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

MARCELO DE OLIVEIRA ROSA PRATES

Learning to Solve $\mathcal{NP}$-Complete Problems

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

Advisor: Prof. Dr. Luís da Cunha Lamb

Porto Alegre
August 2019

CIP — CATALOGING-IN-PUBLICATION

Prates, Marcelo de Oliveira Rosa

Learning to Solve $\mathcal{NP}$-Complete Problems / Marcelo de Oliveira Rosa Prates. – Porto Alegre: PPGC da UFRGS, 2019.

149 f.: il.

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

1. Artificial Neural Network. 2. Deep Learning. 3. Graph Neural Network. 4. Typed Graph Network. 5. Neural Symbolic Reasoning. 6. Traveling Salesman Problem. I. Lamb, Luís da Cunha. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof^{a}. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof^{a}. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ACKNOWLEDGEMENTS

The PhD program is a long and trying road and being able to finish it brings me much joy. I'm grateful to many people inside and outside academia that this thesis was possible. For once, I'm indebted to Henrique Lemos for his insights, companionship and for the works we have developed together towards the end of my doctoral program. I understand that although many people have had their role in the fulfillment of my goal, only a couple were decisive. I owe to Pedro Avelar the greater part of my fascination by deep learning, and this thesis was only really possible because of the time we shared together studying and researching geometric deep learning. Finally, I am extremely grateful to my advisor Luis Lamb for believing in me even when I did not, and for offering me the liberty and the encouragement to carry out research on many varied subjects. To him I owe a plurality in my training as a researcher which is deeply valuable to me.

ABSTRACT

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 thesis, we propose a new formulation for GNNs, which employs the concept of "types" to partition the objects in the problem domain into many distinct classes, yielding the Typed Graph Networks (TGN) model and a Python / Tensorflow library for prototyping TGNs. This thesis is also concerned with the application of GNNs to the Traveling Salesperson Problem (TSP). We show that GNNs can learn to solve, with very little supervision, the decision variant of the TSP, a highly relevant $\mathcal{NP}$-Complete problem. Our model is trained to function as an effective message-passing algorithm in graph in which edges from the input graph communicate with vertices from the input graph for a number of iterations after which the model is asked to decide whether a route with cost $< C \in \mathbb{R}_0^+$ 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 (C) $x\%$ 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 1.5% relative 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, in addition to proposing a modern reformulation of the meta-model.

Keywords: Artificial Neural Network. Deep Learning. Graph Neural Network. Typed Graph Network. Neural Symbolic Reasoning. Traveling Salesman Problem.

Aprendendo a Resolver Problemas $\mathcal{NP}$-Completos

RESUMO

Graph Neural Networks (GNN) constituem uma técnica promisora para conectar programação diferencial e domínios combinatoriais. GNNs lançam mão de módulos treináveis os quais podem ser montados em diferentes configurações, cada uma refletindo a estrutura relacional de uma instância específica. Nessa tese, nós propomos uma nova formulação para GNNs, a qual faz uso do conceito de "tipos" para particionar os objetos no domínio do problema em múltiplas classes distintas, resultando no modelo das Typed Graph Networks (TGN) e numa biblioteca Python / Tensorflow para prototipar TGNs. Esta tese também se preocupa com a aplicação de GNNs no Problema do Caixeiro(a) Viajante (PCV). Nós mostramos que GNNs são capazes de aprender a resolver, com pouquíssima supervisão, a variante de decisão do PCV, um problema $\mathcal{NP}$-Completo altamente relevante. Nosso modelo é treinado para funcionar, efetivamente, como um algoritmo de troca de mensagens em grafos no qual as arestas do grafo de entrada comunicam-se com os vértices do grafo de entrada por um determinado número de iterações, depois do qual o modelo é forçado a responder se o grafo de entrada admite ou não uma rota Hamiltoniana de custo $< C \in \mathbb{R}_0^+$. Nós mostramos que esta rede pode ser treinada com conjuntos de exemplos duais: dado o custo ótimo C^*, produzimos uma instância de decisão com custo alvo (C) $x\%$ menor e uma com custo alvo $x\%$ maior do que C^*. Nós fomos capazes de obter 80% de acurácia treinando o modelo com desvios de $-2\%, +2\%$, e o mesmo modelo treinado foi capaz de generalizar para desvios mais relaxados com melhor performance. Também mostramos que o modelo é capaz de generalizar para problemas maiores. Finalmente, nós oferecemos um método para predizer o custo de rota ótimo dentro de 1.5% de desvio relativo para o valor real. Em resumo, nosso trabalho demonstra que GNNs são suficientemente poderosas para resolver problems $\mathcal{NP}$-Completos que combinam dados simbólicos e numéricos, além de propor uma reformulação moderna do meta-modelo.

Palavras-chave: Rede Neural Artificial, Deep Learning, Graph Neural Network, Typed Graph Network, Raciocínio Neural Simbólico, Problema do Caxeiro(a) Viajante.

LIST OF ABBREVIATIONS AND ACRONYMS

GNN	Graph Neural Network
GCN	Graph Convolutional Neural Network
GRN	Graph Recurrent Neural Network
GAT	Graph Attention Network
ANN	Artificial Neural Network
MLP	Multilayer Perceptron
LSTM	Long Short-Term Memory
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
GAN	Generative Adversarial Network
ReLu	Rectified Linear Unit
TSP	Traveling Salesperson Problem
ML	Machine Learning
DL	Deep Learning
TSP	Traveling Salesman Problem
TGN	Typed Graph Network
AI	Artificial Intelligence
CNF	Conjunctive Normal Form
SAT	Boolean Satisfiability Problem

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Deep learning is rapidly pushing the state of the art in artificial intelligence, with significant advances in many areas including but not limited to computer vision, speech recognition, game playing, natural language processing and bioinformatics. However, the black-box nature of DL systems – which lack a solid theoretical background – creates a divide between them and symbolic AI techniques. It has been argued that DL provides a way to implement "fast thinking" or intuition into machines, while formal logic provides reliable, explainable "slow thinking". Combining fast (DL) and slow (logics) thinking is rapidly becoming a frontier in AI research, with recent projects such as Vadalog aiming to stack a layer of logical deduction on top of associations learned by ML technology (BELLOMARINI; SALLINGER; GOTTLOB, 2018). Intelligent machines should be able not only to learn from experience but crucially to reason about what was learned. In this context, commiting to combinatorial generalization – the ability to learn associations between elements on discrete, symbolic domains – is regarded as a key path forward for AI research today (BATTAGLIA et al., 2018).

In this context, Graph Neural Networks (GNNs) – end-to-end differentiable neural models that learn on relational structure – open exciting new possibilities. The key insight behind GNNs is that there are advantages to not forcing the ANN architecture to be fixed. One can in principle instantiate a small number of trainable neural modules and assemble them in different configurations, the result of which is a set of ANN architectures of combinatorial size all of which are parameterized by the same neural weights. Fundamentally, this allows one to feed an ANN with relational rather than numerical data: the input for a GNN is a graph. Because all combinatorial problems can be expressed in the language of graphs, GNNs offer a way to build models that learn from symbolic data. A CNF formula, for example, can be expressed as a graph in which clauses are connected to the literals pertaining to it. This insight has been applied to the development of NeuroSAT, a GNN which becomes capable of predicting the satisfiability of a CNF formula upon training with randomly-generated instances (SELSAM et al., 2018).

The NeuroSAT experiment was the first to show that GNNs in particular and ANNs in general can solve classical $\mathcal{NP}$-Complete problems. It also raised a number of exciting research directions: when trained with the proposed dataset of

CNF formulas, the NeuroSAT architecture is able to achieve something unheard of in deep learning: the accuracy of the model increases monotonically (although with diminishing gains) with the number of iterations of message-passing (the core operation of GNNs). Additionally, the authors show that even though the model was only trained as a SAT predictor, it nevertheless generates satisfying assignments which can be easily extracted from the network's "memory". Overall, this experiment is an invitation to a deeper investigation on GNNs, which have shown an impressive performance in the recent years (BATTAGLIA et al., 2018) and whose rapid growth suggests they may very well be the "next CNN" in terms of popularity.

In this work, we investigate whether GNNs can solve conceptually harder $\mathcal{NP}$-Complete problems in which symbolic/relational information is combined with numerical information. In this context, the Traveling Salesperson Problem is a prime candidate, both for its relevance in computer science and for the fact that in its general definition edges are labelled with numerical information in the form of weights. We propose a novel GNN architecture to solve the decision variant of the TSP, which relies on the insight of mapping the original graph into an abstract graph representation in which edges are elevated to the status of nodes and can be embedded with their weights. We propose a technique for training this model in which it is fed with two copies of the same graph with a small variation in the target cost X (as in "does graph $\mathcal{G}$ admit a Hamiltonian path with cost $\leq X$?") such that one copy is answered correctly with NO and the other with YES. With this, we were able to train the model to predict the decision problem within a $-2\%, +2\%$ relative deviation from the optimal tour cost. We evaluate our model on different datasets, providing evidence that it is able to generalize (with diminishing accuracy) to other graph distributions and larger problem sizes.

1.1 Research Questions

Combining symbolic reasoning with deep learning is an important and mostly unexplored frontier in AI (BATTAGLIA et al., 2018). GNNs – neural computers which work on relational data – show promise in this direction, but their applicability and operation are still poorly understood. (SELSAM et al., 2018) have shown that GNNs can tackle boolean satisfiability, a hard ($\mathcal{NP}$-Complete) combinatorial problem. The main research question we are interested in is assessing to which extent GNNs can solve conceptually harder $\mathcal{NP}$-Complete problems which combine relational and numerical information. In this context, we want to assess whether GNNs can be trained to solve the decision variant of Traveling Salesperson Problem (TSP) as a case example.

A second issue is the explainability of GNNs. (SELSAM et al., 2018) has shown that the algorithm learned by a GNN can be amenable to investigation, and that some important facts about how a trained GNN processes data can be deduced from the set of refined embeddings. We want to understand under which conditions the choice of the training data can affect the learned algorithm. Most importantly, we want to understand how a GNN defined on a decision problem can be trained to learn a *constructive* algorithm – that is, how can it be trained in such a way that a solution can be extracted from the refined embeddings.

1.1.1 Specific Goals

1. To investigate the extent to which GNNs can be applied to $\mathcal{NP}$-Complete problems which combine relational and numerical information.
2. To propose and train a GNN architecture to solve the decision variant of the Traveling Salesperson Problem.
3. To evaluate the effects of training/evaluating the model with larger/smaller relative deviations from the optimal cost.
4. To evaluate the model's performance on different graph distributions and larger problem sizes compared to what it was trained with.
5. To understand under which conditions the model can be trained to provide constructive solutions (i.e. such that we can extract a Hamiltonian route from the network's memory).
6. To evaluate the effects of training the model with different (possibly parameterized) graph distributions, such as varying the network connectivity.
7. To assess the "data hunger" of our model: because we can create our training instances, we can evaluate how many training examples are required for the model to generalize well.

1.2 Our Contributions

During the course of the PhD program, the author has authored and co-authored a number of scientific papers, spanning three main subjects: 1) the ethics of artificial intelligence 2) phase transitions on hard computational problems and 3) neural symbolic reasoning on graph neural networks. These papers are summarized in this section.

1.2.1 Problem Solving at the Edge of Chaos: Entropy, Puzzles and the Sudoku Freezing Transition – Marcelo Prates, Luis Lamb

Published at ICTAI 2018 (Qualis A2) (PRATES; LAMB, 2018a). Also available as a preprint on <arxiv.org> (PRATES; LAMB, 2018b). Prates was responsible for implementing the Sudoku Gecode solver and adapding the QQWing Sudoku solver for the purposes of the study, as well as implementing the code for carrying out the phase transition analysis. Prates was also responsible for the greater share of the writing of the manuscript, while Lamb contributed with supervision during the structuring of the project and technical and scientific suggestions during the experimental setup and analysis, as well as the revision and writing of portions of the manuscript.

Sudoku is a widely popular $\mathcal{NP}$-Complete combinatorial puzzle whose prospects for studying human computation have recently received attention, but the algorithmic hardness of Sudoku solving is yet largely unexplored. In this paper, we study the statistical mechanical properties of random Sudoku grids, showing that puzzles of varying sizes attain a hardness peak associated with a critical behavior in the constrainedness of random instances. In doing so, we provide the first description of a Sudoku *freezing* transition, showing that the fraction of backbone variables undergoes a phase transition as the density of pre-filled cells is calibrated. We also uncover a variety of critical phenomena in the applicability of Sudoku elimination strategies, providing explanations as to why puzzles become boring outside the typical range of clue densities adopted by Sudoku publishers. We further show that the constrainedness of Sudoku puzzles can be understood in terms of the informational (Shannon) entropy of their solutions, which only increases up to the critical point where variables become frozen. Our findings shed light on the nature of the k-coloring transition when the graph topology is fixed, and are an invitation to the study of phase transition phenomena in problems defined over *alldifferent* constraints. They also suggest advantages to studying the statistical mechanics of popular $\mathcal{NP}$-Hard puzzles, which can both aid the design of hard instances and help understand the difficulty of human problem solving.

1.2.2 Neural Networks Models for Analyzing Magic: the Gathering Cards – Felipe Zilio, Marcelo Prates, Luis Lamb

Published at ICONIP 2018 (Qualis B2) (ZILIO; PRATES; LAMB, 2018). Also avaliable as a preprint on <arxiv.org> (**??**). This paper is the fruit of a Bachelor Thesis Prates has co-advised with Lamb. Zilio was responsible for implementing the code, carrying out the experiments and writing the greater share of the manuscript's text. Prates and Lamb contributed by pointing out this specific research direction (combining the hot topic of deep learning with Zilio's personal interest for MTG), as well as proof-reading and performing revisions to Zilios Bachelor's thesis. Prates additionally contributed by adapting Zilio's thesis into this publication.

> Historically, games of all kinds have often been the subject of study in scientific works of Computer Science, including the field of machine learning. By using machine learning techniques and applying them to a game with defined rules or a structured dataset, it's possible to learn and improve on the already existing techniques and methods to tackle new challenges and solve problems that are out of the ordinary. The already existing work on card games tends to focus on gameplay and card mechanics. This work aims to apply neural networks models, including Convolutional Neural Networks and Recurrent Neural Networks, in order to analyze Magic: the Gathering cards, both in terms of card text and illustrations; the card images and texts are used to train the networks in order to be able to classify them into multiple categories. The ultimate goal was to develop a methodology that could generate card text matching it to an input image, which was attained by relating the prediction values of the images and generated text across the different categories.

1.2.3 On Quantifying and Understanding the Role of Ethics in AI Research: A Historical Account of Flagship Conferences and Journals – Marcelo Prates, Pedro Avelar, Luis Lamb

Published and presented at GCAI 2018 (Qualis unavailable) (PRATES; AVELAR; LAMB, 2018). Also available as a preprint on <arxiv.org> (PRATES; AVELAR; LAMB, 2018). Lamb is credited with the idea of performing a corpus-based analysis on the subject. Avelar is credited with implementing the web scraper to obtain paper titles and abstracts for the selected conferences and journals, as well as performing the statistical tests described at the end of the paper. Prates is credited with performing the greater share of the analysis as well as writing the greater part of the manuscript's text. Prates and Lamb were responsible for the greater share of the literature review present in the paper. Prates, Avelar and Lamb contributed equally with hypothesis and experimental setup scenarios.

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.

1.2.4 Assessing Gender Bias in Machine Translation – A Case Study with Google Translate – Marcelo Prates, Pedro Avelar, Luis Lamb

Accepted for publication at Neural Computing and Applications (Journal, Qualis B1). Also available as a preprint on <arXiv.org> (PRATES; AVELAR; LAMB,). Prates was responsible for designing the experimental setup, implementing the code for carrying out the analyzes in the paper and writing the entirety of its first version, subject to revisions and suggestions by Lamb. Upon rejection, Avelar was included as a co-author and was responsible for obtaining a respected source (in the U.S. Bureau for Labor Statistics) for the job occupations analyzed in the experiments, as well as contributing with revisions to the manuscript's text. This work has received substantial attention in the local and international media.

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.

1.2.5 Multitask Learning on Graph Neural Networks – Learning Multiple Graph Centrality Measures with a Unified Network – Pedro Avelar, Marcelo Prates, Henrique Lemos, Luis Lamb

Accepted for publication at the 28th International Conference on Artificial Neural Networks (ICANN). Also available as a preprint on <arXiv.org> (AVELAR et al., 2018). Prates and mostly Avelar were responsible for the development of the Python / Tensorflow library used to implement the model described in this paper. The idea for this particular model is a joint contribution of Prates and mostly Avelar. The initial code for this model was implemented by Avelar, while Lemos was responsible for analyzing it, debugging it and ultimately getting it to work, while additionally carrying out the analyzes described in the paper. The experimental setup is mostly due to Avelar. Overall, Prates, Avelar and Lamb were responsible for the greater share of the text, in decreasing order of contribution.

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 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.

1.2.6 Learning to Solve NP-Complete Problems – A Graph Neural Network for the Decision TSP – Marcelo Prates, Pedro Avelar, Henrique Lemos, Luis Lamb and Moshe Vardi

Presented at AAAI 2019 (Qualis A1). Main author. Also available as a preprint on <arXiv.org> (PRATES et al., 2018). Prates and mostly Avelar were responsible for the development of the Python / Tensorflow library used to implement the model described in this paper. Prates is credited with the design of the model, including the original idea of projecting edges to multidimensional space to feed it with edge weight data. The experimental setup, including the idea of developing adversarial instance pairs, is also due to Prates. Avelar is credited with the idea and carrying out of the “acceptance curves” experiments, while Lemos is credited with carrying out baseline comparisons with traditional methods. Prates is responsible for the greater share of the manuscript’s text, although Avelar, Lemos and mostly Lamb contributed with revisions. Lamb and Vardi are credited with suggesting and advocating for the research direction of approaching traditional $\mathcal{NP}$-Complete problems with neural networks.

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.

1.2.7 Typed Graph Networks – Marcelo Prates, Pedro Avelar, Henrique Lemos, Luis Lamb, Marco Gori

Under review at IEEE Transactions on Neural Networks and Learning Systems. Available as a preprint on <arxiv.org> (PRATES et al., 2019). Prates and mostly Avelar are responsible for the development of the Python / Tensorflow library which is a companion to and one of the two main subjects of this paper. Prates is credited with the original idea of formalizing Graph Neural Networks in the setting of *types*, yielding the Typed Graph Networks formalization. Prates is responsible for the greater share of the manuscript's text, including algorithms, visualizations and figures. Avelar is credited with the block diagram illustrating the structure of a TGN. Prates is credited with implementing one, Avelar two and Lemos one of the models described in the paper. Lamb is credited with the suggestion of turning the TGN formalization (originally developed for this thesis) into a paper. Lamb and Gori contributed with revisions to the manuscript text. Gori is credited with the development of the original GNN model and unique historical perspective on the differences between it and our modern formalization.

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.

1.2.8 Graph Colouring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems – Henrique Lemos, Marcelo Prates, Pedro H.C. Avelar and Luis C. Lamb

Under review at The IEEE International Conference on Tools with Artificial Intelligence (ICTAI). Available as a preprint on <arxiv.org> (LEMOS et al., 2019). Prates and mostly Avelar are responsible for the development of the Python / Tensorflow library which was used to write the models proposed in this paper. Lemos is credited with the original idea of applying GNNs to graph colouring, with the experimental design and with the models themselves, as well as the greater part of the writing of the manuscript. Prates, Avelar and Lamb contributed with revisions to the text.

Deep learning has consistently defied state-of-the-art techniques in many fields over the last decade. However, we are just beginning to understand the capabilities of neural learning in symbolic domains. Deep learning architectures that employ parameter sharing over graphs can produce models which can be trained on complex properties of relational data. These include highly relevant $\mathcal{NP}$-Complete problems, such as SAT and TSP. In this work, we showcase how Graph Neural Networks (GNN) can be engineered – with a very simple architecture – to solve the fundamental combinatorial problem of graph colouring. Our results show that the model, which achieves high accuracy upon training on random instances, is able to generalise to graph distributions different from those seen at training time. Further, it performs better than the Neurosat, Tabucol and greedy baselines for some distributions. In addition, we show how vertex embeddings can be clustered in multidimensional spaces to yield constructive solutions even though our model is only trained as a binary classifier. In summary, our results contribute to shorten the gap in our understanding of the algorithms learned by GNNs, as well as hoarding empirical evidence for their capability on hard combinatorial problems. Our results thus contribute to the standing challenge of integrating robust learning and symbolic reasoning in Deep Learning systems.

1.2.9 Graph Neural Networks Improve Link Prediction on Knowledge Graphs – Henrique Lemos, Marcelo Prates, Pedro H.C. Avelar and Luis C. Lamb

Under review at 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP). Prates and mostly Avelar are responsible for the development of the Python / Tensorflow library which was used to write the models proposed in this paper. Lemos is credited with the original[1] idea of applying GNNs to knowledge link prediction, with the experimental design and with the models themselves, as well as the greater part of the writing of the manuscript. Prates contributed with the formalization of the problem and model, writing of some sections and decisions regarding the presentation of diagrams. Prates, Avelar and Lamb contributed with revisions to the text.

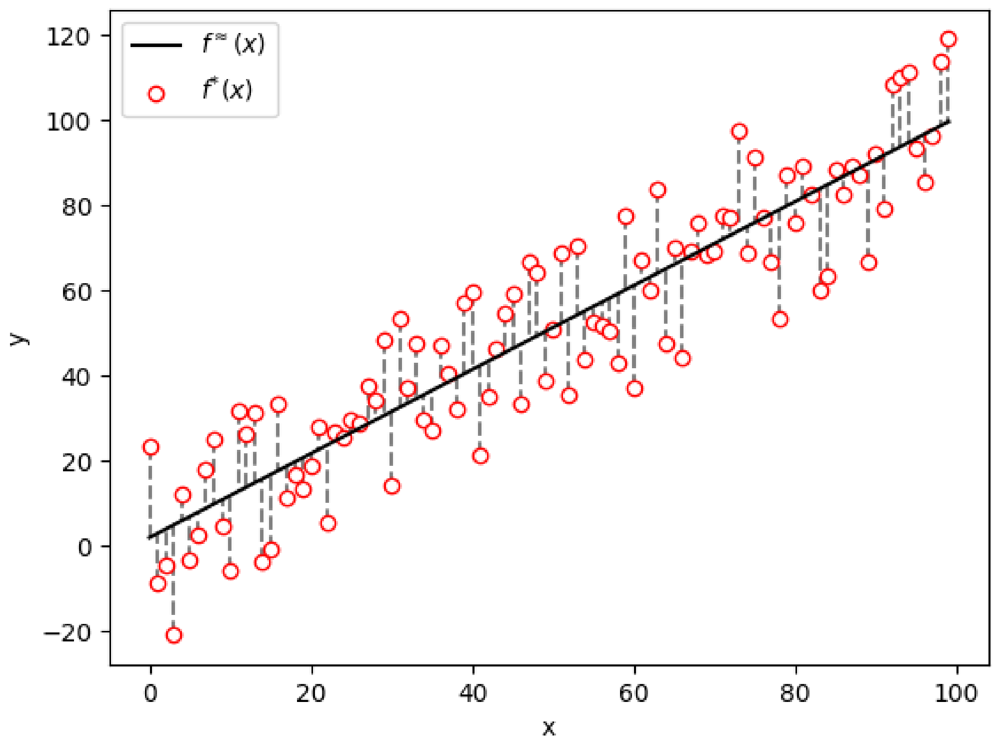
Knowledge graphs are repositories of *facts* about real-world entities. Each *fact* is structured as a 3-tuple connecting a source entity and a target entity with a relation. However these repositories can be incomplete, with several missing facts about the stored entities. Recently, there have been attempts to infer missing facts from existent ones, which can be modelled as a machine learning task. Two notable research efforts try to predict a new fact between two entities e_s and e_t by considering (i) all known facts connecting e_s and e_t and (ii) several *paths* of facts connecting e_s and e_t. In this paper, we propose a new approach based on Graph Neural Networks which enforces learning over all these paths naturally by feeding a model with the minimal subset of the knowledge graph containing all of them. By learning to produce representations for entities and facts akin to word embeddings, we can train a model to decode these representations and predict new facts in a multitask approach. We demonstrate that such a model improves the state-of-the-art mean average precision in the Freebase+ClueWeb link prediction benchmark to 92%.

1

2 MACHINE LEARNING BASICS

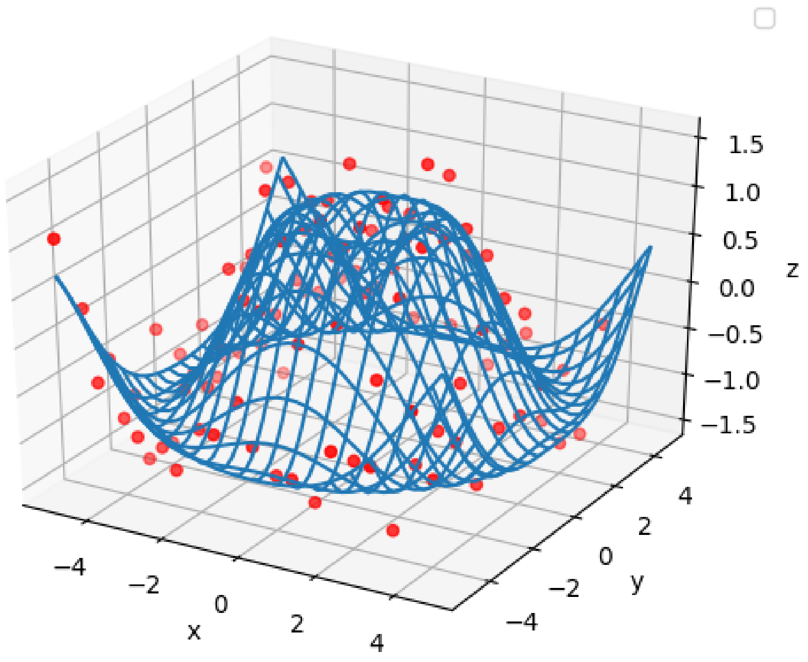
Generally speaking, Machine Learning (ML) refers to the study and engineering of statistical models which are capable of generalizing a function given a limited observation window of it. For example, a ML model can be given a set of example input-output pairs $(X, f^*(X))$ sampled from a function f^*, and its goal is to produce an approximation $f^{\approx}$ which minimizes some loss w.r.t. f^* in the example dataset, such as $f^{\approx} = \min_f \left(\sum_X (f^*(X) - f(X))^2 \right)$ for sum-of-squares loss. This definition encompasses a surprisingly wide range of techniques, ranging from linear regression (Figure 2.1) to (for example) the graph neural networks which are the topic of this thesis.

Figure 2.1: Linear regression example. A nonlinear function $f^*(x)$ (in red) is approximated by a linear model $f^{\approx}(x)$ (in black). Source: Author.



In most cases, ML can be interpreted as *generalized curve-fitting.* Figure 2.2 exemplifies bidimensional curve-fitting (i.e. $f^* : \mathbb{R}^2 \to \mathbb{R}$). Even when it is dealing with higher dimensionalities, we can think of ML applications as producing a hypersurface whose average distance to the example datapoints is minimized. Each deep neural network architecture, which most often than not receives as input vectors of high dimensionality, can be thought of as a family of hypersurfaces. Because neural networks are parameterized by their (many) neural weights and biases, each parameter configuration yields a different hypersurface. When training is successful, the hypersurface obtained is satisfactorily close to all training examples.

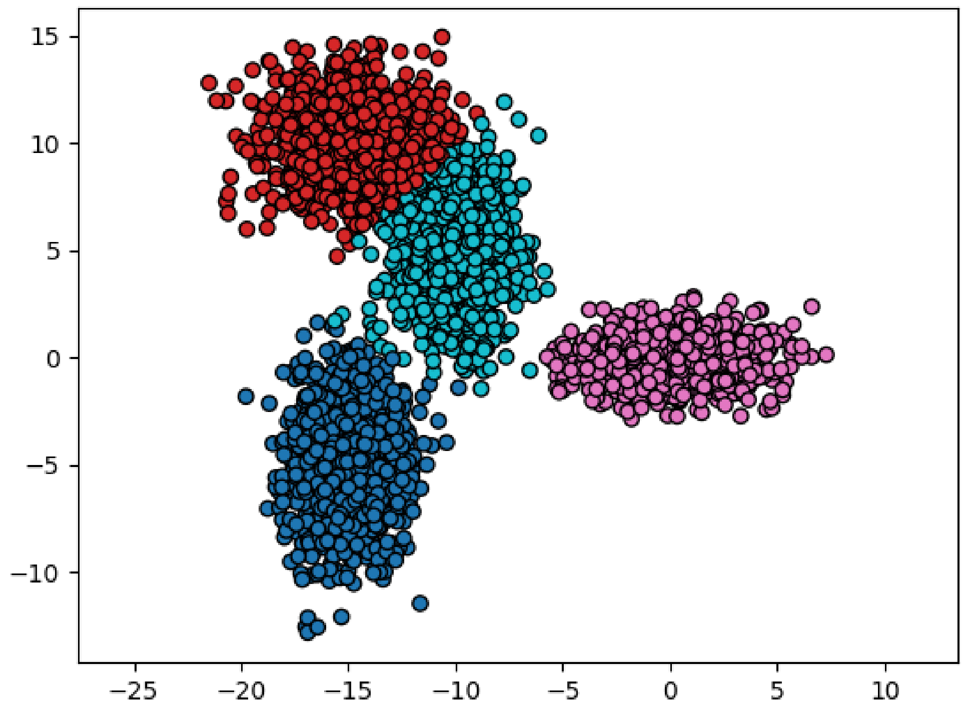
Figure 2.2: Bidimensional curve fitting. Source: Author.



The category of ML discussed above is known as **supervised learning**, as we have access to all input-output pairs $(x, f^*(x))$. We do not always have input-output pairs (also called *labelled data*) readily available for training, but in some contexts ML can learn even without explicitly knowing the desired output for each input example. When performing k-means clustering (Figure 2.3), for instance, we are not allowed to train our model with labelled observations and have it generalize to other domains accordingly. We are instead presented with a single set of observations and asked to cluster them into k distinct categories. This can nevertheless be accomplished if we proceed to arrange observations in such a way as to minimize the variance of each cluster. In this context we are still minimizing some error term, but it is not directly related to the difference between $f^{\approx}$ and f^* (which we do not know). This category of ML is known as **unsupervised learning**. At the interface between supervised and unsupervised learning we have **semi-supervised**

learning when some but not all input examples have outputs associated to them. Related to semi-supervised learning is **active learning**, when the model is only able to obtain a limited number of outputs (based on some budget) and has to actively choose from which inputs it will obtain them.

Figure 2.3: When performing K-clustering, we do not minimize the difference between the approximation $f^{\approx}$ and the ground-truth f^*, because f^* is unknown. We minimize another error function instead (specifically the sum of cluster variances). This is an example of **unsupervised learning**. Source: Author.



Finally, a **reinforcement learning** agent is not fed with the correct outputs for each input, but rather with rewards / penalties associated with each decision it makes while interacting with the environment. An example of reinforcement learning is Q-learning, in which an agent tries to learn the reward / penalty associated with each combination of state and decision. For example in (MNIH et al., 2013) the agent is required to learn the reward / penalty associated with each Atari button press for each configuration of pixels in the screen. The space of possible 260×160 RGB pixel configurations is of course a huge exponential number ($= (260 \times 160)^{3 \times 255} \approx 10^{3533}$), but this difficulty can be bypassed by replacing the usual Q-learning table of size $\#states \times \#actions$ by a convolutional neural network fed with visual inputs.

The main advantage of ML is that it allows one to synthesize algorithms purely from training data, without explicitly programming them. These algorithms are rather simple when few variables are involved, corresponding to a linear equation for example in Figure 2.1, but can become rather involved when more parameters are added. The highly successful model ResNet for image classification, for instance, spans 152 convolutional layers (HE et al., 2016). Even though the status of the functions learned by simple ML systems as “algorithms” may be disputed, the state-of-the-art in deep learning is rife with applications which blur the border between what could be considered an algorithm or not. As we will see in the next sections, ML systems in general and graph neural networks in particular can nowadays be trained to perform complex computations on symbolic domains, extend their computation over a controllable number of iterations and even produce relational output – attributes which are at the very least reminiscent of hand-engineered algorithms.

3 DEEP LEARNING BASICS

In its modern and usual sense, Deep Learning (DL) corresponds to the study and engineering of ML models based on deep (i.e. many-layered) artificial neural networks. An ANN is a model of which the main building block – the artificial neuron – was envisioned to mimick the “all-or-none” rule of neuroscience, which states that a neuron (Figure 3.1) either fires or does not fire depending on its received signal. In other words: the intensity of the received signal does not interfere with the intensity of the signal which is propagated forward (if any), it merely determines, in a boolean fashion, whether a signal will be sent.

For reasons which will become clear briefly, however, it is paramount that the artificial neurons of ANNs do not actually fire in a strictly boolean fashion. For the moment, it suffices to state that the “all-or-none” rule must be approximated by continuous (differentiable) means. Fortunately, this can be done very easily as we have many differentiable functions at our disposal whose outputs saturate to a fixed value approximately above a certain threshold. This is the case of the hyperbolic tangent function and the sigmoid function (Figures 3.2 and 3.3 respectively).

Figure 3.1: Drawing of neurons in the pigeon cerebellum by Spanish neuroscientist Santiago Ramón y Cajal. Source: Wikimedia Commons.

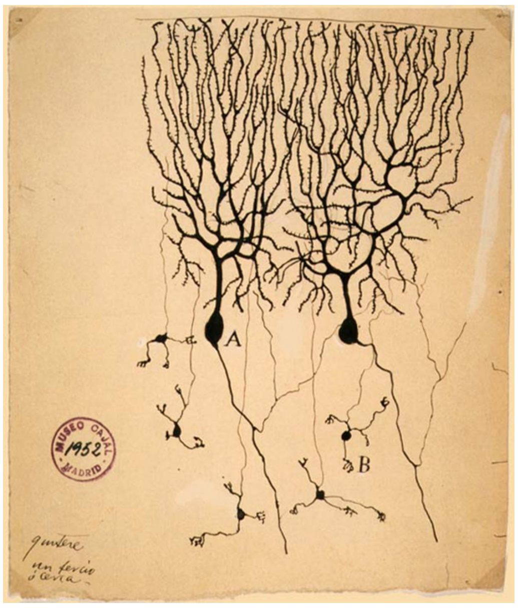


Figure 3.2: Graph of the hyperbolic tangent function ($\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$), which is often used as an activation function for ANNs. Source: Author.

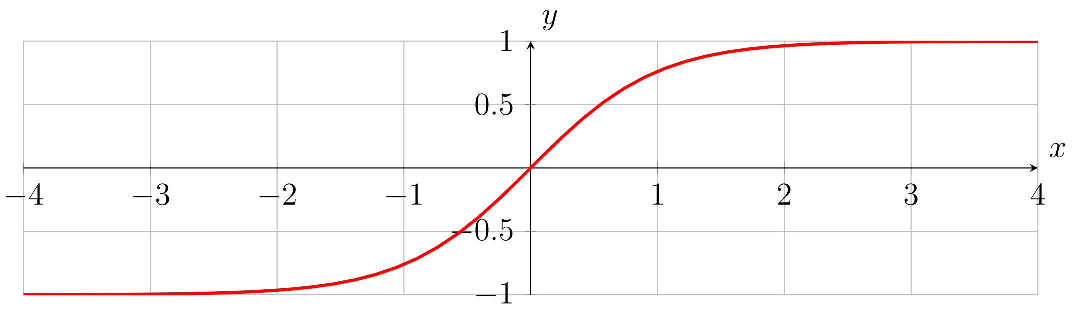
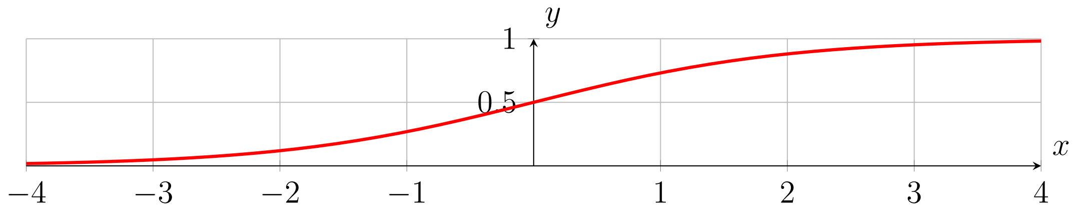
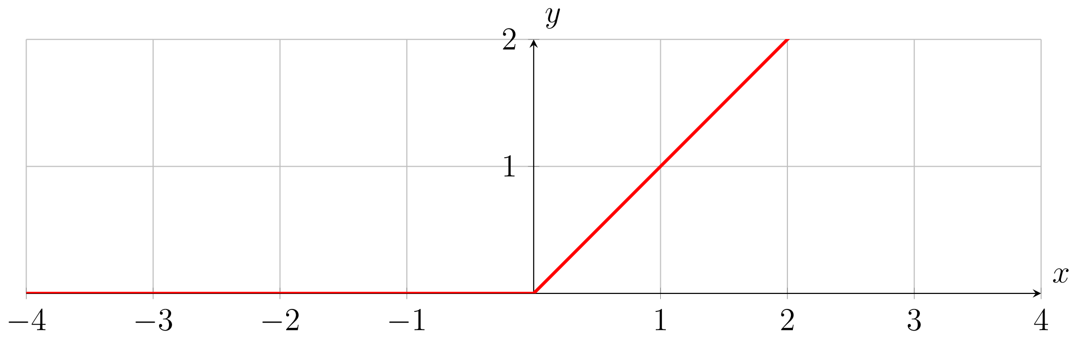


Figure 3.3: Graph of the sigmoid function $S(x) = \frac{e^x}{e^x+1}$, which is often used as an activation function for ANNs.



Although $-1 < \tanh(x) < 1$ and particularly $0 < S(x) < 1$ capture well the desired "all-or-none" behavior while the rectifier function $0 \leq f(x)$ (Figure 3.4) is unbounded to the right and not everywhere differentiable, there are advantages for using it as an ANN activation, the most important one being that computing its derivative is less costly.

Figure 3.4: Graph of the rectifier function $f(x) = x^+ = \max(0, x)$, which is often used as an activation function for ANNs. Source: Author.

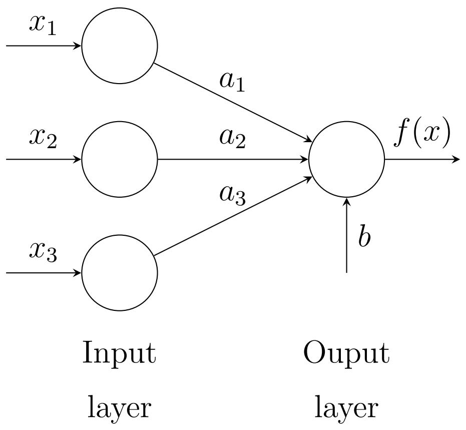


ANNs, in their crudest form, are built by composing these nonlinear activation functions with weighted sums. Something like

$$f(\vec{x}) = \tanh(a_1x_1 + a_2x_2 + a_3x_3 + b) \tag{3.1}$$

can be interpreted as a single-layer ANN where a_1, a_2, a_3 represent the "neural weights" (i.e. how strongly the output neuron is connected to the input signals x_1, x_2, x_3) in our neuroscience metaphor, while b represents the output neuron's *bias*. The greater the bias (which corresponds to a left shift in the graph in Figure 3.2), the closer the activation is to 1, regardless of the received signal. Equation 3.1 can also be visualized in a pictorial representation (Figure 3.5) where each neuron is represented as a circle. The directionality of the computation flow is left $\rightarrow$ right: feeding the initial layer are the inputs x_1, x_2, x_3, while the output $f(x)$ points outwards. Forward neural connections are annotated with their corresponding weights (a_1, a_2, a_3), and the bias b is also fed to the output neuron.

