Electrical and Computer Engineering
University of Thessaly (UTH)

ECE434 - Comlpex Networks

Fall Semester — Educational year 2024-2025

Complex Networks Semester Project

Spyridon Petroglou - AEM: 03185

Vasileios Stergioulis - AEM: 03166

Emmanouil Pantopoulos - AEM: 03222

Abstract

In this project, we will present our work on the node classification problem using Graph Neural Networks (GNNs). We trained four different variations of GNNs on the Cora dataset. From these four networks, two of them were convolutional-based and the remaining two were attention-based. The best of these models was the simple Graph Attention Network (GAN), which yielded an accuracy of 0.8. In addition, to further improve the accuracy, we experimented with some different ensemble methods. The highest accuracy was achieved using the weighted averaging method, where the optimal weights were found by applying the grid search method.

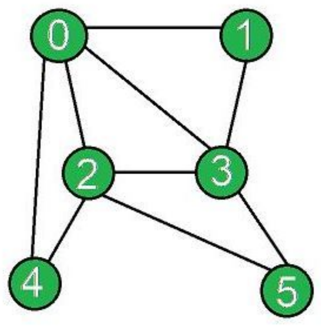
Key Phrases: Graph Neural Network, Graph Convolution Network, Graph Attention Network, Node Classification

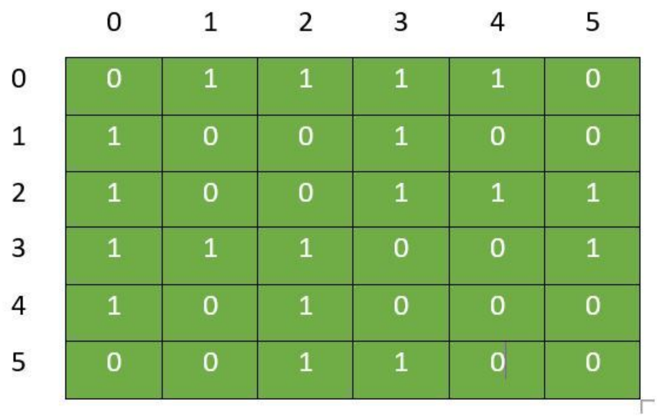
Contents

1 Introduction

What does Instagram have in common with urban transportation maps and proteins? At first glance, these concepts may seem quite irrelevant, but if you think carefully, you will realize that in each case, you have information about entities and about the relationship among these entities. On social media, each account can be represented as an entity and if two accounts are following one another, we can connect these entities with a straight line. We call these entities nodes and the connections edges. In transportation maps, the nodes are the bus stops and the edges are the roads among the stops, whereas in proteins, the nodes represent amino acids or atoms, and edges represent interactions or distances between them.

In computer science, each structure that consists of nodes and edges is called a $\mathbf{Graph}$ and is represented by $\mathbf{G} = \{\mathbf{V}, \mathbf{E}\}$, where $\mathbf{V}$ is the number of vertices (nodes) and $\mathbf{E}$ is the number of edges. A graph is stored in a computer as a matrix, called the adjacency matrix. An adjacency matrix is a 2D matrix of size $\mathbf{V}$x$\mathbf{V}$, where each row and each column represent a node, and each index represents an edge. For unweighted graphs, the adjacency matrix is a binary matrix, where each 1 in (i, j)-place denotes a connection between the nodes i, j, while each 0 denotes that there is no connection between these two nodes.





