UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PEDRO HENRIQUE DA COSTA AVELAR

Learning Centrality Measures with Graph Neural Networks

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Luis da Cunha Lamb

Porto Alegre
Fevereiro 2019

CIP — CATALOGING-IN-PUBLICATION

Avelar, Pedro Henrique da Costa

Learning Centrality Measures with Graph Neural Networks / Pedro Henrique da Costa Avelar. – Porto Alegre: PPGC da UFRGS, 2019.

120 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Luis da Cunha Lamb.

1. Deep neural networks. 2. Recurrent neural networks. 3. Graph neural networks. 4. Graphs. 5. Centrality measures. I. Lamb, Luis da Cunha. II. Título.

“If, then, a machine may have all the properties of a man, and act as a man while driven only by the ingenious plan of its construction and the interaction of its materials according to the principles of nature, then does it not follow that man may also be seen as a machine?”

— THE TALOS PRINCIPLE

ACKNOWLEDGEMENTS

This work was partially funded by CNPQ, CAPES and FAPERGS, which I thank greatly for their financial support. I also thank NVIDIA for the GPU grant conceded to our research group. I'd like to thank my lab colleagues Marcelo de Oliveira Rosa Prates and Henrique Lemos dos Santos, and my advisor Luis da Cunha Lamb for our partnership, joint work and support. I'd like to thank Felipe Roque Martins, Henrique José Avelar, Henrique Lemos dos Santos, Jessica Reinheimer de Lima and Marcelo de Oliveira Rosa Prates for their help proof-reading this dissertation and for their insightful comments. I'd also like to thank all my friends for their emotional support, in special my father Henrique and my mother Iris for their unwavering support throughout my life. Without the support of all these people, I would have never completed this Master's programme.

ABSTRACT

Centrality Measures are important metrics used in Social Network Analysis. Such measures allow one to infer which entity in a network is more central (informally, more important) than another. Analyses based on centrality measures may help detect possible social influencers, security weak spots, etc. This dissertation investigates methods for learning how to predict these centrality measures using only the graph's structure. More specifically, different ways of ranking the vertices according to their centrality measures are shown, as well as a brief analysis on how to approximate the centrality measures themselves. This is achieved by building on previous work that used neural networks to estimate centrality measures given other centrality measures. In this dissertation, we use the concept of a Graph Neural Network – a Deep Learning model that builds the computation graph according to the topology of a desired input graph. Here these models' performances are evaluated with different centrality measures, briefly comparing them with other machine learning models in the literature. The analyses for both the approximation and ranking of the centrality measures are evaluated and we show that the ranking of centrality measures is easier to compute. The transfer between the tasks of predicting these different centralities is analysed, and the advantages of each model is highlighted. The models are tested on graphs from different random distributions than the ones they were trained with, on graphs larger than the ones they saw during training as well as with real world instances that are much larger than the largest training graphs. The internal embeddings of the vertices produced by the model are analysed through lower-dimensional projections and conjectures are made on the behaviour seen in the experiments. Finally, we raise and identify possible future work highlighted by the experimental results presented here.

Keywords: Deep neural networks. recurrent neural networks. graph neural networks. graphs. centrality measures.

Aprendendo Medidas de Centralidade com Redes Grafo-Neurais

RESUMO

Medidas de Centralidade são um tipo de métrica importante na Análise de Redes Sociais. Tais métricas permitem inferir qual entidade é mais central (ou informalmente, mais importante) que outra. Análises baseadas em medidas de centralidade podem ajudar a detectar influenciadores sociais, pontos fracos em sistemas de segurança, etc. Nesta dissertação se investiga métodos para aprender a predizer estas medidas de centralidade utilizando somente a estrutura do grafo de entrada. Mais especificamente, são demonstradas diferentes formas de se classificar os vértices de acordo com suas medidas de centralidade, assim como uma breve análise de como aproximar estas medidas de centralidade. Nesta dissertação utiliza-se o conceito de uma Rede Grafo-Neural – um model de Aprendizagem Profunda que constrói o grafo de computação de acordo com a topologia do grafo que recebe de entrada. Aqui as performances destes modelos são avaliadas com várias medidas de centralidade e são comparadas com outros modelos de aprendizado de máquina na literatura. As análises para tanto a aproximação quanto a classificação das medidas de centralidade são feitas e se mostra que a classificação é mais fácil de ser computada. A transferência entre as tarefas de predizer as diferentes centralidades é analizada e as vantagens de cada modelo são destacadas. Os modelos são testados em grafos de distribuições aleatórias diferentes das quais foram treinados, em grafos maiores daqueles vistos durante o treinamento assim como com instâncias reais que são muito maiores do que as maiores instâncias vistas durante o treinamento. As representações internas dos vértices aprendidas pelo modelo são analisadas através de projeções de menor dimensão e se conjectura sobre o comportamento visto nos experimentos. Por fim, se identifica possíveis futuros trabalhosm destacados pelos resultados experimentais apresentados aqui.

Palavras-chave: redes neurais profundas, redes neurais recorrentes, redes grafo-neurais, grafos, medidas de centralidade.

LIST OF ABBREVIATIONS AND ACRONYMS

GNN	Graph Neural Network
GPU	Graphic Processing Unit
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
ML	Machine Learning
MSE	Mean Squared Error
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SAT	Boolean Satisfiability Problem
TGN	Typed Graph (Neural) Network
TSP	Travelling Salesperson Problem

LIST OF SYMBOLS

$\varpi(a, b)$	Number of shortest paths between a and b.
$\varpi(a, b\|c)$	Number of shortest paths between a and b that pass through c
σ	Sigmoid function
$\mathbb{R}$	Real number set
$\mathbb{1}\{\}$	A Tensor whose values are 1 in the indexes belonging to the set between the curly braces and 0 elsewhere
$\mathbf{1}$	A Tensor with all values set to 1 (its size can generally be inferred from its context)
$\mathbf{A}, \mathbf{B}', \mathbf{C}_{\mathbf{small}}, \ldots$	Tensors
$\mathbf{A}^\intercal$	The matrix $\mathbf{A}$, transposed.
$\mathbf{A}_{(i,j,k,\ldots)}$	Value at indexes $i, j, k, \ldots$ of tensor $\mathbf{A}$
$\mathbf{A}^i$	Tensor $\mathbf{A}$ at the i-th iteration of an algorithm
$\text{shape}(\mathbf{A})$	Tensor $\mathbf{A}$'s shape
$\mathbf{A} \times \mathbf{B}$	Matrix multiplication between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \cdot \mathbf{B}$	Point-wise multiplication between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} + \mathbf{B}$	Point-wise sum between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} - \mathbf{B}$	Point-wise subtraction between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A}/\mathbf{B}$	Point-wise division between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$	Sets
$\|\mathcal{A}\|$	Number of elements in set $\mathcal{A}$

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Recently, deep learning has made significant strides in many areas, replacing hand-engineering solutions with ones inferred from data (LECUN; BENGIO; HINTON, 2015). However, a common trend with this new success seems to be in the hand-engineering of models to solve different tasks, with a plethora of proposed and tested architectures for a variety of problems. In Image Classification/Computer Vision we have witnessed the evolution from using simple Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) (LECUN et al., 1998) for the MNIST, passing through CNNs' success on the ImageNet dataset (KRIZHEVSKY; SUTSKEVER; HINTON, 2012), Residual Networks (ResNets) (HE et al., 2016) and then now with research being done with Capsule Networks (CapsNets) in (SABOUR; FROSST; HINTON, 2017) and (HINTON; SABOUR; FROSST, 2018). In sequence-based data there has been the use of simple Recurrent Neural Networks (RNNs), which then have spawned a variety of Hand-Engineered variations that try to attack some of the issues encountered when learning sequence-based computation, such as the Long Short-Term Memories (LSTMs) (HOCHREITER; SCHMIDHUBER, 1997), their "simplified" version – the Gated Recurrent Units (GRUs) (CHO et al., 2014) – as well as many slight variations of both, the Neural Turing Machines (NTMs) (GRAVES; WAYNE; DANIHELKA, 2014) which adds an external memory that the network can manipulate, Neural GPUs (NGPUs) (KAISER; SUTSKEVER, 2015) and Differentiable Neural Computers (DNCs) (GRAVES et al., 2016). All of these try to avoid some pitfalls of simple end-to-end differentiable neural computation by hand-engineering features on the network's architectural level.

BATTAGLIA et al. (2018), in part, define these modifications as adding levels of relational inductive biases to the network. In their work they also make a survey of what they call "Graph Networks", which can be seen as an umbrella-term for their relational inductive bias hypothesis where they work towards unifying models which exhibit a graph-like architecture. More specifically, GILMER et al. use the term "Message-Passing Neural Networks" (MPNNs) (GILMER et al., 2017) for some instances of Graph Neural Networks (GNNs) (SCARSELLI et al., 2009) in their earlier survey as the umbrella-term for a subset of these networks and this is one way one can visualise the structure of GNNs: as being a network of computers, which work under an algorithm, communicating through message-passing to provide a joint solution to a computational problem. The architectures under the Graph Network umbrella are having a spike recently and have been used to

solve many relational and symbolic problems, and seem to couple with the notion of initial embeddings and dynamic times of computation, as shown in (GILMER et al., 2017) and (SELSAM et al., 2018).

All of this work, along with attentional interfaces, neural programmers and other techniques seem to be moving towards using neural network architectures as modules, integrating these different modules into tools and working with them as heuristics to aid in algorithms (OLAH; CARTER, 2016; BENGIO; LODI; PROUVOST, 2018). Furthermore, recent studies have suggested that advancing combinatorial generalisation is a key step forward in modern AI (BATTAGLIA et al., 2018). The results presented in this dissertation can be seen as a natural step towards this goal presenting, to the best of our knowledge, the first application of GNNs to multiple network centrality measures, a combinatorial problem with very relevant applications in our highly connected world, including the detection of power grid vulnerabilities (WANG; SCAGLIONE; THOMAS, 2010; LIU et al., 2018), influence inside interorganisational and collaboration networks (CHEN et al., 2017; DONG; MCCARTHY; SCHOENMAKERS, 2017), social network analysis (MORELLI et al., 2017; KIM; HASTAK, 2018), pattern recognition on biological networks (TANG et al., 2015; ESTRADA; ROSS, 2018) among others.

In general, vertex-level centralities summarise a vertex's contribution to the network cohesion. Several types of centralities have been proposed and many models and interpretations of these centralities have been suggested, namely: autonomy, control, risk, exposure, influence, etc. (BORGATTI; EVERETT, 2006). Despite their myriad of applications and interpretations, in order to calculate some of the more complex centrality measures one may face both high time and space complexity, thus making it costly to compute them on large networks. Although some studies pointed out a high degree of correlation between some of the most common centralities (LEE, 2006; BATOOL; NIAZI, 2014), it is also stated that these correlations are attached to the underlying network structure and thus may vary across different network distributions (SCHOCH; VALENTE; BRANDES, 2017). Therefore, techniques to allow faster centrality computation are topics of active research (EPPSTEIN; WANG, 2004; SARIYÜCE et al., 2017).

However, this work's goal is to provide more evidence to the power of GNNs on working with relational problems and not to provide a faster algorithm to compute these measures. Thus we focus on four well-known vertex centralities to investigate in this study:

Degree First proposed by (SHAW, 1954), it simply calculates to how many neighbours a vertex is connected.

Betweenness It calculates the number of shortest paths which cross by the given vertex. High betweenness vertices are more important to the graph's cohesion, i.e., their removal may disconnect the graph.

Closeness As defined by (BEAUCHAMP, 1965), it is a distance-based centrality which measures the average geodesic distance between a given vertex and all other reachable vertices.

Eigenvector This centrality uses the largest eigenvalue of the adjacency matrix to compute its eigenvector (BONACICH, 1987) and assigns to each vertex a score based upon the score of the vertices to whom it is connected.

1.1 Research Questions and Hypotheses

After presenting a short overview of the field, we would like to answer the following questions:

1. Can a neural network infer a vertex's centrality value only from the network structure?
2. Can a neural network learn an internal representation that translates into a vertex's centrality in a graph?
3. Can the representation from such a network benefit from the correlations between centrality measures and hold information about multiple centrality measures?
4. Will the algorithm learned by this neural network be scalable and be able to run for more iterations?
5. Will the algorithm learned by this neural network behave correctly for graphs larger than the ones it was trained?

From the previous work surveyed in the literature, it was hypothesised that the Research Question 1 may not to be answered positively. This is due both to some of the high relative absolute errors present in (GRANDO; LAMB, 2015) and to the fact that the end-task is more difficult in itself, since the algorithm receives only the network structure as input. Although it may be possible to rank the vertices from the information provided since there was a high correlation between the predicted values and the real values. The

answer to Research Question 2 was thought to be positive as well, and in a similar vein to (SELSAM et al., 2018) we may even expect that it might form a "centrality measure" of its own inside such a representation. The answer to Research Question 3 was hypothesised to be positive as well, and one could expect that the correlation between the centralities may result in a positive transfer between the multiple tasks. The answer to Research Question 4 was believed to be possible, again due to the results in (SELSAM et al., 2018). And finally, the answer to Research Question 5 was considered to be blurry, since degree distributions presented in the bigger graphs may cause unexpected behaviour on the representation learned by the algorithm due to numerical problems; also that Machine Learning algorithms tend to perform slightly worse on their test datasets; all this paired with the fact that performance drops as the graph size increases in (SELSAM et al., 2018) makes it hard to believe that it will generalise perfectly to bigger graphs, although it may be that, if the answer to Research Question 4 is positive, one can expect a similar behaviour and make the predictor run for more time steps in larger graphs.

1.2 Related Work

1.2.1 Predicting Centrality Measures with Neural Networks

The work of (GRANDO; LAMB, 2015; GRANDO; LAMB, 2016) uses neural networks to estimate centrality measures, but their work uses a priori knowledge of other centralities to approximate a different one, while our network uses only the network structure and produces centrality comparisons (i.e. is a vertex more central than another?). On (GRANDO; LAMB, 2018) they also produce a ranking of the centrality measures, but do so using the degree and eigenvector centralities as input, while the model presented here uses only the network structure and can produce any of the centralities tested. (KUMAR; MEHROTRA; MOHAN, 2015) uses local features such as number of vertices in a network, number of edges in a network, degree and the sum of the degrees of all of the vertex's neighbours, while we use only the network structure and the neighbourhood information is due to the message passing between neighbouring vertices. (SCARSELLI et al., 2005) uses GNNs to compute rankings for the PageRank centrality measure and does not focus on other centralities nor analyses the transfer between centralities.

1.2.2 Graph Neural Networks

Graph Neural Networks are having quite an explosion in their use recently, and one can enumerate some publications that have approaches similar in use to what we have done in here. For a survey of the area of Graph Neural Networks in general, one can look at (GILMER et al., 2017), (BATTAGLIA et al., 2018) and (ZHANG; CUI; ZHU, 2018). (SCARSELLI et al., 2005) learns to rank web pages to a search, but limits himself to approximating a single PageRank-like centrality measure. (LI et al., 2018), (YOU et al., 2018b) and (YOU et al., 2018a) work on learning generative models for graph generation, the first and the latter applying their work to the generation of chemical compounds, this can be seen as a step forward in modelling parameters for graphs, but their analyses don't study the relationships between the latent space and the characteristics of the networks itself. Finally, (SELSAM et al., 2018) explores the latent space learned by his network and uncovers significant emergent behaviour through visualising the PCA reduction of their embeddings as well as clustering these embeddings, some of the techniques we use in this work are very much like the ones applied in their methodology.

1.3 Contributions

During the Master's programme's duration we took part in several research projects, some of which are part of this dissertation. These contributions, if measured in terms of scientific papers, are the describe below. We include the abstracts of these papers to make it easier to understand the context of the contributions.

1.3.1 On Quantifying and Understanding the Role of Ethics in AI Research: A Historical Account of Flagship Conferences and Journals

> Recent developments in AI, Machine Learning and Robotics have raised concerns about the ethical consequences of both academic and industrial AI research. Leading academics, businessmen and politicians have voiced an increasing number of questions about the consequences of AI not only over people, but also on the large-scale consequences on the the future of work and employment, its social consequences and the sustainability of the planet. In this work, we analyse the use and the occurrence of ethics-related research in leading AI, machine learning and robotics venues. In order to do so we perform long term, historical corpus-based analyses on a large number of flagship conferences and journals. Our experiments identify the prominence of ethics-

> related terms in published papers and presents several statistics on related topics. Finally, this research provides quantitative evidence on the pressing ethical concerns of the AI community.

The student was the 2nd author of the paper on the submission to the 4th Global Conference on Artificial Intelligence, Luxembourg (Qualis unavailable), presented at the conference by Marcelo de Oliveira Rosa Prates (PRATES; AVELAR; LAMB, 2018). Code started in <https://github.com/phcavelar/Ethics-AI-Data>, moved to <https://github.com/marceloprates/Ethics-AI-Data> and also available at <https://github.com/phcavelar/Ethics-AI-Data-1>.

The contribution of each of the students are as follows:

Prates' contribution on this paper was doing the bulk of experimental work as well as writing most of the paper and revising it.

The author's contribution on this paper was scraping the data necessary for the study, discussing the results and writing a small part of the paper, as well as revising it.

1.3.2 Assessing Gender Bias in Machine Translation – A Case Study with Google Translate

> Recently there has been a growing concern in academia, industrial research labs and the mainstream commercial media about the phenomenon dubbed as *machine bias*, where trained statistical models – unbeknownst to their creators – grow to reflect controversial societal asymmetries, such as gender or racial bias. A significant number of Artificial Intelligence tools have recently been suggested to be harmfully biased towards some minority, with reports of racist criminal behavior predictors, Apple's Iphone X failing to differentiate between two distinct Asian people and the now infamous case of Google photos' mistakenly classifying black people as gorillas. Although a systematic study of such biases can be difficult, we believe that automated translation tools can be exploited through gender neutral languages to yield a window into the phenomenon of gender bias in AI.
>
> In this paper, we start with a comprehensive list of job positions from the U.S. Bureau of Labor Statistics (BLS) and used it in order to build sentences in constructions like "He/She is an Engineer" (where "Engineer" is replaced by the job position of interest) in 12 different gender neutral languages such as Hungarian, Chinese, Yoruba, and several others. We translate these sentences into English using the Google Translate API, and collect statistics about the frequency of female, male and gender-neutral pronouns in the translated output. We then show that Google Translate exhibits a strong tendency towards male defaults, in particular for fields typically associated to unbalanced gender distribution or stereotypes such as STEM (Science, Technology, Engineering and Mathematics) jobs. We ran these statistics against BLS' data for the frequency of female participation in each job position, in which we show that Google Translate fails to reproduce a real-world distribution of female workers. In summary, we provide experimental evidence that even if one does not expect in principle a 50:50 pronominal gender distribution, Google Translate

> yields male defaults much more frequently than what would be expected from demographic data alone.
>
> We believe that our study can shed further light on the phenomenon of machine bias and are hopeful that it will ignite a debate about the need to augment current statistical translation tools with debiasing techniques – which can already be found in the scientific literature.

Second author for the submission to Neural Computing and Applications (Qualis B1, 2013-2016), published at the journal (PRATES; AVELAR; LAMB, 2019). This work received attention in the international media, appearing on websites such as The Register[1], Tert [2], datanews[3], t3n[4] and others. Some time after this media coverage Google introduced a new feature in its Google Translate tool to provide translations for both genders <https://www.blog.google/products/translate/reducing-gender-bias-google-translate/> (Last accessed 21/01/2018), although this solution does not use gender neural problems, it at least promotes fairness in its available translations. The code was made available in <https://github.com/marceloprates/Gender-Bias> and <https://github.com/phcavelar/Gender-Bias>.

The contribution of each of the students are as follows:

Prates' contribution was the code communicating with Google Translate, experimental work, making the visualisations, as well as writing the majority of the paper and revising it.

The author's contribution on this paper involves gathering and wrangling the data from the United States of America's Bureau of Labor Statistics on job positions as well as the Adjective Corpus, writing part of the paper (mainly on the linguistic aspect of the languages taken into consideration for the study), defining the language templates used, experimental work and revision.

1.3.3 Multitask Learning on Graph Neural Networks – Learning Multiple Graph Centrality Measures with a Unified Network

> The application of deep learning to symbolic domains remains an active research endeavour. Graph neural networks (GNN), consisting of trained neural modules which can be arranged in different topologies at run time, are sound

[1] <https://www.theregister.co.uk/2018/09/10/boffins_bash_google_translate_for_sexist_language/> (Last accessed 21/01/2018)

[2] <https://www.tert.am/en/news/2018/09/12/google-translate/2788710> (Last accessed 21/01/2018)

[3] <https://datanews.knack.be/ict/nieuws/google-translate-is-seksistisch/article-normal-1195859.html> (Last accessed 21/01/2018)

[4] <https://t3n.de/news/google-translate-verstaerkt-sexistische-vorurteile-1109449/> (Last accessed 21/01/2018)

> alternatives to tackle relational problems which lend themselves to graph representations. In this paper, we show that GNNs are capable of multitask learning, which can be naturally enforced by training the model to refine a single set of multidimensional embeddings $\in \mathbb{R}^d$ and decode them into multiple outputs by connecting MLPs at the end of the pipeline. We demonstrate the multitask learning capability of the model in the relevant relational problem of estimating network centrality measures, i.e. is vertex v_1 more central than vertex v_2 given centrality c?. We then show that a GNN can be trained to develop a *lingua franca* of vertex embeddings from which all relevant information about any of the trained centrality measures can be decoded. The proposed model achieves 89% accuracy on a test dataset of random instances with up to 128 vertices and is shown to generalise to larger problem sizes. The model is also shown to obtain reasonable accuracy on a dataset of real world instances with up to 4k vertices, vastly surpassing the sizes of the largest instances with which the model was trained ($n = 128$). Finally, we believe that our contributions attest to the potential of GNNs in symbolic domains in general and in relational learning in particular.

First author (previously joint first author) on the submission the International Conference of Artificial Neural Networks in 2019 (Qualis B1, 2013-2016), presented as a poster by the student for the special session on graph neural networks (AVELAR et al., 2019). Code made available at <https://github.com/phcavelar/centrality-multitask>. Code for the library (version shown in this dissertation) available at <https://github.com/phcavelar/graph-neural-networks> with separate commits available in an earlier staging version in <https://github.com/phcavelar/graph-nn>.

This work is part of the theme of this dissertation. The contribution of each of the students are as follows:

The author's contribution to this paper was developing the bulk of the code for the core GNN and MLP modules, the groundwork for the batch separation code for extracting the results regarding the embeddings for each of the separate problems in the batch, the implementation of models prior to the comparison framework, part of the development of the comparison framework and supervising Lemos during his learning phase in implementing the comparison model and plotting some of the visualisations for analysing the embeddings. The author also generated the datasets, collected some of the real instances, and ran a good part of the experimental results, as well as writing a part of the paper and revising it.

Prates' contribution was part of the code for the core GNN module, part of the development of the comparison framework, supervising Lemos during his learning phase in implementing the comparison model, plotting some of the visualisations for analysing the embeddings, as well as writing a good part of the paper and revising it.

Lemos' contribution was writing a part of the paper, revising it, doing the literature review on the centrality measures and the effects of the graph distributions on them,

reworking the approximation model to the comparison framework, collecting some of the real instances, and plotting some of the visualizations for analysing the embeddings.

1.3.4 Learning to Solve NP-Complete Problems: A Graph Neural Network for the Decision TSP

Graph Neural Networks (GNN) are a promising technique for bridging differential programming and combinatorial domains. GNNs employ trainable modules which can be assembled in different configurations that reflect the relational structure of each problem instance. In this paper, we show that GNNs can learn to solve, with very little supervision, the decision variant of the Traveling Salesperson Problem (TSP), a highly relevant $\mathcal{NP}$-Complete problem. Our model is trained to function as an effective message-passing algorithm in which edges (embedded with their weights) communicate with vertices for a number of iterations after which the model is asked to decide whether a route with cost $< C$ exists. We show that such a network can be trained with sets of dual examples: given the optimal tour cost C^*, we produce one decision instance with target cost $x\%$ smaller and one with target cost $x\%$ larger than C^*. We were able to obtain 80% accuracy training with $-2\%, +2\%$ deviations, and the same trained model can generalize for more relaxed deviations with increasing performance. We also show that the model is capable of generalizing for larger problem sizes. Finally, we provide a method for predicting the optimal route cost within 2% deviation from the ground truth. In summary, our work shows that Graph Neural Networks are powerful enough to solve $\mathcal{NP}$-Complete problems which combine symbolic and numeric data.

Joint first author, with Marcelo de Oliveira Rosa Prates and Henrique Lemos dos Santos, on the submission to the 23rd AAAI Conference on Artificial Intelligence (Qualis A1, 2016), presented by Marcelo de Oliveira Rosa Prates. Pre-print made available (PRATES et al., 2019a). Code available at <https://github.com/machine-reasoning-ufrgs/TSP-GNN> and <https://github.com/phcavelar/TSP-GNN>. Code for the library (version shown here) available at <https://github.com/phcavelar/graph-neural-networks> with separate commits available in an earlier staging version in <https://github.com/phcavelar/graph-nn>.

The contribution of each of the students are as follows:

Prates' contribution to this paper was part of the code for the core GNN module, doing the bulk of the experimental work and setup (running experiments, trying different ways of producing the answer, generating the datasets, etc), writing most of the paper as well as revising it.

The author's contribution to this paper was developing the bulk of the code for the core GNN module as well as the MLP, the groundwork for the batch separation code for extracting the results regarding the embeddings for each of the separate problems in the

batch, the definition and implementation of the "acceptance curve" and its application on extracting route costs from a trained model. The author also contributed in the discussion of possible models and tentative solutions to the problems faced during the development of the experiments, as well as writing a small part of the paper and revising it.

Lemos' contribution to this paper was implementing the baseline models, discussion of possible models and tentative solutions to the problems faced during the development of the experiments, writing a small part of the paper and revising it.

1.3.5 Typed Graph Networks

> Recently, the deep learning community has given growing attention to neural architectures engineered to learn problems in relational domains. Convolutional Neural Networks employ parameter sharing over the image domain, tying the weights of neural connections on a grid topology and thus enforcing the learning of a number of convolutional kernels. By instantiating trainable neural modules and assembling them in varied configurations (apart from grids), one can enforce parameter sharing over graphs, yielding models which can effectively be fed with relational data. In this context, vertices in a graph can be projected into a hyperdimensional real space and iteratively refined over many message-passing iterations in an end-to-end differentiable architecture. Architectures of this family have been referred to with several definitions in the literature, such as Graph Neural Networks, Message-passing Neural Networks, Relational Networks and Graph Networks. In this paper, we revisit the original Graph Neural Network model and show that it generalises many of the recent models, which in turn benefit from the insight of thinking about vertex **types**. To illustrate the generality of the original model, we present a Graph Neural Network formalisation, which partitions the vertices of a graph into a number of types. Each type represents an entity in the ontology of the problem one wants to learn. This allows - for instance - one to assign embeddings to edges, hyperedges, and any number of global attributes of the graph. As a companion to this paper we provide a Python/Tensorflow library to facilitate the development of such architectures, with which we instantiate the formalisation to reproduce a number of models proposed in the current literature.

Joint first author, with Marcelo de Oliveira Rosa Prates and Henrique Lemos dos Santos, on the submission to the IEEE Transactions on Neural Networks and Learning Systems (Qualis A1, 2013-2016), rejected. Pre-print made available (PRATES et al., 2019b). Code for the library (version shown in this dissertation) available at <https://github.com/phcavelar/graph-neural-networks> with separate commits available in an earlier staging version in <https://github.com/phcavelar/graph-nn>. The library was renamed for this publication and is also available at <https://github.com/machine-reasoning-ufrgs/typed-graph-network>.

This work is part of the theme of this dissertation. The contribution of each of the students are as follows:

Prates' contributions were part of the code for the core GNN module, writing the majority of the paper as well as formalising the TGN meta-model.

The Author's contribution was doing the bulk of the implementation of the GNN library used for the TGN formalisation (and in the results collected from the 3 published papers cited) as well as revising the paper.

Lemos' contribution was writing part of the paper, revising it and providing preliminary results in regards to the Graph Colouring problem, which are yet to be published.

1.4 Dissertation Structure

This dissertation is structured as follows:

- This Chapter explains the research problem in Section 1.1, exposes related work in Section 1.2, lists the achieved scientific contributions in Section 1.3 and then defines the structure of the remainder of the document;
- Chapters 2 and 3 give most of the necessary background needed for reading this dissertation, and give pointers to what expected background is assumed. More specifically, the most important graph-theoretical background needed is in Chapter 2, along with the motivating topic of social networks and the definition of centrality measures analysed, and the background on Artificial Neural Networks and, most importantly, Graph Neural Networks are introduced in Chapter 3;
- Chapter 4 describes the carried out experiments and the results achieved throughout the Master's programme, in regard to this Dissertation's topic, and the submitted publications relevant to the dissertation;
- Then, Chapter 5 concludes this dissertation and exposes possible future work on the topics covered here.
- Additionally, Appendix A explains the GNN library used in the implementation of the articles.

2 ON SOCIAL AND COMPLEX NETWORKS

This chapter introduces some of the motivating topics behind this dissertation and also gives the necessary background on graph theory, setting the terminology used in the rest of the work. For further reading on the topic of network science and social networks, see (BARABÁSI et al., 2016)[1], (EASLEY; KLEINBERG, 2010)[2] and (NEWMAN, 2010).

2.1 What is a Network?

Over the last few decades, developed and developing countries have experienced a surge of new technologies enabling people to interact, learn and perform commercial transactions on a unseen scale. People have become highly dependant on the systems that enable such connectedness and many solutions have become so ubiquitous that the hardship of living without them seems threatening. As an example someone can, using only Facebook™, communicate with others through instant messaging, arrange events, enter communities of like-minded individuals, collaborate on documents, and both set up products for sale as well as search for products for buying. All of these functions aggregated on a single platform emphasise how the inherit value of something may depend on how many users it has.

This explosion of connectivity and, most importantly, the ability to gather and analyse data about these connected systems have helped form the field of study known as Network Science (BARABÁSI et al., 2016). This field studies systems which can be represented as entities and relations, such systems are called *networks* and their entities are represented by *nodes* and the relations between them are denoted as *links*. A prime example of such a system is the World Wide Web (the Web, for short) which is, today, a highly connected collection of Web pages that contain pointers (links) to other Web pages.

The Web, however, is not the sole example of highly connected system that one encounters daily. Many other types of networks exist and a person interacts with or is affected by many of these highly connected, highly complex systems, on an everyday basis. To name a few:

- **Social Networks** are the networks that describe the relationship between human beings – a network of people and their friendships, a network of people and their

[1] Available online at <www.networksciencebook.com> as of 29/12/2018
[2] Available online at <www.cs.cornell.edu/home/kleinber/networks-book/> as of 29/12/2018

co-workers, a networks of romantic relationships, **collaboration networks**, etc. This kind of network has been in existence since the first human beings and they determined the spread of knowledge, culture, and resources throughout all of human history.

- **Trade Networks** are the networks that define commercial transactions between different entities, such as the global network of trade between countries. These are the basis for the material prosperity humanity has been enjoying since the last World War.

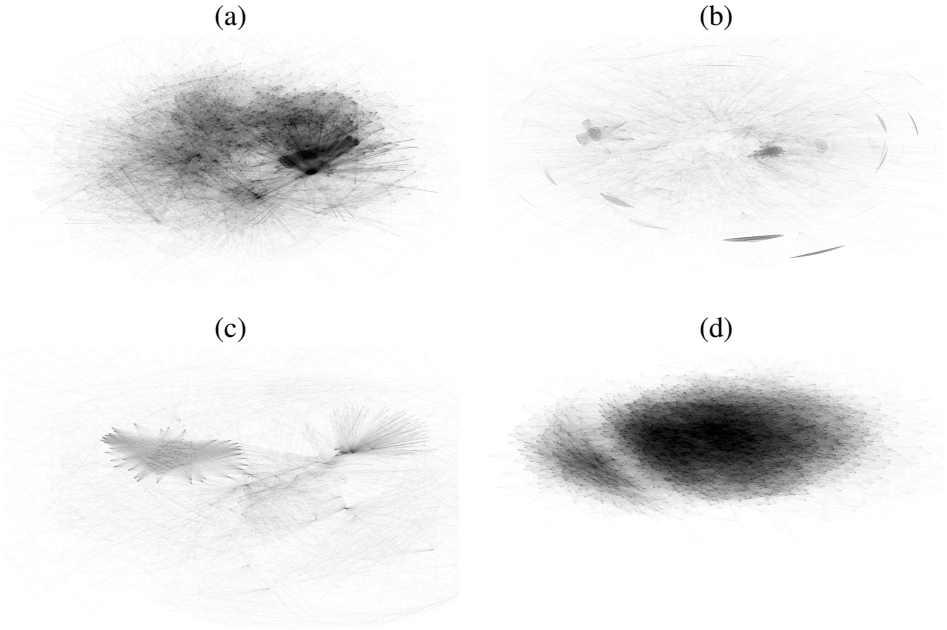
- Systems that transport a certain quantity of something from one point to another, and examples of this involve the logistics of **Distribution Networks** of supplies to armies or factories, **Transportation Networks** that model the transit of people during their daily routines or even the **Power Grids** that distribute the vital electric energy to its end-users;

- **Information Networks** are those systems that model the propagation of information along its nodes, the Web being a prime example of an information network, where a person can follow the links between the nodes of the web in search of information about a related article. The networks formed by the cross references between related items in an encyclopedia are another example of such information networks.

