



Degree Project in Mathematical Statistics

Second cycle, 30 credits

Beyond Stochastic Gradient Descent : Sampling-Based Training for Graph Neural Network

BENOÎT GOUPIL

Abstract

Graph neural networks (GNNs) are a powerful framework for learning graph representations by recursively aggregating information from neighboring nodes. In this work, we propose alternative training methods for GNNs that avoid conventional stochastic gradient descent by employing random weight sampling. We introduce a novel alignment measure (ALI) to quantify the correspondence between graph embeddings and their labels, demonstrating that it correlates well with final model performance and can serve as an effective proxy during model selection. Based on ALI, we first present a random sampling approach that selects the best-performing model from a set of randomly initialized GNNs. This method offers a competitive alternative to traditional training. We further extend this approach by aggregating multiple small, independently sampled GNNs into a single, sparse ensemble model. This ensemble strategy not only enhances overall performance but also significantly reduces computational costs. Evaluations on benchmark datasets such as Mutagenicity, NCI1, and COLLAB demonstrate that our methods are competitive and time efficient, while in some cases outperforming conventional approaches for models with large hidden dimensions. Overall, our findings highlight the potential of random weight sampling and ensemble techniques as viable alternatives to standard training methods for GNNs, opening new perspectives for efficient graph representation learning.

Sammanfattning

Graph neural networks (GNN:er) utgör en kraftfull ram för inlärning av grafrepresentationer, där information från närliggande noder rekursivt aggregeras för att skapa en representation. I detta arbete föreslår vi alternativa träningsmetoder för GNN:er som undviker konventionell stokastisk gradientnedstigning genom att använda slumpmässiga vikter. Vi introducerar ett nytt anpassningsmått (ALI) för att kvantifiera överensstämmelsen mellan grafinbäddningar och deras etiketter, och visar att detta mått korrelerar väl med den slutliga modellprestandan samt kan fungera som en effektiv proxy vid modellval. Baserat på ALI presenterar vi först en slumpmässig genereringsmetod som väljer den bäst presterande modellen från en uppsättning slumpmässigt initierade GNN:er. Denna metod utgör ett konkurrenskraftigt alternativ till traditionell träning. Vi utvidgar vidare detta tillvägagångssätt genom att kombinera flera små, oberoende genererade GNN:er till en enda, gles ensemblemodell, vilket inte bara förbättrar den övergripande prestandan utan även avsevärt minskar de beräkningsmässiga kostnaderna. Utvärderingar på benchmark-datamängder såsom Mutagenicity, NCI1 och COLLAB visar att våra metoder är konkurrenskraftiga och tidsmässigt effektiva, där de i vissa fall överträffar konventionella metoder för modeller med stora dolda dimensioner. Sammanfattningsvis framhäver våra resultat potentialen hos slumpmässig viktgenerering och ensembletekniker som starka alternativ till konventionella träningsmetoder för GNN:er, vilket öppnar nya perspektiv för effektiv inlärning av grafrepresentationer.

Acknowledgements

First and foremost, I express my sincere gratitude to my supervisor, Pascal Welke, for warmly welcoming me to the lab at TU Wien in Vienna. His insightful advice and dedicated guidance have been invaluable throughout my work. I also wish to acknowledge Fabian Jögl for his technical assistance and thoughtful input. I extend my thanks to everyone at the ML lab at TU Wien, whose friendly support and collaboration made my time there both productive and enjoyable. Lastly, I would like to thank my supervisor at KTH, Joakim Andén, for his helpful guidance and constructive feedback, which significantly contributed to refining this thesis.

