DISS. ETH NO. 30015

Explaining and Improving Communication in Graph Neural Networks

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

LUKAS FABER

MSc IT Systems Engineering, University of Potsdam, Germany
born on 19.04.1993

accepted on the recommendation of

Prof. Dr. Roger Wattenhofer, examiner
Prof. Dr. Makoto Yamada, co-examiner

2024

First Edition 2024

Copyright © 2024 by Lukas Faber

Series in Distributed Computing, Volume 44
TIK-Schriftenreihe-Nr. 209
Edited by Roger Wattenhofer

Abstract

This thesis studies improvements for message-passing Graph Neural Networks (GNNs) from two angles. First, we tackle the problem of understanding how GNNs reason. We quantify how much GNN architectures achieve the combined reasoning over features and edges that GNNs uniquely offer. We then present two methods to explain GNN predictions. The first explanation method explains graphs by contrasting them to similar examples. The second explanation method uses decision trees to build recipes that explain the important inputs and how the GNN uses these inputs. We finish our understanding chapter by examining current explanation datasets. We discover several problems with these datasets that can lead to wrong conclusions. Instead, we propose three new datasets to evaluate explanation methods.

For the second angle, we investigate the limits of message passing. We identify the message aggregation step to potentially lose much information, contributing to problems identified by past research, such as Oversmoothing, Underreaching, or the $1-$WL expressiveness limit. We propose a new architecture that avoids aggregation entirely and processes individual messages asynchronously. We also identify a problem with neural architectures to learn comparisons between numbers and propose a new architecture well-aligned to learn comparisons. For example, we can use this architecture in a GNN setting to learn shortest paths.

Zusammenfassung

In dieser Doktorarbeit betrachten wir zwei Ansätze zur Verbesserung von Graphneuronalen Netzwerken (GNNs), die dem Nachrichtenaustauschprinzip folgen. Im ersten Ansatz verfolgen wir zunächst ein besseres Verständnis von GNNs. Wir quantifizieren, inwiefern existierende GNN Modelle das gleichzeitige Berechnen von Knotenattributen und Kanteninformationen ausnutzen können. Im Anschluss präsentieren wir zwei Methoden zur Erklärung von GNN Vorhersagen. Die erste Methode verwendet kontrastiert den Graphen, den wir erklären wollen mit ähnlichen Graphen aus dem Trainingsdatensatz. Die zweite Methode verwendet Entscheidungsbäume, um sogenannte Rezepte zu erstellen. Diese Rezepte erklären für alle Graphen eines Datensatzes welche Teile der Eingabe wichtig sind, und wie das GNN diese Eingaben verwendet um die Vorhersage zu berechnen.

Im zweiten Ansatz betrachten wir die Grenzen des Nachrichtenaustauschprinzips. Wir haben festgestellt, dass sich viele Probleme von GNNs auf den Aggregationsschritt der erhaltenen Nachrichten mindestens teilweise zurückführen lassen. Aus diesem Grund haben wir eine neue GNN Architektur entwickelt die komplett auf Aggregation verzichtet. Stattdessen verwenden wir asynchrone Kommunikation um jede Nachricht einzeln zu verarbeiten. Zuletzt haben wir festgestellt, dass neuronale Netzwerke generell Schwierigkeiten haben, Vergleiche zwischen Zahlen zu lernen. Wir stellen eine neue dedizierte Architektur zum Lernen solcher Vergleiche vor, die wir beispielsweise dazu verwenden, mit GNNs besser kürzeste Pfade zu lernen.

Acknowledgments

I want to thank all of my fellow DisCo students for the last years I had with you. During these 4.5 years, I met a lot of amazing people. Thanks, Béni, Henri, and Roland for making me an honorary member of G94 together with Zeta. I cherished our fun times with Tichu and other things both in and outside of Disco. Also thank you Roland for welcoming me into the group, Henri for the spire-slaying, and Zeta for the Tichu masterclasses and how to play the "literally-worst-hands-I-have-ever-had-in-my-life". I also thank the travel squad Andreas, Benjamin, Florian, Joel, and Karolis for the two amazing trips to Dijon and Milan, and hope we manage some more. Thanks, Luca for carrying my posters around the world to meet up in Hawaii. On a work-related note, thanks Benjamin for the shared late evenings/suffering before deadlines, Joel for the pro memeing, and Karolis for making all the numbers bold. When I joined Disco Pankaj thankfully taught me about the importance of cake. Thanks, Quentin for teaching us about the modern re-interpretation of cake through bananas and Till for teaching that not all sweets have to be cake. I also hope I get to savor Florian's beer eventually

The most important part of the PhD is learning. Some learning was through a reading group that I enjoyed with Béni, Damian, Gino, and Oli. Thanks, Damian for finding exotic papers, Gino for teaching me critical thinking on papers, and Oli for bringing out the good in every paper. I also want to thank Sal for bringing Lulu into our group as a postdoc who gave us veteran experience on PhD life. Some learning was about life where I want to thank Darya for building professional swimming goggles with me and Judy and Robin for the climbing tipps. Thank you Andras for your extraordinarily funny life-commentary of my CoTi miseries and life in general and the some hilarious paper writing sessions. Thank you Andrei for philosophical dis-

cussions about the German language and number systems. I also want to thank Kobi, Tejeaswi, and Yann for the teachings/preachings about crypto, trust, society, the best mensa (the jury is still out on Platte), and what I cannot better describe than life in general. Another important part of the PhD is learning to teach students. From Simon, I learned how to manage many students at the same time, but Peter taught me (or rather all of us) how many students you can actually supervise at once. Ye taught me how to ethically involve students as participants in research and Zhao taught me how to manage lectures efficiently—which I found so particular I coined it Zhaoing a meeting. Thanks, Lioba for the fun and rough times where we learned together to hire professors.

Now that I finish the PhD I will completely switch to industry life, for which I was thankfully prepared by Ard and Diana through their extensive internship-obtained industry experience. I look forward to matching your experience in a few years—maybe. Andreas thankfully interns at my new workplace to keep me up on the latest DisCo news.

I also want to thank Roger for his guidance over the last 4.5 years during the PhD. I learned a lot from you and would not have finished without your help. Outside of work, I really enjoyed playing Tichu with you and Chess against you.

Those last 4.5 years were not always easy so I am very grateful for the support I got from my family and my friends to keep me motivated and on track to finish. Many thanks go to Trier, Mainz, Freiburg, Munich, and Potsdam!

Collaborations and Contributions

Most of the chapters in this book are based on preprints or published papers written in collaboration with fellow researchers. Below is a list of the co-authors per chapter, whom I would like to thank for their contributions and especially for the interesting meetings that we had during these projects. The papers marked with a * list the authors in alphabetical order.

Chapter 3 is based on the preprint Should Graph Neural Networks Use Features, Edges, Or Both? [Faber et al., 2021b]. Coauthors are Yifan Lu and Roger Wattenhofer.

Chapter 4 is based on the publication* Contrastive graph neural network

explanation [Faber et al., 2020]. Coauthors are Amin K Moghaddam and Roger Wattenhofer.

Chapter 5 is based on the publication GraphChef: Learning the Recipe of Your Dataset [Müller et al., 2023]. Coauthors are Peter Müller, Karolis Martinkus, and Roger Wattenhofer.

Chapter 6 is based on the publication* When Comparing to Ground Truth is Wrong: On Evaluating GNN Explanation Methods [Faber et al., 2021a]. Coauthors are Amin K Moghaddam and Roger Wattenhofer.

Chapter 7 is based on the publication GwAC: GNNs with Asynchronous Communication [Faber and Wattenhofer, 2023a]. Coauthor is Roger Wattenhofer.

Chapter 8 is based on the publication Neural Status Registers [Faber and Wattenhofer, 2023b]. Coauthors is Roger Wattenhofer.

Contents

1 Introduction **1**

I Explaining Communication in Graph Neural Networks **4**

2 Understanding Communication **5**

3 Graph Neural Network Necessity **11**
3.1 A Metric for Correct Predictions 12
3.2 Experiments 13

4 Contrastive GNN Explanation **17**
4.1 CoGE: Contrastive GNN Explanations 19
4.2 Experiments 21

5 Decision Tree Recipes to explain GNNs **25**
5.1 GraphChef Recipes 27
5.2 Recipe Improvements 30
5.3 Experiments 32

6 Explanation Evaluation Pitfalls **41**
6.1 Pitfalls for Explanation Evaluation 42
6.2 Three New GNN Explanation Benchmarks 47
6.3 Experiments 53

II Improving Communication in Graph Neural Networks 58

7 GwAC: Asynchronous GNNs 59
7.1 Limitations of GNNs 59
7.2 No Aggregation Asynchronous Communication 62
7.3 The GwAC framework 63
7.4 Expressiveness Analysis 70
7.5 Underreaching and Oversmoothing 76
7.6 Complexity and Efficiency 78

8 Neural Status Registers 82
8.1 Neural Status Registers 84
8.2 Experiments 88

9 Conclusion 98

1 Introduction

Graphs are much like the force (and apparently economics [Feinstein, 2015]): "They surround us, penetrate us, and the graphs' edges hold the galaxy together." We can find graphs almost everywhere: Social Networks between people, Financial Networks in Economics, Traffic Networks in Civil Engineering, Bindings between atoms in molecules in Chemistry, Interactions between Proteins or Enzymes in Biology, or Interactions between particles in Physics. Naturally, the recent surge of neural network-based models cannot ignore a field with such rich applications. Graph Neural Networks became the workhorse for deep learning in graph-structured data. In 2019, GNNs have just started to attract interest. It was an interesting time for research in GNNs because the field was new and unexplored. In particular, what GNNs can or cannot do was still open. The first graph-first explanation methods also just started appearing in that year.

In the first part of the thesis, we connect to this zeitgeist and understand what these GNN models can do. First, we investigate which kind of predictions GNNs newly allow on then-popular graph datasets. While existing methods can easily reason about node features (red nodes are important) or graph structure (degree two nodes are important), GNNs now seamlessly combine both: Only red degree two nodes are important. Then, we dive

into the main topic of GNN explanation methods, presenting two new ones. Explaining GNNs differs from other modalities since we have to consider the discrete edge information. This difference also makes it easy to mistake explanations (What part of the input makes the GNN predict a class) as adversarial attacks (what part of the input can we change to cause a misclassification). We can easily create graphs that differ from every graph in the training set by perturbing edges and the graph structure. GNNs may behave abnormally on such graphs. The first explanation chapter investigates this phenomenon more closely and proposes a training-distribution-compliant method. In the second explanation chapter, we present an explanation method with a new quality: We can identify the important inputs and additionally understand the decision-making process. The recipes unveil the GNN reasoning through a series of decision trees. We found issues with existing datasets during both methods' evaluations that can lead to false conclusions about whether an explanation method works. We finish the explanation part with an analysis of properties we need for explanation datasets and three benchmarks following these properties.

In the second part of the thesis, we connect GNNs to our research group: Distributed Computing. Loukas [2020], Sato et al. [2019], Sato [2020] characterized the similarities between GNNs and the LOCAL distributed computing model. GNNs also inherit some of the problems from these models, which are partially aggravated by aggregating messages before processing them. Oversmoothing is one problem where nodes become more indistinguishable with every GNN layer. Oversmoothing can lead to Underreaching when we cannot use enough GNN layers. Furthermore, GNNs cannot distinguish several classes of different graphs. They are limited in the same way as the Weisfeiler-Lehman graph isomorphism heuristic. We take inspiration from other distributed computing models to tackle these limitations and propose an alternative framework for GNNs following asynchronous communication. Notably, this framework processes every message independently and avoids aggregation completely. We show that this framework can help to remedy the mentioned problems. We finish wiht a case study on shortest paths which are a common algorithmic problem for GNNs. We found that the arithmetic problem of comparing numbers that we need for finding shortest paths (to find the smallest distance) is a task neural networks struggle with. Generally, we can evaluate comparisons on numbers the model has seen during training but not to much larger numbers. This suggests that models do not actually understand how to solve comparisons but find a heuristic that works on the training set. We present a new architecture that is well-aligned to learn comparisons to solve this shortcoming and help in learning shortest paths.

Background on Graph Neural Networks

In this thesis, we look at GNNs in the message-passing framework proposed by Gilmer et al. [2017], Battaglia et al. [2018]. In this framework, nodes operate synchronously in rounds. In every round, every node sends a message to every neighbor based on its state plus potentially edge information and the recipient's state. Next, all nodes aggregate their received messages using permutation-invariant functions. Then, every node transitions to a new state based on its previous state and the message aggregation. Generally, we realize the message computation and node update through learnable neural functions.

Different GNN architectures propose different ways to realize those functions. Graph Isomorphism Networks [Xu et al., 2019c] sum the incoming messages before combining the aggregation with the node state. For example, Graph Convolutional Networks [Kipf and Welling, 2017] scale messages by the involved nodes' degrees to achieve laplacian smoothing. Graph Attention Networks [Veličković et al., 2018] use the attention mechanism to direct the receiver's attention to important messages. There are many more possible turning knobs, which You et al. [2020] mapped out in a design space. Is there edge information to incorporate into the messages? Which aggregation method to use? Do we include skip connections in the model?

Part I

Explaining Communication in Graph Neural Networks

2

Understanding Communication

Graph Neural Networks are no exception compared to other neural architectures when it comes to being able to understand—or rather not being able to understand— their decision process. If we have no insight into the decision-making process, we can use GNNs only as black-box models. This, in turn, limits our possible trust in the model and may prevent us from applying GNNs in safety-critical domains such as healthcare. How can we reveal the decision-making process of GNNs?

The neural functions for computing messages, node, and edge updates allow for complicated reasoning. Furthermore, the GNN decision-making process further uniquely involves the exchange of messages. For example, we additionally need to understand: Which edges between which types of nodes are important? How do we route the important information to the relevant nodes? This unique reasoning over edges is furthermore discrete: In most graphs, edges to propagate messages either exist or do not exist. This differs from the continuous reasoning over activations in most neural network architectures. Many approaches have been proposed in recent years to address these unique explanation requirements, some inspired by explanation methods from other domains, some uniquely built for GNNs.

Instance-Level Explanations

Most explanation methods follow one the approaches in Figure 2.1: Compute an importance score for each node (b); Compute an importance score per edge (c), or identify an explanation subgraph(d). We can easily transform these each ideas into each other: We can transform each importances into node importances by aggregating the importances of all adjacent edges. Vice versa, we can average two adjacent nodes' importances to derive the importance of an edge. To derive a subgraph from importances, we choose all nodes or edges whose importance surpasses a threshold. When we have an explanation subgraph, we assign every node and edge inside the subgraph an importance of 1 and all other inputs 0. Either way, we derive the explanations for a particular graph and explain why the GNN assigns this graph the predicted label.

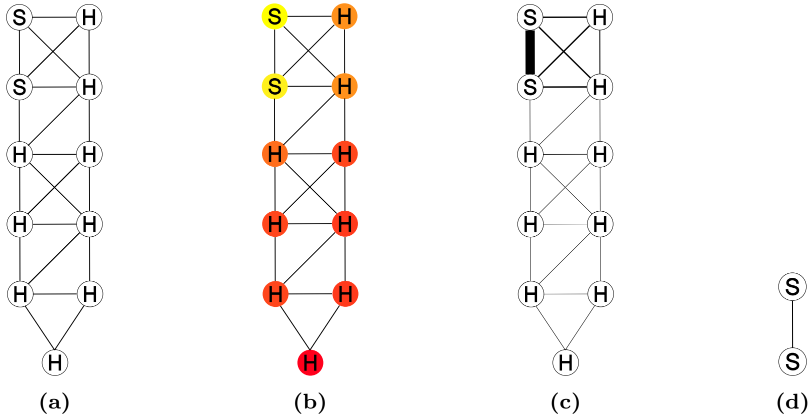


Figure 2.1: Overview of different explanation methods. (a) Graph for which we want to explain a prediction. (b) Node importance explanations, the brighter the node the more important it is. (c) Edge importance explanations, the thicker an edge the more important it is. (d) Subgraph explanation, retaining the important subgraph of the input.

Gradients are one example way to compute such scores. For example, Baldassarre and Azizpour [2019] and Pope et al. [2019] propose gradients with respect to the input whose perturbation maximizes changes in predictions. Gradient-based ideas are commonly used in computer vision, for example, in the Grad-CAM method [Selvaraju et al., 2017a]. Schlichtkrull

et al. [2021] extend this idea with continuous masks over edges, which allows computing edge gradients and edge scores as well. While Feng et al. [2022] is not a gradient method, it backtracks the contributions towards the prediction through the GNN. The backtracked importances are composed into subgraphs.

Counterfactual methods build on top of occlusion from computer vision [Zeiler and Fergus, 2014]: If removing parts of the input changes the GNN prediction drastically, these parts must be important. Indeed, Lucic et al. [2021] show that we can find compact explanations in the form of few counterfactual edges. However, we have to be careful not to mistakenly exploit model weaknesses in the form of adversarial attacks, which also only need a few perturbations [Zügner et al., 2018, Zügner and Günnemann, 2019, Geisler et al., 2020]. To this end, Bajaj et al. [2021] focus on perturbations that simultaneously work for multiple input graphs. The idea is that such perturbations capture the GNN decision process and not adversarial artifacts. Ma et al. [2022] derive counterfactual changes not directly in the graph space but train a variational autoencoder and use its latent space to generate explanations. Chen et al. [2023], Fang et al. [2023] explicitly learn a graph distribution and only accept explanations that respect this distribution.

Optimization-based methods define an optimization objective whose maximum constitutes the explanation. GNNExplainer [Ying et al., 2019] optimizes the mutual information between masks of node features and edges and the prediction labels. They also introduce a set of explanation benchmarks that became popular in many follow-up works. Luo et al. [2020] also build a mutual-information objective. They extend their objective to find explanations that can explain many graphs at once. Chen and Ying [2023] employ mutual-information between graph changes and predictions to find temporal explanations.

Yuan et al. [2021], Ye et al. [2023] use monte-carlo-tree-search to explore the space of all possible **subgraph explanations**, scoring the subgraphs with Shapley values [Shapley, 1953]. Xia et al. [2023] use monte-carlo-tree-search on the space of graph changes to find explanations in temporal graphs. Li et al. [2023] propose an alternative approach to explore the space of subgraphs to build explanations: they incrementally build explaining subgraphs through a learnt policy neywork. Wu et al. [2023] conversely build explaining subgraphs by matching parts of graphs from the training set and extract the commonalities

Example-based methods explain a given graph through similar or dissimilar examples. Huang et al. [2020] adopt the LIME algorithm [Ribeiro et al., 2016] for graphs. They explain a graph through linear approximation of the decision boundary near that graph. They obtain this decision boundary by interpolating the most similar graphs with the same and different labels. Vu and Thai [2020] capture correlations between inputs and predictions with probabilistic graphical models. Dai and Wang [2021] use similar examples to transform their GNN into a k-nearest-neighbor classifier where the nearest neighbors simultaneously decide the prediction and provide an explanation. We will propose one more example-based explanation method in Chapter 4. Example-based explanation methods are naturally resistant to mistake adversarial attacks for explanations since explanations are other graphs from the training distribution.

[Duval and Malliaros, 2021] learn a **simpler** surrogate model from their GNN that they use as an explanation. An alternative to post-hoc explaining complex GNNs is training simpler GNNs that are understandable through **inspection**. The question is how much classification performance such simplicity might sacrifice. Cai and Wang [2018], Chen et al. [2019a], and Huang et al. [2021] show that simpler GNNs must not be much worse than the complex ones. However, these architectures are not built to be more explainable. Chapter 5 will feature the GraphChef model that intentionally sacrifices expressive power but with the goal of being more understandable.

Model-level explanations

Another set of explanation methods does explain individual graphs but instead explains how a GNN generally reasons on the entire dataset. For example, we want to understand the key characteristics that make a GNN predict a certain class. Zhang et al. [2021] learn a prototype per prediction class. A prototype tries to capture the essential structure per class, removing any noise that might be present on training graphs. In the same spirit, Yuan et al. [2020a] and Wang and Shen [2022] define a graph generation problem to generate a graph per class that maximizes the GNN confidence to predict this class. Lv and Chen [2023] define desirable properties of model-level explanations and search for such explanations through greedy optimization. Azzolin et al. [2022] build a framework to leverage the instance-level explanation methods from the previous section. Their framework combines many instance-level explanations to build model-level explanations in the form of simple decision rules. However, all these methods showcase which inputs are important or how the input should be structured. They do not explain how

the GNN uses these inputs and do not reveal the decision-making process. In chapter 5, we will present a new explanation method, GraphChef, that finds model-level explanations and reveals the decision-making process.

Desirable Properties of Explanation Methods

First and foremost, we want an explanation method to find correct explanations. We can measure correctness as either Fidelity or Accuracy. However, there are further desirable properties of an explanation method.

Accuracy Sanchez-Lengeling et al. [2020], Yuan et al. [2020b]] introduce Accuracy for cases when we know which nodes and edges should be the explanation for a GNN prediction. Accuracy measures the overlap between the explanation and these known inputs: the higher Accuracy, the better the explanation.

Fidelity Pope et al. [2019], Yuan et al. [2020b], Amara et al. [2022]] define Fidelity for cases when we do not know which nodes and edges are the correct explanation. A good explanation should highlight the unique necessary parts of the input that drive the GNN prediction. Removing these necessary inputs should then change the model prediction. Fidelity captures the difference in model prediction with the explanation being removed or not. A good explanation should lead to large differences.

Stability Sanchez-Lengeling et al. [2020], Yuan et al. [2020b], Agarwal et al. [2022]] define that if the explanation found the key input, the explanation should not change when we randomly change small amounts of the input.

Faithfulness An explanation method should explain a model, not try to solve the prediction itself. A faithful explanation method captures the model prediction, even when it might be wrongSanchez-Lengeling et al. [2020], Yuan et al. [2020b], Agarwal et al. [2022]. [Agarwal et al., 2022] extend this concept with Fairness: If the underlying model is biased/unbiased, the explanation method should have/avoid the same biases.

Consistency One opposite idea is that an explanation should be consistent across the top performing models [Sanchez-Lengeling et al., 2020]. However, this property assumes that all models reason in the same way. Otherwise, Consistency contradicts Faithfulness, which states that we should explain the model behavior exactly.

Sparsity A good explanation should be understandable by humans. Therefore, it should be sparse and consist of as few nodes and explanations as possible[Pope et al., 2019, Yuan et al., 2020b].

Contrastivity Explanations should be unique for each different class. In turn, this means that explanations for different classes should be different. Otherwise, the explanations do not find class-specific inputs [Pope et al., 2019].

Benchmarking Explanation Methods

Most works either use Fidelity or Accuracy to evaluate a good explanation method. Many works prefer accuracy on the benchmarks proposed by Ying et al. [2019]. For one example benchmark, they create a binary tree as a base graph and attach motifs, such as three-by-three grids, to some nodes in the graph. Every node has to predict if they are part of such a motif. The explanation ground truth is the whole motif for all motif nodes. These benchmarks have become popular and used in evaluation in many later papers. However, we will show in chapter 6 that these benchmarks have pitfalls that make them problematic for evaluation. Low accuracy may not necessarily mean bad explanations. Himmelhuber et al. [2021] found similar issues with these datasets. We will propose three attempts at ground-truth benchmarks that avoid such pitfalls. Recent works by Amara et al. [2022] and Agarwal et al. [2023] also propose new benchmarks to avoid the pitfalls from Chapter 6. Finally, Agarwal et al. [2022] analyze explanation methods theoretically, avoiding empirical pitfalls completely.

Using fidelity for evaluation is not without problems either. If removing a potential explanation takes the graph out of the training distribution, fidelity might be high even though the graph is not an explanation. Instead, we might have found an adversarial attack that takes the input out of the training distribution [Hooker et al., 2019]. In chapter 4, we show one approach to avoid this problem. Fang et al. [2023] explicitly learn the training data distribution to ensure their explanations do not fall outside. One more issue with fidelity arises when there are multiple valid explanations. Removing one correct explanation might not change the model prediction and would falsely receive a low fidelity score. We discuss this problem in more detail in chapter 6.

3

Graph Neural Network Necessity

Before we head to explain concrete GNN predictions, let us first understand GNNs on a higher level. The appealing novelty of GNNs is that they simultaneously exploit the node features and the graph structure with neural functions. Figure 3.1 illustrates the three modes. Learning methods before GNNs either make use of the node features (Figure 3.1a) such as using MLPs or extracting information from the graph structure (Figure 3.1c) with methods such as Node2Vec [Grover and Leskovec, 2016] or VERSE [Tsitsulin et al., 2018].

Let us investigate to what extent GNNs make use of this. Shchur et al. [2018] investigate that simpler models achieve comparable performance to GNNs despite these models using only the node features or the graph structure. Zhu et al. [2020] report that GNN architectures struggle and only perform as well as feature-only models (i.e., MLPs) when facing graphs with low homophily. This paper motivates our study to measure the extent to which we can solve a dataset through (i) node features, (ii) graph structure, or (iii) the combined reasoning of GNNs.

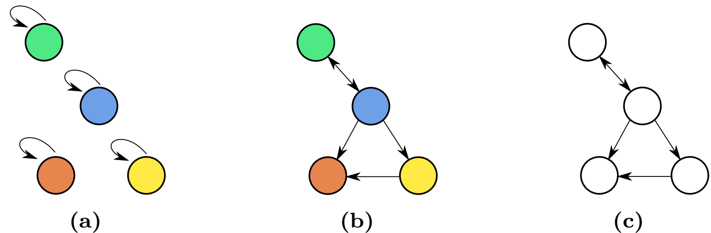


Figure 3.1: What is available for prediction? Left: Every node has only access to its local features. Right: Only the graph structure is available. Middle: A GNN can combine features and structure.

3.1 A Metric for Correct Predictions

Let us first define "solving". The most common form of empirical evaluation of a GNN, or any machine learning model, is to evaluate the correct predictions on the test set. Generally, we measure the best performance, averaging over several different seeds to combat (un)-lucky initialization. However, we want to compute a measure on the dataset that is even more robust against variance from different hyperparameter choices or initial weights. Furthermore, we a measure independent from the number of classes. For example, even a random model is correct 50% of the time.

We propose to define solving on the level of an individual prediction and by contrasting the model across many different instantiations to a random model. We count how many times the model predicts the correct class. We consider that a model solves a prediction when we cannot statistically explain the number of correct predictions through random guessing. When we reject this hypothesis, we can conclude that there is some systemic information that the model could exploit.

Our null hypothesis is that guesses follow a Bernoulli distribution with success probability $\frac{1}{c}$ where c is the number of classes. Suppose we conduct r runs on the model with n successes. We want to confirm or reject if such a Bernoulli process can explain those n successes. In the experiments, we will use a significance level of 0.001. For a dataset, we will define the solvable set S as the set of all models where we can reject the null hypothesis. We will now define operations on these sets.

3.2 Experiments

Evaluating Datasets

Let F be a model that uses only node features and E a model that uses only edge information. Consider the GCN[Kipf and Welling, 2017] convolution for a graph with n nodes:

$$H^{(l+1)} = \sigma(D^{\frac{1}{2}} A D^{\frac{1}{2}} H^{(l)} W^{(l)})$$

This convolution uses a modified graph adjacency matrix A where we add self-loops to every node. The diagonal matrix D of node degrees also includes these self-loops. $H^{(l)} \in \mathbb{R}^{n \times d_l}, H^{(l+1)} \in \mathbb{R}^{n \times d_{l+1}}$ are embeddings for every node per layer and $W^{(l)} \in \mathbb{R}^{d_l \times d_{l+1}}$. Our feature-only model does not have access to the graph information and would only compute $H^{(l+1)} = \sigma(H^{(l)} W^{(l)})$. This is equivalent to a standard MLP over the node features. For an edge-only model, we set the initial node embedding $H^0 \in \mathbb{R}^{n \times d_0}$ to a matrix of all ones. As alternatives, we considered extracting graph features (such as degree, page rank, and centrality measures) and computing an MLP over those. The feature-starved GNN architecture is a better choice because it is a more flexible approach that can compute the graph features it considers most useful. Propagation algorithms such as those used by Shchur et al. [2018] are other alternatives. We will define S_F as the solvable set by the feature MLP and S_E as the solvable set of the no-features GNN.

