# Towards Formal Verification of Graph Neural Networks

## CPSC513-FV Project Report

**Roozmehr Jalilian Hassanpour**

**Instructor:** Prof. Alan Hu

**University of British Columbia**

**December 2024**

# Abstract

Despite being among the most successful Artificial Intelligence (AI) technologies, Neural Networks (NNs) mostly have a "black box" nature, making their decision-making process difficult to trust, especially for safety-critical applications, requiring us to come up with ways to formally verify them.

Many different architectures for NNs have been proposed over the years, one of which being Graph Neural Networks (GNNs). These peculiar networks can process graph-structured input data with arbitrary size which, while making them exceptionally powerful in generalizing to unseen inputs, complicates the process of their formal verification to a great degree.

In this work, we present a framework to formally verify a subtype of message-passing-based GNNs, called graph convolution networks (GCNs), by first converting them into feed-forward NNs based on their input graphs, and then using a Satisfiability Modulo Theory (SMT) solver to perform verification. We then evaluate the performance of this framework for different input graph sizes to assess the scalability of our proposed approach.

# Table of Contents

# List of Figures

# List of Tables

# 1 Introduction

Formal verification of neural networks has been gaining interest recently due to how critical it is to verify their decision-making process, especially for safety-critical applications in fields such as medical, automotive, and aerospace. In fact, the International Verification of Neural Networks Competition (VNN-COMP) has been established since 2021 for this specific reason, where contestants submit their verification tools to be evaluated across multiple NN architectures [1].

## 1.1 Graph Neural Networks

One of the most widely used neural network architectures is GNN, capable of processing graph-structured input data of arbitrary size. There are multiple subtypes of these networks, one of which is based on the concept of *message passing*. These networks operate by aggregating features across neighboring nodes and edges for each node in the graph, which can be interpreted as passing message across different nodes.

In general, message passing in graphs can be performed in three ways: basic weighted summation, graph convolution, and graph attention. GNNs which aggregate features based on these methods are called Message-Passing Neural Networks (MPNNs) [2], Graph Convolutional Networks (GCNs) [3], and Graph Attention Networks (GATs) [4], respectively. The focus of our approach is on GCNs, but the methodology can easily be extended to other architectures as well.

### 1.1.1 Graph Convolution Networks

The node aggregation operation in a GCN, depicted in Figure 1, can be summarized in the formula:

$$h_i = \sigma\left(\sum_{j\in\mathcal{N}_i} c_{ij}\mathbf{W}h_j\right), \qquad c_{ij} = \frac{1}{\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}} \tag{1}$$

Where $h_i$ is the feature vector of the node $i$, $\mathcal{N}_i$ is the set of neighboring nodes of node $i$, $c_{ij}$ is the convolution coefficient between nodes $i$ and $j$, $\mathbf{W}$ is the learnable weight matrix, and $\sigma$ is an activation function.

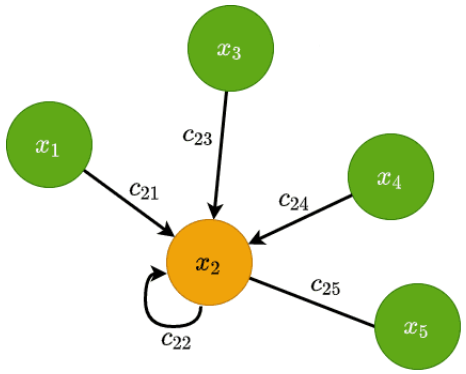


Figure 1: An example of node aggregation in GCNs [5].