Contents

1	Introduction	5
1.1	Research Motivation	6
1.2	Contributions and Outline	6
1.3	Related Work	7
2	Background on Machine Learning for Graphs	8
2.1	Background on Graph Theory	8
2.2	Graph Machine Learning	9
2.3	Graph Neural Network Models	10
2.4	GNN Training	14
2.5	Common GNN limitations	17
2.6	Experimental Setup	18
3	Alignment Measurement	20
3.1	Alignment Definition	20
3.2	Practical Implementation	23
3.3	Limitations	24
3.4	Evolution of the Alignment Score During Training	24
4	Random Sampling	27
4.1	ALI_k -Acc Correlation	27
4.2	Best Model Sampling and Baseline Comparison	36
5	Random Ensemble Sampling	46
5.1	Ensemble-Sampled MPNNs	46
5.2	Comparison of Performance over Time	53
6	Conclusion	55

1 Introduction

Machine learning, and particularly deep learning, has revolutionized artificial intelligence over the past decade, enabling models to achieve state-of-the-art performance across diverse tasks. While many architectures – such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) – excel on Euclidean data (e.g., images, text), they struggle to model systems with intrinsic relational structure, where interactions between entities define the data’s behavior [3]. Graphs provide a natural framework for representing such systems, encoding entities as nodes and relationships as edges [1]. This representation is universal: it captures molecular structures through atomic bonds, social networks via user connections, and physical systems as interacting particles, among many other real-world examples.

One of the first works to apply deep learning on graph has been published by Scarselli et al. [18] as an extension of recurrent neural network. As in a classical neural network, it applies several layers on the features but also exploits the graph structure by transmitting information along the edges to the adjacent nodes. Such architectures are called graph neural networks (GNN) or more specifically message-passing neural networks (MPNN). Many new improved architectures have been proposed. One of the most famous ones is the graph convolutional neural network introduced by Kipf et al. [11].

These architectures offer a powerful framework for using the inherent structure of graphs to extract meaningful information. They can generate a compact representation of a graph, known as an embedding, which is a fixed-dimensional vector. The goal of this embedding is to capture the essential features of the graph as accurately as possible. An important characteristic of these embeddings is that they should reflect the similarity between graphs – similar graphs should have embeddings that are close to each other in the vector space, while dissimilar graphs should have embeddings that are farther apart. The definition of similarity is often related to the specific task. Graphs can for example be similar if they have closely associated labels. This vectorial representation can be used as an input for a more traditional neural network architecture.

As in all machine learning or deep learning models, training is a crucial step to fit the data. Graph neural networks are no exception and are trained using more or less the same techniques as traditional neural networks, called stochastic gradient descent (SGD). This is an iterative method used to optimize a function called the objective function or more commonly the loss function. This function usually represents the discrepancies between the model predictions and the actual ground truth, and the goal is to minimize it.

While stochastic gradient descent has proven itself powerful and efficient, the training part remains time-consuming. Additionally, in the non-convex framework of deep learning models, convergence to a global minimum is never guaranteed. The algorithm also requires careful hyperparameter tuning, adding to its complexity.

1.1 Research Motivation

The performance of graph neural networks is inherently dependent on the quality of their graph representations. A promising research direction is to investigate the extent to which the alignment between embedding similarity and graph label similarity can serve as a reliable predictor for a GNN generalization performance. Specifically, for a network to achieve optimal performance, graphs that are similar in terms of their labels should be represented by similar embeddings. Moreover, recent studies have demonstrated that untrained GNNs – with randomly initialized parameters – can perform competitively compared to their trained counterparts [2]. This finding suggests that the probability of obtaining a well-performing network at initialization is significantly high. This observation raises an intriguing question: could it be possible to directly sample the network’s parameters, bypassing traditional stochastic gradient descent (SGD)-based training, and still discover models that perform well based solely on this alignment property?

1.2 Contributions and Outline

In this thesis, we focus on developing new sampling methods that are competitive with gradient descent in both time efficiency and performance. We introduce a training approach that only relies on random sampling, eliminating the need for traditional gradient descent. This method allows us to train graph neural network architectures more efficiently, reducing both the time and computational power required. Additionally, our approach achieves results that match or even surpass those obtained using gradient descent.

In the second chapter, we present a background on graph machine learning problems. We delve into the problem setting, introducing the notations and concepts relevant to message-passing neural networks (MPNNs). Additionally, we provide an overview of the key architectures that will be used in this work.