Figure 3.5: Pictorial representation of the single-layered ANN in Equation 3.1. Source: Author.

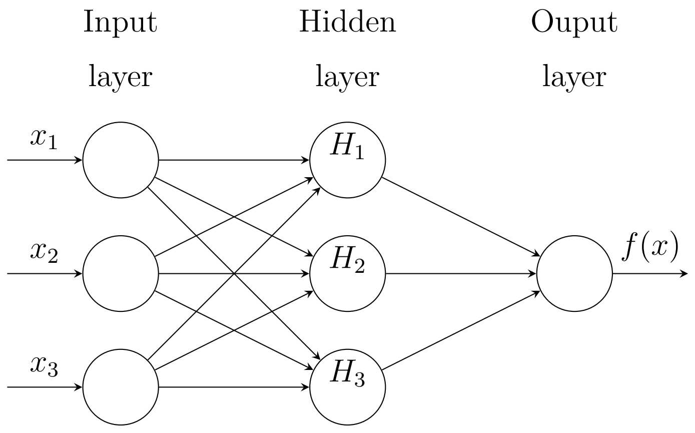


Possibly the major insight from the deep learning renaissance of the last decade is that **depth** is often more powerful than **width** when designing ANNs. The depth corresponds to the number of layers in the network (in Figure 3.1 we have one input layer and one output layer), while the width corresponds to the size of each layer in the number of neurons (in Figure 3.1 three neurons for the input layer and one for the output layer). This observation was initially exemplified in the evolution of convolutional neural network architectures for the ImageNet dataset, a collection of over 50 million cleanly labelled full resolution images used as a benchmark for image classification (DENG et al., 2009). The ~4M parameter architecture ResNet (HE et al., 2016) was able to surpass VGGNet, with ~140M parameters (SIMONYAN; ZISSERMAN, 2014). ResNet sported 152 layers, however, while VGGNet encompassed only 16. A deeper network is simple to visualize as just an iterated composition of an activation function (i.e. tanh) and a weighted sum. For example, adding an additional layer of three neurons to Equation 3.1 corresponds to:

$$f(x) = \tanh\left(b^O + \sum_{i=1}^{3} a_i^O \cdot \tanh\left(b_i^H + \sum_{j=1}^{3} a_{i,j}^H \cdot x_j\right)\right) \tag{3.2}$$

which can also be visualized in pictorial format as the diagram in Figure 3.6. $a_{i,j}^H$ corresponds to the neural weights between the input layer and the hidden layer, while a_h^O corresponds to the neural weights between the hidden layer and the output layer. Analogously, b_i^H and b^O correspond to the biases of the hidden neurons and to the bias of the output neuron respectively. We now have $3 \times 3 + 3 = 12$ neural weights and $3 + 1 = 4$ biases, because for every fully-connected layer of width n_2 connected to the output of a previous layer of width n_1 we add $n_1 \times n_2$ neural weights and n_2 biases to the collection of parameters of our model.

Figure 3.6: Pictorial representation of the ANN in Equation 3.2 with one hidden layer. Hidden neurons are marked with H_1, H_2, H_3, and neural weights and biases are suppressed due to space constraints. Source: Author.



Finally, the collection of weighted sums operated at each layer can be succinctly described by a matrix multiplication. Consider for example the 3 inputs received by neurons H_1, H_2, H_3 in the hidden layer of the ANN in Figure 3.6. In tensorial form, they can be described as

$$\begin{pmatrix} a^H_{1,1}x_1 + a^H_{1,2}x_2 + a^H_{1,3}x_3 + b^H_1 \\ a^H_{2,1}x_1 + a^H_{2,2}x_2 + a^H_{2,3}x_3 + b^H_2 \\ a^H_{3,1}x_1 + a^H_{3,2}x_2 + a^H_{3,3}x_3 + b^H_3 \end{pmatrix} = \begin{pmatrix} a^H_{1,1} & a^H_{1,2} & a^H_{1,3} \\ a^H_{2,1} & a^H_{2,2} & a^H_{2,3} \\ a^H_{3,1} & a^H_{3,2} & a^H_{3,3} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b^H_1 \\ b^H_2 \\ b^H_3 \end{pmatrix} \tag{3.3}$$

A ANN model with N layers with sizes $n_1, n_2, \ldots, n_N$ spans $N - 1$ such matrices $\mathbf{M}_1 \ldots \mathbf{M}_{N-1}$, with sizes $n_1 \times n_2, \ldots, n_i \times n_{i+1}, \ldots n_{N-1} \times n_N$ respectively. The model also spans $N - 1$ bias vectors $\vec{b}_2 \ldots \vec{b}_N$ with sizes $n_2 \ldots n_N$ respectively. One way to think about ANN layers is to focus on the function that each layer applies to the tensor of inputs received from the previous layer, which corresponds to an activation applied to a matrix multiplication plus a bias. Concretely, the function $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$ computed by the i-th layer can be expressed as:

$$f_i(\vec{x}) = \varphi.(\mathbf{M}_i \times \vec{x} + \vec{b_i}) \tag{3.4}$$

Where φ. is the activation function (for example $\tanh(x)$) applied element-wise. As a result, the entire ANN can be interpreted in terms of iterated function composition as

$$f(\vec{x}) = f_N \circ f_{(N-1)} \circ \cdots \circ f_3 \circ f_2 = \left(\bigcirc_{i=N}^{2} f_i \right) (\vec{x}) \qquad (3.5)$$

Where the notation $\bigcirc_{i=n_1}^{n_2} f_i = f_{n_1} \circ \cdots \circ f_{n_2}$ denotes iterated function composition. Notice that this formalization also encompasses networks where layers are not necessarily fully-connected: to disconnect two neurons it suffices to make the corresponding element in the weight matrix equal zero.

3.1 Multilayer Perceptron

A Multilayer Perceptron (MLP) is a class of artificial neural network in which the connections are feedforward (i.e. do not form a cycle) and are organized into at least three layers: an input layer, a hidden layer and an output layer. The ANN in Figure 3.6, for instance, is an example of a MLP.

3.2 Expressiveness of Artificial Neural Networks

The formalization presented above exemplifies the simplicity of ANNs, which can be entirely described in terms of matrix multiplications and a simple activation function such as tanh or the rectifier function (ReLU). This may lead to the conclusion that ANNs are somewhat limited in their capability of approximating complex functions, but since the early 1990s it is known that feed-forward neural networks can approximate with arbitrary precision any continuous function defined over compact subsets of $\mathbb{R}^n$ (i.e. hyperrectangles). This is due to the Universal approximation theorem (CYBENKO, 1989; HORNIK, 1991), whose formal description is the following:

Theorem 1 (Universal approximation theorem (HORNIK, 1991))**.** *Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, continuous bounded function. For every $\varepsilon \in \mathbb{R}$ and function $f : [0,1]^m \to \mathbb{R}$ in the unit hypercube there exists a layer size $N \in \mathbb{N}$, hidden neural weights $\mathbf{M}_1 \in \mathbb{R}^{m \times N}$, hidden biases $\vec{b} \in \mathbb{R}^m$ and output neural weights $\mathbf{M}_2 \in \mathbb{R}^N$*

such that

$$f^{\approx}(\vec{x}) = \mathbf{M}_2 \times \varphi.\left(\mathbf{M}_1 \times \vec{x} + \vec{b}\right) \tag{3.6}$$

satisfies $|f^{\approx}(\vec{x}) - f(\vec{x})| < \varepsilon$ *for all* $\vec{x} \in [0,1]^m$.

3.3 Feasibility of Training Artificial Neural Networks

The Universal approximation theorem asserts the ultimate **capability** of artificial neural networks, showing that there exists an ANN and a parametrization thereof which together yield an approximation for any continuous function. It does not, crucially, say anything about the **feasibility** of training such a ANN to satisfactory levels of accuracy. Regardless of the unquestioned successes of deep learning in the last decades, theoretical arguments in this direction have been scarce. Nevertheless, at least one recent effort has made important advances on the theoretical understanding of gradient descent for ANNs. (ALLEN-ZHU; LI; SONG, 2018) were able to show that deep neural networks can achieve perfect (100%) classification accuracy on the training dataset in polynomial time w.r.t. the number of training samples and the number of layers in the model.

3.4 Vectors, Matrices and Tensors

Although it is not strictly necessary to formalize deep learning concepts in the language of tensors, it is extremely useful to do so. A tensor, put succinctly, generalizes the concept of a matrix in analogy to how a matrix generalizes the concept of a vector and to how a vector generalizes the concept of scalar quantities. A vector $\vec{x}$ can be thought of as an ordered list of real numbers $x_i \in \mathbb{R}$ (although in principle one could define vectors over any set other than $\mathbb{R}$), as exemplified below for a four dimensional real vector.

$$\vec{x} = \begin{pmatrix} 3 \\ -2\pi \\ 0 \\ -1 \end{pmatrix} \tag{3.7}$$

The corresponding *vector space* (i.e. the set to which all such four dimensional vector pertain) can be described by the 4-ary cartesian product $\mathbb{R}^4 = \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$. In this sense, a scalar quantity can be thought of as a unitary vector described by the unary cartesian product $\mathbb{R}^1 = \mathbb{R}$, and we can immediately see how vectors extend the concept of scalars to ordered lists thereof.

Matrices are similar to vectors in the sense that a matrix is an ordered list of vectors analogously to how vectors themselves are ordered lists of scalars:

$$\mathbf{A} = \begin{pmatrix} 3 & 8 & -2 & 0 \\ 3\pi & -2\pi & 5.2 & 9 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.8}$$

Accordingly, in the context of 3×4 real matrices such as $\mathbf{A}$, the corresponding space can be described by a 3-ary cartesian product performed over 4-dimensional vectors: $\mathbb{R}^{3\times 4} = \mathbb{R}^4 \times \mathbb{R}^4 \times \mathbb{R}^4$. This immediately allows us to generalize the concept to ordered lists of (ordered lists of (ordered lists of (...))) indefinitely: one must just define spaces $\mathbb{R}^{n_1 \times n_2 \times \ldots n_N}$. This generalization to vectors of any *rank* (0 for scalars, 1 for vectors, 2 for matrices, 3 for “three-dimensional” matrices, etc.) is called a *tensor*.

$$\mathbb{R}^{\overbrace{n_1 \times n_2 \times \ldots n_N}^{N \text{ is the tensor } \textit{rank}}} \tag{3.9}$$

Tensors are useful in deep learning because training data is usually logically organized into multiple dimensions. In the context of computer vision, for instance, images correspond to rectangular grids of pixels the content of each one usually corresponds to a color defined over three (RGB) channels. In this context, it is customary to represent each channel of an image as a $n \times m$ matrix and the image itself is corresponds to stacking the three channels, yielding a $3 \times n \times m$ tensor. When input images pass through the first convolutional layer (which is usually composed of not one but many convolutional kernels), each of the K learned kernels will yield a different filtered image. In this context, it makes sense to treat all K output images as if they were mere channels on an hypothetical image format with multidimensional pixel colors. In the language of tensors, this corresponds to replacing the $3 \times n \times m$ input to the first layer for a $K \times n \times m$ to the second. Additionally, because deep learning models are customarily trained in *batches* (i.e. the gradients to the loss computed over many different instances are added up and we perform descent on the resulting gradient), the inputs to the first layer usually have one additional "batch" dimension: $b \times 3 \times n \times m$.

3.5 Parameter Space and Gradient Descent

An ANN model is parameterized by its neural weights and biases, meaning that each different assignment of values to these parameters potentially implements a different function. The configuration of all P parameter values in a given ANN model $\mathcal{M}$ can be interpreted in tensorial format as a multidimensional vector $p \in \mathbb{R}^P$, giving rise to a *parameter space* $\mathcal{P} = \mathbb{R}^P$. Given a parameter configuration $p \in \mathcal{P}$ for $\mathcal{M}$, if one has access to a input/output training example (x, y), they can in principle evaluate a *loss* L between the ground-truth output y and the output of the function computed by $\mathcal{M}$ with configuration p given input x for example as:

$$L(p) = (\mathcal{M}_p(x) - y)^2 \tag{3.10}$$

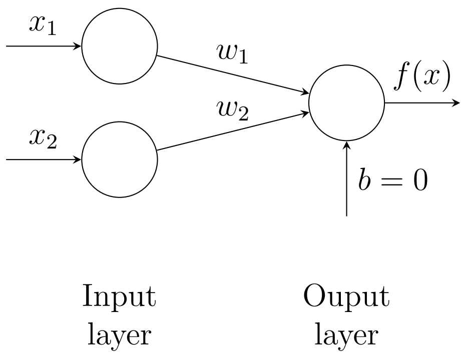
For fixed x and y, the loss $L(p)$ is ultimately a function of the parameter configuration p. Because $\mathcal{M}$ is differentiable w.r.t. each individual parameter, we can compute a partial derivative $\frac{\partial}{\partial p_i}L$ for each parameter $p_1, \dots, p_P$. Ultimately, these partial derivatives can be stacked in a vector to obtain a **gradient** for the loss:

$$\triangledown L = \begin{pmatrix} \frac{\partial}{\partial p_1}L \\ \frac{\partial}{\partial p_2}L \\ \vdots \\ \frac{\partial}{\partial p_P}L \end{pmatrix} \tag{3.11}$$

Because each orthogonal direction in the gradient is weighted by the steepness of the loss function in that direction, the gradient effectively points towards the direction of steepest ascent in the hyperdimensional landscape defined by the loss function. Therefore if one were to take a step in the parameter space in the direction opposite to that pointed by the gradient, they should expect to see the loss diminish. This is the insight behind using gradient descent to train ANN models: repeatedly take small steps in the direction of steepest **descent** (i.e. opposite to the gradient) until the loss is satisfactorily small. This process is visualized in Figure 3.8, where the loss of the small single-layer ANN depicted in Equation 3.12 and Figure 3.7 is decreased throughout 200 gradient descent operations.

$$f(\vec{x}) = w_1 x_1 + w_2 x_2 \tag{3.12}$$

Figure 3.7: Pictorial representation of the single-layered ANN in Equation 3.1. Source: Author.

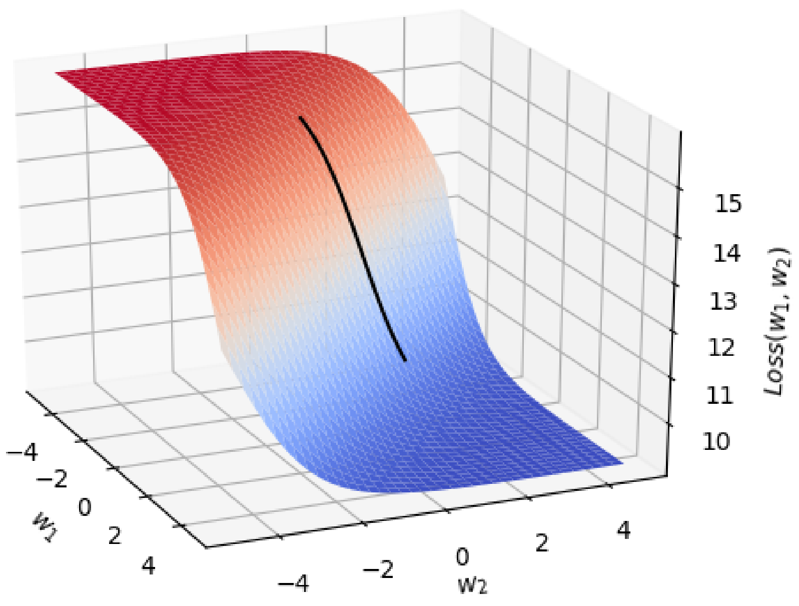


Notice that the smaller the length of a gradient descent step, the closest we are to a displacement on the plane tangent to the point in consideration. Yet we cannot make steps infinitesimally small, so as a result they must be chosen arbitrarily and the gradient descent method is also parameterized by a **learning rate**: a scalar determining the step length. For smooth losses, we should expect large learning rates to be effective, because we expect the gradient to vary little over considerable distances and consequently to be approximated moderately well by a straight line. On the other hand, one should be careful not to employ large learning rates when training on complex loss landscapes.

3.6 Batch Training and Stochastic Gradient Descent

In the last subsection we have illustrated the gradient descent method on a toy problem with a single training instance. On practice, machine learning applications employ thousands to millions of training examples. The gradients of the corresponding models can still be computed in a straightforward way: we must only reduce all individual losses into a single, scalar value. This can be done, for example, by adding them up or by computing their arithmetic mean. In some specific cases it may even be useful to reduce them multiplicatively or in other unusual manner. Nevertheless, when a large number of instances is considered, we must take into

Figure 3.8: The result of performing gradient descent on the loss $L(w_1, w_2) = \left(\mathcal{M}_{w_1,w_2}(\begin{pmatrix} 1 & 1 \end{pmatrix}^\intercal) - 4\right)^2$ of the single-layer ANN $\mathcal{M}_{w_1,w_2}(x) = S(w_1x_1 + w_2x_2)$ w.r.t. the inputs $x_1 = 1, x_2 = 1$ and the output $y = 4$. The model evolves throughout 200 gradient descent operations with initial parameters $w_1 = -2, w_2 = -0.25$ and learning rate $= 0.01$. Source: Author.



account the computational cost of computing such a gradient, which can be very expensive. A solution is to repeatedly sample a random instance, compute its gradient and perform a gradient descent step on it. In this setup we are required to run an expected number of gradient descent operations equal to the size of the training set to sweep over all instances, which might also be costly. An hybrid strategy is to sample not one but a **minibatch** of $n > 1$ random instances and perform a gradient step on its gradient. This is the most commonly used technique to train ANN models, with minibatch sizes varying from implementation to implementation.

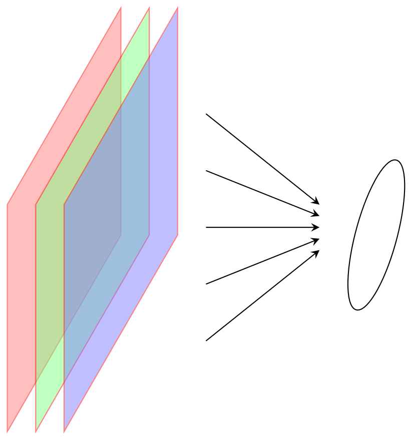
3.7 Artificial Neural Network Building Blocks

In subsection 3.5 we have described ANNs in their crudest form as iterated composition of matrix multiplication operations with activation functions, pointing out how this formalization can capture even networks where layers are not necessarily fully-connected. However, ANN models can – and often do – transcend the weighted sum model. Because the sole restriction to the applicability of the gradient descent method is that the loss function is differentiable, we are only fundamentally limited by the differentiability of candidate ANN building blocks. As a result, we are for example free to feed a layer with the weighted **product** $p = \prod_{i=1}^{n} w_i x_i$ of all its inputs, which can be easily differentiated w.r.t. w_i as $\frac{\partial p}{\partial w_i} = \frac{1}{w_i} \prod_{i=1}^{n} w_i x_i$.

3.8 Convolutional Neural Networks

Neuroscience research in the late 1960s showed that the visual cortices of some mammals, in particular cats and monkeys, contain neurons which are sensitive to small, localized regions of the visual field. The outputs of these neurons, dubbed "simple cells" (Figure 3.9), are maximized upon observation of straight lines with particular orientations. This neural wiring is different from the fully-connected architecture discussed in the previous subsections in two fundamental ways: first because in this case layers are sparse and second and perhaps most importantly because the connections between the retina and a layer of simple cells archives (even if passively) visual information. That is not to say that the **signals** sent through these connections contain visual information (although they certainly do), but that the **neural wiring itself** is enriched with it. By connecting a simple cell with a set of neurons closely localized in the visual field, we are passively feeding the model with the information that those neurons are near one another. This is the main insight behind Convolutional Neural Networks (CNNs), which intelligently engineer layers to imprint visual information into an ANN model.

Figure 3.9: Pictorial representation of a simple cell receiving signals from a square RGB window of the visual field. Source: Author.



There are significant advantages to tying together the weights of all simple cells – that is, making each simple cell feed on a different square window of the visual field (Figure 3.10) but share the same neural weights as all other simple cells. If all simple cells are conditioned on learning the same relationship, the function $f : \mathcal{RGB}^2 \rightarrow \mathbb{R}^2$ learned by the model as a whole will exhibit an important property called **equivariance** to translation. This essentially means that the sole effect of performing an horizontal or vertical shift in the input image will be an equivalent shift in the output of f. Figure 3.11 exemplifies this property on a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$

Figure 3.10: Pictorial representation of a 3×3 matrix of simple cells each receiving signals from a different square RGB window of the visual field. Source: Author.

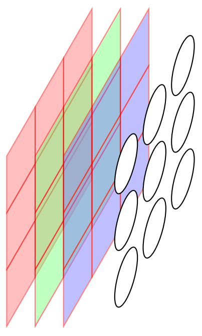
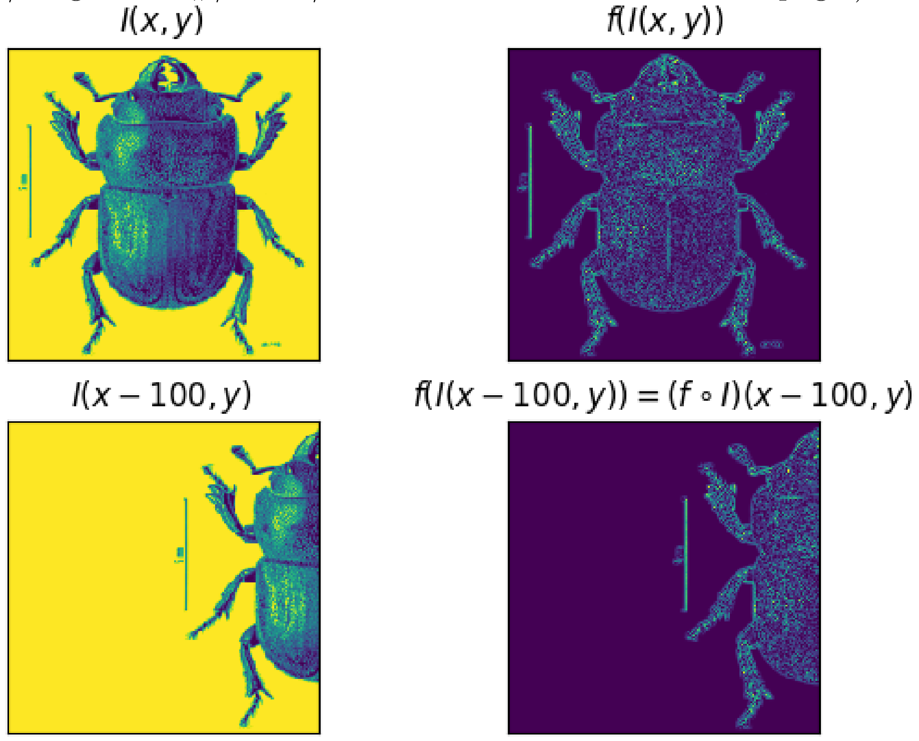


Figure 3.11: Example of an equivariant function $f : \mathbb{R}^2 \to \mathbb{R}^2$: As the input image I undergoes a left shift, the composition $f \circ I$ is shifted by the same amount. Source: Author. Beetle image obtained from Wikimedia Commons (<https://en.wikipedia.org/wiki/Stag_beetle#/media/File:COLE_Lucanidae_Lissotes.png>).



An insight about the kind of functions such a model can possibly learn is the following: notice that each simple cell weighs each of its inputs (i.e. each pixel) by a certain amount which is ultimately determined by the neural weight connecting it to the corresponding neuron in the visual field. For a simple cell connected to a grid of 3×3 pixels, a possible arrangement of neural weights is the following:

$$W = \begin{pmatrix} 0.08 & 0.13 & -0.45 \\ 0.22 & -0.35 & -0.07 \\ 0.17 & 0.29 & -0.29 \end{pmatrix} \tag{3.13}$$

The input signal of such a simple cell would then be given by

$$I = \sum_{i=1}^{3} \sum_{j=1}^{3} W_{ij} x_{ij} \tag{3.14}$$

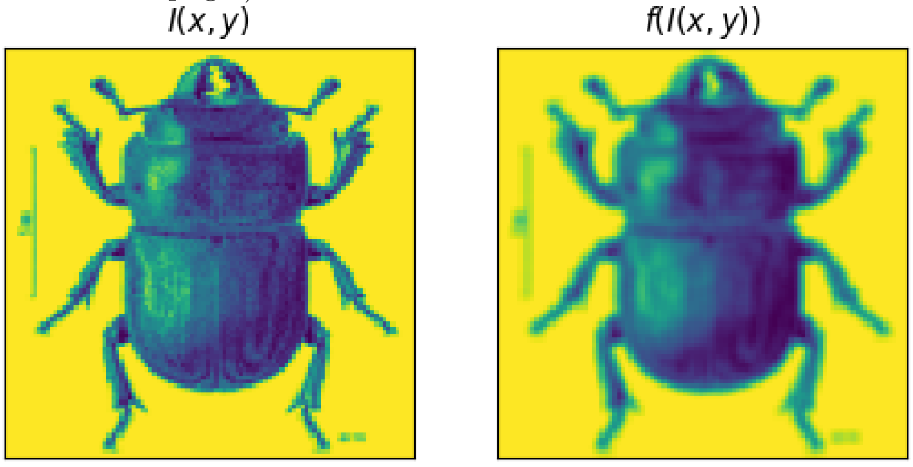
This is exactly equivalent to the 2D *convolution* operation (with stride $= 3$) which is commonly used in digital image processing. In simple terms, what such a model does is that it sweeps over the input image producing an output image for which the value of each pixel (i, j) is a linear combination of the values of its (9-

connected) neighbors (notice that, in this context, each pixel is considered a neighbor of itself). In image processing parlance, matrices such as W are called *convolution kernels*. Ultimately, the convolution kernel determines which kind of filtering will be applied over the input image, but obviously the operation implemented by most matrices $\in \mathbb{R}^{3\times 3}$ have no useful interpretation. Some matrices, nevertheless, have useful effects. The convolution kernel

$$W = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{3.15}$$

for example, can be used to **blur** an image, as Figure 3.12 exemplifies. It is easy to see how such a result emerges from this kernel, as what it effectively implements is a replacement of the original pixel values by the averages of their 3×3 neighborhoods.

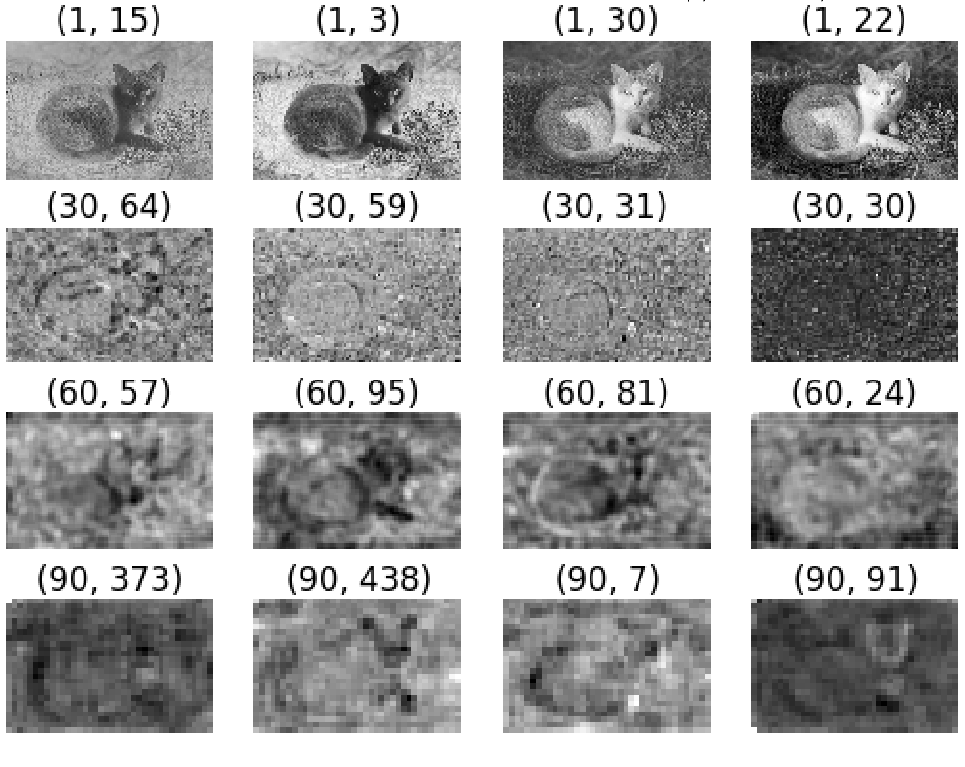
Figure 3.12: Result of performing a convolution with the “blur kernel” in Equation 3.15. Source: Author. Beetle image obtained from Wikimedia Commons (<https://en.wikipedia.org/wiki/Stag_beetle#/media/File:COLE_Lucanidae_Lissotes.png>).



The most important insight behind CNN architectures is that such useful kernels can ultimately be learned through gradient descent on the parameter space of kernel weights. The second to most important insight is that not only 1) many kernels can be learned at the same time but also 2) kernels can be learned at multiple consecutive levels. In simple terms, a CNN learns n_1 convolutional kernels to be applied to the input image, which yield n_1 filtered images, then it learns n_2 kernels to be applied to these n_1 images, and so on for many convolutional steps. The interpretation of this chain of convolution operations is that a successfully trained

CNN model should be able to learn simple associations in the first layer, such as detecting lines in particular orientations, and it should be able to learn filters to effectively combine features learned at a given layer into more complex associations in the next layer. Eventually, upon stacking many layers, a CNN should be able (for example) to have a neuron activate in lesser or greater intensity depending on how much the input image resembles a photograph of a stag beetle, or, as Figure 3.13 exemplifies, high-level features of a cat such as ears, face and tail.

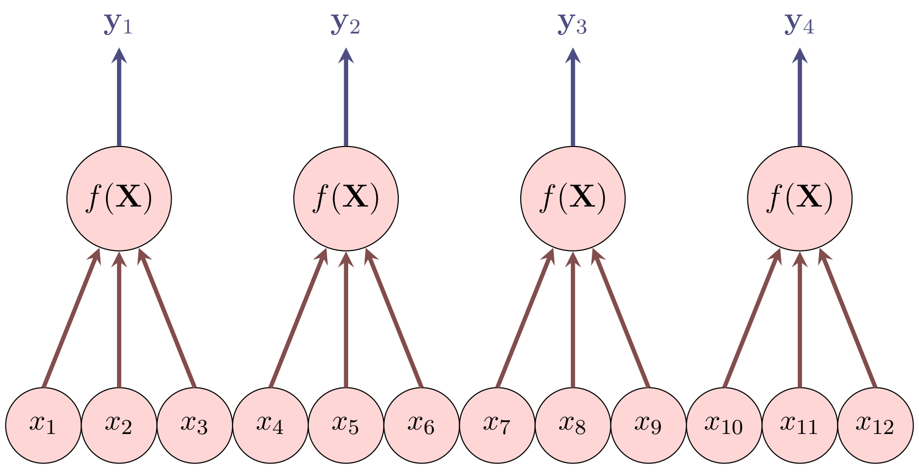
Figure 3.13: Visualizing some of the convolutional kernels learned by the InceptionV3 model (XIA; XU; NAN, 2017) on the ImageNet dataset at four different layers (*conv2d_1*, *conv2d_30*, *conv2d_60*, *conv2d_90*). Kernels are visualized by applying them on a sample image containing a domestic cat. As is usual with convolutional models, images' resolutions become lower as more convolutional layers are stacked, abstracting unimportant pixel information. At the last stages, learned convolutions weigh in high-level features such as the cat's ears, face and tail. This is in contrast with the first kernels, which are sensitive to low-level features such as lines in particular orientations. Cat image obtained from Wikipedia user Basile Morin and licensed under CC BY-SA 4.0 (<https://en.wikipedia.org/wiki/Cat#/media/File:Felis_silvestris_catus_lying_on_rice_straw.jpg>). InceptionV3 ImageNet weights obtained from Keras Applications library <https://keras.io/applications/>.



3.9 Parameter Sharing in Convolutional Neural Networks

Although this thesis does not make direct use of CNNs, there are fundamental insights shared by CNNs and the main topic of this work, the most important one being the concept of *parameter sharing* in ANN models. As discussed above, CNNs benefit from the insight of employing tied weights over sparse, localized neural connections to effectively learn convolutional kernels (Figure 3.14). This makes sense due to the fact that the equivariant property is useful for analyzing images, as our visual ontology is not usually interested on pixel coordinates on a **global** level but rather on a **local** one. In other words: if you are going to learn an association for a small grid of pixels, there are not, in general, good reasons to learn different associations depending on your position on the big picture. Edge detection is a good example: to perform edge detection it suffices to know about the immediate neighborhood of each pixel, and as a result the required operation can be carried out regardless of whether you are performing it (for example) above or below the center of the visual field. Rigorously speaking, our visual cortex is also interested on pixel correlations over long distances (such as combining pixel information from the paws and ears of a dog to identify it), but CNNs analyze these correlations indirectly by stacking multiple layers of abstraction between the pixel level and the conceptual level, progressively reducing the dimensionality of the analyzed domain until paw and ear information are brought spatially closer to each other.

Figure 3.14: Pictorial representation of a one-dimensional convolutional neural network with kernel size = 3 and strides = 3. Because the parameters of all four convolutional blocks are shared, the resulting network can be thought of as sweeping the same convolutional filter throughout the entire 15-dimensional input. Source: Author.



Although parameter sharing can be understood conceptually, its effect can also be exemplified numerically if we take into consideration the sizes of the parameter space in the context of a fully-connected network in comparison to a convolutional architecture. For an image with n pixels and a hidden layer of m neurons, a fully-connected scheme yields $n \times m$ weights and biases, while a convolutional scheme with kernels $\in \mathbb{R}^{k \times k}$ yields $m \times k^2$. If $k^2 \ll n$ (as it is almost surely the case), we obtain a dramatic reduction on the size of the parameter space, which allows for less iterations of gradient descent and ultimately faster training. In this context, the triumph of CNNs is that they reduce the parametrization of the model to the level of pixel neighborhoods. We will see briefly how this relates to the type of parameter space reduction implemented by Graph Neural Networks, which also employ parameter sharing on a different domain.

3.10 Recurrent Neural Networks

When dealing with time series data, it is not much useful to devote the same level of attention to datapoints in different timesteps. Conceptually, older informations should be weighed in more as they potentially have more leverage on the future of the system. Something which has happened at $t = 0$ potentially influences all events up to the last timestep, while the influence of something which happened at $t = n$ does not capture the first n events. This suggests that an effective implementation of ANN models to perform predictions on time series should take this factor into account. A solution is to augment the model with an internal "memory state" (called a "hidden state" in DL parlance) to allow it to store information about what it has already seen. If each temporal observation at time t is a vector $\mathbf{X}^{(t)} \in \mathbb{R}^{d_i}$, each output at time t is a vector $\mathbf{Y}^{(t)} \in \mathbb{R}^{d_o}$ and each hidden state at time t is a vector $\mathbf{H} \in \mathbb{R}^{d_h}$ (where d_i is the input dimensionality and d_o is the output dimensionality), ideally we want to train some function $f : \mathbb{R}^{d_h} \times \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_h} \times \mathbb{R}^{d_o}$ whose role is to receive an observation $\mathbf{X}^{(t)}$, update the current hidden state $\mathbf{H}^{(t-1)}$ into $\mathbf{H}^{(t)}$ and produce an output $\mathbf{Y}^{(t)}$. Concretely:

$$f(\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}) = (\mathbf{H}^{(t)}, \mathbf{Y}^{(t)}) \tag{3.16}$$

This is the fundamental insight behind Recurrent Neural Networks (RNNs), which correspond to the function f in equation 3.16. Programatically, a RNN can be thought of as a neural network block augmented with a self-loop which implements a feedback (Figure 3.15). Architecturally, a RNN can be thought of as a series of identical NN blocks connected in sequence, their weights tied together to enforce parameter sharing (Figure 3.16).

Figure 3.15: Pictorial representation of a RNN block. Source: Author.

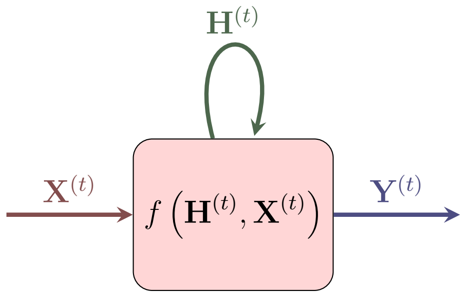
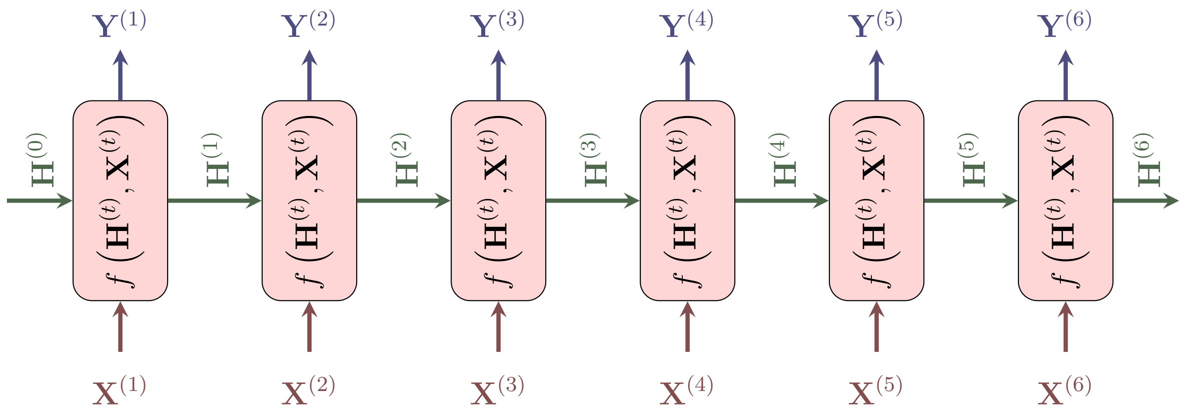


Figure 3.16: Pictorial representation of the unrolling of a recurrent unit f into six iterations. Because the parameters of all six blocks are shared, the resulting network can be thought of as iterating the same operation over the hidden state $\mathbf{H}^{(t)}$ and the input observations $\mathbf{X}^{(t)}$ that many times. Inputs $\mathbf{X}^{(t)}$ are colored red, outputs $\mathbf{Y}^{(t)}$ are colored blue and hidden states $\mathbf{H}^{(t)}$ are colored green. Source: Author.