## 1.2 Formally Verifying Neural Networks

Verifying a neural network is generally done in three ways:

- Encoding the NN into a set of Mixed-Integer Linear Programming (MILP) or SMT formulas and using domain-specific algorithms
- Treating the NN as non-linear functions and using global optimization techniques
- Using abstract representations to certify the NN

There are usually two types of properties to verify in a neural network: Adversarial Robustness (ARP) and Output Reachability (ORP). A property usually has the form of a triple, $P(N, \phi, \psi)$, where $N$ is the neural network, $\phi$ is some input specification, and $\psi$ is some output specification. We denote $\Phi$ and $\Psi$ as sets of input and output specs,

respectively. Using this notation, the two types of properties mentioned earlier can be defined as follows:

$$\text{ARP}(\Phi, \Psi) \text{ is true iff } \forall I \vDash \Phi \colon N(I) \vDash \Psi \tag{2}$$

$$\text{ORP}(\Phi, \Psi) \text{ is true iff } \exists I \vDash \Phi \colon N(I) \vDash \Psi \tag{3}$$

Where $I$ is an input to the network, and the operator $\vDash$ denotes conformation (i.e., the term on the lefthand side conforms to the set of specs on the righthand side). In the context of GNNs, either property can be defined based on the type of network inference (e.g., for a node-classifier network, ARP would be denoted as $\text{ARP}_{\text{node}}$).

# 2 Related Work

At the time of writing, few works in prior literature have tried to formally verify GNNs, and understandably so. Given the fact that these networks usually don't impose any restrictions on the number of nodes and edges of their input graphs, it is extremely challenging to design a general and scalable verification framework for them.

In fact, in a study by M. Sälzer *et al.* it was proven that for MPNNs, both $\text{ARP}_{\text{graph}}(\Phi_{\text{unb}}, \Psi_{\text{class}})$ and $\text{ORP}_{\text{graph}}(\Phi_{\text{unb}}, \Psi_{\text{eq}})$ properties, were $\Phi_{\text{unb}}$ is a set of graph specs allowing for unbounded, but finite size, nontrivial degree and sufficiently expressive labels, are not formally verifiable [6]. Note that $\Psi_{\text{eq}}$ is a set of vector specs able to check if a certain element of a vector is equal to some fixed integer, and $\Psi_{\text{class}}$ is a set of vector specs which can check if a certain element is greater than all others (i.e., checking for exact class assignment).

In another work by T. Ladner *et al.*, a formal verification approach for GCNs with uncertain node features and uncertain graph structures using polynomial zonotopes was proposed [7]. This approach falls into the third category of NN verification (i.e., using

abstract representations), and despite being much more efficient than graph enumeration, it is shown that verification runtime still grows exponentially with respect to the number of edges in the input graph.

# 3 Methodology

Based on the study done by M. Sälzer *et al.*, we know that if bounded input specs are used instead of unbounded ones, both the ARP and ORP properties for node-classifier MPNNs become verifiable [6]. This motivated us to design a verification framework for node-classifier GCNs, which is shown in Figure 2. The implemented framework is publicly available for use under an Apache 2.0 license at [8].