For good GNN datasets, these individual scores are as low as possible, meaning that only the combined information of features and edges provided by GNNs allows for correct predictions. We experiment with co-citation node classification datasets: Cora, CiteSeer, and PubMed [Yang et al., 2018b]. For graph classification, we use several datasets from Morris et al. [2020]: MUTAG, ENZYMES, AIDS, and PROTEINS.

We further experiment with several GNN architectures: Graph Convolutional Networks (GCN) [Kipf and Welling, 2017], GraphSage (GS) [Hamilton et al., 2017], Graph Attention Networks (GAT) [Veličković et al., 2018], and Graph Isomorphism Networks (GIN) [Xu et al., 2019c]. We use the sum-pooling aggregation for GraphSage and GIN. We build all architectures with 4 layers and set the embedding size to 64. Table 3.2 shows the relative sizes of S_F, S_E, and S_{GNN} relative to all possible predictions. We union the feature and edge-only sets to measure how many predictions are left for a GNN to solve. We also measure the best union of features, edges, and GNN to measure how many new predictions the GNNs solve. We use GIN as

the base architecture for the edge-only model, which is the most expressive architecture considered. Furthermore, we report the best-performing GNN.

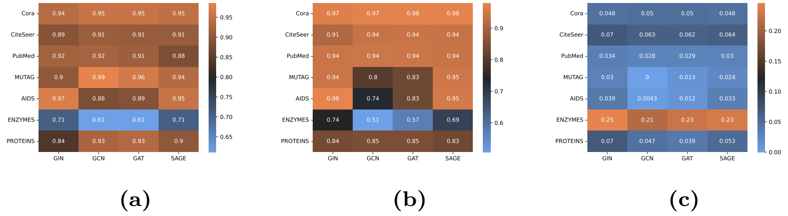
Dataset	S_F	S_E	$S_F \cup S_E$	S_{GNN}	$S_F \cup S_E \cup S_{GNN}$
Cora	0.803	0.780	0.923	0.925	0.969
Citeseer	0.752	0.593	0.85	0.812	0.909
Pubmed	0.892	0.626	0.93	0.877	0.961
MUTAG	0.739	0.910	0.94	0.888	0.973
AIDS	0.770	0.879	0.94	0.949	0.980
ENZYMES	0.373	0.473	0.652	0.570	0.800
PROTEINS	0.665	0.662	0.791	0.693	0.838

Table 3.2: Solvable sets for each dataset. Most datasets are largely solved via features or edges alone, except ENZYMES. However, GNNs learn to solve additional predictions as the last column shows.

Arguably, Cora, PubMed, MUTAG, and AIDS are not suitable datasets for GNN evaluation since we can solve almost all predictions in these datasets through features or edges alone. CiteSeer and PROTEINS can also be largely solved without requiring GNN reasoning. On the other hand, one-third of the predictions in ENZYMES need GNN reasoning. Nevertheless, we can see that the combined GNN reasoning unlocks some new predictions on most datasets. The difference is most noticeable on ENZYMES, where GNNs solve roughly half the predictions that neither features nor edges alone can solve. As the GNN score falls short of this last column, GNNs also forget some predictions. Let us now analyze this forgetting further.

Evaluating GNNs

Next, we will evaluate the different GNN architectures in three ways: 1) How well can they predict the same predictions as the feature-only model, 2) the edge-only model, and 3) go beyond these two models and solve the predictions that neither of these methods can. For each GNN architecture, we compute S_{GNN} and compute the ratio of predictions of 1) S_F, 2) S_E, and the ratio of S_{GNN} that is neither in S_F nor S_E. The first two scores indicate how much GNNs retain the power to explain simpler predictions. The third metric highlights how much the GNN can solve more difficult



(a) **(b)** **(c)**

Figure 3.3: Analyzing solvable sets for GNNs in comparison to a feature-only and edge-only model. (a) What ratio of the feature-only model can be solved by a GNN as well? GNN reasoning is a superset of feature reasoning. Values closer to 1 are better. (b) What ratio of the edge-only model can be solved by a GNN. (c) What ratio of solvable set of the GNN cannot be explained through feature or edge reasoning? Ideally, GNNs capture more complex relationships, higher values are better.

predictions. Figure 3.3 shows the results.

Figure 3.3a and Figure 3.3b show that GNNs can generally solve the same predictions that feature-only and edge-only models can solve. This is expected since GNN reasoning is a superset of these approaches. Figure 3.3c shows that GNNs are not only a sum of features and edges but that combining features and edges allows new predictions. Outside of ENZYMES, most GNNs can predict a few extra predictions on most datasets.

Comparing GNNs

Last, we compare different GNN architectures to what extent the architectural differences allow us to learn different predictions. For every architecture GNN_1, we measure how many predictions are solved by another architecture GNN_2 and report this in Figure 3.4. For example, the number 0.97 in the top right corner of the first heatmap encodes that GIN can solve 97% of the predictions that SAGE can solve on the Cora dataset. Most GNN architectures agree and learn to solve the same predictions on most datasets. However, the interesting cases are when different GNN architectures learn to solve different predictions. This potentially allows for improving dataset results through ensembling different GNNs. For example, if we perfectly ensembled all different architectures on the ENZYMES dataset, we could already solve 86.6% of predictions.

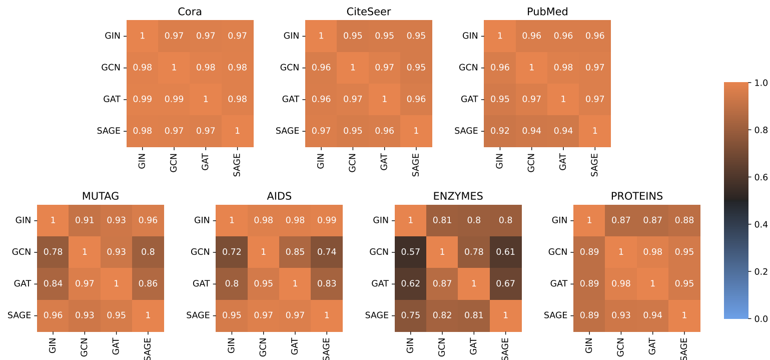


Figure 3.4: Per dataset overlap of different GNN architectures. Every entry in the heatmap encode the ratio of the solvable set of the column GNN that the row GNN can also solve. For example, 0.97 in the top right corner of the Cora heatmap encodes that GIN can solve 97% of the predictions of SAGE.

4

Contrastive Graph Neural Network Explanation

In this chapter, we examine the interplay between explanations and adversarial attacks, which try to achieve almost the same goal: Adversarial attacks want to find a small subset of edges or node features that, when perturbed, change the prediction of the model [Zügner et al., 2018]. The Fidelity interpretation of explanations is finding a subset of edges or node features that are key for the model prediction. Being key to the prediction means a model would change its prediction with those key parts removed. This relationship gives us the following problem: We find a subset of the graph that changes the model output if removed (a common scenario for a counterfactual explanation). How can we decide whether this subset is an adversarial attack that exploits an anomaly or a good model explanation?

Let us look at one extreme illustrating example. We assembled a simple graph classification problem. Randomly created trees as base graphs have cliques attached, while other trees do not. We train a GNN to distinguish which trees have cliques attached. Then we use a simple Occlusion [Zeiler and Fergus, 2014, Ancona et al., 2018] explanation, where we mask each edge once at a time and rerun the GNN. The more the model prediction

changes away the graph containing a clique, the more important the edge. Figure 4.1a shows the results for one example graph. Brighter edges are more important. Surprisingly, Occlusion does not consider the edges in the clique as the most important but different edges (highlighted with red circles).

In this dataset, we know that correct explanations are edges in the clique. This suggests that Occlusion does not find explanations but adversarial attacks. We hypothesize that removing the highlighted edges creates disconnected graphs, which take the underlying GNN out of the distribution of graphs seen during training (all connected graphs). In particular, deleting the most important edge would create an isolated node. Based on similar observations for computer vision [Hooker et al., 2019], a likely reason for this behavior is that the GNN behaves abnormally on these inputs outside of the training set. Occlusion erroneously picks up on these edges being important. Other methods, such as the gradient-based explanation (Figure 4.1b) and GNNExplainer (Figure 4.1c), experience the same phenomenon but to a lesser extent. These methods do not change the graph drastically, but they also consider subgraphs that are not part of the training data.

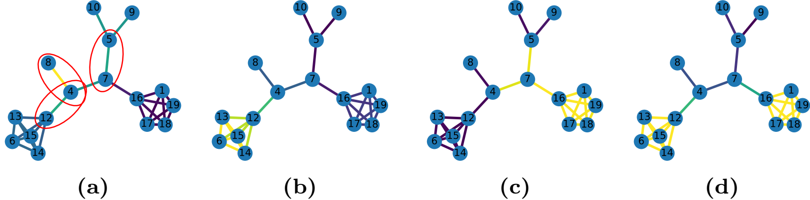


Figure 4.1: Comparison of explanations of different methods for a CYCLIQ dataset with different methods. The two cliques are the correct explanation. (a) Occlusion (b) Gradients (c) GNNExplainer, and (d) CoGE. Many methods highlight wrong edges.

We want to find explanations that we can be certain are not adversarial attacks. We want the explanation to focus on data consistent with the training data distribution to avoid adversarial attacks. We coin this a distribution compliant explanation. We introduce a new distribution-compliant explanation method that explains graphs by contrasting the graph versus very similar graphs (from the same dataset) with the same and different labels. We coin this method **Co**ntrastive **G**raph **E**xplanation. In the introduced taxonomy in Chapter 2, CoGE is an example-based method.

4.1 CoGE: Contrastive GNN Explanations

CoGE is a contrastive explanation method [Dhurandhar et al., 2018, 2019]: We derive explanations by finding commonalities and differences between the graph G to explain and other similar graphs. The process has two steps: 1) Identifying these similar graphs. To cover the full GNN decision space, they should cover graphs with the same label and graphs with other labels. 2) Identifying the explanation as commonalities and differences between G and those other graphs. In CoGE, explanations come in the form of importance scores for every node.

Finding Similar Graphs We need to define a notion of similarity between graphs in the dataset. Graph similarity measures are abundant in the literature that we could choose from [Sanfeliu and Fu, 1983, Heimann et al., 2018, Zhang and Lee, 2019, Wang et al., 2019]. However, we are not interested in generally similar graphs but graphs that are similar according to the GNN decision making. Therefore, we do not evaluate similarity on the raw graphs but compare similarities between the final GNN embeddings. These embeddings encode relevant node features and also the relevant structural information. Equally important, the GNN can drop non-important information before the final embedding. For our similarity purposes, we represent a graph as the multiset of its final node representation $\{x_1, x_2, \dots x_n\}$ where x_i are $d-$ dimensional embedding out of $\mathbb{R}^d$. Suppose we want to measure similarity between two graphs S and T with m and n nodes. In their multiset of final embeddings representation, we note them as

$$
\begin{aligned}
S &= \{s_1, s_2, \dots, s_m\} \\
T &= \{t_1, t_2, \dots, t_m\}
\end{aligned}
$$

What is left is measuring the similarity between two such multisets, which we inspire from Optimal Transport [Fey et al., 2020, Liu et al., 2020, Nguyen et al., 2021, Nikolentzos et al., 2017, Sato et al., 2020, Xu et al., 2019b,a]. We construct a transportation graph F. Nodes in F are the union of nodes from S and T. For edges, we connect every node from S to every node from T. We base similarity on the solution of the following min-cost flow on this graph F: Every node from S acts as a source that needs to send a volume $v_i = 1/m$ flow, every node from T is a sink that can accept a capacity $c_i = 1/n$ flow. Therefore, the total flow in the network will be 1. We do not constrain maximum flow on any edge, but sending flow across an edge (s_i, t_j) occurs cost proportional to the L_2 norm $\|s_i - t_j\|_2$. $L(S, T)$

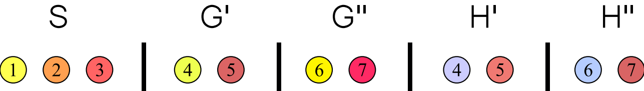


Figure 4.2: Five example graphs with different embeddings, similar embeddings receive similar colors. S is the graph we want to explain, G' and G'' have the same label, H' and H'' have a different label. In this case, node 1 is the explanation because similar nodes exist in G' and G'' but not in H' and H''.

is the cost of the min-cost flow of this graph. The lower the loss, the more similar S and T are. We compare G against every other graph and pick the top k graphs from the same and different classes with the lowest losses.

Finding Explanations We subdivide the graphs from the previous step into the set $\mathbb{G}^{\approx}$ of graphs with the same label as G and the set of graphs $\mathbb{G}^{\not\approx}$ with a different label. The explanation of G will be the nodes where there exist nodes with similar embeddings in $\mathbb{G}^{\approx}$ but such nodes do not exist in $\mathbb{G}^{\not\approx}$. We will again use flows with G taking the role of S above. We consider multiple flows simultaneously where each node in $\mathbb{G}^{\approx}$ and $\mathbb{G}^{\not\approx}$ serves as T in one flow. We optimize the sink volumes v_i (while their sum still remains 1) such that the sum over all minimal flows becomes minimal/maximal.

For a single graph T, if we choose v_i as to <u>maximize</u> the optimal transport between G and T, we assign low volumes on nodes in G that do not have a counterpart in T. Conversely, choosing volume to <u>minimize</u> optimal transport assigns low volumes to the nodes with counterparts. This allows us to define the explanation e for G as the <u>low-volume</u> nodes of the following optimization problem over the volumes $V = \{v_1, v_2, \dots v_m\}$:

$$e = \arg\min_{V} \frac{1}{|\mathbb{G}^{\not\approx}|} \sum_{G' \in \mathbb{G}^{\not\approx}} \mathcal{L}_V(G, G') - \frac{1}{|\mathbb{G}^{\approx}|} \sum_{G'' \in \mathbb{G}^{\approx}} \mathcal{L}_V(G, G'')$$

$L_V(S, T)$ is the same loss as above but using volumes V instead of uniform volumes, while capacities are still uniform. Our overall objective has two components: The first term pushes the volumes to minimize the loss between G and graphs in $\mathbb{G}^{\not\approx}$. This highlights the similarities to graphs with different labels. The second term changes the volumes to maximize the loss between G and graphs in $\mathbb{G}^{\approx}$, highlighting differences to graphs with the same. Nodes with low volumes become the explanation: They have no counterparts in graphs wiht different labels, and they have counterparts in

graphs with the same label.

Figure 4.2 shows an example of this idea. Here, the graphs G' and G'' are graphs from $\mathbb{G}^{\approx}$ and H' and H'' are from $\mathbb{G}^{\not\approx}$. For the first loss term, let us minimize the optimal transport between S and H' or H''. We will put the most volume on node 3, which has counterparts in nodes 5 and 7. For the second loss term, let us maximize the optimal transport between S and G' and 'G''. We will put the most volume on node 2, which has no counterparts. In combining those objectives, node 1 receives consistently low volumes and becomes the explanation.

We additionally use a third regularizing term: We compute the loss between G and its uniformly weighted version. This loss is minimal when using uniform volumes for V. It penalizes deviations from uniform volumes so we only consider clearly beneficial volume changes.

$$e = \arg\min_V \frac{1}{|\mathbb{G}^{\not\approx}|} \sum_{G' \in \mathbb{G}^{\not\approx}} \mathcal{L}_V(G, G') - \frac{1}{|\mathbb{G}^{\approx}|} \sum_{G'' \in \mathbb{G}^{\approx}} \mathcal{L}_V(G, G'') + L_V(G, G). \tag{4.1}$$

Which we abbreviate to the following three terms:

$$\begin{aligned}
L^{\approx} &= \frac{1}{|\mathbb{G}^{\approx}|} \sum_{G'' \in \mathbb{G}^{\approx}} \mathcal{L}_V(G, G'') \\
L^{\not\approx} &= \frac{1}{|\mathbb{G}^{\not\approx}|} \sum_{G' \in \mathbb{G}^{\not\approx}} \mathcal{L}_V(G, G') \\
L^{=} &= L_V(G, G) \\
e &= L^{\not\approx} - L^{\approx} + L^{=}
\end{aligned}$$

4.2 Experiments

We experiment with three datasets. Mutagenicity is a binary graph classification dataset containing 4337 nodes where we predict whether a molecule is mutagenic. REDDIT-BINARY is a binary graph classification dataset where we assign user-comment graphs to Q&A or Discussion style subreddits. CYCLIQ is a synthetic graph classification dataset based on the simple problem from Figure 4.1d. We use random trees as base graphs to which we either attach a cycle or a clique. We predict per graph which motif was attached. As nodes have no features, we initialize each node with feature vectors of all ones. We generate 2000 graphs and split them 80 : 20 into a

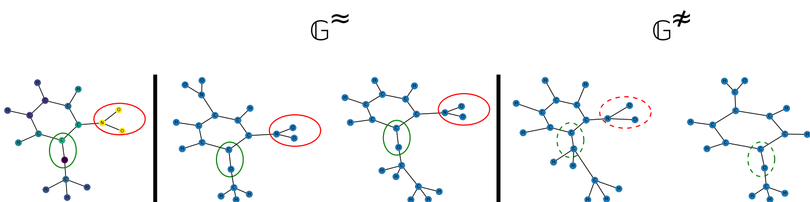


Figure 4.3: MUTAG Explanation showing important substructures in molecules. Left: Original graph, Middle: Similar graphs with the same label, Right: Similar graphs with a different label. Brighter nodes are more important in the explanation.

training and a test set.

For obtaining CoGE explanations, we contrast against 10 graphs, each with the same and different labels. We use the GeomLoss library [Feydy et al., 2019] for computing optimal transport. We minimize the objective in Equation 4.1 using gradient descent with the Adam optimizer [Kingma and Ba, 2014] with a learning rate of 0.01 for REDDIT-BINARY and 0.1 otherwise.

Mutagenicity Results

We expect correct explanations to contain the NO_2 subgraph, a known mutagenic structure [Debnath et al., 1991, Ying et al., 2019]. However, this group is insufficient and can also occur in non-mutagenic graphs. Figure 4.3 shows an example CoGE explanation for a mutagenic graph (left). The explanation consists of two unconnected subgraphs. The known NO_2 substructure and a C atom in a carbon ring connected to an O. We can validate that this explanation might be correct; we find both structures in the other closest mutagenic graphs (middle graphs). However, the structures are not simultaneously present in non-mutagenic graphs (right). We do not have the chemical knowledge to verify the explanation if the CO structure is also chemically relevant.

REDDIT-BINARY Results

We know that the Q&A graphs are more star-like. Few—or even only one—user replies to questions from many users. On the other hand, many users interact with other users in discussion graphs. Figure 4.4 shows a CoGE explanation for a Q&A graph. We can see the explanation highlighting a

star structure with the most importance assigned to the center nodes of the graph.

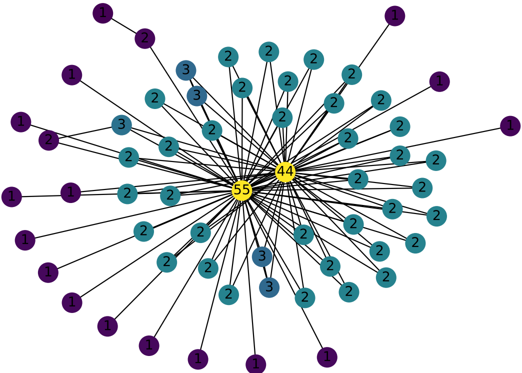


Figure 4.4: Example Q&A explanation. Numbers depict the node degree. Nodes connected to both central nodes are more important.

CYCLIQ Results

The correct explanation for the CYCLIQ dataset is the edges that form the cycle/clique. We can derive edge importances by summing the importance of adjacent nodes. Table 4.5 shows the explanation accuracy for CoGE and several other methods: node-based occlusion method (removing adjacent edges), sensitivity analysis [Pope et al., 2019], and GNNExplainer Ying et al. [2019] and a random baseline. CoGE achieves higher explanation accuracy than the other methods. At the same time, all methods can explain cliques much better than cycles. The reason for this is a defect in the dataset that we found later and will explain in Chapter 6.

CoGE Ablation

We last investigate the individual effects of each loss term in Equation 4.1 through an ablation study on the CYCLIQ dataset. Table 4.6 shows the CYCLIQ explanation accuracy for each subset of loss terms from Equation 4.1. Additionally, we validate the choice of optimal transport by comparing it to measuring similarity as Euclidean distance. We compute the volume-weighted average of nodes in each graph and compute the distance between

Table 4.5: Explanation accuracies on the CYCLIQ dataset.

Method	Cycle Acc.	Clique Acc.	Avg. Acc.
Random	0.41 ± 0.17	0.58 ± 0.13	0.49
Occlusion	0.39 ± 0.23	0.86 ± 0.16	0.62
Sensitivity	0.36 ± 0.2	0.87 ± 0.12	0.61
GNNExplainer	0.43 ± 0.18	0.73 ± 0.14	0.58
CoGE	**0.78 ± 0.18**	**0.99 ± 0.02**	**0.88**

the two averages.

The finer-grained optimization provides important information that leads to better explanations. Additionally, removing any loss term also causes explanation accuracy to drop significantly.

Table 4.6: Explanation accuracies on the CYCLIQ dataset for ablations of the loss function and the distance measure.

Loss	Cycle Acc.	Clique Acc.	Avg. Acc.
$-\mathcal{L}_W^{\approxeq}$	0.63 ± 0.25	0.6 ± 0.35	0.62
$-\mathcal{L}_W^{\approxeq} + \mathcal{L}_W^{=}$	0.62 ± 0.23	0.61 ± 0.27	0.61
$\mathcal{L}_W^{\not\approx}$	0.65 ± 0.23	0.99 ± 0.02	0.81
$\mathcal{L}_W^{\not\approx} + \mathcal{L}_W^{=}$	0.66 ± 0.24	0.99 ± 0.02	0.82
$\mathcal{L}_W^{\not\approx} - \mathcal{L}_W^{\approxeq}$	0.7 ± 0.2	0.99 ± 0.02	0.84
$\mathcal{L}_W$ and Average	0.45 ± 0.24	0.99 ± 0.02	0.71
$\mathcal{L}_W$ and OT	0.78 ± 0.18	0.99 ± 0.02	0.88

5

GraphChef: Decision-Tree Recipes to Explain Graph Neural Networks

This chapter presents a new explanation method, GraphChef. To the best of our knowledge, this is the first explanation method that does not only help to understand the important inputs of a GNN prediction. GraphChef also reveals the decision-making process. Figure 5.1 shows the difference between these two on the PROTEINS dataset.

PROTEINS is a binary graph classification dataset. Graphs are either an enzyme or not. The nodes are one of three secondary biological structures of amino acids: helixes (H, input 0), sheets (S, input 1), or turns (T, input 2). The figure shows explanations: for nodes in Figure 5.1a, for edges in Figure 5.1b, and by finding subgraphs in Figure 5.1c. All three explanations reveal that the double-sheet structure is important, but neither method explains why it is important for an enzyme: Does it need a pair of connected sheets? Does it just need to be more than one sheet? In contrast, Figures 5.1d and 5.1e show the recipe from GraphChef. This recipe allows us to follow the decision-making for any graph, whether it is an Enzyme or not. Furthermore, the recipe allows to understand for the whole dataset what makes an enzyme. In this particular instance, the graph has more than

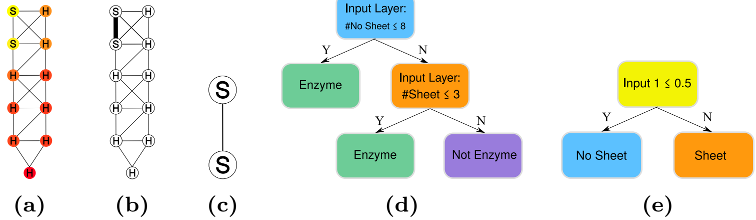


Figure 5.1: Showing the three example explanations from Figure 2.1. All methods highlight the connections between the two S nodes. However, none of the methods explain why these two nodes exactly are important. GraphChef shows the decision process: There must be less than three S nodes to classify the graph as Enzyme. Here, we only have two such nodes.

8 nodes that are not sheets, but it has less than 3 sheets. This explanation is consistent with existing findings for the PROTEINS dataset [Errica et al., 2020]. Therefore, the graph is an enzyme.

As Figures 5.1d and 5.1e show, GraphChef explains by transforming the decision-making into a decision tree. Similar ideas exist for non-graph neural networks. Boz [2002], Craven and Shavlik [1995], Dancey et al. [2004], Krishnan et al. [1999] investigated the first ideas to combine decision trees with neural networks. Schaaf et al. [2019] recently improved the approximation through regularization, encouraging sparsity and orthogonality in the neural network. Wu et al. [2017a] follow a similar regularization idea for time series neural networks. Their regularization penalizes weights that are difficult to model for decision trees. Kontschieder et al. [2015] directly learn neural decision trees by relaxing the routing in decision nodes to be probabilistic and learning the routing. Aytekin [2022] introduced an approximation theorem for decision trees to theoretically express neural networks. However, their approach requires exponentially many decision nodes. Such large trees are not understandable by humans.

GraphChef is closest in spirit to the works of Yang et al. [2018a]. They create a neural layer that learns to split data and put it into different bins. They stack their neural architecture with these layers. GraphChef also introduces a differentiable but inherently explainable layer, the dish layer (**di**fferentiable **s**tone **h**). In contrast to Yang et al. [2018a], GraphChef has to additionally reason about the graph structure and combine the presence/absence of edges with the node features. We will introduce a dish layer as a version of a GNN

layer. Instead of following the CONGEST [Peleg, 2000, Wattenhofer, 2020] distributed communication model, dish layers follow the simpler stone-age communication model [Emek and Wattenhofer, 2013]. It turns out that decision trees can easily model this layer.

5.1 GraphChef Recipes

Introducing dish GNN Layers

The first step to finding GraphChef recipes is building a GNN composed of dish layers. The starting point for these dish layers is GIN layers [Xu et al., 2019c]. These layers internally use **M**ulti-**L**ayer **P**erceptrons. MLPs are sequences of linear weight layers $W^{(l)}$ separated by nonlinearities σ. For example, a $3-layer$ MLP applied on a feature matrix $X \in \mathbb{R}^{n,d}$ looks like the following:

$$MLP(X) = \sigma(\sigma(XW^{(0)})W^{(1)})W^{(2)} \tag{5.1}$$

$W^{(0)}$ first dimension is equal to d we can choose the other dimensions freely as long as dimensions for matrix multiplications match. For example $W^{(0)}$'s second dimensions must equal $W^{(1)}$'s first dimension. GIN takes a node v's current embedding h_v and the sum of this node's neighbors' $Nb(v)$ embeddings and computes v's next embedding through such an MLP:

$$h_v^{l+1} = MLP(h_v^l, \sum_{w \in Nb(v)} h_w^l) \tag{5.2}$$

The internal representations h^l allow the encoding of complex information. These embeddings are mixed with neighborhood embeddings in the next GNN layer, which causes the interpretability of GNN to suffer. We take inspiration from distributed computing to make this layer understandable. The above GIN layers are close to the CONGEST computation model. Instead, we create a GNN layer close to the stone-age model [Emek and Wattenhofer, 2013]. In this model, nodes are more limited and have a categorical state. Stone-age nodes can only count the number of neighbors in each state for aggregating their neighbors. Furthermore, nodes can only count to an upper bound. In this spirit, we can create a GNN layer by transforming the node embeddings h^l into categorical values. We use Gumbel softmax [Jang et al., 2016, Maddison et al., 2016] that allows to draw samples from a categorical distribution. Suppose our GIN update from Equation 5.2 above computes d-dimensional embeddings. Per node, we can interpret the embeddings as per-state-probability logits π_s and sample state probability p_s

as follows:

$$p_s = \frac{exp((\pi_s + g_s)/\tau)}{\sum_{i=1}^{d} exp((\pi_s + g_s)/\tau)}$$

The terms g_s are independently sampled values from a Gumbel(0,1) distribution[Jang et al., 2016]. We can use the temperature τ to control how close the distribution should converge to a one-hot distribution. For one node, we obtain the per-class sampling probabilities by computing above update for each of its d logits. We note $Gumbel(X)$ with $X \in \mathbb{R}^{n \times d}$ as computing per-class probabilities for each of the d classes, computing each of the n nodes independently. This gives us the following dish update equation:

$$h_v^{l+1} = Gumbel(MLP(h_v^l, \sum_{w \in Nb(v)} h_w^l)) \tag{5.3}$$

Having categorical node states means that the summation of neighbor embeddings turns into counting the number of neighbors in each state. Figure 5.2 shows the computation in one dish layer. We compute the update for a node in state 0 and has 1, 2, and 1 neighbors in states 0, 1, and 2, respectively. Equation 5.3 corresponds to the red box. We complete the dish GNN with an encoder that discretizes the initial node features and a decoder with skip connections to every intermediate node state. For graph classification, the decoder can access sum-pooled embeddings for all intermediate layers.

