Deep Learning on Graphs: Directed Graphs, Edge Structures and Graph Estimation

Michael Peter Kenning

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

School of Mathematics and Computer Science
Swansea University

31st July 2023

Declarations

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .... (candidate)

Date ....

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)

Date ....

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate)

Date ....

The University's ethical procedures have been followed and, where appropriate, that ethical approval has been granted.

Signed .... (candidate)

Date ....

To my grandfather Peter George Royal.

Abstract

In the last decade and a half, machine learning has been refounded on a class of techniques called deep learning. The earliest, most prominent techniques of deep learning were restricted in their application to regularly structured domains. A new set of techniques, broadly referred to as geometric deep learning, extends the application of deep learning approaches to irregular domains, in particular the use of the graph. A graph is an effective means of representing irregular relations between discretely sampled points; its use has its attendant research challenges that has brought about a flourishing field of research. In this work we investigate three of those challenges, namely learning on directed graphs, one of the many variants of the graph; learning on the edge-structure of graphs; and graph estimation, i.e. the estimation of graph structure from the data itself.

In the first chapter of our work, we consider the challenge of learning on the edge structure of a graph in application to a datacentre and present a convolution technique for the edge-structure of a directed graph representing a datacentre.

In the second chapter of our work, we present a strategy to estimate two complementary graphs, the long-term or static graph and short-term or dynamic graph from combinations of temporal, cyclical data. Additionally, we propose an attention-based convolution for directed graphs that factorises neighbourhood signals into in- and out-flows.

In the third and final chapter of our work, we draw on the work of the previous two chapters and design a graph estimation strategy to learn the complementary structures of molecular graphs. For this purpose, we define a new kind of graph for the estimation, which we call the graph complement, and use it in predicting molecular properties by incorporating intramolecular forces not present in the original graph. The structure thus learned is used to propagate vertex and edge signals on a directed graph.

The work is concluded with a reflection on the contributions of the thesis and prospective areas of research in the field of graph deep learning.

Acknowledgements

First and foremost I would like to thank my supervisor Prof. Xianghua Xie. He has been my mentor since 2017 when I began my master's degree at Swansea University and took me on later as a doctoral student. As my supervisor throughout the years, he has always been patient, understanding, supportive and encouraging in my academic endeavours. The challenge of supervision was complicated by the pandemic, in the midst of which I moved to Germany in 2020, despite which he did not slacken in his support for me. I am indebted and very grateful to him for my professional development. I would also like to thank my second supervisor, Dr. Jingjing Deng, for his academic support and his comprehensive understanding of deep learning. The idea of using linegraphs in the first study is Dr. Deng's, who suggested it at the beginning of my doctoral studies. The idea then served as a basis for the work in this thesis. I would like to Stavros Georgousis for his indispensable work in co-authoring the survey paper that proved foundational to my doctoral work. The experiments in the latter chapters of this work would not have been possible without the support of AccelerateAI, whose Sunbird supercomputer, managed through Supercomputing Wales and part-funded by the European Regional Development Fund through the Welsh Government, made it possible to conduct the later experiments in this work.

I am grateful to my colleagues at Swansea University, with whom I have sadly not had much contact since the beginning of the pandemic. From 2017 till 2021, they offered their moral support, even after some of them had long completed their doctoral work. To Dr. Michael Edwards I would like to extend my gratitude for his help. A few times during my PhD I have turned to him in desperation for advice and moral support and he has always been kind and forthcoming with his time and his reassurance. Dr. Edwards also lent me a great deal of guidance when I first started my research on graph deep learning and allowed me to establish a firm basis of understanding. His sense of humour and the sharp remarks he makes has been a spur to my better understanding of the field as well as a source of amusement in tough times. I would also like to thank Dr. Joss

Whittle, Dr. David George, Dr. Alex Lee and Dr. Gavin Tsang, who together with Dr. Edwards constituted a vital web of social and academic support during my doctoral studies. As my seniors, at various points they offered advice and reassurance that I would be all the worse off without. I would also like to thank Dr. Hassan Eshkiki, Dr. Majedaldein Almahasneh, Dr. Mohammed Ali, Dr. Hanchi Ren and Dr. Ali Alqahtani, fellow-members of the Vision Lab at Swansea University, and Filippos Pantekis, Olga Petrovska and other colleagues at Swansea University.

My family and friends in Wales and Germany have provided me with unwavering and loving support through the years, especially during the hardest years of the pandemic and my unexpectedly early move to Germany. My deep gratitude goes to my long-suffering and cherished mother, Alison Royal, and my brother, James Kenning, especially for the fact that they boxed up and moved my possessions out of my old flat after I had moved to Germany. My thanks are also owed to my friends from Swansea, Meghan Hagain-Edwards, Dan Williams, Luc Daley, Hannah Sweetapple, Sam Ballantine, Ben Long, Saul Moore, Zoe Dodson, Louis Jawad, Nick Tucker, Edward Blackaby and my wider circle of friends in Swansea, who have been there to joke around with or share thoughts and feelings. I give my thanks to my friends from other parts of the world, too, whose dear friendship has helped me these last years; I would like to thank Aiden Bains, Elisabeth Schober and Vladyslav and Aiym Solokha.

I am especially grateful to Kristina Seit, who unstintingly supported and encouraged me for more than six years of my Master's degree and doctoral degree and through very difficult periods in my life. I would also like to thank her parents, Birgit and Thomas Seit, for hosting me for a whole year after I first moved to Germany, as well as Svenja and Paul Seit, who were good company in hard times. I am also grateful to her wider family and her family friends, who warmly welcomed me when I first arrived.

I owe my deep gratitude to the friends, colleagues and acquaintances in Bamberg who helped me integrate into and prosper in my adopted city, in particular my partner Irene Lo Faro, both for the philosophical discussions that informed the background of this work and for her unconditional support and counsel. I am also grateful to Hannah Brodel, Carmen Marr, Victoria Müller, Konstantin Klug, Johanna Rost, Stefan Holland, Martin Wimmer, Jan Berndt, Dea Xhafa and my other friends and colleagues at Geschichte für Alle e.V., all of whom made Bamberg a home for me.

Contents

List of Publications

The following publications resulted from the work conducted for this doctoral thesis. An outline of contributions may be read in Section 1.3.

1. Stavros Georgousis, Michael P. Kenning, and Xianghua Xie (2021). "Graph Deep Learning: State of the Art and Challenges". In: *IEEE Access* 9, pp. 22106–22140. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3055280

2. Michael Kenning et al. (2021). "Locating Datacenter Link Faults with a Directed Graph Convolutional Neural Network". In: *Proceedings of the International Conference on Pattern Recognition Applications and Methods*. SCITEPRESS - Science and Technology Publications, pp. 312–320. DOI: 10.5220/0010301403120320

3. Michael P. Kenning et al. (2022). "A directed graph convolutional neural network for edge-structured signals in link-fault detection". In: *Pattern Recognition Letters* 153, pp. 100–106. ISSN: 01678655. DOI: 10.1016/j.patrec.2021.12.003

4. Michael Kenning and Xianghua Xie (2023). "Attention-based Graph Estimation and Directed Convolution for Prediction of Traffic Conditions". In: *Proceedings of the 10th Internatinal Workshop on Deep Learning on Graphs (To be published.)*

List of Symbols

A list of symbols used repeatedly in the thesis.

$\boldsymbol{A}, \boldsymbol{L}, \boldsymbol{D}$ Adjacency, Laplacian and degree matrices of the graph.

$d_{\text{in}}(x), d_{\text{out}}(x)$ The in- and out-degree of vertex x.

$d(x)$ The degree of vertex x.

$\bar{G}$ The graph complement of graph G.

$\Gamma(x), \Gamma_i(x)$ The neighbours of the vertex x, the i-hop neighbours of the vertex x.

$\Gamma_{\text{in}}(x), \Gamma_{\text{out}}(x)$ The in- and out-neighbours of a vertex x.

$\boldsymbol{A}, \boldsymbol{a}$ Matrix, vector.

$n = |V|, m = |E|$ The graph order n, i.e., the number of vertices, and the graph size m, i.e., the number of edges.

$x, y \in G, xy \in G$ Vertices x and y in the graph, the edge xy in the graph.

G, V, E Graph, vertices, edges.

Acronyms

P-F model	Perron-Frobenius spectral directed model
AUC	area under the curve
BBBP	blood–brain-barrier penetration
BPLHM	body-part-level hybrid model
CNN	convolutional neural network
CommNet	Communication Network
DGCN	Directed Graph Convolutional Network
DGCNN	directed graph convolutional neural network
DGCNN-in	in-neighbor-only directed graph convolutional neural network
DGCNN-out	out-neighbor-only directed graph convolutional neural network
DGCNN-R	directed graph convolutional neural network with residual connections
DGCNN/I	directed graph convolutional neural network including the inverse edge
EEG	electroencephalography
ESOL	estimated solubility
GAN	generative adversarial network
GAT	graph attention network
GCN	Graph Convolutional Network
GCNN	graph-based convolutional neural network
GECNN	graph edge convolutional neural network
GIN	graph isomorphic network
GMM	Gaussian mixture model
GN	Graph Network
GNN	graph neural network
GPU	graphics processing unit

HMGNN	heterogeneous molecular graph neural network
IN	Interaction Network
KNN	k nearest neighbours
LSTM	long short-term memory
MAE	mean absolute error
MAPE	mean average percentage error
MEGNet	material graph network
MGC	molecular graph convolution
MLP	multi-layer perceptron
MPNN	message-passing neural network
PCA	principal component analysis
ReLU	rectified linear unit
RF	random forest
RMSE	root mean squared error
RNN	recurrent neural network
Saak	subspace approximation with augmented kernels
SVD	singular-value decomposition
SVM	support-vector machine
ToR	top-of-rack
UGCNN	undirected graph convolutional neural network
UGCNN+I	undirected graph convolutional neural network including the inverse edge merged
UGCNN/I	undirected graph convolutional neural network including the inverse edge separately

List of Tables

List of Figures

Chapter 1

Introduction

Contents

1.1 Motivations

Techniques in machine learning have been almost wholly refounded on a class of techniques described as deep learning (Goodfellow, Bengio, and Courville, 2016). The power of deep learning lies in its capacity to learn its own feature representations directly from wholly unprocessed input data. Deep learning as originally conceived is however restricted to regular domains such as images and other discretely sampled domains (LeCun, Bengio, and Hinton, 2015). Over the last decade, the challenge of learning on irregular domains has been met by the development of graph-based deep learning (Shuman et al., 2013; Hamilton, Ying, and Leskovec, 2017a). The graph has proved to be a useful mathematical object for describing irregular relations between discretely sampled objects (Georgousis, Kenning, and Xie, 2021) and also serves as an inductive bias on a learning algorithm (Battaglia et al., 2018). The field of graph deep learning has become fertile ground for a plethora of techniques aiming to learn patterns on diverse graph-structured data.

Indeed, the graph and its attendant learning challenges has given rise to research on unique and exciting problems which will concern us in this thesis. If the original motivation for machine learning is to allow an algorithm to learn feature representations automatically, the corollary is that a graph-based algorithm can learn the structure of the data, too. As a consequence a new direction of research has developed that focusses on learning graph structures with various designations such as neural relational inference, graph generation and structure learning; we use the term *graph estimation*. A second direction of research concerns the learning of signals structured on the edges of a graph, which we term *edge learning*. We also consider techniques for learning on one variant of the the graph in which edges have orientation called the *directed graph* (research is as spirited on other variants of the graph which are beyond the scope of this thesis; see Georgousis, Kenning, and Xie, 2021). In this thesis we present novel techniques for the learning on the directed structure, the edge structures of graphs and the estimation of graph structures. We explore practical issues in their representation and consider the ways in which such techniques can be applied to application areas beyond those we use in this thesis to validate our methods.

1.1.1 Learning on Directed Graphs

A directed graph describes the unidirectional relations between discretely sampled objects. Graph-based convolution techniques are generally designed for undirected graphs. These techniques occasionally neglect the unique learning challenges that could be represented properly in learning; for directed structures act as an inductive bias on a model and serve as a means to provide a more detailed description of the relational structure of a domain. Direction in graphs can be used, among other purposes, to describe the unequal relations between sampled points as well as factor information into in- and out-flows (Li et al., 2018a). In this work we consider domains where directed structures have frequently been used for the purposes of learning on graphs. In Chapter 3 we use a directed graph to structure a datacentre in order to localise the source of link-faults. Direction is inseparable from a domain where the information between vertices is *per se* directed, specifically in that the links between the vertices, representing machines in a network, are either up-links, passing information up a network hierarchy, or down-links, passing information downwards (Arzani, Behnaz et al., 2018). In Chapter 4 we estimate a directed graph in order to make predictions of traffic conditions. The vertices represent traffic sensors and the relations between sensors are not symmetrical. Sensors positioned on major roads, for example, record traffic volumes that will have far greater collateral effects on minor roads than the traffic flow on a single minor road could have on a major road. In Chapter 5, four directed graphs are used to estimate the intermolecular forces and other forces acting upon a molecule. The intermolecular forces again do not necessarily act symmetrically between atoms, and nor do groups of atoms, which together may exert forces upon other atoms or groups of atoms; such physical dynamics may be encoded as weighted, directed edges.

1.1.2 Learning on the Edge Structures of Graphs

Representations may be structured on the vertices or edges of a graph. Existing techniques incorporate vertex and edge signals into their definitions of convolution, notable amongst which are the graph-based models that have been applied to molecular learning (Kearnes et al., 2016). Accordingly, the incorporation of edge features in general is a question of the particular nature of the application domain. The optimal means of incorporating edge features into convolution is an ongoing research question that has led to a range of techniques. One approach is to use linegraphs, which represent the second-order structure of graph and hence the second-order interactions within a domain (Chen, Bruna, and Li, 2019). In Chapter 3 we present a technique for learning on the edge structure of a graph by using a directed linegraph. We consider the theoretical difficulties of definition that arise from applying convolution to a directed linegraph. In Chapter 5 we use a directed linegraph to learn edge representations in application to molecular prediction and also describe an approach for learning the interactions of edge features in a second-order structure, which we call the linegraph complement. We also present an approach for propagating learned edge features over two different second-order structures and for combining the representations for molecular property prediction.

1.1.3 Graph Estimation

Graph estimation has been yet to be formalised comprehensively, such that no consensus has emerged for a single umbrella term (see Georgousis, Kenning, and Xie, 2021 for an attempt to formalise the area; see Ma and Tang, 2021, §15.4 for a brief discussion of related work), although the characteristics of graph estimation can be identified in many existing works (Georgousis, Kenning, and Xie, 2021). The applications of graph estimation are far broader than either learning on directed graphs or edge learning, since it covers all manner of variants on the graphs. In Chapter 3 we apply our proposed graph estimation technique to the learning of graph structure from long- and short-term information in a traffic prediction task. The technique uses different compositions of information to discover complementary but differing structures in order to aid learning on traffic data. In Chapter 5 we apply graph estimation for the purposes of learning a complementary structure in order to predict molecular properties. The complementary structures represent the first- and second-order structures not definitionally present in the original graph. The purpose is to discover implicit relations in the graph-structured molecular features that inform the determination of molecular features and therefore the prediction of properties, namely intramolecular forces (Xiong et al., 2020).

1.1.4 Objective

Although there is already diverse work available on the use of edge representations in the aforementioned domains and directed graphs are not new in the field, the two directions of research, edge learning and learning on directed structures, especially when the two are taken together, receive comparatively less attention compared to graph convolution applied to other domains. An objective of this thesis is to study techniques to work with directed graph and directed linegraphs. By contrast, graph estimation is a formally inchoate field with no single widely used term to describe the overarching objective. It is possible to discern various approaches in one direction or another in the literature (Georgousis, Kenning, and Xie, 2021), but the overall directions are diffuse and difficult to harmonise with one another without a clear theoretical framework. Broadly speaking, techniques are characterised as direct and indirect means of learning the graph structure via the adjacency matrix, or learning an interaction function for entities. Our objective is to broaden the application of graph estimation to different compositions of information, as is the purpose of Chapter 4, and to different combinations of structure, as in Chapter 5, with a focus on directed graphs and the edge structures thereof. The approaches presented in this work can therefore be considered as one of many indirect means of learning the graph. The proposed methods are not limited to the domains they target; they demonstrate of what it is possible to do with an estimated graph structure. Ultimately, the purpose of this work is to consider more advanced problems of learning on graphs and estimating the graphs themselves.

1.2 Overview

In light of the motivations presented in Section 1.1, the aim of this work is to elaborate applications of graph estimation, convolution on directed graphs and edge-focussed learning to new formulations of existing problems. We present two new approaches to learning on directed structures and two new approaches for estimating graph structures. Chapter 3 concerns itself with the unique problems that arise from using a directed linegraph to structure edge learning. In the first graph estimation approach in Chapter 4 we present a technique for learning complementary graphs for a temporal problem from two combinations of cyclical data. This approach has applications to domains where there are static and dynamic temporal structures that need to be balanced in predicting outcomes in domains. The directed attention-based approach to graph convolution presented therein is able to factor features into their respective information flows learned in the graph estimation approach. In the second graph estimation approach in Chapter 5 we draw together the work from the previous two sections by designing a graph estimation procedure to complement two existing graph structures. The estimated graph structures complement both first- and second-order structures in the data and enrich the vertex and edge representations learned for predicting molecular properties.

1.3 Contributions

The main contributions of this work are the following:

A technique for structuring learning on graph edges. We present two methods for learning on the edge structure or second-order structure of graphs representing domains. The first method uses the directed linegraph to define the second-order structure of a datacentre. We consider in particular the special case of inverse edges that arise from the construction of directed linegraphs on directed graphs and present a method to learn on inverse edges. We evaluate the approach on datacentre simulations of link-faults and find that the approach more accurately localises link faults in a datacentre than undirected convolution and non-graph learning algorithms. The second method uses a directed linegraph for the propagation of edge representations, which are further propagated by the linegraph complement, a graph that, as far as we know, has not been presented in the literature. We evaluate the approach on two benchmark molecule datasets and find that the linegraph complement, in combination with an original linegraph, is able to propagate edge features globally and reduce prediction errors, as well as simultaneously reducing the number of propagation steps necessary for learning.

An isotropic method for convolution on directed graphs. We present an isotropic kernel for directed graphs that factors neighbouring signals into two groups according to their incidence to a focal vertex. It is a simple construction with only a few parameters per output channel. The method is evaluated on a datacentre simulation, where we find that the inclusion of inverse edges as a separate term or as part of the neighbourhood worsened the performance, thereby justifying our decision to separate out inverse edges as a term altogether.

A technique for estimating graph structure from temporal, cyclical data. We present an attention-based approach to graph estimation that uses two compositions of cyclical data, which we call *static–dynamic fusion*, to make two complementary estimations of the graph structure inhered in the input. In the evaluation of the approach on two popular traffic datasets, we find that the high quantity of missing data in one dataset leads to large errors in the baselines that did not use graph estimation. On the contrary, graph estimation reduces the error caused by the missing data, which is robustly handled by the static–dynamic fusion approach.

An anisotropic, attention-based convolution for directed graphs. We present an anisotropic, attention-based convolution for directed graphs that separates neighbourhoods into their incidence to focal vertices. In the evaluation on the two traffic datasets, we find that the directed attention was robust to missing data in the more difficult dataset, and in one case is as capable as the static–dynamic fusion approach to estimation described above in reducing prediction error, and moreover even avoids a period of high error that knocks the approaches using static–dynamic fusion off kilter.

A new graph to supplement a predefined structure. We present an attention-based approach to graph estimation that learns the higher-order interactions on a molecular graph. The molecular graphs are defined solely in terms of chemical bonding; the complementary structure, which we call a *graph complement*, can learn the structure of the intramolecular relations in the molecule. The downstream learning is further factorised into the different structures to allow the model to separately learn the two different structures. The approach is validated on two popular molecule benchmark dataset, in which we find that the graph complements improve the prediction errors on two benchmark datasets. Analysis additionally showed that the complement graphs reduce the number of propagation steps necessary to achieve an optimal result. To the best of our knowledge, no technique has been presented to estimate such graph structures end-to-end in learning molecular representations.

The outcomes of this thesis have also contributed to several publications as outlined in the List of Publications. The key contributions of each paper to the contents of the thesis are summarised below:

Stavros Georgousis, Michael P. Kenning, and Xianghua Xie (2021). "Graph Deep Learning: State of the Art and Challenges". In: *IEEE Access* 9, pp. 22106–22140. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3055280.

We conduct survey of the state of the art in graph deep learning. The discussions on the various challenges of deep learning on graphs contributed significantly to the discussions in this thesis in Chapter 2.

Michael Kenning et al. (2021). "Locating Datacenter Link Faults with a Directed Graph Convolutional Neural Network". In: *Proceedings of the International Conference on Pattern Recognition Applications and Methods*. SCITEPRESS - Science and Technology Publications, pp. 312–320. DOI: 10.5220/0010301403120320.

We present the directed graph convolution applied to the structure of the linegraph in application to link-fault detection in datacentre simulations. It contributes the theoretical presentation of the method and its results to Chapter 4.

Michael P. Kenning et al. (2022). "A directed graph convolutional neural network for edge-structured signals in link-fault detection". In: *Pattern Recognition Letters* 153, pp. 100–106. ISSN: 01678655. DOI: 10.1016/j.patrec.2021.12.003.

We present more detailed exploration and set of result of the directed graph applied to the structure of the linegraph in application to link-fault detection in datacentre simulations. It contributes to the discussion, set of results and discussion presented in Chapter 4.

Michael Kenning and Xianghua Xie (2023). "Attention-based Graph Estimation and Directed Convolution for Prediction of Traffic Conditions". In: ***Proceedings of the 10th Internatinal Workshop on Deep Learning on Graphs.*** **(To be published.)**

We present the static–dynamic fusion approach to graph estimation and the directed attention-based graph convolution. A narrower set of results are presented in the paper. The methodology and part of the results and discussion contribute to Chapter 5.

1.4 Outline

The rest of this work is structured as follows:

Chapter 2: Background In this chapter we first review the theoretical underpinnings of deep learning and the motivation for deep learning. The discussion leads to the drawbacks of conventional deep learning, which in turn motivates the use of graphs in structuring some irregular domains. The graph and its characteristics, variants and data structures are described. Next we describe convolution on graphs, enumerating prominent techniques for graph convolution and the key challenges of learning on graphs which we address in the work presented in this thesis. Finally we review some applications of graph deep learning, with particular reference to those challenges.

Chapter 3: Directed Linegraph for Learning on Edges The topic of this chapter is the use of the directed linegraph to structure a learning problem on the edges of a graph, specifically the localisation of link faults in a datacentre. Using a directed linegraph, which is definitionally a directed graph, we design a convolution technique for directed graphs with an isotropic kernel that factorises signals into separate streams of information and the signals of inverse edges. We use the structure of the directed graph and the proposed convolution technique to design a graph-based convolutional neural network, which we evaluate against a feed-forward neural network and a random forest.

Chapter 4: Graph Estimation on Directed Graphs and Directed Graph Attention In this chapter we describe our approach to estimating a traffic graph from temporal, cyclical data. We use a novel combination of different periods of traffic data to estimate two graph structures, one relating to the long-term structure and the other to the short-term structure. We also propose an attention-based convolution technique for directed graphs. We evaluate the resulting model against our own implementation of a state-of-the-art method.

Chapter 5: Estimation of Graph Complements In this chapter we define a new kind of graph, the graph complement. We then describe our approach to graph estimation which we apply to learning molecular structures alongside predefined molecular graphs. The result is evaluated on two commonly used baseline molecular datasets.

Chapter 6: Conclusions and Future Work To conclude, we recapitulate the outcomes of the work in the thesis and consider future work in the field.

Chapter 2

Background

Contents

The world is its own best model.

Rodney A. Brooks, 1991

The majority of the material relevant to graph deep learning that is cited in this literature review was published with the title "Graph Deep Learning: State of the Art and Challenges" as a journal paper in *IEEE Access* (Georgousis, Kenning, and Xie, 2021). The work described therein is supplemented here by additional or more recent work where the subject demands it. This literature review covers a swathe of subject-matter as is relevant as background or supporting work to the thesis of this doctoral work. The more detailed discussions naturally focus on the theory of deep learning in respect of this work.

The first section describes the development of the field of machine learning, from its conceptual origin of self-organising networks in the eighteenth century and the first perceptrons, through to the fall of perceptrons in the 1970s, leading finally to the renascence of neural networks in the second decade of the twenty-first century. The second section traces the development of neural networks into their modern incarnation as deep learning. With the advent of deep learning, the application of deep learning to irregular domains, notably graphs, is of central importance to this thesis. The final section elucidates some domain-specific problems, which serve as the motivation for the theoretical work of the thesis.

The author is indebted to several textbooks which aided in elucidating the theoretical development of machine learning. Any background on computational technology is owed to *A History of Computing Technology* by Williams (1997). The statistics, mathematics and background on machine learning owes a great deal to *Machine Learning and Pattern Recognition* by Bishop (2006), *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman (2009) and *Deep Learning* by Goodfellow, Bengio, and Courville (2016). The sections on graph theory draw on the work of Bollobás (1979). Background information on graph deep learning has also been supplemented by *Deep Learning on Graphs* by Ma and Tang (2021).

2.1 Statistical and Mathematical Background

First we will cover the mathematical and statistical background in order to establish the notation we use in this thesis.

2.1.1 Preliminary Remarks

The general formulation of an input–output mapping is the function $f : X \to Y$, where X is the input variable and Y is the output variable. X could be a continuous function that f maps to another continuous function Y. In machine learning the input and output are *discretised*. X, Y can be scalars x, y, vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^c$, or matrices $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{R}^{k \times c}$. A set of input vectors can also be constructed as a *design matrix* $[\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{k-1}]^\top = \boldsymbol{X} \in \mathbb{R}^{k \times c}$ and $\boldsymbol{x}_i$ is the ith feature vector of $\boldsymbol{X}$. We denote the selection of a row i of $\boldsymbol{X}$ as X_i or the selection of a column i as $X_{:,i} = X_i^\top$.

The feature matrix is a set of *observations* of some space X. Each column corresponds to a *feature* or *channel* and each row is an observation, also called a *row vector*. Each column of X is referred to as a *channel vector*, or more generally a *column vector*. The vector of values for each channel of a given observation is called a *feature vector*. The channel vectors are *linearly independent* if $\sum_{i=0}^{c} \alpha_i X_{:,i} = 0$ iff all $\alpha_i = 0$. When this property does not hold, the vectors are called *linearly dependent*. This means essentially that no column can be expressed as a linear combination of the others. Accordingly the channels can be seen as linearly independent, meaning they represent independent sources of information. This is important later when we consider *dimensionality reduction*.

In cases where $\boldsymbol{X}$ is square and, one may decompose $\boldsymbol{X}$ into its eigenvectors and eigenvalues:

$$\boldsymbol{X} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}, \tag{2.1}$$

where $\boldsymbol{Q}$ is the set of column eigenvectors, $\boldsymbol{Q}^{-1}$ is its inverse and $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues. The design matrix is more often rectangular, however, in which case singular-value decomposition (SVD) can be used to decompose the matrix:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top \tag{2.2}$$

where $\boldsymbol{U} \in \mathbb{R}^{k \times k}$ and $\boldsymbol{V} \in \mathbb{R}^{c \times c}$ are the left- and right-singular vectors of $\boldsymbol{X}$ and $\boldsymbol{\Sigma} \in \mathbb{R}^{k \times c}$ is a diagonal matrix of singular values of $\boldsymbol{X}$.

2.1.2 The Estimation of Data-generating Distributions

The function $f : X \to Y$ represents a mapping from input to output. It may also be viewed as a probability distribution $p(Y|X)$. The aim of machine learning is to approximate this *data-generating distribution* (Goodfellow, Bengio, and Courville, 2016, p. 108). The function $f(X)$ is thus approximating the data-generating distribution $p(Y|X)$. In learning algorithms the function is parametrised with a set of variables $\boldsymbol{\theta}$ such that it takes the form $f(X, \boldsymbol{\theta})$. Formally the approximation is

$$f(X, \boldsymbol{\theta}) = p(Y|X) + \epsilon \tag{2.3}$$

where ϵ is the error of estimation. The set of parameters of the function that best approximates $p(Y|X)$ is denoted $\boldsymbol{\theta}^*$.

The job of any algorithm is to make the error ϵ as small as possible by changing the values of the parameters $\boldsymbol{\theta}$. The whole range of possible values of $\boldsymbol{\theta}$ is the *search space* or *hypothesis space*; the more parameters, the larger the search space, the more complex the function can be. Approximation of the parameters is performed on a *training set* and validated against a *test set*. Ideally the error on one should match the other, but it is often the case that the errors are not proportionate. The first possible problem is that the function $f(X, \boldsymbol{\theta})$ *underfits* the training data, meaning it approximates the data-generating distribution with a large error. The second possible problem is that the function *overfits* the training data, which means that the error is low on the training set but incommensurately high on the test set. Both issues are related to the *capacity* of a model, which is partly connected to the number of parameters in the model. More parameters means a greater capacity which means a higher chance that the model will overfit. Too few parameters means a model is not able to estimate the data-distribution which means the function underfits the data-generating distribution. The challenge in learning algorithms is to balance these two problems.

One process of adjusting the parameters $\boldsymbol{\theta}$ to obtain an approximation of $p(Y|X)$ is termed *maximum likelihood estimation*. Formally the problem is

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J\left(\hat{Y}, Y\right), \tag{2.4}$$

where $J(-,-)$ is the *cost* or *objective function*, which measures the difference between the approximated output $\hat{Y} = f(X)$ and the target Y; in machine learning it is also called the *loss function*. The form of the function f is determined by one's knowledge of the distribution to be approximated. Whatever the function, the error, computed by the objective function $J(-,-)$ must be measured somehow in order that Eq. 2.4 can work. A simple example is the L_1 norm, which measures the sum of absolute differences between a predicted output $\hat{\boldsymbol{y}}$ and an target $\boldsymbol{y}$:

$$J(\hat{\boldsymbol{y}}, \boldsymbol{y}) = |\hat{\boldsymbol{y}} - \boldsymbol{y}|. \tag{2.5}$$

This function has a discontinuity at the y-intersect, however; a simple alternative is the L_2 norm, which is essentially the Euclidean distance between the predictions $\hat{\boldsymbol{y}}$ and an target $\boldsymbol{y}$:

$$J(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sqrt{|\hat{\boldsymbol{y}} - \boldsymbol{y}|^2} = |\hat{\boldsymbol{y}} - \boldsymbol{y}|_2^2. \tag{2.6}$$

This function has the properties of both being continuous around the y-intersect and penalising large errors. Such a function is called convex (Bishop, 2006, p. 56).

There is another aspect to consider when we think of multiple training samples. When we draw a set of samples X from the data-generating distribution, the samples have a *bias* and a *variance*, both of which affect the learning algorithm. The distance of the sample mean from the data-generating distribution's mean is called *bias*. The spread of the samples from the sample mean is measured as *variance*. A biased sample can lead to a biased model, which means the model systematically makes incorrect predictions owing to insufficient data or a poor selection of variables. A high variance on the other hand can lead to a model that makes unstable decisions that poorly generalise to data outside the training set.

2.1.3 Gradient Descent

A straightforward way of adjusting the parameters $\boldsymbol{\theta}$ during maximum likelihood estimation is simply to directly change the parameters according to whatever measure of error. But this does not account for the changes in the gradient as the parameters approach their optimal values. In other words, the parameter updates should shrink as the parameter approaches the correct value to avoid the learning from overshooting the optimum value and jumping around an optimal point because it uses constant weight updates. Using gradients to update weights gradually is called *gradient descent*.

The basic algorithm is to compute the average error across the whole dataset, measured using the objective function, then compute the gradients of the parameters with respect to the error, the magnitudes of which are used to update the model's parameters. The speed of change is modulated by a value called the *learning rate*, denoted η or λ, which is determined outside the learning algorithm. This is why it is important to have a convex loss function; it is then possible to descend a gradient slowly to a low point in the loss function, where the inverse gradient from all direction points to a minimum. This does not mean, however, that the functions of parameters with respect to an input are also convex. Indeed there may be many *local minima* that yield a suboptimal performance but which the learning algorithm can get stuck in, since it is a point at which there are only climbing gradients in each direction of change. The other risk is that the algorithm with respect to its parameters gets stuck at *saddle points*, where there is a local zero gradient. A zero gradient means that there is no value with respect to which the parameter can change. These are issues that are always in the background of machine learning algorithms; one must remain vigilant towards them.

2.2 Early Approaches to Machine Learning

In this section we will describe the purpose of machine learning, the elementary kinds of tasks and some elementary computational techniques to learn patterns in data.

2.2.1 Elementary Tasks in Machine Learning

Typically the objective of a machine learning algorithm is to discover the set of mathematical operations that reliably maps a set of n inputs $X = \{x_0, x_1, \ldots, x_{n-1}\}$ to a set of d outputs Y. The choice of algorithm reflects the nature of the task, for the nature of the algorithm imposes an *a priori* constraint on the solution that may be sought.

Tasks generally fall into one of the following categories:

- **Regression**, the prediction of a target value Y from a set of inputs. For example, predictions of economic growth as a percentage of gross domestic product or the price of stocks from a collection of economic indicators.

- **Classification**, the assignment of a category labels to observations. For example, the identification of plant species from sepal and petal lengths and widths (Fisher, 1936) or the diagnosis of disease from health indicators.

- **Clustering**, the grouping of observations according to their providence, e.g., the grouping of molecules according to the similarity of their structures and/or properties.

These categories are not exhaustive, but such labels help us to determine what form a learning algorithm should take and what measures we require.

A further distinction is drawn between *supervised* and *unsupervised* models. A supervised model is presented with labels at training time and learns to map input to an explicit output. Classification and regression tasks are mainly supervised tasks. On the other hand, unsupervised models are not fed any targets, but rather are permitted to determine whatever pattern is necessary to minimise a measurable outcome that is not tied to a learning objective. We describe examples of supervised and unsupervised models in the next section.

A machine learning algorithm is programmed to optimise an *objective function* or *loss function*. These functions represent the learning task for the algorithm, measuring the closeness of the model's output to the answer we expect. Ideally the *loss* should be zero: the larger the loss, the more incorrect our model. Hence the loss can also be considered as a measure of error.

The output of whatever model we choose is compared to our desired output, the *ground-truth*, what we consider to be correct. If the outputs are very wrong, the model is altered by adjusting its parameters. The parameters are scalar values, so they may be increased or decreased. By calculating the derivative of the loss with respect to each parameter, we know the direction in which a parameter should be changed, given by the sign of the derivative, as well as the magnitude of its change.

The space of the parameters of a model is termed the *hypothesis space* or *search space*. Ideally the search space is concave with respect to the loss, which means there is a single, unique, optimal solution. There is seldom a problem in which this is true in practice, where search spaces contain many local minima, many of which of suboptimal. The model may become stuck in any of these minima, and it is difficult if not impossible to know if it's locally or globally optimal: the complexity of the search space does not allow it. There are ways to reduce the chances of falling into local minima, but there is no general way to avoid it completely.

2.2.2 Elementary Techniques in Machine Learning

The elementary machine learning techniques in this section are described as parametric or non-parametric. Parametric models make an assumption about the distribution of data, from which point it is necessary only to estimate the parameters that best fit the data to the distribution. On the one hand, the advantage of parametric models is that inference is cheap and the models tend to be small. On the other hand, the assumed distribution will not be valid for all data.

By contrast, non-parametric models make no assumption about the distribution, but rely on a large training set. The advantage that a non-parametric model is flexible to the data distribution is offset by the computational expense of inference, since it works by reference to the training data instead of a small set of parameters.

2.2.2.1 Linear Regression

Linear regression is a parametric method. In its simplest form, linear regression is a mapping from a single input x to a single output y, with a single coefficient θ and a single

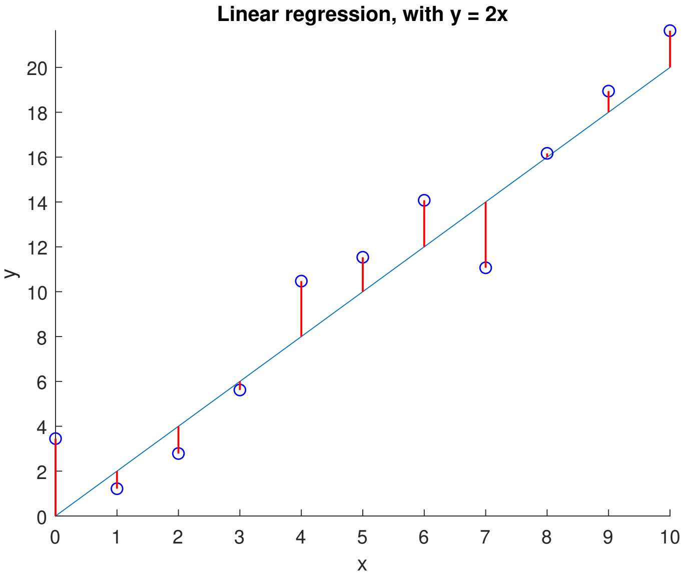


Figure 2.1: Linear regression can be used to separate two classes during classification. In this figure the line $y = 2x$ is plotted. The points above the line can be considered the positive class and the points below the negative class. The figure has been adapted from Kenning (2019).

bias b. In the simplest form, the linear regression has the following formula:

$$y = \theta x + b.$$

Alternatively one may map from multiple inputs $\boldsymbol{x} = [x_0, x_1, \ldots, x_{p-1}]^\top \in \mathbb{R}^p$ with p features to multiple outputs $\boldsymbol{y} = [y_0, y_1, \ldots, y_q]^\top \in \mathbb{R}^q$. The set of parameters is thereby a matrix $\boldsymbol{\Theta} \in \mathbb{R}^{p \times q}$. Including the vector of biases $\boldsymbol{b} \in \mathbb{R}^q$, one per output, the formula is:

$$\boldsymbol{y} = \boldsymbol{\Theta x} + \boldsymbol{b}. \tag{2.7}$$

Non-linear transformations may be applied to the inputs $\boldsymbol{x}$ before they are linearly combined. One example is the radial basis function.

Linear regression is more conventionally used to regress on real values. It can however be adapted to classification, for which a decision boundary is required. Any prediction of y above that line belongs to a positive class; otherwise a negative class. (See Fig. 2.1.)

Logistic regression is a variation on linear regression, where the output is bounded on the interval $(0, 1)$. It has the form

$$\boldsymbol{y} = \sigma\left(\boldsymbol{\Theta x} + \boldsymbol{b}\right) \tag{2.8}$$

where $\sigma(-)$ is the sigmoid function

$$\sigma(\boldsymbol{z}) = \frac{1}{1+e^{-\boldsymbol{z}}} = \frac{e^{\boldsymbol{z}}}{e^{\boldsymbol{z}}+1}\,. \tag{2.9}$$

Let $\boldsymbol{z} = \boldsymbol{\Theta x} + \boldsymbol{b}$. Logistic regression is used for binary classification tasks or decision tasks. The value $\sigma(\boldsymbol{z}) = 0.5$ is the *decision boundary*. A value above the decision boundary is a positive decision; below is a negative decision.

2.2.2.2 k Nearest Neighbours

The k nearest neighbours (KNN) algorithm requires a large training set against which new samples can be compared. The training set usually consists of a set of observations with their own class labels. The inferential set is a set of observations without class labels. The task is to infer the class labels of the inferential set from the class labels of the training set.

The sole decision criterion is the distance measurement. Most simply one measures the Euclidean distance between an inference observation x and every member of the training set $\mathbb{X}$. The closest k training samples determine the label $\hat{y}$ of the observation. If $k = 1$ then the observation x is classified with the same label as the closest sample, $\hat{y} = y_i$, where $i = \arg\min_i ||\mathbb{X}_i - x||_2^2$.

When $k > 1$, what happens depends on the kind of task. If KNN is used for regression, then $\hat{y}$ is an average of the k closest samples. For classification on the other hand, in the simplest case, the modal class among the nearest examples is the assigned class. Of course other refinements are possible.

KNN is only evaluated against other training samples; there are no trained parameters. As such KNN is a non-parametric approach. Inference is consequently very slow, since it is necessary to run a full evaluation of the training set—unless of course some search optimisation is employed. It is also unsupervised since no explicit training target is supplied to the algorithm.

Non-linear projection of the data before KNN is applied is also possible, similar to what is possible with linear regression.

2.2.2.3 k Means

The k means algorithm may be used to cluster a set of data, but it can be further used to classify new datapoints. It is crucially necessary that $k > 1$, for otherwise the algorithm is complete before it has started. The datapoints of a training set $\mathbb{X}$ is first assigned randomly to k different classes. The average position of each class, called the *centroid*, is computed.

The datapoints are reassigned a class according to the nearest centroid. The algorithm thus reiterates until the centroids no longer shift or shift below a threshold.

Like KNN, the sole decision criterion for class membership is the distance measure. Typically, again, this is the Euclidean distance. Other distance measures may also be used, of course. As mentioned above, k means may be used to cluster training data. Like KNN it may also be applied to classification. The decision boundaries of the k means algorithm are more jagged. The algorithm is still non-parametric, but it is more efficient than KNN since one need only compare new points to the small number of centroids.

2.2.2.4 Gaussian Mixture Model

The Gaussian mixture model (GMM) is superficially similar to the k means algorithm. The GMM however requires a labelled training set. Each data class is assumed to have a normal/Gaussian form. Each class consequently has its own Gaussian component with a vector of means $\boldsymbol{\mu}$ and a covariance matrix $\boldsymbol{\Sigma}$. The distributions are moreover weighted according to a *prior probability*, which is the relative frequency of each data class. The *posterior probability* tells us the likelihood of a given class given a sample. Although a Gaussian or normal distribution is parametric, in that it has a mean and a standard deviation, a mixture of Gaussian models is considered non-parametric—or at least semi-parametric.

2.2.2.5 Support Vector Machine

The support-vector machine (SVM) is outwardly very similar to linear regression, specifically because it uses a linear function similar to Eq. 2.7. The SVM is only capable of outputting class labels, however. A single SVM can only classify according to positive and negative classes. In the simplest case, the SVM learns a set of parameters $\boldsymbol{\theta}$ for the following formula:

$$y = \boldsymbol{\theta}^\top \boldsymbol{x} + b. \tag{2.10}$$

An observation $\boldsymbol{x}$ is predicted as a member of the positive class when $y > 0$; when $y < 0$ then $\boldsymbol{x}$ is predicted a member of the negative class. The parameters $\boldsymbol{\theta}$ are optimised to find a line that maximally separates the two classes. The separation is measured using the sum of squared differences, which supplies the loss value for the optimisation procedure, a process called the least squares algorithm.

Normally the SVM assumes that the data is linearly separable. A feature function $\phi(\boldsymbol{x})$ may be defined that maps non-linear features to a smaller space, which permits a certain innovation called the *kernel trick*. This actually allows us to rewrite Eq. 2.10 in terms of the dot-product of the sample $\boldsymbol{x}$ and m training samples $\boldsymbol{x}^{(i)}$

$$\boldsymbol{\theta}^\top \boldsymbol{x} + b = b + \sum_{i=1}^{m} \theta_i \boldsymbol{x}^\top \boldsymbol{x}^{(i)}, \tag{2.11}$$

By using a kernel $\phi(\boldsymbol{x})$, thanks to the kernel trick, we can write it as follows for some kernel $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)}) = k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$:

$$f(\boldsymbol{x}) = b + \sum_{i=1}^{m} \theta_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}). \tag{2.12}$$

The advantage is that the function f, although non-linear with respect to $\boldsymbol{x}$, is linear with respect to $\phi(\boldsymbol{x})$ and separately with respect to $\boldsymbol{\theta}$. This is important as it allows one to use a convex optimisation procedure that is guaranteed to converge efficiently (Goodfellow, Bengio, and Courville, 2016, p. 138) especially when certain loss functions are applied to linear models (Hastie, Tibshirani, and Friedman, 2009, p. 192). Accordingly, the optimisation procedure may consider $\phi(\boldsymbol{x})$ to be fixed and only optimise the coefficients $\boldsymbol{\theta}$. It may be thereby said that the optimisation is independent of the feature function $\phi(\boldsymbol{x})$. This is in fact the standard form of the SVM; the m features $\boldsymbol{x}^{(i)}$ are termed the *support vectors*.

The SVM is of course a supervised method. As there is a linear combination of features weighted by learned coefficients, it is considered parametric. Yet the kernel trick makes the SVM a very slow method, since it requires an evaluation of a single point against the whole training dataset.

2.2.2.6 Decision Trees

A decision tree consists of a series of decision points, represented as *nodes*, which lead to further decision points and eventually a decision on a class or regression value, which is termed a *leaf*. Each leaf requires *at least* one training example. For a single feature x, the decision tree would have one level and one decision: either $x < \theta$ or $x \geq \theta$, which would decide whether it belongs to one class or another. This split into two options is termed a *binary split*. Theoretically further splits could be made, but this is generally inadvisable as it leaves too little data at lower levels. Generally, if multiway splits can be achieved via multiple binary splits, the latter is to be preferred (Hastie, Tibshirani, and Friedman, 2009, p. 311). If each level is constrained to one features, and hence one axis, it can make decisions on the basis of further features. Theoretically however

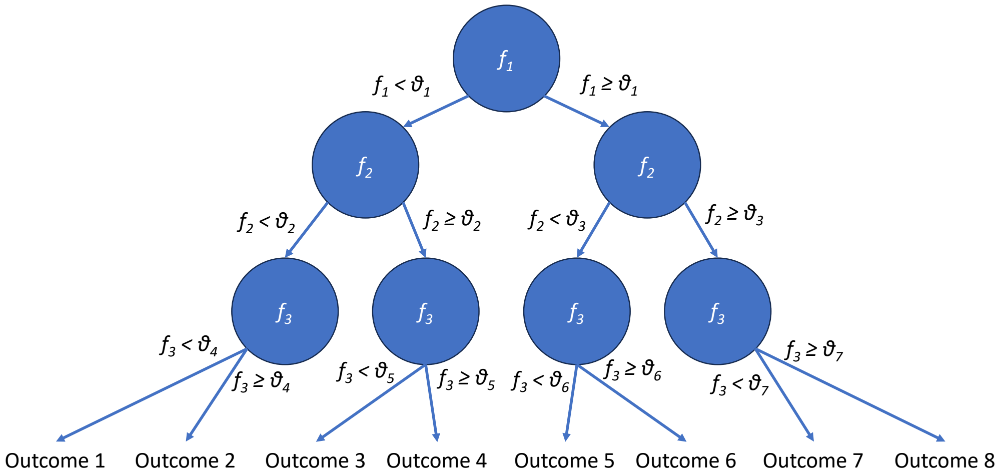


Figure 2.2: A simple decision tree with three features, f_1, f_2, f_3. The passage from the first level to the second level depends on the first decision boundary θ_1; the passage between the second to the third levels depends on the second decision boundaries θ_2, θ_3; the passage between the third level and the decision depends in turn on the decision boundaries $\theta_4, \theta_5, \theta_6, \theta_7$.

multiple, disparate leaves in the tree can lead to the same classification decision. Figure 2.2 illustrates an example of a decision tree.

If a decision tree is allowed to grow indefinitely, then it is considered non-parametric. A decision tree becomes parametric if its growth is constrained according to some limits on, say, depth, number of leaves, the minimum number of samples at each leaf, *etc.*

Decision trees are in fact quite brittle and struggle to solve problems where logistic regression would have no issue (Goodfellow, Bengio, and Courville, 2016, p. 142). Although decision trees have a low bias, they have high variance (Hastie, Tibshirani, and Friedman, 2009, p. 587–8). Consequently decision trees tend to overfit on the training data.

2.2.2.7 Random Forests

A random forest (RF) consists of an ensemble of decision trees. Each tree is constructed from a set of samples drawn randomly from the training set, a process called *bagging*. At each terminal node of the tree, the algorithm is only allowed to use a subset of features in deciding a split at each level. The trees thus constituted resemble each other less and are consequently said to be *de-correlated*. The prediction for a new observation is determined by passing it to every tree in the ensemble. For classification, the modal class is the predicted class. In regression problems, the average of the predictions is the predicted value.

The bias of the ensemble is the same as any individual tree. Although they are not independent, the decision trees of an RF are identically distributed. Accordingly, as the number of trees in the ensemble increases, the variance decreases, leading to a more robust model than an individual tree (Hastie, Tibshirani, and Friedman, 2009, p. 588).

The construction of the decision trees of course implies that RFs are strictly supervised learners. As the number of decision trees and the various parameters previously mentioned in Section 2.2.2.6 are specified at the outset, it is of course a parametric method, too. The cost of inference is curtailed by a simple evaluation of an observation against each tree rather than against the whole training set. The computational complexity of a single decision tree is $\mathcal{O}(\log_2(m))$, where k is the number of datapoints in each bag of samples. With b decision trees in the RF, the computational complexity is $\mathcal{O}(b \log_2(k))$.

2.2.2.8 Principal Component Analysis

More often than not the input features to an algorithm are not linearly independent (Section 2.1.1). Where there are large numbers of features, which is dangerous owing to the curse of dimensionality (Section 2.2.2.9), or a high degree of redundancy in the features, it is helpful to simplify the input data by retaining dimensions in the input space that correspond to the greatest variance in the input.

Principal component analysis (PCA) is one such technique for *dimensionality reduction*. The input design matrix is first decomposed using SVD (Eq. 2.2). The diagonal entries of the singular-value matrix $\mathbf{\Sigma}$ describe the variance accounted for by each of the orthogonal right-singular vectors. Usually the first k right-singular vectors, which account for a percentage of the variance in the data indicated by the same of their corresponding values in diagonal matrix of singular values, form the *principal subspace*. This constitutes a set of column vectors that are used to project the input data into a smaller subspace, hence reducing the dimensionality.

2.2.2.9 The Curse of Dimensionality

Many machine learning algorithms run into tremendous, insuperable difficulties at high dimensions (Hastie, Tibshirani, and Friedman, 2009, *passim*). As we increase the quantity of variables in the data, the number of their configurations increases *exponentially* (Goodfellow, Bengio, and Courville, 2016, p. 151). Essentially it is all well and good to devise a learning algorithm that can seek the nearest datapoint, as in KNN. Nonetheless, if we supply a model with ever greater number of variables, eventually one reaches the point where the number of configurations of those variables exceeds the number of training examples. At such a point it is possible that one witnesses an example for which there is no example in the training set. The assumption that new observations resemble old ones hence becomes tenuous. The inductive principle underlying the learning algorithms too becomes tenuous.

2.2.2.10 No Free Lunch

An addition to the theoretical troubles of machine learning is the no free lunch theorem, presented by Wolpert (1996). In brief, the theorem states that, averaged across all data-generating distributions, every classification algorithm has the same average error rate. The corollary is that no algorithm is universally better than any other (Goodfellow, Bengio, and Courville, 2016, p. 113). It does not however mean that it is impossible to find an algorithm that is optimal for a given problem. It warrants a certain scepticism to the goal of finding a universally applicable algorithm for all learning problems, though. Indeed, in practice, a consideration of what is practical and expedient to a specific purpose is wholly appropriate. The theorem, in combination with the curse of dimensionality, serves as a warning against the imprudence of entrusting an algorithm wholly with a task, especially tasks that concern the wellbeing of human beings. This concern is connected to the following issue.

2.2.2.11 The Insufficiency of Data

A comprehensive discussion of the epistemic limitations of science is undoubtedly beyond the scope of this discussion. Nevertheless it is necessary to discuss the representational limitations of natural phenomena since it bears keenly on a popular theoretical justification for deep learning over conventional methods. The following considerations broadly draw on philosophical traditions in western philosophy (Russell, 2010).

On the elementary questions of the extent to which reality is knowable (which really is beyond the scope of this), it is customary in science to adopt a positivistic or empirical attitude to things. From this perspective a considerate but by no means conclusive discussion of this issue was published by Schrödinger (1992). More generally it regards the epistemological gap between an object (e.g., Kant's thing-in-itself, *noumenon*) and its perception or representation (the thing-as-perceived, *phenomenon*). Kant asserts that the noumenon is wholly inaccessible; a tradition of philosophers going back to Plato assert that it is accessible. We make no judgement either way; the point is that it is undecided (and perhaps undecidable). It is noteworthy that the word *phenomenon* originally had the specific connotation of what is *mentally* perceived by a human subject, what is subjective to the perceiver.

A more relevant and proportionate concern is the extent to which what *is* knowable may be *faithfully* and *accurately* representable not merely mathematically but computationally. Partly it is a question of technological boundaries, which are perpetually shifting; e.g., a machine's ability to represent computational complexity. Another part is the appropriateness of numerical or symbolic representation in itself, for quantophrenia can quickly overtake a scientist's mind against better judgement.

Scientific theories seek to explain natural phenomena using simplified but underdetermining models. It is not necessary that a scientific model wholly and flawlessly explains a phenomenon; it is only necessary that the model *more completely* captures the dynamics or development of a natural system. A scientific theory should reliably and accurately explain other relevant phenomena with a tolerable level of error.