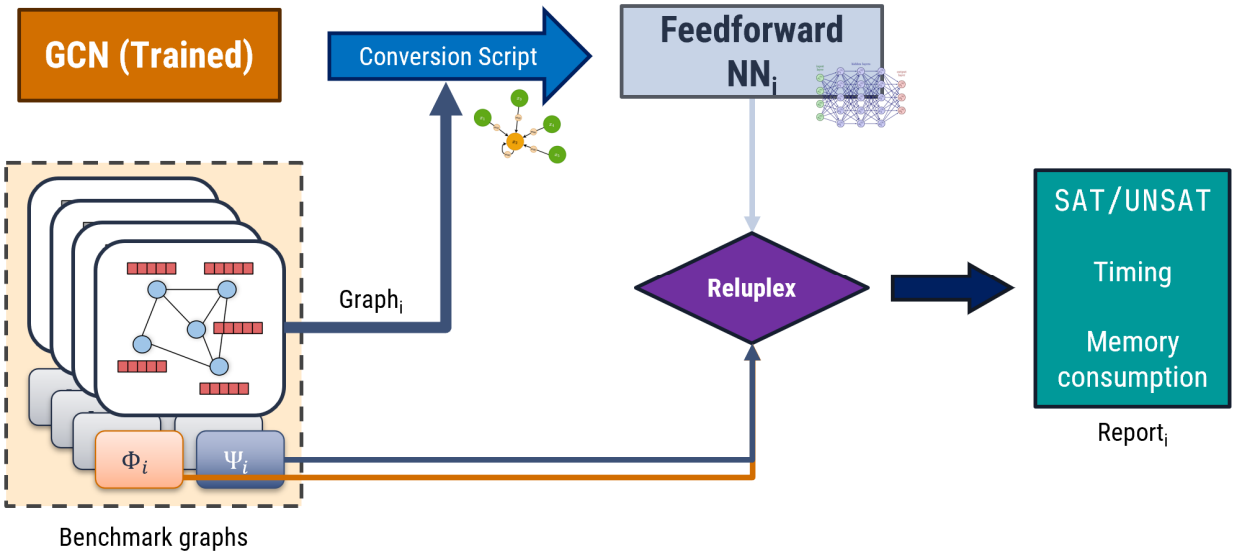


Figure 2: Overview of the implemented GCN verification framework.

In this framework, a feed-forward NN (FFNN) specific to each input graph is created based on the GCN we're trying to verify and the input graph structure. This FFNN is then fed into *Reluplex* [9], an SMT-based verifier for neural networks, and the verification

results are recorded. We used an upgraded version of *Reluplex* in our implementation, *Marabou* [10], due to it providing a convenient Python API.

The key component of this framework is the FFNN generation algorithm. Similar to Equation (1), for a directed graph $\mathcal{G} = \{V, E\}$ where $V$ is the set of nodes and $E$ is the set of edges, the node-wise operation commonly used in graph convolution layers is as follows:

$$\mathrm{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}_i \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathrm{x}_j \tag{4}$$

Where $\mathrm{x}_j \in \mathbb{R}^{F_N}$ is the input feature vector of node $j$, $\mathrm{x}'_i \in \mathbb{R}^{F_H}$ is the output feature vector of node $i$, $\mathbf{\Theta} \in \mathbb{R}^{F_N} \times \mathbb{R}^{F_H}$ is the learnable weight matrix, $e_{j,i} \in \mathbb{R}$ denotes the edge weight from source node $j$ to target node $i$, $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}_i} e_{j,i}$ is the degree of node $i$ with inserted self-loops, and $F_N$ and $F_H$ are the number of input and output node features, respectively.

The key idea here is to break down this operation into two parts: matrix multiplication (shown in blue) and node aggregation (shown in red). The former is graph-independent and transforms the features of each node, and the latter is feature-independent and depends on graph structure.

Consider a simple graph with 2 edges and 3 nodes, with each node having 2 features, depicted in Figure 3. In order to convert the operations in Equation (4) into a set of purely linear transformations similar to that of FFNNs, we first break down the 3 nodes into a vector which has $3 \times 2 = 6$ elements, represented as $x_i \in \mathbb{R}^{(|V|F_N)}$. Each element will act as an input entry to our FFNN.

