any new technology deployed on such infrastructure needs to guarantee correctness.

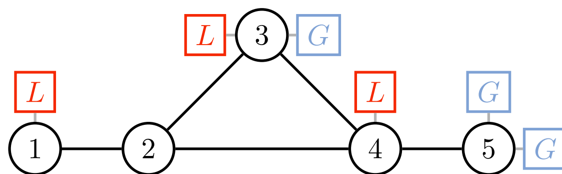


Fig. 6.1: The graph type used in this chapter: Aside from the edge features first introduced in Chapter 5, it also contains different types of nodes: Generators, buses, and loads.

[1] Parts of this chapter have been previously published (Diehl 2019c; Diehl 2019a).

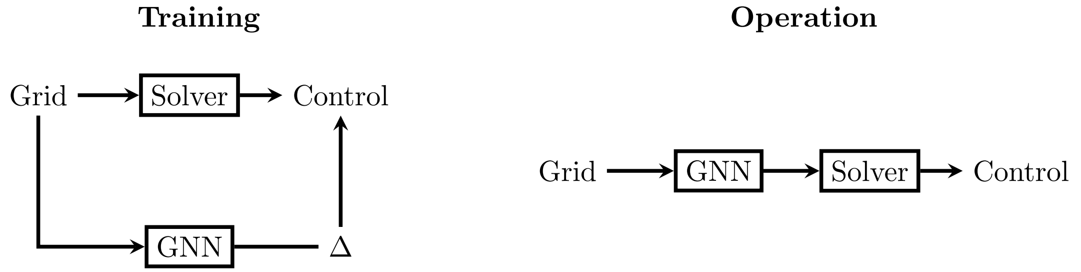


Fig. 6.2: General procedure used in this chapter. (left) During training, a classical solver takes a power grid as input and produces an ACOPF solution, which the GNN model is trained to approximate. (right) During operations, the GNN model's prediction is used to warm-start the ACOPF solver.

Day-to-day operating of an electrical grid requires scheduling of generator outputs. A Transmission System Operator (TSO) needs to optimize the purchase of power from different generators, each of which has different and potentially nonlinear costs and CO_2 emissions per produced MJ. Minimizing cost is known as OPF, and optimizing it using the full Alternating Current (AC) equations (referred to as ACOPF) has been proven to be NP-hard (Bienstock and Verma 2015). In the future, the problem size is bound to increase even more with the proliferation of small renewable generators.

Great progress has been achieved in the last decades (refer to Cain et al. (2012) for an overview). Yet, particularly in day-to-day operations which require solving OPF within a minute every five minutes, TSOs are forced to rely on linear approximations known as DCOPF. Solutions produced by these approximations are inefficient and therefore waste power and overproduce hundreds of megatons of CO_2-equivalent per year (Cain et al. 2012).

6.1.2 Approach

We propose to use machine learning to produce a solution to the OPF problem. Knowing that such a solution will not necessarily be optimal or even feasible (i.e. physically implementable), we can then use it to warm-start an ACOPF solver. Combining both approaches, we gain a significantly faster execution time while still guaranteeing feasibility.

This has previously been proposed by Guha et al. (2019). However, they only showed that a MLP could be used to produce such outputs on very small example datasets. They also did not integrate their results into an actual ACOPF optimizer.

The task is also interesting from the GNN perspective: In contrast to both the default benchmark datasets of Chapter 3 and the traffic prediction task of Chapter 5, the graph modelling this task contains both edge features (modelling the power transmission lines) and different types of nodes (generators, buses, and loads). There is a lack of approaches

for heterogeneous node and edge types. Accordingly, this task also provides us with additional knowledge on applying GNN models to more interesting tasks, which often feature diverse types of nodes and edge features.

We therefore also showcase different ways of modelling the task as a graph and conduct benchmarking to present the reader with an estimate on the different methods' performances. This obviously does not provide a definite answer to the question of how to model heterogeneous graphs; instead, we aim to inspire the reader and provide a starting point for their own exploration.

6.2 Simulating and Controlling Power Grids

The OPF problem is a crucial ingredient to modern power grids (Cain et al. 2012). TSOs use it to plan grid expansion and investment, request next-day commitments by power plant operators, and react to both usage spikes and issues or even losses in generators.

For the following, we follow the introduction by Frank and Rebennack (2016), to which we refer the reader for more detailed information.

6.2.1 Power Grid Equations

Electrical circuits are modelled as nodes and edges, representing physical interconnections and circuit elements respectively. These follow Kirchoff's laws: Kirchoff's Voltage Law (KVL) requires that the voltages around a closed loop sum up to zero[2], while Kirchoff's Current Law (KCL) requires that the incoming and outgoing currents for each nodes are equal.

Given that modern power grids operate using AC, voltage and current vary constantly. They are therefore analyzed under the assumption of sinusoidal steady-state operation, i.e. treating all voltages and currents as sinusoids with fixed magnitude, frequency, and phase shift. We then transform the time-domain representation into the phasor representation, complex equations in the frequency domain:

$$\underbrace{c \sin(2\pi f t + \gamma)}_{\text{time domain}} \quad \Leftrightarrow \quad \underbrace{c e^{i\gamma}}_{\text{frequency domain}}, \tag{6.1}$$

where c is the magnitude, f the frequency (which is fixed, and therefore omitted from the phasor equation), and γ the phase angle. Alternatively, the phasor representation can also be written in a polar representation as $c \cos\gamma + ic \sin\gamma$, with the real part $c \cos\gamma$ and the imaginary part $c \sin\gamma$. Lastly, voltage and current phasor magnitudes are expressed as Root-Mean-Square quantities and scaled by $\frac{1}{\sqrt{2}}$, which avoids introducing a separate scaling factor.

Leveraging the algebraic representation, Ohm's law for AC circuits is

$$\tilde{V} = \tilde{I}\tilde{Z} = \tilde{I}(R + iX), \tag{6.2}$$

[2]Taking voltage direction into account.

with $\tilde{V}$ being the voltage, $\tilde{I}$ the current, and $\tilde{Z}$ the complex impedance, distinguishing between real resistance R and imaginary reactance X. In the following, we use $\tilde{\square}$ to mark complex numbers. Reactance represents the effects of electrical storage, which produce a phase shift between voltage and current.

As with voltage and current, power is also treated as a phasor in an AC system. Complex power $\tilde{S}$ consists of real power P and reactive power Q:

$$\tilde{S} = P + iQ. \tag{6.3}$$

Real power represents net current flow from source to load over time, and occurs when voltage and current are in phase. Reactive power represents circular power flows, a net zero energy transfer over time. It occurs when voltage and current are orthogonal.

6.2.2 Power Flow

Given the a power grid, Power Flow (PF) aims to find bus voltages and power injections such that the power grid ends up in a physically feasible state. For this, a grid of b buses is modelled using

$$\tilde{I} = \tilde{Y}\tilde{V}, \tag{6.4}$$

with $\tilde{V}$ and $\tilde{I}$ being b-dimensional vectors of phasor voltages and currents respectively. $\tilde{Y}$ is the $b \times b$ complex bus admittance matrix, which is constructed from power line parameters. Eq. (6.4) is generally transformed via $S = \tilde{V}\tilde{I}^*$ into

$$\tilde{S} = \tilde{V} \odot \left(\tilde{Y}\tilde{V}\right)^*, \tag{6.5}$$

where $\square^*$ is the complex conjugate and $\odot$ is elementwise multiplication. Here, $\tilde{S} = P + iQ$ is the b-dimensional vector of complex power injections (i.e. of both real and reactive power injections) at each node. That is, $\tilde{S}$ is the difference between generated and consumed power at each node.

The advantage of the power flow-based formulation of Eq. (6.5) over the current-based formulation of Eq. (6.4) is the more direct computation of required electrical energy and the independence of injected power from system voltage angle.

With the decomposition of complex power $\tilde{S}$ above into real and reactive power injections, we can rewrite Eq. (6.5) as

$$P + iQ = \tilde{V} \odot \left(\tilde{Y}\tilde{V}\right)^*. \tag{6.6}$$

Real and reactive power injections P and Q are trigonometric functions of the system voltages and each bus i therefore has four variables: Real and reactive power injections P_i and Q_i, voltage magnitude V_i, and voltage angle δ_i. Since Eq. (6.6) gives us two equations (the real and imaginary part), we need to fix two of the four variables for each bus.

PF classically fixes these variables as follows:

Slack Bus One single slack bus has a fixed voltage magnitude and phase angle[3]. The slack bus is the only bus at which power can be freely varied.

PQ Bus At PQ buses, real and reactive power injections are fixed, while voltage magnitude and phase angle can be controlled.

PV Bus At PV buses, real power injection and voltage magnitude are fixed, while reactive power injection and phase angle can be controlled.