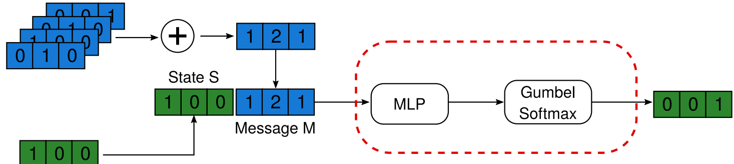


Figure 5.2: One dish layer. Nodes receive their previous state and counts of their neighbor states. A final Gumbel softmax outputs a categorical state.

This design does not need to lose expressiveness if the categorical node states are exponentially larger than the continuous GIN embeddings. However, we aim for small and, therefore, interpretable node states and accept the slight loss in expressive power.

GraphChef Recipes

GraphChef recipes consist of a series of decision trees. We find these trees by distilling the node update from Equation 5.3 into a decision tree. We do not change anything about the remaining architecture, shown in Figure 5.3. The red block again highlights the node update step. Because the dish layers compute categorical states, we can directly formulate the tree distillation as a classification problem: Given the previous node state and neighborhood counts, predict the followup state. Similarly, we distill the encoder and decoder layers.

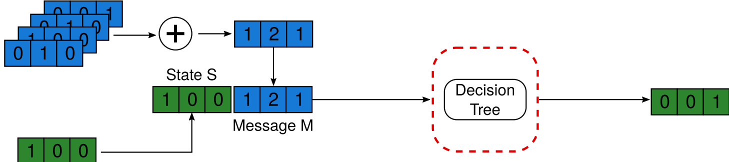


Figure 5.3: A GraphChef layer after distilling the MLP + Gumbel block into a decision tree.

Decision trees struggle with comparing two input features and picking the greater one. Such comparisons require a binary search over two features. To help produce small trees, we provide delta features as full pairwise comparisons of all neighborhood states as additional inputs to the decision trees. Figure 5.4 shows these additional features in yellow. The first entry is 0 because the count of neighbors in state 0 (1 node) is smaller than the number of nodes in state 1 (2 nodes). Similarly the third delta feature is a 1 since there are more nodes in state 1 than 0 (2 versus 1 nodes).

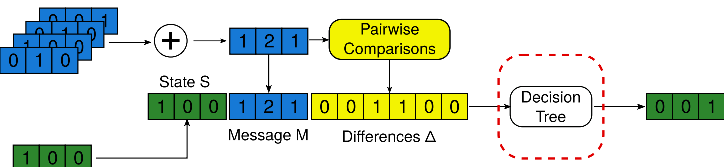
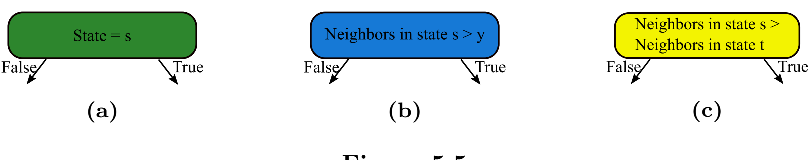


Figure 5.4: An improved version of a GraphChef layer that has additional features for comparing features.

Interpreting GraphChef Recipes

We can now interpret how GraphChef reasons per layer by looking at which features each decision split in the decision trees use. Figure 5.5 shows how we can interpret each feature. The colors match the feature colors in Figure 5.4. The trees can pay attention to the node's state (a) if there are at least a learned number of neighbors in a certain state (b) or if there are more neighbors in one than another state.



(a) **(b)** **(c)**

Figure 5.5

5.2 Recipe Improvements

Decision Tree Pruning

To further improve the human understandability of the recipes, we employ pruning to minimize the number of decision nodes in each tree. We use reduced error pruning for this [Quinlan, 1987], for which we need a quality criterion. We observed that neither the validation set nor the training set accuracy is a good criterion. The validation set would need to be bigger to cover all decision paths, leading to overpruning. The training set does not allow pruning artifacts that stem from overfitting on the training set.

Our criterion allows replacing a decision split with a leaf if (i) does not reduce validation accuracy and (ii) does not drop training accuracy below validation accuracy. The first criterion ensures that we do not overprune. The second criterion allows leveraging the larger training set. By letting the training accuracy drop back to the validation accuracy, we can prune the overfitting artifacts.

We prune our trees by iterating the inner nodes sorted by the number of training samples covered until we find a decision node we can replace with a leaf. We replace this node with a leaf and restart the iteration since this might change the scores for other nodes. We can finish pruning when no more nodes fulfill the quality criterion. Practically, we found it beneficial to keep pruning and allow slight deterioration in validation accuracy to prune

significantly more nodes. The choice of which drop in accuracy is acceptable for which amount of pruning is dependent on the dataset. Therefore, we leave it to the user.

Computing Explanation Scores

We can also use the recipes to compute node-level explanation scores similar to other explanation methods. While these are not as informative as the recipe containing the decision process, these scores can complement a recipe to highlight the important nodes for a particular graph. We compute these explanations as a matrix $E \in \mathbb{R}^{N \times S \times N}$. N is the number of nodes in the graph, and S is the number of classes. An entry (u, s, v) contains a value of how much node v contributes to node u being in state s. We normalize the matrix after every computation step so that values sum up to 1 for every pair u, s. We denote $e(n, s) = E[n, s, :] \in \mathbb{R}^n$ as the explanation for node n being in state s

For the importance computation, we follow a layer-wise importance propagation through the tree layers. We initialize the matrix to contain ones on the diagonal entries (u, s, u) for all states s. This means that every node is initially its own explanation. Per layer, we compute the importance of each feature available to the decision tree using TreeShap values [Lundberg et al., 2018]. Depending on the type of feature, we update explanation importance as follows:

State features Consider the simple example where the tree consists of one decision split where a node transitions into state s if it currently is in state s'. The explanation for the node for state s for this layer is the explanation for the node for state s' for the previous layer. It can also be the other way around that nodes transition to state s when they are **not** in state s'. Then, the explanation for s is the opposite of that for s'. The TreeShap values T_S also capture the interaction between all pairs of states for larger trees. Indicator variables $\mathbb{1}(n, s')$ are 1 when node n is in state s' before the current layer and -1 if it is not. This applies the correct sign for the explanation. We can compute the state explanation:

$$\sigma(n, s) = \sum_{s' \in S} T_S(s', s) \cdot e(n, s') \cdot \mathbb{1}(n, s')$$

Message Features Consider the simple example where a node n transitions to a state s if it has at least some neighbors in state s'. In this case, we define the message explanation for n being in state s as the average of

these neighbors' explanations for state s'. The TreeShap values T_M capture the dependencies between states for neighborhood splits. We compute the message explanation as follows:

$$\mu(n, s) = \sum_{s' \in S} T_M(s', s) \cdot \sum_{n' \in Nb(n,s')} \frac{e(n, s')}{|Nb(n, s')|}$$

Delta Features The delta features work similarly to message features. Consider that we compare if the number of nodes in state s' is larger than state s''. In that case, the explanation of neighbors in state s' contribute positively, but the neighbors in state s'' contribute negatively. However, if the comparison is true the other way around, we must invert which nodes contribute negatively. The TreeShap values T_D capture which comparisons between neighbor counts are important. Indicator variables $\mathbb{1}(n, s', s'')$ are 1 if s' neighbors are in the majority and are -1 otherwise. We can then compute the delta explanation as follows:

$$\delta(n, s) = \sum_{s' \in S} \sum_{s'' \neq s' \in S} T_D(s, s', s'') \cdot \mathbb{1}(n, s', s'') \cdot \frac{\sum_{n' \in Nb(n,s')} e(n, s') - \sum_{n' \in Nb(n,s'')} e(n, s'')}{|Nb(n, s')| + |Nb(n, s'')|}$$

We can then compute the new explanations by adding these three explanations to the previous explanations. Keeping the previous explanations helps to highlight whole motifs. After the addition, we renormalize the explanations $e(n, s)$ to be 1 again. The decoder layer with skip connections requires a slight modification of this approach. We can use the same scheme for node classification problems, except the features and explanations can be from more than just the previous layers. In graph classification problems, we expose the counts of nodes in each state from all layers to the decoder. We propagate the explanations as we do for message and delta explanations from those nodes in the given state in the given layer.

5.3 Experiments

The following experiments show that GraphChef retains much expressiveness compared to the not-explainable GIN. Quantitatively, we will investigate the classification accuracy of Graphchef against several baselines, the effectiveness of the proposed pruning method, and the explanation accuracy of the tree-derived importance scores. Qualitatively, we will present and analyze recipes from several datasets to showcase how we can use GraphChef

to understand GNN decision-making.

We run our experiments on explanation benchmarks: Infection and Negative Evidence dataset, which we will introduce in Chapter 6, BA-Shapes, Tree-Cycle, and Tree-Grid from Ying et al. [2019] and the BA-2Motifs from Luo et al. [2020]. We perform further experiments on real-world datasets: BBBP from molecule net [Wu et al., 2017b] and MUTAG, Mutagenicity, PROTEINS, REDDIT-BINARY, IMDB-BINARY, and COLLAB from TU-Dataset[Morris et al., 2020].

We train five-layer GNNs on each dataset with 10−fold cross-validation. We use a GIN as a baseline and a GNN using our dish layers that we convert to GraphChef trees. GIN layers have an embedding size of 16. We train the dish layers with ten available spaces. We can identify the number of needed states and layers upon inspecting the recipes. We show these in Table 5.6. We train the GNNs for 1500 epochs, allowing an early stopping with a patience of 100. We use the temperature parameter τ inside the Gumbel-Softmax blocks to increasingly converge towards sampling from one-hot distributions. Additionally, we create two decision tree baselines. The first version has access to only the node features; the second version has access to node degrees. These decision trees and the distilled in GraphChef are limited to 100 nodes at most.

Layers	States	
Infection	5	6
Negative	1	3
BA-Shapes	5	5
Tree-Cycles	5	5
Tree-Grid	5	5
BA-2Motifs	4	6

Layers	States	
MUTAG	4	6
Mutagenicity	3	8
BBBP	3	5
PROTEINS	3	5
IMDB-B	3	5
REDDIT-B	2	5
COLLAB	3	8

Table 5.6: Used number of layers and number of states GraphChef uses per dataset. Found through recipe inspection..

Quantitative Results

First, we investigate the expressive power of dish GNN layers and GraphChef trees. Since the dish layers are less expressive than GIN layers, we expect small losses in classification accuracy. On the other hand, these layers allow for better reasoning in graphs compared to vanilla decision trees. Therefore, we expect dish layers and GraphChef to perform better. Table 5.7 shows

the classification accuracy for all methods.

Compared to GIN, neither dish GNN nor GraphChef loses much, if any, classification accuracy on the datasets. This supports that the loss of expressiveness is negligible. The similar performance on the synthetic benchmarks shows that GraphChef can reason over larger graph structures. The weak performance of both vanilla decision tree variants further supports the observation that we require sophisticated graph reasoning for these datasets. All methods perform close to each other on real-world datasets, which suggests most of the reasoning is over the individual node features.

					GraphChef	
Dataset	DT	DT+degrees	GIN	dish GNN	No pruning	Lossless pruning
Infection	0.43±0.00	0.43±0.00	0.98±0.04	1.00±0.00	1.00±0.00	1.00±0.00
Negative	0.51±0.00	0.50±0.00	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00
BA-Shapes	0.43±0.00	0.86±0.02	0.97±0.02	1.00±0.01	0.99±0.01	0.99±0.01
Tree-Cycles	0.59±0.00	0.84±0.04	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00
Tree-Grid	0.58±0.00	0.76±0.03	1.00±0.01	0.99±0.01	0.99±0.01	0.99±0.01
BA-2Motifs	0.50±0.00	0.82±0.03	1.00±0.00	1.00±0.00	1.00±0.00	1.00±0.00
MUTAG	0.83±0.09	0.85±0.07	0.88±0.05	0.88±0.06	0.88±0.06	0.85±0.08
Mutagenicity	0.71±0.02	0.72±0.02	0.81±0.02	0.79±0.02	0.75±0.02	0.74±0.02
BBBP	0.83±0.03	0.83±0.02	0.81±0.04	0.83±0.03	0.82±0.03	0.83±0.03
PROTEINS	0.74±0.02	0.71±0.04	0.70±0.03	0.71±0.02	0.71±0.04	0.71±0.04
IMDB-B	0.57±0.04	0.71±0.04	0.69±0.04	0.70±0.05	0.69±0.03	0.69±0.04
REDDIT-B	0.75±0.03	0.80±0.03	0.87±0.10	0.90±0.03	0.88±0.03	0.87±0.04
COLLAB	0.59±0.01	0.70±0.02	0.72±0.01	0.70±0.02	0.69±0.02	0.69±0.02

Table 5.7: Classification accuracy of simple decision trees without graph reasoning, GIN for comparison, and three GraphChef variants. GraphChef's accuracy is competitive with.

Next, let us investigate the effectiveness of tree pruning on these datasets. We measure the sum of nodes overall decision trees in all layers and compare the (i) unpruned decision trees, (ii) REP-pruned decision trees with no loss on the training set, (iii) REP-pruned trees with no loss on the validation set, (iv) pruned decision trees with no loss on our criterion, and (v) trees with lossy pruning. For lossy pruning, we experimented with 10% steps of nodes to prune and choose what we considered the best tradeoff between size and accuracy. Table 5.8
As expected, many training set artifacts cannot be eliminated with pruning on the training dataset, whereas other methods produce smaller trees. We can also observe, especially on real-world datasets, that pruning on the validation dataset alone tends to over prune and consistently underperform unpruned trees and other methods. Our proposed pruning criterion combines the advantages of both ideas: We obtain decision trees that are almost as

	No pruning		REP Training		REP Validation		REP Ours		REP Lossy	
Dataset	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
Infection	1.00±0.00	205±56	1.00±0.00	26±2	1.00±0.00	25±2	1.00±0.00	26±2	0.98±0.01	17±2
Negative	1.00±0.00	18±14	1.00±0.00	5±0	1.00±0.00	5±0	1.00±0.00	5±0	1.00±0.00	4±0
BA-Shapes	0.99±0.01	30±10	0.99±0.01	21±5	0.97±0.03	15±4	0.99±0.01	21±5	0.98±0.04	17±4
Tree-Cycles	1.00±0.00	19±5	1.00±0.00	11±3	0.99±0.02	9±2	1.00±0.00	11±3	0.99±0.01	9±3
Tree-Grid	0.99±0.01	30±13	0.99±0.01	17±8	0.99±0.01	13±4	0.99±0.01	15±8	0.99±0.01	15±8
BA-2Motifs	1.00±0.00	141±43	1.00±0.00	12±3	1.00±0.01	11±3	1.00±0.00	13±4	1.00±0.00	13±4
MUTAG	0.88±0.06	59±27	0.86±0.08	19±17	0.83±0.07	7±6	0.85±0.08	18±16	0.85±0.08	18±16
Mutagenicity	0.75±0.02	375±13	0.76±0.02	154±19	0.73±0.01	56±16	0.74±0.02	91±36	0.73±0.02	50±19
BBBP	0.82±0.03	366±53	0.84±0.02	88±52	0.79±0.04	8±10	0.83±0.03	46±27	0.82±0.03	31±18
PROTEINS	0.71±0.04	206±90	0.72±0.03	12±13	0.70±0.04	8±6	0.71±0.04	9±6	0.71±0.04	9±6
IMDB-B	0.69±0.03	218±32	0.69±0.04	20±9	0.66±0.06	16±6	0.69±0.04	29±9	0.69±0.04	29±9
REDDIT-B	0.88±0.03	248±28	0.88±0.02	53±14	0.85±0.04	28±8	0.87±0.04	49±21	0.87±0.04	38±15
COLLAB	0.69±0.02	301±1	0.70±0.02	36±15	0.67±0.03	22±12	0.69±0.02	30±18	0.68±0.02	21±12

Table 5.8: GraphChef decision tree sizes and testing accuracy for different tree pruning strategies and no pruning as comparison. Using the training set does not prune enough training artifacts, using the validation set overprunes at cost of accuracy. Our combined objective keeps the best of those two worlds.

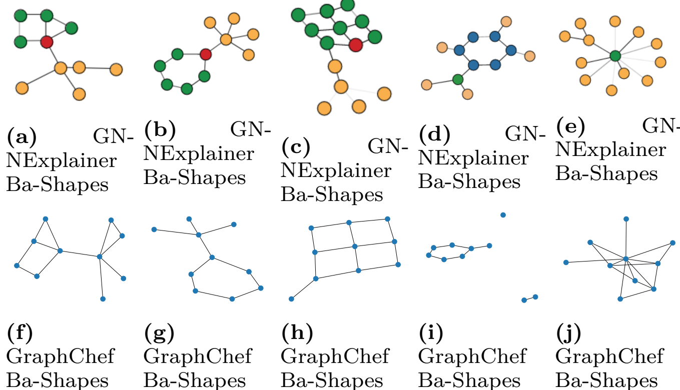
small as validation pruning but without losses in accuracy. Accepting small losses in accuracy, we can reduce the decision tree size even further.

Method	Infection	Saturation	BA-Shapes	Tree-Cycles	Tree-Grid
Gradient	1.00±0.00	1.00±0.00	0.882	0.905	0.667
GNNExplainer	0.32±0.09	0.32±0.05	0.925	0.948	0.875
PGMExplainer	0.38±0.06	0.01±0.01	0.965	0.968	0.892
GraphChef	0.95±0.02	1.00±0.00	0.94±0.02	0.84±0.02	0.927±0.01

Table 5.9: Explanation accuracy measured as the overlap of nodes we score most important to the dataset ground truth. GraphChef produces competitive explanations compared to existing state-of-the-art methods.

Last, let us compare the explanation scores that we can derive from the decision trees to other explanation methods: Gradient-based explanation [Pope et al., 2019], GNNExplainer [Ying et al., 2019], and PGMExplainer [Vu and Thai, 2020]. We evaluate on datasets with an existing explanation ground truth. The explanation accuracy is the relative overlap of the $k-$sized ground truth with the k most important nodes. Table 5.9 shows that the explanation accuracy of GraphChef is comparable to that of other explanation methods (Figure 5.10 shows some examples). Except Tree-Cycles, GraphChef performs at least as well or better than the baselines. We will analyze the recipe for Tree-Cycles to understand the lower score for this dataset. In Chapter 6, we will discuss a principled flaw in the dataset that

surfaces here. Similar flaws are also at work in BA-Shapes and Tree-Grid but do not surface as much.



(a) GNNExplainer Ba-Shapes **(b)** GNNExplainer Ba-Shapes **(c)** GNNExplainer Ba-Shapes **(d)** GNNExplainer Ba-Shapes **(e)** GNNExplainer Ba-Shapes

(f) GraphChef Ba-Shapes **(g)** GraphChef Ba-Shapes **(h)** GraphChef Ba-Shapes **(i)** GraphChef Ba-Shapes **(j)** GraphChef Ba-Shapes

Figure 5.10: Example subgraphs of the 10 most important nodes for GNNExplainer and GraphChef for several datasets.

Qualitative Experiments

Tree-Cycles We now present the new quality of explanation that GraphChef offers: understanding the GNN decision-making process. We first analyze the recipe for the Tree-Cycles dataset to understand the below-baseline performance in Table 5.9. Figure 5.11 shows the decision tree recipe for every layer for this dataset and Table 5.12 an analysis of the recipe. To understand the decision-making, we have to understand what every state across the layers means. The idea is to analyze the states from encoder to decoder like dynamic programming. First, we understand what the encoder states mean. Next, we analyze the decision-making of the first layer, which uses the encoder states. We can then make sense of the first layer by looking up the semantics of encoder states. Then, we proceeded to the next layer with the same approach until we analyzed all states up to the decoder. Per state, we mention the mechanical rule of the decision tree as well as a human understandable explanation.

Notably, the output layer (Figure 5.11d) ignores several message-passing layers. The first message-passing layer already identifies most cycles nodes

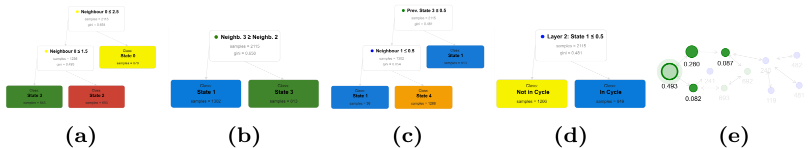


Figure 5.11: Recipe for the Tree-Cycles dataset. Table 5.12 provides and analysis for the state of every layer.

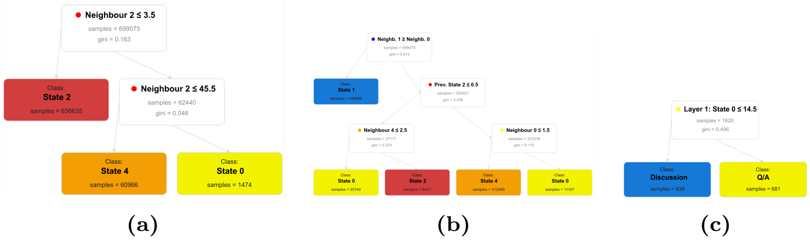
Layer	State	Decision Rule	Interpretation
Encoder	0	All nodes	No node features available for differentiation.
Layer 0	0	Three or more state 0 neighbors	Degree 3 or higher nodes (inner nodes in the tree, cycle node connecting the cycle to the tree).
Layer 0	2	Two neighbors in state 0	Degree 2 nodes (root node and cycles nodes).
Layer 0	3	One or zero neighbors in state 0	Degree 1 nodes (leaves in the connected graph).
Layer 1	1	At least as many state 3 as state 2 neighbors	As least as many leaves as cycle neighbors (true for inner nodes as well having zero of both).
Layer 1	3	More state 2 than state 3 neighbors	Most cycle nodes, nodes connected to degree 2 nodes.
Layer 2	4	Not previous state 3 and at least one state 1 neighbor	Not already a cycle node and connected to a non-cycle node.
Layer 2	1	1) Previous state 3 or 2) no state 1 neighbors	1) already a cycle node or 2) only connected to cycle nodes.
Decoder	Cycle	In layer 2 state 1	See previous state.
Decoder	No Cycle	otherwise	otherwise.

Table 5.12: Analysis of the recipe for Tree-Cycles in Figure 5.11. Generally, cycle nodes are nodes connected to only or almost only degree two nodes. Notably, nodes identify as cycle nodes before seeing the whole motif.

as degree two nodes. Since the base graph is a balanced binary tree, all base graph nodes except the root have a degree different than 2. The next layer almost finished the cycle detection as nodes connected to degree 2 nodes (Layer 1 state 3). Layer 2 finds the few remaining cycle nodes. Figure 5.11e confirms that the explanation only considers nodes two hops away. Since

the recipe does not consider the while motif, it is a Faithful explanation to score the unconsidered parts low.

Reddit-Binary Let us do another analysis for the Reddit Binary dataset. From Ying et al. [2019] and Chapter 4, we expect the Q&A graphs to



(a) (b) (c)

Figure 5.13: Recipe for the Reddit-Binary dataset. Table 5.14 provides and analysis for the state of every layer.

exhibit a more star-like structure. Figure 5.13 and Table 5.14 The first layer in Figure 5.13a splits the nodes into three different groups based on their degrees: inactive users with at most three neighbors, active users with at most 44 neighbors, and central users. The decoder uses only state 0 from the next layer. There are two possibilities to reach this state: (i) inactive users writing with at least two central users, (ii) non-inactive users that write with central users and at most two active users. The conditions limit the interaction of active and central nodes with non-inactive nodes. A graph has few central nodes, and the interaction with active nodes is limited to at most two. This dataset is a nice example of how the recipes allow us new possibilities of explanation because it allows us to quantify what a GNN considers centers in the star (45 nodes) and to what extent deviations from the star structure are acceptable (communication to 2 active nodes).

Limitations of GraphChef

We identified a limitation when applying GraphChef to datasets with many input features on the nodes, for example, citation datasets such as Cora, CiteSeer, Pubmed [Sen et al., 2008], or OGB-Arxiv [Hu et al., 2020]. The dish layers still perform somewhat comparably to GIN, but converting to decision trees leads to clear drops, as Table 5.15 shows. Larger feature spaces are harder to condense into one categorical space. However, the larger effect is that having 500 features (as PubMed) would require 500 decision nodes to

Layer	State	Decision Rule	Interpretation
Encoder	2	All nodes	No differentiation due to no features.
Layer 0	2	Nodes with at most 3 neighbors	Inactive users
Layer 0	4	Between 4 and 45 neighbors	Active users
Layer 0	0	More than 45 neighbors	Central users
Layer 1	0	1) State 0 nodes with more than one state 0 neighbor or 2) Not state 2 nodes that have at most two state 4 neighbors	1) Inactive users writing with at least 2 central users or 2) Active or central users that write with at most 2 active users.
Layer 1	1	No neighbor in state 0	Users that do no write with a central user.
Layer 1	2	Not state 2 nodes with at least 3 state 4 neighbors	Active or central users that write with at least 3 active users.
Layer 1	4	Nodes in state 2 with exactly one state 0 neighbor	Inactive users write with one central user.
Decoder	Q/A	At least 15 nodes in Layer 1 state 0	See interpretation of Layer 1 state 0.
Decoder	Discussion	Otherwise	The GraphChef model looks for evidence of a Q/A graph. Discussions are "not Q/A" graphs.

Table 5.14: Analysis of the recipe for Tree-Cycles in Figure 5.13. The recipe tries to find evidence for Q&A graphs and predicts Discussion otherwise. For Q&A graphs, there must be a limited number of interactions that involve non-low-degree nodes. Low-degree nodes must communicate with one high-degree node, which creates a star-like graph structure. The recipe also provides thresholds what a low-degree node is.

consider each feature once. Since we only allow 100 nodes, we force the encoder layer to disregard 80% of the input space. We confirmed that the drop happens in the encoder layer: When we do not convert the encoder layer into trees, the loss of accuracy becomes much smaller. We consider three possibilities to tackle these datasets in future work: (i) Dimensionality-reduction strategies such as PCA or clustering, (ii) Special neural architectures that allow interpretability, such as the works of Wu et al. [2017a] or Schaaf et al.

[2019], and (iii) Additional features to expose to GraphChef to express the input space more efficiently, similar to how we expose the delta features to compare neighbor counts.

Dataset	Features	GIN	Dish GNN	GraphChef
CORA	1433	0.87±0.02	0.82±0.03	0.69±0.04
CiteSeer	3703	0.77±0.01	0.70±0.03	0.61±0.04
PubMed	500	0.88±0.01	0.87±0.01	0.85±0.01
OBGN-Arxiv	128	0.68±0.02*	0.68±0.01	0.28±0.11

Table 5.15: Accuracy of GraphChef on citation networks. When converting to trees, accuracy drops noticeably. *Since the dataset has 40 classes, we use a state-size of 50 for GraphChef variants and 128 wide embeddings for GIN.

6

When Comparing Against Ground Truth is Wrong

We finish this chapter by wrapping up several observations during the evaluation of the previous two chapters. We made the first observation in Chapter 4 on the CYCLIQ dataset. Is the reason for better performance in clique explanation versus cycle explanation really a problem of CoGE to explain cycles? We also saw a surprising GraphChef recipe in Chapter 5 where the recipe for Tree-Cycles was content only to explore the cycle two layers deep. Does GraphChef also have a problem with Cycle detection? We will show in this chapter that cycles are not the bane of explanation methods. Rather, deficiencies in the evaluation setup can easily make post-hoc explanation methods seem worse than they are. We characterize five pitfalls that can cause such deficiencies. Next, we propose three new explanation benchmarks designed to avoid these pitfalls and finish with experiments on these datasets. The pipeline in Figure 6.1 shows the most common setup for evaluation explanation method, in which these pitfalls can occur.