- **Chemical Networks** and **Biological Networks** are other types of complex networks that have a completely different motivation and evolution history behind them, in contrast to many of the networks exemplified above. They can vary as much as networks of taste compounds in recipes, to networks of metabolic chemical reactions inside a cell. One important networks that served as inspiration to a powerful tool that is also used in this dissertation is the **Neural Network** inside our brain, that was the main biological inspiration for the beginning of deep learning.

One can look at Fig. 2.1 for examples of real networks plotted, including a Biological Network, a Power Grid, a subset of a Social Network and a part of the Web Graph – an Information Network. Also, see Fig. 2.2 for another plot, now of a part of the Technological Network that is the internet. For more examples of different types of networks, one can look at the suggested bibliography for this chapter (BARABÁSI et al., 2016; EASLEY; KLEINBERG, 2010; NEWMAN, 2010).

Figure 2.1: Various instances of real networks plotted with invisible nodes and using the Kamada-Kawai force-directed algorithm (KAMADA; KAWAI, 1989) for laying out the node positions. Subfigure (a) is an example of a biological network, "bio-SC-GT". Subfigure (b) is the collaboration network "CA-GrQc". The network in Subfigure (c) is "econ-mahindas", an economical network. The plot in Subfigure (d) is a social network from Facebook™, "socfb-haverford76". More details of the networks can be seen in Table 4.2. Note that due to the fact that the edge width is very small, edges may only be visible when there is a large grouping of them. Source: Author, using data from (LESKOVEC; KREVL, 2014) and plotting using Networkx (HAGBERG; SWART; CHULT, 2008)



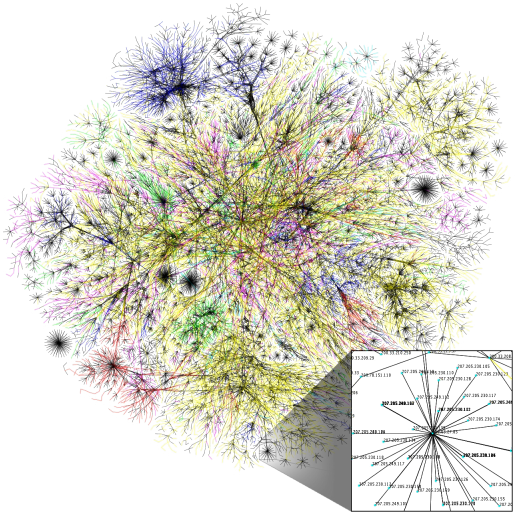
2.2 Characteristics of Complex Networks

One of the most important discoveries of network science is that many naturally occurring complex networks share properties that can be studied and exploited. For example, technological networks such as the internet, information networks such as the World Wide Web, biological networks such as metabolical networks, and social networks such as a snapshot of facebook connections all share some common properties such as low diameter, heavy tailed degree distribution and the presence of community structure.

The low diameter of complex networks was popularised by the principle of "six degrees of separation", which states any two people in the world would be separated by at most 6 "steps" of friendship. This idea gave rise to various metrics such as the *Erdös Number*[3] (NEWMAN, 2001; COLLINS; CHOW, 1998), which is the distance from a mathematician to Paul Erdös in the mathematics collaboration graph, the *Bacon Number*[4]

[3]<https://en.wikipedia.org/wiki/Erd%C5%91s_number>
[4]<https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon>

Figure 2.2: Partial map of the Internet based on the January 15, 2005 data found on opte.org. Each line is drawn between two nodes, representing two IP addresses. The length of the lines are indicative of the delay between those two nodes. This graph represents less than 30% of the Class C networks reachable by the data collection program in early 2005. Lines are colour-coded according to their corresponding RFC 1918 allocation (See source for colour coding). Source: Wikimedia Commons



(COLLINS; CHOW, 1998), that is the distance of an actor or actress to Kevin Bacon in the co-starring graph of artists on films, and even more so the popular game "5-clicks to Jesus"[5], in which a user starts from a random page on wikipedia and tries to reach the page for Jesus through the least amount of steps.

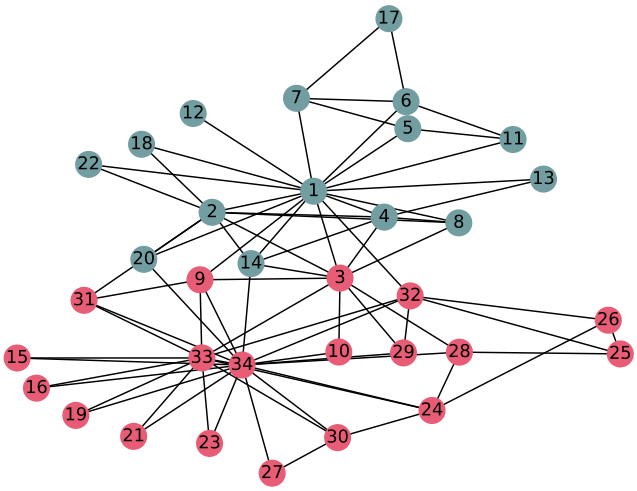
This principle can be understood by considering that there are some people who are central "hubs", that connect to a large amount of people and can serve as "shortcuts" between one entity and another. Such an idea was exploited for the purposes of searching web pages, and is an emergent property of most Complex Networks. This property, however, can be related to their heavy tailed degree distribution, that is also called the *scale-free property* of complex networks, which was first described in (ALBERT; JEONG; BARABÁSI, 1999). It simply means that the degree distribution of the network follows a power law relationship, and thus there is a small quantity of nodes that concentrate a large part of the degree counts of the network – that is, a few nodes that are connected to most nodes – whereas the large majority of nodes have very small degrees. We will revisit this in more detail in Subsection 2.4.1.

Finally, the community structure is the idea that nodes inside a community generally tend to have more connections between nodes in that same community than in other communities. For example, people that work in the same workplace tend to be more connected between themselves than with people from other workplaces. Similar ideas that

[5]<https://en.wikipedia.org/wiki/Wikipedia:Wiki_Game>

pertain to Social Networks are that of friend circles, hobbyist groups, and neighbours, but the idea of community structures has been used in Complex Networks as diverse from Social Networks as Metabolic Networks (RAVASZ; SOMERA ANNA LISA; BARABÁSI, 2002). A small and clear example of community structure is that of Zachary's Karate Club (WAYNE, 1977) which can be seen in Figure. 2.3

Figure 2.3: The connections of the 34 members of Zachary's Karate Club (WAYNE, 1977) where links denote that two members interacted outside the club. The colours are the best community partition predicted the Girvan-Newman method for community detection (GIRVAN; NEWMAN, 2002; NEWMAN; GIRVAN, 2004), which captures closely the split into two groups: The only node being assigned to a different faction being node 9, which should actually was on the same side as node 1 after the split. This anomalous behaviour is explained when one knows the history behind the node – the person corresponding to the node was almost completing a 4-year quest to obtain a black belt, something it could only achieve with the instructor's node (1). Source: Author, data from (WAYNE, 1977) plotted using the Networkx Python package (HAGBERG; SWART; CHULT, 2008)



2.3 Basics of Graph Theory

In this Section we will define the basic graph-theoretical concepts used in this dissertation. We will start with defining a graph, the main mathematical object used to model networks, and then proceed to give some definitions that will be important in the future, such as the notion of paths, shortest paths, distances and connected components. For the sake of uniformity, we will adopt from now on the graph-theoretic nomenclature for nodes which will be called *vertices*. For a more thorough explanation of graph theory itself,

see (BONDY; MURTY, 2008) or an older version (BONDY; MURTY et al., 1976)[6]. The suggested literature for network science cited at the start of this chapter (BARABÁSI et al., 2016; EASLEY; KLEINBERG, 2010; NEWMAN, 2010) also contains some concepts of graph theory.

A *graph* G is a pair $G(V, E)$ of *vertices* V and *edges* between vertices E. The source of an edge $e \in E$ is denoted $src(e)$ and its target $tgt(e)$, both of which are vertices $\in V$. Both vertices and edges may have properties associated with them. A common property for edges to have are numerical weights, denoted w_e. Graphs which have weights associated with their edges are called *weighted* graphs. Normally weights are real numbers, but for the applications of this dissertation one can assume that weights are non-negative real numbers $w_e \in \mathcal{R}_{\geq 0}$ from now on, unless the contrary is specified. If a graph is *unweighted* consider that $\forall e \in E, w_e = 1$. A graph is said to be *undirected* if $\forall e \in E \exists e' \in E$ such that $src(e) = tgt(e')$, $src(e') = tgt(e)$, and any other property of e is equal to the one in e'.

Given two vertices $v_s, v_t \in V$, a *path* between these two vertices $P(v_s, v_t)$ is a sequence of edges $(e_1, e_2, \ldots e_n)$ such that $src(e_{i+1}) = tgt(e_i) \forall i \in [0, n-1]$, $src(e_1) = v_s$ and $tgt(e_n) = v_t$. The distance of a path $|P(v_s, v_t)|$ is the sum of all weights (or distances) of the edges in the path $\sum_{i=0}^{n} w_i$, which is the version assumed here onwards unless specified, or of any other property that an edge may have. A path can also be seen as a sequence of vertices $(v_0, v_1, \ldots v_n)$ such that $v_0 = v_s$ and $v_n = v_t$, where $\forall v_i \in P(v_s, v_t) \setminus \{v_t\} \exists e, src(e) = v_i, tgt(e) = v_{i+1}$.

A *shortest path* between two given vertices $v_s, v_t \in V$ is a path $P'(v_s, v_t)$ such $\forall P(v_s, v_t), |P'(v_s, v_t)| \leq |P(v_s, v_t)|$, we denote the set of shortest paths between two vertices as $\mathcal{P}_{short}(v_s, v_t)$ and the set of paths between two vertices as $\mathcal{P}(v_s, v_t)$. The *distance between two vertices* is simply the distance of their shortest path.

A vertex v_t is said to be *reachable* from vertex v_s if and only if there exists a path from v_s to v_t. A graph is said to be *connected* if any vertex $\in V$ is reachable from any other vertex. If a graph is directed and such that, by making it undirected, it would become connected, it is called *weakly connected*, if a directed graph is connected it is called *strongly connected*. If a strict subgraph $G_c(V_c, E_c) \subset G(V, E), V_c \subset V, V_c \neq V, E_c = \{e \in E | src(e), tgt(e) \in V_c\}$ is connected (or either weakly or strongly connected) and is not a strict subgraph of a connected (or either weakly or strongly connected) strict subgraph of G, it is called a connected (or either weakly or strongly connected) component of the

[6] Available online at <citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.721.3161&rep=rep1&type=pdf> as of 29/12/2018

graph G.

For our applications, however, we will always use *unweighted undirected graphs*, the other definitions given here are mainly for the sake of completeness and to allow for further extensions of the model to adopt directed or weighted graphs.

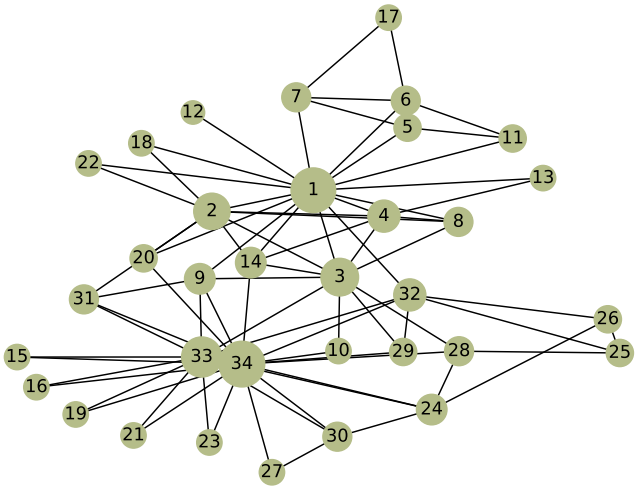
2.4 Centrality Measures

For the purpose of analysing the behaviour of complex networks, many metrics have been devised and tested. A subset of these metrics is what is called a *centrality measure*. In this section we will define and explain some of the uses for the centrality measures that were analysed in this dissertation. One can look at (NEWMAN, 2010) for a Network Science book with definitions for many network centrality measures, including the ones discussed in this dissertation. Also note that, while sometimes we do take care to define centrality measures for graphs with multiple disconnected components, our end-use application will assume that disjoint subgraphs are different graphs entirely and thus one can consider that the graphs are composed of a single connected component. Due to numerical constraints, we will take care to also define a version of each centrality measure that can easily fit into the $[0, 1]$ interval.

2.4.1 Degree and Degree Distribution

A vertex's *degree* is given by the number of edges that connect to or from it. In the case of a directed graph, there can be the definition of *in-degree* and *out-degree* which simply are, respectively, the number of edges that have the vertex as its target and the number of edges that have the vertex as its source. The **Degree Centrality Measure** (Denoted $D(i)$ as the degree of vertex i) of a vertex is simply the vertex's degree. Similarly are defined both the **In-Degree Centrality Measure** and **Out-Degree Centrality Measure**. The idea behind these metrics is that, if a vertex is connected to many other vertices, it can be considered more central than a vertex that has fewer connections. If a normalisation is needed for the the degree centrality measure of a vertex, one can simply divide the vertex's degree by the maximum possible degree of a network the same size as the network it is in (Denoted as $\overline{D}(i)$). One can look at Fig. 2.4 to see an example network with the vertex's size scaled by its degree.

Figure 2.4: The Karate Club example graph shown in Figure 2.3 with the vertices sized according to their degree centrality – the vertex with largest degree is 4 times as big as the one with the smallest degree. Source: Author, data from (WAYNE, 1977) plotted using the Networkx Python package



The whole set of degree values for a network is a very important information to describe the network itself (BARABÁSI et al., 2016), such information can be summarised as the *degree distribution* of a network – which defines the probability of a randomly selected vertex has a certain degree. We can write it as in Equation 2.1, where p_k denotes the probability that a randomly selected vertex has degree k, N_k is the number of vertices that have degree k and N is the total number of vertices.

$$p_k = \frac{N_k}{N} \tag{2.1}$$

With the degree distribution we can also calculate the average degree of a network $\langle k \rangle$, as in Equation 2.2. In real networks, however, we can work the definition as starting from the network's minimum degree k_{min} which is simply the smallest degree value in the network.

$$\langle k \rangle = \sum_{k=0}^{k=\infty} k p_k \tag{2.2}$$

In fact, we can generalise this equation to define the n^{th} moment of a distribution $\langle k^n \rangle$, in which $n = 1$ is the average degree, $n = 2$ helps defining the variance of a network's degree distribution s^2 through $s^2 = \langle k^2 \rangle - \langle k^1 \rangle^2$, and $n = 3$ defines the *skewness* of the distribution. Thus Equation 2.2 can be generalised as Equation 2.3, where we present both

the discrete and a continuous formulation being that the continuous one will be used later.

$$\langle k^n \rangle = \sum_{k=k_{min}}^{k=\infty} k^n p_k = \int_{k=k_{min}}^{\infty} k^n p(k) dk \tag{2.3}$$

In Section 2.2 we briefly described the *scale-free* nature of complex networks, but did not go in its details. With the definition of degrees and degree distributions however, we can now describe this in more detail. Since a Scale-Free network is a network whose degrees follow a power law distribution, we can say that $p_k = k^{-\gamma}$ is followed by the degree distribution[7]. If we consider a continuous formalism for this, we would have Equation 2.4 for the continuous degree distribution.

$$p(k) = Ck^{-\gamma} \tag{2.4}$$

Due to the condition that the function is a probability distribution, we have that $\int_{k_{min}}^{\infty} p(k)dk = 1$, and using Equation 2.3's continuous form, we'd have $\langle k^n \rangle \approx \int_{k=k_{min}}^{\infty} k^n p(k)dk$. We can substitute Equation 2.4 on this continuous form, and obtain the results of Equation 2.5.

$$\langle k^n \rangle = \int_{k=k_{min}}^{k_{max}} Ck^{n-\gamma} dk = C\frac{k_{max}^{n-\gamma+1} - k_{min}^{n-\gamma+1}}{n+\gamma-1} \tag{2.5}$$

Now, we need to analyse the values for $|\mathcal{V}| \to \infty$, under the following assumptions:

1. k_{min} remains bounded (generally 1 or 2).
2. $k_{max} \to \infty$ with $|\mathcal{V}| \to \infty$.
3. Real networks often have a degree exponent $2 \leq \gamma \leq 3$ (BARABÁSI et al., 2016).

We can see that, for very large networks, its moment as defined in Equation 2.5, due to Items 1 and 2, will depend on k_{max} diverging or not. We can see that this will, in fact, depend on $n - \gamma + 1 \leq 0$. Since Item 3 says that γ is between 2 or 3, only the first moment will satisfy this, and the rest will diverge.

But how, one may ask, this relates to anything? The fact is that this property is not seen in a random network that follows the Poisson distribution for degrees. In a random networks with the Poisson distribution, the second momentum and the third momentum have definite values, and these values are relative to the first momentum (the average degree), so that the average degree serves as a scale for the network[8]. In a Scale-Free

[7] Vertices with degree 0 are not considered, for the sake of simplicity

[8] We omit the steps to this claim here, but one could use the same logic to derive the exact values.

network, however, due to the divergence of the second and third momentum, the value for a randomly selected vertex's degree can be as small as k_{min} and arbitrarily big, and thus the average degree of the networks does not serve as a "scale" to it.

2.4.2 Betweenness

The **Betweenness Centrality Measure** is a path-centric metric, in which a vertex is declared central if it belongs to many different paths between vertices. In the case of the Betweenness centrality, a vertex's centrality is the amount of shortest paths which it belongs to, being that if a vertex has more than one shortest path to another vertex, the fraction of shortest paths between those two vertices that pass through the vertex is used instead of a unit value. That is, for every vertex A and B that are connected by a path, we calculate all the k shortest paths from vertex A to vertex B. The betweenness of each vertex that belongs to each of the shortest paths is then increased by $1/k$. One can look at Fig. 2.5 to see an example network with the vertex's size scaled by its betweenness.

In Equation 2.6, $B(i)$ is the betweenness of vertex i, $\mathcal{V}$ is the set of vertices of the Graph, $d(s,t)$ is the shortest distance from vertex s to vertex t (it is ∞ when vertex t is unreachable to vertex s), and $\varpi(s,t)$ is the number of shortest paths from vertex s to t while $\varpi(s,t|k)$ is the number of those shortest paths that pass through vertex k.

$$B(i) = \sum_{s,t \in \mathcal{V}, d(s,t) \neq \infty} \frac{\varpi(s,t|i)}{\varpi s,t} \tag{2.6}$$

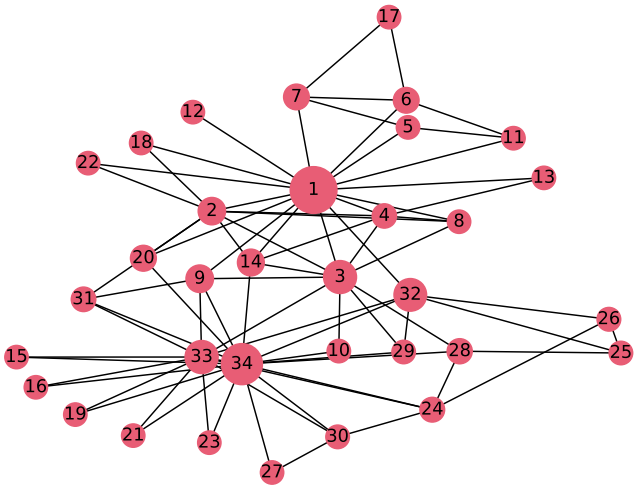
If we want a normalised version of the betweenness centrality measure, we can use the one defined in Equation 2.7, in which we divide the pure betweenness value by the maximum number of shortest paths that could pass through a vertex in that Graph. Note that we expect $|\mathcal{V}| \geq 3$, that the Graph has a single connected component (otherwise we could normalise for each component), and that the 2 in the numerator is due to the graph being undirected (thus we count every edge twice).

$$\overline{B}(i) = B(i) \cdot \frac{2}{(|\mathcal{V}| - 1) \cdot (|\mathcal{V}| - 2)} \tag{2.7}$$

One interpretation of vertices with high betweenness is that they are bridges between different parts of a network, so they, in a certain way, control the flow between different parts of the network. This interpretation is used, to a certain extent, in its use in the Girvan-

Newman method for community detection (GIRVAN; NEWMAN, 2002; NEWMAN; GIRVAN, 2004), in which edges[9] of high betweenness are removed to disconnect parts of a network and reveal community structures inside it.

Figure 2.5: The Karate Club example graph shown in Figure 2.3 with the vertices sized according to their betweenness centrality – the vertex with largest betweenness is 4 times as big as the one with the smallest betweenness. Source: Author, data from (WAYNE, 1977) plotted using the Networkx Python package



2.4.3 Closeness

The **Closeness Centrality Measure** is a centrality measure that uses the notion of inter-vertex distance in its definition. A vertex is called more central if its distance to other vertices is smaller. Therefore, the Closeness of a vertex is simply the reciprocal of the average of the distances between it and all vertices reachable from it. One can look at Fig. 2.6 to see an example network with the vertex's size scaled by its closeness.

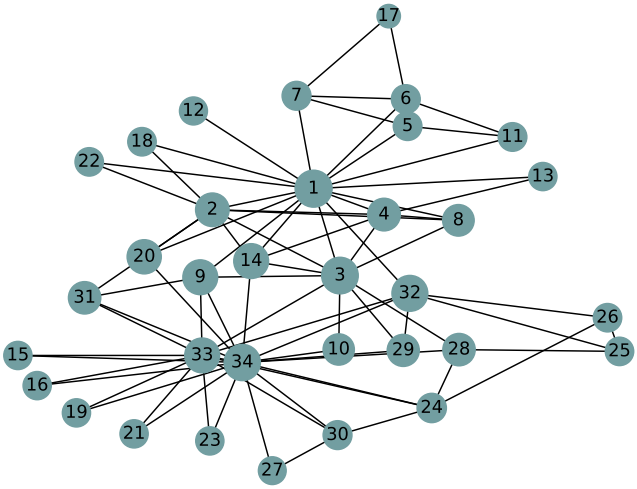
In Equation 2.8, $C(i)$ is the closeness of vertex i, $\mathcal{V}$ is the set of vertices of the Graph, $d(s,t)$ is the distance from vertex s to vertex t (it is ∞ when vertex t is unreachable to vertex s, and is calculated as the distance of the shortest path between them), and $r(s,t)$ is 1 if t is reachable from s and 0 otherwise. Note that if the edges are not weighted or are weighted with weights ≥ 1, the distances will also be ≥ 1 and since the closeness value is the reciprocal of the average of the distances, then its value will already be between 0 and

[9]Note that while we defined vertex betweenness, edge betweenness is defined following the same logic

1.

$$C(i) = \frac{\sum_{t \in \mathcal{V}} r(i,t)}{\sum_{t \in \mathcal{V}, d(i,t) \neq \infty} d(i,t)} \tag{2.8}$$

Figure 2.6: The Karate Club example graph shown in Figure 2.3 with the vertices sized according to their closeness centrality – the vertex with largest closeness is 4 times as big as the one with the smallest closeness. Source: Author, data from (WAYNE, 1977) plotted using the Networkx Python package



2.4.4 Eigenvector

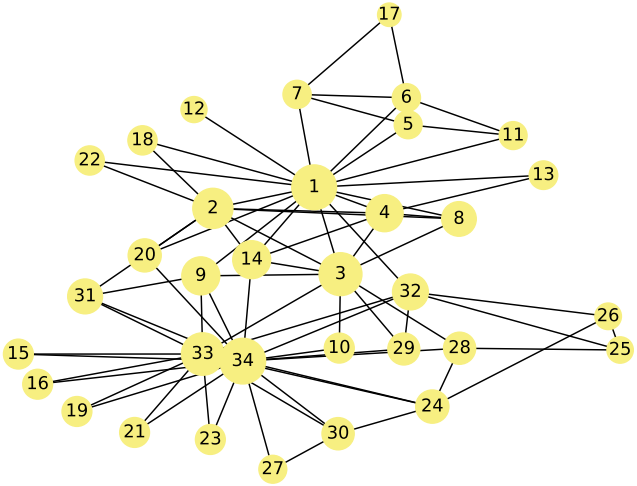
All of the measures described so far (i.e. in Subsections 2.4.1, 2.4.2 and 2.4.3) play with the notion that an important vertex either is one that has many connections (Degree) or that is close to others. In the second case, it can either be that is is in the shortest path between other vertices, as is the case with Betweenness, or it is nearer to other vertices in average, as is the case with Closeness. Another assumption that can be made is that an important (or, say, powerful) vertex is connected to vertices that are themselves also important (WĄS; SKIBSKI, 2018), which is one of the intuitions behind this metric. This is one of the ideas behind centrality measures such as PageRank or **Eigenvector**. One can look at Fig. 2.7 to see an example network with the vertex's size scaled by its Eigenvector centrality.

The Eigenvector Centrality Measure of a vertex i is the value in the i-th position of the eigenvector x generated by solving Equation 2.9, where M is the adjacency matrix of the graph and λ is the largest eigenvalue of the adjacency matrix. Due to the Perron-Frobenius theorem there will be an unique solution x, whose entries are all positive

(CHANG; PEARSON; ZHANG, 2008).

$$\mathbf{M}\mathbf{x} = \lambda\mathbf{x} \tag{2.9}$$

Figure 2.7: The Karate Club example graph shown in Figure 2.3 with the vertices sized according to their eigenvector centrality – the vertex with largest eigenvector centrality is 4 times as big as the one with the smallest eigenvector centrality. Source: Author, data from (WAYNE, 1977) plotted using the Networkx Python package



2.4.5 Summary

Centrality measures can be seen as a way to summarise information about a network's topology within a vertex. Albert-László Barabási especially enforces in his book (BARABÁSI et al., 2016) that the degree distribution is a key factor to determine a network's characteristic. Some authors, however, say that one should take centrality measures with a grain of salt, since, depending on the context they are applied, they can sometimes be misleading (EASLEY; KLEINBERG, 2010). However, for massive graphs, such as but not limited to the World Wide Web, it is hard to be able to analyse the network in detail and one must resort to these metrics.

Due to the Scale-Free property of most real networks, however, the degree of each vertex may vary hugely, and thus normalisations such as the ones proposed in the subsections fall short in being that they disconsider the power law nature of the network's connections. However, this can be ignored if only the ranking of the vertices matters, since it is not changed by the normalisation, and if one wishes only to identify the most important vertices in a network, they should need no further than the ranking of these

vertices.

Many other centrality measures have been proposed in the literature, we chose to work with the vertex-level centralities described in the subsections above simply as a question of scope. One can look at the recommended bibliography suggested at the beginning of this section for examples of other centrality measures, including edge-level centralities, and for some different definitions and interpretations of these centrality measures.

3 DEEP LEARNING AND GRAPH NEURAL NETWORKS

This chapter presents the fundamental concepts of the methodology applied in this work as well as the necessary building blocks to understand the main object of study: A Graph Neural Network. We also set the terminology for when we are dealing with deep learning concepts. Note that here we assume a basic knowledge of Calculus, Linear Algebra and Statistics. For a more thorough explanation of the area in general, except for the part of Graph Neural Networks, see (GOODFELLOW; BENGIO; COURVILLE, 2016)[1] or (NIELSEN, 2015)[2].

3.1 A Brief Introduction to Machine Learning

In Machine Learning (ML), the objective is to allow a computer to learn from data in order to perform a task without being explicitly programmed to do so. One could think of it as the task of fitting a set of *variables*, given some *input examples*, so that they provide the best performance for a given task on both seen and unseen examples. The models to build such a set of variables are manifold, and range from simple function regressors to decision trees. These models often try to minimise a *loss function*, which is often different from the *error function* used to evaluate the model. These functions can be seen as a form of distance metric from the current model to the desired model that fits the data correctly. The definition of a Machine Learning algorithm is, therefore, dependent on how its variables are defined and what is the procedure that optimises these values.

The process of Machine Learning is often categorised into a set of different tasks, some of which relate to the model's relationship with the data it receives. For instance, in *Supervised Learning* the model receives both the input examples (known as *training examples*) and the desired output for these examples (*known as labels*) and tries to optimise a function which maps inputs to outputs for some loss function that defines the distance that the output is from the desired output. One can employ Supervised Learning for both *Classification* problems, where the output is given in a categorical format (the possible values for an output being known as *Classes*), as well as *Regression*, where the desired output is a (possibly continuous) numerical function, and can also be extended to cases where only a fraction of the data has labels, when it is denominated *Semi-supervised*

[1] Available online at <www.deeplearningbook.org> as of 29/12/2018
[2] Available online at <neuralnetworksanddeeplearning.com> as of 29/12/2018

Learning. Some examples of Supervised learning are: identifying whether an image contains a cat or of a dog given examples of pictures with cats and dogs, transcribing speech recordings given as examples pairs of recordings and their respective correct transcriptions, and modelling an unknown function given a set of input values and the expected output value of the function being modelled.

ML algorithms can also deal with data when the desired output cannot be provided, in which the algorithm must then identify patterns present within the data without any labels attached to it. This task is called *Unsupervised Learning* and can either separate the data into different groups, which is known as *Clustering*, learn to identify anomalies in the presented data, known as *Anomaly Detection*, or attempt to build some model that relates the example data to a set of latent variables that describe it, of which a known model is Principal Component Analysis (PCA) (JOLLIFFE, 2011) that can be used to summarise a dataset's information into a lower-dimensional space than its original one.

There is also the task of *Reinforcement Learning*, which models the learning process of a decision-making agent on an environment who receives rewards based on whether the decisions it takes can be seen as desirable behaviour. This is generally done when one cannot possibly model the environment as a whole and analyse all possible states – regarding this, an example is the Chinese boardgame Go, known to have more than 2×10^{170} legal positions in a 19×19 board (the standard playing board) (TROMP; FARNEBÄCK, 2006)[3] – more possible positions than the estimated number of atoms in the observable universe.

In this dissertation we experiment mostly with Supervised Learning, although we use Unsupervised Learning to analyse the representation learned by our algorithm, and thus we will not discuss any more on Reinforcement or Unsupervised Learning. Other sub-tasks of Supervised Learning relevant to this dissertation are *Transfer Learning* and *Multitask Learning*. In Transfer Learning one aims to train an algorithm to solve a problem and use part of the acquired knowledge on this problem to solve yet another problem, which gives the definition of *Positive Transfer* between tasks. We say that a task has positive transfer to another if learning one can be beneficial to learning the other – that is, there is information contained in one that is redundantly required in the other. We are also interested in Multitask Learning, where one trains an algorithm to solve jointly different (and maybe related) tasks, in the hopes that optimising these different objectives may lead to a better performance than if trained separately (BENGIO, 2012).

[3]The number of legal positions for the 19×19 board is shown in <https://tromp.github.io/go/gostate.pdf>, retrieved 10/01/2019.

The optimisation done on Supervised Learning, however, is often not made on the whole dataset. If all the possible input values are known beforehand, a simple algorithm can be devised to always produce the correct input. And the value of ML algorithms is calculated in how well they generalise to unseen data. Thus, one generally splits the dataset into two different *train* and *test* datasets, and applies the variable fitting procedure only the train dataset, saving the test dataset only to keep track of how well the model is generalising. We call the loss/error of the algorithm on the training dataset the train loss/error, and similarly the test loss/error. A machine learning algorithm can often fit a dataset too tightly, presenting a lower training loss/error than test loss/error, which is called *overfitting*, conversely it can also *underfit* a dataset, producing high loss/error values on both train and test datasets.