In the third chapter, we introduce a novel measure called ALI to quantify the alignment between the embeddings produced by a given GNN and the labels of the associated graph. We provide a theoretical definition of ALI, detail its implementation, and examine the evolution of the score throughout training.

In the fourth chapter, we show that the alignment score is a reliable predictor of model performance and generalization across various MPNN architectures and hyperparameters. We further explore the distributions of both the alignment score and performance metrics, analyzing the likelihood of identifying a well-performing model through random sampling. Based on these insights, we present a random sampling method and benchmark its performance and efficiency against stochastic gradient descent on multiple datasets.

In the fifth chapter, we extend the random sampling method by incorporating ensemble techniques. This approach combines multiple small, independently sampled GNNs into a

more effective model, achieving superior time efficiency while outperforming traditional stochastic gradient methods.

Finally, in the last chapter, we conclude the thesis with a comprehensive discussion of our findings, their implications, and potential directions for future work.

1.3 Related Work

To the best of our knowledge, there has been limited investigation into the performance of untrained graph neural networks (GNNs). One of the first studies in this area is by Huang et al. [10], who explore the feasibility of identifying effective untrained GNNs without parameter tuning. However, their method still involves a training phase to optimally prune network connections. Interestingly, their results demonstrate that these sparse networks can match or even surpass the performance of fully trained models.

Another work on untrained GNNs is by Böker et al. [2], which investigates their ability to capture and compare graph structures without training. They show that even with random initialization, GNNs can process graphs effectively to extract meaningful patterns. The paper also observes that untrained GNNs can achieve competitive results in tasks like graph classification, emphasizing that their architecture naturally encodes useful information about graph structure, even without learning through training. However, the experimental part of the study is quite succinct, and the authors focus on specific architectures that are not commonly used in practical applications.

Finally, a recent work by Bui et al. [4] proposes a novel model based on randomness. They use completely random weights for the message-passing propagation. However, the model includes a node feature extractor, and the propagations are entirely independent to avoid interference between them. In this approach, the weight matrix is fully diagonal, ensuring no cross-interaction between different propagation dimensions.

2 Background on Machine Learning for Graphs

Before proceeding with the contributions, we first present the fundamental concepts of graph theory and machine learning as they relate to graph-based problems.

2.1 Background on Graph Theory

We introduce key graph theory concepts, including the definition of a graph and the notion of graph isomorphism.

2.1.1 Notations

We formally define a graph as $G = (V, E)$, where V is the finite set of vertices or nodes, and E is the set of edges. Each element of E is described as an ordered pair of vertices, $(i, j) \in V^2$.

The graph can be fully characterized by its adjacency matrix A . This matrix represents the connections between nodes, where A_{ij} is non-zero if there is an edge between node i and node j . The matrix is symmetric if the graph is undirected, i.e., for all $(i, j) \in E$, we have $(j, i) \in E$.

We can also define the neighbors of a node i by:

$$N_i = \{j \in V \mid (i, j) \in E\}.$$

Each node i can be associated with a feature vector $x_i \in \mathbb{R}^d$. This vector encodes relevant information for the node. The concatenation of all the node features can be denoted as $X \in \mathbb{R}^{|V| \times d}$, where d is the dimension of the features. In some cases, the edges can also have features.

2.1.2 Graph Isomorphism and Expressivity

Graph isomorphism is a common notion in graph theory. When we state that two graphs are isomorphic, it essentially means that those graphs represents the same thing.

Definition 1 (Graph Isomorphism) *Two graphs G and H are said to be isomorphic if there exists a bijection between the vertex sets $V(G)$ and $V(H)$ of G and H :*

$$f : V(G) \rightarrow V(H),$$

such that for any two vertices $u, v \in V(G)$, there is an edge between u and v in G if and

only if there is an edge between $f(u)$ and $f(v)$ in H . Formally:

$$(u, v) \in E(G) \iff (f(u), f(v)) \in E(H),$$

where $E(G)$ and $E(H)$ denote the edge sets of G and H , respectively.