Our measurements of phenomena are always peripheral to the things-in-themselves; we are always using scientific apparatuses to comprehend and in turn explain our universe. It is too cynical to conclude that the implied distance between objects and our perceptions means *all our perceptions* are therefore fundamentally flawed and unreliable. In any case, there is a distance. In conventional learning algorithms, where the threat of the curse of dimensionality is ever looming, the necessity of scientific apparatuses and the measurements therefrom is clear and necessary. It is an expedient that allows us to make tolerably reliable predictions about our universe.

Nonetheless there is a sense in which these measurements summarise too much in the sense that they miss crucial patterns in information that might be discovered by other means. To draw information from what is termed *raw information* is considered an improvement over summarising and thus simplifying measurements. The specific selection of the measurements adds an additional source of bias to the algorithm. Much better, the thought goes, is to allow the algorithm to learn representations automatically and directly from the raw information. In spite of the sense of these points, we aver that the perceptual distance has not disappeared, even if a machine is allowed to automatically learn from raw information.

Automatic representation learning is at the heart of a set of learning techniques termed *deep learning*. At the heart of deep learning is a structure called a neural network. That is the subject of the next section.

2.3 Neural Approaches to Machine Learning

Neural-based approaches, approaches that use some variation on a neural network, characterise the vast majority of today's most advanced algorithms. Central to the modern approaches is the more specialised convolutional neural network (CNN), which is discussed in Section 2.4. In this section we discuss the early rudimentary forms of perceptrons and later more elaborate neural architectures. The models were largely developed before the modern renascence of neural networks that was initiated in the second decade of the twenty-first century.

2.3.1 The Perceptron

The perceptron is the predecessor of the modern neural network. The form of the neurones that constitute the perceptron is derived from the McCulloch–Pitts logical neurone (McCulloch and Pitts, 1943), a simplification of the real neurone. A neurone is a cell body with dendritic branches as chemical inputs and axon branches as chemical outputs. The chemical signal received along the dendrites accumulates until it exceeds a threshold, at which point the neurone fires, releasing neurotransmitter hormones down the axon branches to neighbouring neurones. The activity of a logical neurone on the other hand is an all-or-nothing response. At any point in time, all that is required to activate a logical neurone is that a certain number of its synapses are activated, whereas inhibitory synapses absolutely prevent the excitation of the neurone. The responses of a perceptron are not all-or-nothing—unless one builds in that property. Moreover, whether the input to a neurone is excitatory or inhibitory is a question of relative degree. The neurones of the perceptron with which we are acquainted is for this reason a relaxation of the McCulloch–Pitts logical neurone.

The perceptron, as introduced by Rosenblatt (1958), is a set of neurones, each of which takes a vector of input vectors $\boldsymbol{x} \in \mathbb{R}^c$ which are first transformed by a fixed linear transformation $\phi(\boldsymbol{x}) \in \mathbb{R}^d$. The transformed input features are bound to a set of trainable weights $\boldsymbol{\omega} \in \mathbb{R}^d$ and bias term $b \in \mathbb{R}$ is also added. Conventionally however the zeroth position of the weight vector ω_0 is the bias term, and the corresponding entry in the input vector x_0 is 1. (Later this simplifies the description of the gradient.) Therefore the formula for a single neurone is written

$$z = y(\boldsymbol{x}, \boldsymbol{\omega}) = \boldsymbol{\omega}^\top \phi(\boldsymbol{x}). \tag{2.13}$$

The result z is passed through a step function f, essentially classifying the input as positive or negative (see Fig. 2.3 for an illustration of the perceptron):

$$f(z) = \begin{cases} +1 & z \geq 0, \\ -1 & z < 0. \end{cases} \tag{2.14}$$

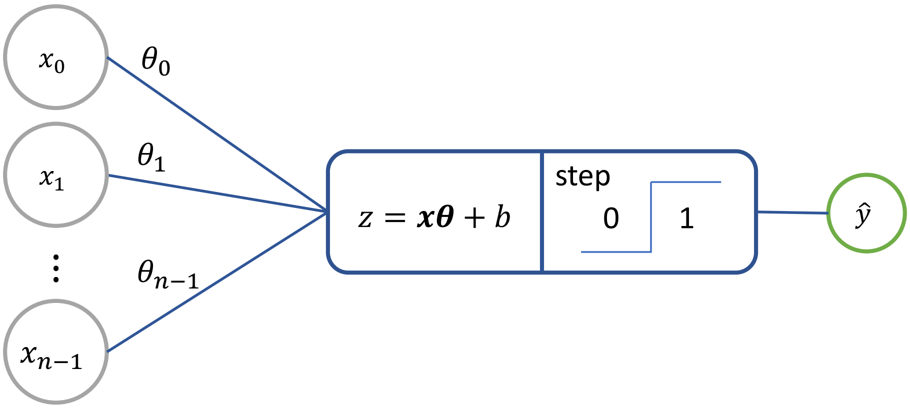


Figure 2.3: The perceptron receives inputs $x_0, \ldots, x_{n-1}$ weighted by a set of parameters $\theta_0, \ldots, \theta_{n-1}$, to which a bias term b is added. The weighted sum gives a prediction $\hat{y}$.

The target value for the ith sample is therefore $t_i = \{-1, +1\}$ rather than $\{0, 1\}$, as is conventional in modern machine learning. The weights $\boldsymbol{\omega}$ are trained on a target by stochastic gradient descent with respect to the classification error, measured by the loss function. Yet the loss function is not merely the number of incorrectly classified samples since the function would be discontinuous (Bishop, 2006, p. 193). Instead a *perceptron criterion* E is used that sums the z for all misclassified samples (correctly classified samples have a zero error):

$$E(\boldsymbol{\omega}) = -\sum_{i \in \mathcal{M}} \boldsymbol{\omega}^\top \phi(\boldsymbol{x}) t_i \tag{2.15}$$

where $\mathcal{M}$ is the set of indices of misclassified examples.

The weight update at time τ from the weights and error at $\tau - 1$ is therefore

$$\boldsymbol{\omega}^{(\tau)} = \boldsymbol{\omega}^{(\tau-1)} - \eta \nabla E(\boldsymbol{\omega}) = \boldsymbol{\omega}^{(\tau-1)} - \eta \phi(\boldsymbol{x}) t_i, \tag{2.16}$$

where η is the learning rate. According to Bishop (2006, p. 194), the learning rate may be set $\eta = 1$ since multiplying the perceptron function $y(\boldsymbol{x}, \boldsymbol{\omega})$ by a constant is unchanged in the output $f(z)$.

The first perceptron to be implemented was the Mark I Perceptron, developed on an IBM 704 computer by Frank Rosenblatt in 1957 at Cornell Aeronautical Laboratory. It was programmed with the world's first machine-learning algorithm, the perceptron learning algorithm (Smith, 2019, p. 195). The Mark I consisted of a set of 400 photoreceptors with wires literally randomly connected to a set of neurones (Bishop, 2006, p. 196).

Potentiometers controlled the contribution of each photoreceptor to the output, the predicted category of the input. Weight updates were performed by motors according to the equation described in Eq. 2.16. The Mark I was trained on illuminated shapes, such as letters of the alphabet.

The perceptron is in fact a linear model and is therefore limited to representing linear functions and therefore limited to distinguishing linearly separable patterns. Minsky and Papert (1972) demonstrated, among the other limitations of single-layer perceptrons, that a single-layer perceptron is incapable of representing a XOR function. Their criticisms were directed at single-layer perceptron and a conjecture that the same problems obtained in other networks was widely misinterpreted as damning on all neural networks that it led at length to a substantial decline in research funding for neural networks (Bishop, 2006, p. 193; Goodfellow, Bengio, and Courville, 2016, p. 218). As is later demonstrated (Cybenko, 1989; Hornik, Stinchcombe, and White, 1989), multi-layer perceptrons (MLPs), obeying certain conditions, do not suffer from these limitations.

2.3.2 Universal Approximation Theorem

The universal approximation theorem states that any Borel measurable function from one finite-dimensional space to another finite-dimensional space can approximate any continuous function to an arbitrary degree of error. Since any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable, it means that a neural network can approximate any function (Goodfellow, Bengio, and Courville, 2016, p.192). The degree of error is dictated by the size of the neural network—in its breadth (the quantity of neurones in a layer) or depth (the number of successive perceptron layers) (Hastie, Tibshirani, and Friedman, 2009, p. 390). The model could contain a single hidden layer (plus a layer of linear outputs), but the layer would be infeasibly large and fail to learn and generalise correctly (Goodfellow, Bengio, and Courville, 2016, p. 193). The alternative is a deeper neural network with more hidden layers, which reduces the requisite number of neurones in the hidden layers.

The neural network is therefore a *universal approximator* (Bishop, 2006, p. 230). A two-layer perceptron for example can approximate any continuous function on a compact input domain (Bishop, 2006, p. 230) provided that the first hidden layer uses a non-linear “squashed” activation function on its neurones, e.g., a tanh or sigmoid function (Goodfellow, Bengio, and Courville, 2016, p. 192).

There are several added difficulties however, both practical and theoretical, described by Goodfellow, Bengio, and Courville (2016, p. 193). Firstly, a multi-layer network may be theoretically able to represent a function, but it does not guarantee that the network *will* learn the function. It may be impossible to find the parameters or the network will simply choose the wrong function because it overfits the training data. Secondly and most crucially there is no universal procedure for estimating parameters from a training set that will generalise to samples that are not in the training set.

All that being said, feed-forward neural networks are powerful tools in the approximation of functions. A deep model is particularly important for statistical reasons and is to be favoured over a statistical model that makes assumptions of the specific statistical form of the data, for it encodes a very general belief that the function we want to learn is a composition of smaller, simple functions (Goodfellow, Bengio, and Courville, 2016, p. 195).

2.3.3 Feed-forward Neural Networks

The *feed-forward neural network* is identical with an MLP. A feed-forward neural network is rarely less than two layers deep. What counts as a layer is in contention; for present purposes a layer is a linear transformation followed by an activation. Feed-forward neural networks are often described as *fully connected*. That is, the neurones in a layer fully connected to every neurone in the next layer. The first and last layers are referred to as the *input* and *output layers*. The input layer is identical with the input features. The layers between are termed *hidden layers*.

The input features $\boldsymbol{x} \in \mathbb{R}^c$ are fed to the first hidden layer of p neurones, where the ith neurone has a matrix of weights $\boldsymbol{\Omega}_{1,i} \in \mathbb{R}^{c \times d}$ (as before the bias is implicitly included at the zeroth position). The output of the ith neurone in the first hidden layer is therefore

$$\boldsymbol{z}_i^{(1)} = \sigma\left(\boldsymbol{\Omega}_{1,i}^\top \boldsymbol{x}\right) \tag{2.17}$$

where the superscript-(1) represents the index of the first layer, σ is a non-linear function and $\boldsymbol{z}^{(l)} \in \mathbb{R}^d$. Every subsequent layer l may then be defined in terms of the previous layer $l - 1$'s outputs $\boldsymbol{z}^{(l-1)}$:

$$\boldsymbol{z}_i^{(l)} = \sigma\left(\boldsymbol{\Omega}_{l,i}^\top \boldsymbol{z}^{(l-1)}\right) \tag{2.18}$$

The output after L layers is yielded as $\hat{y} = z^{(L)} \in \mathbb{R}^d$, where d is the desired number of features in the output. Depending on the task, the output could be bounded or normalised in various ways.

In supervised tasks, the error of the output $\hat{y}$ is measured against a target y with a *loss function*, often referred to simply as the *loss*. A common supervised loss has already been described, namely the L_2 norm or Euclidean distance:

$$L(\hat{y}, y) = ||\hat{y} - y||_2^2. \tag{2.19}$$

Later we discuss how the loss is used to adjust the parameters of the model. Beforehand, it is worth noting something about the structure of the neural network.

In some tasks our target is not a single value z but actually multiple values. In multi-class problems, there are c outputs for the c classes. First the output is normalised with a function called *softmax*, which bounds the outputs to the interval $(0, 1)$. If the output vector is a vector $\boldsymbol{a} \in \mathbb{R}^o$, then for the class output at a_i

$$\hat{\boldsymbol{y}}_i = \text{softmax}(\boldsymbol{a})_i = \frac{e^{a_i}}{\sum_{j=1}^{c} e^{a_j}} \tag{2.20}$$

for the ith class prediction. Softmax is distinct from simply dividing the outputs by the largest value because it emphasises large values and minimises small values. The labels for a multi-class problem are encoded as *one-hot vectors*. The labels of a c-class task would therefore consist of a vector $\boldsymbol{y}$ where all zero entries except the ith entry which is set to 1 for to represent a positive for the ith class. The softmax'd predictions $\hat{\boldsymbol{y}}$ and the true labels $\boldsymbol{y}$ are then compared using cross-entropy, also called Kullback–Leibler divergence $D_{\text{KL}}(\hat{\boldsymbol{y}} \parallel \boldsymbol{y})$, yielding a single value for the loss:

$$H(\hat{\boldsymbol{y}}, \boldsymbol{y}) = D_{\text{KL}}(\hat{\boldsymbol{y}} \parallel \boldsymbol{y}) = -\sum_{i=0}^{c-1} \hat{y}_i \log y_i. \tag{2.21}$$

Effectively one could describe each of the neural network as a linear combination of non-linear basis functions (Bishop, 2006, p. 227). According to this interpretation it is easy to see where the power of approximation lies. In Section 2.2.2 we described the use of the kernel trick in the SVM that enables the algorithm to learn on non-linear basis functions via linear combination. The SVM works independently of those basis functions. By contrast, a neural network actually learns the parameters of the basis functions—and moreover it is linear to those parameters.

The non-linearity is achieved by the use of non-linear activation functions, or as we refer to them in Section 2.3.2, "squashed" functions. It was once common to use the sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.22}$$

or tanh

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{2.23}$$

activation functions, but these functions carry the risk of saturation (Goodfellow, Bengio, and Courville, 2016, p. 189). As a result the use of these activation functions is discouraged except in the final layer of a neural network, where the optimiser can easily correct for saturation.

In multi-class problems, where there are multiple classes and therefore multiple outputs $\boldsymbol{z} \in d$, softmax is used to constrain the outputs to the interval $[0, 1)$. For the ith class output it has the form

$$\hat{y}_i = \text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{d} e^{z_j}} \tag{2.24}$$

The sum of the outputs is therefore $\sum_{i=0}^{d} \boldsymbol{y}_i = 1$, which has the form of a probability distribution. The outputs therefore reflect the *class probabilities* for a multi-class output.

2.3.4 Backpropagation through a Neural Network

Backpropagation, or simply *backprop*, is the process of discovering the gradients present in the forward propagation of signals through a neural network. The cost from the loss function is propagated backwards from the end of the network through all the weights of the network to discover their gradients (Goodfellow, Bengio, and Courville, 2016, p. 197).

The chief mathematical tool behind backpropagation is the chain rule of calculus. The description here is adapted from Goodfellow, Bengio, and Courville (2016, p. 197). Suppose we have an input x and two functions f and g such that $y = f(x)$ and $z = g(f(x)) = g(y)$. The chain rule states that

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}. \tag{2.25}$$

This equation may be generalised to vectors. Suppose we have $\boldsymbol{x} \in \mathbb{R}^c, \boldsymbol{y} \in \mathbb{R}^d$, $f : \mathbb{R}^c \to \mathbb{R}^d$ and $g : \mathbb{R}^d \to \mathbb{R}$. The composition of the functions then looks like $\boldsymbol{y} = f(\boldsymbol{x})$ and $z = g(\boldsymbol{y})$. The chain rule for vectors is

$$\frac{\partial z}{\partial x_i} = \sum_i \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x_i}, \tag{2.26}$$

which in vector notation may be written

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{2.27}$$

where $\frac{\partial y}{\partial x}$ is the $d \times c$ Jacobian matrix of g. Hence we can see that obtaining the gradient of a variable $\boldsymbol{x}$ is a simple multiplication of the Jacobian matrix $\frac{\partial y}{\partial x}$ by the gradient $\nabla_{\boldsymbol{y}} z$. The backpropagation is the progressive computation of these gradients for each operation in the neural network.

The value z in a neural network is the loss. The size of the weights update is controlled by a learning rate η which scales the loss. It is termed a *hyperparameter* because it controls the algorithm that trains the model parameters. Models can be sensitive to the choice of η and it is common that in training a model several learning rates differing in magnitude are compared experimentally to determine the most appropriate learning rate.

2.3.5 Mini-batch Gradient Descent

We have described gradient descent in Section 2.1.3. Normally gradient descent, specifically batch gradient descent, implies a full evaluation of the error on the training set, with which the parameters of the model are optimised. This is computationally very expensive. Additionally there is a high degree of redundancy in training sets, especially in large datasets, that means it is superfluous to evaluate every sample. The extreme alternative is to evaluate the model and update the parameters one sample at a time, a process sometimes called stochastic gradient descent, as we have seen. The disadvantage is of course the length of time it takes to iterate through a large training set one by one. The other disadvantage is the risk of knocking a model's training of kilter by one anomalous or highly unusual example, to protect against which would require a very low learning rate, which would further slow down learning.

A intermediate approach is *mini-batch gradient descent*, sometimes referred to as simple *stochastic* gradient descent (Goodfellow, Bengio, and Courville, 2016, p. 272). For the model training, the training set is subsampled into smaller batches with an arbitrary number of samples. These smaller batches are called *mini-batches*, but also confusingly called *batches* in the literature. Each subsample is used to update the model's weights. A full cycle through every mini-batch is termed an *epoch*. By randomly sampling a subset, one reduces the computational cost of weight updates, but moreover learns on a subset that is distributed identically with the original training set, and therefore lead to unbiased estimates of the gradient, with a variance inversely proportional to the size of the mini-batch.

There are concomitant questions here regarding the appropriate size of the mini-batches, termed the *batch size*. The larger the batch size, the smaller its variance. Larger batch sizes mean fewer iterations per epoch which means faster training. Any edge cases are also subsumed in a larger set of more normal samples. Smaller batch sizes expose edge cases to the model, however, and potentially encourage a model to incorporate edge cases in its approximation of the data-generating distribution. Small batch sizes can induce a small regularising effect on the model, too (Goodfellow, Bengio, and Courville, 2016, p. 272). How large the batch size should be is ultimately a matter of experimentation. One rule of thumb is that batch sizes should be set in powers of 2, especially when using graphics processing units (GPUs), since the GPU may avail itself of these sizes in setting up parallelised operations owing to its architecture.

It is also common that the mini-batches are shuffled. Random selection is one means of shuffling the dataset. Further randomisation is achieved by resampling the mini-batches every epoch. When a model has a high enough capacity, there is a risk that it will overfit on specific configurations of samples. Varying the composition of the mini-batches combats this fact. On the other hand, the risk diminishes with very large datasets even if they are shuffled just once (Goodfellow, Bengio, and Courville, 2016, p. 273).

Finally there is the matter of data imbalance to consider. Data imbalance arises when the quantity of one class is far in excess of the other class. This is very common in modelling medical phenomena where there are far more healthy patients than unhealthy patients. There are two basic remedies in this situation: (1) the sampling of the training set is skewed proportionately to favour the smaller class; or (2) the losses computed on the smaller class are scaled by amount proportionate to its infrequency.

2.3.6 Technological Limitations and Stagnation

The neural network is said to have undergone three waves of popularity (Goodfellow, Bengio, and Courville, 2016, pp. 14–18). The first wave crested with the perceptron (Rosenblatt, 1958) and broke with the publication of *Perceptrons* (Minsky and Papert, 1972), ending in a trough following a substantial decline in research funding (Bishop, 2006, p. 193). The second wave began with the research into symbolic AI, culminating in a discipline of cognitive science called *connectionism*, before finally falling at the onset of the so-called AI Winter that ended in the 1980s (Goodfellow, Bengio, and Courville, 2016, p. 16; Howe, 2007).

Neural networks have been renascent since after 2006 with the presentation of deep belief networks (Hinton, Osindero, and Teh, 2006), which presented an efficient strategy to train deep neural networks, namely by greedy layer-wise pretraining, which reveals the technological problems until that point. The lull in interest till that point is owed in no mean part to the difficulty of training neural networks efficiently (Goodfellow, Bengio, and Courville, 2016, p. 18). Besides, other non-neural techniques were making strides and therefore more attractive, especially in light of the perceived failures of connectionism at the end of the second wave (Goodfellow, Bengio, and Courville, 2016, p. 17). RFs and SVMs continued to be a popular approach to machine learning. Eventually, however, neural models would come to the forefront of machine learning.

2.4 Deep Learning

In the time until the point of renascence in 2007 some fairly impressive demonstrations of neural networks were made, even amid the slump in research and funding. Most notably this took the form of the convolutional neural network, called LeNet-5 (LeCun et al., 1998), which achieved an impressive performance over conventional techniques on a common benchmark, the MNIST dataset: the error rate for a simple linear classifier being 12%, for a carefully designed SVM 0.56% and for a convolutional neural network 0.4% (Bishop, 2006, p. 677). As has been mentioned, a fillip to research was the deep belief network presented by Hinton, Osindero, and Teh (2006). It demonstrated specifically that a neural network could outperform an SVM with an radial basis function on MNIST. Other significant publications drew attention to deep neural networks in the following years (Bengio and LeCun, 2007; Ranzato, Boureau, and Cun, 2007). These papers coincided with the popularisation of the term *deep learning*, used to describe networks that used network depth to attain performances that exceeded conventional techniques relying so-called *hand-crafted features*.

Repeatedly in the literature, culminating in the landmark paper *Deep Learning* by LeCun, Bengio, and Hinton (2015), a repeated justification for deep learning techniques recurs. In conventional learning techniques it is necessary to select a set of basis functions or even design summary features to feed to a learning algorithm. The reasons are two-fold: (1) some assumption needs to be made about the form of the data-generating distribution that is to be modelled; and (2) the curse of dimensionality meant that one must pare out redundant features, hence techniques like PCA.

By contrast, neural networks, especially convolutional neural networks (CNNs), scale well to high-dimensional data and approximate any continuous function. The need of a human intermediary who designs various summary features or selects basis functions is wholly circumvented. Indeed, key to turning away from the hand-crafted features that are fundamental to conventional learning algorithms is that they otherwise stand in the way of modelling the variability and richness present in "natural data", and that it is impossible by such means to build an accurate recognition system entirely by hand

(LeCun et al., 1998). The argument became increasingly convincing over the years too, as the power of GPUs increased and the price of computational power decreased, such that it made neural networks yet more feasible. In short, neural networks work directly on raw information and avoid *hand-crafted features* (LeCun, Bengio, and Hinton, 2015).

2.4.1 Convolution

Before we discuss the CNN, it is necessary to describe the convolutional kernel and the convolution operation. Convolution is the integration of one signal, termed a *kernel* w in the field of deep learning, over another signal, termed the *input*. Convolution can be applied to a sequence of information, for example a function x over time and the function w weights the function according to the distance from t:

$$s(t) = \int_a x(a)w(t-a) = (x * w)(t), \tag{2.28}$$

where $s(t)$ is the convolved signal and $\star$ denotes the convolution operation. The requirement here is that both functions x and w are continuous.

In computing applications convolution must be discrete[1]. Time-series are discretised into even *timesteps*. In the above case, the function x could run forever, in which case the convolution is a infinite sum of the two functions retrospectively at a point in time t:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{2.29}$$

Often the length of the kernel w is smaller than the length of the input, in which case we assume that the function is simply zero but in the finite set of values we are concerned with (Goodfellow, Bengio, and Courville, 2016, p. 323).

Convolution may also be performed on more than one axis, especially in the case of images. Suppose I represents a function over the two-dimensional surface of an image and $I(i,j)$ represents the signal of the pixel in the ith row and jth column. Suppose further that K represents the function of a two-dimensional kernel. Discrete convolution over the kernel is therefore:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, i-n), \tag{2.30}$$

[1] Although analogue computers have been developed Paleja (2023), they are not available.

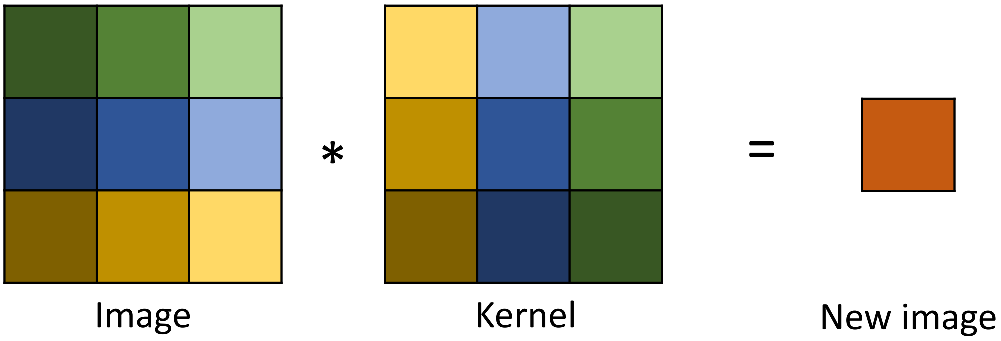


Figure 2.4: Two three-by-three matrices, an image (left) and a kernel (right). The entries that are multiplied together share a colour. The product of the paired entries are summed. Hence the all orders of summation are equivalent. The illustration here assumes that there is no striding and boundary conditions are ignored. The commutativity of convolution arises because the kernel is flipped with respect to the input image, which is useful in writing proofs, but this is usually relaxed in implementations since the kernel entries take on arbitrary values during training (Goodfellow, Bengio, and Courville, 2016, pp. 323–324).

where $S(i, j)$ is the convolved signal at pixel (i, j). The operation as formulated above is commutative and may be equivalently written

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i - m, i - n) K(m, n). \tag{2.31}$$

The two operations are equivalent because the same indices are multiplied in either case. The first starts in the top-left corner of the image and the bottom-right corner of the kernel and iterates through; the second starts at the bottom-right corner of the image and the top-left corner of the kernel and iterates through. Ultimately the orders in which the products of the two function values are summed are thus equal. The result is a sum of the corresponding entries, which means the two are commutative (Fig. 2.4). The kernel is hence flipped in respect of the indices with respect to the image or input.

A related operation where the kernel is not flipped is the *cross-correlation*:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i + m, i + n) K(m, n). \tag{2.32}$$

Cross-correlation is not commutative, however. Many machine learning libraries allegedly implement cross-correlation but call it convolution (Goodfellow, Bengio, and Courville, 2016, p. 324). We follow Goodfellow, Bengio, and Courville (2016) in using the term *convolution* to refer to both, while specifying whether the kernel is flipped where necessary.

In image processing the kernel used for convolution is typically much smaller than the image itself. A Gaussian blur is obtained by convolving the Gaussian kernel, such as the three-by-three Gaussian kernel

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix},$$

over the surface of an image. Two further examples of kernels can be used to detect horizontal and vertical edges respectively:

$$\begin{bmatrix}-1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1\end{bmatrix} \text{ and } \begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix}.$$

Since the kernel is smaller than the image, it must be iterated over the surface of the image. The algorithm is called the *sliding window*.

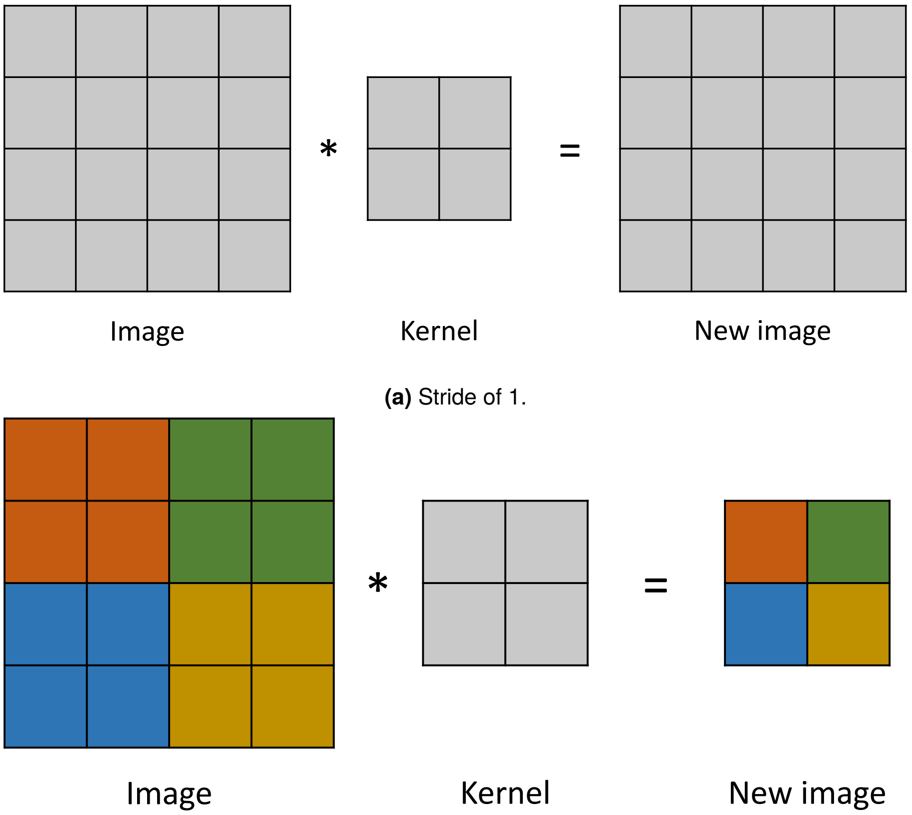
The sliding window has several parameters that affect the output size of the image. The first parameter is the stride, or the steps of the iteration. By default the stride is 1, as in Eq. 2.31, but if the stride is increased beyond 1, the dimension of the output decreases. The second parameter concerns *boundary conditions*. A "valid" convolution would not allow the kernel to overstep the bounds of the image. In such a case, the result of convolution with the identity kernel

$$\begin{bmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix}$$

would yield the same image with the pixels on the boundary missing. The dimension of the result is however smaller than the input. Alternatively the input may be padded, which means that the dimensionality is retained after convolution is performed (see Fig. 2.6).

2.4.2 The Convolutional Neural Network

The CNN is built on convolution (but it does not wholly consist of convolutional operations, as we discuss in Section 2.4.2.1). Yet, rather than using fixed kernels like those described above, the kernels' entries are learned by backpropagation. Like the feed-forward neural



(a) Stride of 1.

(b) Stride of 2.

Figure 2.5: Assuming the use of "same convolution" (Fig. 2.6), a stride of 1 will yield an output image with the same dimensions as the input image (top). A stride of 2 will halve the dimensions of the output relative to the input (bottom). The convolved locations are colour-coded on the bottom image. Each quadrant of four pixels is multiplied with the kernel to yield a corresponding cell in the new image.

network, the CNN is built of a composition of non-linear functions over successive layers, but the functions are more compact.

The operation of a single convolutional layer and its constituent neurones are commonly reckoned as a set of convolution operations. Each neurone corresponds to a single kernel that yields one channel in the output. Continuing the example of an image, suppose that $I(i, j, q)$ represents an image's pixel value in row i, column j and channel q. Suppose I has c channels. The kernel K also has c channels. The output channel o is computed thus:

$$S(i, j, o) = (K * I)(i, j, k) = \sum_q \sum_m \sum_n I(i - m, i - n, q) K(m, n, k). \tag{2.33}$$

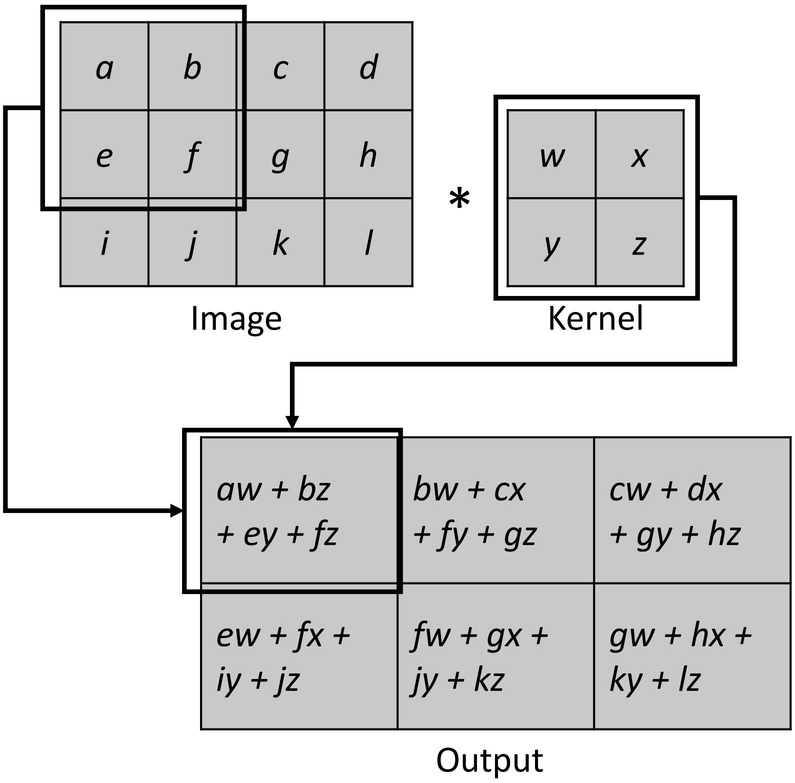


Figure 2.6: If we restrict the convolution to the boundaries of the image, the output will be smaller. This operation is called "valid" convolution. (The figure is adapted from Goodfellow, Bengio, and Courville, 2016, p. 325.) On the other hand, one could pad the image in some way before performing convolution. This is necessary to preserve the image dimensions, for example, which altogether is called "same convolution".

Equation 2.33 describes the computation on a single neurone. If there are d output channels in one layer, there are d neurones in that layer. In fact, after a certain point we ditch the terminology of neurones entirely and speak only of layers and the number of their outputs. Consequently, $I \in \mathbb{R}^{n \times m \times c}$, $K \in \mathbb{R}^{k \times k \times c}$ and $S \in \mathbb{R}^{u \times v \times d}$, such that

$$S = K * I, \tag{2.34}$$

the result of which S is passed through an activation function.

The CNN has three important and interrelated properties. The first is the *sparse interactions*. In contrast to the feed-forward neural network, the CNN has highly sparse interactions between layers. In a feed-forward neural network, a neurone in one hidden layer is fully connected to the output of the previous layer. The connections between two layers are described by a matrix operation with a complexity of $\mathcal{O}(c \times d)$. In the CNN, the connections are sparse owing to the kernel's being magnitudes smaller than the input. The kernel is a square matrix $k \times k$ and the image is $n \times m$, and k^2 is often magnitudes smaller than $n \times m$. The computational complexity of the convolution operation is $\mathcal{O}(k^2 \times d)$. Consequently the convolutional neural networks require far less computation per neurone than the feed-forward neural network.

The second property is *parameter sharing*. Since the same, compact kernel is convolved over the surface of the input, rather than learning a set of parameters for each location, we learn a single set of parameters applied everywhere. From another perspective, shared parameters are called *tied weights*, because the weights applied in one region of the input are tied to those applied to another region. Furthermore the k^2 parameters occupy small amounts of memory compared to the situation in the feed-forward neural network a unique weight is tied to every location in the input.

The third property is equivariant representations. Over successive layers, the small functions represented by the kernels are composed into larger patterns. Low-level features, such as edges, are composed over the multiple layers into higher-level features, such as curves. Moreover, although the low-level kernels are interacting with small regions in the input, with the passage of information through later layers, the later layers interact *indirectly* with large portions of the input (Goodfellow, Bengio, and Courville, 2016, p. 326). In other words, the *receptive field* of low-level kernels is small, but with composition the receptive field grows larger. Moreover, since the kernels small and the weights are tied over the space, the convolutional network is also *translation invariant*. A pattern may occur anywhere in the input and still be detected.

These properties lend the CNN its enormous power, a drastic demonstration of which was made for the 2012 International Large-Scale Visual Recognition Challenge after a CNN designed by Krizhevsky, Sutskever, and Hinton (2017) attained a 11.1 percentage point lead of the top-5 error lead over the next best learning algorithm. Another group of researchers (Zeiler and Fergus, 2014) used this model to demonstrate composition of low-level features into high-level features was demonstrated by using *deconvolution*. They found among other things that invariance to scale and translation improve with network depth, but that CNNs are not invariant to rotations of rotationally asymmetrical inputs. Successive years produced ever lower rates of error with variations on the CNN. Over the years refinements and extensions of the CNN have been presented, which now discuss briefly.

2.4.2.1 Other Deep Neural Networks

Other convolutional neural networks. InceptionNet (Szegedy et al., 2015) use multiple, parallel, differently sized kernels, called an Inception module, in 22 layers. One of those kernels is 1×1 in dimension, which essentially result in a scaled mapping of the input to the output. It could otherwise be termed a skip connection, since the original image signal is being scaled and passed forward. The model uses fewer parameters than that of Krizhevsky, Sutskever, and Hinton (2017) but attains a much lower training error: 6.7% top-5 error rate versus 15.3%.

ResNet (He et al., 2016) uses residual layers where the skip connections are *unaltered*, i.e., they are simply added together. The purpose is to counteract the deleterious effect of increasing network depth on the model's learned representations, whereby a deeper model produces a *higher* training error than a shallower model. Oddly, the authors believe degradation with depth is not a consequence of overfitting, which they did deduce from other experiments on a much larger, 1202-layer model. ResNet nearly halved the top-5 error rate of InceptionNet with 3.7%, although it uses layers (34) than InceptionNet.

Temporal models. Other CNNs have been designed to model temporal data. Recurrent neural networks (RNNs) consume a sequence of temporal data one timestep at a time. The neurones consume each timestep in sequence. The hidden state at each timestep is fed back into the neurone along with the next timestep of the input. At training, the predictions are compared stepwise with the output, with the targets fed to the model instead of the output of the RNN. This is called *teacher forcing* (Goodfellow, Bengio, and Courville, 2016, pp. 372ff.). On occasion the model may be introduced to predicted outputs rather than the targets; this is called using the *free-running inputs* (*ibid.*). Various other training techniques exist. The training error is then passed back through an unrolled computational graph using the algorithm called *backpropagation through time*. At inference, where a predicted sequence is generated, the RNN is left to run on its own predictions.

The output of a RNN can also be used to generate a context for a second RNN, in which case the RNNs are referred to respectively as the encoder and decoder. The output of the decoder is fed to the decoder as an input and as a context, like the way the hidden state are fed to successive timesteps. Such a model is useful for translating one sequence to another, such as predicting stocks or translating natural language. For that purpose there is an entire class of RNNs in the literature grouped under the term *seq2seq* (sequence-to-sequence, e.g., Li et al., 2018c).

So far the description has concerned the single-layer RNN. To achieve multiple layers, the number of neurones is simply multiplied. The kind of connections may vary according to desire (Goodfellow, Bengio, and Courville, 2016, p. 388), but the most straightforward way is to pass the outputs up through a chain of neurones for a single timestep, where each neurones is still fed with its hidden state from the previous timestep.

There are two challenges that arise with RNNs. The first is the challenge of learning long-term dependencies, in that gradients propagated over many stages explode or vanish (Goodfellow, Bengio, and Courville, 2016, p. 390). A notable example that redresses the difficulty is the long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997). Like the RNN, an LSTM unit takes an input, yields an output and passes a hidden state to itself for the following timestep. The difference is that there are extra sigmoid-activated gates. The *input gate* controls how much the input contributes to the computation of a new state. The *forget gate* controls how much of the previous should be contributed to the new state. Finally the *output gate* controls how much of the output, computed from the gated input and state, to emit from the model. These gates moderate the risk of exploding and vanishing gradients, which means that a LSTM learn long-term dependencies more easily than a RNN. LSTM units can be stacked into multiple layers similar to the RNN.

The second issue is autoregression. When a signal at timestep t depends on its value at timestep $t-1$, it is easier for a model to make a prediction that simply matches the value at the previous timestep. Indeed a model can achieve a reasonably low training error by simply duplicating the previous timestep. It is necessary to notice when a sequence is highly autoregressive in order to counter its deleterious effects on learning. One example of highly autoregressive sequences are stock prices. A common remedy in the case of sequences with similar properties to stock prices is to use the change in price rather than the stock price.

The transformer. The transformer is a non-recurrent neural network that is able to map from one sequence to another. The transformer was first presented by Vaswani et al. (2017). The structure is similar to the encoder-decoder architecture discussed in the section above on RNNs, except it is invariant to sequence length. Each step of the input features are first linearly transformed and packed altogether into three spaces for the queries, keys and values $Q, K, V \in \mathbb{R}^{d_{model}}$. The result is then multiplied by three matrices, the query, key and value matrices, $\boldsymbol{W}^Q, \boldsymbol{W}^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $\boldsymbol{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ respectively, where d_{model} is the dimensionality of the model's hidden layers and d_v, d_k is the dimensionality of the values and keys, yielding

$$Q' = Q\boldsymbol{W}^Q, \quad K' = K\boldsymbol{W}^K, \quad V' = V\boldsymbol{W}^V. \tag{2.35}$$

In an operation called *scaled dot-product attention* the queries and keys are multiplied together multiplied by a scaling factor $1/\sqrt{d}$, the result of which is multiplied by the value matrix:

$$\text{Attention}(Q', U', V') = \text{softmax}\left(\frac{Q'K'^{\top}}{\sqrt{d_{\text{model}}}}\right) V' \in \mathbb{R}^{d_{\text{model}} \times d_v} \tag{2.36}$$

The transformer accommodates further for multiple heads h, such that there are h query, key and value matrices. Their output is concatenated along the feature axis d^v, yielding an output $Z \in \mathbb{R}^{hd_v}$, which is multiplied with an output weight matrix $W^o \in \mathbb{R}^{hd_v \times d_{\text{model}}}$:

$$Z' = ZW^o \in \mathbb{R}^{d_{\text{model}}}. \tag{2.37}$$

The input works solely on self-attention; that is the input sequence constitutes the queries, keys and values. This is termed *self-attention*, and the attention blocks in the encoder consist only of self-attention mechanisms. The decoder works slightly differently because its attention blocks consist firstly of self-attention, followed by an attention mechanism where the output of parallel block in the encoder forms the keys and output sequence forms the queries and values.

Since there is no recursion, there is no sense of order in the input. To add a sense of order, Vaswani et al. (2017) add a *positional encoding* to both the input and output sequences that imparts an idea sequential order in the features. An additional problem is that, as the data is supplied simultaneously, the transformer can search ahead of a position at future steps in temporal sequences. A further stage is added in the process that masks out future steps after each step.

At inference, since there is no target sequence, the decoder is simply predicts a sequence step for step, like the RNN. The advantage of parallelism remains during training, however, which means the transformer is much faster to train than the RNN. The transformer is a very powerful tool for sequence-to-sequence processing. It lies at the core of the OpenAI GPT models.

Generative adversarial networks. A whole class of models called generative adversarial networks (GANs) utilise two models to train adversarially to a target, first introduced by Goodfellow et al. (2020). One model, called the generator, is trained to mimic data drawn from a data-generating distribution. The task of the discriminator is to judge which of two samples is the generated image. Training works in cycles and with time the generator generates fake data that the discriminator cannot distinguish from data actually drawn from the distribution. GANs have developed to the point where very convincing generated data is being produced, so much so that there is arising concern about the potential of GANs to be used for nefarious purposes. Such a usage is for the purpose of generating so-called *deep fakes*.

2.4.2.2 Other Layers in Convolutional Neural Networks

Several other layers are also present in CNNs that are worth briefly discussing. It is worth noting at this point, again, that what is considered a layer is ever in contention. Some consider activation functions to be a inseparable part of a convolution layer, while others consider them altogether separate. For the sake of discussion, we treat them separately here.

Activation functions. The most common layers to be found are the activations functions. We have already seen the sigmoid and tanh functions in Section 2.3.3. Today, as we mentioned before, these are generally discouraged within the hidden layers of a model. Instead other, simpler functions are used. The most common is the rectified linear unit (ReLU):

$$\mathrm{ReLU}(a) = \begin{cases} a & a > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.38}$$

It is a computationally very simple layer. The discontinuity at zero presents problems, however, specifically the idea of dead neurones. Variations on ReLU have instead been proposed. It is not necessary to enumerate them all, but the most common variation is the leaky ReLU

$$\mathrm{LeakyReLU}(a) = \begin{cases} a & a > 0 \\ \zeta a & \text{otherwise.} \end{cases} \tag{2.39}$$

The leaky ReLU guarantees that there is a gradient on the negative side of the activation function. Of course there is still a discontinuity, as a result of which other variations use smoother curves, such as the exponential linear unit. Each has its own advantages and they are variously used according to application.

Pooling. Pooling is often used in CNNs to reduce the dimensionality of hidden layers, but it also helps in generalising learned representations. It also helps in making representations approximately invariant to translation (Goodfellow, Bengio, and Courville, 2016, p. 330). In pooling a square of pixels in an image, for example, is extracted and some summary measure is made on it, producing a new value for that location and discarding all others.

A very common pooling method is *max pooling*, where the pixel with the largest value is taken as the new value. The invariance arises because the exact point of detection of a feature matters less, since the highest detection is extracted in any case. Indeed, together, convolution and pooling in a network act as a strong prior on invariance to permutation and translation (Goodfellow, Bengio, and Courville, 2016, pp. 334ff.).

The final dense layers. The final, dense or fully connected layers of a CNN summarise the features detected in the earlier layers. One may add any number of dense layers at the end, although the benefit to training error in the case of learning on images is doubtful (Zeiler and Fergus, 2014). Certainly it is necessary to have at least one dense layer, especially as the dimensionality of the output must be reduced to the output size.

2.4.3 Improving the Optimisation Process

We have already described the most basic process for learning the parameters of a neural network, backpropagation, in Section 2.3.4. Backpropagation can be augmented in a number of ways to accelerate learning or to avoid saddle points in the objective function (see Section 2.1.3).

Momentum. Gradient descent can be made adaptive by the inclusion of a *momentum* term to the loss function. The momentum term keeps a moving average of previous gradients. Common optimisers that use momentum are AdaGrad (Duchi, Hazan, and Singer, 2011), RMSProp (Hinton, 2012), the Adam optimiser (Kingma and Ba, 2015) and a modification of Adam called AMSGrad (Reddi, Kale, and Kumar, 2018). In AdaGrad, every model parameter has its own learning rate that is adjusted inversely proportional to the square root of the sum of all the historical squared values of the gradients (Goodfellow, Bengio, and Courville, 2016, p. 299). RMSProp improves on AdaGrad by using an exponentially weighted average of the gradients. Adam improves on RMSProp further by incorporating the first- and second-order moments of the gradients to calculate weight updates.

AMSGrad fixes a convergence issue in the Adam optimiser, which was observed occasionally not to optimise in certain conditions. The exponential by using the maximum of past squared gradients rather than an exponential weighting. The idea is that parameters with a large historical gradient will change rapidly, while those with historically small gradients will change slowly. RMSProp and Adam both add an additional decay rate on the accumulated gradients that controls the mix of old gradient values with the new. Adam contains two decay rates, one for the first-moment and another for the second-moment.

The foregoing refinements of optimisation algorithms have a drawback in that they add further hyperparameters that need to be evaluated. The Adam paper recommends defaults, but the ultimate choice lies with the architect of the model.

Regularisation. The capacity of a model may be controlled by adding or removing functions or constraining the set of solutions or a functions a model is allowed to learn. One way of forcing the model onto a smaller set of weights, thereby compelling the model to learn weights that are small and do not overly emphasise one feature or another, is to add a penalty to the loss function called a *regulariser*. The two primary and most general forms of regularisation are the L_1 norm, computes the sum of absolute gradient values, and L_2 norm, which computes the sum of squared gradient values. Whichever one is chosen is added to the loss function. The result is passed to the backpropagation algorithm. In general, the aim of regularisation is to reduce the generalisation error but not the training error (Goodfellow, Bengio, and Courville, 2016, p. 117).

Dropout. Dropout (Srivastava et al., 2014) is intended to prevent the co-adaptation of neurones in a network by randomly setting a subset of outputs to zero at training. It means that at each training step, a subnetwork is essentially selected. Each training step is consequently training a different, thinner network. At inference dropout is never applied. Dropout essentially applies masking noise to the hidden layers of a network. It forces a network to seek redundant features or alternative patterns in the data (Goodfellow, Bengio, and Courville, 2016, p. 260).

Layer and batch normalisation. Two methods are used to normalise the inputs to layers: The first method is *batch normalisation* (Ioffe and Szegedy, 2015). The purpose of batch normalisation is to reduce what the authors term *internal covariant shift*, which is the tendency of the distribution of a hidden layer's input values to shift. Essentially a batch normalisation layer maintains a running average and standard deviation of the input layer for inference. During training simply the average and standard deviation of the *batch* are used for normalisation. The stabilisation of the input distribution also speeds up training by making it easier for the model to coordinate weight updates. Owing to this normalisation, batch normalisation also combats vanishing and exploding gradients. It also has a regularising effect on the weights.

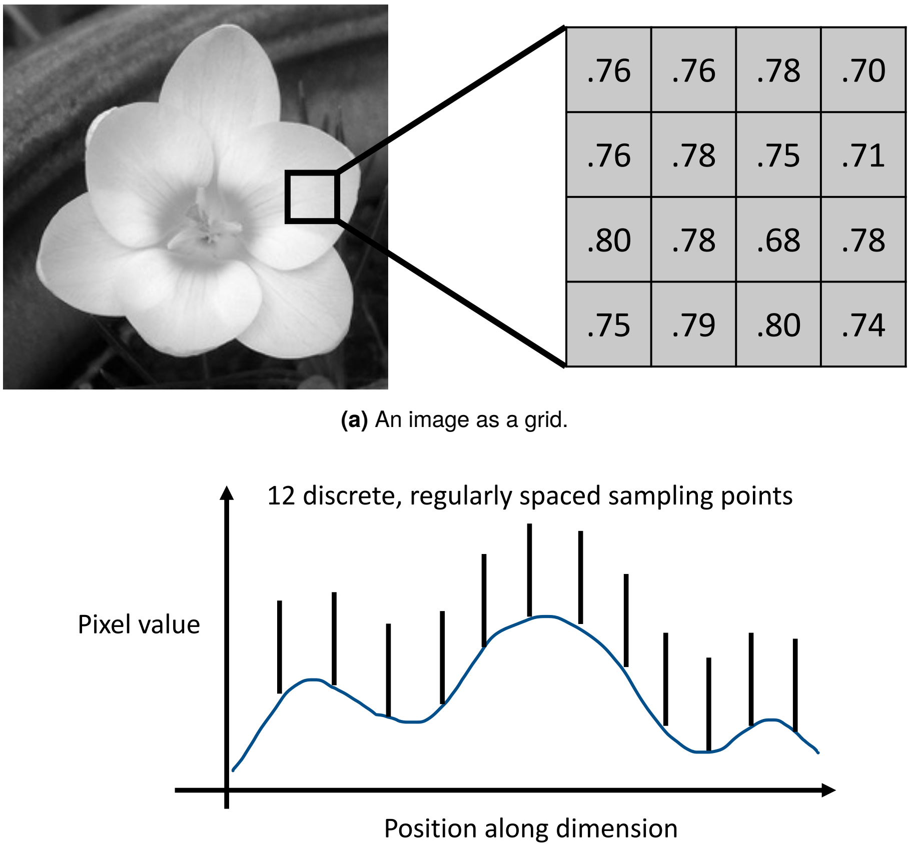
The second method is *layer normalisation* (Ba, Kiros, and Hinton, 2016). Whereas batch normalisation normalises across a batch, layer normalisation computes an average and standard deviation for all the values of a sample to normalise the input. The original motivation for layer normalisation was to address the shortcomings of batch normalisation. Layer normalisation is however best deployed in RNNs and transformers, whereas batch normalisation is usually used in networks that do not involve the modelling of sequences. It is advisable to use either batch normalisation or dropout, but not together.

Alternatives to backpropagation. There is no real alternative to back-propagation, but one extension of the backpropagation algorithm has produced some interesting results. Kuo and Chen (2018) introduced the subspace approximation with augmented kernels (Saak) transform as a reformulation of the convolutional layer. As with rectified correlations on a sphere, introduced in an earlier paper (Kuo, 2016), each layer as a Saak transform is the projection of an input onto a set of anchor vectors. The resultant projection values are rectified using ReLU—with the exception of the zeroth index, which forms the basis of reconstruction of the inverse Saak transformation. Saak transforms permit feed-forward training of a network (hence an architecture constructed using Saak transforms is described as a feed-forward network). In a single pass, the network is able to obtain an accuracy approaching that of a CNN trained using back-propagation of errors. A feed-forward network constructed from Saak transforms is also far more resilient to adversarial input examples than a traditional CNN (Kuo et al., 2019).

2.4.4 Limitations with Respect to Regular Domains

The CNN is only suitable to domains where the data has a regular structure. We have already seen how a discretised time sequence is discretised and an image is indexed by pixel location (Section 2.4.1). The regular, grid-like structure of an image allows one to slide a square kernel over the image's surface, permitting convolution to be defined. Accordingly, allowing for boundary conditions, one may place the kernel anywhere on the image's discrete space and it will fit. There is, in other words, a regular relationship between a pixel and its neighbours. One might think of the relationship of a pixel with a neighbours as the relation of a sampled point with the other sampled points. Each sampled point is a constant distance from its immediate neighbours, which are likewise a constant distant from their neighbours, and so on (Fig. 2.7)—again, boundary conditions allowing.

