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

HENRIQUE LEMOS DOS SANTOS

Solving the decision version of the Graph Coloring Problem: a neural-symbolic approach using graph neural networks.

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

Advisor: Prof. Dr. Luis da Cunha Lamb

Porto Alegre
Fevereiro 2020

CIP — CATALOGING-IN-PUBLICATION

dos Santos, Henrique Lemos

Solving the decision version of the Graph Coloring Problem: a neural-symbolic approach using graph neural networks. / Henrique Lemos dos Santos. – Porto Alegre: PPGC da UFRGS, 2020.

91 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Luis da Cunha Lamb.

1. Deep neural networks. 2. Recurrent neural networks. 3. Graph neural networks. 4. Graphs. 5. Graph coloring. 6. Neural-symbolic computation. I. Lamb, Luis da Cunha. II. Título.

“Above all, don’t lie to yourself.
The man who lies to himself and listens to his own lie comes to a point
that he cannot distinguish the truth within him,
or around him, and so loses all respect for himself and for others.
And having no respect he ceases to love.”

— FYODOR DOSTOEVSKY, THE BROTHERS KARAMAZOV

ACKNOWLEDGEMENTS

This work was partially funded by CNPQ, which I thank for the financial support. I also thank NVIDIA for the GPU granted to our laboratory. I'd like to thank my lab colleagues Marcelo de Oliveira Rosa Prates and Pedro Henrique da Costa Avelar for our friendship, joint work and helpful discussions. Both of them were responsible for the foundational aspects of this dissertation, their admirable knowledge and dedication towards understanding very deep aspects of Graph Neural Networks not only inspired me but also made this dissertation possible. Also, I'd like to thank Anderson Rocha Tavares for his help proof-reading this dissertation and final discussions about this work. I must also thank my advisor Prof. Luis Lamb for his support, 24/7 availability and his inspiring love for science in general. Finally, an enormous part of this work I owe to the unwavering support of my parents, João Carlos Soares dos Santos and Lizete Lemos dos Santos, and for them no acknowledgment would ever be enough.

ABSTRACT

Deep learning (DL) has consistently pushed the state-of-the-art in many fields over the last years. Still there is, however, a lack of understanding on how symbolic and relational problems can benefit from DL architectures. The most promising path towards this long-desired integration comprises deep learning architectures whose parameter sharing strategy is based over graphs and thus can be trained to learn complex properties of relational data. Several NP-Complete problems, such as the boolean satisfiability problem and the traveling salesperson problem, present such properties. In both cases, a meta-model called Graph Neural Network (GNN) can be directly fed with the graph representation of the problem and learn to produce a binary answer at hand. In this dissertation, we are specifically concerned with the application of a GNN model to tackle the graph coloring problem: our proposed model leverages the specific features of such problem by adding internal representations of vertices and colors to the GNN kernel and by performing message-passing iterations over such representations. In this sense, our model's architecture is able to reflect the relational structure of the original problem, with no need of polynomial time reductions, while it still employs parameter sharing over the graph vertices and colors. We also show how to train such model upon very hard instances, which were generated in an adversarial fashion: we generate pairs of instances comprising graphs that are on the verge of satisfiability – a positive and a negative-labeled instance that only differ by a single edge, such edge makes the second instance unsatisfiable given a fixed number of colors C. We were able to obtain 83% accuracy during training and to show that such model is able to generalize, to some extent, its performance to unseen instances coming from different distributions and sizes. We show that such performance defeats two heuristics and an allegedly generalist neural-symbolic approach. Finally, we explore the internal memory of our model and find evidence of how its reasoning is built upon its internal states (vertex and color representations). In summary, our results strongly suggests that GNNs are, indeed, powerful to tackle combinatorial problems but their performance can be largely enhanced when all problem's features are integrated within the GNN neural architecture and no problem translation is required.

Keywords: Deep neural networks. recurrent neural networks. graph neural networks. graphs. graph coloring. neural-symbolic computation.

Resolvendo a versão de decisão do problema de coloração de grafos: uma abordagem neuro-simbólica usando redes grafo-neurais.

RESUMO

Técnicas baseadas em aprendizado profundo têm recorrentemente atingido desempenho de estado-da-arte em diversas áreas ao longo dos últimos anos. Ainda há, no entanto, uma certa falta de compreensão em como problemas simbólicos e relacionais podem se beneficiar de modelos cuja arquitetura é baseada em aprendizado profundo. O caminho mais promissor para essa tão desejada integração consiste em arquiteturas neurais cuja propriedade de compartilhamento de parâmetros baseia-se em grafos e, dessa forma, podem ser treinadas para aprender características complexas de dados relacionais. Diversos problemas NP-Completos, tais como satisfatibilidade booleana e problema do caixeiro viajante, apresentam esse tipo de característica. Em ambos casos, um metamodelo chamado *Graph Neural Network* (GNN) pode trabalhar diretamente com entradas em formato de grafos, que representam uma instância do problema, e aprender a produzir uma resposta binária para o problema em questão. Nessa dissertação, estamos particularmente focados em aplicar um modelo de GNN ao problema da coloração de grafos: o modelo que propomos se aproveita de propriedades específicas desse problema ao contemplar tanto vértices quanto cores com representações internas na sua arquitetura e ao fazer com que tais representações passem por diversas etapas de troca de mensagens. Nesse sentido, a arquitetura que propomos é capaz de refletir a estrutura relacional do problema original, sem necessidade de uma redução em tempos polinomial para outro problema, enquanto ainda emprega uma estratégia de compartilhamento de parâmetros em função de vértices e cores. Nós também demonstramos como treinar tal modelo com instâncias muito difíceis, geradas de uma maneira adversarial: nós geramos pares de instâncias que são grafos no limite da satisfatibilidade – uma instância positiva e outra negativa que diferem apenas por uma única aresta, tal aresta faz com que a segunda instância não seja colorável por um dado número de cores C, enquanto a primeira permanece sendo minimamente colorável com C. Obtivemos uma acurácia de 83% durante treinamento e verificamos que nosso modelo é capaz de generalizar, até certo ponto, esse desempenho para instâncias de teste – não-vistas durante treinamento e que foram amostradas de diferentes distribuições. Nós mostramos que esse desempenho superou o desempenho de duas heurísticas e o desempenho de uma suposta abordagem neuro-simbólica generalista. Por fim, nós exploramos a memória

interna do nosso modelos e encontramos evidências de como o seu raciocínio é construído em volta dos valores de representação de vértices e cores. Em suma, nossos resultados sugerem fortemente que GNNs são, de fato, ferramentas poderosas para resolver problemas combinatoriais mas que seu aprendizado pode ser amplamente melhorado quando as propriedades de um problema são totalmente agregadas à arquitetura neural e nenhuma conversão de problema é feita.

Palavras-chave: redes neurais profundas, redes neurais recorrentes, redes grafo-neurais, grafos, coloração de grafos, computação neuro-simbólica.

LIST OF ABBREVIATIONS AND ACRONYMS

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

LIST OF SYMBOLS

$\mathbb{R}$	Set of Real Numbers
$\mathbb{B}$	Set of Boolean Numbers ($\mathbb{B} = \{0, 1\}$)
σ	Sigmoid function
$\mathbf{A}, \mathbf{B}', \mathbf{C_k}, \ldots$	Tensors
$\mathbf{A}^\intercal$	Transpose of matrix $\mathbf{A}$
$\mathbf{A}^{(t)}$	Tensor $\mathbf{A}$ at the t-th iteration of an algorithm
$\mathbf{A} \otimes \mathbf{B}$	Matrix multiplication between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \odot \mathbf{B}$	Element-wise multiplication between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \oplus \mathbf{B}$	Element-wise sum between tensor $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \frown \mathbf{B}$	Concatenation of tensor $\mathbf{A}$ with $\mathbf{B}$ (if not specified, concatenation over the last axis).
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$	Sets
$\lvert\mathcal{A}\rvert$	Number of elements in set $\mathcal{A}$
$\mathcal{N}(0,\, 1)$	Normal distribution with mean $= 0$ and variance $= 1$
$\mathcal{U}(0,\, 1)$	Continuous uniform distribution between 0 and 1
$\mathcal{U}\{0,\, 1\}$	Discrete uniform distribution between 0 and 1

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Recently, after all the state-of-the-art performances achieved by deep learning (DL) models in a wide range of tasks, there has been a prominent discussion on what is the future for machine learning (ML) and artificial intelligence (AI) in general. While many researchers exposed their fear about an imminent new AI winter (NIELD, 2019; SHEAD, 2020), several others (both from academia and industries) shed light on a paradigm which has the potential to drive new AI breakthroughs and has not yet been fully explored in the last years: neural-symbolic computing (GARCEZ et al., 2019; RAGHAVAN, 2019; SMOLENSKY, 2019).

Traditional DL techniques and architectures such as Multi-Layer Perceptrons (MLPs), Convolutional Networks (CNNs), Recurrent Neural Networks (RNNs) and so on have been consistently used to overcome computer vision (KRIZHEVSKY; SUTSKEVER; HINTON, 2012; LI et al., 2015) and natural language processing (CHO et al., 2014a; BAHDANAU; CHO; BENGIO, 2014) tasks. Also, in the last years, there have been substantial research efforts to alleviate some of the difficulties faced by these seminal models: Residual Networks (ResNets) (HE et al., 2016) ease the training of very deep convolutional networks, Capsule Networks (CapsNets) (HINTON; SABOUR; FROSST, 2018) enables spatial hierarchy in image classification (an important drawback of traditional CNNs) and Long Short-Term Memories (LSTMs) (HOCHREITER; SCHMIDHUBER, 1997) and Gated Recurrent Units (GRUs) (CHO et al., 2014b) avoid long-term dependency problem and vanishing gradient on RNNs, not to mention the recent successes on machine translation and language models achieved by attention-based mechanisms such as Transformers (VASWANI et al., 2017; DEVLIN et al., 2019; RADFORD et al., 2019). Yet, there is also a growing interest on explainability capabilities which are usually not provided by such black-box DL techniques. However, while DL techniques are particularly efficient for "fast thinking" (despite being highly data-driven), symbolic AI can in turn provide reliable and explainable solutions via "slow thinking" (despite usually relying on hard-coded rules). On top of these arguments, one may also understand that an intelligent machine should not only be capable of recognizing patterns but also reasoning about what was learned, as put forward by Turing Award Leslie Valiant.

In a nutshell, combining connectionist and symbolic techniques has the potential to tackle some of the challenges faced by these paradigms when they are used alone, such as lack of reasoning and hard-coded knowledge, respectively (BADER; HITZLER, 2005;

GARCEZ; LAMB; GABBAY, 2009; KHARDON; ROTH, 1999). One way of integrating DL and symbolic reasoning is to apply DL models to combinatorial problems: this set of problems tend to have a complex mathematical structure – usually represented by a graph – but, in most cases, there are exact solvers available for them, allowing one to produce a huge amount of labeled instances. When jointly using combinatorial optimization and DL techniques, one may expect two main enhancements to the learning process: (1) divide-and-conquer; as the model would be able to dissect the symbolic problem into smaller learning tasks and (2) generalized curve-fitting; as the model would explore the space of decisions searching for the best performing behavior (BENGIO; LODI; PROUVOST, 2018). While (1) is due to the natural combinatorial optimization structure, (2) is the main principle of all machine learning strategies. In this sense, the neural-symbolic computing paradigm arises as a promising framework towards reasoning and explainability and as a key path forward for AI research from now on (BATTAGLIA et al., 2018).

One way of incorporating the relational structure of a combinatorial problem into a neural model is to ensure permutation invariance by letting adjacent elements of the problem communicate with themselves through neural modules subject to parameter sharing. That is, the problem *prints* its graph representation onto the configuration of the neural modules. These neural modules are primarily accountable for computing the messages sent among the problem's elements and for updating the internal representation of each of these elements. Moreover, one may define not only edge and vertex-level attributes/representations but also global-level attributes. The family of models that makes use of this message-passing algorithm includes message-passing neural networks (GILMER et al., 2017), recurrent relational networks (PALM; PAQUET; WINTHER, 2017), graph networks (BATTAGLIA et al., 2018) and the pioneer model: graph neural network (GNN) (GORI; MONFARDINI; SCARSELLI, 2005).

GNNs have been proven useful to tackle other combinatorial problems such as the NP-Complete boolean satisfiability problem (SAT) (SELSAM et al., 2019) and the decision version of the Traveling Salesperson Problem (TSP) (PRATES et al., 2019a). In the first case, Selsam et al. (2019) translated randomly generated formulas in conjunctive normal form (CNF) into a graph structure: literals are connected to clauses to which they pertain, clauses are connected to literals they contain and literals are also connected to their complementary variants. Using just one bit as a supervision (the satisfiability of the given CNF), the NeuroSAT model proposed by them is able not only to achieve a high accuracy (85%) upon testing but it also demonstrates two well-desired reasoning characteristics

for a ML model: NeuroSAT architecture is able to generalize its performance for a increasing number of timesteps (or number of iterations of message-passing), suggesting that the model learned an algorithm in order to solve the problem; moreover, even though NeuroSAT was trained only as a binary predictor, the authors were able to decode valid assignments from its memory. Selsam et al. (2019) also demonstrated that NeuroSAT was able to decode valid assignments for several other combinatorial problems reduced to SAT, such as vertex coloring, clique detection, dominating set and vertex cover. Further experiments conducted by Prates et al. (2019a) also verified the possibility of incorporating numeric information to the combinatorial problem tackled by the GNN: the edges of a TSP instance/graph are framed as nodes and embedded with their weights, then the GNN kernel allows them to communicate with their neighboring nodes and vice-versa.

In this work, we introduce a GNN-based model[1] to tackle the decision version of the graph coloring problem (GCP), with no need of prior reductions. Our model mainly relies on GNN's power of coping with several types of edges as we approached the GCP problem by using two different types of vertices. By designing a GNN model to solve an important combinatorial problem (with applications on flow management (BARNIER; BRISSET, 2004), job scheduling (THEVENIN; ZUFFEREY; POTVIN, 2018), register allocation (CHEN et al., 2018) and others), we hope we can foster the adoption and further research on GNN-like models which in turn integrate deep learning and combinatorial optimization. We believe that, from an AI perspective, our work provides useful insights on how neural modules reason over symbolic problems, and also on how their hidden states or embeddings can be further interpreted.

1.1 Research Questions and Hypotheses

Bridging the gap between symbolic reasoning and DL is still a sparsely explored research field (BATTAGLIA et al., 2018). GNNs, end-to-end differentiable neural networks with a distinguished ability to cope with relational data, are definitely the most promising framework to accomplish this integration of two long-departed AI branches. Selsam et al. (2019) have already shown that GNNs are able to tackle a very general NP-Complete problem (boolean satisfiability) and Prates et al. (2019a) have shown that a GNN model is also able to cope with numeric information. However, following the experiments conducted

[1] Our implementation of such model is available at: <https://machine-reasoning-ufrgs.github.io/GNN-GCP/>

in the NeuroSAT paper, one may argue that the NeuroSAT is the ultimate GNN model, as every other problem in NP can be reduced to SAT in polynomial time and thus solved by the NeuroSAT itself. Our main hypotheses is that, by reducing some other problem (such as the GCP) to SAT and feeding it to the NeuroSAT, one may not leverage the full capabilities of symbolic reasoning presented by a GNN-like model. In this sense, we would like to answer the following questions during this dissertation:

1. *Can a Graph Neural Network solve the decision version of the Graph Coloring problem only from the network structure and a number of colors which is close to the chromatic number?*

2. *Does such GNN generalize its performance to larger/smaller number of colors?*

3. *Does the strategy employed by NeuroSAT to decode assignments from GNN internal states work under this new architecture?*

4. *Can this specialized GNN learn meaningful and interpretable internal states?*

Given the related literature, it was hypothesized that the Research Question 1 may be answered positively, at least to some extent, which may be not yet comparable to state-of-the-art solvers, as NeuroSAT's performance also was not. The answer to Research Question 2 was thought to be positive as well as we believe that training a GNN architecture in the verge of satisfiability also improves its performance on easier instances, as shown by (PRATES et al., 2019a) w.r.t. TSP and route costs. Also following NeuroSAT, we would expect Research Question 3 and 4 to be answered positively as the GNN refines their internal embeddings in a fashion that allows for an algorithm to be executed upon them, which indicates that such model is capable of reasoning over a symbolic structure in order to produce an answer to the problem at hand.

1.2 Contributions

During the Master's programme duration the author participated in several research endeavors, some of which are part of this dissertation. The abstracts of the scientific papers produced during these researches are listed below along with a brief description of contributions.

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

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

Joint first author, with Pedro Henrique da Costa Avelar and Marcelo de Oliveira Rosa Prates, on the submission to the 28th International Conference on Artificial Neural Networks (Qualis B1, 2013-2016), accepted as poster, presented by Pedro Henrique da Costa Avelar. Pre-print available (AVELAR et al., 2018).

The author's contribution on this paper consisted in redesigning the approximation model, which provided the initial positive results, gathering and parsing some of the real instances, elaborating the theoretical review on the centrality measures as well as on the graph distributions, generating and plotting the PCA visualizations for the embeddings and writing part of the paper.

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

Graph Neural Networks (GNN) are a promising technique for bridging differential programming and combinatorial domains. GNNs employ trainable modules which can be assembled in different configurations that reflect the relational structure of each problem instance. In this paper, we show that GNNs can learn to solve, with very little supervision, the decision variant of the Traveling Salesperson Problem (TSP), a highly relevant 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 NP-Complete problems which combine symbolic and numeric data.

Joint first author, with Marcelo de Oliveira Rosa Prates and Pedro Henrique da Costa Avelar, on the submission to the 23rd AAAI Conference on Artificial Intelligence (Qualis A1, 2016), accepted for oral presentation, presented by Marcelo de Oliveira Rosa Prates (PRATES et al., 2019a).

The author's contribution to this paper was implementing and calibrating the baseline models (Greedy heuristic and Simulated Annealing), discussing some of the model's features as well as some of the problem's properties (metric and euclidean properties), writing a small part of the paper and revising it.

1.2.3 Typed Graph Networks

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

Joint first author, with Marcelo de Oliveira Rosa Prates and Pedro Henrique da Costa Avelar. on the submission to the IEEE Access journal (Qualis B3, 2016) Pre-print

available (PRATES et al., 2019b).

The author's contribution was writing part of the paper, revising it and providing preliminary discussions and results regarding the Graph Coloring problem.

1.2.4 Graph Colouring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems

> 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 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.

Joint first author, with Marcelo de Oliveira Rosa Prates and Pedro Henrique da Costa Avelar, on the submission to the 31st International Conference on Tools with Artificial Intelligence (Qualis B1, 2016), accepted for oral presentation. Pre-print available (LEMOS et al., 2019).

The author's contribution to this paper was implementing and calibrating the baseline models (Greedy heuristic and Tabucol), designing the GNN-based model, producing the train and test instances and conducting the backbone of the experiments, as well as writing most of the paper.

1.2.5 Neural-Symbolic Relational Reasoning on Graphs Models: Effective Link Inference and Computation from Knowledge Bases

> The recent developments and growing interest in neural-symbolic models has shown that hybrid approaches can offer richer models for Artificial Intelligence. The integration of effective relational learning and reasoning methods is one of the key challenges in this direction, as neural learning and symbolic

> inference offer complementary characteristics that can benefit the development of AI systems. Relational labelling or link prediction on knowledge graphs has become one of the foremost problems in deep learning-based natural language processing research. Moreover, other fields which make use of such neural-symbolic techniques also benefit from such research endeavours. There have been several efforts towards the identification of missing facts from existent ones in knowledge graphs. Two lines of research try and predict knowledge relations between two entities by considering all known facts connecting them or several paths of facts connecting them. We propose a neural-symbolic graph neural network based model, which naturally applies learning over all the paths by feeding the model with the embedding of the minimal subset of the knowledge graph containing such paths. By learning to produce representations for entities and facts corresponding to word embeddings, we show how one trains the proposed model to decode these representations and infer relations between entities in a multitask approach. Our contribution is two-fold: we show how a neural-symbolic methodology leverages the resolution of relational inference in large graphs, and we also demonstrate that such neural-symbolic model is shown more effective than path-based approaches.

Joint first author, with Marcelo de Oliveira Rosa Prates and Pedro Henrique da Costa Avelar, on the submission to the International Joint Conference on Artificial Intelligence 2020 (Qualis A1, 2013-2016), submitted.

The author's contribution was gathering and parsing the main dataset, designing the GNN model as well as the evaluation setup, writing most of the paper and revising it.

1.3 Related Work

1.3.1 Tackling the Graph Coloring problem with Neural Networks

The first endeavor to solve the GCP problem by means of a connectionist architecture dates back to the 80's when Ballard, Gardner and Srinivas (1987) transformed the GCP into a maximum-weight independent set problem and then mapped the optimization problem into the task of minimizing an energy function, where each vertex is associated to one binary neuron in a Hopfield network (HOPFIELD, 1982). Philipsen and Stok (1991) noticed that the update rule used by Ballard, Gardner and Srinivas (1987) had no guarantees to converge to a valid local minimum: usually clusters of neurons/vertices had no neuron activated, or more than one neuron activated, thus producing inconsistent color assignments. Instead, they proposed the use of Potts neurons to each cluster of vertices, whose solution space guarantees, in the high temperature limit, that there will be exactly one neuron on per cluster. These two seminal approaches, although lacking of exhaustive experiments and reasoning features, provided a sneak-peek of how connectionist architectures could be engineered to solve combinatorial problems such as the GCP. More

recently, on (TOENSHOFF et al., 2019), the authors tackled two versions of the GCP (3-COL and MAX-3-COL), translated into Constraint Satisfaction problems (CSPs), with a message-passing network trained in an unsupervised fashion: their model simply returns the best solution it can find, given an unlabeled instance of the problem. When it comes to architectural aspects, their model is composed of a simple linear transformation, which creates messages from internal representations of the variables, a LSTM to aggregate all incoming messages and produce a new internal representation, and a softmax function applied over the internal representations in order to compute a valid assignment. The main difference to our work, besides the unsupervised training, is that their model does not verify the unsatisfiability of a given instance, instead it returns an incorrect assignment. Also, Das, Ahmad and Venkataramanan (2019) proposed a hybrid approach to solve the GCP in a context of register allocation (where no invalid assignment is acceptable): an LSTM is fed with the adjacency vector of each vertex and produces an output sequence with the same size of the number of vertices, this sequence is then decoded by a dense layer+ReLU into a final color value for a given node. As there are no constraints hard-coded in this neural module, the authors also propose a greedy algorithm to correct invalid color assignments, which works independently of the LSTM and needs to be used in almost 85% of the cases.