$\mathbf{Y}^{(1)}$ $\mathbf{Y}^{(2)}$ $\mathbf{Y}^{(3)}$ $\mathbf{Y}^{(4)}$ $\mathbf{Y}^{(5)}$ $\mathbf{Y}^{(6)}$
$\mathbf{H}^{(0)}$ $\mathbf{H}^{(1)}$ $\mathbf{H}^{(2)}$ $\mathbf{H}^{(3)}$ $\mathbf{H}^{(4)}$ $\mathbf{H}^{(5)}$ $\mathbf{H}^{(6)}$
$f\left(\mathbf{H}^{(t)}, \mathbf{X}^{(t)}\right)$
$\mathbf{X}^{(1)}$ $\mathbf{X}^{(2)}$ $\mathbf{X}^{(3)}$ $\mathbf{X}^{(4)}$ $\mathbf{X}^{(5)}$ $\mathbf{X}^{(6)}$

The process of repeating a recurrent unit many times in sequence as Figure 3.16 illustrates is called "unrolling". It is analogous to the loop unrolling implemented by optimizing compilers in the context of programming languages (Listings 3.1 and 3.2). Unrolling allows for the computation of gradients in a straightforward way, as one would compute for a feed-forward NN. The twist here is that weights and biases are identical for every layer, or, in other words, that all layers are parameterized by the same reduced set of values. This is acceptable, as gradients can still be computed w.r.t. these parameters.

```c
int H = 0;
for (t = 1; t <= 6; t++)
{
    H += X[t];
}
return H;
```

Listing 3.1: For loop in the C language.

```c
int H = 0;
H += X[1];
H += X[2];
H += X[3];
H += X[4];
H += X[5];
H += X[6];
return H;
```

Listing 3.2: Equivalent unrolled implementation of the for loop in 3.1.

3.11 Parameter Sharing in Recurrent Neural Networks

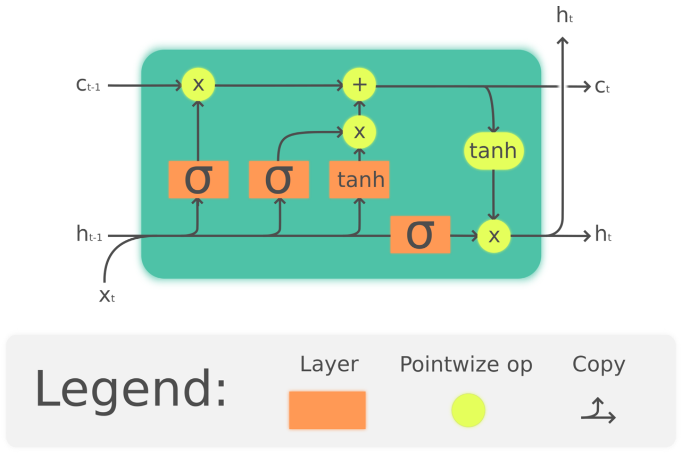
As discussed in Section 3.9, one of the triumphs of convolutional neural networks is that they employ parameter sharing over the spatial domain to enforce a key property for image analysis: equivariance to (spatial) translation. As the reader will recall, this ties into the intuition that it makes sense for image analysis techniques working from the bottom-up to be indifferent to pixel coordinates. In the same way, it makes sense for models predicting over the temporal domain not to differentiate between identical sequences of observations occurring on different moments in time. Recurrent Neural Networks, consequently, employ parameter sharing over the temporal domain (i.e. by tying weights sequentially) to enforce equivariance to translation in **time**.

3.12 Exploding / Vanishing Gradients and Long Short-Term Memory

As previously mentioned, depth is a very important feature of neural models, since stacking additional layers can yield more gains in performance than expanding the size of the existent layers correspondingly. Ironically, depth can also harm models, in what are conventionally called the vanishing and exploding gradient problems. These problems are linked to the numerical artifacts which arise from the backpropagation algorithm when we compute gradients for deep neural networks. This is due to the fact that the gradients of long series of function compositions are defined over products of multiple partial derivatives. If sufficiently many of these partial derivatives are slightly larger / smaller than 1, such a product may result in very large / small gradients, corresponding respectively to the "exploding" and "vanishing" gradient problems. This is especially problematic for recurrent neural networks, which are trained via unrolling. This essentially means that to compute gradients we will be forced to multiply the partial derivatives of the same function (the RNN) w.r.t. the learnable parameters N times.

One (very successful) proposal to overcome this difficulty is the Long Short-Term Memory architecture, which enables a RNN to learn to prevent the accumulation of information over arbitrarily long durations by empowering it with a "forget gate".

Figure 3.17: Architecture of a LSTM cell. Source: Wikipedia user Guillaume Chevalier. Licensed under Creative Commons.



The parameters of the forget gate can be optimized in such a way that the model can learn to forget the old state at particular moments when the corresponding information is no longer useful in the recurrent computation. This potentially prevents the long-term dependencies which are ultimately responsible for vanishing or exploding gradients. From the outside, a LSTM cell behaves just like a RNN, feeding on input features and producing feature outputs throughout all computation steps. From the inside, however, a LSTM cell has an internal recurrence (a self-loop) in addition to the outer recurrence shared with typical RNNs (Figure 3.17). LSTMs have been sucessfully applied to a wide range of learning tasks over sequential domains, such as handwriting (GRAVES et al., 2009) and speech (GRAVES; MOHAMED; HINTON, 2013) recognition, handwritting generation (GRAVES, 2013) and machine translation (BAHDANAU; CHO; BENGIO, 2014).

3.13 Recurrent Learning Beyond Neural Networks

Over the last decades, RNNs have paved the way for a revolution in machine learning in particular and artificial intelligence as a whole, but they also have their shortcomings. Although one can in principle unroll the same RNN any number of times and thus train it over any arbitrary number of timesteps, the decision is still arbitrary, and this is often undesirable. Consider for example a learning task on the medical history of a patient, in which events can be distributed in any way imaginable (and with any *variance* imaginable) over time. A deep learning engineer will be forced to decide at which level of granularity this timeline must be quantized in order to be fed into a RNN in discrete steps. Ideally, we want to be able to drop this arbitrariness entirely, but to do that one must break out from the concept of a RNN unit. This is what (CHEN et al., 2018a) have done, by intelligently generalizing the definition of a RNN to the continuous domain in a step not much different – kept in due proportion – to what Newton and Leibniz have done with calculus in the eighteenth century. The authors' insight is the following: A RNN, parameterized by the set of parameters ϕ, can be formalized as a function $f(\vec{h}_t, \phi)$ which can be used to compute additive increments to the previous hidden state, updating it:

$$\vec{h}_{t+1} = \vec{h}_t + f(\vec{h}_t, \phi) \tag{3.17}$$

This recurrence relation has a natural mapping to the continuous domain as a differential equation, namely:

$$\frac{\partial \vec{h}(t)}{dt} = f(\vec{h}(t), t, \phi) \tag{3.18}$$

That is to say that the **sequence** of hidden states in the previous formalization is now replaced by a **continuous-time function** of hidden states $\vec{h}(t) \in \mathbb{R} \rightarrow \mathbb{R}^n$. The recurrent neural network analougue $f(\vec{h}(t), t, \phi)$ is now rewritten as the **derivative** of the function of hidden states $\vec{h}(t)$ w.r.t. time.

This approach drops the concept of a neural network entirely, replacing it for a functional unknown which is not to be learned by gradient descent as usual, but *discovered* by (numerically) solving an ordinary differential equation. The feasibility and applications of such a technique are still unknown as we are only a few months ahead of this publication, but one can hypothesize that neural ordinary differential equations (as the authors call them) can have a very disruptive effect on sequence modelling. In particular, it is possible that the technique can be coupled with Graph Neural Networks (the main subject of this thesis, to be described in the next chapter), which suffer from a similar problem as the medical application described before: most often than not, the number of iterations the model runs must be defined on an arbitrary basis.

4 GRAPH NEURAL NETWORKS

Convolutional Neural Networks and Recurrent Neural Networks are applicable, respectively, to domains where spatial and temporal invariance make sense. There is another type of invariance, however, which is central to discrete mathematics in general and combinatorial problem solving in particular: permutation invariance. Consider, as an example, the problem of boolean satisfiability – i.e. determining whether a boolean formula $F(x)$ in the Conjunctive Normal Form[1] admits at least one assignment $x \in \mathbb{B}^n$ of boolean values to its variables which renders its truth value $F(x) = \top$ (SELSAM et al., 2018). Whether $F(x)$ admits a satisfying assignment or not is indifferent to any particular ordering on clauses or literals[2] inside clauses (Figure 4.1).

Figure 4.1: Literals $\neg x_5$ and x_3 can exchange places with no effect to the function $f : \mathbb{B}^4 \to \mathbb{B}$ expressed by this clause. The same applies for any exchange; in effect, the number of equivalent clauses one can obtain this way corresponds to the number of permutations with 5 elements, computed by $5! = 120$.

$$\begin{tikzcd}[column sep=tiny] (x_1 \lor & \neg x_5 \arrow[rr, bend right, leftrightarrow] & \lor\, x_2 \lor & x_3 & \lor \neg x_4) \end{tikzcd} \tag{4.1}$$

However, traditional ANN techniques would require one to encode a CNF formula in a way such that for every possible formula over n variables and m clauses, there would be a number of equivalent encodings combinatorial on n and m. For example, if one enumerates all $2n$ literals, clauses can then be encoded by assigning for each of them a vector $\in \mathbb{B}^{2n}$ with zeros everywhere except for ones in those positions corresponding to the specific literals they contain. For example, for a formula with 5 variables, literals $x_1 \dots x_5$ can be mapped to integers $1 \dots 5$, literals $\neg x_1 \dots \neg x_5$ can be mapped to integers $6 \dots 10$ and thus the clause $(x_1 \lor \neg x_2 \lor x_3)$ can be encoded with the vector $\vec{v} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$[3]. However, all $10!$ different enumerations of literals correspond to exactly the same instance. Encoding all clauses could be done by concatenating their encodings, but, to make matters worse, all $m!$ different permutations of clauses in the encoding also correspond to exactly the same instance, yielding $10! \times m!$ equivalent encodings per

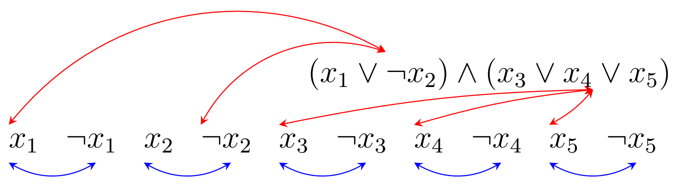
[1] A boolean formula is said to be in the Conjunctive Normal Form if it is described by a logical conjunction over logical disjunctions of boolean variables, i.e. $F = (x_1 \lor x_2 \lor x_5) \land (\neg x_4 \lor x_3)$

[2] A *literal* is a possibly negated variable, e.g. $x_1, \neg x_2$

[3] Mapping integers $0 \leq i \leq n$ to binary, unitary vectors $\in \mathbb{B}^n$ is called *one-hot encoding* in DL parlance.

instance. To train an ANN model to generalize over CNF formulas, one must be able to show it sufficiently many equivalent encodings for the same formula to allow for the memorization of the encoding bias. If we fix $m = 5$, an exhaustive enumeration of equivalent encodings amounts to 4.35456×10^8 per instance, which is still within the computational limits of training ANN models but imposes a significant overhead. Dealing even with CNF formulas of small size becomes rapidly unfeasible ($n = 10, m = 20$ yields $> 8.8 \times 10^{24}$ equivalent encodings per formula).

Figure 4.2: CNF formula $F = (x_1 \vee \neg x_2) \wedge (x_3 \vee x_4 \vee x_5)$ represented as a **graph**: clauses and literals correspond to nodes, edges between clauses and literals are painted red and edges between literals and their complements are painted blue.


$(x_1 \vee \neg x_2) \wedge (x_3 \vee x_4 \vee x_5)$
x_1 $\neg x_1$ x_2 $\neg x_2$ x_3 $\neg x_3$ x_4 $\neg x_4$ x_5 $\neg x_5$

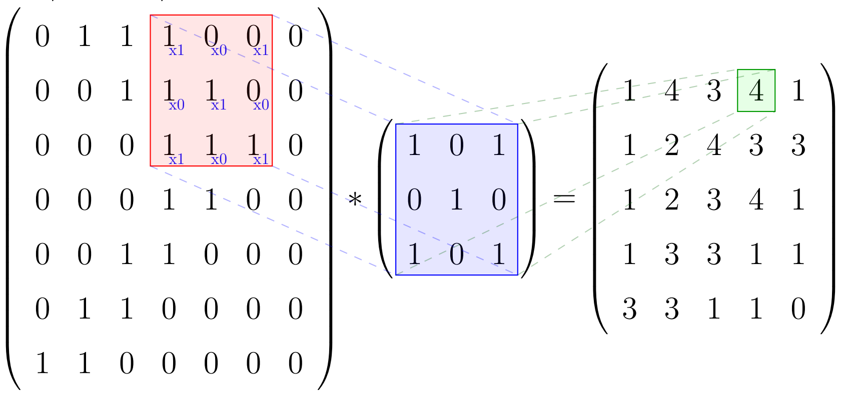
(4.2)

Note that the boolean satisfiability problem discussed above can be described in terms of graphs, because it is **relational** in nature. Each clause references a set of literals, and each negated literal (such as $\neg x_5$) references its non-negated variant (x_5). Or, in other words, each CNF formula can be defined by an adjacency matrix between clauses and literals and an adjacency matrix between literals and literals (Figure 4.2). The same applies for virtually every other symbolic/combinatorial problem defined over variables. The issue raised in the last paragraph, accordingly, applies for ANN models over graphs in general, fundamentally because graph vertices are invariant to permutations. The only true information encoded by a (non-labelled) graph is in its connections (i.e. which vertices connect with what other vertices), which also respond to no particular order. This imposes a significant barrier for bridging deep learning with combinatorial problem solving: in order to learn over graphs, one must be able to enforce permutation invariance on nodes whilst still feeding the model with relational (i.e. edge) data. This problem motivates the study of graph-based deep learning models, of which the fundamental operation is the *graph convolution*.

4.1 Graph Convolutions

Although the neurons of convolutional neural networks' layers have no true position in space (each layer is uniquely defined by its connections with the previous one), it is useful to think that the neurons in each layer are arranged in a rectangular grid. In this context, each neuron of a given layer feeds (by the means of a weighted sum) on the outputs of a rectangular window of neurons in the previous layer. The analogy with a discrete convolution operation is clear: the input layer can be seen as a 2D signal $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2}$ and the set of neural weights associated with the weighted sum (each one directly mapped to a position of that rectangular window) can be seen as a 2D signal $\mathbf{Y} \in \mathbb{R}^{m_1 \times m_2}$. Remember that because CNNs employ parameter sharing, this set of weights is the same for all neurons. Then, one can easily see that a discrete convolution $\mathbf{X} \circledast \mathbf{Y}$ between these two signals is effectively computed by the outputs of the neurons in the second layer.

Figure 4.3: Pictorial representation of a 2D discrete convolution operation. Source: Adapted from Tex Stack Exchange answer <https://tex.stackexchange.com/questions/437007/drawing-a-convolution-with-tikz>.



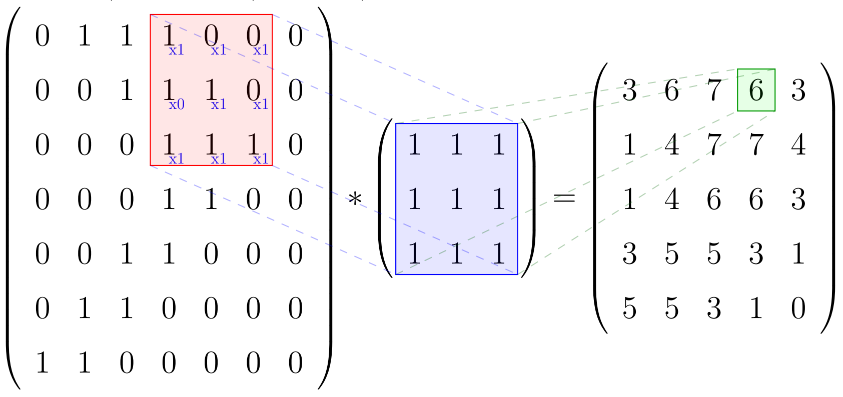
In this context, every CNN logically enforces a grid topology on the input layer (Figure 4.3). The motivation for defining a convolution operation for graphs stems from the necessity of training deep learning models on non-grid topologies, including irregular ones. Conceptually, a graph convolution operation is fairly simple, but the analogy with convolutions in the traditional signal processing context is not immediate. The reader will be referred to (KIPF; WELLING, 2016) for a throughout explanation.

To introduce graph convolutions, let us exploit an analogy with the image context. In the image context, we have a visual signal $\mathbf{X} : \mathbb{N}^{n \times m} \rightarrow \mathbb{R}^3$ corresponding to a mapping between pixel coordinates and RGB triples. By analogy, in the context of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ we have a signal $\mathbf{X} : \mathcal{V} \rightarrow \mathbb{R}^d$ corresponding to a mapping between graph vertices and real-valued multidimensional vectors. The goal of a graph convolution operation is to produce a new signal $\mathbf{X}'$ from $\mathbf{X}$, in the same way that a typical 2D convolution produces a new image by convolving the input image with a convolution kernel. For example, a new vector can be computed for each vertex simply by adding up the vectors of all of its neighbors after applying a transformation:

$$\mathbf{X'}_i = \sum_{v_j \in \mathcal{N}_{v_i}} \mathbf{X}_j \tag{4.3}$$

Where $\mathbf{X}_i$ and $\mathbf{X'}_i$ denote the vectors corresponding to vertex v_i in the mappings $\mathbf{X}$ and $\mathbf{X}'$ respectively. Note how this is analogous to performing a traditional 2D convolution with a "sum" kernel (i.e. a matrix of ones). In the context of images, this corresponds to computing the value of each pixel in the resulting image as the sum of the values of all of its 9-connected neighbors (Figure 4.4).

Figure 4.4: Pictorial representation of a 2D discrete convolution operation with a "sum" kernel. Source: Adapted from Tex Stack Exchange answer <https://tex.stackexchange.com/questions/437007/drawing-a-convolution-with-tikz>.



However, up until now, our operation has no trainable parameters, which blocks us from perform deep learning over graphs. This can be amended by defining a trainable function $f : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$. We can then update Equation 4.3 to have the new vector for each vertex be computed by adding up the vectors of all of its neighbors **after applying the transformation** f:

$$\mathbf{X'}_i = \sum_{v_j \in \mathcal{N}_{v_i}} f(\mathbf{X}_j) \tag{4.4}$$

Note that what enables us to perform deep learning over graphs using graph convolutions is the transformation f, which can be parameterized and thus trainable with gradient descent.

Equation 4.4 is usually referred to as a graph convolution *layer*. One can in principle stack an arbitrary number of such layers sequentially. This corresponds to a production of a sequence of signals $\mathbf{X}', \mathbf{X}'', \mathbf{X}''', \ldots$. Note that because the transformations f can map to vector spaces of any dimensionality, the dimensionality of the vectors associated with each vertex at each point in the sequence can vary. This is analogous to how the number of "channels" for each pixel in a CNN architecture can vary throughout layers.

One may also want to apply another transformation g to the result of the sum in Equation 4.4, for example:

$$\mathbf{X}'(v_i) = g\left(\sum_{v_j \in \mathcal{N}_{v_i}} f(\mathbf{X}(v_j))\right) \tag{4.5}$$

In some definitions (KIPF; WELLING, 2016), the vertex-wise transformation f corresponds to applying a linear layer to the original vector, and the sum-wise transformation g corresponds to a non-linearity σ:

$$\mathbf{X}'(v_i) = \sigma\left(\sum_{v_j \in \mathcal{N}_{v_i}} \mathbf{W} \times \mathbf{X}(v_j)\right) \tag{4.6}$$

It also should be noted that if the input graph has no self edges, the original vector $\mathbf{X}(v_i)$ will not be considered in the equation which computes the new vector $\mathbf{X}'(v_i)$. This can be undesirable, as the new state for a given vertex will be computed only as function of its neighbors, not itself. This can be amended by adding a self-edge to every vertex in the graph, a technique which is common practice on graph convolutional neural networks.

Algorithm 1 Graph Convolutional Neural Network.

```
1: procedure GraphConvNet($\mathcal{G} = (\mathcal{V}, \mathcal{E})$)
2:     Initialize vertex embeddings $\mathbf{X}_i \in \mathbb{R}^{d_1} \mid \forall v_i \in \mathcal{V}$
3:     for $k = 1 \ldots K$ do
4:         for $v_i \in \mathcal{V}$ do
5:             // $f_k$ is a function $f_k : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}}$
6:             $\mathbf{X}_i \leftarrow \sum_{v_j \in \mathcal{N}(v_i)} f_k(\mathbf{X}_i)$
7:             // Thus, $X_i$ is now $\in \mathbb{R}^{d_{k+1}}$
8:     return $\{\mathbf{X}_i\}_{i=1\ldots|\mathcal{V}|}$
```

We are now enabled to define a generic graph convolutional neural network architecture with K layers (Algorithm 1). Note that we initialize vertex embeddings with dimensionality d_1. This is analogous to how the pixels of the input layer of a typical CNN have dimensionality 3 (i.e. three channels). We also define a series of functions $f_1, f_2, \ldots f_K$. Each of these functions will be applied to each vertex embedding, and it may be the case that they map from a given dimensionality to a different one. For example, one can have initial vertex embeddings $\in \mathbb{R}^{30}$ and $f_1 : \mathbb{R}^{30} \to \mathbb{R}^{50}$. In this scenario, after the first convolutional layer, vertex embeddings will be $\in \mathbb{R}^{50}$.

4.2 Graph Recurrent Neural Networks

Instead of stacking many different graph convolution layers sequentially, one may borrow insights from recurrent models and share the parameters of graph convolution layers over time. Or, in other words, one can define a single convolution layer and "unroll" it, in a RNN fashion, for many iterations. This corresponds to defining, in addition to the function $f : \mathbb{R}^d \to \mathbb{R}^{d'}$ of Equation 4.4, another function $\phi : \mathbb{R}^{d'} \times \mathbb{R}^{d'} \to \mathbb{R}^d$ such that we can define:

$$\mathbf{X}_i^{(t+1)} = \phi\left(\mathbf{X}_i^{(t)}, \sum_{v_j \in \mathcal{N}_{v_i}} f\left(\mathbf{X}_j^{(t)}\right)\right) \tag{4.7}$$

Equation 4.7 can be read in the following way:

1. ϕ is a recurrent neural network

2. $\mathbf{X}_i^{(t)}$ is the vector associated with vertex v_i at time $= t$. $\mathbf{X}_i^{(t)}$ is also the *hidden state* of the RNN ϕ at time $= t$
3. Upon receiving a hidden state $h = \mathbf{X}_i^{(t)}$ and a input $x = \sum_{v_j \in \mathcal{N}_{v_i}} f\left(\mathbf{X}_j^{(t)}\right)$ (which corresponds to an aggregation of messages), the RNN ϕ produces a new hidden state $h' = \mathbf{X}_i^{(t+1)}$.

We are now enabled to describe a graph recurrent neural network in pseudocode (Algorithm 2). Note that, in opposition to Algorithm 1, we are annotating vertex embeddings with their "timestamps" (for example, $\mathbf{X}_i^{(t)}$ is the vertex embedding at time $= t$). We **could** have done the same for the graph convolutional model, annotating the output at each k-th layer as $\mathbf{X}_i^{(k)}$, but instead we chose to overwrite the same variable $\mathbf{X}_i$ many times for simplicity.

Algorithm 2 Graph Recurrent Neural Network.

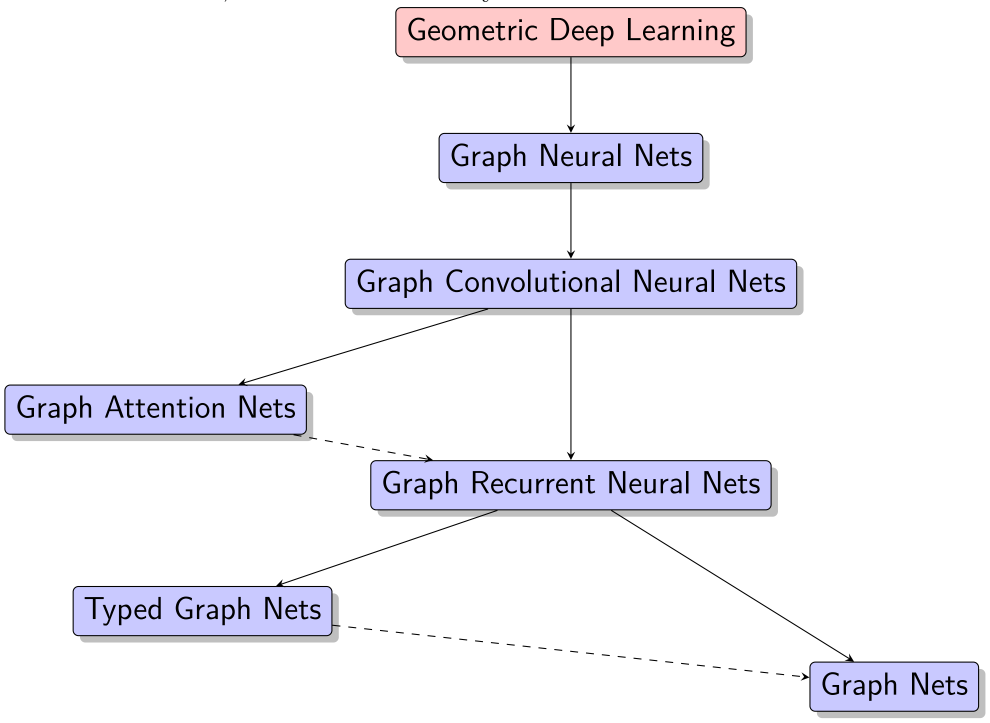
```
1: procedure GraphRecNet($\mathcal{G} = (\mathcal{V}, \mathcal{E})$)
2:     Initialize vertex embeddings $\mathbf{X}_i^{(1)} \in \mathbb{R}^d \mid \forall v_i \in \mathcal{V}$
3:     for $t = 1 \dots T$ do
4:         for $v_i \in \mathcal{V}$ do
5:             // $f$ is a function $f : \mathbb{R}^d \to \mathbb{R}^{d'}$
6:             // $\phi$ is a function $\phi : \mathbb{R}^{d'} \times \mathbb{R}^{d'} \to \mathbb{R}^d$
7:             $\mathbf{X}_i^{(t+1)} \leftarrow \phi\left(\mathbf{X}_i^{(t)}, \sum_{v_j \in \mathcal{N}(v_i)} f_k\left(\mathbf{X}_i^{(t)}\right)\right)$
8:             // Thus $\mathbf{X}_i^{(t+1)} \in \mathbb{R}^d$
9:     return $\{\mathbf{X}_i^{(T)}\}_{i=1\dots|\mathcal{V}|}$
```

It should be noted that although we have defined graph **convolutional** neural networks and graph **recurrent** neural networks as two distinct architectures, the convolutional and recurrent aspects are both present in graph recurrent neural networks. This is due to the fact that vertex embeddings are updated in a recurrent fashion with inputs computed by graph convolutional layers.

4.3 Graph Neural Networks in General

Because deep learning over graphs is a relatively recent field, the taxonomy of different architectures is somewhat still debated, with some authors preferring one label over another for the same underlying model. For this reason, we have made an effort to describe in detail the taxonomy adopted for this thesis, a diagram of which is presented in Figure 4.5. First of all, the overarching *field* of deep learning we are interested in is *geometric deep learning*. Geometric deep learning is concerned with learning problems defined over non-euclidean spaces, which can be modelled as graphs. Then, for the purposes of this thesis, a Graph Neural Network (GNN) is any neural architecture which feeds on graphs enforcing permutation invariance among vertices. A Graph Convolutional Neural Network (GCN) is any kind of GNN which makes use of graph convolutional layers. A Graph Attention Network (GAT) is any kind of GNN which employs attention mechanisms over neighbors (VELIČKOVIĆ et al., 2017). A Graph Recurrent Neural Network (GRN) is any kind of GNN which updates vertex embeddings with a recurrent unit (RNN, LSTM, GRU). A Graph Network, as defined by (BATTAGLIA et al., 2018), is a GRN which assigns embeddings to vertices, embeddings to edges and an embedding to the entire graph. Finally, a Typed Graph Network (TGN) (PRATES et al., 2019), which is a conttribution of this thesis, extends the concept of vertex / edge / graph embeddings by partitioning graph vertices into different *types*.

Figure 4.5: Taxonomy of neural architectures in the graph neural network family. The overarching *field* of deep learning to which these architectures belong to, painted in red, is *geometric deep learning*. Architecture families are painted in blue. The dashed line indicates that a graph recurrent neural net can be implemented using attention mechanisms, but not necessarily.



4.4 Mechanics of Graph Neural Networks

Up until now, we have defined GNNs based on their architectural description. It remains to be seen, however, how GNNs operate to solve a problem. In this context, it should be noted that even though GNNs can be run for any number of iterations and feed on relational / symbolic data, their *modus operandi* could not be more different than that of traditional methods in the symbolic AI family.

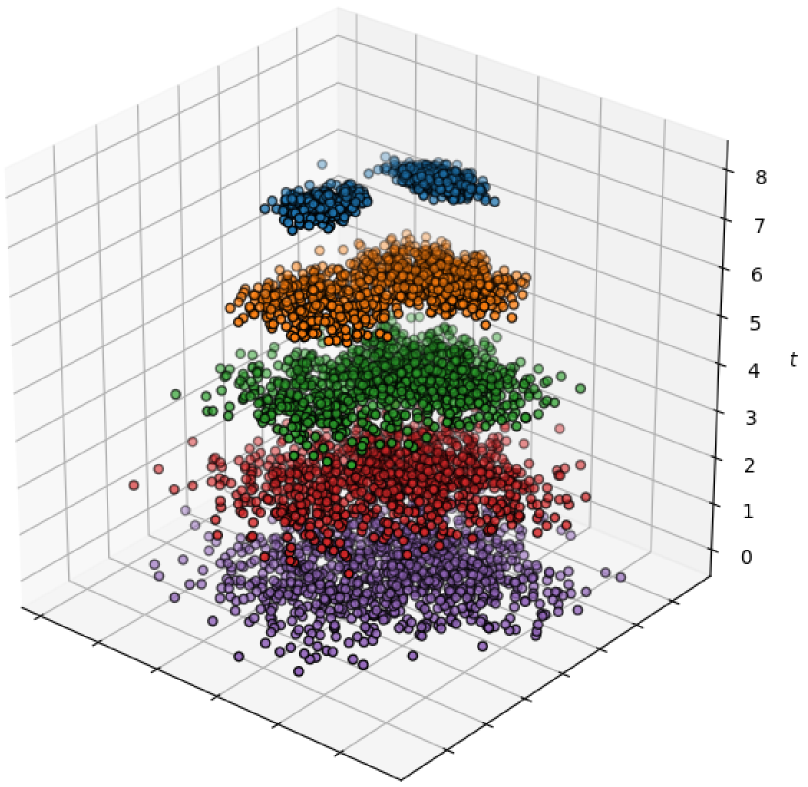
Figure 4.6: Example of a symbol manipulation operation: the operational semantics of a while loop command in an imperative language.

$$\frac{\langle B, s\rangle \implies \top}{\langle \textbf{while } B \textbf{ do } C, s\rangle \to \langle C; \textbf{while } B \textbf{ do } C, s\rangle} \qquad \frac{\langle B, s\rangle \implies \bot}{\langle \textbf{while } B \textbf{ do } C, s\rangle \to s}$$

GNNs do not **manipulate** symbols (Figure 4.6) – at least not directly –, but rather refine **projections** of these symbols into hyperdimensional real spaces. The divide between symbol manipulation and tensor algebra is at the heart of important discussions regarding artificial intelligence today, such as the longstanding debate between Gary Marcus and some members of the deep learning community such as Yann LeCun. A sharp critic of some aspects of the recent deep learning hype (MARCUS, 2018a; MARCUS, 2018b) – such as "the notion that deep learning is without demonstrable limits and might, all by itself, get us to general intelligence" (MARCUS, 2018c), Marcus has defended that Artificial General Intelligence[4] will require a combination of deep learning techniques (relying on tensor algebra) with discrete symbol manipulation operations. LeCun, by contrast, suggests that "whatever we do [to achieve AGI], DL is part of the solution", from which he deduces that fundamentally vectors/tensors are required instead of symbols (MARCUS, 2018c). Graph Neural Networks fit into an interesting position in this debate, as the triumph of GNNs is that, in many cases, a manipulation of projections can be learned which captures symbolic transformations as an emergent property. Although there is not sufficient evidence pointing towards any direction, we have reason to believe that the polished and well-behaved rules governing symbolic domains can emerge from the messy world of tensors, as something similar is conceivably what happens inside our brains.

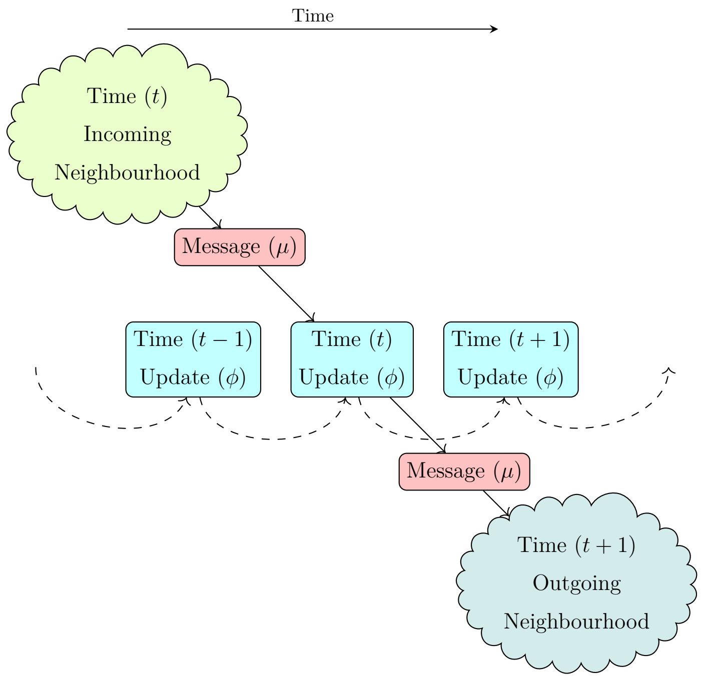
[4]Artificial General Intelligence, or AGI refers to an artificial intelligence capable of performing any intellectual task a human being can (KURZWEIL, 2010).

Figure 4.7: The computation performed by a Graph Recurrent Neural Network can be interpreted as the iterative refinement (over many message-passing iterations) of an initial projection $\mathcal{P}_0 : \mathcal{V} \rightarrow \mathbb{R}^d$ of graph vertices into hyperdimensional space. A successfully trained GNN model will be capable of refining a projection which captures some property of the learned problem, for example a 2-partitioning of $\mathcal{V}$. The Figure shows a pictorial representation of a sequence of progressively refined projections $\mathcal{P}_0, \mathcal{P}_2, \mathcal{P}_4, \mathcal{P}_6, \mathcal{P}_8$ over $t_{max} = 8$ message-passing timesteps. Source: Author.



Recall that a GRNN assigns a vector (called *embedding*) $\mathbf{X_i} \in \mathbb{R}^d$ to each vertex $\in \mathcal{V}$. This vector acts as a memory storage unit for that vector, which is updated (by the function ϕ) at every new message-passing iteration. If we consider all embeddings at once, this collection can be thought both as a distributed memory storage system and as a **projection** of graph vertices $\in \mathcal{V}$ into the hyperdimensional real space $\mathbb{R}^d$. Following the second interpretation, the operation of a GNN can be understood as the iterative refinement of an initial projection $\mathcal{P}_0 : \mathcal{V} \rightarrow \mathbb{R}^d$ into a final projection $\mathcal{P}_{t_{max}} : \mathcal{V} \rightarrow \mathbb{R}^d$ which reshapes the spatial distribution of vertex embeddings into some geometrical shape from which useful information can be decoded. Figure 4.7 for example imagines the execution of a GNN whose ultimate effect is to 2-partition vertex embeddings into two clusters. In general we are not directly interested on the distribution of these embeddings in hyperdimensional space but rather on some reduction which is to be performed over all of them. For example, a GNN model can be trained such that the magnitude of the average embedding (over all vertices) predicts some scalar property of the input graph. In practice, however, it makes sense to perform some complex computation over each vertex embedding before reduction. The rationale here is that vertex embeddings can be left free to act as memory storage units rather than as the output for the learned problem, which can be decoded from them by some function. Architecturally speaking, this function can be implemented as a MLP and its parameters can be learned by the trained model.

Figure 4.8: Pictorial representation of a Graph Neural 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 μ and messages are aggregated and fed to the update function ϕ, which produces an updated embedding for v. Simultaneously, v sends messages to vertices in its outgoing neighbourhood, which will undergo the same update process. Reproduced with authorization from Pedro Avelar (PRATES et al., 2019).


Time
Time (t) Incoming Neighbourhood
Message (μ)
Time ($t-1$) Update (ϕ)
Time (t) Update (ϕ)
Time ($t+1$) Update (ϕ)
Message (μ)
Time ($t+1$) Outgoing Neighbourhood

GNNs can also be understood in terms of the information flow running through their pipeline, in the spirit of the diagram in Figure 3.16. Although the computation flow is still centered on RNNs, in this case the inputs fed to the recurrent units at each timestep do not originate from the external world but are rather indirectly produced by themselves in the last timestep. Figure 4.8 illustrates this process in a diagram.