To solve the PF problem, we need to determine voltage angles for all buses except the slack bus. For PQ buses, we also have to determine voltage magnitudes. This requires us to solve the real part of Eq. (6.6) for all but the slack bus in addition to the imaginary part for PQ buses.

If such a solution exists, the power grid is in a feasible state.

6.2.3 Optimal Power Flow

Whereas PF concerns itself only with the feasibility of a given grid, OPF aims to produce a feasible solution that minimizes the cost of electricity generation while ensuring the safe operation of the grid. It assumes a number of controllable generators connected to buses. The cost of buying electricity from the generators varies, but is usually not linear. To ensure safe grid operation, real and reactive power, voltage magnitude, and phase angle for each bus have to remain within some minimum and maximum values. Similarly, the current magnitude flowing through branches has to remain below a certain limit[4]. Eq. (6.6) still has to be fulfilled.

First formulated in 1962 (Carpentier 1962), no solution technique exists that is both dependable and fast (Cain et al. 2012). In particular, Eq. (6.6) is non-convex and non-linear; and the function space is very uneven and contains many local minima. Indeed, Bienstock and Verma (2015) showed that OPF is NP-hard.

In practice, OPF is often simplified by using DC approximations (referred to as DCOPF in contrast to an optimization using the full AC equations, referred to as ACOPF). However, DCOPF can be very inaccurate, particularly for power lines under heavy load. Unfortunately, the latter is often a critical period in which precise control is necessary. There have also been approaches to use DCOPF to warm-start the full ACOPF solver, but this often does not work well.

Thus, reliable and scalable ACOPF solution methods remain an open and very interesting problem.

[3]Usually, PF uses the so-called unit system, in which voltage, power, and current are defined relative to reference magnitudes. There, the slack bus usually is fixed to a voltage of 1.0 p.u. and an angle of 0°.

[4]In the real world, thermal expansion of the power lines leads to sagging which in turn might lead to electrical arcs between lines and, for example, trees. This can start forest fires.

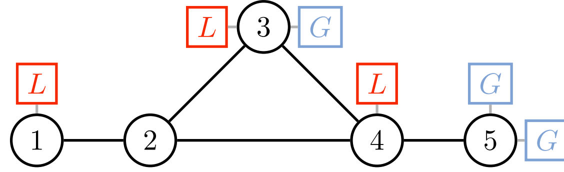


Fig. 6.3: Exemplary five-bus power grid. Buses are depicted as (numbered) circles, loads and generators are depicted with an L and a G respectively. Branches are black lines, while intra-bus connections are grey lines. As can be seen, buses can have multiple generators associated with them (bus 5), can have both generators and loads (bus 3) or might act only as an interconnection (bus 2). Buses, generators, loads, and branches all have features associated with them, while bus-to-component connections do not.

6.3 Methodology

Some approaches use a DCOPF solution to warm-start an ACOPF solver, thereby hopefully accelerating convergence and making its application feasibly. However, the large number of local minima noted in the previous section mean that the inaccurate solutions produced by the DCOPF solver often provide only little speed-up[5]. However, using the ACOPF solver guarantees feasible solutions. A better estimate than DCOPF provides should therefore provide larger speed-ups while at the same time ensuring a feasible solution.

GNN models can naturally be applied to the graph representation of power grids and are powerful models that should be able to provide better initial solutions. Our general idea is train a GNN model to predict the output of an ACOPF solver. Since the GNN model's output on new scenarios is not guaranteed to be feasible, we use its output instead to initialize the ACOPF solver, accelerating its convergence while guaranteeing feasibility of the solution.

6.3.1 Modelling the Power Grid

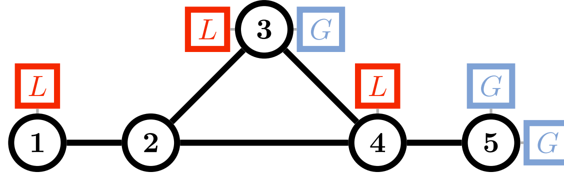
We model the power grid as a graph: Nodes are physical buses (each of which might contain loads and/or generators) connected by branches. Each component is characterized by both physical parameters (for example a branch's resistance) and constraints (for example a generator's maximum power output). An example is shown in Fig. 6.3.

This model of our problem closely follows the classical branch/bus model introduced in Section 6.2.

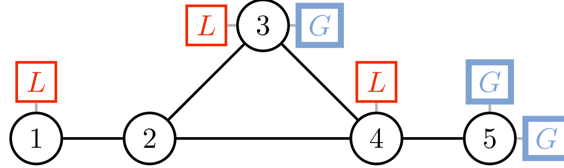
6.3.2 GNN Models for Power Grids

By modelling the problem as a graph, we can apply GNN models to it. GNN models are an attractive method to learn on power grids, since they allow us independence of actual

[5] As we will see in Section 6.4.3, mean speed-ups are less than 5%, though worst-case time is reduced by up to 50%.



(a) Components providing inputs. These are buses, generators, loads, and inter-bus connections.



(b) Components requiring predictions. These are only the generators.

Fig. 6.4: Components providing inputs and require predictions respectively. Relevant components have been bolded.

net topology and are data-efficient (contrary to modelling this with standard MLPs) and are well-suited to the sparse connectivity of the power grid graph.

Following the problem description in Section 6.2, buses, loads, and generators provide us with input features, as do branches. We require predictions for the generators. This is depicted in Fig. 6.4.

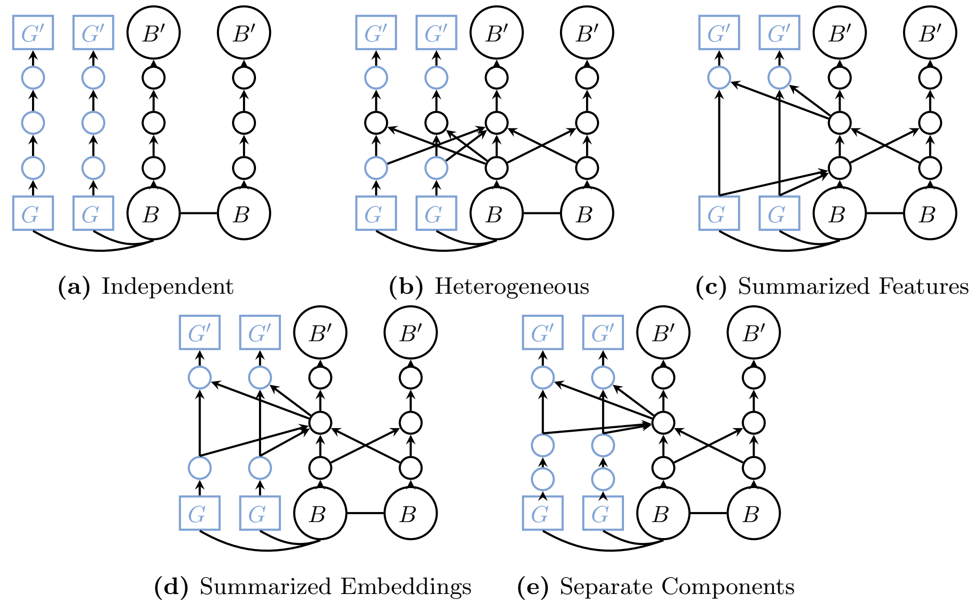
Most GNN methods consider only node features and adjacencies, ignoring edge features. We draw from the Graph Network framework (see Section 2.5.3.5), which can operate on node, edge, and global features. We adapt this framework, ignoring global features. This changes Eq. (2.46) to

$$\mathcal{E}'_{ij} = \mathcal{U}_e \left(\mathcal{E}_{ij}, \mathcal{V}_i, \mathcal{V}_j \right), \tag{6.7}$$

where $\mathcal{E}$ are the edge features, $\mathcal{U}_e$ is the edge update neural network, and $\mathcal{V}$ are the node features. Afterwards, the new node features $\mathcal{V}$ are computed by adapting Eq. (2.47) and using the learned node update function $\mathcal{U}_v$:

$$\mathcal{V}'_i = \mathcal{U}_v \left(\bigoplus_{\forall e_{ij} \in E} \left(\mathcal{E}_{ij} \right) . \mathcal{V}_i \right). \tag{6.8}$$

This formulation gives us a network architecture which supports transforming edge features. However, GNN models are usually applied to heterogeneous nodes, i.e. nodes that are of the same type. In contrast, this problem features three different types of components: Buses, generators, and nodes. Modelling these requires changes to the GNN model, several of which we evaluate. They are depicted in Fig. 6.5 for better comparison.



(a) Independent **(b)** Heterogeneous **(c)** Summarized Features

(d) Summarized Embeddings **(e)** Separate Components

Fig. 6.5: Graphical overview over the different model types operating on a very small graph and the resulting dataflow. All networks have one encoding, one GNN layer (excepting the Independent model), and one classification layer. The two generators are assumed to be connected to one bus. We do not show the influence of edge features onto computation. Colour-coding is used to denote parameter sharing for that layer, organized by components.