1.3.2 Graph Neural Networks

Graph Neural Networks have drawn a lot of attention lately and their usefulness has spread to several fields of research, from computer vision (CV) to natural language processsing (NLP). For a thorough and historical review of GNNs one may refer to Battaglia et al. (2018), Zhang, Cui and Zhu (2018) and Wu et al. (2019). GNNs first set of applications, however, was far simpler than CV or NLP tasks, as Gori, Monfardini and Scarselli (2005) benchmarked their GNN model on three types of problems: connection-based problems – clique problem and neighborhood size counting; label-based problem – classifying the parity of a boolean-valued vector assigned to each vertex and a general problem of identifying subgraphs. GNNs potential to calculate centrality measures was also explored by (Scarselli et al., 2005) and (AVELAR et al., 2018). There was also GNN-based models proposed to learn how to generate graphs following a certain distribution, specifically related to mimetizing chemical molecules (YOU et al., 2018a; YOU et al., 2018b). GNNs are also a promising technique for image classification tasks, as demonstrated on the work of (QUEK et al., 2011), where the authors used local scale-invariant region detectors to transform

the original image into a graph which is then fed to a GNN able to maintain topological hierarchy and process undirected connections, and on the work of (SHEN et al., 2018) which specifically focused on the person re-identification problem. Another fertile ground for GNNs are NLP tasks based on knowledge graphs, such as entity classification and link prediction. In this context, one may highlight the R-GCN (SCHLICHTKRULL et al., 2018), which acts like a vertex encoder by applying a graph convolution operator which accumulates transformed feature vectors of neighboring vertices through a normalized sum, and the KBGAT (NATHANI et al., 2019) which adds up attention layers to a GNN-like model, enabling the network to pay different attention (or to assign distinct weights) to each vertex in a given neighborhood. These two models were hand-engineered to solve both entity classification and link prediction tasks, achieving outstanding results and insights on how a neural-symbolic model can foster explainable reasoning over a knowledge graph. Finally, Selsam et al. (2019) were able to uncover meaningful behavior on the internal embeddings of a GNN model designed to solve the decision version of the boolean satisfiability problem. They do so by clustering the final literals embeddings and reducing their dimensionality (Principal Component Analysis (PCA)) to create valid assignments, a strategy which we also followed during this work.

1.4 Dissertation Structure

The remainder of this dissertation has the following structure:

- Chapters 2 and 3 are designed to provide most of the necessary technical background needed for reading this dissertation. More specifically, Chapter 2 formally defines the graph coloring problem along with some insights on its characteristics and computational complexity and Chapter 3 contains a set of Deep Learning basic concepts and techniques which served as foundations for this work, specially highlighting a full description of our base model, the Graph Neural Network.
- Chapter 4 details the conducted experiments in regards to generating instances, chosen baselines and evaluation setup, together with the results of our method.
- Finally, Chapter 5 concludes this dissertation and envisions avenues for future research.

2 THE GRAPH COLORING PROBLEM

The Graph Coloring Problem (GCP) may be understood as an umbrella-term for any problem concerned with assigning colors to certain elements of a graph subject to certain constraints. Usually these elements are the vertices of the graph and the constraints are related to the adjacency information. In a historical perspective, the term "coloring" came from a student of University College London who, in 1852, was coloring a map of counties in England and noticed that only 4 colors were enough to ensure that all neighboring counties were assigned to different colors. Actually, several years later (1976) it was proven, by Kenneth Appel and Wolfgang Haken, that 4 colors are sufficient to color any map (or any planar[1] graph). Still a very important and studied problem in optimization and theoretical computer science due to its applications to many other areas, the GCP may be divided into three main versions:

1. Decision: "Does graph $\mathcal{G}$ admit a proper vertex coloring with c colors so that no adjacent vertices have the same color?". Proven to be NP-Complete by Karp (1972).

2. Optimization: "What is the smallest number of colors (chromatic number – χ) needed to color the vertices of $\mathcal{G}$ so that no adjacent vertices share the same color?". NP-Hard problem.

3. Counting: "How many different valid color assignments does the graph $\mathcal{G}$ accept given c different colors available?". Also known as chromatic polynomial.

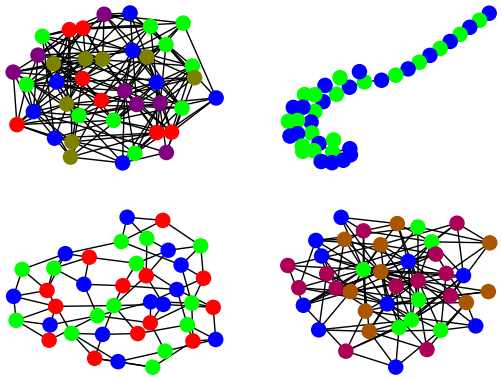
It is worth mentioning that graph coloring problems are solved in polynomial time if $\mathcal{G}$ is 2-colorable, i.e. bipartite graphs without an odd cycle (ASRATIAN; DENLEY; HÄGGKVIST, 1998). Moreover, there are polynomial-time algorithms (GRÖTSCHEL; LOVÁSZ; SCHRIJVER, 1984) to efficiently solve graph coloring problems if $\mathcal{G}$ is a perfect graph, i.e. for every induced subgraph $\mathcal{H} \subseteq_i \mathcal{G}$, the chromatic number of $\mathcal{H}$ – $\chi(\mathcal{H})$ – equals its clique number[2] (GOLOVACH et al., 2014).

The aforementioned variations of GCP and many others are extensively studied in the context of register allocation (CHAITIN et al., 1981), circuit board testing (GAREY; JOHNSON; SO, 1975), mobile radio frequency assignment (GAMST, 1986) and other applications that can be seen as scheduling or resource allocation problems. To show the

[1] A planar graph is a graph that can be drawn with no crossing edges.

[2] Maximum size of a subset of vertices of an undirected graph such that every two distinct vertices in the subset are adjacent.

Figure 2.1: Pictorial representation of four vertex coloring problems applied to different graph distributions. Clockwise: random, power-law tree, power-law cluster, small-world). All graphs are colored with their chromatic number. Source: author.



generality of the GCP definition, we may suppose that several tasks need to be scheduled, each task will demand some resource and we need to ensure that no resource is being used during the same time slot. Such problem can be framed as a vertex coloring problem, where the tasks are defined as the vertices and two tasks demanding the same resource are connected with an edge. In this sense, the minimum number of colors needed corresponds to the minimum number of time slots required to complete all tasks.

2.1 Formulation: Vertex Coloring and Chromatic Number

In this work, we are particularly concerned with a mix of two versions of the GCP: the decision and the optimization ones. Although our model is intended to answer a decision question (yes or no), our training procedure, which will be later described, leverages the definition of chromatic number in order to produce very hard instances of GCP. Also, our evaluation setup emulates the optimization problem: we repeatedly fed our model with a given instance and increasing number of colors, until the first positive answer is reached – which indicates the chromatic number. In this sense, the next paragraphs will provide general and formal definitions for both versions of GCP.

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a structure consisting of a finite set $\mathcal{V}$ of vertices and a finite edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, the GCP can be generally defined as the problem of assigning a color c to each vertex such that adjacent vertices – $v \in \mathcal{V}$ and $u \in \mathcal{V}$ are adjacent if and only if $e = (u, v) \in \mathcal{E}$ – are not assigned to the same color. Traditionally, GCP is applied to undirected graphs, i.e. if $e = (u, v) \in \mathcal{E}$ then $e = (v, u) \in \mathcal{E}$. The decision version of GCP, however, only intends to give a binary answer to the described problem, with no prior

need to build up a valid color assignment: given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of colors $\mathcal{C}$, is it possible to create a mapping function $F : \mathcal{V} \rightarrow \mathcal{C}$ such that no adjacent vertices are mapped into the same color $c \in \mathcal{C}$?

As aforementioned, if one is interested in finding the smallest number of colors which yield a valid assignment, then the GCP becomes an optimization problem concerned with minimizing the size of $\mathcal{C}$ and that can be formulated as an Integer Linear Programming (ILP) as in Definition

1, for a graph $\mathcal{G}$ where $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$.

This ILP formulation, proposed by (LIMA; CARMO, 2018), makes the natural assumption that n colors are enough to properly build an assignment for any given graph. The minimizing function is applied to the sum of the binary variables x_j, which indicate whether color j is part of a solution. Each variable y_{vj} indicates whether vertex v is colored with j. The first set of constraints enforces that each vertex can have just one color assigned to it. The second set of constraints is the one responsible for avoiding two neighboring vertices to have the same color.

Definition 1 (GCP Chromatic Number ILP Formulation)**.**

$$\min \sum_{j=1}^{n} x_j$$

subject to:

$$\begin{aligned}
&\sum_{j=1}^{n} y_{vj} = 1 && \forall v \in \mathcal{V} \\
&y_{vj} + y_{uj} \leq x_j && \forall (v,u) \in \mathcal{E}, j \in 1 \dots n \\
&y_{vj} \in \{0,1\} && \forall v \in \mathcal{V}, j \in 1 \dots n \\
&x_j \in \{0,1\} && j \in 1 \dots n
\end{aligned} \tag{2.1}$$

2.2 Computational Complexity

We already had a glimpse on the fact that GCP is NP-Complete in its decision version – where a graph and a number of colors is informed to the solver – and that its optimization version (minimizing number of colors needed) is NP-Hard. In the field of computational complexity, problems are usually posed as decision problems, requiring the algorithm only to output “yes” or “no”. Decision problems for which there is an algorithm

whose execution time can be expressed via polynomials, i.e. not prone to combinatorial explosion, belong to the class P, which stands for polynomial-time problems.

On the other hand, NP problems, i.e. non-deterministic polynomial time, are the ones that can have a candidate answer evaluated in polynomial time. For instance, assuming the decision version of GCP "Does the graph $\mathcal{G}$ admit a proper vertex coloring with $\rfloor$ colors so that no adjacent vertex has the same color?" it is quite straightforward to verify if a given assignment of colors is valid or not. Obviously, any problem in P is also in NP – we can solve a problem in P by using its polynomial-time algorithm to build a proper solution. That basically means that $P \subseteq NP$, however it is still not known if $P = NP$[3]. Another class of problems is the NP-Complete one, whose problems can have their candidate solutions verified in polynomial time (NP-Complete $\subseteq NP$) but there are no known algorithm available to give a binary answer in polynomial time – that is, an algorithm for identifying a "yes" solution to NP-Complete problems would have to resort to enumerating and checking a significant portion of the solution space, whose size grows exponentially compared with the problem size.

Regarding GCP and many other intractable computational problems, it is common to see a version of them referred to as NP-Hard problems. NP-Hard problems are at least as difficult as NP-Complete problems but they are not required to be in NP, i.e. they are not required to be stated as decision problems. This implies that an NP-Complete problem can be transformed into a NP-Hard by changing its goal, instead of asking "Does the graph $\mathcal{G}$ admit a proper vertex coloring with c colors so that no adjacent vertex has the same color?", we ask "What is the smallest number of colors (chromatic number – χ) needed to color the vertices of $\mathcal{G}$ so that no adjacent vertices share the same color?" (LEWIS, 2016). But up to this point, we have not yet shown how the decision version of GCP fits into these classes of problems. Cook (1971) introduced the concept of NP-Completeness and polynomial time reduction, and also proved that the boolean satisfiability problem (SAT) is NP-Complete: Cook showed that there is no known algorithm to efficiently solve all instances of SAT, specially due to the UNSAT instances which requires the algorithm to test all possible assignments – leading to combinatorial explosion – before giving a negative answer. The decision GCP is NP-Complete because it can generalize to the 3-SAT (a special case of SAT where each clause has strictly three literals), in other words, a 3-SAT problem can be reduced to a graph coloring problem. And 3-SAT itself can generalize to SAT, which is known to be NP-Complete (COOK, 1971). Seminal work of (KARP, 1972)

[3]This is actually one of the Millenium Prize Problems stated by the Clay Mathematics Institute. It remains an unsolved problem, together with Riemann Hypothesis, Hodge Conjecture, among others.

demonstrated these two polynomial time reductions and others between 3-SAT and several combinatorial problems.

2.3 Phase Transition on Graph Coloring

As in any other combinatorial problem, GCP also undergoes a phase transition phenomenon: when some graph parameter reaches a specific threshold, no c-color assignment can be found anymore. For GCP, this parameter is usually related to the vertex connectivity (degree), but in principle it could be any parameter related to the amount of edges in the graph. Particularly, as the average vertex degree increases, a threshold phenomenon is observed: the graph accepts less valid c-coloring solutions, until no solution can be found.

Zdeborová and Krzakala (2007) defined five transitions of connectivity which cause the space of solutions to be gradually more sparse until it becomes completely empty. The last transition, called COL/UNCOL transition, can also be used to build very hard instances of the GCP, since it is empirically known that deciding the feasibility of a c-coloring becomes much harder near to the coloring threshold COL/UNCOL than far away from it, where the average vertex degree is lower (CHEESEMAN; KANEFSKY; TAYLOR, 1991). In this work, we explore this property in order to create both training and test instances, as it will be described later on Section 4.2.

3 MACHINE LEARNING, DEEP LEARNING AND GRAPH NEURAL NETWORKS

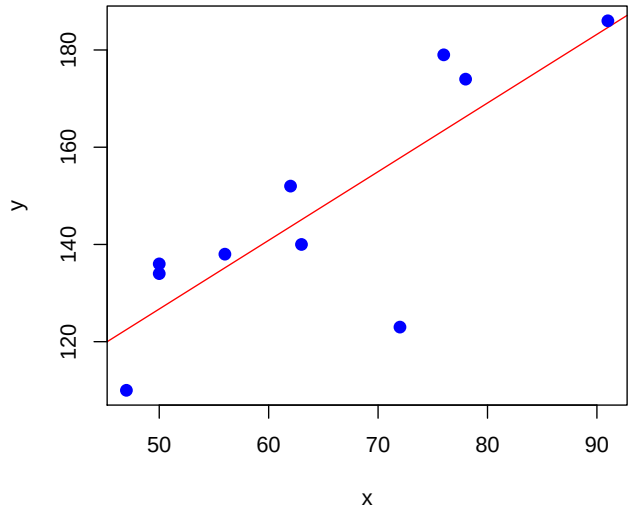
This chapter introduces the underlying concepts of the neural-symbolic methodology used throughout this research along with a brief explanation of the basic neural models that constitute the core of our work: a Graph Neural Network. We assume a basic knowledge of linear algebra and calculus, specifically with regards to vectors, matrices and their basic operations. For a more rigorous explanation of Machine Learning (ML) and Deep Learning (DL) in general, the reader may refer to (WITTEN; FRANK; HALL, 2011), (GOODFELLOW; BENGIO; COURVILLE, 2016)[1] or (NIELSEN, 2015)[2].

3.1 An Overview of Machine Learning Goals, Methods and Pitfalls

Machine Learning in general deals with the approximation and generalization of some function given a limited observation range of it. This process can be seen as a curve-fitting model whose objective is to minimize the distance between the limited observation range and the learned function. Note that learning the function is a key-aspect to accomplish in Machine Learning, since if the ML model can only memorize what it has seen, then it will not be able to extrapolate its knowledge to unseen instances – in fact, one may argue that there would be no knowledge at all in this case. During the learning phase, usually called **training**, an ML model is fed with N examples of the function it needs to learn, in the form of $(x_i, y_i), \cdots, (x_N, y_N)$ such that x_i stands for the feature vector of the i-th example and y_i stands for its target value. The model is supposed to learn a function $g : X \rightarrow Y$, which can be seen as a part of the hypothesis space, or space of possible functions, G. In this sense, ML models employ the concept of a **loss function**, a measure of how far away are $g(x_i)$ and y_i. One of the most basic loss functions is the mean average error, given by $MAE = \frac{\sum_{i=1}^{n} |g(x_i) - y_i|}{n}$. This is the basic definition for **supervised** learning, where each input feature vector has a correspondent target value and whose container models may range from linear regression (see Figure 3.1), logistic regression, support vector machines and even graph neural networks. Supervised learning is well-suited for both classification and regression problems: the former deals with categorical data (y_i is a discrete target value) and the latter has its targets in a continuous domain. As there are many ML algorithms, one way of distinguishing between them is the form taken by g,

[1] Available online at <www.deeplearningbook.org> as of 15/01/2020
[2] Available online at <www.neuralnetworksanddeeplearning.com> as of 15/01/2020

Figure 3.1: Linear regression algorithm applied to approximate a non-linear one-dimensional function (dots). Source: author.



for instance, Naive Bayes classifiers learn $g = P(x, y)$ (joint probability) while a logistic regression model learns $g = P(y|x)$ (conditional probability).

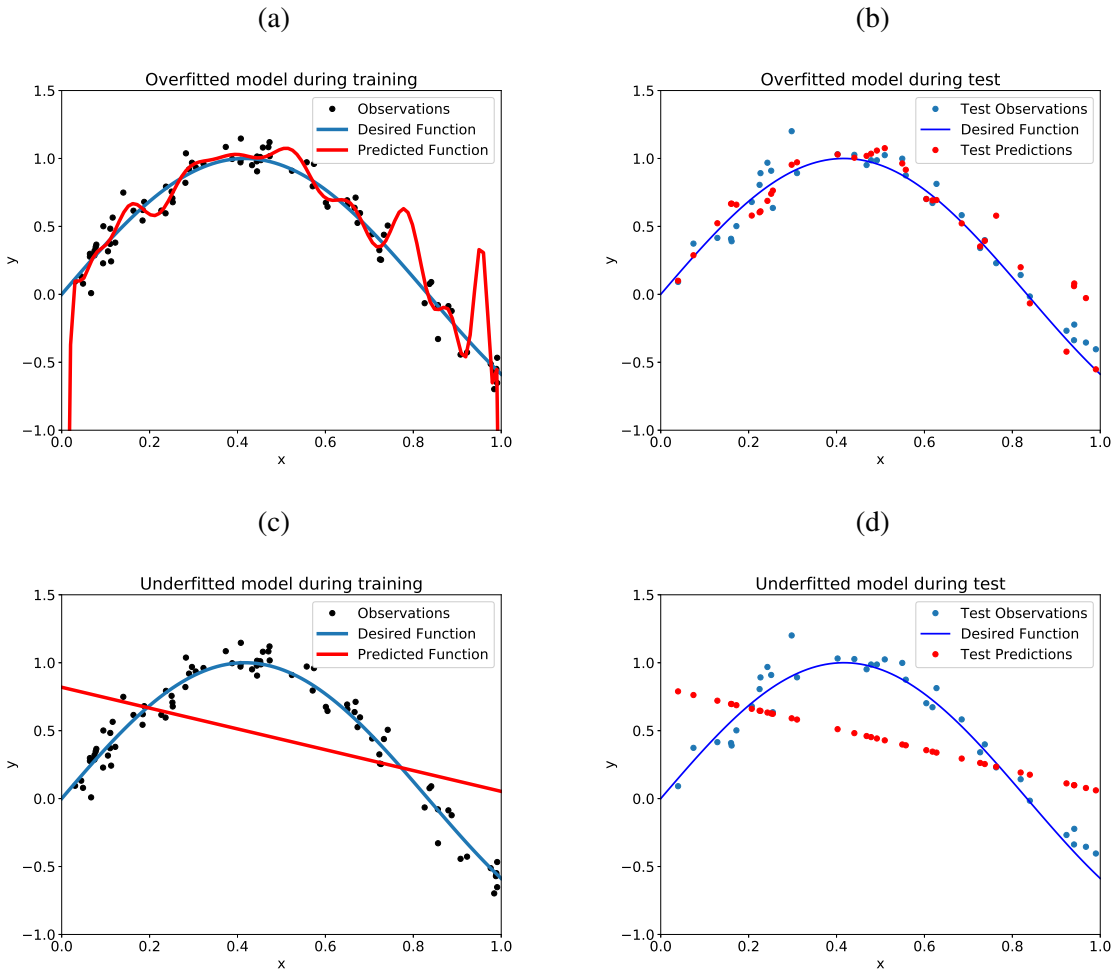
The presence of labeled data is not, however, a premise for an ML algorithm to learn a desired behavior or function. There are successful examples of ML algorithms applied to partially labeled data (**Semi-supervised** learning) to build generative models (KINGMA et al., 2014) and also to identify biomolecules (KÄLL et al., 2007), for instance. It is also feasible to perform some kind of learning over completely unlabeled data – **unsupervised** learning. In this case, the ML algorithm may either attempt to cluster the unlabeled data according to some inferred pattern/similarity, to find frequent co-occurrences of items (association rules), to identify outliers in the input distribution or to project the original datapoints into some latent space, which may summarize some of the input's properties.

Instead of a label, one may also envision a concept of reward. In this case, a ML model, usually renamed to agent, participates in a decision-making process, in which the agent's outputs, or decisions, yield a reward or a penalty depending on the environment's nature. This is particularly useful when the entire learning environment can not be exactly modeled. This learning paradigm is known as **Reinforcement learning** and one of its most traditional algorithms is Q-Learning, which rewards or penalizes the agent based not only on its action but also on its current state. The Q-Learning agent defined by (MNIH et al., 2013), for instance, is required to learn how to play an Atari game by pressing some button and observing the related reward / penalty. The state definition is given by

the configuration of pixels in the screen – yielding a huge state space ($\approx 10^{3533}$), and thus requiring the Q-Learning table associated with states and actions to be modeled as a convolutional neural network fed directly with the screenshot image of the current state.

In this dissertation, our experiments solely rely on supervised learning techniques, although we also apply unsupervised learning methods to explore the meaningfulness of the representations produced by our model. In this sense, it is paramount to state that supervised learning is always divided into two stages: the training, where a loss function is computed and used to adjust internal parameters, and the test, where an error function defines some accuracy-like metric. The training and the test set of instances must be disjoint, so that the error function reflects the model's capability of generalizing its performance to unseen instances, but the test data must – or, at least, should - come from the same distribution as the training data, to prevent a problem known as covariate shift, which decreases the model's performance during test. Moreover, an ML algorithm can also fit the training dataset too tightly, yielding a very low loss metric but potentially a very low accuracy as well, a problem known as **overfitting** and that can be tackled by adjusting a parameter called learning rate or even by applying some regularization strategy, which try to balance the amount of parameters used within the model. The opposite pitfall is known as **underfitting** – when the model fails to fit both training and test data. These two ML pitfalls are shown in Figure 3.2.

Figure 3.2: An overfitted regression model during training in Subfigure (a) and its impact during the test in Subfigure (b). An underfitted model during training in Subfigure (c) and its impact during the test in Subfigure (d). Source: Author.



3.2 Deep Learning Preliminaries

Essentially, Deep Learning refers to any learning algorithm whose most basic block consists in (some form of) an artificial neuron and whose architecture is defined by many layers of connected artificial neurons. As its name suggests, the artificial neuron mechanics is inspired by the mechanics of biological neurons, or nerve cells. Basically, a biological neuron (see Figure 3.3) receives inputs through its dendrites, processes them in its body and sends them outside via its axon terminal. This last stage is in fact governed by the all-or-none law: the output of a neuron is a binary value caused by the input stimulus being either below or above a threshold potential, i.e. the strength of the input signal is not perfectly reflected in the output but rather indicates if the output is on or off.