3.2 On Tensors in Deep Learning

In the next session we will explain the advantages of looking at Deep Learning through the lenses of tensors and linear algebra. In this section we will explain some basic definitions about tensors which will be used throughout the dissertation and provide the basic knowledge to understand the rest of the content.

One can view a *Tensor* as a generalisation of a *Matrix*, which in itself is a generalisation of a *Vector*, which can be see as a collection of *Scalars*. A scalar is simply a number, say 1.0, and encodes ground information about something – for example a constant in physics. We say that a vector has d dimensions to mean that it holds d scalars in itself which we can access by indexing it, an example of a vector is the representation of a point in three dimensional space like $\mathbf{V} = (3.2, 2.3, -1.1)$, where 3.2 is the first element, 2.3 the second and -1.1 the third.

Now, a matrix, which is a fundamental concept used in linear algebra, is much like a vector, only that it has two "indexes" with which addresses the values it stores. A $n \times m$ matrix holds $n \cdot m$ values, and we can access those values through two indexes $1 \leq i \leq n$ and $1 \leq j \leq m$. A vector can be represented as a matrix with $d \times 1$ or $1 \times d$ dimensions. For example, Equation 3.1 shows a matrix, and we can access the number π through indexes $i = 1$ and $j = 3$, where we can see that our definitions lead us to see that i is the index for the *row* and j for the *column*. Equation 3.2 shows the previously defined

vector represented as a *column vector* and Equation 3.3 as a *row vector*.

$$\mathbf{M} = \begin{bmatrix} x_{1,1} & x_{1,2} & \pi & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,m} \\ & & \dots\dots\dots\dots & & \\ x_{n,1} & x_{n,2} & x_{n,3} & \dots & x_{n,m} \end{bmatrix} \tag{3.1}$$

$$\mathbf{V} = \begin{bmatrix} 3.2 \\ 2.3 \\ -1.1 \end{bmatrix} \tag{3.2}$$

$$\mathbf{V}^\intercal = \begin{bmatrix} 3.2 & 2.3 & -1.1 \end{bmatrix} \tag{3.3}$$

One can then easily extend this idea to having multiple *ranks*[4], being that a d-dimensional vector is something with rank 1, a $m \times n$ matrix is of rank 2 and a scalar is of rank 0. And thus, we are led to Tensors – which are objects $\mathbf{T}$ of rank k, that hold a dimensionality $d_i \geq 1$ for every $1 \leq i \leq k$. We call this description a tensor's *shape*, which itself is an array of integer values $\text{shape}(\mathbf{T}) = (d_1, d_2, \dots, d_k)$. One can also see that these tensors hold in them $\prod_{i=1}^{k} d_i$ values in total, so a rank with dimensionality 1 does not really add to the number of values that a Tensor holds. One does not really need to concern oneself with the specifics of tensorial algebra, however, since most of the operations used in deep learning are not specific to tensorial algebra but can be understood from linear algebra, such as point-wise operations and matrix multiplications between tensors.

3.3 Neural Networks

Usually, Neural Networks can be considered as graphs that connect inputs and operation outputs to operation inputs or outputs, but it is more mathematically, algorithmically, and numerically convenient to consider them as a series of tensor operations, that are differentiable end-to-end, in which some of these tensors are to be considered *variables* and can be optimised.

These variable tensors can be flattened into a set of variables. With this, we can calculate the gradient on such set given the input and the error based on the desired output of the function. With the gradients for the variables we can update the model to produce

[4]Note that the definition of rank here is different from that one may be accustomed to

a, supposedly, better predicting model (the error function we note here can be, and most times is, substituted by a loss function that is different to the error function itself but carries information about the distance of a variable to its desired state).

More formally, given a n_x-dimensional point x, its desired n_y-dimensional label y and a model M, composed of linear tensor transformations and (not necessarily linear) function activations, which use a set of n optimisable variables $\mathcal{V} = v_1, v_2, \dots v_n$. We denote the output of the model as being $y' = M(x, v_1, v_2, \dots, v_n)$. To perform gradient descent in such a model, one could define an error function $e(x, y, M, v_1, v_2, \dots v_n)$ which would define a distance from y' to y. This function can then be differentiated to produce the partial derivatives of it, in respect to each of the model's variables, producing the gradients $\nabla(\mathcal{V}) = \nabla(v_1, v_2, \dots v_n) = \{\frac{\delta e(x,y,M,v_1,v_2,\dots v_n)}{\delta v_1}, \frac{\delta e(x,y,M,v_1,v_2,\dots v_n)}{\delta v_2}, \dots \frac{\delta e(x,y,M,v_1,v_2,\dots v_n)}{\delta v_n}\}$. These gradients can be applied to $\mathcal{V}$ by subtracting from each variable its gradient, producing a new set of variables $\mathcal{V}' = \{v'_1, v'_2, \dots v'_n\} = \{v_i - \frac{\delta e(x,y,M,v_1,v_2,\dots v_n)}{\delta v_i}, v_i \in \mathcal{V}\}$, which when applied to the model M on the same input should produce a smaller error – that is, $e(x, y, M, v_1, v_2, \dots v_n) \geq e(x, y, M, v'_1, v'_2, \dots v'_n)$.

Such an idea can be extended to provide gradients for an arbitrary number b of points, where the model will be applied for every point-label pair, and then the errors will be calculated in tandem, producing a set of b errors. These errors are then aggregated into singular errors for each of the variables, being that the most common and straightforward aggregation of these errors would be a simple sum. That is, for a set of b points $\mathcal{P} = \{p_i, 1 \leq i \leq b\}$ and labels $\mathcal{Y} = \{y_i, 1 \leq i \leq b\}$, we calculate the errors of the model given $\mathcal{V}$ for every point-label pair as $\mathcal{E} = \{e_i(x_i, y_i, M, v_1, v_2, \dots v_n), 1 \leq i \leq b\}$, and then we have a set of b gradient-sets as $\nabla^B(\mathcal{V}) = \{\nabla_i(\mathcal{V}), 1 \leq i \leq b\}$. We can then get the set of aggregated gradients, given an aggregation r, as $\nabla^R(\mathcal{V}) = \nabla^R(v_1, v_2, \dots v_n) = \{r(\{\frac{\delta e_i(x,y,M,v_1,v_2,\dots v_n)}{\delta v_j}, 1 \leq i \leq b\}), 1 \leq j \leq n\}$. This aggregated gradient is then applied to the variables to produce the new variable set, as was discussed in the previous paragraph.

This application of the training process to a number of b point-label pairs is known as *batching* and it is used due to the fact that it accelerates the training process, taking advantage of data parallelism. This also serves to us as an example of parameter-sharing. We could see this batched model as a single model, which is basically the repetition of the same model b times to accommodate all the input points. The model's variables, however, are tied to each other across every instance of the model. In fact, our definition of a model does not limit how many times a variable can be used in a single application of the model. It could be that variable v_k is used many times in the model, and the errors calculated in

respect to it would then be calculated for each of these times and aggregated as in a batch update.

Theoretically, if one wishes to fit a model for a specified dataset, they could produce a single batch consisting of the entire dataset and run the gradient descent procedure above until the model produces a good fit (keeping in mind, of course, problems such as overfitting, underfitting, etc). However, due to memory constraints, most datasets cannot possibly fit in memory through such a procedure, and smaller batches are used. This technique, called mini-batching, produces models that are similar in performance to those with large batches, making it possible to train models on datasets that cannot possibly fit on memory and achieving similar results in terms of convergence to the desired performance.

Also, the gradient descent procedure as defined above is highly deterministic, which sometimes may lead to overfitting and to the model being trapped into local minima. Therefore the procedure is often augmented with extra stochastic and/or deterministic parts to the underlying algorithm to prevent such errors. These procedures will be referred to collectively as Stochastic Gradient Descent (SGD), and examples of these are the Adam (KINGMA; BA, 2014), Adagrad (DUCHI; HAZAN; SINGER, 2011) and Adadelta (ZEILER, 2012). For a more thorough overview of Gradient Descent in its many variants, one can look at the overview provided by (RUDER, 2016).

The main idea of using tensors, however, will only be readily visible when there is a need for multidimensional data such as images or videos. In these cases there are operations that operate on tensors of rank k and they consider that any excess rank "to the left" of the tensor itself can be simply flattened into an array for the purposes of computation. In this way, matrix multiplication in tensors is also simply defined as the matrix multiplication for the matrices, flattening the left of the left hand side tensor into a matrix and the right of the right hand side tensor into another matrix, and multiplying them as such. The mostly used matrix multiplications, however, involve simply tensors of rank 3 of shape (b, n, m) being multiplied to variable tensors (which are mostly of rank 1 with shape m), thus producing b matrix multiplications of (n, m) matrices with $(m, 1)$ matrices.

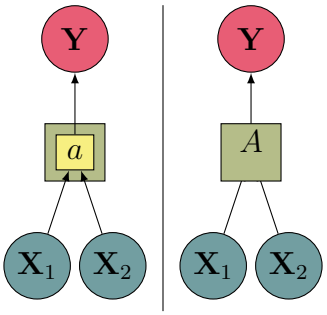
3.4 Convolutional Neural Networks and Parameter Sharing

In the dicussion of batching in Subsection 3.3 we saw a glimpse of a phenomenon that is quite common in neural architectures – parameter sharing. We discussed how the application of SGD to a batch of b points could be seen as the instance of b copies of a

model M, with the variables of each instance tied to each other, so that the update of a variable in one of the instances would be reflected across all others. This is one of the main principles that are applied in Convolutional Neural Networks (CNNs).

In convolution neural networks, a *kernel* is learned that is applied to parts of a input multiple times, by striding (or convolving) the kernel through the input. We can see the analogy between the kernels used in deep learning and the ones in mathematics if we think of some of the input Tensor's ranks as a discrete n-dimensional space through which we will perform a discrete convolution.

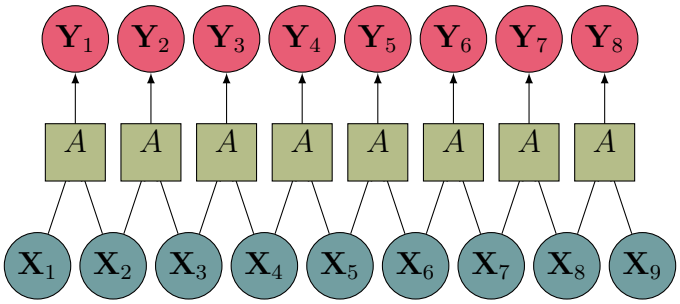
Figure 3.1: A small convolutional kernel A with activaction a that receives two inputs and produces one output. On the left the simple internal structure of the kernel is shown, containing only a single fully connected layer, on the right one can see the kernel as a module A. Yellow squares are neural network layers, blue circles are inputs, red circles are outputs and green backgrounds are to represent the whole neural network block. Source: Author, based on (OLAH, 2014)



Thus, we say that a 1-dimensional convolutional kernel convolves through the discrete 1-dimensional space defined by one of the tensor's ranks, a 2-dimensional convolutional kernel convolves through the discrete 2-dimensional space defined by two of the tensor's ranks, and so on and so forth. Through this convolution, however, we use the same weights over each of the points that are being applied, applying *parameter sharing* throughout the convolutions on the n-dimensional space, as if instantiating the same network with tied weights throughout each of the network's application to a slice of data. An example of a 1-dimensional convolutional kernel that receives 2 data-points can be seen in Figure 3.1 (note how it is simply a fully connected neural network layer), and the application of this kernel to a discrete 1-dimensional space consisting of points $x_i, 1 \leq i \leq 9$ can be seen in Figure 3.2. Note that the same kernel, with its parameters shared, is used throughout the discrete 1-dimensional space.

It is also common that the kernel operates on data which has a number of *features*, in a certain way performing a *filter* on those features, which must be stored in one of the tensor's ranks, and producing a feature of itself. So we can see that a k-dimensional

Figure 3.2: A discrete 1-dimensional space whose points are $x_i, 1 \leq i \leq 9$ having a convolutional kernel A applied to its input multiple times, producing a discrete convolution of the input values into the output values shown. Note how the convolutional kernel is simply a *module* who is repeated many time with shared parameters, and how some inputs are used multiple times in different input positions of the kernel. Yellow squares are neural network layers, blue circles are inputs, red circles are outputs and green backgrounds are to represent the whole neural network block. Source: Author, based on (OLAH, 2014)



convolutional kernel with dimensions $d_1, d_2, \ldots d_k$, which receives f_{in} features as input, is simply a neuron that receives $f_{in} \cdot \prod_{i=1}^{k} d_i$ inputs – thus we can see that a convolutional layer that produces f_{out} features is simply a layer of f_{out} neurons that are applied to every $d_1, d_2, \ldots d_k$ slice of an input tensor.

The way that it is organised is that the convolutional layers act on the last $k + 1$ ranks of a tensor, being the last rank the f_{in} features that the layer will work on and the other k ranks are the discrete k-dimensional space in which we will convolve the kernel through. All of the values in the $d_1, d_2, \ldots d_k, f_{in}$ slice can be flattened and fed as input to the neural network layer, thus producing f_{out} new features for every time the convolution is applied. The outputs of the convolutional layer keep the k-dimensional structure between themselves, and produce f_{out} feature maps, which in themselves are rank k (with the added ranks of the tensor which are not used and considered as batches), only with a reduced number of dimensions in each rank.

The application of such a network to large n-dimensional spaces can be costly, however, and some techniques are used to avoid the large number of operations that need to be done. One way to reduce the number of convolutional operations done is to convolve with a *stride*, which simply means that some points in the n-dimensional space are going to be skipped, and not all slices are going to be used. Another is to perform *pooling* in the output, from which the most common is max-pooling, where one takes, a slice of the feature map and combines these points into a single point, with only the largest value being taken into consideration.

Now, with a Convolutional Neural Network layer defined, we can see that one can

apply this to many different areas where one would benefit from invariance throughout some dimension. Some examples are audio files, where one can use 1-d convolutions to produce features to be processed (ABDEL-HAMID et al., 2012; ABDEL-HAMID; DENG; YU, 2013; ABDEL-HAMID et al., 2014), images, which are by far the most prominent use of convolutional neural networks and has produced state-of-the-art models in many computer vision competitions (KRIZHEVSKY; SUTSKEVER; HINTON, 2012; SIMONYAN; ZISSERMAN, 2014; LI et al., 2015). The main take-out here, however, is that applying parameter sharing over a certain data-structure can provide networks which use less parameters and present better performance than assuming that "everything is connected to everything". This is the main idea behind this chapter's final object of study: Graph Neural Networks.

3.5 Recurrent Neural Networks

Before we can build upon Graph Neural Networks from the idea of parameter-sharing over a data-structure, we must first discuss Recurrent Neural Networks – both because they, as will be clear throughout this section, use the same principle of parameter-sharing as well as because they can be used as a building block in Graph Neural Networks.

Recurrent Neural Networks are a subset of the common neural architectures in which the network can be used multiple times and, each time it is used, it will receive either its output, or its "internal state", or both the internal state and its output, produced on the last use. This is a form of parameter sharing, much like CNNs, in which the parameters are shared throughout the input sequence, and can be seen as a module that learns a *fold* or *map* operation. In this section we will adopt that h^t means the output of the module at timestep t, x^t is its input at the same timestep and $a()$ is the activation function used, when non-specified in some way.

One of the simplest RNN architectures simply takes its last output as part of its input. Other architectures that are commonly used are the Long Short-Term Memory (LSTM) (HOCHREITER; SCHMIDHUBER, 1997) and Gated Recurrent Unit (GRU) (CHO et al., 2014), along with their variants. They both provide a small "memory unit" for the RNN to store information in, alongside operations for controlling deleting from and writing to these memories. There are many different variants and different proposed architectures, but we focus on these due to their ubiquity in Deep Learning Practice and Research.

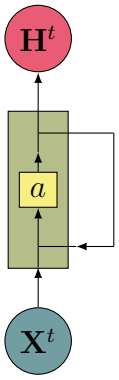
3.5.1 Simple Recurrent Neural Network

One of the simplest forms of RNN is one that receives as input both its previous output and its input, with its first input being that of the normal input and, commonly, a zero-tensor as its "previous output". Look at Fig. 3.3 for a graphical visualisation of this module and at Equation 3.4 below for it in mathematical notation[5]:

$$\mathbf{H}^t = a(concat(\mathbf{H}^{t-1}, \mathbf{X}^t) \times \mathbf{W} + \mathbf{B}) \tag{3.4}$$

However, with this simplicity, although proved to be Turing-complete (SIEGELMANN; SONTAG, 1991), this neural module has difficulties learning due to the exploding/vanishing gradient problem[6], and thus it is difficult to make one learn properties with long time dependencies (BENGIO; SIMARD; FRASCONI, 1994). As such, we will use it here only as an example on how Recurrent Neural Networks are trained.

Figure 3.3: The diagram of a simple recursive neural network with a layer whose activation is a that receives an input tensor $\mathbf{X}^t$ at time-step t, creating its output $\mathbf{H}^t$ who is then fed back on the network on the next time-step. If two lines joins there is a concatenation of both tensors, where the lines separate two copies of the same tensor are produces one for dividing arrow. Yellow squares are neural network layers, blue circles are inputs, red circles are outputs and green backgrounds are to represent the whole neural network block. Source: Author, based on (OLAH, 2015)



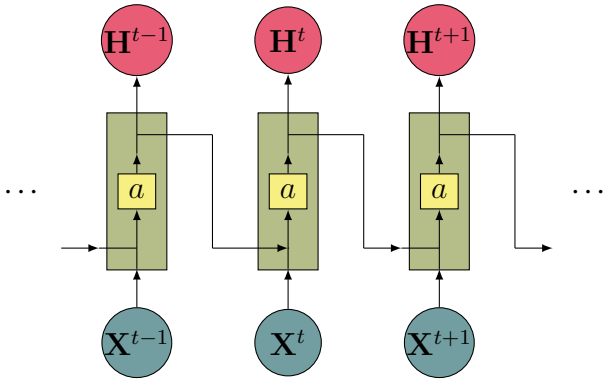
The recurrent part of the network can be seen as a deep network in which the same weights are used for the stacked layers. This is known as *unfolding* the network over time, and in this way backpropagation can be applied as normal, this being called *backpropagation through time*. As an example, look at Fig. 3.4 where one can see how the

[5] note that $concat$ is the concatenation between two tensors on an axis, here done on the last axis, which is the one that holds the output of the neurons in the RNN

[6] Due to the application of the chain rule repeatedly on the same function, if the derivatives tend towards smaller/bigger values, the gradients then will be vanishingly small/explodingly big, preventing the network from learning any meaningful information

layers, stacked along the input sequence, provide a deep neural network through which we can backpropagate the error function and update the weights jointly as if in a batch update.

Figure 3.4: The diagram elucidating the unrolling of a simple recursive neural network for backpropagation through time. This RNN has a layer whose activation is a that receives an input tensor $\mathbf{X}^t$ at time-step t, creating its output $\mathbf{H}^t$ who is then fed back on the network on the next time-step. If two lines joins there is a concatenation of both tensors, where the lines separate two copies of the same tensor are produces one for dividing arrow. Yellow squares are neural network layers, blue circles are inputs, red circles are outputs and green backgrounds are to represent the whole neural network block. Source: Author, based on (OLAH, 2015)



3.5.2 Long Short-Term Memory

The LSTM cell (HOCHREITER; SCHMIDHUBER, 1997) has, alongside its output $\mathbf{H}^t$ and input $\mathbf{X}^t$, a internal state $\mathbf{C}^t$, which, theoretically, can only be operated by the the recurrent cell itself. A simple LSTM cell is controlled by the following equations[7]:

$$\begin{aligned}
\mathbf{X}'^t &= concat(\mathbf{H}^{t-1}, \mathbf{X}^t) \\
\mathbf{F}^t &= \sigma(\mathbf{X}'^t \times \mathbf{W_f} + \mathbf{B_f}) \\
\mathbf{I}^t &= \sigma(\mathbf{X}'^t \times \mathbf{W_i} + \mathbf{B_i}) \\
\mathbf{G}^t &= a(\mathbf{X}'^t \times \mathbf{W_w} + \mathbf{B_w}) \\
\mathbf{C}^t &= (\mathbf{C}^{t-1} \cdot \mathbf{F}^t) + (\mathbf{I}^t \cdot \mathbf{G}^t) \\
\mathbf{O}^t &= \sigma(\mathbf{X}'^t \times \mathbf{W_o} + \mathbf{B_o}) \\
\mathbf{H}^t &= a(\mathbf{C}^t) \cdot \mathbf{O}^t
\end{aligned} \tag{3.5}$$

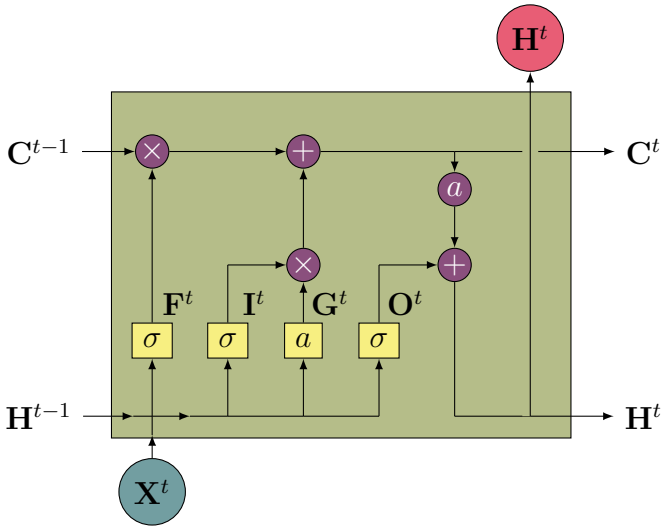
One can see from the onset that the LSTM cell has 4 different sets of weights and biases, which it uses to different purposes throughout its operation. These correspond

[7] σ is the sigmoid function

to 4 operations that the LSTM does in order to update its internal state and produce its output and the sigmoid operations serve to control the values to be considered ($\sigma(v) = 1$) or ignored ($\sigma(v) = 0$) completely, or anything in-between. These gates can be seen as the following:

- The $\mathbf{F}$ gate is the "forget" gate, where the LSTM picks what it will delete from its internal state, given the current input and its previous output.
- The $\mathbf{I}$ gate is the "input" gate, where the LSTM picks what part of the input it will write to its memory.
- The $\mathbf{G}$ gate is the "transform" gate, where the LSTM transforms the input before writing it to its memory.
- The $\mathbf{O}$ gate is the "output" gate, where the LSTM decides which part of its hidden state it will output.

Figure 3.5: The diagram of a LSTM neural network with a layer whose activation is a that receives an input tensor $\mathbf{X}^t$ at time-step t, creating its output $\mathbf{H}^t$ and its internal state $\mathbf{C}^t$ who is then fed back on the network on the next time-step. If two lines joins there is a concatenation of both tensors, where the lines separate two copies of the same tensor are produces one for dividing arrow. Yellow squares are neural network layers, blue circles are inputs, red circles are outputs, green backgrounds are to represent the whole neural network block and purple circles are point-wise operations. Source: Author, based on (OLAH, 2015)



A graphical visualisation of a LSTM cell can be see in Fig. 3.5, for an easier understanding of its internal connections. It can be seen that, if it so desires, the LSTM cell can completely wipe its internal memory, completely ignore the input, or anything

in-between, and thus it may be able to learn long time dependencies by keeping this channel freer from external noise, allowing the gradients to flow through them more easily.

3.5.3 Gated Recurrent Unit

A GRU cell (CHO et al., 2014) takes a approach that is similar to the LSTM, albeit more minimalistic. It makes the assumption that if something is going to be written into the internal memory something can be erased from it as well, and thus combines the forget and input gates into an update gate. It also merges the cell state and its output, instead of having a separate internal state. The formulas for the GRU cell are as following:

$$\begin{aligned} \mathbf{U}^t &= \sigma(concat(\mathbf{X}^t, \mathbf{H}^{t-1}) \times \mathbf{W_u} + \mathbf{B_u}) \\ \mathbf{R}^t &= \sigma(concat(\mathbf{X}^t, \mathbf{H}^{t-1}) \times \mathbf{W_r} + \mathbf{B_r}) \\ \mathbf{N}^t &= a(concat(\mathbf{R}^t \cdot \mathbf{X}^t, \mathbf{H}^{t-1}) \times \mathbf{W_n} + \mathbf{B_n}) \\ \mathbf{h}^t &= (\mathbf{1} - \mathbf{U}^t) \cdot \mathbf{H}^{t-1} + \mathbf{U}^t \cdot \mathbf{N}^t \end{aligned} \tag{3.6}$$

Here the cell has three sets of different weights and biases, which has been seen as its main advantage against a LSTM, given that some studies showed that, overall, they don't have a significant difference in performance (CHUNG et al., 2014). The three gates can be seen as the following:

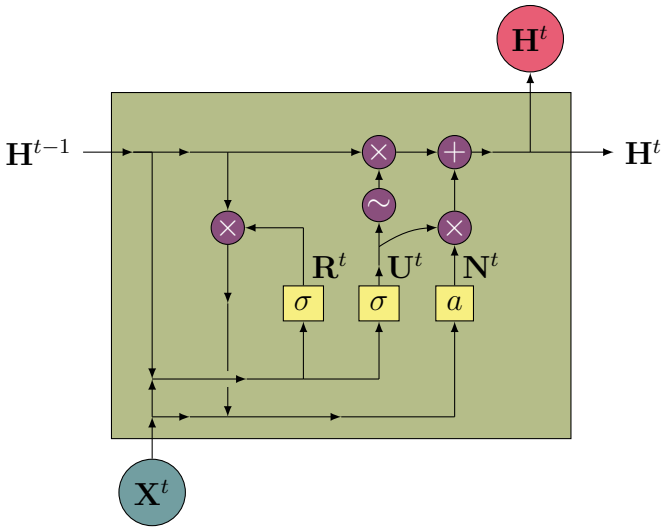
- The $\mathbf{U}$ gate is the "update" gate, where the GRU decides how much of its internal memory it will maintain and how much will be taken from the new input.

- The $\mathbf{R}$ gate is the "reset" gate, where the GRU decides to reset some part of its internal memory before using it as part of its new input.

- The $\mathbf{N}$ gate is the "new" gate, where the GRU performs the activation and generates the new input to its masked memory and input that will be written alongside the existing content, controlled by the update gate.

A graphical visualisation of a GRU cell can be see in Fig. 3.6, for a easier understanding of its internal connections.

Figure 3.6: The diagram of a GRU neural network with a layer whose activation is a that receives an input tensor $\mathbf{X}^t$ at time-step t, creating its output $\mathbf{H}^t$, which also is its internal state, who is then fed back on the network on the next time-step. If two lines joins there is a concatenation of both tensors, where the lines separate two copies of the same tensor are produces one for dividing arrow. Yellow squares are neural network layers, blue circles are inputs, red circles are outputs, green backgrounds are to represent the whole neural network block and purple circles are point-wise operations. The tilde $(\sim)$ operator is an unary set complement on the probabilities outputted by $\mathbf{U}^t$, and therefore is $(\mathbf{1} - \mathbf{U}^t)$. Source: Author, based on (OLAH, 2015)



3.6 Graph Neural Networks

Expanding on the reuse of weights alongside a discrete n-dimensional space, such as with Convolutional Neural Networks, or along a sequence, such as with Recursive Neural Networks, (GORI; MONFARDINI; SCARSELLI, 2005; SCARSELLI et al., 2009) proposed a model in which one can reuse weights using graph-structures to make an Artificial Neural Network learn to solve problems in graphs. This is a very powerful idea, since it opens a plethora of problems that were previously difficult to map to a Neural Network to work on, without breaking its end-to-end differentiability. The idea of Graph Neural Networks was then worked into many different formats, taking into some different intuitions on how it should be worked. In (GILMER et al., 2017) they are seen as Message-Passing Neural Networks, in (DUVENAUD et al., 2015; KEARNES et al., 2016) they are seen as Graph Convolutions and in (BATTAGLIA et al., 2018) it was even suggested that one could expand this idea not only to neural networks, but to any machine learning algorithm, giving it the name Graph Networks. Here we will look at the main idea behind this kind of neural network, alongside different intuitions to it.

The first obvious viewpoint available to explain a GNN is that of the original papers

(GORI; MONFARDINI; SCARSELLI, 2005; SCARSELLI et al., 2009). In these papers a graph structure would be reproduced by having is a state for every vertex, that contains that vertex's information, and local transition functions where the vertex's state is updated depending on itself, the state of its neighbourhood and the differing types of relations between the vertex and its neighbours. Note also that in the (SCARSELLI et al., 2009) formalisation it is explicitly defined that different types (kinds, as named in the paper) of vertices there are different transition functions.

(BATTAGLIA et al., 2018) present a more direct formalisation in which the Graph Neural Network, called Graph Network in their paper, restrained the model to have *vertices*, *edges* and *graph* states, and made no further points to different types of each entity. This model, which translates the language of graphs directly into itself, considers that one could try to reason about properties of the vertices, being vertex-centric (or node-centric, depending on the nomenclature being used), about the relations between the vertices, being edge-centric, or about the whole system in general. It provides reductions of some models alongside it but presents no further results.

The main idea underlying BATTAGLIA et al.'s paper, however, is that it raises the importance of parameter-sharing and notes the property of what they call Relational Inductive Bias. With this, it exposes the advances made by parameter sharing between time-steps or indices of a sequence on RNNs and between areas of an image on CNNs, and then poses the same idea to graphs. In this view, Graph Neural Networks are simply a generalization of parameter sharing to graphs, which capture the intrinsic relational inductive bias of the input's topology and uses it, alongside end-to-end differentiable systems with stochastic gradient descent to provide a tool that learns to classify information in graphs.

However, as noted before, in (BATTAGLIA et al., 2018) there is a small restriction on different entity types, which make the case that the model would not translate naturally for, say, hypergraphs, since its edges allow only communication between pairs of vertices. In this sense, (SCARSELLI et al., 2009) is even more general when one considers that every one of the three entity types cited (vertex, edge and graph) could be simply reduced to a different type (or kind, as referred in SCARSELLI et al.'s article) of vertex.

This extension works well for Hypergraphs and also for many other types of relational structures. In the reduction of a hypergraph to a graph, every hyperedge is simply transformed into a "hyperedge" vertex, and the sources and targets are themselves appropriately connected to said vertex as either pointing to it or being pointed by it. Other

problems such as Boolean Satisfiability and Graph Coloring, for example, are naturally represented with this definition.

A boolean satisfiability (SAT) problem[8] has two types of vertices – namely literals (or variables) and clauses. Clauses are related to many different literals, or they define a relation between a set of literals, and can thus be seen as a hyperedge between literals, whereas the problem itself is a relation between clauses.

A graph coloring problem may be seen as having two types of vertices: vertex-vertices and color-vertices. The color vertices are very like (BATTAGLIA et al., 2018) graph states, only that there are more than one color. In fact, one may assume that there are k colors for a k-coloring problem. With this, the vertex-vertices must decide whether they are colorable by analysing their neighbours and choosing with which global "color" state they would be colored. Note that this allows one to limit the available number of colors for specific nodes as well, setting more constraints than a simple graph coloring problem.

3.6.1 Typed Graph Networks

Thus explained, one can now define a simpler[9] definition of a GNN, which will be called Typed Graph Networks (TGN), that exposes the simplicity and the power of thinking about Graph Neural Networks through vertex-types, bringing back the original GNN definition with types of nodes discussed in (SCARSELLI et al., 2009), with a more modern terminology aligned with (GILMER et al., 2017; SELSAM et al., 2018). In fact, we will see the Graph Network model (BATTAGLIA et al., 2018) as being a GNN with three types of vertices, each of them representing a different type of entity: one type of vertex represents vertices proper; one type of vertex represents edges; and the last type of vertex represents the graph-level operations. However, we won't make generalisations to fit our model in domains other than Deep Learning and Neural Networks (i.e. we won't define it to work with other machine learning algorithms like in (BATTAGLIA et al., 2018)) and will leave out this more general version as one can easily extend it from this formalisation.