In the first step, we train a GNN on a problem. Then, we employ a post-hoc explanation method to generate explanations for the GNN. Because of the nature of the problem, we know which nodes are the correct explana-

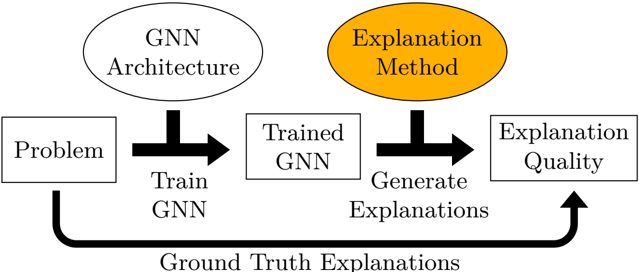


Figure 6.1: Common setup to evaluate explanation accuracy for an explanation method (yellow). First we train a GNN and then apply the explanation method. We measure explanation quality as accuracy to the problem ground truth.

tion. For example, the correct explanations in the CYCLIQ dataset are the clique or cycle nodes. As another example, the correct explanation in the BA-Shapes dataset is all the nodes in the house. Because we have this explanation ground truth, we can define the quality of explanation methods as the overlap between its explanation and the explanation ground truth.

6.1 Pitfalls for Explanation Evaluation

In this chapter, we will focus our definitions on explaining edges. However, the following arguments can be equally defined on nodes, or we can derive edge explanations from nodes. Two sets are commonly considered important for evaluation explanations: the set of ground truth explanation edges $\mathcal{T}$ versus the set of explanation edges produced by our method to test $\mathcal{E}$. Some explanation types, such as subgraphs, directly produce a set of edges while others produce importance scores, which we use to assemble the set $\mathcal{E}$. With ground truth labels, we use accuracy to measure explanation quality instead of fidelity. In this notation, accuracy is the overlap of these two sets relative to the size of $\mathcal{T}$:

$$Acc = \frac{\mathcal{T} \cap \mathcal{E}}{|\mathcal{T}|} \tag{6.1}$$

However, we argue that this approach fails to acknowledge another crucial set of edges $\mathcal{M}$, which are the edges that the GNN actually used. The above definition makes sense if we assume that the GNN used exactly the ground truth edges and $\mathcal{T} = \mathcal{M}$. However, what if the model partially uses some

other edges?

Because we want our explanation method to explain the model behavior, we would also want $\mathcal{E}$ to reflect those edges (Faithfulness). However, Equation 6.1 would show low accuracy scores even for perfect explanation methods where $\mathcal{M} = \mathcal{E}$. Therefore, it is imperative to take measures to ensure $\mathcal{T} = \mathcal{M}$ as much as possible. In the following, we discuss five pitfalls during the dataset or model design that can cause a divergence and should, therefore, be avoided.

Bias Terms

The first pitfall comes from failing to acknowledge that GNNs can use bias terms in their decision-making. When using a bias term, the GNN does not use any edges. Therefore, $\mathcal{M} = \emptyset$, and there is no way to reasonably expect an explanation method to find the correct explanation $\mathcal{T}$. We observed this pitfall in a previous chapter in the CYCLIQ dataset. Table 4.5 shows that the accuracy for cliques is much higher than for cycles. We can explain this difference by bias terms. We investigated the GNNs on this dataset and found they consistently use bias terms to predict cycles and only switch to clique predictions upon seeing enough data in favor of it.

How can we solve this pitfall? Unfortunately, not allowing bias terms in the GNN is not enough. GNNs can easily create their own bias terms. Suppose one input feature is a one-hot encoded categorical value. The first layer can sum up all these values, thereby creating a constant one that subsequent layers can use again as a bias term. Therefore, we have to solve this pitfall in the dataset design. When designing an explanation benchmark, we must include a target class that input features cannot easily solve. One common approach we will also use later is including a class for “all other cases”.

Redundant Evidence

Another pitfall arises when the ground truth evidence contains two or more sets of edges that each are sufficient for the GNN to make its prediction. Suppose $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$. For example, the CYCLIQ dataset from the previous chapter but a graph has two cliques attached instead of one. It may be use only one of the clique ($\mathcal{M} = \mathcal{T}_1$ or $\mathcal{M} = \mathcal{T}_2$) or both of the cliques ($\mathcal{M} = \mathcal{T}_1 \cup \mathcal{T}_2$). We should only expect the explanation method to find all

edges in the latter case.

A similar effect, but more subtly, can happen when the ground truth is larger than necessary. This happens in the Tree-Cycles dataset from Ying et al. [2019]: In this node classification dataset, we attach cycles of length six to random parts in a balanced binary tree. The GNN learns to separate the nodes in the cycle from the other nodes. Intuitively, the whole cycle should be the ground truth $\mathcal{T}$. However, we can identify every cycle node as nodes with two degree-two neighbors. We train a $2-$layer GNN on this dataset to validate this idea, achieving 95% accuracy. We can largely solve the dataset without observing the whole cycle. Therefore, it may be that not even a deeper GNN considers the full cycle before making a prediction, which leads to "incomplete" cycles in the explanation. We can see one such occurrence in the GraphChef recipe in Figure 5.11. The figure also shows that the explanation, consistent with the recipe, only assigns importance to nodes at most two hops away.

How can we solve this pitfall? We can address this pitfall in several ways: When creating the benchmark dataset, we must ensure that the ground truth only contains the truly necessary edges to find the correct prediction. This set may be counterintuitive, so we should double-check that smaller ground truth sets are not informative enough. We also propose having the set be unique, for example, not having two cliques in the CYCLIQ dataset. Instead, we account for redundant evidence in our evaluation and accept either clique as an explanation.

Trivial Evidence

Adebayo et al. [2018] present an "explanation" of a convolutional neural network for a bird. Except the explanation is not an actual explanation but an edge detector. The model-agnostic edge detector should not be considered a good explanation method since it disregards the model decision-making process– -even though the output looks reasonable. The same can happen in graphs: For example, a model agnostic method that returns close neighbors would perform well in the Tree-Grid [Ying et al., 2019] node classification dataset, where we attach grids to balanced binary trees. For most nodes in the grid, the closest neighbors are exactly the other grid nodes. Homophilic community detection datasets such as Cora [Sen et al., 2008] are another example where explaining a node through its neighbors may yield perceptively good results while disregarding the GNN decision-making process altogether.

How can we solve this pitfall? During benchmark creation, we need to ensure that simple heuristics, such as the nodes' closest neighbors, do not cover the entire explanation $\mathcal{T}$. We vet our later-proposed benchmarks by ensuring that nearest-neighbor, random-walk, and entirely random methods perform poorly.

Weak GNN Architecture

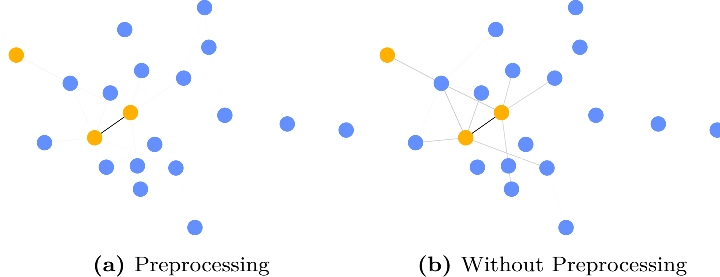
While the three previous pitfalls concerned the dataset used in the evaluation, this and the next pitfall concern the GNN we want to explain. First, let us consider the implications if the GNN performance is significantly below 100%, which means the GNN makes many mistakes. If the GNN were to use all the ground truth edges $\mathcal{M} = \mathcal{E}$, it would have access to all the necessary information. Therefore, it stands to reason that the GNN could solve the dataset (close to) perfectly. On the other hand, a GNN that makes mistakes is a strong indicator that $\mathcal{M} \neq \mathcal{T}$, which in turn means that the GNN explanation method will also contain wrong edges and/or miss correct ones.

How can we solve this pitfall? For evaluating explanation methods, we should limit ourselves to benchmarks where we can train the GNN we want to explain to near-perfect accuracy. Note that we can still use explanation methods on real-world datasets where we commonly do not reach such accuracy scores. However, we should strive for a controlled environment to evaluate explanation methods. Alternatively, we should compare the different explanation methods on the very same GNN so the GNN mistakes are consistent for evaluating all explanation methods.

Misaligned GNN Architecture

The last pitfall is the most subtle and the most difficult to remedy. Even when we have a (close-to) perfect scoring GNN, and none of the other pitfalls are in place, we can find a mismatch between $\mathcal{M}$ and $\mathcal{T}$ because the model architecture does not fit the problem at hand.

Consider the simple graph classification problem where graphs have yellow and blue nodes, and we classify whether a graph contains two connected yellow nodes. However, we encode node color deliberatively in a suboptimal way. Instead of representing the categorical color as a one-hot vector, we encode color in a scalar where 1 maps to blue and 2 maps to yellow. We now train a GNN in two versions: both consist of three message-passing layers, and only one has two preprocessing layers. These two layers allow every



(a) Preprocessing **(b)** Without Preprocessing

Figure 6.2: Example of how a misaligned model architecture can cause bad explanations. The GNN on the right has no preprocessing layers and has to use message-passing layers to transform the node features (color) into a more usable representation. Neighbors of yellow nodes receive such messages and become important when they should not be.

node to transform its features into a more useful representation before starting the message-passing layers. There is no difference in performance; both GNNs achieve quasi-perfect accuracy. However, we observe clear differences in the explanations. Here, we use a graph adopted version of integrated gradients [Sundararajan et al., 2017] in Figure 6.2.

Figure 6.2a explains the GNN with preprocessing layers, which is the expected edge between the two yellow nodes. However, the explanation for the architecture without preprocessing layers includes many more nodes, as Figure 6.2b shows. This GNN needs to transform the scalar color value into a more useful representation using the message-passing layers. Because of those layers, several nodes that do not carry useful information participate in this transformation.

How can we solve this pitfall? Unfortunately, we did not find a principled way to address this pitfall. Usually, it requires intricate knowledge of the benchmark to know if the GNN architecture is aligned with the reasoning required by the dataset. You et al. [2020] laid out a design space with twelve dimensions for GNN architecture design that each can help or hurt aligning the GNN architecture. We will discuss for each following benchmark individually how we think we found a well-aligned GNN architecture. Generally, preprocessing layers allow node-local computation to happen locally in the node. Additionally, we found skip connections helpful: In case a

GNN can solve some predictions without using all layers, information does not need to be passed around additional layers.

6.2 Three New GNN Explanation Benchmarks

We propose three new benchmarks tailored to circumvent the previously mentioned pitfalls. The benchmarks are designed to cover the most common GNN reasoning tasks: a simple pattern detection benchmark and one benchmark requiring homophilic reasoning. The third benchmark investigates the ability of explanation methods to handle negative evidence and abundant information.

Infection Benchmark

The infection is a simple pattern detection benchmark. Pattern detection is a common task, for example, in chemical applications where we want to identify subgraphs known for some chemical function. Two example benchmarks are MUTAG and Mutagenicty [Morris et al., 2020] dataset. Our proposed dataset is close to the infection benchmark proposed by Baldassarre and Azizpour [2019] with some modifications.

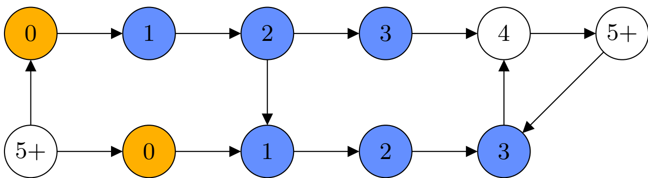


Figure 6.3: One example graph from the infection dataset. Yellow nodes are infected nodes, we classify the nearest distance of every node to an infected node. We only explain on nodes the GNN can find (no nodes with label 5+) and nodes with a unique shortest path (not on the node with distance 4.)

We create a random connected Erdös-Rényi graph where nodes are either healthy or infected. The task is node classification, where every node classifies how far they are away from their nearest infected node, up to a maximum distance. Therefore, the possible labels reach from 0 (the node itself is infected) to 5+ for nodes five or more hops away from an infected node. We also give the label 5 to unreachable nodes. This task differs from the dataset by Baldassarre and Azizpour [2019] in three ways: (i) We allow for more than one step of infection propagation, so explanations become graphs; (ii)

We do not incorporate immune edges; (iii) We exclude nodes from consideration whose shortest path to an infected node is not unique. We show an example graph with labels in Figure 6.3. Yellow nodes are infected, and white nodes are not considered for computing the explanation accuracy either because there is no unique path (as for the white node at distance 4) or there is no ground truth (for the 5+ nodes). Every blue node has a unique ground truth explanation, which is the unique shortest path of length 4 or less—if it exists. This dataset circumvents the pitfalls as follows:

Bias Terms The label 5+ is overloaded: It contains nodes in distance 5, larger than distance 5, and even unreachable nodes. Especially for the last group, there is no information in the graph that the GNN can leverage. These nodes have their class because of the absence of any information. To solve all classes correctly, the GNN must use its bias for this class. Therefore, the evaluation of relevant classes is bias-free.

Redundant Evidence We remove nodes with non-unique shortest paths from consideration for evaluation explanation accuracy, particularly to avoid this pitfall. Every considered node (the blue nodes in Figure 6.3) has a unique shortest path to an infected node, and every node along this path is necessary for a correct explanation.

Trivial Evidence The explanations are paths, therefore, not restricted to a node's immediate neighborhood. We further validate that path-based baselines, such as random walks, also perform poorly on this dataset.

Weak GNN architecture The GNNs used in the test solve this dataset with 100% accuracy.

Misaligned GNN architecture The longest paths among all the classes have length 4, therefore we employ a 4−layer GNN. Additionally, we use skip connections so the GNN does not have to preserve the information for distance 1 nodes for 3 more steps. This preservation would not be guaranteed to be on the nodes themselves.

We create 6 training and 4 test graphs, each having 1000 nodes and roughly 4000 edges on average. Since we are employing 4 GNN layers, there are 6 possible labels: distances from 0 to 4 plus one label for unreachable nodes. Since we only consider nodes with a unique shortest path, we have an average of 491 nodes for testing per graph.

Community Benchmark

Another very common form of graphs is social networks. Typically, these networks exhibit homophily, which makes it a frequent reasoning requirement for GNNs. This benchmark evaluates whether an explanation method can explain such homophilic reasoning.
We base our graphs on the directed stochastic block model (SBM) [Abbe, 2017]. We build a graph with n nodes that we split into k equally sized communities to which we assign different colors. The goal is for every node to predict their community/color. The probability for any two nodes in the same community is p, and the probability for any other two nodes is q. We choose (i) $p > q$ and (ii) $p < (k-1) * q$. Therefore, every node has more neighbors from other communities than from its community in expectation. On the other hand, the community with the most neighbors is a node's community in expectation. To prevent a node from just predicting the majority class among its direct neighbors, we randomly re-assign a color to a fraction f of nodes (without changing their prediction label). Several graph generation processes involve randomness, so we base our explanation evaluation on relative importance [Yang and Kim, 2019].

We generate a base graph as described above. Figure 6.4 shows one example graph. When explaining node v, we will rewire this base graph within v's receptive field in two different ways. First, we increase the community structure of v's community by rewiring edges that start outside of v's community but end in v's community. After rewiring, the edges also start in v's community, strengthening the community structure. If the prediction confidence for v now increases, then we expect the explanation score for these edges to increase. Second, we rewire edges that start and end outside of v's community to end them in v's community, overall decreasing community structure. If this leads to decreased prediction confidence, we require the explanation score to decrease. We define the overall explanation accuracy as the ratio where the prediction confidence of the GNN leads to an equivalent change in explanation score by the explanation method. This avoids the above pitfalls as follows:

Bias Terms There is no special class accounting for the bias term, which is why we contrast the GNN prediction before and after rewiring and look for differences in confidence. Both predictions use the same biases, so a prediction change cannot stem from the bias term.

Redundant Evidence There must be a tradeoff between rewiring too few edges (with no impact because there is redundant evidence available)

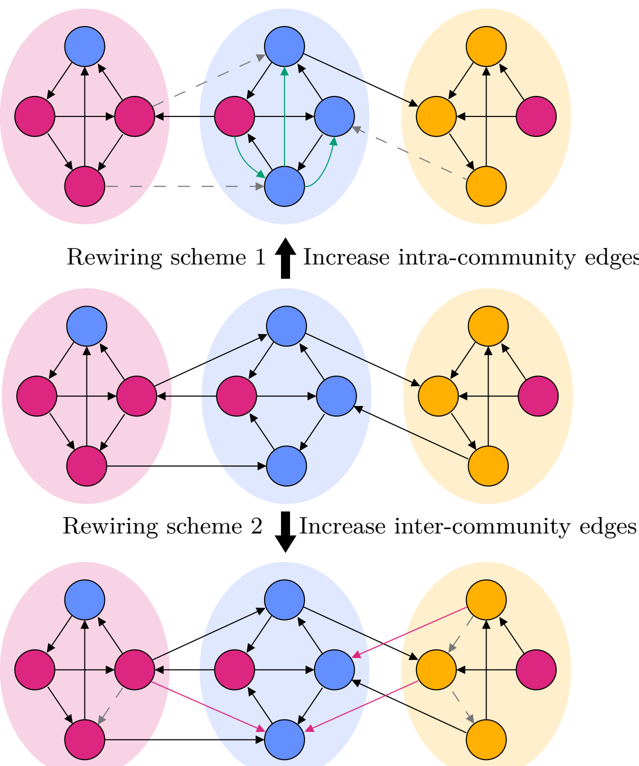


Figure 6.4: An example graph from the community dataset. Nodes need to classify the community they belong to (background color) while potentially starting with a different color. This benchmark uses relative importance on two rewirings: The first rewiring (top) increases intro-community edges, and the second rewiring(bottom) increases inter-community edges. Explanations for the rewired edges should copy model confidence changes.

versus too many rewirings that create graphs that no longer represent the training distribution.

Trivial Evidence Nearby nodes do not make for an explanation since the majority of immediate neighbors come from a different class. High-centrality nodes help neither since such nodes tend to connect to all communities.

Weak GNN architecture While the GNN does not reach 100% on the prediction task, we only evaluate the explanation accuracy where the model predicts correctly and detects the change in community structure.

Misaligned GNN architecture We found a 4− layer GNN to produce the best results and again provide skip connections to allow each layer to directly provide a signal to the final output.

We create 45 training and 5 test graphs, each with 1000 nodes and, on average, 11250 edges. We found the best model accuracy for the parameters $p = 0.05, q = 0.007$, and $f = 0.5$. The explanations are not easy, yet the GNN still reaches an accuracy of 0.81.

Negative Evidence and Saturation

This last benchmark tests two properties of graph explanation methods: Handling negative evidence and robustness to surplus explanation. Negative evidence differs from unimportant edges in that they actively discourage the GNN from certain predictions. Understanding negative evidence allows us to understand the model reasoning more deeply. Further, we want models to be robust to surplus evidence. When faced with more evidence, some methods will actually become less confident in their explanation. Shrikumar et al. [2017] refer to this as the saturation effect. Resilience to saturation is important since we cannot guarantee that real-world datasets will avoid the second pitfall. We propose the following benchmark to test both properties.

We create a graph with white, blue, and red nodes. There are 10 blue and red nodes each. We connect every white node to other white nodes such that every white node has mostly white neighbors. Furthermore, we connect each white node to b blue and r red nodes, such that $b \neq r$. The target prediction for the GNN is the majority color. We do not learn to predict anything for red and blue nodes. Multiple white nodes exist for every combination (b, r). We only use nodes for testing where the difference between red and blue nodes is larger than 1, so occlusion never observes out-of-distribution

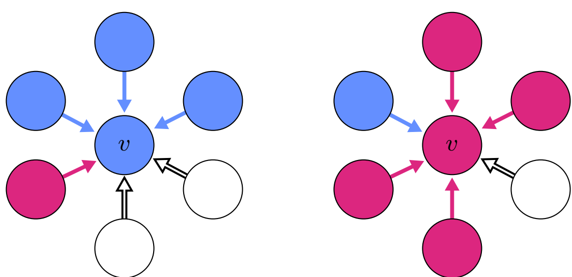


Figure 6.5: Two example graphs of the negative evidence and saturation benchmark: Node labels are the majority non-white color in their neighborhood. Node v in the left graphs has more blue neighbors, and more red neighbors in the right graph. We expect white edges (uninformative) to be attributed differently from minority edges (negative evidence). Evidence can saturate when the difference between majority and minority grows.

graphs. Figure 6.5 shows two example graphs. The left graph shows a white node with a blue majority, and the node in the right graph has a red majority.

Blue edges are negative evidence for a node with more red edges and vice versa. At the same time, white edges do not carry any information. Depending on the semantics of an explanation method, we could imagine a higher attribution for a white edge (a white edge is more "helpful" than a minority edge) or a lower attribution (a white edge is less "informative" than a negative edge). However, white edges should be attributed differently from minority edges. We verify that white edges differ from minority edges in their attribution. We run a two-sided Kolmogorov-Smirnov test with a significance level of 0.99. A method can detect negative evidence when it creates different importance distributions.

It is easy to cause saturation in this benchmark by having a large difference between the majority and minority color. No matter how large the difference, the prediction stays the same. Therefore, we expect that the explanation strength should not decrease when the difference between majority and minority increases. The benchmark addresses the five pitfalls as follows:

Bias Terms No class is suited to catch bias terms. However, the task symmetry plus the fact that both labels require considering both colors means that the model cannot reach 100% through using bias terms.

Redundant By design, this benchmark does contain redundant evidence. Therefore, we do not measure explanation accuracy as overlap. We measure exposure to saturation by measuring if the explanation strength decreases with more redundant evidence.

Trivial Evidence Most neighbors are white and unimportant for every white node. This eliminates simple "explanation" methods just as a majority vote of the neighbors or path-based approaches.

Weak GNN architecture GNNs reach 100% accuracy on this dataset.

Misaligned GNN architecture We can classify correctly using only 1 GNN layer without any skip connections or preprocessing. Colors are available as one-hot inputs.

We create 6 training and 4 test graphs, each with 1980 white nodes. We connect white nodes with probability 0.015 to other white nodes. In expectation, every white node has 30 white neighbors, and the graph has 103955 edges. We have 22 white nodes for every combination of blue and red neighbors. By restricting the difference between the majority and minority to be at least 2, each graph has 1232 nodes available for testing.

6.3 Experiments

Setup

We evaluate several explanation methods. To evaluate the third pitfall on trivial solutions, we also run several simple "explanation" methods:

Random A simple baseline where we assign a random number between 0 and 1 to every edge.

Nearest Neighbor A simple baseline where we assign importance inversely proportional to the distance from the target node.

PageRank A simple baseline where we assign importance equivalent to the node's personalized pagerank [Page et al., 1999] explain.

Node Gradients We measure node importance by computing gradients with respect to the node features. We derive the edge importance as the average value of its adjacent nodes' importance.

Edge Gradients To compute edge gradients, we allow every edge to have continuous weights and multiply passed messages with the edge weight. This allows us gradients with respect to these edge weights. The gradients form the edge importance.

Integrated Gradients Integrated gradients (abbreviated with IG) are proposed from Sundararajan et al. [2017]. We accumulate gradients for several inputs where we gradually interpolate from an all-zeros input to the actual input. We use this version for both nodes (Node IG) and edges (Edge IG).

GradCAM GradCAM [Selvaraju et al., 2017b] is a gradient-based method that weighs the gradients by the node activation. Instead of only using the last layer, which restricts explanations to one-hop neighborhoods, we base node importance on gradients and, additionally, activations from intermediate layers.

Occlusion Occlusion [Zeiler and Fergus, 2014] removes one node or edge at a time and defines the importance as the difference in the model confidence on the perturbed graph.

PGMExplainer PGMExplainer [Vu and Thai, 2020] also creates perturbed graph versions and employs probabilistic graphical models to capture dependencies and the prediction target. We use a single-instance version of PGMExplainer that we modify to predict the explanation for a graph of our choice.

GNNExplainer Similar to gradient methods GNNExplainer [Ying et al., 2019] relaxes the edges and features to continuous values. Instead of gradients, GNNExplainer defines node importance as the mutual information between these inputs and the prediction target. We also modify GNNExplainer to compute an explanation for a graph of our choice instead of the model-predicted class.

We run these methods on the three newly proposed benchmarks we created, as mentioned in the previous section. We use a GIN with access to the previous state and two sets of weights: one for the aggregated messages and one for the previous state. We use skip connections if outlined for the benchmark. The learning rate is 0.0003 for Infection and Community and 0.005 for the third benchmark. All results are computed only on eligible graphs in the test set.

Explanation Accuracy

Table 6.6 shows the explanation accuracy for all methods. Generally, edge-based methods work better than node-based methods. This is unsurprising since the here-proposed benchmarks define their ground truth as edges, and edges allow for finer-grained explanations. The first Infection column shows the overlap of the shortest path edges with the most important edges in the

Method	Infection	Community	Negative Evidence
Random	0.00 ± 0.00	0.50 ± 0.00	0.01 ± 0.00
NearestNeighbor	0.44 ± 0.01	0.57 ± 0.00	N/A
PageRank	0.05 ± 0.00	0.56 ± 0.00	N/A
Node Gradients	0.03 ± 0.03	0.40 ± 0.03	0.00 ± 0.00
Edge Gradients	0.74 ± 0.12	**0.92 ± 0.01**	**1.00 ± 0.00**
Node IG	0.53 ± 0.13	0.55 ± 0.02	**1.00 ± 0.00**
Edge IG	**1.00 ± 0.00**	0.71 ± 0.13	**1.00 ± 0.00**
GradCAM	0.50 ± 0.11	0.33 ± 0.11	0.00 ± 0.00
Occlusion	**1.00 ± 0.00**	0.89 ± 0.01	0.25 ± 0.16
PGMExplainer	0.38 ± 0.06	0.53 ± 0.01	0.01 ± 0.01
GNNExplainer	0.32 ± 0.09	0.53 ± 0.04	0.32 ± 0.05

Table 6.6: Results for baselines and explanation methods for the three proposed benchmark. First group of methods are baselines. Second group of methods are white-box explanation methods. Second group are black-box methods. Infection shows the overlap ratio with the shortest path to an infected node. Community shows the ratio of rewired edges where the explanation change is consistent with model confidence changes. Negative Evidence shows for how many nodes the distribution of white and minority explanations are different.

explanation. The strongest baseline is the nearest neighbor explanation, which finds at least one correct edge. Node methods, PGMExplainer, and GNNExplainer struggle to beat this baseline. On the other hand, Integrated edge gradients and Occlusion achieve perfect scores.

For the Community benchmark, the table contains the ratio of edges that pass the relative importance test. Many methods struggle to beat the random baseline of 0.5 by a large margin. Only the same methods that also worked best on the last benchmark do. This time, vanilla edge gradients worked better than the integrated variant and performed best.

For the third benchmark, we first evaluate the negative evidence. We perform one Kolgomorov-Smirnov test for each node with at least 2 minority edges to determine whether we can reject the null hypothesis that the distribution of minority and white edges are different. Table 6.6 contains the ratio of nodes for which this test is successful with at least 0.99 confidence. Edge gradients and both integrated gradient methods achieve perfect results. We show the results for the saturation effect in Figure 6.7. If a method is almost

parallel to the $x-$ axis, it does not suffer from saturation. Edge gradients and Occlusion suffer from saturation. When saturation is too strong, Occlusion fails.

Table 6.6 shows that white box explanation methods (those have access to the model weights) generally perform better than black-box explanation methods. This result is not surprising since access to the model weights allows, for example, to use negative weights to identity negative evidence. Black-box models can also detect negative evidence, but the task is harder. Current black-box models do not explicitly have the goal to detect such evidence, therefore models like GNNExplainer or PGMExplainer struggle with this dataset. The result of black-box models apart from Occlusion on the Infection dataset are surprisingly low. The construction of the dataset means that removing any edge on the path switches the label. This is why Occlusion easily performs perfectly. One hypothesis for the worse results of PGMExplainer and GNNExplainer is that they might build their explanations additively instead of removing like Occlusion. In such cases they need to consider several apparently useless edges to uninfected nodes, before finally connecting to an infected node. In particular for GNNExplainer, it might matter if we optimize upwards from an empty graph or downwards from the computational subgraph. We see a similar pattern for the Community benchmark where Occlusion performs better. When the actual community is unclear including the right edges is difficult since the majority of neighboring edges lead to wrong communities.