A single artificial neuron (see Figure 3.4 can be modeled as an activation function applied to a weighted sum. The weighted sum is obtained by multiplying each of the inputs x_n with a correspondent weight w_n and adding them up with a special parameter called bias, which simply permits the output of the function to be shifted up or down, thus precisely fitting a desired output function. Then, the weighted sum is fed to an activation function σ which produces the predicted output

Figure 3.3: Structure of a biological neuron. The dendrites capture the input signal, which is then processed by the cell body and expelled out via axon terminals. Source: Wikimedia Commons.

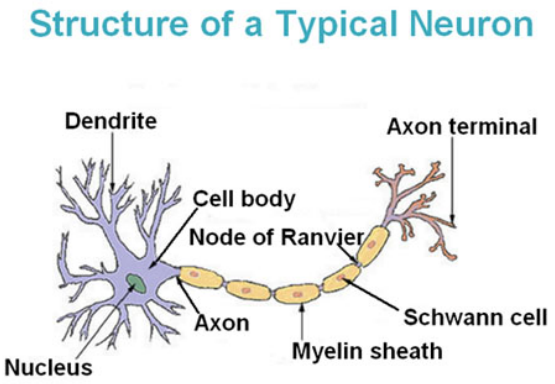
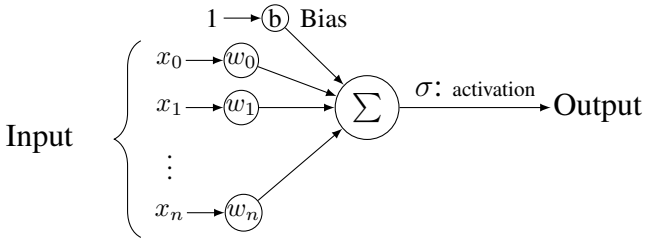


Figure 3.4: Pictorial representation of a single artificial neuron, also known as perceptron. Source: author.



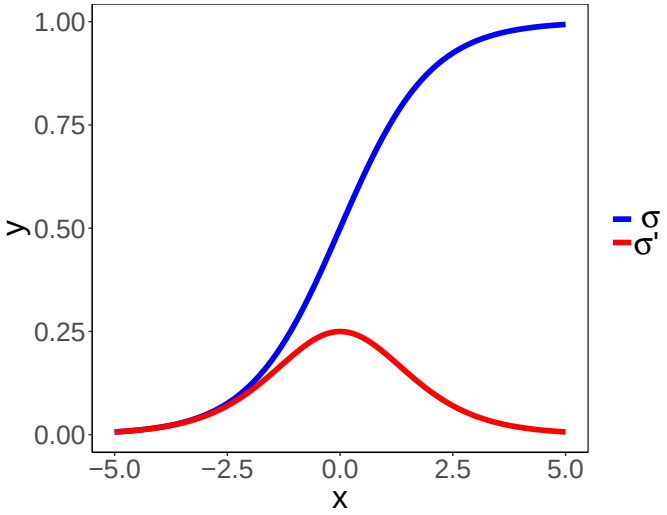
In this sense, a single artificial neuron outputs a function given by:

$$g(x) = \sigma(b + w_1x_1 + w_2x_2 + \cdots + w_nx_n) \tag{3.1}$$

where $w_1, w_2, \cdots, w_n$ are the neural weights which are learned during training.

As biological neurons can be connected to other biological neurons to perform more complex tasks, artificial neurons can also be arranged in a network to produce answers to difficult tasks (Artificial Neural Network – ANN). This network configuration can be achieved by connecting the output of a neuron to the input of another neuron in the next layer, or, mathematically, by iterating a composition of activation functions applied over weighted sums. In order to learn a desired function, ANNs need to update their internal weights according to their contribution to the final output. This process is usually done by backpropagating (RUMELHART; HINTON; WILLIAMS, 1986) the loss during training, which requires that all activation functions are differentiable or, at least, differentiable almost everywhere. It is also important that the activation function presents non-linear properties so the neural network can learn nonlinear relationships in the data.

Figure 3.5: Graph of the sigmoid function and its derivative, while the first one is often used as an activation function in ANNs, the second one is computed in regards to the internal weights in order to update them during the backpropagation stage. Source: author.



All in all, choosing an appropriate activation function is essentially what determines the expressiveness of an ANN. For instance, one of the most common activation functions is the sigmoid function – or logistic function (Figure 3.5) – whose original formula and derivative corresponds to:

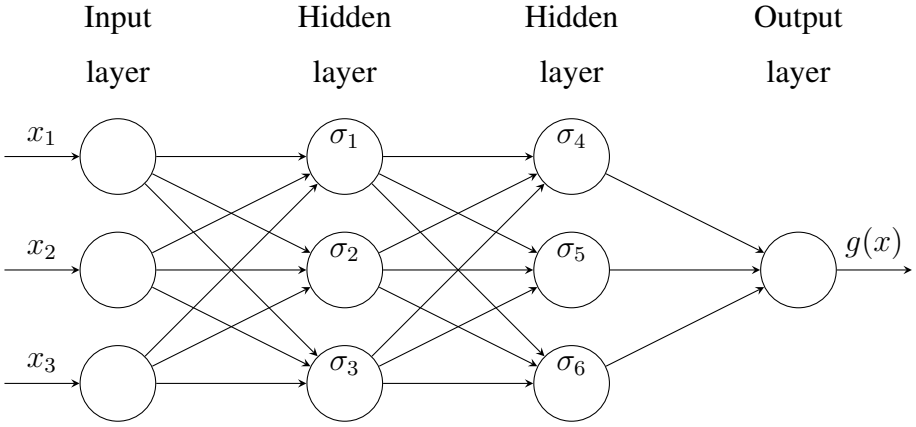
$$\begin{aligned} \sigma(z) &= \frac{1}{1+e^{-z}} \\ \sigma(z)' &= \sigma(z)(1-\sigma(z)) \end{aligned} \tag{3.2}$$

Note that the sigmoid function reflects the all-or-none law we previously described, except for a narrow margin around 0. But this property is not always required, and we may forgo it in exchange for computational efficiency and select an unbounded function such as the rectified linear unit – $ReLU(z) = max(0, z)$. Choosing an appropriate activation function is particularly important at the output layer, since this last function determines the form of the output: a sigmoid activation at the last layer means that the ANN will be a binary classifier, whereas an identity function implies that the ANN will be a regressor.

Deep Learning major premise is that increasing the **depth** of an ANN yields better expressiveness and results than increasing its **width**. In an ANN, width corresponds to the number of single neurons in a given layer whereas depth is equal to the total number of layers, including the input one, the hidden ones and the final output layer. For instance, the ANN depicted on Figure 3.6 is a 4-layered neural network whose input and hidden layers have a width equals to 3. This is also an example of a **feedforward** artificial neural network, since there are no connections between neurons at the same layer, no connections

between an $n+1$-th layer and an n-th layer and no self-loops, thus no cycle can be formed. Moreover, a feedforward artificial neural network with at least 3 layers (input, hidden and output) is also known as Multilayer Perceptron (MLP).

Figure 3.6: 4-layered ANN with two hidden layers indicated by their activation functions $\sigma_1, \sigma_2, \cdots, \sigma_6$, we do not show neural weights and biases to enhance readability. Source: Author.



The forward pass of an ANN can be understood as a series of matrix multiplications, one multiplication per layer, without violating the differentiability constraint. For instance, the inputs of the first hidden layer of the ANN depicted in Figure 3.6 are given by:

$$\begin{pmatrix} w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + b_1 \\ w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3 + b_2 \\ w_{3,1}x_1 + w_{3,2}x_2 + w_{3,3}x_3 + b_3 \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{3.3}$$

where $w_{1,1}$ corresponds to the neural weight associated with the hidden neuron 1 and x_1, whereas $w_{2,1}$ corresponds to the weight associated with hidden neuron 2 and x_2, and so on. The second hidden layer, however, deals with the input provided by the first hidden layer, thus its input operation can be formulated as:

$$\begin{pmatrix} w_{4,1}g_1 + w_{4,2}g_2 + w_{1,3}g_3 + b_4 \\ w_{5,1}g_1 + w_{5,2}g_2 + w_{5,3}g_3 + b_5 \\ w_{6,1}g_1 + w_{6,2}g_2 + w_{6,3}g_3 + b_6 \end{pmatrix} = \begin{pmatrix} w_{4,1} & w_{4,2} & w_{4,3} \\ w_{5,1} & w_{5,2} & w_{5,3} \\ w_{6,1} & w_{6,2} & w_{6,3} \end{pmatrix} \begin{pmatrix} g_1 \\ g_2 \\ g_3 \end{pmatrix} + \begin{pmatrix} b_4 \\ b_5 \\ b_6 \end{pmatrix} \tag{3.4}$$

where g_1 stands for the activated output of first neuron in the first hidden layer, or $g_1 = \sigma_1(w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + b_1)$. In a nutshell, each of the i layers of an ANN is

responsible for computing:

$$g_i(\vec{x}) = \sigma(\mathbf{W}_i \otimes \vec{x} \oplus \vec{b_i}) \tag{3.5}$$

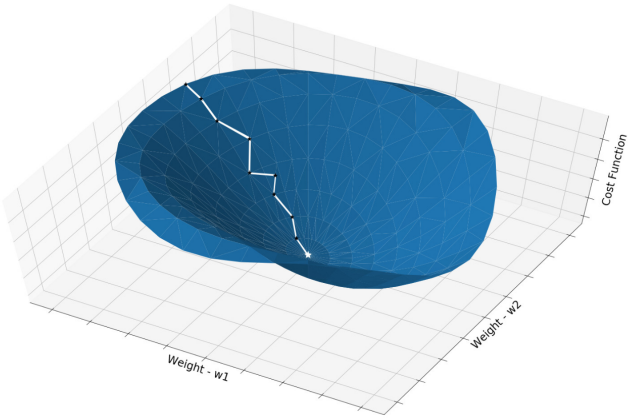
where $\vec{x}$ stands for the input tensor fed to the given layer, $\mathbf{W}_i$ corresponds to the matrix of weights of that layer (each row associated to a neuron) and $\vec{b_i}$ stands for its bias vector.

It is easy to verify that increasing the depth of an ANN implies that more matrix multiplications will be required, while increasing the size of the layers (width) results in larger matrices to be multiplied. Fortunately, both drawbacks are easily handled by modern Graphics Processing Units (GPUs), which are mainly designed to accelerate matrix multiplications. Such GPUs allowed, for instance, the feasibility of training a convolutional neural network such as the ResNet (HE et al., 2016) with 152 layers and $\sim$4M parameters to achieve state-of-the-art performance in image classification tasks and such as the FaceNet (SCHROFF; KALENICHENKO; PHILBIN, 2015) with 22 layers and $\sim$140M parameters, a current state-of-the-art architecture to tackle the face recognition problem. Treating each layer of an ANN as matrix multiplications is usually called **vectorization** and not only enables speeding up training but also saves a huge amount of time for any programmer designing a neural architecture, as several for-loops can be replaced by a single matrix multiplication operation. Here, it is also worth defining that an n-dimensional array can be generalized to a **tensor** of rank n. For instance, the weight matrix of the right side of Equation 3.4 is a tensor of rank 2 whose shape is $(3, 3)$ – size in each dimension. While the bias vector is a tensor of rank 1 and shape 3. We will not expand on a mathematical definition of a tensor but it is quite important to shed light on this basic terminology as we will describe further neural modules in terms of tensorial operations.

3.3 On Training an Artificial Neural Network

In the last section, we briefly mentioned that it is important to have only differentiable functions along the pipeline of an ANN. This is due to the fact that we need to compute the contribution of each variable inside our model (weights and biases) to the resulting loss and a faster way of doing so is to partially derive the loss w.r.t. each set of variables.

Figure 3.7: The result of applying gradient descent on a surface produced by a loss function with 2 parameters. Source: (SHARMA, 2018)



Formally, after one forward pass, we have a n_x-dimensional input datapoint x, a target n_y-dimensional label y and a model M, which is essentially a composition of matrix multiplications and activation functions, which operate over a set of m updatable variables $\mathcal{W} = w_1, w_2, \ldots w_m$. The output of our model is represented by $g(x, \mathcal{W}) = M(x, \mathcal{W})$. We formally define a general loss function, for instance an MAE loss:

$$\mathcal{L}(\mathcal{W}) = |g(x, \mathcal{W}) - y| \tag{3.6}$$

As $g(x)$ is parameterized by $\mathcal{W}$, we can compute a partial derivative of $\mathcal{L}$ for each variable $w \in \mathcal{W}$, yielding the following gradient:

$$\bigtriangledown\mathcal{L} = \begin{pmatrix} \frac{\partial\mathcal{L}}{\partial w_1} \\ \frac{\partial\mathcal{L}}{\partial w_2} \\ \vdots \\ \frac{\partial\mathcal{L}}{\partial w_m} \end{pmatrix} \tag{3.7}$$

Graphically this gradient would be the slope of the tangent line to the loss function given the current $\mathcal{W}$. An example of gradient produced by a loss (or cost) function parameterized by 2 weights is shown in Figure 3.7. The mathematical purpose of the gradient vector is to provide us the direction on which the loss function increases, thus we need to adjust our variables towards the opposite direction pointed by $\bigtriangledown\mathcal{L}$.

In practice, the **backpropagation** algorithm (RUMELHART; HINTON; WILLIAMS, 1986) computes each layer's partial derivative individually, starting from the last layer until reaching the first layer, based on the chain rule of composite functions. Once the full gradient vector $\bigtriangledown\mathcal{L}$ is obtained, each of our variables w_n is subtracted by its correspondent gradient value $\frac{\partial\mathcal{L}}{\partial w_n}$ and multiplied by a **learning rate** which allows one to adjust the size

of the step taken towards a smaller loss – this is particularly important as higher learning rates will deteriorate the convergence of the model when dealing with complex or not regularly-shaped losses.

It is not feasible, however, to individually apply the above algorithm for each instance of a problem, as DL models usually need thousands of instances and several epochs[3] to converge. To overcome this issue, only the forward pass is computed for any number b of instances, effectively producing b values of $\mathcal{L}(\mathcal{W})$ – note that $\mathcal{W}$ does not change during this forward pass. These losses are then aggregated, via sum or mean for instance, and the resulting gradients can be computed. Ideally, one would fit the entire dataset into this technique called **Batch Gradient Descent**, as it leverages data parallelism and parameter sharing to their maximum. However, due to computational constraints, the most common strategy is to decrease the b number of instances (**Minibatch Gradient Descent**) prior to applying the forward and the backward passes. Particularly, when b is of any size lesser than the entire dataset size, most of the researchers call this learning technique **Stochastic Gradient Descent**. Several other learning algorithms based on stochastic gradient descent exist and one may highlight three of them, which are consistently used in several DL well-established models: Adam (KINGMA; BA, 2014), Adagrad (DUCHI; HAZAN; SINGER, 2011) and Adadelta (ZEILER, 2012).

3.4 Convolutional Neural Networks

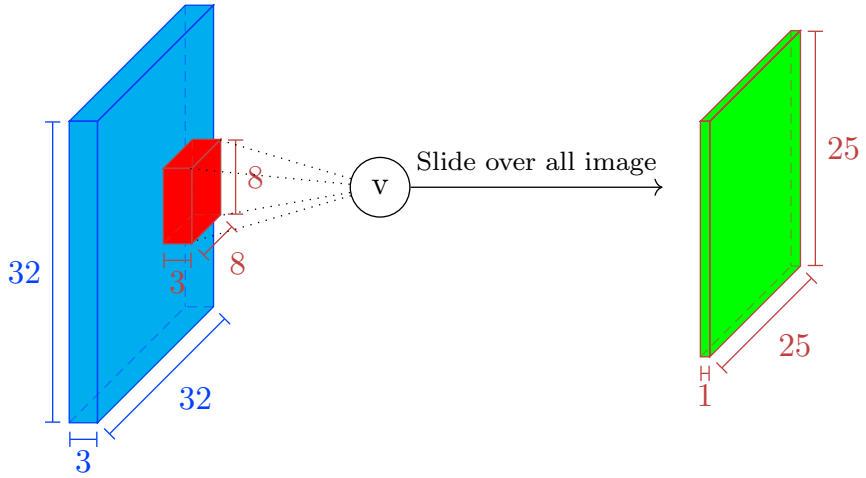
Although MLPs can be very powerful performing classification or regression tasks over an input vector, there are some drawbacks when one needs to process spatial data, such as images, with them. For instance, an MLP connects a neuron of the input layer to every input datapoint, which is, considering an RGB image, a pixel multiplied by three – thus a 640×480 image would require around 920k weights, for just one neuron. Another problem is that MLPs are not translation invariant, i.e. an MLP could output a different prediction for a given image and its shifted version. This translation problem happens because when one flattens an image prior to feed it to the MLP, all spatial information is lost and pixels that were originally close to each other will be considered as single and independent units of information. Convolutional Neural Networks (CNNs) were designed to tackle these issues related to n-dimensional inputs.

[3] One epoch means that the whole dataset was passed both forward and backward through the neural network.

CNNs biological inspiration is the visual cortex – a portion of the brain filled with specialized clusters of cells. Neuroscience research, back in the 1960s, showed that some neuronal cells in the visual cortex fired only when exposed to horizontal edges, a different group of cells, however, fired only when exposed to vertical edges, and so on (HUBEL; WIESEL, 1962). When considered all together, these specialized cells produce what we call visual perception – the capability of distinguish what is being framed in a given image. To emulate such property in a DL model, we basically need to tie together all the weights of such specialized cells – or sliding them over different regions of an image while still sharing the same neural weights.

Concretely, the CNN block designed to perform the operation described in the last paragraph is called convolutional layer, and it is always placed as the first layer of any CNN-based architecture. The main component of the convolutional layer is its kernel, sometimes also known as **filter**, which is essentially a n-rank tensor of weights, i.e. a structure analogous to our previous definition of a neuron but with no activation defined. Convolving the filter with the image basically consists in a weighted sum of the filter weights by respective image pixels. The amount of scalar values produced by a filter is dependent on how we slide it throughout the entire image, or on the size of each sliding step (stride). Figure 3.8 shows a convolutional layer whose 3-dimensional filter (stride=1) has shape $(8, 8, 3)$ and produces a $(25, 25, 3)$-shaped feature map. Note that the size of the last dimension of the filter must match its analogous value in the image in order to enable the correct multiplications. Usually, a convolutional layer j has a set of i independent filters, i.e., in our case, each filter would be independently convolved with the image, producing i feature maps of shape $(25, 25, 1)$ or a single volume shaped as $(25, 25, i)$.

Figure 3.8: The operation performed by a convolutional layer over an image (blue volume): a filter (red volume) slides over all unique grid of pixels (stride=1) and produces a single scalar value v, which is inserted into a resulting 2-dimensional array (green)– also called feature map. Source: author.



In a high level perspective, each filter can be seen as a feature identifier, such as curves, straight edges and so on. Mathematically, each filter $\mathbf{W}_{ji}$ learns to adjust its weights in order to operate a linear transformation on the pixels it has seen. Some sets of values for the elements of $\mathbf{W}_{ji}$ are already well-known in image processing, since their operations are quite fundamental. For instance, the Laplacian filter in Equation 3.8 is used to detect edges.

$$\mathbf{W} = \frac{1}{9}\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{3.8}$$

After a convolutional layer, it is useful to apply a non-linear activation – such as ReLU – in order to enhance the expressiveness of the learned transformation, as the convolutional layer itself only performs a linear transformation (element-wise multiplications and sum). In a nutshell, a ReLU applied to a given feature map only changes its negative values to 0. Also, researchers already verified that ReLU activations speed up the training of CNNs in image classification tasks as well as alleviates the problem of vanishing/exploding gradient[4].

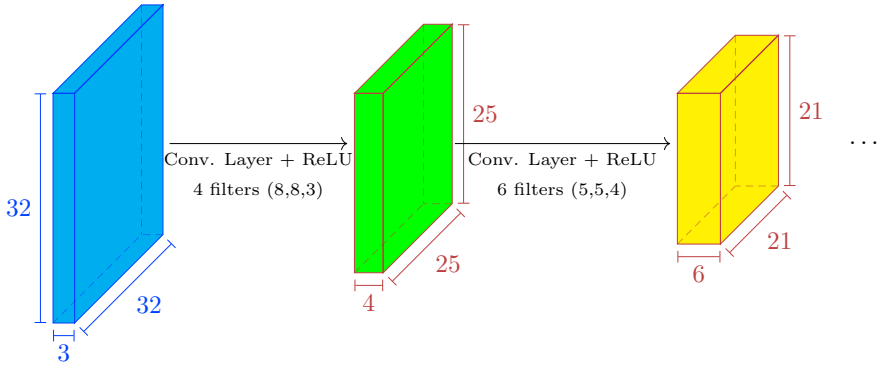
It is also common to apply some kind of downsampling to the resulting feature maps, particularly because once a specific feature is uncovered, we do not need its exact location but rather a relative location to other uncovered features. Naturally, the number of

[4]Due to the chain rule applied to a huge number of functions, at lower layers the gradient may become so small/big that no significant/a drastic update is made in the weights of those layers

parameters of the next layer can also be reduced when the previous layer is a downsampling one. Another obvious benefit of downsampling is that we reduce the chances of overfitting our model. In such context, the most common pooling method is max-pooling, which essentially returns the max value of a (n, n) region of a feature map.

As the CNN architecture goes deeper, filters become more and more specialized since they are able to associate all simpler feature maps from the previous layer. For instance, the 4 feature maps produced by the first conv. layer in Figure 3.9 would highlight low-level features, such as edges, straight lines and so on, whereas the 6 next feature maps would identify higher-level features, such as the presence of a triangle or another polygon. Identification actually means that a feature map will present a certain configuration of values to indicate whether a feature is found in the input. But so far no classification or detection has been properly made. To accomplish that, the output of the last convolutional layer (highest-level feature maps) is usually fed into a fully-connected MLP whose loss function enables some of the tasks we mentioned – for instance, one may apply the sigmoid σ loss to a binary task ("Is there a person in the original image?") or more sophisticated losses to categorical tasks. Moreover, despite CNNs' very pronounced success in computer vision tasks such as image classification (KRIZHEVSKY; SUTSKEVER; HINTON, 2012), pose estimation (YANG et al., 2019) and semantic segmentation (CHOY; GWAK; SAVARESE, 2019), their mechanics can be applied also to analogous tasks on 1-dimensional data, such as speech recognition on audio files (ABDEL-HAMID et al., 2014) and relation prediction over entities embeddings in a knowledge graph (NGUYEN et al., 2018).

Figure 3.9: Part of a convolutional neural network architecture. The original image (blue) is convolved into 4 low-level feature maps, which are then convolved into 6 high-level feature maps. Source: author.