In a Typed Graph Neural Network, we have different types of entities, all of which will be considered vertices in a graph, with each type of entity reflecting similarly in a different type of vertex. Every vertex is then initialised with an *initial embedding* that represents any initial information we may have about a vertex – for example, the position,

[8] We assume that the problem is in Conjunctive Normal Form for simplicity and without loss of generality

[9] Than the one presented in (BATTAGLIA et al., 2018)

velocity, acceleration, mass, etc, of a particle in a physical system. Each vertex then, computes its *update function* on such an embedding, which updates the state of such vertex to contain this information, and outputs whatever information it may deem relevant to be processed and sent to its neighbouring vertices. Every vertex, then, will have its outputted embedding *translated* into a message for every type of vertex it connects to[10]. Then, every vertex will collect all the messages it may have received and aggregate them by vertex type using an aggregating function, these messages are then appended to form the input of to the vertex proper, which will be repeatedly applying this update function, sending messages along its neighbours and collecting and aggregating messages into its input in the next time step. This procedure can be executed throughout many time-steps, with parameter sharing along the time-steps using an RNN or can also be done only once, where its output can be fed to other blocks in the Deep Learning Model.

Note, however, that the previous explanation expected that the model firstly produced its update. We can assume, without any loss of generality, that the input embedding we received is in fact the first generated output to the update function, and proceed to apply the message-passing procedure first, having the update function be on the last part of the TGN block. One can look at Figure 3.7 for a pictorial representation of a vertex's view in a multi-timestep version of a Typed Graph Network model.

Definition 1 (Typed Graph Network)**.** In the context of deep learning, a TGN with N vertex types that communicate between themselves through K edge types is:

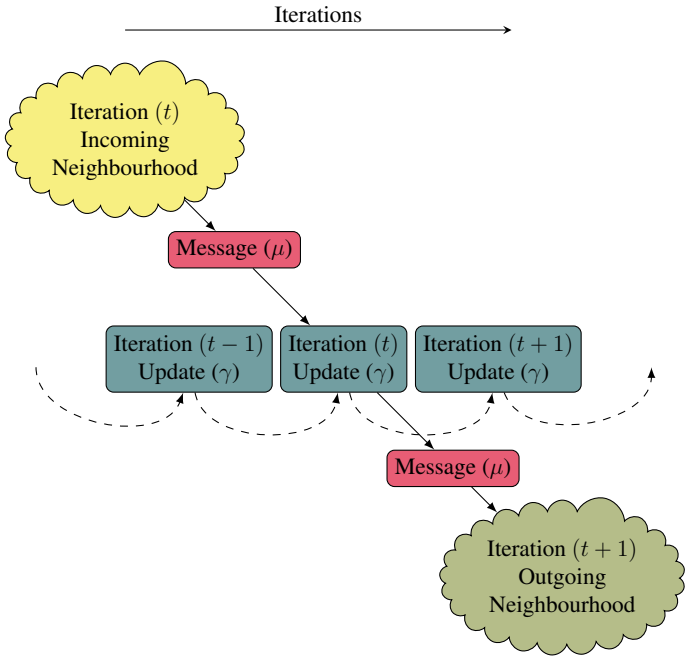
1. A set of **vertex types** $\mathcal{T}_v = \{\tau_i = \mathbb{R}^{d_i} \mid 1 \leq i \leq N\}$, which all have embedding dimension $d_{\tau_a} \forall \tau_a \in \mathcal{T}_v$. We define additional types $\notin \mathcal{T}_v$ for intermediate steps here.

2. A set of K **edge types** $\mathcal{T}_e = \{k \mid 1 \leq i \leq K\}$ whose edges define adjacencies between vertices of two vertex types. The set of vertex types that each edge type communicate is $\mathcal{P} = \{\pi_k = (s, t) \mid 1 \leq k \leq K\}$. A value $\pi_k = (s, t)$ set defines that the adjacencies of the edges of type k are between vertices of type τ_s to ones of type τ_t. Note here that there may be more than one edge type for the same pair of vertex types.

3. A set of K **message** functions $\mathcal{M} = \{\mu_k : \tau_i \to \tau_{\mu_k} \mid 1 \leq k \leq K, \tau_i \in \mathcal{T}_v\}$ which will take a set of vertices of type τ_i and a set of adjacencies $\mathcal{E}_k$ between those vertices and produce a set of messages $\overline{\mu}_k$ each of type τ_{μ_k}.

[10]There can be multiple message functions for the same pair of vertex types, for sake of simplicity we omit this from this explanation and keep it to the final model only

Figure 3.7: Pictorial representation of a Typed Graph Network from the perspective of a vertex v. A set of embeddings is received from vertices in its incoming neighbourhood, a message is computed from each embedding with the message function and messages are aggregated and fed to the update function, which produces an updated embedding for v. Simultaneously, v sends messages to vertices in its outgoing neighbourhood, which will undergo the same update process. Source: Author



4. A set of K **message aggregating** functions $\mathcal{A} = \{\alpha_k : \{\tau_{\mu_k}\} \to \tau_{\alpha_k} \mid 1 \leq k \leq K\}$ which will take a set of messages $\overline{\mu}_k$ produced by μ_k and aggregate them into message aggregates $\overline{\alpha}_k$ of type τ_{α_k}.

5. A set of N **input** functions $\mathcal{R} = \{\rho_i : \{\tau_{\alpha_k} \mid 1 \leq k \leq K, \pi_k = (s, i)\} \to \tau_{\rho_i} \mid 1 \leq i \leq N\}$ which takes message aggregates $\overline{\alpha}_k$ produced by α_k and aggregate them into update inputs $\overline{\rho}_i$ with elements of type τ_{ρ_i}

6. And a set of **update** functions $\mathcal{U} = \{\gamma i : \tau_i \times \tau_{\rho_i} \to \tau_i \mid \forall \tau_i \in \mathcal{T}_v\}$

Where the message and update functions are the sole trainable components of the model.

The idea here in Definition 1 again is that all vertices of a certain type will send information about themselves to their neighbouring vertices through message functions which will be aggregated into message aggregates, which will be again aggregated to be finally used by the update functions as information to update the vertex's embedding. An inversion of this path can also be used to facilitate seeing the need for the intermediate functions: A vertex has to update its embedding through the information it receives from its neighbourhood, this information may come from many different types of vertices,

which themselves may have many vertices that are adjacent to the vertex in question. Therefore, the vertex aggregates all the messages from every type of vertex separately and then aggregates these messages again to form the final information it may have. In our definition, we say that only the message and update functions are learnable parameters, but one may also wish to learn the aggregation functions. We simply not consider this here for the sake of simplicity.

As an example of different message aggregating functions and input functions, imagine that there are three types of relational data that one wants to propagate, through separate channels, in the network. The first type of information is sum-invariant, the second one is product-invariant, and the third one only makes sense as an average of the values. In this case, one could define three different edge types, each of them representing a different communication channel, each with its specific learned messaging function and with the desired message aggregation function on each of them. The messaging functions of each should, then, be expected to learn to pass along the information that pertains to each channel through the correct channel, since this would minimise the loss more rapidly. Then, the input function only serves to prepare these aggregates into an input for the network, generally being a simple concatenation of the different message aggregates, but there could be a case where it would make the most sense to join the aggregates through another operation, such as a sum, or difference.

These learnable functions can be any neural module one wants. But one fruitful way of thinking about them is that the message functions are MLP blocks, which are simply conversion functions, taking an input embedding and filtering any relevant information that could be sent throughout this adjacency, or maybe a type casting function that converts the internal embedding representation to one interpretable to vertices of other types. The update function then is an RNN module which updates its embedding and keeps internal information about itself throughout the message-passing process, as if it was a small neural computer that has an internal memory and can only communicate with the outside environment through message passing. These neural computers are then arranged in the network structure of a graph, with many types of modules which all behave through the same learned algorithm.

With this ensemble of vertices which communicate with each other through learned message and update functions the network propagates information about the graph structure and the property of the vertices in a way that satisfies the requirements after the TGN block. That is, if one wishes to learn about a specific structure in a network, the TGN block

will tune itself to provide that information, only propagating the information it may find relevant. Thus, a TGN can be simplified into Equations 3.7, 3.8, 3.9, 3.10, 3.11.

$$K_i = \{k \mid \forall i, \pi_k = (s, i)\} \tag{3.7}$$

Equation 3.7 defines the subset of edge types that target a certain vertex type i. With this, Equation 3.8 defines the subset of messages from a type k that target a certain vertex b.

$$\overline{\mu}_{k,b}^{(t)} = \{\mu_k(\mathbf{V}_{\mathbf{s}_{(a)}}^{(t-1)}) \mid \forall v_a \in \mathcal{V}_s, (v_a, v_b) \in \mathcal{E}_k\} \tag{3.8}$$

Then Equation 3.9 aggregates these messages to be processed for input in the update function of that vertex type.

$$\overline{\alpha}_{k,b}^{(t)} = \alpha_k(\overline{\mu}_k^{(t)}) \mid 1 \leq k \leq K \tag{3.9}$$

Equation 3.10 uses the aggregates produced as in Equation 3.9 that target vertex b and aggregates them for the final use in the update function.

$$\overline{\rho}_{i,b}^{(t)} = \rho_i(\overline{\alpha}_{k,b}^{(t)}) \quad \forall 1 \leq i \leq N, v_b \in \mathcal{V}_i \tag{3.10}$$

Finally, Equation 3.11 shows how every vertex embedding in the tensor of vertex embeddings is updated for vertex type i, by update the embeddings for every vertex v_b.

$$\mathbf{V}_{\mathbf{i}_{(b)}}^{(t)} = \gamma(\mathbf{V}_{\mathbf{i}_{(b)}}^{(t-1)}, \overline{\rho}_i^{(t)}) \tag{3.11}$$

One can look at Algorithm 1 for an example of how one may implement this model. See that the TGN works by receiving only the initial embeddings for the vertices and the graph structure, with the vertices partitioned into the N types and the edges into K edge types defining as many modes of communication between two vertex types as desired.

3.6.2 A Simpler TGN Formalisation

The formalisation in Subsection 3.6.1 is an attempt to build upon the formalisation proposed in (PRATES et al., 2019b), which adds more freedom to the model, and while the such formalisation may be quite daunting, we see it in its original version as Definition 2,

Algorithm 1 Typed Graph Network Model

// The input for a TGN is a graph whose vertices are partitioned into N **vertex types**, and the edges into K **edge types**, as well as an initial embedding for every vertex
procedure TGN($\mathcal{G} = (\mathcal{V} = \bigcup_{i=1}^{N} \mathcal{V}_i, \mathcal{E} = \bigcup_{k=1}^{K} \mathcal{E}_k), \mathcal{I} = \bigcup_{i=1}^{N} \mathbf{V_i}^{(0)}$)
    // Run for t_{max} message-passing iterations
    for $t = 1 \dots t_{max}$ **do**
        // For every vertex type
        for $i = 1 \dots N$ **do**
            // Find every edge type that targets that vertex type
            Let $K_i \leftarrow \{k \mid \forall k, \pi_k = (s, i)\}$
            // For every vertex of that type
            for all $v_b \in \mathcal{V}_i$ **do**
                for all $k \in K_i$ **do**
                    // Find every message that is targeting that vertex
                    $\overline{\mu}_{k,b}^{(t)} \leftarrow \{\mu_k(\mathbf{V_s}_{(a)}^{(t-1)}) \mid \forall v_a \in \mathcal{V}_s, (v_a, v_b) \in \mathcal{E}_k\}$
                    // Aggregate all messages from all edge types that target that vertex
                    $\overline{\alpha}_{k,v_t}^{(t)} \leftarrow \alpha_k(\overline{\mu}_{k,b}^{(t)})$
                end for
                // Combine the aggregated messages into the input format
                $\overline{\rho}_{i,b}^{(t)} = \rho_i(\{\overline{\alpha}_{k,b}^{(t)} \mid \forall k \in K_i\})$
                // Update the vertex embedding
                $\mathbf{V_i}_{(b)}^{(t)} \leftarrow \gamma_i(\mathbf{V_i}^{(t)}, \overline{\rho}_{i,b}^{(t)})$
            end for
        end for
    end for
    return $\{\mathbf{V_i}^{(t_{max})} \mid i = 1 \dots N\}$
end procedure

which assumes that the aggregated messages are concatenated. Also look at Algorithm 2 for the algorithm of this definition, where the aggregating operator for messages is fixed as a simple point-wise sum. This aggregation method was chosen both due to the fast computation of matrix multiplications in GPUs as well as due to the fact that the network may tune the representation on its vertices' embeddings to work with the point-wise sum as an aggregation method, even if it is at a loss of performance. The input functions are all simple concatenations of the aggregated messages, and this is done simply to keep the communication channels of each type separate from each other.

Definition 2 (Typed Graph Network)**.** In the context of deep learning, the simplified version of a TGN with with N types that communicate between themselves through K messaging functions can be described as:

1. A set of **types** $\mathcal{T} = \{\tau_i = \mathbb{R}^{d_i} \mid 1 \leq i \leq N\}$, which all have embedding dimension $d_\tau \forall \tau \in \mathcal{T}$.

Algorithm 2 Original Typed Graph Network Model

1: // The input for a TGN is a graph whose vertices are partitioned into N **vertex types**, and the edges into K **edge types**
2: **procedure** TGN($\mathcal{G} = (\mathcal{V} = \bigcup_{i=1}^{N} \mathcal{V}_i, \mathcal{E} = \bigcup_{k=1}^{K} \mathcal{E}_k), \mathcal{I} = \bigcup_{i=1}^{N} \mathbf{V_i}^{(0)}$)
3: **for** $k = 1 \dots K$ **do**
4: // Compute an adjacency matrix between types $\#i$ and $\#j$ with 1s where an edge exists
5: $\mathbf{M_k}[a, b] = \mathbb{1}\{(v_a, v_b) \in \mathcal{E}_k \mid v_a \in \tau_a, v_b \in \tau_b, \mu_k : \tau_a \to \tau_b\}$
6: **end for**
7: // Run for t_{max} message-passing iterations
8: **for** $t = 1 \dots t_{max}$ **do**
9: **for** $i = 1 \dots N$ **do**
10: **for** $\mu_k \in \mathcal{M} \mid \mu_k : \tau_i \to \tau_j$ **do**
11: $\overline{\mu}_k \leftarrow \mathbf{M_k} \times \mu_k(\mathbf{V}_j^{(t-1)})$
12: **end for**
13: $\mathbf{V_i}^{(t)} \leftarrow \gamma_i(\mathbf{V_i}^{(t-1)}, \{\overline{\mu}_k \mid \mu_k \in \mathcal{M}, \mu_k : \tau_i \to \tau_j\})$
14: **end for**
15: **end for**
16: **return** $\{\mathbf{V_i}^{(t_{max})} \mid i = 1 \dots N\}$
17: **end procedure**

2. A set of K **message** functions $\mathcal{M} = \{\mu_k : \tau_1 \to \tau_2 \mid 1 \leq k \leq K, \tau_1, \tau_2 \in \mathcal{T}\}$. Note that one may define more than one message function for a pair of vertices.

3. And a set of **update** functions $\mathcal{U} = \{\gamma_{\tau_i} : \mathbb{R}^{d_{\tau_i} + \sum d_{\tau_i} \forall \mu : \tau_j \to \tau_i \in \mathcal{M}} \to \mathbb{R}^{d_\tau} \mid \forall \tau_i \in \mathcal{T}\}$

Where the message functions $\tau_i \to \tau_j$ and the update functions γ_τ are the sole trainable components of the model.

This formalisation allows us to take away with the relative cumbersome definitions required for both the message aggregation functions as well as the input functions, but comes at the cost of flexibility. Another simplification to note here is that the messages all take from τ_1 to τ_2, which is also done for simplicity, with the rationale that these message functions are in fact "type-casting" the embeddings of a type to another, and thus the embedding dimensionality a message function's output is the same than the target vertex's embedding dimensionality.

As one can see, the model builds adjacency matrices for all messages[11] and then proceeds to run for a number of message-passing iterations (Line 8), for every vertex of every vertex type (Line 9) it aggregates the messages sent to it through all the messages (Line 10) that target it by making the matrix multiplication between the adjacency matrix of that messaging function and the vertices of the type that are sending the message. With

[11] The adjacencies defined may be redundant in the final programming model, but for generality we assume that they might be different

this description of the simpler TGN formalisation we can see that it can be simplified into Equations 3.12 and 3.13.

$$\overline{\mu}_k^{(t)} = \mathbf{M_k} \times \mu_k(\mathbf{V_j}_{(a)}^{(t-1)}) \mid \forall \mu_k \in \mathcal{M} | \mu_k : \tau_i \rightarrow \tau_j \tag{3.12}$$

Equation 3.12 shows how the matrix multiplication of every state tensor with the adjacency matrix produces the aggregated messages for every vertex being targeted and Equation 3.13 shows that the update function is then simply calculated through the update on the last embeddings and all the aggregated messages that target that type of node.

$$\mathbf{V_i}^{(t)} = \gamma_i(\mathbf{V_i}^{(t-1)}, \{\overline{\mu}_k \mid \mu_k \in \mathcal{M}, \mu_k : \tau_i \rightarrow \tau_j\}) \tag{3.13}$$

These two equations show how much is simplified under this more constrained formalisation, which relies on assuming that the embedding representation learned by the TGN will be able to deal with the sum aggregation of the embeddings even if that is not the optimal aggregation. This has as a payoff the fact that the matrix multiplication can be done very efficiently in its code implementation, and provides a more concise and elegant solution to the full definition.

3.6.3 Discussion

With this formalisation of GNNs one can set many of the literature's models into a simple framework for working with relational data. The first example is that we can have the Graph Network (BATTAGLIA et al., 2018) framework simplified as a TGN with three types of vertices: A *Vertex* type, an *Edge* type and a *Graph* type. We specify the connections between these as follows: Vertices will communicate with the Edges that they have[12]. Each edge will communicate with the two vertices they are connected with[13]. Finally, there will be a single vertex of the graph type, which communicates with all edges-vertices and vertex-vertices.

By means of this reduction it is shown as well that the TGN formalisation extends and generalises all other models cited and shown to be generalised in BATTAGLIA et al.'s model. These include, but are not limited to, Independent Recurrent Blocks (SANCHEZ-GONZALEZ et al., 2018), Message Passing Neural Networks (GILMER et al., 2017),

[12] If one needs to specify in-edges and out-edges this can be simply done by making two messages for each type as well.

[13] This communication can also take place with an in-edge and out-edge definition

Non-Local Neural Networks (WANG et al., 2018), Relational Networks (RAPOSO et al., 2017; SANTORO et al., 2017) and Deep Sets (ZAHEER et al., 2017).

All of the ideas here shown can also be framed into multiple viewpoints, such as the Message-Passing viewpoint, in which every vertex type is considered as a type of stateful computer in a communication networks, which then transfers and receives messages in parallel along its adjacencies (GILMER et al., 2017); and the Graph-Convolutional viewpoint, in which we are operating a convolutional kernel through a graph structure, then feeding the produced features on to the next layer (DUVENAUD et al., 2015; KEARNES et al., 2016) (in fact one can see Graph Convolutional Networks as generalisations of convolutions on images, being that every pixel in an image is adjacent to the ones directly adjacent to it).

Nonetheless, power of applying Deep Learning models on graphs is astounding – and the main takeaway from these models is the idea that neural networks can be reused by mirroring a certain data-structure and produce approximations to values on these structures. Graph Neural Networks have produced significant advances in many areas, such as molecular fingerprinting (DUVENAUD et al., 2015; KEARNES et al., 2016), generating new molecules (YOU et al., 2018a; LI et al., 2018), language modelling (SANTORO et al., 2017), learning approximations to NP-Complete problems (SELSAM et al., 2018; PRATES et al., 2019a) and many others. Thus it is reasonable to assume that work on deep learning models that work on the underlying structure of a graph can be used in a myriad of applications and will be of great use in the near future.

4 LEARNING CENTRALITY MEASURES WITH GRAPH NEURAL NETWORKS

In this chapter we present the the main results of the research carried out during the Master's programme, relevant to this dissertation's theme. This will be done in three sections: Section 4.1 explains the datasets used for the experiments, Section 4.2 explains the experiments done to approximate centrality values using GNNs and Section 4.3 explains the experiments done to rank vertices according to a centrality (in both these sections we also discuss the experiments done to evaluate the transfer between centralities by learning a joint embedding for every centrality type), and Section 4.4 analyses visually the representations learned by some of the models.

4.1 Datasets

In this section we describe the details of the datasets used in the experiments described in the following sections. For the experiments regarding this dissertation we prepared six datasets to be used across all experiments, these are "train", "test", "large", "sizes", "different", "real". Of these train, test, large and sizes are all synthetic datasets sampled from random graph models and were all generated with the Python NetworkX package (HAGBERG; SWART; CHULT, 2008). The real dataset consists of real instances.

Table 4.1: Training instances generation parameters. Source: Author.

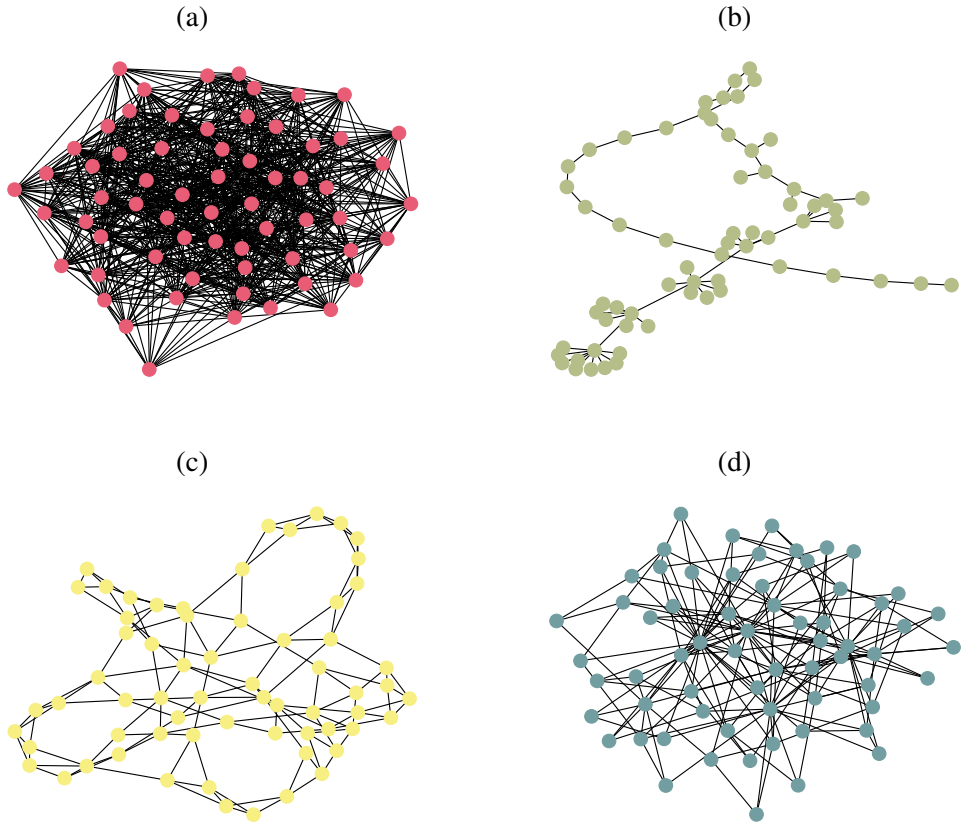
Graph Distribution	Parameters
Erdős-Rényi	$p = 0.25$
Random power law tree	$\gamma = 3$
Watts-Strogatz	$k = 4, p = 0.25$
Holme-Kim	$m = 4, p = 0.1$

The "train" dataset was created by producing 4096 graphs between 32 and 128 vertices for each of the four following random graph distributions (total 16384): 1) Erdős-Rényi (BATAGELJ; BRANDES, 2005), 2) Random power law tree[1], 3) Connected Watts-Strogatz small-world model (WATTS; STROGATZ, 1998), 4) Holme-Kim model (HOLME; KIM, 2002). Further details about the random distributions used are reported in Table 4.1. Examples sampled from each distribution are shown in Figure 4.1.

The "large" was created in a similar fashion as the "train" dataset, only that the sizes of the graphs were larger – from 128 to 256 vertices – and less graphs were created – only

[1]This refers to a tree with a power law degree distribution specified by the parameter γ

Figure 4.1: Examples of training instances with $n = 64$ vertices for each graph distribution: Erdős-Rényi in Subfigure (a) coloured red, Random power law tree in Subfigure (b) coloured green, Watts-Strogatz in Subfigure (c) coloured yellow and Holme-Kim in Subfigure (d) coloured blue. Source: Author, using Networkx (HAGBERG; SWART; CHULT, 2008)



64×4 graphs – with instances sampled from the same distributions shown in Table 4.1. The "test" dataset was also created much like the "train" dataset, only being comprised of newly sampled instances.

The "sizes" is a set of datasets which are also comprised of instances samples from the distributions in Table 4.1, but with the specific sizes on a range from 32 to 256 with strides of 16 to allow us to analyse the generalisation to larger instance sizes and how the performance of the models vary with the graph's size, beginning in the range with which the model was trained up until the size of the largest instances of the "large" dataset.

The "different" dataset is comprised of instances from two previously unseen random distributions, and consists of 256 graphs sampled from the Barabási-Albert model (ALBERT; BARABÁSI, 2002) and 256 shell graphs[2] (SETHURAMAN; DHAVAMANI,

[2]The shell graphs used here were generated with the number of points on each shell proportional to the "radius" of that shell. I.E., $n_i \approx \pi \times i$ with n_i being the number of vertices in the i-th shell.

2000), all of which have between 32 and 128 vertices.

The "real" dataset consists of real instances obtained from either the Network Repository (ROSSI; AHMED, 2015) or from the Stanford Large Network Dataset Collection (LESKOVEC; KREVL, 2014), which were *power-eris1176*, a power grid network, *econ-mahindas*, an economic network, *socfb-haverford76* and *ego-Facebook*, Facebook networks, *bio-SC-GT*, a biological network, and *ca-GrQc*, a scientific collaboration network. The statistics of each of these graphs is further presented in Table 4.2 where one can easily see that these networks significantly surpass the size range of the "train" dataset, which will be used for training, overestimating from $\times 9$ to $\times 31$ the size of the largest ($n = 128$) networks which will be seen during training, while also pertaining to entirely different graph distributions than those described in Table 4.1. To see the difference in network structure between these networks, compare Figure 2.1 which contains pictures of the *bio-SC-GT*, *ca-GrQc*, *econ-mahindas* and *socfb-haverford76* networks and the example networks from the training datasets in Figure 4.1.

Table 4.2: Statistics for the real instances and their source, where NR stands for (ROSSI; AHMED, 2015) and SN for (LESKOVEC; KREVL, 2014) Source: Author with data from (ROSSI; AHMED, 2015; LESKOVEC; KREVL, 2014).

Name	Source	Vertices	Edges	Degree		
				Maximum	Average	Minimum
power-eris1176	NR	1174	9861	100	16.8	2
econ-mahindas	NR	1258	7619	206	12.1	2
socfb-haverford76	NR	1446	59590	374	82.4	1
ego-Facebook	SN	4036	88243	1044	43.7	1
bio-SC-GT	NR	1708	33982	549	39.8	1
ca-GrQc	SN	4158	13428	81	6.46	1

Each and every one of the graphs in the datasets mentioned had the centrality measure values for each vertex calculated, having both a normalised and a not-normalised version of the Betweenness, Closeness and Degree centralities as well as the Eigenvector centrality. All centralities were calculated using the algorithms available on the Networkx package (HAGBERG; SWART; CHULT, 2008), with the centralities that needed de-normalisation either de-normalised through a parameter to the function call or de-normalised outside the function call. The graphs were then stored as json files and read at a later date, being sampled randomly from the graphs avaialable on every dataset when loaded for training or testing.

4.2 Learning to Approximate Centrality Measures

One of the first approaches for learning centrality measures is to approximate their values, normalised or not, directly. In this section we aim to show how we devised the GNN model and show the results obtained through training, as well as discuss any issues that arose during the experimental phase.

4.2.1 Centrality measure normalisation and numeric issues

As discussed in Chapter 2 we have, for some centrality measures, both normalised and non-normalised versions. Now it is necessary to ponder whether one wishes to approximate the metric normalised or not and provide a rationale for the choices made.

One of the first and foremost issues is a numeric one. Neural networks work best on a defined range of values and do not generalise well to values on unexpected ranges (BENGIO, 2012; IOFFE; SZEGEDY, 2015; BA; KIROS; HINTON, 2016; BENGIO; LODI; PROUVOST, 2018). This poses a strong contender for using the normalised versions of the centralities. However the most common way to normalise these centralities is to divide them by a number polynomially dependant on the number of vertices in the graph. This is a most salient problem when normalising Degrees for real networks, due to their power-law nature, which makes it so that the centrality values for small degrees tend to be really small if binned polynomially as suggested.

Another problem, related to the numeric one, that may arise is that centralities with different scales for their values may end up making the network have to work with many different ranges of values, which may degrade performance. We show that this seems to be somewhat true although the way our models will be built can possibly overcome such problem due to the separation between the modules that calculate the predictions for each of the metrics proper.

With these arguments, one may prefer to work with normalised values, however there is also the problem that, due to how the GNN models operate, each vertex will have to somehow gather information on the size of the network itself to cope with these normalisations. We propose, however, to normalise the output values of the network in a similar way to the centralities themselves depending on every problem's number of vertices, so that the model will not need to worry about the normalisation by itself, but will be able to benefit from the increased numerical stability of the normalised values.

We will see, however, that none of the results are as appealing as they could be, which lead us to trying to rank the vertices with respect to the measures instead of learning the numerical value of the centrality measures themselves, something we experiment upon on 4.3.

4.2.2 The model

All of the models used had the same GNN module in them, which is represents a directed graph[3], with two different MLP message functions, one for an edge's source and another for its target both with 4 layers with 64 neurons each, and ReLU activations for all but the last layer. The message passing is done with both using the matrix and the source messaging function as well as the transposed matrix and the target messaging function. One can look at the Code Snippet 1 for the code of the GNN block used, as per the GNN library definition (See Appendix A).

Code Snippet 1 The GNN block used for all the centrality experiments, where d is the dimensionality chosen for the embeddings. Source: Author

```
gnn = GNN{
    {
        "N": d
    },
    {
        "M": ("N","N")
    },
    {
        "Nsource": ("N","N"),
        "Ntarget": ("N","N")
    },
    {
        "N": [
            {
                "mat": "M",
                "var": "N",
                "msg": "Nsource"
            },
            {
                "mat": "M",
                "transpose?": True,
                "var": "N",
                "msg": "Nsource"
            }
        ]
    },
    name = "centrality"
}
```

For all of the models used the initial embedding for every vertex was the same and

[3]Note that we only work with undirected graphs. The use of "directed" edges here was simply to allow more "bandwidth" for the message passing, as well as allowing for one to use the same model with directed graphs in future work.

it was a learnable d dimensional array, where d is the dimensionality we choose for the vertices' embeddings. This d dimensional array is broadcasted for every vertex in the input graphs/batch.

For such method we considered three different setups,

With that in mind, we have three models which we consider: In one we learn the normalised centrality measures directly, which we name *AN* (for "Approximate the Normalised centrality"), in the second one we learn the unnormalised version of the centrality measures, which we call *AU* (for "Approximate the Unnormalised centrality"), and a third approach was to learn from the normalised centrality values, but perform a normalisation of the model's approximated value before using it as its final output, and dub it *AM* (for "Approximate the normalised centrality, with normalisation on the Model"). These models are the three discussed possible models in the last section and all of them have a similar pipeline following the GNN. In fact, models AU and AN both have the exact same pipeline, which is simply a MLP for every centrality the model has to predict, with 2 layers with d neurons and ReLU non-linearities and 1 layer with a single output neuron with no non-linearity applied to it. The AM pipeline is almost the same as the other two, with the only modification being that, for every problem in the batch, the output of the output MLP is normalised depending on the centrality being calculated – being divided by n if the centrality in question is degree and by $\frac{(n-1)\cdot(n-2)}{2}$ if it was the betweenness centrality, with n being the number of vertices in that graph.