	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

Figure 1: A simple graph along with its adjacency matrix

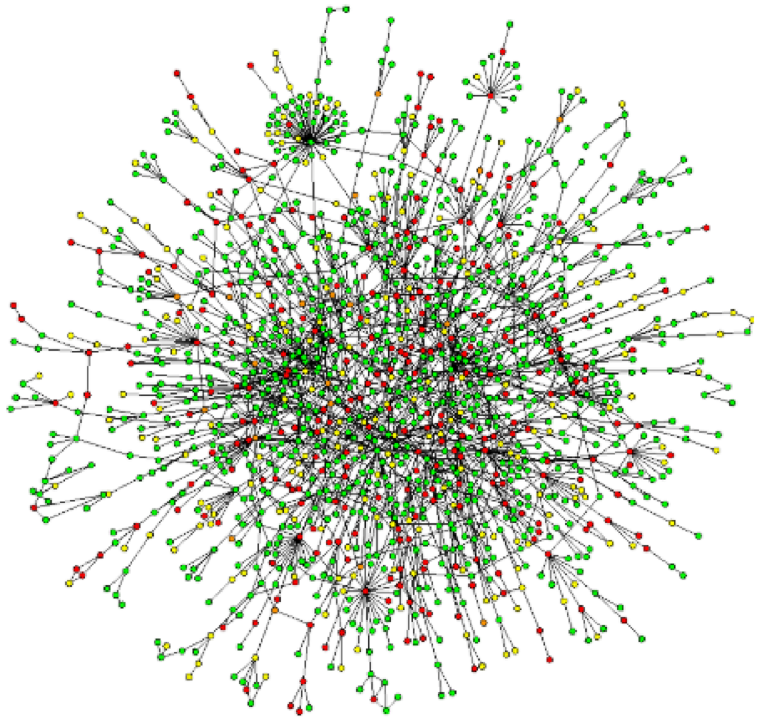


Figure 2: A complex real-world graph

In **Figures 1, 2** we can see some graph instances. We can easily notice that some graphs may be very simple while some others may be much more complicated. These complex graphs require more sophisticated analysis techniques.

From a geometrical point of view, a graph can be seen as a 2D array in the Euclidean space without an explicit shape. A comparison between a structured 2D grid and a graph is illustrated in **Figure 3**. Since each node can be connected with an arbitrary number of nodes, the result is a non-structured grid. Thus, a 2D array, or an image, can be seen as a specific type of graph, a graph where each node is connected only to their upper, lower, left and right nodes.

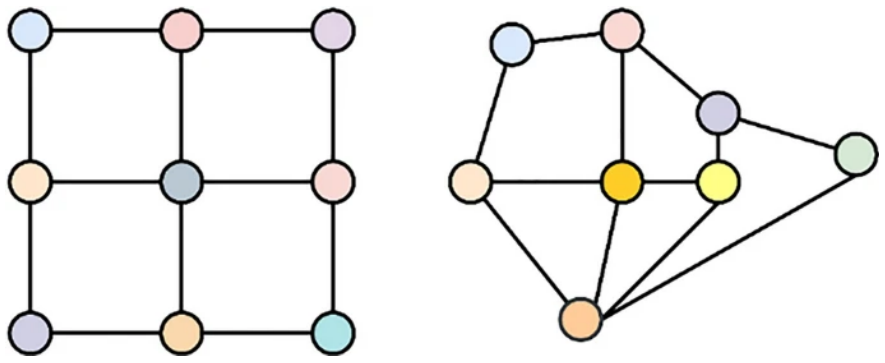


Figure 3: The topological difference between structured and non-structured grids (Image from [1])

This was the main idea behind the graph convolution operation. Since the classical convolution is applicable in grid-like structures, which is a specific type of graph, we can extend this concept to be applicable in non-grid structures too. To perform convolution in images, we use a fixed-size kernel and then, for a specific pixel, we update its value by multiplying the pixel values of its neighboring pixels with a kernel's weight and aggregating all these products. Essentially, we accumulate information for one specific pixel from its adjacent pixels. If the kernel's shape is 3x3 (this is the most usual kernel size), the pixel accumulates information only from its one-hop neighbors, if it is 5x5 it accumulates information from both one-hop and two-hop neighbors, and so on. As we can see in **Figure 4**, the main difference between traditional 2D convolution and graph convolution is that the kernel does not have a fixed size because each node may have a different number of one-hop neighbors.

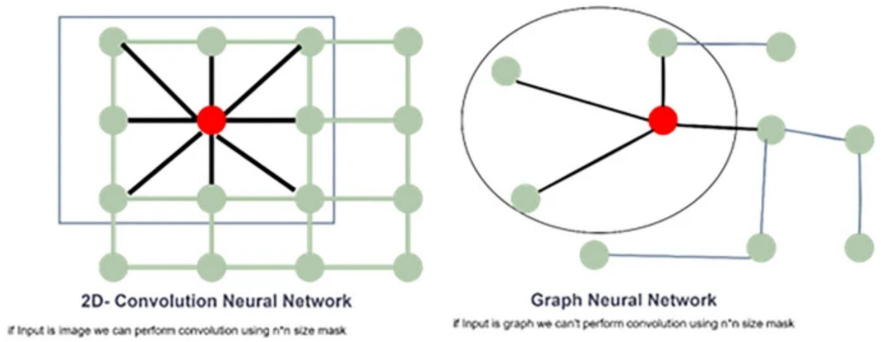

2D- Convolution Neural Network
If Input is image we can perform convolution using n*n size mask
Graph Neural Network
If Input is graph we can't perform convolution using n*n size mask