Although we do not use a proper CNN in our work, it is quite important to highlight some of its basic insights, as they will be revisited when we discuss the graph neural

network model. First of all, the CNN filters, or kernels, are all prone to gradient descent algorithm thus leveraging the already mentioned data parallelism (Section3.3) – many filters are learned at once and at different consecutive levels. Second, CNNs enable parameter sharing across the spatial relationships of the input volume: by using the same weights, or filter, $\mathbf{W}_{jn}$ over all parts of an input, CNNs present properties such as equivariance and invariance to some transformations (translation, rotation, etc.) at some level (LENC; VEDALDI, 2015; KAUDERER-ABRAMS, 2018) – these properties are particularly useful to image processing tasks since important features arise locally (and not globally). Third, also due to parameter sharing over space, we refrain from having a fully-connected neural structure[5], which would require different groups of weights for each pixel. Numerically, we can exemplify such drawback with a black-and-white image of $640 \times 480 \approx 307$k pixels: in the first hidden layer of an MLP with width equals to 5, our model would span $5 \times 307\text{k} \approx 1.5$M parameters (weights and biases); on the other hand, the first convolutional layer of a CNN composed also by 5 filters (e.g. $(100, 100)$-shaped) would yield a total of $100^2 \times 5 = 50$k parameters. The basic assumption is that filters whose shapes are much lesser than the original image are sufficient to detect any kind of feature, as long as they are correctly slid along the entire image, leveraging neighborhood relationships. For instance, a filter which identifies vertical edges matches the left border of an image but will also match the right border. Ultimately, this huge decrease in the parameter space size allows one to train a CNN very fast in comparison to an analogous fully-connected network (less matrix multiplications and partial derivatives during gradient descent). Note, however, that the parameter sharing provided by a CNN architecture relies on grid-shaped data, such as images or a simple 1-dimensional input, and to enable such advantage on non-regularly shaped inputs, as an arbitrary graph, one may have to design a different neural architecture – this is the case of a Graph Neural Network, whose parameter sharing property is based on a different domain.

3.5 Recurrent Neural Networks

When dealing with sequential data, it is a key-feature to have persistence over time, that is, once the model receives a new instance, it must consider all, or at least a part of,

[5]Note that a convolutional kernel with the same size of the input image is equivalent to a fully-connected neural network. The main insight of a CNN is to define a fully-connected kernel with a smaller size and to slide it over the image.

the previous inputs it has already seen. In other words, the full meaning of an instance is closely related to its neighboring instances in the time domain. Such property calls for a model which is not only able to store knowledge gathered from previously-seen data but also to take into account the ordering of its inputs. The Recurrent Neural Networks (RNNs) are a subfamily of ANNs whose main principle relies on the reuse of a single neural cell across time to process sequential data. Note that RNNs can be classified by what they effectively reuse and how they reuse it, for instance, an RNN cell can reuse only its previous output, or both its previous output and an internal state, in any case this property can also be seen as a case of parameter sharing, much like the one presented by CNNs but in the temporal domain.

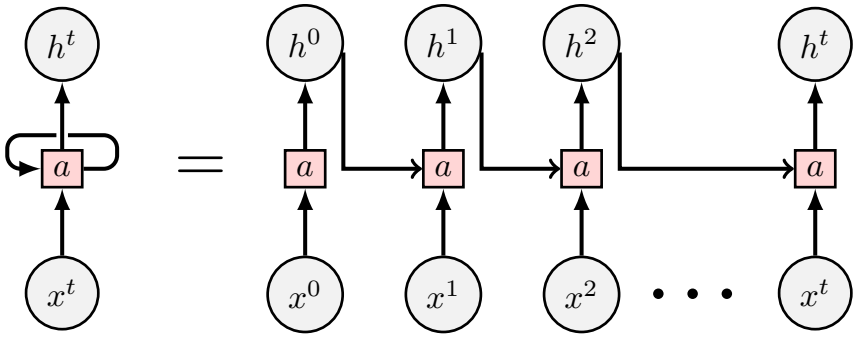
The most basic example of an RNN is depicted in Figure 3.10. Intuitively, one may think that, at timestep t, the RNN cell must learn a tensor of neural weights $\mathbf{W}$ and biases $\mathbf{B}$ such that its output is given by:

$$h^t = a((h^{t-1} \frown x^t) \otimes \mathbf{W} \oplus \mathbf{B}) \tag{3.9}$$

where a stands for an activation function (usually the hyperbolic tangent) and $\frown$ denotes the concatenation of the previous output with the current input. Note that we deliberately do not assign timestep subscripts to both $\mathbf{W}$ and $\mathbf{B}$. As we stack more and more cells (right side of Figure 3.10), we still rely on the same tensors of weights $\mathbf{W}$ and biases $\mathbf{B}$ to compute an output h^t (parameter sharing).

Allegedly there could be no stacked cells – one could implement a full RNN as a single cell with feedback which runs throughout any t number of inputs. Nevertheless, from an architectural perspective it is useful to **unroll** an RNN into a series of repeated and connected cells, thus allowing the backpropagation algorithm (Section 3.3) to perform gradient descent in the same fashion as it does to feed-forward ANNs.

Figure 3.10: Basic cell of a simple RNN (left of the equation) and its unrolled version over t timesteps. Note that, in the single cell, the self-loop corresponds to the output of the block in a previous timestep and h_t to its current output. Also, at $t = 0$ the second input is usually a vector of 0s. Source: author.



Unfortunately, in practice, a simple RNN model as described so far has shown several drawbacks in regards to learning long-term dependencies (BENGIO; SIMARD; FRASCONI, 1994) or, large temporal gaps between relevant information and where it is needed. To overcome such drawback, Hochreiter and Schmidhuber (1997) proposed a modified RNN called Long Short-Term Memory (LSTM) whose cell's structure is far more complex than a single activation function, weights and biases, such as the one presented by the original RNN.

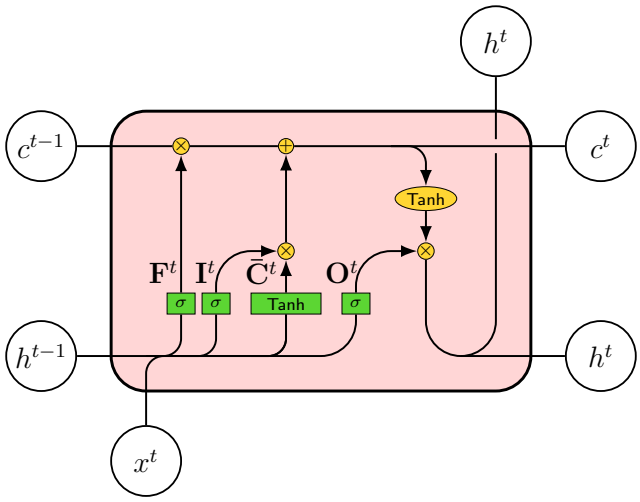
An LSTM cell (Figure 3.11) has not only an input x^t and an output h^t but also an internal state c^t, whose information can be deleted, augmented or even remain unchanged throughout the timesteps. Inside an LSTM cell, there are four neural layers, usually called **gates**, each one is accountable for the following operations:

- In the $\mathbf{F}$ gate (forget) the LSTM decides what will be removed from its internal state c^t, given the input x^t and its previous output h^{t-1}.
- In the $\mathbf{I}$ gate (input) the LSTM decides which parts of the input x^t will be written to its internal state c^t.
- In the $\bar{\mathbf{C}}$ gate (transform) the LSTM transforms the current input x^t prior effectively writing it to the internal state c^t.
- Finally, in the $\mathbf{O}$ gate (output) the LSTM decides which parts of its hidden state will be sent to the next timestep or to the final output.

Each of these gates is an independent fully-connected layer with its own weights and biases and an appropriate activation function. Precisely, the ability of turning off any

changes to its internal state is what allows LSTMs to learn long-term dependencies, this is accomplished by a correct composition of $tanh$ and σ-activated layers.

Figure 3.11: An LSTM cell at timestep t composed by neural layers (green), element-wise operations (yellow) and inputs and outputs (circles). Source: author.



We do employ LSTMs as building blocks of our GNN architecture, as it will be shown later, but the main take-away of this section is that parameter sharing can also enhance learning over sequential data, when the model has to learn both temporal dependency and the abstract concept of ordering – RNNs present invariance properties related to the temporal domain, such as invariance to time rescaling or to any kind of time warping (MATCOVSCHI; PASTRAVANU, 2009; TALLEC; OLLIVIER, 2018; WANG, 2018). Such properties are particularly useful for Natural Language Processing (NLP), the research area where RNNs achieved their most prominent successes, as in (MELIS; KOCISKÝ; BLUNSOM, 2019) for language modeling, (NALLAPATI; ZHAI; ZHOU, 2017) for summarization and (FOLAND; MARTIN, 2017) for semantic parsing. All in all, by reusing its weights sequentially, RNNs in general leverage a parameter sharing property analogous to that of CNNs, but in a different domain.

3.6 Graph Neural Networks

We have described neural modules whose parameter sharing strategies enable spatial and temporal invariance, CNNs and RNNs, respectively. In other words, while RNNs leverage on neighboring information along the 1-dimensional time domain, CNNs take advantage of local neighboring information along n-dimensional grid-shaped data.

Under these assumptions, still there is a lack of understanding on how symbolic/relational problems would fit into an ANN model, such as MLPs, CNNs or RNNs. First of all, such problems are prone to *permutation* invariance: the order of their variables is meaningless – for instance, the following two CNF SAT problems are equivalent: $(x_1 \vee x_3) \wedge (x_2 \vee \neg x_5) \equiv (x_2 \vee \neg x_5) \wedge (x_3 \vee x_1)$. Second, due to its relational structure, these problems are naturally mapped onto graphs whose structure may be not as regular as grid-shaped data – traditional approaches deal with graphs by flattening them in some form prior to feeding them to an ANN, potentially losing topological information (GORI; MONFARDINI; SCARSELLI, 2005). Joining these two properties, we may envision a neural model whose input is indeed a graph and one of its capabilities is to identify isomorphic[6] graphs as well as the Weisfeiler-Lehman test does (WEISFEILER; LEHMAN, 1968) – this is particularly important because it is not desirable to encode a single problem into all of its equivalent representations. We already discussed that parameter sharing enables invariance over some domains. In this graph domain, we need the neural kernels of the model to be repeatedly applied over the entire graph representation of a problem (as a CNN kernel does over an image). Such model is the **Graph Neural Network**, which is able to fulfill both of these requirements (XU et al., 2019).

Besides having a kernel with tied weights across the computation for each vertex, there are three other requirements a general GNN must satisfy:

1. The information accumulated for each vertex must come from its neighborhood – this indicates some kind of message-passing algorithm enriched with adjacency information.

2. The prior requirement also imposes that an internal memory or state[7] for vertices is needed.

3. To enforce that the communication flow along the graph, such algorithm must execute over t timesteps, where ideally t is equal to the diameter of the graph.

Then, given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$[8] where $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$, this basic GNN architecture would have a time-labeled d-dimensional state for each vertex

[6] Two graphs $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ and $\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ are isomorphic if there is a bijection $f : \mathcal{V}_\mathcal{G} \to \mathcal{V}_\mathcal{H}$ such that $(u, v) \in \mathcal{E}_\mathcal{G} \iff (f(u), f(v)) \in \mathcal{E}_\mathcal{H}$.

[7] Henceforth we will use state, embedding and projection interchangeably, and all of them can be seen as a real-valued vector $\mathbf{V} \in \mathbb{R}^d$.

[8] This definition can be easily generalized to a directed graph, but we keep it as simple as possible since GCP instances are usually defined over undirected graphs.

(also called vertex embedding):

$$\mathbf{V_i}^t \in \mathbb{R}^d \qquad \forall i \in 1, \cdots, n \tag{3.10}$$

and a function ϕ that takes a vertex current state and its neighbors states to produce the vertex next state:

$$\mathbf{V_i}^{t+1} = \phi\left(\mathbf{V_i}^t, \left\{\mathbf{V_j}^t \;\middle|\; (i,j) \in \mathcal{E}\right\}\right) \qquad \forall i, j \in 1, \cdots, n \tag{3.11}$$

Note that instead of simply using the states of each neighbor, we may need to translate each neighbor state into a "message" via any desired function μ. That is, a vertex state $\mathbf{V_i}^t$ is translated to $\mathbf{M_i}^t$ by μ, then $\mathbf{M_i}^t$ is sent to all its neighbors. Another issue is related to the accumulated set of messages that arrive to each vertex at each timestep (second argument of ϕ in Equation 3.11). To be used as an input for a DL module, this set of messages must be summarized in a way that its size is fixed regardless the amount of neighbors of each vertex. Thus, a simple concatenation is not allowed. Instead, we may use *sum* or even *average* operations, but the second option is known to fail at capturing structures of unlabeled graphs (XU et al., 2019). Finally, we modify Equation 3.11 to:

$$\mathbf{V_i}^{t+1} = \phi\left(\mathbf{V_i}^t, \sum_{(i,j)\in\mathcal{E}} \mu\left(\mathbf{V_j}^t\right)\right) \qquad \forall i, j \in 1, \cdots, n \tag{3.12}$$

Each of these functions, ϕ and μ, is applied to each vertex $V_i \in \mathcal{V}$, over t_{max} timesteps. This GNN basic skeleton will henceforth be referred to as **GNN-Basic** and can be seen in its complete version in Algorithm 1. Moreover, both ϕ and μ functions could be learned and thus designed as Multilayer Perceptrons (MLPs), but as ϕ is assigned with performing a computation over sequential data (sequence of messages over time), it is appropriate to implement it as an RNN, where the RNN's hidden state corresponds to $\mathbf{V_i}^t$.

Algorithm 1 GNN-Basic

procedure GNN($\mathcal{G} = (\mathcal{V}, \mathcal{E})$)
  Initialize vertex embeddings $\mathbf{V_i} \in \mathbb{R}^d \qquad \forall v_i \in \mathcal{V}$
  // Over t_{max} iterations, do:
  for $t = 1 \dots t_{max}$ **do**
    // For each vertex $v_i \in \mathcal{V}$
    for $i = 1 \dots n$ **do**
      // Update vertex embedding given its state and aggregated received messages
      $\mathbf{V_i}^{(t+1)} \leftarrow \phi(\mathbf{V_i}^{(t)}, \sum_{v_j \mid (i,j) \in \mathcal{E}} \mu(\mathbf{V_j}))$
    end for
  end for
  // Return refined vertex embeddings
  return $\{\mathbf{V_i}^{(t_{max})}\}_{i=1\dots n}$
end procedure

So far, the **GNN-Basic** we introduced can be seen as a message-passing algorithm (see Figure 3.12) whose ultimate goal is to refine projections/embeddings of vertices (see Figure 3.13). There are two main issues yet to be addressed. Refining embeddings is not enough to solve any combinatorial problem: we still need to decode the embeddings into a proper solution. If their values hold any useful knowledge for a given problem, we can assume that a learnable function γ, applied to each resulting embedding or to an aggregation of them, is able to decode a valid solution, either in a categorical or in a binary format. Usually, γ can be designed as an MLP, although more complex tasks would require different DL models.

Figure 3.12: Seen as a message-passing algorithm, a basic GNN must, at each timestep, compute messages from the incoming neighborhood of a vertex and use them to update the given vertex for the next timestep. $\mathcal{N}(v)$ stands for the neighborhood of vertex v. Source: author based on a similar picture proposed by Pedro Avelar.

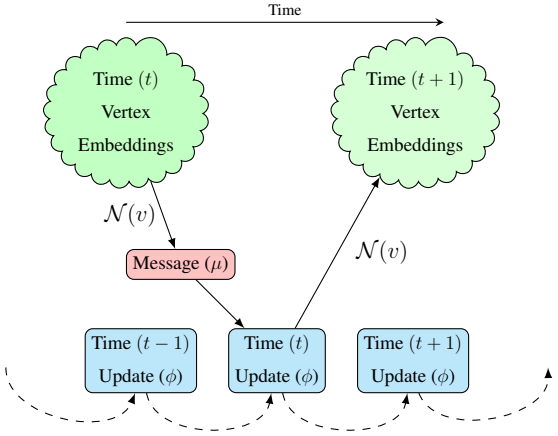
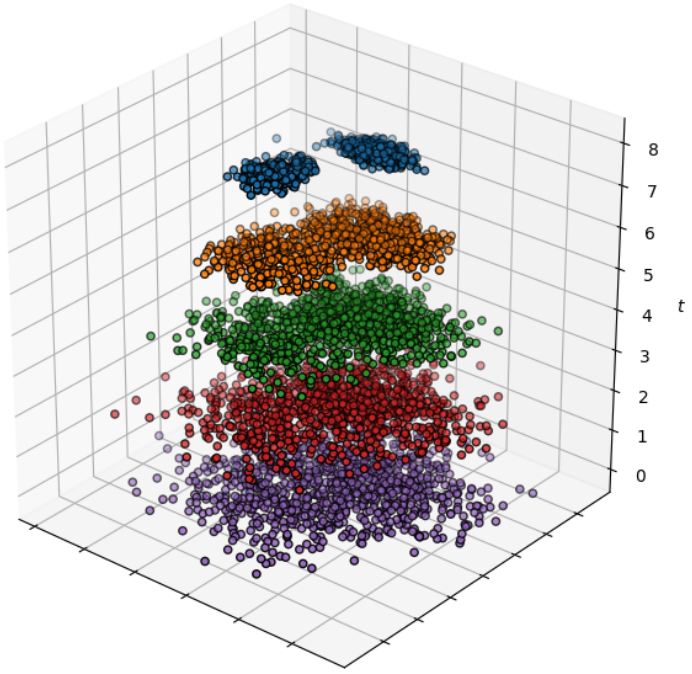


Table 3.1: Relational inductive biases raised by different neural network architectures. Source: (BATTAGLIA et al., 2018)

Component	Elements	Relations	Rel. inductive bias	Domain of invariance
Fully connected	Units	All-to-all	Weak	-
Convolutional	Grid elements	Local	Locality	Spatial
Recurrent	Timesteps	Sequential	Sequentiality	Time
Graph network	Vertices	Edges	Arbitrary	Vertex, edge

Figure 3.13: Considering 2-dimensional vertex embeddings, initially gathered from a distribution $\mathcal{P}$, a GNN should be able to refine them over t_{max} timesteps. Their final position ideally represents some property of the original problem, such as the 2-clustering depicted in this picture. Source: (PRATES et al., 2019b)



The second issue is a bit more entangled and sparked several research endeavors since the appearance of the first GNN model. A full-fledged GNN should be able to generalize its computation to any element in a graph, and even to perform specialized computation over types of elements. Such issue was already partially tackled by the first authors whom described the pioneer GNN model (GORI; MONFARDINI; SCARSELLI, 2005; Scarselli et al., 2005; SCARSELLI et al., 2009): in their work, each vertex is updated according to its own state, the states of its neighbors and the different types of relations connecting them. Particularly, Scarselli et al. (2009) explicitly stated that each "kind" k of vertex must have its own transition function ϕ_k, a function to update a vertex state given its own kind and neighbors' messages, and its own message-computing function μ_k, a function to compute the message sent by vertices of type k to their respective neighbors.

However, recent work of Battaglia et al. (2018) shed light on a new definition

for GNN-based algorithms: the **Graph Network** model. The authors claim that their formalization can "generalize and extend various approaches for neural networks which operate on graphs". They compare the parameter sharing properties of CNNs and RNNs to the one presented by GNNs, which raises the idea of **relational inductive bias** – the input's topology forces constraints on the message-passing algorithm – and the discussion on how such bias can be used in an end-to-end differentiable model with gradient descent-based learning (see Table 3.1). In regards to their model, the underlying idea is to define vertices, edges and graph states, thus augmenting the **GNN-Basic** architecture by promoting edges and the entire graph to elements in the GNN pipeline, giving them the same treatment vertices received in the original GNN (GORI; MONFARDINI; SCARSELLI, 2005). The **Graph Network**'s mechanics can be divided into three main procedures, one for each element:

$$\mathbf{e_k}^{(t+1)} \leftarrow \phi_e(\mathbf{e}_k^{(t)}, \mathbf{v}_{v_k}^{(t)}, \mathbf{v}_{u_k}^{(t)}, \mathbf{u}^{(t)}) \qquad \forall k \in 1, ..., m \tag{3.13}$$

which stands for the update on edge embeddings, considering their current state, their two endpoints embeddings and the current global state.

$$\mathbf{v}_i^{(t+1)} \leftarrow \phi_v(\mathbf{v}_i^{(t)}, \mu(\{\mathbf{e}_k^{(t+1)} \mid \mathbf{e}_k = (i,j)\}), \mathbf{u}^{(t)}) \qquad \forall i \in 1, ..., n \tag{3.14}$$

where the vertices embeddings updates are computed given their current state, the output of a function μ which aggregates all messages sent by its edges[9] and the global state.

$$\mathbf{u}^{(t+1)} \leftarrow \phi_u(\mathbf{u}, \{\mathbf{v}_i^{(t+1)}\}_{\forall i \in 1...n}, \{\mathbf{e}_k^{(t+1)}\}_{\forall k \in 1...m}) \tag{3.15}$$

which finally computes the updated global state, considering all edges and all vertices embeddings inside the graph.

This generalization is quite powerful but it is still lacking expressiveness: it is powerful due to its flexibility – with it, one may approach a combinatorial problem from a vertex, edge or even a global perspective, this is useful for problems whose edges are labeled, such as the Traveling Salesperson Problem, or problems assigned with computing some global property of a given graph, for instance its global clustering

[9]Please note that we are considering an undirected graph, where, given that $\mathbf{e}_k = (i,j)$ exists, then $\mathbf{e}_k = (j,i)$ also exists. But any equation can be easily adapted to a directed graph.

coefficient. However, besides the criticism about its mandatory presence of edges and global states (even when the problem does not require them), the **Graph Network** model lacks expressiveness for two main reasons: it is unable to process **hypergraph** structures, since its edge definition only allows two endpoints, and has no direct translation to problems demanding more than one global feature. The first issue refrains one from using the **Graph Network** model to solve SAT problems with more than 2 literals per clause[10]. The second issue is related to the global graph state – a state which is connected to every other element in the graph, and prevents the decision version of GCP to be tackled by a **Graph Network** as there is no way to define k color/global states for any $k > 1$.

To define a full-fledged GNN formalization, Prates et al. (2019b) expand on the idea of types of vertices. We propose a **Typed Graph Network** (TGN) model whose embeddings are separately defined for each type τ_N of vertex and each type K of connection. In practice, in a TGN model each element in the graph is framed as a vertex of a given type: for instance, the **Graph Network** (BATTAGLIA et al., 2018) can be seen as a TGN with a set vertices of type *edge*, a set of vertices of type *vertex* and an 1-sized set of vertices of type *global*. Note that the authors also expand on the definition of edge, in this sense, K stands for the number of types of connections (not to be misled with the vertex of type edge) between the vertices – or the amount of message functions μ (see Equation 3.12) needed to enable inter-vertex communication.