Explanation Runtimes

Finally, we compare the runtimes of the different methods. For gradients, the measured code is on Pytorch Geometric [Fey and Lenssen, 2019] and Captum [Kokhlikyan et al., 2020]. We use the Infection Benchmark and run it on a TITAN Xp GPU with 12GB memory. We measure the runtimes per explanation and show the results in Figure 6.8. Gradient-based methods are the most efficient. The Integrated Gradients compute 50 gradients. Therefore, they are slower than vanilla gradients. GNNExplainer optimizes a mutual-information objective over several iterations. Similarly, Occlusion evaluates different permutations of a graph. PGMExplainer is the slowest method that also evaluates permutations and evaluates them in graphical models.

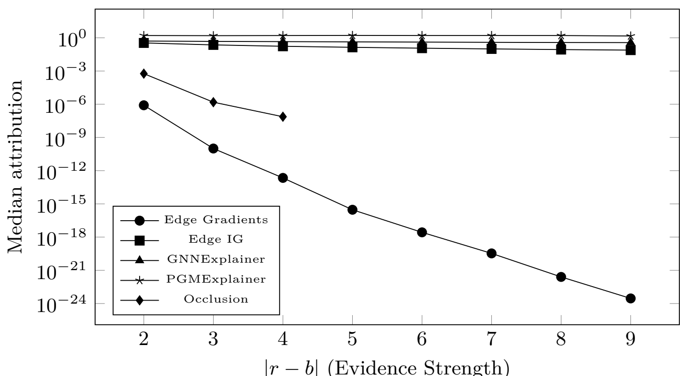


Figure 6.7: Saturation effect measured on the third benchmark. Evidence saturates along the $x-$axis with $|r-b|$. The $y-$ axis shows the median attribution of non-white edges in log-scale. Methods robust to saturation do not decrease in attribution strength and stay parallel to the $x-$axis.

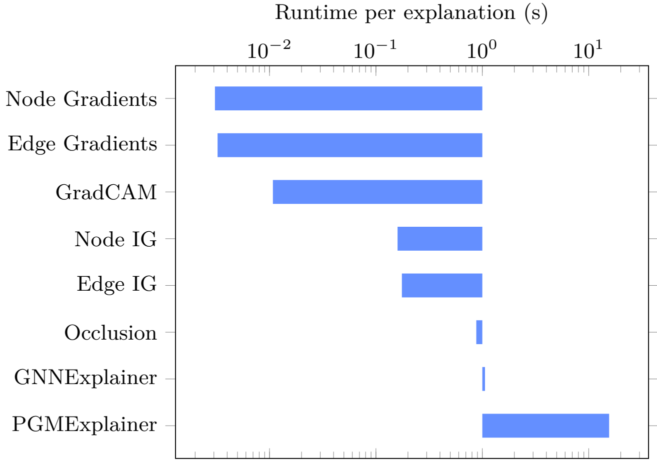


Figure 6.8: Time (in seconds) that explanation methods need on average to compute on explanation.

Part II

Improving Communication in Graph Neural Networks

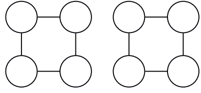
7

GwAC: GNNs with Asynchronous Communication

7.1 Limitations of GNNs

Let us first stay in the spirit of understanding: this time, the limits of GNNs: What are the theoretical limits of GNNs? Which problems can GNNs practically [not] learn to solve well?

Theoretical Limits One important observation is that GNNs operate similarly to nodes in the synchronous communication model in distributed computing [Loukas, 2020, Sato et al., 2019, Sato, 2020]. Both models operate in rounds. In each round, every node sends a message to every neighbor. There are two distributed models: LOCAL and CONGEST [Peleg, 2000, Wattenhofer, 2020]. In the LOCAL model, nodes can send an arbitrary amount of information; in CONGEST, only a limited amount. Practically, GNN layers are width-bound, which corresponds to the CONGEST model. While LOCAL and CONGEST can perform arbitrary computations on individual messages they receive from their neighbors, GNNs apply a learned neural function on an aggregated view of the messages. Does this limit what



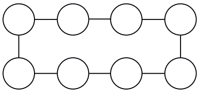


Figure 7.1: Two graphs that are not distinguishable by 1−WL or most GNNs. The left graph consists of two cycles of length 4. The right graph consists of one cycle of node 8. In every GNN message-passing step, there are 8 nodes; no node has any features and receives a message from exactly 2 featureless nodes. No node can gains information to differentiate between the two graphs.

GNNs can compute?

Loukas [2020] show that GNNs can compute any function on a graph with several conditions: (i) The neural functions in the GNN need to be sufficiently deep and wide to allow them to approximate any function following the universal approximation theorem [Royden and Fitzpatrick, 1988]. (ii) The GNN needs to have sufficient many layers such that the information can propagate through the entire graph. Therefore, number of layers needs to be at least as large as a graph's diameter. (iii) The nodes in a graph need to be distinguishable by unique identifiers.

What is the expressive power of GNNs such identifiers are not available? Xu et al. [2019c] show that their GIN architecture is as powerful as the 1−WL test, a heuristic for graph isomorphism [Weisfeiler and Leman, 1968]. 1−WL is the upper bound for expressiveness without any mechanism to distinguish nodes. Figure 7.1 from Garg et al. [2020] shows two simple graphs that a GNN cannot distinguish. One graph consists of two disconnected cycles of length 4 versus one cycle of length 8. Every node in both graphs has the same features (none) and exactly two neighbors. Therefore, every node always receives identical information, and no node learns information to distinguish the two graphs.

There are many proposed approaches how to add additional information to nodes. For example, there exist random features [Loukas, 2020, Sato et al., 2021], ports [Sato et al., 2019], subgraphs [Wijesinghe and Wang, 2021], or additional domain-specific information such as spatial data in chemistry [Gasteiger et al., 2020]. Other methods mirror higher-order versions of the WL test and use GNNs on higher-order graphs [Chen et al., 2019b, Maron et al., 2019, Morris et al., 2019]. Another line of approaches computes multiple perturbed graphs and combines the GNN prediction across those

permutations [Bevilacqua et al., 2022, Papp et al., 2021, Vignac et al., 2020].

Morris et al. [2023], Wang et al. [2023], Zhang et al. [2023] recently proposed alternatives to measure expressiveness, but we will measure expressiveness with the WL framework in the scope of this thesis.

Practical Limits Similar to CONGEST, we can run into scenarios where there is too much information we would have to pass over a single edge. The edge becomes a bottleneck and causes congestion in the overall communication Sarma et al. [2012]. We can observe the same effect in GNNs when the information is too much to learn with the given node and message dimensions [Alon and Yahav, 2021, Topping et al., 2022]. Practically, we can address congestion by introducing further edges into the graph [Brüel-Gabrielsson et al., 2022]. Such additional edges can act as shortcuts. They can free intermediate nodes from needing to pass information. Alternatively, we can reduce the information load in a layer by dropping some nodes or edges [Rong et al., 2020, Feng et al., 2020, Hasanzadeh et al., 2020].

Two more practical problems stem from the aggregation in GNNs. Usually, not all messages are important; non-important messages can bury important ones. Nodes average these messages with their state. Over time, the embeddings of all nodes converge to the same state [Li et al., 2018, 2019], which is known as the Oversmoothing problem [Oono and Suzuki, 2020]. The previously mentioned dropping methods also help with Oversmoothing. In particular, skip connections also help against oversmoothing since they help nodes preserve their information [Chen et al., 2020b, Xu et al., 2018]. Chen et al. [2020a], Zhao and Akoglu [2020], Zhou et al. [2020].

Second, a node in a GNN needs k layers to propagate information to a node that is k hops away. Since oversmoothing limits the number of GNN layers, pairs of nodes may be too far away from each other to exchange pertinent information. We call this limitation Underreaching [Barceló et al., 2020]. Adding or rewiring edges can help to reduce distances in the graph [Brüel-Gabrielsson et al., 2022]. In the extreme case, we can also connect all nodes to a global node to facilitate short distances [Gilmer et al., 2017, Wu et al., 2021]. Such nodes are especially prone to suffer from Oversquashing. Scarselli et al. [2008], Klicpera et al. [2019] propose partially breaking from the message-passing paradigm and propagating information through diffusion.

7.2 No Aggregation Asynchronous Communication

All these problems stem partially from GNNs aggregating the messages before processing them. What if we did not aggregate any messages but would process any message individually? We investigate such a GNN framework in Chapter 7. The synchronous message-passing framework generates many messages, which would be costly to all processes. Furthermore, not all messages are relevant. Therefore, the proposed framework will instead follow asynchronous communication. We present GwAC as an architecture that avoids aggregation entirely. Instead, GwAC uses asynchronous communication and processes each message individually. Figure 7.2 shows an example of the dynamics.

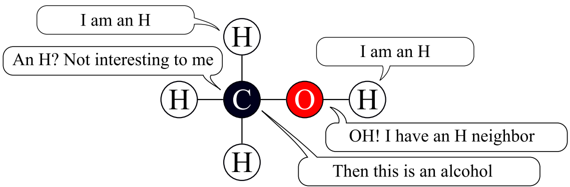


Figure 7.2: Example communication in the GwAC framework. Messages happen one at a time. Only important messages (such as from H to O) trigger a reaction from the receiver.

The topmost H sends a message to its neighbor. This message is not relevant for the C atom and gets discarded. Furthermore, the C atom does not communicate with its neighbors since it has no relevant information. The other H atom messages the O atom. Since this is relevant information, the O, in turn, communicates with its neighbors. This message reaches the C atom, which can correctly classify the graph. Note that the bottom two H atoms never communicate.

This different communication style helps with the previously-introduced problems. Since different nodes act a different number of times at different moments, asynchrony can serve as an information mechanism. Nodes receive every message individually; the important messages cannot become buried between many not-important messages. Suppose oversquashing were to occur on a node state. We expect asynchronous communication to help since all information can be directly forwarded and stored in the message instead. Furthermore, nodes can choose not to act when there is no impor-

tant information, not interrupting the flow of important information. This helps combat underreaching.

7.3 The GwAC framework

Algorithm 1: GNNs in the synchronous model.

```
1 repeat L times                                   ▷ L is the number of GNN layers
2   foreach Node v in parallel do
3     z_v^{i+1} = UPDATE(h_v^i, Agg_{w∈NB(v)}, MESSAGE(h_w^i))
4     g_v^{i+1} = GATE(z_v^{i+1})
5     h_v^{i+1} = g_v^{i+1} · z_v^{i+1} + (1 − g_v^{i+1}) · h_v^i
```

Algorithm 1 shows a simplified algorithm for GNNs following the message passing framework parametrized by three functions: `MESSAGE` is a function $f : R^{n\times d} \rightarrow R^{n\times m}$ that converts $d-$dimensional node states to $m-$dimensional messages. Common choices for messages are the identity functions where nodes directly share their state or MLPs such as the ones in Equation 5.1. Line 3 of the algorithm shows how to derive the new node embedding form the old embedding. `UPDATE` is a function $f : R^{n\times x} \rightarrow R^{n\times d'}$ to compute the new $d'-$ dimensional node states from information, the exact information available differs by GNN architecture. For example, the GIN from equation 5.2 uses an MLP for `UPDATE`, the identity for `MESSAGE` and summation as aggregation. Gin does not require lines 4 or 5 but directly uses this result as new node embedding. However, Li et al. [2016] propose to allow easy preservation of previous information. In line 4 `GATE` learns a value between 0 and 1 which we use to interpolate between the old embedding and the new embedding. However, other gating mechanisms exist that take into account the full history of prevous node embeddings such as in Universal Transformers [Dehghani et al., 2018].
In contrast, Algorithm 2 shows the asynchronous GwAC model, which uses the same kinds of functions. `MESSAGE` works exactly as in synchronous GNNs in that a node derives a message from its state. `UPDATE` differs slightly: in the asynchronous setting, we receive a single message so `UPDATE` computes a new embedding (or state) from the old embedding/state and the received message. We use the same kind of neural functions that we use for the synchronous `UPDATE`, for example, linear layers or MLPs. `GATE` also uses a single message but uses the same architectures as in the synchronous case to compute a value between 0 and 1. We also consider variants of GwAC

Algorithm 2: GNNs in the asynchronous model.

```
1  unreceivedMessages = []                    ▷ Tuples (receiver, message)
2  initializeList()                 ▷ e.g., a message to single or all nodes
3  repeat M times             ▷ M is the number of processed messages
4      v, message = unreceivedMessages[0]
5      g_v^{i+1} = GATE(h_v^i, message)
6      z_v^{i+1} = UPDATE(h_v^i, message)
7      h_v^{i+1} = g_v^{i+1} · z_v^{i+1} + (1 − g_v^{i+1}) · h_v^i
8      newMessage = MESSAGE(h_v^{i+1}, message)
9      foreach w in NB(v) do
10         unreceivedMessages.add(w, newMessage)
```

that discard messages if g_v^{i+1} is small. This corresponds to a conditional `continue` in the loop in line 3

The major difference between GwAC and synchronous GNNs is how messages are processed and emitted. In synchronous GNNs, every node sends and receives messages for a specified number of times. This number is the same for all nodes. On the other hand, we use a queue of unprocessed messages in GwAC. The initialization can depend on the task at hand. For example, we send an initial message to the node from which we want to compute the shortest path. Alternatively, we can pick one or more nodes at random. We iterate this queue and deliver the first message. As part of reacting to the message, a node may emit messages to its neighbors, which we append to the end of the queue. We also investigate an alternative model that uses a priority queue, and every message incurs a random delay.

Let us understand the introductory example from Figure 7.2 in this algorithm. We initialize a queue of unreceived messages with a message to each the H at the top and at the right of the molecule. In principle, we can also insert messages to the other two H but they will be processed identically like the top H. During the first run of the loop in line 3 of Algorithm 2, we extract the message to the top H: The atom decides that the message is interesting (line 5), so it updates its state (the actual value is irrelevant) and sends a message encoding “I am an H” to its neighbor. The H atom on the right does the same. Next, the C atom receives such a message. However, the message is not relevant for this atom, so `GATE` learns a small value (line 5) and we employ a `continue`. We insert no new messages. Finally, the O receives the message “I am an H”, which is relevant. After updating its

state (the value again does not matter) it puts a "OH! I have an H neighbor" in the queue for both neighboring nodes. When the C atom receives this message it can update its state to encode that the molecule is an alcohol.

GwAC can simulate GNNs

This GwAC framework is at least as expressive as message passing, which we prove by demonstrating that we can simulate a message passing GNN, namely GIN, with GwAC. Let S be a synchronous GNN that computes node embeddings h_v^l for all layers l and all nodes v. An asynchronous GwAC model A simulates S if A computes the same embeddings for every node for all possible input graphs. There may be intermediate states on the nodes, so A has to be aware of when an embedding is valid. The underlying idea for simulation comes from techniques in distributed computing to synchronize an asynchronous network to operate in rounds.

We follow Awerbuch [1985]'s α synchronizer that uses the notion of safe nodes. A node becomes safe if it receives its information from all neighbors and its information is received and confirmed by all neighbors. If a node and all its neighbors are safe, it starts the next round and sends new information. We will implement this idea in neural layers to simulate layers in GwAC. Besides the actual information, messages contain type information, assigning them as one of the four following types: (i) `pulse` messages encode that the sender starts a new layer, and the sent information is valid. The other messages are synchronization overhead, and nodes ignore the message information. (ii) `ack` messages acknowledge receiving a pulse message. A node must receive one acknowledgment from every neighbor before becoming safe again. The (iii) `origin` message only occurs once at the beginning to bootstrap the computation and is sent to an arbitrary node. Finally, (iv) nodes discard `noop` messages without any action. We need these messages in our definition of GwAC, where receipt is optional, but sending after receiving is not.

We simulate a simple GIN with sum aggregation. We will discuss simulating other architectures later. If the GIN has embeddings of size d, our GwAC model has a state size of $2d + 5$. The first d entries are the node state s that is equivalent to the GIN embedding up to the simulated point. The next d entries are an accumulator a summing up the neighborhood states as individual messages come in. The last four entries are for synchronization: The number of nodes w stores on whom the node is waiting for their state. If we receive the state from every neighbor, the accumulator contains the sum of all neighborhood states. Then, we can update the node. Nodes store

the number of unsafe nodes u. On every pulse, the node marks itself and every neighbor as unsafe. All nodes must become safe again before a node can send the next pulse. The third entry stores the number of layers l left to simulate. Since the first time a node acts needs slightly different handling, the fourth entry is a boolean if the node has ever reacted to a message. The last entry is a node's degree. The degree is available for simplicity. We could also let every node figure out its degree.

Table 7.3: Update function for GwAC simulating GIN. We execute the state update and message type on the first matching condition. Input to the function are the current node state s, accumulator value a, number w of neighbors the node is waiting on, number u of neighbors that are unsafe, if the node acted before i, and the currently simulated layer number l. The function computes a new value for every of these inputs in addition to a message type.

Condition	s'	a'	w'	u'	i'	l'	message type
`noop`=1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
l=0	⊥	⊥	⊥	⊥	⊥	⊥	⊥
`origin`=1	s	a	w	D	0	l	`pulse`
`i`=1	s	a+m	w-1	u	0	l	`pulse`
`pulse=1` ∧ w=0	UPDATE([s,(a+m)])	0	D-1	u-1	0	l-1	`ack`
`pulse`=1	s	a+m	w-1	u	0	l	`noop`
`ack`=1 ∧ u=0	s	a	w	D	0	l	`pulse`
`ack`=1	s	a	w	u-1	0	l	`noop`

Table 7.3 shows the node update function, depending on the values of w, u, l, and i and the message type. The table shows how to update the node state and which message type to send. Furthermore, messages always contain the current state. The table is a series of cascading `if` statements where we execute the first matching condition. ⊥ means that the message is discarded.

An example execution Before we prove that the function in this table does indeed simulate GIN, let us walk through one example. Let $G = v_1, v_2, v_3, v_4$ and $E = v_1, v_2, v_1, v_3, v_2, v_4, v_3, v_4$ be the graph on which we simulate. Initially, the nodes are in the following states:

Node	s	a	w	u	i	l
v_1	s_1	0	1	2	1	L
v_2	s_2	0	1	2	1	L
v_3	s_3	0	1	2	1	L
v_4	s_4	0	1	2	1	L

Now we send v_2 the initial message, on which it uses the third rule and sends pulse messages. Node v_1 receives this message, updates with the fourth rule, and replies with a pulse message. Node v_2 receives this reply from v_1 and also updates. However, since v_2 acted before, it uses the sixth rule. Node v_3 also receives the pulse message from v_1 and updates with the fourth rule, which means sending pulse messages. This gives the following node states with the pulse message from v_2 to v_4 still in transit.

Node	s	a	w	u	i	l
v_1	s_1	s_2	0	2	0	L
v_2	s_2	s_2	0	2	0	L
v_3	s_3	s_1	0	2	0	L
v_4	s_4	0	1	2	1	L

Next, we deliver the pulse message from v_3 to v_1. Now, node v_1 received information from all neighbors. Therefore, it can compute the next layer's state with the fifth rule. Note that v_1 is not yet safe since its neighbors have yet to acknowledge its pulse message. As part of the node update, v_1 now sends `ack` messages. Both v_2 and v_4 receive this information and apply the eighth rule. Finally, the pulse message from v_2 to v_4 also arrives. Node v_4 updates with the fourth rule and sends a pulse to its neighbors. Node v_3 receives it and also updates with the fifth rule, emitting an `ack` message. Nodes v_1 and v_4 receive this message, yielding the following state.

Node	s	a	w	u	i	l
v_1	s'_1	0	1	0	0	$L-1$
v_2	s_2	s_1	0	1	0	L
v_3	s'_3	0	1	1	0	$L-1$
v_4	s_4	s_2	0	0	0	L

Once v_2 is safe, and v_1 receives the `ack` message, v_1 is safe and can start simulating the next layer. Let v_2 receive the pulse message from v_4, update with the fifth equation, and send the `ack` message. Let v_1 receive this message and now use the seventh rule and send new pulse messages.

Node	s	a	w	u	i	l
v_1	s'_1	0	1	2	0	$L-1$
v_2	s'_2	0	1	1	0	$L-1$
v_3	s'_3	0	1	1	0	$L-1$
v_4	s_4	s_2	0	0	0	L

Unlike the other nodes, v_1 already proceeded to the next layer. Its neighbors v_2 and v_3 finished processing and are waiting for v_4 to acknowledge their pulse. Therefore, v_2 and v_3 can already receive the next pulse by v_1 and store it in their accumulator. Let us assume that they both do, which leaves us in the following state:

Node	s	a	w	u	i	l
v_1	s_1'	0	1	2	0	$L-1$
v_2	s_2'	s_1'	0	0	0	$L-1$
v_3	s_3'	s_1'	0	0	0	$L-1$
v_4	s_4	s_2	0	0	0	L

The one pending message is the pulse message from v_3 to v_4. When v_4 receives this message, it can update using the fifth rule and cause v_2, v_3, and v_4 to become safe and start the next layer.

Lemma 7.4. *Nodes receive a `pulse` message when executing the $i = 1$ condition.*

Proof. Preceding messages capture messages of type `noop` and `origin`. A node u could only receive an `ack` message from a neighbor v after v received a `pulse` message from every neighbor, including u. By elimination, the message type must be `pulse`. □

This allows us to proof the following Lemma via induction over i.

Lemma 7.5. *When node v emits its i-th `pulse` message, it must have already received $(i-1) \cdot D + 1$ pulse or origin messages.*

Proof. We start with $i = 1$, i.e., when nodes emit their first `pulse` message. After initialization, all nodes will send a `pulse` message after receiving one `origin` (third condition) or `pulse` message (Lemma 7.4).

Suppose the lemma holds for node v that just sent its i-th pulse. Before v another pulse, it must reach the seventh condition in Table 7.3. For this, u must decrease to 0, which requires reaching the fifth condition. This requires decrementing w to zero, for which v must receive $D-1$ `pulse` messages in the sixth condition and one further one in the fifth. In total, v has now received $(i-1) \cdot D + 1 + D = i \cdot D + 1$ `pulse` messages. □

Corollary 7.6. *Nodes require D `pulse` messages between their own (Lemma 7.5). Thus, nodes receive one `pulse` from every neighbor.*

We now know that a node receives a pulse message from every neighbor between two pulses. This proves that every node can access all the information it requires to compute the next GIN embedding.

Lemma 7.7. *When node v emits its i-th `pulse` message, it's state s equals the synchronous GNN representation h_v^{L-i-1}.*

Proof. We prove this by induction. The lemma holds for $i = 1$ when the node state s is h_v^0 based on initialization.

Suppose the lemma holds for node v that just sent its i-th pulse. We know from Lemma 7.5, that v now needs to receive D `pulse` messages (Lemma 7.5), one from each neighbor(Corollary 7.6) before it can send `pulse` $i + 1$. According to the induction hypothesis, node v receives z_w^{l-i-1} from every neighbor w in their ith `pulse` message. Upon receiving the last of these messages, v computes locally $\text{UPDATE}(z_v^{l-i-1}, \sum_w z_w^{l-i-1}) = z_v^{l-i}$ following the fifth condition. This state is unchanged until v emits `pulse` $i + 1$. □

This lemma confirms that the simulation is correct. With every `pulse` message, nodes in A proceed to the next embedding h_v^l.

Lemma 7.8 (Synchronous Simulation)**.** *$\mathcal{A}$ simulates GIN, where one round of `pulse` messages maps to one synchronous layer.*

Other Simulation Scenarios We can extend the above proof that uses simplifying assumptions to more general scenarios:

Disconnected Graphs The above proof requires connected graphs but can be quickly changed to allow any number of connected subcomponents. The easiest extension is to send one `origin` message to a random node in each connected component.

Variable Layer Sizes We used a constant embedding size d for GIN across layers, but we might want to have different dimensions $d_1, d_2, \ldots d_L$ for each layer. We can also simulate such GNNs by having a state and an embedding of appropriate size for each layer. Furthermore, we send the layer the sender simulates as part of the message. We can infer which accumulator to store the message by the layer.

Max Aggregation We can support other aggregation methods than summation by changing the logic of the accumulator. For example, if we want to simulate a `max`-aggregating network, we accumulate with element-wise maximum in the accumulator.

Mean Aggregation With a simple change in the fifth condition in Table 7.3, we can switch from a `sum` aggregation to `mean` aggregation. We only have to divide $a + m$ by the node's degree.

GCN GCN smoothes incoming messages by the neighboring nodes' degrees. We can simulate these smoothing factors using the `GATE` functions. For this, we also include the sender's degree in messages. Furthermore, we need to slightly change the accumulator updates in lines 4 and 6 in Table 7.3 to $a + \texttt{GATE(s,m)} \cdot m$.

GAT Unfortunately, GAT is one GNN architecture that we cannot simulate this way. GAT also scales each message by a factor. However, this factor is dependent on all other messages. We could only compute these factors after receiving messages from all neighbors, which is too late for processing an incoming message.

7.4 Expressiveness Analysis

Uniform Delays

We will now analyze GwAC's expressiveness in the WL framework. Lemma 7.8 gives us an easy lower bound:

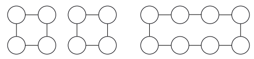
Corollary 7.9 (1WL)**.** *Models following the GwAC framework generally are at least as powerful as the 1 Weisfeiler-Lehman test.*

GNN models such as GIN are as powerful as the $1-$WL test. If GwAC can simulate such models, it must be as powerful as the 1WL test. Moreover, GwAC is more powerful, as we show in several examples in Figure 7.10, which shows three example graphs that are not distinguishable by $1-$WL. The left graph is the example from the chapter introduction to distinguish cycles of different lengths. The middle graph is another example with simple node features of indistinguishable graphs: Are the pairs of white nodes connected to the same blue node or not? The right graph is an example of cliques of different sizes: We consider the node classification problem; it was easy to decide as graph classification by just counting nodes.

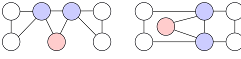
It suffices to learn to detect cycles of different lengths to distinguish the first two graphs. We can solve the first example by answering if there is a cycle of length 8. and the second example by answering if there is a cycle of length tree with two white nodes.

Lemma 7.11. *In GwAC, nodes $v_1, v_2, \dots v_k$ can determine if they are a k cycle.*

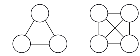
Proof. Let $v_1, v_2, \dots v_k$ be a cycle of k nodes and let computation start from node v. Every node executes the following protocol. If a node receives a message `COUNT-`i and it never received a message before, it stores i and



(a) From Garg et al. [2020]



(b) From Papp et al. [2021], added red node.



(c) 3− versus 4− clique

Figure 7.10: a) GwAC can separate the two graphs by identifying the 8−cycle and b) by identifying a 3− cycle with one blue node (Lemma 7.11. GwAC must not start from the red node to be able to separate the graphs. c) GwAC can separate the graphs (Lemma 7.12). This Lemma also allows to separate Rook-Shrikande graphs which are beyond 3−WL.

messages `COUNT-`$(i+1)$. If the same node then receives a `COUNT-`j message with $j > i$, the node ignores the message. Node v starts the computation by sending `COUNT-`0. The nodes on both paths store numbers as in a BFS from v. Eventually, the two BFS branches meet when one node w that first received a `COUNT-`i message receives a `COUNT-`j message with $i \geq j$. Then, w knows the circle is closed and sends a `FOUND-`$(j+i)$ that every node forwards once. Thus, all nodes become aware of the cycle and its length. □

The third example requires distinguishing cliques of different sizes. Separating cliques is a surprisingly difficult problem that requires expressive power beyond even $3-WL$ since we can reduce distinguishing the Rook-Shrikande graphs to clique detection [Chen et al., 2019b, Martinkus et al., 2022]: Rook graphs have four cliques. We now show that GwAC can solve this problem:

Lemma 7.12. *In GwAC, nodes $v_1, v_2, \dots v_k$ can determine if they are a k clique.*

Proof. Let $v_1, v_2, \dots v_k$ be in a k clique. The nodes iteratively find out they are in $2, 3, \dots k+1$ cliques. The starting node v will coordinate the other nodes. Initially, every node stores that they are in a 1−clique. To find a clique of size j, node v sends a `CLIQUE-`j message, which every neighbor of v forwards once. If neighbors of v receive $j-1$ such messages and are in a $j-1$ clique according to their state, they send a `CLIQUE-`j`-ACK` message to all neighbors (including v) and update their state to be in a j-clique. If v receives k many `CLIQUE-`j`-ACK` messages, it sends out a `CLIQUE-`$(j+1)$ message. Upon receiving k many `CLIQUE-`$(k+1)$`-ACK` messages, v knows there exists a $(k+1)$-clique and can propagate this information to its neighbors. □

Random Delays

There are limits to the GwAC model following Algorithm 2 that can be observed in the middle graph from Figure 7.10. Suppose we have one starting node for our message queue and start from the red node. In this case, the blue nodes receive the same message from the red node and propagate it to their white neighbors. Every white node receives the same message. No node can separate the two graphs. The symmetry-breaking of GwAC is not always strong enough. One practical solution we validate later is to start computation from multiple starting nodes. We now explore a theoretically even more powerful version of GwAC as shown in Algorithm 3. The key difference is that we insert messages into a priority queue with a random delay.