6.3.2.1 Independent Model

The simplest model (see Fig. 6.5a) assumes independence of components, and learns one MLP for each component type. That is, each layer updates each component's features c according to

$$c' = \mathcal{U}_c\left(c\right), \tag{6.9}$$

where $\mathcal{U}_c$ is a component-specific MLP.

This does not take graph topology or any component interaction into account, and therefore represents nothing more than a baseline.

6.3.2.2 Heterogeneous Model

We can treat each component as a separate node (see Fig. 6.5b). Edges are then both the original branches and component-to-bus connections.

Components and buses c are first embedded into the same feature space using separate linear embedding functions E_c to produce node embeddings $\mathcal{V}'$:

$$\mathcal{V}'_i = E_c\left(c_i\right). \tag{6.10}$$

Since component-to-bus connections do not have any features, we learn fixed representations for each type of component-to-bus connection. Afterwards, node and edge features are transformed using the base Graph Network from Section 6.3.2, with final predictions extracted from relevant nodes only.

6.3.2.3 Summarized Features Model

Classically, we could also concatenate component features to their corresponding bus (see Fig. 6.5c), creating initial node features v_i for each bus from both its own features b_i and the generator g_i and load l_i features:

$$\mathcal{V}'_i = \left(b_i \| \bigoplus^{l}\left(l_i\right) \| \bigoplus^{g}\left(g_i\right)\right). \tag{6.11}$$

Since one bus might have multiple components of the same type, we rely on existing knowledge on how to combine them to hand-craft the aggregation functions $\bigoplus^l$ and $\bigoplus^g$. For example, we can sum up maximum and minimum power generation capabilities for generators.

To enable different predictions for generators assigned to the same bus, the output layer predicts using the concatenation of both the final representation of the bus b_o and the original generator input features g_i:

$$g_o = O\left(b_o \| g_i\right). \tag{6.12}$$

6.3.2.4 Summarized Embeddings Model

Instead of summarizing raw features, we can also embed each component using a per-component linear embedding function E_c before summarizing multiple components per bus (see Fig. 6.5d).

$$\mathcal{V}' = \left(b_i \| \sum E_l\left(l_i\right) \| \sum E_g\left(g_i\right)\right). \tag{6.13}$$

Here, we use the sum instead of a hand-crafted aggregation function. This is similar to the approach taken by Zaheer et al. (2017) for representing object sets. As with the Summarized Feature model, we produce the output using the initial (embedded) generator features

$$g_o = O\left(b_o \| E\left(g_i\right)\right). \tag{6.14}$$

In contrast to the Summarized Features model, this does not require hand-crafted aggregation functions and can potentially distinguish better between similar components attached to the same bus.

6.3.2.5 Separate Components Model

Lastly, we can include existing knowledge (see Fig. 6.5e): We know that components are only connected to buses, and that they lack features. We therefore define updates to the components as

$$c' = \mathcal{U}_c\left(b \| c\right), \quad h_b = \mathcal{U}_h\left(b \| \sum_{g \in \mathcal{N}(b)} g \| \sum_{l \in \mathcal{N}(b)} l\right), \tag{6.15}$$

where c is a type of component (generator or load), b, g, l are the buses, generators, and loads respectively, and $\mathcal{U}$ is an MLP. h is an intermediate value per bus.

Branch features e are then computed according to their source and target buses b_s and b_t,

$$e' = \mathcal{U}_e\left(b_s \| e \| b_t\right). \tag{6.16}$$

Lastly, final bus features are computed based on the neighbouring buses and their corresponding branch features:

$$b' = \mathcal{U}_b\left(h \| \sum_{h,e \in \mathcal{N}(b)} \mathcal{U}_{h,e}\left(o \| e\right)\right). \tag{6.17}$$

This model architecture allows us to include existing knowledge as a bias: Each component is treated differently but is connected to one (or several) nodes, edge features are included, and component output depends on the connected bus. There are a total of six MLPs defining each layer of the model: $\mathcal{U}_g$, $\mathcal{U}_l$, $\mathcal{U}_o$, $\mathcal{U}_e$, $\mathcal{U}_{o,e}$, and $\mathcal{U}_b$.

6.3.3 Ensuring Feasibility

As noted in Sections 6.2 and 6.3, ensuring the feasibility of created solutions is a crucial problem. The usually-used DCOPF formulation of the problem often produce suboptimal or even infeasible results. For a power grid, where infeasible solutions can result in brownouts or blackouts, we cannot rely on the output of an approximate and uninterpretable model.

We first train the GNN model to predict the outputs of an ACOPF solver. While applying the ACOPF solver to a power grid has high computational requirements, this can happen offline during training. The trained GNN is extremely fast to execute, but we have no guarantee as to the feasibility of the produced solutions.

Instead, we use the prediction of the GNN model not as a control input to the power grid, but rather to warm-start the ACOPF solver. This guarantees a feasible solution and, since the ACOPF solver starts from a better starting point, accelerates final computation.

Since the final control output is always produced by an ACOPF solver, any solution has to be feasible and can therefore be used to control a power grid.

6.4 Experiments

We structure our experiments to answer two different questions:

Q1: Which of the adapted GNN models performs best?

Q2: Can an ACOPF solver be warm-started by a GNN model?

6.4.1 Datasets and Experimental Setup

Due to security concerns, real-world datasets on power networks are sparse, out-dated, or inaccurate (Birchfield et al. 2018). While a variety of example scenarios have been published, most of these are used for software correctness tests and therefore both unrealistic and orders of magnitude smaller than real power networks.

Birchfield et al. (2018) introduced a methodology to construct synthetic power networks based on known powerplants, census information, and geographic information. They produced several example datasets, of which we use *case_ACTIVSg200* (referred to as ILLINOIS) and *case_ACTIVSg2000* (referred to as TEXAS) datasets. These are synthetic representations of high-voltage transmission grids of the central part of Illinois (containing 200 buses with 230 and 115 kV networks) and of Texas (with 2 000 buses with 500, 230, 161, and 115 kV networks). The grids are depicted in Fig. 6.6 and Fig. 6.7 respectively.

We rely on the PowerModels.jl package (Coffrin et al. 2018) to model the power grid and convert from a power grid representation to a general non-linear optimization problem, which we then solve using the IPOPT package (Wächter and Biegler 2006). We implement the model using pytorch (Paszke et al. 2017) and the *pytorch-geometric* package (Fey and Lenssen 2019).

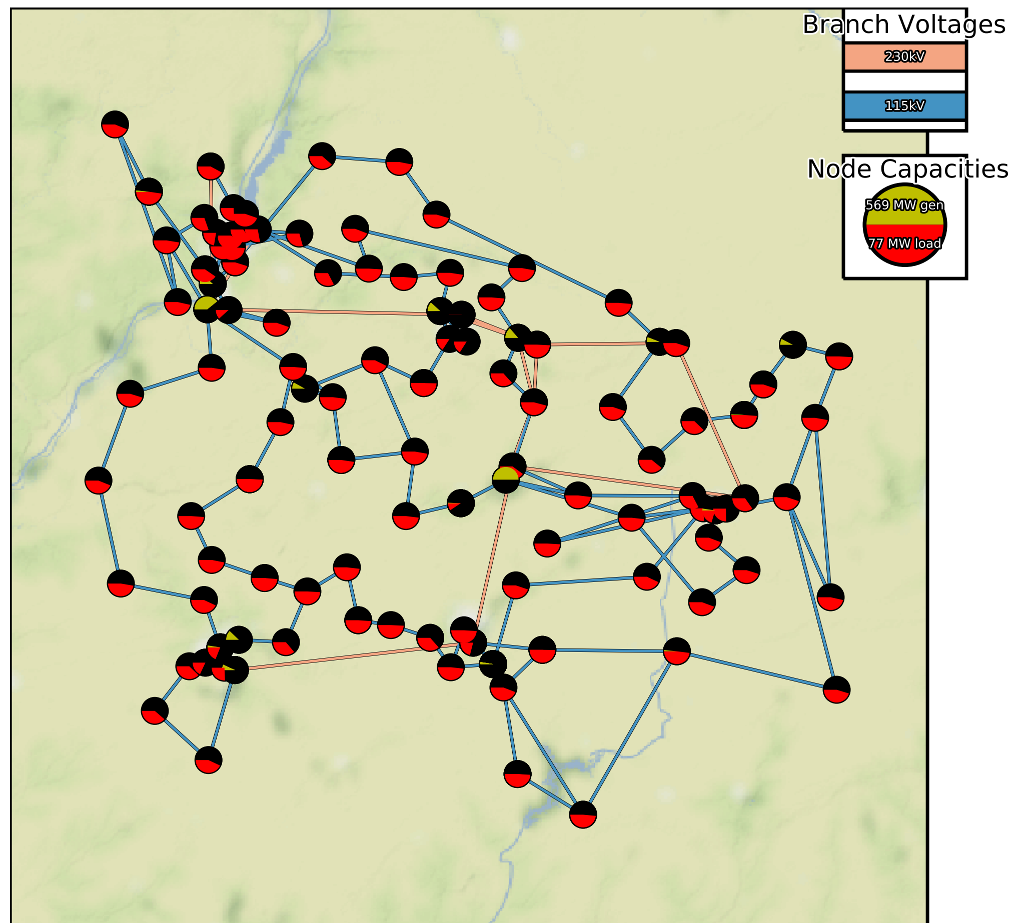


Fig. 6.6: Map of the synthetic ILLINOIS dataset. Each bus is annotated with its connected maximum generation capacity (yellow) and load (red), with a half-circle filled relative to the maximum generation capacity or load. Each branch is colour-coded according to its voltage.