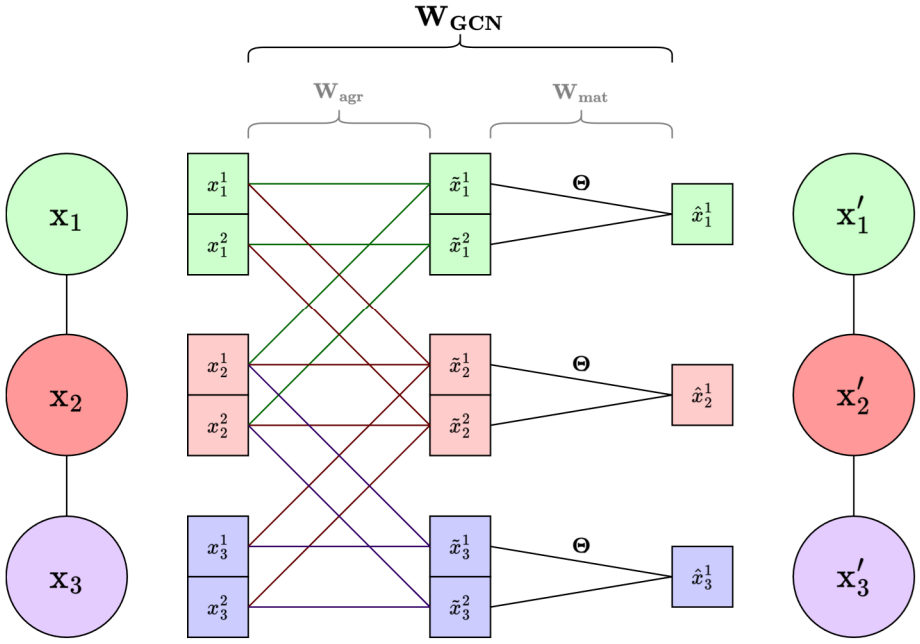


Figure 3: Conversion example of a graph convolution into a linear transform.

Next, we perform node aggregation by computing the coefficients in the red component of Equation (4), combining them into matrix form, $\mathbf{W_{agr}} \in \mathbb{R}^{(|V|F_N)} \times \mathbb{R}^{(|V|F_N)}$, and performing multiplication across the input vector, yielding aggregated node feature vector $\tilde{x}_i \in \mathbb{R}^{(|V|F_N)}$.

Then, we apply the blue component of the equation by concatenating the weight matrix $\boldsymbol{\Theta}$ as many times as the number of nodes in the graph, which can be combined into matrix form, $\mathbf{W_{mat}} \in \mathbb{R}^{(|V|F_N)} \times \mathbb{R}^{(|V|F_H)}$, and performing multiplication across the aggregated vector $\tilde{x}_i$, yielding the output feature vector $\hat{x}_i$. Finally, in order to represent the entire process as a single linear transformation, we compute the matrix $\mathbf{W_{GCN}} = \mathbf{W_{agr}} \times \mathbf{W_{mat}} \in \mathbb{R}^{(|V|F_N)} \times \mathbb{R}^{(|V|F_H)}$, which acts as the weight matrix of the feed-forward (FF) equivalent layer of the graph convolution layer. Note that bias vectors have been omitted from these calculations but can be easily obtained by embedding them inside the weight matrices.

# 4 Experiments and Results

**Disclaimer:** All experiments were conducted on a machine running Ubuntu 22.04 LTS, equipped with an AMD Ryzen 7840HS processor with 32 gigabytes of memory.

To test our framework, we first designed and trained a GCN using the *PyTorch* library on the *KarateClub* dataset [11], which consists of one graph with 34 nodes and 156 edges, with each node having 34 features. Each node in the graph is assigned a label, which is an integer belonging to the set {0, 1, 2, 3}. The task of the GCN is to predict this label for each node in the input graph, which can be any subgraph of the original 34-node graph.

The designed GCN consists of 2 graph convolution layers and 1 ReLU layer in-between, with a Softmax layer at the end for classification. The first convolution layer has a dimension of $(F_N = 34, F_H = 8)$, and the second one has a dimension of $(F_H = 8, F_O = 4)$. Note that we won't include the Softmax layer in our feed-forward conversions since, unlike ReLU, it isn't piecewise-linear.

We chose a set of 10 subgraphs from the original graph to act as inputs to our framework, a description of which is provided in

Table 1. Here $D_{\text{in}}$ and $D_{\text{out}}$ denote the input and output sizes, and $P_{\text{GCN}}$ and $P_{\text{FF}}$ denote the number of parameters in the GCN and FF-equivalent networks, and $T_{\text{convert}}$ denotes the GCN-to-FF network conversion time (in seconds).