Algorithm 3: GNNs in the asynchronous model.

```
1 openMessages = []                ▷ Triples (delay, receiver, message)
  initializeList()   ▷ e.g., a message to single or all nodes repeat M
  times                    ▷ M is the number of processed messages
2   delay, v, message = openMessages[0]
3   g_v^{i+1} = GATE(h_v^i), message
4   z_v^{i+1} = UPDATE(h_v^i, message)
5   h_v^{i+1} = g_v^{i+1} · z_v^{i+1} + (1 − g_v^{i+1}) · h_v^i newMessage = MESSAGE(h_v^{i+1},
     message)
6   foreach w in NB(v) do
7     openMessage.add(delay + randomDelay(), w, newMessage)
```

Generating unique identifiers These delays can break otherwise existing symmetries. Let us revisit the above scenario of sending a message from the red to the two blue nodes in Figure 7.10b. With random delays, we might incur a large delay in one of those messages, such that the message from red to blue to the other blue node is faster than from red to blue. In this case, the two blue nodes are distinguishable. We will now show that we can use these delay races to create unique identifiers for the outer nodes in a star graph:

For simplicity, we connect all outer nodes. The algorithm also routes all communication through the center node c. This node will offer an ID to all neighbors, and nodes without an ID will try to claim it. Some nodes will hear of the claim before they receive the message from c. These nodes will

not claim this ID. If exactly one node claims the ID, they receive it, and c assigns the next ID. In case of more than one claim, c retries this ID with the claiming nodes. We can model this idea in the following network:

Nodes can have for states: `offering` is only used by c, `claiming` and `surrendering` used by the outer nodes and `done` used by all. We have five different message types: `offer` and `claim` sent by the c, `claim` and `surrender` used by the outer nodes and `noop`. Every node has an embedding of 6 values: One of the above one-hot encoded node states, its ID, and the current try to assign the ID offered by c. Additionally, c uses three additional features: a bit p if there was a previous claim on the offered ID in this try, the number w in the try on whose reply c is waiting, and the number n of neighbors with no ID yet. Messages are triples of a message type, the offered ID O-id, and the offering try O-try. We can model `UPDATE` and `MESSAGE` functions with the following tables using the same cascading condition semantics as Table 7.3. Practically, the two tables combine into the same function. We can use a binary feature we can inform nodes if they are the central or an outer node and choose the right set of conditions:

Table 7.13: `UPDATE` (a) and `MESSAGE` function (b) for the central node v in GwAC to assign unique IDs in a star graph.

(a)

Condition	state	try	ID	p	w	x
noop	⊥	⊥	⊥	⊥	⊥	⊥
done	⊥	⊥	⊥	⊥	⊥	⊥
try ≠ attempt	⊥	⊥	⊥	⊥	⊥	⊥
origin	offering	0	0	0	D-1	D-1
claim ∧ c = 0	offering	try	0	1	w-1	x
claim ∧ c > 0	offering	try+1	0	0	x-1	x
surrender	offering	try	0	c	w-1	x
w = 0 ∧ x > 0	offering	try+1	0	0	x-2	x-1
x = 0	done	0	0	0	0	0

(b)

type	O-id	O-try
⊥	⊥	⊥
⊥	⊥	⊥
⊥	⊥	⊥
offer	1	0
noop	⊥	⊥
offer	O-id	try+1
noop	⊥	⊥
offer	O-id+1	try+1
confirm	⊥	⊥

Lemma 7.15. *These two functions assign a unique identifier to every node in the star graph.*

Proof. We prove that (i) that no two nodes receive the same identifier and (ii) every node receives an identifier.

(i) The center node takes ID 0 for itself and only proposes IDs of 1 and above to its neighbors. If a neighbor does not change its ID, it stays with

Table 7.14: `UPDATE` (a) and `MESSAGE` function (b) for the outer nodes in GwAC to assign unique IDs in a star graph.

(a)

Condition	state	try	ID	c	w	x
noop	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
done	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
surrendering $\wedge$ offer $\wedge$ CID=ID	surrendering	O-try	ID	$\bot$	$\bot$	$\bot$
offer $\wedge$ O-try $>$ try	claiming	O-try	CID	$\bot$	$\bot$	$\bot$
claiming $\wedge$ O-try $>$ try	surrendering	O-try	CID	$\bot$	$\bot$	$\bot$
claiming $\wedge$ offer $\wedge$ ID $\neq$ CID	done	0	ID	$\bot$	$\bot$	$\bot$
claiming $\wedge$ confirm	done	0	ID	$\bot$	$\bot$	$\bot$

(b)

type	CID	O-try
$\bot$	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$
surrender	CID	O-try
claim	CID	O-try
surrender	CID	O-try
noop	$\bot$	$\bot$
noop	$\bot$	$\bot$

-1. Thus, ID 0 is unique. Suppose that two nodes w_1, w_2 received the same ID. Since IDs are increasing, w_1 and w_2 must have received the ID at the same time. Since the c only sends a `confirm` when the last node responds, w_1 and w_2 must have received the ID via the second to last condition in Table 7.14a. This is only possible if they were in state `claiming`, meaning they both send a `claim` message. However, in such cases, v would have started another try with that ID.

(ii) The center node receives ID 0. We have to show that all other nodes also receive an ID. Node v counts internally how many IDs it could offer successfully and only stops when this counter reaches 0. We initialize the counter with v's degree. Therefore, v will offer sufficiently many IDs. Let us suppose that w did not receive an ID. Node w will always try to claim an ID unless it is already surrendering this ID. Therefore, w surrenders for all IDs, even the last one. However, this is not possible since there is only w left, and no node could have sent a `claim` message to make w surrender. □

We can easily extend this protocol to arbitrary graphs: We send one node the first message that assigns itself 0 and IDs to its neighbors. After all neighbors have an ID, the node received 1 as ID repeats the protocol in the role of the center node, skipping the nodes who received an ID from 0. Eventually, every node will be adjacent to a center node and receive an ID. Alternatively, we create a global node that distributes IDs that we connect to all other nodes.

Lemma 7.16. *GwAC with random message delays can create IDs in arbitrary graphs.*

Solving Graph Isomorphism We now have a toolkit to solve graph isomorphism and distinguish any two graphs. Let G_1 and G_2 be two graphs we

want to check. We connect them via a bridge node u so that the resulting graph has a diameter of δ_G. We know from Loukas [2020] that a GNN with δ_G many layers, sufficiently wide layers, and unique identifiers can solve all problems on G. From Lemma 7.16, we know GwAC can create unique identifiers. From Lemma 7.8, we can simulate such a GNN for δ_G many rounds after assigning identifiers to nodes.

Despite this impressive theoretical result, we found that a simple queue assuming constant delays works better in practice. The random delays introduce noise in the gradient signal and the training process. Therefore, the same graph can cause many different outputs and gradients. We will present some practical results on expressiveness now.

Experiments

We conduct experiments on the recently proposed expressivness benchmarks: Limits1 and Limits2 by Garg et al. [2020], Triangles and LCC by Sato et al. [2021], $4-$cycles by Loukas [2020], and Skip-cycles and Rook-Shrikande by Chen et al. [2019b]. Furthermore, we use the aggregation-restricted constructions from Xu et al. [2019c] for `Max` and `Mean` aggregation. We want to test the capabilities of the asynchronous communication. Therefore, we choose a simple GwAC model GwAC-S: linear projections for the `UPDATE` and `MESSAGE` function in GwAC and choose to discard the `GATE` function. We also try a version that uses non-uniform delays GwAC-R. We compare against other powerful beyond-$1-$WL architectures: PPGN [Maron et al., 2019], SMP [Vignac et al., 2020], DropGNN [Papp et al., 2021], ESAN [Bevilacqua et al., 2022], and AgentNet [Martinkus et al., 2022]. Furthermore, we run a $1-$WL GIN for control.

We follow the setup from Papp et al. [2021] to use four message passing layers, except for Skip-Circles, where we use the better of 4 or 9 layers. Like DropGNN and ESAN, we compute one GwAC computation per node, making it the starting node. Per computation, we process $5n$ messages. We use 16 hidden layers for all problems except `Max` and `Mean` and Skip-Circles which we give 32 units. We train all models with the Adam optimizer with a learning rate of 0.01 for 1000 epochs. Table 7.17 shows the accuracy for each model:

We can see that the simple GwAC-S performs close to perfectly on all datasets, including Skip-Cycles and the beyond $3-WL$ datasets. This sets GwAC-S apart from the other methods that tend to struggle on Skip-Cycles, which require long-range information propagation or aggregation-restricted

Table 7.17: GwAC-S solves all beyond 1−WL benchmarks quasi-perfect even the challenging ones require long-range propagation (Skip-Cycles) or beyond 3−WL reasoning (Rook-Shrikande)

Dataset	GNN	PPGN	SMP	DropGNN	ESAN	AgentNet	GwAC-S	GwAC-R
Limits1	0.50±0.00	0.60±0.21	0.95±0.16	**1.00±0.00**	**1.00±0.00**	N/A	**1.00±0.00**	0.89±0.11
Limits2	0.50±0.00	0.85±0.24	**1.00±0.00**	**1.00±0.00**	**1.00±0.00**	N/A	**1.00±0.00**	**0.98±0.01**
Triangles	0.52±0.15	**1.00±0.02**	0.97±0.11	0.93±0.13	**1.00±0.01**	N/A	**1.00±0.01**	0.88±0.15
LCC	0.38±0.08	0.80±0.26	0.95±0.17	**0.99±0.02**	**0.96±0.06**	N/A	**0.96±0.03**	0.80±0.04
MAX	0.05±0.00	0.36±0.16	0.74±0.24	0.27±0.07	0.05±0.00	N/A	**1.00±0.00**	0.37±0.10
MEAN	0.28±0.31	0.39±0.21	0.91±0.14	0.58±0.34	0.18±0.08	N/A	**1.00±0.00**	0.71±0.11
4-cycles	0.50±0.00	0.80±0.25	0.60±0.17	**1.00±0.01**	0.50±0.00	**1.00±0.00**	**1.00±0.00**	**0.99±0.02**
Skip-Cycles	0.10±0.00	0.04±0.07	0.27±0.05	0.82±0.28	0.40±0.16	**1.00±0.00**	**1.00±0.00**	0.56 ±0.18
Rook-Shrikande	0.50±0.00	0.50±0.00	0.50±0.00	0.50±0.00	**1.00±0.00**	**1.00±0.00**	**1.00±0.00**	0.50 ±0.50

datasets. Again, this highlights the benefit of not using any aggregation. The inferior results of GwAC-R highlight the practical advantages of having a stable message order, even at the cost of theoretical power.

7.5 Underreaching and Oversmoothing

Theoretical Analysis

In synchronous GNNs, both Underreaching and Oversmoothing occur when information has to travel large distances. We may use too many GNN layers and lose information in the node states (Oversmoothing), or we might try to avoid Oversmoothing but use too few layers, and the information does not reach its target (Underreaching). Let us look at the simple example of a graph G with m where we want to pass information from node u to a node v that is distance d away. A normal GNN will require d rounds of message passing, which totals dm edges. We would only require d messages being sent along the path from u to v. However, in the synchronous framework, the nodes in this path must keep steady until the information from u passes them. This is true even when we use gating. On the other hand, nodes in GwAC only react when receiving an important message, drastically reducing the number of nodes.

Lemma 7.18. *Let G be a graph with n nodes and m edges. GwAC can send a message from any node u to any node v in $O(m)$ messages.*

Proof. The following protocol sends a message from u to v. Any number of nodes $w_1, w_2, \ldots w_n$ may receive an initial message `seek`. Every node $w \neq u$ will forward a `seek` message exactly once. When u receives a `seek` message, it emits a `send` message instead. Every node also forwards `send` messages exactly once. In the worst case, we have one `seek` and one `send` message

traveling over every edge, which is $2m$ messages total. The actual number may be lower: If nodes receive a `send` message first, they do not need to propagate `seek` messages. □

This protocol does not require u to be the starting node; in this case, we would send m messages at most. Crucially, the number of messages is independent of d, and whether we need to send information over short or long distances does not matter. This helps against Underreaching, and fewer messages also help with Overreaching.

Experiments

We experimentally validate these observations with a shortest-path-based experiment. Shortest paths are a common benchmark [Tang et al., 2020, Velickovic et al., 2020, Xu et al., 2021] to test information propagation. We slightly modify shortest paths to not compute the path length but only the parity of the length. This allows us to disregard the arithmetic problems that neural networks that we introduced in Chaper **??**. We will propose an architecture to improve learning shortest paths in the next Chaper 8.

We create a training set of 100 graphs with 10 nodes each. We base graphs on random spanning trees to which we add 2 random additional edges. We pick a starting node s for every graph to compute paths. In GNNs, we give this node a special embedding; in GwAC, we send this node an initial message. We use Neural Execution of Graph Algorithm)s (NEG) [Velickovic et al., 2020], Universal Transformers (UT) [Dehghani et al., 2018], and IterGNNs [Tang et al., 2020]. We employ the same GwAC-S model from the expressiveness section. Furthermore, we create two versions of GwAC that use a gating function inspired by the termination logic used in Graves [2016] and UT or IterGNN, which we call GwaC-UT and GwAC-Iter. After training for 1000 epochs with Adam, we test on increasingly large sets of 10 graphs with $10, 25, 50, 100, 250, 500, and 1000$ nodes each. We show the test results in Table 7.19.

The GwAC models perform better than the synchronous architectures, even the simple GwAC-S. This highlights the benefit of the better-aligned communication scheme for this task. We will now investigate the effects of Underreaching and Oversmoothing on the results in Table 7.19. To investigate Underreaching, we measure accuracy not overall but per distance. To capture a model collapsing to a constant prediction, we combine distances into pairs, i.e., we group accuracies 1 and 2, 3 and 4, and further pairs. Figure 7.20 shows the accuracies per pair. The results show that Underreaching

Table 7.19: Accuracy for predicting the parity of shortest paths to a starting node. Columns show different test graph sizes. Training size was 10 nodes. GwAC models outperform synchronous baseline, even the simpler GwAC-S without termination logic.

Model	10	25	50	100	250	500	1000
NEG	0.73±0.14	0.59±0.11	0.53±0.07	0.52±0.04	0.51±0.03	0.49±0.02	0.50±0.01
Universal	0.87±0.03	0.71±0.04	0.62±0.03	0.56±0.02	0.52±0.01	0.51±0.00	0.50±0.00
IterGNN	0.98±0.03	0.86±0.05	0.74±0.03	0.65±0.04	0.57±0.01	0.53±0.01	0.51±0.00
GwAC-S	0.99±0.00	0.91±0.05	0.82±0.07	0.76±0.10	0.64±0.10	0.58±0.13	0.53±0.10
GwAC-UT	1.00±0.00	0.99±0.00	0.98±0.02	0.97±0.03	0.96±0.05	0.95±0.05	0.94±0.06
GwAC-Iter	1.00±0.00	0.99±0.01	0.99±0.01	0.98±0.03	0.98±0.05	0.97±0.05	0.97±0.05

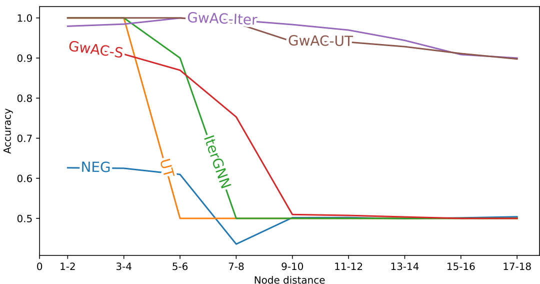


Figure 7.20: Accuracy for predicting the parity of shortest paths by distance to the starting node. Methods suffer from Underreaching if the accuracy for close-by nodes is better than for far-away nodes.

can explain a lot of the accuracy differences in Table 7.19.

We investigate Oversmoothing by only measuring the accuracy on close nodes that have a distance to s that we also observed in the training set. If we see deterioration on these nodes as the graphs and required iterations grow, we can suspect Oversmoothing. Figure 7.21 shows the accuracy on these close nodes. All methods show resistance against Oversmoothing, and some methods are even close to perfect results.

7.6 Complexity and Efficiency

Complexity We finish by comparing the complexity of GwAC to other GNN architectures. A simple GIN with L layers sends $O(LnD)$ messages and updates $O(Ln)$ nodes, L usually being a constant. Powerful architec-

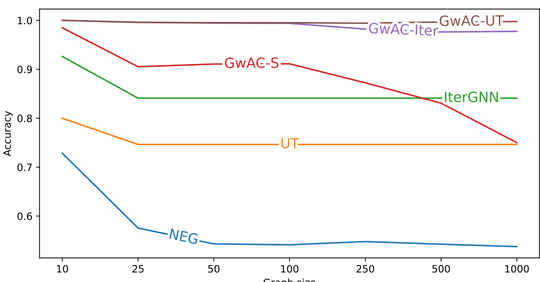


Figure 7.21: Accuracy for predicting the parity of shortest paths by graph size, only considering nodes in training-set distance to the starting node. Methods suffer from Oversmoothing if the accuracy stays constant with larger graph sizes.

tures such as ESAN [Bevilacqua et al., 2022] or DropGNN [Papp et al., 2021] achieve higher expressiveness by higher complexity. With the node-dropping augmentation, ESAN performs n rounds, each with the complexity of a single GIN pass. This totals $O(Ln^2D)$ messages and $O(Ln^2)$ node updates. Similarly, DropGNN computes r graphs with different dropped nodes, totaling $O(LnDr)$ messages and $O(Lnr)$ node updates.

Instead of a layer number L, the complexity driving hyperparameter in GwAC is the number of messages M to process. For M messages, GwAC computes M node updates; each update sends D messages to its neighbors, $O(MD)$ in total. For the expressiveness experiments, we used $M = n^2$, which makes the complexity comparable to ESAN/DropGNN. For the underreaching/oversmoothing experiments we tried $M = 15n$ and $M = 25$ yielding a comparable complexity to GIN.

Efficient Implementation GwAC's practical runtime is much slower despite comparable complexity. The reason lies in the more sequential Algorithm 2: we process one message at a time. This hinders hardware acceleration to compute large amounts of computation in parallel using GPUs. Furthermore, our prototypical implementation is entirely written in Python instead of languages like C++ like many contemporary libraries. However, we can still gain significant speedups by parallelizing computation in GwAC. For example, we use multithreading to run Algorithm 2 in parallel for differ-

ent graphs. If we run with different starting nodes, we can also parallelize those. We built a proof of concept in Python that uses—due to Python's global interpreter lock—multiprocessing instead of multithreading. This causes additional overhead to synchronize memory between processes. Nevertheless, Figure 7.22 shows a promising inversely proportional relationship between epoch time and number of processes available.

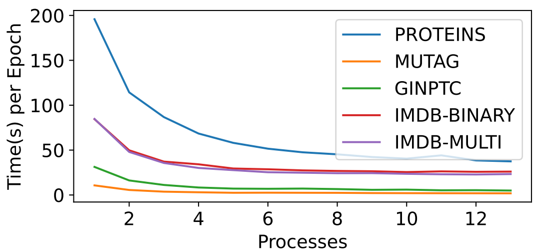


Figure 7.22: Time per one GwAC epoch for different datasets and number of processess available. Runtime is inversely proportional to the available compute.

Graph Classification Finally, we also evaluate the classification accuracy of GwAC on several well-known datasets from the TUDataset collection [Morris et al., 2020]. These datasets are on the smaller side to honor the slower runtime: MUTAG, PTC, PROTEINS, IMDB-B, and IMDB-M. We follow the evaluation scheme from Xu et al. [2019c]: measuring the test set accuracy over the overall best epoch across 10 different random seeds. The first three datasets use 16 or 32 hidden units, the latter two 64. For GwAC, we again use each node once as a starting node, experimenting with $M = 15$ and $M = 25$ per start. Furthermore, we attempt a simple attempt at skip connections: The final embeddings contain the last four node states. Table 7.23 shows that GwAC achieves comparable results as several $1-$WL and beyond $1-$WL GNN architectures.

Table 7.23: Graph classification accuracy (%). All GNNs are powerful and able to separate graphs beyond 1-WL. GwAC-S produces competitive results. *multiple versions, the score is the best-performing model

Model	MUTAG	PTC	PROTEINS	IMDB-B	IMDB-M
GraphSAGE Hamilton et al. [2017]	90.4±7.8	63.7±9.7	75.6±5.5	76.0±3.3	51.9±4.9
GCN Kipf and Welling [2017]	88.9±7.6	79.1±11.4	76.9±4.8	83.4±4.9	57.5±2.6
GAT Veličković et al. [2018]	85.1±9.3	64.5±7.0	75.4±3.8	74.9±3.8	52.0±3.0
GIN Xu et al. [2019c]	89.4±5.6	66.6±6.9	76.2±2.6	75.1±5.1	52.3±2.8
1-2-3 GNN Morris et al. [2019]	86.1	60.9	75.5	74.2	49.5
DropGNN Papp et al. [2021]	90.4±7.0	66.0±9.8	76.3±6.1	75.7±4.2	51.4±2.8
PPGN Maron et al. [2019]*	90.6±8.7	66.2±6.5	77.2±4.7	73±5.8	50.5±3.6
ESAN Bevilacqua et al. [2022]*	92.0±5.0	69.2±6.5	77.3±3.8	77.1±2.6	53.7±2.1
AgentNet Martinkus et al. [2022]	93.6±8.6	67.4±5.9	76.7±3.2	75.2±4.6	52.2±3.8
GwAC-S	90.4±4.1	63.7±9.1	76.7±7.1	74.6±3.6	52.1±3.6

8

Neural Status Registers as a Shortest-Path Case Study

Xu et al. [2021] and Velickovic et al. [2020] show that GNNs, especially with max aggregation, share similarities well with the Bellman-Ford algorithm: One GNN layer corresponds to one iteration in the Bellman-Ford algorithm. The authors found some success in learning shortest paths but we will show later that there is room for improvement. The underlying problem is that Neural networks struggle with learning arithmetic operations. This motivated us not to learn the actual lengths of shortest paths—a quantity— but instead learn their parity— a categorical value. Xu et al. [2020] define a property called algorithmic alignment to explain such behavior. Algorithmic alignment explains how efficiently we can train a neural architecture and how well the trained model will extrapolate to unseen tasks. If an architecture aligns well with how a task can be solved, we need fewer training examples because we do not need the solution structure. Furthermore, well-aligned models also extrapolate better to harder instances.
When it comes to arithmetics, Trask et al. [2018] showed that normal neural architectures quickly fail to compute simple arithmetic operations such as addition, subtraction, and multiplication when we input numbers outside the training set. They propose a better-aligned architecture called Neural

Comparison	Model	10^1	10^2	10^3	10^4	10^5	10^6	10^7
=	MLP	0.18 ±0.16	0.28 ±0.17	0.47 ±0.14	0.52 ±0.05	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	NPU	0.40 ±0.15	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	NALU	0.45 ±0.10	0.45 ±0.10	0.48 ±0.08	0.52 ±0.07	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	NAU	0.33 ±0.11	0.33 ±0.11	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12

Table 8.1: Comparing numbers with = after training on digits. The column header denotes the number that is compared against all numbers at most 5 apart. The table cells shows the mean absolute error to the correct prediction, therefore lower numbers are better.

Arithmetic Logic Units (NALU). Subsequent work by [Schlör et al., 2020] and [Madsen and Johansen, 2020] improve upon the NALU, Heim et al. [2020] propose a unit that can compute more operations, including division and power functions.

Table 8.1 shows a motivating example where we compare increasingly large numbers to their immediate neighbors after training only with single digits. Vanilla MLPs struggle a lot in this task. Surprisingly, so do these existing arithmetic architectures such as the NPU [Heim et al., 2020]], NALU [Trask et al., 2018], or NAU [Madsen and Johansen, 2020]. In this final chapter, we present an algorithmically well-aligned neural architecture to solve comparisons, the Neural Status Register (NSR). The NSR, as the NALU, NAU, or NPU is a quantitative architecture: The model directly operates on the numbers and manipulates them.

The alternative paradigm to tackle arithmetic problems is neuro-symbolic models: These models encode the number in a sequence of symbols and reason over these symbol sequences. A popular choice is text-encoding numbers and using powerful text-based architectures such as transformers [Saxton et al., 2019, Lample and Charton, 2020, Lewkowycz et al., 2022]. Recent large language models also show a surprising aptitude for arithmetic reasoning [Brown et al., 2020]. Kim et al. [2021] also encode the numbers as text but arrange the equations in a grid and use convolutional operations. Similarly, [Kaiser and Sutskever, 2016] and Freivalds and Liepins [2017] also use convolutions as reasoning architecture but encode the numbers in binary form. There also exist various other architectures that draw inspiration from circuits for other problems than arithmetic reasoning.: Graves et al. [2014, 2016], Grefenstette et al. [2015], Le et al. [2020], Weston et al. [2015], Zaremba and Sutskever [2016], Zaremba et al. [2016] all propose different versions of building differentiable memory. There furhter exist several works aiming to learn algorithms using different programming paradigms:

Chen et al. [2018], Li et al. [2017], Reed and de Freitas [2016] follow imperative programming and learn algorithms by composition of instructions. Cai et al. [2017], Feser et al. [2017] use functional programming and with recursive function calls. Dong et al. [2019], Evans and Grefenstette [2018] follow declarative programming and logic reasoning.

8.1 Neural Status Registers

The inspiration for Neural Status Registers is physical status registers: circuits we encounter in virtually all processors nowadays. These circuits are part of a processor's arithmetic logic unit (ALU) to evaluate conditionals such as `if x == y`. Internally, the status register subtracts y from x and inspects so-called condition code bits. For our purposes, we are interested in two bits: the sign bit, which is 1 if y is greater than x, and the zero bit, which is only 1 when $x == y$. For comparing the above `if` statement, we can directly inspect the zero bit.

The NSR translates these ideas into a fully-differentiable neural architecture. As with other quantitative architectures, the NSR takes a vector of numbers as inputs. The NSR's output is a probability p how confident it is about a comparison being true. The NSR independently learns which numbers are relevant for the comparison and also which comparison to evaluate with the architecture shown in Figure 8.2:

The input to the NSR is a vector $\boldsymbol{x}$. The first set of parameters W_M and W_S (shaded red) learn which two numbers in $\boldsymbol{x}$ to compare. We activate each of them with Softmax. Note that W_M and W_S are not symmetric. W_M learns the minuend m, number to subtract from, and W_S learns the subtrahend s, the number to subtract. We activate the difference $m - s$ with a $\hat{S}$ and a $\hat{Z}$ function that approximates the sign and zero bits from circuit status registers. The second set of learnable parameters W_+, W_0 combined with a bias term b learn to combine the output of these two functions to represent any comparison. The final probability p is the sigmoid activation of this combination. Formally, the NSR computes:

$$\begin{aligned} m &= \langle \boldsymbol{x}, softmax(W_M) \rangle \\ s &= \langle \boldsymbol{x}, softmax(W_S) \rangle \\ p &= \sigma(b + W_+ \hat{S}(m - s) + W_0 \hat{Z}(m - s)) \end{aligned}$$

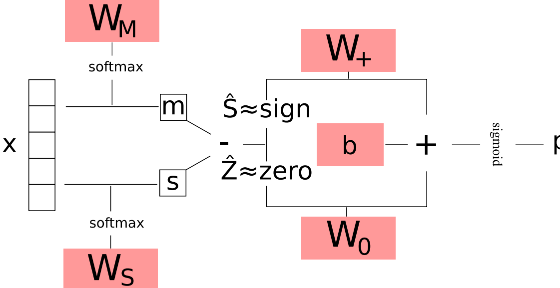


Figure 8.2: Architecture for a Neural Status Register. Learnable parameters are shaded in red. A first set of parameters W_M learns a number based on the input to subtract from, and W_S learns a number to subtract. We activate the difference with continuous approximations for sign and zero bits. A second set of weights W_+ and W_0 learns to weigh these bits with a learnable bias term. We output a confidence of a true comparison by activating this weighted sum with a sigmoid.