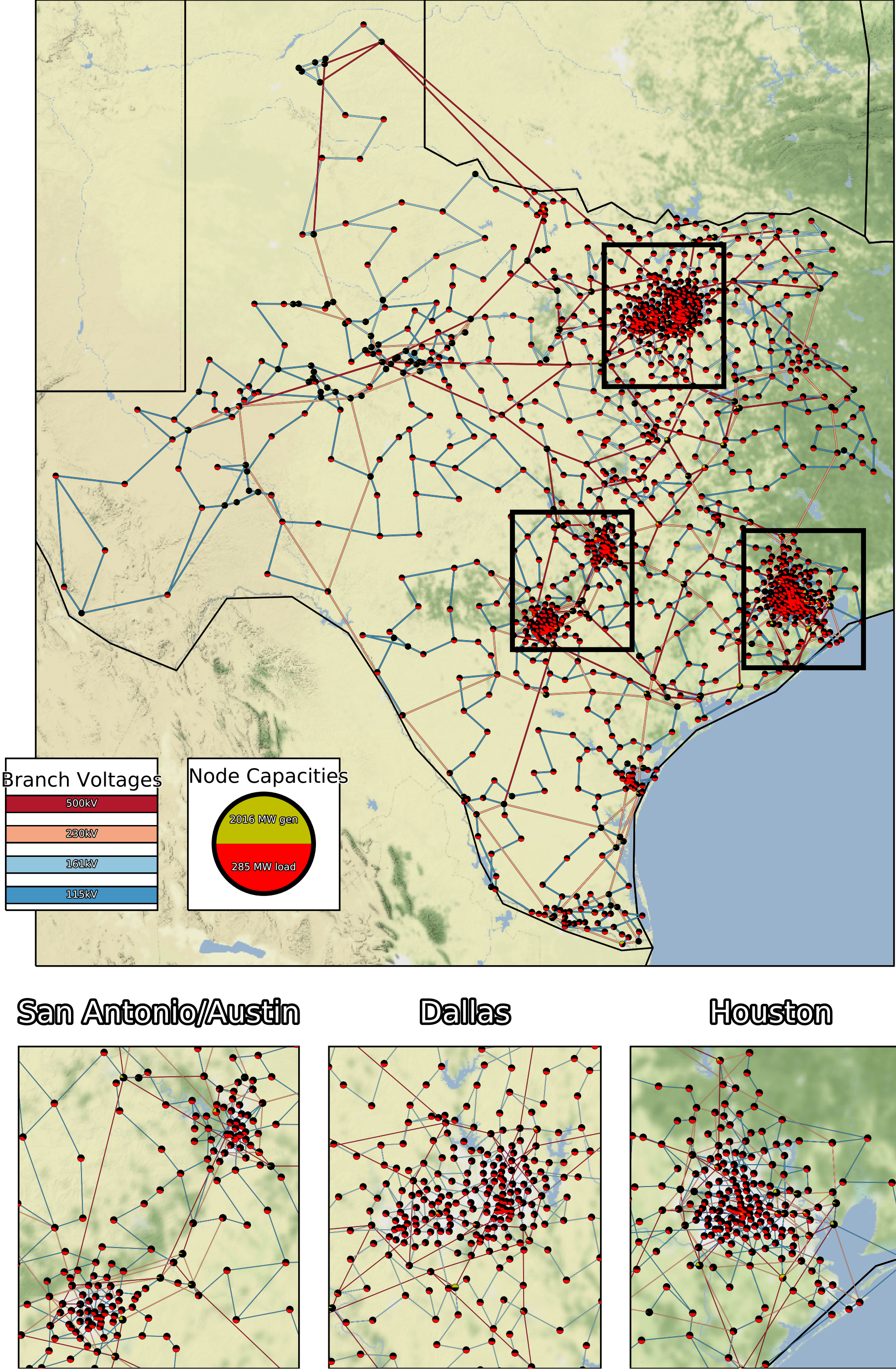


Fig. 6.7: Map of the synthetic TEXAS dataset. Depicted in the same style as Fig. 6.6

Table 6.1: Comparison of performances for different model architectures from Section 6.3.2.

Method	MAE
Independent	0.250 9
Heterogeneous	0.020 3
Summarized Features	0.018 6
Summarized Embeddings	0.017 1
Separate Components	0.021 8

Both datasets have synthetic hourly load distributions available for one year, and we randomly split these by day into train, validation, and test dataset. In the TEXAS dataset, total load drops below minimum generator output for about half the datapoints. Since our OPF solver does not support disabling generators, we ignore generator minimum production for TEXAS.

6.4.2 Q1: Which of the Adapted GNN Models Performs Best?

To answer **Q1**, we evaluate their performance on the TEXAS dataset. We measure performance according to the Mean Absolute Error (MAE) between the control outputs of the GNN model and the ACOPF solver.

6.4.2.1 Experimental Setup

We train all models identically using the Adam optimizer (Kingma and Ba 2014) at default parameters, and train for 500 epochs using the Mean Squared Error (MSE) loss. All models use 8 layers, residual connections, and batch normalization. We scale all models to the same number of parameters (about 200 k); accordingly, the Independent model uses 105 units per layer, Separate Components model 32. All other models[6] use 48 units per layer.

6.4.2.2 Results and Discussion

We report the MAE in Table 6.1. We make several observations:

- It is immediately apparent that the Independent model performs extremely badly, with errors an order of magnitude worse than the second-worst method. This is to be expected, as no generator prediction can depend on any other component.
- All but the Independent and the Separate Components models produce feasible solutions for every single one of our 2 208 test samples without any further enforcement of physical knowledge. The Independent model fails in about a quarter of the cases, while the Separate Component model fails on two cases.

[6]The Heterogeneous model also uses nine instead of eight layers to have the same receptive field as the other graph models.

- The Heterogeneous model, which treats all nodes as heterogeneous nodes, i.e. encoding them into the same representation space and applying a GNN model, performs well. This is arguably a useful starting point for other problems, too.
- Our adaptation of the Separate Components model does not perform better than the simpler Heterogeneous model.
- Both the Summarized Features and Summarized Embeddings model perform well, with the latter reducing the MAE by 16% compared to the Heterogeneous model.

6.4.2.3 Conclusion

In summary, the results clearly show that introducing prior knowledge through network architecture is necessary to achieve the best performance on more complex datasets. Given the disparity of such datasets, no single recommendation can be made on how to achieve it; however, we have shown several potential strategies which readers can use as a base for experiments. Yet, even a simple general-purpose representation can already deliver good results, leaving the inclusion of prior knowledge for hyper-parameter tuning.

For the concrete task of OPF, we use the Summarized Embeddings model going forward.

6.4.3 Q2: Can an ACOPF Solver be Warm-Started by a GNN Model?

To answer **Q2**, we apply the methodology on both the Illinois and the Texas dataset. We are interested in two parameters: The runtime of each method and the feasibility of the produced solutions.

6.4.3.1 Experimental Setup

We again train the model using the Adam optimizer (Kingma and Ba 2014) at default parameters, and train for 500 epochs using the MSE loss based on the ACOPF solver's result. The model has eight layers of 48 units each, and uses both residual connections and batch normalization.

During evaluation, we either use the result from the model, DCOPF solver, or ACOPF solver directly or use one of the former two to warm-start the ACOPF solver.

6.4.3.2 Results and Discussion

We discuss the results on Illinois and Texas separately. We report both mean and 95 % quantile runtime. The latter is important for deployed systems. For evaluation, we do not run the model on a GPU but note that even evaluated on the CPU, the GNN produces its results in less than half a second.

Table 6.2: Performance on ILLINOIS. Here, all methods find feasibly solutions. DC and AC are the corresponding power flow models, while → depicts warm-starting the ACOPF, either using the DC or the GNN model. All times are noted both in seconds and relative to the ACOPF optimizer.