Table 1: Specifications of the input graphs and their corresponding FF-equivalent networks.

| **Graph #** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| $\|V\|$ | 3 | 4 | 5 | 6 | 7 | 12 | 12 | 13 | 29 | 34 |
| $\|E\|$ | 4 | 8 | 12 | 20 | 18 | 38 | 48 | 50 | 136 | 156 |
| **Original GCN network properties** | | | | | | | | | | |
| $D_{\text{in}}$ | $\|V\|$ & $\|E\|$ = input-dependent, $F_N = 34$ | | | | | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $D_{\text{out}}$ | $\lvert V\rvert$ & $\lvert E\rvert$ = input-dependent, $F_O = 4$ | | | | | | | | | |
| $P_{\text{GCN}}$ | 316 | | | | | | | | | |
| **FF-equivalent network properties** | | | | | | | | | | |
| $D_{\text{in}}$ | 102 | 136 | 170 | 204 | 238 | 408 | 408 | 442 | 986 | 1156 |
| $D_{\text{out}}$ | 12 | 16 | 20 | 24 | 28 | 48 | 48 | 52 | 116 | 136 |
| $P_{\text{FF}}$ | 2,772 | 4,912 | 7,660 | 11,016 | 14,980 | 43,920 | 43,920 | 51,532 | 256,012 | 351,832 |
| $T_{\text{convert}}$ (s) | 0.30 | 0.33 | 0.31 | 0.34 | 0.34 | 0.39 | 0.42 | 0.43 | 0.98 | 1.18 |

As we can see, the number of parameters in the FF-equivalent network is of order $\mathcal{O}(|V|^2)$, assuming the feature dimensions of the GCN network remain constant. After conversion, the FF-equivalent network is exported in .nnet format and then fed into *Marabou*. Note that all inputs need to be normalized to zero-mean, one-range values before being fed into *Marabou*. Thus, it is necessary to also specify the min, max, mean, and range values for all inputs in the .nnet file.

In this work, we excluded the Softmax layer from the conversion process, and instead tried to verify if all the outputs of the second-to-last layer (i.e., the second graph convolution layer) fall within certain bounds. Specifically, we set bounds of the form $(-2^b, 2^b - 1)$, where $b$ is an integer ranging from 1 to 13. This represents a real-world scenario in which we'd like to implement our GCN on resource-limited hardware, and the goal is to use the least number of bits required (i.e., $B_{\text{min}} = b_{\text{min}} + 1$; with the extra bit accounting for sign) to store the integer portion of the values at the output of second-to-last layer, without the risk of overflow or underflow, over all possible input combinations.

To verify these properties, we perform two queries per output variable: one for the lower bound (i.e., $-2^b$), and one for the upper bound (i.e., $2^b - 1$). If, for a given value of $b$, one of the queries returns SAT, this means at least one variable would fall out of that bound, implying that the desired property does not hold. If, and only if, all queries for every output variable return UNSAT, the desired property holds.

The results of this verification procedure are provided in Table 2. Here $B_{\text{min}}$ denotes the least number of bits required to store the output values of the FF-equivalent network without overflow/underflow, $T_{\text{SAT}}$ @ $b$=1 denotes the earliest time it takes for Marabou to return SAT for a query when $b$ is set to one, $T_{\text{UNSAT}}$ denotes the time it takes for Marabou to return UNSAT for the query corresponding to the bound $(-2^{b_{\text{min}}}, 2^{b_{\text{min}}} - 1)$, in seconds, and $M_{\text{peak}}$ denotes the peak memory usage of the verification program (in MiB).