Figure 4: The difference between the standard 2D convolution and the graph convolution (Image from [1])

Apart from convolution, we can extend more concepts used in deep learning to operate on graph networks, such as attention mechanism, residual connections, recurrent layers and more. All models that use these generalized concepts and operate in graph data are called **Graph Neural Networks** , or GNNs in short, and are used to perform machine learning tasks in graphs. Some common tasks for GNNs are:

- Graph Classification: Assign a class label to each graph in a dataset consisting of many graphs.
- Node Classification: Assign a class label to each non-labeled node in a graph. In this case, the dataset consists of a single graph.
- Link Prediction: Find missing links in a graph.

For the sake of our project, we utilized GNNs to perform node classification in the Cora dataset. The rest of our report is organized as follows. In section **2** we analyze the motivation behind the development of the GNNs and in section **3** we describe the way we tackled the node classification task in our dataset.

2 Related Work

There are three important properties that distinguish graphs from any grid structure, such as images, and render traditional deep learning models incapable of handling and processing graph data. The first one is that not all graphs in a dataset are necessarily of the same size. This might also be true for image data, but in comparison to graphs, you can easily crop or reshape images to be the same size. The second one is that graphs are permutation invariant, which means that if you flip a graph in any direction, you still get the same graph. This is not true for image data. If you flip an image, you get a completely new image. This property is also called the isomorphism property of graphs. The last property is that graphs do not have a grid structure in Euclidean space. All of these properties led to the development of GNNs.

Before the development of GNNs, hand-crafted graph features were fed as input to neural networks to perform regression and classification tasks. However, this method was suboptimal since none of the aforementioned properties were taken into consideration. Graph Neural Networks were introduced in 2005 in [2], but were not widely used until 2017, when a variant of the classical convolution method was introduced in graph networks [3]. This type of neural network was named **Graph Convolutional Network** (GCN). A GCN consists of multiple layers, where each layer performs the following operation:

$$X^{(l+1)} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X^{(l)}W^{(l)}) \tag{1}$$

where:

$X^{(l)}$ are the node features at layer l,
$\hat{A} = A + I$ is the adjacency matrix with ones in the main diagonal (self loops),
$\hat{D}$ is the diagonal matrix of $\hat{A}$,
$W^{(l)}$ is the learnable weight matrix and
σ is the activation function.

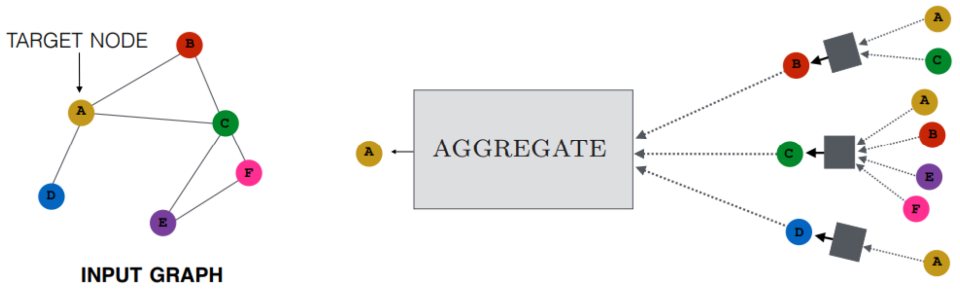


Figure 5: Illustration of Two-Layer Information Flow in a Simple Graph (Image from [4])

By applying a single GCN layer to our graph data, we allow our nodes to aggregate information from its one-hop nodes before updating their state. By stacking multiple layers one after the other, the nodes can gather information from more distant nodes before performing the update. Each GCN layer is also called a **Message Passing Layer** [4]. A simple two-layer message passing is illustrated in **Figure 5**. Note that node A updates its condition based on the information from its one-hop neighbors B, C and D, but these nodes have already aggregated information from their one-hop neighbors, including nodes E and F. Thus, node A will indirectly be affected from the information of both one-hop and two-hop neighbors. As we increase the number of message passing layers, we increase the range in which each node gathers information in the network.

The outputs of the Message Passing Layers are called **Embeddings** [5] (specifically node embeddings since we operate on graphs). An embedding is a low-dimensional representation of a node feature that contains useful information not only about itself but also about the nodes in its vicinity. The exact size of the embedding vector is a hyperparameter. An embedding update after a single message passing layer can be seen in **Figure 6**. Note that the embedding after the update contains information from all of its one-hop neighbors.