Method	Mean Time [s] (rel.)	95% Time [s] (rel.)
GNN	0.03 (3%)	0.04 (1%)
DC	0.13 (13%)	0.18 (5%)
GNN → AC	0.75 (72%)	1.96 (55%)
DC → AC	0.98 (95%)	2.58 (72%)
AC	1.03	3.58

Illinois The results on the ILLINOIS dataset, consisting of 200 nodes, can be found in Table 6.2. We make several observations:

- All models find feasible solutions for all samples in our test set.
- The ACOPF solver warm-started by the DCOPF solver saves about 5% runtime on average. This is increased for longer-running scenarios, reducing the 95% quantile runtime by 28%.
- In simple models, the pure GNN model runs 4.5× faster than the DCOPF solver and 36× faster than the ACOPF solver.
- Even for such a small power grid, the combination of model and ACOPF saves 25 % runtime compared to the pure ACOPF. It also reduces 95% quantile runtime by almost half.

In summary, warm-starting the ACOPF solver using a GNN is worth it compared to the alternatives.

Texas The results on the larger and more realistic TEXAS dataset can be found in Table 6.3. The runtimes are graphically shown in Fig. 6.8. We can make several observations:

- The pure DCOPF model fails to find solutions to 6.6% of our scenarios. All other models succeed for every scenario.
- The ACOPF solver warm-started by the DCOPF solver only saves 2% runtime on average. This is again increased for longer-running scenarios, reducing the 95% quantile runtime by 45%.
- In simple models, the pure GNN model runs more than 10× faster than the DCOPF solver and is more than four orders of magnitude faster than the ACOPF solver.
- The combination of model and ACOPF saves 72 % runtime compared to the pure ACOPF. It also reduces 95% quantile runtime by 63 %.

Table 6.3: Performance on TEXAS. As can be seen, the DC power flow model fails to find solutions in 6.6% of scenarios.

Method	Mean Time [s] (rel.)	95% Time [s] (rel.)	Legal (of 2208)
GNN	0.2 (0.02%)	0.3 (0.02%)	2208
DC	3.2 (0.37%)	3.6 (0.22%)	2062
Model → AC	244 (28%)	621 (37%)	2208
DC → AC	850 (98%)	919 (55%)	2208
AC	863	1 670	2208

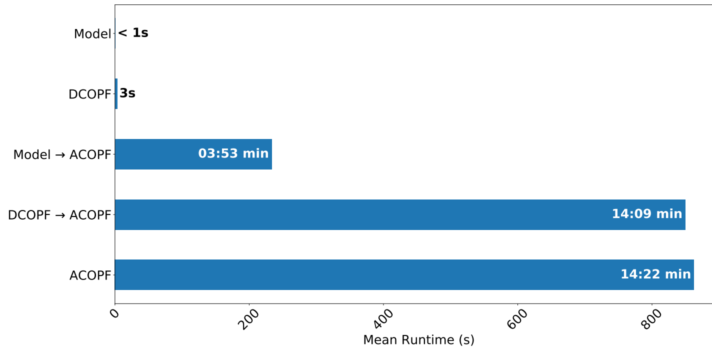


Fig. 6.8: Runtime comparison on TEXAS.

6.4.3.3 Conclusion

In summary, the results clearly show that the GNN model is superior both when used as an approximation, in which it produces better approximations than the DCOPF model in far less time, and when used to warm-start the ACOPF optimizer, in which it reduces runtime significantly. In itself, applying a GNN model to a power grid is almost free computation-wise. We believe that a better optimizer, in particular one that has been implemented with an eye towards warm-starting, should improve these advantages further.

6.5 Conclusion

Power grid optimization is an interesting problem to apply GNN models to, since GNN models are usually applied to simple problems of homogeneous nodes and lacking edge weights, and there is a lack of discussion on dealing with more interesting problems featuring nodes of different types and edge weights. Our case study on applying GNN models to a power grid task is therefore useful beyond the immediate task. By comparing five different approaches, we have shown that incorporating more problem-specific knowledge than just a graph structure into the model results in better performance. At the same time, our experiments indicate that the simplest approach can already give

good results. We therefore give practitioners a valuable starting point to their own experiments to apply GNNs to an ever larger class of problems.

We have shown that a GNN model can be trained offline on power grid optimization results. As an approximator, it performs four orders of magnitude faster than the ACOPF solver and 10× faster than the DCOPF approximator. Using it to warm-start the ACOPF solver allows us to dispense with the currently-used linear approximations and accelerates ACOPF computation by a factor of 3.8× while guaranteeing feasible solutions. Contrary to a vanilla ACOPF, it is feasible to deploy in short-time control scenarios.

However, these results should not be understood as more than a first proof-of-concept: (a) Deployed optimizers are more complex and problem-specific. (b) We rely on synthetic data, since actual grid data is not publicly accessible. (c) Deploying these solutions into existing power grid infrastructure is a gigantic task both from an engineering and organizational perspective.

Nonetheless, wide-spread adoption of such a system could save billions of dollars per year (Cain et al. 2012), and reduce emissions in the order of a hundred $MtCO_2$-eq./year.

7 Conclusion & Outlook

In this chapter, we draw conclusions from the preceding chapters. We first summarize the work we have presented previously (Section 7.1), presenting the main conclusions from each of the previous chapters. In Section 7.2, we draw three main conclusions from this thesis as a whole: We recommend modelling problems as graphs, how to approach a new graph task, and combining GNN models and classical models for critical infrastructure tasks. Lastly, in Section 7.3, we look to the future, recommend further work, and note that aversions to the simple graph paradigm are already spreading throughout the GNN community.

7.1 Summary

Within this thesis, we have seen how GNN models can be applied to tasks that defy standard ML approaches, allowing us to learn on problems that are most easily structured as graphs.

- In **Chapter 1**, we outlined both the need for GNN models and the shortcomings of current models.
- In **Chapter 2**, we provided the reader with the necessary background. This included how to represent graphs (Section 2.2), the necessary spectral graph theory which many methods rely on to apply the concept of convolution to graphs (Section 2.3), how to formulate graph problems (Section 2.4), and finally a number of GNN methods (Section 2.5). The latter include recurrent graph networks, and spectral and spatial graph networks. We also introduced theoretical analysis of GNN models (Section 2.5.3.6) and special computational considerations necessary for implementing GNN models (Section 2.5.4).
- In **Chapter 3**, we compared a large number of the GNN models from Section 2.5 and their tweaks on simple graphs, i.e. those with only node features and featureless connections between nodes. We evaluated our research questions on three semi-supervised node classification tasks and five graph classification tasks. We drew several conclusions: GNN models show a large variability in their results. GCN layers show good performance on a large number of datasets, which makes them a good starting point for constructing an initial model. Lastly, GNN models do not yet profit from larger depth.
- In **Chapter 4**, we showed the first departure from the restrictions of simple graphs: We introduced a pooling layer which modifies graph topology, allowing further

layers to act on groups of nodes instead of single nodes. We showed that EdgePool, a local and hard pooling layer, outperforms other pooling methods and can be easily integrated in both graph- and node-level prediction problems. We also showed that it carries a 50% computational time penalty, but that it improved memory usage on graphs.

- In **Chapter 5**, we explored the task of traffic prediction on highways. Here, we departed from simple graphs in yet another manner by introducing simple edge features. We showed that representing a traffic scene as a graph and applying GNN models allows for powerful and expressive models to be applied while keeping model complexity limited. We showed how the changes we introduced into two GNN models improve prediction quality, with the best model reducing prediction error by 30%. We also evaluated construction methods for the scene representation, and showed how to make the model more interpretable.
- In **Chapter 6**, we applied GNN models to the far more complex problem of power grid optimization. By modelling a power grid as a graph—and thereby departing from the simple graph assumption by having both complex edge features and heterogeneous nodes—we were able to predict the output of a classical optimization algorithm. This model produced results four orders of magnitude faster than the original solver. And by using the model's output to warm-start the optimization algorithm, we improved the latter's runtime by a factor of 3.8× while keeping the optimizer's guarantees for a feasible solution. This is crucial for deploying such techniques to real-life power grids.
- Lastly, in **Chapter 7**, we summarize this thesis, draw three main conclusions from our experiments, and close with an outlook based on averting the simple graph paradigm.

7.2 Conclusion

In Chapter 1, we motivated this thesis with the observation that most publications of GNN models concentrate on the case of simple graphs, i.e. those with fixed graph topology and only node features. Arguing that this was a very limited set of tasks, we set out to explore different tasks in which the simple graph assumptions would not hold.

Modelling the world using graphs is a powerful and well-applicable technique, and we have easily found three example tasks in which the simple graph assumptions do not hold. Changing graph topology as a general task and both traffic prediction and power grid control as applications each violate the simple graph assumptions normally underlying GNN models, but in each of these tasks these additions are necessary for a good performance. The ease with which we have found such problems is proof they are everywhere, and we accordingly need to find guidelines for designing models for such tasks.