Many domains do not exhibit regular structure, however, and it is inappropriate to apply a CNN to them. Forcing irregularly structured information into a regular space risks undermining a model's ability to learn data by losing important information embedded in the irregular structure (Shuman et al., 2013). Electroencephalography (EEG) sensors (Song et al., 2019; Jang, Moon, and Lee, 2018; Li et al., 2019b; Rui, Nejati, and Cheung, 2016), sensors on the surface of the Earth or on a road network (Li et al., 2018a; Cai



.76	.76	.78	.70
.76	.78	.75	.71
.80	.78	.68	.78
.75	.79	.80	.74

(a) An image as a grid.

(b) A single dimension of an image as discretely sampled function.

Figure 2.7: The pixels of an image can be represented as a grid (*top*). Each cell of the grid is equidistant to its immediate neighbours. The distance to further points are a distance that is a factor of the distance of each pixel to its neighbours. An image's pixels can also be reframed as discrete, regular sampling of a continuous function (*bottom*); in this case we visualise a single dimension as a function.

et al., 2020), the positional readings from points on the human body (Edwards and Xie, 2017)—each are examples of domains where there is no regular relationship between the datapoints in an observation.

Consequently a flourishing field of deep learning on irregular domains has arisen that takes irregular domains as its learning target (Shuman et al., 2013; Bronstein et al., 2017), in particular on graphs, reflected in the growing number of surveys on the topic (Wu et al., 2019b; Zhou et al., 2018; Zhang, Cui, and Zhu, 2020; Georgousis, Kenning, and Xie, 2021). In the next section we discuss the field of deep learning on graphs.

2.5 Graph Deep Learning

The graph is a useful and very general way of representing discretely sampled information. It may be used to describe very complex relations between the sampled points. The assumption of regularity, embedded in the grid structure, that is made by the CNN is relaxed in the graph: related sampled points are not constrained to have a certain relationship to other sampled points. Indeed, graphs can even be used to structure images (Edwards and Xie, 2016); regularity violates no constraint.

In this section we describe deep learning techniques applied to graphs. First we will review what an irregular domain is. Then we will elucidate graph-theoretical definitions which we will use throughout this thesis. Final we will discuss techniques and common arising problems in graph deep learning.

2.5.1 What is an Irregular Domain?

In Euclid's treatise *Elements*, written in the third century B.C.E., the set of definitions, axioms and postulates are described that underpin regular geometry. The postulates are self-evident facts of geometry that cannot be derived from any other geometric rules. The fifth postulate in particular states that

> if a line, falling on two lines, should make two angles on the same side [of that intersecting line] smaller than right angles, the two lines, having been drawn to infinity, are to meet at a point on the same side [of the intersecting line] as the two angles smaller than right angles[2] (author's translation).

The postulate states in a effect that two lines will meet at some point if no line drawn across them produces two right angles. The corollary of that statement means that two lines never meet if that intersecting line produces two right angles. For this reason, the fifth postulate is also called the parallel postulate.

The postulate describes regular geometry, but it is not self-evidently true. New fields of geometry can be defined without the fifth postulate, called non-Euclidean geometry, such as hyperbolic geometry. The graph is a topology that can be used to represent functions in non-Euclidean, hence irregular spaces or domains.

2.5.2 Graph-theoretical Definitions

We now enumerate the various theorems and corollaries of graphs that will be helpful in this thesis. In Sections 2.5.2.4 to 2.5.2.7 we describe several extensions of the graph, but

[2] καὶ ἐὰν εἰς δύο εὐθείας εὐθεῖα ἐμπίπτουσα τὰς ἐντὸς καὶ ἐπὶ τὰ αὐτὰ μέρη γωνίας δύο ὀρθῶν ἐλάσσονας ποιῇ, ἐκβαλλομένας τὰς δύο εὐθείας ἐπ᾽ ἄπειρον συμπίπτειν, ἐφ᾽ ἃ μέρη εἰσὶν αἱ τῶν δύο ὀρθῶν ἐλάσσονες.

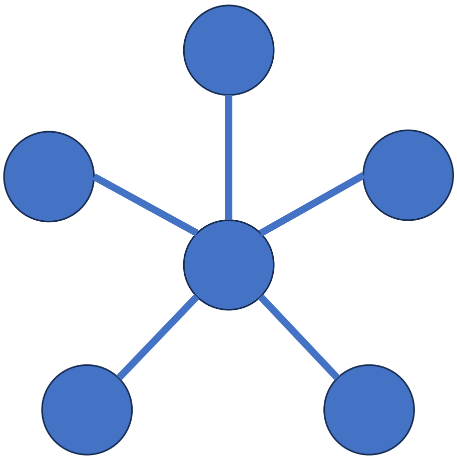


Figure 2.8: A common example of a graph is the *star graph*. It consists of four *vertices* or *nodes* connected to another, central vertex by four *edges*.

the key thing to bear in mind is that these are all still graphs. That is, the rules that we enumerate about graphs in general apply just as much to these graphs as others.

2.5.2.1 General Graph Definitions

A graph G (Fig. 2.8) is an ordered pair of finite sets (V, E), where V is the set of *vertices* or *nodes* and *edges* E is the set of edges. E is a subset of unordered pairs of V. The subset consists of all vertices x, y that are *adjacent*, such that $\{x, y\} \in E$. $\{x, y\}$ may also be written xy, e_{xy} or e. Reflexively, an edge is always adjacent to two vertices x and y, called its *endvertices*. Two edges are said to be adjacent if they share a vertex. The vertices and edges of a graph can also be described as functions $V(G)$ and $E(G)$. The number of vertices in a graph is its order $n = |V|$ and its size is the number of edges $m = |E|$. A graph with n vertices is termed an n-graph and is denoted G^n. Generally graph vertices have no self-loops, i.e., an edge joining a vertex to itself, unless otherwise specified. A subgraph of G is a graph $H = \{W, F\}$ such that $W \subset V$ and $F \subset E$. A graph is *weighted* if the edges map to a set of real values.

2.5.2.2 Degree and Neighbourhoods

The number of edges incident to a vertex x is denoted $d(x)$, a quantity termed the *degree* of x. The minimum degree of a graph is denoted $\delta(G)$ and the maximum degree is denoted $\Delta(G)$. The degree of a vertex x is equal to the *first-order*, *one-hop* or *first neighbours*, denoted $\Gamma(x)$, hence $\Gamma(x) = d(x)$. The vertex at the centre of a neighbourhood is the *target* or *locus* vertex. The ith neighbourhood of a vertex $\Gamma_i(x)$ is the set of vertices at most i steps from

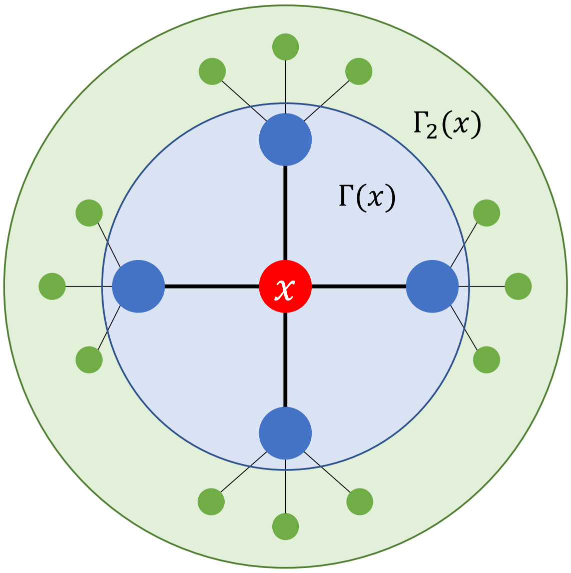


Figure 2.9: The neighbourhood of a vertex is conventionally the set of adjacent vertices. One can expand the definition of neighbours by thinking of different orders of neighbours that correspond to vertices at increasing distance from the focal vertex x. x is not guaranteed to be a member of $\Gamma(x)$. This figure has been adapted from Georgousis, Kenning, and Xie, 2021.

the locus vertex. The neighbourhood Γ is fundamental to spatial definitions of graph convolution because they describe the receptive field.

It should be noted that by the standard definition $x \notin \Gamma(x)$ unless the graph contains self-loops. It could be argued that inasmuch as $\Gamma(x) \subset \Gamma_2(x)$, if $\Gamma_0(x) = \{x\}$ then $\Gamma_0(x) \subset \Gamma(x)$. See Fig. 2.9 for an illustration of the graph neighbourhoods. The foregoing definitions of graph convolution generally assume to the contrary that $x \in \Gamma(x)$ unless otherwise stated.

2.5.2.3 The Connectivity of Graphs

A graph with no edges between its n vertices is an *empty* n-graph E^n. A *complete* n-graph K^n has an edge between every pair of vertices. A graph where the $\delta(G) = \Delta(G) = k$ is a termed a k-regular graph and is denoted R^k. A graph with a minimum degree $k = \delta(G) < \Delta(G)$ is k-connected. All n-complete graphs are $(n-1)$-connected. A graph is *connected* if there is a path between every pair of vertices in the graph.

2.5.2.4 Directed Graphs

The graphs described hitherto are *undirected*, meaning $\forall xy \in G, \exists yx \in G$ s.t. $xy = yx$. A graph is described as *directed* if $xy \in E \not\Rightarrow yx \in E$. That is, the edge set E is

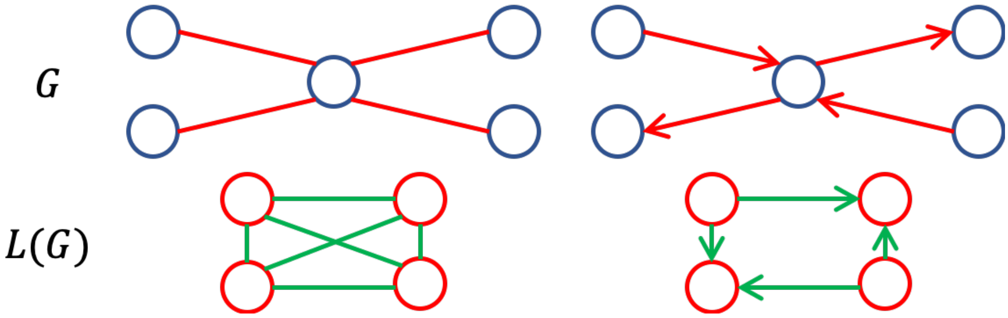


Figure 2.10: The graphs on the left describe the construction of an undirected linegraph from an underlying graph. The graphs on the right show the construction of a directed linegraph from a directed graph. The red edges in the upper graphs correspond to the red vertices in the lower graphs. This figure has been adapted from Kenning et al. (2021).

the set of *ordered* pairs of V. Every directed edge joins a vertex x, the *startvertex*, to a vertex y, the *endvertex*. An additional property arises from the orientation of edges in the directed graph. It is possible for two given edges to have both vertices in common in opposite directions. Suppose $xy, yx \in E$: We call a directed edge $yx \in E$, which joins an startvertex y to the endvertex x, the *inverse edge* of $xy \in E$. Inverse edges are central to the work presented in Chapter 3.

Since a vertex x in a directed graph can be both a start- and endvertex, the neighbourhood $\Gamma(x)$ is split into two groups of neighbours. The first group $\Gamma_{\text{in}}(x) = \{y|yx \in E\}$ is the set of in-neighbours, and the corresponding measure of degree is denoted $d_{\text{in}}(x) = |\Gamma_{\text{in}}(x)|$. The second group $\Gamma_{\text{out}}(x) = \{y|xy \in E\}$ is the set of out-neighbours, and the corresponding measure of degree is denoted $d_{\text{out}}(x) = |\Gamma_{\text{out}}(x)|$. The neighbourhood of x is thus redefined as $\Gamma(x) = \Gamma_{\text{in}}(x) \cup \Gamma_{\text{out}}(x)$ and degree is redefined as $d(x) = d_{\text{in}}(x) + d_{\text{out}}(x)$. Note that the vertex x is a member of neither $\Gamma_{\text{in}}(x)$ nor $\Gamma_{\text{out}}(x)$. Additionally, strictly speaking, an inverse edge belongs to both the in- and out-neighbourhoods, but one could also argue for its exclusion from $\Gamma_{\text{in}}(x)$ nor $\Gamma_{\text{out}}(x)$ but simultaneously its inclusion in $\Gamma(x)$.

The orientation of the graph's edges results in additional properties not present in undirected graphs. A directed graph where there is a path between every pair of vertices is *strongly connected*. A directed graph that is connected, i.e., there is a path between every pair of vertices but not in both directions, is *weakly connected*.

2.5.2.5 Linegraphs

A linegraph $L(G) = (G(E), E_L)$ is constructed from an underlying graph $G = (V, E)$ (Temperley, 1981, p. 15; see Fig. 2.10). The edges of the underlying graph bijectively map to

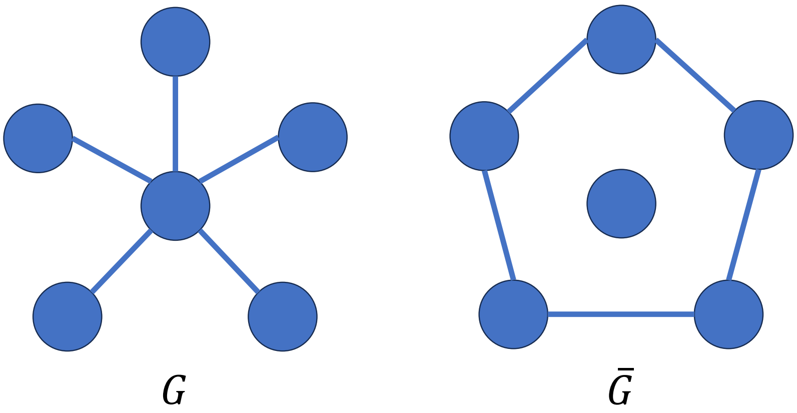


Figure 2.11: Suppose the graph G is a star graph. The complement $\bar{G}$ of the star graph G is illustrated on the right. It has the same vertices but all the edges not present in the graph G.

the vertices of the linegraph $V(L(G)) = E(G)$. Suppose a mapping $g : V(L(G)) \to E(G)$ bijectively maps the vertices of the linegraph back to the edges of the underlying graph G. A pair of vertices α, β in the linegraph are adjacent iff their corresponding edges in the underlying graph $g(\alpha) = e_\alpha, g(\beta) = e_\beta$ are also adjacent.

A directed linegraph is constructed from an underlying directed graph. As defined by Aigner (1967), a directed edge in the linegraph is drawn between two of its vertices α, β if the underlying edges on the original graph G have the same orientation, i.e., $\alpha = xy$ and $\beta = yz$. By either definition, a linegraph represents the edge-adjacency or second-order structure of its underlying graph. Moreover, since the linegraph is in fact a graph, it has all the properties of a graph described in this section.

2.5.2.6 The Graph Complement

Given an undirected graph $G = (V, E)$, its *graph complement* is $\bar{G} = (V, \bar{E})$, where $\bar{E} = \mathcal{E} \setminus E$ and $\mathcal{E}$ is the set of unordered pairs of V. An example using the star-graph is illustrated in Fig. 2.11.

2.5.2.7 Subgraphs and Spanning Subgraphs and Graphs

$G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subset V$ and $E' \subset E$. The subgraph G' is a *spanning subgraph* of G if there is a function $\phi : E' \to E$ such that $\forall x, y \in V'$, $\exists xy \in E'$ such that $\exists w = \phi(x), z = \phi(y) \in V$, $\exists wz \in E$. In other words, for every pair of adjacent vertices in the subgraph G', if there is a mapping to a pair of adjacent vertices in the graph

G, G' is a spanning subgraph of G. As far as we know, there is no term to describe a graph $G' = (V', E')$ where $E' \subset E$ but $V' = V$. Such a graph we designate a *spanning graph*.

2.5.2.8 Matrix Representations of Graphs

There are several ways to represent graph structure as a matrix. When multiple matrices are used simultaneously, there must be assumption of some order to the vertices and edges. In other words, the vertex and edge sets acquire an order. It is accomplished by assigning each vertex and edge an index. The order is arbitrary and does not matter. When the vertices of a graph have been indexed, we denote the ith vertex x_i; likewise the ith edge is e_i. If the graph already has some *cardinal order*, where vertices have some domain-defined order, then the indices are already defined. Graphs with a cardinal ordering are called *positional graphs* (Gori, Monfardini, and Scarselli, 2005). Otherwise graphs are assumed to be non-positional.

An *adjacency matrix* of an n-graph is a binary $n \times n$ matrix $\boldsymbol{A}$, where a non-zero entry at A_{ij} means that there is an edge between vertices $x_i, x_j \in V$. For undirected graphs, the entries are symmetrical; so $A_{ij} = A_{ji}$. Otherwise the entries are zero. The n-graph's degree matrix $\boldsymbol{D}$ is a diagonal matrix where the ith diagonal entry is $d(x_i)$. More compactly, $\boldsymbol{D} = \mathrm{diag}(\boldsymbol{A}\mathbf{1})$, where $\mathbf{1}$ is a vector of 1's with a length equal to the number of rows in $\boldsymbol{A}$.

From the adjacency and degree matrices we can compute the graph Laplacian matrix $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$, often simply called the *graph Laplacian* or *Laplacian*. The normalised graph Laplacian matrix is $\hat{\boldsymbol{L}} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{L}\boldsymbol{D}^{-1/2}$. The Laplacian matrix has some useful properties: since it is symmetric and semidefinite, an eigenvalue decomposition yields a full set of eigenvectors and eigenvalues. As we will see later (Section 2.5.4.1), spectral approaches to graph convolution avail themselves of the eigenvalue composition.

If a graph is weighted, we use a *weight matrix* $\boldsymbol{W}$ for the adjacency matrix in the calculation of the Laplacian. $\boldsymbol{W}$ is defined similarly to $\boldsymbol{A}$, except it is a real-valued matrix. Entries correspond to real-valued weights of edges rather than the mere presence of an edge. The degree of the weight matrix is accordingly computed differently: $\boldsymbol{D} = \mathrm{diag}(\boldsymbol{W}\mathbf{1})$.

There are differences in some matrices for directed graphs. The adjacency and weight matrices of a directed graph are not guaranteed to be symmetrical, since an edge in one direction does not mean there is an edge in the other direction. Concretely this means that a non-zero entry at row i and column j of the adjacency or weight matrix of a directed graph does not imply a non-zero entry at row j, column i. As a consequence, there are two different degree matrices for directed graphs, one recording the in-degrees $\boldsymbol{D}_{\mathrm{in}} = \mathrm{diag}(\boldsymbol{A}^{\top}\mathbf{1})$, since columns correspond to in-neighbours, and $\boldsymbol{D}_{\mathrm{out}} = \mathrm{diag}(\boldsymbol{A}\mathbf{1})$, since rows correspond to out-neighbours.

An additional consequence of the asymmetry of the directed graph is that the Laplacian matrix is not symmetric. This presents problems later in Section 2.5.4.1, where defining spectral techniques for convolution on directed graphs is more of a challenge.

For undirected and directed graphs alike, we can also compute a *transition matrix* $\boldsymbol{P} = \boldsymbol{D}^{-1}\boldsymbol{A}$, where the entry P_{ij} is the probability of transitioning from vertex x_i to vertex x_j in one step. Transition probabilities over further steps can be obtained by powers of the transition matrix, where $\boldsymbol{P}^k$ is the matrix of probabilities over k steps. It is trivial to show that the transition matrix of a directed graph is not guaranteed to be symmetrical.

2.5.2.9 Signals Structured on Graphs

A signal over a graph's vertices is a mapping $f : V \to \mathbb{R}^c$, thus every signal has a c-channel signal associated with it. Likewise one may speak of signals over a graph's edges, a mapping $f : E \to \mathbb{R}^{c'}$. Overloading notation one may talk of signals on the vertices $f(V) \in \mathbb{R}^{n \times c}$ and on the edges $f(E) \in \mathbb{R}^{n \times c'}$. There is also a mapping from individual vertices $f(x) \in \mathbb{R}^c$. We write f to refer to signals structured on the graph when it is clear from the context that the signal is structured on the vertices or edges. The graph signal, unless otherwise stated, is the data that is supplied to the first layer of a graph convolutional model. Equivalently we write $\boldsymbol{h}_0 = f(G)$. The graph features attributed to vertex x in layer l is denoted $\boldsymbol{h}_l(x)$ or $\boldsymbol{h}_{l,x}$. Unless otherwise stated, we denote the number of input features to a given layer by c and the number of output features by d.

2.5.3 Early Approaches to Learning on Graphs

There are some very early examples of neural networks on graphs. The earliest to the best of our knowledge was presented by Sperduti and Starita (1997), where a graph is first encoded into a vector that is passed to a feed-forward neural network to classify the whole graph. A RNN is used to propagate labels across a directed graph and use sigmoidal activation. Since the graphs are cyclic, the RNN units, in this context called *recursive neurones*, re-feed the output information at previous timesteps as input at successive timesteps.

Gori, Monfardini, and Scarselli (2005) presented the first graph neural network (GNN). In the GNN, each vertex has a state and a label which are propagated over several stages until a stable point is reached. Notably the vertex's own state and label is propagated to neighbours, but not to itself—though its label is. The whole propagation process is iterative and defined by a parametric function common to every vertex. The parametric function is a sum over neighbours' features, termed *contributions* in the paper, when the graph is non-positional; otherwise the weights associated with vertex positions are used to scale contributions. For non-positional graphs the parametric function has the following form:

$$h_{l+1,x} = \sum_{y \in \Gamma(x)} h_\theta \left(l(x), h_{l,y}, l(y) \right) , \tag{2.40}$$

where the function $l(-)$ is a mapping from vertices to their labels. The output of the parametric function is passed to a sigmoid function (Eq. 2.9) and is therefore non-linear. The general form of the output function is

$$o(x) = g_\theta(h_{l,x}, l(x)) , \tag{2.41}$$

where l is the number of the final layer. The algorithm works on directed edges, too, for which the parametric function need only be extended to indicate whether a neighbour is an in-neighbour or an out-neighbour. A paper published later (Di Massa et al., 2006) compares the two approaches and demonstrates that the GNN almost always outperforms the RNN-based approaches. Scarselli et al. (2009) extended Gori, Monfardini, and Scarselli's model to include edge labels, too.

Finally Micheli (2009) proposed a neural network for graphs, essentially a variation on the GNN that does not produce a distinction between vertex states and labels in the propagations and takes no edge labels. The only vertex label is present at the first iteration on each node, after which it is consumed by the first propagation. The result of each propagation is sigmoid-activated. In these respects it is similar to the network proposed by Sperduti and Starita (1997), since it includes the focal vertex in the computation of new vertex representations. A key difference is that it incrementally adds hidden units that are frozen and all contribute to the new representation of each added hidden state with some learned weighting. Each hidden unit is learned separately on the previous hidden units, already trained, rather than propagating an error backwards through an unfolded network.

The methods described above share some common attributes that recur in modern graph-based convolutional neural networks (GCNNs). Firstly the features of a focal vertex's neighbours are combined with the focal vertex's features to produce a new representation at each iteration, followed by an activation function, altogether termed the function is called the *local transition function* (Scarselli et al., 2009), *state transition function* (Di Massa et al., 2006) or simply *transition function* (Gori, Monfardini, and Scarselli, 2005). A second function, the *local output function* (Scarselli et al., 2009) or simply *output function* (Gori, Monfardini, and Scarselli, 2005; Di Massa et al., 2006), is then used to map the final vertex representations to the output space. Although the other papers (Sperduti and Starita, 1997; Micheli, 2009) do not have names for their functions, they do have functions with alike purposes. The approach by Scarselli et al. (2009) also accepts edge labels, which today we more commonly refer to as *edge attributes*. Similar functions appear in the frameworks we discuss in the next section.

2.5.4 Definitions of Convolution on Graphs

Concurrently with the development of graph convolutional models, several frameworks have been described to generalise the common attributes of GCNNs and formalise a general procedure for GCNN architectures.

The earliest framework is the message-passing neural network (MPNN) (Gilmer et al., 2017). The MPNN, like the GNN proposed by Scarselli et al. (2009), accepts an undirected graph with vertex and edge attributes, but also works with directed graph and multigraphs. The framework consists of two phases: a message-passing phase and a readout phase, similar to the transition function and output function described in Section 2.5.3. The earlier approaches iterate the passage of graph signals through recurrent units until a stable point is reached using a contracting function or weight decay to enforce the property. The MPNN does not strictly adhere to this property and simply uses the propagations to learn some vertex representations. Multiple message-passing phases stacked one after the other creates model depth and expands the receptive field of the model. The terminology introduced by Gilmer et al. (2017) is still in use today and many techniques can be described in this manner.

Note that these two processes of message-passing and readout bear a resemblance to the Weisfeiler–Lehman test of isomorphism. The purpose of the test is to produce a canonical form for a graph. Two isomorphic graphs will share the same canonical form. In the test, for each vertex the neighbouring vertices' labels are gathered at the focal vertex along with its current label, from which a new label is computed by compressing the gathered labels using, for instance, a hash function. The labels are gathered and hashed repeatedly until the partitioning of the graph on according to the labels no longer changes. In GCNN architectures the compression function is replaced by a neural network. It has been demonstrated, among other related findings, that GCNNs are at most as powerful as the Weisfeiler–Lehman test of isomorphism in distinguishing graph structures (Xu et al., 2019b).

The Graph Network (GN) framework (Battaglia et al., 2018) is more broadly defined than the MPNN. In contrast to the MPNN, it is designed to operate on directed multigraphs, which lends a designer a great deal of flexibility in designing a convolutional layer. The MPNN can be thought of as a more constrained GN. A single GN block consists of three separate update functions for attribute updates on the edges and vertices and attributes for the whole graph:

$$\boldsymbol{e}'_k = \phi^e(\boldsymbol{e}_k, \boldsymbol{v}_{r_k}, \boldsymbol{v}_{s_k}, \boldsymbol{u}), \tag{2.42}$$

$$\boldsymbol{v}'_i = \phi^v(\bar{\boldsymbol{e}}_i, \boldsymbol{v}_i, \boldsymbol{u}), \tag{2.43}$$

$$\boldsymbol{u}' = \phi^u(\bar{\boldsymbol{e}}', \bar{\boldsymbol{v}}', \boldsymbol{u}). \tag{2.44}$$

The terms $\bar{\boldsymbol{e}}_i, \bar{\boldsymbol{e}}', \bar{\boldsymbol{v}}'$ are computed from three aggregation functions: one local function that aggregates neighbours' vertex features to focal vertices, and two global functions, one which aggregates edge features and the other that aggregates vertex features:

$$\bar{\boldsymbol{e}}'_i = \rho^{e \to v}(E'_i), \tag{2.45}$$

$$\bar{\boldsymbol{e}}' = \rho^{e \to u}(E'), \tag{2.46}$$

$$\bar{\boldsymbol{v}}' = \rho^{v \to u}(V'). \tag{2.47}$$

The terms $\bar{\boldsymbol{e}}', \bar{\boldsymbol{v}}'$ are variables that are computed from all vertex and edge attributes respectively, namel The global vertex and edge features are subsequently used in the computation of a new global feature (Eq. 2.44) and included in the computation of edge and vertex features (Eqs. 2.42 and 2.43).y E') and V'. The hat on the symbols thus represents an intermediate value before the computation of the global attribute. The attribute of a given edge is denoted E'_i.

The update procedure at each layer has three steps, stated in Eqs. 2.42 to 2.44. Firstly the current attributes of each edge, its endvertices and the global representation are used to update the edge attributes (Eq. 2.42). Secondly, for every vertex, the incident edge attributes are aggregated with each vertex's current attributes and the global attribute (Eq. 2.43). The global attributes are then updated (Eq. 2.44) with the aggregation of all edge attributes (Eq. 2.46), all vertex attributes (Eq. 2.47) and the previous global attributes.

Various GCNNs can be constructed by composing multiple GN blocks or excluding different parts of the GN block. Most of the techniques described below do not have global attributes, for example, which means the corresponding components may be excluded from the GN block. As with the MPNN, the stacking of multiple GN blocks in depth yields a deeper model with a larger receptive field.

The frameworks discussed above appear to more accurately suit the definition of *spatial* definitions of convolution. There is also a category of convolutions which we term *spectral* definitions. In fact these frameworks fit both definitions since spectral convolution is formulaically equivalent to spatial convolution (Bracewell, 2000, p. 163; Shuman et al., 2013). Indeed there are cases where definitions of graph convolutions that have developed from spectral definitions are equivalent to spatial formulations in certain conditions (Kipf and Welling, 2017).

2.5.4.1 Spectral Graph Convolution

Graph Fourier transformation. First we must consider how one obtains a spectrum from graph signals. The graph Laplacian matrix represents the graph structure (Section 2.5.2.8). Since the Laplacian matrix is symmetric and semidefinite, its eigenvalue decomposition yields a full set of eigenvalues and eigenvectors,

$$\boldsymbol{L} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top, \tag{2.48}$$

where $\boldsymbol{U} \in \mathbb{R}^{n\times n} = [\boldsymbol{u}_0^\top, \dots, \boldsymbol{u}_{n-1}^\top]$, the Fourier basis of n eigenvectors, and $\boldsymbol{\Lambda} = \mathrm{diag}([\lambda_0, \dots, \lambda_{n-1}]) \in \mathbb{R}^{n\times n}$. The eigenvectors can be used to transform graph signals into the spectral domain, termed *graph Fourier transformation* or simply *Fourier transformation*. *Forward* Fourier transformation yields the spectrum of the graph signals

$$\tilde{f} = \boldsymbol{U}^\top f. \tag{2.49}$$

and we can recover the spatial graph signals using *reverse* Fourier transformation:

$$f = \boldsymbol{U}\tilde{f}. \tag{2.50}$$

In recent work (Xu et al., 2019a; Li et al., 2020b; Zheng et al., 2020), the graph Fourier transformation is generalised as the matrix multiplication of a matrix Φ with the graph signal:

$$\tilde{f} = \boldsymbol{\Phi}^\top f. \tag{2.51}$$

and likewise,

$$f = \boldsymbol{\Phi}\tilde{f}. \tag{2.52}$$

Φ is thus the basis of the transformation that could be the eigenvectors or collection of wavelets.

Eigenvectors as a Fourier basis. On this basis, the spectral graph convolution was first described by Bruna et al. (2014) and uses the eigenvectors as the Fourier basis. A spatial convolution is in fact equivalent to the Hadamard product of a filter in the spectral domain:

$$f * \boldsymbol{g} = \boldsymbol{U}\left(\left(\boldsymbol{U}^{\top}\right) f \odot \tilde{\boldsymbol{g}}\right), \tag{2.53}$$

where $\odot$ is the Hadamard product and $\tilde{\boldsymbol{g}}$ is the spectrum of the spatial kernel $\boldsymbol{g}$. Let $g_\theta = \operatorname{diag}(\boldsymbol{U}^{\top}\boldsymbol{g})$. Equation 2.53 consequently becomes

$$f * \boldsymbol{g} = \boldsymbol{U}\boldsymbol{g}_\theta\boldsymbol{U}^{\top} f, \tag{2.54}$$

where the entries of $\boldsymbol{g}_\theta$ are the trainable parameters of the convolution. The term $\boldsymbol{g}_\theta$ is a function of $\boldsymbol{L}$ and equivalently a function of its eigenvalues $\boldsymbol{g}_\theta(\boldsymbol{L}) = \boldsymbol{g}_\theta(\boldsymbol{U}\boldsymbol{\lambda}\boldsymbol{U}^{\top}) = \boldsymbol{U}\boldsymbol{g}_\theta(\boldsymbol{\Lambda})\boldsymbol{U}^{\top}$, so a more appropriate representation would be $\boldsymbol{g}_\theta(\boldsymbol{\Lambda})$. This statement holds for both normalised and unnormalised Laplacian matrices as long as they are well defined. The operation on the spectral domain described in Eq. 2.54 may be viewed from another perspective as a spectral filter. The result of the convolution is passed to an activation layer, giving

$$\boldsymbol{h}_l = \sigma\left(\boldsymbol{h}_{l-1} * \boldsymbol{g}\right) = \sigma\left(\boldsymbol{U}\boldsymbol{g}_\theta\boldsymbol{U}^{\top}\boldsymbol{h}_{l-1}\right), \tag{2.55}$$

for the lth layer, where $\sigma(-)$ is the activation function.

This definition of convolution is expensive. The graph need only be calculated once with $\mathcal{O}(n^2)$, which is however computationally prohibitively expensive as the order of the graph grows. An additional $\mathcal{O}(n^2)$ computational cost is present in the forward Fourier transformation. The number of weights in the spectral filter $\boldsymbol{g}_\theta$ also grows linearly $\mathcal{O}(\theta)$ per layer, so the memory consumption grows linearly with the number of vertices at every layer. Alternative ways to approximate the Fourier basis and define a spectrum are at the core of later refinements to graph convolution.

Approximating the Fourier basis with Chebyshev polynomials. Hammond, Vandergheynst, and Gribonval (2011) proposed a polynomial approximation of the wavelet basis Φ using Chebyshev polynomials. Defferrard, Bresson, and Vandergheynst (2016) used the definition as a basis for a graph convolutional layer.

The Chebyshev polynomial of order k is

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \tag{2.56}$$

where $T_0 = 1$ and $T_1 = x$. Defined in terms of the eigenvalues, the ith term of the Chebyshev polynomial expansion is denoted $T_i(\boldsymbol{\Lambda})$. The computation of the Chebyshev polynomials is $\mathcal{O}(kn)$ for the polynomials $0 \leq i < k$. The filter $\boldsymbol{g}_\theta$ is then defined as a k-order polynomial of $\bar{\boldsymbol{\Lambda}}$:

$$\boldsymbol{g}_\theta(\boldsymbol{\Lambda}) \approx \sum_{i=0}^{k-1} \theta_i T_i(\bar{\boldsymbol{\Lambda}}), \tag{2.57}$$

where $T_i(\Lambda) \in \mathbb{R}^{n \times n}$, θ_i is its learned coefficient and $\bar{\boldsymbol{\Lambda}} = 2\boldsymbol{\Lambda}/\lambda_{\max} - \boldsymbol{I}_n$ is the rescaled set of eigenvalues, where $\lambda_{\max}$ is the largest eigenvalue in $\boldsymbol{\Lambda}$. To avoid eigendecomposition, we can instead approximate $\boldsymbol{g}_\theta$ with the normalised Laplacian matrix $\hat{\boldsymbol{L}}$.

Redefining the convolution with the Chebyshev filter $\boldsymbol{g}_\theta$ we obtain

$$\boldsymbol{g}_\theta(\bar{\boldsymbol{L}})f \approx \sum_{i=0}^{k-1} \theta_i T_i(\bar{\boldsymbol{L}})f \tag{2.58}$$

where $\bar{\boldsymbol{L}} = 2\hat{\boldsymbol{L}}/\lambda_{\max} - \boldsymbol{I}_n$ is the rescaled Laplacian matrix. The output of the lth convolutional layer is therefore

$$\boldsymbol{h}_l = \sigma\left(\boldsymbol{h}_{l-1} * \boldsymbol{g}\right) = \sigma\left(\sum_{i=0}^{k-1} \theta_i T_i(\bar{\boldsymbol{L}})\boldsymbol{h}_{l-1}\right). \tag{2.59}$$

First-order approximation of the spectral convolution. Approximation using the kth-order Chebyshev polynomial means that the model is k-localised. Kipf and Welling (2017) proposed using a first-order approximation of the Chebyshev polynomial to define convolution. This restricts the receptive field of the convolution to the immediate neighbourhood of each vertex, but by stacking the layers over successive steps, one can obtain larger receptive fields by depth instead.

Equation 2.59 is correspondingly altered in accordance with the first-order Chebyshev polynomial expansion in Eq. 2.56 to

$$\begin{aligned} f * \boldsymbol{g}_\theta \approx \sum_{i=0}^{1} \theta_i T_i(\hat{\boldsymbol{L}}) &= \theta_0 f + \theta_1(\boldsymbol{L} - \boldsymbol{I}_n)f \\ &= \theta_0 f - \theta_1 \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}f. \end{aligned} \tag{2.60}$$

where θ_0, θ_1 are free parameters and assuming $\lambda_{\max} \approx 2$. In practice Kipf and Welling (2017) constrained the free parameters such that $\theta_0 = -\theta_1 = \theta$, giving

$$f * \boldsymbol{g}_\theta = \theta\left(\boldsymbol{I}_n + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}\right)f. \tag{2.61}$$

The authors approximate $\lambda_{\max} \approx 2$ and state that the neural network can be expected to adapt to this change in scale in the process of learning. If $\lambda_{\max} \approx 2$, the eigenvalues of $\boldsymbol{I}_n - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$ are within the interval $(0, 2)$, which leads to numerical instabilities over multiple layers. The authors therefore introduce what they call the *renormalisation trick* to combat this, where the normalisation $\boldsymbol{I}_n - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$ is redefined to $\tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$ where $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_n$ and $\tilde{\boldsymbol{D}} = \operatorname{diag}(\tilde{\boldsymbol{A}}\mathbf{1})$. With the renormalisation trick the Chebyshev convolution reduces to

$$f * \boldsymbol{g}_\theta = \theta \left(\boldsymbol{I}_n + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2} \right) f = \theta \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2} f, \tag{2.62}$$

such that $\Phi = \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$.

The convolutional layer using the first-order Chebyshev polynomial is therefore

$$\sigma(f * \boldsymbol{g}_\theta) = \sigma \left(\theta \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2} f \right). \tag{2.63}$$

Since $\tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$ remains constant during training, it can be implemented as a sparse matrix and thereby the complexity is $\mathcal{O}(mcd)$.

Cayley filters. The Cayley polynomial expansion is a series of rational complex functions that have been proposed as a Fourier basis (Levie et al., 2019). Like the Chebyshev filters proposed by Defferrard, Bresson, and Vandergheynst (2016), a Fourier basis approximated by Cayley polynomials does not require an eigendecomposition of the Laplacian matrix, has the same computational complexity as a sparse Laplacian of $\mathcal{O}(n)$ and the locality of the spectral convolution. One disadvantage of the Cayley filters is that its spectrum of polynomials is limited to the interval $[-1, 1]$ linearly, limiting their ability to specialise in small spectral bands. To mitigate this drawback, a spectral zoom coefficient is introduced.

Wavelet methods. The graph wavelet neural network (Xu et al., 2019a) uses a collection of wavelet functions as a Fourier basis. Each wavelet function is localised on the vertices spatially. Each wavelet is also able to specialise on different regions of the spectrum with a scaling parameter. The set of wavelet basis functions together produce a very sparse basis for transformation, making it a computationally efficient mode of spectral transformation. However this advantage only stands as long as one uses a sparse approximation, such as the Chebyshev polynomial expansion, which does not require an eigendecomposition. The method has a computational complexity of $\mathcal{O}(kn)$ for k basis functions. Two approaches using Haar wavelets as a Fourier basis have been proposed (Li et al., 2020b; Zheng et al., 2020). Haar functions are likewise sparse approximations of the Fourier basis and are likewise more efficient.

Spectral convolution on directed graphs. As we described in Section 2.5.2.8, the Laplacian matrix of a directed graph is asymmetric, therefore one cannot compute the full set of eigenvectors and eigenvalues from it. The Laplacian matrix therefore cannot be used to define spectral convolution on graphs; an alternative formulation is necessary.

Two distinct methods have been proposed. The first method proposed by Ma et al. (2019) avails the Perron–Frobenius theorem to compute the eigenvalues from the transition probability matrix $\boldsymbol{P}$. The theorem states that an irreducible, semidefinite matrix has a unique positive real-valued eigenvalue $\rho = \lambda_{\max}$ with an associated eigenvector $\boldsymbol{\psi} \in \mathbb{R}^n$ called the *Perron vector*. The transition matrix $\boldsymbol{P}$ is irreducible and semidefinite, we can apply this theorem such that $\psi_i > 0$ where $0 \leq i < n - 1$ and $\boldsymbol{\psi P} = \rho(\boldsymbol{P})\boldsymbol{\psi}$. A diagonal matrix $\boldsymbol{\Psi} = \mathrm{diag}\psi_{\mathrm{norm}}$ can then be constructed such that

$$\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2} \tag{2.64}$$

Substitute $\boldsymbol{P}$ for $\boldsymbol{A}$:

$$\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{P}\boldsymbol{D}^{-1/2} \tag{2.65}$$

Substitute $\boldsymbol{\Psi}$ for $\boldsymbol{D}$:

$$\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{\Psi}^{-1/2}\boldsymbol{P}\boldsymbol{\Psi}^{-1/2}. \tag{2.66}$$

As we stated in Section 2.5.2.8, $\boldsymbol{P}$ is not guaranteed to be symmetric for directed graphs. The normalised symmetric Laplacian of the directed graph is therefore

$$\boldsymbol{L}_{\mathrm{sym}} = \boldsymbol{I} - \frac{1}{2}\left(\boldsymbol{\Psi}^{1/2}\boldsymbol{P}\boldsymbol{\Psi}^{-1/2} + \boldsymbol{\Psi}^{-1/2}\boldsymbol{P}^{\top}\boldsymbol{\Psi}^{1/2}\right) \tag{2.67}$$

This may be rescaled using a variation of the renormalisation trick, whereby $\boldsymbol{P}$ is derived instead from $\overline{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_n$, $\overline{\boldsymbol{D}}_{\mathrm{out}} = \overline{\boldsymbol{A}}\boldsymbol{1}$, $\overline{\boldsymbol{D}}_{\mathrm{in}} = \overline{\boldsymbol{A}}^{\top}\boldsymbol{1}$ and $\overline{\boldsymbol{P}} = \overline{\boldsymbol{D}}_{\mathrm{out}}\overline{\boldsymbol{A}}$, such that

$$\boldsymbol{L}_{\mathrm{sym}} = \frac{1}{2}\left(\boldsymbol{\Psi}^{1/2}\boldsymbol{P}\boldsymbol{\Psi}^{-1/2} + \boldsymbol{\Psi}^{-1/2}\boldsymbol{P}^{\top}\boldsymbol{\Psi}^{1/2}\right) \tag{2.68}$$

which is very similar to $\boldsymbol{L} = (\boldsymbol{L} + \boldsymbol{L}^{\top})/2$ for undirected graphs.

Since $\boldsymbol{L}_{\mathrm{sym}}$ is symmetric, with it we can approximate the Fourier basis using Chebyshev polynomials and obtain a function similar to that of Eq. 2.63.

A refinement of this approach, proposed by Li et al. (2020a) and called the fast directed graph convolutional network, uses an approximation of $\boldsymbol{\Psi}$, since an evaluation of the

Perron vector has polynomial time complexity. By fixing the Perron vector $\boldsymbol{\psi}$ such that $\psi_i = (1/n)^{n\,3}$ for all $0 \leq i < n-1$, a decomposition-free convolutional layer may be written

$$f * \boldsymbol{g}_\theta \approx \frac{\theta}{2}\left(\left(\frac{1}{\sqrt{n}}\boldsymbol{I}_n\right)\boldsymbol{P}\left(\sqrt{n}\boldsymbol{I}_n\right) + \left(\sqrt{n}\boldsymbol{I}_n\right)\boldsymbol{P}^\top\left(\frac{1}{\sqrt{n}}\boldsymbol{I}_n\right)\right) f \tag{2.69}$$

$$= \frac{\theta}{2}\boldsymbol{P} + \boldsymbol{P}^\top f \tag{2.70}$$

A problem arises in weakly-connected directed graphs whereby a vertex may not have any out-neighbours. As a consequence there is a risk that a row of $\boldsymbol{P}$ might sum to zero, which does not conform to the definition of a probability matrix. For this reason a self-loop is added to $\boldsymbol{A}$, which in effect yields the same set of renormalisations as earlier for $\overline{\boldsymbol{A}}$ and $\overline{\boldsymbol{D}}_{\text{out}}$, such that

$$f * \boldsymbol{g}_\theta \approx \frac{\theta}{2}\boldsymbol{P} + \boldsymbol{P}^\top f \tag{2.71}$$

$$= \frac{\theta}{2}\overline{\boldsymbol{D}}_{\text{out}}^{-1}(\boldsymbol{A}+\boldsymbol{I}_n) + (\boldsymbol{A}+\boldsymbol{I}_n)^\top\overline{\boldsymbol{D}}_{\text{out}}^{-1} f \tag{2.72}$$

$$= \frac{\theta}{2}\overline{\boldsymbol{D}}_{\text{out}}^{-1/2}\overline{\boldsymbol{A}}\overline{\boldsymbol{D}}_{\text{out}}^{-1/2} + \overline{\boldsymbol{D}}_{\text{out}}^{-1/2}\overline{\boldsymbol{A}}^\top\overline{\boldsymbol{D}}_{\text{out}}^{-1/2} f \tag{2.73}$$

$$= \frac{\theta}{2}\overline{\boldsymbol{D}}_{\text{out}}^{-1/2}\left(\overline{\boldsymbol{A}} + \overline{\boldsymbol{A}}^\top\right)\overline{\boldsymbol{D}}_{\text{out}}^{-1/2} f. \tag{2.74}$$

In a convolutional layer this approximation of $f * \boldsymbol{g}_\theta$ is passed through an activation layer.

2.5.4.2 Spatial Graph Convolution

Definitions of spatial graph convolution are devised solely on the space of the graph. The definitions are in general easier to understand. It is also easier to translate domain-specific goals into spatial definitions of graph convolution. The frameworks we discussed at the beginning of this section are phrased largely in terms of physical convolution. In this section we discuss particular techniques in the literature.

The definition of local structure is most simply each vertex x's immediate neighbours $\Gamma(x)$, the one-step neighbours Theoretically one could include neighbours any number of steps away from the focal vertex x, much like one sets the size of a convolutional kernel to be some odd width and height larger than 1. Vertices that are more distantly are however usually captured by stacking multiple layers. The graph is irregular, however, and the sparsity of the graph can affect the computational expense of convolutions depending

[3] In the original paper the authors state that the vector is fixed at $(1/n, \ldots, 1/n)^n$. We assume that the n is either supposed to be the symbol for transposition $\top$ or is a statement of the vector length n. Later calculations would not make sense if it meant a power.

on the application. Sometimes it is appropriate, according to the domain's requirements, to subsample vertices for convolution. Subsampling vertices however becomes difficult when there is a highly variable vertex degree in the graph. Instead a cap might be placed on the number of vertices in a neighbourhood. Or again one might choose some heuristic by which to prune edges. Graphs offer freedom and a great deal of choice, for all the burden of choice it also causes.

The bigger question is what function to apply to the signals once they have been aggregated at each vertex. (In the context of the earlier approaches, this is the transition function.) It is in this respect that the approaches we discuss below will differ most. There are two classes of function: isotropic and anisotropic. Isotropic functions assume that the signals of neighbours contribute equally—or statically, in the case where a weight matrix determines the contribution but does not change throughout training. Anisotropic functions allow the contributions to vary throughout training. The weights applied to the neighbours need not be independent of one another, however. The weights of an anisotropic kernel are either explicit, where they are adjusted directly with backpropgation, or implicit, where some secondary process latent in the weight is determining its value. Isotropic functions are quicker to train and take inferences from and generally have fewer parameters, but their modelling capacity is lower.

There is an additional problem before we continue with our discussion of approaches. A pressing issue is, if one is to use an isotropic approach, how one should associate neighbours' signals with a parameter. In images it is simple because a kernel is applied to regularly sized grids of pixels and there is always a correspondence in the sizes. With graphs, however, (1) the degree of vertices are not guaranteed to be equal, and (2) the vertices do not have a natural ordering (unless specified in the domain). There is therefore (1) no way of defining a fixed vector of parameters that works at every focal vertex and (2) there is no generic way to create, say, a dictionary of weights mapped onto individual vertices that is invariant up to isomorphism. There are two corresponding exceptions: (1) the graph is *regular*, in which case vertex degree is constant across the graph, and (2) the graph is *positional*, in which case there is a cardinal order on the vertices (see Section 2.5.2.8). Some interesting ways around this have been developed, which we also discuss in this section.

Molecular fingerprints using graphs. Duvenaud et al. (2015) proposed a GCNN to learn molecular fingerprints, a vector that embeds a whole graph. This recalls the work of Sperduti and Starita (1997), except here the graph embedding is learned end-to-end in the neural network. Duvenaud et al.'s model has k successive convolutional layers. Each layer aggregates and sums each atom/vertex's signals together with its neighbours' and multiplies the result with a matrix of weights indexed by the focal vertex's degree:

$$\boldsymbol{h}_{l,x} = \sigma\left(\left[\boldsymbol{h}_{l-1,x} + \sum_{y\in\Gamma(x)} \boldsymbol{h}_{l-1,y}\right] \boldsymbol{\Theta}_{l-1,d(x)}\right), \tag{2.75}$$

where $\boldsymbol{\Theta}_{l-1,d(x)} \in \mathbb{R}^{c\times d}$. The output of each layer is then projected by another weight matrix into the vector-space of the molecular fingerprint, to which softmax is applied to sparsify the embedding.

Since the model is learning on organic compounds, the maximum valency is five; correspondingly there are five sets of weights per layer $\boldsymbol{\Theta}_{l,i}$ for $0 \leq l < k-1, 1 \leq i \leq 5$. The method would scale poorly to domains where there is a highly variable vertex degree, however, the parameters growing by $\mathcal{O}(k\log(k))$ where $k = \Delta(G)$. Since the local signals are summed before the weight matrix is applied, the kernel is isotropic. Moreover, the summation is commutative, which renders the model invariant to the order of vertex neighbours. The approach does not appear to make an allowance for directed graphs either.

PATCHY-SAN. PATCHY-SAN (Niepert, Ahmed, and Kutzkov, 2016) consists of three stages: selection, aggregation and normalisation. The selection stage produces a subset of k vertices that will be the focal vertices during the aggregation process. The subset is selected using a ranking based on a canonical ordering generated by the Weisfeiler–Lehman test described in Section 2.5.3. After the vertices have been ordered, every pth vertex is selected until k vertices have been selected.

In aggregation, the second stage, the signals around the k selected vertices are aggregated. The neighbouring signals at increasing steps from the focal vertices are collected into a set B until the point where a threshold q is reached or breached, i.e., $|B| \geq q$ or there are no more vertices to select. For each selected vertex there a subgraph is therefore constructed of its neighbourhood set B.

The final stage, normalisation, takes each subgraph and normalises it using some labelling procedure, which also produces a ranking of the vertices. The top k vertices in the subgraph B are selected for the convolution. Two subgraphs with similar structures will therefore have a similar ranking over their vertices. A learned one-dimensional parameter vector is applied over the ordered signals and summed. Thus the vertex signals are weighted according to their structural role.

The labelling process is expensive as it needs to be applied to input graph and every subgraph. Canonicalisation with the Weisfeiler–Lehman test however ensures that PATCHY-SAN is not sensitive to the ordering of vertices, as all inputted graphs are represented as their canonical graphs. The sequence of selected vertices of two isomorphic graphs would be identical.

PATCHY-SAN copes with the varying vertex degrees by subsampling neighbourshoods and padding receptive fields to a fixed size k. With vertices of small degree, however, the algorithm would necessitate the sampling of distant, uninformative features and make the signals in the subgraph noisy. In the worst case it includes counter-informative signals, harming learning. It is not clear whether a high variation in vertex degree would lead to comparable vertex rankings across the whole graph. The advantage of the imposed positionality is that the algorithm is learning an anisotropic kernel over the signals. There seems to be no allowance for directed graphs, though.

MoNet. MoNet (Monti, Bronstein, and Bresson, 2017) is a general framework for learning on irregular geometries. The convolutional kernel is a mixture of k normal distributions with learned means and covariances in a so-called pseudo-coordinate space. The model parameters are thus learned indirectly. Altering certain aspects of the model allows for a definition on the graph.

An initial set of coordinates for each vertex and its neighbours is computed, for which the vertex degree is used:

$$u(x,y) = \left(\frac{1}{\sqrt{d(x)}}, \frac{1}{\sqrt{d(y)}} \right)^{\top}, \tag{2.76}$$

where x and y are two vertices in the graph and x is the focal vertex. One could substitute whatever other structural measures here instead of vertex degree.

The pseudo-coordinates are transformed in a dense layer with a tanh activation function

$$\tilde{u}(x,y) = \tanh(\boldsymbol{W} u(x,y) + \boldsymbol{b}) \tag{2.77}$$

with learned weights $\boldsymbol{W} \in \mathbb{R}^{r \times 2}$ and biases $\boldsymbol{b} \in \mathbb{R}^{r}$. The dimensionality of the pseudo-coordinates r is a hyperparameter; the authors chose $r \in \{2, 3\}$, depending on the dataset.