Table 2: Summary of the verification results.

| **Graph #** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| $\|V\|$ | 3 | 4 | 5 | 6 | 7 | 12 | 12 | 13 | 29 | 34 |
| $\|E\|$ | 4 | 8 | 12 | 20 | 18 | 38 | 48 | 50 | 136 | 156 |
| $B_{\text{min}}$ | 11 | 12* | 12 | 13* | 13 | 15 | 15 | 15 | 17 | 17* |
| $T_{\text{SAT}}$ @ $b$=1 (s) | 0.01 | 0.01 | 0.02 | 0.05 | 0.09 | 0.65 | 0.64 | 0.83 | 19.47 | 38.03 |
| $T_{\text{UNSAT}}$ (s) | 0.02 | -- | 0.93 | -- | 0.32 | 1.11 | 1.08 | 1.35 | 16.01 | -- |
| $M_{\text{peak}}$ (MiB) | 564.79 | 566.41 | 568.36 | 571.55 | 581.97 | 605.23 | 602.86 | 609.15 | 779.37 | 858.50 |

Note that entries in the table marked with an asterisk are based on speculation, since the verification program returned UNSAT on some of the output variables but hanged on verifying the remaining queries. Based on our experiments, if one were to do an exhaustive search to find the value of $B_{\text{min}}$, it would take around 5 minutes when the input graph has 34 nodes, compared to around 24 milliseconds for a graph with 5 nodes. This shows the runtime of this verification-based exhaustive search for optimum bit-width scales exponentially with respect to input graph size.

# 5 Conclusion and Future Work

In this work, we proposed a framework to formally verify graph convolutional networks, by first breaking the convolution operation into aggregation and matrix multiplication

components, converting these components into linear transformations, mapping the original network to an equivalent FF neural network based on the structure of the input graph, and feeding the derived FF network into an SMT solver. We then studied how well this framework would scale with input graph size, and performed a formal-verification-based exhaustive search to find the smallest possible bound which the outputs of the network would fall within. We observed that the execution time of this approach scales exponentially with respect to number of nodes of the input graphs.

In the future, we'd like to optimize the derivation of the FF-equivalent weight matrix by leveraging the sparsity of the matrices involved in its computation, implement a more "intelligent" search algorithm that uses information obtained from prior input graphs, and also extend our framework to support other types of message-passing-based GNNs.

# References

[1] "5th International Verification of Neural Networks Competition (VNN-COMP'24)," 10 October 2024. [Online]. Available: https://sites.google.com/view/vnn2024. [Accessed 21 December 2024].

[2] J. Gilmer et al., "Neural Message Passing for Quantum Chemistry," in *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[3] T. N. Kipf et al., "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations*, 2017.

[4] P. Veličković et al., "Graph Attention Networks," in *International Conference on Learning Representations*, 2018.

[5] S. Karagiannakos, "Best Graph Neural Network architectures: GCN, GAT, MPNN and more," AI Summer, 23 September 2021. [Online]. Available: https://theaisummer.com/gnn-architectures/. [Accessed 21 December 2024].

[6] M. Sälzer et al., "Fundamental Limits in Formal Verification of Message-Passing Neural Networks," in *The Eleventh International Conference on Learning Representations*, 2023.

[7] T. Ladner et al., "Formal Verification of Graph Convolutional Networks with Uncertain Node Features and Uncertain Graph Structure," in *arXiv:2404.15065v1 [cs.LG]*, 2024.

[8] R. Jalilian, "Towards Formal Verification of GNNs," GitHub repository, 2024. [Online]. Available: https://github.com/roozmehrjh79/CPSC513_FV4GNN.

[9] G. Katz et al., "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," *CAV: International Conference on Computer Aided Verification,* pp. 97-117, 13 July 2017.

[10] H. Wu et al., "Marabou 2.0: A Versatile Formal Analyzer of Neural Networks," *36th International Conference on Computer Aided Verification,* pp. 249-264, 25 July 2024.

[11] "KarateClub," PyG Documentation, [Online]. Available: https://pytorch-geometric.readthedocs.io/en/2.5.3/generated/torch_geometric.datasets.KarateClub.html. [Accessed 21 December 2024].

[12] "conv.GCNConv," PyG Documentation, [Online]. Available: https://pytorch-geometric.readthedocs.io/en/2.5.2/generated/torch_geometric.nn.conv.GCNConv.html. [Accessed 21 December 2024].