No single work can produce guidelines that apply to every such task. Nonetheless, we feel we are able to draw several conclusions that are generally applicable.

7.2.1 Modelling Tasks as Graphs is Powerful

In this thesis, we have seen a large number of tasks which can be modelled well by using a graph. These tasks range from citation networks, recommender systems, molecule attribute prediction, and traffic participant prediction to control of a power transmission grid. In each of these, transforming the samples into a graph allowed applying similar models. When looking at algorithmic problems, "it is amazing how often messy applied problems have a simple description and solution in terms of classical graph properties." (Skiena 2008, p. 146). The same applies to messy applied problems in ML—transforming them into a suitable graph representation simplifies ML models and provides a useful structure for engineers to think about.

That is not to say a graph approach is always well-suited. Indeed, on most of the tasks on which today's ML algorithms are deployed, they are unneeded. On video, image, time-series, or simple vector tasks, there exists a spatial and fixed-grid structure and using GNN models would mean giving up structural information and having to deal with the computational inefficiencies. Yet, environments less structured than image or time-series data will profit from graph modelling.

7.2.2 Recommendations for Exploring New Problems

Building architectures for a new task is always a challenge. In areas like image or text processing, the domain is generally well-understood and tasks share a certain common structure, which allows engineers to apply well-known and tested models. In contrast, tasks that are modelled as graphs are far more diverse, with their structure, attributes, and the tasks itself varying significantly between problems. This thesis alone has explored tasks that differ for example in which components are labelled, in graph attributes and topology, in the complexity of the prediction, and in the output type.

Nevertheless, we can point to several observations we have made within this thesis as good starting points for work on any tasks:

- The GCN layer is an excellent baseline layer, since it performs well on many tasks.
- Contrary to other deep learning models, current GNN models suffer when introducing too much depth. Thus, depth should be carefully controlled.
- Care should be taken to model the task as close as possible. Introducing edge feature, for example, often improves performance significantly. This might require adaptations to the models used.
- A simple model applied to a task provides an excellent baseline, even in the face of heterogeneous nodes.

In summary, these observations suggest to use graph data but to start with the simplest models possible. The improvements mostly stem from better modelling the input data as a graph rather than far more complex models.

7.2.3 Building ML Models for Critical Infrastructure

Aside from the modelling aspects of Chapter 6, that task also showed us a way to integrate ML models into highly critical infrastructure such as high-voltage power transmission grids: Not by replacing existing algorithms but by supporting them.

The GNN model produces solutions to the power grid scenarios four orders of magnitude faster than the ACOPF solver. Yet, it cannot be relied on to always produce feasible solutions. This is particularly critical for anomalous situations, since they are not likely to be found in training data. Yet, anomalous situations are exactly the critical situations in which operation margins are thin and blackouts or brownouts loom. These are also the scenarios in which ACOPF optimizers cannot be used because they are too slow.

However, we have shown that combining these two produces a system that is both faster and as robust as before: The GNN model produces a starting point for the ACOPF solver and does so in negligible runtime. The ACOPF solver, accelerated by the initial state, can solve the situation far faster while still guaranteeing feasibility. Indeed, we believe that both more engineering on the ACOPF solver to improve performance from warm-started starting points and on the GNN model to produce warm-starting points better suited for the ACOPF solver will increase the speed-up even more. The advantage of a solution guaranteed to be feasible remains.

7.3 Outlook

As this thesis has argued, we believe the concentration on simple GNN models based benchmarks of simple graphs to lead to less applicability to real-world tasks. Accordingly, we believe there is a case to be made for more complex benchmarks and tasks. Luckily, the recent Open Graph Benchmark datasets (Hu et al. 2021) are a significant step into this direction, having both larger graph sizes and including graphs with edge features.

Of course, neither these graphs nor the tasks explored in this thesis are the end-all model that can be applied to all graphs. Particularly global features and more complex topologies such as multi-graphs are an interesting expansion from the simple graph model, and research in these has also accelerated.

From a more task-specific point of view, Kelly et al. (2020) have recently created a series of competitions to control power grids, concentrating on reinforcement learning algorithms. We find it heartening to see transmission grid operators involved in this field, since the interest and inclusion of infrastructure operators is the best approach to, at some point, deploy ML models on such critical infrastructure.

In summary, we believe the modelling of tasks using graphs to be a powerful tool well-suited to many problems. We look forward to future developments in this field and to the deployment of such models on real-world tasks.

Publications

First-Author Publications

Frederik Diehl (2019c). “Warm-Starting AC Optimal Power Flow with Graph Neural Networks”. In: *Climate Change Workshop at the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*

Frederik Diehl (2019a). “Applying Graph Neural Networks on Heterogeneous Nodes and Edge Features”. In: *Graph Representation Learning Workshop at the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*

Frederik Diehl, Thomas Brunner, et al. (2019a). “Graph neural networks for modelling traffic participant interaction”. In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE

Frederik Diehl (May 2019b). “Edge Contraction Pooling for Graph Neural Networks”. In: *arXiv:1905.10990 [cs, stat]*. arXiv: 1905.10990

Frederik Diehl, Thomas Brunner, et al. (2019b). “Towards graph pooling by edge contraction”. In: *ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data*

Frederik Diehl and Alois Knoll (2019). “Tree Memory Networks for Sequence Processing”. In: *International Conference on Artificial Neural Networks*. Springer, pp. 431–443

Further Publications

Thomas Brunner, Frederik Diehl, Michael Truong Le, et al. (2019a). “Guessing smart: Biased sampling for efficient black-box adversarial attacks”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 4958–4966

Thomas Brunner, Frederik Diehl, and Alois Knoll (2019). “Copy and paste: A simple but effective initialization method for black-box adversarial attacks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*

Thomas Brunner, Frederik Diehl, Michael Truong Le, et al. (2019b). “Leveraging Semantic Embeddings for Safety-Critical Applications”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*

Wieland Brendel et al. (2020). “Adversarial vision challenge”. In: *The NeurIPS’18 Competition*. Springer, pp. 129–153

Michael Truong Le et al. (2018). “Uncertainty Estimation for Deep Neural Object Detectors in Safety-Critical Applications”. In: *21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, pp. 3873–3878

Chih-Hong Cheng et al. (2018). “Neural networks for safety-critical applications—Challenges, experiments and perspectives”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, pp. 1005–1006

David Lenz et al. (2017b). “Deep neural networks for Markovian interactive scene prediction in highway scenarios”. In: *Intelligent Vehicles Symposium (IV), 2017 IEEE*. IEEE, pp. 685–692

Gereon Hinz, Martin Büchel, et al. (2017). “Designing a far-reaching view for highway traffic scenarios with 5G-based intelligent infrastructure”. In: *8. Tagung Fahrerassistenz*. Lehrstuhl für Fahrzeugtechnik mit TÜV SÜD Akademie. München

Bibliography

Adebayo, Julius, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim (2018). "Sanity checks for saliency maps". In: *Advances in Neural Information Processing Systems.*

Battaglia, Peter W., Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu (June 4, 2018). "Relational Inductive Biases, Deep Learning, and Graph Networks". In: arXiv: `1806.01261 [cs, stat]`.

Bienstock, Daniel and Abhinav Verma (Dec. 22, 2015). "Strong NP-hardness of AC power flows feasibility". In: *arXiv:1512.07315 [math].* arXiv: `1512.07315`.

Birchfield, Adam B., Ti Xu, Komal Shetye, and Thomas Overbye (Jan. 3, 2018). "Building Synthetic Power Transmission Networks of Many Voltage Levels, Spanning Multiple Areas". In: *Proceedings of the 51st Hawaii International Conference on System Sciences.* ISBN: 978-0-9981331-1-9. DOI: `10.24251/HICSS.2018.349`.

Borgwardt, Karsten M., Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel (June 1, 2005). "Protein Function Prediction via Graph Kernels". In: *Bioinformatics* 21 (suppl_1), pp. i47–i56. ISSN: 1367-4803. DOI: `10.1093/bioinformatics/bti1007`.

Brendel, Wieland, Jonas Rauber, Alexey Kurakin, Nicolas Papernot, Behar Veliqi, Sharada P Mohanty, Florian Laurent, Marcel Salathé, Matthias Bethge, Yaodong Yu, et al. (2020). "Adversarial vision challenge". In: *The NeurIPS'18 Competition.* Springer, pp. 129–153.

Bronstein, Michael M., Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst (July 2017). "Geometric Deep Learning: Going beyond Euclidean Data". In: *IEEE Signal Processing Magazine* 34.4, pp. 18–42. ISSN: 1053-5888. DOI: `10.1109/MSP.2017.2693418`. arXiv: `1611.08097`.

Bruna, Joan, Wojciech Zaremba, Arthur Szlam, and Yann LeCun (Dec. 20, 2013). "Spectral Networks and Locally Connected Networks on Graphs". In: arXiv: `1312.6203 [cs]`.