The transformed pseudo-coordinates are used in defining the k weight functions $w_i : \mathbb{R}^r \to \mathbb{R}$ by taking the distance of the pseudo-coordinates from the centre of each Gaussian distribution i parametrised by a set of means $\boldsymbol{\mu}_i \in \mathbb{R}^r$ and covariances $\boldsymbol{\Sigma}^{r \times r}$, constrained to be orthogonal:

$$w_i(\tilde{u}(x,y)) = \exp\left(-\frac{1}{2} \left(\tilde{u}(x,y) - \boldsymbol{\mu}_i \right)^{\top} \boldsymbol{\Sigma}_i^{-1} \left(\tilde{u}(x,y) - \boldsymbol{\mu}_i \right) \right). \tag{2.78}$$

The vertex signals neighbouring vertex x are then scaled by their respective weights:

$$D_i(x)\boldsymbol{h}_l = \sum_{y \in \Gamma(x)} w_i(\tilde{u}(x,y)\boldsymbol{h}_{l-1,y}). \tag{2.79}$$

Note that an implicit assumption of this approach is that $x \in \Gamma(x)$. The contribution of each i Gaussian kernel to the new vertex representation is weighted with an additional parameters $\theta_i \in \mathbb{R}$:

$$\boldsymbol{h}_l(x) = \sum_{i=0}^{k-1} \theta_i D_i(x)\boldsymbol{h}_{l-1}. \tag{2.80}$$

Each MoNet layer has k kernels, yielding altogether $k(r^2 + r + 1)$ parameters, and there are an additional k parameters to combine the output of each distribution.

MoNet manages varying vertex degrees by using an auxiliary mechanism to compute the vertex weights. Indirectly, all pairs of vertices with the same ordered pair of degrees will compute the same initial pseudo-coordinates in Eq. 2.76 and consequently the same weight is applied to the neighbouring signals following the transformation in the dense layer in Eq. 2.77. Yet it avoids depending on an ordering of the vertices because it does not need to index weights. Although the convolution is limited to the first-order neighbourhood, multiple layers may be composed to obtain wider receptive fields. It is not clear how this approach could be adapted to directed graphs, however.

GraphSAGE. GraphSAGE (Hamilton, Ying, and Leskovec, 2017b) is a conceptually simpler GCNN. Unlike PATCHY-SAN, where the graph is subsampled before and during convolution, GraphSAGE takes all vertices in the graph as focal vertices. Each layer consists of two principal procedures: sampling and aggregation. Because GraphSAGE is designed to work on very large graphs with high vertex degrees, in the sampling stage, a subset of each vertex's neighbours is uniformly sampled. A different sampling is made at each layer and the number of samples is a hyperparameter.

The aggregation stage follows on the back of the sampling stage. The features of the sampled vertices are aggregated,

$$\boldsymbol{h}_{l,\Gamma(x)} = \text{aggregate}_l\left(\{\boldsymbol{h}_{l-1,y} \forall y \in \Gamma(x)\}\right) \in \mathbb{R}^c, \tag{2.81}$$

and concatenated with the focal vertex's features and passed through a dense layer:

$$\boldsymbol{h}_{l,x} = \sigma\left((\boldsymbol{h}_{l-1,x} \parallel \boldsymbol{h}_{l,\Gamma(x)})\boldsymbol{W}_l\right), \tag{2.82}$$

where $\boldsymbol{W}_l \in \mathbb{R}^{2c\times d}$ is a matrix of learned weights for the lth layer and σ is a non-linear activation function.

There are three options for the aggregator function $\mathrm{aggregator}_l$ proposed by the authors, although conceivably any other symmetric function could be used. The LSTM aggregator is one example. The LSTM unit makes an implicit assumption of order, though, so the inputs have to be shuffled to enforce the property of symmetry. The pooling aggregator simply takes the maximum value for each channel in the aggregation.

For the mean aggregator Eqs. 2.81 and 2.82 need to be short-circuited. Instead, assuming the neighbourhood includes the focal vertex, the output of the layer is

$$\boldsymbol{h}_{l,x} = \sigma\left(\left[\frac{1}{|\Gamma(x)|}\sum_{y\in\Gamma(x)}\boldsymbol{h}_{l-1,y}\right]\boldsymbol{W}_l\right), \tag{2.83}$$

where $\boldsymbol{W}_l \in \mathbb{R}^{c\times d}$ is the weight matrix for the lth layer.

By fixing the number of vertices sampled at each layer, GraphSAGE is able to constrain the computational complexity of the graph, while the different sampling at each layer means that GraphSAGE can capture overlapping, distinct structures at each vertex. The sampling also has an additional mechanism whereby vertices with degrees lower than the sample size are oversampled, which ensures that unhelpful information is not included as in PATCHY-SAN. With the aggregator function it uses pooling, for example, the oversampling is obscured by the discarding of non-maximal values, and the average in the mean aggregator flattens out the contributions of oversampled vertices.

The weight applied to the vertices is however isotropic, since the vertex features are summed without distinction. This reduces the modelling capacity, which might not be an issue in the tasks with the large networks that GraphSAGE was evaluated. Ultimately, the role that vertices play structurally within a layer is lost and only recovered partially in the stacking of layers with subsamples of vertices. There is however no clear way of adapting this approach to directed graphs.

The Graph Convolutional Network. We have already described the Graph Convolutional Network (GCN) by Kipf and Welling (2017) in Section 2.5.4.1. Yet it is worth mentioning among the various definitions of spatial convolutions for its superficial similarity to spatial convolution. Indeed, the definition of the GCN reveals how spectral definitions can be constrained to become identical with spatial definitions. The Fourier basis in Eq. 2.63 $\tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$, obtained from the first-order Chebyshev approximation of the Laplacian, when multiplied with the graph signals, is essentially summing the signals of the focal vertex and its neighbours and scaling the result by a learned parameter θ. As before, there is no clear way of adapting this approach to directed graphs unless we use some alternative definition of the Laplacian matrix. Many spatial techniques take the GCN as a basis for defining spatial convolution.

The Directed Graph Convolutional Network. The Directed Graph Convolutional Network (DGCN) proposed by Tong et al. (2020) defines a spatial convolution that incorporates first- and second-order neighbourhoods in the aggregation. Each convolutional layer of the model consists of three adjacency matrices: a first-order convolution, a second-order convolution of in-neighbours, and a second-order convolution of out-nighbours. Each convolution is implemented using the GCN definition of spatial convolution. Effectively the first-order convolution describes the undirected interactions of each vertex with its neighbours. The second-order contributions describe the contribution of the focal vertex's neighbours as a fraction that the edge weighs of the neighbours' own adjacent vertices.

The first-order adjacency matrix $\boldsymbol{A}$ is the symmetricised adjacency matrix of the directed graph. It is not explicitly described in the paper, but a symmetric adjacency matrix of a directed (therefore asymmetric) adjacency matrix is a simple addition of the adjacency matrix with the transpose of itself.

The second-order matrices are descriptions of the relative contributions of neighbours to focal vertices. Consider the case with the out-neighbours. Suppose we have a vertex $x \in G$, which has an out-neighbour $y \in \Gamma_{\text{out}}(x)$. Suppose $K = \{z \mid z \in G, y \in \Gamma_{\text{out}}(z) \wedge x \in \Gamma_{\text{out}}(x)\}$, the set of all vertices that also have y as an out-neighbour. Suppose further that A_{xy} is the edge weight between vertices x and y. The second-order matrix for the out-neighbours $\boldsymbol{A}_{\text{out}}$ is calculated with the following formula:

$$A_{\text{out},xy} = \sum_{z \in K} \frac{A_{xy} A_{zy}}{\sum_{v \in K} A_{vy}}. \tag{2.84}$$

The second-order in-neighbour adjacency matrix $\boldsymbol{A}_{\text{in}}$ is calculated similarly.

A useful property of this approach is that the three matrices $\boldsymbol{A}$, $\boldsymbol{A}_{\text{in}}$, $\boldsymbol{A}_{\text{out}}$ are symmetric, which makes the choice of definition of convolution for designing GCNN flexible, all the while maintaining the difference between the in- and out-neighbours. The matrices $\boldsymbol{A}_{\text{in}}, \boldsymbol{A}_{\text{out}}$ are moreover computed using a symmetric measure, so really only $(n^2 + n)/2$ entries need to be computed for the upper triangles of both matrices.

The Graph Attention Network. The graph attention network (GAT) (Veličković et al., 2018) is an interesting example of a spatial technique. At the core is an attention mechanism that maps every pair of adjacent vertices' signals to a coefficient. Consider the case of two adjacent vertices $x, y \in G$ with vertex features $\boldsymbol{h}_{l-1,x}, \boldsymbol{h}_{l-1,y}$ at layer $l-1$. The vertex features are first projected through a weight matrix $\boldsymbol{W} \in \mathbb{R}^{d' \times c}$ such that $\boldsymbol{h}'_{l-1,x} = \boldsymbol{W}\boldsymbol{h}_{l-1,x} \in \mathbb{R}^{d'}$ and $\boldsymbol{h}'_{l-1,y} = \boldsymbol{W}\boldsymbol{h}_{l-1,y} \in \mathbb{R}^{d}$. The projected representations are passed into the attention mechanism to yield a coefficient $C_{l,xy}$ like so:

$$C_{l,xy} = a(\boldsymbol{h}'_{l-1,x}, \boldsymbol{h}'_{l-1,y}). \tag{2.85}$$

Although it could be defined otherwise, the attention mechanism is in this case a single dense layer with a weight vector $\boldsymbol{a} \in \mathbb{R}^{2d'}$ applied to the concatenated vertex features:

$$C_{l,xy} = \mathrm{LeakyReLU}(\boldsymbol{a}[(\boldsymbol{h}_{l-1,x} \parallel \boldsymbol{h}_{l-1,y})]) \tag{2.86}$$

where the slope in the leaky ReLU layer is set to 0.2.

The attention coefficients are then normalised by a softmax function over their neighbourhoods:

$$\alpha_{l,xy} = \mathrm{softmax}_{\Gamma(x)}(C_{l,xy}) = \frac{\exp(C_{l,xy})}{\sum_{z \in \Gamma(x)} \exp(C_{l,cz})}, \tag{2.87}$$

where $\alpha_{l,xy}$ is the normalised attention coefficient for the vertex pair x, y. In fact the attention mechanism here is centred on the focal vertex x. Consequently $\alpha_{l,xy} \neq \alpha_{l,yx}$.

With the attention coefficients calculated, we can finally carry out the convolution by applying the weights:

$$\boldsymbol{h}_{l,x} = \sigma\left(\sum_{y \in \Gamma(x)} \alpha_{l,xy} \boldsymbol{W}_l \boldsymbol{h}_{l-1,y}\right) \tag{2.88}$$

where $\boldsymbol{W}_l \in \mathbb{R}^{d \times d'}$ is the weight matrix for layer l, σ is some non-linear activation function. Again, the focal vertex is being implicitly included in the neighbourhood of x.

The GCN also optionally includes k attention heads in each layer. Till now the method we have described has one attention head, but we can combine the outputs of several parallel attention mechanisms by duplicating the whole attention mechanism, yielding $\alpha_{l,i,xy}$ for $0 \leq i < k-1$. The output of the attention mechanisms can then be concatenated along the channels or alternatively averaged, such that

$$\boldsymbol{h}_{l,x} = \sigma\left(\sum_{i=0}^{k} \sum_{y \in \Gamma(x)} \alpha_{l,i,xy} \boldsymbol{W}_l \boldsymbol{h}_{l-1,y}\right) \in \mathbb{R}^{d}. \tag{2.89}$$

Several parallel attention heads can learn different structures like several convolutional kernels on an image. This is called a *multi-head attention mechanism*. There composition hence yields a higher-capacity model.

The attention mechanism, especially with multiple heads, therefore allows the model to learn anisotropic functions over the vertex neighbourhoods. Moreover, as before, composing several multi-head attention mechanisms produces a broader receptive field.

The Graph Isomorphism Network. Finding that the GCN and GraphSAGE are incapable of discriminating certain graph structures, the graph isomorphic network (GIN) (Xu et al., 2019b) was proposed as a simple yet elegant spatial convolution approach. Each layer l of the GIN is its own feed-forward neural network or MLP, defined

$$\boldsymbol{h}_{l,x} = \mathrm{MLP}_l \left((1 + \epsilon_l) \cdot \boldsymbol{h}_{l-1,x} + \sum_{y \in \Gamma(x)} \boldsymbol{h}_{l-1,y} \right), \tag{2.90}$$

where MLP is a multi-layer perceptron and $\epsilon_l \in \mathbb{R}$ is a learned parameter of each layer or a constant across all layers. Experimentally it was found that the model attains a good performance when $\epsilon_l = 0$ for all l. Unless the vertex features are one-hot vectors, the features are first projected into a new space through another MLP before being passed to the first layer of the GIN.

The GIN is as powerful as the Weisfeiler–Lehman test for isormorphism. The MLP of each layer embeds the vertex representations in the previous layer into a lower-dimensional space where similar structures can be embedded closely. This embedding allows the network to discriminate different structures that the authors demonstrate previous techniques could not discriminate.

Like GraphSAGE, GIN uses an isotropic kernel, although the MLP may of course approximate any function per universal approximation theorem. Once again, it is not clear how a directed graph could work in this model.

Directional message-passing. We now come to some methods that do work on directed graphs. The DimeNet proposed by Gasteiger, Groß, and Günnemann (2020) factorises the contributions of neighbours to the focal vertex's new representation into incoming and outgoing messages. It is specifically intended to be applied to molecules, so the terminology used here will sometimes reflect that. The vertices of the molecular graph represent atoms and edges represent chemical bonds. The representation of a vertex x at layer l is denoted

$\boldsymbol{h}_{l,x}$ and is computed by summing the messages passed from its adjacent vertices, such that

$$\boldsymbol{h}_{l,x} = \sum_{y \in \Gamma(x)} \boldsymbol{m}_{l,yx} \tag{2.91}$$

where $\boldsymbol{m}_{l,yx}$ is a message from vertex y to vertex x and is defined in terms of its adjacent messages:

$$\boldsymbol{m}_{l,yx} = f_{\text{update}} \left(\boldsymbol{m}_{l-1,yx}, \sum_{z \in \Gamma(y) \setminus \{x\}} f_{\text{int}} \left(\boldsymbol{m}_{l-1,zy}, e\left(y,x\right), a\left(zy,yx\right) \right) \right) \tag{2.92}$$

where $e(y,x)$ is the radial basis function of the atomic distance of the atoms represented by y and x and $a(zy,yx)$ is the bonding angle of the bonds between the atoms represented by vertices z and y on the one hand and y and z on the other. The authors note that the messages effectively act as edge embeddings as used in the Weisfeiler–Lehman test, claiming in turn that DimeNet can provably distinguish molecules that a regular GCNN cannot.

The model is therefore able to model directed graphs, but it is not able to model the in-neighbours and out-neighbours as distinctive flows.

Directed diffusion. Directed diffusion on graphs was originally proposed Li et al. (2018a) as a way of modelling flows of traffic in a network. It is an extension of the k-step diffusion process on a graph. It is an interesting formulation that measures the contributions of in- and out-neighbours of a focal vertex separately over several diffusion steps. Convolution is defined as a summation of k diffusion steps over the input:

$$\boldsymbol{h}_l * \boldsymbol{g}_{\theta,l} = \sum_{i=0}^{k-1} \left(\theta_{l,k,1} (\boldsymbol{D}_{\text{out}}^{-1} \boldsymbol{W})^i + \theta_{l,k,2} (\boldsymbol{D}_{\text{in}}^{-1} \boldsymbol{W}^\top)^i \right) \boldsymbol{h}_{l-1}, \tag{2.93}$$

where $\boldsymbol{W}$ is the graph's weight matrix. The result of the diffusion is then passed through an activation function.

The parameters of the network factors the flows of information to and from a given vertex into two learning problems, but the kernels are effectively anisotropic over each direction and each diffusion step. The directed diffusion therefore does not share the flexibility of the kernel that, say, the GAT has.

The diffusion convolution is able to adjust the coefficients of a directed process across the graph in a receptive field, the breadth of which is proportionate to the value of k chosen as a hyperparameter. The risk of stacking layers with a broad receptive field is graph smoothing, which can occur over multiple successive graph layers unless remedial means are taken to mitigate it. We briefly discuss over-smoothing in the next section on graph-related problems.

2.5.5 Over-smoothing in Deep Graph Neural Networks

A full treatment of learning issues on graphs is beyond the scope of this work, for which see Georgousis, Kenning, and Xie (2021). Here we discuss one problem that plagues GCNNs: *graph smoothing*.

It has been observed empirically that networks with a depth greater than two convolutional layers suffer degradation in performance on vertex classification tasks (Kipf and Welling, 2017; Wu et al., 2019a). The term *over-smoothing* was first coined by Li et al. (2019a), who found that graph convolution is a kind of Laplacian smoothing. Over several layers the vertex signals become more alike to one another and eventually the only differentiating information between vertices lies in the structure. Since the vertex representations have become so alike, deep GCNNs suffer from poor performance on vertex classification tasks. The degradation of performance appears to be related to the spectral properties of a graph, too, namely the smallest positive eigenvalue of the normalised Laplacian matrix. The problem is worsened as the order and density of the graph grows (Oono and Suzuki, 2020). These results were confirmed by Cai and Wang (2020), who additionally found that the ReLU and leaky ReLU activation functions *reduce* expressive power and thereby contribute to over-smoothing. Another work found that tanh retains the linear independence of features (Luan et al., 2019). Another paper suggests that the coupling of transformation and propagation layers worsens the problem, showing that using k successive MLP layers followed by a k aggregations a model's performance degrades in terms of test accuracy and a measure of graph smoothness.

Whatever the case, there are three common solutions for mitigating over-smoothing: residual connections, weight or feature normalisation and edge sampling (Georgousis, Kenning, and Xie, 2021). Residual connections mitigate over-smoothing and serve to delay its appearance in a model, allowing deeper models (Li et al., 2019a; Chen et al., 2020b). Finally, adding a weighted residual connection imposes a degree of weight normalisation (Oono and Suzuki, 2020).

The purpose of normalisation of weights and features is to regularise the model, as with layer normalisation (Ba, Kiros, and Hinton, 2016) and batch normalisation (Ioffe and Szegedy, 2015). One approach involves normalising intermediate vertex representations by scaling them to the same L_2 norm (Zhao and Akoglu, 2020). Another approach is to normalise groups of similar vertices independently by learning a clustering assignment matrix (Zhou et al., 2020).

Edge sampling on the other hand sparsifies the graph and slows down the onset of over-smoothing (Oono and Suzuki, 2020). The random sampling of edges in each training step has been experimentally demonstrated to reduce over-smoothing and even improve classification accuracy (Rong et al., 2020a).

We conjecture that the radius of the graph is the upper bound for a graph convolution before graph smoothing occurs, which is somewhat related to graph density.

2.5.6 Learning on Graph Edges

We have already seen an early GNN (Di Massa et al., 2006) that included edge attributes in the transition function. An example among recent works is the GN framework (Hamilton, Ying, and Leskovec, 2017a), which includes edge attributes in the updates of vertex attributes. Most prominent modern architectures forgo edge attributes altogether, however. A weaker example of edge updates is also found in the DimeNet (Gasteiger, Groß, and Günnemann, 2020), which in essence computes a second- and third-order attribute corresponding to the atom distance and bonding angle of two atoms respectively. We consider it a weak example because such scalar values serve auxiliary purposes in the course of learning vertex representations and, as scalar real-valued weights, insofar as they are positive, can be encoded in a graph's weight matrix $\boldsymbol{W}$.

By and large, however, edge attributes occupy a peripheral, auxiliary role to the vertices, contextualising or enriching details on vertex relations. In fact, edge attributes can express all manners of relations not wholly independent but expressive beyond the entities represented by graph vertices. There are sundry ways of incorporating edge attributes in the aggregation and transition functions of a convolutional layer. In the terminology of the message-passing network, edge attributes can alter the passage of a message, alter the message itself, or form its own message-passing network.

The ways in which edge attributes may be incorporated into definitions of convolution fall into three kinds: (1) edge attributes for modifying, selecting or indexing message-passing functions; (2) edge attributes that are integrated into a message between two vertices; and (3) edge attributes as a representation that is learned. We discuss each in turn in the following sections.

2.5.6.1 Edge Attributes as Function Selectors

Edge attributes as *edge types* have been used to characterise the kind of interaction between two vertices. Discrete edge types can be used to describe bonds in a molecule, for example, where the bond-type is encoded as a one-hot vector as is very common in molecular graphs (Kearnes et al., 2016; Danel et al., 2020; Maziarka et al., 2020; Ma et al., 2022). The Know-Evolve network (Trivedi, Yang, and Zha, 2020) and edge-conditioned convolution (Simonovsky and Komodakis, 2017) are two examples where edge types are used to select functions.

2.5.6.2 Edge Attributes as Auxiliary Features

Incorporating edge attributes as auxiliary information is common in molecular applications (Gasteiger, Groß, and Günnemann, 2020; Kearnes et al., 2016; Danel et al., 2020; Maziarka et al., 2020; Ma et al., 2022). Vertex features describe properties of atoms and edge features describe the bonding between pairs of atoms. The two sets of features are combined in graph convolution. Modern GCNNs usually lack a mechanism to include edges, as we saw in Sections 2.5.4.1 and 2.5.4.2, and as a consequence the GCNNs applied to molecules make alterations on the usual definitions to include them. As a consequence, molecular models are the source of domain-inspired innovations in GCNN architectures.

On the one hand, edge attributes can directly inform convolution, as in the material graph network (MEGNet). In each layer of the MEGNet, edge features are updated by a function over the edge features and source and destination vertices' features and a set of global features coming from the previous layer. The new edge features are then used to update the vertex representations in another update.

On the other hand, edge attributes can form their own networks whose representations are intermingled with the vertex representations produced in their own network. For instance, the molecular graph convolution (MGC) (Kearnes et al., 2016) incorporates both vertex and edge features into its network. The MGC consists of successive Weave modules that separately convolve vertex and edge features before pleating the features together in four different configurations of vertices and edges. Two transition or message-passing functions are applied separately over each vertex and edge. Combining the features allows the model to learn from complementary information that would otherwise be inaccessible to a conventional modern GCNN.

We have already described the DimeNet (Gasteiger, Groß, and Günnemann, 2020) in Section 2.5.4.2, which includes second- and third-order structural information in the message passing process. The model is unique in that, according to the authors, previous approaches do not include information on torsion and the angles of atomic bonds; these potential energies are usually otherwise inferred from the higher-order interactions learned in multiple graph layers.

2.5.6.3 Edge Attributes as a Learning Goal

Whereas the MGC described above (Kearnes et al., 2016) consists of two parallel and interweaved networks, the line graph neural network (Chen, Bruna, and Li, 2019) learns edge signals on the vertices of a linegraph (Section 2.5.2.5). The linegraph permits the definition of a non-backtracking operator (Krzakala et al., 2013). Unfortunately the use of a linegraph incurs a high memory complexity of $\mathcal{O}(2m)$ linegraph vertices with number of edges that grows considerably depending on the density of the graph with an upper bound of $\mathcal{O}((2m(2m-1))/2)$ for undirected linegraphs. Linegraphs do not scale well to large graphs. Nonetheless, several useful extensions of the linegraph can be inferred. We have already noted in Section 2.5.2.5 that the linegraph represents the second-order structure of a graph. In this vein of thought, the authors point out that a hierarchy of graphs may be constructed from recursive definitions of the linegraph, e.g., the linegraph of a graph's linegraph describes the third-order structure. The question is whether this would lead to denser and denser graphs as one advances up the hierarchy and therefore eventually become computationally unfeasible. Linegraph-like structures have also been deployed in the SeqGNN (Xie et al., 2019). The edge features are structured as a linkage attribute graph, definitionally very similar to a directed linegraph.

Similar to the structure of the MGC, Zhang et al. (2019) designed a graph edge convolutional neural network (GECNN) for the convolution of edge attributes. They design a hybrid network that combines the output of two models, one for vertices and one for edges. The output of the two is combined in two different ways: in the first model, the sequence-level hybrid model, the output is simply concatenated and passed through a dense layer; in the other, the body-part-level hybrid model (BPLHM), it is passed through a shared convolutional layer, a pooling layer and finally a dense layer. They compared these two models against a single-stream GECNN. The results show that all three models outperformed the baseline. The BPLHM perfomed the best of the three and the GECNN performed the worst. It shows that separate edge models, which do not simply incorporate edge attributes into the vertex convolution, are able to obtain significant results on some domains.

2.5.6.4 Lacunae

Notably none of the above techniques considers a situation where the signals exist solely on the edge structure, i.e., the relations of vertices. As a consequence, there is an absense of methods learning vertex-focused tasks where a task must be completed on edges from the edge signals, despite the fact that there is plenty of work on edge-focussed tasks working solely on vertex features, e.g. edge prediction.

Edge features have already been incorporated into GCNNs for the prediction of molecular properties and traffic conditions. Such models might feasibly be extended to, say, physical systems, where there are forces acting at very high degrees, where a higher-order linegraph might represent the interactions well. A fusion of information at different levels might accomplish tasks in such cases rather well, but this requires further investigation.

2.5.7 The Estimation of Graph Structure

In this section we focus on the estimation of graph structure, e.g., the estimation of the presence of interactions not observed in a fixed graph and how one can estimate their presence from raw information.

2.5.7.1 General Theoretical Considerations

Before we continue, it is worth briefly considering what a graph is supposed to be representing. In a given irregular domain one defines a set of discrete points from which one samples information. The discrete points are related to one another in some way defined by the domain, determined by the domain, whether it be physical or geometric distance or some measure of likeness or whatever. These discrete points could also be called entities, for in some domains the discrete points represent indivisible sources of information, such as a sensor. The relationships between entities describe their interactions within the context of a system. Entities are thus represented by vertices and the edges represent their interactions that are significant by some measure. A graph built of these vertices and edges thus represents our knowledge of the domain.

A gap in our understanding, our nescience or ignorance, is represented by missing vertices or edges. One may consider that nescience has two grades in a domain: either (1) a lack of, little or no knowledge of the relations, but full knowledge of the entities, and therefore an incomplete representation the edges, which we term *relational nescience*; or (2) a lack of, little or no knowledge of both the relations and entities, and therefore an incomplete structural representation of both vertices and edges, which we term *formal nescience*. There are several terms used in the literature to describe the process of estimating the structure from raw information: e.g., neural relational inference, graph estimation, graph generation and structure learning. Formal nescience is beyond the scope of this discussion (see Section VI.B of Georgousis, Kenning, and Xie, 2021). For relational nescience, we prefer the term *graph estimation*.

The crucial assumption of graph estimation is that the structure of a domain can be recovered from raw information. For temporal problems in physical system this means learning the fixed rules governing the interaction of physical objects over time (Kipf et al., 2018; Santoro et al., 2017; Watters et al., 2017; Steenkiste et al., 2018; Alet et al., 2019). One might for instance discover the discriminative structure inhered in electroencephalography data (Song et al., 2019; Li et al., 2019b), which would be helpful in a domain where several possible graph connectivities exist to represent brain signals, each capturing a different neurological perspective (Rui, Nejati, and Cheung, 2016). Graph estimation strategies have also been applied to text and object prediction (Henaff, Bruna, and LeCun, 2015), traffic prediction (Wu et al., 2019c), interactions in multi-agent systems (Sukhbaatar, Szlam, and Fergus, 2016), reasoning problems (Santoro et al., 2017; Johnson, 2017), the generation of new molecules (Simonovsky and Komodakis, 2017; Li et al., 2018b; You et al., 2018; Rigoni, Navarin, and Sperduti, 2020; Jin, Barzilay, and Jaakkola., 2018), the mapping of airwaves in the lungs (Selvan et al., 2020), human-action recognition (Kipf et al., 2018) and citation networks (Franceschi et al., 2019), although to the best of our knowledge none has been applied to the estimation of molecular properties.

2.5.7.2 Obstacles to Graph Estimation

There are several obstacles to graph estimation that need to be considered in the design of a graph estimation strategy. Firstly, the computational cost of estimating a graph is a serious impediment. Combinatorially the search-space of all possible n-graphs is 2^{n^2}. This fact limits some approaches to the estimation of graphs with only tens of vertices.

Secondly, a graph is a discrete structure and therefore taken as a whole is non-differentiable. Either the graph estimation therefore becomes a continuous relaxation or it is represented as a probability distribution over the vertices and edges. That the graph is discrete and therefore non-differentiable is not so much a serious impediment, but it requires one to think of certain optimisations to sparsify its weights if a continuous relaxation is used.

Thirdly, one must consider the type of graph one wants to create, directed or undirected *etc.* Constraints on these properties must be incorporated in the graph estimation procedure or the loss function or both. There may be domain-motivated structural constraints, such as those placed on the chemical stability of molecular substructures that would render certain generated graphs invalid (You et al., 2018; Rigoni, Navarin, and Sperduti, 2020). On the one hand, lacing constraints offers its own advantages in limiting the 2^{n^2} search-space; on the other hand, constraints might burden an algorithm with further checks to its capacity.

Fourthly and finally, one must consider what the original graph ought to be. It is an open question how the choice of the initial graph affect graph estimation. There are several elementary choices: (1) start with an empty or complete graph, making no *a priori* assumption of graph structure, from which point edges are either added or subtracted respectively; (2) start with a random graph, like starting with random weights in a model; (3) start with a graph that represents domain knowledge, from which point the graph is manipulated in whatever direction; (4) start with a statistical metric, such as the correlation coefficient of every pair of vertex signals (Henaff, Bruna, and LeCun, 2015; Jang, Moon, and Lee, 2018).

2.5.7.3 Approaches to Graph Estimation

The two approaches to graph estimation that concern us in this thesis are graph estimation via learning the entries of a weight matrix and via interaction functions. Two other approaches to generating graphs are described in Georgousis, Kenning, and Xie (2021), namely the sequential model-based generation of graphs and model-based generation of whole graphs. We will mention one special case, however, of an evolutionary algorithm for network fault localisation (Yang et al., 2022), where what is effectively a graph topology is evolved by the addition of hidden units and connections. It represents a radical departure from the other kinds of graph estimation, but it is closest aligned with sequential graph generation.

Learning the entries of a weight matrix. With a continuous relaxation of the problem, it is possible to learn the entries of the graph's weight matrix end-to-end in the model via backpropagation. Henaff, Bruna, and LeCun (2015) used the parameters of the first layer of an MLP trained on a learning task with vertex-wise labelling to compute a weight matrix. The distance between weights of each vertex pair constitute the entries of the weight matrix. The advantage of using distance is its commutativity, which means the weight matrix is consequently symmetric, allowing it to be used with most spectral methods. Clearly, however, some non-commutative metric would need to be used for computing a directed graph.

In contrast to Henaff, Bruna, and LeCun (2015), Song et al. (2019) and Wu et al. (2019c) proposed a method where the the weight matrix's entries are adjusted directly by backpropagation. The model proposed by Song et al. (2019), like that proposed by Henaff, Bruna, and LeCun (2015), can only learn undirected graphs, owing to the choice of convolutional technique. The model proposed by Wu et al. (2019c) incorporates existing directed knowledge by factorising the graph weight matrix into a known component, representing in- and out-flows, and an unknown component that is directly modelled. The model also makes use of a definition of convolution that does not depend on symmetry in the weight matrix. Yet another technique is present in the model proposed by Li et al. (2019b), where the model also backpropagates over the weight matrix, but learns a separate adjacency matrix at each layer. A graph is then constructed of an average of the edge weights in each layer. The initial graph for this model is a complete graph.

Learning interaction functions. Rather than learn graph representations directly as above, some approaches to graph approximation learn an interaction function. It has the advantage that one need no predefined graph to model interactions. The Interaction Network (IN) (Battaglia et al., 2016) consists of a set of functions—in their case MLP—which jointly learn the interactions of objects in a physical system, notably as different kinds of forces acting on the objects. The Relational Network (Santoro et al., 2017) simplifies the model by modelling edges as a function of vertex attributes, modelled again using an MLP. These approaches are however computationally intensive, since the functions are applied to all n^2 edges. The problem is doubly bad in the IN where there are two edge-wise functions.

The Communication Network (CommNet) (Sukhbaatar, Szlam, and Fergus, 2016) models the interactions in a multi-agent system. The interactions between multiple agents varies with time, so the model is designed to scale to accommodate an increase in the number of agents. It is accomplished by two functions, one to model communication and the other to map vertex attributes from one step to the next. The latter function makes no distinction between the adjacent edges of a vertex. An extension of CommNet, the Vertex Attention Interaction Network (Hoshen, 2017), circumvents the problem by learning an attention vector for each vertex and an additional communication vector on each vertex rather than each pair. The negative Euclidean distance of each vertex pair's attributes, passed through a softmax function, weights the communication vectors. The attention mechanism has several advantages over IN and CommNet. Firstly it reduces the computational expense by evaluating the MLP that models the communication vector n times rather than $n^2 - n$ times. Secondly, unlike CommNet, the interactions are explicitly modelled, taking the burden of modelling instead of a later MLP. The interactions may potentially be better interpreted, too. Thirdly, the attention coefficients can model the higher-order interactions of agents indirectly. Fourthly and finally it permits local modelling of objects and consequently the model needs no predefined graph as an input.

2.6 Applications of Graph Deep Learning

In this section we discuss applications of learning algorithms to three domains: datacentres, traffic-flow and molecules. In each case the structure of the domain is realised easily as a graph. Each has its attendant challenges of course, which we use in the course of this work to evaluate some challenges of machine learning. Our summaries of each area are general but motivate the methods proposed in later chapters.

2.6.1 Localising Link-Faults in Datacentres

In the first section we cover the motivation for machine learning approaches as applied to networks. In the second section we review diagnostic approaches for networks, specifically how they measure network problems.

2.6.1.1 Machine Learning on Networks

A host of machine learning techniques have been applied to all manner of networking problems. Datacentres are one kind of network. The issues and challenges of datacentre management, in particular diagnosing various issues in the datacentres (Gill, Jain, and Nagappan, 2011), are general issues arising in any networks (Mello et al., 2016). Quick and effective diagnosis of network problems is necessary to ensure stability in a network, but the size and complexity of some systems means manual diagnosis is slow. Network visualisation is one way of improving the manual diagnosis of network problems (Pelkonen et al., 2015). To some extent the choice of architecture mitigates, but does not eliminate, network issues. The FAT tree is a popular choice of architecture as it has in-built redundancy in each level of the architecture, which is important for mitigating the deleterious effect of faults on traffic-flow (Leiserson, 1985). An alternative is the Clos network (Clos, 1953). Alternatively a more efficient or refined routing algorithm can be used that routes dynamically in response to traffic conditions (Gálvez and Ruiz, 2013). But again, networks are so large that automation of certain, tedious diagnostic processes using machine learning is an attractive propsect for network engineers. The DARPA dataset, for example, is one dataset that was proposed as a basis specifically for the evaluation of intrusion-detection algorithms (Lippmann et al., 2000). Indeed, machine learning has been otherwise used to generate visualisations for diagnostic purposes (Chircu et al., 2019).

The challenge for conventional machine learning is that no one measure provides a total description of network; however, a suite of complementary metrics is necessary to diagnose network failures (Gill, Jain, and Nagappan, 2011). Moreover, a large number of diagnostic tasks can be automated using machine learning, such as the prediction of traffic volume, traffic classification, congestion control and job scheduling among others (Wang et al., 2018a). The term coined for a framework for machine learning techniques applied to networking problems is *anomography*, a portmanteau of *anomalous* and *tomography* (Zhang et al., 2005). It its simplest form, solving network problems is most simply the solution to the linear equation

$$\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x} \tag{2.94}$$

where $\boldsymbol{b}$ is the vector of link measurements and $\boldsymbol{A}$ is the routing matrix; $\boldsymbol{x}$ is therefore the unknown traffic elements as a vector, which a learning algorithm is to solve. The actual interactions in the network are more complicated, of course. The paper (Zhang et al., 2005) elucidates the various, more elaborate approaches to diagnosing traffic conditions, including spatial anomographical techniques, such as PCA; temporal anomographical techniques, such as auto-regressive integrated moving average, Fourier analysis, wavelet analysis and temporal analysis; and finally inferential anomographical techniques, such as the pseudoinverse solution (of the linear equation above), sparsity maximisation and greedy algorithms.

Beyond these basic analytical techniques, MLP, RF and SVM has been applied to localisation of link-faults (Srinivasan, Truong-Huu, and Gurusamy, 2019), where the RF performed best. MLPs have also been applied to networks for diagnostic purposes. Rafique et al. (2018) applied a three-layered MLP to fault detection. DeepTP (Feng et al., 2018) is a sequence-to-sequence model of LSTM units applied to model cellular traffic. DeepTP also uses an attention mechanism to weight the initial features supplied to the decoder by the encoder. The hierarchical spatial-temporal features-based intrusion detection system (Wang et al., 2018b), applied to intrusion detection, transforms temporal traffic data into two-dimensional images, on which a CNN outputs a sequence of packet vectors that are fed to an LSTM. Ji et al. (2018) proposed a CNN to step through the entries of a log file for log prediction. Xiao et al. (2019) reshape a one-dimensional vector of features from a network intrusion dataset into a matrix, to which they applied a CNN to predict network intrusions. We believe that the CNN models use an inappropriate mechanism to learn the network data. In the first case (Wang et al., 2018b) traffic data is rendered as two-dimensional images. Convolution simply does not properly represent the irregular structure of a network. The second case of stepping through log files (Ji et al., 2018) is less clear in the assumption of structure, but the third case (Xiao et al., 2019), where feature vectors are reshaped into matrices, places the features in arbitrary and potentially uncorrelated relation to one another.

A graph avoids the awkwardness that the CNN imposes on data domains by representing the structure of a network directly. A datacentre is represented as a graph when machines are assigned vertices and edges are drawn between vertices whose machines are directly connected (Section 3.3.1). Undirected graphs have been used to structure a GCNN to model traffic loads on cellular networks (Fang et al., 2018) and anomaly detection in a Internet-of-Things network (Protogerou et al., 2020). Directed graphs have been used as early as 2013 to represent network topology (Gálvez and Ruiz, 2013). The diffusion convolution network, which as we have seen separates flows into in- and out-flows in a directed graph, has been applied to predicting traffic loads on links (Andreoletti et al., 2019). A GCNN has also been applied to predicting cellular traffic (Wang et al., 2019), as well as to modelling and optimising traffic flow (Li, Sun, and Hu, 2020).

A radically different technique uses a deep evolutionary neural network (Yang et al., 2022) to evolve a neural network for fault localisation. This approach, as we briefly mention in the introduction to Section 2.5.7.3, is effectively a graph estimation procedure.

2.6.2 Traffic Prediction

We now consider graph deep learning applied to road traffic prediction. We first consider some graph-based techniques from the literature, followed by a discussion of approaches to graph estimation as applied to traffic prediction.

2.6.2.1 Graph Deep Learning Applied to Traffic Prediction

Various temporal models have been employed to predict traffic conditions. The majority uses recursive neural networks to model the temporal behaviour of traffic and a graph-based convolution to model the spatial behaviour (Li et al., 2018a; Zhao et al., 2019; Li et al., 2019c; Wang et al., 2020; Li et al., 2022; Deng et al., 2022). Alternatively, some authors avoid techniques employing recurrent units, owing to their needing to be unrolled, by using temporal convolution (Yu, Yin, and Zhu, 2018; Ta et al., 2022).

One successful implementation of transformers to traffic prediction is the Traffic Transformer (Cai et al., 2020). The traffic transformer essentially replaces the word embeddings in the original transformer (Vaswani et al., 2017) with graph embeddings computed by a graph convolutional layer. A vector is thus computed from the graph signal at each timestep in the source and target input sequences. The output of the decoder is projected to the shape of the graph, yielding a graph-wide prediction of each timestep. The traffic transformer has attained excellent results on the METR-LA and PeMS-Bay datasets, far exceeding the performance of its baselines.

2.6.2.2 Graph Estimation Applied to Traffic Prediction

The principle of using data to estimate graphs is derived from the assumption that the influence of one point/sensor/road on another is to be discovered in the dynamics inhered in the data (Yu, 2022). Hitherto the graph estimation techniques for traffic prediction can be distinguished according to whether they estimate the adjacency matrix's entries directly or whether they use an attention mechanism to do it indirectly. The separation of the two kinds is crude, but it is a scheme of learning that is often adopted in the literature. The separation occurs in time-series data, too, but it is of a different nature, specifically of stability, which is only a question of degree. Sometimes it is useful in traffic prediction problems to make a distinction between static, stable structures of a network and dynamic, transitory structures. Two graphs are consequently learned that complement one another and therefore yield two different interpretations of traffic data. A static graph is a stable structure, observed over long periods, relatively unchanged with the passage of time. The dynamic graph is less stable and corresponds to ephemeral but consequential changes in structure. In the context of a traffic network, we may conceive of the static (more stable) structure as the roads and its junctions, while the dynamic (less stable) structure will be disruptions, such as road closures. A road closure causes the displacement of traffic, overwhelming nearby junctions or alternative routes through the traffic grid, which the static graph, representing the more stable structures, would not capture. Static structures are only stable relative to a frame of time, therefore a method that is able to adapt to changes in long-term structure is desirable.

One example of direct estimation is found in Guo et al.'s work (2022), where they propose learning a weight matrix directly and adding it to an existing graph, which together supplement the adjacency matrices learned from the data for each timestep of the LSTM.

There are several examples of techniques using attention mechanisms. Tang et al. (2020) used a simple global attention mechanism, such that the attention coefficients are used to fill the entries of the weight matrix, which structures the spatial convolutions, implemented using the GCN, of a spatio-temporal model. There is no restriction on the weight matrix apart from being normalised. Likewise Guo et al. (2019) modified the values of the Chebyshev polynomial terms of a predefined adjacency matrix with values of a global attention mechanism, a spatial attention matrix, computed from the previous hour of traffic data. The same spatial attention matrix is used to modify the spatial convolutions in the three model streams for the three time-segments: hourly-, daily- and weekly-periodic data.

The graph estimator GALEN (Yu, 2022) uses a multi-layer GAT to reconstruct a target sequence from a source sequence; it is not intended as a predictive model, but as an adjunct graph estimator, the structure from which is fed to the traffic predictor. The GAT's attention coefficients are scaled by the vertex-wise reconstruction percentage error at each timestep of the edge's source vertex. The average of the scaled coefficients over all layers and timesteps yields an attention score, which is used to structure the predictor. To make the modelling more efficient by reducing the initial adjacencies, the initial graph to the model is a set of k mutually exclusive subgraphs, constructed by an k-medoid algorithm from the geographic distance of the vertices.

The top s vertices for each vertex according to the attention score are selected to form the new graph; thus the degree of each vertex is fixed and the complexity of prediction restrained. The multi-layer GAT thus allows geographically distant vertices to be joined according to their similarity as adjudged by the attention score.

Several methods use a fusion of two complementary sources of structural information to estimate the graph. Kong et al. (2020) propose two complementary pipelines, one where the model is structured using a static graph, and the other where the graph structure is learned from the data; that is to say, the weight matrix is learned directly. The graph is initialised on the latter branch as a matrix of ones. Likewise Ta et al. (2022) used two parallel streams of to learn the graph, which they refer to as micro- and macro-level learning. The macro-level graph learner supplements a predefined graph structure with a identically sized matrix of directly learned parameters. Its purpose is to supplement the existing graph structure with missing relations. On the contrary, the micro-level graph learner computes edge weights from the dot-product of the pairs of learned vertex representations. Its purpose is to capture temporary changes in graph structure. In contrast to Kong et al. (2020), in Ta et al.'s work the two graphs are combined and used in a temporal model to structure graph convolutions. Zhang et al. (2020) describe a more elaborate model consisting of a global and a local module. Both modules partition the modelling of static and dynamic structures. The static structure is the structure that is consistent for all samples, whereas the dynamic structure varies depending on the input. Chen et al. (2020a) takes a different tact by first separating the data into weekly, daily and hourly streams. Each stream is then passed through a sequence of diffusion convolutions that are augmentations on the original (Li et al., 2018a): an additional global weight matrix is constructed by taking 3-step random walks over a given graph. The forward, backward and global matrices are altered by a learned attention matrix.

A important delineation between the architectures in the literature regards the means by which different graph structures are learned and the information supplied to the graph estimation approaches. The first kind is such methods that fuse the outputs of two distinct models, the result of which is regressed against a learning objective after being passed through an interposing dense layer (Kong et al., 2020; Chen et al., 2020a).

The second kind is the set of methods that start the pipeline with a graph estimation module which computes a graph from the input data in some way and feeds the learned graph to downstream graph convolutional layers (Ta et al., 2022). Methods of the third kind estimates a new graph on the input to each layer of the model (Zhang et al., 2020). Nonetheless the aforementioned approaches all likewise conceptually distinguish between long-term or static structures and short-term or dynamic structures.

2.6.3 Prediction of Molecular Properties

Over the last decade and a half, deep learning has risen to the fore in molecular applications (Chen et al., 2018). Graph-based methods in particular have arisen to compete with the traditional techniques in computational chemistry and biology (Zhang et al., 2022). Two main areas of focus obtain in molecular deep learning: graph generation and molecular representation learning. A range of graph-based deep learning techniques have been proposed for molecular graph generation (Chen et al., 2018; Xu et al., 2019c), particularly attractive in an application that is otherwise very expensive (Chen et al., 2018). Increasingly research is also conducted into graph-based deep learning techniques for molecular representation learning, which is the focus of this section.

Several surveys over recent years have demonstrated that molecular representations learned by deep learning techniques perform on par or better than earlier human-designed rules-based methods and other traditional models (Wu et al., 2018; Faber et al., 2017; Yang et al., 2019; St. John et al., 2019; Atz, Grisoni, and Schneider, 2021), with one exception (Mayr et al., 2018). GCNN are counted under an umbrella of approaches called message-passing neural networks (MPNNs) (Gilmer et al., 2017), a framework we described in Section 2.5.4, consisting of a message-passing phase and readout phase. Part of the advantage of graph-based methods is that they do not rely on hand-crafted features (Shui and Karypis, 2020); indeed, graph-based approaches have obtained state-of-the-art performance with molecular representations learned on only a few features (Chen et al., 2019). Graphs also act as a useful inductive bias on a model (Gilmer et al., 2017; Battaglia et al., 2018; Morgan, Paiement, and Klinke, 2022). Graphs also however embed certain kinds of invariance, which is vital to learning stable and generalisable molecular models (Chen et al., 2019).

On the basis of the GCNN presented by Duvenaud et al. (2015) and presented in Section 2.5.4.2, which first described how molecular fingerprints could be learned in a differentiable algorithm, several extensions and improvements have been produced that address various problems arising with learning on molecules, in particular problems of invariance. Duvenaud et al. (2015) proposed commutative functions as a way to make graph models invariant to vertex/atom ordering. Originally the atoms would have to be ascribed a canonical ordering based on vertex and edges features (Rogers and Hahn, 2010). The Weave network (Kearnes et al., 2016) encodes vertex and edge features in two separate streams, whose outputs are merged together. SchNet (Schütt et al., 2017) introduces continuous functions, which make predictions of molecular energy less sensitive to minor variations in molecule position. GROVER (Rong et al., 2020b) masks substructures in a pre-training stage to improve the model's ability to predict motifs in molecular structures. The heterogeneous molecular graph neural network (HMGNN) (Shui and Karypis, 2020) uses a heterogeneous graph to model the interactions in a molecule between different molecule groups.

Directed graphs serve as a useful inductive bias, too, in describing one-way interactions between atoms. DimeNet (Gasteiger, Groß, and Günnemann, 2020), and the more efficient DimeNet++ (Gasteiger et al., 2020), use bonding angles in a directed graph to encode directional invariance by including bonding angles and rotations in the convolution. Yang et al. (2019) on the other hand use a special definition of convolution on directed graphs to mitigate tottering in the propagation of signals.

In some approaches a global vertex or global feature or some mechanism for propagating features globally is used to enhance molecular descriptions. The MPNN framework introduces a global master vertex that is connected to all vertices in the graph, allowing the vertex representations to travel long distances in the graph (Gilmer et al., 2017). Attention has been used similarly to pass representations locally and globally in the graph. An auxiliary adjacency matrix where the entries correspond to interatomic distances has been used within a transformer (Maziarka et al., 2020). Attention to attend to the whole or parts of the graph. Sun et al. (2021) uses two views of information: local structures of molecules and global relationships between different molecules. The local structures allow for augmentation of molecules without changing molecular semantics. Xiong et al. (2020) propose an attention mechanisms for molecular property prediction that incorporates non-local, intramolecular effects in the attention convolution.

2.7 Summary

At the beginning of this chapter we reviewed the elementary techniques and concepts of machine learning. We saw that the field has been refounded on a new class of learning algorithms, collectively termed deep learning. We discussed the drawbacks of the conventional techniques in deep learning, namely their restriction to regularly structured domains; when these conventional techniques are applied to irregular domains, various transformations are necessary that lead it invalid structural assumptions and the loss of structural information that is otherwise informative to a model. The deficiency of deep learning in this regard motivated our discussion of deep learning on graphs. We presented a range of deep learning techniques on graphs, both spatial and spectral, and considered the various graph-related issues that arise in graph deep learning. When discussing these techniques we repeatedly considered the methods with respect to their application to directed graphs. We then considered two additional challenges on graphs in more depth: learning on the edge structures of graphs and graph estimation. Lastly we described existing work on three application areas: datacentres, road traffic prediction and molecular property prediction. The three application domains supply a unique setting within which to investigate techniques to address these issues. The datacentre is a useful domain on which to develop edge-structured approaches to graph deep learning. The traffic dataset allows us to consider the use of combinations of information to estimate graph structures. The two concerns are then drawn together in the final domain, where the task is to learn on both edge structures and estimate graphs.

Chapter 3

Directed Linegraph for Learning on Edges: Localising Faults on Datacentres

Contents

In this section we present our work on detection of link faults in datacentres. Specifically we use the concept of a linegraph to describe the problem and train a model to identify the location of multiple link faults in a datacentre simulation. The results of this section has been published as two works. Part of the results were published as a conference paper:

> Michael Kenning et al. (2021). "Locating Datacenter Link Faults with a Directed Graph Convolutional Neural Network". In: *Proceedings of the International Conference on Pattern Recognition Applications and Methods*. SCITEPRESS - Science and Technology Publications, pp. 312–320. DOI: 10.5220/0010301403120320.

The full results were published as a journal paper:

> Michael P. Kenning et al. (2022). "A directed graph convolutional neural network for edge-structured signals in link-fault detection". In: *Pattern Recognition Letters* 153, pp. 100–106. ISSN: 01678655. DOI: 10.1016/j.patrec.2021.12.003.

3.1 Introduction

In Section 2.6.1 we considered the range of machine learning approaches that have been applied to network problems. It costs network administrators time and money when network errors occur that inhibit proper functioning of the system. Diagnostic systems are often deployed to assist network administrators in diagnosing the cause of network problems (Gill, Jain, and Nagappan, 2011; Pelkonen et al., 2015; Chircu et al., 2019). Machine learning approaches offer a quick and effective means of modelling the high-order interactions between the machines in a computer networks. To that end various conventional learning methods have been applied (Wang et al., 2018a; Zhang et al., 2005; Srinivasan, Truong-Huu, and Gurusamy, 2019; Ren et al., 2020). Neural methods using MLPs have also been used (Srinivasan, Truong-Huu, and Gurusamy, 2019; Rafique et al., 2018; Feng et al., 2018). There are also CNN-based approaches (Wang et al., 2018b; Ji et al., 2018; Xiao et al., 2019), some of which, as we have seen in Section 2.6.1, force data into grid-like structures and therefore are questionable with respect to the structural assumptions implicit in a CNN (Wang et al., 2018b; Xiao et al., 2019).

Networks are very represented well by graphs, however, and unsurprisingly a range of approaches have appeared in recent year that use graphs to structure learning (Fang et al., 2018; Protogerou et al., 2020), in particular directed graphs (Gálvez and Ruiz, 2013; Andreoletti et al., 2019; Wang et al., 2019; Li, Sun, and Hu, 2020).

Part of the difficulty of detecting faults in datacentres is on the one hand the shear complexity of large networks; on the other is the range of measures that only ever produce a partial explanation of the network at any one time (Gill, Jain, and Nagappan, 2011). Usually a diagnostic system has deployed agents making live measurements on the system. This comes with its own issues because it should not counter-productively burden the switches in the network with packets sent out ultimately to *prevent* traffic problems, called a *probing trade-off* (Arzani, Behnaz et al., 2018). Various systems exist to diagnose network faults, each with their attendant advantages and disadvantages (Guo et al., 2015; Ren et al., 2020; Arzani, 2018).

A diagnostic system alone is not sufficient, however, since the data it produces still needs to be studied before a problem is detected. Consider the 007 system, for example (Arzani, 2018). The diagnostic system consists of agents situated on host machines of a datacentre which monitor for retransmission errors. When a retransmission error is detected, the host issues a traceroute command that finds the likely path taken by the packet. All links found on that route are marked as jointly responsible, called a *blame score*. After some time, the blame scores of all hosts are aggregated and summed. The bad link does not always have the highest aggregated blame score, though; an extra level of processing must occur. Arzani, Behnaz et al. (2018) propose an optimisation algorithm that progressively pares links to discover the subset of links that correspond to the most likely sources of failure. It is possible however to imagine improvements on this algorithm that uses the network structure as a graph to diagnose efficiently the location of link-faults.