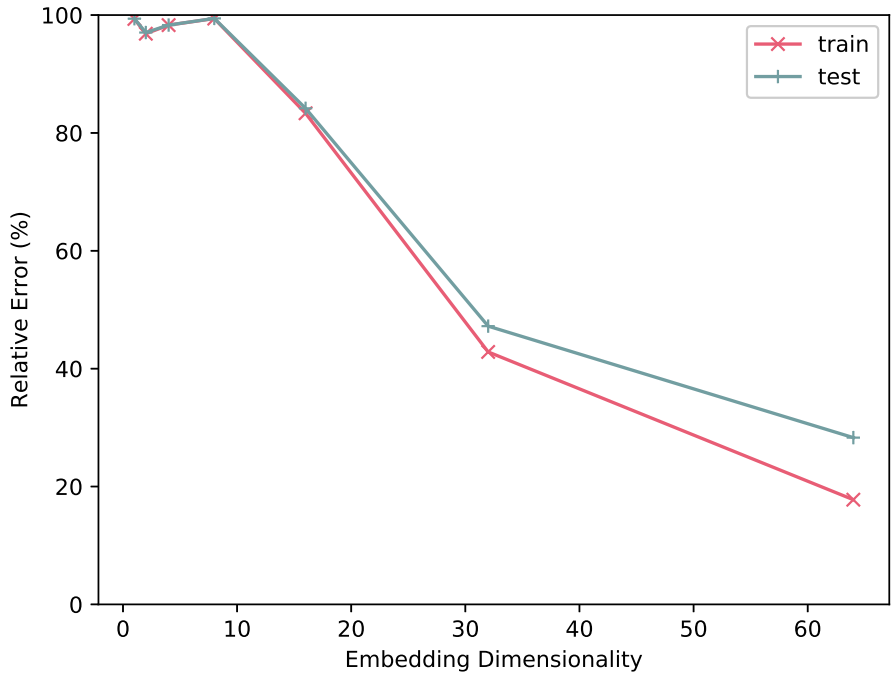
The loss function that was optimised by these models was the *mean squared error* of all of the model's output with the label, being that model AU received the centrality measure without any normalisation as label and models AN and AM received the normalised centrality as their label. The performance metric we are interested in is the *relative error* between the predicted value and the label, being normalised by the label's value, except in the case of the eigenvector centrality, which we are interested in the *absolute error*, due to numerical problems with the aforementioned relative error.

All the models were trained in 32 epochs, each consisting of 32 batches of 32 training instances each, through Stochastic Gradient Descent to minimise the mean squared error loss and a L2 normalisation loss on the parameters. More specifically, the Adam (KINGMA; BA, 2014) implementation was used.

4.2.3 Fitting the dimensionality of the model

All of our model's parameters are dependant on the dimensionality d of the vertices' embeddings. For choosing this value d, embedding dimension sizes of $d = 2^k$ were tested with $1 \leq k \leq 6$, being that $d = 128$ was avoided due to being considered too big (the Neurosat model uses $d = 128$ to solve a NP-Complete problem (SELSAM et al., 2018)) and powers of two were used to ease performance optimisations on the GPU operations. The values we settled for the dimensionality were $d = 2^6 = 64$ for all models, which was surprising due to the fact that the performance had an increase up until such point. We tested these values for both a model predicting only the degree centrality measure as well as one predicting all 4 at the same time, with the loss being computed as the average of the losses for each centrality.

Figure 4.2: Plot of the Relative Error for the degree centrality on the "train" (in red) and "large" datasets (in blue) for an increasing embedding dimensionality d for the AM model, showing how the error continued to decrease as the dimensionality of the embedding increased, allowing for the embeddings to hold more information at the cost of computational power required. Source: Author



One can see a plot of the relative error of the degree centrality for a non-multitasking AM model in relation with the embedding size in Figure 4.2, where we can see that the error decreases as the embedding size for the vertices increase, possibly due to the ease of accumulating more information given by the larger embedding sizes. The number of

parameters in every model is the same, due to the similarities in the pipeline, and the increase in the number of parameters seemed to be polynomial with the increase of the dimensionality, seemingly following a quadratic equation. A graph with the increase in the number of parameters due to the increase in the embedding size can be seen in Figure 4.8 in Subsection 4.3.3 being the same as that section's AM and AM models[4]. Note that the multitask model has roughly twice the amount of parameters, even though it produces the results of equivalent to 4 non-multitasking models. The training times were not altered significantly due to the manual computing of the metrics outside of the GPU, but there was a slight increase of the runtime.

We can see that some models behave quite differently. Comparison between models show that although the Eigenvector centrality fluctuates, every model's pipeline actually is the same for this centrality, and thus we do not take it into account. The Closeness centrality has only the difference that the normalised ones use a slightly improved definition (WASSERMAN; FAUST, 1994) that takes in account the connected component that the vertex belongs to, and is also not as relevant to the analysis. These two values fluctuate, then, mostly due to the interference caused by the numerical differences in the labels and value while multitasking and to differences due to the stochastic processes involved. Thus, one is to take note mostly of the difference between the performance when training on the "train" dataset and the performance on the larger "large" dataset, to account for the generalisation to larger graphs, which we want to achieve.

Seeing this, we can look at Table 4.3 and see that the model AM has the most desirable performance of all of them while multitasking, being between the AU model's better generalisation to larger instances as well as being the best model in the performance in three of the 4 metrics for the last training epoch as well as having the best overall performance on the "large" dataset. With this in mind we then decided to train it on the 4 proposed centrality measures individually, as well as the multitask model on all 4 of them.

4.2.4 Results

Having defined the dimensionality of each vertex's embedding, the model was trained anew for each of the centrality measures individually and for all of them in tandem

[4]The results are presented there instead of here so that we can compare the size of the two models used on that section. The AM and AN models are presented as "RC" in the plot since they have the same number of parameters.

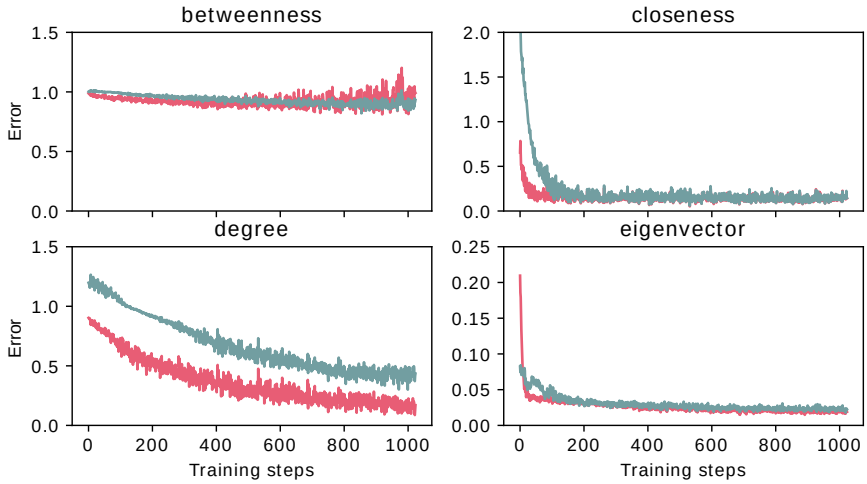
Table 4.3: The Relative Errors of the multitask learning performance for the Betweenness, Closeness and Degree centrality measures and the Absolute Error for the Eigenvector centrality, for the proposed models trained with $d = 64$ on a sample of the "train" dataset, on the full "large" dataset as well as these proportion between these two values. For every line we **bold** the best value. Source: Author

		AU	AN	AM
"train"	Betweenness	92.7%	119%	**89.6%**
	Closeness	77.8%	16.3%	**15.3%**
	Degree	55.5%	**38.9%**	43.6%
	Eigenvector	0.0438	0.0251	**0.0230**
"large"	Betweenness	**91.7%**	419%	94.2%
	Closeness	274%	85.9%	**75.0%**
	Degree	58.9%	210%	**50.6%**
	Eigenvector	0.0569	**0.0518**	0.0734
$\frac{\text{"large"}}{\text{"train"}}$	Betweenness	**0.989**	3.52	1.05
	Closeness	**3.52**	5.26	4.90
	Degree	**1.06**	5.40	1.16
	Eigenvector	**1.3**	2.06	3.19

for the multitask learning analysis. One can see the loss curves for every centrality in Figure 4.3, where the loss for every centrality is separated, with the multitask model's loss being separated for the loss relative to every centrality. Then the model's final performance was evaluated on the entirety of the "test" and "large" datasets, the results of which one can see at Table 4.4. Note that the error for the Eigenvector centrality is the absolute one, instead of the relative error, this is due to the fact that the relative errors for the eigenvector centrality were enormously high, and even though numerical problems were guessed at first, the tested corrections didn't produce great results. One can see that the multitask model is outperformed by the non-multitask one quite frequently, and sometimes erring for more than twice the relative amount of the non-multitask model.

To see how the model fared to novel distributions, its performance was also tested on the "different" dataset, which can also be seen in Table 4.4, and while the errors were not as high as for the "large" dataset, they were higher than on the "test" dataset. This was, in a sense, expected, and it shows that the model somehow generalised its results to different distributions. Then, the model was tested on the real-world instances of the "real" dataset, which can be seen in Table 4.5 where one can see a significantly worsening of the results for all centrality measures, with the degree centrality measure being the least affected one. Interestingly, the multitask model was not outclassed in this test, retaining a better overall relative error on the dataset, while the relative error stayed close to the

Figure 4.3: Evolution of training error per batch for all four centrality metrics throughout the training process for the AM model. The error is plotted in red and blue for training without and with multitasking, respectively. The loss plot was similar, but in another scale, so we chose to omit it here. The error metric is the relative error for the betweenness, closeness and degree centrality and absolute error for the eigenvector centrality. Source: Author



non-multitask model.

To see how the model's performance was affected by the size of the input graph, it was tested with the "sizes" dataset, and the results for a multitasking model are summarised in Figure 4.4. In short, the multitask model's accuracy presents a seemingly linear, but expected, decay in performance with increasing problem sizes. This shows that there is some difficulty generalising to larger instance sizes, and reinforces the results presented in Tables 4.4 and 4.5. One can also see that the model performs better on the problems near the average size with which the model was trained, which is expected behaviour as well. The absolute error for the eigenvector centrality measure is shown on the right of Figure 4.4 with a similar, but more pronounced, behaviour as the overall relative error for the other centralities.

In Figure 4.5 one can also see the plots for the individual models for each centrality and note some of the results are similar to those in Figure 4.4, while showing that for some centralities the decay in the model's performance behaves differently than the aggregate shown in Figure 4.4.

The results obtained here are not satisfactory. Other machine learning models such as REPTrees and simpler neural models such as in (GRANDO; LAMB, 2015) already obtained better relative errors. However, the task here is significantly harder than the task presented in GRANDO; LAMB's paper, since here the model only has the network structure to work with. In their other works (GRANDO; LAMB, 2016; GRANDO;

Table 4.4: Loss (Mean Squared Error) and performance metrics (Relative error for Betweenness, Closeness and Degree, and Absolute error for Eigenvector) for the AM model during test on the "test", "large" and "different" datasets (without/with multitasking). The best values for a metric under a centrality are in **bold**. Source: Author

Error Type	Centrality	"test"	"large"	"different"
Relative (%)	Betweenness	95.96/**89.54**	**91.03**/94.17	99.94/**83.88**
	Closeness	13.49/**13.38**	79.00/**74.76**	**19.13**/21.35
	Degree	**16.75**/43.39	**27.88**/50.06	**25.84**/45.32
	Average	**42.07**/48.77	**65.97**/72.99	**48.30**/50.18
Absolute	Eigenvector	**0.01946**/0.02286	0.08311/**0.07214**	**0.04854**/0.05282
MSE	Betweenness	**0.01462**/0.01464	**0.01462**/0.01464	**0.001376**/0.001445
	Closeness	0.004785/**0.003710**	0.004785/**0.003710**	**0.008956**/0.01079
	Degree	**0.03465**/0.03705	**0.03465**/0.03705	**0.01026**/0.01758
	Eigenvector	0.01694/**0.008880**	0.01694/**0.008880**	**0.003974**/0.004940
	Average	0.01775/**0.01607**	0.01775/**0.01607**	**0.006142**/0.008689

Table 4.5: Accuracy performance for the AM model during test on the "real" dataset (without/with multitasking). The graph names are abbreviated to PE for power-eris1176, EM for econ-mahindas, SH for socfb-haverford76, SC for bio-SC-GT, GQ for Ca-GrQc and EGO for ego-facebook. The number of decimal digits was reduced so that the results fit on page. Source: Author

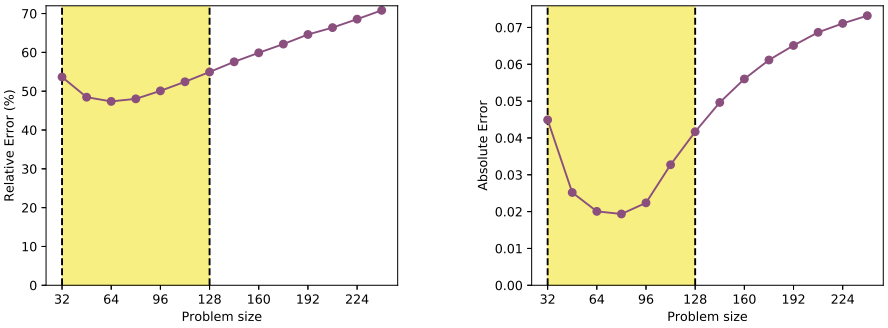
Centrality	Relative Error (%)						
	PE	EM	SH	SC	GQ	EGO	Average
Betweenness	123/**102**	**95**/96	147/**118**	562/**344**	148/**125**	428/**275**	250/**177**
Closeness	**325**/330	57/**54**	**24**/35	**41**/48	116/**107**	**100**/107	**110**/114
Degree	37/**36**	**22**/40	**79**/87	**53**/71	**17**/47	**57**/72	**44**/59
Average	162/**156**	**58**/63	83/**80**	219/**155**	93/**93**	195/**151**	135/**117**
	Absolute Error						
	PE	EM	SH	SC	GQ	EGO	Average
Eigenvector	0.083/ **0.0098**	**0.061**/ 0.10	0.21/ **0.17**	**0.097**/ 0.11	**0.074**/ 0.11	**0.12**/ 0.13	**0.11**/ 0.12

GRANVILLE; LAMB, 2018; GRANDO; LAMB, 2018) these results are more focused on ranking the centrality measure and on the correlation values than the relative error per-se, which provides some rationality behind the hardness of predicting these values.

In these works, however, the models were able to learn to predict both the betweenness and closeness centrality measures, from which the AM model was able to predict here only the closeness centrality measure, and with little success at generalising to larger instances. The incapability of predicting the betweenness centrality measure may be because the network does not have enough time to process the information required to provide an accurate measure. However, when looking at the Mean Squared Error loss for the values (in Table 4.4), one can see that the AM model performed better in minimising it, so maybe further pre-processing of the data could be used to achieve better results.

Figure 4.4: The overall relative error for the multitask model (Subfigure (a)) and the absolute error for the eigenvector centrality (Subfigure (b)). The overall relative error increases with increasing problem sizes, with a small valley centered near the average training instance size. There is a similar behaviour with the absolute error, with a more pronounce valley. The dotted lines delimit the range of problem sizes used to train the network ($n = 32 \ldots 128$). Source: Author

(a) (b)



Another unfortunate but surprising result was the high relative errors of the eigenvector centrality measure, which made it impossible to measure relative errors, even after some attempts at correcting the issue aimed at mitigating numerical problems with floating point arithmetic. The suprising part of this result is due to the fact that the procedure executed by Graph Neural Networks can be seen as similar to the procedure of the eigenvector centrality measure itself. Thus the model achieved a mediocre performance only on the degree centrality measure, which has no need for an approximate solution due to its low cost of computation. Nonetheless, on the next sections there will be a focus on *ranking* centrality measures instead of approximating them directly.

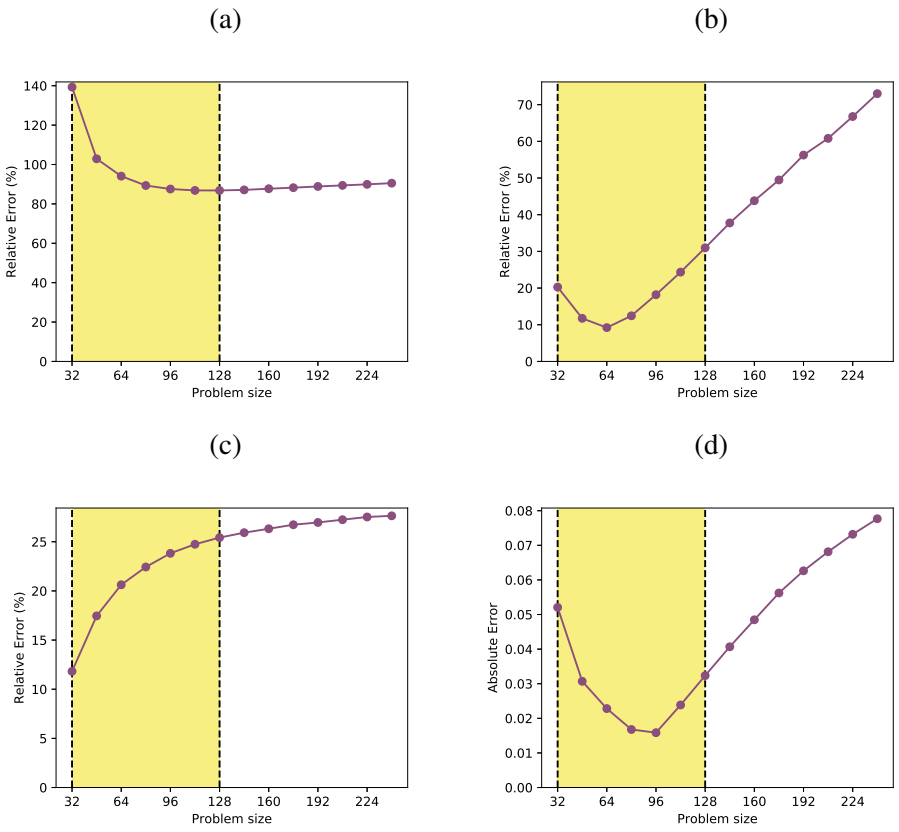
4.3 Ranking Centrality Measures

In this section we show the results we obtained while trying to *rank* centrality measures instead of predicting their values per se, as well as discuss any issues that arose during the experimental phase.

4.3.1 Ranking by Comparison

The main idea behind ranking is to compare the desired score of different entities and then sort them from the most desirable to the least desirable. Here this idea will be

Figure 4.5: The overall accuracy of the non-multitask models for problems with an increasing number of vertices with relative errors for the betweenness, closeness and degree centralities, and absolute error for the eigenvector centrality. The plot in Subfigure (a) is for the model that predicts the betweenness centrality, the one in Subfigure (b) for closeness, Subfigure (c) for degree and Subfigure (d) for eigenvector. The error seems to increase in a seemingly linear fashion for the closeness centrality and logarithmically for the degree centrality, with eigenvector behaving similarly to the closeness centrality. Surprisingly, the error for the betweenness centrality did not seem to increase significatively increasing problem sizes. The dotted lines delimit the range of problem sizes used to train the network ($n = 32 \ldots 128$). Source: Author



taken into the concept of a *comparison matrix*. In our case dealing with centrality measures, for each pair of vertices $(v_i, v_j) \in \mathcal{V}$, under the definition of a specific centrality measure, the comparison matrix M is indexed $M_{i,j}$ so that $M_{i,j} = 1$ if vertex i is more central than vertex j.

With such a matrix, we can define the Precision, Recall (also known True Positive Rate), True Negative Rate and Accuracy metrics on the model's predictions. Firstly, we say that the *Accuracy* of the model is simply the number of comparisons it happens to predict correctly, which can be seen as a more strict version of the Kendall-τ correlation coefficient (KENDALL, 1938), commonly used as a measure of the correspondence between two rankings, since the accuracy metric here penalises ties in the ranking as much as discordant

rankings. The same rationale can be followed for the other metrics: defining them as if the problem was a binary classification one. The results will be shown for all these four metrics, but the focus will be on accuracy for the selection of the working model.

This comparison matrix is calculated on the predicted centrality values that the trained network outputs, but one can also build a neural network classifier that answers the same question: given two representations of a vertex's centrality, which representation is the first representation higher than the second? With this we can build a fuzzy comparison matrix, which is composed of the probabilities that one vertex is more central than another, instead of containing only the true value. If the classifier is well trained, we can convert the probabilities to prediction by thresholding at 50% and use this to calculate the rankings instead. An example of such a fuzzy comparison matrix, which also elucidates the non-fuzzy case, can be see in Figure 4.6

Figure 4.6: Example of a fuzzy comparison matrix $\mathbf{M}_{\widetilde{\geq}_c}$ at the left with an upper triangular matrix $\mathbf{T}$ at the right, for a graph with three vertices $\mathcal{V} = \{v_1, v_2, v_3\}$ sorted in ascending centrality order as given by the centrality measure c. Source: Adapted from (AVELAR et al., 2019)

$$\begin{pmatrix} P(v_1 >_c v_1) & P(v_2 >_c v_1) & P(v_3 >_c v_1) \\ P(v_1 >_c v_2) & P(v_2 >_c v_2) & P(v_3 >_c v_2) \\ P(v_1 >_c v_3) & P(v_2 >_c v_3) & P(v_3 >_c v_3) \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

4.3.2 The models

Two of the models we use here are very similar to those in Subsection 4.2.2, with both the start of the pipeline and the GNN block being exactly the same. We experimented with three models in total, with one trained to perform rankings natively on the final embeddings produced by the GNN block, denoted as *RN* (for "Ranking centralities Natively"), and the other three models reusing their names from Subsection 4.2.2: *AN*, and *AM*[5]. These last two models are, in fact, almost the same as their versions from model from Subsection 4.2.2, with the main difference being that we also produce a comparison matrix with the Centrality measures it generates, which will then be used to compare this model with the one who learns to make comparisons natively on the vertices' embeddings.

Thus, the end of the pipeline for the RN model is different in that, its MLP takes

[5]We use both AN and AM since in preliminary tests, the ranking AM model performed poorly in some aspects, as will be seen below. We did run tests for AU model.

two embeddings as input, having then 3 layers with sizes $[2d, 2d, 1]$, ReLU nonlinearities for the first two layers and no nonlinearity for the last one. The value outputted by this MLP is interpreted as the *logit* of the probability[6] that the first vertex has a centrality larger than the second one. This operation is done by reshaping the final embedding tensor N of each problem with n vertices. We reshape N from shape (n, d) to $(1, n, d)$ which we'll call $N1$. Then we transpose $N1$ swapping its first and second rank so that the shape is $(n, 1, d)$, which we call $N2$. We then broadcast both of these tensors on the rank they have 1 as dimension so that they are both of shape (n, n, d) and then concatenate them both on the d rank, so that we have a tensor of shape $(n, n, 2d)$ which is the tensor of all possible pairs of embeddings, so that in the i-th position on the first rank and the j-th position on the second rank is the embedding of the i-th vertex in the first d dimensions and the embedding of the j-th vertex in the remaining dimensions. With such a tensor in our hands, we simply run the MLP over this tensor, producing a tensor with shape $(n, n, 1)$, which we shrink to (n, n) to produce the final output: The fuzzy comparison matrix with the logits of the aforementioned probabilities.

These logits are used to be trained with the label comparison matrix by calculating the sigmoid-cross entropy between the logits of every point in the matrix and the desired value. The probability for every vertex is used for calculating performance metrics, where we apply sigmoid to all the logits and then round the probability and take that as the predicted value. A description of the algorithm for the RN model is presented in Algorithm 3.

4.3.3 Fitting the dimensionality of the model

As was done in Subsection 4.2.3, we fitted the dimensionality of the vertices' embeddings d to choose the best value. Again the value which was settled for was $d = 64$ for both models, being that AN, and AM inherited this value due to their performance on the last section and RN had this value chosen due to the performance observed in the tests by rising the d value.

One can see a plot of the accuracy, relative to the dimensionality of the vertices' embeddings in the RN model, in Figure 4.7. The increase in the number of parameters in both models is in Figure 4.8. One can see that the RN model does not present a significant

[6]The logit function is the inverse of the sigmoidal function, and it maps a probability $\in [0, 1]$ to $(-\infty, +\infty)$

Algorithm 3 Graph Neural Network Centrality Ranking Predictor

procedure GNN-CENTRALITY($\mathcal{G} = (\mathcal{V}, \mathcal{E}), \mathcal{C}$)
  // Compute adj. matrix
  $\mathbf{M}[i, j] \leftarrow 1$ if $(v_i, v_j) \in \mathcal{E}$ else 0
  // Initialise all vertex embeddings with the learned initial embedding V_{init}
  $V^1[i, :] \leftarrow V_{init} \mid \forall v_i \in \mathcal{V}$
  // Run t_{max} message-passing iterations
  for $t = 1 \dots t_{max}$ **do**
    // Refine vertex embeddings with messages from incoming edges
    $\mathbf{V}^{t+1}, \mathbf{V}_h^{t+1} \leftarrow V_u(\mathbf{V}^t, \mathbf{M} \times \mathrm{src}_{\mathrm{msg}}(\mathbf{V}^t), \mathbf{M}^T \times \mathrm{tgt}_{\mathrm{msg}}(\mathbf{V}^t))$
  end for
  for $c \in \mathcal{C}$ **do**
    // Compute a fuzzy comparison matrix $M_{\gtrapprox_c} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$
    $\mathbf{M}_{\gtrapprox_\mathbf{c}}[i, j] \leftarrow \mathrm{cmp}_c(\mathbf{V}^{t_{max}}[i, :], \mathbf{V}^{t_{max}}[j, :]) \mid \forall\, v_i, v_j \in \mathcal{V}$
    // Compute a strict comparison matrix $M_{>_c} \in \{\top, \bot\}^{|\mathcal{V}| \times |\mathcal{V}|}$
    $\mathbf{M}_{>_\mathbf{c}} \leftarrow M_{\gtrapprox_c} > \frac{1}{2}$
  end for
end procedure

increase in the number of parameters in relation with the approximation-based models, which kept the number of parameters from the last section. One can also note that the number of parameters in the multitask model, again, is approximately twice the number of parameters of the non-multitasking model, therefore using only the multitasking model would reduce the total memory use by roughly 2 times than if one used the four separate models, one for each centrality. If more centralities are used, this memory saving can be even higher. Again, the training times were not altered significantly, but one could see a slight increase of the runtimes as d increased.

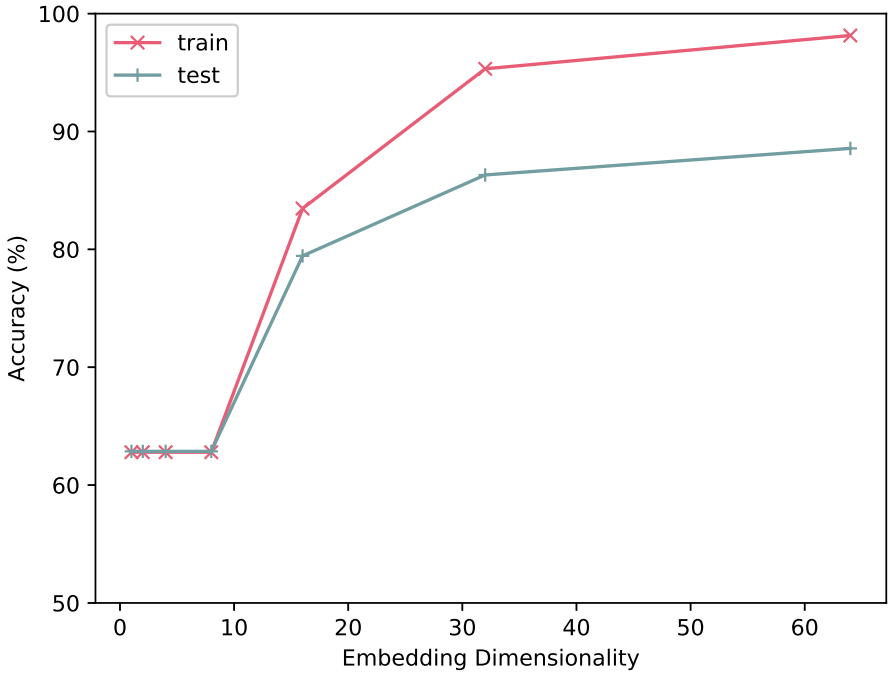
We can see that the RN model is comparatively heavier than the AM and AN models, but, as will be shown in the experimental results, its performance for ranking centrality measure is superior in some points.

4.3.4 Results

Having defined the dimensionality of each vertex's embedding, all of the models were trained anew for each of the centrality measures individually[7] as well as for all of them in tandem, for the multitask learning analysis. One can see the loss curves for every centrality in Figure 4.9 for the RN model and in Figure 4.10 for the AN model, and in Figure 4.11 for the AM model, where the loss for every centrality is separated, with the

[7] Please keep in mind that the non-multitasking AN and AM models are identical.

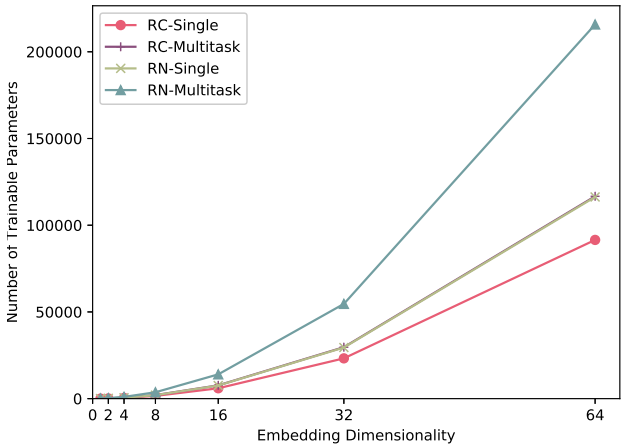
Figure 4.7: Plot of the Accuracy for the degree centrality on the “train” (in red) and “large” datasets (in blue) datasets for an increasing embedding dimensionality d for the RN model, showing how the accuracy continued to increase as the dimensionality of the embedding increased, allowing for the embeddings to hold more information at a larger computational cost. Source: Author



multitask model’s loss being separated for the loss relative to every centrality. Then the models’ final performances were evaluated on the entirety of the “test” dataset, the results of which one can see at Table 4.6 and reflect the models’ performances in unseen instances the same size and distribution as the ones in the training dataset.

Looking at Table 4.6 one can see that the native comparison method learned by the RN model outperforms the most of the rankings produced by the approximation-based models, sometimes by a wide margin. However, looking at Table 4.7 one can see that the RN model isn’t as dominant, suggesting that the approximation-based models extend their performance better for larger instances. The only failing is that the AM model seem to fail completely for the betweeness centrality, and its non-multitasking version fails is all but the eigenvector centrality, which is normalised by default and whose pipeline is equivalent to the AN model. One interesting synergy, however, is that the multitasking AM model seem to dominate in the closeness centrality, maybe showing a potential multitasking combination to be used in practice, with both the closeness and eigenvector centralities being the ones with a decent performance, whereas the AN model fails at the closeness centralities without the normalisation being made in the model.

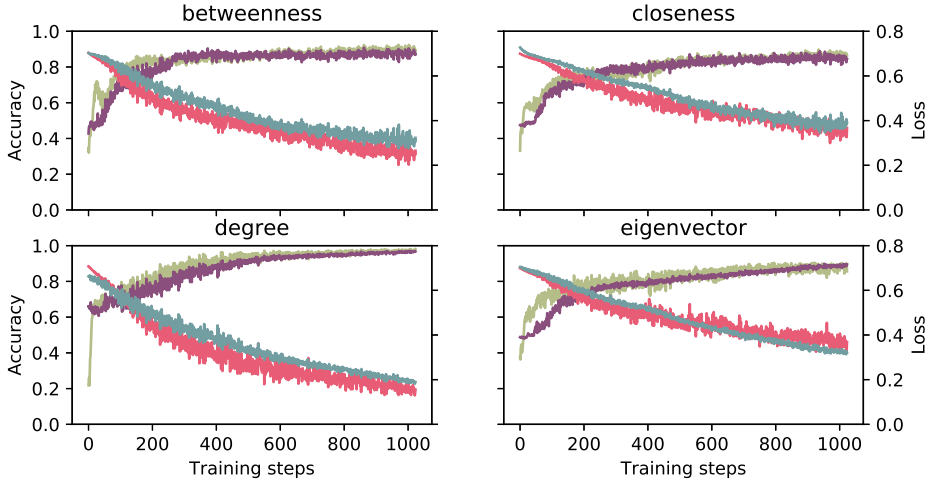
Figure 4.8: Number of parameters of an approximating model (RC here stands for either the AM or AN model) and a native comparison model (RN). The non-multitasking approximating model is shown in red and the multitasking one in purple, while the RN non-multitasking model is in green and the multitasking one is shown in blue. Note that the size of the multitasking models is less than twice the amount of the non-multitasking ones, and that the increase in the total number of parameters seem to follow a quadratic curve. The values for the approximating-multitask and RN non-multitask are very close and overlap. The larger size of the RN model is only due to the final MLP having to work with $2d$ as its input and hidden layer dimensions. Source: Author



In table 4.8, one can see the results that should reflect performance for the models with distributions with different topologies, being that the "different" dataset has two distributions not seen in training: one similar to a training one and one completely alien to the model. In this configuration one can see that a multitasking approach seems to alleviate the issues of graphs with different topologies, even though the multitasking is done in the centrality measures. However, the non-multitasking models are still slightly superior than the multitasking ones, with the multitasking AN model showing signgs of a slight overfit with regards to the eigenvector centrality, and the AM model showing surprisingly better response with regards to the degree centrality, since the average number of nodes in between this distribution and the training one does not change.