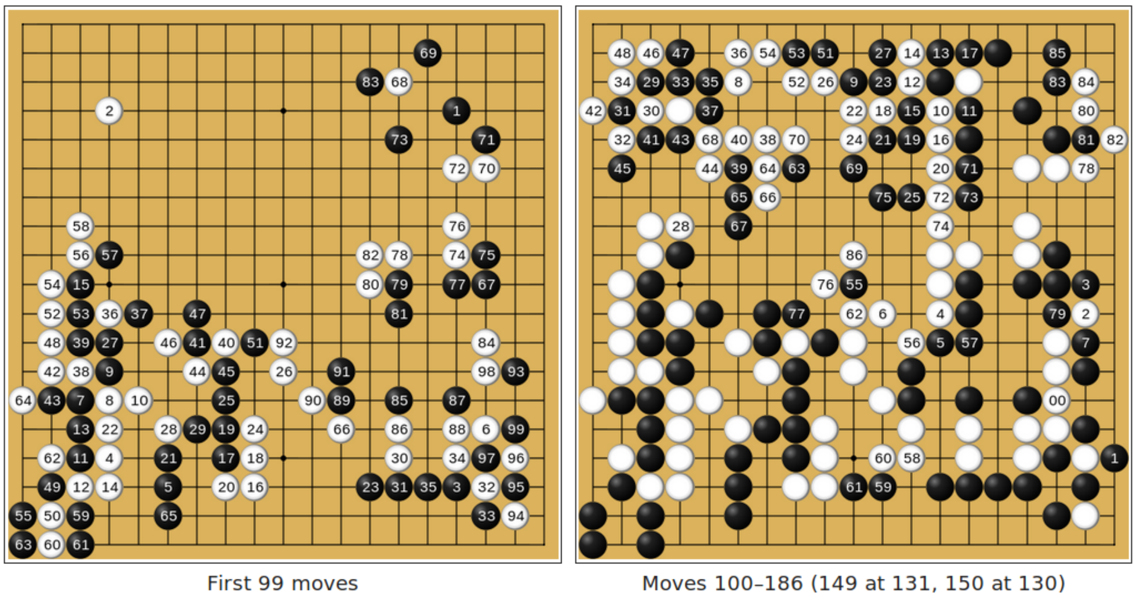
4.5 Motivations for Graph Neural Network Research and Recent Advances in the GNN Family

Graph neural network research has underwent significant progress in the last few years, most notably since 2016. The plot in Figure 4.10 visualizes the growth in the number of publications related to topics in GNN family, as measured according to a manually curated list of relevant publications enumerated below:

- 2005 (3 publications): (GORI; MONFARDINI; SCARSELLI, 2005; SCARSELLI et al., 2005; BIANCHINI et al., 2005)
- 2006 (4 publications): (MONFARDINI et al., 2006; MASSA et al., 2006; PUCCI et al., 2006; YONG et al., 2006)
- 2009 (4 publications): (SCARSELLI et al., 2009b; SCARSELLI et al., 2009a; MICHELI, 2009; LU et al., 2009)
- 2010 (3 publications): (NOI et al., 2010; BANDINELLI; BIANCHINI; SCARSELLI, 2010; MURATORE et al., 2010)
- 2011 (2 publications): (QUEK et al., 2011; UWENTS et al., 2011)
- 2013 (2 publications): (BORDES et al., 2013; BRUNA et al., 2013)
- 2015 (4 publications): (LI et al., 2015; DUVENAUD et al., 2015; TANG et al., 2015; HENAFF; BRUNA; LECUN, 2015)
- 2016 (10 publications): (DEFFERRARD; BRESSON; VANDERGHEYNST, 2016; NIEPERT; AHMED; KUTZKOV, 2016; ATWOOD; TOWSLEY, 2016; KIPF; WELLING, 2016; BATTAGLIA et al., 2016; BELLO et al., 2016; GROVER; LESKOVEC, 2016; DEFFERRARD; BRESSON; VANDERGHEYNST, 2016; CHANG et al., 2016; KIPF; WELLING, 2016)
- 2017 (21 publications): (RAPOSO et al., 2017; SANTORO et al., 2017; WU et al., 2017; WATTERS et al., 2017; HOSHEN, 2017; OÑORO-RUBIO et al., 2017; HAMAGUCHI et al., 2017; GILMER et al., 2017; VASWANI et al., 2017; HAMRICK et al., 2017; PASCANU et al., 2017; TOYER et al., 2017; NOWAK et al., 2017; KHALIL et al., 2017; JOHNSON, 2017; DURAN; NIEPERT, 2017; BRONSTEIN et al., 2017; MORAVČÍK et al., 2017; GARCIA; BRUNA, 2017; ALLAMANIS; BROCKSCHMIDT; KHADEMI, 2017; VELICKOVIC et al., 2017)
- 2018 (22 publications): (LI et al., 2018b; HU et al., 2018; STEENKISTE et

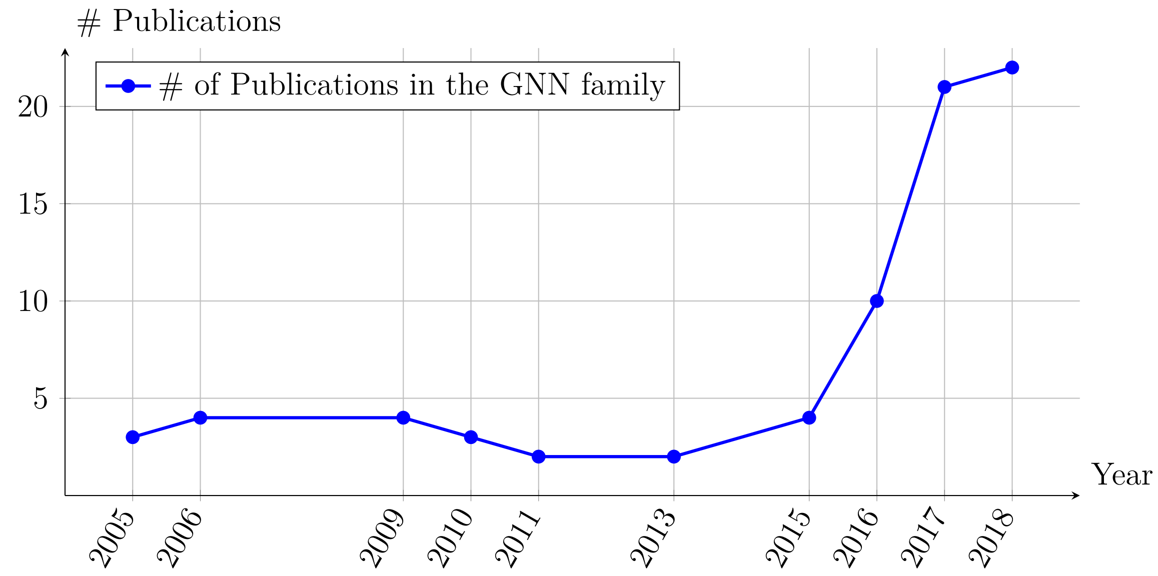
al., 2018; SANCHEZ-GONZALEZ et al., 2018; KIPF et al., 2018; CUI et al., 2018; WANG et al., 2018b; WANG et al., 2018c; CHEN et al., 2018b; SHAW; USZKOREIT; VASWANI, 2018; GULCEHRE et al., 2018; WANG et al., 2018a; HAMRICK et al., 2018; ZAMBALDI et al., 2018; SELSAM et al., 2018; YOON et al., 2018; LI et al., 2018a; CAO; KIPF, 2018; YOU et al., 2018; BOJCHEVSKI et al., 2018; PRATES et al., 2018; AVELAR et al., 2018)

Figure 4.9: Moves 1-186 of AlphaGo Master vs professional Go player Tang Weixing (31 December 2016), won by resignation by AlphaGo Master. AlphaGo is an example of a DL-fueled technology to solve combinatorial problems, but it does not constitute end-to-end learning.



The significant growth in number of GNN-related publications in the last two years coheres with a recent interest from the AI community in applying deep learning to combinatorial / symbolic domains. (BATTAGLIA et al., 2018) suggest that “a key path forward for modern AI is to commit to combinatorial generalization as a top priority”. (BENGIO; LODI; PROUVOST, 2018) additionally argue that expert knowledge may not be sufficient to design effective heuristics for combinatorial optimization problems, suggesting that ML offers itself as an alternative as it is well suited for signals for which no clear mathematical formulation is known, adding that “deep learning excels when applied in high dimensional spaces with a large number of data points”. Moreover, as (BENGIO; LODI; PROUVOST, 2018) also argue, operations research and machine learning are closely related through optimization, as ML is interested in minimizing the error between predictions and targets. This is

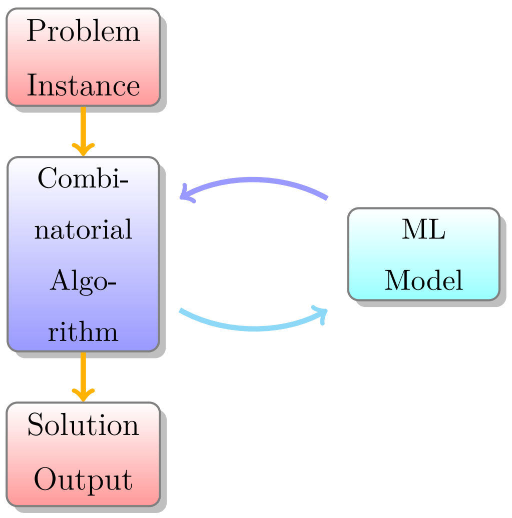
Figure 4.10: Growth in the number of publications related to topics in the GNN family, as measured according to a manually curated set of relevant papers. Since 2016, the field has experienced a significant boom in quantity of publications as a result of the increased interest by end-to-end differentiable models of relational reasoning.



an invitation for bridging both fields, as they can overlap or complement each other in some areas.

Naturally, deep learning has been applied to combinatorial domains in more than one occasion. One can envision a decision-making process fueled by a DNN "brain" which can be trained to yield appropriate decisions for each scenario. This is essentially how DeepMind's highly successful Go playing algorithm AlphaGo (SILVER et al., 2018) (Figure 4.9) structures itself: the evaluation of the quality of each board configuration is up to a DNN, but ultimately this DNN module fits into a Monte Carlo tree search algorithm. The structure of the AlphaGo algorithm has sparked controversy in the AI community when DeepMind published AlphaGo Zero, a successor to AlphaGo which was able to reach superhuman levels purely through

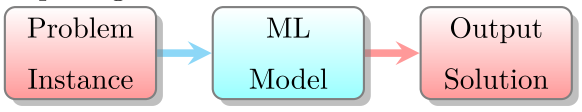
Figure 4.11: When ML is combined with combinatorial algorithms, a ML module is called upon possibly many times but an output solution is ultimately produced by the combinatorial algorithm.



self-play. DeepMind promoted AlphaZero as evidence that ML models can learn complex tasks from scratch, but (MARCUS, 2018b) argued that the claim that AlphaGo Zero starts *tabula rasa* is overstated considering that the Monte Carlo tree search structure was not **learned** from data but rather **built innately** into the model. This brings us into the distinction between using machine learning **alongside** combinatorial algorithms (as AlphaGo does) and **end-to-end learning**. End-to-end learning refers to training a ML model which is able to provide output solutions directly from the input instance (Figure 4.12), as opposed to training an ML model which acts as a subroutine of a larger algorithm (Figure 4.11).

End-to-end learning has a number of advantages over other approaches, both from the engineering perspective and perhaps most importantly from the scientific standpoint.

Figure 4.12: In end-to-end learning, machine learning provides a solution directly from the problem instance input, without requiring external tools.



By reducing the impact of engineers' biases in the model, we might be able to achieve greater levels of accuracy in the proposed algorithm, which is desirable for pragmatical reasons. But leaving ML free to learn its own associations is also exciting as it potentially opens doors in algorithm design, shedding light on as of yet esoteric novel ways to solve a traditional problem. This partially justifies the recent increased interest by GNNs, which are **end-to-end differentiable** models of machine learning – that is, they are self-contained recipes for solving problems which are differentiable throughout their entire pipeline, allowing for gradients to flow from input to output and therefore lending themselves to improvement purely through gradient descent. Another way to think about it is to understand that GNNs have no "moving parts" in the same way that AlphaGo, for example, has: a GNN is entirely described by a continuous hyperdimensional function. This yields for free support for a significant range of techniques which have been perfected for deep learning in the last decade, such as using GNNs to implement Generative Adversarial Networks (GANs) (Figure 4.13) (GOODFELLOW et al., 2014) or producing feature visualizations of the associations learned by GNNs by performing gradient ascent on the activation of some neurons as a function of the problem inputs, as in DeepDream (Figure 4.14) (CASTELVECCHI, 2016).

Figure 4.13: Convolutional and Deconvolutional Neural Networks are end-to-end differentiable ML models. This enables one to connect the output of a "generator" model, implemented as a deconvolutional network, to the input of a "discriminator" model, implemented as a CNN. The discriminator is fed with 50% real images and 50% artificial images (produced by the generator) with the goal of discriminating between the two classes, while the goal of the generator is to maximize the classification error of the discriminator. Because both models are end-to-end differentiable, one can flow gradients from the beginning of the generator pipeline until the end of the discriminator's. The end result are two models which evolve simultaneously through competition in a zero-sum game. Ultimately, the generator network becomes able to produce photo-realistic images of high quality, such as in these examples. Source: images produced by the author using the Google Colab BigGAN demo <https://colab.research.google.com/github/tensorflow/hub/blob/master/examples/colab/biggan_generation_with_tf_hub.ipynb>.

In the original GNN publication (GORI; MONFARDINI; SCARSELLI, 2005), the authors instantiate the proposed model to learn A) the subgraph matching problem, B) the mutagenesis problem from bioinformatics and C) web page ranking. The initial experiments advocate the model's applicability to a diverse set of fields, a prediction which has stood the test of time over the last decade. Since the early stages of GNN research, it became clear that it held the potential for learning object recognition tasks on images, which in many contexts can be effortlessly segmented into regions and represented as graphs (BIANCHINI et al., 2005). The triumph of training object recognition tasks on preprocessed graph descriptions of images as opposed to the original images themselves is that permutation invariance among objects in each scene is automatically enforced by the graph representation, a feature not shared by typical approaches such as CNNs (although capsule networks take a step in this direction (SABOUR; FROSST; HINTON, 2017)). In a recent work, (RAPOSO et al., 2017) show how GNNs can perform object-relation reasoning. The authors factor images into "scenes" comprised of multiple interconnected objects and successfully train "relational networks" (a specialized GNN) to annotate edges with the inferred relations between the corresponding objects. Similarly, (SANTORO et al., 2017) show how DL models can be trained to answer complex relational reasoning queries over images obtained from the CLEVR dataset (JOHN-

Figure 4.14: The end-to-end differentiability of CNNs enables one to perform gradient ascent on the activation of a given neuron (or set thereof), where the parameters considered are the pixels of the input image. Over sufficient gradient ascent iterations, an input image is morphed to exaggerate the features which excite those neurons, possibly yielding surreal looking images conventionally called "DeepDreams". Strange as it is, the DeepDream treatment is a powerful technique for feature visualization, helping researchers and engineers understand which features are weighed in the most by each neuron (SIMONYAN; VEDALDI; ZISSERMAN, 2013). Source: Wikipedia user MartinThoma. Licenced under CC0 1.0 Universal.

SON et al., 2017). GNNs have also shown promise on symbolic domains: (LI et al., 2015) successfully train GNN models to solve bAbI formulations (WESTON et al., 2015) describing relational tasks such as path finding and basic deductions, also showing how can they can achieve state-of-the-art results in program verification tasks such as inferring program invariants, which have also been explored by (ALLAMANIS; BROCKSCHMIDT; KHADEMI, 2017). In a similar context, GNNs have been applied to combinatorial optimization (BELLO et al., 2016; NOWAK et al., 2017; KHALIL et al., 2017) and constraint satisfaction (SELSAM et al., 2018), as well as to the task of simulating varied types of automata (JOHNSON, 2017). GNNs have found a fertile niche in chemical learning tasks, having been applied to quantum chemistry (GILMER et al., 2017) and to learning molecular fingerprints (DUVENAUD et al., 2015), as well as producing molecules from examples (CAO; KIPF, 2018; LI et al., 2018a). GNNs have also been successfully applied to control and planning on more than one occasion (WANG et al., 2018a; HAMRICK et al., 2017; HAMRICK et al., 2018; ZAMBALDI et al., 2018; TOYER et al., 2017).

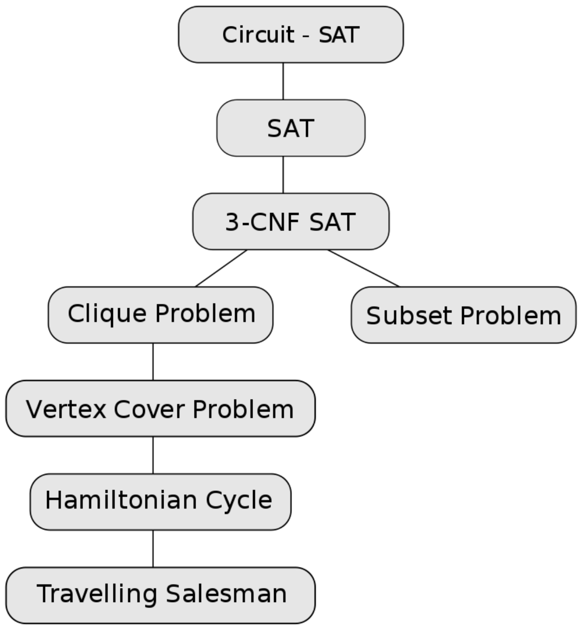
Recently an effort has been made into training generative models of graphs, neural models which are able to reproduce and extrapolate distributions of graphs which they have seen during training (LI et al., 2018a; CAO; KIPF, 2018; YOU et al., 2018; BOJCHEVSKI et al., 2018). Many approaches have been taken, but fundamentally such a model should be able to project input graphs into a hyperdimensional real space, a task particularly suited for GNNs.

In the remainder of this section, we will explore some relevant examples of GNNs proposed in the scientific literature, explaining how they work and how they fit into the state of the art.

4.5.1 NeuroSAT

The boolean satisfiability problem (SAT) has occupied an important position in theoretical and applied computer science since its inception, particularly as a result of how it fits historically in the development of computational complexity theory. SAT was the first problem proved to be $\mathcal{NP}$-Complete, by (COOK, 1971) and independently by (LEVIN, 1973). The Cook-Levin theorem, as it is now known, works by showing that for every nondeterministic Turing Machine M running on polynomial time[5] and for each possible input string x to M, we can construct a boolean expression B, whose number of variables is polynomial on $|x|$, which is satisfiable[6] if and only if M accepts x. Because B can be translated to the CNF, this result ultimately describes the difficulty of solving SAT as at least as hard as that of every other problem solvable in polynomial time by a nondeterministic Turing Machine. Or, in computational complexity theory

Figure 4.15: Tree of reductions between $\mathcal{NP}$-Complete problems, showing which problems A have their $\mathcal{NP}$-Completeness proof relying on another problem B to perform polynomial-time reduction from B to A. Source: Wikipedia user Gian Luca Ruggero. Public Domain.



[5]i.e. accepting or rejecting each input string of size n in time $\leq p(n)$ where $p = n^k$ is a polynomial function

[6]i.e. admits an assignment of truth values to every variable which renders the expression $\top$

parlance, one can say that SAT is at least as hard as every problem in $\mathcal{NP}$[7], making it a $\mathcal{NP}$-Hard problem. Because SAT itself is in $\mathcal{NP}$, it additionally belongs to the intersection $\mathcal{NP} \cap \mathcal{NP}$-Hard $= \mathcal{NP}$-Complete.

For reasons which will become clear in the following chapters, the complexity class $\mathcal{NP}$-Complete plays a fundamental role in theoretical computer science. The importance of SAT, in this context, is in its position as one of the conceptually simplest $\mathcal{NP}$-Complete problems, making it the backbone of a large number of $\mathcal{NP}$-Completeness proofs (Figure 4.15). The simplicity of SAT also enables one to effortlessly describe more complex problems in terms of boolean expressions and solve them indirectly using SAT solvers, the efficiency of which has been consistently perfected over the last decades. SAT is of paramount importance to a wide range of fields such as model checking, formal verification, automated theorem proving and circuit design.

Most approaches to SAT solving consist on backtracking-based search, with perhaps the most significant example being the Davis–Putnam–Logemann–Loveland (DPLL) algorithm (DAVIS; PUTNAM, 1960). The DPLL algorithms relies on enhancements to the backtracking procedure, such as unit propagation and pure literal elimination, which prune several computation paths and on practice make the solving process more efficient, although its worst-case time complexity remains $\mathcal{O}(2^n)$. Over the decades, industrial SAT solvers have accumulated a collection of tricks and heuristics, pushing the empirical complexity of SAT solving closer and closer to acceptable values (AUDEMARD; SIMON, 2018). A significant number of important decisions which must be taken by SAT solvers is somewhat arbitrary in nature, such as deciding which variable to branch next. From a ML perspective, this suggests that there are aspects of SAT solving which could be learned from example. This is an invitation to tackle SAT with deep learning, an opportunity which (SELSAM et al., 2018) have explored in their paper.

As we have already discussed in the beginning of this chapter, training a SAT solver with deep learning could be done by encoding CNF formulas with tensors in a one-hot encoding fashion. This approach comes with severe limitations, however, as there is an exponential number of equivalent encodings of the same formula which can be obtained by permutation over clauses or literals. Ideally, we want to enforce permutation invariance in the model, a task for which GNNs are particularly well-

[7]$\mathcal{NP}$ is defined as the class of problems solvable in polynomial time by a nondeterministic Turing Machine

Figure 4.16: CNF formula $F = (x_1 \vee \neg x_2) \wedge (x_3 \vee x_4 \vee x_5)$ represented as a **graph**: clauses and literals correspond to nodes, edges between clauses and literals are painted red and edges between literals and their complements are painted blue. Source: Author.

$$\begin{tikzcd}
 & & & & (x_1 \vee \neg x_2) \arrow[dllll, bend right, <->] \arrow[dl, bend right] & \wedge & & & (x_3 \vee x_4 \vee x_5) \arrow[dllll] \arrow[dll] \arrow[d] & \\
x_1 \arrow[r, bend right, <->] & \neg x_1 & x_2 \arrow[r, bend right, <->] & \neg x_2 & x_3 \arrow[r, bend right, <->] & \neg x_3 & x_4 \arrow[r, bend right, <->] & \neg x_4 & x_5 \arrow[r, bend right, <->] & \neg x_5
\end{tikzcd} \tag{4.8}$$

suited. In (SELSAM et al., 2018), the authors' idea was to think about a CNF formula in terms of both its literals and its clauses, connecting literals to clauses to which they pertain and additionally connecting literals to their negated variants (x_2 and $\neg x_2$). For illustration, we repeat here the same diagram from the beggining of the chapter in Figure 4.16. Immediately, we see that a SAT instance can be represented as a graph, allowing the problem to be instantiated into a GNN model. However, the distinction between clauses and literals (which correspond to different "types" of nodes in the graph) imposes a challenge for the training process: how can we expect the model to differentiate between nodes corresponding to literals and nodes corresponding to clauses? Without some tricks, the GNN formalization defined in Algorithm 2 does not support this. However, we will use the "NeuroSAT" model defined by (SELSAM et al., 2018) to show how this differentiation can be achieved, and during the course of the explanation establish the basis of what must be modified in the original formalization to allow for the additional expressiveness required.

Suppose that instead of projecting nodes of a graph into hyperdimensional real space, we separately project literal vertices into one space and clause nodes into a separate one. Concretely, this corresponds to instantiating two different RNNs to perform embedding updates on literals and clauses separately. In this context, we can simply send messages between literals and clauses according to the connections illustrated in Figure 4.16. We also wish to define a MLP $\mu^{\mathcal{L}\to\mathcal{C}}$ to compute messages from literals to clauses and a MLP $\mu^{\mathcal{C}\to\mathcal{L}}$ to compute messages from clauses to literals. Because there are now two "types" of vertices, it makes sense to define three adjacency matrices, one between clauses and literals ($\mathbf{M}_{\mathcal{C}\mathcal{L}}$), one between literals and literals ($\mathbf{M}_{\mathcal{L}\mathcal{L}}$) and one between clauses and clauses ($\mathbf{M}_{\mathcal{C}\mathcal{C}}$), although we will only make use of the first two. If we denote by ${\mathbf{V}_{\mathcal{L}}}^{(t)}$ the tensor of literal embeddings and by ${\mathbf{V}_{\mathcal{C}}}^{(t)}$ the tensor of clause embeddings, we can finally describe the model with the follwing set of equations:

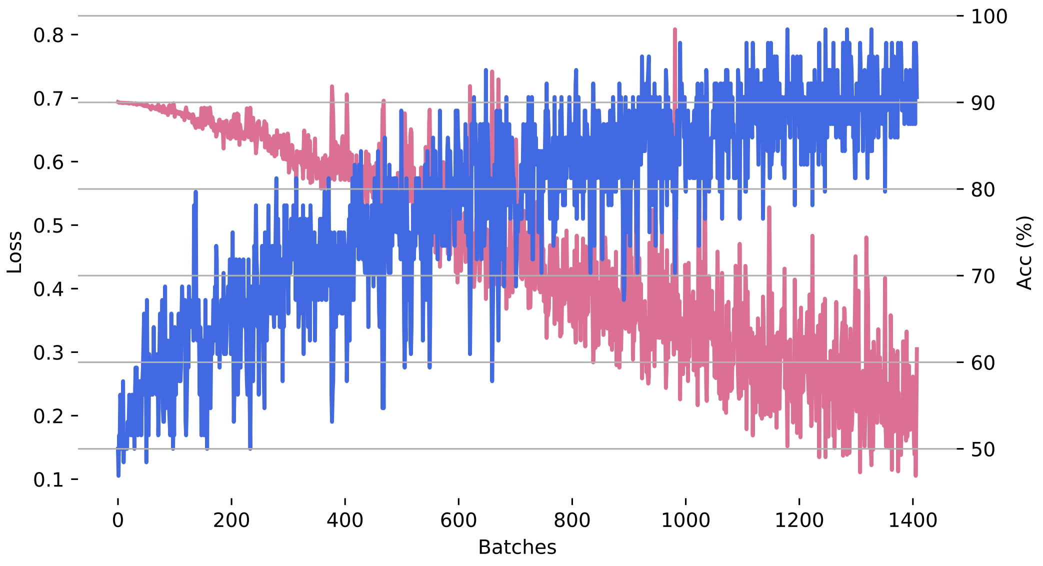
$$\begin{aligned} \mathbf{V_{\mathcal{L}}}^{(t+1)} &\leftarrow \phi_{\mathcal{L}}(\mathbf{V_{\mathcal{L}}}^{(t)}, \mathbf{M}_{\mathcal{CL}} \times \mu^{\mathcal{C}\to\mathcal{L}}(\mathbf{V_{\mathcal{C}}}^{(t)}), \mathbf{M}_{\mathcal{LL}} \times \mathbf{V_{\mathcal{L}}}^{(t)}) \\ \mathbf{V_{\mathcal{C}}}^{(t+1)} &\leftarrow \phi_{\mathcal{C}}(\mathbf{V_{\mathcal{C}}}^{(t)}, \mathbf{M}_{\mathcal{CL}}{}^{\intercal} \times \mu^{\mathcal{L}\to\mathcal{C}}(\mathbf{V_{\mathcal{L}}}^{(t)})) \end{aligned} \tag{4.9}$$

Equations 4.9 can be interpreted in the following way:

1. The literal update function $\phi_{\mathcal{L}}$ updates literal embeddings given a tensor of messages $\mu^{\mathcal{C}\to\mathcal{L}}(\mathbf{V}_{\mathcal{C}})$ computed from clause embeddings by the message computing function $\mu^{\mathcal{C}\to\mathcal{L}}$, which is multiplied by the adjacency matrix $\mathbf{M}_{\mathcal{CL}}$. From each literal's perspective, this multiplication has the effect of "masking" messages sent by clauses for which it does not pertain. The second argument of the update function is a tensor of (unaltered) literal embeddings, which is multiplied by the adjacency matrix $\mathbf{M}_{\mathcal{LL}}$. From each literal's perspective, this multiplication has the effect of "masking" all $2n-2$ literal embeddings which do not correspond to its negated variant.
2. The clause update function $\phi_{\mathcal{C}}$ updates literal embeddings given a tensor of messages $\mu^{\mathcal{L}\to\mathcal{C}}(\mathbf{V}_{\mathcal{L}})$ computed from clause embeddings by the message computing function $\mu^{\mathcal{L}\to\mathcal{C}}$, which is multiplied by the (transposed) adjacency matrix $\mathbf{M}_{\mathcal{CL}}{}^{\intercal}$. From each clause's perspective, this multiplication has the effect of "masking" messages sent by literals it does not contain.

The authors decided to train the model on pairs of complementary SAT / UNSAT instances, which are engineered to differ only by the polarity of a single literal in a single clause. The rationale behind this decision is that it should speed up the training process, as it forces the model to distinguish even the most similar instances with respect to their satisfiability. Instance pairs are generated by iteratively adding clauses to an initially empty CNF formula until it becomes unsatisfiable. At this point, the polarity of a single literal in the most recently added clause is flipped, rendering the instance satisfiable once again. Both instances (before / after flipping) are added to the training dataset, which naturally enforces a 50/50 distribution among positive and negative examples for the corresponding binary classification task. Upon training, the model is able to achieve $\approx 85\%$ test accuracy on instances it had never seen before (Figure 4.17).

Figure 4.17: Loss and accuracy curves of the NeuroSAT model over 1400 gradient descent operations on batches of 32 CNF instances with sizes $n \sim \mathcal{U}(20, 40)$. Results reproduced by the author in Tensorflow according to the description of the model by the original authors in (SELSAM et al., 2018).



4.6 Graph Networks and Typed Graph Networks

In Section 4.5.1, we saw how some problems may require that their instances be modelled as graphs where nodes are partitioned into more than one "type". The example in question, which draws from the work of (SELSAM et al., 2018), is boolean satisfiability, where CNF formulas are represented as a bipartite graph between literals and clauses. In that context, both literals and clauses have their own separate update function ($\phi_{\mathcal{L}}$ and $\phi_{\mathcal{C}}$ respectively), and the model also incorporates "message-computing functions" between literals and clauses and between clauses and literals ($\mu^{\mathcal{L}\to\mathcal{C}}$ and $\mu^{\mathcal{C}\to\mathcal{L}}$ respectively). The formalization of GRNs in Algorithm 2, however, does not support this. In this section, we will show how the previous definition of GNNs can be extended to allow for the definition of models such as NeuroSAT. Additionally, we will review the formalization recently proposed by (BATTAGLIA et al., 2018), explaining how it fails to generalize Graph Neural Networks to allow the definition of relatively simple models while augmenting it with unnecessary complexity.

In their recent paper, which is part literature review, part formalization effort and part position paper, (BATTAGLIA et al., 2018) propose the "Graph Network" model, which according to the authors is intended to "generalize and extend various approaches for neural networks which operate on graphs". The authors' concept was to augment the simplest GNN formalization (defined on Algorithm 2) with edge embeddings and a "global attribute" embedding. In their formalization, embeddings are assigned to each vertex, to each edge and a single embedding is assigned to entire graph. Over the course of many message-passing timesteps, vertices communicate with edges of which they are endpoints and all edges plus all vertices communicate with the graph, in a process which culminates in the refinement of embeddings for all three categories (vertices / edges / graph). Their effort makes sense: assigning embeddings to edges corresponds to promoting them to "first-class citizens" in the GNN formalization, which enables one (for example) to train problems on graphs with labelled edges, as their embeddings can now be initialized according to the corresponding labels. This feature also gives for free a projection of edges into a hyperdimensional space, which may be useful for varied reasons. Having a "graph embedding" is also useful, as it may be used to refine some global property of the problem instance which is intentended to be learned.

However, there are contexts in which is acceptable not to assign embeddings to edges, and even more contexts in which it does not make sense to assign an embedding to the entire graph. In these scenarios, the authors' formalization forces one to define models with "appendages" which either harm the model's performance, confuse the programmer, or both. Moreover, it is undesirable from the standpoint of parsimony to define a formalization with such unnecessary complexities. This is not the main problem with the authors' formalization, however. Instead, the most problematic issue is the fact that it is incomplete, as, by focusing on edges, the authors end up removing support for richer relational structures. This brings us back to the original example of NeuroSAT, which illustrates the issue very well. In the context of CNF formulas, clauses are nothing more than sets of literals. They are relational structures which link many nodes at the same time, or, in other words, they correspond to **hyperedges** on an hypergraph (FENG et al., 2019) where literals are equated with nodes. The model proposed by (BATTAGLIA et al., 2018), which supports only regular edges, is bound to model only a special case of NeuroSAT where all clauses contain exactly two literals (2-SAT).

One is not required, however, to explicitly augment the previous model with support for hyperedges. Hyperedges can be simulated in a far more succint and elegant manner by partitioning the graph's nodes into N distinct "types" and as-

Algorithm 3 Graph Network Formalisation (BATTAGLIA et al., 2018) run for t message-passing timesteps.

```
1: procedure GraphNetwork($\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{u})$)
2:     for $t = 1 \ldots t_{max}$ do
3:         for $k = 1 \ldots |\mathcal{E}|$ do
4:             // Compute updated edge embeddings
5:             $\mathbf{e_k}^{(t+1)} \leftarrow \phi_e(\mathbf{e}_k^{(t)}, \mathbf{v}_{r_k}^{(t)}, \mathbf{v}_{s_k}^{(t)}, \mathbf{u}^{(t)})$
6:         for $i = 1 \ldots |\mathcal{V}|$ do
7:             // Aggregate messages sent to node $i$ into $\overline{\mu}$
8:             $\overline{\mu} \leftarrow \rho_{e \to v}(\{\mathbf{e}_k^{(t+1)} \mid r_k = i\})$
9:             // Update vertex embedding
10:            $\mathbf{v}_i^{(t+1)} \leftarrow \phi_v(\mathbf{v}_i^{(t)}, \overline{\mu}, \mathbf{u}^{(t)})$
11:        // Compute updated global embedding
12:        $\mathbf{u}^{(t+1)} \leftarrow \phi_u(\mathbf{u}, \{\mathbf{v}_i^{(t+1)}\}_{i=1 \ldots |\mathcal{V}|}, \{\mathbf{e}_i^{(t+1)}\}_{i=1 \ldots |\mathcal{E}|})$
13:    // Return refined vertex embeddings, edge embeddings and global embedding
14:    return $\{\mathbf{v}_i^{(t_{max})}\}_{i=1 \ldots |\mathcal{V}|}, \{\mathbf{e}_i^{(t_{max})}\}_{i=1 \ldots |\mathcal{E}|}, \mathbf{u}^{(t_{max})}$
```

signing update functions (and eventually message-computing functions) for each of them. This corresponds exactly to what (SELSAM et al., 2018) have done in their paper, by assigning embeddings to hyperedges (clauses). In fact, the authors' insight in (BATTAGLIA et al., 2018) to project edges and the graph to hyperdimensional space in addition to nodes can be understood as empowering these objects with a "node treatment". Effectively, edges are treated as if they were **nodes of a different type**, whose embeddings are updated with an "update" function different from that used to update regular nodes. The same applies for the global attribute, with the notable difference that there is only one "graph embedding" in contrast to possibly many node and edge embeddings. We include here, for reference, the authors' formalization in Algorithm 3.

The crux of the formalization in Algorithm 3 is that node, edge and graph embeddings are logically separated by the definition of three update functions ϕ_v, ϕ_e, ϕ_u. It should be noted that the Graph Network model **fixes** objects into these three types. But in theory, one could envision extensions to the Graph Network model with additional types of objects different from nodes, edges and graph, the only requirement being that an additional update function is defined for each additional type. In fact, (SCARSELLI et al., 2009b) already tackled this issue with the original Graph Neural Network model by assigning to each kind of node its own message-computing function and update function. For didatic purposes, we initially defined a simplified variant of the authors' original model with a single vertex type in Algorithm 1. But now hopefully we have made the point clear that it would be useful to have not one but k embeddings for global attributes, with corresponding update functions $\phi_{u_1}, \phi_{u_2} \dots \phi_{u_k}$. Thus posed, we wish to emphasize the original authors' contribution by proposing an updated terminology focused on the possibility of having several types of objects, which is in line with more recent work (GILMER et al., 2017) and improves understandability, removing unnecessary complexities from the original model such as the support for multiple edge types – since the concept of node types immediately adds the same expressiveness.

Algorithm 4 Typed Graph Network Model

```
1: // The input for a TGN is a graph whose vertices are partitioned into $N$ **types**
2: **procedure** TGN($\mathcal{G} = (\mathcal{V} = \bigcup_{i=1}^{N} \mathcal{V}_i, \mathcal{E})$)
3:     // Compute an adjacency matrix between types $\tau_i$ and $\tau_j$
4:     **for** $i = 1 \dots n,\ j = 1 \dots N$ **do**
5:         $\mathbf{M}_{ij}[a,b] = \mathbb{1}\{(v_a, v_b) \in \mathcal{E} \mid v_a \in \mathcal{V}_i, v_b \in \mathcal{V}_j\}$
6:     **for** $i = 1 \dots n$ **do**
7:         Init vertex embeddings $\mathbf{X}_i^{(1)}[a] \in \tau_i \mid \forall v_a \in \mathcal{V}_i$
8:     // Run for $T$ message-passing iterations
9:     **for** $t = 1 \dots T$ **do**
10:        // For each receiving type $\tau_i$
11:        **for** $i = 1 \dots N$ **do**
12:            // For each message sent from type $\tau_j$
13:            **for** $\mu_k \in \mathcal{M} \mid \mu_k : \tau_j \to \tau_i$ **do**
14:                // Accumulate messages sent to vertices of type $\tau_i$ by vert. of type $\tau_j$
15:                $\overline{\mu}_k \leftarrow \mathbf{M}_{ij} \times \mu(\mathbf{X}_j^{(t)})$
16:            // Compute updated embeddings for type #$i$
17:            $\mathbf{X}_i^{(t+1)} \leftarrow \phi_i(\mathbf{X}_i^{(t)}, \{\overline{\mu}_k \mid \mu_k \in \mathcal{M}, \mu_k : \tau_i \to \tau_j\})$
18:    // Return set of refined embeddings over $T$ iterations
19:    **return** $\{\mathbf{X}_i^{(T)} \mid i = 1 \dots n\}$
```

Our proposal is described in Algorithm 5. Note that the set of vertices is partitioned into N "types", corresponding to disjoint subsets: $\mathcal{V} = \bigcup_{i=1}^{N} \mathcal{V}_i$. We define an adjacency matrix for each pair of types i, j (N^2 total) in the straightforward way, by reading from the edges $e \in \mathcal{E}$ in the input graph. Vertex embeddings can be initialized in any way imaginable: they can all be initialized with the same value, embeddings from each type can be initialized with a different value, and embeddings can even be initialized independently for each individual vertex, perhaps as a function of the labels assigned to each node in the context of a labelled graph. Then, for the message-passing loop itself, note that in order to update embeddings for each vertex type we accumulate messages sent from all other types which send communicate with it. So, for example, if type #1 receives messages from types #2 and #5, we will reduce all messages received from types #2 and #5 separately into two tensors $\overline{\mu}_2$ and $\overline{\mu}_5$, and these two tensors will correspond to two distinct arguments for the update function ϕ_1 as in $\phi_1(\mathbf{V_i}^{(t)}\{\overline{\mu}_2, \overline{\mu}_5\})$. Finally, after all embeddings from all types are refined over the programmed number of iterations, the algorithm returns them. Formally, a Typed Graph Network, as we named it, can also be defined according to Definition 4.6 below.

[Typed Graph Network] A Typed Graph Network with N types is described by

1. A set of **embedding sizes** $n_1, n_2 \dots n_N$
2. A set of **types** $\mathcal{T} = \{\tau_i \in \mathbb{R}^{n_i} \mid i = 1 \dots N\}$.
3. A set of K **message** functions
   $\mathcal{M} = \{\mu : \tau_1 \to \tau_2 \mid \tau_1, \tau_2 \in \mathcal{T}\}$
   OBS. Note that for each type combination (τ_1, τ_2) one can define many message functions.
4. And a set of **update** functions
   $\{\phi_i : \mathbb{R}^{n_i + D(i)} \to \mathbb{R}^{n_i} \mid i = 1 \dots N\}$, where $D(i) = \sum\limits_{\forall \mu : \tau_j \to \tau_i \in \mathcal{M}} n_i$

Where the message functions $\mu_{i \to j}$ and the update functions ϕ_i are the sole trainable components of the model.

4.6.1 A Note on the Number of Message-Computing Functions

At this point, the reader may possibly have asked herself whether it is really necessary to allow for the existence of more than one message-computing functions between the same combination of types (τ_i, τ_j). Would it not suffice to instantiate a message-computing function between each such combination? And what is the interpretation of having two message-computing functions, say μ_1 and μ_2, between the same two types?