007 presents its own troubles in that the blame scores describe the suspected level of fault of links between machines, meaning that the features of the dataset are consequently associated with edges rather than vertices. Additionally problems are presented by the network architecture used by Arzani, Behnaz et al. (2018). In the Clos topology (Clos, 1953) described by Arzani, Behnaz et al. (2018), every pair of machines has two separate flows, upstream and downstream, flowing in opposite directions between the two machines and corresponding to two different wires. This presents two additional issues: (1) the edges are directed and must be separately accounted for; and (2) the edges are the inverse of one another. Added to the fact that, whereas the usual construction of a directed graph on networks associates vertices with machines (Gálvez and Ruiz, 2013), on which a graph signal is most commonly structured, the features from 007 are associated with edges. Moreover the edges are directed and every edge from A to B has an accompanying edge from B to A. The theoretical problems of defining a graph model in this context are thus very interesting.

3.2 Proposed Approach

In the introduction we described a particular problem that arises from the architecture of the network described by Arzani, Behnaz et al. (2018), namely that (1) in light of the usual definition, the signals are structured on a graph's edges, (2) the edges are directed and (3) each edge has an accompanying edge in the opposite direction, which is referred to as an *inverse edge* in a directed graph. To address these challenges, we propose the following:

1. We construct a directed linegraph on the directed graph representing the datacentre. The blame scores are therefore be structured on the vertices of the directed linegraph rather than the edges of a directed graph. Since graph convolution assumes that graph signals are structured on vertices, not edges, this transformation permits us to define graph convolution along the conventional lines described in Section 2.5.4.2.

2. To preserve the directional information, we propose a convolutional technique that distinguishes neighbouring vertex information by its orientation with respect to a focal vertex.

3. We propose accounting for the inverse edge separately from the directed neighbours and the focal vertex.

In Section 3.3.1 we describe how the representation of the datacentre as a directed linegraph. In Section 3.3.2 we formulate the directed graph convolution that we use in the directed graph convolutional neural network (DGCNN) described in Section 3.3.3.

3.3 Methodology

3.3.1 The Datacentre as a Directed Linegraph

We have already described the directed graph and the directed linegraph in Sections 2.5.2.4 and 2.5.2.5 respectively. The directed linegraph $L(G)$ is constructed on an underlying graph G by assigning a new vertex in $L(G)$ to every edge in G. An edge for vertices $\alpha, \beta \in L(G)$ is added to $L(G)$ if their the two underlying directed edges (1) share a vertex and (2) are oriented in the same directed. Such is the algorithm explained by Aigner (1967). Moreover, since the directed linegraph is essentially a graph, the properties of the directed graph outlined in Section 2.5.2.4 obtain equally in the directed linegraph. As we said in Section 2.5.2.5, the linegraph, directed or undirected, represents the second-order structure of the underlying graph. The directed linegraph that represents a datacentre thus represents the link-level, second-order structure of the datacentre.

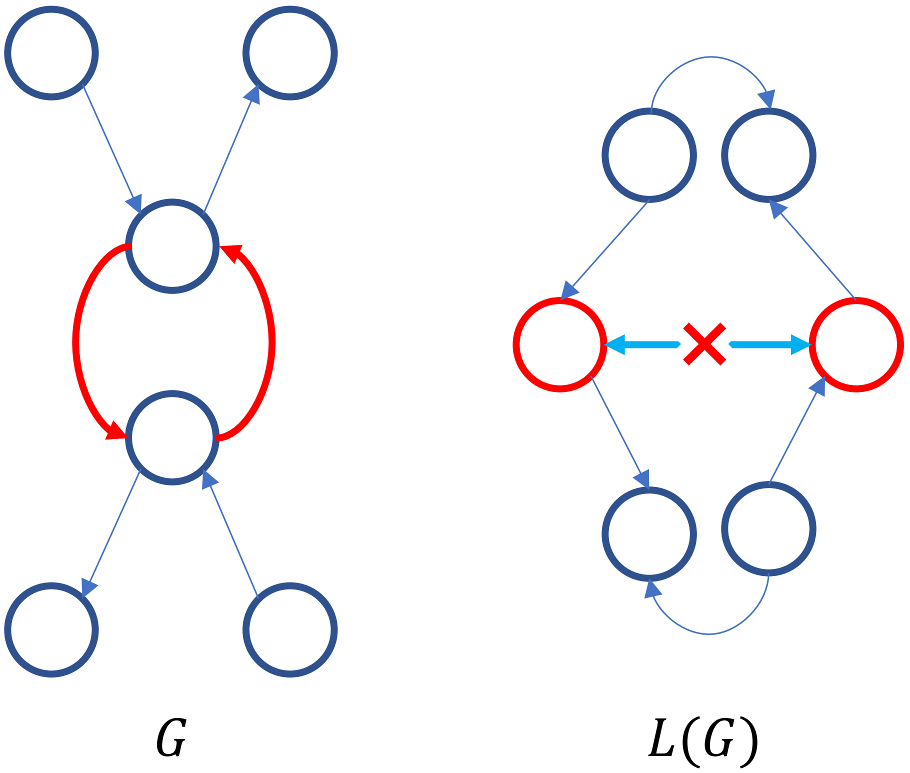


Figure 3.1: If a directed graph (left, G) has a pair of inverse edges, the directed linegraph constructed from it (right, $L(G)$) would contain a pair of inverse edges (represented by the light-blue, horizontal, double-headed arrow). As a consequence, a question arises as to whether the inverse edges in the directed linegraph ought to be excluded altogether, since they lead to problems in the definition of in- and out-neighbourhoods (see Section 2.5.2.4).

A problem however arises from the nature of the underlying graph in this case, namely the inverse edges. Since every edge in the underlying graph has a vertex in the directed linegraph, the two inverse edges in the underlying graph have two vertices in the directed linegraph. How should the two vertices be connected? The problem is illustrated in Fig. 3.1. Strictly, following Aigner's definition, the inverse edges should be joined by two inverse edges in the directed linegraph. A consequence thereof is that the two vertices of the inverse edges in the underlying graph would be in both the in- and out-neighbourhoods of each other, thus occurring twice (see Section 2.5.2.2). The question therefore is whether to consider the underlying graph's inverse edges neighbours *at all* in the directed linegraph, or whether to make a separate account of the inverse edges. In the next section, when we define directed graph convolution, we do just that. For clarity we factorise the adjacency matrix $\boldsymbol{A}$ into its inverse edges $\boldsymbol{B}$; the other adjacencies are simply denoted $\boldsymbol{A}$, overloading the notation.

3.3.2 Directed Graph Convolution

In this section we describe our proposed approach for graph convolution. As we noted above, the directed linegraph for all intents and purposes may be treated as a graph (Section 2.5.2.5). Our definition of directed graph convolution is therefore not restricted to use on directed linegraphs but the broader class of directed graphs. We will therefore be describing the method as applied to a directed graph rather than a directed linegraph.

In Sections 2.5.4.1 and 2.5.4.2 we enumerated various techniques for convolution on graphs. Few of the methods accounted for direction on the graph. We propose a method for convolution on directed graphs, where

$$\begin{aligned}\boldsymbol{h}_{l,x} =& \theta_0 \boldsymbol{h}_{l-1,x} \\ &+ \frac{\theta_1}{d_{\text{in}}(x)} \sum_{y \in \Gamma_{\text{in}}(x)} \boldsymbol{h}_{l-1,y} \\ &+ \frac{\theta_2}{d_{\text{out}}(x)} \sum_{y \in \Gamma_{\text{out}}(x)} \boldsymbol{h}_{l-1,y},\end{aligned} \tag{3.1}$$

where $\theta_0, \theta_1, \theta_2 \in \mathbb{R}^c$ are the learned parameters for the target vertex's signals and in- and out neighbours' signals respectively. The equation may be formulated generally for all graph vertices and multiple output channels as

$$\boldsymbol{h}_l = \boldsymbol{h}_{l-1}\boldsymbol{\Theta}_0 + \hat{\boldsymbol{A}}^\top \boldsymbol{h}_{l-1}\boldsymbol{\Theta}_1 + \hat{\boldsymbol{A}}\boldsymbol{h}_{l-1}\boldsymbol{\Theta}_2, \tag{3.2}$$

where $\hat{\boldsymbol{A}}$ is the row-normalised adjacency matrix and $\boldsymbol{\Theta}_0, \boldsymbol{\Theta}_1, \boldsymbol{\Theta}_2 \in \mathbb{R}^{c \times d}$ are the sets of learned parameters. This formulation does not account for inverse edges, however, so we extend Eq. 3.2 to include inverse edges as an additional term

$$\boldsymbol{h}_l = \boldsymbol{h}_{l-1}\boldsymbol{\Theta}_0 + \gamma_0 \hat{\boldsymbol{A}}^\top \boldsymbol{h}_{l-1}\boldsymbol{\Theta}_1 + \gamma_1 \hat{\boldsymbol{A}}\boldsymbol{h}_{l-1}\boldsymbol{\Theta}_2 + \gamma_2 \boldsymbol{B} h_{l-1}\boldsymbol{\Theta}_3, \tag{3.3}$$

where $\boldsymbol{B} \in \mathbb{R}^{n \times n}$ is the weight matrix of the non-inverse-edges in the graph. γ_0 and γ_1 are flags for the inclusion of in- and out-neighbours in the convolution and γ_2 is a flag for including the inverse edges. By default $\gamma_0 = \gamma_1 = 1$ and $\gamma_2 = 0$. Note that $\boldsymbol{A}$ and $\hat{\boldsymbol{A}}$ do not contain diagonal entries, for the graph it represents has no self-loops.

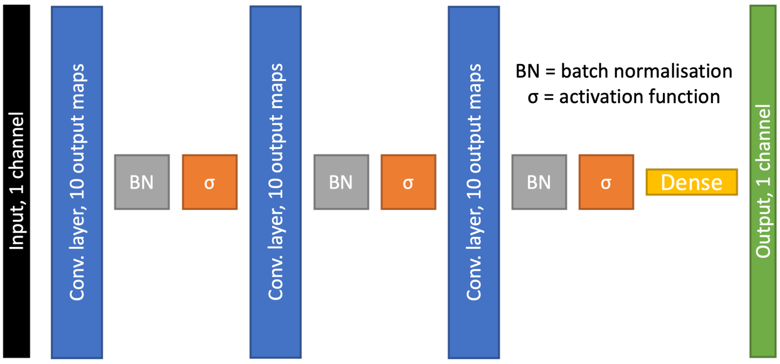


Figure 3.2: Three DGCNN layers are included, whose outputs are passed through a batch-normalisation layer and an activation function, namely leaky ReLU where $\alpha = 0.3$. In the directed graph convolutional neural network with residual connections (DGCNN-R), skip connections are added to the model to mitigate over-smoothing. The illustration is adapted from Kenning et al., 2022.

3.3.3 Directed Graph Convolutional Neural Network

The DGCNN we propose, illustrated in Fig. 3.2, consists of three layers of the directed graph convolution proposed above. The output of each layer is normalised by batch normalisation to centre the outputs of each layer. We use a momentum of 0.99, which in experimentation we found stabilises learning. The batch-normalised output is passed through a leaky ReLU layer, where $\alpha = 0.3$.

3.4 Experiment

The experiments described below were programmed in Python 3.6.6 and trained in Tensorflow 2.4.1. The GraphSAGE model is an implementation from the library Spektral (Grattarola and Alippi, 2020) version 1.0.4.

3.4.1 Dataset

Our model was trained on a dataset generated by the 007 simulator (Arzani, 2018) developed to test the 007 diagnostic system Arzani, Behnaz et al. (2018). The diagnostic system is designed to discover link-failures in a Clos network topology (Clos, 1953). The programme simulates the passage of packets in a datacentre in thirty-second runs. In total we ran 2,880 30-second-long simulations, representing 24 hours of simulations.

There are several kinds of links in the datacentre simulation owing to the Clos network topology. The simulated datacentre has four device types: host machines at the lowest level, to which users have access; top-of-rack (ToR) switches, to which the

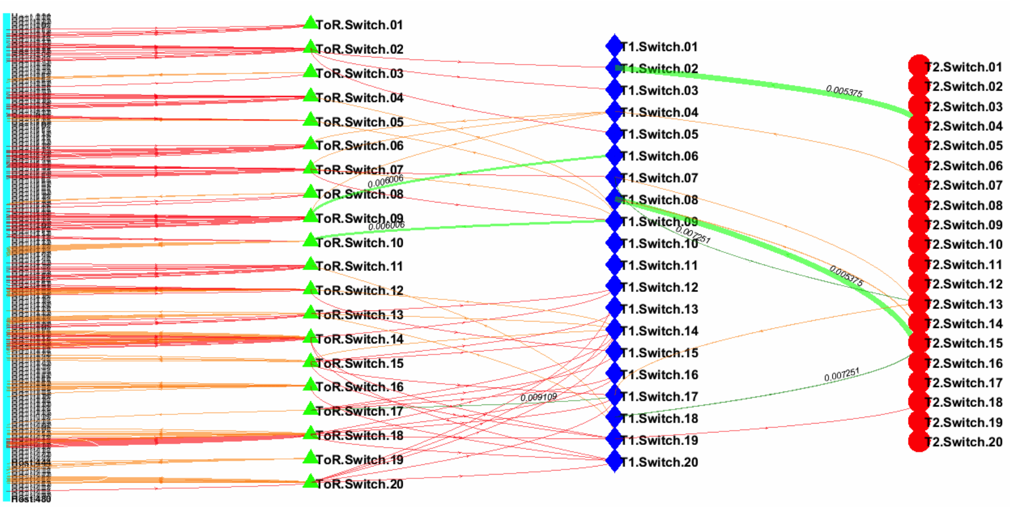


Figure 3.3: The illustration here describes the blame scores on the links of the dataset after one thirty-second simulation. The green lines indicate the faulty links; the numbers they are labelled with give their probability of dropping their packet. In this example there are seven link faults. The dark-green lines represent uplinks; the light green lines represent downlinks. The red lines indicate functioning links. The orange lines represent functioning downlinks and the red lines represent functioning uplinks. This figure has been adapted from Kenning et al. (2021).

host machines are connected; T1 switches, to which the ToR switches connect; and finally the T2 switches, to which the T1 switches connect. The Clos network used by Arzani, Behnaz et al. (2018) consists of two pods of T1 switches, ToR switches and host machines. The ToR and T2 switches of a pod are fully connected. Each host machine is connected to a single ToR switches. Each T1 switch connects to two T2 switches. Each T2 switch connects to a maximum of one T1 switch from each pod. he outcome of one simulation is supplied in Fig. 3.3.

We are not interested in the link faults on the host machines: it is trivial to diagnose link faults at that level because there is a one-to-one pairing between host machine and ToR switch. We are interested in the four link types that are more difficult to diagnose: the downstream T1–ToR links type 1; the upstream ToR–T1 links, type 2; the upstream T1–T2 links, type 3; and the downstream T2–T1 links, type 4. For every upstream link joining a pair of devices in one directed, there is another link joining the pair in the opposite direction.

In our simulations we set the datacentre to have 10 T2 switches and two pods of 10 T1 switches and 10 ToR switches. Each ToR switch is connected to 24 hosts. In total there are therefore 540 devices and 1,440 links. As we stated above, each simulation is run for 30 seconds. A subset of the datacentre links, between 2 and 10, are selected to be faulty and drop 1% and 10% of the packets that pass through; that rate is randomly selected

Table 3.1: A comparison of the models we are evaluating in the experiments. This table is adapted from the journal paper presented by Kenning et al. (2022).

Name	Type	Directed	Inverse edges	Excluding neighbours
DGCNN	Spatial	✓	—	—
DGCNN-R	Spatial	✓	—	—
DGCNN/I	Spatial	✓	✓	—
DGCNN-out	Spatial	✓	—	✓
DGCNN-in	Spatial	✓	—	✓
UGCNN	Spatial	—	—	—
UGCNN/I	Spatial	—	✓	—
UGCNN+I	Spatial	—	✓	—
GraphSAGE	Spatial	—	—	—
P-F model	Spectral	✓	—	—
Fusion model	Spatial	✓	—	—
MLP	—	—	—	—
RF	—	—	—	—

in that interval and fixed for the duration of the simulation. To add noise, healthy links also drop packets but at a much lower rate of 0.01% of packets.

There is a high inherent degree of class imbalance that must be managed in the learning algorithm, a problem we described in Section 2.3.5. The datacentre has 1,440 links and the number of failures is $k \sim U(2, 10)$, which has an expectation of $\mathbb{E}(k) = 2 + \frac{10-2}{2} = 6$. The expected class imbalance is therefore $\rho = \frac{\mathbb{E}(k)}{1{,}440-\mathbb{E}(k)} = 4.18 \times 10^{-3} \ll 1$, where 1 would indicate equally balanced classes. We describe our remedies for the class imbalance in Section 3.4.3.

3.4.2 Baselines and Comparisons

To evaluate the DGCNN architecture we compare it with seven architectural variations on itself, differing only in the implementation of the convolutional layers. A table comparing each model in the experiments is presented in Table 3.1.

The DGCNN is run in the experiments as several variations. Recall that the parameters $\gamma_0, \gamma_1, \gamma_2$ in Eq. 3.3 control the inclusion of in- and out-neighbours' and the inverse edges' signals in the convolution. By default $\gamma_0 = \gamma_1 = 0$ and $\gamma_2 = 1$, but we can create three additional variants on the DGCNN by turning the flags off. When $\gamma_0 = \gamma_2 = 0, \gamma_1 = 1$, the in-neighbours' signals are excluded from the convolution; we call this variation the DGCNN-out. When $\gamma_1 = \gamma_2 = 0, \gamma_0 = 1$, the out-neighbours' signals are excluded from the convolution; this variation is called the DGCNN-in. When $\gamma_0 = \gamma_1 = \gamma_2 = 1$, the inverse edges' signals are included; this variation is called the DGCNN/I.

As we noted in Section 2.5.5, over-smoothing in graph models begins to manifest itself after two layers (Kipf and Welling, 2017; Wu et al., 2019a). Since the DGCNN has three layers, we test a variant that includes a skip connection between the input and output of each layer, which we term the directed graph convolutional neural network with residual connections (DGCNN-R). The output $\boldsymbol{z}_l$ after the batch normalisation β and non-linearity σ at layer l is therefore

$$\boldsymbol{z}_l = \sigma \circ \beta \left(\boldsymbol{h}_l\right) + \boldsymbol{h}_{l-1}\boldsymbol{\Theta} \tag{3.4}$$

where $\boldsymbol{\Theta} \in \mathbb{R}^{c \times d}$ is the learned projection matrix of the original signal and $\boldsymbol{z}_l$ is the input to the next layer; c and d are respectively the number of input and output channels. This is necessary in cases where the input dimensionality is different from the output dimensionality. The output following the skip connection is passed to the next layer of the network.

We also reformulate the DGCNN to remove the factorisation of the neighbourhoods into two groups. The DGCNN thus loses its distinction of in- and out-vertices; instead they are collapsed together and the model becomes undirected. In effect the redefinition means that Eq. 3.3 is redefined as

$$\boldsymbol{h}_l = \boldsymbol{h}_{l-1}\boldsymbol{\Theta}_0 + (\hat{\boldsymbol{A}}^\top + \hat{\boldsymbol{A}})\boldsymbol{h}_{l-1}\boldsymbol{\Theta}_1 + \gamma\boldsymbol{B}h_{l-1}\boldsymbol{\Theta}_2, \tag{3.5}$$

where γ is a flag for the inclusion of the inverse edges as a separate term. Where $\gamma = 1$, the undirected model is called the undirected graph convolutional neural network including the inverse edge separately (UGCNN/I); where $\gamma = 0$, thus excluding the term for the inverse edge, the model is simply called undirected graph convolutional neural network (UGCNN). Collapsing the in- and out-neighbourhoods into one group allows the inverse edges to be included in the neighbourhood term without doubly accounting for the inverse edges. Another variation of the UGCNN model therefore includes the inverse edges together with the neighbourhoods, as per a strict adherence to Aigner's definition of linegraphs (Aigner, 1967), which formulated as

$$\boldsymbol{h}_l = \boldsymbol{h}_{l-1}\boldsymbol{\Theta}_0 + (\hat{\boldsymbol{A}}^\top + \hat{\boldsymbol{A}} + \boldsymbol{B})\boldsymbol{h}_{l-1}\boldsymbol{\Theta}_1, \tag{3.6}$$

which we simply call the undirected graph convolutional neural network including the inverse edge merged (UGCNN+I).

We also evaluated the DGCNN against five baselines. The first three baselines are layers for graph convolution proposed in the literature that have been described in Sections 2.5.4.1 and 2.5.4.2. The first is an undirected, spatial technique, GraphSAGE Hamilton, Ying, and Leskovec (2017b), for which we use the mean aggregator. The second is the Perron-Frobenius spectral directed model (P-F model), a directed, spectral technique. The third is the DGCN, a directed, spatial technique, which we called the Fusion technique in this section to disambiguate it further from the DGCNN. We wish to compare the efficacy of our directed technique against two directed techniques in the literature and one undirected technique.

For the two additional baselines we include an MLP, essentially the DGCNN where the layers are replaced with dense, fully-connected layers. The MLP allows us to validate the use of graph-based models at all. The other is an RF, against which we evaluate the deep learning models.

3.4.3 Experimental Conditions

Each model was trained for 50 epochs. The initial training rate is 0.1 and is decayed to 1×10^{-7} during training using cosine decay. The loss-function is the binary cross-entropy of the model output and the ground-truth. We make several modifications to the loss function to mitigate the deleterious effect of the high class-imbalance on learning (see Section 3.4.1).

Firstly, we apply a threshold, the lowest blame-score among the positive samples in the training set, to exclude the loss on vertices with blame-scores lower than the threshold. The risk is that the model misses the few samples in the test set where the blame-score of a defective link falls below the threshold. Though, given the gap between the failure rate of the healthy and faulty links (see Section 3.4.1), the risk is very low. We also exclude the loss on links to the Host machines since no fault ever occurs on those links in the simulations.

Secondly, we weight the loss on each vertex by the balanced odds-ratio of positive/faulty to negative/healthy links. The weight of a positive/faulty link is $(\mathrm{neg} + \mathrm{pos})/2\mathrm{pos}$ and $(\mathrm{neg} + \mathrm{pos})/2\mathrm{neg}$ for negative/healthy links, where pos and neg are the counts of positive and negative samples above the threshold in each simulation.

Thirdly, we set the bias of the final layer such that the model outputs reflect the prior distribution of the positive links. The log-ratio of positive to negative links was computed

$$b = \ln\left(\frac{\mathrm{pos}}{\mathrm{neg}_{\mathrm{total}}}\right) , \tag{3.7}$$

where $\mathrm{neg}_{\mathrm{total}}$ is the count of negative links, ignoring the threshold. The sigmoid layer consequently yields the prior:

$$\sigma(b) = \frac{\mathrm{pos}}{\mathrm{neg}_{\mathrm{total}}}. \tag{3.8}$$

These conditions largely cannot apply to the RF. The RF we used in our experiments consists of 100 estimators, each estimator constructed on a subsample as large as 10% of the training set. The classes are balanced in each tree according to their proportions in the tree's subsample.

3.4.4 Metrics

We compare the model performances on the test set across three primary measures: precision, recall and F_1-score. Precision describes the proportion of links classified as faulty that are actually faulty. Recall describes the proportion of faulty links correctly classified as faulty. Ideally the model would maximise both; the F_1-score describes the balance between the precision and the recall. These measures, in particular the F_1-score, help us in the discussion to compare the DGCNN with the comparison models.

We use a secondary measure, McNemar's test (McNemar, 1947), specifically Edwards' correction of the test (Edwards, 1948), to compare directly the predictions of two different models:

$$\chi^2 = \frac{(|n_{\mathrm{sf}} - n_{\mathrm{fs}}| - 1)^2}{n_{\mathrm{sf}} + n_{\mathrm{fs}}}, \tag{3.9}$$

where n_{sf} is the number of samples correctly classified by model 1 (hence "s" for "success") that were incorrectly classified by model 2 (hence "f" for "failure"), and n_{fs} is the number of samples correctly classified by model 2 that were incorrectly classified by model 1. The value χ^2 is approximated by a chi-squared distribution, and therefore we may use a binomial test to evaluate the null hypothesis, that the performances of the two models are equal. In our analysis, model 1 is always the DGCNN, i.e., the left side, to which we compare the outputs of every other model in the experiment.

3.5 Results

The results of our experiments are presented in Table 3.2. With the exception of DGCNN-in and DGCNN-out, the graph-based models attain better F_1-scores than the non-graph models, the MLP and the RF. With the exception of DGCNN-in and DGCNN-out the variants of the DGCNN attain higher F_1-scores than the variants of UGCNN and the undirected GraphSAGE. The two directed methods from the literature, the P-F model and the Fusion model, have the worst F_1-scores. Whether this is owed to the natures of the models or faulty implementations, we do not know. The parameter count alone cannot explain the differences in performance between the DGCNN variants and the UGCNN variants. The relationship between mere parameter count and performance is not clear-cut anyway in general (Neyshabur et al., 2017).

It is worth noting a subtle trend in the size of the n_{sf} and n_{fs} for the McNemar's test statistic in comparison to the reference model. The chief trend in these numbers lies in the number of samples that model 1, the DGCNN correctly predicts that the other models do not. The numbers counts the samples that one model can identify but the other cannot. It appears that a model that learns to classify one set of observations correctly cannot correctly classify another set. It is not that it is impossible, since the other model is capable of correctly identifying the other subset. The trends suggest therefore that an ensemble of such predictors might pick up on those unsuccessful predictions. Specifically an ensemble of graph-based methods might work better, since the ensemble of trees in the RF fails on average on far more samples ($n_{\text{sf}} = 1043.0$) than it succeeds ($n_{\text{fs}} = 553.2$) in comparison to the DGCNN.

It is also worth noting that the difference in performance observed between the reference model, the DGCNN, and two directed models, DGCNN-R and the DGCNN/I, all undirected models, and GraphSAGE are statistically insignificant, indicated by the p-value higher than 0.05. In spite of the statistical insignificance of the performance differences, it is still possible to draw some conclusions about the different levels of performance of the models.

A random forest versus deep learning. By F_1-score the RF is outperformed by all DGCNN variants, all UGCNN variants and the MLP. The RF does have the second-highest recall score, but it comes at the expense of a low precision score in comparison to the better-performing deep learning techniques.

Graph deep learning versus non-graph deep learning. The MLP, representing non-graph deep learning, has a higher F_1-score than the DGCNN-in and DGCNN-out. This is unsurprising since DGCNN-in and DGCNN-out essentially exclude signals from the convolution. The other DGCNN and UGCNN variants outperform MLP, suggesting that the graph structure is helping the models to learn better. In other words, the inductive bias inhered in the graph structure is advantageous for a model.

Table 3.2: The results on the models are averaged over the five folds of the test set. The F_1-score, precision and recall are rounded to five decimal places, and the McNemar's test statistics to two decimal places. The p-value is rounded to five decimal places. The reference model for the McNemar's tests is the directed model with the residual connections because it had the best performance by F_1-score. The McNemar's test-statistic for the reference model is therefore not given. The parameter counts of each model are given for the perspective they give on capacity, training and inference times, *etc.* This table was adapted from the paper by Kenning et al. (2022).

Model	No. Params.	F_1-score	Precision	Recall	n_{sf}	n_{fs}	χ^2	p-value
DGCNN	731	$\mathbf{0.77985 \pm 0.00787}$	0.72250 ± 0.01407	0.84853 ± 0.03682	—	—	—	—
DGCNN-R	941	0.77761 ± 0.01215	$\mathbf{0.72577 \pm 0.01237}$	0.83921 ± 0.04552	418.0	416.4	1.01	0.40969
DGCNN/I	941	0.77628 ± 0.01021	0.72381 ± 0.01409	0.83870 ± 0.04299	530.8	516.8	0.57	0.51480
DGCNN-out	521	0.71514 ± 0.05162	0.66750 ± 0.07296	0.77880 ± 0.08151	1139.6	643.2	175.28	0.00000
DGCNN-in	521	0.74134 ± 0.03282	0.70836 ± 0.04189	0.79203 ± 0.11719	789.0	560.6	40.10	0.00000
UGCNN	731	0.77675 ± 0.01645	0.71655 ± 0.03239	0.85514 ± 0.07976	549.2	513.8	2.30	0.31816
UGCNN/I	941	0.76341 ± 0.01571	0.73282 ± 0.02456	0.80067 ± 0.06326	667.4	612.2	3.30	0.28927
UGCNN+I	731	0.74535 ± 0.06347	0.67650 ± 0.09306	0.84210 ± 0.07511	883.6	522.4	177.17	0.19262
GraphSAGE	521	0.70562 ± 0.07261	0.61386 ± 0.10334	0.83998 ± 0.01248	1369.6	543.6	467.21	0.16386
P-F model	521	0.11800 ± 0.05497	0.06662 ± 0.03498	0.59747 ± 0.04498	34123.6	696.6	32091.83	0.00000
Fusion	1243	0.11580 ± 0.07609	0.06543 ± 0.04822	0.63700 ± 0.08147	40394.0	828.6	37980.77	0.00000
MLP	311	0.74303 ± 0.00603	0.64016 ± 0.00532	$\mathbf{0.88539 \pm 0.01167}$	975.6	514.4	144.34	0.00000
RF	100	0.73698 ± 0.00463	0.63952 ± 0.00576	0.86954 ± 0.00741	1043.0	553.2	151.24	0.00000

Directed versus undirected graph models. The DGCNN and DGCNN-R outperform the variants of the UGCNN by F_1-score. The recall scores of the UGCNN variants are higher than the DGCNN and DGCNN-R, but the precision is poorer. Additionally, the directed models were generally more stable than the undirected models. Recall that the only difference between the UGCNN and the DGCNN is the adjecency matrix and the additional term. We may therefore conclude that the improvement in performance is owed to the factorisation by vertex incidence in the model. This conclusion is not firm, however, since the difference between the best UGCNN variant and the best DGCNN variant is not statistically significant.

Excluding neighbours. We have already seen that excluding the in- and out-neighbours in DGCNN-in and DGCNN-out respectively so markedly inhibited their performance that even the MLP attained better results. The exclusion of out-neighbours is more deleterious than the exclusion of in-neighbours, which suggests that the in-neighbours are more informative than the out-neighbours. Moreover the stability of the models was vastly undermined.

Inverse edges. The signals on the inverse edges, when included, impart no clear advantage on the DGCNN. The DGCNN/I performed worse than the DGCNN by F_1-score, and moreover the model was less stable. The same trend is apparent in when comparing the UGCNN with the UGCNN+I or UGCNN/I. The inverse edge worsens the performance far more when it is included with the neighbours, as in UGCNN+I, as opposed to when it is included as a separate term in UGCNN/I. Interesting, although the stability of the UGCNN/I is barely different from the UGCNN, whereas it is worsened between the DGCNN and DGCNN/I, the stability of the UGCNN+I is nearly four times worse than the UGCNN. Altogether, these results suggest the inverse edges' signals are deleterious to the performance of the model in whatever way they are included.

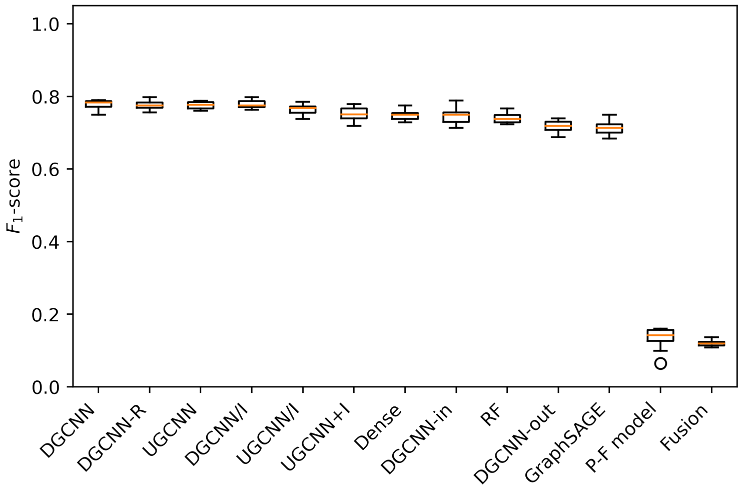
Spatial versus spectral graph-based approaches. The P-F model and Fusion models perform by far the worst of all the models. We cannot exclude the possibility that our implementation of the P-F model and Fusion models were at fault. GraphSAGE is the best performing of the three models from the literature, the implementation of which comes from the Spektral library (Grattarola and Alippi, 2020). It does not attain the same level of performance as even the UGCNN variants, however, which does not distinguish in the model between the focal vertex's features and the out-neighbours' features. This suggests that factorising the convolution into the focal vertex's signals and neighbouring signals provides the model with an interpretative advantage.

Performance stability over link-types and number of failures. We visualised the performance of the models with respect to the F_1-scores in Fig. 3.4. In Fig. 3.4a the test simulations were split into nine groups according to the number of failures present in each simulation. The average F_1-score was then computed within in each group, and the average of the group averages was plotted for each fold. Across all folds, the performance of all the models varies little with respect to the number of failures in the simulation, although the P-F model has a large spread of average.

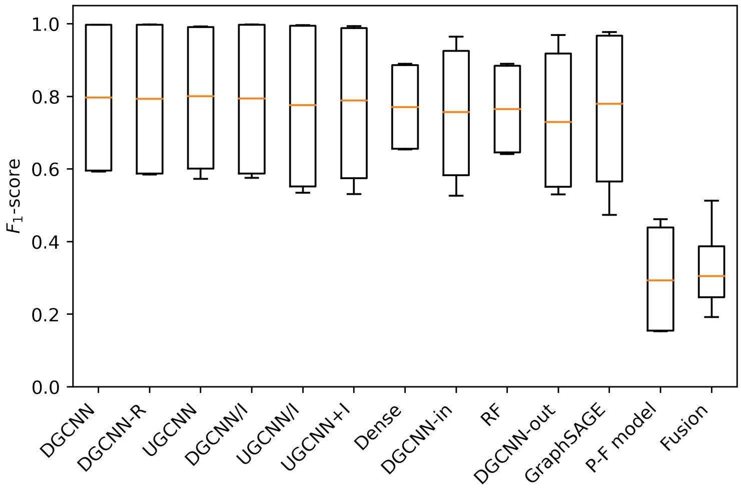
In Fig. 3.4b for each fold we masked for each edge-type and computed the masked F_1-score over all test simulations. We then computed the average F_1-score over the folds for each edge-type and plotted the results as a box-plot. We can see that there is a great degree of variance in the results across models. Further analysis showed that the models struggled particularly to localise fault on the type-3 (T1–T2) and type-4 (T2–T1), the links that reside furthest from the host machines. We suspect that the connectivity of these links, and therefore the quantity of information propagated to those vertices, is responsible. Type-3 links have 11 in-neighbour and 1 out-neighbours and type-4 links have 1 in-neighbour and 11 out-neighbours. Type-1 and type-2 links on the contrary have 11 in-neighbours and 33 out-neighbours and 33 in-neighbours and 11 out-neighbours respectively. The sparsity of the connections between T1 and T2 is the cause of this discrepancy in comparison to those connections between T1 and ToR.

Moreover, a greater number of the type-3 and type-4 links in comparison to the number of type-1 and type-2 links, are likely to be suspected as a source of link-faults. The upstream links from the ToR switches are fully connected to the T1 switches. Although each T1 switch connects to two T2 switches, the host machines are still indirectly connected to every T2 switch at two degrees, as the ToR switches and T1 switches are nonetheless fully connected. This increases the tendency that type-3 and type-4 switches are falsely implicated in a failure, decreasing the usefulness of the blame scores, which in turn renders the localisation of link-faults more difficult, which explains the variance in the results visualised in Fig. 3.4b.

Additionally, when the faulty links are selected uniformly randomly, the proportion of failures on the type-1 and type-2 links (there are type-1 and type-2 links) outnumber the failures on type-3 and type-4 links (there are type-3 and type-4 links). The multiple levels, and therefore several propagation steps, do not appear to make up for the sparsity of information.



(a) Number of failures.



(b) Link-types.

Figure 3.4: In (a) for each model on each fold we grouped the simulations by the number of failures and computed the F_1-scores. In (b) for each model on each fold we masked for each edge type and then computed the F_1-scores. In both cases we average the scores across the folds. The box-plot shows the average, first standard deviation and 25th and 75th percentiles of those F_1-scores across the folds. Circles are outlier samples. The models in the visualisation have been ordered on the x-axis by the F_1-score in Table 3.2. These figures have been adapted from Kenning et al. (2022).

3.6 Summary

The most important finding of these experiments is that interpreting neighbouring signals according to direction aids link-fault localisation. This is an important validation of the directed graph convolution that we propose in Section 3.3.2. An interesting point for future examination is whether randomly dividing the neighbours into two neighbourhoods, irrespective of direction, bestows a similar advantage. It is not clear whether the model is benefiting from factorising direction or just *any* division of the data into two parts, which allows the model to make finer distinctions. This question is particularly pertinent when considering the DGCNN versus DGCNN/I and the UGCNN versus UGCNN/I and UGCNN+I. We suspect that an arbitrary division of the neighbours irrespective of direction would not help the model, as the power results on UGCNN suggest, but it would be interesting to study that experimentally. The exclusion of either direction's information is even more harmful to localisation performance.

A second area of future work would be to study experimentally the ways in which the discrepancy between the F_1-scores of different link types may be addressed. One possible way forward is to manipulate the loss function further to balance the learning on the link failures within each simulation.

Finally we have demonstrated that in certain domains, in particular datacentres, it is necessary to disconnect vertices that by the strict construction of linegraphs would otherwise be connected. The inverse edge's signals contribute deleteriously to the model performance. The structure of the datacentre, where the routing table would exclude the passage of packets down the link in the opposing direction, motivates the exclusion of inverse edges.

The ecological validity of these approaches applied to a real datacentre is in question, however. Although the motivation of these experiments was to study the theoretical viability of the DGCNN, it is an open question how the approach would translate to a data from a real, non-simulated datacentre.

Chapter 4

Graph Estimation on Directed Graphs and Directed Graph Attention: Predicting Traffic Conditions

Contents

Part of the results in this chapter are to be published as a workshop paper:

> Michael Kenning and Xianghua Xie (2023). "Attention-based Graph Estimation and Directed Convolution for Prediction of Traffic Conditions". In: *Proceedings of the 10th Internatinal Workshop on Deep Learning on Graphs*

4.1 Introduction

Traffic prediction is one task to which graph deep learning has applied: the interactions in a traffic network are complex and the relations between the entities in traffic datasets are highly irregular. Graphs are often constructed by leveraging knowledge of the domain to render the graph structure. In traffic networks it is no different; the structure is computed based using a heuristic that measures, for instance, the geographic distance between sensors on the traffic network. Such a way of structuring a traffic network, specifically on the METR-LA and PeMS-Bay datasets, was presented by Li et al. (2018a). The distances between all sensors having been computed, a Gaussian kernel (Shuman et al., 2013) is applied to each vertex with respect to all other sensors, and all transformed sensors more than a given distance from one another are excluded from the graph. The same graph has been used in later works (Wu et al., 2019c; Kong et al., 2020; Ta et al., 2022) and similar constructions using a Gaussian kernel have been used elsewhere on different datasets (Yu, Yin, and Zhu, 2018). Alternatively, some data-driven approximations of the graph structure are made by measuring correlations between points of data in the graph Henaff, Bruna, and LeCun, 2015; Jang, Moon, and Lee, 2018. In any case, the predefined graph structures are fixed throughout learning, however, and cannot adjust to ephemeral changes in the traffic structure.

An improvement on predefined, fixed graphs would be to learn the graphs directly from the data. If the motivation for deep neural networks was the automatic learning of feature representations directly from data rather than learn on features computed by intermediary heuristics Section 2.4, it follows that the same logic may be extended to graph structures: If features may be automatically learned from data, its structure can be learned, too. Indeed, increasingly, graph structure is being learned end-to-end in the model (Georgousis, Kenning, and Xie, 2021), which in Section 2.5.7 we termed *graph estimation*.

In Section 2.6.2 we discussed the graph estimation techniques present in the literature that have been applied to the problem of traffic estimation. We saw that approaches to graph estimation on traffic datasets distinguish between the long-term or static structures and the short-term or dynamic structures, each of which is assumed to be present in the given dataset (Kong et al., 2020; Ta et al., 2022; Zhang et al., 2020; Chen et al., 2020a). We propose a technique that similarly reckons with the data according to its periodicity—weekly, daily, hourly—but it uses two compositions of the periodicities to discover static and dynamic structures in the data. The graph estimation procedure we propose is described in Section 4.3. We evaluate our approach on the METR-LA and PeMS-Bay datasets in Section 4.4.

4.2 Proposed Approach

In Section 4.1 we described the problem of graph estimation for traffic prediction. We saw namely that traffic data can be understood as distinct flows of information, from which a model can learn different kinds of structures. In this chapter we propose the following:

1. An approach to graph estimation that fuses the long-term, static graph structure and short-term, dynamic structure learned from traffic data. The static structure is learned from a composition of cyclical information from the previous week, day and hour relative to the input, hence from long-term traffic data. The dynamic structure is learned only from the previous hour and, at training time, the target sequence, hence the short-term data.

2. A new directed attention mechanism for graph convolution. The directed diffusion used in the Traffic Transformer proposed by Cai et al. (2020) separates the signals around a vertex into two distinct flows, but it uses an isotropic kernel over each neighbourhood. The directed attention mechanism we propose learns attention coefficients for the two distinct flows, thus an anisotropic kernel.

4.3 Methodology

The task of the traffic model is to predict traffic conditions on a road network. The model must predict an hour-long target sequence from three historical hour-long sequences of traffic data. The three historical sequences record traffic conditions one hour, one day and one week prior to the target sequence. The Traffic Transformer (Cai et al., 2020) consists of an encoder and decoder. The encoder is fed the three historic sequences; the decoder is fed the target sequence at training time as well as the embedding from the hidden layers of the encoder. It is posited that the historical sequences also encode weekly, daily and hourly cyclical information of which the encoder can avail itself in learning patterns in traffic conditions, information which is then passed to the decoder for prediction.

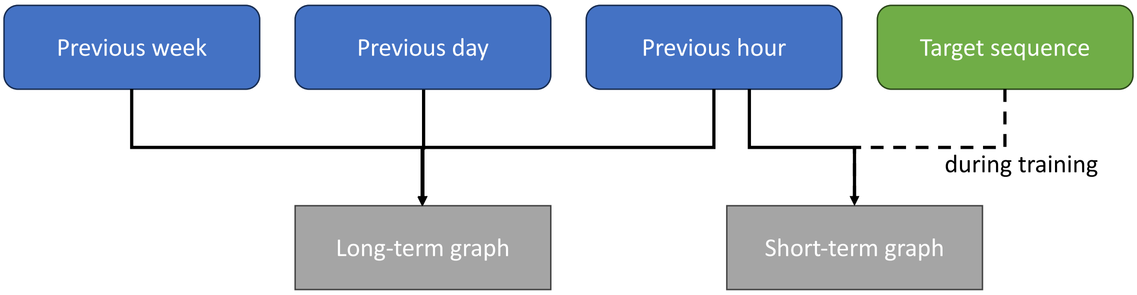


Figure 4.1: The graph estimation process yields two graphs: a long-term graph, estimated from the weekly-, daily- and hourly-periodic traffic sequences, representing the more stable structures in the traffic network; and a short-term graph, estimated from the hourly-periodic and target traffic sequences, representing the ephemeral structures in the traffic network.

All the sequences are structured as graphs. Each sensor on the network corresponds to a vertex in the traffic graph. The interrelation of the vertices describes the structure of the graph, which is constructed using a thresholded Gaussian kernel in the original paper; we describe the construction in Section 4.3.2. The description of the graph structure, represented as a weight matrix, is fed to the initial modules of the Traffic Transformer, which use graph convolution to propagate information across the graph. Of course, we may replace the description of the graph structure supplied to the graph convolutions in any way we like. In this section, we describe how we use the sequences to learn different graph structures from the data.

The model we propose thus consists of two successive phases: A graph is first estimated as a weight matrix $\overline{\boldsymbol{W}}$, a weighted combination of an estimated static/long-term graph represented as a weight matrix $\overline{\boldsymbol{W}}_{\mathrm{stat}}$ and an estimated dynamic/short-term graph represented as a weight matrix $\overline{\boldsymbol{W}}_{\mathrm{dyn}}$. The static graph, encoding long-term, more stable structures in the traffic data, is estimated using the three historical sequences, whereas the dynamic graph, encoding short-term, ephemeral structures, is estimated from the hourly historical sequence, plus the target sequence during training. Figure 4.1 describes which sequences are used to predict which graphs. The estimated graph is then used in the Traffic Transformer architecture to structure the graph convolution.

In the rest of this section we describe our approach to graph estimation. In Section 4.3.1 we formulate the problem of traffic prediction, describing the form of the input data. In Section 4.3.2 we describe the predefined graph used to structure the traffic data in conventional techniques that do not use graph estimation. The drawbacks of the conventional structure of the graph motivates our choice of the graph estimation procedure, which is presented in Section 4.3.3. In the final section, Section 4.3.4, we describe our modifications to the Traffic Transformer (Cai et al., 2020) to which the estimated graph structure is fed. Namely we introduce the attention-based directed graph convolutional neural network, which uses an anisotropic kernel to learn neighbouring features with two distinct attention mechanisms for the two orientations of the neighbours. In our experiments we compare a modification of the traffic transformer using directed attention, which is the subject of the final subsection.

4.3.1 Traffic Prediction

Our approach builds on the model for traffic prediction presented by Cai et al. (2020). The traffic transformer presented in that paper receives a source sequence of graph observations at the encoder, and the decoder predicts the subsequent hour of traffic readings that are expected from the activations computed by the encoder. Each graph is embedded in a k-dimensional space before it is passed to the transformer.

The first innovation is in its positional encoding. Since the transformer needs a sense of order in time, the solution is to modify the data with a positional encoding of some kind. Usually this is achieved by adding a plain d-dimensional positional encoding to the data at each timestep i for dimension a, a point on the following function:

$$\pi(i)_a = \begin{cases} a \pmod 2 \equiv 0 & \sin\left(\frac{i}{10{,}000^{\frac{2a}{k}}}\right) \\ a \pmod 2 \equiv 1 & \cos\left(\frac{i}{10{,}000^{\frac{2a}{k}}}\right) \end{cases} \tag{4.1}$$

thus $\pi(i) \in \mathbb{R}^d$. However, rather than adding positional embeddings to the data of each timestep as in a conventional transformer, termed addition-based combination, the model computes the dot-product of the positional encodings, giving the positional scalar $p(i,j)$ for each encoding at timesteps i and j

$$p_{i,j} = \pi(i)\pi(j), \tag{4.2}$$

which scale the coefficients in the attention mechanism, called similarity-based combination. If the source sequence has l steps and the target sequence has m steps, then the positional encoding is a matrix $\pi \in \mathbb{R}^{l+m,d}$ and the scalars form a matrix $p \in \mathbb{R}^{l+m,l+m}$.

The second innovation is the use of periodic data. In addition to the source sequence, Cai et al. propose feeding the encoder with an additional two sets of sequences: the traffic readings exactly a week (or several weeks) before the target sequence and those exactly a day (or several days) before the target sequence. The sets of sequences are respectively labelled the weekly-periodic and daily-periodic data; the preceding hour of data is thus labelled the hourly-periodic data. The weekly- and daily-periodic sequences are assigned the positional scalars of the target sequence, since they are viewed as coincident in periodicity.

We denote the l-timestep-long source sequence as $\boldsymbol{X} \in \mathbb{R}^{l \times n \times c}$ and the m-timestep-long target and predicted sequences respectively as $\boldsymbol{Y}, \hat{\boldsymbol{Y}} \in \mathbb{R}^{m \times n \times c}$ for n vertices with c channels. We denote sets of the weekly-periodic sequences $X(w)$ and the daily-periodic sequences as $X(d)$, where w and d are constants stating the number of preceding weeks and days in the sets respectively, thus denoting an offset from the source. As in Cai et al., 2020, we set $w = d = h = 1$, meaning we supply the model with sequences previous to the target sequence by 1 week, 1 day and 1 hour, each an hour long, therefore $l = m = 12$ timesteps long,

The model's parameters are adjusted according to the mean absolute error of the traffic prediction with respect to a target sequence:

$$l(\boldsymbol{Y}, \hat{\boldsymbol{Y}}) = \frac{1}{b} \sum_{i=0}^{b-1} |\hat{\boldsymbol{Y}}_i - \boldsymbol{Y}_i|, \tag{4.3}$$

where b is the batch size and $\hat{\boldsymbol{Y}}_i, \boldsymbol{Y}_i$ are the prediction and target respectively at the ith timestep. The model implements scheduled sampling to slowly move the model from learning on the target sequence to learning from its own predictions. The speed of the prediction is determined by the scheduling function, a value that amortises with the training step i

$$\epsilon_i = \frac{i}{i + \exp(i/s)}, \tag{4.4}$$

where the speed s controls the speed with which the model moves to training on its own predictions. Rather than choose a speed directly, it is selected indirectly by deciding the epoch by which, of the sequences supplied to the decoder, the ratio of target sequences to predicted sequences, what we call the scheduling ratio, is 50 per cent. This point can be computed with the Lambert W-function $W_0(-)$, such that

$$s = s_{0.5} = \frac{r}{W_0(b_s \cdot r)}, \tag{4.5}$$

where b_s is the number of batches per epoch and r is the epoch at which the scheduling ratio is 50 per cent.

4.3.2 The Distance-based Traffic Graph

The static traffic graph is often computed with a Gaussian kernel (Shuman et al., 2013) using the distances between each pair of vertices $\mathrm{dist}(a, b)$:

$$\boldsymbol{W}_{xy} = \begin{cases} \exp\left(-\text{dist}(a,b)^2/\sigma^2\right) & \text{if dist}(a,b) \leq \kappa, \\ 0 & \text{otherwise}, \end{cases} \tag{4.6}$$

where σ is the standard deviation of the training samples, κ is a threshold and $\boldsymbol{W}_{xy}$ is the entry at the xth row and yth column. We also refer to these graphs in this chapter as the *original graphs*, against which we will compare our estimated graphs. Such a construction has been used subsequently in the Traffic Transformer (Cai et al., 2020) and others (Ta et al., 2022; Yu, Yin, and Zhu, 2018). (In contrast Zhao et al. (Zhao et al., 2019) use an unweighted adjacency matrix.) The graph hence describes the topological proximity of sensors, but it does not necessarily describe long-distance interactions in traffic network beyond the relations described by the graph, nor can it adapt to changes in the traffic topology over time. This is the main motivation for the graph estimation strategies presented in the literature and in this work.

4.3.3 Graph Estimation from Cyclical Data

The graph estimation procedure that we propose learns a fusion of two graphs, one representing the long-term dynamics and the other the short-term dynamics of the traffic network. In splitting the graph estimation into separate streams, our approach bears a resemblance to the techniques discussed in the literature review (Kong et al., 2020; Ta et al., 2022; Zhang et al., 2020; Chen et al., 2020a). The graph structure is interpreted from the data using two attention-like mechanisms, one for the static structure and the other for the dynamic structure. We distinguish the attention coefficients computed in the graph estimation layer from those computed in the convolutional layer by the terms *global* and *local* attention coefficients. There is a nice symmetry in the attention approaches (for a description of the full set of models, see Section 4.4.1). On the one hand the global attention coefficients are used to learn global relationships, while the local attention coefficients learn local interactions, the existence and strength of which is in turn determined by the global attention coefficients.

Previous work has demonstrated the effectiveness of two fused streams of information (Ta et al., 2022). Each attention mechanism consists of two d-length vectors for each vertex in the set V, such that $dn \ll n^2$, where d is the dimensionality of the graph embedding

$W_{\text{emb}} \in \mathbb{R}^{c \times d}$. This is referred to as a self-adaptive matrix elsewhere (Yu, 2022). We also use multiple attention heads to diversify the structures identified in the data. Our use of a multi-head attention mechanism avoids directly modelling the n^2 parameters of the adjacency matrix; this is therefore not a direct method as described in Section 2.5.7.2. There are three advantages to this approach. Firstly the graph structure is determined by the data and not fixed after training as in direct methods. Secondly the number of parameters is lower than a direct method. Thirdly a graph can thereby be constructed for each prediction problem separately, as we do not average the estimated graphs over the batches.

The long-term structure of the traffic network is inferred from the weekly-, daily- and hourly-periodic data, $X(w), X(d), X(h) \in \mathbb{R}^{t \times n \times c}$, with t timesteps, n vertices and c input channels, composing the patterns of traffic over a long period of time. We identify the long-term structure observed with the static structure. Each sequence is first projected into a d-dimensional space using the embedding matrix $\boldsymbol{Z} \in \mathbb{R}^{c \times d}$ learned end-to-end in the model:

$$\left.\begin{aligned} X'(w) &= X(w)\boldsymbol{Z} \\ X'(d) &= X(d)\boldsymbol{Z} \\ X'(h) &= X(h)\boldsymbol{Z} \end{aligned}\right\} \in \mathbb{R}^{t \times n \times d}; \tag{4.7}$$

then the sequences are concatenated on the timestep dimension:

$$X'_{\text{stat}} = \big(X'(w) \,\|_0\, X'(d) \,\|_0\, X'(h)\big) \in \mathbb{R}^{3t \times n \times d}, \tag{4.8}$$

where $\|_0$ is a concatenation operation along the first dimension representing timesteps. An attention vector $\boldsymbol{a}_{\text{stat}} \in \mathbb{R}^{2d}$ is then applied to the concatenated features such that:

$$\overline{W}_{\text{stat,xy}} = \text{ReLU}\left(\frac{1}{3t}\sum_{i=0}^{3t-1}\left[X'_{\text{stat},i,x} \,\|\, X'_{\text{stat},i,y}\right]\boldsymbol{a}_{\text{stat}}\right), \tag{4.9}$$

where $X'_{\text{stat,i,x}} \in \mathbb{R}^d$ is the set of features for vertex x at timestep i. The ReLU is applied to constrain the values of the dynamic matrix in the interval $[0, \infty)$. The learned static weight matrix is denoted $\overline{\boldsymbol{W}}_{\text{stat}}$.