These results show that both ranking natively as well as ranking through an approximated centrality measure are feasible techniques, with each showing a particular strength in some settings. Ranking natively seems to favour distributions with a similar number of nodes as seen in training, a better overall performance, as well as a more robust response to multitasking (possibly since the centralities need not share their numeric properties within the embedding directly). Whereas ranking through approximated centralities seem to be more amenable to changes in the number of nodes in the graph, with the exception of the

Figure 4.9: Evolution of training loss and training accuracy per batch for all four centrality metrics throughout the training process for the RN model. The loss is plotted in red and blue and the accuracy in green and purple for training without and with multitasking, respectively. Source: Author



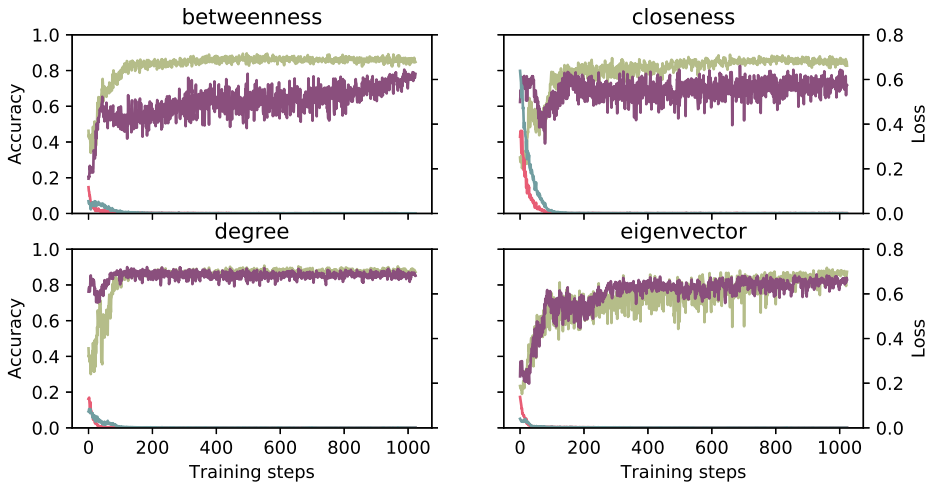
degree centrality in the AM model.

Although the multitasking model is outperformed by the basic model in many cases, the overall accuracy is not changed as much (see Tables 4.6, 4.7, and 4.8), especially for the RN model, with these models having roughly half the number of parameters when compared with having a separate model for each centrality. In this context, recall that the multitask learning model is required to develop a "lingua franca" of vertex embeddings from which information about any of the studied centralities can be easily extracted, so in a sense it is solving a harder problem.

To see how the model's performance is affected by the size of the input graph, it was tested with the "sizes" dataset, and the results are summarised in Figures 4.12, 4.14, and 4.16. In short, the multitask RN and AM models' accuracy presents an expected decay with increasing problem sizes, however, this decay is not a free fall towards 50% for instances with almost twice the size of the ones used to train the model, in fact the overall accuracy remains above 80% accuracy when $n = 240$ in the RN model, which reinforces that some level of generalisation may be achievable for larger instances.

The individual plots for every non-multitask model are shown in Figures 4.13, 4.15, and 4.17, where one can see that the RN models present a minor skewing in some specific problem sizes and that the points where the decay started/ended were somewhat different, but overall the curves behaved in a similar fashion, while the AM model shows a greater stability in degree and eigenvector centrality, also showing the worst overall performance in centralities other than eigenvector and a peculiar decay within the training interval for

Figure 4.10: Evolution of training loss and training accuracy per batch for all four centrality metrics throughout the training process for the AN model. The loss is plotted in red and blue and the accuracy in green and purple for training without and with multitasking, respectively. Remember that the losses here are different from the RN model. Source: Author

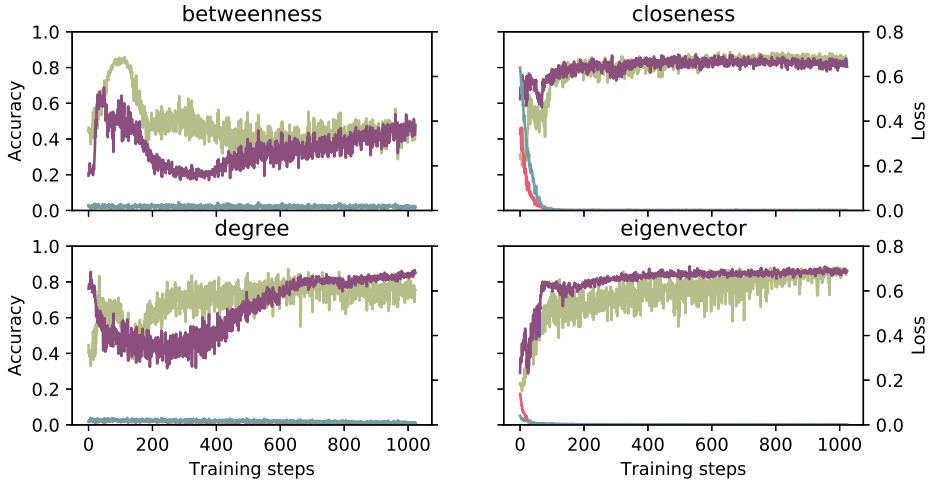


the degree centrality, but a more stable curve after that. The AN model, however, shows to be the most stable model overall, only worsening its performance for larger instances in the closeness centrality.

A final benchmark for the models was for them to be tested on the real-world instances of the "real" dataset, with the results being reported in Table 4.9. The RN model was able to obtain up to 86% accuracy (on degree) and 82% average accuracy on the best case (*bio-SC-GT*), and 58% accuracy (closeness) and 65% average accuracy on the worst case (*ego-Facebook*), while being the only model that did not go below baseline. The model with best accuracy overall, however, was the non-multistasking AN model, with 77% accuracy, closely followed by the non-multitasking and multitasking RN models, with 75% and 74% accuracy, showing a slight superiority for ranking with the normalised centrality measures, which had a heavier loss of performance while multitasking. Therefore, to make a greater use of the reduced number of parameters while multitasking, one should probably learn to rank natively on the embeddings.

Due to the fact that the largest networks significantly surpass the size range that the network has been trained on and may be from completely different graph distributions, it is impressive that the RN models can predict betweenness centrality with 75% and 75% accuracy (or 81% and 77% without multitasking) on graphs as large as *ca-GrQc* and *ego-facebook*, both with more than four thousand vertices and more than fourteen, on ca-Gr-Qc, or eighty eight thousand edges. It is also notable that one of the worst performances occur

Figure 4.11: Evolution of training loss and training accuracy per batch for all four centrality metrics throughout the training process for the AM model. The loss is plotted in red and blue and the accuracy in green and purple for training without and with multitasking, respectively. Remember that the losses here are different from the RN model. Source: Author



on the smallest real network (*power-eris1176*) – an overall accuracy just below 70% for both models. This perhaps can be explained in (HINES; BLUMSACK, 2008; HINES et al., 2010) who highlighted the significant topological differences between power grid networks and Erdős-Rényi and Watts-Strogatz small-world models (two of the models used to train the network). The performance of the AN models is also impressive, achieving 91% and 81% (83% and 95% while multitasking) on such large networks, really shining on the degree centrality, while also achieving decent results on the other centralities. However, the eigenvector centrality misperformed on these graphs, even with some of them being power-law in nature.

We tested if the results of the model generalised if we ran the GNN block for more message-passing iterations, however we found out that the performance fluctuated as one strayed from the number of message-passing iterations the model was trained with. This was unfortunate since it does not allow us to remedy the accuracy decay with larger sizes as does (SELSAM et al., 2018). Possible, but untested ways to circumvent this would be to train the model while randomising the number of timesteps it runs for, doing gradient descent on all intermediate results at the same time, this second technique showing promise for this in (PALM; PAQUET; WINTHER, 2017), whereas the first one can be seen as an approximation of the second, while being less resource-demanding.

Table 4.6: Performance metrics (Precision, Recall, True Negative rate, Accuracy) for the AN, AM and RN models during test on the "test" dataset referring to the predictions of the comparison matrix. Please note that the differences between the AN and AM models without multitasking on the Eigenvector centrality are only due to the stochastic nature of the training, being that the pipeline is the same in both models for this centrality. The best values for a metric under a centrality are in **bold**. Source: Author

	AN				AM				RN			
Centrality	P	R	TN	Acc	P	R	TN	Acc	P	R	TN	Acc
Without multitasking												
Betweenness	82.4	**88.6**	84.6	86.0	39.9	45.6	43.5	44.1	**90.3**	88.5	**91.0**	**89.8**
Closeness	83.1	**85.4**	84.3	84.8	83.1	**85.4**	84.4	84.9	**88.4**	84.5	**89.8**	**87.3**
Degree	75.3	**98.4**	81.6	87.5	63.3	85.8	70.4	75.6	**99.3**	94.9	**99.4**	**97.6**
Eigenvector	87.0	87.0	87.6	87.3	86.5	86.5	87.0	86.8	86.2	**90.2**	82.3	86.3
Average	82.0	**89.8**	84.5	86.4	68.2	75.8	71.3	72.8	**91.0**	89.5	90.6	**90.3**
With multitasking												
Betweenness	73.6	79.6	76.1	77.3	41.9	46.7	46.3	46.2	87.2	87.2	88.9	87.9
Closeness	71.4	73.0	73.6	73.3	80.2	82.5	81.7	82.1	86.9	82.0	88.7	85.5
Degree	73.7	96.0	80.4	85.9	72.4	94.3	79.3	84.7	98.3	92.4	99.0	96.4
Eigenvector	83.5	83.5	84.2	83.9	85.5	85.5	86.2	85.9	**89.8**	88.3	**90.4**	**89.4**
Average	75.5	83.0	78.6	80.1	70.0	77.3	73.4	74.7	90.5	87.5	**91.8**	89.8

4.4 Analysing the Internal Representations Learned by the Network

Machine Learning algorithms have achieved impressive feats in the past decade, but the interpretability of the computation that takes place within Deep Learning models is still limited (BREIMAN et al., 2001; LIPTON, 2016; LIPTON, 2018). SELSAM et al. (2018) have shown that one can extract useful information from the embeddings learned by a model, even in hard problems as boolean satisfiability. They used this information to extract solutions even though their model was trained exclusively to produce a binary answer of SAT/UNSAT.

We can have insights about what possible algorithm it might have learned in the same fashion as (SELSAM et al., 2018) by projecting the $\mathcal{R}^d$ vertex embeddings onto a one-dimensional space by the means of Principal Component Analysis (PCA) (JOLLIFFE, 2011) and plotting such projections throughout the message-passing iterations. Here, we analyse this behaviour for Section 4.3's RN model. In the results obtained from the trained model it was observable that in *some* instances the projection presented a strong fit with the logarithm of the centrality measure, and it seemed that the embeddings were being used to sort the vertices by their centrality measures as can be seen in Figure 4.18. On some cases, however, even though the PCA seemed to provide a good visual fit, the model had performed poorly, such as in Figure 4.19. Some of theses cases can be traced due to

Table 4.7: Performance metrics (Precision, Recall, True Negative rate, Accuracy) for the AN, AM and RN models during test on the "large" dataset referring to the predictions of the comparison matrix. Please note that the differences between the AN and AM models without multitasking on the Eigenvector centrality are only due to the stochastic nature of the training, being that the pipeline is the same in both models for this centrality. The best values for a metric under a centrality are in **bold**. Source: Author

	AN				AM				RN			
Centrality	P	R	TN	Acc	P	R	TN	Acc	P	R	TN	Acc
Without multitasking												
Betweenness	79.2	85.6	81.3	**82.8**	37.5	43.2	40.4	41.3	78.7	**87.4**	69.4	78.4
Closeness	64.8	64.9	66.0	65.5	64.9	65.0	66.0	65.5	63.3	61.2	**89.0**	75.6
Degree	74.0	**98.5**	80.8	86.7	50.7	74.6	58.2	63.4	**77.2**	73.4	**99.8**	**87.4**
Eigenvector	88.9	88.8	89.1	89.0	**89.1**	**89.0**	89.3	**89.2**	71.0	67.7	**89.8**	78.8
Average	76.7	**84.5**	79.3	**81.0**	60.5	68.0	63.5	64.9	72.5	72.4	**87.0**	80.0
With multitasking												
Betweenness	77.6	83.8	79.9	81.3	39.1	43.9	42.6	43.0	**85.1**	75.9	**88.1**	81.8
Closeness	58.6	58.6	60.0	59.3	**81.6**	**82.4**	82.1	**82.3**	70.0	58.4	88.9	74.1
Degree	73.4	97.4	80.3	86.0	49.7	71.9	58.0	62.5	76.8	70.9	98.8	85.9
Eigenvector	85.7	85.6	85.9	85.7	83.7	83.6	84.0	83.8	77.6	87.2	65.9	76.5
Average	73.8	81.3	76.5	78.1	63.5	70.5	66.7	67.9	**77.4**	73.1	85.4	79.6

the fact that there was very little variation for centrality measure in question, which was very common on Erdös-Renyi graphs.

There are also examples of the PCA of the vertex embeddings providing bad visual fits and the model achieving a good performance, such as in Figure 4.20, as well as bad perfomances on such PCAs with poor visual fit, as in Figure 4.21. This variety of examples show how much the PCA of the embeddings varied and how one cannot take solid conclusions out of them that are valid to all graph distributions. In fact, the PCA of the embeddings were highly similar for graphs of the same distribution. On the previous images there were examples of 4 of the 6 random distributions that the synthetic graphs were drawn from. Figures 4.22 and 4.23 provide examples of the evolution of the PCA embeddings for the other two distributions.

This apparent relation between the PCA of the embedding and the log scale of the centrality being predicted, which happened in a significant number of the cases, can be seen as promising, since it relates to the nature of the centrality values themselves in some of these distributions, and might reflect that the model learned to sometimes pinpoint this information inside the model for specific cases or distributions. However, since the graph distributions used are mostly synthetic, this analysis is inconclusive and further work should be done to elucidate whether there is significant information to be extracted relating the approximation learned by the model.

Table 4.8: Performance metrics (Precision, Recall, True Negative rate, Accuracy) for the AN, AM and RN models during test on the "different" dataset referring to the predictions of the comparison matrix. Please note that the differences between the AN and AM models without multitasking on the Eigenvector centrality are only due to the stochastic nature of the training, being that the pipeline is the same in both models for this centrality. The best values for a metric under a centrality are in **bold**. Source: Author

	AN				AM				RN			
Centrality	P	R	TN	Acc	P	R	TN	Acc	P	R	TN	Acc
Without multitasking												
Betweenness	80.2	**80.3**	80.8	**80.5**	43.1	43.2	44.8	44.0	**81.2**	77.5	**81.8**	79.7
Closeness	80.8	82.4	81.6	82.0	80.8	82.4	81.6	82.0	81.7	75.3	**84.2**	79.9
Degree	82.7	94.6	85.2	89.0	75.3	86.8	78.3	81.7	86.4	72.5	89.0	82.1
Eigenvector	70.4	70.4	71.2	70.8	69.9	69.9	70.7	70.3	**84.9**	**87.9**	**83.8**	**85.8**
Average	78.5	**81.9**	79.7	80.6	67.3	70.6	68.9	69.5	**83.6**	78.3	**84.7**	**81.9**
With multitasking												
Betweenness	73.4	73.6	74.3	73.9	46.6	46.8	48.2	47.5	77.9	77.0	78.5	77.8
Closeness	81.3	82.9	82.2	82.5	**81.9**	**83.5**	82.7	**83.1**	79.6	77.5	81.4	79.5
Degree	85.0	**97.0**	87.4	**91.3**	84.1	96.1	86.5	90.3	**87.4**	74.9	**91.0**	84.0
Eigenvector	67.5	67.5	68.4	68.0	70.4	70.4	71.1	70.7	79.6	80.5	79.9	80.2
Average	76.8	80.3	78.1	78.9	70.8	74.2	72.1	72.9	81.1	77.5	82.7	80.4

As said in Subsection 4.3.4, the model did improve its performance for larger numbers of message-passing iterations, the PCA of these values sometimes did not hold their structure, even though sometimes the performance of the model remaind somewhat stable. Figure 4.25 shows an example of unstable behaviour of the performance after the 32 iterations that the model was trained with and Figure 4.24 shows a case where the performance was more stable throghout the iterations.

Figure 4.12: Overall accuracy of the multitask RN model for problems with an increasing number of vertices. The overall accuracy decays with increasing problem sizes, but it does not approach the baseline of 50% (random guess) for the largest instances tested. The dotted lines delimit the range of problem sizes used during training. Source: Author

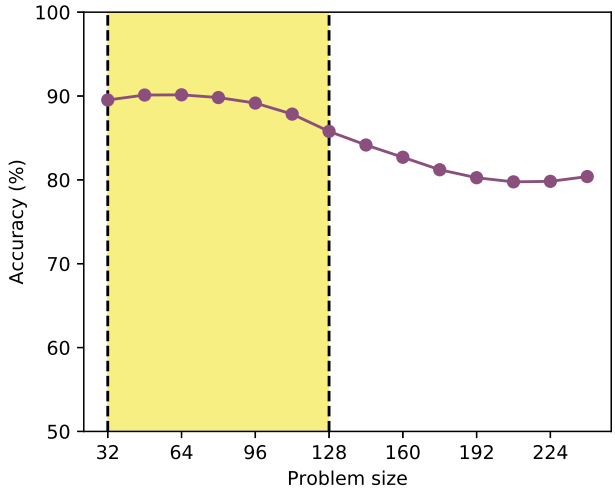


Figure 4.13: Overall accuracy of the non-multitask RN models for problems with an increasing number of vertices. Subfigures (a), (b), (c), and (d) are for the betweeness, closeness, degree, and eigenvector models. The overall accuracy decays with increasing problem sizes, but it does not approach the baseline of 50% (random guess) for the largest instances tested. The dotted lines delimit the range of problem sizes used during training. Source: Author

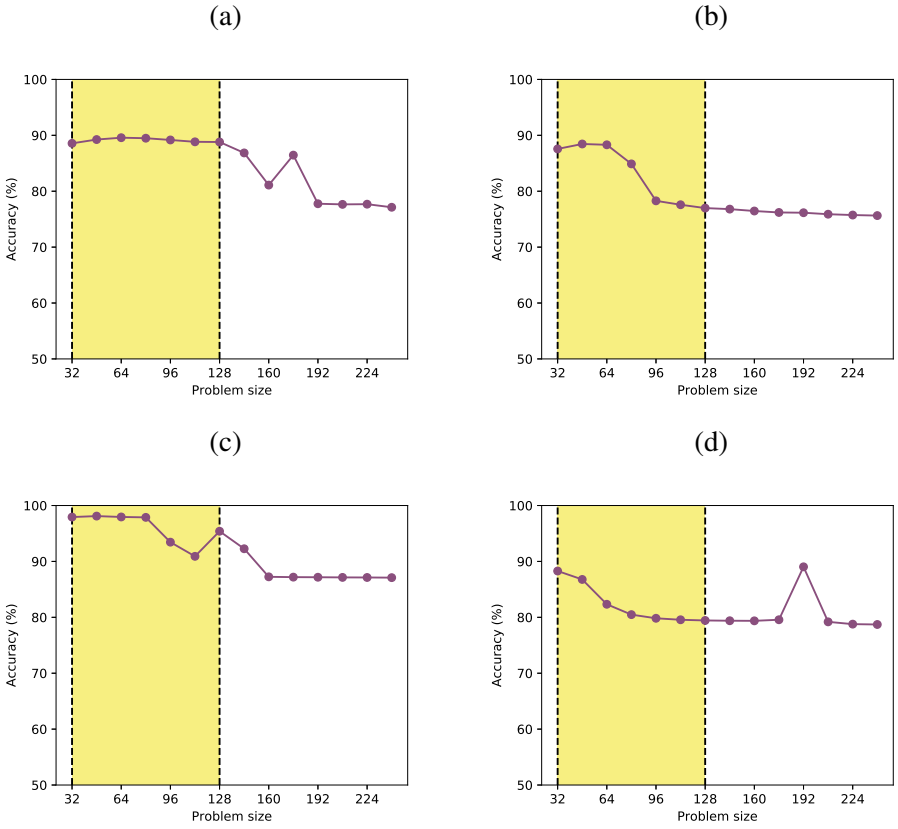


Figure 4.14: Overall accuracy of the multitask AM model, akin to Figure 4.12 for the RN model. The overall accuracy decays with increasing problem sizes, but its decay and performance are both smaller than for the RN model. Source: Author

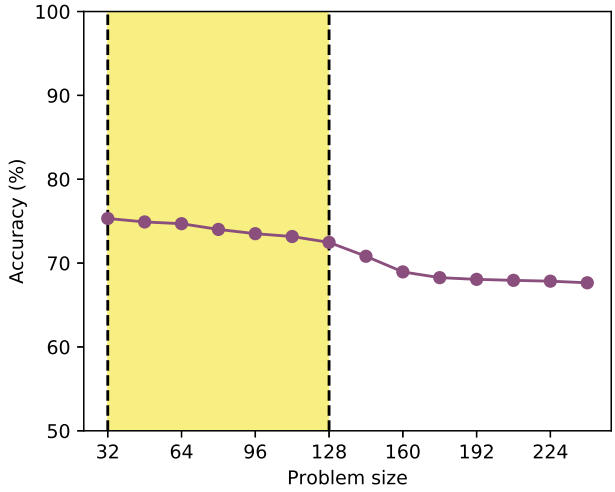


Figure 4.15: The overall accuracy of the non-multitask AM models, organised as in Figure 4.13. It fails to reach the baseline for the betweeness centrality, has a sharp decay at larger instances for the closeness centrality, decays in the range of training for the degree centrality and seems to be stable for the eigenvector centrality. Source: Author

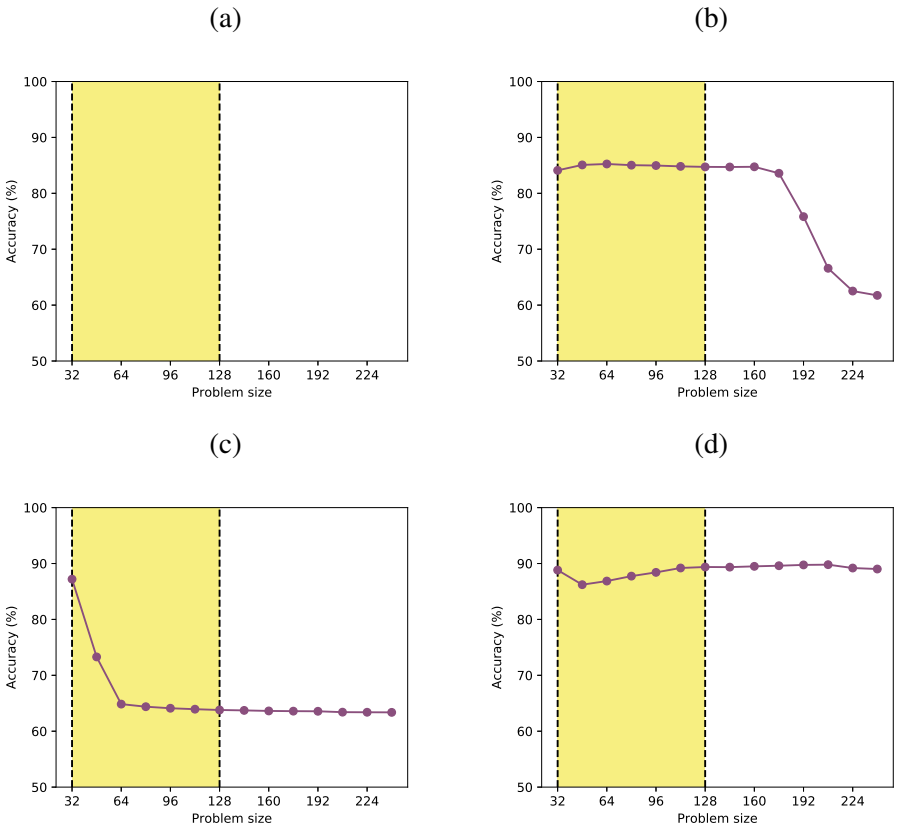


Figure 4.16: The overall accuracy of the multitask AN model, akin to Figure 4.12 for the RN model. The overall accuracy seems to stabilize for larger problem sizes. Source: Author

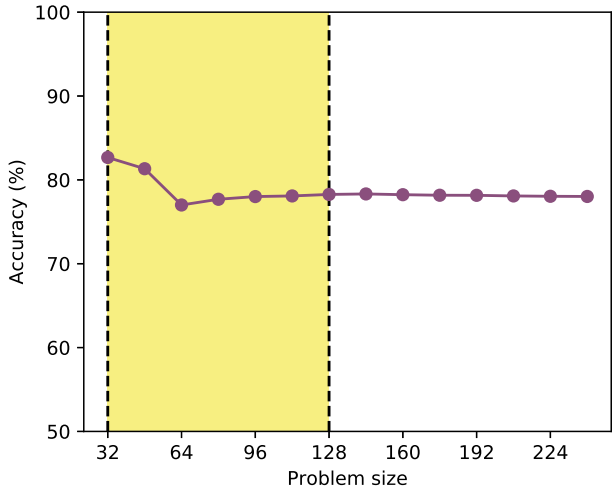


Figure 4.17: The overall accuracy of the non-multitask AN models, organised as in Figure 4.13. The overall accuracy seems to stabilize with increasing problem sizes for all centralities, except closeness. Source: Author

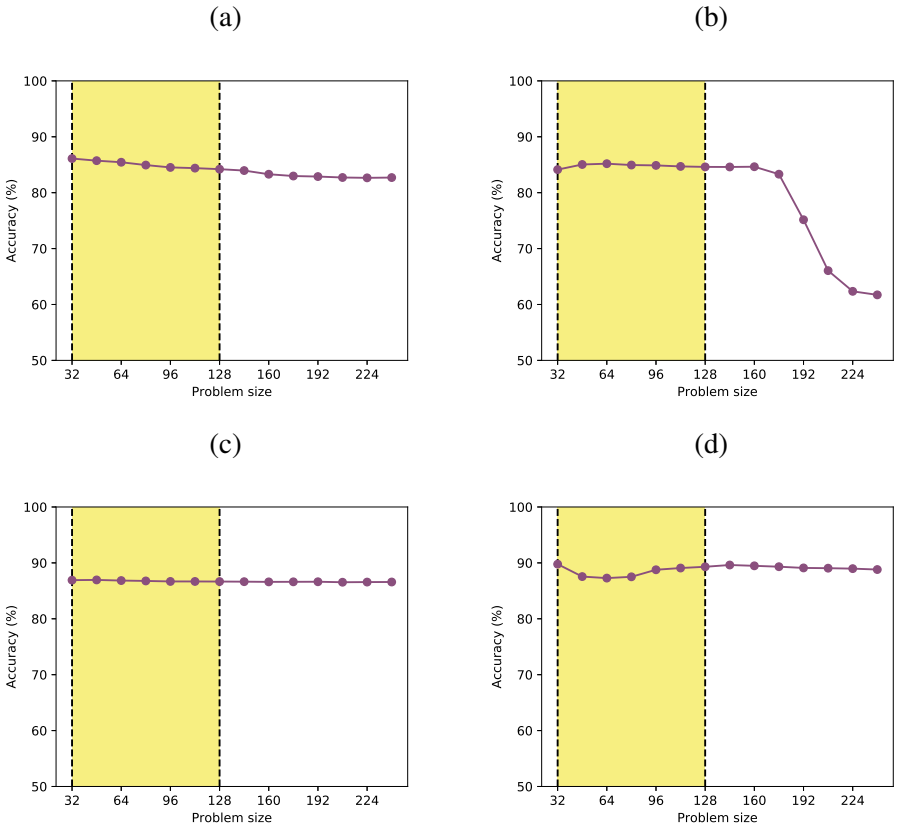


Table 4.9: Accuracy performance of the models during test on the "real" dataset (without/with multitasking) referring to the predictions of the comparison matrix. The graph names are abbreviated as in Table 4.5 and the number of decimal digits was also reduced so that the results fit on page. The best values under a centrality and a graph are in **bold**. Source: Author

Centrality		Accuracy (%)						
		PE	EM	SH	SC	GQ	EGO	Average
Betweenness	RN	64/**66**	77/**81**	84/85	**84**/83	**81**/75	**77**/75	**78**/66
	AM	60/58	35/40	16/18	38/37	59/58	28/28	39/40
	AN	65/57	78/70	**90**/76	**84**/52	73/60	**77**/43	**78**/60
Closeness	RN	71/65	81/83	60/74	77/80	62/68	64/58	69/61
	AM	**76**/71	89/**89**	66/**80**	80/**81**	**74**/71	**65**/55	**75**/**75**
	AN	**76**/73	**89**/88	65/19	80/61	**74**/**74**	**65**/48	**75**/61
Degree	RN	78/82	86/83	67/73	80/80	82/84	74/72	78/68
	AM	85/86	90/**94**	15/45	60/80	**93**/92	47/68	65/77
	AN	**88**/81	93/93	93/**96**	91/**94**	91/83	81/**95**	89/**90**
Eigenvector	RN	**67**/63	**73**/**73**	**87**/69	**86**/79	62/**64**	**66**/57	**74**/58
	AM	60/51	71/65	79/80	60/69	62/58	65/46	66/62
	AN	59/56	71/66	77/80	59/68	61/61	63/46	65/63
Average	RN	70/69	79/80	74/75	**82**/81	72/73	70/65	75/74
	AM	70/67	71/72	44/56	60/67	72/70	51/49	61/63
	AN	**72**/67	**83**/79	**81**/68	79/69	**75**/69	**71**/58	**77**/68

Figure 4.18: Evolution of the 1D projection of vertex embeddings of a non-multitask model plotted against the corresponding eigenvector centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the Watts-Strogatz small world distribution of the "large" dataset. In this one can see how the PCA of the embeddings seem to sort the vertices along their eigenvector centrality while the model improves its performance. The cause for the most central vertex shifting from being on the left to the right is most likely irrelevant. Source: Author

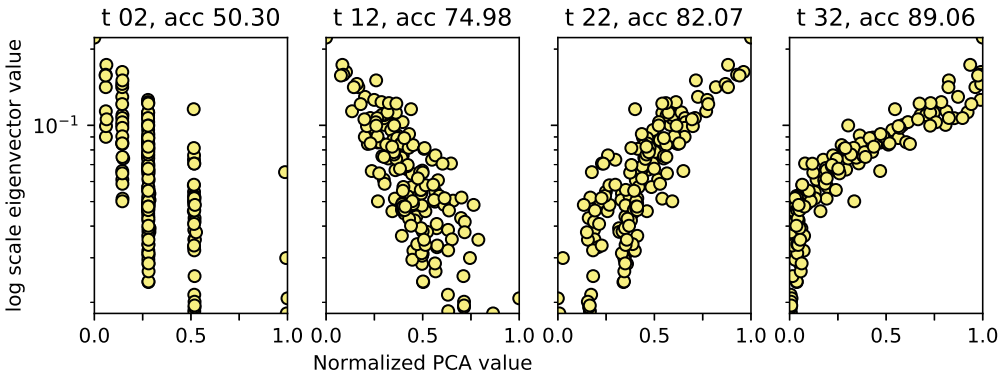


Figure 4.19: Evolution of the 1D projection of vertex embeddings of a non-multitask model plotted against the corresponding betweenness centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the Erdös-Renyi distribution of the "large" dataset. In this one can see how the PCA of the embeddings seem to sort the vertices along their betweenness centrality, but the performance of the model is at the 50% baseline. Source: Author