A simple TGN model to perform over N types of vertices using K message-functions can be defined by the following sets:

1. A set of N **embedding sizes** $n_1, n_2 \dots n_N$ (one embedding size per type of vertex).
2. A set of N **vertex types** $\mathcal{T} = \{\tau_i \in \mathbb{R}^{n_i} \mid \forall i \in 1 \dots N\}$.
3. A set of K **message** functions $\mathcal{M} = \{\mu_k : \tau_1 \to \tau_2 \mid 1 \leq k \leq K, \tau_1, \tau_2 \in \mathcal{T}\}$. Each combination of types (τ_1, τ_2) may have many message-functions assigned to.
4. And a set of N **update** functions $\mathcal{U} = \{\phi_i : \mathbb{R}^{n_i + D_{acc}(i)} \to \mathbb{R}^{n_i} \mid \forall i \in \{1 \dots N\}\}$, where $D_{acc}(i) = \sum_{\mu:\tau_j \to \tau_i \in \mathcal{M}} n_i$

where the message functions $\mu_k \in \mathcal{M}$ and the update functions $\phi_i \in \mathcal{U}$ are the only trainable modules of the TGN architecture. Note that each different type of vertex can be projected into a different hyperspace, as there are no constraints regarding $n_1, n_2, \cdots, n_N$.

[10]Note that each clause in a k-SAT problem can be mapped to a hyperedge connecting k literals.

Algorithm 2 Typed Graph Network Model defined by (PRATES et al., 2019b)

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

To translate from a τ_i hyperspace to τ_j hyperspace, TGN kernel may have any number of message functions $\mu_k : \tau_i \rightarrow \tau_j$.

Algorithm 2 defines the TGN full procedure in terms of two types of vertices, $\#i$ and $\#j$, but its mechanics can be generalized to any pair of types $\in \mathcal{T}$. First of all, we need to define the relevant adjacency information between any communicating types (Line 3). We also need to initialize all vertex embeddings (Line 6), this can be done in a plethora of forms, either by a simple random initialization or by enriching the initial embedding with some relevant information (e.g. for a vertex of type "edge" in a TSP problem, we may insert its cost as an initial embedding value). Finally, the TGN runs for t_{max} message-passing iterations: for each type i, all messages sent from vertices $v_b \in \mathcal{V}_j$ to vertices $v_a \in \mathcal{V}_i$ are accumulated via matrix-multiplication of the adjacency information $\mathbf{M}_k$ and its correspondent message-function μ_k. The core operation of a matrix-multiplication is a sum, thus this is a sum-aggregation of messages from neighboring vertices of type j (Line 11). Then all vertices of type i have their embeddings updated (Line 13) by ϕ_i, which takes into account their current embeddings and the sum of all messages sent to them by vertices of all types.

We provided this somewhat in-depth explanation of TGN not only because we chose to frame the GCP in a TGN model but mainly to show that TGN has all components a combinatorial problem may require. TGN is able to generalize not only the Graph Networks (BATTAGLIA et al., 2018), but also several others (ad-hoc and general) GNN

architectures, such as Message Passing Neural Networks (GILMER et al., 2017), Relational Networks (RAPOSO et al., 2017; SANTORO et al., 2017), Deep Sets (ZAHEER et al., 2017), Non-Local Neural Networks (WANG et al., 2018) and Independent Recurrent Blocks (SANCHEZ-GONZALEZ et al., 2018).

It is also important to mention that, even though we have described the GNN general algorithm as a message-passing one, where each vertex can be seen as a stateful element which exchanges information with neighboring elements, such as envisioned by (GILMER et al., 2017), there is also a convolutional perspective (DUVENAUD et al., 2015; KIPF; WELLING, 2017) upon which a GNN operates a convolutional kernel along the entire graph, identifying features according to local neighborhood and accumulating them over time.

Finally, although other DL models have been applied to combinatorics in recent years, they are usually engineered as an ancillary module to a proper combinatorial algorithm, such as a local search procedure, that in fact produces the output. In this perspective, one may highlight the well-known algorithm AlphaGo (SILVER et al., 2018), which defeated Go's World Champion by coupling a deep neural network with a Monte Carlo tree search – while the former evaluates the current state of a Go's board, the latter is fed with the evaluation and determines the best move available. Strategies such as the one employed by Alpha Go are not end-to-end differentiable – gradient descent learning is not feasible to the whole pipeline – and thus are prone to human bias. On the other hand, end-to-end learning over combinatorics – powered by GNN-based models – are free from biases, as there are no algorithms hard-coded, and also very promising given that the feasibility of "printing" a graph structure into a connectionist architecture may enable several groundbreaking insights on how machines reason over given inputs to produce a desired solution. As of now, besides GNN models designed to solve NP-Complete problems (SELSAM et al., 2019; PRATES et al., 2019a), which are very related to this dissertation, there have been several state-of-the-art pushes achieved by GNNs in other areas, namely language modeling (SANTORO et al., 2017), link prediction (NATHANI et al., 2019), human-object interaction (QI et al., 2018), program verification (LI et al., 2016), explainable recommender systems (MONTI; BRONSTEIN; BRESSON, 2017) and many others.

3.6.1 NeuroSAT

We mentioned earlier that GNN models may foster novel ways of understanding and solving combinatorial problems. Perhaps the first glimpse of such potential was achieved by the NeuroSAT (SELSAM et al., 2019). The satisfiability problem (SAT) consists in evaluating if any truth-values assignments to variables in an expression E leads to its satisfiability (E is true). Usually the expression is represented by a conjunctive normal formula – a set of clauses connected by disjunctions, such that each clause contains literals connected by conjunctions. SAT plays a fundamental role in theoretical computer science, it was the first problem proved to be NP-Complete (COOK, 1971) and it is often used to verify the NP-Completeness of a different problem $\mathbb{P}$: we basically need to verify that $\mathbb{P}$ is in NP and that SAT can be reduced to $\mathbb{P}$. Moreover, the opposite reduction (P to SAT) allowed SAT exact solvers to be of paramount importance in regards to a broad range of problems, going from automated theorem proving to model checking and circuit design.

In regards to their usability in DL models, SAT problems can be encoded in a one-hot vector and used as input in a RNN-based model, for instance. However, several properties of propositional logic would be lost, particularly related to invariances of permutation and negation. NeuroSAT's underlying idea is to represent a CNF expression as a graph: one node per literal, one node per clause, an edge connecting a literal to every clause containing it and another edge connecting a literal and its negated version. In its GNN-based architecture, NeuroSAT assigns embeddings to literals and to clauses and refines them over t timesteps of message-passing, where each clause receives an update based on the messages sent by its literals and each literal receives an update based on the messages sent by its clauses as well as by its negated variants.

Bringing this idea to a TGN definition, there are two types of nodes, C for clauses and L for literals, and each type is projected into a different hyperspace, thus leading to the necessity of two update functions, ϕ_C and ϕ_L. Moreover, according to the adjacency information we described in the last paragraph, NeuroSAT requires two message-functions, μ_{LC} which translates literals embeddings into messages to their respective clauses, and μ_{CL} that translates clauses embeddings into messages to their literals. Note that each literal also receives information from its negated variant, which yields the following update

equations (analogous to Lines 11 and 13 of Algorithm 2):

$$\begin{aligned} \mathbf{V_L}^{(t+1)} &\leftarrow \phi_L(\mathbf{V_L}^{(t)}, \mathbf{M_{CL}} \times \mu_{CL}(\mathbf{V_L}^{(t)}), \mathbf{M_{LL}} \times \mathbf{V_L}^{(t)}) \\ \mathbf{V_C}^{(t+1)} &\leftarrow \phi_C(\mathbf{V_C}^{(t)}, \mathbf{M_{CL}}^{\intercal} \times \mu_{LC}(\mathbf{V_L}^{(t)})) \end{aligned} \tag{3.16}$$

Equation 3.16 can be separately interpreted as: 1) to update literal embeddings $\mathbf{V_L}$, ϕ_L takes into account the current literal embeddings, a tensor of accumulated messages computed by μ_{CL} multiplied by the adjacency information between clauses and literals $\mathbf{M_{CL}}$ and a tensor of unchanged[11] literal embeddings multiplied by the adjacency information between literals and negated literals $\mathbf{M_{CL}}$. These matrix multiplications between messages/unchanged embeddings and adjacency information are needed to correctly mask the input to the update function. In other terms, μ_{CL}, for instance, computes the message from all clauses to all vertices, but not all of them are effectively connected, thus it is paramount to mask these resulting computations; 2) ϕ_C updates clause embeddings given their current values and the aggregated messages sent by literals – note that here again a adjacency information is multiplied by those messages to enforce neighboring constraints. NeuroSAT implements μ_{CL} and μ_{LC} as MLPs and ϕ_C and ϕ_L as LSTMs[12]

NeuroSAT was trained upon batches containing pairs of SAT/UNSAT instances whose only difference was the polarity of a single literal in a single clause, thus leading to almost indistinguishable instances. These pairs of instances were generated by progressively adding clauses (with k literals whose polarity probability was 50%) to a CNF formula until it becomes unsatisfiable. Moreover, NeuroSAT was trained in a supervised single-bit fashion: each instance was associated to a label $y_i \in \mathbb{B} = \{0, 1\}$. NeuroSAT was able to achieve around 85% of accuracy during test but its most impressive feature is related to the knowledge encoded by its internal embeddings. Although having been trained only to solve a decision problem – NeuroSAT's output is effectively a binary answer decoded by an MLP at the end of its pipeline – Selsam et al. (2019) were able decode valid assignments to 70% of their test instances by 2-clustering literal final embeddings $\mathbf{V_L}^{(t_{max})}$. As any problem in NP can be reduced to SAT, the authors also sampled instances from other combinatorial problems, including k-coloring $k \in \{3, 4, 5\}$, reduced them to SAT and trained NeuroSAT upon them. Particularly regarding the k-coloring problems, NeuroSAT was able to decode valid assignments for 64%, 69% and 54% of the satisfiable instances, for $k \in \{3, 4, 5\}$, respectively.

[11] Unchanged means that these embeddings do not undergo any message-function

[12] Note that we omit LSTMs hidden states in the Equation to improve readability.

4 TYPED GRAPH NETWORKS FOR THE GRAPH K-COLORING PROBLEM

In this chapter we will present and provide an in-depth discussion of the main subject of this dissertation: a graph neural network designed to solve the k-coloring problem on graphs. As seen in Section 3.6, GNN-based models assign multidimensional representations, or embeddings $\in \mathbb{R}^d$, to vertices and edges. These embeddings communicate between themselves according to some adjacency information throughout a given number of message-passing iterations. Such meta-architecture is quite appropriate to symbolic domains as its neural modules can be engineered in various configurations, each one reflecting a graph representation of a given instance of the problem (SCARSELLI et al., 2009).

One of the most successful GNN research endeavors in symbolic domains is the NeuroSAT (SELSAM et al., 2019), which solved SAT problems with a very good accuracy and decoded valid assignments from its internal states. This could suggest that the NeuroSAT architecture is the ultimate GNN-based architecture for combinatorial problems in NP, but in this dissertation we investigate if an architecture specifically designed to the GCP itself, with no need of prior polynomial-time reductions, can at least have the same performance NeuroSAT achieved when it was fed with GCP instances (reduced to SAT). It is also worth mentioning that the original NeuroSAT was trained and tested with k-coloring ($k \in \{3, 4, 5\}$) instances whose number of vertices was equal to 10, thus its usability and generalizing features are not yet well-established.

The next section will provide the details of our model (**GNN-GCP**), while Sections 4.2 and 4.3 will describe the training methodology and provide a brief description of the baselines, respectively. Finally, in Section 4.4 we will show the **GNN-GCP** performance on several scenarios and details on how it may be reasoning over the graph coloring problem.

4.1 Our Model

GNN-based models can be defined by which graph elements they allow to communicate and how such communication is performed. In general, graph neural network models assign multidimensional embeddings $\in \mathbb{R}^d$ to vertices. These embeddings are then refined according to some adjacency information throughout a given number of message-passing iterations. The adjacency information controls which are the valid incoming messages

for a given vertex, these filtered messages undergo an aggregating function and finally an update function receives the aggregated messages and computes the embedding update for the given vertex. The Graph Network model (BATTAGLIA et al., 2018) also allows the instantiation of global graph attributes which seems useful to the k-colorability problem. In our case, however, we choose to treat each possible color as a global attribute, thus there must be multiple global graph attributes, i.e. each color has its own embedding – which essentially means that we "promote" colors to a vertex level. Because of that, we chose to model the k-colorability problem in the Typed Graph Network (TGN) framework (PRATES et al., 2019b) (Section 3.6, Algorithm 2), a formalization that leverages the capability of coping with several types of nodes and that was built upon the seminal GNN model (GORI; MONFARDINI; SCARSELLI, 2005).

Given a GCP instance $I = (\mathcal{G}, C)$ composed of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a number of colors $C \in \mathbb{N} \mid C > 2$, each color is associated with a random initial embedding $\in \mathbb{R}^{d_c}$ over an uniform distribution $\mathbf{C}^{t=0}[i] \sim \mathcal{U}(0,\, 1) \mid \forall i \in C$ and each vertex is assigned to the same embedding $\in \mathbb{R}^{d_v}$ which is initially sampled from a normal distribution $\mathbf{V}^{t=0}[i] \sim \mathcal{N}(0,\, 1) \mid \forall i \in \mathcal{V}$. Following the procedure of Prates et al. (2019a), we make this randomly initialized common vertex embedding a trained parameter learned by the model. To enable the communication among neighboring vertices and among vertices and colors, besides the vertex-to-vertex adjacency matrix $\mathbf{M}_{\mathcal{V}\mathcal{V}} \in \{0, 1\}^{|\mathcal{V}|\times|\mathcal{V}|}$, our model also requires a vertex-to-color adjacency matrix $\mathbf{M}_{\mathcal{V}\mathbf{C}} \in \{1\}^{|\mathcal{V}|\times|\mathcal{C}|}$, that connects each color to all vertices – this means that no prior information is given to the model and any vertex can be assigned to any color. After this initialization, adjacent vertices and colors communicate and update their embeddings throughout t_{max} rounds of message-passing. Then the resulting vertex embeddings are decoded by a Multilayer Perceptron (MLP) which computes a logit probability corresponding to the model's prediction of the answer to the decision problem: "does the graph G accept a C-coloration?". This procedure is summarized in Algorithm 3 and the overall view of our architecture can be seen in Figure 4.1.

In a nutshell, our proposed neural-symbolic model learns the following seven tasks:

- To generate a single $\mathbb{R}^{d_v}$ vector, used to initialize all vertex embeddings.
- A function $C_{msg} : \mathbb{R}^{d_c} \rightarrow \mathbb{R}^{d_v}$ that computes a message to a vertex given a color embedding. Implemented by an MLP.
- A function $V_{msg} : \mathbb{R}^{d_v} \rightarrow \mathbb{R}^{d_c}$ that computes a message to a color given a vertex

Algorithm 3 Graph Neural Network Model for GCP

procedure GNN-GCP($\mathcal{G} = (\mathcal{V}, \mathcal{E}), C$)

// Compute binary vertex-to-vertex adjacency matrix

$\mathbf{M}_{\mathcal{V}\mathcal{V}}[i, j] \leftarrow 1$ iff $(\exists e \in \mathcal{E} | e{=}(v_i, v_j)) | \; \forall v_i {\in} \mathcal{V}, v_j {\in} \mathcal{V}$

// Compute all-ones vertices-to-colors adjacency matrix

$\mathbf{M}_{\mathcal{V}\mathbf{C}}[i, j] \leftarrow 1 \forall v_i {\in} \mathcal{V}, c_j {\in} C$

// Compute initial vertex embeddings

$\overset{(1)}{\mathbf{V}}[i] \sim \mathcal{N}(0,\, 1) \mid \forall i \in \mathcal{V}$

// Compute initial color embeddings

$\overset{(1)}{\mathbf{C}}[i] \sim \mathcal{U}(0,\, 1) \mid \forall i \in \mathcal{C}$

// Run t_{max} message-passing iterations

for $t = 1 \ldots t_{max}$ **do**

// Refine each vertex embedding with messages received from its neighbors and candidate colors

$\overset{(t+1)}{\mathbf{V}_h}, \; \overset{(t+1)}{\mathbf{V}} \leftarrow V_u(\overset{(t)}{\mathbf{V}_h}, \mathbf{M}_{\mathcal{V}\mathcal{V}} {\times} \overset{(t)}{\mathbf{V}}, \mathbf{M}_{\mathcal{V}\mathbf{C}} \times \underset{msg}{C}(\overset{(t)}{\mathbf{C}}))$

// Refine each color embedding with messages received from all vertices

$\overset{(t+1)}{\mathbf{C}_h}, \; \overset{(t+1)}{\mathbf{C}} \leftarrow C_u(\overset{(t)}{\mathbf{C}_h}, \mathbf{M}_{\mathcal{V}\mathbf{C}}{}^T {\times} \underset{msg}{V}(\overset{(t)}{\mathbf{V}}))$

end for

// Translate vertex embeddings into logit probabilities

$V_{logits} \leftarrow V_{vote}\left(\overset{t_{max}}{\mathbf{V}}\right)$

// Average logits and translate to probability (the operator $\langle\rangle$ indicates arithmetic mean)

prediction $\leftarrow$ sigmoid($\langle \mathbf{V_{logits}} \rangle$)

end procedure

embedding. Implemented by an MLP.

- A function $V_u : \mathbb{R}^{3d_v} \rightarrow \mathbb{R}^{d_v}$ to compute an updated vertex embedding and an updated RNN hidden state given the current RNN hidden state, the embeddings of neighboring vertices and a message sent by neighboring colors. Implemented by an LSTM.

- A function $C_u : \mathbb{R}^{2d_c} \rightarrow \mathbb{R}^{d_c}$ to compute an updated color embedding and an updated RNN hidden state given the current RNN hidden state and a message sent by all vertices. Implemented by an LSTM.

- And finally, a function $V_{vote} : \mathbb{R}^{d_v} \rightarrow \mathbb{R}^1$ to compute logit probabilities given each vertex embedding. Implemented by an MLP. We transform this unbounded value

into a proper probability by applying a sigmoid function to the outputs of the MLP V_{vote}.

We chose to design such neural-symbolic model under the Typed Graph Network framework, in the sense that we leverage TGN's vertex typing features to insert the colors into the graph and to allow them to communicate with "real" vertices. The TGN framework[1] requires definitions of four main elements: which graph elements will have embeddings (in our case, vertices and colors), the adjacency matrices between each type of vertex, the message-functions and the update-functions. We provide a Python dictionary containing such definitions in Appendix A.

Lastly, our model updates vertex and color embeddings, along with their respective hidden states, according to the following equations:

$$\begin{aligned}\mathbf{V}^{(t+1)}, \mathbf{V}_h^{(t+1)} \leftarrow \mathcal{V}_u(\mathbf{V}_h^{(t)}, \mathbf{M}_{\mathcal{V}\mathcal{V}} \times (\mathbf{V}^{(t)}), \\ \mathbf{M}_{\mathcal{V}\mathbf{C}} \times \underset{msg}{C}(\mathbf{C}^{(t)}))\end{aligned} \tag{4.1}$$

$$\mathbf{C}^{(t+1)}, \mathbf{C}_h^{(t+1)} \leftarrow \mathcal{C}_u(\mathbf{C}_h^{(t)}, \mathbf{M}_{\mathcal{V}\mathbf{C}}{}^{\intercal} \times \underset{msg}{V}(\mathbf{V}^{(t)})) \tag{4.2}$$

Note that, following NeuroSAT (SELSAM et al., 2019), to update vertex embeddings we use computed messages from a different type of vertex (colors) and the raw embeddings of vertex of the same type (vertices) – we do not employ a function to produce a message sent by a vertex to another vertex. For a given vertex, the second argument of V_u is simply a sum-aggregated embedding of all its vertex neighbors, whereas the third argument is a sum-aggregated message computed over its color neighbors. As we implement V_u as an RNN (specifically an LSTM), its own hidden state acts as a memory of its current embedding (first argument). On the other hand, to update color embeddings C_u (also an LSTM) only uses its hidden state (current embedding of a color) and sum-aggregated messages from adjacent vertices.

4.2 Training Methodology

To train these message computing and updating modules, MLPs and RNNs respectively, we used a Stochastic Gradient Descent algorithm implemented via TensorFlow's

[1] Available at: <https://github.com/machine-reasoning-ufrgs/typed-graph-network>

Adam optimizer (KINGMA; BA, 2014). We defined as loss the binary/sigmoid cross entropy between our model final prediction and the ground-truth for a given GCP instance – a boolean value indicating that the graph accepts (or not) a target C number of colors. The MLPs C_{msg} and V_{msg} are three-layered $(64, 64, 64)$ with ReLU nonlinearities as the activations for all layers except for the linear activation on the output layer. And the RNN cells C_u and V_u are basic LSTM cells with layer normalization and ReLU activation. We trained our model using a learning rate of 2×10^{-5}, 32 iterations of message-passing, i.e. $t_{max} = 32$, and $d_c = d_v = 64$ for the size of embeddings.

When it comes to the GCP instances, following (SELSAM et al., 2019) and (PRATES et al., 2019a) we intended to train our model with very hard graph coloring problems. To produce such instances, we leverage the phase transition phenomenon briefly described in Section 2.3. In this sense, the most obvious procedure to generate GCP instances would consist in gradually adding edges to an initial empty graph until it becomes unsatisfiable for a given number of colors. We found such procedure to be very slow and added a preprocessing stage to define initial parameters for generating GCP graphs. First, we set the candidate number of colors $c \sim \mathcal{U}\{3, 4, \cdots, 7\}$ and the number of vertices $n \sim \mathcal{U}\{40, 41, \cdots, 60\}$, for each value of n we generated 500 graphs whose edge probability $p \sim \mathcal{U}\{10, 20, \cdots, 90\}$. Then, each graph was fed to a CSP-Solver[2] together with each value of c. With that we were able to identify which ranges of p and n yielded, in average, a balanced number of satisfiable and unsatisfiable instances, that is, when the CSP-Solver only solved around 50% of the instances for a given n and a given p. We used such combinations of n and p as our initial parameters to produce training and test instances.