We denote this isomorphism as $G \cong H$.

Expressivity refers to a model’s ability to represent and approximate a wide range of functions. In the context of graphs, this concept also encompasses the capacity to differentiate between distinct graph structures. In other words, a highly expressive graph model should be able to produce unique representations for non-isomorphic graphs. This requirement is directly linked to the graph isomorphism problem, which involves determining whether two graphs have the same structure. Ideally, a model with strong expressivity ensures that if graphs G and H are not isomorphic, their outputs are clearly distinguishable. This property is crucial for tasks such as graph classification, where accurately capturing structural differences is essential.

2.2 Graph Machine Learning

Graph machine learning is a recent and growing field that operates on graphs. It allows us to process graphs to make more complex predictions. These tasks can be divided into three levels: graph-level tasks, node-level tasks, and edge-level tasks.

2.2.1 Graph-Level Tasks

At the graph level, the entire graph is associated with a label, and the objective is to predict this characteristic accurately. These tasks typically involve predicting either categorical properties, i.e., classification tasks, or numerical properties, i.e., regression tasks.

In a chemical setting, typical graph-level prediction tasks include toxicity classification – predicting whether a molecule is toxic or non-toxic based on its molecular structure – and molecular activity prediction, which determines whether a molecule acts as an active or inactive compound against a specific biological target.

2.2.2 Node-Level Tasks

In node-level tasks, each node in the graph is associated with a label, and the goal is to predict the label for individual nodes. These tasks also fall into the categories of classification and regression.

In the context of social networks, classical node-level tasks cover social network analysis, where the goal is to predict the community or group a user belongs to within a

social network. Regression tasks, on the other hand, can involve influence scoring in social networks, which involves estimating a user’s influence or importance.

In a such scenario, the node-set V is divided in two non-overlapping subsets V_a and V_b such that $V_a \cup V_b = V$ and the objective is to predict the labels of the nodes that belong to V_a based on the information obtained on the graph structure and the node labels in V_b .

2.2.3 Edge-Level Tasks

Finally, edge-level tasks provide another important class of graph prediction problems. Edge regression tasks, for instance, involve predicting numerical values associated with edges, such as estimating the traffic flow between two connected intersections in a road network. Similarly, edge prediction focuses on determining the existence or likelihood of an edge between two nodes, such as predicting whether two users in a social network are likely to form a connection based on shared interests or mutual friends.

2.3 Graph Neural Network Models

As described above, GNNs are machine learning models that operate on graphs. They aim at exploiting the topological structure of the data to obtain meaningful representations and make predictions.

2.3.1 Invariance and equivariance

One important characteristic of graphs is that there is no specific ordering of the nodes. This means that any function applied on the graph should not consider the nodes nor the nodes features in a specific order. This property is called permutation invariance.

A function f applied to the graph should satisfy:

$$f(G) = f(\pi(G)),$$

where $\pi(G)$ represents the graph G with a permutation π applied to the nodes, i.e., the node feature matrix X is transformed to $\pi(X)$ and the adjacency matrix A to $\pi(A)$. Here, $\pi(X)$ permutes the rows of X according to π , and $\pi(A)$ permutes the rows and columns of A correspondingly.

This property ensures that the function f operates on the graph structure and features without being influenced by an arbitrary ordering of the nodes.

At the node level, a function f that operates on the feature vector of each node to produce a new representation should respect the permutation equivariance property. Formally, let $G = (V, E)$ be a graph with a node feature matrix $X \in \mathbb{R}^{|V| \times d}$, where d is

the feature dimension. The function $f : \mathbb{R}^{|V| \times d} \rightarrow \mathbb{R}^{|V| \times d'}$, which maps the input feature matrix to an output feature matrix, satisfies permutation equivariance if:

$$f(\pi(X)) = \pi(f(X)),$$

where $\pi(X)$ represents the feature matrix X with rows permuted according to a permutation π , and $\pi(f(X))$ permutes the rows of the output feature matrix $f(X)$ in the same way.