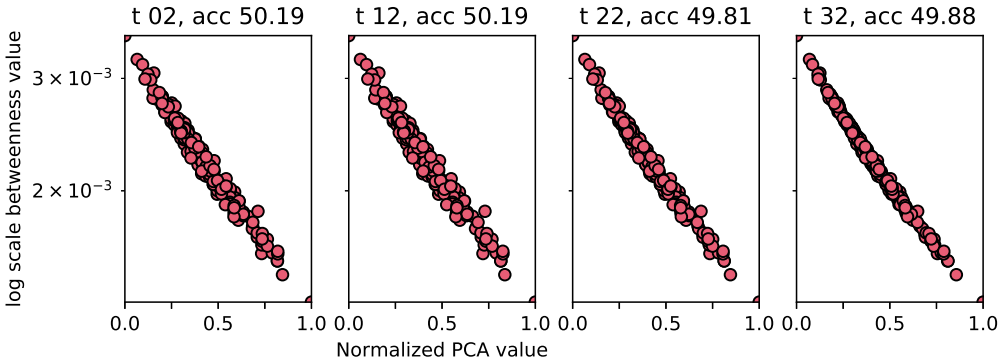


Figure 4.20: Evolution of the 1D projection of vertex embeddings of a multitask model plotted against the corresponding degree centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the powerlaw tree distribution of the "test" dataset. In this the PCA of the embeddings do not provide a good visual fit for sorting the vertices along their degree centrality, but the model manages to achieve a good performance nonetheless. Source: Author

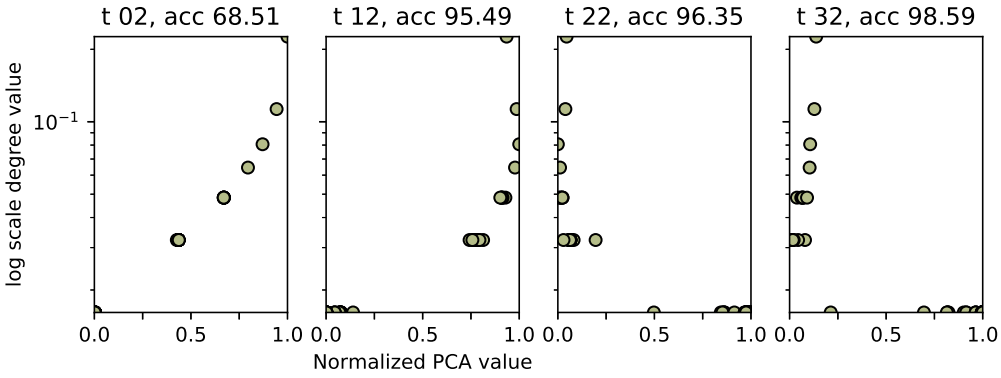


Figure 4.21: Evolution of the 1D projection of vertex embeddings of a multitask model plotted against the corresponding closeness centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the shell distribution of the “different” dataset. In this the PCA of the embeddings do not provide a good visual fit for sorting the vertices along their degree centrality, and the model does not manage to achieve a very good performance. Source: Author

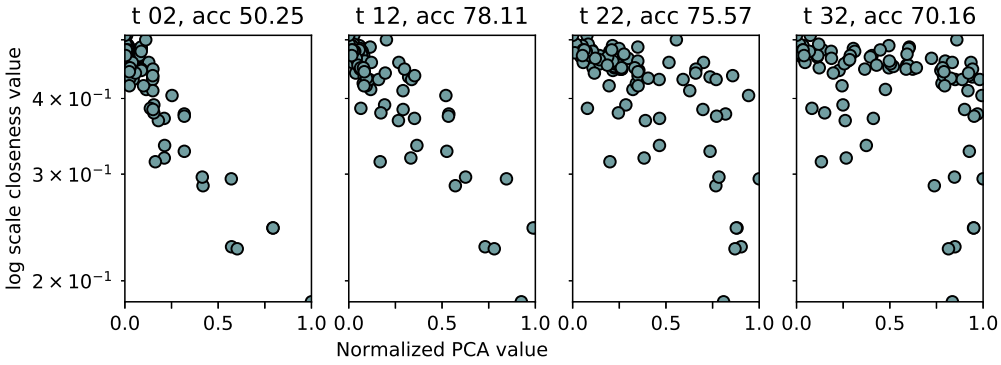


Figure 4.22: Evolution of the 1D projection of vertex embeddings of a non-multitask model plotted against the corresponding betweenness centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the Barabasi-Albert distribution of the “different” dataset. Source: Author

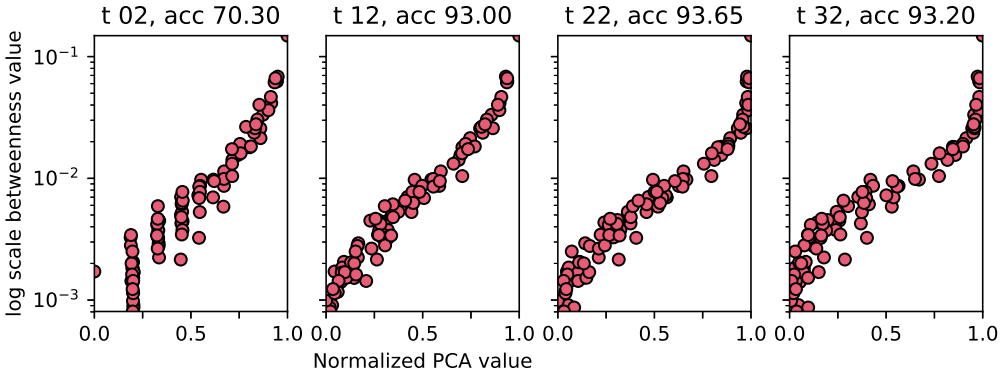


Figure 4.23: Evolution of the 1D projection of vertex embeddings of a non-multitask model plotted against the corresponding degree centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the powerlaw cluster distribution of the “large” dataset. Source: Author

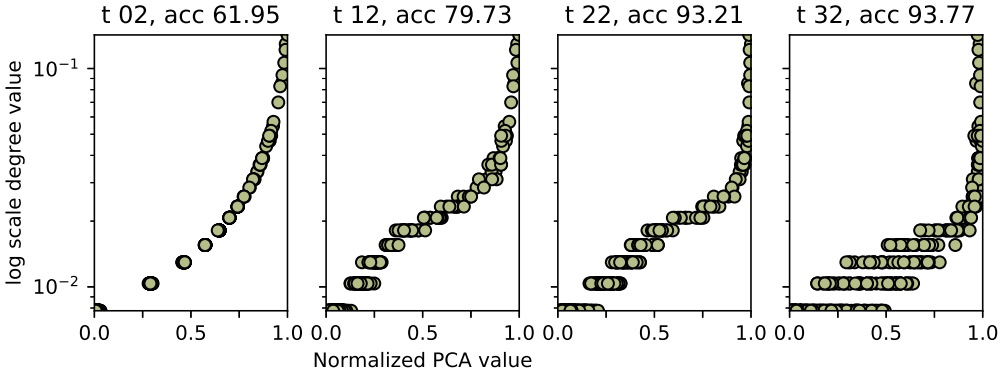


Figure 4.24: Evolution of the 1D projection of vertex embeddings of a multitask model plotted against the corresponding eigenvector centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the powerlaw tree distribution of the "test" dataset. Here, we extend the model to work with more message-passing iterations than it was trained with, and its performance remains relatively stable throughout. Source: Author

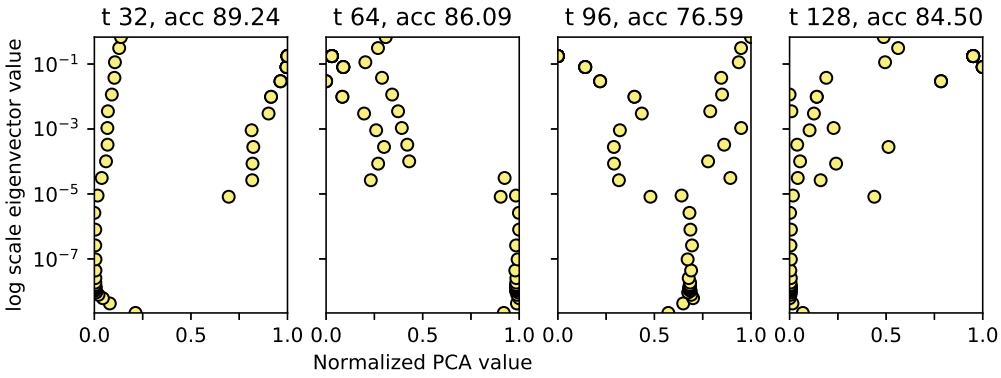
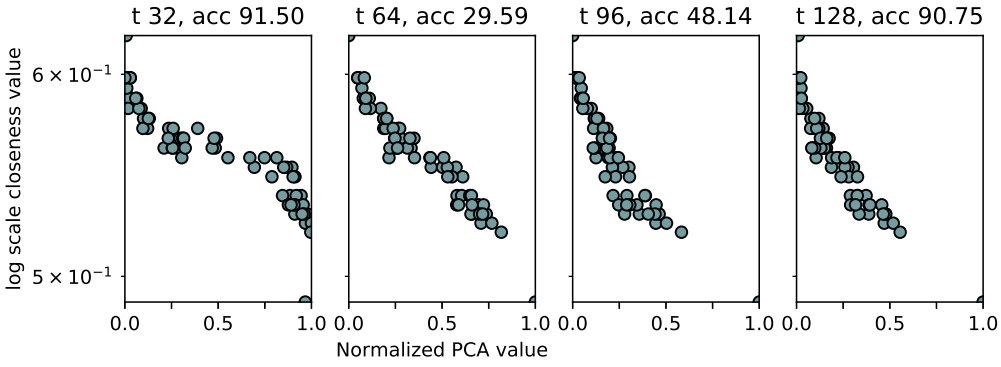


Figure 4.25: Evolution of the 1D projection of vertex embeddings of a multitask model plotted against the corresponding closeness centralities (plotted on a log scale) through the message-passing iterations for a graph sampled from the Erdös-Renyi distribution of the "test" dataset. Here, we extend the model to work with more message-passing iterations than it was trained with, and its performance varies throughout this extended range. Source: Author



5 CONCLUSIONS AND FUTURE WORK

In this dissertation we demonstrated how one can train a neural network to predict graph centrality measures, while feeding it with only the raw network structure. To achieve this, we implemented a library for Graph Neural Networks so that it enforced permutation invariance among graph elements through its message-passing algorithm composed of neural modules with shared weights. These modules were assembled in configurations reflecting the network structure of each problem.

Experiments were made both with approximating centrality measures directly and with ranking centrality measures to answer Research Question 1. It was seen that the task of approximating centrality measures was hard, but also that the proposed model can be trained to predict centrality comparisons (i.e. is vertex v_1 more central than vertex v_2 given the centrality measure c?) with high accuracy.

In order to answer Research Question 3 the model was instantiated separately for each centrality measure as well as for predicting all of them simultaneously, a technique was shown where there is a minimal effect to the overall accuracy while providing significant savings in the number of parameters. In most cases, it seems that the models could not benefit from similarities in the sub-problems, although one may say that this could be due to the fact that the model had the same space to work as the specialised models, with a specific configuration of the model showing an unexpected increase in performance on the multitask variant. For the multitasking version the model had to encode all information about all the centrality measures jointly in its vertices' embeddings so that they would be extracted separately for each centrality by an MLP on the end of the execution pipeline.

In order to shed light on Research Question 2 the PCA projections of the vertices' embeddings was analysed for the models trained to perform the ranking along the centrality measures. It seems that some information pertaining the centrality measures themselves is present in the PCAs, but they vary wildly along instances and do not seem to provide a robust method for extracting further information from the model. However, the PCA visualisations seem to suggest that the neural network model can learn a representation that translates into a vertex's centrality in a graph, but further work is needed to extract this information in a robust way.

The model was also tested for its generalisation for running more message-passing iterations, in order to answer Research Question 4. The model's performance, however, seemed to suffer from this more than it gained – presenting instabilities in its accuracy and

overall not gaining much from the increased computation.

Finally, the model was tested with both graphs from different distributions than the ones with which it was trained as well as with larger graphs. In the larger graphs most models had a decay in performance, but showed some level of generalisation to the larger instances, even for real world graphs which overestimate from 9 to 31 times the size of the largest graphs the model saw during training. With this it seems safe to say that the answer to Research Question 5 is that some level of generalisation is achievable, with the best generalisation across domains being with multitasking and the best generalisation along the number of nodes being with approximation-based ranking, but one should expect a decay on the performance of the model while working out-of-sample. Unfortunately, however, since the model did not increase its performance through more computation time (Research Question 4), this seems hard to tackle, while techniques exist in the literature that allow this (PALM; PAQUET; WINTHER, 2017), we wished it so that our model could achieve this performance while working in the same vein as (SELSAM et al., 2018).

The future directions to this work are manifold. One could experiment with different centrality measures (possibly even edge-level centralities), different architectures for the message-passing neural network model, working with weighted and/or directed graphs, as well as try different forms of normalisation while training the model. Since the analysis of the embeddings learned by the model could not be conclusive, it would also be an interesting future direction to evaluate how to train this model to learn a more interpretable representation. Another clear path for future work is working on how to efficiently scale up the model to networks with a higher number of vertices, with a possible venue of investigation mentioned in the end of Section 4.3.4.

REFERENCES

ABADI, M. et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Available from Internet: <http://tensorflow.org/>.

ABDEL-HAMID, O.; DENG, L.; YU, D. Exploring convolutional neural network structures and optimization techniques for speech recognition. In: BIMBOT, F. et al. (Ed.). **INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France, August 25-29, 2013**. ISCA, 2013. p. 3366–3370. Available from Internet: <http://www.isca-speech.org/archive/interspeech_2013/i13_3366.html>.

ABDEL-HAMID, O. et al. Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition. In: **2012 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2012, Kyoto, Japan, March 25-30, 2012**. IEEE, 2012. p. 4277–4280. Available from Internet: <https://doi.org/10.1109/ICASSP.2012.6288864>.

ABDEL-HAMID, O. et al. Convolutional neural networks for speech recognition. **IEEE/ACM Trans. Audio, Speech & Language Processing**, v. 22, n. 10, p. 1533–1545, 2014. Available from Internet: <https://doi.org/10.1109/TASLP.2014.2339736>.

ALBERT, R.; BARABÁSI, A.-L. Statistical mechanics of complex networks. **Reviews of modern physics**, APS, v. 74, n. 1, p. 47, 2002.

ALBERT, R.; JEONG, H.; BARABÁSI, A.-L. Internet: Diameter of the world-wide web. **nature**, Nature Publishing Group, v. 401, n. 6749, p. 130, 1999.

AVELAR, P. H. C. et al. Multitask learning on graph neural networks: Learning multiple graph centrality measures with a unified network. Springer, v. 11731, p. 701–715, 2019. Available from Internet: <https://doi.org/10.1007/978-3-030-30493-5_63>.

BA, L. J.; KIROS, R.; HINTON, G. E. Layer normalization. **CoRR**, abs/1607.06450, 2016. Available from Internet: <http://arxiv.org/abs/1607.06450>.

BARABÁSI, A.-L. et al. **Network science**. [S.l.]: Cambridge university press, 2016.

BATAGELJ, V.; BRANDES, U. Efficient generation of large random networks. **Physical Review E**, APS, v. 71, n. 3, p. 036113, 2005.

BATOOL, K.; NIAZI, M. Towards a methodology for validation of centrality measures in complex networks. **PLOS ONE**, Public Library of Science, v. 9, n. 4, p. 1–14, 2014.

BATTAGLIA, P. W. et al. Relational inductive biases, deep learning, and graph networks. **CoRR**, abs/1806.01261, 2018. Available from Internet: <http://arxiv.org/abs/1806.01261>.

BEAUCHAMP, M. An improved index of centrality. **Behavioral Science**, v. 10, n. 2, p. 161–163, 1965.

BENGIO, Y. Practical recommendations for gradient-based training of deep architectures. In: MONTAVON, G.; ORR, G. B.; MÜLLER, K. (Ed.). **Neural Networks: Tricks of the Trade - Second Edition**. Springer, 2012, (Lecture Notes in Computer Science, v. 7700). p. 437–478. Available from Internet: <https://doi.org/10.1007/978-3-642-35289-8_26>.

BENGIO, Y.; LODI, A.; PROUVOST, A. Machine learning for combinatorial optimization: a methodological tour d'horizon. **CoRR**, abs/1811.06128, 2018. Available from Internet: <http://arxiv.org/abs/1811.06128>.

BENGIO, Y.; SIMARD, P. Y.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. **IEEE Trans. Neural Networks**, v. 5, n. 2, p. 157–166, 1994. Available from Internet: <https://doi.org/10.1109/72.279181>.

BONACICH, P. Power and centrality: A family of measures. **American Journal of Sociology**, University of Chicago Press, v. 92, n. 5, p. 1170–1182, 1987. ISSN 00029602, 15375390.

BONDY, J. A.; MURTY, U. S. R. **Graph theory, volume 244 of Graduate Texts in Mathematics**. [S.l.]: Springer, New York, 2008.

BONDY, J. A.; MURTY, U. S. R. et al. **Graph theory with applications**. [S.l.]: Citeseer, 1976.

BORGATTI, S. P.; EVERETT, M. G. A graph-theoretic perspective on centrality. **Social Networks**, v. 28, n. 4, p. 466–484, 2006. Available from Internet: <https://doi.org/10.1016/j.socnet.2005.11.005>.

BREIMAN, L. et al. Statistical modeling: The two cultures (with comments and a rejoinder by the author). **Statistical science**, Institute of Mathematical Statistics, v. 16, n. 3, p. 199–231, 2001.

CHANG, K.-C.; PEARSON, K.; ZHANG, T. Perron-frobenius theorem for nonnegative tensors. **Communications in Mathematical Sciences**, International Press of Boston, v. 6, n. 2, p. 507–520, 2008.

CHEN, K. et al. Do research institutes benefit from their network positions in research collaboration networks with industries or/and universities? **Technovation**, 2017. ISSN 0166-4972. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0166497217307836>.

CHO, K. et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: MOSCHITTI, A.; PANG, B.; DAELEMANS, W. (Ed.). **Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL**. ACL, 2014. p. 1724–1734. Available from Internet: <http://aclweb.org/anthology/D/D14/D14-1179.pdf>.

CHUNG, J. et al. Empirical evaluation of gated recurrent neural networks on sequence modeling. **CoRR**, abs/1412.3555, 2014. Available from Internet: <http://arxiv.org/abs/1412.3555>.

COLLINS, J. J.; CHOW, C. C. It's a small world. **Nature**, Nature Publishing Group, v. 393, n. 6684, p. 409, 1998.

DONG, J. Q.; MCCARTHY, K. J.; SCHOENMAKERS, W. W. How central is too central? organizing interorganizational collaboration networks for breakthrough innovation. **Journal of Product Innovation Management**, v. 34, n. 4, p. 526–542, 2017.

DUCHI, J. C.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. **Journal of Machine Learning Research**, v. 12, p. 2121–2159, 2011. Available from Internet: <http://dl.acm.org/citation.cfm?id=2021068>.

DUVENAUD, D. K. et al. Convolutional networks on graphs for learning molecular fingerprints. In: CORTES, C. et al. (Ed.). **Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada**. [s.n.], 2015. p. 2224–2232. Available from Internet: <http://papers.nips.cc/paper/5954-convolutional-networks-on-graphs-for-learning-molecular-fingerprints>.

EASLEY, D. A.; KLEINBERG, J. M. **Networks, Crowds, and Markets - Reasoning About a Highly Connected World**. Cambridge University Press, 2010. ISBN 978-0-521-19533-1. Available from Internet: <http://www.cambridge.org/gb/knowledge/isbn/item2705443/?site_locale=en_GB>.

EPPSTEIN, D.; WANG, J. Fast approximation of centrality. **J. Graph Algorithms Appl.**, v. 8, p. 39–45, 2004. Available from Internet: <http://jgaa.info/accepted/2004/EppsteinWang2004.8.1.pdf>.

ESTRADA, E.; ROSS, G. J. Centralities in simplicial complexes. applications to protein interaction networks. **Jnl. Theoret. Biology**, v. 438, p. 46–60, 2018. ISSN 0022-5193.

GILMER, J. et al. Neural message passing for quantum chemistry. PMLR, v. 70, p. 1263–1272, 2017. Available from Internet: <http://proceedings.mlr.press/v70/gilmer17a.html>.

GIRVAN, M.; NEWMAN, M. E. Community structure in social and biological networks. **Proceedings of the national academy of sciences**, National Acad Sciences, v. 99, n. 12, p. 7821–7826, 2002.

GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In: TEH, Y. W.; TITTERINGTON, D. M. (Ed.). **Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010**. JMLR.org, 2010. (JMLR Proceedings, v. 9), p. 249–256. Available from Internet: <http://www.jmlr.org/proceedings/papers/v9/glorot10a.html>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

GORI, M.; MONFARDINI, G.; SCARSELLI, F. A new model for learning in graph domains. In: IEEE. **Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on**. [S.l.], 2005. v. 2, p. 729–734.

GRANDO, F.; GRANVILLE, L. Z.; LAMB, L. C. Machine learning in network centrality measures: Tutorial and outlook. **CoRR**, abs/1810.11760, 2018. Available from Internet: <http://arxiv.org/abs/1810.11760>.

GRANDO, F.; LAMB, L. C. Estimating complex networks centrality via neural networks and machine learning. In: **2015 International Joint Conference on Neural Networks, IJCNN 2015, Killarney, Ireland, July 12-17, 2015**. IEEE, 2015. p. 1–8. Available from Internet: <https://doi.org/10.1109/IJCNN.2015.7280334>.

GRANDO, F.; LAMB, L. C. On approximating networks centrality measures via neural learning algorithms. In: **2016 International Joint Conference on Neural Networks, IJCNN 2016, Vancouver, BC, Canada, July 24-29, 2016**. IEEE, 2016. p. 551–557. Available from Internet: <https://doi.org/10.1109/IJCNN.2016.7727248>.

GRANDO, F.; LAMB, L. C. Computing vertex centrality measures in massive real networks with a neural learning model. In: **2018 International Joint Conference on Neural Networks, IJCNN 2018, Rio de Janeiro, Brazil, July 8-13, 2018**. IEEE, 2018. p. 1–8. Available from Internet: <https://doi.org/10.1109/IJCNN.2018.8489690>.

GRAVES, A.; WAYNE, G.; DANIHELKA, I. Neural turing machines. **CoRR**, abs/1410.5401, 2014. Available from Internet: <http://arxiv.org/abs/1410.5401>.

GRAVES, A. et al. Hybrid computing using a neural network with dynamic external memory. **Nature**, v. 538, n. 7626, p. 471–476, 2016. Available from Internet: <https://doi.org/10.1038/nature20101>.

HAGBERG, A.; SWART, P.; CHULT, D. S. **Exploring network structure, dynamics, and function using NetworkX**. [S.l.], 2008.

HE, K. et al. Deep residual learning for image recognition. In: **2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016**. IEEE Computer Society, 2016. p. 770–778. Available from Internet: <https://doi.org/10.1109/CVPR.2016.90>.

HINES, P.; BLUMSACK, S. A centrality measure for electrical networks. In: **41st Hawaii International International Conference on Systems Science (HICSS-41 2008), Proceedings, 7-10 January 2008, Waikoloa, Big Island, HI, USA**. IEEE Computer Society, 2008. p. 185. Available from Internet: <https://doi.org/10.1109/HICSS.2008.5>.

HINES, P. et al. The topological and electrical structure of power grids. In: **43rd Hawaii International International Conference on Systems Science (HICSS-43 2010), Proceedings, 5-8 January 2010, Koloa, Kauai, HI, USA**. IEEE Computer Society, 2010. p. 1–10. Available from Internet: <https://doi.org/10.1109/HICSS.2010.398>.

HINTON, G. E.; SABOUR, S.; FROSST, N. Matrix capsules with em routing. 2018.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural Computation**, v. 9, n. 8, p. 1735–1780, 1997. Available from Internet: <https://doi.org/10.1162/neco.1997.9.8.1735>.

HOLME, P.; KIM, B. J. Growing scale-free networks with tunable clustering. **Physical review E**, APS, v. 65, n. 2, p. 026107, 2002.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. JMLR.org, v. 37, p. 448–456, 2015. Available from Internet: <http://jmlr.org/proceedings/papers/v37/ioffe15.html>.

JOLLIFFE, I. T. Principal component analysis. In: LOVRIC, M. (Ed.). **International Encyclopedia of Statistical Science**. Springer, 2011. p. 1094–1096. Available from Internet: <https://doi.org/10.1007/978-3-642-04898-2_455>.

KAISER, L.; SUTSKEVER, I. Neural gpus learn algorithms. **CoRR**, abs/1511.08228, 2015. Available from Internet: <http://arxiv.org/abs/1511.08228>.

KAMADA, T.; KAWAI, S. An algorithm for drawing general undirected graphs. **Inf. Process. Lett.**, v. 31, n. 1, p. 7–15, 1989. Available from Internet: <https://doi.org/10.1016/0020-0190(89)90102-6>.

KEARNES, S. M. et al. Molecular graph convolutions: moving beyond fingerprints. **Journal of Computer-Aided Molecular Design**, v. 30, n. 8, p. 595–608, 2016. Available from Internet: <https://doi.org/10.1007/s10822-016-9938-8>.

KENDALL, M. G. A new measure of rank correlation. **Biometrika**, JSTOR, v. 30, n. 1/2, p. 81–93, 1938.

KIM, J.; HASTAK, M. Social network analysis: Characteristics of online social networks after a disaster. **Int J. Information Management**, v. 38, n. 1, p. 86–96, 2018. Available from Internet: <https://doi.org/10.1016/j.ijinfomgt.2017.08.003>.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **CoRR**, abs/1412.6980, 2014. Available from Internet: <http://arxiv.org/abs/1412.6980>.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: BARTLETT, P. L. et al. (Ed.). **Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.** [s.n.], 2012. p. 1106–1114. Available from Internet: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>.

KUMAR, A.; MEHROTRA, K. G.; MOHAN, C. K. Neural networks for fast estimation of social network centrality measures. In: SPRINGER. **Proceedings of the Fifth International Conference on Fuzzy and Neuro Computing (FANCCO-2015)**. [S.l.], 2015. p. 175–184.

LECUN, Y.; BENGIO, Y.; HINTON, G. E. Deep learning. **Nature**, v. 521, n. 7553, p. 436–444, 2015. Available from Internet: <https://doi.org/10.1038/nature14539>.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, IEEE, v. 86, n. 11, p. 2278–2324, 1998.

LEE, C.-Y. Correlations among centrality measures in complex networks. **arXiv preprint physics/0605220**, 2006.

LESKOVEC, J.; KREVL, A. **SNAP Datasets: Stanford Large Network Dataset Collection**. 2014. <http://snap.stanford.edu/data>.

LI, H. et al. A convolutional neural network cascade for face detection. In: **IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015**. IEEE Computer Society, 2015. p. 5325–5334. Available from Internet: <https://doi.org/10.1109/CVPR.2015.7299170>.

LI, Y. et al. Learning deep generative models of graphs. **CoRR**, abs/1803.03324, 2018. Available from Internet: <http://arxiv.org/abs/1803.03324>.

LIPTON, Z. C. The mythos of model interpretability. **CoRR**, abs/1606.03490, 2016. Available from Internet: <http://arxiv.org/abs/1606.03490>.

LIPTON, Z. C. The mythos of model interpretability. **Commun. ACM**, v. 61, n. 10, p. 36–43, 2018. Available from Internet: <https://doi.org/10.1145/3233231>.

LIU, B. et al. Recognition and vulnerability analysis of key nodes in power grid based on complex network centrality. **IEEE Trans. on Circuits and Systems**, v. 65-II, n. 3, p. 346–350, 2018. Available from Internet: <https://doi.org/10.1109/TCSII.2017.2705482>.

MORELLI, S. A. et al. Empathy and well-being correlate with centrality in different social networks. **PNAS**, National Academy of Sciences, v. 114, n. 37, p. 9843–9847, 2017. ISSN 0027-8424.

NEWMAN, M. **Networks: An Introduction**. [S.l.]: OUP Oxford, 2010.

NEWMAN, M. E. The structure of scientific collaboration networks. **Proceedings of the national academy of sciences**, National Acad Sciences, v. 98, n. 2, p. 404–409, 2001.

NEWMAN, M. E.; GIRVAN, M. Finding and evaluating community structure in networks. **Physical review E**, APS, v. 69, n. 2, p. 026113, 2004.

NIELSEN, M. A. **Neural networks and deep learning**. [S.l.]: Determination press USA, 2015.

OLAH, C. Conv nets: A modular perspective. Jul 2014. Online; accessed 13-January-2019. Available from Internet: <https://colah.github.io/posts/2014-07-Conv-Nets-Modular/>.

OLAH, C. Understanding lstm networks. Aug 2015. Online; accessed 13-January-2019. Available from Internet: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

OLAH, C.; CARTER, S. Attention and augmented recurrent neural networks. **Distill**, v. 1, n. 9, p. e1, 2016.

PALM, R. B.; PAQUET, U.; WINTHER, O. Recurrent relational networks for complex relational reasoning. **CoRR**, abs/1711.08028, 2017. Available from Internet: <http://arxiv.org/abs/1711.08028>.

PRATES, M. O. R.; AVELAR, P. H. C.; LAMB, L. C. On quantifying and understanding the role of ethics in AI research: A historical account of flagship conferences and journals. In: LEE, D. D.; STEEN, A.; WALSH, T. (Ed.). **GCAI-2018, 4th Global Conference on Artificial Intelligence, Luxembourg, September 18-21, 2018**. EasyChair, 2018. (EPiC Series in Computing, v. 55), p. 188–201. Available from Internet: <http://www.easychair.org/publications/paper/Z7D4>.

PRATES, M. O. R.; AVELAR, P. H. C.; LAMB, L. C. Assessing gender bias in machine translation: a case study with google translate. **Neural Computing and Applications**, Mar 2019. ISSN 1433-3058. Available from Internet: <https://doi.org/10.1007/s00521-019-04144-6>.

PRATES, M. O. R. et al. Learning to solve np-complete problems: A graph neural network for decision TSP. AAAI Press, p. 4731–4738, 2019. Available from Internet: <https://doi.org/10.1609/aaai.v33i01.33014731>.

PRATES, M. O. R. et al. Typed graph networks. **CoRR**, abs/1901.07984v3, 2019. Available from Internet: <http://arxiv.org/abs/1901.07984>.

RAPOSO, D. et al. Discovering objects and their relations from entangled scene representations. **CoRR**, abs/1702.05068, 2017. Available from Internet: <http://arxiv.org/abs/1702.05068>.

RAVASZ, E.; SOMERA ANNA LISA, M. D. A. O. Z. N.; BARABÁSI, A.-. Hierarchical organization of modularity in metabolic networks. **Science**, v. 297, n. 5586, p. 1551–1555, 2002.

ROSSI, R. A.; AHMED, N. K. The network data repository with interactive graph analytics and visualization. In: **AAAI**. [s.n.], 2015. Available from Internet: <http://networkrepository.com>.

RUDER, S. An overview of gradient descent optimization algorithms. **CoRR**, abs/1609.04747, 2016. Available from Internet: <http://arxiv.org/abs/1609.04747>.

SABOUR, S.; FROSST, N.; HINTON, G. E. Dynamic routing between capsules. In: GUYON, I. et al. (Ed.). **Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA**. [s.n.], 2017. p. 3859–3869. Available from Internet: <http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules>.

SANCHEZ-GONZALEZ, A. et al. Graph networks as learnable physics engines for inference and control. JMLR.org, v. 80, p. 4467–4476, 2018. Available from Internet: <http://proceedings.mlr.press/v80/sanchez-gonzalez18a.html>.

SANTORO, A. et al. A simple neural network module for relational reasoning. In: GUYON, I. et al. (Ed.). **Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA**. [s.n.], 2017. p. 4974–4983. Available from Internet: <http://papers.nips.cc/paper/7082-a-simple-neural-network-module-for-relational-reasoning>.

SARIYÜCE, A. E. et al. Graph manipulations for fast centrality computation. **TKDD**, v. 11, n. 3, p. 26:1–26:25, 2017. Available from Internet: <https://doi.org/10.1145/3022668>.

SCARSELLI, F. et al. The graph neural network model. **IEEE Trans. Neural Networks**, v. 20, n. 1, p. 61–80, 2009. Available from Internet: <https://doi.org/10.1109/TNN.2008.2005605>.

SCARSELLI, F. et al. Graph neural networks for ranking web pages. In: SKOWRON, A. et al. (Ed.). **2005 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2005), 19-22 September 2005, Compiegne, France**. IEEE Computer Society, 2005. p. 666–672. Available from Internet: <https://doi.org/10.1109/WI.2005.67>.