And for the learnable parameters W_M, W_S, W_+, W_0, and b, we can derive the following partial derivatives:

$$\frac{\partial p}{\partial b} = p(1-p) \qquad \frac{\partial p}{\partial W_+} = p(1-p)\hat{S}(d) \qquad \frac{\partial p}{\partial W_0} = p(1-p)\hat{Z}(d)$$

$$\frac{\partial p}{\partial m} = p(1-p)(\hat{S}'W_+ + \hat{Z}'W_0) \qquad \frac{\partial p}{\partial s} = -p(1-p)(\hat{S}'W_+ + \hat{Z}'W_0)$$

Choosing Bit Approximations The last two equations require our approximations for the sign and zero bits to be differentiable and to produce helpful gradient landscapes. For example, consider the following function approximation for the zero bit: $\hat{Z} = 1 - 2\tanh(m-s)^2$. The issue is that the sign and zero bits have a value of 0 for a significant part of the inputs. Furthermore, these bit approximations are a multiplier in most gradients, causing gradient vanishing. Figure 8.3a shows the gradient landscape for W_0 when we choose $\hat{Z} = 1 - 2\tanh(m-s)^2$. The gradients become vanishingly small except for the tiny part of the input space where $m \approx s$. Therefore, we deviate from an exact bit mapping and use -1 as the value for a false bit. We propose the following functions, which we also visualize in Figure 8.4a.

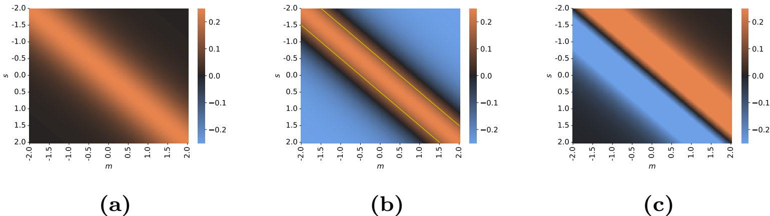


Figure 8.3: Gradient landscapes for different approximates for sign and zero bits. (a) landscape for W_0 when $\hat{Z} = 1 - \tanh{(m-s)}^2$: most gradients are zero. (b) Landscape for W_0 improved when $\hat{Z} = 1 - 2\tanh{(m-s)}^2$: mostly non-zero gradients. Positive gradients for $x_1 \approx x_2$ but gradients are also positive for $|x_1 - x_2| = 0.5$. We say that this difference falls below the resolution limit. (c) Landscape for m when $\hat{S} = \tanh{(m-s)}, W_0 = 0, W_+ = 1$. Gradients shrink towards 0 as $m - s$ grows.

$$
\begin{aligned}
\hat{S}(d) &= \tanh{(m-s)} \\
\hat{Z}(d) &= 1 - 2\tanh{(m-s)}^2
\end{aligned}
$$

Figure 8.3b shows the gradient landscape for W_0, which is now different from zero for virtually all values. However, we have two more issues to solve: First, the sign approximation is approximately 0 when $m \approx s$. If we want to learn comparisons between almost similar numbers, this function will be close to 0 most of the time, again causing vanishing gradients. Second, there

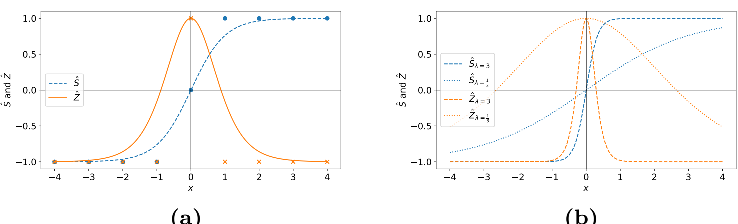


Figure 8.4: (a) Discrete sign (blue) and zero (orange) bits and the proposed continuous approximations. (b) Two different rescaled approximations with $\lambda = \frac{1}{3}$ or $\lambda = 3$.

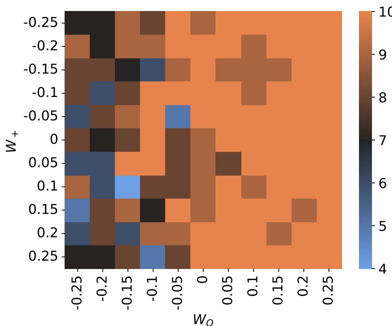


Figure 8.5: Number of successes to learn equality comparisons with the given weight initialization. Learning may fail if W_0 is negative highlighting initialization dependence on W_M and W_S.

is always a region where $m \neq s$ but $\hat{Z} > 0$, giving a wrong approximation. In these cases, we say that m and s are below the resolution limit. The proposed $\hat{Z}$ has a resolution limit of roughly 0.88. If we wanted to detect that numbers that are 0.5 apart are not equal, this version of the NSR would struggle. To address both issues, we introduce a factor λ to scale $m - s$ before applying the approximations. We can use $\lambda > 1$ to amplify differences between $m - s$, which lowers the resolution limit and the range where $\hat{S} \approx 0$. On the other hand, we can also use $\lambda < 1$ to diminish differences between $m - s$. This may help when we compare numbers that are far apart. For large differences $m - s$ both $\hat{S}$ and $\hat{Z}$ are saturated. Their derivatives occur as a factor in the gradients for W_M and W_S, which can vanish these gradients as Figure 8.4b. Figure 8.4b shows this scaled variants of $\hat{S}$ and $\hat{Z}$, which we define mathematically as:

$$\hat{S}(d) = \tanh\left(\lambda(m - s)\right)$$
$$\hat{Z}(d) = 1 - 2\tanh\left(\lambda(m - s)\right)^2$$

Redundancy in the NSR

Figure 8.5 gives a motivating example for building redundancy into the NSR. We initialize W_M and W_S Glorot Uniform [Glorot and Bengio, 2010] and W_+ and W_0 to each value in 0.05 increments in $[-0.25; 0.25]$. We repeat each combination for W_+ and W_0 10 times. The figure shows how often the NSR, with this initialization, learns the equality comparison successfully. We can see that—especially for small values for W_0— the NSR does not

Table 8.6

Task	Example input	Example output	Extrapolation
Comparisons	$10 > 5$ $1 > 5$	1 0	Larger numbers
Functions	$f(11, 10, 22, 44, 62)$ $f(10, 11, 22, 44, 62)$ $g(10, 10, 22, 44, 62)$ $g(11, 10, 22, 44, 62)$	66 22 66 22	Larger numbers for comparisons
MNIST	0 > 6 5 = 5	0 1	N/A
Recurrent	$\min([3, 5, 2, 7, 1])$ $count([3, 3, 5, 2, 3, 1])$	1 2	Longer sequences and/or larger numbers
SSSP			More nodes and/or larger edge weights

always succeed. This result is consistent with previous findings on learning inequality (the bitwise XOR) where Bland [1998] and Frankle and Carbin [2019]: Learning success is initialization dependent.

Kaiser and Sutskever [2016] address initialization dependence in the neural GPU by creating several redundant parameter sets that are forced to converge eventually. We apply a similar idea to the NSR. We adapt W_M and W_S to learn multiple versions of m and s and adapt W_+ and W_0 to combine all the differences into one logit for p. Unlike Kaiser and Sutskever [2016], we did not find regularization to make these versions converge helpful.

8.2 Experiments

We experimentally validate the NSR in various settings, in isolation, with other architectures, and recurrently with itself. Since we design the NSR to be algorithmically well-aligned with comparisons, we expect that the NSR can extrapolate well. Table 8.6 shows an overview of each experiment with its inputs and outputs and how we extrapolate to more difficult predictions. To align with the quantitative nature of the models, we supervise with the mean absolute error. We use 10 redundant NSR units and the Adam [Kingma and Ba, 2014] optimizer and average all results over 10 seeds.

ChatGPT

In addition to the NSR and other arithmetic architectures, we attempt to solve these problems with the large generative language model ChatGPT[1] with the default settings. We experimented with the following prompts

Comparison Which of the numbers x_1 and x_2 is larger

Functions Let $f(a, b, c, d, e)$ be a piecewise defined function that computes $e + 4$ if $a > b$ and $d - c$ otherwise. What is $f(x_1, x_2, x_3, x_4, x_5)$

Minimum What is the smallest number in $[x_1, x_2, \dots]$

Counting How many times does x_1 occur in $[x_2, x_3, \dots]$

Shortest Paths I am describing to you a graph in a set of triples (a,b,c) of undirected edges. a and b are node IDs and c is the distance between the two nodes. The edge list is $[e_1, e_2, \dots]$. Can you compute the lengths of shortest paths from v to all other nodes. You can skip the intermediate steps and only provide the final result

ChatGPT solved all instances of the first three problems. The solutions extrapolated to larger numbers. On the other hand, ChatGPT often failed on the counting task, especially as sequences became longer. ChatGPT also failed to compute shortest paths, though most distances were in the right ballpark. We found the final instruction necessary to ensure the reply does not exceed the output size. We also experimented with encoding MNIST images in text but found no encoding that ChatGPT understood. This small study shows the impressive progress general-purpose symbolic architectures have made on arithmetic tasks. However, it also demonstrates that quantitative architectures are still the better choice for some tasks.

Comparisons

First, we run the NSR on the motivating problem at the beginning of the chapter. The inputs are two digits; the output is the binary evaluation of one comparison between the two numbers. We run this experiment in six versions and try each comparison of $>, <, =, \neq, \geq, \leq$. We train with all pairs of digits. For testing, we use larger numbers around the powers of 10. For each power, we create a set of comparisons testing the power against each number at most 5 apart. Crafting the test set this way allows us to test on the hardest comparisons around the decision boundary: Large numbers that are almost the same. We balance the test set by only taking one different

[1] https://openai.com/product/chatgpt

number instead of all 10. This allows us to identify model failures when the score becomes 0.5. We compare the NSR with the four architectures from the chapter introduction: A multilayer perceptron (MLP), Neural Power Units (NPU) [Heim et al., 2020], Neural Arithmetic Units (NAU) [Madsen and Johansen, 2020], and Neural Arithmetic Logic Units (NALU) [Trask et al., 2018]. We append a linear readout to these models to allow a fair comparison to the NSR in terms of parameters. We train for 50000 epochs. Table 8.7 shows the results for all models. As we measure with mean absolute error, smaller numbers are better.

Comparison	Model	10^1	10^2	10^3	10^4	10^5	10^6	10^7
	MLP	0.00 ±0.00	0.00 ±0.00	0.01 ±0.01	0.23 ±0.16	0.49 ±0.12	0.52 ±0.01	0.52 ±0.01
	npu	0.75 ±0.19	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
$>$	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.06	0.50 ±0.09	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	MLP	0.00 ±0.00	0.00 ±0.01	0.09 ±0.14	0.33 ±0.18	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01
	npu	0.83 ±0.14	0.52 ±0.02	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01	0.52 ±0.01
$<$	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.05	0.51 ±0.03	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	MLP	0.18 ±0.16	0.28 ±0.17	0.47 ±0.14	0.52 ±0.05	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	npu	0.40 ±0.15	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
$=$	nalu	0.45 ±0.10	0.45 ±0.10	0.48 ±0.08	0.52 ±0.07	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	nau	0.33 ±0.11	0.33 ±0.11	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	MLP	0.18 ±0.11	0.27 ±0.12	0.42 ±0.11	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
	npu	0.43 ±0.15	0.52 ±0.05	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00	0.50 ±0.00
$\neq$	nalu	0.38 ±0.12	0.38 ±0.12	0.41 ±0.13	0.50 ±0.10	0.52 ±0.07	0.50 ±0.00	0.50 ±0.00
	nau	0.33 ±0.11	0.33 ±0.11	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12	0.35 ±0.12
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	MLP	0.00 ±0.00	0.00 ±0.00	0.01 ±0.01	0.23 ±0.16	0.49 ±0.12	0.52 ±0.01	0.52 ±0.01
	npu	0.75 ±0.19	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
$\geq$	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.06	0.50 ±0.09	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	MLP	0.00 ±0.00	0.00 ±0.00	0.02 ±0.04	0.25 ±0.19	0.52 ±0.01	0.52 ±0.00	0.52 ±0.00
	npu	0.74 ±0.20	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
$\leq$	nalu	0.00 ±0.00	0.00 ±0.00	0.11 ±0.06	0.50 ±0.08	0.52 ±0.00	0.52 ±0.00	0.52 ±0.00
	nau	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	nsr	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00

Table 8.7: Learning comparisons following the setup from Table 8.1 for six different comparisons and using the NSR. The NSR learns and extrapolates all comparisons, including the equality-based comparisons that are difficult for the other methods.

As expected, the NPU struggles most with this task. The NPU architecture learns power functions and is not well-aligned for comparisons that require addition and subtraction. MLPs are universal function approximators that learn all comparisons to some extent. However, MLPs are not aligned enough to extrapolate fully. The NALU weights saturate before learning the exact solution, so it does not extrapolate fully. On the other hand, the

NAU is well-aligned to learn a subtraction between numbers. This allows the NAU to learn all of $>, <, \geq, \leq$ and extrapolate over all tests. The NSR also perfectly solves these comparisons and learns the equality-based information $=$ and $\neq$. The zero bit allows for a direct signal to learn (in)equality. This experiment shows that the NSR can learn all comparisons. Next, we integrate the NSR into larger models with other architectures.

Piecewise-defined Functions

First, we combine the NSR with NAUs and NALUs. We wire NSR output p and $1 - p$ each to a separate model to allow learning two different branches of piecewise functions. We aim to learn the following two functions:

$$f(a, b, c, d, e) = \begin{cases} e + 4 & \text{if } a > b \\ d - c & \text{otherwise} \end{cases}$$

$$g(a, b, c, d, e) = \begin{cases} e + 4 & \text{if } a == b \\ d - c & \text{otherwise} \end{cases}$$

We sample a and b as in the previous experiment: training with digits and using powers and numbers close by for testing. We sample the remaining numbers used in addition and subtraction from $[-100; 100]$. Table 8.8 shows the mean average error. As before, we train for 50000 epochs.

Function	Comparison	2^1	2^2	2^3	2^4	2^5	2^6	2^7
f	MLP+NALU	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.96 ±2.72	3.92 ±4.69	3.95 ±5.57
	MLP+NAU	1.46 ±1.67	4.45 ±8.09	6.42 ±8.88	17.72 ±24.86	27.41 ±37.65	49.67 ±56.88	103.28 ±152.32
	NALU+NALU	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.02 ±0.03	0.30 ±0.81	2.77 ±4.77
	NALU+NAU	7.28 ±12.25	14.83 ±21.82	9.72 ±12.24	14.12 ±14.42	21.86 ±23.19	25.78 ±27.84	65.97 ±114.87
	NAU+NALU	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.01 ±0.00	0.02 ±0.01	0.03 ±0.01	0.06 ±0.03
	NAU+NAU	7.19 ±12.28	13.45 ±21.69	9.76 ±12.22	16.15 ±18.70	21.41 ±22.59	37.93 ±53.33	93.63 ±142.86
	NSR+NALU	0.00 ±0.00	0.00 ±0.00	0.01 ±0.00	0.01 ±0.00	0.01 ±0.01	0.02 ±0.01	0.05 ±0.06
	NSR+NAU	1.75 ±1.83	2.27 ±2.54	4.01 ±5.34	5.84 ±6.28	10.28 ±15.71	15.58 ±23.08	24.01 ±42.46
g	MLP+NALU	4.78 ±8.78	5.81 ±9.18	4.71 ±8.52	5.42 ±8.73	4.26 ±6.55	12.72 ±13.34	25.93 ±16.09
	MLP+NAU	8.28 ±9.89	8.64 ±10.52	9.30 ±9.71	15.96 ±13.28	18.37 ±13.11	33.30 ±9.29	37.38 ±10.04
	NALU+NALU	25.39 ±9.36	30.76 ±14.19	37.31 ±29.58	43.95 ±47.93	75.85 ±119.67	139.51 ±254.32	204.69 ±357.51
	NALU+NAU	34.51 ±10.76	36.23 ±13.06	36.75 ±14.96	34.85 ±15.66	33.97 ±14.21	32.75 ±11.56	40.14 ±18.70
	NAU+NALU	24.89 ±9.20	28.86 ±12.74	29.62 ±14.80	32.59 ±21.83	57.92 ±71.28	82.04 ±120.31	148.11 ±256.99
	NAU+NAU	33.86 ±11.40	37.59 ±11.18	35.92 ±13.19	37.52 ±13.16	40.80 ±16.17	42.15 ±24.79	57.77 ±37.23
	NSR+NALU	0.04 ±0.03	0.04 ±0.04	0.12 ±0.19	0.08 ±0.11	0.06 ±0.05	0.08 ±0.08	0.21 ±0.30
	NSR+NAU	5.66 ±8.20	6.48 ±8.34	7.24 ±6.48	12.72 ±11.35	17.48 ±12.49	23.94 ±24.45	41.20 ±50.47

Table 8.8: Combining comparison models with a downstream arithmetic unit to learn piecewise-defined functions. The NSR performs best with the most noticeable difference on the equality-based comparison.

Generally, all architectures can better learn comparisons when paired with a downstream NALU over a downstream NAU. In the previous experiment,

we saw that the clipping weights in NAU allow for much better extrapolation. However, we hypothesize that this clipping, with its associated zero gradients, makes gradients for upstream units harder. Looking only at the comparison units, the NSR again outperforms the remaining architectures: The NSR achieves the best extrapolation for f, tied with NAU, and is the only method that learns g at all. This function requires learning equality, so this result is consistent with the equality results from Table 8.7.

Digit Image Comparisons

We now combine architectures the opposite way: We combine the NSR with a convolutional neural network (CNN) but this time, the NSR is the downstream unit. We train a simple CNN for MNIST[2], multiply the class probabilities with the respective digit, and input pairs of these numerical predictions into an NSR. We want this NSR to predict which if comparisons $>$ and $=$ are equal. We train the entire architecture, including the CNN, end to end. Since the initial predictions of the CNN are close to uniform, we found it helps to lower the resolution limit by setting $\lambda = 10$. To set up pairs for $>$, we batch the MNIST dataset in batches of 50 and create ordered pairs from all images in each batch. To set up pairs for $=$, we create batches of 100. We use every correct equality comparison per batch to create balanced pairs and add one wrong pair of images. We train for 13 epochs over the MNIST training set and test with the same batching setup on the test dataset. Table 8.9 shows the correct predictions where the distance to the correct label is less than 0.5.

Model	Digit $>$	Digit$=$
MLP	95.18±0.78	70.79±5.25
NALU	75.90±20.36	64.99±0.97
NAU	76.01±20.44	63.26±1.68
NSR	96.92±1.06	80.05±12.07

Table 8.9: Combining comparison models with an upstream convolutional neural network to compare images of digits. The NSR produces the best results for both comparisons, especially for the equality comparison.

The accuracy does not reach that of the image classification task, which is above 99%, which is not surprising given this is a harder problem. Clas-

[2] `https://github.com/pytorch/examples/blob/234bcff4a2d8480f156799e6b9baae06f7ddc96a/mnist/main.py`

sifying an image provides a direct supervision signal for the CNN to learn to distinguish images. In this comparison task, the only supervision signal for the CNN is if an image is larger/equal to another digit. To identify the digit, we need to compare the same image to many others. The comparison-aligned architecture of the NSR allows it to outperform the other models on this task, again with the most clear difference on the equality comparison.

Recurrent Problems

We now use the NSR in two recurrent scenarios. First, we are learning the minimum out of a sequence of numbers. Second, we count how often the first element occurs in the rest of the sequence. To find the minimum, we iterate the sequence and compare the current number with our belief of the minimum in the NSR. We then update our minimum as the NSR-weighted average of the current number and our previous minimum. For counting, we also iterate the sequence and increment a counter with the NSR output, comparing the current element against the first sequence element (the element we want to count).

We train the minimum search with 400 sequences of length 5 with numbers randomly chosen from $[1, 10]$. We train counting by sampling 6 elements where the last five are at most 5 different. For testing, we extrapolate to larger numbers by multiplying the upper limit with powers of two, and we extrapolate to larger sequences in steps of 5 up to 50 (51) elements. We use LSTMs [Hochreiter and Schmidhuber, 1997] as additional baselines, which can count as shown by Suzgun et al. [2019a,b], Weiss et al. [2018]. Figure 8.10 shows the results for LSTMS and the previous comparison models.

The NSR is the only method to learn both problems and extrapolate in both dimensions consistently. Consistent with existing literature, LSTMs learn to count in the training set but fail to learn the minimum and to extrapolate to more difficult counting problems. The previous baselines make mistakes in the training set for both tasks. They show resilience to extrapolating to larger numbers but not to larger sequences.

8.2.1 Shortest Paths with Graph Neural Networks

Finally, we combine the NSR with our main thesis topic: message-passing graph neural networks [Gilmer et al., 2017, Battaglia et al., 2018] (GNNs). GNNs require an aggregation of messages in one round of message passing. Motivated by the strong results of learning a minimum with the NSR, we use it as the aggregation function in learning a shortest path setting. Tech-

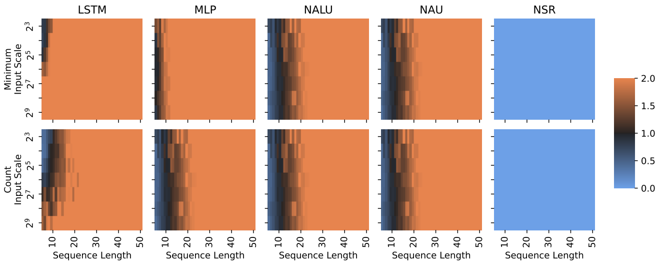


Figure 8.10: Combining comparisons models recurrently with themselves to learn the minimum in a sequence and to count occurrences. The NSR learns and extrapolates both problems, while baselines either struggle with the task themselves or with extrapolation.

nically, predicting with the NSR over a sequence has no guarantee of being permutation invariant, but if we learn successfully, we learn a permutation invariant function. Unlike current GNN architectures for shortest paths that we give the information to use the minimum/maximum, our NSR-GNN will learn this aggregation. Formally we contrast against IterGNN [Tang et al., 2020] and NEG [Velickovic et al., 2020]:

$$\text{NEG/IterGNN:} h^{l+1} = \texttt{UPDATE}(h^l, \min_{v \in N(v)} (\texttt{MESSAGE(v)}))$$
$$\text{NSR-GNN:} h^{l+1} = \texttt{NSR}(h^l, \Vert_{v \in N(v)} (\texttt{MESSAGE(v)}))$$

$N(v)$ contains the neighbors of v, MESSAGE and UPDATE are linear functions. Compared to NEG, IterGNN uses MESSAGE and UPDATE functions that must be homomorphic; NEG makes no restriction.

For training, we follow Velickovic et al. [2020] and learn to imitate the Bellman-ford algorithm with supervision per iteration. We create 10 graphs with 10 nodes each and randomly sampled edge weights from $[1, 10]$. We base graphs on a randomly chosen spanning tree to which we add on expectation one random edge. Additionally, we add self-loops to every node to retain their previous information. We train for 1000 epochs and then test in two extrapolation dimensions: First, we increase the number of nodes by multiplying the training size with powers of two. Second, we also scale the upper edge weight by powers of two. Figure 8.11 shows the mean average

error between predictions and the actual distances.

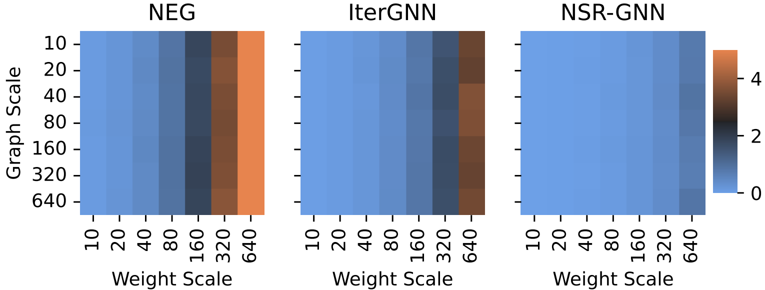


Figure 8.11: Learning and extrapolating shortest paths. The top left corner equals the training setting. Along the x-axis, we increase the maximum edge weights. Along the y-axis, we increase the graph size. All models can extrapolate to larger graphs. The NSR is the only architecture extrapolating in both dimensions.

All models show extrapolation capability to larger graphs. However, NEG struggles with extrapolating to larger weights. The homomorphic functions of IterGNN allow for better extrapolation than NEG. The NSR also outperforms IterGNN with almost perfect extrapolation in both dimensions.

Resolution Limit Ablation on λ

In most previous examples, the input numbers required the NSR to find differences of 1 or larger. We now conduct an ablation on λ to show that smaller or larger differences are fine, as suggested by the MNIST experiment (where the intermediate state had no guarantees of minimum differences). We copy the setup for simple comparisons but rescale the inputs by different powers of 10, such that the smallest difference δ between two non-equal numbers varies from 10^{-3} to 10^{5}. Orthogonally, we vary lambda 10^{-3} to 10^{3}. Figure 8.12 shows the mean average error for the training set. We observed that training errors of 0 extrapolate, so we emit test errors for conciseness.

In any case, the NSR learns successfully when the differences δ between numbers are large. The same is not true for small distances because of

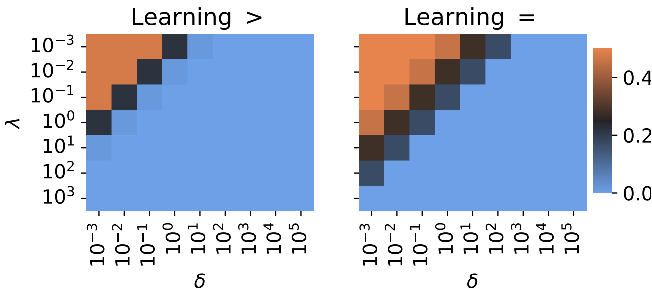


Figure 8.12: Mean average error of the NSR for learning $>$ and $=$ when we vary the distance between different digits (δ, x-axis) and internal scaling parameter λ. If both δ and λ are small, we experience the first resolution error, and the NSR fails to learn $>$. If both $\delta < \lambda^{-1}$, we experience the second resolution error, and the NSR fails to learn $=$.

the resolution limit. We can see that the NSR fails to learn $>$ when the difference between δ and λ becomes too large, in our case, three orders of magnitude or more. For learning $=$ we even need λ to be as large as δ^{-1}. Both plots show that values of λ satisfying these conditions can counteract the effects of smaller minimum differences.

Redundancy Ablation

Last, let us conduct an ablation on the effect of redundant units in the NSR. We revisit the piecewise-defined functions with an identical setup, except we vary the redundant units in the NSR from 1 to 15. Figure 8.13 shows the mean average error of the testing set without extrapolation for each redundancy level.

Both functions benefit from at least some level of redundancy. For learning f it seems already sufficient to have 2 or more NSR units. On the other hand, learning g requires 9 or more redundant NSR units to train robustly.

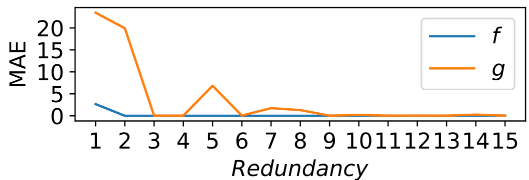


Figure 8.13: Mean average error for the NSR with varying redundancy to learn piecewise-defined functions from Table 8.8. At least some level of redundancy is required to learn either function robustly.