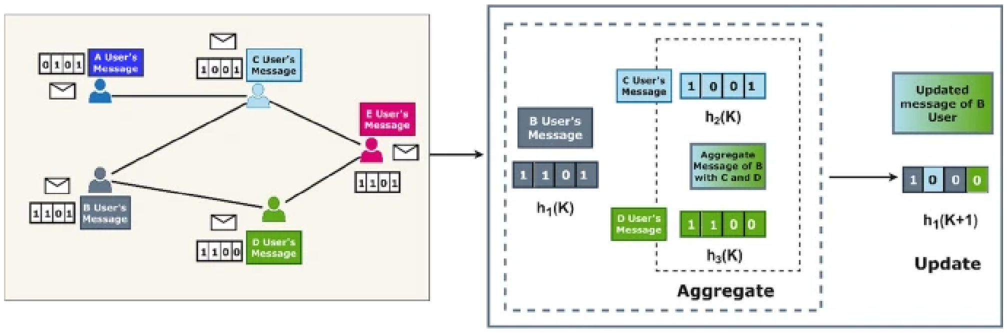


Figure 6: Processing of the Embedding Vectors (Image from [1])

Another common type of GNN is the **Graph Attention Network**, or GAT in short [6]. In comparison to GCN layers, a GAT layer uses the following operation for weight updating:

$$h_i' = \sigma(\sum_{j \in N(i)} a_{ij} W h_j) \tag{2}$$

where

$$a_{ij} = \frac{exp(LeakyReLU(a^T[Wh_i||Wh_j]))}{\sum_{k \in N} exp(LeakyReLU(a^T[Wh_i||Wh_k]))} \tag{3}$$

and
h_i is the feature of node i,
W is the shared learnable weight matrix,
a is the learnable vector for computing attention,
$||$ is the concatenation operation and
σ is the activation function.

The main difference between this method and the convolutional one is that this method assigns a learnable attention score to the neighbors in order to gather more information from more important neighbors. If we convert equation (1) to an element-wise form, we obtain:

$$x_i^{(l+1)} = \sigma\left(\sum_{j \in N(i) \cup \{i\}} \frac{1}{\sqrt{d_i d_j}} W^{(l)} x_j^{(l)}\right) \tag{4}$$

By removing the normalization factor $(\sqrt{d_i d_j})^{-1}$ and the self-loop, we obtain a simplified version of the equation:

$$x_i^{(l+1)} = \sigma(\sum_{j \in N(i)} W^{(l)} x_j^{(l)}) \tag{5}$$

or, equally:

$$h_i' = \sigma(\sum_{j \in N(i)} W h_j) \quad (6)$$

By comparing the equations (2) and (6), we can easily notice that the only difference among them is the attention coefficient a_{ij}. In a GCN layer, there is no coefficient, which means that a specific node treats all its neighbors equally, while in a GAT layer, the nodes with a higher attention coefficient are considered more influential. Thus, in that case, the network aggregates more information from these nodes before updating its condition.

Since both of these GNN architectures have the same backbone, we can generalize them and create a new generalized formula [4]:

$$h_i^{(k+1)} = UPDATE^{(k)}(h_i^{(k)}, AGGREGATE^{(k)}(\{h_j^{(k)}, \forall j \in N(i)\})) \quad (7)$$

In a node i, the $AGGREGATE$ function performs an aggregation operation (most common operations are sum, mean and max) on the embeddings of the adjacent nodes j, and then the $UPDATE$ function updates the node's embeddings. Most message passing GNNs are based on this general formulation. The only thing that changes among the different architectures is the $AGGREGATE$ and $UPDATE$ functions.

Due to the effectiveness of these architectures in graph data, several modifications of the simple GCN and GAT architectures have been proposed. For example, in [7], a new convolutional network is defined that, in addition to the node features, also includes edge features. In comparison to the node features that provide information about the nodes, edge features provide information about the relationship between two connected nodes. On the other hand, [8] transposes the order of the activation function and the linear transformation in the computation of the attention coefficients.

Lastly, apart from the GCN and GAT networks, there are more GNN architectures that obey the equation (7), but use different $AGGREGATE$ and $UPDATE$ functions, such as the Graph Sampling and Aggregate (GraphSAGE) [9] and the Graph Isomorphism Network (GIN) [10]. Furthermore, this article [11] provides a summary of all four different networks mentioned so far.