The short-term structure is inferred from only the hourly-periodic data $X(h)$, combined additionally with the target sequence $\boldsymbol{Y}$ during training. (Since the target sequence is what the model is trying to predict during training, it is used to estimate the graph structure at inference.) The source and target sequence are especially relevant to the task; they alone contain any structural information on short-term disturbances, such as road closures, which are important to understand in predicting traffic. The model is allowed to interpret the less stable changes in structure from the short-term information. Therefore we identify the short-term structure with the dynamic structure. The short-term graph is computed similarly. The two sequences $X(h), \boldsymbol{Y}$ are first projected into a d-dimensional space with the embedding matrix $\boldsymbol{U} \in \mathbb{R}^{c \times d}$, learned end-to-end in the model as in Eq. 4.7, then the average over the timesteps and of the three sequences is computed,

$$X'_{\text{dyn}} = [(X\boldsymbol{U})_i \,\|\, \zeta \cdot (\boldsymbol{Y}\boldsymbol{U})_i] \in \mathbb{R}^{2t \times n \times d}\,, \tag{4.10}$$

where ζ is a flag indicating whether the algorithm is training, meaning the target sequence—and therefore the sequence length, too—is excluded. As above, an attention vector $\boldsymbol{a}_{\text{dyn}} \in \mathbb{R}^{2d}$ is then applied to the concatenated features that are subsequently passed through a rectified linear unit:

$$\overline{W}_{\text{dyn,xy}} = \text{ReLU}\left(\frac{1}{(1+\zeta)\cdot t}\sum_{i=0}^{(1+\zeta)t-1}\left[X'_{\text{dyn},i,x} \,\|\, X'_{\text{dyn},i,y}\right]\boldsymbol{a}_{\text{dyn}}\right)\,, \tag{4.11}$$

where $\boldsymbol{X}'_{\text{dyn,i,x}} \in \mathbb{R}^d$ is the feature of vertex x at timestep i. As above, the ReLU is applied to constrain the values of the dynamic matrix in the interval $[0, \infty)$. The learned dynamic weight matrix is denoted $\overline{\boldsymbol{W}}_{\text{dyn}}$. Note that we do not use softmax in computing the attention coefficients. Otherwise it would require us to choose an axis along which to perform the operation, meaning we would have to decide between doing it on the in-edges or out-edges. We want to compute a graph that will account for both flows; consequently there is no way to perform softmax on both axes simultaneously. We decide not to use softmax, and instead rely on regularisation and thresholding to control the values.

The static and dynamic graphs are then fused with a softmax-activated learned mixing coefficient $\beta \in (0,1)$:

$$\overline{\boldsymbol{W}} = \beta \cdot \overline{\boldsymbol{W}}_{\text{stat}} + (1-\beta)\cdot \overline{\boldsymbol{W}}_{\text{dyn}}\,. \tag{4.12}$$

In fusing the static structural information with the dynamic structure, we are thereby combining a stable structure with the unstable but presently relevant structure.

Additionally we implement multi-head attention in graph estimation. The multiple heads enable the model to estimate several structures from the input data simultaneously. After the structure has been estimated on multiple heads, the average of the several graphs is taken. The output of the graph estimation is therefore a single graph for each sample. Note that we do not apply the same graph to every sample; instead we allow a new graph to be estimated for every sample. This follows our assumption that, although the static structures are relatively stable in relation to the dynamic structures, the static structures still may change over time. Furthermore, it makes little sense to average the graphs over the samples since during learning the batches are drawn randomly from the training set, which means the structures would potentially differ massively if a single graph was learned for the whole batch. The drawback of course is the increased memory complexity of learning.

In order to validate our fusion approach, we define another graph estimation approach that does not estimate graphs from two different streams, which we call *simple graph estimation*. In effect it means concatenating the weekly-, daily- and hourly-periodic data, and the target $\boldsymbol{Y}$ during training,

$$X'_{\text{simple}} = \big(X'(w) \,\|_0\, X'(d) \,\|_0\, X'(h) \,\|_0\, \zeta \cdot \boldsymbol{YZ}\big) \in \mathbb{R}^{(3+\zeta)t \times n \times d}\,, \tag{4.13}$$

and, with an attention vector $\boldsymbol{a}_{\text{simple}} \in \mathbb{R}^{2d}$ altering Eq. 4.12 to

$$\overline{\boldsymbol{W}} = \text{ReLU}\left(\frac{1}{(3+\zeta)\cdot t} \sum_{i=0}^{(3+\zeta)(t-1)} \left[X'_{\text{simple},i,x} \,\|\, X'_{\text{simple},i,y}\right] \boldsymbol{a}_{\text{simple}}\right)\,. \tag{4.14}$$

Estimating the weight matrix $\overline{\boldsymbol{W}}$ presents a unique problem, however. There will be no zero entries in the matrix unless one of the vertex embeddings is zero. Therefore the weight matrix will be dense. It is necessary to sparsify the weight matrix somehow. There are many ways to achieve this end (Zhu et al., 2022). We impose a hard threshold $\gamma = 0.1$ on the values of the weight matrix, such that all weights in $\overline{\boldsymbol{W}}$ below γ are set to zero. Simultaneously we also add L_1 regularisation of the sum of the weight matrix's entries. We also constrained the values of the projection matrix and attention mechanism to the interval $[0, 1]$ by normalising the weights by the minimum and maximum values in each attention head. The entries of the graph are directly optimised with respect to the learning goal by back-propagation. Lastly, we apply dropout at 80% to the output of the graph estimator during training to sparsify the estimated graph, which yields three advantages: Firstly the sparser graph enables the speed-up gained from sparse operations. Secondly the graph estimator is effectively focussing on a sampled set of vertices from the estimated graph. We assume that the most informative subset of relations in the graph constitutes a small spanning graph over the hypothesis space. Thirdly dropout has been shown to prevent over-smoothing (Rong et al., 2020a).

4.3.4 Modifications to the Traffic Transformer

In this section we present our modification to the Traffic Transformer proposed by Cai et al. (2020). At the core of the Traffic Transformer is the transformer proposed by Veličković et al. (2018). The transformer, as we have already seen in Section 2.4.2.1, consists of an encoder and decoder. The input is structured as a sequence of vectors which is passed simultaneously to the encoder as input. The decoder also receives vectors as input during training and testing. The output of the decoder, the prediction, is also a vector. Somehow the sequence of graph data supplied to the transformer needs to be embedded as vectors. The Traffic Transformer uses single layer of directed diffusion proposed by (Li et al., 2018a, see Section 2.5.4.2) to embed graph-structured traffic data into vectors. The directed diffusion able to process the input information as directed flows over several diffusion steps. But, as we noted in Section 2.5.4.2, directed diffusion uses an anisotropic kernel over the in- and out-neighbourhoods, meaning it may less flexibly determine the contributions of signals *within* neighbourhoods. We propose a modification of the Traffic Transformer whereby the directed diffusion layers are replaced with what we term *directed attention convolution*. We describe directed attention convolution in the following section.

4.3.4.1 Directed Attention Convolution

As we remarked in Section 2.4.2.1, transformers are non-recursive, which lends it an advantage in training time, as it allows the model to receive whole sequences simultaneously. This is accomplished by the implementation of an attention mechanism, which simplifies the recurrence in RNNs by replacing the stepwise multiplications with dot-products across the timesteps. Here we describe principally the attention mechanism, as it is the relevant part of our discussion. Further details on the model, such as the masking of inaccessible timesteps and further details of the architecture can be read in the original paper (Vaswani et al., 2017), to which Cai et al. (2020) make few alterations.

The transformer output in the Traffic tTansformer (Cai et al., 2020) is modified from what is present in the original transformer (Vaswani et al., 2017). In the original transformer the decoder output for each timestep is probabilistic. In the Traffic Transformer it is scalar. Moreover, the output of the decoder is expanded to the size of the graph, since the number of output channels is lower than the number of vertices at each timestep; this allows the model to make predictions for each vertex.

We propose a modification of the usual graph attention (Veličković et al., 2018, see Section 2.5.4.2) that factors input data into in- and out-streams of information, we call the *directed attention convolution*. The form of the directed attention convolution is inspired by directed diffusion. Whereas the original formulation makes no distinction between in- and out-neighbours, we adopt two parallel attention mechanisms that learn coefficients for the two distinct flows of information into a vertex.

The pre-normalised coefficient $\boldsymbol{C}_{xy}$ is unique to the relationship between a focal vertex x and neighbouring vertex y. This is a consequence of the concatenation operation of the two projected vectors $\boldsymbol{h'}_{l,x}$ and $\boldsymbol{h'}_{l,y}$, which is intransitive. Essentially the first d dimensions of $\boldsymbol{a}$ learn on the (projected) features of focal vertices, while the second d dimensions learn on the (projected) features of their neighbours. This relationship does not represent an orientation of the edge between the two, however. The attentional mechanism hence lacks a means to represent directed flows of information on the graph—unless we say that the coefficient $\boldsymbol{C}_{xy}$ represents either the in- or out-edges of the focal vertex x—but not both.

To represent the orientation of edges, we make a small adjustment to the attentional mechanism: instead of $\boldsymbol{a} \in \mathbb{R}^{2d}$, we expand the vector to $\boldsymbol{a} \in \mathbb{R}^{3d}$, where the first d elements $\boldsymbol{a}_0$ learn on in-neighbours' features, the second d elements of $\boldsymbol{a}_1$ learn on the focal vertices' features, and the final d elements $\boldsymbol{a}_2$ learn on out-neighbours' features. The pre-normalisation attention coefficient matrix in Eq. 2.86 instead depends on several computations. The matrix is formulated as

$$\boldsymbol{C}_{l,xy} = \begin{cases} \text{LeakyReLU}\left(c_{l,x}(x)\right) & x = y \\ \text{LeakyReLU}\left(c_{l,x}(\vec{yx}) + c_{l,x}(\vec{xy})\right) & x \neq y \end{cases} \tag{4.15}$$

where

$$c_{l,x}(x) = \boldsymbol{h}'_{l-1,x}\boldsymbol{a}_1 \,, \tag{4.16}$$

$$c_{l,x}(\vec{yx}) = \begin{cases} \left[\boldsymbol{h}'_{l-1,y} \| \boldsymbol{h}'_{l-1,x}\right] \boldsymbol{a}_{0:1} & \vec{yx} \in E \,, \\ 0 & \text{otherwise} \,, \end{cases} \tag{4.17}$$

$$c_{l,x}(\vec{xy}) = \begin{cases} \left[\boldsymbol{h}'_{l-1,x} \| \boldsymbol{h}'_{l-1,y}\right] \boldsymbol{a}_{1:2} & \vec{xy} \in E \,, \\ 0 & \text{otherwise} \,, \end{cases} \tag{4.18}$$

where $\boldsymbol{a}_{0:1}, \boldsymbol{a}_{1:2} \in \mathbb{R}^{2d}$ are slices of attention vector $\boldsymbol{a}$. This minimal extension preserves the simplicity of the attentional mechanism while allowing in- and out-flows to be modelled separately, as they are in the directed diffusion graph. The subsequent Eqs. 2.86 and 2.88 are unchanged. The directed graph attention uses a multi-head attention mechanism for the reasons described in Section 2.5.4.2. The form of the multiple attention heads in the convolution is identical to what is presented in Eq. 2.89.

4.3.5 Attention-based Graph Estimation and Graph Convolution

The coefficients obtained in graph estimation are joined together with the local attention coefficients in the convolution. The estimated structure computes a weight matrix for every sample. The weights correspond to adjacencies in the graph, but they also constitute a global weighting of the vertices with respect to one another over the whole dataset, which we term *global attention coefficients*. This is informative for the graph convolution, where a local weighting is also computed in the attention mechanisms, which we term *local attention coefficients*. We combine the global and local coefficients in all models, including the edge weights present in the original graph, by modifying that

$$\boldsymbol{h}_{l,x} = \sigma\left(\sum_{i=0}^{k}\sum_{y\in\Gamma(x)} \overline{W}_{x,y}\alpha_{l,i,xy}\boldsymbol{W}_l\boldsymbol{h}_{l-1,y}\right). \tag{4.19}$$

where $\overline{W}_{xy}$ is the global attention coefficient for the vertices $x, y \in G$.

4.4 Experiment

In this section we describe the experiments we carried out to evaluate the comparative performance of the models, which we describe in the Section 4.4.1. In Section 4.4.2 we describe the control variables for the experiment. In Section 4.4.3 we describe the datasets, METR-LA and PeMS-Bay, the structure of the input data, the structure of the conventional graph used for the two datasets, and the metrics for performance, as well as the loss function. Finally in Section 4.4.4 we enumerate the hypotheses in advance of our analysis in the next section of this chapter.

4.4.1 The Models

We compare the performance of two approaches to graph estimation mixed with four approaches to convolution. We also consider the four convolution approaches without graph estimation. In total, therefore, we consider twelve models. There are two attention-based convolution models: undirected attention, the GAT introduced by Veličković et al. (2018), and the directed GAT, our variation on the GAT described in Section 4.3.4.1.

We also consider two diffusion models, using one-step diffusion and two-step diffusion. The diffusion convolution was introduced by Li et al. (2018a) and the two-step diffusion subsequently used in the Traffic Transformer (Cai et al., 2020) as in the original paper.

Li et al. (2018a) demonstrated experimentally that the diffusion lends an interpretative power to a traffic model.

Each model is fed either with the modified original graph described in Section 4.3.2 or with a graph estimated in one of two ways. We have described the principal way we propose to estimate the traffic graph in Section 4.3.3. Solely for the sake of comparison and verification, we use in the experiment a second, simplified version of the static–dynamic fusion that does not separate the data into static and dynamic streams of information. Instead there is only one stream of data, namely the static stream in Eq. 4.9, where at training, when the scheduled sampling allows it, the target sequence is included in the summation as in Eq. 4.11. By eliminating the distinction between long- and short-term structures in the simplified approach to graph estimation, we can verify whether our static–dynamic fusion approach is learning structural information that helps prediction.

In this chapter, when we refer to the "fully attention-based" models, we mean those models that use either simple graph estimation or graph estimation by static–dynamic fusion *and* the undirected or directed GAT.

4.4.2 The Control Variables

The models were trained on 4 NVIDIA V100 GPUs and 16 Intel Xeon Gold 6148 CPU cores hosted by the Sunbird supercomputer at Swansea University. Each model was trained for 100 epochs with a batch-size of 4 for the METR-LA dataset and 2 for the PeMS-Bay dataset. The batch size must be kept low because the graph estimator demands a large memory capacity owing to the large number of parameters. We used the Adam optimiser with a learning rate $\lambda = 1 \times 10^{-4}$. For the scheduled sampling, we set $r = 5$ in Eq. 4.5 so that predicted sequences are supplied to the decoder half the time by the 5th epoch. As the loss function we used the mean absolute error (MAE),

$$\mathrm{MAE}(y, \hat{y}) = \frac{1}{nt} \sum_{j=0}^{t-1} \sum_{i=0}^{n-1} |y(i,j) - \hat{y}(i,j)|, \tag{4.20}$$

and add a regularisation term of the estimated graph $\overline{\boldsymbol{W}}$:

$$||\overline{\boldsymbol{W}}|| = \frac{1}{b} \sum_{i=0}^{b-1} \sum_{j=0}^{n-1} \sum_{k=1}^{n} |\overline{W}_{i,j,k}|, \tag{4.21}$$

where b is the batch size, yielding the loss function:

$$l(y, \hat{y}) = \mathrm{MAE}(y, \hat{y}) + ||\overline{\boldsymbol{W}}||. \tag{4.22}$$

4.4.3 The Datasets and Metrics

The models described in Section 4.4.1 are evaluated on their predictive performance on METR-LA and PeMS-Bay datasets, which are commonly used in evaluating traffic prediction models. In this subsection we describe these datasets briefly and two ways to construct their graphs. The two datasets are collections of speed readings from vehicles driving over loop detectors across two traffic networks. The readings of the loop detectors have been summarised into sequences of five-minute averages of observed speeds of vehicles that passed over the loop detectors.

The METR-LA dataset is a network of 207 loop detectors in Los Angeles County. It is a subset of the larger set of 8,900 sensors introduced by Jagadish et al. (2014). The dataset contains 23,974, 3,425 and 6,850 training, validation and testing sequences respectively. The PeMS-Bay dataset consists of speed readings 325 loop detectors from the San Francisco Bay Area (Li et al., 2018a). The dataset is split into 36,465, 5,209 and 10,419 training, validation and testing sequences respectively. The input data to the encoder and decoder is z-normalised, where the mean and standard deviation are computed from the training set. The target sequences are not normalised; therefore the model is trained to map normalised speeds to unnormalised speeds.

To construct any graph from real-world data we need to make two decisions: (1) Which set of elements of the dataset constitute the vertices? (2) What are the criteria by which we decide whether two elements of that set are related, i.e., how do we decide which vertices to connect to one another? For traffic data, the vertices usually represent the sensors, in this case the loop detectors in the METR-LA and PeMS-Bay datasets. The crux of the construction is how vertices are connected, usually by reference to some material relationship between the vertices; i.e. it is not random.

In Section 4.3.2, we described the topological graphs computed using a thresholded Gaussian kernel over the road distance between the sensors to compute the edge weights in the sensor graph. There are two issues with the graph constructed by Li et al. that mean we must alter the graph. Firstly the graphs contain diagonal entries that are always 1, effectively giving them all self-loops. We remove these entries, because there is already a self-loop implicit in the definition of the diffusion and attention convolutions. In the diffusion convolution, it is there by default as the zeroth diffusion step, the identity matrix, and in the attention mechanisms a self-loop on each vertex is implicit in their definitions. There are also some disconnected vertices in each graph. In the METR-LA graph, there is 1 disconnected vertex, and 6 disconnected vertices in the PeMS-Bay graph. We delete the disconnected vertices and use the resultant graph in the experiments. The reduced graphs therefore have 206 instead of 207 vertices for the METR-LA dataset and 319 instead of 325 vertices for the PeMS-Bay dataset.

The models are assessed on the basis of three metrics, lower values of all representing a better prediction. The first metric is the MAE (Eq. 4.20), which measures the average absolute deviation of the predicted value from the target value. The second metric, the root mean squared error (RMSE),

$$\mathrm{RMSE}(y, \hat{y}) = \frac{1}{nt} \sum_{j=1}^{t} \sum_{i=0}^{n-1} \left(y(i,j) - \hat{y}(i,j)\right)^2, \tag{4.23}$$

measures the quality of the predictions, namely how far from the target value the predicted values lie. A large RMSE relative to the MAE will indicate that the model is making outlandish predictions. By extension, if two models are measured to have a similar MAE, by comparing the RMSE we can measure the relative stability of the prediction errors. The third metric, the mean average percentage error (MAPE),

$$\mathrm{MAPE}(y, \hat{y}) = \frac{1}{nt} \sum_{j=1}^{t} \sum_{i=0}^{n-1} \frac{y(i,j) - \hat{y}(i,j)}{y(i,j)}, \tag{4.24}$$

measures the relative error of the predicted values with respect to the target value. For example, in the traffic dataset, an error of 5 m.p.h. is a more significant error when the correct speed is 10 m.p.h. than when it is 20 m.p.h.

We follow the example of Cai et al. (2020) by excluding datapoints in the measurement where the true value $y(i, j)$ is zero for two reasons: (1) no prediction is necessary at these points; and (2) a zero speed would lead to a zero-division error in Eq. 4.24.

4.4.4 Hypotheses

In the analysis of our results we test the following hypotheses:

Hypothesis 1 The directed diffusion layer attains lower prediction errors than the GAT and the directed GAT when the problem is structured using the modified original graph.

Hypothesis 2 2-step directed diffusion will attain a lower prediction error than the 1-step directed diffusion when the problem is structured using the modified original graph.

We believe that the directed diffusion will produce more accurate predictions compared to the attention methods. We also expect that 2-step diffusion will produce more accurate predictions than 1-step diffusion because it can model more elaborate interactions.

Hypothesis 3 The GAT and directed GAT will make better predictions when the graph is estimated compared to the original graph.

We believe that graph estimation will produce a more informative graph if it is learned on the learning objective than if it is fixed.

Hypothesis 4 The fusion of the static and dynamic graphs will yield a estimated graph that reduces the prediction error compared to a graph that is estimated on a single, undifferentiated stream of information.

We believe that the static and dynamic graphs will encode different kinds of information as we described above because the dynamic changes observed in the short-term data will inform the structure and thus the traffic prediction.

Hypothesis 5 The GAT and directed GAT will attain lower prediction errors than the directed diffusion model when the graph for each is estimated.

Hypothesis 6 The directed GAT will attain a lower prediction error than the GAT when the graph is estimated.

Since a fully attention-based model uses both global and local attention, the capacity of the model is greater than the graph-estimating traffic model using directed diffusion. The directed diffusion model cannot flexibly alter the contributions of vertices within diffusion steps like the attention approaches can. We also expect that the directed GAT will attain a lower prediction error than the GAT since the directed GAT assigns in- and out-edges separate coefficients, whereas the GAT does not.

Hypothesis 7 The prediction error over time will increase at a lower rate when the graph is estimated as opposed to when it is fixed.

Predictions made that lie further away in time from the period covered by the dataset are less accurately predicted. This problem worsens with distance in time. Because we allow the graph estimation to produce a graph according to the input data, the graph estimation is able to produce a graph that is more relevant to structure of the data it presently sees.

Table 4.1: The prediction errors of each model on the PeMS-Bay dataset at 15, 30 and 60 minutes. The best results appear in bold; the second-best results are underlined. This table of results has been adapted from Kenning and Xie (2023).

Models on PeMS-Bay		MAE			MAPE			RMSE		
Estimation		15′	30′	60′	15′	30′	60′	15′	30′	60′
None	Attention	2.787	2.841	2.885	6.575	6.686	6.773	4.810	4.915	5.018
	Directed attention	**2.652**	**2.681**	**2.704**	6.450	**6.505**	**6.533**	**4.656**	**4.719**	**4.782**
	1-step diffusion	2.730	2.791	2.837	**6.424**	6.573	6.684	4.711	4.827	4.931
	2-step diffusion	2.741	2.782	2.812	6.452	6.537	6.594	4.757	4.844	4.924
Simple	Attention	**2.615**	**2.642**	**2.659**	6.386	6.424	**6.438**	4.631	4.682	**4.732**
	Directed attention	2.627	2.657	2.684	**6.331**	**6.383**	6.439	**4.598**	**4.668**	4.743
	1-step diffusion	2.754	2.802	2.836	6.522	6.611	6.673	4.770	4.870	4.961
	2-step diffusion	2.718	2.756	2.784	6.410	6.485	6.536	4.717	4.800	4.879
Static–dynamic	Attention	**2.634**	**2.660**	**2.680**	**6.368**	**6.406**	**6.430**	**4.640**	**4.693**	**4.751**
	Directed attention	2.659	2.680	2.710	6.476	6.479	6.510	4.647	4.699	4.777
	1-step diffusion	2.759	2.798	2.835	6.518	6.594	6.663	4.769	4.853	4.943
	2-step diffusion	2.743	2.782	2.812	6.565	6.656	6.723	4.759	4.840	4.920

Table 4.2: The prediction errors of each model on the METR-LA dataset at 15, 30 and 60 minutes. The best results appear in bold; the second-best results are underlined. This table of results has been adapted from Kenning and Xie (2023).

Models on METR-LA		MAE			MAPE			RMSE		
Estimation		15′	30′	60′	15′	30′	60′	15′	30′	60′
None	Attention	4.894	5.089	5.463	13.457	13.877	14.603	7.515	7.798	8.285
	Directed attention	4.880	4.717	**4.695**	13.926	13.541	**13.476**	7.755	7.678	**7.748**
	1-step diffusion	**4.657**	**4.686**	4.927	13.521	**13.486**	13.852	**7.415**	**7.495**	7.812
	2-step diffusion	5.008	5.160	5.415	**13.318**	13.541	13.937	7.653	7.893	8.253
Simple	Attention	4.670	4.719	5.007	12.748	12.756	13.214	7.326	7.460	7.842
	Directed attention	5.521	5.403	5.520	14.766	14.351	14.488	8.356	8.366	8.604
	1-step diffusion	**4.230**	**4.353**	**4.732**	**11.656**	**11.880**	**12.577**	**6.766**	**6.989**	**7.473**
	2-step diffusion	4.529	4.632	4.924	13.629	13.781	14.229	7.363	7.440	7.760
Static–dynamic	Attention	4.308	**4.314**	4.438	12.452	12.328	12.492	7.119	7.137	7.326
	Directed attention	4.980	4.869	4.914	14.350	13.915	13.907	7.846	7.836	8.016
	1-step diffusion	**4.289**	4.456	4.771	**11.892**	**12.173**	12.784	**6.852**	7.097	7.531
	2-step diffusion	4.374	4.336	**4.413**	12.503	12.251	**12.251**	7.133	**7.083**	**7.208**

4.5 Results

For each dataset we present a table of results across three metrics, MAE, MAPE and RMSE, measured on the predictions of each model. We assess the prediction errors against the hypotheses we enumerated in Section 4.4.4.

4.5.1 PeMS-Bay

In the full table of results on the PeMS-Bay dataset presented in Table 4.1, we observe that, contrary to our expectations in **Hypothesis 1**, the directed attention layer attains the lowest average errors when there is no graph estimation. Directed diffusion is however fairly consistently the runner-up, more often two-step than one-step directed diffusion, outperforming undirected convolution, meaning **Hypothesis 1** can be partly accepted.

To test **Hypothesis 3**, we compute the error difference between no graph estimation and the two kinds of graph estimation at sixty minutes on the three metrics within each model. We visualise the differences as graphs in Fig. 4.2. It is clear to see that simple graph estimation is a help, but the static–dynamic fusion of graph structures is apparently a hindrance. It is a greater help to the GAT than the directed GAT.

To test **Hypothesis 4**, we again study the bar-graphs in Fig. 4.2. The static–dynamic fusion clearly worsens the prediction error in comparison to no graph estimation. The simple graph estimation on the contrary lowers the prediction error. The one-step directed diffusion is the exception in both cases. The degradation and improvement of performance is however negligible. The largest improvement in the bar-graphs is 0.335 in the MAPE for the attention model.

The two GAT variants attain lower prediction errors than the directed diffusion variants when the graph is estimated, confirming **Hypothesis 5**. The confirmation is not total, however. The prediction error of the undirected GAT is consistently the worst-performing model when the graph is not estimated but the directed GAT is the best. Interestingly, the undirected GAT largely outperforms the directed GAT on average when the graph is estimated; consequently we cannot accept **Hypothesis 6**, especially as the differences are so small.

4.5.2 METR-LA

In the full table of results on the METR-LA dataset is presented in Table 4.2 we observe in contradiction to our expectations in **Hypothesis 1** that the two-step directed diffusion layer does not attain the lowest prediction errors. On only one metric does it obtain the lowest error, namely at the 15-minute prediction horizon according to MAPE. Otherwise it attains the highest or second-highest error. Indeed, a complete reversal appears to us, as often the 1-step diffusion has the lower errors, meaning we have to reject **Hypothesis 2**. The messiness of the results corresponds to the reputed difficulty of the dataset.

To test **Hypothesis 3** by computing the error difference between no graph estimation and the two kinds of graph estimation at sixty minutes on the three metrics within each model. We visualise the differences as graphs in Fig. 4.2. On all models but the 1-step directed diffusion, the fusion of the static and dynamic graph estimations reduces the prediction error. The simple graph estimation reduces the errors in all cases but the directed attention layer. In all but two cases, then, graph estimation reduces prediction error, confirming **Hypothesis 3**.

We turn again to Fig. 4.2 to draw our conclusions about **Hypothesis 4**. The static–dynamic graph estimation leads to superior performance on average over simple graph estimation only on directed diffusion. The performance of the 2-step directed diffusion with static–dynamic graph estimation is roughly on par with the same model with simple graph estimation. The performance of GAT with static–dynamic graph estimation is on par or worse. The performance of the 1-step directed diffusion is a little worse than no graph estimation; simple graph estimation is barely better. What we do see however is that in three cases (1- and 2-step directed diffusion and attention) the MAPE is substantially lower when a graph structure is learned. MAPE measures the relative dispersion of the prediction from the target. Graph estimation appears to reduce the dispersion around the target.

Regarding **Hypothesis 5**, we see that it is partially confirmed in observing Fig. 4.2. The undirected GAT undergoes a reduction in prediction error with both graph-estimation schemes. The directed GAT's prediction error only reduced with static-dynamic graph estimation, however. We cannot confirm **Hypothesis 6**, however, since the directed GAT performs worse than the undirected GAT across the board when some graph estimation technique is used.

4.5.3 Post-hoc Analyses

In the post-hoc analyses we study (1) the errors over time and then (2) the structure of the estimated graphs for each model and each graph estimation strategy.

4.5.3.1 Data Comparisons

First in the *post hoc* analyses, we will measure the prediction error as the observations over time for the test set. We present the errors on the test set over time as three-hour rolling averages of the test error on the PeMS-Bay (Fig. 4.3) and METR-LA (Fig. 4.4) datasets. The purpose of the rolling average is to flatten out cyclical errors in the daily and weekly cycles. In respect of **Hypothesis 7**, it is difficult to discern any trends in error on the METR-LA set in Fig. 4.4, but we can see a tendency for the error to increase over time in general, irrespective of model, on the PeMS-Bay dataset in Fig. 4.3. The same trend is visible when we fit a linear regression on the three-hour rolling average and consider

Table 4.3: For each model and strategy we fit a linear regression on the test prediction errors on each dataset. The matrices below gives the slopes of each regression for the PeMS-Bay and METR-LA datasets respectively. Positive values represent errors that increased with time; negative values represent errors that decreased with time.

PeMS-Bay				
Graph estimation strategy	Directed diffusion		Attention	Directed Attention
	1-step	2-step		
None	4.8×10^{-5}	1.3×10^{-5}	-1.4×10^{-5}	2.0×10^{-5}
Simple	1.3×10^{-5}	7.2×10^{-6}	-7.3×10^{-6}	1.5×10^{-5}
Static–dynamic	3.6×10^{-6}	2.4×10^{-5}	-1.0×10^{-5}	2.5×10^{-5}

METR-LA				
Graph estimation strategy	Directed diffusion		Attention	Directed Attention
	1-step	2-step		
No	-1.6×10^{-4}	-4.3×10^{-4}	-3.3×10^{-4}	2.3×10^{-4}
Simple	-2.7×10^{-4}	-1.4×10^{-4}	-2.0×10^{-4}	5.5×10^{-4}
Static-dynamic	-2.5×10^{-4}	3.1×10^{-5}	-5.4×10^{-6}	2.5×10^{-4}

the gradient. Table 4.3 shows an increase in the error over time on PeMS-Bay. We can also see that occasionally the gradients are lower for the learned graphs. But there are no discernable trends and so we have to reject **Hypothesis 7**. For the METR-LA dataset, we cannot conclude anything with confidence, namely that graph learning is consistently reducing that error. A proper analysis would require much longer stretches of data and much broader windows of time in the rolling windows.

A very noticeable difference on the METR-LA dataset in Fig. 4.4 is the period between the 1,000th and 3,000th windows, where the graph estimation using static–dynamic fusion suppresses the high proportion of errors present in the other models, with the exception of directed diffusion. Further analysis of the missing data in the test set, visualised in Fig. 4.5, reveals that in this period there is a high concentration of missing data. The same visualisation also explains the disturbance around the 3,750th window in Fig. 4.4 for the directed attention models. It seems that static–dynamic attention is largely able to resist the error to which simple graph estimation is very susceptible. The model structured by the original graph, and so with no estimated graph, avoids the error altogether.

The suggestion that errors will increase over time, as we see in Fig. 4.3, is a sensible suspicion. That we do see the error increasing suggests perhaps that the model would benefit from some sense of the global passage of time, some awareness of the distance of unobserved samples from the last training point.

4.5.3.2 Structural Comparisons

In this section we compare the topological graphs that accompany the METR-LA and PeMS-Bay datasets, which we term here the "original graphs", and the two-step diffusion graph with a set of estimated graphs from each model. We sampled 20 observations uniformly randomly from the test set, computed their respective estimated graphs and took their average structure. Each edge in the average structure is thus an average of the the same edge in the 20 estimated graphs. In Figs. 4.6 and 4.7 we have visualised the average estimated graphs. Henceforth we will refer to these graphs simply as "estimated graphs" for simplicity's sake.

We observe firstly that, in stark contrast to the originals graphs, the estimated graphs are very dense. This is evident in a comparison of the visualisations and the minima, maxima, means and medians of the edge weights presented in Tables 4.5 and 4.6. The graphs estimated on the PeMS-Bay dataset are completely dense, where the minimum in any graph was never zero; although, interestingly, the more difficult dataset METR-LA led to graph estimations with some zero edge weights. It is difficult to understand why, if the reason is not that the vertices are all helpful in the comparatively simple PeMS-Bay network, where it is paradoxically more difficult for the algorithm to wholly rule out certain vertex pairings. A consequence is that optimisations of later convolutions that leverage sparsity are ineffective; computation with the estimated graphs is hence protracted. It is clear that neither dropout—which at least speeds training up—nor L_1 regularisation of the graph's edge weights sufficiently low to be excluded with the threshold. In other words, the techniques we used failed to sparsify the graphs.

We can also see in Figs. 4.6 and 4.7 that the attention mechanism is modifying whole columns and rows, visible as the conspicuous lines that criss-cross the graph visualisations. It is clear that the attention coefficients of a given vertex strongly determine the edge weights of its adjacent edges. We can also see consistent patterns in the vertices that the graph estimation strategies are selecting as relevant. The greatest difference in these patterns occur in the 1-step diffusion models, especially in the simple strategy of graph estimation. Namely we see that the strength of the horizontal lines is greater than that of the vertical lines.

The coefficients of the models show that there are roughly equal mixtures of the static and dynamic graphs with slight preferences for one structure or another in some attention heads; see Tables 4.7 and 4.8. There is no overwhelming preference for one

Table 4.4: The time to train each model in hours:minutes:seconds. Training time includes the training and validation steps, recording the results for Tensorflow, the final save of the model's weights and saving the raw outputs of the model to disk.

Estimation	Models	PeMS-Bay	METR-LA
None	Attention	86:01:04	30:12:25
	Directed attention	85:12:54	33:05:29
	1-step diffusion	121:48:42	38:15:20
	2-step diffusion	115:23:35	39:18:05
Simple	Attention	78:02:12	41:00:41
	Directed attention	127:32:35	40:57:20
	1-step diffusion	195:08:29	84:18:21
	2-step diffusion	197:30:40	88:35:95
Static–dynamic	Attention	87:30:07	43:28:22
	Directed attention	137:03:32	44:18:14
	1-step diffusion	218:13:40	128:27:31
	2-step diffusion	210:15:05	133:34:33

stream of information or another (static/long-term vs. dynamic/short-term) in either the simple or fused graph estimation.

Curiously, specifically in the PeMS-Bay dataset, there are a few communities in the bottom-left corner of the 2-step diffusion that are assigned quite low scores in the graph estimation. The 2-step diffusion graph is interesting to consider because it is where the convolution is expanding the neighbourhood to include vertices at a distance. The idea behind that expansion of the neighbourhood was to capture long-distance effects in the network. It suggests that at the very least the structure determined by geographical distance does not furnish the algorithm informatively.

4.5.4 Runtimes

The runtimes for the models are presented in Table 4.4. Directed attention appears to require minimal overhead in comparison to undirected attention. Meanwhile one- and two-step diffusion consistently have a larger overhead in training than the attention-based methods. This is unsurprising since the computation of the diffusion matrix is a $\mathcal{O}((k-1) \cdot 2 \cdot n^3)$ for k diffusion steps. The computational burden is hence magnified when the graph is estimated; as illustrated in Figs. 4.6 and 4.7; in this case, since each input sample is associated with a graph, the diffusion matrix is computed as many times as there are samples in the batch at each training step. Although interestingly the difference between 1- and 2-step diffusion is not as drastic as expected: in some cases, 2-step diffusion is *faster* than 1-step diffusion to train.

The attention mechanism is comparatively simpler, having a much smaller overhead, since the global attention coefficients are simply directly incorporated into the convolution. Indeed, overall, the time to train the attention-based techniques when graph estimation is added is tolerable in comparison to the differences between graph estimation and none for directed diffusion. A second surprising result is that directed attention is on two occasions quicker to train than attention, despite the added overhead of a broader attention mechanism (compare no graph attention on PeMS-Bay and simple graph convolution on METR-LA).

Overall it seems that the graphs estimated for the PeMS-BAy datasets lead to a greater overhead overall. The undirected attention appears to be barely affected by the dataset, but a significant training burden is imposed on the directed attention and directed diffusion models when graph estimation is introduced on the PeMS-Bay dataset.

4.6 Discussion

Before we begin the discussion, we must note a few general facts. The METR-LA dataset is regarded as more difficult than the PeMS-Bay (Li et al., 2018a). That METR-LA is more difficult is clear from the elevated errors above PeMS-Bay in our results (Tables 4.1 and 4.2) and the common periods of difficult in the predictions on METR-LA (Fig. 4.4). Secondly, in respect of earlier papers, we have been unable to replicate the results. The design of our models, with the exception of the graph estimation modules, closely follows the traffic transformer presented by (Cai et al., 2020). We contacted the authors to ask for comment on our results but we received no response.

On both datasets the models where the graph is estimated in some way generally outperform the models using the default topological graph. The error on the diffusion models is likewise ameliorated. In the few cases where graph estimation does not help, the error worsens by no more than 1 mile an hour on MAE. It is clear therefore that by means of graph estimation a model can discover structures that are otherwise present neither in the original graph nor diffusion graph.

Whether attention improves prediction *on average* is not clearly confirmed one way or another in our results according to the tables of results Tables 4.1 and 4.2. On the PeMS-Bay dataset, directed attention performs best without graph estimation. With graph estimation it stands largely second-best to undirected attention. On the METR-LA dataset, however, directed attention figures as best or second-best only when there is no graph

Table 4.5: The minima, maxima, means and medians of the weights of the original graph, the 2-step diffusion graph and the estimated graphs on 20 uniformly randomly sampled observations from the test set of the PeMS-Bay dataset. The statistics on the estimated graphs is the minimum *etc.* across all graphs before the average graph is computed. "A" is attention, "DA" is directed attention, "1d" is 1-step diffusion and "2d" is 2-step diffusion.

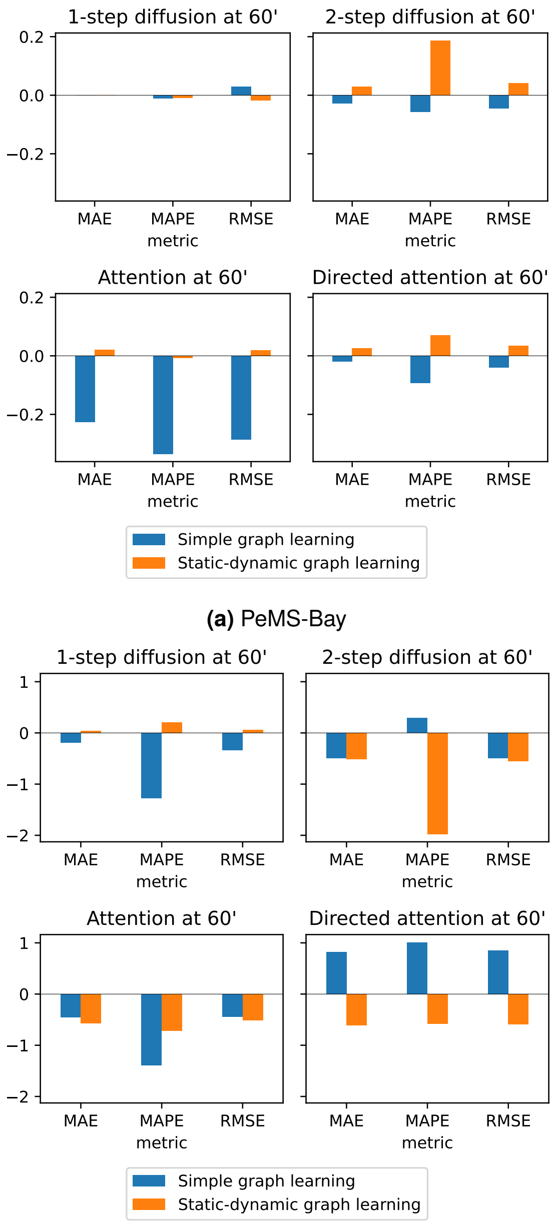
Stats	Original	2-step diffusion	Simple estimation				Static–dynamic			
			A	DA	1d	2d	A	DA	1d	2d
min.	0.00	0.00	0.71	0.67	0.66	0.61	0.53	0.58	0.68	0.68
max.	2.00	3.42	3.39	3.19	3.14	2.93	3.07	3.30	3.15	3.22
mean	0.03	0.06	2.93	2.78	2.72	2.53	2.61	2.84	2.73	2.78
median	0.00	0.00	3.00	2.84	2.79	2.59	2.69	2.93	2.80	2.85

Table 4.6: The minima, maxima, means and medians of the original graph, the 2-step diffusion graph and the estimated graphs on 20 uniformly randomly sampled observations from the test set of the METR-LA dataset. The statistics on the estimated graphs is the minimum *etc.* across all graphs before the average graph is computed. "A" is attention, "DA" is directed attention, "1d" is 1-step diffusion and "2d" is 2-step diffusion.

Stats	Original	2-step diffusion	Simple estimation				Static–dynamic			
			A	DA	1d	2d	A	DA	1d	2d
min.	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
max.	2.00	2.77	4.62	2.51	4.17	5.44	4.59	5.54	5.57	5.82
mean	0.03	0.06	3.22	1.61	3.00	3.79	3.25	4.09	4.05	4.22
median	0.00	0.00	3.09	1.69	3.01	3.67	3.45	4.31	4.30	4.27

estimation. When there is graph estimation, the 1-step diffusion is predominately better. Directed attention never vies for first or second place when graph estimation is used.

A similarly mixed picture presents itself with respect to the *average* errors with the different strategies to graph estimation. On the PeMS-Bay dataset, the best models using simple graph estimation attain better predictions on average across the board than the best models using static–dynamic fusion. Additionally the second-best models using simple graph estimation outperform the second-best of those models using static–dynamic fusion.



(a) PeMS-Bay

(b) METR-LA

Figure 4.2: The difference between the error when there is no graph estimation and simple graph learning (blue) and the static–dynamic graph learning (orange) at sixty minutes for each metric on each dataset. Each plot is a comparison within a single model. The y-axes of each graph have the same range to allow comparison of differences between models.

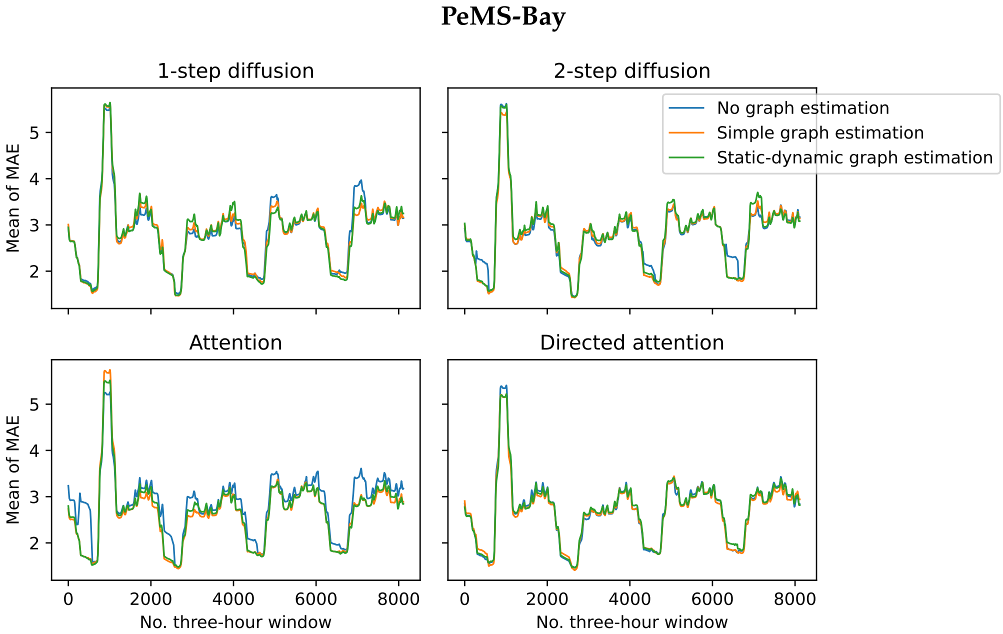


Figure 4.3: A linegraph plotting the rolling average in three-hour windows on the test set of the PeMS-Bay dataset. In a few places graph estimation flattens spikes in the error.

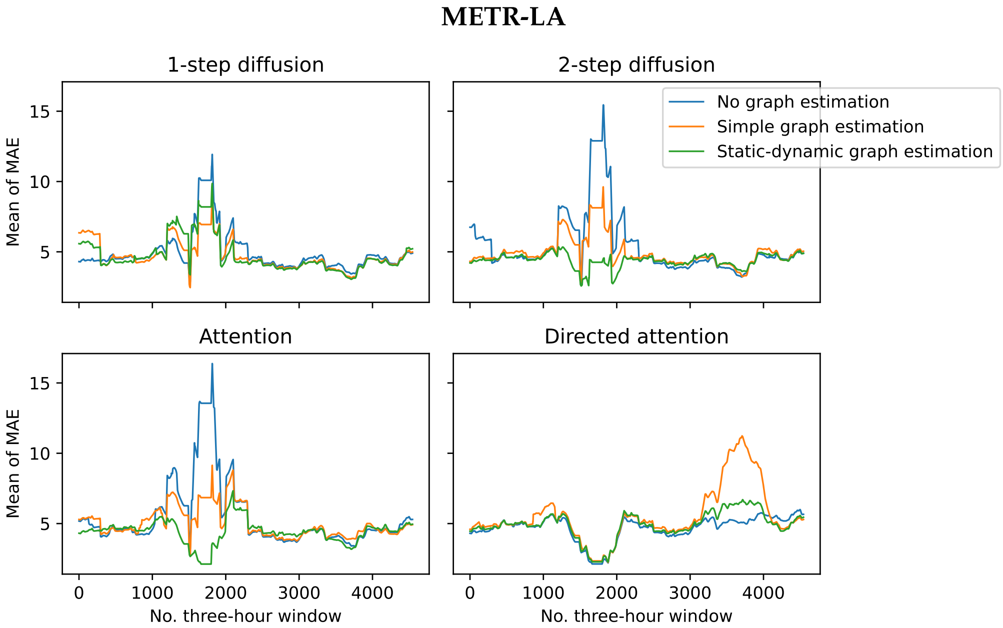


Figure 4.4: A linegraph plotting the rolling average in three-hour windows on the test set of the METR-LA dataset. Note that between the 1,000th and 3,000th hours the error consistently rises except in the directed attention model.

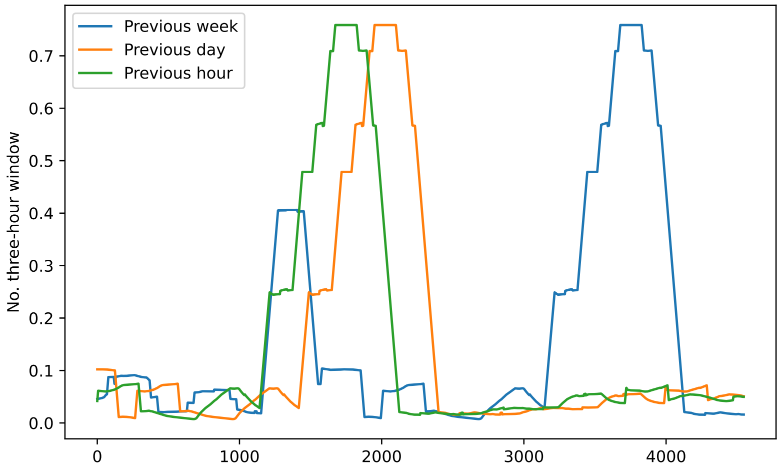


Figure 4.5: The proportion of sensors with missing data was computed at each timestep and a three-hour rolling average was taken over the predictions on test sequence on the METR-LA dataset. This figure has been adapted from Kenning and Xie (2023).

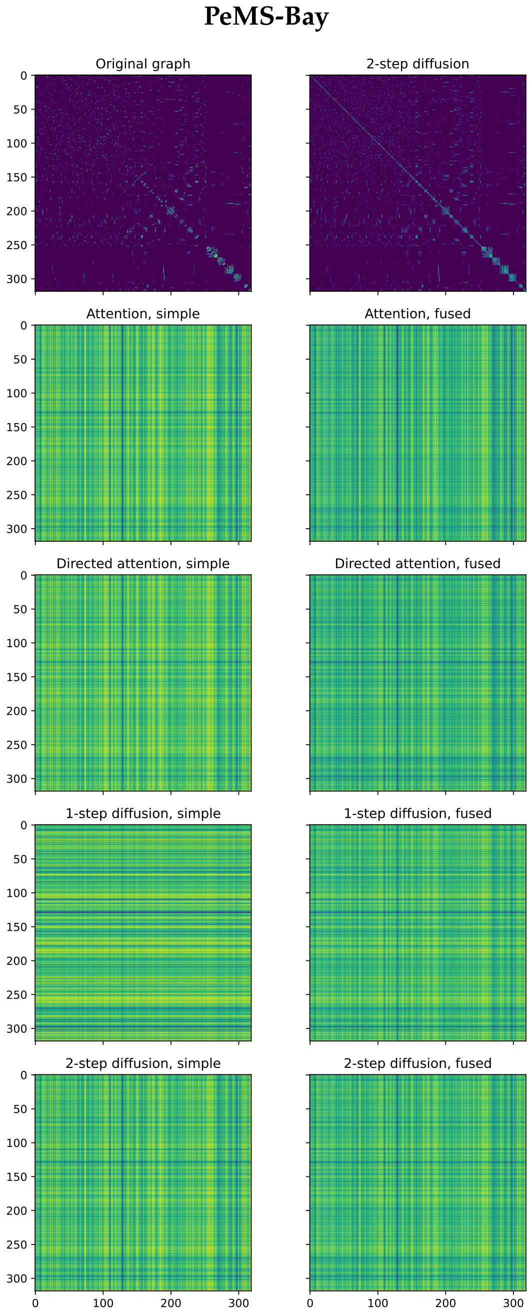


Figure 4.6: The figure visualises the original graph of the PeMS-Bay dataset and the learned graphs of each estimation approach. The first row are visualisations of the topological graphs—on the left the original graph and on the right the two-step diffusion graph. The remaining rows are divided into two columns. Each row corresponds to a model. The visualisations on the left come from the simple graph estimation, and the columns on the right come from static–dynamic fusion.

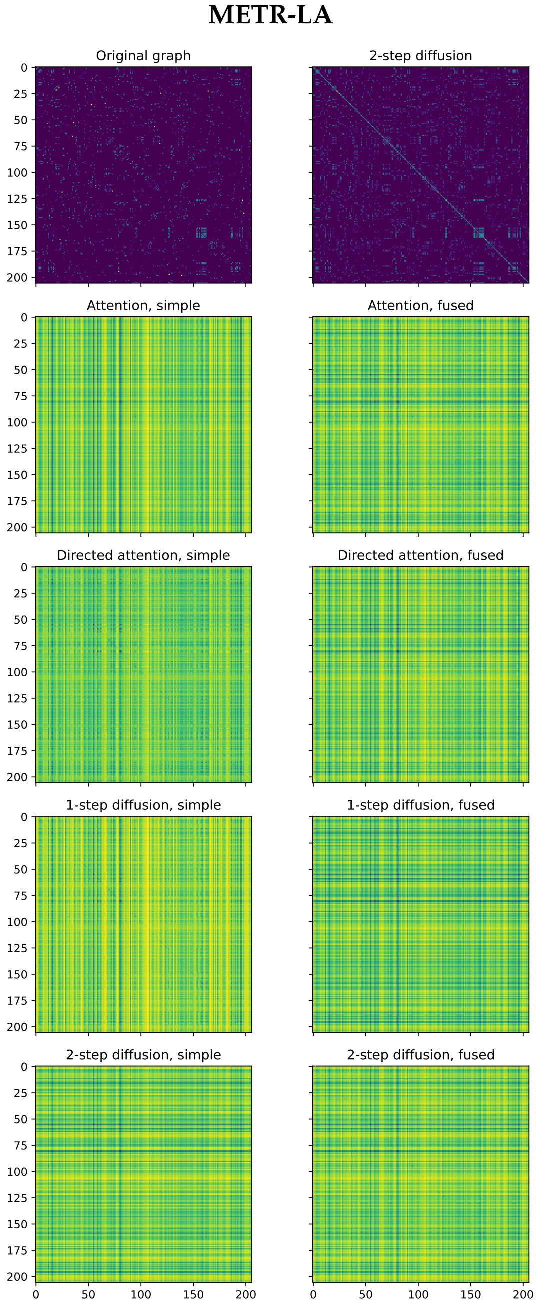


Figure 4.7: The figure visualises the original graph of the METR-LA dataset and the learned graphs of each estimation approach. The first row are visualisations of the topological graphs—on the left the original graph and on the right the two-step diffusion graph. The remaining rows are divided into two columns. Each row corresponds to a model. The visualisations on the left come from the simple graph estimation, and the columns on the right come from static–dynamic fusion.