With regards to training instances, besides being very difficult to solve, we also wanted them to be fed to our model in an adversarial fashion, i.e. two very similar instances with opposite labels are fed to the model in the same batch. Formally, we may define such procedure in the following way: let χ be the chromatic number, i. e. the smallest value to obtain a valid coloring, then for each positive instance $I = (G = (\mathcal{V}, \mathcal{E}), C)$, there is also an adversarial negative instance $\overline{I} = (G' = (\mathcal{V}, \mathcal{E}'), C)$ such that $\mathcal{E} \neq \mathcal{E}'$ only for a single edge (v_i, v_j), thus $C = \chi(G) = \chi(G') - 1$. This edge is usually called *frozen edge* since for every valid coloring C of G, then $C[v_i] = C[v_j]$, which implies that this edge cannot belong to any C-colorable graph containing G as a subgraph (CULBERSON; GENT, 2001). To produce a pair of these instances, we randomly chose a target chromatic number

[2]<https://developers.google.com/optimization/cp/cp_solver>

χ between 3 and 7 and $n \sim \mathcal{U}\{40, 41, \cdots, 60\}$, then we populate the adjacency matrix $\mathbf{M}_{\mathcal{VV}}$ with a probability p adjusted to the selected chromatic number and n combination, as described before. Finally, a CSP-Solver is used to ensure that the undirected graph represented by the initial matrix $\mathbf{M}_{\mathcal{VV}}$ has a chromatic number χ: if it has, then we proceed adding edges[3] to the graph until the CSP-Solver is no longer able to solve the GCP for χ. The last two generated instances were added to the dataset (both having $C = \chi$), ensuring that the dataset is not only composed by hard instances but also perfectly balanced: 50% of the instances do not accept a C-coloring while the remaining 50% do accept. A total of 2×2^{15} such instances were produced.

These instances were randomly joined into a larger graph during training to produce a batch-graph containing 2×8 instances. This was done with a disjoint union so that the resulting $\mathbf{M}_{\mathcal{VV}}$ and the $\mathbf{M}_{\mathcal{VC}}$ of a batch do not allow communication between vertices and colors of different subgraphs. Therefore, we ensure that despite being on the same batch-graph, there is no inter-graph communication. The logit probabilities computed for each vertex within the batch are separated and averaged according to which instance the vertex belongs to. Finally, we calculated the binary cross entropy between these predictions and their instances labels.

Figure 4.1 also depicts how a toy problem of graph coloring (left side) would be processed within our proposed model. Such toy problem has $\mathbf{M}_{\mathcal{VC}}$ with shape $(3, 2)$ and $\mathbf{M}_{\mathcal{VV}}$ with shape $(3, 3)$. We will assume $d_c = d_v = d$ for simplicity and to perfectly emulate the model we trained. In this sense, initially the GNN instantiates two internal memories: V with shape $(3, d)$ and C with shape $(2, d)$. The first message-passing iteration then begins by each of the MLPs V_{msg} and C_{msg} computing, in parallel, messages from each vertex to all colors and from each color to all vertices, respectively. Thus, the output of V_{msg} has shape $(3, d)$ – each vertex send a message of size d to colors – and the output of C_{msg} has shape $(2, d)$ to perform the analogous operation. Then, the model needs to leverage its adjacency information: the transpose of $\mathbf{M}_{\mathcal{VC}}$ is multiplied by the output of V_{msg} ($(2, 3) \times (3, d)$) yielding a tensor of shape $(2, d)$ which we will call Cv_{agg} (orange arrow inside the GNN) as it is the aggregated vertex messages for each color; on the bottom level of our pipeline, we multiply $\mathbf{M}_{\mathcal{VC}}$ by the output of C_{msg}, and the result is a tensor of shape $(3, d)$, which we will refer to as Vc_{agg} (blue arrow) – also in the bottom, we use the vertex-to-vertex adjacency matrix $\mathbf{M}_{\mathcal{VV}}$ $(3, 3)$ by multiplying it with raw embeddings of vertices V $(3, d)$, yielding a tensor with shape $(3, d)$ with aggregated vertex embeddings

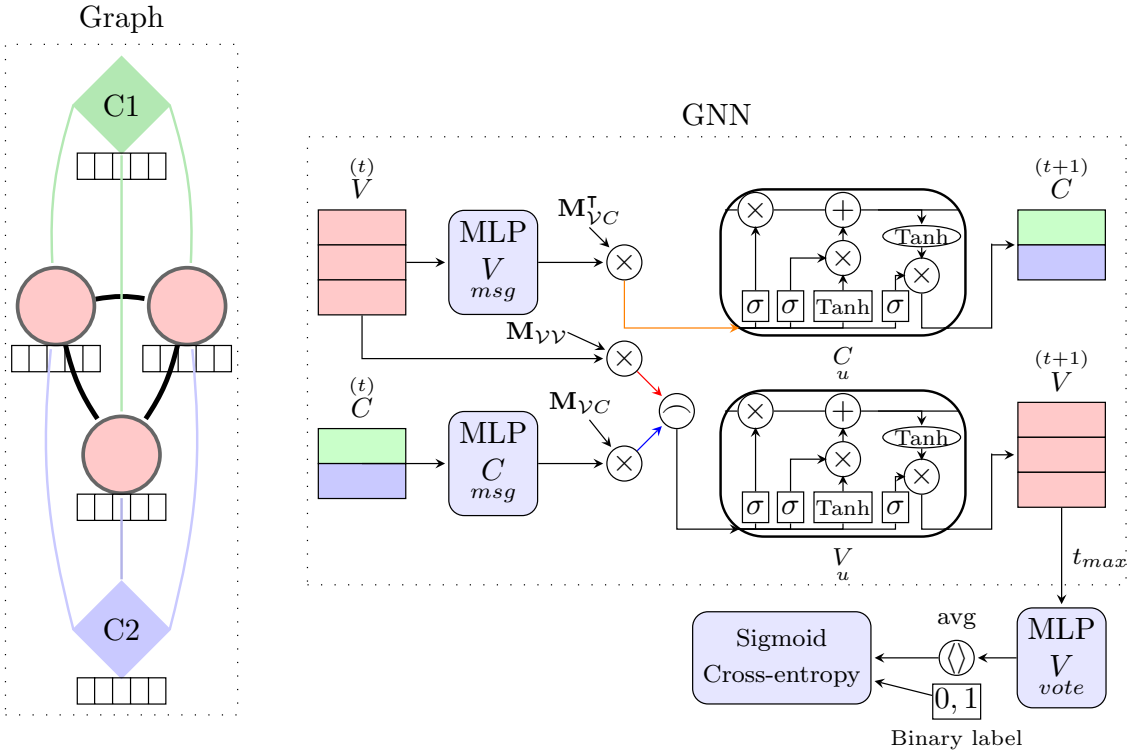
[3]Here we found that adding edges to vertices whose degree is already high speeds up the procedure.

of neighbors Vv_{agg} (red arrow). These 3 aggregations are all that our model needs to update vertex and color embeddings. To update color embeddings, the LSTM C_u uses its hidden state $(2, d)$ (not shown in the figure) and Cv_{agg}, producing a color updated tensor of shape $(2, d)$ and another hidden state to be used in the next iteration. And to update vertex embeddings, V_u is fed with its hidden state $(3, d)$ and a concatenation of Vc_{agg} with Vv_{agg} (resulting shape is $(3, 2d)$), its output has shape $(3, d)$[4] and corresponds to the updated vertex embeddings.

Finally, outside of the GNN scope, MLP V_{vote} computes a single logit probability for each vertex, whose average is fed to a sigmoid cross-entropy function to produce a loss value. Neural weights of V_{vote}, V_u, C_u, V_{msg} and C_{msg} are then updated via gradient descent. Note that the above description considers only one timestep of iteration. Thus, only direct neighbors were able to communicate. Larger information flows are enabled by increasing the number of timesteps.

[4]Note that both LSTMs receive a concatenation of its hidden state and the input. For instance, V_u actually receives a tensor of shape $(3, 3d)$ and produces (besides the next hidden state) an output of shape $(3, d)$. This change of shape is due to the shape of $\mathbf{W}$ in Equation 3.9

Figure 4.1: Overall view of our architecture. Initially, each color is mapped into a GNN internal memory $\in C$ and each vertex is mapped into an internal state $\in V$ (initial value for vertex is learned). Then, in parallel, messages are computed from all vertices to all colors and vice-versa. Messages from colors to vertices are concatenated with raw embeddings from neighboring vertices (operation $\frown$). Two LSTMs (V_u and C_u) update vertex and color embeddings, respectively. All matrix multiplications are required to enforce that adjacency information is respected prior to update a vertex or color embedding. At t_{max} we gather all vertex embeddings and feed it to an MLP, whose output is one logit probability per vertex. These probabilities are then averaged into a final answer. To produce a proper loss value, we compute the sigmoid cross-entropy between the final answer and the problem's label. We omit the LSTMs hidden and cell states to improve readability. Colored arrows inside the GNN corresponds to aggregated embeddings and messages. Source: author.

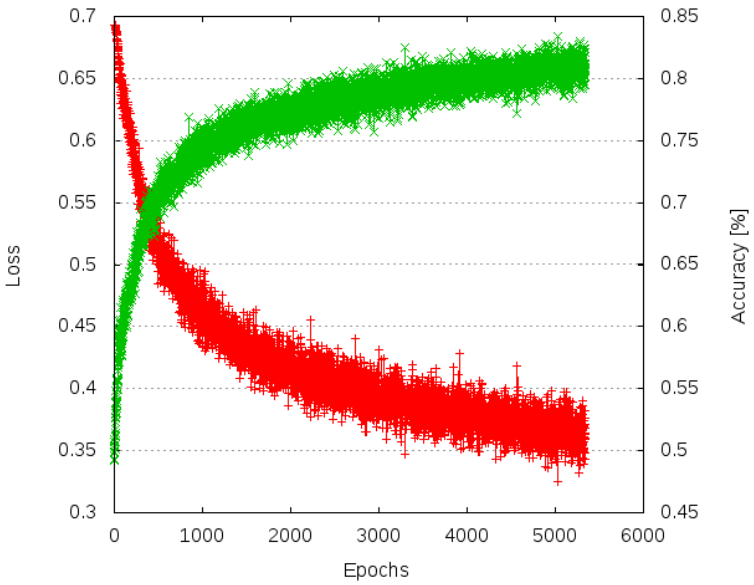


We stopped the training procedure when our model achieved around 82% of accuracy and 0.35 of Sigmoid Cross Entropy loss averaged over 128 batches containing 16 instances at the end of 5300 epochs. Figure 4.2 shows the learning evolution of our model during training.

4.3 Baselines

We compared our model against 3 other methods: two heuristics and one neural-symbolic approach. The first heuristic we chose is a **greedy** algorithm which simply assigns the first available color to a given node. Our greedy procedure to the GCP is fully

Figure 4.2: Evolution of the sigmoid cross entropy loss (red curve) and accuracy (green curve) throughout 5300 training epochs on a dataset of 2×2^{15} graphs. Note that after an epoch our model only sees 128×16 instances and the accuracy is computed regarding this number. We refrained from having an epoch containing 2×2^{15} instances due to memory constraints. Source: author.



described in Algorithm 4, its output is the minimum number of colors that yielded a valid assignment.

The second heuristic approach is **Tabucol** (HERTZ; WERRA, 1987), an algorithm based on tabu search. A tabu search, in general, can be seen as an enhancement of a local search: in a local search, an initial illegal coloring is provided and the algorithm reaches a neighboring coloring by making local changes, there is also some criterion on which neighbor will be selected – however, a simple local search tends to be stuck in sub-optimal solutions when no neighboring coloring presents a better outcome. Tabu search (GLOVER, 1989; GLOVER, 1990) improves on such strategy by keeping a record of recently visited solutions, these solutions are added to a tabu list so the algorithm will not explore them in the near future. Tabucol basically assumes that a solution/coloring c is evaluated w.r.t. internal conflicts – adjacent vertices with the same color – by a function f. For a given pivot coloring, the neighbor coloring c' whose $f(c')$ is lesser than $f(c)$ is chosen and c is added to the tabu list. There are several versions of Tabucol, the one we used is described by Algorithm 5 and, besides the input graph $\mathcal{G}$ and the number of colors C, comprises three main parameters: maximum number of iterations it_{max}, the size of the tabu list L and the amount of neighboring colorings to be evaluated rep. For our experiments, we set $it_{max} = 10000$, $L = 7$ and $rep = 100$.

After constructing a random initial solution and initializing some variables, the

Algorithm 4 Greedy coloring

```
1:  procedure GREEDY-COLORING(G = (V, E))
2:      // Create a solution vector, initially containing no valid color for each vertex.
3:      Sol[v_i] ← −1        ∀v_i ∈ V
4:      // Define a set with all possible colors (n colors are enough for a fully-connected
        graph)
5:      A ← {0, 1, ··· , n − 1}
6:      // Define an empty set of unavailable colors
7:      U ← ∅
8:      // Assign the first color to the first vertex
9:      Sol[0] ← 0
10:     // For each other vertex
11:     for i = 1 ... n do
12:         for j = 0 ... n do
13:             if (v_i, v_j) ∈ E ∧ Sol[j] ≠ −1 then
14:                 // If a neighbor is already colored by a color z, add z to the unav. set
15:                 U ← U + Sol[j]
16:             end if
17:         end for
18:         // Candidate colors for i are all avail. colors minus the unav. ones
19:         C ← A − U
20:         // Color of i is the lowest candidate color
21:         Sol[i] ← min(C)
22:         // Reset unavailable set to the next vertex
23:         U ← ∅
24:     end for
25:     // Minimal coloring returned
26:     return max(Sol) + 1
27: end procedure
```

algorithm starts by identifying candidate vertices to be changed on the next solution – i.e. vertices that, in the current solution, are part of a conflict (Lines 12 to 16). Then Tabucol selects a random neighbor of the given solution (Lines 19 and 20) and evaluate if such neighbor has less conflicts (Line 21) – if not, it will select another neighbor; if it has, then the aspiration level of the given solution may be updated and the pair (vertex, new color) is removed from the tabu list, and the algorithm goes to Line 35-37 where it adds the current solution to the tabu list and updates the current solution with the new solution. If the new solution has not achieved a better aspiration level (Line 28), it evaluates if the new solution is in the tabu list: if positive, it rejects the new solution and continues searching neighboring colorings; otherwise the new solution is accepted and Tabucol moves on to Line 35-37. Its output is a truth-value indicating if it found a valid coloring given the input number of colors.

Algorithm 5 Tabucol

1: **procedure** TABUCOL($\mathcal{G} = (\mathcal{V}, \mathcal{E}), C, it_{max}, L, rep$)
2: $Col \leftarrow [0, 1, \cdots, C-1]$
3: $Sol[v_i] \leftarrow randompick(Col) \qquad \forall v_i \in \mathcal{V}$
4: $Tabu \leftarrow queue(size = L)$
5: $it \leftarrow 0$
6: // Initially, from n conflicts we aspire to reach $n-1$ conflicts
7: $Asp \leftarrow \{n : n-1, \cdots, 1 : 0\}$
8: **while** $it < it_{max} \wedge f(Sol) > 0$ **do**
9: $Cand \leftarrow \emptyset$
10: **for** $(i, j) \in \mathcal{E}$ **do**
11: **if** $Sol[i] == Sol[j]$ **then**
12: $Cand \leftarrow Cand + i + j$
13: **end if**
14: **end for**
15: $Sol' \leftarrow copy(Sol)$
16: **for** $r = 0 \ldots rep$ **do**
17: $vert \leftarrow randompick(Cand)$
18: $Sol'[vert] \leftarrow randompick(Col - Sol[vert])$
19: **if** $f(Sol') < f(Sol)$ **then**
20: **if** $f(Sol') < Asp[f(Sol)]$ **then**
21: $Asp[f(Sol) \leftarrow f(Sol') - 1$
22: **if** $(vert, Sol'[vert]) \in Tabu$ **then**
23: $Tabu \leftarrow Tabu - (vert, Sol'[vert])$
24: **end if**
25: **break** ▷ Jump to Line 33
26: **else**
27: **if** $(vert, Sol'[vert]) \notin Tabu$ **then**
28: **break** ▷ Jump to Line 33
29: **end if**
30: **end if**
31: **end if**
32: **end for**
33: $Tabu \leftarrow Tabu + (vert, Sol[vert])$
34: $Sol \leftarrow Sol'$
35: $it \leftarrow it + 1$
36: **end while**
37: **return** $TRUE$ **if** $f(Sol) == 0$ **else** $FALSE$
38: **end procedure**

We also compared our model to a NeuroSAT (previously described in Section 3.6.1) version, under the TGN framework, which reproduced training and test accuracy of the original model. We set NeuroSAT parameters to the same values of our model, that is $d = 64$, $t_{max} = 32$ and learning rate $= 2 \times 10^{-5}$. We also needed to reduce our k-GCP instances (training and test) to CNF SAT problems. Although such reduction can be done in polynomial time, its main drawback is the huge amount of variables it generates. To reduce from GCP to SAT one is required to create $C * |\mathcal{V}|$ variables, $C * |\mathcal{V}|$ clauses to ensure that there will be no uncolored vertex, $C * |\mathcal{E}|$ clauses to ensure that each edge has its source and target colored differently and $(C - 1) * |\mathcal{V}|$ clauses to ensure that at most one color will be assigned to each vertex, thus causing a significant increase of nodes and edges in the resulting SAT graph in comparison to the original GCP graph, whose requirement is just C embeddings for colors and $|\mathcal{V}|$ embeddings for vertices. Algorithm 6 presents the entire procedure to reduce a GCP to a SAT formula. We have trained this NeuroSAT version until it reached a Sigmoid Cross Entropy loss of 0.41 at epoch 1222, considering the same training dataset we described before.

Note that both heuristics are able to output a valid color assignment while our model and NeuroSAT are trained solely as classifiers.

4.4 Experimental Results

We trained our model on an NP-complete GCP version, since we fed it not only the graph but also a target number of colors C which was either equal to the chromatic number χ of that graph (positive label) or equal to the chromatic number minus 1 (negative label). During test, however, we emulate the NP-hard problem by feeding each graph repeatedly to the model, with C ranging[5] from 2 to $\chi + 3$. The same procedure was applied to test NeuroSAT and Tabucol – the greedy algorithm already produces a local minimal coloring. In total, 4096 unseen test instances were fed to our model, NeuroSAT, Tabucol and Greedy algorithm following this procedure. These instances were produced in the same fashion as the training ones (see Section 4.2).

Figure 4.3 shows how our model and these three baselines performed over the test instances (average of 10 runs). For both our model and NeuroSAT, we selected as the best solution (vertical axis) the first C which yielded a positive prediction ($prediction > 0.5$). As the horizontal axis stands for the actual χ of those instances, one can evaluate each

[5]We ignored the trivial case of a fully-disconnected graph with $\chi = 1$

Algorithm 6 k-GCP to SAT reduction

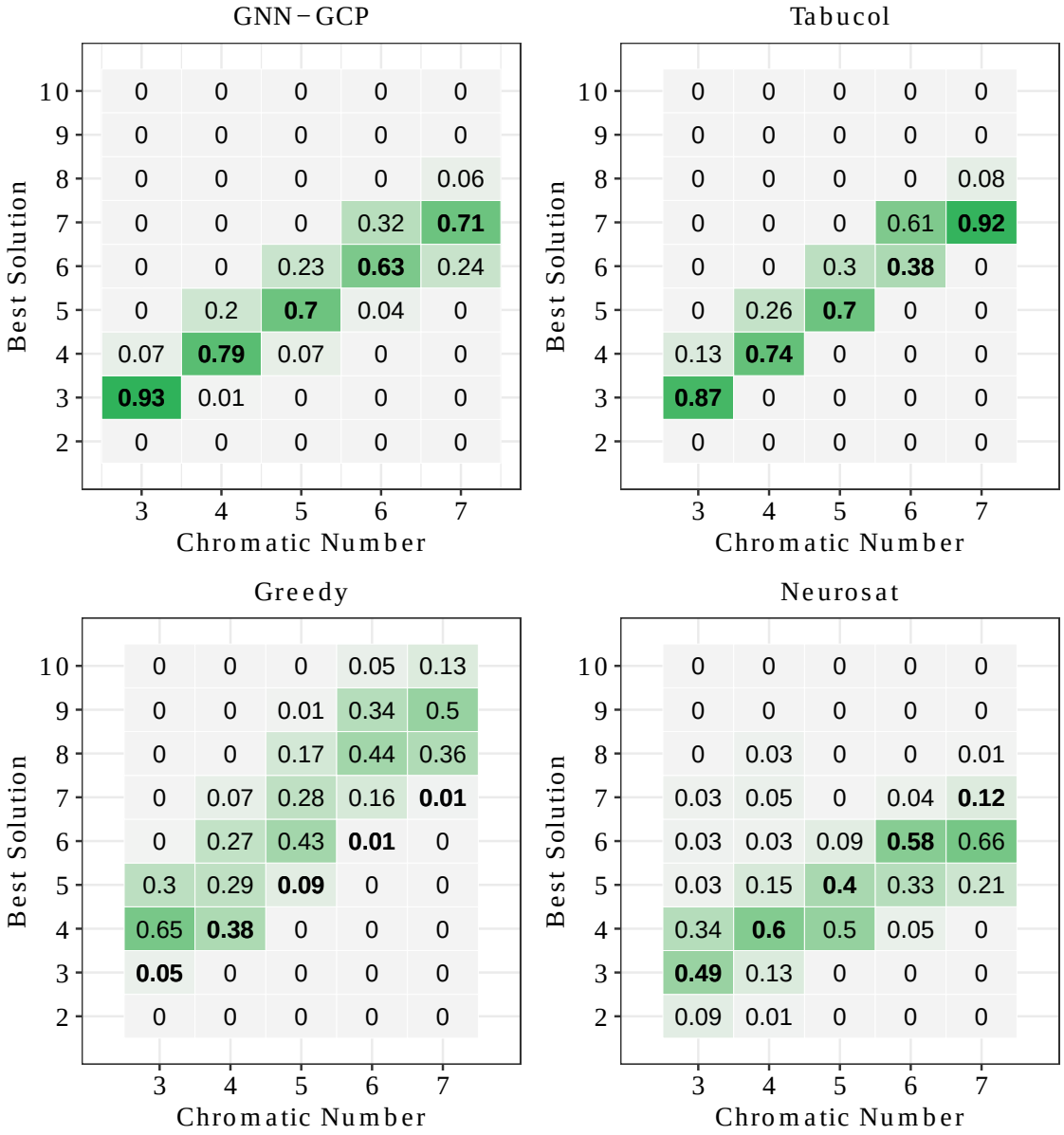
procedure REDUCTION($\mathcal{G} = (\mathcal{V}, \mathcal{E}), C$)
    // A formula is a set of clauses, initially empty
    $F \leftarrow \emptyset$
    // Create variables matrix: each cell must have a unique value
    $Vars[v_i, c_j] \leftarrow i \frown j \quad \forall v_i \in \mathcal{V} \quad \forall c_j \in \{0, \cdots, C-1\}$
    // Ensure that at least one color is assigned to each vertex
    for $v_i \in \mathcal{V}$ **do**
        $CL \leftarrow [\,]$
        for $j = 0 \ldots C-1$ **do**
            $CL \leftarrow CL + Vars[i, j]$
        end for
        $F \leftarrow F + CL$
    end for
    // Ensure that each edge has its endpoints colored differently
    for $(u, v) \in \mathcal{E}$ **do**
        for $j = 0 \ldots C-1$ **do**
            $CL \leftarrow -Vars[u, j] + -Vars[v, j]$
            $F \leftarrow F + CL$
        end for
    end for
    // Ensure that at most one color is assigned to each vertex
    for $v_i \in \mathcal{V}$ **do**
        for $j = 0 \ldots C-2$ **do**
            $CL \leftarrow -Vars[i, j] + -Vars[i, j+1]$
            $F \leftarrow F + CL$
        end for
    end for
    return F
end procedure

model by the main diagonal values. Along the first six chromatic numbers our model slightly outperforms the Tabucol algorithm regarding a binary accuracy metric – hit only when the exact chromatic number is achieved – however, it demonstrates a drop on its performance at the highest chromatic number, where the Tabucol achieves around 90% of accuracy. Nevertheless, our model's average accuracy across all test instances was 75.09% and its absolute deviation from the exact chromatic number averaged 0.25, while Tabucol scored 70.80% and 0.29, NeuroSAT scored 46.20% and 0.69 and the greedy algorithm achieved only 13.56% and 1.55, respectively. The performance of NeuroSAT is quite disappointing as it can roughly achieve around 60% of accuracy for $\chi = 4$ and $\chi = 6$, whereas our model keeps its performance always above 60% for all chromatic numbers. However, at the top border of our experiment, $\chi = 7$, Tabucol achieves the best performance among all models. Both our method and Tabucol presented a noisy

performance for $\chi = 6$, as they misclassified a significant portion of such instances with best solution $C = 7$ (32% for our model and 68% for Tabucol). We analyzed the edge density[6] of our test instances (see Figure 4.4) and verified a great amount of overlap between the distribution of edge density of instances with $\chi = 6$ and $\chi = 7$, this may have led both our model and Tabucol to be unable to distinguish between them as they did to other chromatic numbers. Note that the same phenomenon occurred to NeuroSAT, but in the backward direction, as it classifies most of the $\chi = 7$ instances as $C = 6$. The overlap presented by instances with $\chi = 3$ and $\chi = 4$, however, did not decrease the performance of GNN-GCP and Tabucol.