This property ensures that the node-level representations produced by f are consistent with any reordering of the nodes in the graph, making f invariant to the arbitrary indexing of nodes while preserving their relational structure.

2.3.2 Evolution of the GNNs models

The early foundations of graph neural networks (GNNs) were introduced by Gori et al. [8] and further developed by Scarselli et al. [18]. These studies introduced a framework for learning node representations by iteratively propagating information between neighboring nodes until convergence to a stable fixed point. Although this approach demonstrated great promise, it was computationally expensive, motivating the development of more efficient architectures.

Inspired by convolutional layers in computer vision, which aggregate information from neighboring pixels to update a pixel’s value, graph convolutional neural networks (GCNs) were introduced [11]. GCNs adapt this concept to graph-structured data by updating node representations through a fixed number of layers, each of which aggregates information from a node’s immediate neighbors. This architecture is referred to as spatial-based GCN. In contrast, spectral-based GCNs operate in the spectral domain of the graph using the eigendecomposition of the Laplacian and uses the Fourier transform to define graph convolutions by introducing filter from the perspective of graph signal processing [20]. While spectral methods are effective for certain tasks, they often require costly eigenvalue computations and do not scale to large graphs. We will not delve further into spectral GCNs in this discussion.

Other architectures have been proposed such as graph attention network (GAT) [19]. It uses attention mechanisms to assign learnable importance weights to neighboring nodes during the message-passing process. This allows GATs to learn which neighbors contribute most to a node’s representation.

Another notable architecture is the graph isomorphism network (GIN) introduced by Xu et al. [21]. The paper makes two main contributions. First, it demonstrates that the expressive power of any MPNN architecture cannot exceed that of the Weisfeiler–Lehman (WL) test for graph isomorphism. This implies that if two graphs cannot be distinguished by the WL test, no MPNN can distinguish them either, regardless of its architecture or

parameters. The second contribution is the introduction of GIN, which updates node representations by aggregating neighboring features using a sum operation followed by a multi-layer perceptron, and which they prove to be as expressive as the WL test.

2.3.3 Formal Message Passing Framework

In this study, we focus on a subcategory of GNNs known as message-passing neural networks (MPNNs). These models are constructed by stacking several layers that follow the same principle: at each layer, every node’s representation is updated by propagating messages from its neighboring nodes along the edges. Formally, let $x_i^{(k)} \in \mathbb{R}^{d_k}$ denote the representation of node i at the k^{th} layer. Then, the update rule for the k^{th} layer of an MPNN can be expressed as:

$$x_i^{(k+1)} = \phi^{(k)} \left(x_i^{(k)}, \mathbf{Agg} \left(\{ \psi^{(k)}(x_i^{(k)}, x_j^{(k)}) \mid j \in \mathcal{N}(i) \} \right) \right),$$

where we have the following components:

- $\phi^{(k)}$: The node update function for layer k , which transforms the node’s current state and aggregated information into a new representation. This function can be a linear transformation, an MLP, or another learnable mapping.
- $\psi^{(k)}$: The message function for layer k , which computes the information sent from a neighboring node j to the target node i . This function can depend on both $x_i^{(k)}$ and $x_j^{(k)}$, or only on $x_j^{(k)}$.
- $\mathbf{Agg}(\cdot)$: The aggregation function, which combines messages from all neighboring nodes $j \in \mathcal{N}(i)$. Common aggregation functions include summation, mean, or max pooling.
- $\mathcal{N}(i)$: The set of neighboring nodes for node i in the graph.

This general form ensures that the GNN can handle arbitrary graph structures, as it is invariant to the ordering of nodes (permutation invariance) and aggregates information in a structured manner. Using this representation, we can formally express the GCN and the GIN architectures that we are going to use throughout this work.