We can answer this with an example: consider an application in which you have graphs with directed edges and you want to differentiate between a message sent from an **incoming** vertex and a message sent from an **outcoming** vertex. In this context, it would make perfect sense to instantiate two message-computing functions μ_1, μ_2, and have each learn the corresponding task.

4.6.2 Typed Graph Networks with Customizable Aggregation

In Algorithm 5 and Definition 4.6, there are two levels of aggregations performed on the messages exchanged between vertices. For each combination of types (τ_i, τ_j), any number of message-computing functions can be defined. Each of them will be used to compute a message for every single vertex of type τ_j, and these messages will be sent to adjacent vertices of type τ_i. Upon receiving a set of messages, a vertex of type τ_i must aggregate them somehow into a single tensor. In our formalization, the model is forced to implement this aggregation as a element-wise sum (Line 10 of Algorithm 5). But there is a second level of aggregation, because each vertex of type τ_i can receive multiple sets of messages, each corresponding to a particular message-computing function. Even though these sets are aggregated each one into a tensor, we still have a set of tensors at the second level, which must be aggregated somehow. In our formalization, the model is forced to implement this aggregation by concatenating all tensors (Line 11 of Algorithm 5).

However, one could in principle generalize TGNs to leave the aggregation functions at the first and second levels subject to the decision of the DL engineer. To do so, it suffices to enhance our formalization with two aggregator operators ρ_1 and ρ_2. The updated algorithm is the following:

4.7 Typed Graph Networks Python / Tensorflow Library

Part of the contribution of this thesis is the development of a Python / Tensorflow Library intended for significantly easing the implementation and prototyping of TGN models. The TGN library was developed together with fellow graduate student Pedro Avelar, and was used for the development of all models proposed in or reproduced in this dissertation. Our library allows one to specify a TGN succinctly, at a description level similar to that of the formalisation in Algorithm 5. This description is compiled into a set of Parameters that, when the model is called upon a input, constructs a computation graph accordingly, thus yielding a module whose inputs and outputs can be connected to any other operations at the desired point in

Algorithm 5 Generalized Typed Graph Network Model

```
 1: // The input for a Generalized TGN is a graph whose vertices are partitioned into $N$ **types**
 2: procedure GENERALIZED TGN($\mathcal{G} = (\mathcal{V} = \bigcup_{i=1}^{N} \mathcal{V}_i, \mathcal{E})$)
 3:     // Compute an adjacency matrix between types $\tau_i$ and $\tau_j$
 4:     for $i = 1 \dots n,\ j = 1 \dots N$ do
 5:         $\mathbf{M}_{ij}[a,b] = \mathbb{1}\{(v_a, v_b) \in \mathcal{E} \mid v_a \in \mathcal{V}_i, v_b \in \mathcal{V}_j\}$
 6:     for $i = 1 \dots n$ do
 7:         Init vertex embeddings $\mathbf{V_i}^{(1)}[a] \in \tau_i \mid \forall v_a \in \mathcal{V}_i$
 8:     // Run for $T$ message-passing iterations
 9:     for $t = 1 \dots T$ do
10:         // For each receiving type $\tau_i$
11:         for $i = 1 \dots N$ do
12:             // For each message sent from type $\tau_j$
13:             for $\mu_k \in \mathcal{M} \mid \mu_k : \tau_j \to \tau_i$ do
14:                 // Accumulate messages sent to vertices of type $\tau_i$ by vert. of type $\tau_j$
15:                 $\overline{\mu}_k \leftarrow \rho_1(\mu(\mathbf{V_j}^{(t)}))$
16:             // Compute updated embeddings for type $\#i$
17:             $\mathbf{X}_i^{(t+1)} \leftarrow \phi_i(\mathbf{X}_i^{(t)}, \rho_2(\{\overline{\mu}_k \mid \mu_k \in \mathcal{M}, \mu_k : \tau_i \to \tau_j\}))$
18:     // Return set of refined embeddings over $T$ iterations
19:     return $\{\mathbf{V_i}^{(T)} \mid i = 1 \dots n\}$
```

the trainable model's pipeline. It is important to note here that our implementation, due to practical reasons, implements the update function exclusively as a LSTM-like RNN, which operates on both the last hidden state and output.

The TGN builder identifies each vertex type by a string, for example 'V' for vertex vertices and 'E' for edge vertices. Adjacency matrices are also identified by strings, for example 'EV' for an edge-to-vertex adjacency matrix $\in \mathbb{B}^{|\mathcal{E}|\times|\mathcal{V}|}$, as well as message functions, for example 'V_cast_E' for a message function $\tau_{\mathcal{V}\to\mathcal{E}} : \mathbb{R}^{n_\mathcal{V}} \to \mathbb{R}^{n_\mathcal{E}}$ mapping vertex embeddings to edge embeddings. An update function is automatically instantiated for each type, but each of its arguments must be specified by an adjacency matrix, a message function and the type of the sender vertices.

Concretely, the TGN builder receives 4 arguments:

1. A Python dictionary mapping type names to embedding sizes, such as {'V': d_v, 'E': d_e}. Equivalent to $\tau_\mathcal{V} = \mathbb{R}^{d_v}, \tau_\mathcal{E} = \mathbb{R}^{d_e}$.
2. A Python dictionary mapping matrix names to 2-uples of type names, such as {'EV': ('E','V')}. Equivalent to: $\mathbf{EV} \in \mathbb{R}^{|\mathcal{E}|\times|\mathcal{V}|}$.
3. A Python dictionary mapping message function names to 2-uples of type names, such as

{’V_cast_E: (’V’,’E’), ’E_cast_V’: (’E’,’V’)}. Equivalent to: $\mu_{\mathcal{V}\to\mathcal{E}} : \tau_{\mathcal{V}} \to \tau_{\mathcal{E}}, \mu_{\mathcal{E}\to\mathcal{V}} : \tau_{\mathcal{E}} \to \tau_{\mathcal{V}}$.

4. A Python dictionary mapping type names to lists of Python dictionaries each specifying an aggregation of messages, such as {’E’: [{’mat’: ’EV’, ’msg’: ’V_cast_E’, ’var’: ’V’}]}. Equivalent to:
   $\mathbf{E}^{(t+1)} \leftarrow \phi_{\mathcal{E}}(\mathbf{E}^{(t)}, \mathbf{M}_{\mathcal{E}\mathcal{V}} \times \mu_{\mathcal{V}\to\mathcal{E}}(\mathbf{V}^{(t)}))$. Note that this argument corresponds to a **list** of dictionaries. This is so because the update function can receive multiple arguments.

Then, the TGN itself is called with the desired inputs, effectively coupling it with the rest of the model’s pipeline and producing the output of the last states of each type of vertex. This also permits, in case of the Tensorflow implementation, to set the builder to a specific variable scope and choose whether to make use of parameter sharing or not. When coupling the TGN with the rest of the model, it receives 3 dictionaries and 1 integer, with one of the dictionaries being optional, specifying the adjacency matrices, the initial embeddings and the initial hidden-states of the RNNs (being that the hidden state is assumed to be a zero tensor, if missing) as well as how many time-steps of computation should be performed.

Concretely, upon calling the TGN and coupling it with the rest of the pipeline, it receives:

1. A Python dictionary mapping matrix names to adjacency matrices between two vertex types, such as {’EV’: $M_{|E|\times|V|}$}.
2. A Python dictionary mapping type names to the initial embeddings of each vertex of that type, such as {’V’: V, ’E’: E}, with $V \in \mathbb{R}^{|V|\times d_v}$ and $E \in \mathbb{R}^{|E|\times d_e}$ being the tensors containing the initial embedding for each of the vertices in the Graph.
3. A single integer t_{max}, defining how many timesteps of message-passing are to be performed.
4. A Python dictionary mapping type names to the initial hidden states embeddings of each vertex of that type, such as {’V’: $\mathbf{V}_0$, ’E’: $\mathbf{E}_0$}, with $\mathbf{V}_0 \in \mathbb{R}^{|\mathcal{V}|\times d_v}$ and $\mathbf{E}_0 \in \mathbb{R}^{|\mathcal{E}|\times d_e}$ being the tensors containing the initial hidden-state embedding for each of the vertices in the Graph.

Another, implementation-specific, note that can be raised is also that the batching of different graphs can be done by simply concatenating different graphs, without making any adjacencies between one another. In this way, information from a node in a graph will never reach information in a node in another graph, and thus one can make even better use of parallelism to perform faster computation on the inputs.

On the following subsections, we will give some examples of how to implement some models using the TGN formalisation and our library, and specifically in Subsections 4.7.2 and 4.7.5 we explain how to implement models in which the TGN formalization works more naturally than others explained in the literature.

4.7.1 Technical Overview

The core of our TGN library is the Python file "tgn.py", in addition to Python file "mlp.py" which acts as a helper to build MLP blocks. Both files can be found in Appendix A. "tgn.py" corresponds to a single Python class, `TGN`. Overall, `TGN` is composed of four methods: `__init__()`, `check_model()`, `_init_parameters()`, `__call__()` and `check_run()`. The paragraphs which will follow explain them in detail.

The `__init__()` method is assigned with processing the input dictionary and calling `check_model()` to check the model for consistency and `_init_parameters()` to initialize the parameters of the model's trainable LSTM and MLP components.

The `_check_model()` method is assigned with checking the user-defined model for consistency. This includes checking for vertex types in the TGN model which are not associated to any update rule and update rules, message-computing functions or adjacency matrices referring to undeclared vertex types.

The `_init_parameters()` method is assigned with initializing the parameters of the trainable components of the model, corresponding to message-computing MLPs and LSTMS implementing vertex update functions.

The `__call__()` method is the core of the TGN library. It is assigned with calling `check_run()` to perform a runtime check of the model, after which it essentially runs the algorithm described in Algorithm 5. This corresponds to running a `tf.while_loop()` whose stopping condition relates to the maximum number of timesteps and whose body performs one iteration of embedding refinement through message-passing. Concretely, in Tensorflow parlance, our code implements message-passing by feeding a tensor of vertex embeddings into the message-computing MLP to yield a rank 2 tensor of messages. The columns of this tensor are then masked by a binary mask which erases the contribution, to each vertex, of messages sent by non-neighbors. This is done via matrix multiplication as described in Algorithm 5. Finally, the aggregated messages for each vertex are organized into a tensor which is fed to the layer-norm LSTMs implementing vertex updates, yielding a tensor with the updated embeddings for this iteration.

The `check_run()` method performs a runtime check on the model. This is in contrast with `check_model()` which performs a check at compilation time. This process is called upon whenever the model is actually run on a Tensorflow session, and is assigned with checking for inconsistencies between the sizes of the inputs fed to the model at runtime (initial embeddings and adjacency matrices).

By default, as defined in “mlp.py”, all MLPs are initialized with Xavier initialization (GLOROT; BENGIO, 2010) for weights and initializes biases with zeros. Also by default, the activation functions for both LSTMs and MLPs are ReLUs. Batching can be naturally implemented in the TGN library by performing the disjoint union of a set of graphs to obtain a single disconnected graph. Message-passing will carry out as expected and vertex embeddings will be refined normally. The fact that subgraphs are disconnected means that no information will bleed out from one instance to another, and at the end of the process the refined vertex embeddings for each instance can be aggregated separately to yield outputs for each instance.

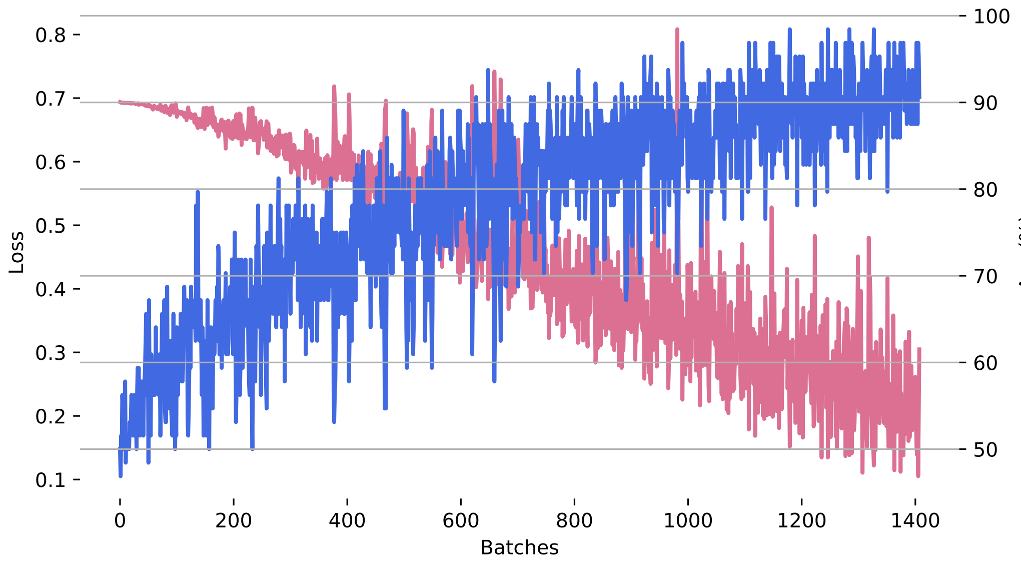
4.7.2 NeuroSAT

(SELSAM et al., 2018) have shown that neural models in the GNN family can learn to solve the problem of boolean satisfiability with up to 85% accuracy upon being fed with complementary (SAT/UNSAT) instances which differ by a single literal’s polarity on a single clause. The insight behind the authors’ architecture is to assign embeddings both to literals (i.e. $x_1, \neg x_5$, etc.) and clauses (i.e. $(x_2 \vee \neg x_{10} \vee \neg x_3)$). Literals send messages to clauses to which they pertain, and clauses send messages to literals they contain. Additionally, literals send messages to their negated variants (i.e. x_1 sends messages to $\neg x_1$ and vice-versa). This can be formalised by the following update equations:

$$\begin{aligned} \mathbf{L}^{(t+1)} &\leftarrow \phi_{\mathcal{L}}(\mathbf{L}^{(t)}, \mathbf{M}_{\mathcal{LC}}{}^{\intercal} \times \mu_{\mathcal{C}\to\mathcal{L}}(\mathbf{C}^{(t)}), \mathbf{M}_{\mathcal{LL}} \times \mathbf{L}^{(t)})) \\ \mathbf{C}^{(t+1)} &\leftarrow \phi_{\mathcal{C}}(\mathbf{C}^{(t)}, \mathbf{M}_{\mathcal{LC}} \times \mu_{\mathcal{L}\to\mathcal{C}}(\mathbf{L}^{(t)})) \end{aligned} \tag{4.10}$$

Where $\mathcal{L} = \{x_1, \neg x_1 \ldots x_i, \neg x_i \ldots x_N, \neg x_N\}$ is the set of all $2N$ literals, $\mathcal{C} \in \mathcal{P}(\mathcal{L})^M$ is the set of all M clauses and a CNF formula can be described by an adjacency matrix $\mathbf{M}_{\mathcal{LC}} \in \mathbb{B}^{2N\times M}$ between literals and clauses as $\mathbf{M}_{\mathcal{L}\times\mathcal{C}} = \mathbb{1}\{(l,c) \in \mathcal{L}\times\mathcal{C} \mid l \in c\}$. To connect literals with their negated variants, it suffices to define an adjacency matrix $\mathbf{M}_{\mathcal{LL}} \in \mathbb{B}^{2L\times 2L}$ between literals and literals as $M_{\mathcal{LL}} = \mathbb{1}\{(l_1,l_2) \in \mathcal{L}^2 \mid l_1 = \neg l_2\}$.

Figure 4.18: Results obtained with the NeuroSAT model proposed by (SELSAM et al., 2018) implemented in the Typed Graph Networks library (see Code Listing 4.1).



```
tgn = TGN(
  {
    # $\tau_{\mathcal{L}} = \mathbb{R}^{d_l}$
    'L': dl,
    # $\tau_{\mathcal{C}} = \mathbb{R}^{d_c}$
    'C': dc
  },
  {
    # $\mathbf{M}_{\mathcal{LL}} \in \mathbb{B}^{2N \times 2N}$
    'LL': ('L','C')
    # $\mathbf{M}_{\mathcal{LC}} \in \mathbb{B}^{2N \times M}$
    'LC': ('L','C')
  },
  {
    # $\mu_{\mathcal{L}\to\mathcal{C}} : \tau_{\mathcal{L}} \to \tau_{\mathcal{C}}$
    'L_msg_C': ('L','C'),
    # $\mu_{\mathcal{C}\to\mathcal{L}} : \tau_{\mathcal{C}} \to \tau_{\mathcal{L}}$
    'C_msg_L': ('C','L')
  },
  {
    # $\mathbf{L}^{(t+1)}, \mathbf{L}_h^{(t+1)} \leftarrow \phi_{\mathcal{L}}(\mathbf{L}_h^{(t)},$
    #          $\mathbf{M}_{\mathcal{LC}}^{\intercal} \times \mu_{\mathcal{L}\to\mathcal{C}}(\mathbf{C}^{(t)}), \mathbf{M}_{\mathcal{LL}} \times \mathbf{L}^{(t)})$
    'L': [
      {
        'mat': 'LC',
        'msg': 'L_msg_V',
        'transpose?': True,
        'var': 'L'
      },
      {
        'mat': 'LL',
        'var': 'L'
      }
    ],
    # $\mathbf{C}^{(t+1)}, \mathbf{C}_h^{(t+1)} \leftarrow \phi_{\mathcal{C}}(\mathbf{C}_h^{(t)}, \mathbf{M}_{\mathcal{LC}} \times \mu_{\mathcal{L}\to\mathcal{C}}(\mathbf{L}^{(t)}))$
    'C': [
      {
        'mat': 'LC',
        'msg': 'L_msg_C',
        'var': 'L'
      }
    ]
  }
)
```

Listing 4.1: TGN kernel of an end-to-end differentiable model to solve the boolean satisfiability problem (SAT), implemented with our Python library. Lines of code are commented with the corresponding equations in the proposed formalisation for TGNs in Algorithm 5.

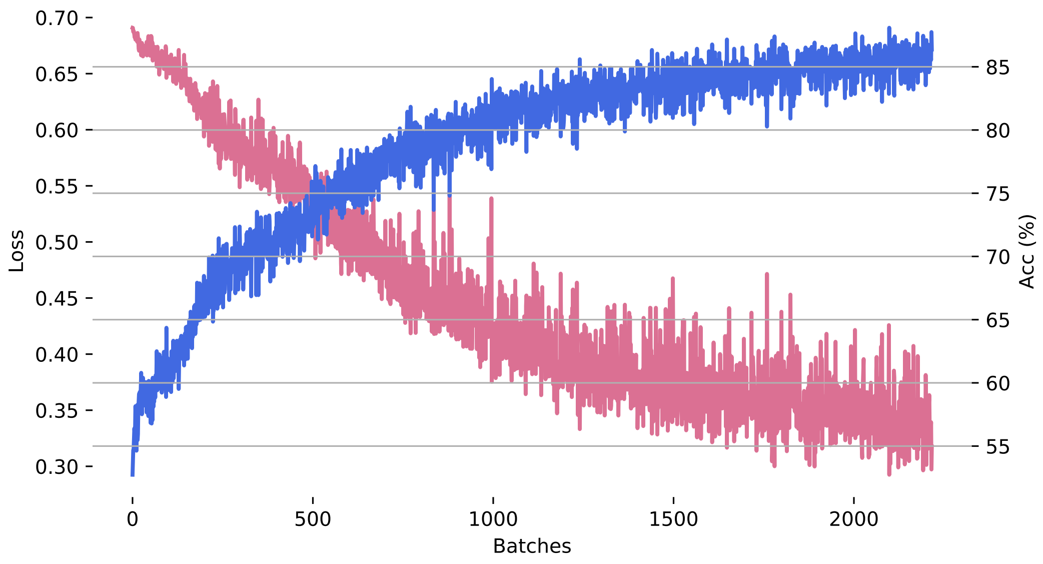
4.7.3 Solving the decision TSP

The TGN formalisation can be instantiated into the kernel of a end-to-end differentiable model to solve the decision version of the Traveling Salesperson Problem (TSP), although the corresponding block could conceivably be used to learn any graph problem with weighted edges (given an appropriate loss function). In (PRATES et al., 2018) we were able to train a GNN model with up to 80% test accuracy on the decision problem (i.e. "does graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ admit a solution with cost no greater than C?") by feeding it with pairs of complementary decision instances $X^- = (\mathcal{G}, 0.98C^*)$ and $X^+ = (\mathcal{G}, 1.02C^*)$ where $\mathcal{G}$ is a random euclidean graph with optimal TSP cost C^*. The ground truth answers for the decision problem are **NO** and **YES** respectively for X^- and X^+, forcing the model to learn to solve the problem within a 2% relative deviation from the optimal cost.

Because edges are labeled with numerical information (i.e. their weights), it is convenient to assign embeddings to edges and nodes alike, and initialise edge embeddings with their corresponding weights. Then a TGN model can be instantiated in which nodes send messages to their incoming and outcoming edges, and edges send messages to their source and target nodes over many iterations of message passing. This model can be instantiated into the proposed TGN formalisation by defining an adjacency matrix $\mathbf{M}_{\mathcal{E}\mathcal{V}} \in \mathbb{B}^{|\mathcal{E}|\times|\mathcal{V}|}$ between edges and vertices (i.e. mapping each edge to its source and target vertices) and having the update step for vertex and edge embeddings be defined by:

$$\begin{aligned} \mathbf{V}^{(t+1)} &\leftarrow \phi_{\mathcal{V}}(\mathbf{V}^{(t)}, \mathbf{M}_{\mathcal{E}\mathcal{V}}{}^{\intercal} \times \mu_{\mathcal{E}\rightarrow\mathcal{V}}(\mathbf{E}^{(t)})) \\ \mathbf{E}^{(t+1)} &\leftarrow \phi_{\mathcal{E}}(\mathbf{E}^{(t)}, \mathbf{M}_{\mathcal{E}\mathcal{V}} \times \mu_{\mathcal{V}\rightarrow\mathcal{E}}(\mathbf{V}^{(t)})) \end{aligned} \tag{4.11}$$

Figure 4.19: Results obtained with the TSP GNN model proposed (PRATES et al., 2018) implemented in the Typed Graph Networks library (see Code Listing 4.2).



```
tgn = TGN(
  {
    # $\tau_{\mathcal{V}} = \mathbb{R}^{d_v}$
    'V': dv,
    # $\tau_{\mathcal{E}} = \mathbb{R}^{d_e}$
    'E': de
  },
  {
    # $\mathbf{M}_{\mathcal{E}\mathcal{V}} \in \mathbb{B}^{|\mathcal{E}|\times|\mathcal{V}|}$
    'EV': ('E','V')
  },
  {
    # $\tau_{\mathcal{V}\to\mathcal{E}} : \tau_{\mathcal{V}} \to \tau_{\mathcal{E}}$
    'V_msg_E': ('V','E'),
    # $\tau_{\mathcal{E}\to\mathcal{V}} : \tau_{\mathcal{E}} \to \tau_{\mathcal{V}}$
    'E_msg_V': ('E','V')
  },
  {
    # $\mathbf{V}^{(t+1)}, \mathbf{V}_h^{(t+1)} \leftarrow \phi_{\mathcal{V}}(\mathbf{V}_h^{(t)}, \mathbf{M}_{\mathcal{E}\mathcal{V}}{}^{\intercal} \times \mu_{\mathcal{E}\to\mathcal{V}}(\mathbf{E}^{(t)}))$
    'V': [
      {
        'mat': 'EV',
        'msg': 'E_msg_V',
        'transpose?': True,
        'var': 'E'
      }
    ],
    # $\mathbf{E}^{(t+1)}, \mathbf{E}_h^{(t+1)} \leftarrow \phi_{\mathcal{E}}(\mathbf{E}_h^{(t)}, \mathbf{M}_{\mathcal{E}\mathcal{V}} \times \mu_{\mathcal{V}\to\mathcal{E}}(\mathbf{V}^{(t)}))$
    'E': [
      {
        'mat': 'EV',
        'msg': 'V_msg_E',
        'var': 'V'
      }
    ]
  }
)
```

Listing 4.2: TGN kernel of an end-to-end differentiable model to solve the decision version of the Traveling Salesperson Problem, implemented with our Python library. Lines of code are commented with the corresponding equations in the proposed formalisation for TGNs in Algorithm 5.

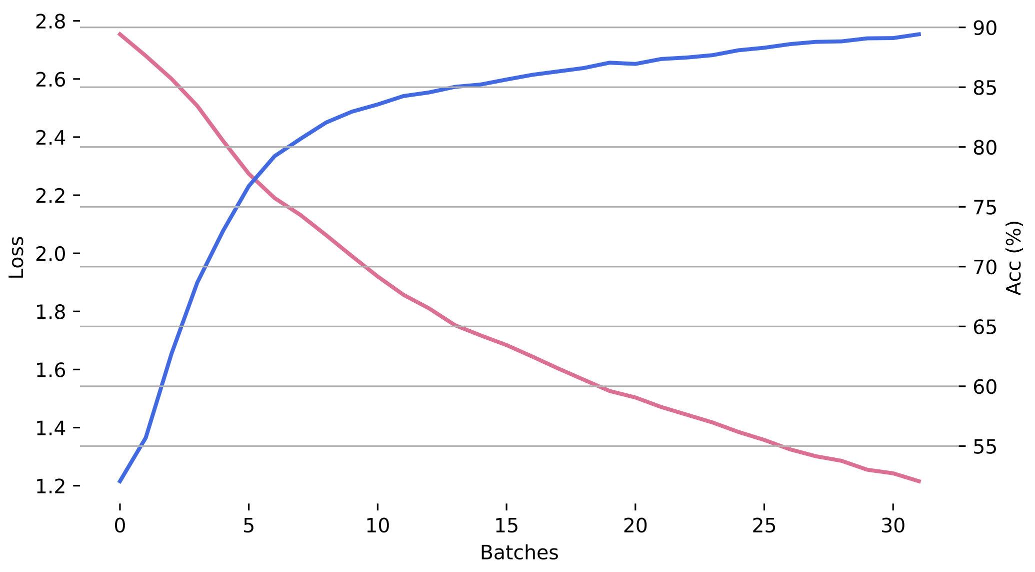
4.7.4 Ranking graph vertices by their centralities

Recently in (AVELAR et al., 2018) we trained a GNN model to predict centrality comparison on graphs (i.e. "given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a centrality measure $c : \mathcal{V} \rightarrow \mathbb{R}$ and two vertices $v_1, v_2 \in \mathcal{V}$, does $c(v_1) < c(v_2)$ hold?"). The authors were able to achieve 89% test accuracy on a dataset composed by random instances with up to 128 vertices. Additionally, we were able to distill the same refined vertex embeddings into multiple centrality predictions, each corresponding to a different centrality measure, by training multiple MLPs fed with these embeddings at the end of the computation pipeline (multitask learning). This scenario can exemplify one of the simplest TGN models possible, corresponding to an iteration of the original Graph Neural Network model (SCARSELLI et al., 2009b) with a single vertex type.

In our model we differentiate between a message received from a source and from a target vertex. The update function can be described by the following equation:

$$\mathbf{V}^{(t+1)} \leftarrow \phi_{\mathcal{V}}(\mathbf{V}^{(t)}, \mathbf{M} \times {\mu^{S}}_{\mathcal{V}\rightarrow\mathcal{V}}(\mathbf{V}^{(t)}), \mathbf{M}^{\intercal} \times {\mu^{T}}_{\mathcal{V}\rightarrow\mathcal{V}}(\mathbf{V}^{(t)})) \tag{4.12}$$

Figure 4.20: Results obtained with the graph centrality GNN model proposed by (AVELAR et al., 2018) implemented with the Typed Graph Networks library (see Code Listing 4.3).



```
tgn = GraphNN(
  {
    # $\tau_{\mathcal{V}} \in \mathbb{R}^d$
    'V': d
  },
  {
    # $\mathbf{M} \in \mathbb{B}^{|\mathcal{V}|\times|\mathcal{V}|}$
    'M': ('V','V')
  },
  {
    # $\mu^S_{\mathcal{V}\to\mathcal{C}} : \tau_{\mathcal{V}} \to \tau_{\mathcal{V}}$
    'Vsource': ('V','V'),
    # $\mu^T_{\mathcal{V}\to\mathcal{C}} : \tau_{\mathcal{V}} \to \tau_{\mathcal{V}}$
    'Vtarget': ('V','V')
  },
  {
    # $\mathbf{V}^{(t+1)}, \mathbf{V}_h^{(t+1)} \leftarrow \phi_{\mathcal{V}}(\mathbf{V}_h^{(t)}, \mathbf{M} \times \mu^S{}_{\mathcal{V}\to\mathcal{V}}(\mathbf{V}^{(t)}),$
                       $\mathbf{M}^\intercal \times \mu^T{}_{\mathcal{C}\to\mathcal{V}}(\mathbf{V}^{(t)}))$
    'V': [
      {
        'mat': 'M',
        'var': 'V',
        'msg': 'Vsource'
      },
      {
        'mat': 'M',
        'transpose?': True,
        'var': 'V',
        'msg': 'Vtarget'
      }
    ]
  },
)
```

Listing 4.3: TGN kernel of an end-to-end differentiable model to predict vertices ordering according to a given centrality, note that the resulting vertices embeddings should further be fed to c MLPs in order to compute vertex-to-vertex comparisons

4.7.5 Solving the Vertex k-Colorability Problem

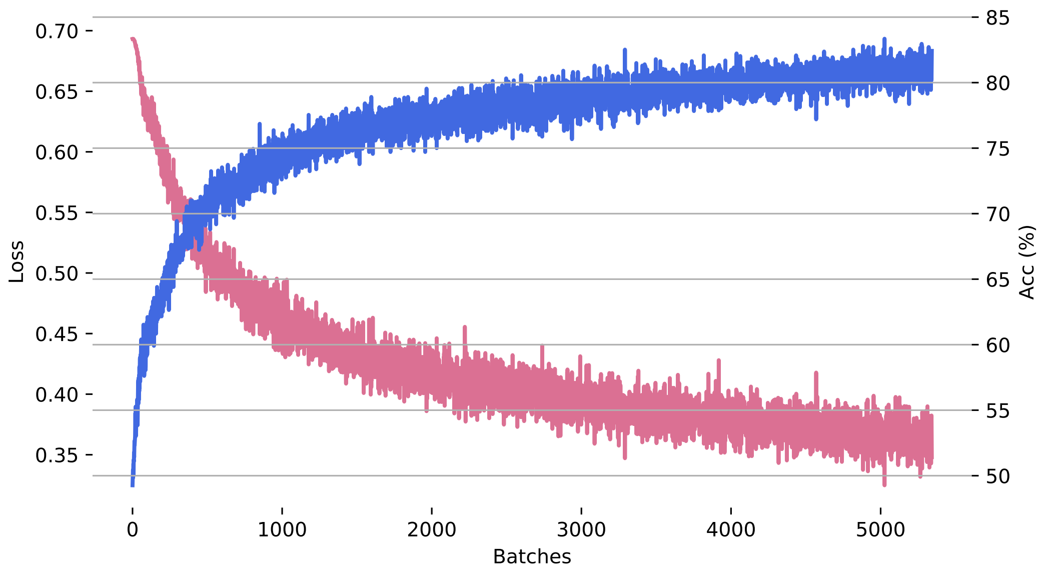
The Graph Network model does not allow multiple embeddings corresponding to multiple global attributes for a graph. However this feature can be useful to solve a wide range of graph problems. We exemplify this issue here with the vertex coloring problem – in which, given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and an integer $k \in \mathbb{N}$, we must decide whether all the vertices $v\mathcal{V}$ can be partitioned into k disjoint subsets (each representing a color) such that no two vertices v_i, v_j from the same subset are connected by an edge $(i,j) \in \mathcal{E}$. In this context, it is useful to assign k embeddings, one for each color, in addition to vertices embeddings. Color embeddings correspond to global attributes, as they should communicate with all vertex embeddings (i.e. each embedding is unconstrained, in principle, regarding which color it can assume).

A TGN-based algorithm to tackle this problem could be defined as follows: initially, in addition to the vertices adjacency matrix $\mathbf{M}_{\mathcal{V}\mathcal{V}}$, we need to instantiate an adjacency matrix between vertices and colors $\mathbf{M}_{\mathcal{V}\mathcal{C}}$, initialised with ones since a priori any color can be assigned to any vertex (this can be changed if one wishes to change the amount of initial information fed to the algorithm); then both vertices and colors embeddings are initialised - the later ones can be randomly initialised ($\sim \mathcal{U}[0,1)$) but ideally they could be placed equidistant over a hypersphere. Next, both vertices and colors embeddings should be updated throughout t_{max} iterations according to the following set of equations:

$$\begin{aligned} \mathbf{V}^{(t+1)} &\leftarrow \phi_{\mathcal{V}}(\mathbf{V}^{(t)}, \mathbf{M}_{\mathcal{V}\mathcal{V}} \times (\mathbf{V}^{(t)}), \mathbf{M}_{\mathcal{V}\mathcal{C}} \times \mu_{\mathcal{C}\to\mathcal{V}}(\mathbf{C}^{(t)})) \\ \mathbf{C}^{(t+1)} &\leftarrow \phi_{\mathcal{C}}(\mathbf{C}^{(t)}, \mathbf{M}_{\mathcal{V}\mathcal{C}}^{\intercal} \times \mu_{\mathcal{V}\to\mathcal{C}}(\mathbf{V}^{(t)})) \end{aligned} \tag{4.13}$$

Finally, each vertex embedding sums up to a final vote on whether the entire graph is k-colorable or not.

Figure 4.21: Results obtained with the implementation of a k-colorability solver in the Typed Graph Networks library (see Code Listing 4.4).



```
tgn = GraphNN(
  {
    # $\tau_\mathcal{V} = \mathbb{R}^{d_v}$
    'V': dv,
    # $\tau_\mathcal{C} = \mathbb{R}^{d_c}$
    'C': de
  },
  {
    # $\mathbf{M} \in \mathbb{B}^{|\mathcal{V}|\times|\mathcal{V}|}$
    'VV': ('V','V'),
    # $\mathbf{VC} \in \mathbb{1}^{|\mathcal{V}|\times|\mathcal{C}|}$
    'VC': ('V','C')
  },
  {
    # $\mu_{\mathcal{V}\to\mathcal{C}} : \tau_\mathcal{V} \to \tau_\mathcal{C}$
    'V_msg_C': ('V','C'),
    # $\mu_{\mathcal{C}\to\mathcal{V}} : \tau_\mathcal{C} \to \tau_\mathcal{V}$
    'C_msg_V': ('C','V')
  },
  {
    # $\mathbf{V}^{(t+1)}, \mathbf{V}_h^{(t+1)} \leftarrow \phi_\mathcal{V}(\mathbf{V}_h^{(t)}, \mathbf{M}_{\mathcal{V}\mathcal{V}} \times \mathbf{V}^{(t)},$
                $\mathbf{M}_{\mathcal{V}\mathcal{C}} \times \mu_{\mathcal{C}\to\mathcal{V}}(\mathbf{C}^{(t)}))$
    'V': [
      {
         'mat': 'M',
         'var': 'V'
      },
      {
         'mat': 'VC',
         'var': 'C',
         'msg': 'C_msg_V'
      }
    ],
    # $\mathbf{C}^{(t+1)}, \mathbf{C}_h^{(t+1)} \leftarrow \phi_\mathcal{C}(\mathbf{C}_h^{(t)}, \mathbf{M}_{\mathcal{V}\mathcal{C}}{}^\intercal \times \mu_{\mathcal{V}\to\mathcal{C}}(\mathbf{V}^{(t)}))$
    'C': [
      {
        'mat': 'VC',
        'msg': 'V_msg_C',
        'transpose?': True,
        'var': 'V'
      }
    ]
  },
)
```

Listing 4.4: TGN kernel of an end-to-end differentiable model to answer whether or not a given graph accepts a k coloring, in this model only vertices embeddings are taken into account to compose the final answer

5 TRAVELING SALESPERSON PROBLEM

The Traveling Salesperson Problem (TSP) is one of the most studied problems in optimization, computational complexity theory and theoretical computer science in general. In simple terms, it asks the following question: “given a set of cities and the knowledge of all intercity distances, what is the shortest possible route that visits each city exactly once and returns to the starting point?”[1]. The TSP has numerous applications on a wide range of areas. The TSP and variations thereof are extensively studied in the context of *planning*, yielding solutions to practical problems such as vehicle routing (GENDREAU; LAPORTE; VIGO, 1999). The concepts of *cities* and *intercity distances* are easily interchangeable with other elements such as DNA fragments and soldering points, which renders the TSP applicable to problems as diverse as DNA sequencing and microchip design.

5.1 Formulation

The TSP is defined over the concept of a *Hamiltonian path*, for which a definition is available below.

[Hamiltonian path] An *Hamiltonian path* in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined as a sequence of edges $P = \{e_1, e_2 \dots e_{|\mathcal{V}|} \mid e_i \in \mathcal{E}\}$ such that each vertex $v \in \mathcal{V}$ is covered by exactly one edge in the path. In other words, a Hamiltonian path is a path in the edges of $\mathcal{G}$ which visits every vertex exactly once.

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the TSP can be defined as the problem occupied with finding the Hamiltonian path of shortest length in $\mathcal{G}$. Concretely, it asks for the Hamiltonian path $P = \{e_1, e_2 \dots e_{|\mathcal{V}|} \mid e_i \in \mathcal{E}\}$ which minimizes $\sum_{i=1}^{|\mathcal{V}|} w_i$, where $w_i \in \mathbb{R}$ is the weight of the corresponding edge.

As a combinatorial optimization problem, the TSP lends itself easily to a integer linear programming formulation. The formulation below, which is due to (DANTZIG; FULKERSON; JOHNSON, 1954), implements binary variables to encode sets of edges representing tentative solutions. The ILP formulation minimizes

[1]Note that, in this context, a *city* does not correspond necessarily to a human settlement but rather to a generalized notion of a point in (some) space. Concretely, the TSP is defined over graphs where cities correspond to vertices and intercity distances correspond to edge weights.

the sum of all such variables, weighted by the corresponding edge weights. The weighted sum effectively computes the length of the corresponding Hamiltonian path, which is to be optimized. Two sets of restrictions enforce that each vertex has exactly one incoming edge and exactly one outcoming edge, respectively, while the last set of restrictions prohibits solutions containing sub-cycles (i.e. when all vertices are visited exactly once but the selected edges do not form a proper path).