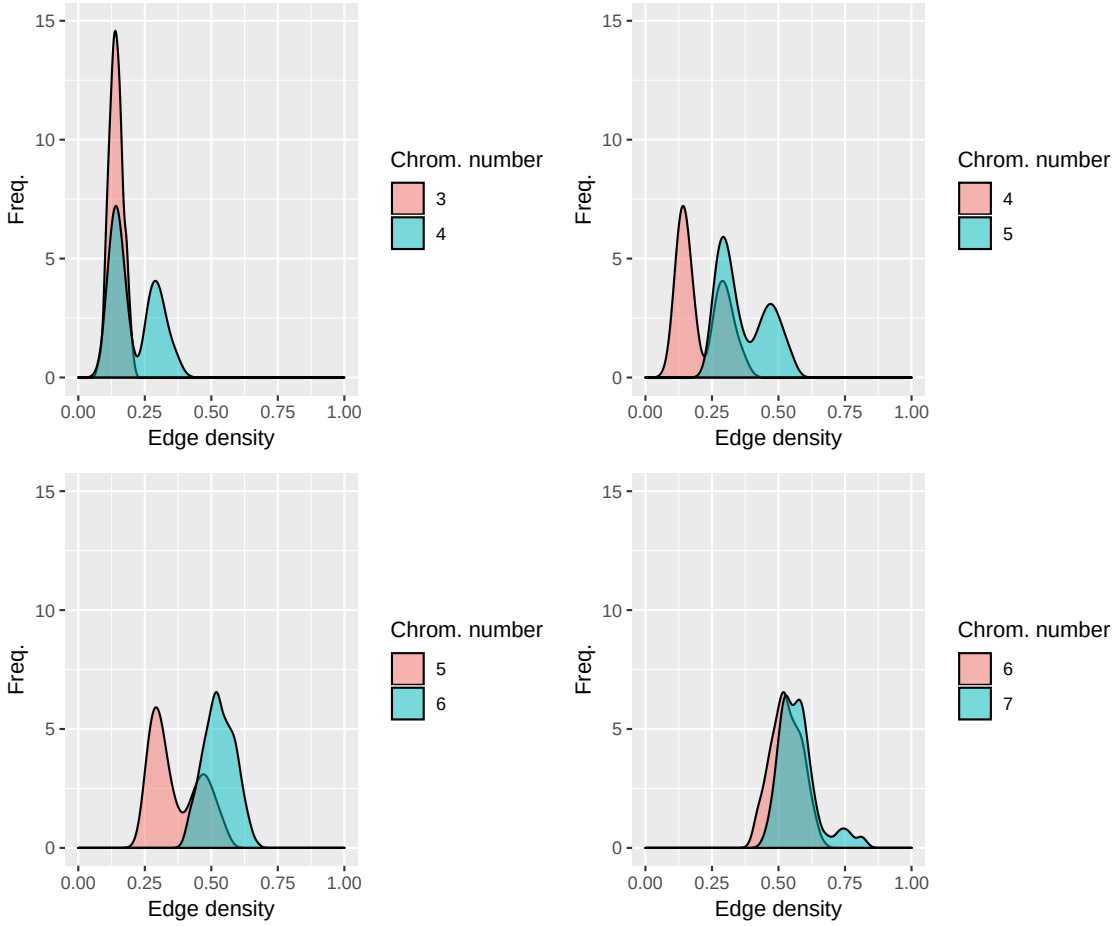
[6]The ratio between the actual number of edges and the total amount of possible edges. In an undirected graph this is: $dens = m/(n \times (n-1)/2)$

Figure 4.3: Prediction distributions over 4096 unseen test instances, with similar features to those seen in training, for our model (GNN-GCP), Tabucol, a Greedy heuristic and NeuroSAT. Note that the darker the main diagonal (highlighted in bold), the better the results. Source: author.



All 4 methods were able to keep a stable performance regardless of the number of vertices of the test instances (see Figure 4.5, left plot). While GNN-GCP and Tabucol achieved around 75% of accuracy, NeuroSAT remained stable around 50% and the Greedy algorithm never guessed the right χ for more than 25% of the instances of any number of vertices. When the performance is compared to the edge density of our instances, however, some trends arise: there is a clear drawback of using both heuristics when the edge density increases; GNN-GCP also presents a lower performance as the edge density increases, but NeuroSAT remains somewhat stable. The most dense instances, from 0.7 onwards,

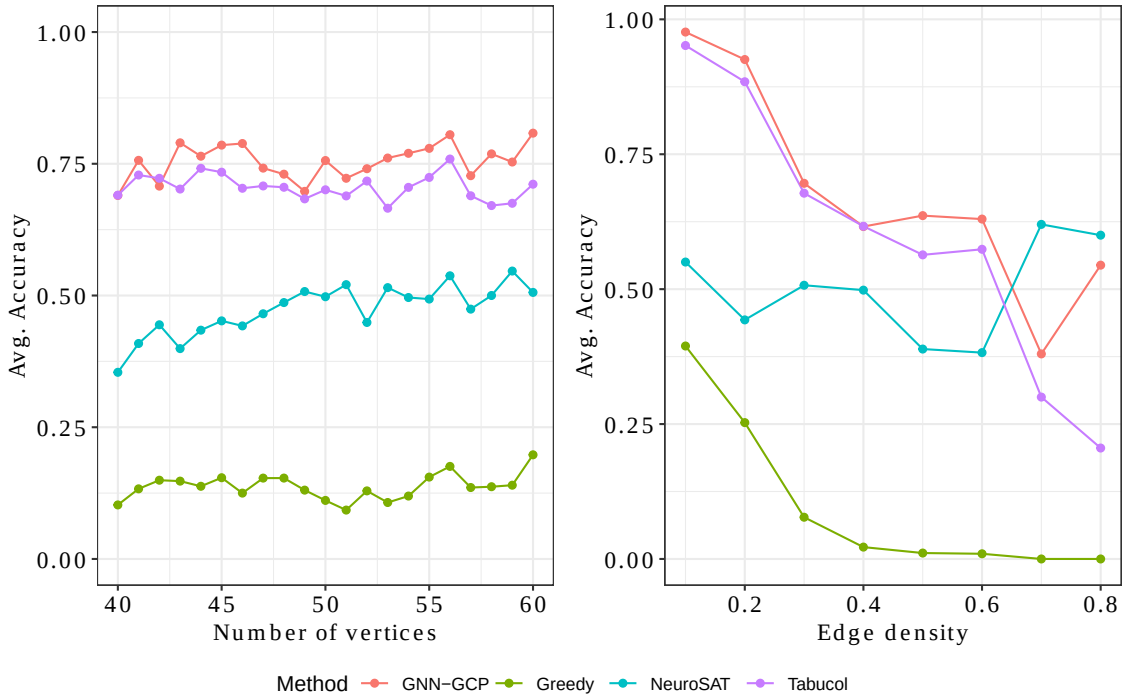
Figure 4.4: Density plot of the edge density of our test instances grouped by their chromatic number. Instances with chromatic number 6 and 7 presented a large overlap as well as 3 and 4. Source: author.



however represent only 1% of our test dataset, thus the noisy performance of GNN-GCP and NeuroSAT upon them may be not statistically relevant. Such instances are very hard to generate as their search space are huge and even a CSP-Solver takes very long to solve them or even returns an uncertainty flag.

So far we have seen **GNN-GCP** performance on the chromatic number problem, but even though we trained GNN-GCP on instances whose C was true ($C = \chi$) or false ($C = \chi - 1$) only by a narrow margin, we could only argue it can solve the GCP problem (decision version) if as the margin goes wider, the model still provides the right answer, despite being positive or negative. We can see that in Fig. 4.6: our model's predictions undergo a regime remindful of a phase transition – as we fed our model with C values closer to χ it became unsure about its prediction, nevertheless the model is quite sure about its prediction on the upper ($C = \chi + 2$) and lower ($C = \chi - 2$) bounds.

Figure 4.5: Average accuracy of each method regarding number of vertices of test instances (left plot) and binned edge density (right plot). Source: author.

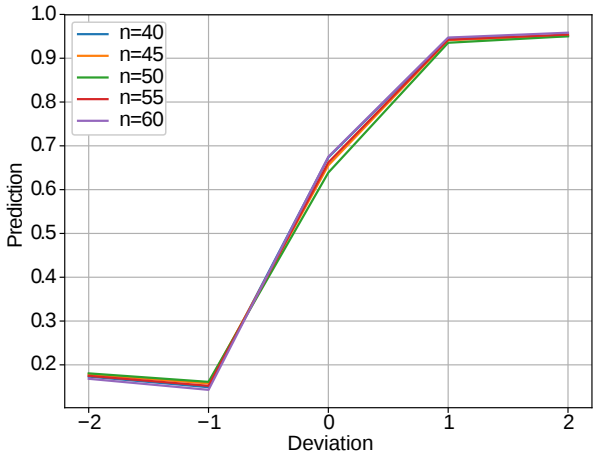


4.4.1 Performance on Different Graph Distributions

In order to assess the performance of **GNN-GCP** in unseen instances, both larger and smaller than those it was trained upon, we gathered 20 instances from the COLOR02/03/04 Workshop dataset[7] and fed them to our model and to the previously cited heuristics (Tabucol and Greedy). "Queen" graphs are drawn from chessboards, where each vertex corresponds to a square in the board and each edge represents a legal move by the queen. Queen graphs are hamiltonian and biconnected. "Myciel" graphs are triangle-free but the coloring number increases in problem size. "Insertions" graphs are similar to "Myciel" but with additional disconnected vertices to increase graph size while keeping the same edge density. "mug88_1" is an almost 3-colorable graph with a hard-to-find 4-clique inside it. The remainder are graphs created by Donald Knuth and built upon book's characters, where each vertex is a character and an edge indicates that these two characters met each other during the history. These instances have up to 835% more vertices than the training ones, and also have chromatic numbers exceeding the boundaries seen during training. We also fed them to the NeuroSAT model, but it was not able to output a positive answer within the range of $C \in [2, 3, \cdots, \chi + 5]$ for all instances. This suggests that these different graph distributions are not well suited for the trained NeuroSAT model.

[7] <https://mat.tepper.cmu.edu/COLOR02/>

Figure 4.6: Average prediction – above 0.5 means a positive answer – extracted from **GNN-GCP** fed with testing instances with size ranging from 40 to 60. Each instance was fed seven times to the model with target $\mathcal{C} \in [\chi - 2, \chi + 2]$



As both Tabucol and the greedy algorithm produce a valid color assignment, they never underestimate the chromatic number, as our model eventually does (see Table 4.1). When it comes to predict the exact chromatic number, our model only achieved 5 hits, against 12 and 7 from Tabucol and the greedy algorithm, respectively. Nevertheless, our model's absolute deviation from the actual chromatic number accounted for 1.15, standing in between Tabucol (0.33) and the greedy algorithm (2.15). Despite its overall low performance on these instances, it is worthy reminding that upon training our model have never seen instances with more than 60 vertices and χ higher than 7.

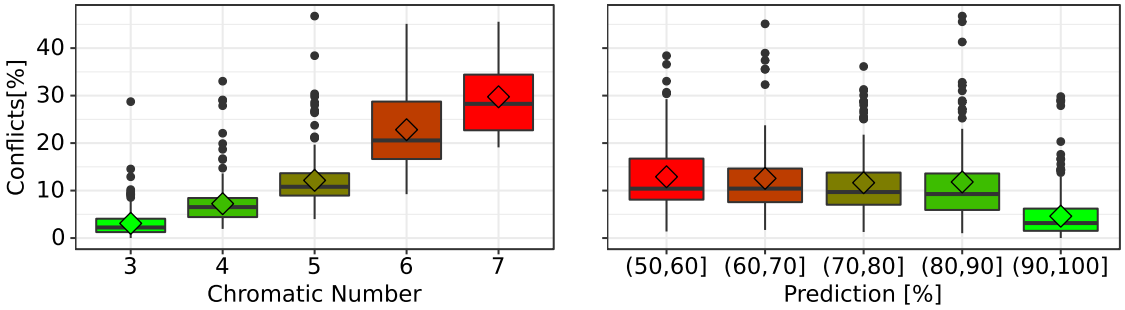
4.4.2 Exploring Vertex Embeddings

The capability of GNN-like models to generate meaningful embeddings and to learn an algorithm to solve its task was already highlighted by Selsam et al. (2019). Following their findings, even though we trained our model only to produce a boolean answer, we also expected to decode valid color assignments to each vertex. To achieve that, we extracted vertex embeddings from test instances which were fed to our model along with their exact chromatic number as C and resulted in a **GNN-GCP**'s positive prediction. These embeddings were then clustered into C groups whose centroids were initialized with the final color embeddings from our model – we made the *a priori* assumption that the model internally places non-adjacent vertices near each other, thus resulting into color assignments. For each cluster of each instance we computed how many conflicts were

Table 4.1: Chromatic number produced by our model and two heuristics on some instances of the COLOR02/03/04 dataset. As our model faces unseen graph sizes and larger chromatic numbers it tends to underestimate its answers.

Instance	Size	χ	Computed χ		
			GNN-GCP	Tabucol	Greedy
queen5_5	25	5	6	**5**	8
queen6_6	36	7	**7**	8	11
myciel5	47	6	5	**6**	**6**
queen7_7	49	7	8	8	10
queen8_8	64	9	8	10	13
1-Insertions_4	67	4	**4**	5	5
huck	74	11	8	**11**	**11**
jean	80	10	7	**10**	**10**
queen9_9	81	10	9	11	16
david	87	11	9	**11**	12
mug88_1	88	4	3	**4**	**4**
myciel6	95	7	**7**	**7**	**7**
queen8_12	96	12	10	**12**	15
games120	120	9	6	**9**	**9**
queen11_11	121	11	12	NA	17
anna	138	11	**11**	**11**	12
2-Insertions_4	149	4	**4**	5	5
queen13_13	169	13	14	NA	21
myciel7	191	8	NA	**8**	**8**
homer	561	13	14	**13**	15
Average	116.9	8.6	8	8.55	10.75
		HITS	5	**12**	7
		Abs. Dev. from χ	1.15	**0.33**	2.15

Figure 4.7: After performing a k-means ($k = C$) algorithm on the vertex embeddings, we computed the ratio of conflicts (adjacent vertices on the same cluster) for each cluster. In this experiment, we fed our model with the exact chromatic number for each instance and selected the embeddings only for positive predictions (above 50%) of the **GNN-GCP**. The left plot shows that the clusters have less meaning as the chromatic number grows. The right plot shows how the clustering correlates to the final prediction – when the model is more confident that there is a valid coloring, the clusters have less conflicts.

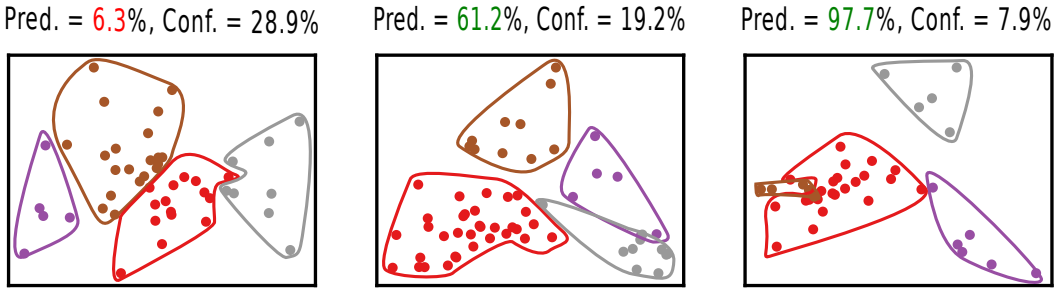


raised – the ratio between the amount of adjacent vertices belonging to that cluster and the number of 2-combinations without repetition of these vertices. We then computed the average conflicts per cluster/color. Naturally, a perfect valid color assignment for a given instance would yield zero conflicts.

According to Figure 4.7 (leftmost boxplots) our model faces more difficulties in assigning correct vertices to clusters when C grows – regarding instances with $C = 7$ the average conflicts ratio within the seven clusters achieved around 30%. On the other hand, this metric drops off to below 5% regarding 3-coloring instances. We also verified a moderate negative correlation (-0.6 – Spearman's Rank Correlation) between the positive certainty of our model and the number of conflicts, as seen in Figure 4.7 (rightmost boxplots), which goes along with the natural thought that the fewer the conflicts within each instance's clusters, the more certain our model is of a positive answer. Among these instances, our model together with the clustering procedure were able to provide a valid color assignment to only 3 examples, nevertheless its behavior suggests that it is able to respond positively when some inferior threshold w.r.t. conflicts is reached.

Figure 4.8 helps visualizing the distribution of vertices along the dimensions (internal outcome of our model) and how their clustering affects the GNN predictions. The clusters separability (measured with a silhouette score $S \in [-1, +1]$ over the 64-dimensional clustered embeddings) increases together with our model's certainty of a positive answer and the inverse ratio of conflicts within each cluster: the leftmost example resulted in a clustering with 28.9% of average conflicts and a S equals to 0.29, even though our model should have answered it positively, its prediction was only 6.3% (any answer

Figure 4.8: Vertex embeddings (after a PCA-2D procedure) of three different test instances, with $\chi = 4$. The axes and the surrounding curves are meaningless as we are simply interested in visualizing how the clusters behavior are related to our model outcomes. All these three instances should imply in a positive answer, but our model only answered positively to the second and to the third one. Pred. stands for the certainty of our model and Conf. stands for the average ratio of conflicts of all clusters.



below 50% is considered negative); when fed with the middle instance, however, our model was able to answer properly (61.2%) as not only the average of conflicts decreased to 19.2% but also the silhouette coefficient increased to 0.39; finally, in the last example our model was quite sure about its positive answer, which goes along with the clustering procedure outcomes: only 7.9% of conflicts within each cluster and a silhouette score of 0.44.

5 CONCLUSIONS AND FUTURE WORK

Throughout this dissertation we explained how a Graph Neural Network model can be designed to solve the decision version of GCP, such model works by allowing several timesteps of message-passing between the projected elements (vertices and colors) of the given problem – note that GNNs do not directly manipulate a problem's symbols but rather refine their hyperdimensional projections. Our model employs a connectionist architecture in its kernel in a way that allows the symbolic structure of the problem to be reflected upon its computation: messages are properly computed from vertices to colors (and vice-versa), then they are aggregated according to the graph's adjacency information and used to update the next vertex and color embeddings. We also demonstrated how the proposed model can be trained on very hard instances, whose candidate number of colors is either equal to the chromatic number or just one unit smaller.

We compared our model to 3 other baselines, including a neural-symbolic one, in regards to their strict accuracy and how their performance relates to some features of the test graphs. The results we provided in Section 4.4 help answering the research questions we postulated before:

1. *Can a Graph Neural Network solve the decision version of the Graph Coloring problem only from the network structure and a number of colors which is close to the chromatic number?*

   Yes, when we evaluated our model under the chromatic number answer, it correctly indicated the chromatic number of 75% of the test instances, while Tabucol heuristic achieved 70% and NeuroSAT only 46%. Such performances are not, however, equally distributed over the five chromatic numbers we tested. Our model clearly presents some difficulties while classifying instances whose chromatic number is higher. However, when we consider the most traditional version of GCP (3-coloring) our model presents an outstanding performance of 93%.

2. *Does such GNN generalize its performance to larger/smaller number of colors?*

   Yes, we plotted an acceptance curve (Figure 4.6) which shows that as we increase the target number of colors C the model becomes more and more certain of its positive answer, and it also becomes more certain (of a negative answer) when we decrease C to a number lower than the actual chromatic number.

3. *Does the strategy employed by NeuroSAT to decode assignments from GNN internal states work under this new architecture?*

   The original NeuroSAT paper described a simple 2-clustering procedure over literal embeddings in order to separate them into "True" and "False" literals. We followed such strategy and clustered vertex embeddings into C groups, where the initial centroid of each group was set to the corresponding final color embedding, but only 3 out of 4096 instances accepted the assignment yielded by such clustering procedure.

4. *Can this specialized GNN learn meaningful and interpretable internal states?*

   Yes, despite not being able to decode valid assignment, we found evidences suggesting that intra-cluster conflicts – neighboring vertices that had their final embeddings near each other – are correlated to our model prediction. That is, instead of building a perfect clustering, our model seems to give a positive/negative answer based on a threshold of separability among vertex embeddings.