GCNs use a weighted aggregation of neighboring node features, normalized by the degree of the nodes, and apply a shared linear transformation followed by a non-linear activation. The layer-specific representation for GCN can be expressed as:

$$x_i^{(k+1)} = \sigma \left(W^{(k)} x_i^{(k)} + \sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{d_i d_j}} W^{(k)} x_j^{(k)} \right),$$

where we have:

- $\psi^{(k)}(x_i^{(k)}, x_j^{(k)}) = \frac{1}{\sqrt{d_i d_j}} W^{(k)} x_j^{(k)}$, where d_i and d_j are the degrees of nodes i and j , and $W^{(k)}$ is the layer-specific learnable weight matrix.
- $\phi^{(k)}(x_i^{(k)}, z) = \sigma \left(W^{(k)} x_i^{(k)} + z \right)$, where z denotes the aggregated messages and σ is a non-linear activation function (e.g., ReLU).
- $\mathbf{Agg}(\{z_1, \dots, z_l\}) = \sum_{j=1}^l z_j$.

The normalization term is applied to balance the contributions of nodes with different degrees, ensuring numerical stability and mitigating the effects of varying connectivity.

GINs aim to distinguish graph structures by using a sum aggregation and an expressive multi-layer perceptron (MLP) for updating node features. The layer-specific representation for GIN is the following:

$$x_i^{(k+1)} = \text{MLP}^{(k)} \left((1 + \epsilon^{(k)}) x_i^{(k)} + \sum_{j \in \mathcal{N}(i)} x_j^{(k)} \right),$$

where we have:

- $\psi^{(k)}(x_i^{(k)}, x_j^{(k)}) = x_j^{(k)}$ (simple identity message function for neighbors).
- $\phi^{(k)}(x_i^{(k)}, z) = \text{MLP}^{(k)} \left((1 + \epsilon^{(k)}) x_i^{(k)} + z \right)$, where $\epsilon^{(k)}$ is a learnable parameter or a fixed scalar, z denotes the aggregated messages, and $\text{MLP}^{(k)}$ is a layer-specific multi-layer perceptron.
- $\mathbf{Agg}(\{z_1, \dots, z_l\}) = \sum_{j=1}^l z_j$.

As mentioned earlier, the GIN architecture was designed to maximize expressivity. It aggregates the complete multiset of neighboring features using an unnormalized summation, thereby retaining more detailed information than normalized approaches. By replacing a simple linear transformation with an MLP, the network can learn more complex functions, significantly broadening the range of approximable functions and enhancing theoretical expressivity. Additionally, the learnable parameter ϵ allows the model to finely balance the contribution of a node’s own features with those aggregated from its neighbors.

2.3.4 Graph-Level Prediction Tasks

To address graph-based problems, we require a global representation of the entire graph. This global graph representation is essential for tasks such as graph classification or regression, where the focus is on the properties of the graph as a whole rather than individual

nodes or edges. To compute this global representation, a read-out function is employed, which aggregates node representations across the graph into a single vector.

Formally, the global graph representation h_G can be expressed as:

$$h_G = \mathbf{Readout}(\{x_i^{(K)} \mid i \in V\}),$$

where $x_i^{(K)}$ represents the final node representation in the last GNN layer K , V is the set of nodes in the graph, and **Readout** is a permutation-invariant function, such as summation, mean, or max pooling. It is worth mentioning that more advanced pooling techniques, such as Attention Pooling [12] and DiffPool [22], have been proposed to capture richer graph representations. However, in this study, we will limit our focus to the simpler and more commonly used sum and mean pooling functions.

After obtaining a meaningful vectorial representation (or embedding) of the graph through the GNN model, this embedding can be used for graph-level tasks, such as classification or regression. Typically, a multi-layer perceptron (MLP) head is employed, taking the graph embedding as input to predict the desired output.

2.4 GNN Training

Machine learning models require a training phase to learn from data. This process varies in complexity depending on the model. For example, in the k -nearest neighbors (k -NN) method, training involves storing the training data in memory to enable direct comparisons during inference. However, for many other models, the training phase is more complex and often involves optimization techniques.