Brunner, Thomas, Frederik Diehl, and Alois Knoll (2019). "Copy and paste: A simple but effective initialization method for black-box adversarial attacks". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops.*

Brunner, Thomas, Frederik Diehl, Michael Truong Le, and Alois Knoll (2019a). "Guessing smart: Biased sampling for efficient black-box adversarial attacks". In: *Proceed-*

ings of the IEEE/CVF International Conference on Computer Vision, pp. 4958–4966.

Brunner, Thomas, Frederik Diehl, Michael Truong Le, and Alois Knoll (2019b). "Leveraging Semantic Embeddings for Safety-Critical Applications". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops.*

Cain, Mary B., Richard P. O'Neill, and Anya Castillo (2012). "History of Optimal Power Flow and Formulations". In: *Federal Energy Regulatory Commission 1 (2012).*

Cangea, Cătălina, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò (Nov. 3, 2018). "Towards Sparse Hierarchical Graph Classifiers". In: arXiv: `1811.01287 [cs, stat]`.

Carpentier, J (1962). "Contribution to the economic dispatch problem". In: *Bulletin de la Societe Francoise des Electriciens* 3.8, pp. 431–447.

Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud (Dec. 2019). "Neural Ordinary Differential Equations". In: *arXiv:1806.07366 [cs, stat]*. arXiv: 1806.07366.

Cheng, Chih-Hong, Frederik Diehl, Gereon Hinz, Yassine Hamza, Georg Nührenberg, Markus Rickert, Harald Ruess, and Michael Truong Le (2018). "Neural networks for safety-critical applications—Challenges, experiments and perspectives". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, pp. 1005–1006.

Cho, Kyunghyun, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (Sept. 2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *arXiv:1406.1078 [cs, stat]*. arXiv: 1406.1078.

Coffrin, Carleton, Russell Bent, Kaarthik Sundar, Yeesian Ng, and Miles Lubin (June 2018). "PowerModels.jl: An Open-Source Framework for Exploring Power Flow Formulations". In: *2018 Power Systems Computation Conference (PSCC)*. DOI: `10.23919/PSCC.2018.8442948`.

Dai, Hanjun, Zornitsa Kozareva, Bo Dai, Alex Smola, and Le Song (July 2018). "Learning Steady-States of Iterative Algorithms over Graphs". en. In: *International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, pp. 1106–1114.

Defferrard, Michaël, Xavier Bresson, and Pierre Vandergheynst (June 30, 2016). "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: arXiv: `1606.09375 [cs, stat]`.

Diehl, Frederik (2019a). "Applying Graph Neural Networks on Heterogeneous Nodes and Edge Features". In: *Graph Representation Learning Workshop at the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019).*

— (May 2019b). "Edge Contraction Pooling for Graph Neural Networks". In: *arXiv:1905.10990 [cs, stat]*. arXiv: 1905.10990.

— (2019c). "Warm-Starting AC Optimal Power Flow with Graph Neural Networks". In: *Climate Change Workshop at the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019).*

Diehl, Frederik, Thomas Brunner, Michael Truong Le, and Alois Knoll (2019a). "Graph neural networks for modelling traffic participant interaction". In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE.

— (2019b). "Towards graph pooling by edge contraction". In: *ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data*.

Diehl, Frederik and Alois Knoll (2019). "Tree Memory Networks for Sequence Processing". In: *International Conference on Artificial Neural Networks*. Springer, pp. 431–443.

Dobson, Paul D and Andrew J Doig (2003). "Distinguishing enzyme structures from non-enzymes without alignments". In: *Journal of molecular biology* 330.4, pp. 771–783.

Federal Highway Administration (FHWA) (2005). *US highway 80 dataset*. Tech. rep. FHWA-HRT-06-137.

Fey, Matthias and Jan Eric Lenssen (Mar. 6, 2019). "Fast Graph Representation Learning with PyTorch Geometric". In: arXiv: 1903.02428 [cs, stat].

Frank, Stephen and Steffen Rebennack (Aug. 11, 2016). "An introduction to optimal power flow: Theory, formulation, and examples". In: *IIE Transactions 48.12 (2016)*. ISSN: 10.1080/0740817X.2016.1189626.

Gao, Hongyang and Shuiwang Ji (2019). "Graph U-Net". In: *International Conference on Machine Learning*. PMLR.

Giles, C Lee, Kurt D Bollacker, and Steve Lawrence (1998). "CiteSeer: An automatic citation indexing system". In: *Proceedings of the third ACM conference on Digital libraries*, pp. 89–98.

Gilmer, Justin, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl (Apr. 4, 2017). "Neural Message Passing for Quantum Chemistry". In: arXiv: 1704.01212 [cs].

Gori, Maria Cristina, Gabriele Monfardini, and Franco Scarselli (2005). "A New Model for Learning in Graph Domains". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* International Joint Conference on Neural Networks 2005. Vol. 2. Montreal, Que., Canada: IEEE, pp. 729–734. ISBN: 978-0-7803-9048-5. DOI: 10.1109/IJCNN.2005.1555942.

Guha, Neel, Zhecheng Wang, and Arun Majumdar (2019). "Machine Learning for AC Optimal Power Flow". In: *arXiv:1910.08842 [cs.LG]*. arXiv: 1910.08842.

Hamilton, William L., Rex Ying, and Jure Leskovec (June 7, 2017). "Inductive Representation Learning on Large Graphs". In: arXiv: 1706.02216 [cs, stat].

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). "Deep Residual Learning for Image Recognition". In: *Arxiv.Org* 7.3. arXiv: 1512.03385 ISBN: 978-1-4673-6964-0, pp. 171–180. ISSN: 1664-1078. DOI: 10.3389/fpsyg.2013.00124.

Henaff, Mikael, Joan Bruna, and Yann LeCun (June 2015). "Deep Convolutional Networks on Graph-Structured Data". In: *arXiv:1506.05163 [cs]*. arXiv: 1506.05163.

Hinz, Gereon, Martin Büchel, Frederik Diehl, Guang Chen, Annkathrin Kraemmer, Juri Kuhn, Venkatnarayanan Lakshminarasimhan, Malte Schellmann, Uwe Baumgarten, and Alois Knoll (2017). "Designing a far-reaching view for highway traffic scenarios

with 5G-based intelligent infrastructure". In: *8. Tagung Fahrerassistenz*. Lehrstuhl für Fahrzeugtechnik mit TÜV SÜD Akademie. München.

Hinz, Gereon, Martin Buechel, Frederik Diehl, Guang Chen, Annkathrin Krämmer, Juri Kuhn, Venkatnarayanan Lakshminarasimhan, Malte Schellmann, Uwe Baumgarten, and Alois Knoll (2017). "Designing a far-reaching view for highway traffic scenarios with 5G-based intelligent infrastructure". In: 8. Tagung Fahrerassistenz, p. 8.

Hu, Weihua, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec (Feb. 2021). "Open Graph Benchmark: Datasets for Machine Learning on Graphs". In: *arXiv:2005.00687 [cs, stat]*. arXiv: 2005.00687.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv:1502.03167*, pp. 1–11. ISSN: 0717-6163. DOI: 10.1007/s13398-014-0173-7.2. pmid: 15003161.

IPCC (2014). *Climate Change 2014: Synthesis Report. Contribution of Working Groups I, II and III to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change.*

Kelly, Adrian, Aidan O'Sullivan, Patrick de Mars, and Antoine Marot (Mar. 2020). "Reinforcement Learning for Electricity Network Operation". In: *arXiv:2003.07339 [cs, eess, stat]*. arXiv: 2003.07339.

Kersting, Kristian, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann (2020). *Benchmark Data Sets for Graph Kernels.*

Kim, Sunghwan, Jie Chen, Tiejun Cheng, Asta Gindulyte, Jia He, Siqian He, Qingliang Li, Benjamin A Shoemaker, Paul A Thiessen, Bo Yu, et al. (2021). "PubChem in 2021: new data content and improved web interfaces". In: *Nucleic Acids Research* 49.D1, pp. D1388–D1395.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: arXiv: 1412.6980.

Kipf, Thomas N. and Max Welling (Sept. 9, 2016). "Semi-Supervised Classification with Graph Convolutional Networks". In: arXiv: 1609.02907 [cs, stat].

Krajewski, Robert, Julian Bock, Laurent Kloeker, and Lutz Eckstein (2018). "The highD Dataset: A Drone Dataset of Naturalistic Vehicle Trajectories on German Highways for Validation of Highly Automated Driving Systems". In: *2018 IEEE 21st International Conference on Intelligent Transportation Systems (ITSC).*

Krämmer, Annkathrin, Christoph Schöller, Dhiraj Gulati, and Alois Knoll (2019). "Providentia - a large scale sensing system for the assistance of autonomous vehicles". In: *arXiv preprint arXiv:1906.06789.*

Kuefler, Alex, Jeremy Morton, Tim Wheeler, and Mykel Kochenderfer (June 2017). "Imitating driver behavior with generative adversarial networks". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017 IEEE Intelligent Vehicles Symposium (IV), pp. 204–211. DOI: 10.1109/IVS.2017.7995721.