Neural-symbolic approaches have recently received a lot of attention due to their capabilities of interpretability, explainability and combining low and high level reasoning. The Graph Neural Network model (and its subfamilies) is arguably the most promising technique to bridge the gap between connectionistic and symbolic artificial intelligence as their applications are both manifold and successful so far. Nevertheless, much of these applications – particularly regarding NP-Complete problems – are still not comparable to state-of-the-art solvers, such as the NeuroSAT itself (SELSAM et al., 2019) and the work by Prates et al. (2019a) on the TSP. Even so, we are just beginning to understand how such GNN-like models can be engineered to solve combinatorial problems. For instance, our results suggest that a general model, such as NeuroSAT, may not be as competitive as one may envision at first sight: as any problem in NP can be reduced to SAT, one may think all SAT results will hold to any reduced problem. We verified that this is not true for graph coloring problems whose reduction yielded more than 120 variables on average – Selsam et al. (2019) only reported GCP results for graphs with 10 vertices. But, as happened to NeuroSAT as well, our results are not yet comparable to state-of-the-art solvers of GCP – for instance, we know for a fact that CSP and SAT state-of-the-art solvers would achieve a better performance than our model given a fair computation time budget.

The most prominent benefit of our technique, and of neural-symbolic ones in general, is to dismiss the need of hand-coding constraints of problems – e.g. both Tabucol

and any other GCP heuristic would require some explicit rule to determine that two adjacent vertices cannot be assigned to the same color. While such benefit may not be as relevant as an overall accuracy to final stakeholders, we expect that our research helps fostering the research over specialized GNN models and how they reason over their specific problems.

With respect to the current state of our model, we can envision some further research paths: we chose to not feed *a priori* information to our model, each color embedding is connected to all vertices. This could be changed so that an initial greedy coloring is informed to the GNN, ensuring that adjacent vertices are not communicating with the same color embedding, this will require our model to use more than C color embeddings during its computation, but one could expect that the additional colors may have their embeddings clustered/collapsed into the original C embeddings at the final timestep. A much simpler change can also be done regarding the initial color embeddings: we chose to randomly initialize them, but ideally they could be placed equidistant over a hypersphere. This, however, may not yield relevant improvements as we saw that GNNs are capable of refining such embeddings until they acquire some useful meaning. When it comes to decoding valid assignments from vertex embeddings, we also have designed a variant of GNN-GCP whose goal was to minimize the number of conflicts intra-cluster: basically, we changed the MLP V_{vote} to a Softmax output with size C, so it becomes a multi-class classifier, then we used matrix multiplications with the adjacency matrix to identify color assignments which yielded a conflict: the sum of the fuzzy values of conflicting color assignments is defined as loss. Unfortunately, such model was still not able to learn anything useful, as its loss almost never dropped. It is also worth mentioning that we successfully decoded valid assignments from GNN-GCP for three 3-coloring problems. This and the findings of Avelar et al. (2018) and recently of Toenshoff et al. (2019), who trained specific GNN-based models for each problem, may suggest that we are currently "polluting" our GNN internal states by feeding graph coloring problems with C ranging from 3 to 7, and that training a GNN-GCP for each specific c-coloring may lead to better results regarding decoding assignments.

REFERENCES

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

ASRATIAN, A. S.; DENLEY, T. M.; HÄGGKVIST, R. **Bipartite graphs and their applications**. [S.l.]: Cambridge university press, 1998.

AVELAR, P. H. C. et al. Multitask learning on graph neural networks - learning multiple graph centrality measures with a unified network. **CoRR**, abs/1809.07695, 2018. Available from Internet: <http://arxiv.org/abs/1809.07695>.

BADER, S.; HITZLER, P. Dimensions of neural-symbolic integration - A structured survey. In: **We Will Show Them! Essays in Honour of Dov Gabbay**. [S.l.: s.n.], 2005. p. 167–194.

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

BALLARD, D. H.; GARDNER, P.; SRINIVAS, M. **Graph problems and connectionist architectures**. [S.l.]: University of Rochester, Computer Science Department, 1987.

BARNIER, N.; BRISSET, P. Graph coloring for air traffic flow management. **Annals of Operations Research**, v. 130, n. 1, p. 163–178, Aug 2004.

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

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

BENGIO, Y.; SIMARD, P. Y.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. **IEEE Trans. Neural Networks**, v. 5, n. 2, p. 157–166, 1994.

CHAITIN, G. J. et al. Register allocation via coloring. **Comput. Lang.**, v. 6, n. 1, p. 47–57, 1981. Available from Internet: <https://doi.org/10.1016/0096-0551(81)90048-5>.

CHEESEMAN, P. C.; KANEFSKY, B.; TAYLOR, W. M. Where the really hard problems are. In: MYLOPOULOS, J.; REITER, R. (Ed.). **Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991**. Morgan Kaufmann, 1991. p. 331–340. Available from Internet: <http://ijcai.org/Proceedings/91-1/Papers/052.pdf>.

CHEN, W. et al. Register allocation for intel processor graphics. In: **CGO 2018**. [s.n.], 2018. p. 352–364. ISBN 978-1-4503-5617-6. Available from Internet: <http://doi.acm.org/10.1145/3168806>.

CHO, K. et al. On the properties of neural machine translation: Encoder-decoder approaches. In: **Proc. of SSST@EMNLP**. [s.n.], 2014. p. 103–111. Available from Internet: <http://aclweb.org/anthology/W/W14/W14-4012.pdf>.

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

CHOY, C. B.; GWAK, J.; SAVARESE, S. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In: **CVPR**. [S.l.]: Computer Vision Foundation / IEEE, 2019. p. 3075–3084.

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.

CULBERSON, J.; GENT, I. Frozen development in graph coloring. **Theoretical Computer Science**, v. 265, n. 1, p. 227 – 264, 2001. ISSN 0304-3975. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0304397501001645>.

DAS, D.; AHMAD, S. A.; VENKATARAMANAN, K. Deep learning-based hybrid graph-coloring algorithm for register allocation. **arXiv preprint arXiv:1912.03700**, 2019.

DEVLIN, J. et al. BERT: pre-training of deep bidirectional transformers for language understanding. In: **Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)**. [s.n.], 2019. p. 4171–4186. Available from Internet: <https://www.aclweb.org/anthology/N19-1423/>.

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

DUVENAUD, D. K. et al. Convolutional networks on graphs for learning molecular fingerprints. In: CORTES, C. et al. (Ed.). **Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada**. [S.l.: s.n.], 2015. p. 2224–2232.

FOLAND, W.; MARTIN, J. H. Abstract meaning representation parsing using LSTM recurrent neural networks. In: **Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)**. Vancouver, Canada: Association for Computational Linguistics, 2017. p. 463–472. Available from Internet: <https://www.aclweb.org/anthology/P17-1043>.

GAMST, A. Some lower bounds for a class of frequency assignment problems. **IEEE transactions on vehicular technology**, IEEE, v. 35, n. 1, p. 8–14, 1986.

GARCEZ, A. d'Avila; LAMB, L.; GABBAY, D. **Neural-Symbolic Cognitive Reasoning**. Springer, 2009. (Cognitive Technologies). ISBN 978-3-540-73245-7. Available from Internet: <https://doi.org/10.1007/978-3-540-73246-4>.

GARCEZ, A. S. d'Avila et al. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. **FLAP**, v. 6, n. 4, p. 611–632, 2019.

GAREY, M. R.; JOHNSON, D. S.; SO, H. C. An application of graph coloring to printed circuit testing (working paper). In: **16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975**. IEEE Computer Society, 1975. p. 178–183. Available from Internet: <https://doi.org/10.1109/SFCS.1975.3>.

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

GLOVER, F. W. Tabu search - part I. **INFORMS Journal on Computing**, v. 1, n. 3, p. 190–206, 1989.

GLOVER, F. W. Tabu search - part II. **INFORMS Journal on Computing**, v. 2, n. 1, p. 4–32, 1990.

GOLOVACH, P. A. et al. A survey on the computational complexity of colouring graphs with forbidden subgraphs. **CoRR**, abs/1407.1482, 2014.

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

GORI, M.; MONFARDINI, G.; SCARSELLI, F. A new model for learning in graph domains. In: **Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.** [S.l.: s.n.], 2005. v. 2, p. 729–734 vol. 2. ISSN 2161-4407.

GRÖTSCHEL, M.; LOVÁSZ, L.; SCHRIJVER, A. Polynomial algorithms for perfect graphs. **Ann. Discrete Math**, v. 21, p. 325–356, 1984.

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

HERTZ, A.; WERRA, D. de. Using tabu search techniques for graph coloring. **Computing**, v. 39, n. 4, p. 345–351, 1987.

HINTON, G. E.; SABOUR, S.; FROSST, N. Matrix capsules with EM routing. In: **6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings**. [s.n.], 2018. Available from Internet: <https://openreview.net/forum?id=HJWLfGWRb>.

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

HOPFIELD, J. J. Neural networks and physical systems with emergent collective computational abilities. **Proceedings of the national academy of sciences**, National Acad Sciences, v. 79, n. 8, p. 2554–2558, 1982.

HUBEL, D. H.; WIESEL, T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. **The Journal of physiology**, Wiley Online Library, v. 160, n. 1, p. 106–154, 1962.

KÄLL, L. et al. Semi-supervised learning for peptide identification from shotgun proteomics datasets. **Nature methods**, Nature Publishing Group, v. 4, n. 11, p. 923–925, 2007.

KARP, R. M. Reducibility among combinatorial problems. In: **Complexity of Computer Computations**. [S.l.]: Plenum Press, New York, 1972. (The IBM Research Symposia Series), p. 85–103.

KAUDERER-ABRAMS, E. Quantifying translation-invariance in convolutional neural networks. **CoRR**, abs/1801.01450, 2018.

KHARDON, R.; ROTH, D. Learning to reason with a restricted view. **Machine Learning**, v. 35, n. 2, p. 95–116, 1999. Available from Internet: <https://doi.org/10.1023/A:1007581123604>.

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

KINGMA, D. P. et al. Semi-supervised learning with deep generative models. In: **NIPS**. [S.l.: s.n.], 2014. p. 3581–3589.

KIPF, T. N.; WELLING, M. Semi-supervised classification with graph convolutional networks. In: **ICLR (Poster)**. [S.l.]: OpenReview.net, 2017.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. Imagenet classification with deep convolutional neural networks. In: **NIPS**. [S.l.: s.n.], 2012. p. 1097–1105.

LEMOS, H. et al. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. **CoRR**, abs/1903.04598, 2019. Available from Internet: <http://arxiv.org/abs/1903.04598>.

LENC, K.; VEDALDI, A. Understanding image representations by measuring their equivariance and equivalence. In: **CVPR**. [S.l.]: IEEE Computer Society, 2015. p. 991–999.

LEWIS, R. M. R. **A Guide to Graph Colouring - Algorithms and Applications**. [S.l.]: Springer, 2016.

LI, H. et al. A convolutional neural network cascade for face detection. In: **CVPR**. [S.l.: s.n.], 2015.

LI, Y. et al. Gated graph sequence neural networks. In: **ICLR (Poster)**. [S.l.: s.n.], 2016.

LIMA, A. M. de; CARMO, R. Exact algorithms for the graph coloring problem. **RITA**, v. 25, n. 4, p. 57–73, 2018. Available from Internet: <https://doi.org/10.22456/2175-2745.80721>.

MATCOVSCHI, M.-H.; PASTRAVANU, O. Invariance properties of recurrent neural networks. In: **Intelligent Systems and Technologies**. [S.l.]: Springer, 2009. p. 105–119.

MELIS, G.; KOCISKÝ, T.; BLUNSOM, P. Mogrifier LSTM. **CoRR**, abs/1909.01792, 2019.

MNIH, V. et al. Playing atari with deep reinforcement learning. **CoRR**, abs/1312.5602, 2013.

MONTI, F.; BRONSTEIN, M. M.; BRESSON, X. Geometric matrix completion with recurrent multi-graph neural networks. In: **NIPS**. [S.l.: s.n.], 2017. p. 3697–3707.

NALLAPATI, R.; ZHAI, F.; ZHOU, B. Summarunner: A recurrent neural network based sequence model for extractive summarization of documents. In: **AAAI**. [S.l.]: AAAI Press, 2017. p. 3075–3081.

NATHANI, D. et al. Learning attention-based embeddings for relation prediction in knowledge graphs. In: **Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers**. [s.n.], 2019. p. 4710–4723. Available from Internet: <https://www.aclweb.org/anthology/P19-1466/>.

NGUYEN, D. Q. et al. A novel embedding model for knowledge base completion based on convolutional neural network. In: **NAACL-HLT (2)**. [S.l.]: Association for Computational Linguistics, 2018. p. 327–333.

NIELD, T. Is another ai winter coming? Feb 2019. Online; accessed 15-January-2020. Available from Internet: <https://hackernoon.com/is-another-ai-winter-coming-ac552669e58c>.

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

PALM, R.; PAQUET, U.; WINTHER, O. Recurrent relational networks for complex relational reasoning. **arXiv preprint arXiv:1711.08028**, 2017.

Philipsen, W. J. M.; Stok, L. Graph coloring using neural networks. In: **1991., IEEE International Symposium on Circuits and Systems**. [S.l.: s.n.], 1991. p. 1597–1600 vol.3.

PRATES, M. O. R. et al. Learning to solve np-complete problems: A graph neural network for decision TSP. In: **The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019**. [S.l.: s.n.], 2019. p. 4731–4738.

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

QI, S. et al. Learning human-object interactions by graph parsing neural networks. In: **ECCV (9)**. [S.l.]: Springer, 2018. (Lecture Notes in Computer Science, v. 11213), p. 407–423.

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

RADFORD, A. et al. Language models are unsupervised multitask learners. **OpenAI Blog**, v. 1, n. 8, 2019.

RAGHAVAN, S. 2020 AI predictions from IBM research. Dec 2019. Online; accessed 15-January-2020. Available from Internet: <https://www.ibm.com/blogs/research/2019/12/2020-ai-predictions/>.

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

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **nature**, Nature Publishing Group, v. 323, n. 6088, p. 533–536, 1986.

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

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

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

Scarselli, F. et al. Graph neural networks for ranking web pages. In: **The 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI'05)**. [S.l.: s.n.], 2005. p. 666–672. ISSN null.

SCHLICHTKRULL, M. S. et al. Modeling relational data with graph convolutional networks. In: **The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings**. [s.n.], 2018. p. 593–607. Available from Internet: <https://doi.org/10.1007/978-3-319-93417-4_38>.

SCHROFF, F.; KALENICHENKO, D.; PHILBIN, J. Facenet: A unified embedding for face recognition and clustering. In: **CVPR**. [S.l.]: IEEE Computer Society, 2015. p. 815–823.

SELSAM, D. et al. Learning a SAT solver from single-bit supervision. In: **7th International Conference on Learning Representations, ICLR 2019**. [S.l.: s.n.], 2019.

SHARMA, R. **Machine Intelligence in Design Automation**. [S.l.]: Independent, 2018.

SHEAD, S. Researchers: Are we on the cusp of an 'ai winter'? Jan 2020. Online; accessed 15-January-2020. Available from Internet: <https://www.bbc.com/news/technology-51064369>.

SHEN, Y. et al. Person re-identification with deep similarity-guided graph neural network. In: **The European Conference on Computer Vision (ECCV)**. [S.l.: s.n.], 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.

SMOLENSKY, P. Next-generation architectures bridge gap between neural and symbolic representations with neural symbols. Dec 2019. Online; accessed 15-January-2020. Available from Internet: <https://www.microsoft.com/en-us/research/blog/next-generation-architectures-bridge-gap-between-neural-and-symbolic-representations-with-neural-symbol
>.

TALLEC, C.; OLLIVIER, Y. Can recurrent neural networks warp time? In: **ICLR (Poster)**. [S.l.]: OpenReview.net, 2018.

THEVENIN, S.; ZUFFEREY, N.; POTVIN, J. Graph multi-coloring for a job scheduling application. **Discrete App. Math.**, v. 234, p. 218 – 235, 2018.

TOENSHOFF, J. et al. Run-csp: Unsupervised learning of message passing networks for binary constraint satisfaction problems. **arXiv preprint arXiv:1909.08387**, 2019.

VASWANI, A. et al. Attention is all you need. In: **Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA**. [s.n.], 2017. p. 5998–6008. Available from Internet: <http://papers.nips.cc/paper/7181-attention-is-all-you-need>.

WANG, B. Disconnected recurrent neural networks for text categorization. In: **ACL (1)**. [S.l.]: Association for Computational Linguistics, 2018. p. 2311–2320.

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

WEISFEILER, B.; LEHMAN, A. A. A reduction of a graph to a canonical form and an algebra arising during this reduction. **Nauchno-Technicheskaya Informatsia**, v. 2, n. 9, p. 12–16, 1968.

WITTEN, I. H.; FRANK, E.; HALL, M. A. **Data mining: practical machine learning tools and techniques, 3rd Edition**. Morgan Kaufmann, Elsevier, 2011. ISBN 9780123748560. Available from Internet: <http://www.worldcat.org/oclc/262433473>.

WU, Z. et al. A comprehensive survey on graph neural networks. **CoRR**, abs/1901.00596, 2019. Available from Internet: <http://arxiv.org/abs/1901.00596>.

XU, K. et al. How powerful are graph neural networks? In: **ICLR**. [S.l.]: OpenReview.net, 2019.

YANG, L. et al. Parsing R-CNN for instance-level human analysis. In: **CVPR**. [S.l.]: Computer Vision Foundation / IEEE, 2019. p. 364–373.

YOU, J. et al. Graph convolutional policy network for goal-directed molecular graph generation. In: BENGIO, S. et al. (Ed.). **Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.** [s.n.], 2018. p. 6412–6422. Available from Internet: <http://papers.nips.cc/paper/7877-graph-convolutional-policy-network-for-goal-directed-molecular-graph-generation>.

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

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

ZDEBOROVÁ, L.; KRZAKALA, F. Phase transitions in the coloring of random graphs. **CoRR**, abs/0704.1269, 2007. Available from Internet: <http://arxiv.org/abs/0704.1269>.

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

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

APPENDIX A — TYPED GRAPH NETWORK FOR THE GRAPH COLORING PROBLEM: DEFINITIONS

The Typed Graph Network framework provided by (PRATES et al., 2019b) requires 4 kinds of definitions. First, which graph elements will be assigned to a type of embedding – we define that vertices of type V will have an embedding size d, the same size of embeddings for vertices of type C. Second, we need to define adjacency matrices between these different types of embeddings – note that, if one type is not used to update another type, then no adjacency information between these types is needed, we defined a VV adjacency matrix between vertices V and VC between V and C vertices. Optionally, we may need to translate embeddings of one type to messages to another type: we defined message-functions from embeddings V to messages received by C, and vice-versa. Finally, we set the update-functions: an update of vertices of a given type may comprise any other types as long as their adjacency information is provided. To update vertices C we used only the incoming messages sent by their neighbors of type V. To update vertices V, however, we used not only incoming messages from its C neighbors but also raw embeddings of their own type neighbors.

```
tgn = GraphNN(
  {
    # τ_V = ℝ^d
    'V': d,
    # τ_C = ℝ^d
    'C': d
  },
  {
    # M ∈ 𝔹^{|V|×|V|}
    'VV': ('V','V'),
    # VC ∈ 𝟙^{|V|×|C|}
    'VC': ('V','C')
  },
  {
    # μ_{V→C} : τ_V → τ_C
    'V_msg_C': ('V','C'),
    # μ_{C→V} : τ_C → τ_V
    'C_msg_V': ('C','V')
  },
  {
    # V^{(t+1)}, V_h^{(t+1)} ← φ_V(V_h^{(t)}, M_VV × V^{(t)},
    #                M_VC × μ_{C→V}(C^{(t)}))
    'V': [
      {
         'mat': 'M',
         'var': 'V'
      },
      {
         'mat': 'VC',
         'var': 'C',
         'msg': 'C_msg_V'
      }
    ],
    # C^{(t+1)}, C_h^{(t+1)} ← φ_C(C_h^{(t)}, M_VC^⊺ × μ_{V→C}(V^{(t)}))
    'C': [
      {
        'mat': 'VC',
        'msg': 'V_msg_C',
        'transpose?': True,
        'var': 'V'
      }
    ]
  },
)
```

Listing A.1: TGN kernel of an end-to-end differentiable neural-symbolic model to answer whether or not a given graph accepts a k coloring.

APPENDIX B — RESUMO EXPANDIDO

Técnicas baseadas em aprendizado profundo têm recorrentemente atingido desempenho de estado-da-arte em diversas áreas ao longo dos últimos anos. Ainda há, no entanto, uma certa falta de compreensão em como problemas simbólicos e relacionais podem se beneficiar de modelos cuja arquitetura é baseada em aprendizado profundo. O caminho mais promissor para essa tão desejada integração consiste em arquiteturas neurais cuja propriedade de compartilhamento de parâmetros baseia-se em grafos e, dessa forma, podem ser treinadas para aprender características complexas de dados relacionais. Diversos problemas NP-Completos, tais como satisfatibilidade booleana e problema do caixeiro viajante, apresentam esse tipo de característica. Em ambos casos, um metamodelo chamado *Graph Neural Network* (GNN) pode trabalhar diretamente com entradas em formato de grafos, que representam uma instância do problema, e aprender a produzir uma resposta binária para o problema em questão. Nessa dissertação, estamos particularmente focados em aplicar um modelo de GNN ao problema da coloração de grafos: o modelo que propomos se aproveita de propriedades específicas desse problema ao contemplar tanto vértices quanto cores com representações internas na sua arquitetura e ao fazer com que tais representações passem por diversas etapas de troca de mensagens. Nesse sentido, a arquitetura que propomos é capaz de refletir a estrutura relacional do problema original, sem necessidade de uma redução em tempos polinomial para outro problema, enquanto ainda emprega uma estratégia de compartilhamento de parâmetros em função de vértices e cores. Nós também demonstramos como treinar tal modelo com instâncias muito difíceis, geradas de uma maneira adversarial: nós geramos pares de instâncias que são grafos no limite da satisfatibilidade – uma instância positiva e outra negativa que diferem apenas por uma única aresta, tal aresta faz com que a segunda instância não seja colorável por um dado número de cores C, enquanto a primeira permanece sendo minimamente colorável com C. Obtivemos uma acurácia de 83% durante treinamento e verificamos que nosso modelo é capaz de generalizar, até certo ponto, esse desempenho para instâncias de teste – não-vistas durante treinamento e que foram amostradas de diferentes distribuições. Nós mostramos que esse desempenho superou o desempenho de duas heurísticas e o desempenho de uma suposta abordagem neuro-simbólica generalista. Por fim, nós exploramos a memória interna do nosso modelos e encontramos evidências de como o seu raciocínio é construído em volta dos valores de representação de vértices e cores. Em suma, nossos resultados sugerem fortemente que GNNs são, de fato, ferramentas poderosas para resolver proble-

mas combinatoriais mas que seu aprendizado pode ser amplamente melhorado quando as propriedades de um problema são totalmente agregadas à arquitetura neural e nenhuma conversão de problema é feita.