3 Proposed Method

3.1 Dataset

The dataset that we used for the node classification task was the Cora dataset. The Cora dataset is a citation network, which means that each node in the graph represents a scientific paper and each edge represents a citation between two papers (for example, if paper A cites paper B or vice versa, there is an edge among these two nodes). It consists of 2.708 nodes that are connected with 5.429 links.

The node attributes are represented as bag of words, meaning that there is a pre-defined dictionary of words. This dictionary includes 1.433 words in total and in each paper we count how many times a specific word appears. Also, each node has a target attribute that denotes the class that each paper belongs. There are 7 classes in total: *'Case Based'*, *'Genetic Algorithms'*, *'Neural Networks'*, *'Probabilistic Methods'*, *'Reinforcement Learning'*, *'Rule Learning'* and *'Theory'*.

The most common machine learning challenges associated with this dataset are node classification and link prediction. In the link prediction task, you try to predict the missing connections, given a subset of connections. But for this project, we used this dataset for node classification, where given a partial set of labeled nodes, we classified all the other nodes in the seven aforementioned classes. It should also be noted that only 140 of our nodes are labeled, which corresponds to a percentage of 5%.

3.2 Network Architecture

Our architecture is an ensemble of four GNN-based models. Two of these models are GCN-based whereas the remaining two are GAT-based. More specifically:

- The first model is a 2-layer GCN followed by a linear output layer. It takes as input the 1.433-dimensional input, it projects it into a 16-dimensional space, and then the linear layer is used to classify these vectors into one of the seven classes.
- The second model is a 2-layer GAT with 4 heads, followed by a linear output layer. Similar to the GCN, it processes the 1.433-dimensional node features and projects them into a 16-dimensional hidden space before the final 7-class classification layer.
- The third model is a 2-layer GCN-V2 which extends the traditional GCN by incorporating edge attributes. This model uses NNConv layers, which learn a weight matrix for each edge based on its attributes. The input, hidden, and output dimensions are consistent with the other models.
- The fourth and final model is a 2-layer GAT-V2, an attention-based network that also incorporates edge attributes. It uses GATv2Conv layers, allowing for dynamic attention weights that take into account both node features and edge attributes.

The main emphasis of our approach was to understand and leverage the strengths of different GNN architectures, particularly those that can handle edge attributes, and combine their predictions through and ensemble method. The final prediction is not derived from a single model, but rather from a combination of the raw output logits of these four base models. The ensemble strategy aims to improve the overall robustness and accuracy of the node classification task.

3.3 Experimental Setup

For our experiments, we used the Cora dataset as described in the previous sections. All models were implemented using the PyTorch and PyTorch Geometric libraries. The training process for each individual model was configured with the Adam optimizer. A learning rate of 0.005 and a weight decay of 5×10^{-4} were applied to regularize the models and prevent overfitting. The CrossEntropyLoss function was used as the optimization objective, focusing only on the labeled nodes within the training mask. Each model was trained for up to 1000 epochs. During the independent training for each of the four models (GCN, GAT, GCN-V2, GAT-V2) different attribute generation methods were used with the purpose of finding the most suitable for each model. The best performing checkpoint, meaning the model checkpoint with the lowest validation loss, was saved, ensuring that the best-performing iteration of each model was preserved for the ensemble phase. The Cora dataset comes with predefined training, validation, and test masks, which were used to split the labeled nodes for training and evaluating both individual models and ensemble methods.

3.4 Evaluation and Results

The primary evaluation metric for the node classification task was test accuracy, calculated as the ratio of the correctly predicted nodes to the total number of nodes in the test set. After training the individual models and selecting their best checkpoints, we proceeded to evaluate their performance on the test set and then apply various ensemble strategies. The best performing (those with the least amount of loss) individual model and the corresponding edge attribute types that were used for the V2 models are as follows:

- **GCN** 0.776 at epoch 974 with a loss of 0.0332 with feature concation
- **GAT** 0.800 at epoch 956 with a loss of 0.0229 with feature concation
- **GCN-V2** 0.791 at epoch 760 with a loss of 0.0082 with the absolute difference of features
- **GAT-V2** 0.799 at epoch 893 with a loss of 0.0174 with feature concation

The results are presented compactly in **Table 1** We then explored various ensemble techniques to combine the predictions (logits) of these four models:

3.4.1 Logit Averaging

The simplest ensemble approach involved directly averaging the raw output logits from all four base models. The final classification was then determined by taking the argmax of these averaged logits. This method yielded a test accuracy of **0.801**

3.4.2 Weighted Averaging

To further optimize the ensemble, we investigated weighted averaging, where each model's logits contribute with a specific weight to the final average. A brute-force grid search was performed to identify the combination of weights (summing to 1) that maximized the test accuracy. This systematic search led to an improved test accuracy of **0.811** with optimal weights **[0.05, 0.55, 0.2, 0,2]**.