Kurtzer, Gregory M., Vanessa Sochat, and Michael W. Bauer (2017). "Singularity: Scientific Containers for Mobility of Compute". In: *Plos One* 12.5, e0177459. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459.

Lee, Junhyun, Inyeop Lee, and Jaewoo Kang (Apr. 17, 2019). "Self-Attention Graph Pooling". In: arXiv: 1904.08082 [cs, stat].

Lenz, David, Frederik Diehl, Michael Truong Le, and Alois Knoll (June 2017a). "Deep neural networks for Markovian interactive scene prediction in highway scenarios". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017 IEEE Intelligent Vehicles Symposium (IV), pp. 685–692. DOI: 10.1109/IVS.2017.7995797.

— (2017b). "Deep neural networks for Markovian interactive scene prediction in highway scenarios". In: *Intelligent Vehicles Symposium (IV), 2017 IEEE*. IEEE, pp. 685–692.

Levie, Ron, Federico Monti, Xavier Bresson, and Michael M. Bronstein (May 22, 2017). "CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters". In: arXiv: 1705.07664 [cs].

Li, Ruoyu, Sheng Wang, Feiyun Zhu, and Junzhou Huang (Apr. 2018). "Adaptive Graph Convolutional Neural Networks". en. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1. Number: 1. ISSN: 2374-3468.

Li, Yujia, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel (Sept. 2017). "Gated Graph Sequence Neural Networks". In: *arXiv:1511.05493 [cs, stat]*. arXiv: 1511.05493.

Lim, Lek-Heng (Aug. 2020). "Hodge Laplacians on Graphs". en. In: *SIAM Review*. Publisher: Society for Industrial and Applied Mathematics. DOI: 10.1137/18M1223101.

Lucic, Mario, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet (Oct. 2018). "Are GANs Created Equal? A Large-Scale Study". In: *arXiv:1711.10337 [cs, stat]*. arXiv: 1711.10337.

McCallum, Andrew Kachites, Kamal Nigam, Jason Rennie, and Kristie Seymore (July 1, 2000). "Automating the Construction of Internet Portals with Machine Learning". In: *Information Retrieval* 3.2, pp. 127–163. ISSN: 1573-7659. DOI: 10.1023/A:1009953814988.

Morris, Christopher, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe (Oct. 4, 2018). "Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks". In: arXiv: 1810.02244 [cs, stat].

Morton, Jeremy, Tim A Wheeler, and Mykel J Kochenderfer (2016). "Analysis of Recurrent Neural Networks for Probabilistic Modeling of Driver Behavior". In: *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–10.

NT, Hoang and Takanori Maehara (May 2019). "Revisiting Graph Neural Networks: All We Have is Low-Pass Filters". In: *arXiv:1905.09550 [cs, math, stat]*. arXiv: 1905.09550.

Olah, Chris, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev (2018). "The Building Blocks of Interpretability". In: *Distill*. https://distill.pub/2018/building-blocks. DOI: 10.23915/distill.00010.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer (2017). "Automatic Differentiation in PyTorch". In: *NeurIPS Autodiff Workshop*.

Pineau, Joelle (Dec. 2018). *Reproducible, Reusable, and Robust Reinforcement Learning*. Montreal, Quebec, Canada.

Sanchez-Gonzalez, Alvaro, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia (Sept. 2020). "Learning to Simulate Complex Physics with Graph Networks". In: *arXiv:2002.09405 [physics, stat]*. arXiv: 2002.09405.

Scarselli, Franco, Maria Cristina. Gori, Ah Chung Tsoi, M. Hagenbuchner, and Gabriele Monfardini (Jan. 2009). "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1, pp. 61–80. ISSN: 1045-9227, 1941-0093. DOI: 10.1109/TNN.2008.2005605.

Schöller, Christoph, Vincent Aravantinos, Florian Lay, and Alois Knoll (2020). "What the constant velocity model can teach us about pedestrian motion prediction". In: *IEEE Robotics and Automation Letters* 5.2, pp. 1696–1703.

Selvaraju, Ramprasaath R, Abhishek Das, Ramakrishna Vedantam, Michael Cogswell, Devi Parikh, and Dhruv Batra (2016). "Grad-CAM: Why did you say that?" In: *arXiv preprint arXiv:1611.07450*.

Sen, Prithviraj, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad (Sept. 6, 2008). "Collective Classification in Network Data". In: *AI Magazine* 29.3, pp. 93–93. ISSN: 2371-9621. DOI: 10.1609/aimag.v29i3.2157.

Shchur, Oleksandr, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann (Nov. 14, 2018). "Pitfalls of Graph Neural Network Evaluation". In: arXiv: 1811.05868 [cs, stat].

Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman (Dec. 2013). "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: *arXiv:1312.6034 [cs]*.

Skiena, Steven S. (2008). *The Algorithm Design Manual*. London: Springer. DOI: 10.1007/978-1-84800-070-4.

Smilkov, Daniel, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg (2017). "Smoothgrad: removing noise by adding noise". In: *arXiv preprint arXiv:1706.03825*.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15, pp. 1929–1958. ISSN: 15337928. DOI: 10.1214/12-AOS1000. arXiv: 1102.4807.

Sundararajan, Mukund, Ankur Taly, and Qiqi Yan (2017). "Axiomatic attribution for deep networks". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 3319–3328.

Telgarsky, Matus (June 2016). "Benefits of Depth in Neural Networks". In: *29th Annual Conference on Learning Theory*. Ed. by Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir. Vol. 49. Proceedings of Machine Learning Research. Columbia University, New York, New York, USA: PMLR, pp. 1517–1539.

Thiemann, Christian, Martin Treiber, and Arne Kesting (2008). "Estimating Acceleration and Lane-Changing Dynamics Based on NGSIM Trajectory Data". In: *Transportation Research Record: Journal of the Transportation Research Board* 2088, pp. 90–101.

Treiber, Martin (2000). "Congested traffic states in empirical observations and microscopic simulations". In: *Physical Review E* 62.2, pp. 1805–1824. DOI: 10.1103/PhysRevE.62.1805.

Truong Le, Michael, Frederik Diehl, Thomas Brunner, and Alois Knoll (2018). "Uncertainty Estimation for Deep Neural Object Detectors in Safety-Critical Applications". In: *21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, pp. 3873–3878.

Veličković, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio (Oct. 2017). "Graph Attention Networks". en. In.

Wächter, Andreas and Lorenz T Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical programming* 106.1, pp. 25–57.

Wale, Nikil and George Karypis (Dec. 2006). "Comparison of Descriptor Spaces for Chemical Compound Retrieval and Classification". In: *Sixth International Conference on Data Mining (ICDM'06)*. Sixth International Conference on Data Mining (ICDM'06), pp. 678–689. DOI: 10.1109/ICDM.2006.39.

Weisfeiler, Boris and Andrei Leman (1968). "The reduction of a graph to canonical form and the algebra which appears therein". In: *NTI, Series* 2.9. (Translation from Russian by Grigory Ryabov), pp. 12–16.

Wu, Zonghan, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu (Jan. 2, 2019). "A Comprehensive Survey on Graph Neural Networks". In: arXiv: 1901.00596 [cs, stat].

Xu, Keyulu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka (Oct. 1, 2018). "How Powerful Are Graph Neural Networks?" In: arXiv: 1810.00826 [cs, stat].

Xu, Keyulu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka (July 3, 2018). "Representation Learning on Graphs with Jumping Knowledge Networks". In: *International Conference on Machine Learning*. International Conference on Machine Learning, pp. 5453–5462.

Yanardag, Pinar and S. V. N. Vishwanathan (Oct. 8, 2015). "Deep Graph Kernels". In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 1365–1374. ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2783417.

Ying, Rex, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec (July 2018). "Graph Convolutional Neural Networks for Web-Scale Recommender Systems". In: ACM, pp. 974–983. ISBN: 978-1-4503-5552-0. DOI: 10.1145/3219819.3219890.

Ying, Rex, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec (June 22, 2018). "Hierarchical Graph Representation Learning with Differentiable Pooling". In: arXiv: 1806.08804 [cs, stat].

Zaheer, Manzil, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R. Salakhutdinov, and Alexander J. Smola (2017). "Deep Sets". In: Advances in Neural Information Processing Systems, pp. 3391–3401.

Zhang, Muhan and Yixin Chen (2018). "Link Prediction Based on Graph Neural Networks". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., pp. 5167–5177.

Zhang, Muhan, Zhicheng Cui, Marion Neumann, and Yixin Chen (2018). “An End-to-End Deep Learning Architecture for Graph Classification”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.