Table 4.7: The average (μ) and standard deviation (σ) of the set of mixture coefficients β of each model and graph estimation strategy on the METR-LA dataset. A value more than 0.5 indicates that the static graph is weighted more heavily. A value less than 0.5 indicates that the dynamic graph is weighted more heavily.

Model, strategy	μ	σ
Attention, simple	0.502	0.055
Directed attention, simple	0.499	0.057
1-step diffusion, simple	0.503	0.057
2-step diffusion, simple	0.488	0.055
Attention, fused	0.498	0.009
Directed attention, fused	0.500	0.010
1-step diffusion, fused	0.502	0.013
2-step diffusion, fused	0.504	0.007

Table 4.8: The average (μ) and standard deviation (σ) of the set of mixture coefficients β of each model and graph estimation strategy on the PeMS-Bay dataset. A value more than 0.5 indicates that the static graph is weighted more heavily. A value less than 0.5 indicates that the dynamic graph is weighted more heavily.

Model, strategy	μ	σ
Attention, simple	0.503	0.049
Directed attention, simple	0.496	0.049
1-step diffusion, simple	0.497	0.051
2-step diffusion, simple	0.489	0.048
Attention, fused	0.494	0.013
Directed attention, fused	0.506	0.015
1-step diffusion, fused	0.499	0.013
2-step diffusion, fused	0.497	0.016

The graph estimation strategies distinguish themselves best on the METR-LA dataset. It is not clear initially from the average prediction errors alone that the graph strategies are much better than the others: a more complicated picture emerges from that perspective. While the best models using simple graph estimation are better than the best ones using static–dynamic fusion in the *short-term*, the second-best models using simple graph estimation are *worse* than the second-best ones using static–dynamic fusion. In the long-term (i.e. 60 minutes) the static–dynamic graph estimation is better. We can see some similarities in the performance improvements compared to no graph estimation by studying Fig. 4.2. Directed attention and 2-step diffusion experience similar profiles of improvement and worsening on the three metrics for simple and static–dynamic graph learning respectively. Undirected attention and 1-step diffusion experience similar profiles of improvement on the three key metrics with simple graph learning. Static–dynamic graph learning improves undirected and directed attention and 2-step diffusion in similar ways, although 2-step diffusion is improved to a greater degree. 1-step diffusion does not undergo any significant improvements or worsening in error with graph estimation on both datasets, with the exception of simple graph estimation on the METR-LA dataset, where the greatest difference is in the MAPE.

The greatest and most significant differences manifest themselves in the three-week rolling average of the MAE, as we remarked in Section 4.5.3.1. Further analysis in Fig. 4.5 shows that at points of high concentrations of missing data, the static–dynamic fusion is better able to correct itself than the other models. The static–dynamic fusion approach is therefore generally more robust to missing data than no estimation and even the simple graph estimation. This fact is bourne out in comparison with simple graph estimation, which, in combination with directed attention, undergoes a significant surge in error around the 3,750th window in Fig. 4.4. In studying Fig. 4.5, it is clear that static–dynamic fusion is able to balance the two sources of information from the long- and short-term, which is what renders it more robust to error. Moreover, even if the predictions are *on average* as accurate or only less accurate, the models using the static–dynamic fusion for graph estimation are more robust to missing data. The only exception is the directed attention with no estimation at all, which appears to be better on the test set than directed attention fed by a graph formed of static–dynamic fusion.

The graphs estimated in both approaches are both extremely dense. As we remarked in Section 4.5.3.2, there are conspicuous lines that criss-cross the graph visualisations. It suggests that vertices are globally determining the contributions in neighbourhoods. The structure is thus significantly different from the original and diffusion graphs in that it includes more global connections. We believe that the graph estimations are learning redundant edges, which is the reason that predictions are so stable in comparison to the other models (see Fig. 4.4). The static–dynamic architecture might be producing redundant structures that are separately discovered in the long- and short-term sequences. The dropout layer after the two structures are merged could account for the high-level of redundancy in the structure, in which case it is producing a dense graph.

Our observations of the error over time, in light of the proportion of missing data in the METR-LA dataset and the dense but highly distributed and redundant structure in the graph, lead us to two conclusions: (1) the static–dynamic fusion is able to correct for an over-reliance on new information by considering old information flexibly; and (2) the high level of redundancy in the traffic graphs improves the robustness of the model to missing data by drawing traffic information from several sources. The two conclusions would explain why the static–dynamic fusion approach to graph estimation lowers sensitivity to missing data across the board.

4.7 Summary

The difference between the performances of the models on the PeMS-Bay and METR-LA datasets is likely a matter of data complexity. As has been remarked in earlier work (Li et al., 2018a), the METR-LA traffic network has more complicated traffic conditions than the PeMS-Bay dataset. Certainly the graph estimation approaches are leaving the predictions more robust to sudden changes, and combined with the directed graph attention it is yet stabler.

Certainly there is room for improvement. The differences in performance between the four convolutional approaches are not significant on average. The advantage afforded by the directed attention is principally visible in the stability of the error, which is apparently better still than the static–dynamic fusion of information. The static–dynamic fusion technique needs to be refined, though. The last third of the METR-LA test set is curious (Fig. 4.4). It is not clear why the static–dynamic estimation causes such an error, whereas elsewhere it is *more* stable than no graph estimation.

There are several aspects of the fusion approach that need to be addressed. (1) **Density.** As we observed, the graphs are very dense and it leads to inefficient models. If the graph is detecting redundant edges that help robustness, this is good, but too many redundant edges is computationally suboptimal. Some means of pruning the graph more harshly is necessary; the L_1 norm and dropout are not alone sufficient—and dropout may even be making it worse. A simple threshold is desirable but arbitrary, and would need to be theoretically justified. We noticed that the median value on the original graph and diffusion graph is zero. Perhaps a target of keep the median as close to zero as possible would serve to counteract the tendency to very dense graphs. As we avoided the use of softmax in the graph estimation's attention mechanism, for reasons given in Section 4.3.3, we had no means of normalising the weights of the graph, which constitutes another point of improvement. (2) **Efficiency.** A sparser graph would lead to more efficient calculations. A second speed-up would come from the way the graph is used. Instead of using the attention coefficients to build a much larger graph, perhaps they could be used directly in the convolutional layers.

Chapter 5

Estimation of Graph Complements: Predicting Molecular Properties

Contents

5.1 Introduction

Graph-based deep learning approaches have been applied in a notably different way to the prediction of molecular properties. It is common in the literature on graph deep learning for molecular representations to consider both vertex and edge features (see Sections 2.5.6 and 2.6.3). Edge features are incorporated into convolutions on molecular graphs in various ways. They can be included in the convolution itself (Yang et al., 2019; Gasteiger, Groß, and Günnemann, 2020; Gasteiger et al., 2020). Alternatively the edge features can be fed to a separate pipeline based on edge adjacencies, which has a similar form to linegraphs, as node-central and edge-central encoders (Kearnes et al., 2016; Ma et al., 2022). The cross-dependent multi-view graph neural network proposed by Ma et al. (2022), for example, considers the atom and bond features equally for constituting a molecular representation. It consists of a node-central and edge-central autoencoder for propagating the features of vertices and edges. The representations learned for each are used in the other's aggregation function in the next propagation step. Although implicitly used, as far as we know, linegraphs have never been explicitly used in the literature.

The aforementioned methods principally concern the features on vertices and edges that exist in the molecular graph. Other methods introduce mechanisms to learn higher-order interactions on the graph. Maziarka et al. (2020) modified a transformer architecture (see Section 2.4.2.1) to apply attention over the molecular graph's attention matrix augmented by an identically shaped matrix of interatomic distances—which in fact could be replaced by any other measure of proximity or relation. The model is therefore able to attend over the entire molecule. Additionally the model contains a dummy vertex that connects to every vertex in the graph. Interestingly, the addition of edge features to the model worsened results. The HMGNN (Shui and Karypis, 2020) compiles higher-order interactions in the graphs by composing a global representation from several orders of partial structure in a heterogeneous graph, so-called many-body interactions. Although representations are passed between orders, the final output of each order is learned as a separate task, so the whole model operates as a multi-task problem. Removing these connections between orders worsens the performance.

The ever-present question is what interpretative advantage higher-order interactions grant a model that their absence removes. Few answers are apparent in the literature. We speculate that the global features are representing intramolecular interactions in the molecule. Various forces act upon a molecule that are relevant to a graphs, forces that act upon the particles in a molecule in ways beyond what is specified by chemical bonds. For example, van der Waals forces occur within molecules between chemically unbonded atoms and the structure of large proteins is determined by intramolecular forces. The simple model of chemically bonded atoms represented as a graph does not immediately capture these intermolecular forces. Alternatively the learned global representations might also represent outside forces or surfaces in the molecule that act as a repository for information during specialisation (Xia et al., 2022; Xiong et al., 2020). Global features or dummy vertices (Gilmer et al., 2017; Maziarka et al., 2020) and many-body interactions (Shui and Karypis, 2020). Alternatively, as we propose here, one could however capture these excluded forces (Xiong et al., 2020) by representing them as a complement graph. Moreover one could infer and learn these interactions from molecular features using some graph estimation approach. This thought motivates our use of complement graphs to estimate molecular structure.

There is an additional advantage in estimating the graph complement rather than the full graph. In full graph estimation, as in Section 4.3.3, the full weight matrix of the graph was computed using an attention mechanism. The attention mechanism is of a fixed size, of course, and does not grow with the size of the graph; but the number of computations means that an attention coefficient is computed for every pair of vertices in the graph, for which a gradient has to be computed, n^2 in total. We had to use various mechanisms during training to reduce this load, such as a weight threshold and a dropout layer. In contrast, the estimation of graph complements entails that only a subset of the weights need to be estimated, specifically $n^2 - m$, where n is the graph order, i.e., the number of vertices, and m is the graph size, i.e., the number of edges.

The directed graph is likewise a useful tool in graph learning. In DimeNet (Gasteiger, Groß, and Günnemann, 2020; Gasteiger et al., 2020) a directed graph is used to model vertex-wise interactions with neighbours, specifically the relative contributions of neighbours to a focal vertex with respect to bonding angle and atomic distance. Yang et al. (2019) proposed the use of a directed graph to prevent tottering by preventing message updates from receiving the contributions of non-adjacent directed edges, a technique that they call bond-level message-passing.

In this chapter we propose several advances on previous approaches. Firstly we design a method to approximate intramolecular structures from molecular features to complement the chemical structures represented in the molecular graph. We expect that the estimated structure will reflect higher-order, longer-distance interactions in the molecule. Consequently, since the propagation is broadened by the estimation of additional structure, we expect that the optimal number of propagation steps will be lower with graph estimation than without. At the core of the graph estimation approach is a new graph formulation, the *complement graph*, which reduces the number of parameters to estimate in the graph estimation procedure. The graphs we use are directed. We therefore use the a modified version of the directed attention-based graph convolution proposed in Section 4.3.4.1. Convolution is performed on the predefined and estimated graphs separately, which allows the model to factorise contributions over the vertex and edge features into chemical bonds on the one hand and intramolecular forces on the other. The model is thus equipped with the interpretive power to distinguish the two structures, unlike Maziarka et al. (2020), in addition to distinguishing two pipelines for vertices and edges, as in Kearnes et al. (2016) and Ma et al. (2022).

5.2 Proposed Approach

We propose a new approach to joint graph estimation on molecules that we term *complement learning*. The basis of complement learning is the complement graph, which together with the molecular graph is the basis for the propagation of vertex features. We also use the complement of a linegraph to learn the propagation of edge features. The contributions of this chapter are the following:

1. a new graph which we refer to as the *complement graph*;
2. a new way to learn edge features that is structured jointly by a *linegraph* and *linegraph complement*;
3. a new graph estimation strategy that leverages the complement graph; and
4. a new model for learning graph complements end-to-end for molecule representation learning.

5.3 Methodology

In this section we describe the methodology used to estimate graphs on molecules that are applied in predicting molecular properties. In Section 5.3.1 we describe in theoretical terms what the graph complement is. In Section 5.3.2 we describe how we estimate the graph complement. Finally in Section 5.3.3 we describe the model in which we learn the graph complements end-to-end and how they are used in learning molecular representations.

5.3.1 The Graph Complement

The *graph complement* $\bar{G} = (V, \bar{E})$ is constructed on a graph $G = (V, E)$. $\bar{G}$ shares the same vertex set V as G, but they differ in the edges. For an undirected graph G the edge set E is may be described as a subset of the set of unordered pairs of V denoted $\mathcal{V} = \{\{x, y\} \mid \forall x, y \in V\}$ where $\{x, y\}$ is an unordered pair. The graph complement's vertex set is therefore defined as $\overline{E} = \mathcal{V} \setminus E$. In other words, the edges in $\overline{G}$ are those edges that are not present in G. The *directed graph complement* is defined similarly. Overloading notation, let $\mathcal{V} = \{(x, y) \mid \forall x, y \in V\}$ denote the set of ordered vertex pairs (x, y) over V of a directed graph G.

The *linegraph complement* is again defined similarly. As we described in Section 2.5.2.5, the linegraph $L(G)$ is defined on an underlying graph $G = (V, E)$ is defined $L(G) = (E(G), E_L)$, where E_L records the adjacencies between edges E in the underlying graph (see Section 2.5.2.5). We define the linegraph complement as $\overline{L(G)} = (E(G), \overline{E_L})$, such that $\overline{E_L} = \mathcal{E}_L \setminus E_L$, where $\mathcal{E}_L$ is the set of unordered pairs of linegraph vertices. The *directed linegraph complement*, like the directed graph complement, uses the set of ordered rather than unordered pairs of linegraph vertices, and the rest of the definition follows the same lines. To all graph complements, since they are all essentially graphs, the same properties apply as are described Section 2.5.2.

5.3.2 Estimating the Graph Complements

We use the idea of the complement graph described in Section 5.3.1 to design our graph estimation. The estimation model we use utilises an attention mechanism similar to the one described in Section 4.3.3. We rely on the definitions of the directed graph and linegraph complements and the description from this point assumes that directed graphs are used across the board. In total there are four graphs in the GCNN: a graph G, a linegraph $L(G)$ and their respective complements $\overline{G}$ and $\overline{L(G)}$. The graph is the fixed structure of the molecule. A molecule is conventionally represented as a graph G by taking the atoms as vertices and the chemical bonds as edges. The linegraph is constructed from the graph in a deterministic manner (Aigner, 1967). The linegraph represents the second-order structure of the graph, specifically the bonding structure of a given molecule. The graph and linegraph complements are learned end-to-end in the graph estimator. Strictly speaking, the complements estimated by graph estimation approach are spanning graphs (see Section 2.5.2.7) of their complements; it is possible that only a subset of the edges of the complement are present in the estimated graphs. For simplicity's sake we refer to the estimated spanning graphs as complements.

The graph complement estimation is separate for the graph complement and linegraph complement, but the formulation is the same either way. In our model we estimate the two graph with identical processes in separate blocks with different learned parameters. Therefore, although we describe the model for the graph and its complement, the same process applies to the linegraph and its complement. A crucial difference, however, is that the linegraph complement estimator is fed edge features for vertex features. The estimation block is supplied with a set of vertex features $f(V) \in \mathbb{R}^{n \times c}$ and a graph G with a weight matrix $\boldsymbol{W} \in \mathbb{R}^{n \times n}$. The vertex features are then linearly projected into the dimensionality of the estimator d_e:

$$f'(V) = f(V)\boldsymbol{W}, \tag{5.1}$$

where $\boldsymbol{W} \in \mathbb{R}^{c \times d_e}$ is the projection matrix.

The projected vertex features are then used to compute the attention coefficients for each pair of vertices. For vertices $x, y \in G$, the attention coefficient is computed by concatenating their features and applying an attention mechanism:

$$C_{xy} = (f'(x) \parallel f'(y))\boldsymbol{a} \tag{5.2}$$

where $\boldsymbol{a} \in \mathbb{R}^{2d_e}$ is the attention mechanism. The operation yields a full attention matrix $\boldsymbol{C} \in \mathbb{R}^{n \times n}$, which stores an attention coefficient for each adjacency in the full graph complement. In practice, the full matrix need not be estimated. As we mentioned in the introduction in Section 5.1, only the edges of the graph complement need to be estimated. In this theoretical explanation, however, we can simply use a mask $\boldsymbol{M} \in \mathbb{R}^{n \times n}$, defined as

$$\boldsymbol{M} = \begin{cases} 1 & A_{xy} = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.3}$$

where A_{xy} is the entry in the adjacency matrix $\boldsymbol{A}$ of the input graph G for the vertices $x, y \in G$. The map is then used to mask the entries corresponding to positions in the attention matrix and the result is passed through a leaky ReLU layer with $\alpha = 0.2$:

$$\boldsymbol{C}' = \mathrm{LeakyReLU}\left(\boldsymbol{C} \odot \boldsymbol{M}\right), \tag{5.4}$$

where $\boldsymbol{C}'$ is the masked attention matrix and $\odot$ is a binary operation representing the Hadamard product of $\boldsymbol{C}$ and $\boldsymbol{M}$. Unlike Eqs. 4.9 and 4.11 in Section 4.3.3, in this model we replace the ReLU layer applied to the attention coefficients with a leaky ReLU layer with $\alpha = 0.2$. In experiments we found that the estimated graph very quickly dies as all values become zero. As a consequence the estimated weights are not bound between $[0, \infty)$ rather $(-\infty, \infty)$, although there is an implicit bias towards positive values.

Additionally we introduce no regularisation term as in Section 4.3.3 for the same reason that we do not use ReLU.

The estimation process described above is computed over h attention heads. It therefore yields h estimates of the graph complement. As in Section 4.3.3, we average the matrices over these heads to yield a single attention matrix, which is the estimated weight matrix of the graph complement.

At this point we could apply several pruning mechanisms as we did in Section 4.3.3, but in reality it is not necessary. The orders of the graph and graph complement are actually equal, i.e., both graphs have the same number of vertices by definition, and the datasets we use in the experiment in Section 5.4 are not large enough to warrant it. We therefore do not apply any dropout or threshold to the output. We also do not add a regularisation to the loss function of the sum of weights, nor do we constrain the values of the projection matrix and attention function to be in the interval $[0, 1]$ as in Section 4.3.3.

Complement estimation is reliant entirely on the vertex features to estimate the graph complement. It is therefore affected by the choice of features for the vertices, or in the case of the linegraph complement, the choice of edge features. Potentially this mechanism could be extended to include edge features to estimate the graph complement, or vertex features to estimate the linegraph complement. We leave a development of such an approach to future work.

5.3.3 Model Structure

The full model architecture is illustrated in Fig. 5.1 and the pipeline is described algorithmically in Algorithm 1. The model consists of an estimator module, consisting of two blocks for the two estimated graph complements, and a molecule representation learner, the message-passing phase. The output of the molecule representation learner is a set of embeddings on each vertex; the average of those representations is taken as the input to the readout layer, which computes the prediction on an observation.

The model is supplied with a set of vertex features $f(V) \in \mathbb{R}^{n \times c}$, a set of edge features $f(E) \in \mathbb{R}^{m \times c}$, a graph G as an adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and a linegraph $L(G)$ as an adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{m \times m}$. For ease of reading, we do not refer to the graphs as their adjacency matrix here unless the mathematics demands it. On these inputs, the model predicts a target $\boldsymbol{y} \in \mathbb{R}^{d}$ where d is the number of targets. In the first step, the vertex $f(V)$ and edge features $f(E)$ are linearly projected into a d_m-dimensional feature space

Algorithm 1 The pipeline of the molecular prediction using our model. Note that the vertices and edges are flipped in order when they are propagated over the linegraph $L(G)$. This is because the edges of the graph G are mapped to the vertices in $L(G)$; correspondingly the edge features structured on the vertices of G are structured on the vertices of $L(G)$. Likewise the vertex features structured on the vertices of G are structured on the edge of $L(G)$.

```
$k$, maximum number of iterations
$s$, flag indicating whether graph estimation is enabled $\hbar$
$G \leftarrow$ structure of the molecule
$L(G) \leftarrow$ generate the linegraph from $G$
$f(V) \leftarrow$ projected vertex features                                ▷ Eq. 5.5
$f(E) \leftarrow$ projected edge features                                  ▷ Eq. 5.6
$\overline{G} \leftarrow$ estimated from $G$ and $f(V)$
$\overline{L(G)} \leftarrow$ estimated from $L(G)$ and $f(E)$
$i \leftarrow 0$
while $i < k$ do
    $f'(V) \leftarrow$ propagate $f(V)$ and $f(E)$ over $G$
    $f'(E) \leftarrow$ propagate $f(E)$ and $f(V)$ over $L(G)$
    if $s$ is true then
        $f''(V) \leftarrow$ propagate $f'(V)$ over $\overline{G}$
        $f''(E) \leftarrow$ propagate $f'(E)$ over $\overline{L(G)}$
    else
        $f''(V) \leftarrow f'(V)$
        $f''(E) \leftarrow f'(E)$
    end if
    $f(V) \leftarrow f''(V)$
    $f(E) \leftarrow f''(E)$
    $i \leftarrow i + 1$
end while
$f(V) \leftarrow$ merge embeddings $f(V), f(E)$ according to direction     ▷ Eq. 5.12
$|(z) \leftarrow$ compute molecular properties from $f(V)$
```

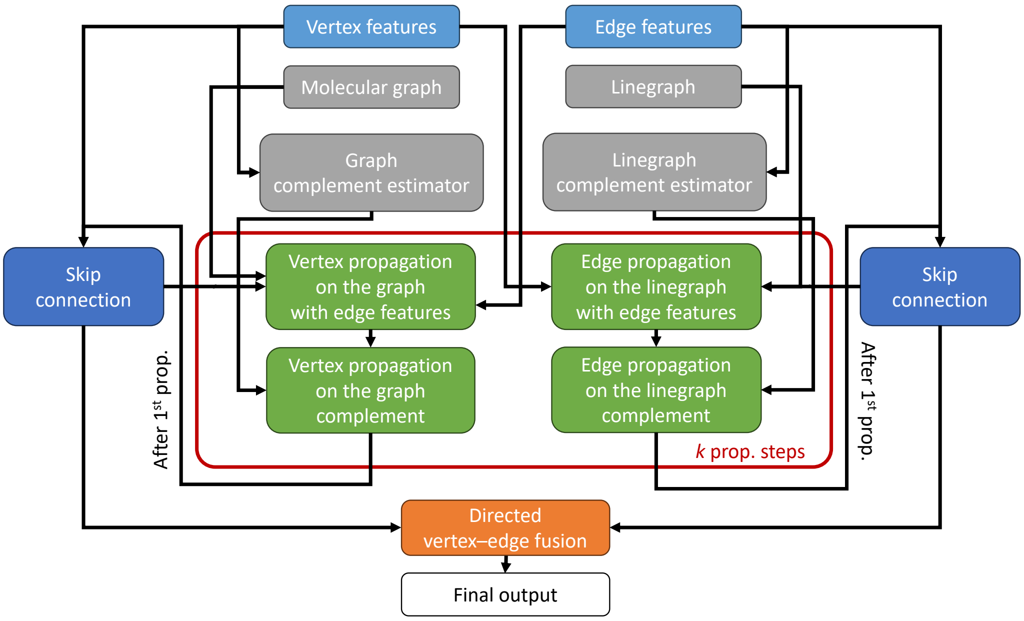

Vertex features
Edge features
Molecular graph
Linegraph
Graph complement estimator
Linegraph complement estimator
Skip connection
Vertex propagation on the graph with edge features
Edge propagation on the linegraph with edge features
Skip connection
Vertex propagation on the graph complement
Edge propagation on the linegraph complement
After 1st prop.
After 1st prop.
k prop. steps
Directed vertex–edge fusion
Final output

Figure 5.1: The full model architecture for complement estimation and molecular property prediction. The model consists of an estimator module (the grey boxes), from which the graph and linegraph complements are estimated and fed to the propagation layers (the green boxes), the output of which is passed to a layer that fuses the vertex and edge representations. The fused features are passed to a dense layer, which computes the prediction for the task. In the models without graph complement estimation, there is no second row of grey boxes for the complement estimation and therefore no second vertex and edge propagation, the second row of green boxes.

by projection matrices $W_V \in \mathbb{R}^{c \times d}, W_E \in \mathbb{R}^{c \times d}$:

$$f'(V) = f(V)W_V \tag{5.5}$$

$$f'(E) = f(E)W_E \tag{5.6}$$

where d_m is the size of the model's hidden layers.

The next step is the complement estimation, which is performed as described in Section 5.3.2. There are two estimator blocks with independent parameters to predict the complements. Together we call the two estimator blocks a module. To predict the graph complement $\bar{G}$, one estimator block is fed the embedded vertex features $f'(V)$ and the graph G. To predict the linegraph complement $\overline{L(G)}$, the other estimator block is fed with the embedded edge features $f'(E)$ and the linegraph G. The two complements are represented as weight matrices, reflecting a continuous relaxation of the graph estimation problem described in Section 2.5.7. Like the two input graphs, for ease of reading we refer to the complements simply as their objects $\overline{G}, \overline{L(G)}$ unless the mathematics demands a reference to their weight matrices.

The next step is propagation, the message-passing phase, which follows a procedure that is similar to methods we have discussed above (Kearnes et al., 2016; Ma et al., 2022). The propagation of the vertex and edge signals across the graph and linegraph has two stages using two consecutive directed attention-based graph convolution operations in two streams. The first operation modifies the form of the directed attention graph convolution in that the the vertex pairs' signals are joined in the attention mechanism by a third set of features: the edge feature of the edge that joins the two vertices. As such the vector of coefficients $\boldsymbol{a} \in \mathbb{R}^{3d}$ is broadened to $\boldsymbol{a} \in \mathbb{R}^{5d}$, where $\boldsymbol{a}_3$ and $\boldsymbol{a}_4$ are the vectors of attention coefficients for the edge features on in- and out-edges respectively. Supposing that the $\boldsymbol{h}_{l-1,xy}$ represents the previous layer's edge features, where $\boldsymbol{h}_{0,xy} = f(xy), xy \in E$, in practice it means that Eqs. 4.16 to 4.18 are modified in the following way:

$$c_{l,x}(x) = (3 \cdot \boldsymbol{a}_1)\boldsymbol{h}'_{l-1,x}, \tag{5.7}$$

$$c_{l,x}(\vec{yx}) = \begin{cases} \left[\boldsymbol{h}'_{l-1,y} \| \boldsymbol{h}'_{l-1,x}\right] \boldsymbol{a}_{0:1} + \boldsymbol{h}_{l-1,xy}\boldsymbol{a}_3 & \vec{yx}\,, \in E\,, \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

$$c_{l,x}(\vec{xy}) = \begin{cases} \left[\boldsymbol{h}'_{l-1,x} \| \boldsymbol{h}'_{l-1,y}\right] \boldsymbol{a}_{1:2} + \boldsymbol{h}_{l-1,xy}\boldsymbol{a}_4 & \vec{xy} \in E\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{5.9}$$

An additional modification is present here beyond the edge features. In Eq. 5.7 the attention on the focal vertex is tripled. Our reasoning is that the sum of three different sources of partial attention in Eqs. 5.8 and 5.9 will always be greater than the contribution of the vertex feature. The tripling of the focal vertex's partial attention counteracts this effect.

The second of the two convolution operations has the same form as the directed graph convolution we presented in Section 4.3.4.1. It receives the output of the first convolution, but no edge features. Indeed, there are no edge features to speak of: the edge features reflect the characteristics of the chemical bonds, and there are no characteristics to speak of between atoms that are not bonded. It is conceivable that some set of features could be measured for the second purpose; we leave such an extension of the present method to future work.

In both cases of convolution, with or without edge features, the graph convolution is modified to include the global attention coefficients from the respective graphs, as in Section 4.3.5. It must be noted here, too, that the weights of the graph and linegraph's edges are always 1, whereas the weights of the graph and linegraph complements are real numbers.

After each propagation step, there is a skip connection where the features before the propagation are added to the new features to mitigate against graph smoothing. The output of the second pair of convolutions yields vertex and edge features, which are used as input to the first convolution operations of the next propagation step. After all propagation steps have completed, the message-passing phase yields vertex and edge embeddings. We fuse the vertex and edge embeddings by aggregating the edge embeddings around incident vertices. For each vertex, the embeddings of its adjacent edges are aggregated and averaged according to incidence:

$$\boldsymbol{h}_{\Gamma_{\text{out}},x} = \frac{1}{d(x)} \sum_{y \in \Gamma_{\text{out}}(x)} \boldsymbol{h}_{l-1,xy} \,, \tag{5.10}$$

$$\boldsymbol{h}_{\Gamma_{\text{in}},x} = \frac{1}{d(x)} \sum_{y \in \Gamma_{\text{in}}(x)} \boldsymbol{h}_{l-1,yx} \,, \tag{5.11}$$

where $\boldsymbol{h}_{\Gamma_{\text{out}},x} \in \mathbb{R}^d$ is the average embedding of the out-edges incident to x and $\boldsymbol{h}_{\Gamma_{\text{in}},x} \in \mathbb{R}^d$ is the average embedding of the in-edges incident to x. We denote the full set of vertex-wise aggregations $\boldsymbol{h}_{\Gamma}$. Then the focal vertex's embedding and the averaged in- and out-edge embeddings are concatenated and passed through a dense layer consisting of a weight matrix $\boldsymbol{W} \in \mathbb{R}^{3d \times d}$, a bias vector $\boldsymbol{b} \in \mathbb{R}^d$ and an activation layer σ:

$$\boldsymbol{h}' = \sigma \left([\boldsymbol{h}_{l-1} \,\|\, \boldsymbol{h}_{\Gamma_{\text{out}}} \,\|\, \boldsymbol{h}_{\Gamma_{\text{in}}}] \, \boldsymbol{W} + \boldsymbol{b} \right) \,. \tag{5.12}$$

Finally for each molecule the vertex representations $\boldsymbol{h}'$ are averaged yielding a single d-dimensional representation for the whole molecule. The final representation is passed through an unactivated dense layer that yields the appropriate number of logits for the task.

5.4 Experiment

The experiments described here were conducted in July 2023. The code was programmed in Python version 3.9.12 and the models implemented in Tensorflow version 2.11.1. Each model was trained on 1 NVIDIA V100 GPU and 1 Intel Xeon Gold 6148 CPU core hosted by the Sunbird supercomputer at Swansea University. Owing to the nature of the implementation, namely the use of ragged tensors, Tensorflow had to be run in eager execution mode, which means that training and inference took significantly longer than it otherwise would.

5.4.1 Evaluated Models

We use variations on the same basic model to evaluate the efficacy of the proposed approach to graph estimation and its effect on feature propagation. We evaluated the model with and without the complement estimation modules. Disabling the complement module also meant disabling the convolutional layers structure by the graph and linegraph complements. In such a case the output of the first pair of convolutional layers was fed straight into the convolutional layers of the next propagation step. We also evaluated the model, with and without complement estimation, with different numbers of propagation steps ranging from one to five propagation steps. In total ten models were evaluated.

5.4.2 Datasets

We evaluated our models against two common benchmark datasets: the blood–brain-barrier penetration (BBBP) dataset and the estimated solubility (ESOL) dataset. The datasets were loaded using the DeepChem library version version 2.7.1 (Ramsundar et al., 2019), which implements Python loaders for the MoleculeNet benchmark datasets (Wu et al., 2018).

The BBBP dataset has 2,039 compounds represented as SMILES strings, from which the graph structure, vertex features and edge features were computed using the DeepChem library. Each compound is recorded as a binary label, where a positive label indicates that the compound is able to penetrate the blood–brain barrier. The task of the model is to learn from the structure of the compounds and its features whether the compound is able to penetrate the blood–brain barrier or not. As recommended by MoleculeNet (Wu et al., 2018), the dataset was split according to the scaffold splitting implemented in the DeepChem library.

The ESOL dataset has 1,128 compounds represented as SMILES strings, from which the graph structure, vertex features and edge features were computed using the DeepChem library. Associated with each compound is a measure of its log solubility in mols per litre. The task is to learn that solubility measure from the compound structure and its features. As recommended by MoleculeNet (Wu et al., 2018), the dataset was split randomly as implemented in the DeepChem library.

The SMILES strings of the compounds in both datasets were processed to obtain a graph, a set of vertex features and a set of edge features using the directed message-passing neural network featuriser provided by the DeepChem library. After featurising the compounds, they were filtered to remove any samples where the number of vertex features did not match the number of atoms in the molecule. In the end, of the 2,037 compounds in the BBBP dataset, 2,031 were used and 6 discarded, and of the 1,128 compounds in the ESOL dataset, 1,127 were used and 1 discarded.

5.4.3 Control Variables

For each dataset we used a batch size of 64; the training set from which the batches were drawn was shuffled after every epoch. The model used the AdamW optimiser with a learning rate $\eta = 1 \times 10^{-2}$ annealed using cosine decay over the first k steps to 1×10^{-4}. We trained the both models for 50 epochs. For the ESOL dataset we set $k = 625$ and for the BBBP dataset we set $k = 1125$. The directed graph attention layers has 64 channels across 8 attention heads, as do the attention mechanisms in the complement estimator. Likewise the hidden layers of the model have 64 channels. All activation layers use the ELU activation (Clevert, Unterthiner, and Hochreiter, 2016) where $\alpha = 1.0$.

The loss function for both datasets was the RMSE. The models trained on the BBBP dataset are evaluated on the area under the curve (AUC) and the model on the ESOL dataset are evaluated on RMSE.

5.4.4 Hypotheses and Questions

In this section we enumerate the hypotheses for the experiment, which namely correspond to our expectations of the results. These are the main points that we will be investigating in our analyses below.

Hypothesis 1 As the number of propagation steps increases, the RMSE/AUC will improve with or without complement estimation.

Hypothesis 2 The models using complement estimation will make more accurate predictions than the models without complement estimation.

The purpose of our investigation is to answer the following questions:

Question 1 Does graph estimation lower the number of propagation steps necessary to compute molecular properties?

Question 2 What effect does increasing the propagation steps have on the structure of the estimated graphs?

In answer to **Question 1**, if it is true, then we should observe the models' performance peaking sooner with fewer convolutions when graph estimation is used compared to where it is not. We do not expect that the issue of graph over-smoothing will not occur

Table 5.1: The test results on the BBBP dataset given as area under the curve (AUC). A higher value is better.

Complement estimation	No. propagation steps				
	1	2	3	4	5
No	0.703 68	0.704 31	0.699 39	0.685 47	0.699 63
Yes	0.722 71	0.708 93	0.720 54	0.703 87	0.702 09

Table 5.2: The test results on the ESOL dataset measure as root mean squared error (RMSE). A lower value is better.

Complement estimation	No. propagation steps				
	1	2	3	4	5
No	0.593 14	0.548 51	0.534 30	0.557 99	0.550 91
Yes	0.516 37	0.534 83	0.570 16	0.535 27	0.551 15

(see Section 2.5.5). We do expect that the performance will improve more quickly with fewer propagations, however, since the vertex and edge information is propagated more widely sooner than merely using the molecular graph. Whether the structures estimated in our approach reflect useful structures in learning is an extension of the question.

Question 2 relates to the structure of the complement graphs learned in the estimation. We wish to study whether there is a relationship between the structure of the graph and linegraph complements and the prediction errors of the models. As opposed to the molecular graph and linegraph, which have binary adjacency matrices, the graph and linegraph complements are represented by real-valued weight matrices. The values of the weight matrices are learned end-to-end in the model and are adjusted by gradient descent according to the performance on the given task. An investigation with respect to this question establishes the relationship between the prediction and the graph complements' structures.

Note that we are not aiming for state-of-the-art results in this analysis; rather we are using the datasets to elucidate the effect of complement estimation on regression and classification performance.

5.5 Results

For each dataset we present a table of results describing the performance of the models over increasing numbers of propagation steps. Firstly we examine the results on each dataset separately by considering the hypotheses we enumerated above, before drawing broader conclusions about the graph estimation procedure in a later subsection.

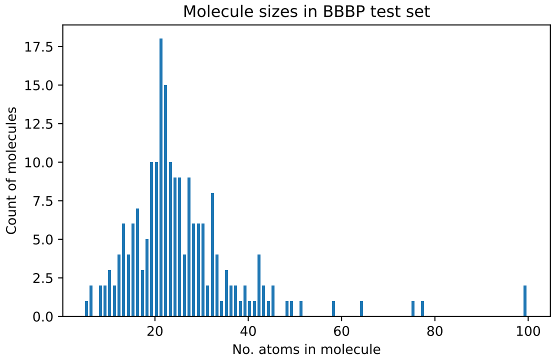


Figure 5.2: A histogram of molecule size, i.e. the number of atoms in each molecule, in the BBBP dataset.

Blood–brain-barrier penetration (BBBP) The results of the models with and without graph estimation over 1 to 5 propagation steps are presented in Table 5.1. The best result on BBBP without complement estimation was attained at 1 propagation step. The best result on BBBP with complement estimation was attained at 2 propagation steps. We could not confirm **Hypothesis 1** since the results of neither model describes an upwards trend in AUC as the number of propagation steps increases. We can however confirm **Hypothesis 2** partially. At every propagation step the model with complement estimation attains a higher AUC than the model without graph estimation. Moreover the model with complement estimation, namely at 1 propagation step, attains the highest AUC score across the board.

Estimated solubility (ESOL) The results of the models with and without graph estimation over 1 to 5 propagation steps are presented in Table 5.2. The best result on ESOL without complement estimation was attained at 3 propagation steps. The best result on ESOL with complement estimation was attained at 1 propagation steps. We could not confirm **Hypothesis 1**, since there is no clear downward trend in the RMSE for models both with and without complement estimation as the number of propagation steps increases. We can however partially confirm **Hypothesis 2**: The error with complement estimation is lower at 1, 2 and 4 propagation steps than the model without graph estimation, whereas the error of the model with graph estimation is higher than the model without at 3 propagation steps and almost on par at 5 propagation steps. Overall the lowest error is attained by a model using complement estimation, namely at 1 propagation step.

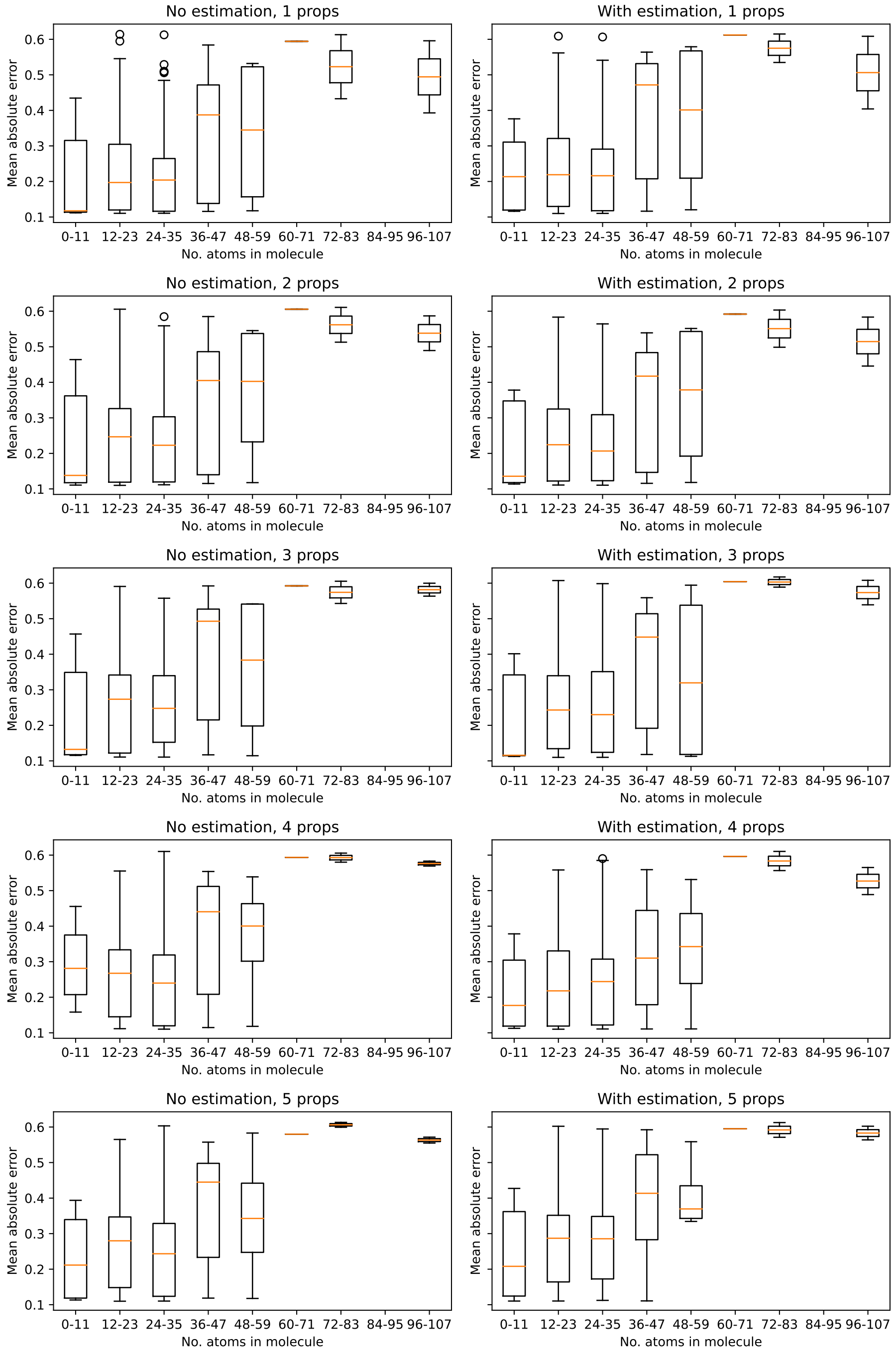


Figure 5.3: The mean absolute error of the prediction was aggregated into 9 bins in the BBBP test set and plotted statistics of the prediction errors of each bin as a boxplot. The plots in the left column visualises the predictions from models without complement estimation; the right column visualises the predictions from models using complement estimation. Each row represents a number of propagation steps, going from the top, 1 propagation step, to the bottom, 5 propagation steps.

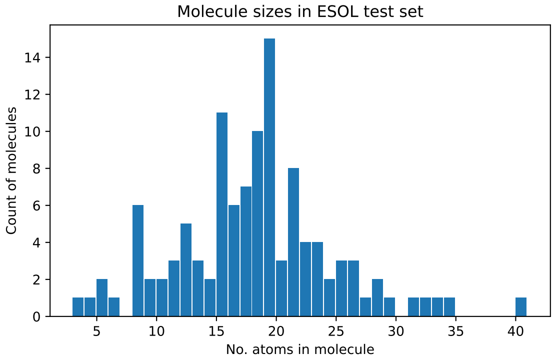


Figure 5.4: A histogram of molecule size, i.e. the number of atoms in each molecule, in the ESOL dataset.

5.5.1 Data Analysis

In answer to **Question 1**, we now consider some characteristics of the dataset. For Figs. 5.2 and 5.4 we plotted histograms of the molecule sizes in the test sets of the BBBP and ESOL datasets respectively. For Figs. 5.3 and 5.5 we aggregated the errors in different sized bins and visualised statistics on the prediction errors at each bin as boxplots.

BBBP. We can see that between the models with and without complement estimation there is not a great deal of difference in the performance of the models with respect to molecule size, as shown in Fig. 5.3. The majority of the molecules are between 10 and 30 atoms large. There are not many discernable differences between the models with respect to the atoms of those sizes. The predictions on these larger molecules are on average worse than the predictions yielded by the model with graph estimation on the same molecules. Compared at each propagation in general we can see that as the number of propagation steps increases, the predictions on the larger molecules become more precise, but also more wrong, although this is probably because there are so few that size; though the models with complement estimation are slightly more correct at every propagation step. We can also see that the model with complement estimation at 1 propagation step has fewer outliers for molecules 24 to 35 atoms large in comparison to the models without complement estimation. The same model with complement estimation has a higher interquartile range on molecules between 36 and 59 molecules large in comparison to the model without complement estimation at 1 propagation step. Ultimately that accounts for comparatively few molecules. Comparing the best model with complement estimation (1 propagation step) with the best model without complement estimation (4 propagation steps), we can

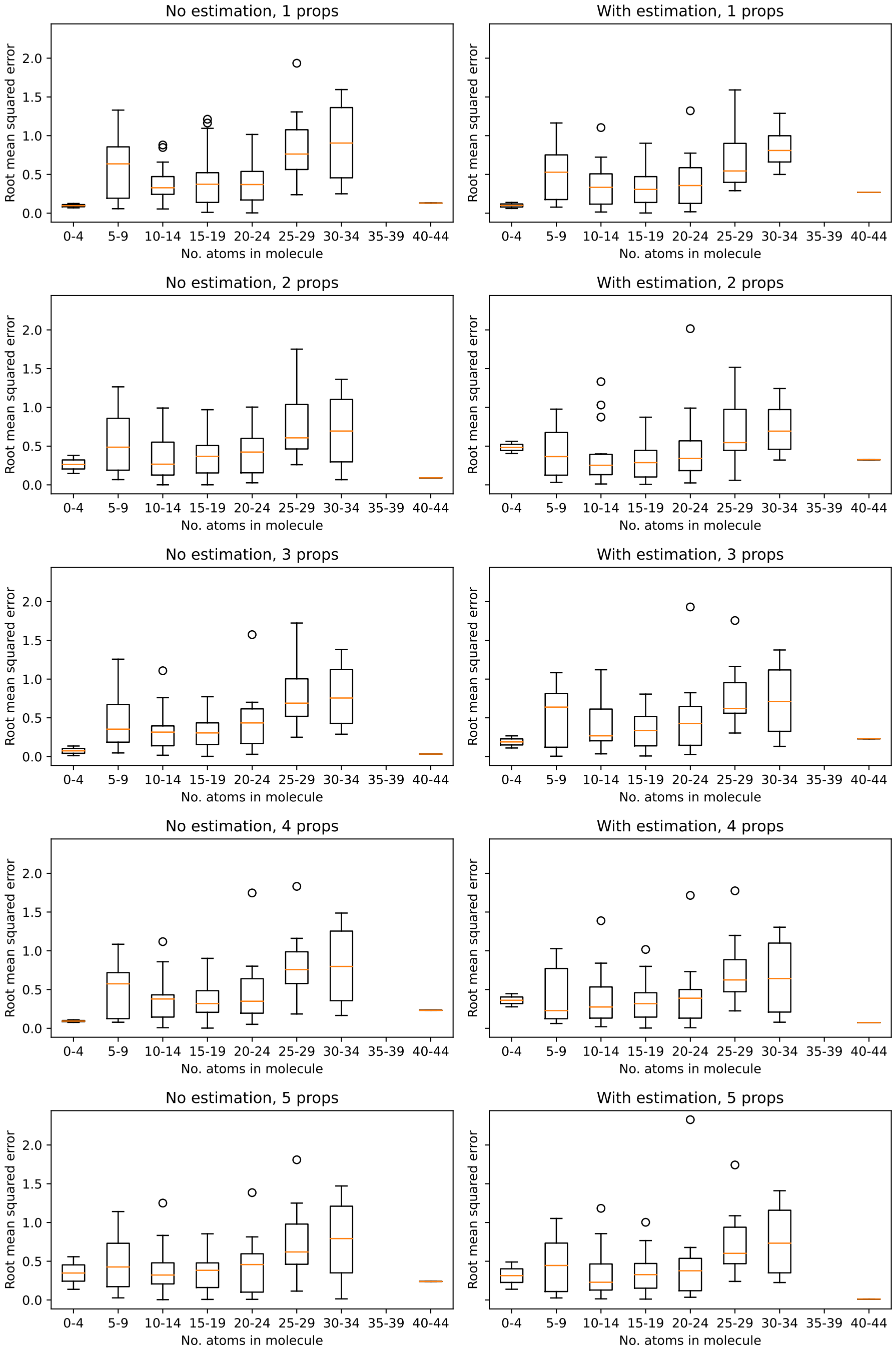


Figure 5.5: The root mean squared error of the prediction was aggregated into 9 bins in the BBBP test set and plotted statistics of the prediction errors of each bin as a boxplot. The plots in the left column visualises the predictions from models without complement estimation; the right column visualises the predictions from models using complement estimation. Each row represents a number of propagation steps, going from the top, 1 propagation step, to the bottom, 5 propagation steps.

see that the major difference is that the model without complement estimation has a much narrower range of prediction values on the very few, very large molecules.

ESOL. The differences between the distributions of predictions in Fig. 5.5 are minimal. The majority of molecules are between 15 and 25 atoms large (Fig. 5.4); in these bins of the boxplot, there differences are not easily discernable. Comparing the propagation step at which a model with complement estimation worked best (1 propagation step) with the best model without complement estimation (3 propagation steps), we can see that for the small number of larger molecules between 25 and 34 atoms large that the interquartile range of errors is smaller in the model with complement estimation than that without. We believe that the graph complements are distributing information further across larger graphs at earlier propagation steps owing to the global propagation structured by the complement graph. Perhaps one sees worse prediction errors in these same regions later on owing to graph over-smoothing induced by multiple convolutions (see Section 2.5.5). This is further evidence in respect of **Question 1** that the graph complements are lowering the necessary number of propagations.

5.5.2 Structure Analysis

In this section, in order to answer **Question 2**, we consider the effect of structure on the model performance on the two datasets separately before drawing general conclusions. For both datasets, for each estimated graph and linegraph complements on each test set sample and for each propagation step, we computed the proportion of non-zero entries, the minima, maxima, mean and standard deviation and present average of those values across all test samples in Tables 5.3 and 5.4 for BBBP and Tables 5.5 and 5.6 for ESOL.

We can see that on both datasets the estimated complements are highly dense in comparison to the original graph in the left-most column. The graph and linegraph complements have much larger maximum values—which unsurprising given that the original graphs have binary weight matrices. On both datasets the weights of the complement graphs are largely positive; the graph and linegraph complements on the ESOL dataset wholly positive and almost totally positive respectively. The maximum and mean graph weights generally tend to decrease with the increase in propagation steps, with the exception of the graph complement learned for the BBBP dataset. The larger weights with increasing propagation steps does not produce a lower prediction error on the BBBP dataset, however Table 5.1. The same effect is seen in the standard deviation of the graph weights, which decrease for all estimated graph and linegraph complement except the graph on BBBP. From these observations it appears that the graph

Table 5.3: The average density of the estimated linegraph complements on the BBBP test set, the average number of positive entries, the average minimum, maximum, mean and standard deviation are reported in this table. The first column gives the corresponding values of the original graph. The density is computed by dividing the number of non-zero positions in the adjacency matrix by the total number of positions. We can see in this table that combined the graph and graph complement are 100% dense.

Statistic	BBBP	No. propagation steps				
	graph	1	2	3	4	5
density	0.098	0.902	0.902	0.902	0.902	0.902
positive	0.098	0.740	0.648	0.902	0.599	0.652
min.	0.000	-0.211	-0.329	-0.011	-0.635	-0.455
max.	1.000	4.307	8.131	9.116	10.701	16.150
μ	0.098	1.071	1.377	3.559	1.695	3.599
σ	0.248	1.073	1.770	1.797	2.288	3.741

Table 5.4: The average density of the estimated linegraph complements on the BBBP test set, the average number of positive entries, the average minimum, maximum, mean and standard deviation are reported in this table. The first column gives the corresponding values of the original linegraph. The density is computed by dividing the number of non-zero positions in the adjacency matrix by the total number of positions. We can see in this table that combined the linegraph and linegraph complement are 100% dense.

Statistic	BBBP	No. propagation steps				
	linegraph	1	2	3	4	5
density	0.003	0.949	0.949	0.949	0.949	0.949
positive	0.003	0.865	0.865	0.949	0.899	0.516
min.	0.000	-0.157	-0.154	0.000	-0.060	-0.190
max.	1.000	9.336	8.945	7.260	4.853	4.918
μ	0.051	3.226	2.791	3.654	1.684	0.449
σ	0.182	2.040	1.861	1.498	1.149	0.905

complement learned for the BBBP is vulnerable to learning very large weights. Given the large graph weights, the contributions from neighbours on the graph complement are consequently weighted highly. A subsequent question is to what extent the learning on the BBBP dataset was hindered by these very large weights in the graph complement; the average graph weight remains proportionally smaller and indeed decreases. On the face of it, performance does not appear to have been hit; at every propagation step the models with complement estimation still outperform the models without complement estimation.