SCHOCH, D.; VALENTE, T. W.; BRANDES, U. Correlations among centrality indices and a class of uniquely ranked graphs. **Social Networks**, v. 50, p. 46–54, 2017. Available from Internet: <https://doi.org/10.1016/j.socnet.2017.03.010>.

SELSAM, D. et al. Learning a SAT solver from single-bit supervision. **CoRR**, abs/1802.03685, 2018. Available from Internet: <http://arxiv.org/abs/1802.03685>.

SETHURAMAN, G.; DHAVAMANI, R. Graceful numbering of an edge-gluing of shell graphs. **Discrete Mathematics**, v. 218, n. 1-3, p. 283–287, 2000. Available from Internet: <https://doi.org/10.1016/S0012-365X(99)00360-X>.

SHAW, M. Group structure and the behavior of individuals in small groups. **The Journal of Psychology**, Routledge, v. 38, n. 1, p. 139–149, 1954.

SIEGELMANN, H. T.; SONTAG, E. D. Turing computability with neural nets. **Applied Mathematics Letters**, Elsevier, v. 4, n. 6, p. 77–80, 1991.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. **CoRR**, abs/1409.1556, 2014. Available from Internet: <http://arxiv.org/abs/1409.1556>.

TANG, Y. et al. Cytonca: A cytoscape plugin for centrality analysis and evaluation of protein interaction networks. **Biosystems**, v. 127, p. 67–72, 2015. Available from Internet: <https://doi.org/10.1016/j.biosystems.2014.11.005>.

TROMP, J.; FARNEBÄCK, G. Combinatorics of go. In: SPRINGER. **International Conference on Computers and Games**. [S.l.], 2006. p. 84–99.

WANG, X. et al. Non-local neural networks. In: **2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018**. IEEE Computer Society, 2018. p. 7794–7803. Available from Internet: <http://openaccess.thecvf.com/content_cvpr_2018/html/Wang_Non-Local_Neural_Networks_CVPR_2018_paper.html>.

WANG, Z.; SCAGLIONE, A.; THOMAS, R. J. Electrical centrality measures for electric power grid vulnerability analysis. In: **Proceedings of the 49th IEEE Conference on Decision and Control, CDC 2010, December 15-17, 2010, Atlanta, Georgia, USA**. IEEE, 2010. p. 5792–5797. Available from Internet: <https://doi.org/10.1109/CDC.2010.5717964>.

WĄS, T.; SKIBSKI, O. An axiomatization of the eigenvector and katz centralities. In: **Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)**. [s.n.], 2018. Available from Internet: <https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16141/15817>.

WASSERMAN, S.; FAUST, K. **Social network analysis: Methods and applications**. [S.l.]: Cambridge university press, 1994. 201 p.

WATTS, D. J.; STROGATZ, S. H. Collective dynamics of 'small-world'networks. **Nature**, Nature Publishing Group, v. 393, n. 6684, p. 440, 1998.

WAYNE, Z. W. An information flow model for conflict and fission in small groups. **Journal of anthropological research**, v. 33, n. 4, p. 452–473, 1977.

YOU, J. et al. Graph convolutional policy network for goal-directed molecular graph generation. In: BENGIO, S. et al. (Ed.). **Advances in Neural Information**

Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. [s.n.], 2018. p. 6412–6422. Available from Internet: <http://papers.nips.cc/paper/7877-graph-convolutional-policy-network-for-goal-directed-molecular-graph-generation>.

YOU, J. et al. Graphrnn: Generating realistic graphs with deep auto-regressive models. In: DY, J. G.; KRAUSE, A. (Ed.). **Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018**. JMLR.org, 2018. (JMLR Workshop and Conference Proceedings, v. 80), p. 5694–5703. Available from Internet: <http://proceedings.mlr.press/v80/you18a.html>.

ZAHEER, M. et al. Deep sets. In: GUYON, I. et al. (Ed.). **Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA**. [s.n.], 2017. p. 3394–3404. Available from Internet: <http://papers.nips.cc/paper/6931-deep-sets>.

ZEILER, M. D. ADADELTA: an adaptive learning rate method. **CoRR**, abs/1212.5701, 2012. Available from Internet: <http://arxiv.org/abs/1212.5701>.

ZHANG, Z.; CUI, P.; ZHU, W. Deep learning on graphs: A survey. **CoRR**, abs/1812.04202, 2018. Available from Internet: <http://arxiv.org/abs/1812.04202>.

APPENDIX A — GRAPH NEURAL NETWORK LIBRARY

Part of the contribution achieved on the Master's Programme was the design and development of a library that instantiates a GNN/MPNN with several message functions and node types (SCARSELLI et al., 2009; GILMER et al., 2017), heavily based upon SELSAM et al.'s implementation of the Neurosat model (SELSAM et al., 2018), including most of the module/activation functions, design decisions, as well as the interface which was made to mimick the form they presented the equations in the paper. The main differences to SELSAM et al.'s implementation, however, is that our model is fully parallel, not performing sequential operations between different types throughout a time-step.

An earlier version of library was used for an unpublished "quiver" formalisation[1] of the TSP paper (PRATES et al., 2019a), and to be used for fast prototyping of new GNN models. It was used in all of the published papers produced by the laboratory as the foundational technological support for the experiments, greatly facilitating the speed with which one can prototype new models that use Graph Structures inside the Tensorflow Framework (ABADI et al., 2015). In this Appendix we will explain in detail how the library code works (in Section A.1, alongside example code of how to build Graph Neural Networks to be used in solving different problems (Subsections A.2, A.3, A.4), as well as the code for the library itself on the last two Sections (Sec. A.5 and A.6). The code is present in this appendix so that the text alludes to static piece of code, instead of a possibly dynamic and changing piece of code in an online repository, in any case there is a version made available by the author in his github account, which will possibly remain static, in the url <https://github.com/phcavelar/graph-neural-networks> and also in <https://github.com/phcavelar/graph-nn> where one can see the evolution of the library throughout the commits. The library, after being used for the TGN paper, was renamed and is available at <https://github.com/machine-reasoning-ufrgs/typed-graph-network>, the author does not give any guarantees that this version will remain static, as it might be reworked for future versions of the paper.

A.1 The library

The library consists of two files, "gnn.py" and "mlp.py", which contain one python class each. The "mlp.py" file contains an Mlp class which is simply a helper for building

[1] The quiver formalisation was basically a GNN with memories for both nodes and edges

MLP blocks and will not be discussed here, but one can see the code in Section A.6. The code for the “gnn.py” file can be seen in Appendix A, Section A.5 and will be referenced throughout this section as we explain the library itself. In this section there will be **(text in bold font)** showing which part of the GNN class is being explained in depth at the moment.

(__init__) The GNN class, contained in the “gnn.py” file, was the base for Algorithm 2 and 1, the second being closer to the how the library works, and implements a GNN model that uses Layer-Norm LSTMs (BA; KIROS; HINTON, 2016) for its update functions and Multilayer Perceptrons for its message functions. As said in Subsection 3.6.2, the output dimensionality for each of the message MLPs is the same as the target vertex’s embedding dimensionality, and one can only control the depth for the MLP, being that the dimensionality of each of its layers is equal to the source vertex’s embedding dimensionality, only the last layer having the target vertex’s dimensionality, but this can easily be changed in the library if one so desires.

The MLPs, by default, use Xavier Initialisation (GLOROT; BENGIO, 2010) for its weight matrices and initialises its bias matrix to zeros. The LSTMs use Tensorflow’s default initialisation for its parameters. The activation for both the MLPs and the LSTMs are Rectified Linear Units[2] (ReLUs) by default, but can be set when instancing a GNN block, also, the MLPs have no nonlinearities on their last layer as a default, which can also be separately changed.

To create an instance of a GNN block, however, one is required to pass as arguments to its constructor four dictionaries: “var”, “mat”, “msg” and “loop”. The “var” dictionary is one that maps from variable names to embedding sizes, that is, an entry var["V1"] = 10 means that the nodes of the type labeled as "V1" will have an embedding size of 10. “mat” is a dictionary from matrix names to variable pairs, which means that an entry mat["M"] = ("V1","V2") means that the matrix "M" can be as an adjacency matrices from vertices of type "V1" to those of type "V2". The “msg” dictionary maps function names to variable pairs, being that an entry msg["cast"] = ("V1","V2") means that one can apply "cast" to send messages from vertices of type "V1" to vertices of type "V2".

Finally, the “loop” dictionary is the one that defines which vertex types will receive messages from which vertex types, as well as defining other operations one may wish. This dictionary maps from the vertex types that will be updated to a list of “update” dictionaries containing instructions on how to process the input. The results from that list will all be

[2] A Rectified Linear Unit, or ReLU for short, consists in a line $y = x \forall x \geq 0$ and $y = 0 \forall x < 0$. There are some other variants but this is the one used here.

concatenated in the end before being used as input for each update LSTM itself. The dictionaries used inside the loop list contain five fields: “mat” which is the adjacency matrix that will be used for this part of the update along with a “transpose?” field that can be used to transpose the matrix before using it[3], a “fun” field which can hold an arbitrary python function built using Tensorflow operations, a “msg” field which defines which message MLP will be used, as well as a “var” field that defines from which vertex type the messages are being sent from.

Note that it expected that the “var” field matches the first vertex type of the message in the “msg” field, and the second vertex type matches the one the update is being applied to. Both the matrices being used as well as the messages must map from the same vertex type and map to the same variable type. If the “mat” field is empty (None) it defaults to an identity matrix, if the “transpose?” field is empty it defaults to False (which does not transpose the matrix), if “fun” is empty (None) no function is applied, if “msg” is empty (None) no message function is applied to that communcation, and if “var” is empty a tensor with values 1 for every vertex is used in its place.

These are all the arguments one may use when building a GNN block with this class (the “__init__” function, lines 5-82). With these arguments the GNN class saves them inside instance attributes and checks for inconsistencies in the definitions passed (with the “check_model” function, lines 84-115, called on line 74) and, if no fatal error was found, builds an instance of itself by instancing its parameters (using the “_init_parameters” function, lines 117-138, called on line 79): The LSTM updates and the MLP messaging functions.

(__call__) When the GNN is called upon to be run (the “__call__” function, in lines 140-195) it expects, one dictionary called “adjacency_matrices” which holds the instances of the matrices whose labels were defined in the “__init__” method , another one called “initial_embeddings” that holds a tensor containing the initial embeddings for every vertex for every type of vertex whose labels were defined in the constructor, an integer “time_steps” which says how many message-passing iterations the GNN block must be run for, and an optional dictionary called “LSTM_initial_states”, where one may also define the initial “C” states for the LSTMs.

With these inputs in hand, the GNN checks if the shapes of the tensors are all consistent by creating a dependency on the assertions that these are in fact equal (the “check_run” function, in lines 197-283, called in line 143 of the “__call__” function), then

[3]This allows us to re-use a matrix when we want to send messages back and forth between two types

from line 146 to line 151 it builds the dictionary containing the de-facto initial states for every vertex type, which it uses in line 187 as an argument to the call to the message-passing while loop. The looping function of the while loop is defined as a closure inside the call to the model, in lines 154-185, and it expects both the current iteration number as well as the states, and it accesses the dictionaries, variables and functions defined for the model through the environment it is scoped into.

This while loop creates a new dictionary containing the new states such that for every vertex type (line 156), it initialises the inputs as an empty list of values (line 157) and for every update dictionary in that vertex type's loop dictionary (line 158) it can process that input in two distinct ways. If the "var" field of the update dictionary is defined (line 159), it picks the embeddings that will be used as input (lines 160-173), passes it through a messaging function, if required (lines 164-165) and, if required, multiplies it by the adjacency matrix that defines the edges alongside between the source and target vertex types (lines 167-172) and finally append it to the update function's list of inputs. Otherwise (lines 175-177) it will simply use the adjacency matrix as input, assuming that it instead contains numeric data (such as features of every node) that will be used by the vertices at every iteration of the message-passing process (line 176) and appends it to the update function's list of inputs.

With the list of inputs to the update function, we aggregate them by concatenating along the embedding axis (line 179) and then call the update function proper on this input and on the outputted vertex embeddings and hidden states of each of the vertices' LSTMs, producing the new states, for that vertex type, for the next message-passing iteration (lines 180-181). Having done this for every vertex type, the while loop returns the new LSTM states and an incremented message-passing iteration number to be used on the next iteration of the loop (line 184).

(check_model) The function used in the class constructor that checks for inconsistencies (line 84-115) checks if any vertex type does not have an update dictionary defined in the loop dictionary and raises a warning if it such vertex type will never be updated in the current definition (lines 86-90). Then it runs through the "loop" dictionary keys and checks if any vertex type that has an update defined there wasn't defined in the "var" dictionary (lines 92-96), throwing an exception if that is the case. It does the same for the "mat" (lines 98-105) and "msg" (lines 107-115) dictionaries, checking if any of the target or source vertex types weren't previously defined, throwing an exception if that is the case. Since this function accesses all dictionaries through the classes' attributes.

(_init_parameters) The function for initialising the parameters used by the network (lines 117-138) simply builds dictionaries containing LSTMs (for the update functions, in lines 119-124) and MLPs (for the messaging functions, in lines 126-137), indexed by the labels of the vertex types which will be updated and the labels for the messaging functions passed through as arguments to the class constructor.

(check_run) Finally, the function that creates the assertions that will be checked before running the message-passing loop itself (lines 197-283). It appends all the assertions in a list that is initially empty (line 198). One of the assertions done is checking that the embedding dimensionality for the tensors of vertex type match the ones defined in the constructor, if the LSTM initial hidden states were passed for that vertex type it checks if dimensionality of the hidden states matches the embedding dimensionality of that vertex type as well, and throughout this it saves how many vertices are defined by the initial embeddings (lines 201-241). Then, it checks if the number of vertices that are defined in the adjacency matrices are consistent with the number expected from the previous information on the initial embeddings and adjacency matrices (lines 243-281). With these checked, the model can be sure that the number of embeddings is consistent and there will be no shape mismatches along message-passing iterations.

(Batching in the GNN library) The GNN library assumes that every tensor containing the vertex embeddings is of shape (b, d), where d is the dimensionality of the embbedings for that specific node type and b is the batch dimension. The first dimension in Tensorflow is commonly considered to be the batch dimension, which is a (generally unspecified) dimension that holds all the values of a batch to be inputted into the model, and as of such it is used as an array of values. In the GNN library, we hold all graphs in a batch in the same batch dimension by joining these graphs into a larger batch-graph whose disjoint connected components are the input graphs themselves.

This technique allows one to input as many graph as can fit on memory through this disjoint union between them, and the adjacency matrices will have no connections between every graph. With this, the message-passing operation will not propagate information from one batched graph to another one, since no messages will be passed between those two.

A.2 Graph Networks

To exemplify the framing of BATTAGLIA et al. (2018)'s model, we show in Code Snippet 2 how one can use the library to make to instantiate a full GN block.

Again, we only allow sum as the reduction operator between the messages, and the update functions are all learned in the format of LSTMs. The "VE" matrix maps each vertex to which edge it is connected, the "VG" and "EG" matrix can be generated procedurally by simply setting all indexes to 1 for every vertex/edge that is is that graph.

Code Snippet 2 A Full Graph Network block from (BATTAGLIA et al., 2018). Source: Author

```
gnn = GNN{
    { "V": d_v, "E": d_e,"G": d_g },
    { "VE": ("V","E"), "VG": ("V","G"), "EG": ("E","G") },
    {   "VmsgE": ("V","E"),
        "EmsgV": ("E","V"),
        "VmsgG": ("V","G"),
        "EmsgG": ("E","G"),
        "GmsgV": ("G","V"),
        "GmsgE": ("G","E"), },
    {   "E": [ { "mat": "VE",
                 "var": "V",
                 "msg": "VmsgE" },
             { "mat": "EG",
                 "transpose?": True,
                 "var": "G",
                 "msg": "GmsgE" } ],
        "V": [ { "mat": "VE",
                 "transpose?": True,
                 "var": "E",
                 "msg": "EmsgV" },
             {   "mat": "VG",
                 "transpose?": True,
                 "var": "G",
                 "msg": "GmsgV" } ],
        "G": [ { "mat": "VG",
                 "var": "V",
                 "msg": "VmsgG" },
             {   "mat": "EG",
                 "var": "E",
                 "msg": "EmsgG" } ] },
    name = "GraphNetwork"
}
```

With this instance in mind one can easily see how to produce similar instances to the other types of neural networks that work on relational data which have seen to be reduced into the Graph Network framework. Therefore with this library one can easily prototype models that work with relational data without having to worry so much with the inner workings of the model itself.

A.3 Conjunctive Normal Form Boolean Satisfiability

SELSAM et al. (2018) used Graph Neural Networks/Message Passing Neural Networks to learn an algorithm for the boolean satisfiability problem and called their model Neurosat. In order to do so, they posed that a conjunctive normal form SAT problem is simply a hypergraph, in which the clauses are the hyperedges and the vertices are

literals, which are joined by each other. This formalisation encodes nicely the relational information of the SAT problem, and allowed his algorithm not only to learn to predict the satisfiability of a formula with a relatively high accuracy but for the network to be able to learn a representation, and an algorithm, whose latent information can be extracted to produce literal assignments as well. The GNN core of his Neurosat model can be synthetised in the GNN library as in Code Snippet 3.

Code Snippet 3 An instance of a GNN block for solving boolean satisfiability, following closely (SELSAM et al., 2018). Source: Author

```
gnn = GNN(
  {
    "L": d_l,
    "C": d_c
  },
  {
    "LL": ("L","L"),
    "LC": ("L","C")
  },
  {
    "L_msg_C": ("L","C"),
    "C_msg_L": ("C","L")
  },
  {
    "L": [
      {
        "mat": "LC",
        "msg": "L_msg_V",
        "transpose?": True,
        "var": "L"
      },
      {
        "mat": "LL",
        "var": "L"
      }
    ],
    "C": [
      {
        "mat": "LC",
        "msg": "L_msg_C",
        "var": "L"
      }
    ]
  },
  name = "SAT"
)
```

The Neurosat model learns an initial embedding for the literal-vertices and one for the clause-vertices and uses the same initial embedding for all vertices of each type. Then, after the message-passing iterations it runs every literal's embedding through a "vote" MLP, which produces a single value with which each literal votes on whether the formula is satisfiable or not. These votes are then averaged on each formula as the overall vote for the on the satisfiability of the problem itself.

A.4 Decision Travelling Salesperson Problem

In (PRATES et al., 2019a), the authors used this library to build an approximate solver for the decision version of the Travelling Salesperson Problem. One can see how the GNN core of their work was instanced using the GNN library in Code Snipper 4.

Code Snippet 4 An instance of a GNN for solving the decision TSP, as explained in (PRATES et al., 2019a). Source: (PRATES et al., 2019a)

```
gnn = GNN(
  {
    'V': d_v,
    'E': d_e
  },
  {
    'EV': ('E','V')
  },
  {
    'V_msg_E': ('V','E'),
    'E_msg_V': ('E','V')
  },
  {
    'V': [
      {
        'mat': 'EV',
        'msg': 'E_msg_V',
        'transpose?': True,
        'var': 'E'
      }
    ],
    'E': [
      {
        'mat': 'EV',
        'msg': 'V_msg_E',
        'var': 'V'
      }
    ]
  },
  name = "TSP"
)
```

In PRATES et al.'s model, the initial vertex-vertices' embeddings are learnable parameters, but the edge-vertices' embeddings are initialised through an MLP that receives the target-cost for the route as well as the edge's weight as input. The end of the model's pipeline is much like the one in (SELSAM et al., 2018), except that the edge-vertices vote on whether there exists a route or not, instead of the vertex-vertices being the voters (one can see the clauses as hyperedges between the literals, thus the analogy drawn here).

A.5 gnn.py

```
import tensorflow as tf
from mlp import Mlp
```

```
3
4  class GNN(object):
5    def __init__(
6      self,
7      var,
8      mat,
9      msg,
10     loop,
11     MLP_depth = 3,
12     MLP_weight_initializer = tf.contrib.layers.xavier_initializer,
13     MLP_bias_initializer = tf.zeros_initializer,
14     RNN_cell = tf.contrib.rnn.LayerNormBasicLSTMCell,
15     Cell_activation = tf.nn.relu,
16     Msg_activation = tf.nn.relu,
17     Msg_last_activation = None,
18     float_dtype = tf.float32,
19     name = 'GNN'
20   ):
21
22     """
23     Receives four dictionaries: var, mat, msg and loop.
24
25     * var is a dictionary from variable names to embedding sizes.
26       That is: an entry var["V1"] = 10 means that the variable "V1" will have an
       embedding size of 10.
27
28     * mat is a dictionary from matrix names to variable pairs.
29       That is: an entry mat["M"] = ("V1","V2") means that the matrix "M" can be used
       to mask messages from "V1" to "V2".
30
31     * msg is a dictionary from function names to variable pairs.
32       That is: an entry msg["cast"] = ("V1","V2") means that one can apply "cast" to
       convert messages from "V1" to "V2".
33
34     * loop is a dictionary from variable names to lists of dictionaries:
35       {
36         "mat": the matrix name which will be used,
37         "transpose?": if true then the matrix M will be transposed,
38         "fun": transfer function (python function built using tensorflow operations,
39         "msg": message name,
40         "var": variable name
41       }
42       If "mat" is None, it will be the identity matrix,
43       If "transpose?" is None, it will default to false,
44       if "fun" is None, no function will be applied,
45       If "msg" is false, no message conversion function will be applied,
46       If "var" is false, then [1] will be supplied as a surrogate.
47
48       That is: an entry loop["V2"] = [ {"mat":None,"fun":f,"var":"V2"}, {"mat":"M","
       transpose?":true,"msg":"cast","var":"V1"} ] enforces the following update rule
       for every timestep:
49         V2 = tf.append( [ f(V2), matmul( M.transpose(), cast(V1) ) ] )
```

```
50
51      You can also specify:
52
53      * MLP_depth, which indicates how many layers before the output layer each message
         MLP will have, defaults to 3.
54      * MLP_weight_initializer, which indicates which initializer is to be used on the
        MLP layers' kernels, defaults to tf.contrib.layers.xavier_initializer.
55      * MLP_bias_initializer, which indicates which initializer is to be used on the
        MLP layers' biases, defaults to tf.zeros_initializer.
56      * Cell_activation, which indicates which activation function should be used on
        the LSTM cell, defaults to tf.nn.relu.
57      * Msg_activation, which indicates which activation function should be used on the
         hidden layers of the MLPs, defaults to tf.nn.relu.
58      * Msg_last_activation, which indicates which acitvation function should be used
        on the output, defaults to None (linear activation).
59      * float_dtype, which indicates which float type should be used (not tested with
        others than tf.float32), defaults to tf.float32.
60      * name, which is the scope name that the GNN will use to declare its parameters
        and execution graph, defaults to 'GNN'.
61      """
62      self.var, self.mat, self.msg, self.loop, self.name = var, mat, msg, loop, name
63
64      self.MLP_depth = MLP_depth
65      self.MLP_weight_initializer = MLP_weight_initializer
66      self.MLP_bias_initializer = MLP_bias_initializer
67      self.RNN_cell = RNN_cell
68      self.Cell_activation = Cell_activation
69      self.Msg_activation = Msg_activation
70      self.Msg_last_activation  = Msg_last_activation
71      self.float_dtype = float_dtype
72
73      # Check model for inconsistencies
74      self.check_model()
75
76      # Initialize the parameters
77      with tf.variable_scope(self.name):
78        with tf.variable_scope('parameters'):
79          self._init_parameters()
80        #end parameter scope
81      #end GNN scope
82    #end __init__
83
84    def check_model(self):
85      # Procedure to check model for inconsistencies
86      for v in self.var:
87        if v not in self.loop:
88          raise Warning('Variable {v} is not updated anywhere! Consider removing it
        from the model'.format(v=v))
89        #end if
90      #end for
91
92      for v in self.loop:
```

```
93        if v not in self.var:
94          raise Exception('Updating variable {v}, which has not been declared!'.format(
      v=v))
95        #end if
96      #end for
97
98      for mat, (v1,v2) in self.mat.items():
99        if v1 not in self.var:
100         raise Exception('Matrix {mat} definition depends on undeclared variable {v}'.
      format(mat=mat, v=v1))
101       #end if
102       if v2 not in self.var and type(v2) is not int:
103         raise Exception('Matrix {mat} definition depends on undeclared variable {v}'.
      format(mat=mat, v=v2))
104       #end if
105     #end for
106
107     for msg, (v1,v2) in self.msg.items():
108       if v1 not in self.var:
109         raise Exception('Message {msg} maps from undeclared variable {v}'.format(msg=
      msg, v=v1))
110       #end if
111       if v2 not in self.var:
112         raise Exception('Message {msg} maps to undeclared variable {v}'.format(msg=
      msg, v=v2))
113       #end if
114     #end for
115   #end check_model
116
117   def _init_parameters(self):
118     # Init LSTM cells
119     self._RNN_cells = {
120       v: self.RNN_cell(
121        d,
122        activation = self.Cell_activation
123       ) for (v,d) in self.var.items()
124     }
125     # Init message-computing MLPs
126     self._msg_MLPs = {
127       msg: Mlp(
128         layer_sizes          = [ self.var[vin] for _ in range( self.MLP_depth ) ],
129         output_size          = self.var[vout],
130         activations          = [ self.Msg_activation for _ in range( self.MLP_depth )
       ],
131         output_activation    = self.Msg_last_activation,
132         kernel_initializer   = self.MLP_weight_initializer(),
133         bias_initializer     = self.MLP_weight_initializer(),
134         name                 = msg,
135         name_internal_layers = True
136       ) for msg, (vin,vout) in self.msg.items()
137     }
138   #end _init_parameters
```

```
139
140   def __call__( self, adjacency_matrices, initial_embeddings, time_steps,
        LSTM_initial_states = {} ):
141     with tf.variable_scope(self.name):
142       with tf.variable_scope( "assertions" ):
143         assertions = self.check_run( adjacency_matrices, initial_embeddings,
        time_steps, LSTM_initial_states )
144       #end assertion variable scope
145       with tf.control_dependencies( assertions ):
146         states = {}
147         for v, init in initial_embeddings.items():
148           h0 = init
149           c0 = tf.zeros_like(h0, dtype=self.float_dtype) if v not in
        LSTM_initial_states else LSTM_initial_states[v]
150           states[v] = tf.contrib.rnn.LSTMStateTuple(h=h0, c=c0)
151         #end
152
153         # Build while loop body function
154         def while_body( t, states ):
155           new_states = {}
156           for v in self.var:
157             inputs = []
158             for update in self.loop[v]:
159               if 'var' in update:
160                 y = states[update['var']].h
161                 if 'fun' in update:
162                   y = update['fun'](y)
163                 #end if
164                 if 'msg' in update:
165                   y = self._msg_MLPs[update['msg']](y)
166                 #end if
167                 if 'mat' in update:
168                   y = tf.matmul(
169                     adjacency_matrices[update['mat']],
170                     y,
171                     adjoint_a = update['transpose?'] if 'transpose?' in update else
        False
172                     )
173                 #end if
174                 inputs.append( y )
175               else:
176                 inputs.append( adjacency_matrices[update['mat']] )
177               #end if var in update
178             #end for update in loop
179             inputs = tf.concat( inputs, axis = 1 )
180             with tf.variable_scope( '{v}_cell'.format( v = v ) ):
181               _, new_states[v] = self._RNN_cells[v]( inputs = inputs, state = states[
        v] )
182             #end cell scope
183           #end for v in var
184           return (t+1), new_states
185         #end while_body
```

```
186
187         _, last_states = tf.while_loop(
188           lambda t, states: tf.less( t, time_steps ),
189           while_body,
190           [0,states]
191         )
192       #end assertions
193     #end Graph scope
194     return last_states
195   #end __call__
196
197   def check_run( self, adjacency_matrices, initial_embeddings, time_steps,
      LSTM_initial_states ):
198     assertions = []
199     # Procedure to check model for inconsistencies
200     num_vars = {}
201     for v, d in self.var.items():
202       init_shape = tf.shape( initial_embeddings[v] )
203       num_vars[v] = init_shape[0]
204       assertions.append(
205         tf.assert_equal(
206           init_shape[1],
207           d,
208           data = [ init_shape[1] ],
209           message = "Initial embedding of variable {v} doesn't have the same
      dimensionality {d} as declared".format(
210             v = v,
211             d = d
212           )
213         )
214       )
215       if v in LSTM_initial_states:
216         lstm_init_shape = tf.shape( LSTM_initial_states[v] )
217         assertions.append(
218           tf.assert_equal(
219             lstm_init_shape[1],
220             d,
221             data = [ lstm_init_shape[1] ],
222             message = "Initial hidden state of variable {v}'s LSTM doesn't have the
      same dimensionality {d} as declared".format(
223               v = v,
224               d = d
225             )
226           )
227         )
228
229         assertions.append(
230           tf.assert_equal(
231             lstm_init_shape,
232             init_shape,
233             data = [ init_shape, lstm_init_shape ],
234             message = "Initial embeddings of variable {v} don't have the same shape
```

```
        as the its LSTM's initial hidden state".format(
235               v = v,
236               d = d
237             )
238           )
239         )
240       #end if
241     #end for v
242
243     for mat, (v1,v2) in self.mat.items():
244       mat_shape = tf.shape( adjacency_matrices[mat] )
245       assertions.append(
246         tf.assert_equal(
247           mat_shape[0],
248           num_vars[v1],
249           data = [ mat_shape[0], num_vars[v1] ],
250           message = "Matrix {m} doesn't have the same number of nodes as the initial
        embeddings of its variable {v}".format(
251             v = v1,
252             m = mat
253           )
254         )
255       )
256       if type(v2) is int:
257         assertions.append(
258           tf.assert_equal(
259             mat_shape[1],
260             v2,
261             data = [ mat_shape[1], v2 ],
262             message = "Matrix {m} doesn't have the same dimensionality {d} on the
        second variable as declared".format(
263               m = mat,
264               d = v2
265             )
266           )
267         )
268       else:
269         assertions.append(
270           tf.assert_equal(
271             mat_shape[1],
272             num_vars[v2],
273             data = [ mat_shape[1], num_vars[v2] ],
274             message = "Matrix {m} doesn't have the same number of nodes as the
        initial embeddings of its variable {v}".format(
275               v = v2,
276               m = mat
277             )
278           )
279         )
280       #end if-else
281     #end for mat, (v1,v2)
282     return assertions
```

```
283   #end check_run
284 #end GNN
```

A.6 mlp.py

```
1  import tensorflow as tf
2
3  class Mlp(object):
4    def __init__(
5      self,
6      layer_sizes,
7      output_size = None,
8      activations = None,
9      output_activation = None,
10     use_bias = True,
11     kernel_initializer = None,
12     bias_initializer = tf.zeros_initializer(),
13     kernel_regularizer = None,
14     bias_regularizer = None,
15     activity_regularizer = None,
16     kernel_constraint = None,
17     bias_constraint = None,
18     trainable = True,
19     name = None,
20     name_internal_layers = True
21   ):
22     """Stacks len(layer_sizes) dense layers on top of each other, with an additional
       layer with output_size neurons, if specified."""
23     self.layers = []
24     internal_name = None
25     # If object isn't a list, assume it is a single value that will be repeated for
       all values
26     if not isinstance( activations, list ):
27       activations = [ activations for _ in layer_sizes ]
28     #end if
29     # If there is one specifically for the output, add it to the list of layers to be
        built
30     if output_size is not None:
31       layer_sizes = layer_sizes + [output_size]
32       activations = activations + [output_activation]
33     #end if
34     for i, params in enumerate( zip( layer_sizes, activations ) ):
35       size, activation = params
36       if name_internal_layers:
37         internal_name = name + "_MLP_layer_{}".format( i + 1 )
38       #end if
39       new_layer = tf.layers.Dense(
40         size,
41         activation = activation,
42         use_bias = use_bias,
43         kernel_initializer = kernel_initializer,
```

```
44          bias_initializer = bias_initializer,
45          kernel_regularizer = kernel_regularizer,
46          bias_regularizer = bias_regularizer,
47          activity_regularizer = activity_regularizer,
48          kernel_constraint = kernel_constraint,
49          bias_constraint = bias_constraint,
50          trainable = trainable,
51          name = internal_name
52        )
53        self.layers.append( new_layer )
54      #end for
55    #end __init__
56
57    def __call__( self, inputs, *args, **kwargs ):
58      outputs = [ inputs ]
59      for layer in self.layers:
60        outputs.append( layer( outputs[-1] ) )
61      #end for
62      return outputs[-1]
63    #end __call__
64  #end Mlp
```