9 Conclusion

In this thesis, we study two major angles to improve current message-passing Graph Neural Networks: Understanding what GNNs learn and how to explain their decision-making to humans. On the other hand, understanding the limits of the message-passing model and improving on the limits.

We examined the improvements of the new GNN architecture on many early benchmark datasets. The novelty in GNNs is their ability to combine simultaneous reasoning over features and edges. We proposed a measure to evaluate how much the datasets require this superior reasoning. We also evaluated several GNN architectures to what extent they leverage this combined reasoning practically. The notion we use for both measurements is solvable nodes. Solvable nodes are nodes that a model can consistently predict correctly. We also compare agreement between different GNN architectures. It might be interesting to investigate the solvable and unsolvable nodes with explanation methods to understand systematic successes or failures of GNNs. Furthermore, future work could try to leverage disagreement between GNN architectures to build better models through ensembling.

We then proposed two new methods to explain GNN predictions. For the first method, we highlighted the importance of finding explanations consis-

tent with the training data distribution. Otherwise, we might mistake adversarial attacks on a model as explanations for it. Our proposed Contrastive Graph Explanation (CoGE) method uses a contrastive approach that compares a graph against other examples in the training set. We can ensure that these explanations are consistent with the training distribution since explanations consist of training graphs. Our second model, GraphChef, allows for a new level of explanation: While current explanation models highlight what parts of the input are important, they do not reveal how the GNN uses the inputs and why they are important. GraphChef reveals the full decision process: Which inputs are important, and how are these inputs used. We consider improving the performance on high-dimensional datasets as the main angle to improve Graphchef in future work.

Finally, we showed several problems that the currently-used benchmarks to evaluate explanation methods have. These problems allow a GNN model to solve the benchmarks differently from our expectations. Explaining such a GNN and comparing these explanations against our expectations will suggest that the explanation method performs poorly, which may not be true. We proposed three new benchmarks that avoid these problems. However, we do not believe the proposed benchmarks are exhaustive for GNN usages, so future work should investigate further benchmarking sets.

We also investigated how to break the current limits of message-passing GNNs and identified that message aggregation aggravates many problems. We proposed a new framework for GNNs with asynchronous communication (GwAC) that avoids aggregation completely and handles every message individually. This framework shows promising improvements on many GNN problems: Expressiveness bounds by $1-$WL, Oversmoothing, and Underreaching. However, we only briefly studied how to train GwAC models effectively and robustly. For future work, we might investigate how to adapt techniques for GNN training, such as residual connections, dropouts, or normalizations. Another promising line for future work is building more efficient implementations for GwAC.

Finally, we investigated the seemingly simple problem of number comparisons with neural networks. We found that current architectures, including architectures built for arithmetics, struggle to learn and extrapolate this problem. We proposed the Neural Status Register (NSR) architecture that is algorithmically well-aligned to solve comparisons. We used the NSR in various comparison-related experiments and got improvements. Future work might incorporate the NSR as a building block in more complex algorithmic reasoning architectures.

Bibliography

Emmanuel Abbe. Community detection and stochastic block models: recent developments. The Journal of Machine Learning Research, 2017.

Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. Sanity checks for saliency maps. In Conference on Neural Information Processing Systems (NeurIPS), Montreal, Canada, December 2018.

Chirag Agarwal, Marinka Zitnik, and Himabindu Lakkaraju. Probing gnn explainers: A rigorous theoretical and empirical analysis of gnn explanation methods. In International Conference on Artificial Intelligence and Statistics, pages 8969–8996. PMLR, 2022.

Chirag Agarwal, Owen Queen, Himabindu Lakkaraju, and Marinka Zitnik. Evaluating explainability for graph neural networks. Scientific Data, 2023.

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In International Conference on Learning Representations (ICLR), 2021.

Kenza Amara, Rex Ying, Zitao Zhang, Zhihao Han, Yinan Shan, Ulrik Brandes, Sebastian Schemm, and Ce Zhang. Graphframex: Towards systematic evaluation of explainability methods for graph neural networks. arXiv preprint arXiv:2206.09677, 2022.

Marco Ancona, Enea Ceolini, Cengiz Öztireli, and Markus Gross. Towards better understanding of gradient-based attribution methods for deep neural networks. In International Conference on Learning Representations (ICLR), Vancouver, Canada, April 2018.

Baruch Awerbuch. Complexity of network synchronization. Journal of the ACM, 1985.

Caglar Aytekin. Neural networks are decision trees. arXiv preprint arXiv:2210.05189, 2022.

Steve Azzolin, Antonio Longa, Pietro Barbiero, Pietro Liò, and Andrea Passerini. Global explainability of gnns via logic combination of learned concepts. arXiv preprint arXiv:2210.07147, 2022.

Mohit Bajaj, Lingyang Chu, Zi Yu Xue, Jian Pei, Lanjun Wang, Peter Cho-Ho Lam, and Yong Zhang. Robust counterfactual explanations on graph neural networks. In Conference on Neural Information Processing Systems (NeurIPS), volume 34, 2021.

Federico Baldassarre and Hossein Azizpour. Explainability techniques for graph convolutional networks. arXiv preprint arXiv:1905.13686, 2019.

Pablo Barceló, Egor Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan-Pablo Silva. The logical expressiveness of graph neural networks. In International Conference on Learning Representations (ICLR), 2020.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261, 2018.

Beatrice Bevilacqua, Fabrizio Frasca, Derek Lim, Balasubramaniam Srinivasan, Chen Cai, Gopinath Balamurugan, Michael M. Bronstein, and Haggai Maron. Equivariant subgraph aggregation networks. In International Conference on Learning Representations (ICLR), 2022.

Richard Bland. Learning XOR: exploring the space of a classic problem. Department of Computing Science and Mathematics, University of Stirling, 1998.

Olcay Boz. Extracting decision trees from trained neural networks. In ACM SIGKDD international conference on Knowledge discovery and data mining (KDD), 2002.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Neural Information Processing Systems (NeurIPS), virtual, 2020.

Rickard Brüel-Gabrielsson, Mikhail Yurochkin, and Justin Solomon. Rewiring with positional encodings for graph neural networks. arXiv preprint arXiv:2201.12674, 2022.

Chen Cai and Yusu Wang. A simple yet effective baseline for non-attributed graph classification. In International Conference on Learning Representations (ICLR) Workshop on Representation Learning on Graphs and Manifolds, 2018.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In International Conference on Learning Representations (ICLR), Toulon, France, 2017.

Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In AAAI Conference on Artificial Intelligence (AAAI), 2020a.

Jialin Chen and Zhitao Ying. Tempme: Towards the explainability of temporal graph neural networks via motif discovery. In Thirty-seventh Conference on Neural Information Processing Systems, 2023.

Jialin Chen, Shirley Wu, Abhijit Gupta, and Zhitao Ying. D4explainer: In-distribution explanations of graph neural network via discrete denoising diffusion. In Thirty-seventh Conference on Neural Information Processing Systems, 2023.

Kaiyu Chen, Yihan Dong, Xipeng Qiu, and Zitian Chen. Neural arithmetic expression calculator. arXiv preprint arXiv:1809.08590, 2018.

Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In International Conference on Machine Learning (ICML), 2020b.

Ting Chen, Song Bian, and Yizhou Sun. Are Powerful Graph Neural Nets Necessary? A Dissection on Graph Classification. ArXiv, 2019a.

Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. On the equivalence between graph isomorphism testing and function approximation with gnns. In Conference on Neural Information Processing Systems (NeurIPS), 2019b.

Mark Craven and Jude Shavlik. Extracting tree-structured representations of trained networks. In Conference on Neural Information Processing Systems (NeurIPS), 1995.

Enyan Dai and Suhang Wang. Towards Self-Explainable Graph Neural Network. In ACM International Conference on Information & Knowledge Management (CIKM), 2021.

Darren Dancey, Dave Mclean, and Zuhair Bandar. Decision Tree Extraction from Trained Neural Networks. In International Florida Artificial Intelligence Research Society Conference (FLAIRS), 2004.

Asim Kumar Debnath, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. Journal of Medicinal Chemistry, 1991.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In International Conference on Learning Representations (ICLR), 2018.

Amit Dhurandhar, Pin-Yu Chen, Ronny Luss, Chun-Chen Tu, Paishun Ting, Karthikeyan Shanmugam, and Payel Das. Explanations based on the missing: Towards contrastive explanations with pertinent negatives. In Conference on Neural Information Processing Systems (NeurIPS), Montreal, Canada, December 2018.

Amit Dhurandhar, Tejaswini Pedapati, Avinash Balakrishnan, Pin-Yu Chen, Karthikeyan Shanmugam, and Ruchir Puri. Model agnostic contrastive explanations for structured data. arXiv preprint arXiv:1906.00117, 2019.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In International Conference on Learning Representations (ICLR), New Orleans, USA, 2019.

Alexandre Duval and Fragkiskos D Malliaros. Graphsvx: Shapley value explanations for graph neural networks. In Joint European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), 2021.

Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In ACM Symposium on Principles of distributed computing (PODC), 2013.

Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. In International Conference on Learning Representations (ICLR 2020), 2020.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. Journal of Artificial Intelligence Research, 2018.

Lukas Faber and Roger Wattenhofer. Asynchronous message passing: A new framework for learning in graphs, 2023a. URL `https://openreview.net/forum?id=2_I3JQ70U2`.

Lukas Faber and Roger Wattenhofer. Neural status registers. In International Conference on Machine Learning, pages 9508–9522. PMLR, 2023b.

Lukas Faber, Amin K Moghaddam, and Roger Wattenhofer. Contrastive graph neural network explanation. arXiv preprint arXiv:2010.13663, 2020.

Lukas Faber, Amin K. Moghaddam, and Roger Wattenhofer. When comparing to ground truth is wrong: On evaluating gnn explanation methods. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, pages 332–341, 2021a.

Lukas Faber, Yifan Lu, and Roger Wattenhofer. Should graph neural networks use features, edges, or both? arXiv preprint arXiv:2103.06857, 2021b.

Junfeng Fang, Wei Liu, Xiang Wang, Zemin Liu, An Zhang, Yuan Gao, and He Xiangnan. Evaluating post-hoc explanations for graph neural networks via robustness analysis. In Conference on Neural Information Processing Systems (NeurIPS), 2023.

Zachary Feinstein. It's a trap: Emperor palpatine's poison pill. arXiv preprint arXiv:1511.09054, 2015.

Qizhang Feng, Ninghao Liu, Fan Yang, Ruixiang Tang, Mengnan Du, and Xia Hu. DEGREE: decomposition based explanation for graph neural networks. In International Conference on Learning Representations (ICLR), 2022.

Wenzheng Feng, Jie Zhang, Yuxiao Dong, Yu Han, Huanbo Luan, Qian Xu, Qiang Yang, Evgeny Kharlamov, and Jie Tang. Graph random neural networks for semi-supervised learning on graphs. In Conference on Neural Information Processing Systems (NeurIPS), 2020.

John K. Feser, Marc Brockschmidt, Alexander L. Gaunt, and Daniel Tarlow. Neural functional programming. In International Conference on Learning Representations (ICLR), Toulon, France, 2017.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In ICLR Workshop on Representation Learning on Graphs and Manifolds, 2019.

Matthias Fey, Jan E Lenssen, Christopher Morris, Jonathan Masci, and Nils M Kriege. Deep graph matching consensus. arXiv preprint arXiv:2001.09621, 2020.

Jean Feydy, Thibault Séjourné, François-Xavier Vialard, Shun-ichi Amari, Alain Trouve, and Gabriel Peyré. Interpolating between optimal transport and mmd using sinkhorn divergences. In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS), Naha, Japan, April 2019.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In International Conference on Learning Representations (ICLR), New Orleans, USA, 2019.

Karlis Freivalds and Renars Liepins. Improving the neural gpu architecture for algorithm learning. arXiv preprint arXiv:1702.08727, 2017.

Vikas Garg, Stefanie Jegelka, and Tommi Jaakkola. Generalization and representational limits of graph neural networks. In International Conference on Machine Learning (ICML), 2020.

Johannes Gasteiger, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. In International Conference on Learning Representations (ICLR), 2020.

Simon Geisler, Daniel Zügner, and Stephan Günnemann. Reliable graph neural networks via robust aggregation. Advances in Neural Information Processing Systems, 33:13272–13284, 2020.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In International Conference on Machine Learning (ICML), 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In International Conference on Artificial Intelligence and Statistics (AISTATS. JMLR Workshop and Conference Proceedings, 2010.

Alex Graves. Adaptive computation time for recurrent neural networks. arXiv preprint arXiv:1603.08983, 2016.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. arXiv preprint arXiv:1410.5401, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. Nature, 2016.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In Neural Information Processing Systems (NeurIPS), Montreal, Canada, 2015.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, 2016.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In Conference on Neural Information Processing Systems (NeurIPS), volume 30, 2017.

Arman Hasanzadeh, Ehsan Hajiramezanali, Shahin Boluki, Mingyuan Zhou, Nick Duffield, Krishna Narayanan, and Xiaoning Qian. Bayesian graph neural networks with adaptive connection sampling. In International Conference on Machine Learning (ICML), 2020.

Niklas Heim, Tomáš Pevnỳ, and Václav Šmídl. Neural power units. Neural Information Processing Systems (NeurIPS), remote, 2020.

Mark Heimann, Haoming Shen, Tara Safavi, and Danai Koutra. Regal: Representation learning-based graph alignment. In International Conference on Information and Knowledge Management (CIKM), Turin, Italy, October 2018.

Anna Himmelhuber, Mitchell Joblin, Martin Ringsquandl, and Thomas Runkler. Demystifying Graph Neural Network Explanations. In Joint European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), 2021.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 1997.

Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. A benchmark for interpretability methods in deep neural networks. In Proceedings of 33th Conference on Neural Information Processing Systems, pages 9737–9748, 2019.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. Advances in neural information processing systems, 33, 2020.

Qian Huang, Horace He, Abhay Singh, Ser-Nam Lim, and Austin R. Benson. Combining Label Propagation and Simple Models Out-performs Graph Neural Networks. International Conference on Learning Representations (ICLR), 2021.

Qiang Huang, Makoto Yamada, Yuan Tian, Dinesh Singh, Dawei Yin, and Yi Chang. GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks. ArXiv, 2020.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax. International Conference on Learning Representations (ICLR), 2016.

Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In International Conference on Learning Representations (ICLR), San Juan, Puerto Rico, 2016.

Segwang Kim, Hyoungwook Nam, Joonyoung Kim, and Kyomin Jung. Neural sequence-to-grid module for learning symbolic rules. In AAAI Conference on Artificial Intelligence (AAAI), remote, 2021.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In International Conference on Learning Representations (ICLR), 2017.

Johannes Klicpera, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. In Conference on Neural Information Processing Systems (NeurIPS), 2019.

Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi Yan, and Orion Reblitz-Richardson. Captum: A unified and generic model interpretability library for pytorch, 2020.

Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo. Deep neural decision forests. In IEEE International Conference on Computer Vision (CVPR), 2015.

R. Krishnan, G. Sivakumar, and P. Bhattacharya. Extracting decision trees from trained neural networks. Pattern Recognition, 1999.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia, 2020.

Hung Le, Truyen Tran, and Svetha Venkatesh. Neural stored-program memory. In International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia, 2020.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. Neural Information Processing Systems (NeurIPS), New Orleans, USA, 2022.

Chengtao Li, Daniel Tarlow, Alexander L. Gaunt, Marc Brockschmidt, and Nate Kushman. Neural program lattices. In International Conference on Learning Representations (ICLR), Toulon, France, 2017.

Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In IEEE international conference on computer vision (ICCV), pages 9267–9276, 2019.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In AAAI Conference on Artificial Intelligence (AAAI), 2018.

Wenqian Li, Yinchuan Li, Zhigang Li, Jianye Hao, and Yan Pang. DAG matters! gflownets enhanced explainer for graph neural networks. In The Eleventh International Conference on Learning Representations (ICLR), 2023.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In International Conference on Learning Representations (ICLR), 2016.

Yanbin Liu, Linchao Zhu, Makoto Yamada, and Yi Yang. Semantic correspondence as an optimal transport problem. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020.

Andreas Loukas. What graph neural networks cannot learn: depth vs width. In International Conference on Learning Representations (ICLR), 2020.

Ana Lucic, Maartje ter Hoeve, Gabriele Tolomei, Maarten de Rijke, and Fabrizio Silvestri. Cf-gnnexplainer: Counterfactual explanations for graph neural networks. ArXiv, 2021.

Scott M. Lundberg, Gabriel G. Erion, and Su-In Lee. Consistent Individualized Feature Attribution for Tree Ensembles. ArXiv, 2018.

Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. In Conference on Neural Information Processing Systems (NeurIPS), 2020.

Ge Lv and Lei Chen. On data-aware global explainability of graph neural networks. Proceedings of the VLDB Endowment, 2023.

Jing Ma, Ruocheng Guo, Saumitra Mishra, Aidong Zhang, and Jundong Li. Clear: Generative counterfactual explanations on graphs. Advances in Neural Information Processing Systems, 2022.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In International Conference on Learning Representations (ICLR), 2016.

Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. In International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia, 2020.

Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. In Conference on Neural Information Processing Systems (NeurIPS), 2019.

Karolis Martinkus, Pál András Papp, Benedikt Schesch, and Roger Wattenhofer. Agent-based graph neural networks. arXiv preprint arXiv:2206.11010, 2022.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In AAAI Conference on Artificial Intelligence (AAAI), 2019.

Christopher Morris, Nils M Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. arXiv preprint arXiv:2007.08663, 2020.

Christopher Morris, Floris Geerts, Jan Tönshoff, and Martin Grohe. Wl meet vc. arXiv preprint arXiv:2301.11039, 2023.

Peter Müller, Lukas Faber, Karolis Martinkus, and Roger Wattenhofer. Graphchef: Learning the recipe of your dataset. In ICML 3rd Workshop on Interpretable Machine Learning in Healthcare (IMLH), 2023.

Vu Nguyen, Tam Le, Makoto Yamada, and Michael A Osborne. Optimal transport kernels for sequential and parallel neural architecture search. In International Conference on Machine Learning, 2021.

Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. Matching node embeddings for graph similarity. In Conference on Artificial Intelligence (AAAI), San Francisco, USA, February 2017.

Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. In International Conference on Learning Representations (ICLR), 2020.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, 1999.

Pál András Papp, Karolis Martinkus, Lukas Faber, and Roger Wattenhofer. Dropgnn: random dropouts increase the expressiveness of graph neural networks. In Conference on Neural Information Processing Systems (NeurIPS), 2021.

David Peleg. Distributed computing: a locality-sensitive approach. SIAM, 2000.

Phillip E Pope, Soheil Kolouri, Mohammad Rostami, Charles E Martin, and Heiko Hoffmann. Explainability methods for graph convolutional neural networks. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2019.

J.R. Quinlan. Simplifying decision trees. International Journal of Man-Machine Studies, 1987.

Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In International Conference on Learning Representations (ICLR), San Juan, Puerto Rico, 2016.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2016.

Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In International Conference on Learning Representations (ICLR), 2020.

Halsey Lawrence Royden and Patrick Fitzpatrick. Real analysis. Macmillan New York, 1988.

Benjamin Sanchez-Lengeling, Jennifer Wei, Brian Lee, Emily Reif, Peter Wang, Wesley Wei Qian, Kevin McCloskey, Lucy Colwell, and Alexander Wiltschko. Evaluating attribution for graph neural networks. Proceedings of 34th Conference on Neural Information Processing Systems, 33, 2020.

Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. IEEE transactions on systems, man, and cybernetics, 1983.

Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. SIAM Journal on Computing, 2012.

Ryoma Sato. A survey on the expressive power of graph neural networks. arXiv preprint arXiv:2003.04078, 2020.

Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Approximation ratios of graph neural networks for combinatorial problems. Advances in Neural Information Processing Systems, 32, 2019.

Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Fast unbalanced optimal transport on a tree. Advances in neural information processing systems, 2020.

Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Random features strengthen graph neural networks. In SIAM International Conference on Data Mining (SDM), 2021.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In International Conference on Learning Representations (ICLR), New Orleans, USA, 2019.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. IEEE transactions on neural networks, 2008.

Nina Schaaf, Marco F. Huber, and Johannes Maucher. Enhancing Decision Tree based Interpretation of Deep Neural Networks through L1-Orthogonal Regularization. 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), 2019.

Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. Interpreting graph neural networks for NLP with differentiable edge masking. In International Conference on Learning Representations (ICLR), 2021.

Daniel Schlör, Markus Ring, and Andreas Hotho. inalu: Improved neural arithmetic logic unit. arXiv preprint arXiv:2003.07629, 2020.

Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In IEEE International Conference on Computer Vision (CVPR), 2017a.

Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In Proceedings of the IEEE international conference on computer vision, 2017b.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. AI magazine, 2008.

L. S. Shapley. 17. A Value for n-Person Games. Contributions to the Theory of Games, 1953.

Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. arXiv preprint arXiv:1811.05868, 2018.

Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. arXiv preprint arXiv:1704.02685, 2017.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In International Conference on Machine Learning (ICML), Sydney, Australia, August 2017.

Mirac Suzgun, Yonatan Belinkov, and Stuart M Shieber. On evaluating the generalization of lstm models in formal languages. Society for Computation in Linguistics (SCiL), New York City, USA, 2019a.

Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov12, and Stuart M Shieber. Lstm networks can perform dynamic counting. In Association for Computational Linguistics (ACL), Florence, Italy, 2019b.

Hao Tang, Zhiao Huang, Jiayuan Gu, Bao-Liang Lu, and Hao Su. Towards scale-invariant graph-related problem solving by iterative homogeneous gnns. Conference on Neural Information Processing Systems (NeurIPS), 2020.

Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M Bronstein. Understanding over-squashing and bottlenecks on graphs via curvature. In International Conference on Learning Representations (ICLR), 2022.

Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In Neural Information Processing Systems (NeurIPS), Montreal, Canada, 2018.

Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. Verse: Versatile graph embeddings from similarity measures. In Proceedings of the 2018 world wide web conference, 2018.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In International Conference on Learning Representations (ICLR), 2018.

Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, 2020.

Clement Vignac, Andreas Loukas, and Pascal Frossard. Building powerful and equivariant graph neural networks with structural message-passing. In Conference on Neural Information Processing Systems (NeurIPS), 2020.

Minh Vu and My T Thai. Pgm-explainer: Probabilistic graphical model explanations for graph neural networks. In Conference on Neural Information Processing Systems (NeurIPS), 2020.

Qing Wang, Dillon Ze Chen, Asiri Wijesinghe, Shouheng Li, and Muhammad Farhan. N-WL: A New Hierarchy of Expressivity for Graph Neural Networks. In The Eleventh International Conference on Learning Representations, ICLR, Kigali, Rwanda, May 2023.

Runzhong Wang, Junchi Yan, and Xiaokang Yang. Learning combinatorial embedding networks for deep graph matching. In International Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, USA, June 2019.

Xiaoqi Wang and Han-Wei Shen. Gnninterpreter: A probabilistic generative model-level explanation for graph neural networks. arXiv preprint arXiv:2209.07924, 2022.

Roger Wattenhofer. Mastering Distributed Algorithms. Inverted Forest Publishing, 2020.

Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. nti, Series, 1968.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. In Association for Computational Linguistics (ACL), Florence, Italy, 2018.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In International Conference on Learning Representations (ICLR), San Diego, USA, 2015.

Asiri Wijesinghe and Qing Wang. A new perspective on" how graph neural networks go beyond weisfeiler-lehman?". In International Conference on Learning Representations, 2021.

Fang Wu, Siyuan Li, Xurui Jin, Yinghui Jiang, Dragomir Radev, Zhangming Niu, and Stan Z Li. Rethinking explaining graph neural networks via nonparametric subgraph matching. In International Conference on Machine Learning, 2023.

Mike Wu, Michael C. Hughes, Sonali Parbhoo, Maurizio Zazzi, Volker Roth, and Finale Doshi-Velez. Beyond Sparsity: Tree Regularization of Deep Models for Interpretability. In AAAI Conference on Artificial Intelligence (AAAI), 2017a.

Zhanghao Wu, Paras Jain, Matthew Wright, Azalia Mirhoseini, Joseph E Gonzalez, and Ion Stoica. Representing long-range context for graph neural networks with global attention. In Conference on Neural Information Processing Systems (NeurIPS), 2021.

Zhenqin Wu, Bharath Ramsundar, Evan N. Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Karl Leswing, and Vijay Pande. MoleculeNet: A Benchmark for Molecular Machine Learning. Chemical Science, 2017b.

Wenwen Xia, Mincai Lai, Caihua Shan, Yao Zhang, Xinnan Dai, Xiang Li, and Dongsheng Li. Explaining temporal graph models through an explorer-navigator framework. In The Eleventh International Conference on Learning Representations (ICLR), 2023.

Hongteng Xu, Dixin Luo, and Lawrence Carin. Scalable gromov-wasserstein learning for graph partitioning and matching. In Conference on Neural Information Processing Systems (NeurIPS), Vancouver, Canada, December 2019a.

Hongteng Xu, Dixin Luo, Hongyuan Zha, and Lawrence Carin Duke. Gromov-wasserstein learning for graph matching and node embedding. In International conference on machine learning (ICML), Long Beach, USA, June 2019b.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In International Conference on Machine Learning (ICML), 2018.

Keyulu Xu, Stefanie Jegelka, Weihua Hu, and Jure Leskovec. How Powerful are Graph Neural Networks? International Conference on Learning Representations (ICLR), 2019c.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In International Conference on Learning Representations(ICLR), Addis Ababa, Ethiopia, 2020.

Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In International Conference on Learning Representations (ICLR), remote, 2021.

Mengjiao Yang and Been Kim. Benchmarking attribution methods with relative feature importance. arXiv, 2019.

Yongxin Yang, Irene Garcia Morillo, and Timothy M. Hospedales. Deep Neural Decision Trees. ArXiv, 2018a.

Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In International conference on machine learning (ICML), 2018b.

Ziyuan Ye, Rihan Huang, Qilin Wu, and Quanying Liu. Same: Uncovering gnn black box with structure-aware shapley-based multipiece explanations. In Thirty-seventh Conference on Neural Information Processing Systems, 2023.

Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In Conference on Neural Information Processing Systems (NeurIPS), Vancouver, Canada, December 2019.

Jiaxuan You, Zhitao Ying, and Jure Leskovec. Design space for graph neural networks. In Conference on Neural Information Processing Systems (NeurIPS), virtual, December 2020.

Hao Yuan, Jiliang Tang, Xia Hu, and Shuiwang Ji. Xgnn: Towards model-level explanations of graph neural networks. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 430–438, 2020a.

Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. Explainability in graph neural networks: A taxonomic survey, 2020b.

Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. On Explainability of Graph Neural Networks via Subgraph Explorations. In International Conference on Machine Learning (ICML), 2021.

Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines-revised. arXiv preprint arXiv:1505.00521, 2016.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In International Conference on Machine Learning (ICML), New York City, USA, 2016.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In European Conference on Computer Vision (ECCV), Zurich, Switzerland, September 2014.

Bohang Zhang, Shengjie Luo, Liwei Wang, and Di He. Rethinking the expressive power of gnns via graph biconnectivity. In The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023, 2023.

Zaixi Zhang, Qi Liu, Hao Wang, Chengqiang Lu, and Cheekong Lee. ProtGNN: Towards Self-Explaining Graph Neural Networks. In AAAI Conference on Artificial Intelligence (AAAI), 2021.

Zhen Zhang and Wee Sun Lee. Deep graphical feature learning for the feature matching problem. In International Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, USA, June 2019.

Lingxiao Zhao and Leman Akoglu. Pairnorm: Tackling oversmoothing in gnns. In International Conference on Learning Representations (ICLR), 2020.

Kaixiong Zhou, Xiao Huang, Yuening Li, Daochen Zha, Rui Chen, and Xia Hu. Towards deeper graph neural networks with differentiable group normalization. In Conference on Neural Information Processing Systems (NeurIPS), 2020.

Jiong Zhu, Yujun Yan, Lingxiao Zhao, Mark Heimann, Leman Akoglu, and Danai Koutra. Beyond homophily in graph neural networks: Current limitations and effective designs. Advances in Neural Information Processing Systems, 2020.

Daniel Zügner and Stephan Günnemann. Adversarial attacks on graph neural networks via meta learning. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019.

Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial attacks on neural networks for graph data. In Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining, 2018.