3.4.3 Non-linear Ensembles (Stacking)

Beyond linear combinations, we experimented with non-linear meta-learners to combine the base model logits. The concatenated logits from the base models served as input to these meta-learners, which were trained on the validation set.

- **MetaStack (Simple MLP)**: A two-layer Multi-Layer Perceptron (MetaStack) was used as a meta-learner. This model combined the base logits to produce the final classification. This ensemble achieved a test accuracy of **0.796**.

- **MetaLinear (Logistic Regression)**: A single-layer linear model (MetaLinear) was employed as a meta-learner, acting as a logistic regression on the concatenated logits. This method yielded a test accuracy of **0.793**.
- **MetaMLP**: A more complex three-layer MLP (MetaMLP) with BatchNorm, GELU activation, and Dropout was also used as a meta-learner. This ensemble achieved a test accuracy of **0.801**.
- **MetaConv1D**: This meta-learner utilized 1D convolutional layers, treating each class's logits as a channel and applying depth-wise and point-wise convolutions across the models. Following global average pooling, it produced a final classification with a test accuracy of **0.805**.

Overall, the ensemble methods, especially the weighted averaging and certain non-linear stacking approaches, demonstrated the potential to achieve higher classification accuracies compared to individual models, showcasing the benefits of combining diverse GNN architectures for node classification on the Cora dataset.

System	**Edge Attributes**	**Accuracy (%)**
GCN	No	77.6
GAT	No	80.0
GCN-V2	absolute difference	79.1
GAT-V2	concatenation	79.9
Ensembling		
Averaging	-	80.1
Weighted Average	-	**81.1**
MetaStack	-	79.6
MetaLinear	-	79.3
MetaMLP	-	80.1
MetaConv1D	-	80.5

Table 1: Results from all the systems.

4 Conclusion

This project successfully explored the application of the Graph Neural Networks (GNNs) to the node classification problem in our case on the Cora dataset. We delved into the GNN architectures demonstrating the effectiveness and robustness of both convolutional-based (GCN, GCN-V2) and attention-based (GAT, GAT-V2) models on there ability to learn meaningful representation on graph-structured data. Our individual model evaluation showed that the Graph Attention Network (GAT) without explicit edge attributes achieved the highest standalone accuracy of 0.8 with the GAT model with the addition of edge attributes being a very close second with an accuracy of 0.799, emphasizing the power of attention mechanisms in aggregating neighborhood information in the graph.

We also experimented with various ensemble methods to further boost classification performance. We tried simple logit averaging, weighted averaging, and several non-linear stacking approaches using different meta-learners (MLPs, Logistic Regression, and 1D Convolutional networks). The results unequivocally demonstrated that ensemble learning can significantly enhance the robustness and accuracy of node classification. Notably, the weighted averaging method, with its optimized weights found through grid search, yielded the highest accuracy of 0.811.

In summary, this project showcased the power of GNNs for node classification and, more importantly, the significant performance gains achievable through thoughtful ensemble strategies. The weighted averaging of diverse GNN architectures proved to be a particularly effective approach, suggesting a promising direction for future research in complex network analysis and GNN applications. Future work could explore more sophisticated methods for learning ensemble weights or investigate other GNN architectures.

References

[1] B. Khemani, S. Patil, K. Kotecha, and S. Tanwar, "A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions," *Journal of Big Data*, vol. 11, p. 18, Jan 2024.

[2] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2, pp. 729–734 vol. 2, 2005.

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[4] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, no. 3, pp. 47–67.

[5] W. L. Hamilton, "Graph representation learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, no. 3, pp. 29–37.

[6] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[7] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," 2017.

[8] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?," 2022.

[9] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018.

[10] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," 2019.

[11] A. Daigavane, B. Ravindran, and G. Aggarwal, "Understanding convolutions on graphs," *Distill*, 2021. https://distill.pub/2021/understanding-gnns.