[TSP ILP Formulation]

$$
\begin{aligned}
& x_{ij} = \begin{cases} 1 & \text{edge } (i,j) \text{ is in path} \\ 0 & \text{otherwise} \end{cases} \\
& \min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} w_{ij} x_{ij} : \\
& x_{ij} \in \mathbb{B} \\
& \sum_{i=1, i \neq j}^{n} x_{ij} = 1 && \forall j = 1 \ldots n \\
& \sum_{j=1, j \neq i}^{n} x_{ij} = 1 && \forall i = 1 \ldots n \\
& \sum_{i \in \mathbf{Q}} \sum_{j \in \mathbf{Q}} x_{ij} < |\mathbf{Q}| - 1 && \forall \mathbf{Q} \subset 2 \ldots n
\end{aligned}
\tag{5.1}
$$

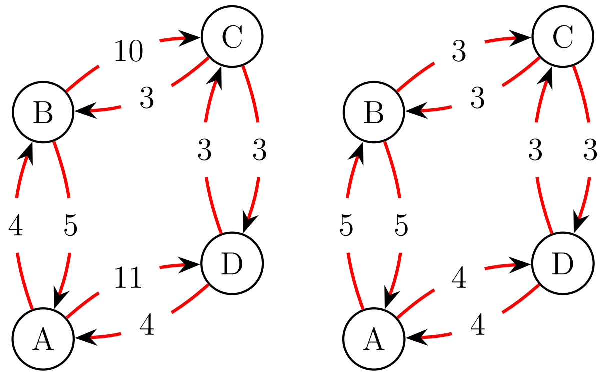
5.2 Variants and Special Cases

As a intensely studied research problem, the TSP has a number of variants. These variants directly influence the computational complexity and the practical feasibility of solving the problem, and as such demand a brief discussion. Some important cases are described here.

5.2.1 Asymmetric / Symmetric

TSP instances can either be **asymmetric** or **symmetric**, a property which is inherited from the graph for which we wish to compute the optimal route. The symmetric property, in this context, refers to when intercity distances are always equal in opposite directions ($w_{ij} = w_{ji}$) (Figure 5.1). Asymmetric and symmetric

Figure 5.1: Example of an asymmetric (left) and symmetric (right) graph. Source: Author.



correspond, respectively, to directed and undirected graphs. Asymmetry imposes additional difficulties for solving TSP instances, as paths may not necessarily exist in both directions and even on complete graphs the number of paths is duplicated w.r.t. their symmetric counterparts.

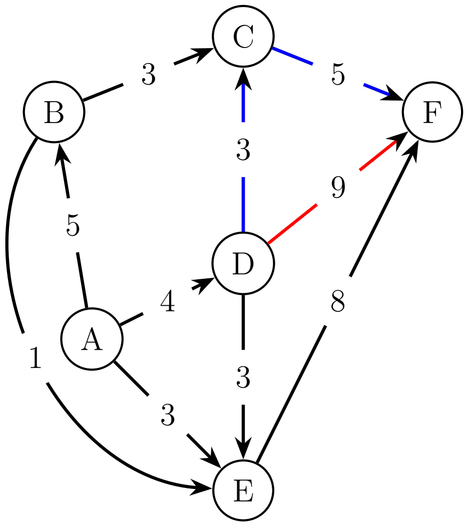
5.2.2 Metric

TSP instances can also be classified according to whether they respect or not the **metric property**. A graph is said to be metric when the set of cities, coupled with intercity distance values, form a **metric space**. In order to do so, it is required that intercity distances respect the **triangle inequality**. That is, it is required that the direct route $A \to B$ between two cities A and B is never longer than a shortcut $A \to C \to B$ passing through a third city C (Figure 5.2).

The metric property is relevant in the context of *approximation algorithms*[2]. Christofides' algorithm, which produces a valid Hamiltonian path for a graph $\mathcal{G}$ in polynomial time through a smart manipulation of its minimum spanning tree, was proven to yield a $\frac{3}{2}$-approximation for the TSP (in the sense that the generated paths measure no more than 1.5× the optimal tour length). Christofides' algorithm however requires that $\mathcal{G}$ have the metric property, and it is in fact known and easy to show that any graph which does not obey the triangle inequality cannot be approximated in polynomial time with a constant approximation factor unless $\mathcal{P} = \mathcal{NP}$ (AN; KLEINBERG; SHMOYS, 2015).

[2]In the context of combinatorial optimization, it is said that a problem admits an *approximation algorithm* when one can prove that said algorithm ensures that its solutions, while not necessarily optimal, are within a constant factor of the optimal value.

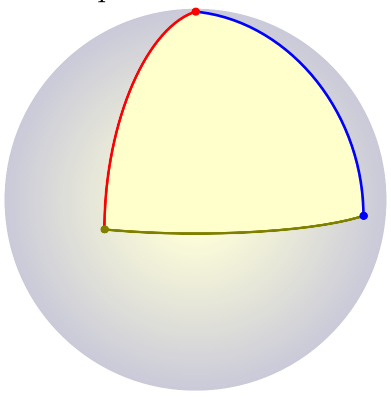
Figure 5.2: Example of a violation to the triangle inequality. The path composed by edges $D \to C$ and $C \to F$ underestimates the length of the direct connection $D \to F$.



5.2.3 Euclidean

A particularly relevant type of metric space is the *euclidean space*, which does not only respect the metric property but also enforces the *parallel postulate*, which states that given a line L and a point p not on it, at most one line parallel to L can be drawn passing through p (Figure 5.3). Euclidean spaces can have any dimensionality $\mathbb{R}^d \mid \forall d \in \mathbb{N}$.

Figure 5.3: A triangle drawn on the surface of a sphere. Two vertices are laid at the equator at 90° from each other while the third one is laid at the north pole, yielding three right angles. This is only possible because spherical geometry violates the parallel postulate. This is an example of a non-euclidean geometry. Source: Author.



5.3 Computational Complexity

Because solving the TSP includes finding a Hamiltonian path as a subproblem, the complexity of intuitive decision problem formulations of the TSP naturally inherit the $\mathcal{NP}$-Complete property from the latter. The Hamiltonian path problem, which is concerned with deciding whether a given graph contains a Hamiltonian path, is known to be $\mathcal{NP}$-Complete (by reduction from 3-SAT) for a long time, being one of the 21 selected Karp's $\mathcal{NP}$-Complete problems in his seminal paper (KARP, 1972). The usual decision version formulation of the TSP is concerned with deciding whether a given graph admits a Hamiltonian path of length no longer than a "target cost" C. In its usual (optimization) formulation, the TSP is known to be $\mathcal{NP}$-Hard. Specifically, the TSP is $\mathcal{FP}^{\mathcal{NP}}$-Complete. As $\mathcal{FP}$ is the class of *function problems*[3] solvable in polynomial time, the TSP is at least as hard as any function problem solvable in polynomial time given a polynomial-time oracle to a $\mathcal{NP}$-Complete problem[4]. This is the analogue of $\mathcal{NP}$-Completeness to the realm of function problems. From the viewpoint of optimization, the TSP is $\mathcal{NPO}$-Complete, and the viewpoint of approximation, the TSP is $\mathcal{APX}$-Complete (i.e. at least as hard as any problem which allows a constant-factor approximation in polynomial time).

[3]A function problem is a computational problem in which the output expected for each instance is expressed in terms of strings. This is in contrast with decision problems, in which the expected output is a binary answer (YES / NO).

[4]In the context of computational complexity theory, the "exponentiation" operator A^B expresses a variant of class A in which the regular Turing machine is replaced by a Turing machine with access to a polynomial-time oracle to a complete problem from class B.

6 TYPED GRAPH NETWORKS FOR THE DECISION TSP

6.1 Model

6.1.1 Intuitive Description

Concretely, our learning task is the following: we want to train a DL model to predict whether a given weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ admits a Hamiltonian cycle with cost no larger than a given real number $C \in \mathbb{R}_0^+$. In this context, each of our instances consists of a pair $X = (\mathcal{G}, C)$ of a graph and a real number.

Immediately, the fact that we are dealing with weighted edges imposes a difficulty in engineering a model to solve our problem using typical GNNs. At this point, the reader should be reminded that in their simplest formulation, GNNs solely assign embeddings to vertices and refine these embeddings iteratively. Connections between vertices are manifest in the graph's adjacency matrix $\mathbf{A} = \mathbb{B}^{|\mathcal{V}|\times|\mathcal{V}|}$, which is also fed to the model. Because vertex embeddings are subject to initialization, if we are dealing with labelled graphs[1], one could in principle feed labels to the model by injecting each of them into the corresponding initial vertex embedding. But in the context of a graph with labelled **edges** (which is our case), this is impossible. In a sense, this has to do with the fact that, in a typical GNN model, edges are not treated as "first-class citizens". Because they are not endowed with embeddings of their own, one could say that they are somewhat removed from the models' ontology. Edges do not have memories of their own, nor can they "vote" at the end of the process.

The model detailed in this section, which was developed chronologically before and served as inspiration for the formalization of Typed Graph Networks, benefits from the insight of assigning embeddings to **edges** in addition to vertices. The idea here is that by assigning embeddings to edges, each edge $e_i \in \mathcal{E}$ embedding can be initialized with the numerical information $w_i \in \mathbb{R}$ corresponding to the edge weight. In our setup, each edge (v_i, v_j) will send messages to and receive messages from vertices v_i and v_j. Over many message-passing iterations, we expect that edges and vertices' embeddings will become refined with relational (and numerical) information about the graph $\mathcal{G}$, which we can hopefully exploit to solve the problem at

[1]In a labelled graph, vertices have labels associated to them.

hand.

Of course, in order for the proposed TGN to be able to learn to refine edges and vertices' embeddings with valuable information (in contrast to garbage), we need to design a proper loss function and train our model on it using gradient descent. In our case, the loss function is fairly straightforward: after the decided upon number of iterations of message-passing, we extract a "vote" from each edge embedding. These votes are computed by a MLP which receives edge embeddings as input and produces a logit probability as output. With all votes in hand, we compute their average and the resulting scalar is the logit probability with which the model thinks that a Hamiltonian cycle with the appropriate length exists in $\mathcal{G}$. Having access to the (binary) correct labels for each training instance, we can now define our loss as the binary cross entropy between the predicted probability and the ground-truth, and gradient descent is applied over this loss.

6.1.2 Concrete Definition

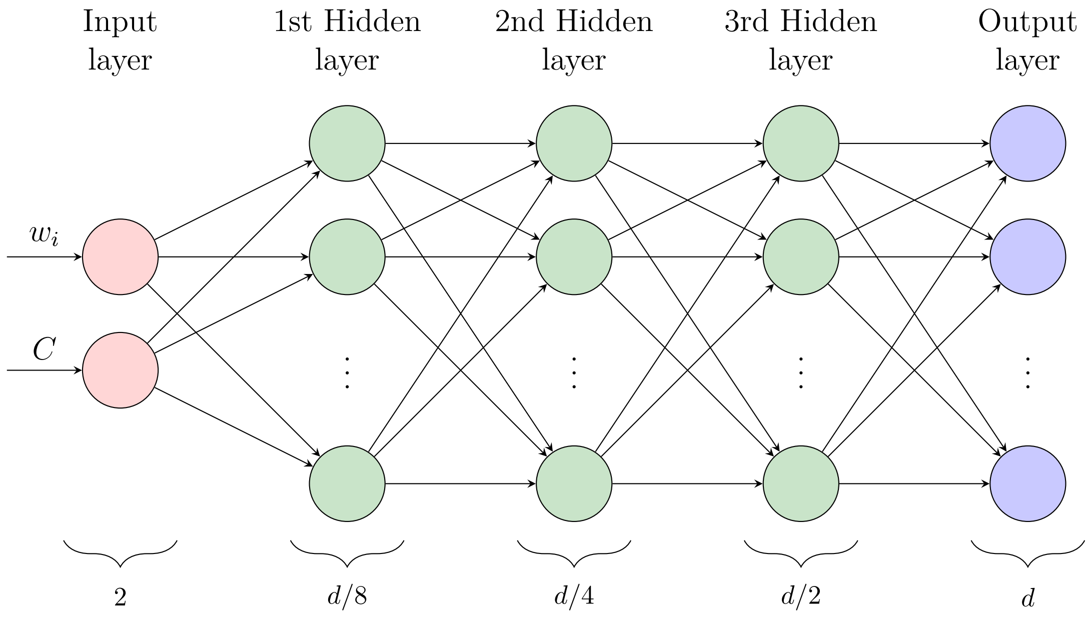
6.1.2.1 Embeddings and Embedding Initialization

Given a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, our model is structured in the following way. Each vertex $v \in \mathcal{V}$ is assigned a "memory" (idiomatically called an "embedding"), which corresponds to a multidimensional vector $\mathbf{V}_i \in \mathbb{R}^d$ (the dimensionality d is a hyper-parameter of our model). This memory unit will accompany the vertex throughout all iterations of message-passing, a process during which it will have its content updated step by step. Each edge $e \in \mathcal{E}$ will also be assigned an embedding, corresponding to a multidimensional[2] vector $\mathbf{E}_i : \mathbb{R}^d$. Both vertex and edge embeddings are subject to an initialization. In the case of vertex embeddings, our choice was to make the initial embedding equal for all vertices. This is line with the fact that, in our task, vertices do not have labels associated to them, and thus it makes sense not to differentiate between them in this initial stage. The initial embedding which is shared by all vertices is conceptualized as a **parameter** of our model (i.e. the model learns this initial embedding as it would learn the weights and biases of a feedforward network). In the context of edge embeddings, because we need to "inject" edge weights into them, our choice was to define a MLP E_{init}

[2]Note that the dimensionality d is shared between vertex and edge embeddings, but this must not be necessarily so. This was decided for simplicity.

which receives a scalar edge weight $w \in \mathbb{R}$ and expands it into the dimensionality of an edge embedding ($\mathbb{R}^d$). We also realized that this MLP can be used to inject C, the "target cost" of the TSP instance $\mathbf{X} = (\mathcal{G}, C)$ into each initial embedding. This way, edge embeddings will initially retain all numerical information relating to our problem. In this context, our MLP is a function $E_{init} \in \mathbb{R}^2 \to \mathbb{R}^d$ (i.e. it maps from ordered pairs of an edge weight and a target cost to initial edge embeddings). In our implementation, E_{init} is implemented as a MLP with fully-connected hidden layer containing $d/8, d/4, d/2$ neurons respectively and output size $= d$. A diagram representation of the architecture of E_{init} is offered in Figure 6.1.

Figure 6.1: Diagram representation of the MLP $E_{init} : \mathbb{R}^2 \to \mathbb{R}^d$ mapping ordered pairs of an edge weight an a target cost to an initial edge embedding. The curly braces at the bottom indicate the size (in number of neurons) of each layer.

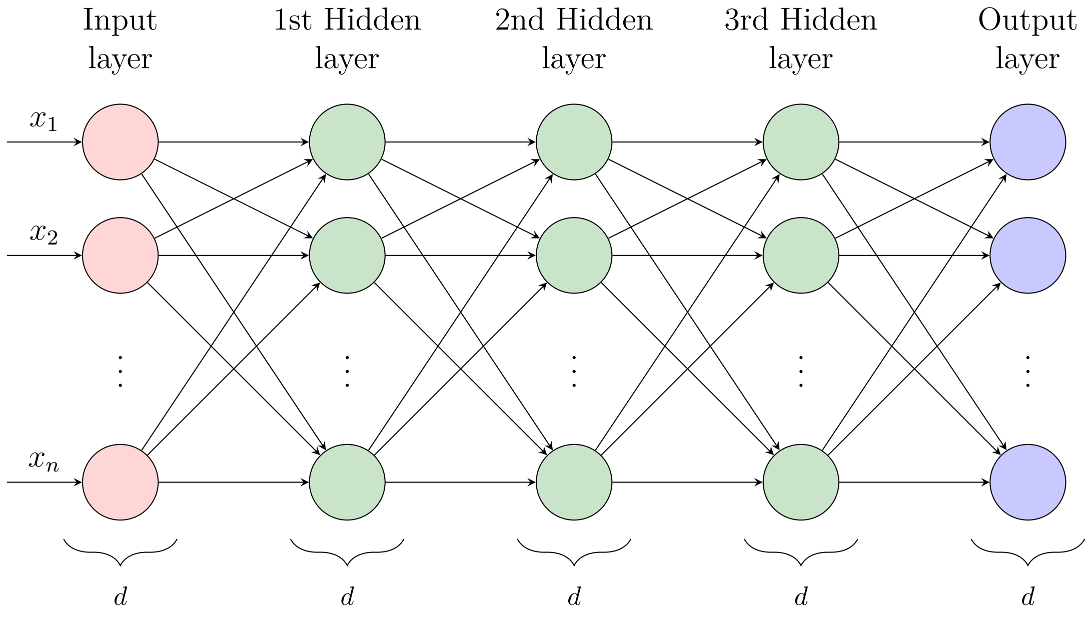


The reader should be reminded that embedding initialization can be visualized in two different paradigms. Programatically, we can think of initial embeddings as multidimensional vectors which are learned (in the case of vertices) or computed by a MLP (in the case of edges). But mathematically, we can think of initial embeddings as the image of a **map**, or a **projection** $\mathcal{V} \to \mathbb{R}^d$ or $\mathcal{E} \to \mathbb{R}^d$ between vertices or edges respectively and a multidimensional real space. Being able to interchange between these two paradigms is useful as it helps one to visualize the way that TGNs solve problems, which is qualitatively different from typical iterative computation. Instead of accessing and updating memory positions in a discrete manner, what TGNs do instead is to iteratively refine projections of vertices into multidimensional real space.

6.1.2.2 Message-Computing Functions

In our setup, vertices will send messages to edges and vice-versa. Part of the insight behind TGNs is that the computation which produces these messages is parameterized by a neural network. These messages are computed from edge or vertex embeddings. The reasoning here is that we want to read the memory (or the "state") associated with each object and from this information compute a message that this object wishes to send to its neighbors. Concretely, we instantiate two MLPs: $V_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$, to compute messages to send from vertices to edges, and $E_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$, to compute messages to send from edges to vertices. Because the embedding size of both vertices and edges is the same and also equal to the dimensionality of the computed messages ($\mathbb{R}^d$), both MLPs have been designed with the same architecture (although naturally we expect them to fit different roles and learn different relationships upon training), consisting of three fully-connected layers each with d neurons each, with an output layer of the same size. A diagram for the architecture of V_{msg} and E_{msg} is offered in Figure 6.2.

Figure 6.2: Diagram for the architecture of the message-computing functions V_{msg} and E_{msg}.

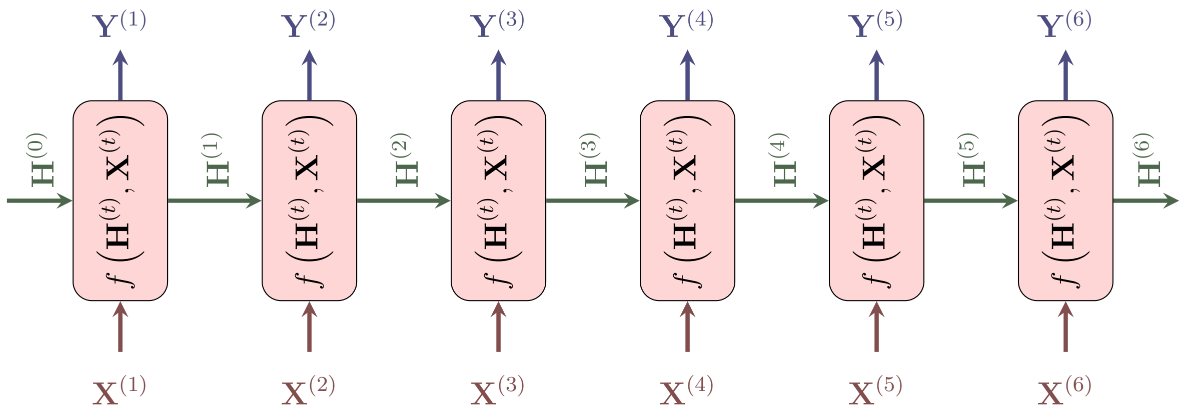


6.1.2.3 Update Functions

Crucially, vertices and edges should be able to update their embeddings upon receiving a set of messages. In our model, this update operation is parameterized by a recurrent neural network (specifically a LSTM). It should be noted however that because vertices and edges are treated as different "types" in our model, each of them is assigned a different RNN: V_u for vertices and E_u for edges.

Each vertex and each edge receives, at each iteration, a set of messages of variable size. RNN cannot be fed with a number of inputs which varies at execution time. Additionally, because each message is a vector of considerable size ($\mathbb{R}^d$), even if we had a fixed number K of messages received at each iteration, the layer size of our RNN would have to be $= K \times d$, which rapidly becomes prohibiting as K grows to moderate proportions. Here it should be briefly noted that the parameter space of a neural network grows exponentially with the size of its layers. As a result, each additional message received imposes a multiplicative overhead in the size of the parameter space, which severely hinders the learning capability of the model. This difficulty can be overcome quite simply, however, by performing a reduction, or an "aggregation", on the set of received messages. In our case, each vertex performs an element-wise addition of all of its received $\in \mathbb{R}^d$ message tensors, yielding an aggregated tensor which is also $\in \mathbb{R}^d$. This tensor is then fed to the RNN to yield an updated embedding for the vertex. The exact same reduction is pefomed by edges.

Figure 6.3: Pictorial representation of the unrolling of a recurrent unit f into six iterations. Because the parameters of all six blocks are shared, the resulting network can be thought of as iterating the same operation over the hidden state $\mathbf{H}^{(t)}$ and the input observations $\mathbf{X}^{(t)}$ that many times. Inputs $\mathbf{X}^{(t)}$ are colored red, outputs $\mathbf{Y}^{(t)}$ are colored blue and hidden states $\mathbf{H}^{(t)}$ are colored green. Repeated here for clarity. Source: Author.



Further explanation is warranted here, as this mechanism is possibly the least straightforward part of the model. The reader will remember that a RNN receives inputs and produces outputs as any other neural network, but additionally keeps record of an internal "hidden state" which is updated at each iteration. Conventionally, the process of training a RNN is fairly straightforward: the engineer has access to a list of training examples, each one of them being a sequence of feature / label pairs, and the RNN is unrolled and trained in order to receive a feature at each timestep and output the corresponding label at the same timestep (Figure 6.3). The main difference between training a RNN in the traditional fashion and training it in the context of a TGN is that, in the traditional case, the source of the features fed to the network is entirely external: they come from a training dataset. On the other hand, in the context of a TGN, the features fed to the network are indirectly produced by the same network on previous timesteps, because each feature is an aggregation of messages (which have been produced from the embeddings of vertices / edges which have in turn been updated by the RNN at hand).

In equations, the process of updating a vertex embedding and an edge embedding can be described as follows:

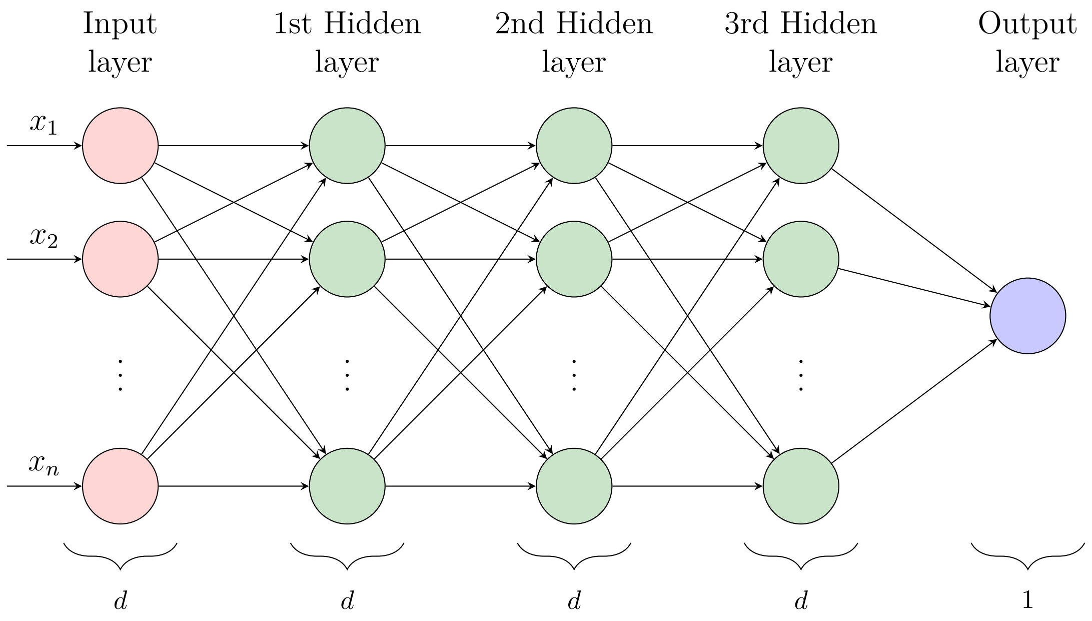
$$\begin{aligned} \mathbf{V}^{(t+1)} &\leftarrow V_u(\mathbf{V}^{(t)}, \mathbf{EV}^T \times E_{msg}(\mathbf{E}^{(t)})) \\ \mathbf{E}^{(t+1)} &\leftarrow E_u(\mathbf{E}^{(t)}, \mathbf{EV} \times V_{msg}(\mathbf{V}^{(t)})) \end{aligned} \tag{6.1}$$

These equations can be read like this: the RNN V_u, upon receiving (via self-loop) its own previous hidden state $\mathbf{V}_h^{(t)}$ and the aggregation of all messages received from edge embeddings, will produce a new hidden state $\mathbf{V}_h^{(t+1)}$ and a new vertex embedding $\mathbf{V}^{(t+1)}$ (the reading of the second equation is analogous). At this point the reader is possibly confused by the fact that instead of a proper aggregation of messages we have the expession $\mathbf{EV}^\intercal \times E_{msg}(\mathbf{E}^{(t)})$. This is due to the fact that these equations are performing embedding updates for all vertices / edges in parallel with tensor operations. $\mathbf{E}^{(t)}$ is not a single edge embedding but a $|\mathcal{E}| \times d$ **tensor** of edge embeddings. Accordingly, $E_{msg}(\mathbf{E}^{(t)})$ corresponds to a tensor of messages computed from edge embeddings. When we multiply this tensor at the left by the adjacency matrix between edges and vertices $\mathbf{EV}^\intercal$, the resulting value corresponds to a $|\mathcal{V}| \times d$ tensor where each i-th line is the aggregation over all messages received by the corresponding vertex v_i.

6.1.2.4 Voting Multilayer Perceptron

The final trainable component of our model is a MLP $E_{vote} : \mathbb{R}^d \rightarrow \mathbb{R}$ which "distills" an edge embedding into a logit probability, here referred to as a "vote" because it represents the probability with which that edge "thinks" that the problem instance at hand is solvable (i.e. that there exists a Hamiltonian route with the appropriate lenght in $\mathcal{G}$). The MLP is implemented with three hidden layer of sizes d, d, d and output size $= 1$, and a diagram of its architecture is offered in Figure 6.4.

Figure 6.4: Diagram for the architecture of the voting function E_{vote}.



6.1.2.5 Complete Model

Having defined all trained parameters of our model, we can now describe the entire algorithm. A single initial vertex embedding (which is a learnable parameter) is shared among all vertices and a different edge embedding is computed for each edge by a MLP $E_{init} : \mathbb{R}^2 \rightarrow \mathbb{R}^d$ which expands ordered pairs (w_i, C) of an edge weight and a route target cost into the dimensionality of edge embeddings. Then, for t_{max} iterations, we run the following process: vertex embeddings are updated by feeding the RNN $V_u : \mathbb{R}^{2\times d} \rightarrow \mathbb{R}^{2\times d}$ with the aggregation (element-wise addition) over all messages received by that vertex from its neighbor edges (each of which is computed by the MLP E_{msg} fed with an edge embedding). Analogously, edge embeddings are updated by feeding the RNN $E_u : \mathbb{R}^{2\times d} \rightarrow \mathbb{R}^{2\times d}$ with the aggregation (element-wise addition) over all messages received by that edge from its neighbor vertices (each of which is computed by the MLP V_{msg} fed with a vertex embedding). Finally, after all iterations, the refined set of edge embeddings is fed into the voting MLP E_{vote} to compute a logit probability from each edge embedding. These logits are averaged together to produce a final prediction. This process is also available as a pseudocode in Algorithm 6.

Algorithm 6 Typed Graph Network TSP Solver

```
1: procedure GNN-TSP($\mathcal{G} = (\mathcal{V}, \mathcal{E}), C$)
2:     // Compute binary adjacency matrix from edges to source & target vertices
3:     $\mathbf{EV}[i, j] \leftarrow 1$ iff $(\exists v' | e_i = (v_j, v', w)) | \forall e_i \in \mathcal{E}, v_j \in \mathcal{V}$
4:     // Compute initial edge embeddings
5:     $\mathbf{E}^{(1)}[i] \leftarrow E_{init}(w, C) \mid \forall e_i = (s, t, w) \in \mathcal{E}$
6:     // Run $t_{max}$ message-passing iterations
7:     for $t = 1 \ldots t_{max}$ do
8:         // Refine each vertex embedding with messages received from edges in which
           it appears either as a source target vertex
9:         $\mathbf{V}^{(t+1)} \leftarrow V_u(\mathbf{V}^{(t)}, \mathbf{EV}^T \times E_{msg}(\mathbf{E}^{(t)}))$
10:        // Refine each edge embedding with messages received from its source and its
           target vertex
11:        $\mathbf{E}^{(t+1)} \leftarrow E_u(\mathbf{E}^{(t)}, \mathbf{EV} \times V_{msg}(\mathbf{V}^{(t)}))$
12:    // Translate edge embeddings into logit probabilities
13:    $\mathbf{E_{logits}} \leftarrow E_{vote}\left(\mathbf{E}^{(t_{max})}\right)$
14:    // Average logits and translate to probability (the operator $\langle\rangle$ indicates arithmetic
       mean)
15:    prediction $\leftarrow$ sigmoid($\langle \mathbf{E_{logits}} \rangle$)
```

Note that even though we were able to provide diagrams for all sub-modules, the task of drawing a diagram for the entire model is prohibitively difficult, for many reasons. The most important of them is the fact that a TGN exploits a huge number of shared parameters. The sub-modules drawn in the last subsections are “cloned” multiple times in the complete model (i.e. once for each vertex or edge in $\mathcal{G}$).

6.2 Adversarial Training Concept

Having defined the model, we are left with the task of deciding how to train it. There are many approaches possible, the most intuitive of all being simply generating pairs of random graphs and random target costs $C \in \mathbb{R}_0^+$, solving the decision TSP on these instances and feeding the corresponding feature / label pairs to the model. However, we can do better than this. Because we have the power of generating our own instances (which is usually not the case in the context of deep learning), we can force the model to develop a sensitivity to even the slightest changes. Our idea is the following: generate a random graph $\mathcal{G}$, compute the cost C^* of its optimal solution and then present the model with two examples containing $\mathcal{G}$, one with a target cost slightly smaller than the optimal (for which the correct prediction would be NO as there is no route cheaper than the optimal) and one with a target cost slightly greater than the optimal (for which the correct prediction would be YES as there is in fact routes more expensive or equal to the optimal).

6.2.1 A Note on Adversarial Instances

Adversarial machine learning is one of the greatest developments of the last decade in artificial intelligence. Generative Adversarial Networks (GANs) now produce photorrealistic images capable of fooling the human eye, but for all their impressiveness, the fundamental insight behind the technique is rather simple (yet ingenious). The base idea behind adversarial machine learning as a whole, metaphorically speaking, is to remove a classification model from its comfort zone by engineering “adversarial instances” – instances whose generation is trained to reduce the classification accuracy of the model. In the context of GANs, these adversarial instances are produced by a “generator” network, which is trained to fool the classificator. By training both the generator and a “discriminator” (which differentiates real from fake images) in parallel, the adversarial setup imposed yields improved performance for both. The generator ultimately learns to produce images which are difficult to distinguish from real ones, while the discriminator ultimately sees its performance enhanced after being forced to distinguish even the slightest artifacts in fake images in order to correctly classify them.

An important note about the typical type of work GANs are trained to do, such as generating realistic-looking human faces, is that we do not have access to an exact solution for verifying whether a RGB image contains a face or not: this task is better suited to a neural network model. But consider our case: we can verify exactly whether a given graph $\mathcal{G}$ admits a route shorter than a given length. So in a sense, the training setup we propose is very similar to regular adversarial training, with an important twist: we replace the discriminator network by a non-neural, exact solver.

6.3 Experimental Setup

6.3.1 A Note on Hyperparameters, Reproducibility and Deep Learning "Alchemy"

As it is the case with virtually every modern deep learning architecture, our model sports a substantial number of hyperparameters. It is generally unfeasible to perform parameter tuning on such a large hyperparameter space. This is a well-known and largely discussed Achilles heel in modern DL research, which is often compared to *alchemy* for the lack of a structured approach to selecting hyperparameters (HUTSON, 2018). The models discussed in this thesis are by no means an exception to this problem. In this context, however, the reader should be reminded that the lack of such an approach – which we fully acknowledge – is not a caveat of this study specifically, but rather reflects a structural problem which permeates the entire field. Deep learning, in its modern formulation, is a fairly recent field with a largely disruptive effect on computer science, and the scientific community has not yet been able to agree upon the best practices on carrying out experiments and designing models. We also face an additional difficulty in the fact that training DL models is computationally costly, and not every research team – certainly not ours – has the resources to perform exhaustive searches on the space of hyperparameters. That being said, we are keenly interested in enabling the full reproducibility of our results. As DL researchers ourselves, we are particularly aware of the damage that the lack of reproducibility poses to our field. Papers are often very hard or impossible to reproduce, and this partially stems from the fact that even the most acute hyperparameter decisions can have a decisive impact in the behavior of the trained model. For this reason, we went out of our way to report our hyperparameters exhaustively. The reader can consult them on Table 6.1. Also, for reproducibility purposes, the training setup parameters are equally important. For this reason we also report them on Table 6.2.

6.3.2 Hyperparameters of our Model

Our model sports four MLPs and two RNNs. Each of these neural modules can be expanded into a large number of hyperparameters. The complete collection of hyperparameters in the entire model is described in Table 6.2, below.

Table 6.1: Hyperparameters of the model

Vertex embedding size (d)		64
Edge embedding size (d)		64
Vertex-to-edge message embedding size (d)		64
Edge-to-vertex message embedding size (d)		64
Edge initialization MLP (E_{init})	Input size	2
	Hidden layers	3
	Hidden layer sizes	8,16,32
	Output size	64
	Non-linearities	ReLU
Vertex-to-edge message-computing MLP (V_{msg})	Input size	64
	Hidden layers	3
	Hidden layer sizes	64,64,64
	Output size	64
	Non-linearities	ReLU
Edge-to-vertex message-computing MLP (E_{msg})	Input size	64
	Hidden layers	3
	Hidden layer sizes	64,64,64
	Output size	64
	Non-linearities	ReLU
Vertex embedding update RNN (V_u)	Input size	64
	Hidden state size	64
	Output size	64
	Non-linearities	ReLU
Edge embedding update RNN (E_u)	Input size	64
	Hidden state size	64
	Output size	64
	Non-linearities	ReLU
Voting MLP (E_{vote})	Input size	64
	Hidden layers	3
	Hidden layer sizes	64,64,64
	Output size	1
	Non-linearities	ReLU

6.3.3 Training Setup Parameters

In our experiments, the model is trained for 2000 epochs, each one sporting 128 batches of 16 instances each. Overall, the training setup is also parameterized by a series of other decisions, such as the optimizer used, L2 normalization, clipping, among others. All such decisions are exhaustively described in Table 6.2.

Table 6.2: Training setup parameters

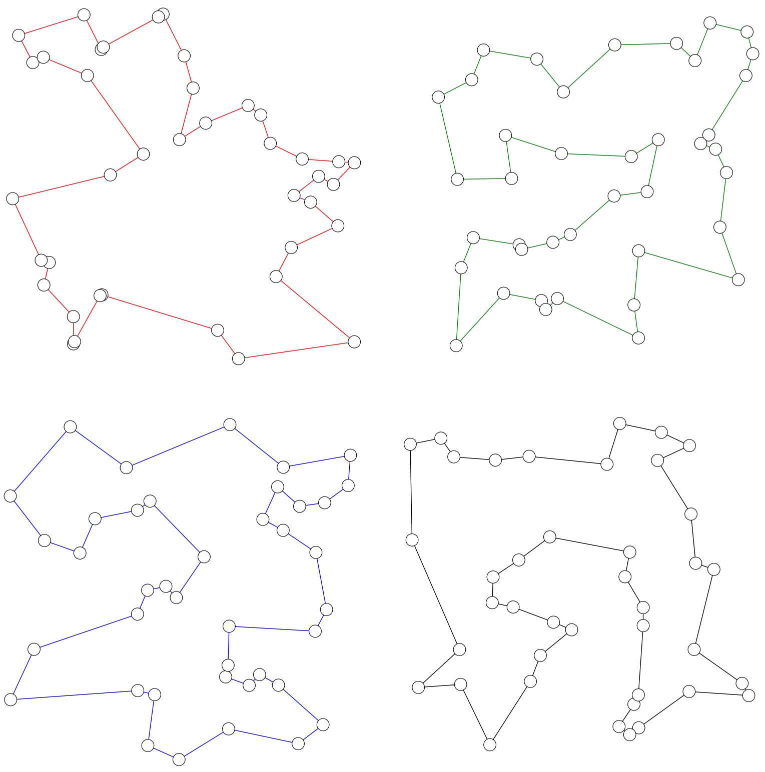
Epochs	2000
Batches per epoch	128
Batch size	16
Time steps	32
MLP Kernel Initializer	Xavier Initializer
MLP Bias Initializer	Zeros Initializer
Vertex Embeddings Initialization	Random Normal
Optimizer	Adam Optimizer
Learning rate	2×10^{-5}
L2 Normalization?	Yes
L2 norm scaling	10^{-10}
Clip by global norm?	Yes
Global norm gradient clipping ratio	0.65

6.3.4 Training Instances

We have decided to train our model with complete, 2D Euclidean instances. They are generated in the following way: initially, we sample n points in the $\sqrt{\frac{1}{2}} \times \sqrt{\frac{1}{2}}$ square[3]. We then use the Concorde TSP (exact) solver (HAHSLER; HORNIK, 2007) to compute the optimal route (and associated route length) for each instance. As described in Section 6.2, we employ an adversarial training concept which consists in producing two training instances from each generated graph $\mathcal{G}$. Having computed the optimal TSP cost C^* for $\mathcal{G}$, we produce two instances $X^- = (\mathcal{G}, (1-\Delta)C^*)$ and $X^+ = (\mathcal{G}, (1+\Delta)C^*)$. The reader will realize that a correct binary classfier for the decision TSP should answer NO for X^-, as it asks whether $\mathcal{G}$ admits a route shorter than the optimal, and YES for X^+, as it asks whether $\mathcal{G}$ admits a route longer than the optimal. In this context, Δ is the "deviation" parameter, defining by how much

[3]The reason for this is that we want, for simplicity, that the distance D_{ij} between any two points p_i, p_j is $0 < D_{ij} < 1$. The choice of a square of side length $\sqrt{\frac{1}{2}}$ gives us this for free as it yields a diagonal of length $= 1$.

Figure 6.5: Examples of 2D Euclidean graphs sampled from the distribution used to train our model, with the associated optimal TSP routes drawn in red, green, blue and black.



these two instances deviate from the optimal. In our experiments, we used $\Delta = 2\%$. This and all other parameters for instance generation are described in Table 6.3, and four examples of graphs, as well as their optimal TSP tours, are presented in Figure 6.5.