Table 5.5: The average density of the estimated linegraph complements on the ESOL test set, the average number of positive entries, the average minimum, maximum, mean and standard deviation are reported in this table. The first column gives the corresponding values of the original graph. The density is computed by dividing the number of non-zero positions in the adjacency matrix by the total number of positions. We can see in this table that combined the graph and graph complement are 100% dense.

Statistic	ESOL	No. propagation steps				
	graph	1	2	3	4	5
density	0.136	0.864	0.864	0.864	0.864	0.864
positive	0.098	0.864	0.864	0.864	0.864	0.864
min.	0.000	0.000	0.000	0.000	0.000	0.000
max.	1.000	4.284	2.158	3.622	1.917	2.732
μ	0.136	2.355	1.466	1.657	0.897	1.494
σ	0.305	1.056	0.586	0.878	0.471	0.717

Table 5.6: The average density of the estimated linegraph complements on the ESOL test set, the average number of positive entries, the average minimum, maximum, mean and standard deviation are reported in this table. The first column gives the corresponding values of the original linegraph. The density is computed by dividing the number of non-zero positions in the adjacency matrix by the total number of positions. We can see in this table that combined the linegraph and linegraph complement are 100% dense.

Statistic	ESOL	No. propagation steps				
	linegraph	1	2	3	4	5
density	0.008	0.930	0.930	0.930	0.930	0.930
positive	0.008	0.928	0.930	0.887	0.930	0.930
min.	0.000	−0.002	0.000	−0.038	0.000	0.000
max.	1.000	1.326	0.226	2.288	0.424	0.077
μ	0.070	0.828	0.134	1.361	0.267	0.052
σ	0.225	0.467	0.078	0.689	0.135	0.020

We can see that there is some interplay between the structure of the estimated graphs and the spread of graph weights in the estimated complements. For example, on the ESOL dataset, the worst result is attained at 3 and 5 propagation steps. At both points, the maximum, mean and standard deviation of the graph weights are high for both the graph and linegraph complements. On the BBBP dataset, for the worst result at 5 propagation steps, the maximum, average and standard deviation of graph complement are at their highest; the statistics on the linegraph complement at 5 propagation steps are however not atypical. This suggests that the prediction errors are being unduly influenced by the graph complement, or at least that learning is able to be led astray by the algorithm's focus on the graph complement to the detriment of the linegraph complement.

In answer to **Question 2** we can only draw mixed conclusions. In most cases, the additional propagation steps lower the magnitude of the graph weights. We suspect that the model is undergoing the process of graph over-smoothing: as the features become less distinct, it becomes more difficult to discern the intramolecular structures in learning the graph and linegraph complements. It is difficult to apply the same explanation to the graph complements learned on the BBBP dataset, however. Perhaps we are witnessing an obverse case here where uninformative data across the graph leads the model to overemphasise single sources of information. That would explain why there is such a large maximum graph weight in the graph complements estimated for BBBP but a proportionally much smaller average graph weight. In either case, the estimated graphs become less informative and the performance of the models worsens; in both cases we suspect that the cause is increasingly indistinct data that gives rise to different outcomes. Indeed a similar effect appears to affect the results at increasing propagation steps on the models without graph estimation, where the prediction errors do not improve with more propagation steps.

The large graph weights indicate that the weights of the graphs need to be controlled somehow. In this experiment we removed all regularisation and loosened the constraint that the graph weights should be negative. That the graph weights are negative does not appear to be an issue, but certainly large graph weights appear to be causing problems. The average and standard deviation of the graph weights provides insight into the the relationship between the graph weights and the prediction errors. Additionally, we are learning two graphs simultaneously, and it appears that learning the graph and linegraph complements simultaneously can lead learning astray as in the case of the BBBP dataset. Additional controls that balance the learning objectives on the two graphs are necessary. For example, one could somehow dually constrain the magnitudes of the complements so that they never grow in excess of one another. This might be realised as a regularisation term.

5.6 Discussion

The best result on both datasets was attained on both datasets by a model using complement estimation, but in both cases it was a model that uses a single propagation step that attains the best result. This is interesting for two reasons. Firstly it suggests that the graph and linegraph complements are essentially mapping vertex and edge features globally to one another after the first propagation step with the normal graph and linegraph. It is inaccurate to say that the best performance was attained after one propagation step on the model with complement estimation. After all, a single propagation step in the model without complement estimation is really one propagation across the graph, whereas there is an additional step in the models with complement estimation that propagates signals across the estimated graph and linegraph complements. Indeed it is reasonable to compare the result of one propagation step on the models with complement estimation to the result of two propagation steps on the model without estimation. Yet notice that the second propagation step of the models with complement estimation do not thereby acquire a similar performance to the models with complement estimation using a single "propagation step". The analysis can be continued further by comparing the result of *two* propagation steps on the models with complement estimation to the result of *four* propagation steps on the model without estimation. Again, from this perspective the performance is still better on the models with graph estimation. This is evidence that the model is estimating complement graphs with useful structural information for interpreting molecular properties.

Secondly the models with complement estimation are attaining better predictions at lower propagation steps, which gives us an interesting answer to **Question 2**. As we have said, on both datasets the models with complement estimation attain the best results at 1 propagation step. While the best results without complement estimation are attained at epochs 1 and 2 on the BBBP dataset, the best result on the ESOL dataset without complement estimation is attained at 4 propagation steps. This result suggests that the global propagation of vertex and edge features across the estimated graph and linegraph complements respectively is partially removing the need to propagate vertex and edge signals across the graph. If the motivation for multiple propagation steps is to expand the receptive field on the graph (see Section 2.5.4), the global propagation removes the need to do so over multiple steps since the signals are theoretically propagated over the whole graph. Additionally the directed graph attention allows the model to select the contribution of local and global vertices more flexibly, which further broadens the interpretative capacity of the model. This global propagation of signals may not be desirable in all problem domains, but it appears to be helpful on small graphs like molecules. We speculate that it is most appropriate domains where graph-wise tasks are performed, such as here, where the model is ultimately learning a single representation for the molecule via a readout phase (see Section 2.5.4).

We can see from the analysis in Section 5.5.1 that the models using complement estimation tend to work better on large molecules, therefore larger graphs, than models without complement estimation. We believe in both cases that this is owed to the use of the global propagation structured by the graph and linegraph complements, which are able to propagate along supplementary structures represented by the graph and linegraph complements. There are however issues of large graph weights disrupting learning and the estimation and use of one graph complement dominating learning of another. An additional problem arises in the use of multiple propagations. A possible cause is graph over-smoothing (Section 2.5.5) which has two possible, related sources here. Firstly the estimated complements may simply be too dense. It has been observed that sparsifying the graph by sampling edges is one way to alleviate graph over-smoothing. Secondly the graph diameter is very small, which, in comparison to other graphs, could be reducing the number of propagation steps before which graph over-smoothing sets in and disrupts, learning. Yet propagation appears to affect models both with and without complement estimation, which suggests that there is a problem with structure of our proposed architecture beyond the estimation of graph complements.

5.7 Summary

In this chapter we presented two novel graphs, which we termed the *graph* and *linegraph complements*, to represent the intramolecular interactions of a molecule. We presented an architecture to learn the structure of the graph complements, which we call *complement estimation*. The approach allows the model to factorise the graph representation learning into structure represented by chemical bonds and its complementary structure, both on the structure of the vertices/atoms and the edges/bonds. The graph and linegraph of a molecule are used to structure the propagation of vertex and edge features, after which the graph and linegraph complements are used to propagate information across the complementary structures. In our experiments we find that complement estimation reduces prediction error. We conclude that the complementary structures learned in complement estimation are learning intramolecular interactions that inform the molecular representation and thereby improve performance on the given task.

This work is limited in a few ways, however. Firstly the analysis we conducted is limited to validating the use of complement graphs. Further experiments could consider the use of random graphs, for example, in order to evaluate whether the model is learning structure that is optimised for the input data, or if any complementary structure will improve the performance of the model; this latter point would be interesting to explore in cases such as this chapter where a graph-based attention mechanism is used, which can select for information in the end-to-end training. We have additionally not examined the effect of different starting-points for graph estimation, an issue that we mentioned in

Section 2.5.7.1. The use of multiple attention heads in generating the complement graphs is a good starting point, since it is a combination of several different starting-points.

Secondly the use of linegraphs and their complements limits this approach to very small graphs. The size of linegraphs is prohibitive in combination with small graphs. The same issue occurred in Chapter 3, where we used a linegraph to structure a datacentre. The linegraph is however an effective way to represent higher-order structures and has already been used directly or indirectly in other papers, as we mentioned in Section 2.6.3. But applying the technique herein to proteins would be very expensive, incurring a significant overhead and memory cost. One way forward to use complement graphs might be the use of heterogeneous graphs, for example, which is able to model higher-order interactions of groups of atoms (Shui and Karypis, 2020).

Thirdly it is possible that the global and local attention mechanisms could be simplified. For example, rather use the attention coefficients to directly compute the adjacency matrix, they could be incorporated into later convolutions. An exploration of the more effective use of attention coefficients, specifically the interaction between global and local attention coefficients in this chapter and in Chapter 4, would be an interesting future direction of practical research.

Fourthly, the estimated graph and linegraph complements are very sparse here. There are two key issues, as were discussed in Section 5.5.2: (1) the graph complements as estimated in the experiments presented in this chapter tend to have much larger weights than the original graphs that their structure is derived from; and (2) the graphs are completely dense. There are many ways to incorporate a regularisation term in the loss that encourages the algorithm to learn smaller weights. We experimented with some techniques in Chapter 4. Investigations of further techniques of regularisation or the explicit model architecture that indirectly produces sparser graphs is a fertile ground for research. In place of the attention vector, for example, some function might be applied that implicitly enforces sparsity—some kernel such as a the Gaussian function, for example.

Extensions of this work could include looking at ways of constraining the vertex and edge embeddings. For example, the set of bond embeddings for a given vertex pair could be constrained to be close to one another. That way the embeddings *can* differ, but they are encouraged to be identical. The model might therefore be inclined to pay more attention to the edge attributes governing the type of each atom/bond in the graph. Alternatively additional embeddings of higher order could be supplied to the model, as in the work of Gasteiger, Groß, and Günnemann (2020) and Gasteiger et al. (2020), such as bonding angles, which could be represented as a higher-order linegraph.

It is difficult to know how applicable the techniques developed in this chapter can be extended to other molecular datasets, too, since we only consider BBBP and ESOL. Notably it is reportedly particularly hard to create pretrained networks with attention mechanisms (Xia et al., 2022). Additionally, there is an open question in the extent of the applicability of the learned graph complements to other molecular problems and other domains.

Lastly, the datasets here are in equilibrium. We have not considered the challenge of learning molecules out of equilibrium and therefore have not considered how the graph estimation approach herein could be adapted to out-of-equilibrium graphs (Schütt et al., 2017). In such a case as this technique could not be applied, a statistical method might be more suitable.

Chapter 6

Conclusions and Future Work

Contents

6.1 Conclusions

The work presented in this thesis addresses three challenges in graph deep learning: learning edge representations on graphs, convolution on directed graphs and the estimation of graph structure from input data. In Chapter 3 we developed a method to learn on the edge structure of a graph with the use of a directed linegraph. The construction of the directed graph presented a definitional difficulty in light of the application, namely the presence of inverse edges. This difficulty motivated the definition of an isotropic kernel for directed graphs, where the inverse edge was factored out of construction as a separate term. The experiments revealed that in application to datacentres, the inverse edge contributes information to a convolution that is deleterious when mixed with neighbouring signals.

In Chapter 4 we addressed the challenge of estimating graph structure from cyclical information. We used two combinations of the input to estimate two structures, the long-term, static structure of the graph and the short-term, dynamic structure of the graph from traffic data. The two graphs complement one another in estimating the structure of the input at a given point in time in order to predict the subsequent development of traffic. Moreover, the graphs that were estimated were directed. We defined an anisotropic convolution operation on directed relations that factorises the input data into its two streams with respect to each vertex in the convolution.

Lastly, in Chapter 5, we considered the challenge of learning molecular representations to predict molecular properties. We proposed a new graph, the graph complement, to learn the structure of a molecule beyond what is specified in the predefined graph. In turn the graph complements, constructed on a directed graph and a directed linegraph, were themselves directed, and allowed us to structure the propagation of information across the graph by factorising the relations into chemical bonds and other intramolecular relations.

6.2 Contributions

The contributions of this thesis are the following:

A technique for structuring learning on graph edges. We present two methods for learning on the edge structure or second-order structure of graphs representing domains. The first method uses the directed linegraph to define the second-order structure of a datacentre. We consider in particular the special case of inverse edges that arise from the construction of directed linegraphs on directed graphs and present a method to learn on inverse edges. The second method uses a directed linegraph for the propagation of edge representations, which are further propagated by the linegraph complement, a graph that, as far as we know, has not been presented in the literature.

An isotropic method for convolution on directed graphs. We present an isotropic kernel for directed graphs that factors neighbouring signals into two groups according to their incidence to a focal vertex. It is a simple construction with only a few parameters per output channel. The method was evaluated on a datacentre simulation, whereby we found that the inclusion of inverse edges as a separate term or as part of the neighbourhood worsened the performance, thereby justifying our decision to separate out inverse edges as a term altogether.

A technique for estimating graph structure from temporal, cyclical data. We present an attention-based approach to graph estimation that uses two compositions of cyclical data, which we call *static–dynamic fusion*, to make two complementary estimations of the graph structure inhered in the input.

An anisotropic, attention-based convolution for directed graphs. We present an an anisotropic, attention-based convolution for directed graphs that separates neighbourhoods into their incidence to focal vertices.

A new graph to supplement a predefined structure. We present an attention-based approach to graph estimation that learns the higher-order interactions on a molecular graph. The molecular graphs defines solely the chemical bonds; the complementary structure, which we call a *graph complement*, can learn the structure of the intramolecular relations in the molecule. The downstream learning is further factorised into the different structures to allow the model to separately learn the two different structures.

6.3 Future Work

Learning with edge features and learning on the edge structures themselves is an approach primarily motivated by the given domain. More formal, principled considerations of their inclusion has, to the best of knowledge, not yet come to the fore. The majority of general definitions of deep learning approaches are restricted to vertex information that is propagated across vertices, although there are clearly various ways in which to use edge features, which can be incorporated into vertex-wise convolution or used to index functions. It is an exciting area of research where a greater degree of formalisation, even within application domains such as molecules, would yield some very interesting results in revealing the various possible routes of explanation in the theory. The results of the various previous models and our own models have shown the value of edge representations in successful model learning.

The learning of edge representations with respect to directed graphs, and directed linegraphs as a complimentary structure, is in its infancy in particular. One future development of linegraphs could be the use of a hierarchical set of linegraphs of increasing order to learn multi-order structures on top of a graph, which would allow the inclusion of additional, higher-order features in a convolution. Techniques to incorporate the learned features of multiple orders into a single model are worth further exploration, however. A significant obstacle to the use of linegraphs in general is the difficulty of the ever-growing order and size of successive orders of linegraphs constructed on one another. Sparse implementations have their place, but also their limit; it is not clear how drastically the density of the graph would grow with increasing orders, nor the laws governing the growth. Methods to limit the growth or prune edges would be an interesting future direction for work. In particular, it would be interesting to see how the linegraph might be applied to incorporating third-order features, such as bond angles, in learning.

The further development of graph estimation is a rich seam of research that is presently in its infancy. Graph estimation is a natural extension to structure of the foundational hypothesis that models learn their own representations. The methods presented in this thesis, combined with works elsewhere, demonstrate the feasibility of graph estimation. Implementations of graph estimation approaches however face various problems of computational complexity, memory load and the highly complex nature of estimating especially large graphs. The shear size of the hypothesis space of graphs is unfathomably large. It is clear that graph estimation requires some kind of restriction in searching the space of possible structures. What has recurred in our studies, for example, is the density of the estimated graphs, which hamper any sparse implementations of the algorithms that would speed up training. Though it is not clear how sparsity can be enforced in estimated graph structures without losing the remarkable degree of redundancy that we discovered in our estimated graphs. It is also unclear how the structure of the model affects the path of graph estimation. The number of propagation steps in the message-passing phase of a graph-based convolutional neural network will learn different representations, which is turn effects the performance and therefore the structure of the graph. Future work would therefore focus on its refinement, optimisation, the elaboration of differing strategies to learn different kinds of structure in data while remaining sufficiently redundant and sparse, and the effect of model architecture on the graph structures that are learned.

Overall graph deep learning faces considerable theoretical and practical challenges. These challenges are not insuperable and the results presented hitherto indicate a promising direction for modelling. A key advantage of graphs is their unrestrictive spatial definition, which permits the definition of a broadly applicable set of techniques.

Bibliography

Aigner, Martin (1967). “On the linegraph of a directed graph”. In: *Mathematische Zeitschrift* 102.1, pp. 56–61. ISSN: 00255874. DOI: 10.1007/BF01110285.

Alet, Ferran et al. (2019). “National Relational Inference with Fast Modular Meta-learning”. In: *Conference on Neural Information Processing Systems*.

Andreoletti, Davide et al. (2019). “Network Traffic Prediction based on Diffusion Convolutional Recurrent Neural Networks”. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 246–251. DOI: 10.1109/INFCOMW.2019.8845132.

Arzani, Behnaz (2018). *Simulation source codes*. Tech. rep. Microsoft Research.

Arzani, Behnaz et al. (2018). “007: Democratically Finding the Cause of Packet Drops”. In: *USENIX Symposium on Networked Systems Design and Implementation*, pp. 419–435.

Atz, Kenneth, Francesca Grisoni, and Gisbert Schneider (2021). “Geometric Deep Learning on Molecular Representations”. In: *Nature Machine Intelligence* 3, pp. 1023–1032. DOI: 10.1038/s42256-021-00418-8.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). *Layer Normalization*. DOI: 10.48550/arXiv.1607.06450.

Battaglia, Peter W. et al. (2016). “Interaction Networks for Learning about Objects, Relations and Physics”. In: *Neural Information Processing Systems*, pp. 4502–4510.

Battaglia, Peter W. et al. (2018). “Relational inductive biases, deep learning, and graph networks”. In: *ArXiv*.

Bengio, Yoshua and Yann LeCun (2007). “Scaling Learning Algorithms towards AI”. In: *Large-Scale Kernel Machines*. Ed. by Léon Bottou et al. MIT Press.

Bishop, Christopher M. (2006). *Pattern recognition and machine learning*. Information science and statistics. Springer.

Bollobás, Béla (1979). *Graph Theory: An Introductory Course*. Ed. by J. H. Ewing, F. W. Gehring, and P. R. Halmos. Springer-Verlag.

Bracewell, Ronald Newbold (2000). *The Fourier Transform and its Applications*. McGraw-Hill Higher Education.

Bronstein, Michael M. et al. (2017). “Geometric Deep Learning: Going beyond Euclidean data”. In: *IEEE Signal Processing Magazine*. ISSN: 10535888. DOI: 10.1109/MSP.2017.2693418.

Brooks, Rodney A. (1991). “Intelligence without reason”. In: *International Joint Conference on Artificial Intelligence*, pp. 569–595.

Bruna, Joan et al. (2014). “Spectral Networks and Locally Connected Networks on Graphs”. In: *International Conference on Learning Representations*, pp. 1–14.

Cai, Chen and Yusu Wang (2020). “A Note on Over-Smoothing for Graph Neural Networks”. In: *International Conference on Machine Learning*, pp. 1–13.

Cai, Ling et al. (2020). “Traffic transformer: Capturing the continuity and periodicity of time series for traffic forecasting”. In: *Transactions in GIS* 24.3, pp. 736–755. ISSN: 1361-1682. DOI: 10.1111/tgis.12644.

Chen, Chi et al. (2019). “Graph Networks as a Universal Machine Learning Framework for Molecules and Crystals”. In: *Chemistry of Materials* 31.9, pp. 3564–3572. ISSN: 0897-4756, 1520-5002. DOI: 10.1021/acs.chemmater.9b01294.

Chen, Hongming et al. (2018). “The rise of deep learning in drug discovery”. In: *Drug Discovery Today* 23.6, pp. 1241–1250. ISSN: 1359-6446. DOI: 10.1016/j.drudis.2018.01.039.

Chen, Liujuan et al. (2020a). “GDCRN: Global Diffusion Convolutional Residual Network for Traffic Flow Prediction”. In: *Knowledge Science, Engineering and Management*. Ed. by Gang Li et al. Lecture Notes in Computer Science. Springer International Publishing, pp. 438–449. DOI: 10.1007/978-3-030-55393-7_39.

Chen, Ming et al. (2020b). “Simple and Deep Graph Convolutional Networks”. In: *International Conference on Machine Learning*. PMLR, pp. 1725–1735.

Chen, Zhengdao, Joan Bruna, and Lisha Li (2019). “Supervised community detection with line graph neural networks”. In: *International Conference on Learning Representations*, pp. 1–23.

Chircu, Alina et al. (2019). “Visualization and Machine Learning for Data Center Management”. In: *INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik – Informatik für Gesellschaft (Workshop-Beiträge)*. Ed. by Claude Draude, Martin Lange, and Bernhard Sick. Gesellschaft für Informatik e.V., pp. 23–35. DOI: 10.18420/inf2019_ws02.

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2016). “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”. In: *Proceedings of the International Conference on Learning Representations*. arXiv. DOI: 10.48550/arXiv.1511.07289.

Clos, Charles (1953). "A study of non-blocking switching networks". In: *The Bell System Technical Journal* 32.2, pp. 406–424. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1953.tb01433.x.

Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems* 2.4, pp. 303–314. ISSN: 0932-4194, 1435-568X. DOI: 10.1007/BF02551274.

Danel, Tomasz et al. (2020). "Spatial Graph Convolutional Networks". In: *Neural Information Processing*. Ed. by Haiqin Yang et al. Communications in Computer and Information Science. Springer International Publishing, pp. 668–675. DOI: 10.1007/978-3-030-63823-8_76.

Defferrard, Michaël, Xavier Bresson, and Pierre Vandergheynst (2016). "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: *Advances in Neural Information Processing Systems*, pp. 3844–3852.

Deng, Leyan et al. (2022). "Graph Convolutional Adversarial Networks for Spatiotemporal Anomaly Detection". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.6, pp. 2416–2428. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2021.3136171.

Di Massa, V. et al. (2006). "A Comparison between Recursive Neural Networks and Graph Neural Networks". In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pp. 778–785. DOI: 10.1109/IJCNN.2006.246763.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12, pp. 2121–2159.

Duvenaud, David et al. (2015). "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: *Advances in Neural Information Processing Systems*, pp. 2224–2232.

Edwards, Allen L. (1948). "Note on the "correction for continuity" in testing the significance of the difference between correlated proportions". In: *Psychometrika* 13.3, pp. 185–187. ISSN: 0033-3123. DOI: 10.1007/BF02289261.

Edwards, Michael and Xianghua Xie (2016). "Graph Based Convolutional Neural Network". In: *Proceedings of the British Machine Vision Conference*. arXiv.

— (2017). "Graph-Based CNN for Human Action Recognition from 3D Pose". In: *Deep Learning in Irregular Domains Workshop*.

Faber, Felix A. et al. (2017). "Prediction Errors of Molecular Machine Learning Models Lower than Hybrid DFT Error". In: *Journal of Chemical Theory and Computation* 13.11, pp. 5255–5264. ISSN: 1549-9618. DOI: 10.1021/acs.jctc.7b00577.

Fang, Luoyang et al. (2018). "Mobile Demand Forecasting via Deep Graph-Sequence Spatiotemporal Modeling in Cellular Networks". In: *IEEE Internet of Things Journal* 5.4, pp. 3091–3101. ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2832071.

Feng, Jie et al. (2018). "DeepTP: An End-to-End Neural Network for Mobile Cellular Traffic Prediction". In: *IEEE Network* 32.6, pp. 108–115. ISSN: 1558-156X. DOI: 10.1109/MNET.2018.1800127.

Fisher, R. A. (1936). "The Use of Multiple Measurements in Taxonomic Problems". In: *Annals of Eugenics* 7.2, pp. 179–188. ISSN: 20501420. DOI: 10.1111/j.1469-1809.1936.tb02137.x.

Franceschi, Luca et al. (2019). "Learning Discrete Structures for Graph Neural Networks". In: *International Conference on Machine Learning*.

Gasteiger, Johannes, Janek Groß, and Stephan Günnemann (2020). "Directional Message Passing for Molecular Graphs". In: *Proceedings of the International Conference on Learning Representations*.

Gasteiger, Johannes et al. (2020). "Fast and Uncertainty-Aware Directional Message Passing for Non-Equilibrium Molecules". In: *Machine Learning for Molecules Workshop*.

Georgousis, Stavros, Michael P. Kenning, and Xianghua Xie (2021). "Graph Deep Learning: State of the Art and Challenges". In: *IEEE Access* 9, pp. 22106–22140. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3055280.

Gill, Phillipa, Navendu Jain, and Nachiappan Nagappan (2011). "Understanding network failures in data centers: measurement, analysis, and implications". In: *ACM SIGCOMM Computer Communication Review* 41.4, pp. 350–361. ISSN: 0146-4833. DOI: 10.1145/2043164.2018477.

Gilmer, Justin et al. (2017). "Neural Message Passing for Quantum Chemistry". In: *International Conference on Machine Learning*. Vol. 3, pp. 1263–1272. DOI: 10.5555/3305381.3305512.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.

Goodfellow, Ian et al. (2020). "Generative adversarial networks". In: *Communications of the ACM* 63.11, pp. 139–144. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3422622.

Gori, M., G. Monfardini, and F. Scarselli (2005). "A new model for learning in graph domains". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

Grattarola, Daniele and Cesare Alippi (2020). "Graph Neural Networks in TensorFlow and Keras with Spektral". In: *ArXiv*.

Guo, Chuanxiong et al. (2015). "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis". In: *ACM SIGCOMM Computer Communication Review* 45.4, pp. 139–152. ISSN: 0146-4833. DOI: 10.1145/2829988.2787496.

Guo, Kan et al. (2022). “Dynamic Graph Convolution Network for Traffic Forecasting Based on Latent Network of Laplace Matrix Estimation”. In: *IEEE Transactions on Intelligent Transportation Systems* 23.2, pp. 1009–1018. issn: 1558-0016. doi: 10.1109/TITS.2020.3019497.

Guo, Shengnan et al. (2019). “Attention Based Spatial-Temporal Graph Convolutional Networks for Traffic Flow Forecasting”. In: *AAAI Conference on Artificial Intelligence*. Vol. 33, pp. 922–929. doi: 10.1609/aaai.v33i01.3301922.

Gálvez, Juan J. and Pedro M. Ruiz (2013). “Efficient rate allocation, routing and channel assignment in wireless mesh networks supporting dynamic traffic flows”. In: *Ad Hoc Networks* 11.6, pp. 1765–1781. issn: 1570-8705. doi: 10.1016/j.adhoc.2013.04.002.

Hamilton, William L, Rex Ying, and Jure Leskovec (2017a). “Representation Learning on Graphs: Methods and Applications”. In: *IEEE Data Engineering Bulletin* 40.3, pp. 52–74.

Hamilton, William L., Zhitao Ying, and Jure Leskovec (2017b). “Inductive Representation Learning on Large Graphs”. In: *Advances in Neural Information Processing Systems*. Ed. by I Guyon et al. Curran Associates, Inc., pp. 1024–1034.

Hammond, David K, Pierre Vandergheynst, and Rémi Gribonval (2011). “Wavelets on graphs via spectral graph theory”. In: *Applied and Computational Harmonic Analysis* 30.2, pp. 129–150. issn: 10635203. doi: 10.1016/j.acha.2010.04.005.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York. doi: 10.1007/978-0-387-84858-7.

He, Kaiming et al. (2016). “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.

Henaff, Mikael, Joan Bruna, and Yann LeCun (2015). “Deep Convolutional Networks on Graph-Structured Data”. In: *ArXiv*.

Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh (2006). “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural Computation* 18.7, pp. 1527–1554. issn: 0899-7667, 1530-888X. doi: 10.1162/neco.2006.18.7.1527.

Hinton, Geoffrey (2012). *Neural Networks for Machine Learning*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural Computation* 9.8, pp. 1735–1780. issn: 0899-7667. doi: 10.1162/neco.1997.9.8.1735.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5, pp. 359–366. issn: 08936080. doi: 10.1016/0893-6080(89)90020-8.

Hoshen, Yedid (2017). “VAIN: Attentional Multi-agent Predictive Modeling”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc.

Howe, Jim (2007). *History of Artificial Intelligence at Edinburgh: A Perspective*.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning*. PMLR, pp. 448–456.

Jagadish, H. V. et al. (2014). "Big data and its technical challenges". In: *Communications of the ACM* 57.7, pp. 86–94. ISSN: 0001-0782. DOI: 10.1145/2611567.

Jang, Soobeom, Seong Eun Moon, and Jong Seok Lee (2018). "EEG-Based Video Identification Using Graph Signal Modeling and Graph Convolutional Neural Network". In: *International Conference on Acoustics, Speech and Signal Processing*. DOI: 10.1109/ICASSP.2018.8462207.

Ji, Weiliang et al. (2018). "A CNN-based network failure prediction method with logs". In: *Chinese Control And Decision Conference*. IEEE, pp. 4087–4090. DOI: 10.1109/CCDC.2018.8407833.

Jin, Wengong, Regina Barzilay, and Tommi Jaakkola. (2018). "Junction Tree Variational Autoencoder for Molecular Graph Generation". In: *International Conference on Machine Learning*.

Johnson, Daniel D. (2017). "Learning Graphical State Transitions". In: *International Conference on Learning Representations*, pp. 1–19. DOI: https://openreview.net/forum?id=HJ0NvFzxl.

Kearnes, Steven et al. (2016). "Molecular graph convolutions: moving beyond fingerprints". In: *Journal of Computer-Aided Molecular Design*. ISSN: 15734951. DOI: 10.1007/s10822-016-9938-8.

Kenning, Michael and Xianghua Xie (2023). "Attention-based Graph Estimation and Directed Convolution for Prediction of Traffic Conditions". In: *Proceedings of the 10th Internatinal Workshop on Deep Learning on Graphs*.

Kenning, Michael et al. (2021). "Locating Datacenter Link Faults with a Directed Graph Convolutional Neural Network". In: *Proceedings of the International Conference on Pattern Recognition Applications and Methods*. SCITEPRESS - Science and Technology Publications, pp. 312–320. DOI: 10.5220/0010301403120320.

Kenning, Michael P. (2019). "Generative Modelling in Non-Euclidian Domains". MA thesis. Swansea University.

Kenning, Michael P. et al. (2022). "A directed graph convolutional neural network for edge-structured signals in link-fault detection". In: *Pattern Recognition Letters* 153, pp. 100–106. ISSN: 01678655. DOI: 10.1016/j.patrec.2021.12.003.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *Proceedings of the International Conference for Learning Representations*.

Kipf, Thomas et al. (2018). “Neural relational inference for Interacting systems”. In: *International Conference on Machine Learning*.

Kipf, Thomas N. and Max Welling (2017). “Semi-Supervised Classification with Graph Convolutional Networks”. In: *International Conference on Learning Representations*, pp. 1–14.

Kong, Xiangyuan et al. (2020). “STGAT: Spatial-Temporal Graph Attention Networks for Traffic Flow Forecasting”. In: *IEEE Access* 8, pp. 134363–134372. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3011186.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2017). “ImageNet classification with deep convolutional neural networks”. In: *Communications of the ACM*. ISSN: 15577317. DOI: 10.1145/3065386.

Krzakala, Florent et al. (2013). “Spectral redemption in clustering sparse networks”. In: *National Academy of Sciences of the United States of America* 110.52, pp. 20935–20940. ISSN: 00278424. DOI: 10.1073/pnas.1312486110.

Kuo, C.-C. Jay (2016). “Understanding convolutional neural networks with a mathematical model”. In: *Journal of Visual Communication and Image Representation*. ISSN: 10959076. DOI: 10.1016/j.jvcir.2016.11.003.

Kuo, C.-C. Jay and Yueru Chen (2018). “On data-driven Saak transform”. In: *Journal of Visual Communication and Image Representation*. ISSN: 10959076. DOI: 10.1016/j.jvcir.2017.11.023.

Kuo, C.-C. Jay et al. (2019). “Interpretable convolutional neural networks via feedforward design”. In: *Journal of Visual Communication and Image Representation*. ISSN: 10959076. DOI: 10.1016/j.jvcir.2019.03.010.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). “Deep learning”. In: *Nature* 521.7553, pp. 436–444. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539.

LeCun, Yann et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2323. ISSN: 00189219. DOI: 10.1109/5.726791.

Leiserson, Charles E. (1985). “Fat-trees: Universal networks for hardware-efficient supercomputing”. In: *IEEE Transactions on Computers* C-34.10, pp. 892–901. ISSN: 1557-9956. DOI: 10.1109/TC.1985.6312192.

Levie, Ron et al. (2019). “CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters”. In: *IEEE Transactions on Signal Processing*. ISSN: 1053587X. DOI: 10.1109/TSP.2018.2879624.

Li, Chensheng et al. (2020a). “Scalable Graph Convolutional Networks With Fast Localized Spectral Filter for Directed Graphs”. In: *IEEE Access* 8, pp. 105634–105644. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2999520.

Li, Guohao et al. (2019a). "DeepGCNs: Can GCNs Go As Deep As CNNs?" In: *IEEE/CVF International Conference on Computer Vision*, pp. 9266–9275. doi: 10.1109/ICCV.2019.00936.

Li, Junfei, Penghao Sun, and Yuxiang Hu (2020). "Traffic modeling and optimization in datacenters with graph neural network". In: *Computer Networks* 181, p. 107528. issn: 1389-1286. doi: 10.1016/j.comnet.2020.107528.

Li, Ming et al. (2020b). "Fast Haar Transforms for Graph Neural Networks". In: *Neural Networks* 128, pp. 188–198. issn: 0893-6080. doi: https://doi.org/10.1016/j.neunet.2020.04.028.

Li, Xiaoyu et al. (2019b). "Classify EEG and reveal latent graph structure with spatio-temporal graph convolutional neural network". In: *International Conference on Data Mining*. Vol. 2019-Novem. IEEE, pp. 389–398. doi: 10.1109/ICDM.2019.00049.

Li, Yaguang et al. (2018a). "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting". In: *International Conference on Learning Representations*, pp. 1–16.

Li, Yujia et al. (2018b). "Learning Deep Generative Models of Graphs". In: *International Conference on Machine Learning*, pp. 1–22.

Li, Zhishuai et al. (2019c). "A Hybrid Deep Learning Approach with GCN and LSTM for Traffic Flow Prediction*". In: *IEEE Intelligent Transportation Systems Conference*, pp. 1929–1933. doi: 10.1109/ITSC.2019.8916778.

Li, Zhishuai et al. (2022). "A Multi-Stream Feature Fusion Approach for Traffic Prediction". In: *IEEE Transactions on Intelligent Transportation Systems* 23.2, pp. 1456–1466. issn: 1558-0016. doi: 10.1109/TITS.2020.3026836.

Li, Zuchao et al. (2018c). "Seq2seq Dependency Parsing". In: *Proceedings of the 27th International Conference on Computational Linguistics*. Association for Computational Linguistics, pp. 3203–3214.

Lippmann, R P et al. (2000). "Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation". In: *DARPA Information Survivability Conference and Exposition*. Vol. 2, pp. 12–26. doi: 10.1109/DISCEX.2000.821506.

Luan, Sitao et al. (2019). "Break the Ceiling: Stronger Multi-scale Deep Graph Convolutional Networks". In: *Advances in Neural Information Processing Systems*. Ed. by Hanna M Wallach et al., pp. 10943–10953.

Ma, Hehuan et al. (2022). "Cross-dependent graph neural networks for molecular property prediction". In: *Bioinformatics*. Ed. by Zhiyong Lu, btac039. issn: 1367-4803, 1460-2059. doi: 10.1093/bioinformatics/btac039.

Ma, Yao and Jiliang Tang (2021). *Deep Learning on Graphs*. 1st ed. Cambridge University Press. doi: 10.1017/9781108924184.

Ma, Yi et al. (2019). *Spectral-based Graph Convolutional Network for Directed Graphs*. DOI: 10.48550/arXiv.1907.08990.

Mayr, Andreas et al. (2018). "Large-scale comparison of machine learning methods for drug target prediction on ChEMBL". In: *Chemical Science* 9.24, pp. 5441–5451. ISSN: 2041-6539. DOI: 10.1039/C8SC00148K.

Maziarka, Łukasz et al. (2020). *Molecule Attention Transformer*. DOI: 10.48550/arXiv.2002.08264.

McCulloch, Warren S. and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4, pp. 115–133. ISSN: 0007-4985, 1522-9602. DOI: 10.1007/BF02478259.

McNemar, Quinn (1947). "Note on the sampling error of the difference between correlated proportions or percentages". In: *Psychometrika* 12.2, pp. 153–157. ISSN: 00333123. DOI: 10.1007/BF02295996.

Mello, Micael O. M. C. de et al. (2016). "Improving load balancing, path length, and stability in low-cost wireless backhauls". In: *Ad Hoc Networks* 48, pp. 16–28. ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2016.05.002.

Micheli, Alessio (2009). "Neural Network for Graphs: A Contextual Constructive Approach". In: *IEEE Transactions on Neural Networks* 20.3, pp. 498–511. ISSN: 1941-0093. DOI: 10.1109/TNN.2008.2010350.

Minsky, Marvin and Seymour A. Papert (1972). *Perceptrons: an introduction to computational geometry*. 2. print. with corr. The MIT Press.

Monti, Federico, Michael M. Bronstein, and Xavier Bresson (2017). "Geometric matrix completion with recurrent multi-graph neural networks". In: *Advances in Neural Information Processing Systems*.

Morgan, Jay, Adeline Paiement, and Christian Klinke (2022). *Domain-informed graph neural networks: a quantum chemistry case study*. DOI: 10.48550/arXiv.2208.11934.

Neyshabur, Benham et al. (2017). "Exploring Generalization in Deep Learning". In: *Neural Information Processing Systems*, pp. 1–10.

Niepert, Mathias, Mohamed Ahmed, and Konstantin Kutzkov (2016). "Learning Convolutional Neural Networks for Graphs". In: *International Conference on Machine Learning*, pp. 2014–2023. DOI: 10.5555/3045390.3045603.

Oono, Kenta and Taiji Suzuki (2020). "Graph Neural Networks Exponentially Lose Expressive Power for Node Classification". In: *International Conference on Learning Representations*, pp. 1–37.

Paleja, Ameya (2023). *Microsoft unveils the world's first analog optical computer to solve optimization problems*.

Pelkonen, Tuomas et al. (2015). "Gorilla: a fast, scalable, in-memory time series database". In: *Proceedings of the VLDB Endowment* 8.12, pp. 1816–1827. ISSN: 2150-8097. DOI: 10.14778/2824032.2824078.

Protogerou, Aikaterini et al. (2020). "A graph neural network method for distributed anomaly detection in IoT". In: *Evolving Systems*. ISSN: 1868-6478. DOI: 10.1007/s12530-020-09347-0.

Rafique, Danish et al. (2018). "Cognitive Assurance Architecture for Optical Network Fault Management". In: *Journal of Lightwave Technology* 36.7, pp. 1443–1450. ISSN: 1558-2213. DOI: 10.1109/JLT.2017.2781540.

Ramsundar, Bharath et al. (2019). *Deep Learning for the Life Sciences*. O'Reilly Media.

Ranzato, Marc'Aurelio, Y-lan Boureau, and Yann Cun (2007). "Sparse Feature Learning for Deep Belief Networks". In: *Advances in Neural Information Processing Systems*. Vol. 20. Curran Associates, Inc.

Reddi, Sashank J, Satyen Kale, and Sanjiv Kumar (2018). "On the Convergence of Adam and Beyond". In.

Ren, Haoshi et al. (2020). "NetCruiser: Localize Network Failures by Learning from Latency Data". In: *Proceedings of the IEEE International Conference on Smart Internet of Things (SmartIoT)*, pp. 23–30. DOI: 10.1109/SmartIoT49966.2020.00013.

Rigoni, Davide, Nicolò Navarin, and Alessandro Sperduti (2020). *Conditional Constrained Graph Variational Autoencoders for Molecule Design*.

Rogers, David and Mathew Hahn (2010). "Extended-Connectivity Fingerprints". In: *Journal of Chemical Information and Modeling* 50.5, pp. 742–754. ISSN: 1549-9596. DOI: 10.1021/ci100050t.

Rong, Yu et al. (2020a). "DropEdge: Towards Deep Graph Convolutional Networks on Node Classification". In: *International Conference on Learning Representations*, pp. 1–17.

Rong, Yu et al. (2020b). "Self-Supervised Graph Transformer on Large-Scale Molecular Data". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., pp. 12559–12571.

Rosenblatt, F. (1958). "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6, pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519.

Rui, Liu, Hossein Nejati, and Ngai-Man Cheung (2016). "Dimensionality reduction of brain imaging data using graph signal processing". In: *IEEE International Conference on Image Processing (ICIP)*. IEEE, pp. 1329–1333. DOI: 10.1109/ICIP.2016.7532574.

Russell, Bertrand (2010). *History of Western Philosophy*. Routledge.

Santoro, Adam et al. (2017). “A simple neural network module for relational reasoning”. In: *Conference on Neural Information Processing Systems*, pp. 4967–4976.

Scarselli, Franco et al. (2009). “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1, pp. 61–80. ISSN: 1941-0093. DOI: 10.1109/TNN.2008.2005605.

Schrödinger, Erwin (1992). *What Is Life?: With Mind and Matter and Autobiographical Sketches*. Canto classics. Cambridge University Press.

Schütt, Kristof et al. (2017). “SchNet: A continuous-filter convolutional neural network for modeling quantum interactions”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc.

Selvan, Raghavendra et al. (2020). “Graph refinement based airway extraction using mean-field networks and graph neural networks”. In: *Medical Image Analysis* 64. DOI: 10.1016/j.media.2020.101751Get.

Shui, Zeren and George Karypis (2020). “Heterogeneous Molecular Graph Neural Networks for Predicting Molecule Properties”. In: *2020 IEEE International Conference on Data Mining (ICDM)*, pp. 492–500. DOI: 10.1109/ICDM50108.2020.00058.

Shuman, David I. et al. (2013). “The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains.” In: *IEEE Signal Processing Magazine*. ISSN: 10535888. DOI: 10.1109/MSP.2012.2235192.

Simonovsky, Martin and Nikos Komodakis (2017). “Dynamic edge-conditioned filters in convolutional neural networks on graphs”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. DOI: 10.1109/CVPR.2017.11.

Smith, Robert Elliott (2019). *Rage inside the Machine: The Prejudice of Algorithms, and How to Stop the Internet Making Bigots of Us All*. Bloomsbury Publishing.

Song, Tengfei et al. (2019). “EEG Emotion Recognition Using Dynamical Graph Convolutional Neural Networks”. In: *IEEE Transactions on Affective Computing*, pp. 1–1. ISSN: 1949-3045. DOI: 10.1109/TAFFC.2018.2817622.

Sperduti, A. and A. Starita (1997). “Supervised neural networks for the classification of structures”. In: *IEEE Transactions on Neural Networks* 8.3, pp. 714–735. ISSN: 1941-0093. DOI: 10.1109/72.572108.

Srinivasan, Srinikethan Madapuzi, Tram Truong-Huu, and Mohan Gurusamy (2019). “Machine Learning-Based Link Fault Identification and Localization in Complex Networks”. In: *IEEE Internet of Things Journal* 6.4, pp. 6556–6566. ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2908019.

Srivastava, Nitish et al. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15, pp. 1929–1958.

St. John, Peter C. et al. (2019). "Message-passing neural networks for high-throughput polymer screening". In: *The Journal of Chemical Physics* 150.23, p. 234111. ISSN: 0021-9606. DOI: 10.1063/1.5099132.

Steenkiste, Sjoerd van et al. (2018). "Relational Neural Expectation Maximization: Unsupervised Discovery of Objects and their Interactions". In: *International Conference on Learning Representations*, pp. 1–15.

Sukhbaatar, Sainbayar, Arthur Szlam, and Rob Fergus (2016). "Learning Multiagent Communication with Backpropagation". In: *Conference on Neural Information Processing Systems*, pp. 2244–2252.

Sun, Mengying et al. (2021). "MoCL: Data-driven Molecular Fingerprint via Knowledge-aware Contrastive Learning from Molecular Graph". In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21. Association for Computing Machinery, pp. 3585–3594. DOI: 10.1145/3447548.3467186.

Szegedy, Christian et al. (2015). "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

Ta, Xuxiang et al. (2022). "Adaptive Spatio-temporal Graph Neural Network for traffic forecasting". In: *Knowledge-Based Systems*, pp. 1–34. ISSN: 09507051. DOI: 10.1016/j.knosys.2022.108199.

Tang, Cong et al. (2020). "A General Traffic Flow Prediction Approach Based on Spatial-Temporal Graph Attention". In: *IEEE Access* 8, pp. 153731–153741. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3018452.

Temperley, H. N. V. (1981). *Graph Theory and Applications*. Ellis Horwood, Ltd.

Tong, Zekun et al. (2020). *Directed Graph Convolutional Network*. DOI: 10.48550/arXiv.2004.13970.

Trivedi, Rakshit, Jiachen Yang, and Hongyuan Zha (2020). "GraphOpt: Learning Optimization Models of Graph Formation". In: *International Conference on Machine Learning*.

Vaswani, Ashish et al. (2017). "Attention Is All You Need". In: *Conference on Neural Information Processing Systems*, pp. 1–11.

Veličković, Petar et al. (2018). "Graph Attention Networks". In: *International Conference on Learning Representations*, pp. 1–12.

Wang, Chunnan et al. (2020). "Auto-STGCN: Autonomous Spatial-Temporal Graph Convolutional Network Search Based on Reinforcement Learning and Existing Research Results". In.

Wang, Mowei et al. (2018a). “Machine Learning for Networking: Workflow, Advances and Opportunities”. In: *IEEE Network* 32.2, pp. 92–99. ISSN: 0890-8044. DOI: 10.1109/MNET.2017.1700200.

Wang, Wei et al. (2018b). “HAST-IDS: Learning Hierarchical Spatial-Temporal Features Using Deep Neural Networks to Improve Intrusion Detection”. In: *IEEE Access* 6, pp. 1792–1806. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2780250.

Wang, Xu et al. (2019). “Spatio-Temporal Analysis and Prediction of Cellular Traffic in Metropolis”. In: *IEEE Transactions on Mobile Computing* 18.9, pp. 2190–2202. ISSN: 1558-0660. DOI: 10.1109/TMC.2018.2870135.

Watters, Nicholas et al. (2017). “Visual Interaction Networks: Learning a Physics Simulator from Video”. In: *Conference on Neural Information Processing Systems*.

Williams, Michael R. (1997). *A History of Computing Technology*. 2nd ed. IEEE Computer Society Press.

Wolpert, David H. (1996). “The Lack of A Priori Distinctions Between Learning Algorithms”. In: *Neural Computation* 8.7, pp. 1341–1390. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1996.8.7.1341.

Wu, Felix et al. (2019a). “Simplifying Graph Convolutional Networks”. In: *Proceedings of the 36th International Conference on Machine Learning*. PMLR, pp. 6861–6871.

Wu, Zhenqin et al. (2018). “MoleculeNet: a benchmark for molecular machine learning”. In: *Chemical Science* 9.2, pp. 513–530. DOI: 10.1039/C7SC02664A.

Wu, Zonghan et al. (2019b). “A Comprehensive Survey on Graph Neural Networks”. In: *ArXiv*.

Wu, Zonghan et al. (2019c). “Graph WaveNet for Deep Spatial-Temporal Graph Modeling”. In: *International Joint Conference on Artificial Intelligence*, pp. 1907–1913. DOI: 10.24963/ijcai.2019/264.

Xia, Jun et al. (2022). *A Systematic Survey of Molecular Pre-trained Models*. DOI: 10.48550/arXiv.2210.16484.

Xiao, Yihan et al. (2019). “An Intrusion Detection Model Based on Feature Reduction and Convolutional Neural Networks”. In: *IEEE Access* 7, pp. 42210–42219. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2904620.

Xie, Z et al. (2019). “Sequential Graph Neural Network for Urban Road Traffic Speed Prediction”. In: *IEEE Access*, p. 1. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2915364.

Xiong, Zhaoping et al. (2020). “Pushing the Boundaries of Molecular Representation for Drug Discovery with the Graph Attention Mechanism”. In: *Journal of Medicinal Chemistry* 63.16, pp. 8749–8760. ISSN: 0022-2623, 1520-4804. DOI: 10.1021/acs.jmedchem.9b00959.

Xu, Bingbing et al. (2019a). “Graph Wavelet Neural Network”. In: *International Conference on Learning Representations*, pp. 1–13.

Xu, Keyulu et al. (2019b). “How Powerful are Graph Neural Networks?” In: *International Conference on Learning Representations*, pp. 1–17.

Xu, Youjun et al. (2019c). “Deep learning for molecular generation”. In: *Future Medicinal Chemistry* 11.6, pp. 567–597. ISSN: 1756-8919, 1756-8927. DOI: 10.4155/fmc-2018-0358.

Yang, Hui et al. (2022). “Accurate Fault Location using Deep Neural Evolution Network in Cloud Data Center Interconnection”. In: *IEEE Transactions on Cloud Computing* 10.2, pp. 1402–1412. ISSN: 2168-7161. DOI: 10.1109/TCC.2020.2974466.

Yang, Kevin et al. (2019). “Analyzing Learned Molecular Representations for Property Prediction”. In: *Journal of Chemical Information and Modeling* 59.8, pp. 3370–3388. ISSN: 1549-9596. DOI: 10.1021/acs.jcim.9b00237.

You, Jiaxuan et al. (2018). “Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation”. In: *Conference on Neural Information Processing Systems*, pp. 6412–6422.

Yu, Bing, Haoteng Yin, and Zhanxing Zhu (2018). “Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting”. In: *IJCAI International Joint Conference on Artificial Intelligence*, pp. 3634–3640. DOI: 10.24963/ijcai.2018/505.

Yu, James J. Q. (2022). “Graph Construction for Traffic Prediction: A Data-Driven Approach”. In: *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–13. ISSN: 1524-9050. DOI: 10.1109/TITS.2021.3136161.

Zeiler, Matthew D. and Rob Fergus (2014). “Visualizing and Understanding Convolutional Networks”. In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Vol. 8689. Springer International Publishing, pp. 818–833. DOI: 10.1007/978-3-319-10590-1_53.

Zhang, Qi et al. (2020). “Spatio-Temporal Graph Structure Learning for Traffic Forecasting”. In: *AAAI Conference on Artificial Intelligence* 34.01, pp. 1177–1185. ISSN: 2374-3468. DOI: 10.1609/aaai.v34i01.5470.

Zhang, Xikun et al. (2019). “Graph Edge Convolutional Neural Networks for Skeleton-Based Action Recognition”. In: *IEEE Transactions on Neural Networks and Learning Systems*. ISSN: 2162-237X. DOI: 10.1109/tnnls.2019.2935173.

Zhang, Yin et al. (2005). “Network Anomography”. In: *ACM SIGCOMM Conference on Internet Measurement*. ACM Press, p. 1. DOI: 10.5555/1251086.1251116.

Zhang, Yue et al. (2022). “Application of Computational Biology and Artificial Intelligence in Drug Design”. In: *International Journal of Molecular Sciences* 23.21, p. 13568. ISSN: 1422-0067. DOI: 10.3390/ijms232113568.

Zhang, Ziwei, Peng Cui, and Wenwu Zhu (2020). “Deep Learning on Graphs: A Survey”. In: *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1. ISSN: 1041-4347. DOI: 10.1109/TKDE.2020.2981333.

Zhao, L et al. (2019). “T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction”. In: *IEEE Transactions on Intelligent Transportation Systems*. IEEE, pp. 3848–3858. DOI: 10.1109/TITS.2019.2935152.

Zhao, Lingxiao and Leman Akoglu (2020). “PairNorm: Tackling Oversmoothing in GNNs”. In: *International Conference on Learning Representations*, pp. 1–17.

Zheng, Xuebin et al. (2020). “Graph Neural Networks with Haar Transform-Based Convolution and Pooling: A Complete Guide”. In: *ArXiv*.

Zhou, Jie et al. (2018). “Graph Neural Networks: A Review of Methods and Applications”. In: *ArXiv*.

Zhou, Kaixiong et al. (2020). “Towards Deeper Graph Neural Networks with Differentiable Group Normalization”. In: *Advances in Neural Information Processing Systems*. Ed. by Hugo Larochelle et al., pp. 1–12.

Zhu, Yanqiao et al. (2022). *A Survey on Graph Structure Learning: Progress and Opportunities*. DOI: 10.48550/arXiv.2103.03036.