Table 6.3: Training instances generation parameters

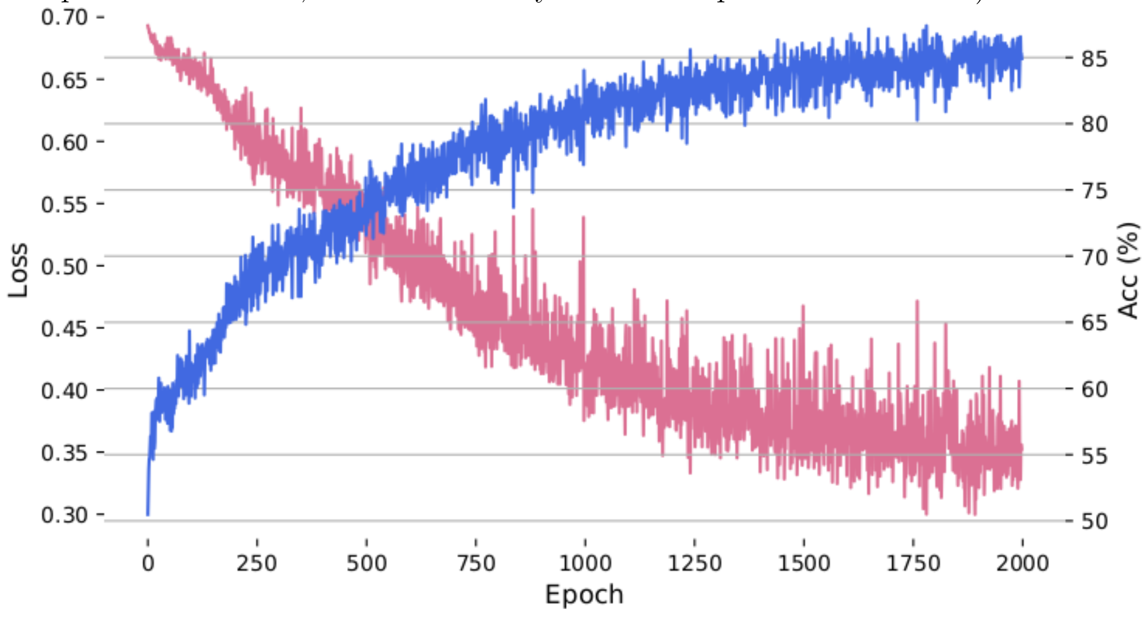
Random distribution	2D Euclidean
Edge density	100% (complete)
Sizes (n)	$n \sim \mathcal{U}(20, 40)$
Deviation (Δ)	2%

6.4 Results and Analyzes

Upon 2000 training epochs, the model achieved 80.16% accuracy averaged over the 2^{21} instances of the training set, having also obtained 80% accuracy on a testing set of 2048 instances it had never seen before. Instances from training and test datasets were produced with the same configuration ($n \sim \mathcal{U}(20, 40)$ and 2%

percentage deviation). Figure 6.6 shows the evolution of the binary cross entropy loss and accuracy throughout the training process.

Figure 6.6: Evolution of the binary cross entropy loss (downward curve in red) and accuracy (upward curve in blue) throughout a total of 2000 training epochs on a dataset of 2^{20} graphs with $n \sim \mathcal{U}(20, 40)$. Each graph with optimal TSP route cost C^* is used to produce two instances to the TSP decision problem – "is there a route with cost $< 1.02C^*$?" and "is there a route with cost $< 0.98C^*$?", which are to be answered with YES and NO respectively. Each epoch is composed of 128 batches of 16 instances each (please note that at each epoch the network sees only a small sample of the dataset, and the accuracy here is computed relative to it).



6.4.1 Stochastic Gradient Descent and Accuracy Variation for the Same Training Setup

The reader should be reminded that DL models are conventionally trained using Stochastic Gradient Descent, or SGD for short. In this setup, each individual gradient descent operation is applied w.r.t. a gradient computed on a batch of possibly many instances, sampled at random from the training set. For this reason, two subsequent trainings of the same exact model with the same exact training setup can – and will – yield different parameterizations due to the fact that there is randomness inherent to the choice of each batch of instances. Unfortunately, there are often not enough compute resources to run the same model multiple times and evaluate the variation in its overall accuracy. Our resources are limited, but we were able to repeat the training process ten times in order to analyze this variation and report the corresponding averages.

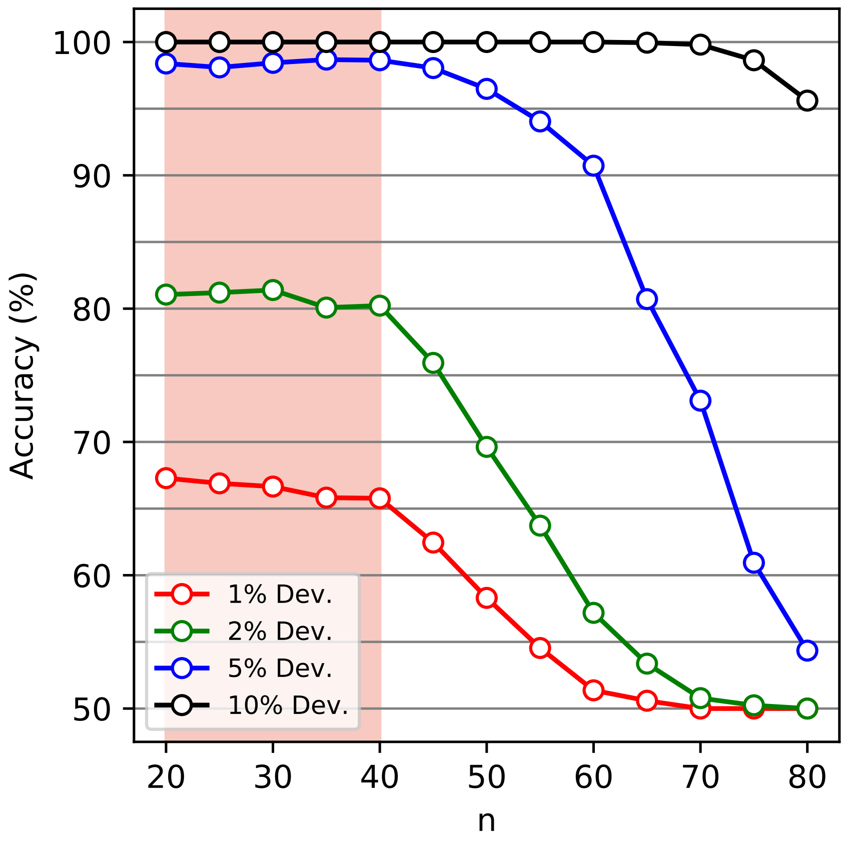
6.5 Generalization at Test Time

One of the conventional approaches to validating a DL model is to promote a division in the set of available examples in which part of them are used to train the model while the remaining is used to test it. The rationale is that, by evaluating the model's accuracy on a set of instances it has never seen during training, we can be more confident that the good accuracy seen on the training set is not a result of overfitting. In this context, we have substantially more flexibility than the conventional DL researcher, because we are enabled to produce our our training and test instances. For this reason, we have chosen to go a step ahead and test the accuracy of our model not only on *instances* it has never seen before but also on *distributions of instances* it has never seen before.

6.5.1 Different Sizes

An aspect in which our work deviates from what is conventional in DL research is that we have the ability to evaluate the accuracy of our model on varying instance sizes, regardless of whether it has seen any of them during training. This is due to the fact that TGNs operate on the level of the **nodes** of a graph, and in this context are naturally applicable to graphs of any size. This presents us with a perfect opportunity to evaluate how the accuracy of our model changes with larger instance sizes. Naturally, we expect that the model will be either unable to generalize to larger instances than those seen at training time or that its performance will degrade for increasing sizes (as their search spaces increasingly become exponentially larger). Upon testing, what we actually observe is the second scenario: the model is in fact able to generalize to larger sizes, but with diminishing results as their sizes grow beyond is "comfort zone" (between 20 and 40 nodes). Figure 6.7 illustrates this behavior, showing how the trained model is able to mantain the accuracy approximately constant inside its comfort zone, but sees it drop progressively for larger sizes. As well as plotting the varying accuracy for the distribution of $-2\%, +2\%$ instances for which training examples are sampled, we also show the same curves for different deviation values either smaller or larger than 2% $(-1\%, 5\%, 10\%)$. In this setup, it becomes clear that while the model's accuracy decreases for increasing sizes, it also increases for incresing deviations. This result, albeit not necessarily an expected finding, makes sense: larger instances have larger search spaces, and larger deviations yield more relaxed constraints.

Figure 6.7: Accuracy of the trained model evaluated on datasets of 1024 instances with varying numbers of cities (n). The model is able to obtain $> 80\%$ accuracy for $-2\%, +2\%$ deviation on the range of sizes it was trained on (painted in pink), but its performance degenerates progressively for larger instance sizes before reaching the baseline of 50% at $n \approx 75$. Larger deviations yield higher accuracy curves, with the model obtaining $> 95\%$ accuracy for $-10\%, +10\%$ deviation even for the largest instance sizes.



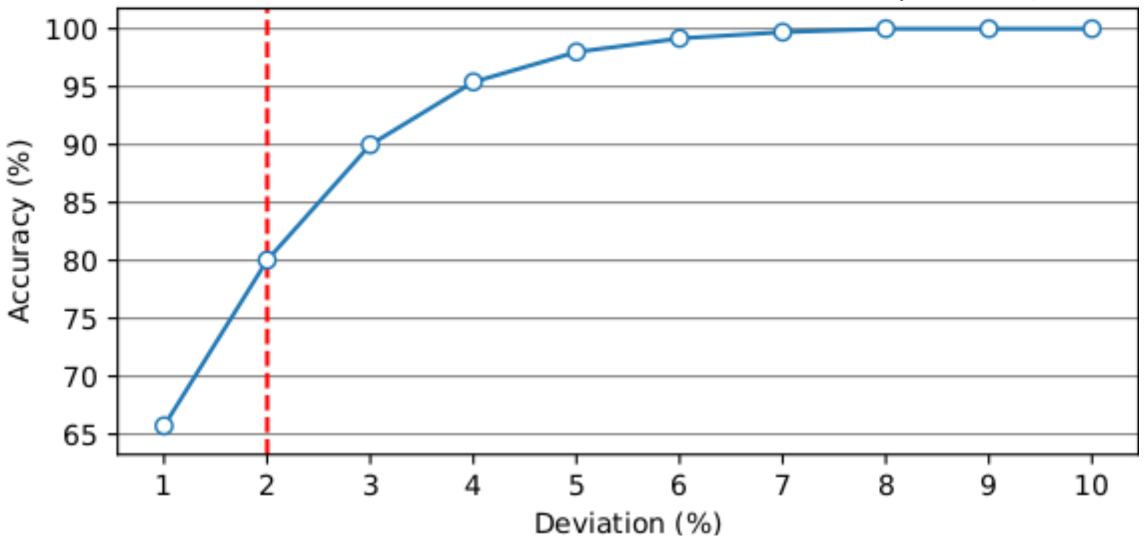
6.5.2 Different Deviations

Another generalization test we can perform without effectively changing the underlying random graph distribution is to evaluate how the trained model's performance changes with the deviation Δ from the optimal tour cost. As commented in Section 6.5.1, we expect larger and smaller deviations to yield easier and harder instances, respectively, because the smaller the deviation the more constrained our decision instances become. This is in fact what we observe in this experiment, as Figure 6.8 illustrates and Table 6.4 reports. In this context, we can see that one must not raise Δ too much in order to obtain almost perfect accuracy: at $\Delta = 5\%$ the model is already at 2% from no mistakes whatsoever.

Table 6.4: Test accuracy averaged over 1024 n-city instances with $n \sim \mathcal{U}(20, 40)$ for varying percentage deviations from the optimal route cost.

Deviation (Δ)	Accuracy (%)
1%	66
2%	80
5%	98
10%	100

Figure 6.8: Accuracy of the trained model evaluated on the same test dataset of 1024 n-city instances with $n \sim \mathcal{U}(20, 40)$ for varying deviations from the optimal tour cost. Although it was trained with target costs $-2\%, +2\%$ from the optimal (dashed line), the model can generalize for larger deviations with increasing accuracy. Additionally, it could still obtain accuracies above the baseline (50%) for instances more constrained than those it was trained on, with 65% accuracy at $-1\%, +1\%$.



6.5.3 Different Graph Distributions

A particularly interesting experimentation is to evaluate whether the model – which, the reader will recall, was trained solely on complete 2D Euclidean graphs – is capable of generalizing to extraneous graph distributions. Naturally, there is an insurmountable number of graph distributions to experiment with, rendering an exhaustive analysis unfeasible. In this context, probably the most important set of graphs to initially experiment with is that where edge weights are drawn from an uniform distribution. We justify this by pointing out that every weighted graph is equiprobable in this context, in contrast with (for example) Euclidean distributions in which some weight matrices are improbable and some even impossible. This is further explained in Appendix **??**.

We averaged accuracy results over 1024 n-city instances defined over uniformly random weight matrices, with $n \sim \mathcal{U}(20, 40)$ as conventional. To our surprise, the model experienced the worst performance possible, consistently yielding 50% accuracies. Note that the model produces 50% accuracy **regardless** of how relaxed we choose the deviation Δ to be. As we enforce a 50/50 divide between positive and negative labels, the model's accuracy in this context is equivalent to that of a coin toss. However, it is known that TSP instances which do not conform to the metric property pose a significant difficulty to heuristic methods (AN; KLEINBERG; SHMOYS, 2015). In this light, we experimented with producing metric variants of the same random weight matrices, which can be done straightforwardly by replacing each intercity distance D_{ij} by the corresponding shortest path length between cities p_i, p_j. Upon performing this modification, we were surprised once again to see that the model's performance not only raises above the 50% baseline but also resumes the typical behavior of yielding better accuracies the larger the deviation Δ is (Table 6.5).

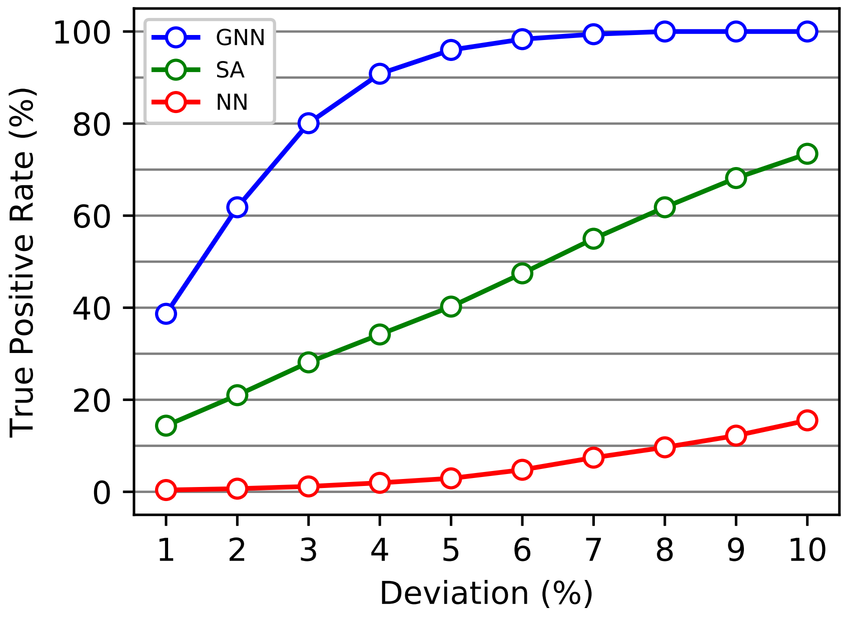
Table 6.5: Test accuracy averaged over 1024 n-city instances with $n \sim \mathcal{U}(20, 40)$ for varying percentage deviations from the optimal route cost for differing random graph distributions: two-dimensional euclidean distances, "random metric" distances and random distances.

Deviation	Accuracy (%)		
	Euc. 2D	Rand. Metric	Rand.
1	66	57	50
2	80	64	50
5	98	82	50
10	100	96	50

6.5.4 Comparing with Classical Baselines

This work would not be complete without performing a comparison between the performance of the trained model and some kind of baseline. However, this is not as simple as it would seem, mostly because training DL models to solve relational problems is a fairly recent research effort. In this context, we chose to perform an admittedly modest comparison with two classical baselines: a Nearest Neighbor (NN) and a Simulated Annealing (SA) search. Nevertheless, we face an additional difficulty in the fact that the TSP problem is not usually solved in its decision problem formulation, as we are frequently interested in obtaining a Hamiltonian path. Because of this, we were forced to adapt the NN and SA heuristics to produce predictors for the decision TSP. This was done by measuring, for a given decision instance $X = (\mathcal{G}, C)$ the frequency with which either of these algorithms produced a solution with cost no greater than C.

Figure 6.9: Nearest Neighbor (NN) and Simulated Annealing (SA) do not yield a prediction for the decision variant of the TSP but rather a feasible route. To compare their performance with our model's, we evaluate the frequency in which they yield solutions below a given deviation from the optimal route cost and plot alongside with the True Positive Rate (TPR) of our model for the same test instances (1024 n-city graphs with $n \sim \mathcal{U}(20, 40)$).



This frequency can be directly compared with the True Positive Rate (TPR) of our trained model. The TPR was chosen instead of the accuracy because NN and SA cannot ever *decide* that a route shorter than a given length does not exist, as they are forced to produce a valid (yet not provably optimal) Hamiltonian path. Our experiments show that the trained model outperforms both SA and NN in this setup. This result can be alternatively read as: our model is able to decide that a given graph $\mathcal{G}$ admits a route shorter than a given length C more frequently than either SA or NN can *produce* a route shorter than C. One can immediately see that the comparison is not fair, as deciding that a route exists is obviously easier than producing said route. Nevertheless, up until this point we cannot be completely sure that the trained model does not effectively search for a valid Hamiltonian route before producing its prediction, eventually finding it and projecting it into the hyperdimensional space of vertex and edge embeddings. For this reason, we believe this particular experiment is important.

6.6 Interpretability

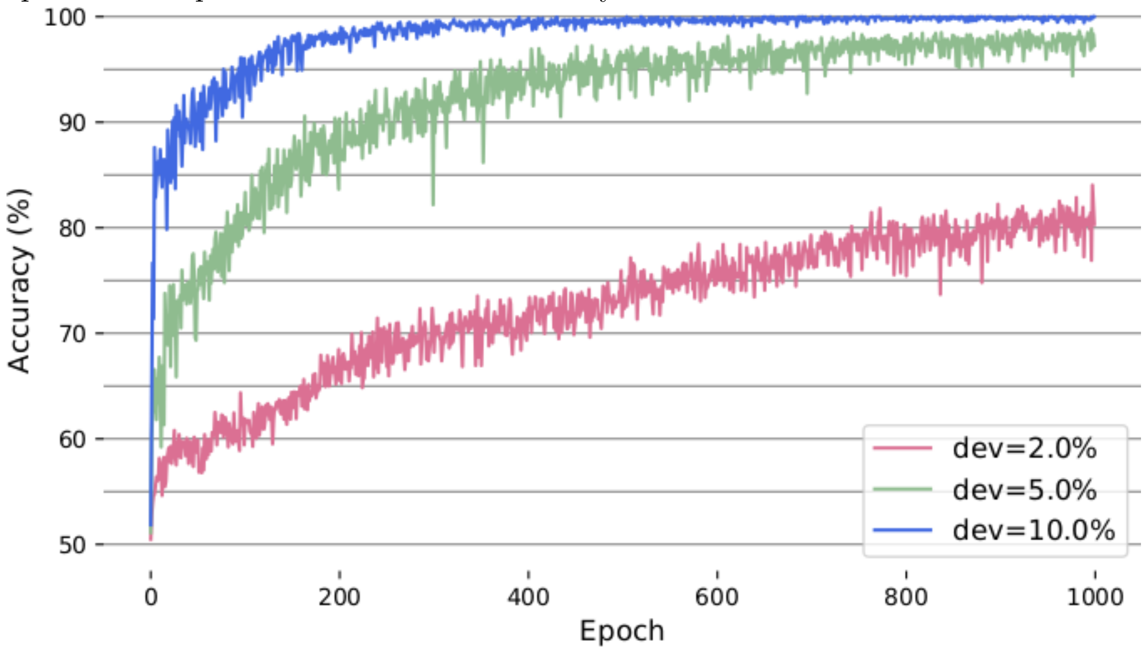
6.7 Training

One of the most interesting, yet costly, aspects to experiment with in our setup is *training*. In a way, the model proposed in this thesis is an object of study in itself, and its importance is justified by the answers it provides concerning the applicability of TGNs in particular and ANNs in general on routing problems. Nevertheless, in a very real sense, the work developed here can also turn the tables around and use the TSP as a case study to try and crack open the black box of TGNs, as it is of significant interest to the DL community to understand how exactly they work. In this context, experimenting with different training setups can possibly help us understand what effects the training data and some selected hyperparameters can have on the algorithm learned by the model.

6.8 Training with Varying Deviations

An initial simple experiment, in the light of what was observed at test time, is to evaluate whether training the model with larger deviations has an effect. By training the same model with $2\%, 5\%, 10\%$ deviations we can clearly see that more relaxed deviations yield close to perfect accuracy much sooner, with 10% for example yielding approximately 100% accuracy upon 400 training epochs. This is in contrast to the 2000 training epochs required for the original model to achieve 80% accuracy (Figure 6.10).

Figure 6.10: The larger the deviation from the optimal cost, the faster the model learns: we were able to obtain $> 95\%$ accuracy for 10% deviation in 200 epochs. For 5%, that performance requires double the time. For 2% deviation, two thousand epochs are required to achieve 85% accuracy.

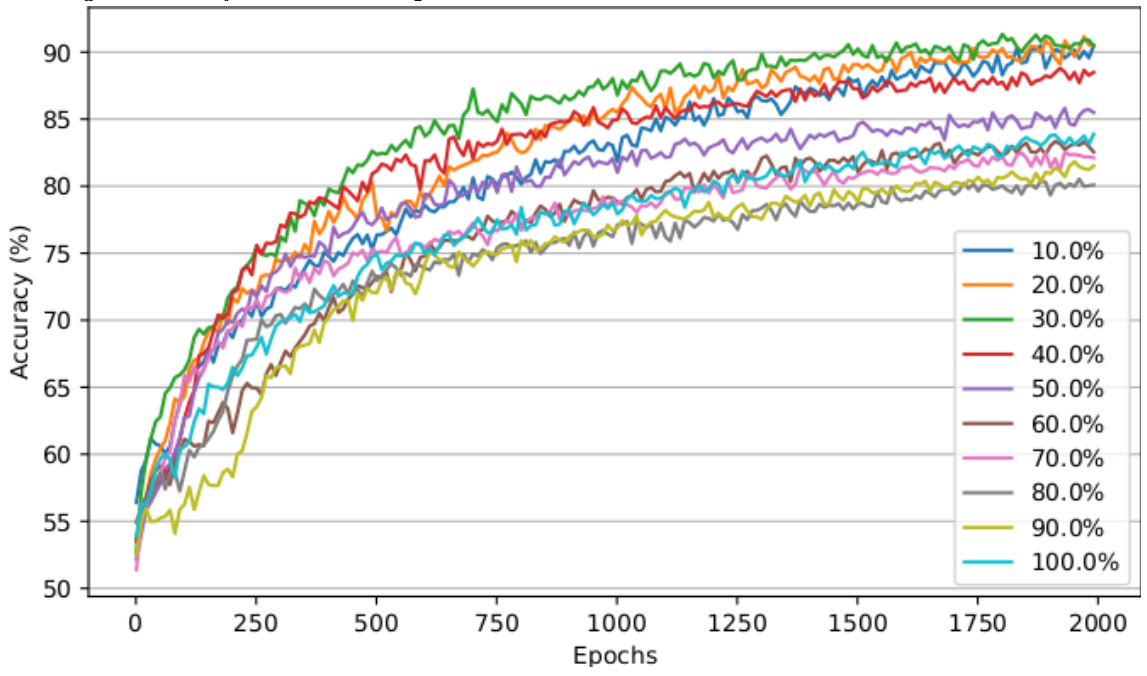


6.9 Training with Sparse Graphs

Another important experiment to assess the model's potential for generalization is to train it under multiple different edge densities. The initial experiment, the reader will remember, was performed on a dataset of complete graphs. In this experiment we produce 10 datasets with edge densities $10\%, 20\%, \ldots, 100\%$. This is done by initially generating a complete graph and then sampling a fraction of its edges uniformly at random.

Upon training, as Figure 6.11 shows, one can see that smaller edge densities are generally associated with faster learning curves. It should be noted, however, that this plot is potentially misleading as the accuracies reported are computed on the training set under consideration (i.e. the accuracies for the 20% regime are computed for instances which themselves contain only 20% of the total edges).

Figure 6.11: The result of training the same model with graphs of varying edge densities is shown. To ease visualization and make the plot smoother, we report the average accuracy at each 10 epoch interval.



A more insightful analysis, in this context, is to examine the generalization potential of each trained model, or, in other words, to examine the accuracy of a model trained with (for example) instances of 20% edge density for test datasets of instances of $10\%, 30\%, \ldots 100\%$ edge density. We performed such an experiment on test datasets of 1024 instances for each edge density. The corresponding results are detailed in Table 6.6 and can also be visualized on Figure 6.12.

Table 6.6: Exhaustive evaluation of the accuracies computed for all combinations of train and test datasets. In other words: cell (i, j) reports the accuracy of the model trained with the i-th train dataset w.r.t. the j-th test dataset. Train datasets consist of 2^{20} instances as usual, while test datasets consist of 2^{15} instances. All accuracies are reported in percentages, but the % symbol is removed due to space limitations. The average accuracy, computed over all test datasets, is reported at the last column. This table can also be visualized as a heatmap in Figure 6.12.

		Tested on										Average
		10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
Trained on	10%	90	86	71	60	54	52	51	52	54	54	62
	20%	89	88	81	74	67	61	56	53	52	51	67
	30%	77	92	93	87	73	62	56	53	51	50	69
	40%	73	89	93	92	88	83	78	73	70	67	80
	50%	69	82	88	88	85	80	76	73	70	67	78
	60%	59	74	79	83	86	86	82	77	72	68	77
	70%	52	64	75	81	83	82	80	76	73	70	74
	80%	54	68	74	74	72	73	76	80	82	81	73
	90%	51	52	57	64	69	73	78	82	84	81	69
	100%	50	50	57	59	59	58	58	60	66	87	60

Figure 6.12: Visual representation of the data reported in Table 6.6. For more details of how this data was collected refer the original Table.

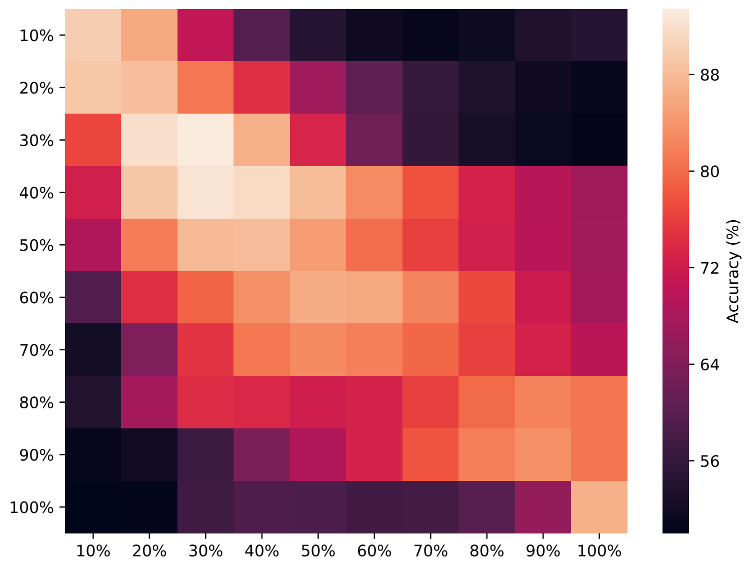


Table 6.6 shows that models trained at different edge distributions

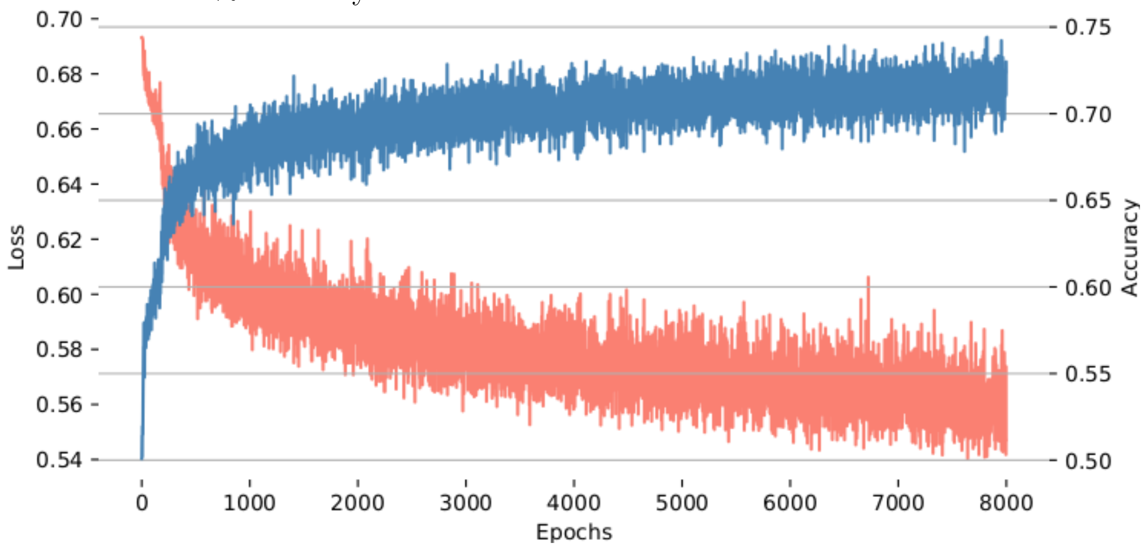
6.10 Training with General Graphs

Up until now, we have only examined our model under a fairly limited family of graphs: Euclidean 2D graphs. In Section 6.9 we show that the same model can be trained on complete and sparse graphs. In this section, we examine our model's capability of predicting the decision TSP on non-Euclidean graphs (i.e. graphs with random edge weights). To render the training dataset as general as possible, we also generate graphs with variable edge densities. To generate a n-vertex graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in our proposed random distribution, one must do as follows:

1. Choose an edge density $d \sim \mathcal{U}(0.2, 1)$ between 20% and 100% uniformly at random.
2. Sample dn^2 edges from the complete n-vertex graph uniformly at random.
3. For each edge $e_{ij} \in \mathcal{E}$, choose its edge weight $w_{ij} \sim \mathcal{U}(0, 1)$ uniformly at random.
4. Sample a random sequence of $\pi = \mathcal{N}^n$ of n nodes $\in \mathcal{N}$ and add the edges of the corresponding path to the graph, if inexistent, to enforce the existence of at least one Hamiltonian tour.
5. Enforce the metric property by replacing each edge weight w_{ij} by the shortest path distance between vertices v_i and v_j.

We generate 2^{15} such graphs. The size $n \sim \mathcal{U}(20, 40)$ of each graph is chosen uniformly at random between 20 and 40 vertices.

Figure 6.13: Loss and accuracy curves corresponding to the training setup of general graphs described above. Although training was significantly slower, we were able to achieve over 70% accuracy on this dataset.



6.11 Acceptance Curves & Extracting Route Costs

6.11.1 Acceptance Curves

More interesting than the results obtained by the trained model is how it effectively works. Understanding this is a challenge, as TGNs learn, upon training, a distributed algorithm which is not only “messy” in terms of the number of parameters involved in linear algebra operations but also conceptually very different from the algorithms we humans are familiar with. Thus, as a first step, it is profitable to start with a modest experiment and assess how the model’s prediction changes w.r.t. the target cost.

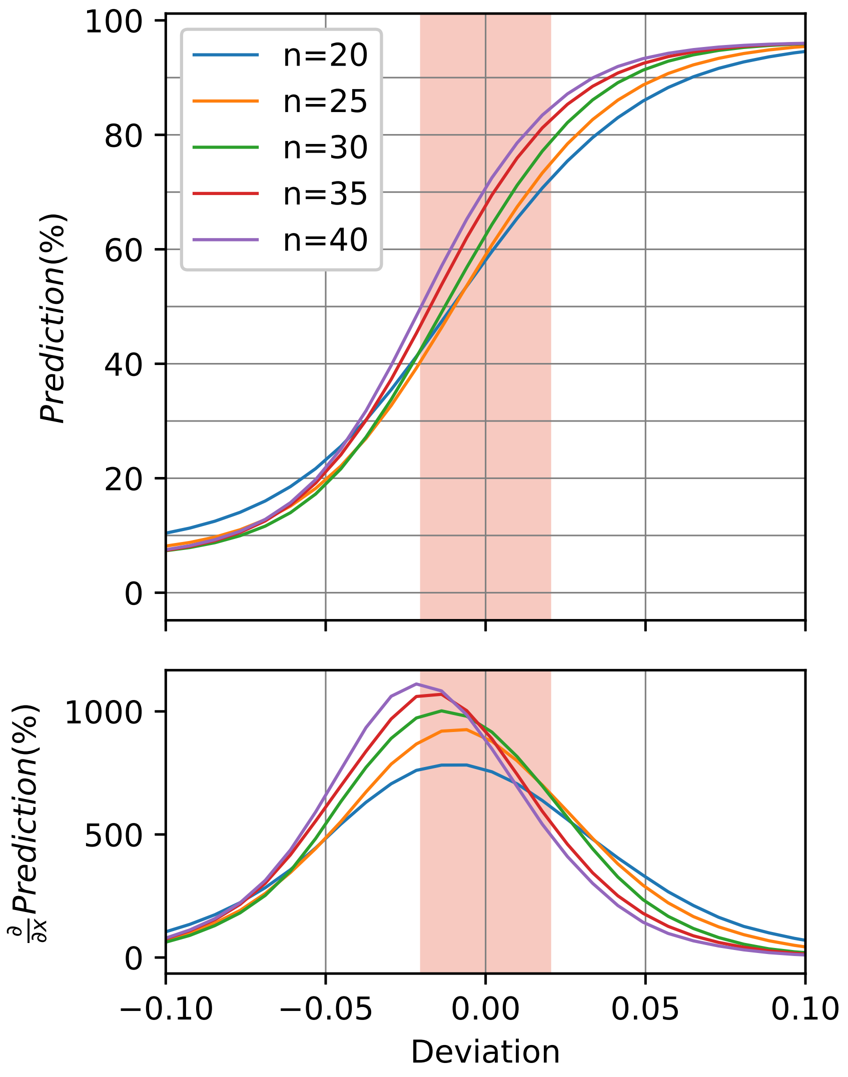
We propose the concept of an “acceptance curve”, which measures the average prediction of the model (i.e. the % of times it guesses YES for a test sample) as a function of the relative deviation of the target cost fed to the model from the ground truth. Such acceptance curves can be seen on Figure 6.14, in which a number of interesting phenomena can be observed.

Firstly, one can see that the curve undergoes a transition as the deviation increases. The model’s prediction seemingly exhibits two phases, one in which it (generally) predicts NO and one in which it (generally) predicts YES. The transition between these phases occurs (as expected) in the vicinity of $\Delta = 0$. Furthermore, one can see that larger instances yield sharper curves. We can even observe a phenomenon reminiscent of finite size scaling, in which smaller instances deviate more from the true critical point (which in theory happens at $\lim\limits_{n \to \infty}$) than larger ones. Note that the average optimal route length for n-city 2D Euclidean instances is proportional to $\sqrt{n}$ (BEARDWOOD; HALTON; HAMMERSLEY, 1959), so it makes sense as an heuristic for the model to guess, at $\Delta = 0$, that routes exist with more probability for larger instances.

6.11.2 Binary Search

Considering the findings described in Section 6.11.1, we can safely suggest that we are closest to the optimal route cost (i.e. $\Delta = 0$) when the model performs most poorly. In an idealized scenario, we could say that when the model exhibits absolute uncertainty (guessing with 50% probability), this is a symptom that the

Figure 6.14: Average prediction obtained from the model as a function of the deviation between the target cost and the optimal cost for varying instance sizes (the pink band indicates the $[-2\%, +2\%]$ interval). As expected, the curve is S-shaped, signalling that the model is very confident that routes with sufficiently large/small costs do/do not exist. The average prediction undergoes a phase transition as we traverse from negative to positive deviations. Larger instances exhibit smaller critical points, as evidenced by the left shifts on the derivatives of the acceptance curves in the bottom subfigure. The prediction for each deviation is averaged over 1024 instances.



target cost fed is the optimal TSP cost for the instance under consideration.

This line of reasoning suggests that it may be possible to exploit our trained binary classified to yield predictions for the optimal TSP cost, even though it was never trained to do so. In this context, we propose performing a simple binary search on the target cost, with the goal of maximizing the model's uncertainty. The procedure is described in detail in Algorithm 7.

Algorithm 7 Binary Search

```
1: procedure BINARY-SEARCH($\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $p$, $\delta$)
2:     // Choose an initial guess for the optimal route cost. $w^{n-}$ and $w^{n+}$ are the sets of the costs of the n edges $\in \mathcal{E}$ with smallest / largest costs respectively.
3:     $C_{min} \leftarrow \sum w_i^{n-}$
4:     $C_{max} \leftarrow \sum w_i^{n+}$
5:     $C \sim \mathcal{U}(C_{min}, C_{max})$
6:     while $C_{min} < C(1-\delta) \vee C(1+\delta) < C_{max}$ do
7:         if GNN-TSP$(\mathcal{G}, C) < p$ then
8:             $C_{min} \leftarrow C$
9:         else
10:            $C_{max} \leftarrow C$
11:        $C \leftarrow (C_{min} + C_{max})/2$
       return $C$
```

Table 6.7: The relative deviations from the optimal route cost are compared for the prediction obtained from the trained model with Algorithm 7 (GNN) and the Simulated Annealing heuristic (SA). Lines referring to instances in which the trained model outperformed and underperformed the SA heuristic are colored blue and red respectively. Note that deviations obtained from the trained model are negative in general, as expected given the discussion in the subsection about Extracting route costs above.

Instance	Size	Relative Deviation (%)	
		GNN	SA
ulysses16*	16	−22.80	+1.94
ulysses22*	22	−27.20	+1.91
eil51	51	−18.37	+18.07
berlin52	52	−8.73	+21.45
st70	70	−11.87	+14.47
eil76	76	−13.91	+19.24
kroA100	100	−2.00	+30.73
eil101	101	−9.93	+20.46
lin105	105	+6.37	+17.77

* These instances had their distance matrix computed according to Haversine formula (great-circle distance).

Upon running the binary search procedure on the test dataset (1024 Complete, 2D Euclidean n-city instances with $n \sim \mathcal{U}(20, 40)$), we were able to obtain predictions within 1.5% of the ground truth with on average 8.9 iterations. We also compare the absolute value of the relative deviation from the optimal cost of our method with that of a simulated annealing routine, the results of which are reported in Table 6.7. Our method outperforms SA for six out of the seven Euclidean instances tested, but underperforms SA substantially on the non-Euclidean instances *ulysses161* and *ulysses221*.

7 RECENT DEVELOPMENTS

7.1 Pytorch Geometric Library

Pytorch Geometric is a recent extension library for geometric deep learning in Pytorch. It has been officially presented in a workshop paper at ICRL 2019 (FEY; LENSSEN, 2019). Pytorch geometric encapsulates and simplifies the code behind graph convolutions, message-passing neural networks and other families of graph-based deep learning modules. In this context, it cares about the same goal as our Typed Graph Networks library, although it is generally more flexible than ours. In a sense, Pytorch Geometric operates at a slightly lower level than our library, which allows for more flexibility but requires comparatively more code. Nevertheless, because Pytorch is substantially less verbose than Tensorflow, codes in Pytorch Geometric are expected to be smaller on practice.

Overall, Pytorch Geometric allows for a high level of expressivity in terms of graph-based deep learning models, whilst keeping the code relatively short and simple. This is a virtue of both Pytorch Geometric and Pytorch itself, which is substantially less verbose than Tensorflow. Additionally, Pytorch Geometric's methods are optimized for performance by leveraging dedicated CUDA kernels (FEY; LENSSEN, 2019). For these reasons, we feel that this thesis would be incomplete without an acknowledgement of the excellent work the Pytorch Geometric team has done. We do this not *despite* the fact that Pytorch Geometric and our library are in a sense adversaries, but rather because of it. Nevertheless, it should be noted that although the development of both libraries started at approximately the same time (October 2017), we would not become aware of the latter until very near the end of the writing of this thesis, as the original paper describing it was published as a workshop paper at ICRL 2019. Keeping up with the fast pace of advancements in deep learning is no easy task.

7.1.1 Our Model in Pytorch Geometric

To demonstrate the differences between our proposed Typed Graph Networks library and the Pytorch Geometric library, we provide below some code snippets showing how our TSP solving module could be implemented in it.

Pytorch Geometric encapsulates the message-passing operation in a Python class which implements a Pytorch module. You can define a message-passing module by extending the Pytorch Geometric class “MessagePassing”. Pytorch Geometric provides a method named “propagate” which takes care of computing a message from each vertex embedding and aggregating all messages sent to a given vertex. In order to define how messages should be computed or aggregated, the programmer must override the “message” and “update” methods of the “MessagePassing” class. We can define the vertices-to-edges message-computing module of our proposed TSP GNN model in the following way:

```python
# Define vertices-to-edges message passing module
class Vertices2Edges(MessagePassing):
    def __init__(self,d):
        # Aggregate messages by adding them up
        super(Vertices2EdgesMsg,self).__init__(aggr='add')
        # Define message-computing MLP
        self.msg = Sequential(Linear(d,d), ReLU(), Linear(d,d), ReLU(), Linear(d,d), ReLU(), Linear(d,d))
        self.updater = LSTM(d,d)

    def forward(self, x, connections):
        return self.updater(self.propagate(connections, x=self.msg(x)))

    def message(self, x_i, connections):
        return x_i

    def update(self, aggregation):
        return aggregation

# Since the vertices-to-edges and edges-to-vertices architectures are equal
Edges2VerticesMsg = Vertices2EdgesMsg
```

Conceptually, it would make sense to compute a message from each vertex embedding separately in the overriden “message” function. However it is much faster to apply the transformation to the entire tensor of vertex embeddings at once, which is why we do it in the “forward” method.

The entire model now can be defined in a straigtforward way. The crux of the model is in lines 42-46, in which we use the vertices-to-edges and edges-to-vertices message-passing modules jointly with the recurrent modules to update embeddings for edges and vertices, for many iterations.

```python
class TSP_GNN(torch.nn.Module):
    def __init__(self.d):
        super(TSP_GNN, self).__init__()
        # Create vertices-to-edges and edges-to-vertices message passing layers
        self.V2E = Vertices2EdgesMsg(d)
        self.E2V = Edges2VerticesMsg(d)
        # Create initial vertex embedding
        self.Vinit = Variable(torch.randn(1,d), requires_grad=True)
        # Create edge embedding initializer
        self.Einitializer = Sequential(Linear(2,d//8), ReLU(), Linear(d//8,d//4), ReLU(), Linear(d//4,d//2), ReLU(), Linear(d//2,d))
        # Create edge 'voter'
        self.voter = Sequential(Linear(d,d), ReLU(), Linear(d,d), ReLU(), Linear(d,d), ReLU(), Linear(d,1))

    def forward(self, batch, timesteps):
        # Get:
        # n_vertices: N  of vertices for each instance
        # n_edges: N  of edges for each instance
        # edges: list of all edges in the batch
        # W: edge weights, in order
        # C: Vector of target costs (one for each instance)
        n_vertices, n_edges, VE, W, C = batch
        n_instances = len(n_vertices)
        total_vertices, total_edges = sum(n_vertices), sum(n_edges)

        # Init tensor of vertex embeddings V
        V = self.Vinit.repeat(total_vertices,1)

        # Repeat each instance's target cost for the No of edges in it
        # (This is done so that we can feed each initial edge embedding with its instance's target cost)
        C_expanded = torch.cat([c.unsqueeze(0).repeat(n) for n,c in zip(n_edges, C)], axis=0)]
        # Init tensor of edge embeddings E
        E = self.Einitializer(torch.cat([W,C_expanded], axis=1))

        # Precompute vertices-to-edges and edges-to-vertices communication masks
        conn_VE = [(i,total_vertices+e) for e,(i,j) in enumerate(edges)] + [(j, total_vertices+e) for e,(i,j) in enumerate(edges)]
        conn_EV = [(y,x) for (x,y) in connections]

        # Run many message-passing iterations
        for t in range(timesteps):
            # Concat V and E into a single tensor
            VE = torch.cat([V,E], axis=0)
            # Send messages from vertices to edges
            E = self.V2E(x=VE, connections=conn_VE)[total_vertices:,:]
            # Send messages from edges to vertices
            V = self.E2V(x=VE, connections=conn_EV)[:total_vertices,:]

        # Compute votes from edges' embeddings
        votes = self.voter(E)

        # Average votes for each instance
        n_edges_acc = n_edges.copy()
        n_edges_acc.append(n_edges_acc[-1])
        for i in range(1,n_instances+1): n_edges_acc[i] += n_edges_acc[i-1];
        preds = torch.cat([votes[n_edges_acc[i]:n_edges_acc[i+1]].mean() for i
    in range(n_instances)], axis=0)
        # Convert from logits to probabilities
        preds = torch.sigmoid(preds)

        # Return predictions
        return preds
```

7.2 Other Applications of Geometric Deep Learning to the Traveling Salesman Problem

Since our paper was published, there have been other attempts at using models in the graph neural network family to solve variants of the TSP. A notable example is the paper "An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem" (JOSHI; LAURENT; BRESSON, 2019), which uses a technique very similar to ours. The authors also focus on Euclidean instances and also embed both vertices and edges in multidimensional spaces (using graph convolution layers). They additionally train a MLP which computes a probability for each edge, which is architecturally similar to our approach. These probabilities are interpreted differently from ours, however: while our probabilities are averaged in order to produce a prediction for the decision problem, in their approach the probability that each edge will feature in the optimal TSP route is approximated. The same approach was attempted by us in an earlier version of the TSP-GNN model, but was discarded because, as the authors of the new paper argue, "directly converting the probabilistic heat-map to an adjacency matrix representation of the predicted TSP tour $\hat{\pi}$ via an argmax function will generally yield invalid tours with extra or not enough edges in $\hat{\pi}$". However, by starting with a given node and iteratively building a route with beam-search, they were able to train the model to solve the optimization problem, in an unsupervised way (the loss function is defined the length of the reconstructed TSP route).

8 DISCUSSION AND FUTURE WORK

In the recent years, the deep learning community has devoted a growing amount of interest to relational, or graph-structured, learning tasks. This interest is motivated by the necessity of extending traditional DL approaches to non-euclidean domains, in which the topology of problem instances can only be described in terms of graphs. This incipient field of DL, dubbed “geometric deep learning”, has advanced at a vary fast pace in the last two years. Due to the velocity of such developments, the state-of-the-art has advanced past some of the techniques and tools developed for this thesis. Nevertheless, this thesis occupies a relevant position in the chronology of GNN models. Along with NeuroSAT (SELSAM et al., 2018), we were one of the first to show that GNNs can learn to solve $\mathcal{NP}$-Complete problems from data in a supervised way. In particular, we were able to show that GNNs can be trained to solve TSP instances, which are conceptually more complex than CNF formulas as they combine relational and numerical information. This was achieved with a GNN architecture which assigns hyperdimensional embeddings not only to graph vertices but also to edges, which enables one to feed them with their corresponding weights. This allows one to effectively feed an weighted graph to an end-to-end differentiable DL model. We train our model with a distribution of TSP instances defined over random Euclidean fully-connected 2D graphs. For each graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we compute the optimal Hamiltonian tour cost C^* and train the model to solve two decision problem instances: 1) “does graph $\mathcal{G}$ admit a route with cost $< C^-$?” and 2) “does graph $\mathcal{G}$ admit a route with cost $< C^+$?”, where $C^- = 0.98C^*$ and $C^+ = 1.02C^*$. In doing that, we effectively train the model to solve the decision problem within a $-2\%, +2\%$ deviation from the optimal tour cost. Our experiments show that, upon training, the model achieves 80% test accuracy and can generalize to some extent to larger instances and different graph distributions, although with reduced accuracy.

Another key contribution of this thesis is the formalization of a new architecture in the GNN family, named “Typed Graph Networks”. While simpler GNN architectures employ a single type of graph convolution operation to all vertices in the graph and more recent reformulations suggest training convolution layers for vertices, edges and graphs separately, we go a step further and propose partitioning graph vertices into N “types”. This formalization naturally allows one to define models which differentiate between different types of elements in the problem domain (nodes / edges / graphs / cliques / colors / literals / clauses). This formalization is accompanied by a Python / Tensorflow library which enables one to prototype a TGN model in a succint way. As of 2019, our library has since been rivalled by other projects, but it has nevertheless fulfilled an important role and served as the basis for a number of publications before alternative libraries were readily available.

REFERENCES

ALLAMANIS, M.; BROCKSCHMIDT, M.; KHADEMI, M. Learning to represent programs with graphs. **arXiv preprint arXiv:1711.00740**, 2017.

ALLEN-ZHU, Z.; LI, Y.; SONG, Z. A convergence theory for deep learning via over-parameterization. **arXiv preprint arXiv:1811.03962**, 2018.

AN, H.-C.; KLEINBERG, R.; SHMOYS, D. B. Improving christofides' algorithm for the st path tsp. **Journal of the ACM (JACM)**, ACM, v. 62, n. 5, p. 34, 2015.

ATWOOD, J.; TOWSLEY, D. Diffusion-convolutional neural networks. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2016. p. 1993–2001.

AUDEMARD, G.; SIMON, L. On the glucose sat solver. **International Journal on Artificial Intelligence Tools**, World Scientific, v. 27, n. 01, p. 1840001, 2018.

AVELAR, P. H. et al. Multitask learning on graph neural networks-learning multiple graph centrality measures with a unified network. **arXiv preprint arXiv:1809.07695**, 2018.

BAHDANAU, D.; CHO, K.; BENGIO, Y. Neural machine translation by jointly learning to align and translate. **arXiv preprint arXiv:1409.0473**, 2014.

BANDINELLI, N.; BIANCHINI, M.; SCARSELLI, F. Learning long-term dependencies using layered graph neural networks. In: IEEE. **Neural Networks (IJCNN), The 2010 International Joint Conference on**. [S.l.], 2010. p. 1–8.

BATTAGLIA, P. et al. Interaction networks for learning about objects, relations and physics. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2016. p. 4502–4510.

BATTAGLIA, P. W. et al. Relational inductive biases, deep learning, and graph networks. **arXiv preprint arXiv:1806.01261**, 2018.

BEARDWOOD, J.; HALTON, J. H.; HAMMERSLEY, J. M. The shortest path through many points. In: CAMBRIDGE UNIVERSITY PRESS. **Mathematical Proceedings of the Cambridge Philosophical Society**. [S.l.], 1959. v. 55, n. 4, p. 299–327.

BELLO, I. et al. Neural combinatorial optimization with reinforcement learning. **arXiv preprint arXiv:1611.09940**, 2016.

BELLOMARINI, L.; SALLINGER, E.; GOTTLOB, G. The vadalog system: datalog-based reasoning for knowledge graphs. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 11, n. 9, p. 975–987, 2018.

BENGIO, Y.; LODI, A.; PROUVOST, A. Machine learning for combinatorial optimization: a methodological tour d'horizon. **arXiv preprint arXiv:1811.06128**, 2018.

BIANCHINI, M. et al. Recursive neural networks for processing graphs with labelled edges: Theory and applications. **Neural Networks**, Elsevier, v. 18, n. 8, p. 1040–1050, 2005.

BOJCHEVSKI, A. et al. Netgan: Generating graphs via random walks. **arXiv preprint arXiv:1803.00816**, 2018.

BORDES, A. et al. Translating embeddings for modeling multi-relational data. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2013. p. 2787–2795.

BRONSTEIN, M. M. et al. Geometric deep learning: going beyond euclidean data. **IEEE Signal Processing Magazine**, IEEE, v. 34, n. 4, p. 18–42, 2017.

BRUNA, J. et al. Spectral networks and locally connected networks on graphs. **arXiv preprint arXiv:1312.6203**, 2013.

CAO, N. D.; KIPF, T. Molgan: An implicit generative model for small molecular graphs. **arXiv preprint arXiv:1805.11973**, 2018.

CASTELVECCHI, D. Can we open the black box of ai? **Nature News**, v. 538, n. 7623, p. 20, 2016.

CHANG, M. B. et al. A compositional object-based approach to learning physical dynamics. **arXiv preprint arXiv:1612.00341**, 2016.

CHEN, T. Q. et al. Neural ordinary differential equations. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2018. p. 6572–6583.

CHEN, X. et al. Iterative visual reasoning beyond convolutions. **arXiv preprint arXiv:1803.11189**, 2018.

COOK, S. A. The complexity of theorem-proving procedures. In: ACM. **Proceedings of the third annual ACM symposium on Theory of computing**. [S.l.], 1971. p. 151–158.

CUI, Z. et al. High-order graph convolutional recurrent neural network: A deep learning framework for network-scale traffic learning and forecasting. **arXiv preprint arXiv:1802.07007**, 2018.

CYBENKO, G. Approximations by superpositions of a sigmoidal function. **Mathematics of Control, Signals and Systems**, v. 2, p. 183–192, 1989.

DANTZIG, G.; FULKERSON, R.; JOHNSON, S. Solution of a large-scale traveling-salesman problem. **Journal of the operations research society of America**, INFORMS, v. 2, n. 4, p. 393–410, 1954.

DAVIS, M.; PUTNAM, H. A computing procedure for quantification theory. **Journal of the ACM (JACM)**, ACM, v. 7, n. 3, p. 201–215, 1960.

DEFFERRARD, M.; BRESSON, X.; VANDERGHEYNST, P. Convolutional neural networks on graphs with fast localized spectral filtering. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2016. p. 3844–3852.

DENG, J. et al. Imagenet: A large-scale hierarchical image database. In: IEEE. **Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on**. [S.l.], 2009. p. 248–255.

DURAN, A. G.; NIEPERT, M. Learning graph representations with embedding propagation. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 5119–5130.

DUVENAUD, D. K. et al. Convolutional networks on graphs for learning molecular fingerprints. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2015. p. 2224–2232.

FENG, Y. et al. Hypergraph neural networks. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2019. v. 33, p. 3558–3565.

FEY, M.; LENSSEN, J. E. Fast graph representation learning with pytorch geometric. **arXiv preprint arXiv:1903.02428**, 2019.

GARCIA, V.; BRUNA, J. Few-shot learning with graph neural networks. **arXiv preprint arXiv:1711.04043**, 2017.

GENDREAU, M.; LAPORTE, G.; VIGO, D. Heuristics for the traveling salesman problem with pickup and delivery. **Computers & Operations Research**, Elsevier, v. 26, n. 7, p. 699–714, 1999.

GILMER, J. et al. Neural message passing for quantum chemistry. **arXiv preprint arXiv:1704.01212**, 2017.

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. Disponível em: <http://www.jmlr.org/proceedings/papers/v9/glorot10a.html>.

GOODFELLOW, I. et al. Generative adversarial nets. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2014. p. 2672–2680.

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.

GRAVES, A. Generating sequences with recurrent neural networks. **arXiv preprint arXiv:1308.0850**, 2013.

GRAVES, A. et al. A novel connectionist system for unconstrained handwriting recognition. **IEEE transactions on pattern analysis and machine intelligence**, IEEE, v. 31, n. 5, p. 855–868, 2009.

GRAVES, A.; MOHAMED, A.-r.; HINTON, G. Speech recognition with deep recurrent neural networks. In: IEEE. **2013 IEEE international conference on acoustics, speech and signal processing**. [S.l.], 2013. p. 6645–6649.

GROVER, A.; LESKOVEC, J. node2vec: Scalable feature learning for networks. In: ACM. **Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.], 2016. p. 855–864.

GULCEHRE, C. et al. Hyperbolic attention networks. **arXiv preprint arXiv:1805.09786**, 2018.

HAHSLER, M.; HORNIK, K. Tsp-infrastructure for the traveling salesperson problem. **Journal of Statistical Software**, American Statistical Association, v. 23, n. 2, p. 1–21, 2007.

HAMAGUCHI, T. et al. Knowledge transfer for out-of-knowledge-base entities: a graph neural network approach. **arXiv preprint arXiv:1706.05674**, 2017.

HAMRICK, J. B. et al. Relational inductive bias for physical construction in humans and machines. **arXiv preprint arXiv:1806.01203**, 2018.

HAMRICK, J. B. et al. Metacontrol for adaptive imagination-based optimization. **arXiv preprint arXiv:1705.02670**, 2017.

HE, K. et al. Deep residual learning for image recognition. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2016. p. 770–778.

HENAFF, M.; BRUNA, J.; LECUN, Y. Deep convolutional networks on graph-structured data. **arXiv preprint arXiv:1506.05163**, 2015.

HORNIK, K. Approximation capabilities of multilayer feedforward networks. **Neural networks**, Elsevier, v. 4, n. 2, p. 251–257, 1991.

HOSHEN, Y. Vain: Attentional multi-agent predictive modeling. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 2701–2711.

HU, H. et al. Relation networks for object detection. In: **Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2018. v. 2, n. 3.

HUTSON, M. Blog, **AI researchers allege that machine learning is alchemy**. 2018. <https://www.sciencemag.org/news/2018/05/ai-researchers-allege-machine-learning-alchemy>.

JOHNSON, D. D. Learning graphical state transitions. In: **5th International Conference on Learning Representations**. [S.l.: s.n.], 2017.

JOHNSON, J. et al. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In: IEEE. **Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on**. [S.l.], 2017. p. 1988–1997.

JOSHI, C. K.; LAURENT, T.; BRESSON, X. An efficient graph convolutional network technique for the travelling salesman problem. **arXiv preprint arXiv:1906.01227**, 2019.

KARP, R. M. Reducibility among combinatorial problems. In: **Complexity of computer computations**. [S.l.]: Springer, 1972. p. 85–103.

KHALIL, E. et al. Learning combinatorial optimization algorithms over graphs. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 6348–6358.

KIPF, T. et al. Neural relational inference for interacting systems. **arXiv preprint arXiv:1802.04687**, 2018.

KIPF, T. N.; WELLING, M. Semi-supervised classification with graph convolutional networks. **arXiv preprint arXiv:1609.02907**, 2016.

KURZWEIL, R. **The singularity is near**. [S.l.]: Gerald Duckworth & Co, 2010.

LEMOS, H. et al. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. **arXiv preprint arXiv:1903.04598**, 2019.

LEVIN, L. A. Universal sequential search problems. **Problemy Peredachi Informatsii**, Russian Academy of Sciences, Branch of Informatics, Computer Equipment and . . . , v. 9, n. 3, p. 115–116, 1973.

LI, Y. et al. Gated graph sequence neural networks. **arXiv preprint arXiv:1511.05493**, 2015.

LI, Y. et al. Learning deep generative models of graphs. **arXiv preprint arXiv:1803.03324**, 2018.

LI, Y. et al. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. 2018.

LU, L. et al. Ranking attack graphs with graph neural networks. In: SPRINGER. **International Conference on Information Security Practice and Experience**. [S.l.], 2009. p. 345–359.

MARCUS, G. Deep learning: A critical appraisal. **arXiv preprint arXiv:1801.00631**, 2018.

MARCUS, G. Innateness, alphazero, and artificial intelligence. **arXiv preprint arXiv:1801.05667**, 2018.

MARCUS, G. **Why Robot Brains Need Symbols**. 2018. [Online; accessed 19-December-2018]. Disponível em: <http://nautil.us/issue/67/reboot/why-robot-brains-need-symbols>.

MASSA, V. D. et al. A comparison between recursive neural networks and graph neural networks. In: IEEE. **Neural Networks, 2006. IJCNN'06. International Joint Conference on**. [S.l.], 2006. p. 778–785.

MICHELI, A. Neural network for graphs: A contextual constructive approach. **IEEE Transactions on Neural Networks**, IEEE, v. 20, n. 3, p. 498–511, 2009.

MNIH, V. et al. Playing atari with deep reinforcement learning. **arXiv preprint arXiv:1312.5602**, 2013.

MONFARDINI, G. et al. Graph neural networks for object localization. In: IOS PRESS. **Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29–September 1, 2006, Riva del Garda, Italy**. [S.l.], 2006. p. 665–669.

MORAVČÍK, M. et al. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. **Science**, American Association for the Advancement of Science, v. 356, n. 6337, p. 508–513, 2017.

MURATORE, D. et al. Sentence extraction by graph neural networks. In: SPRINGER. **International Conference on Artificial Neural Networks**. [S.l.], 2010. p. 237–246.

NIEPERT, M.; AHMED, M.; KUTZKOV, K. Learning convolutional neural networks for graphs. In: **International conference on machine learning**. [S.l.: s.n.], 2016. p. 2014–2023.

NOI, L. D. et al. Web spam detection by probability mapping graphsoms and graph neural networks. In: SPRINGER. **International Conference on Artificial Neural Networks**. [S.l.], 2010. p. 372–381.

NOWAK, A. et al. A note on learning algorithms for quadratic assignment with graph neural networks. **arXiv preprint arXiv:1706.07450**, 2017.

OÑORO-RUBIO, D. et al. Representation learning for visual-relational knowledge graphs. **arXiv preprint arXiv:1709.02314**, 2017.

PASCANU, R. et al. Learning model-based planning from scratch. **arXiv preprint arXiv:1707.06170**, 2017.

PRATES, M.; AVELAR, P.; LAMB, L. C. On quantifying and understanding the role of ethics in ai research: A historical account of flagship conferences and journals. **arXiv preprint arXiv:1809.08328**, 2018.

PRATES, M.; LAMB, L. Problem solving at the edge of chaos: Entropy, puzzles and the sudoku freezing transition. In: IEEE. **2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)**. [S.l.], 2018. p. 686–693.

PRATES, M.; LAMB, L. Problem solving at the edge of chaos: Entropy, puzzles and the sudoku freezing transition. **arXiv preprint arXiv:1810.03742**, 2018.

PRATES, M. O.; AVELAR, P. H.; LAMB, L. C. Assessing gender bias in machine translation: a case study with google translate. **Neural Computing and Applications**, Springer, p. 1–19.

PRATES, M. O. et al. Learning to solve np-complete problems-a graph neural network for the decision tsp. **arXiv preprint arXiv:1809.02721**, 2018.

PRATES, M. O. et al. Typed graph networks. **arXiv preprint arXiv:1901.07984**, 2019.

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: **GCAI-2018, 4th Global Conference on Artificial Intelligence, Luxembourg, September 18-21, 2018**. [s.n.], 2018. p. 188–201. Disponível em: <http://www.easychair.org/publications/paper/Z7D4>.

PUCCI, A. et al. Applications of graph neural networks to large-scale recommender systems some results. In: **Proceedings of the International Multiconference on Computer Science and Information Technology**. [S.l.: s.n.], 2006. v. 1, p. 189–195.

QUEK, A. et al. Structural image classification with graph neural networks. In: IEEE. **Digital Image Computing Techniques and Applications (DICTA), 2011 International Conference on**. [S.l.], 2011. p. 416–421.

RAPOSO, D. et al. Discovering objects and their relations from entangled scene representations. **arXiv preprint arXiv:1702.05068**, 2017.

SABOUR, S.; FROSST, N.; HINTON, G. E. Dynamic routing between capsules. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 3856–3866.

SANCHEZ-GONZALEZ, A. et al. Graph networks as learnable physics engines for inference and control. **arXiv preprint arXiv:1806.01242**, 2018.

SANTORO, A. et al. A simple neural network module for relational reasoning. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2017. p. 4967–4976.

SCARSELLI, F. et al. Computational capabilities of graph neural networks. **IEEE Transactions on Neural Networks**, IEEE, v. 20, n. 1, p. 81–102, 2009.

SCARSELLI, F. et al. The graph neural network model. **IEEE Transactions on Neural Networks**, IEEE, v. 20, n. 1, p. 61–80, 2009.

SCARSELLI, F. et al. Graph neural networks for ranking web pages. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence**. [S.l.], 2005. p. 666–672.

SELSAM, D. et al. Learning a sat solver from single-bit supervision. **arXiv preprint arXiv:1802.03685**, 2018.

SHAW, P.; USZKOREIT, J.; VASWANI, A. Self-attention with relative position representations. **arXiv preprint arXiv:1803.02155**, 2018.

SILVER, D. et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. **Science**, American Association for the Advancement of Science, v. 362, n. 6419, p. 1140–1144, 2018.

SIMONYAN, K.; VEDALDI, A.; ZISSERMAN, A. Deep inside convolutional networks: Visualising image classification models and saliency maps. **arXiv preprint arXiv:1312.6034**, 2013.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. **arXiv preprint arXiv:1409.1556**, 2014.

STEENKISTE, S. van et al. Relational neural expectation maximization: Unsupervised discovery of objects and their interactions. **arXiv preprint arXiv:1802.10353**, 2018.

TANG, J. et al. Line: Large-scale information network embedding. In: INTERNATIONAL WORLD WIDE WEB CONFERENCES STEERING COMMITTEE. **Proceedings of the 24th International Conference on World Wide Web**. [S.l.], 2015. p. 1067–1077.

TOYER, S. et al. Action schema networks: Generalised policies with deep learning. **arXiv preprint arXiv:1709.04271**, 2017.

UWENTS, W. et al. Neural networks for relational learning: an experimental comparison. **Machine Learning**, Springer, v. 82, n. 3, p. 315–349, 2011.

VASWANI, A. et al. Attention is all you need. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 5998–6008.

VELIČKOVIĆ, P. et al. Graph attention networks. **arXiv preprint arXiv:1710.10903**, 2017.

VELICKOVIC, P. et al. Graph attention networks. **arXiv preprint arXiv:1710.10903**, v. 1, n. 2, 2017.

WANG, T. et al. Nervenet: Learning structured policy with graph neural networks. 2018.

WANG, X. et al. Non-local neural networks. In: **The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2018. v. 1, n. 3, p. 4.

WANG, Y. et al. Dynamic graph cnn for learning on point clouds. **arXiv preprint arXiv:1801.07829**, 2018.

WATTERS, N. et al. Visual interaction networks: Learning a physics simulator from video. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 4539–4547.

WESTON, J. et al. Towards ai-complete question answering: A set of prerequisite toy tasks. **CoRR**, abs/1502.05698, 2015. Disponível em: <http://arxiv.org/abs/1502.05698>.

WU, J. et al. Learning to see physics via visual de-animation. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 153–164.

XIA, X.; XU, C.; NAN, B. Inception-v3 for flower classification. In: IEEE. **Image, Vision and Computing (ICIVC), 2017 2nd International Conference on**. [S.l.], 2017. p. 783–787.

YONG, S. L. et al. Document mining using graph neural network. In: SPRINGER. **International Workshop of the Initiative for the Evaluation of XML Retrieval**. [S.l.], 2006. p. 458–472.

YOON, K. et al. Inference in probabilistic graphical models by graph neural networks. **arXiv preprint arXiv:1803.07710**, 2018.

YOU, J. et al. Graphrnn: A deep generative model for graphs. **arXiv preprint arXiv:1802.08773**, 2018.

ZAMBALDI, V. et al. Relational deep reinforcement learning. **arXiv preprint arXiv:1806.01830**, 2018.

ZILIO, F.; PRATES, M.; LAMB, L. Neural networks models for analyzing magic: the gathering cards. In: SPRINGER. **International Conference on Neural Information Processing**. [S.l.], 2018. p. 227–239.

APPENDIX A — TYPED GRAPH NETWORKS LIBRARY CODE

A.1 tgn.py

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

class TGN(object):
  def __init__(
    self,
    var,
    mat,
    msg,
    loop,
    MLP_depth = 3,
    MLP_weight_initializer = tf.contrib.layers.xavier_initializer,
    MLP_bias_initializer = tf.zeros_initializer,
    RNN_cell = tf.contrib.rnn.LayerNormBasicLSTMCell,
    Cell_activation = tf.nn.relu,
    Msg_activation = tf.nn.relu,
    Msg_last_activation = None,
    float_dtype = tf.float32,
    name = 'TGN'
  ):
    """
    Receives three dictionaries: var, mat and msg.

    * var is a dictionary from variable names to embedding sizes.
      That is: an entry var["V1"] = 10 means that the variable "V1" will have an embedding size of 10.

    * mat is a dictionary from matrix names to variable pairs.
      That is: an entry mat["M"] = ("V1","V2") means that the matrix "M" can be used to mask messages from "V1" to "V2".

    * msg is a dictionary from function names to variable pairs.
      That is: an entry msg["cast"] = ("V1","V2") means that one can apply "cast" to convert messages from "V1" to "V2".

    * loop is a dictionary from variable names to lists of dictionaries:
      {
        "mat": the matrix name which will be used,
        "transpose?": if true then the matrix M will be transposed,
        "fun": transfer function (python function built using tensorflow operations,
        "msg": message name,
        "var": variable name
      }
      If "mat" is None, it will be the identity matrix,
      If "transpose?" is None, it will default to false,
      if "fun" is None, no function will be applied,
      If "msg" is false, no message conversion function will be applied,
      If "var" is false, then [1] will be supplied as a surrogate.

      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:
        V2 <- tf.append( [ f(V2), transpose(T) x cast(V1) ] )
    """
    self.var, self.mat, self.msg, self.loop, self.name = var, mat, msg, loop, name

    self.MLP_depth = MLP_depth
    self.MLP_weight_initializer = MLP_weight_initializer
    self.MLP_bias_initializer = MLP_bias_initializer
    self.RNN_cell = RNN_cell
    self.Cell_activation = Cell_activation
    self.Msg_activation = Msg_activation
    self.Msg_last_activation  = Msg_last_activation
    self.float_dtype = float_dtype

    # Check model for inconsistencies
    self.check_model()

    # Initialize the parameters
    with tf.variable_scope(self.name):
      with tf.variable_scope('parameters'):
        self._init_parameters()
      #end parameter scope
    #end TGN scope
  #end __init__

  def check_model(self):
    # Procedure to check model for inconsistencies
    for v in self.var:
      if v not in self.loop:
        raise Warning('Variable {v} is not updated anywhere! Consider removing it from the model'.format(v=v))
      #end if
    #end for

    for v in self.loop:
      if v not in self.var:
        raise Exception('Updating variable {v}, which has not been declared!'.format(v=v))
      #end if
    #end for

    for mat, (v1,v2) in self.mat.items():
      if v1 not in self.var:
        raise Exception('Matrix {mat} definition depends on undeclared variable {v}'.format(mat=mat, v=v1))
      #end if
      if v2 not in self.var and type(v2) is not int:
        raise Exception('Matrix␣{mat}␣definition␣depends␣on␣undeclared␣variable␣{v}'.format(mat=mat, v=v2))
      #end if
    #end for

    for msg, (v1,v2) in self.msg.items():
      if v1 not in self.var:
        raise Exception('Message␣{msg}␣maps␣from␣undeclared␣variable␣{v}'.format(msg=msg, v=v1))
      #end if
      if v2 not in self.var:
        raise Exception('Message␣{msg}␣maps␣to␣undeclared␣variable␣{v}'.format(msg=msg, v=v2))
      #end if
    #end for
  #end check_model

  def _init_parameters(self):
    # Init LSTM cells
    self._RNN_cells = {
      v: self.RNN_cell(
       d,
       activation = self.Cell_activation
      ) for (v,d) in self.var.items()
    }
    # Init message-computing MLPs
    self._msg_MLPs = {
      msg: Mlp(
        layer_sizes          = [ self.var[vin] for _ in range( self.MLP_depth ) ],
        output_size          = self.var[vout],
        activations          = [ self.Msg_activation for _ in range( self.MLP_depth ) ],
        output_activation    = self.Msg_last_activation,
        kernel_initializer   = self.MLP_weight_initializer(),
        bias_initializer     = self.MLP_weight_initializer(),
        name                 = msg,
        name_internal_layers = True
      ) for msg, (vin,vout) in self.msg.items()
    }
  #end _init_parameters

  def __call__( self, adjacency_matrices, initial_embeddings, time_steps, LSTM_initial_states = {} ):
    with tf.variable_scope(self.name):
      with tf.variable_scope( "assertions" ):
        assertions = self.check_run( adjacency_matrices, initial_embeddings, time_steps, LSTM_initial_states )
      #end assertion variable scope
      with tf.control_dependencies( assertions ):
```

```
    states = {}
    for v, init in initial_embeddings.items():
      h0 = init
      c0 = tf.zeros_like(h0, dtype=self.float_dtype) if v not in LSTM_initial_states else LSTM_initial_states[v]
      states[v] = tf.contrib.rnn.LSTMStateTuple(h=h0, c=c0)
    #end

    # Build while loop body function
    def while_body( t, states ):
      new_states = {}
      for v in self.var:
        inputs = []
        for update in self.loop[v]:
          if 'var' in update:
            y = states[update['var']].h
            if 'fun' in update:
              y = update['fun'](y)
            #end if
            if 'msg' in update:
              y = self._msg_MLPs[update['msg']](y)
            #end if
            if 'mat' in update:
              y = tf.matmul(
                adjacency_matrices[update['mat']],
                y,
                adjoint_a = update['transpose?'] if 'transpose?' in update else False
              )
            #end if
            inputs.append( y )
          else:
            inputs.append( adjacency_matrices[update['mat']] )
          #end if var in update
        #end for update in loop
        inputs = tf.concat( inputs, axis = 1 )
        with tf.variable_scope( '{v}_cell'.format( v = v ) ):
          _, new_states[v] = self._RNN_cells[v]( inputs = inputs, state = states[v] )
        #end cell scope
      #end for v in var
      return (t+1), new_states
    #end while_body

    _, last_states = tf.while_loop(
      lambda t, states: tf.less( t, time_steps ),
      while_body,
      [0,states]
    )
  #end assertions
#end Graph scope
return last_states
```

```
  #end __call__

  def check_run( self, adjacency_matrices, initial_embeddings, time_steps, LSTM_initial_states ):
    assertions = []
    # Procedure to check model for inconsistencies
    num_vars = {}
    for v, d in self.var.items():
      init_shape = tf.shape( initial_embeddings[v] )
      num_vars[v] = init_shape[0]
      assertions.append(
        tf.assert_equal(
          init_shape[1],
          d,
          data = [ init_shape[1] ],
          message = "Initial␣embedding␣of␣variable␣{v}␣doesn't␣have␣the␣same␣dimensionality␣{d}␣as␣declared".format(
            v = v,
            d = d
          )
        )
      )
      if v in LSTM_initial_states:
        lstm_init_shape = tf.shape( LSTM_initial_states[v] )
        assertions.append(
          tf.assert_equal(
            lstm_init_shape[1],
            d,
            data = [ lstm_init_shape[1] ],
            message = "Initial␣hidden␣state␣of␣variable␣{v}'s␣LSTM␣doesn't␣have␣the␣same␣dimensionality␣{d}␣as␣declared".format(
              v = v,
              d = d
            )
          )
        )

        assertions.append(
          tf.assert_equal(
            lstm_init_shape,
            init_shape,
            data = [ init_shape, lstm_init_shape ],
            message = "Initial␣embeddings␣of␣variable␣{v}␣don't␣have␣the␣same␣shape␣as␣the␣its␣LSTM's␣initial␣hidden␣state".format(
              v = v,
              d = d
            )
          )
        )
      #end if
    #end for v

    for mat, (v1,v2) in self.mat.items():
      mat_shape = tf.shape( adjacency_matrices[mat] )
      assertions.append(
        tf.assert_equal(
          mat_shape[0],
          num_vars[v1],
          data = [ mat_shape[0], num_vars[v1] ],
          message = "Matrix␣{m}␣doesn't␣have␣the␣same␣number␣of␣nodes␣as␣the␣initial␣embeddings␣of␣its␣variable␣{v}".format(
            v = v1,
            m = mat
          )
        )
      )
      if type(v2) is int:
        assertions.append(
          tf.assert_equal(
            mat_shape[1],
            v2,
            data = [ mat_shape[1], v2 ],
            message = "Matrix␣{m}␣doesn't␣have␣the␣same␣dimensionality␣{d}␣on␣the␣second␣variable␣as␣declared".format(
              m = mat,
              d = v2
            )
          )
        )
      else:
        assertions.append(
          tf.assert_equal(
            mat_shape[1],
            num_vars[v2],
            data = [ mat_shape[1], num_vars[v2] ],
            message = "Matrix␣{m}␣doesn't␣have␣the␣same␣number␣of␣nodes␣as␣the␣initial␣embeddings␣of␣its␣variable␣{v}".format(
              v = v2,
              m = mat
            )
          )
        )
      #end if-else
    #end for mat, (v1,v2)
    return assertions
  #end check_run
#end TGN
```

A.2 mlp.py

```
import tensorflow as tf
```

```

class Mlp(object):
 def __init__(
  self,
  layer_sizes,
  output_size = None,
  activations = None,
  output_activation = None,
  use_bias = True,
  kernel_initializer = None,
  bias_initializer = tf.zeros_initializer(),
  kernel_regularizer = None,
  bias_regularizer = None,
  activity_regularizer = None,
  kernel_constraint = None,
  bias_constraint = None,
  trainable = True,
  name = None,
  name_internal_layers = True
 ):
  """Stacks len(layer_sizes) dense layers on top of each other, with an additional layer with output_size neurons, if specified."""
  self.layers = []
  internal_name = None
  # If object isn't a list, assume it is a single value that will be repeated for all values
  if not isinstance( activations, list ):
   activations = [ activations for _ in layer_sizes ]
  #end if
  # If there is one specifically for the output, add it to the list of layers to be built
  if output_size is not None:
   layer_sizes = layer_sizes + [output_size]
   activations = activations + [output_activation]
  #end if
  for i, params in enumerate( zip( layer_sizes, activations ) ):
   size, activation = params
   if name_internal_layers:
    internal_name = name + "_MLP_layer_{}".format( i + 1 )
   #end if
   new_layer = tf.layers.Dense(
    size,
    activation = activation,
    use_bias = use_bias,
    kernel_initializer = kernel_initializer,
    bias_initializer = bias_initializer,
    kernel_regularizer = kernel_regularizer,
    bias_regularizer = bias_regularizer,
    activity_regularizer = activity_regularizer,
    kernel_constraint = kernel_constraint,
    bias_constraint = bias_constraint,
    trainable = trainable,
    name = internal_name
   )
   self.layers.append( new_layer )
  #end for
 #end __init__

 def __call__( self, inputs, *args, **kwargs ):
  outputs = [ inputs ]
  for layer in self.layers:
   outputs.append( layer( outputs[-1] ) )
  #end for
  return outputs[-1]
 #end __call__
#end Mlp
```