Most training processes focus on minimizing a loss function, which quantifies the error or difference between the model’s predictions and the actual data. In deep learning, where models have a large number of parameters, finding a closed-form solution to minimize the loss function is infeasible. Instead, iterative optimization methods are used, with gradient descent and its many variants being the most common approaches. These methods adjust the model’s parameters step-by-step based on the gradient of the loss function, gradually improving the model’s performance.

Graph neural networks, like other deep learning models, rely on gradient descent to iteratively adjust and optimize their weights.

2.4.1 Loss Functions

A loss function defines the objective of a model and measures the discrepancy between the model’s predictions and the true labels. The goal is to select parameters that minimize this loss, ideally making it as small as possible.

In a classification setting, the cross-entropy loss is frequently used. Suppose there are N training examples and C classes. The cross-entropy loss is given by

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c}),$$

Where $y_{i,c}$ is an indicator (often one-hot encoded) that equals one if example i belongs to class c and zero otherwise, and $\hat{y}_{i,c}$ is the probability output by the model that example i belongs to class c .

In a regression setting, a common loss function is the mean squared error (MSE). Let y_i be the true value for the i^{th} example and \hat{y}_i the predicted value. The mean squared error is defined as

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2.$$

2.4.2 The Gradient Descent Algorithm

Gradient descent is an iterative optimization method used to find a local minimum of a differentiable multivariate function. Let the objective function be defined as an average over a dataset with N samples:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}_{\boldsymbol{\theta}}(\mathbf{x}_i),$$

where $\mathcal{L}_{\boldsymbol{\theta}}(\mathbf{x}_i)$ represents the contribution of the sample \mathbf{x}_i when using the parameters $\boldsymbol{\theta} \in \mathbb{R}^n$. Our goal is to find

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}).$$

The idea behind gradient descent is to use the gradient of \mathcal{L} to update $\boldsymbol{\theta}$ in the direction that locally decreases the function's value. Starting from an initial guess $\boldsymbol{\theta}^{(0)}$, the parameters are updated iteratively as

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k)}),$$

where η is the learning rate and $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{(k)})$ denotes the gradient of \mathcal{L} with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}^{(k)}$.

Since computing the exact gradient

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}}(\mathbf{x}_i)$$

requires summing over all N data points, it can be computationally expensive for large datasets. To address this, stochastic gradient descent (SGD) and its variants approximate the full gradient by using a small subset (mini-batch) of the data at each iteration. Let \mathcal{B} be a randomly sampled mini-batch of data points (or their indices). The mini-batch gradient estimator is then defined as

$$\hat{\mathcal{L}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}}(\mathbf{x}_i),$$

which approximates the full gradient $\nabla \mathcal{L}_{\boldsymbol{\theta}}(\boldsymbol{\theta})$. Although $\hat{\mathcal{L}}(\boldsymbol{\theta})$ is not the exact gradient, it is an unbiased estimator provided that the mini-batch is chosen uniformly at random from the dataset.

Choosing the mini-batch size is a trade-off: smaller batches allow faster updates and add useful randomness that can help escape local minima, while larger batches provide more accurate gradient estimates but require more computation.

2.4.3 Computing the Gradient in Neural Networks

In a neural network, we first perform a forward pass by feeding inputs through each layer to produce an output, from which we compute a loss \mathcal{L} . We then compute the gradient of \mathcal{L} with respect to every parameter by systematically applying the chain rule backward through the network, a process known as backpropagation. If $\boldsymbol{\theta} = (\theta_1, \dots, \theta_m)$ denotes the current parameter configuration, then the gradient of \mathcal{L} at $\boldsymbol{\theta}$ is given by

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \left(\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \theta_1}, \dots, \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \theta_m} \right)$$

Each partial derivative $\frac{\partial \mathcal{L}}{\partial \theta_i}$ is found by tracing how changes in θ_i propagate to the final loss, allowing us to update all parameters simultaneously in a direction that locally reduces \mathcal{L} .

In a typical message-passing neural network (MPNN), each layer is fully differentiable, which means we can apply standard backpropagation to compute gradients with respect to the learnable parameters. After calculating these gradients, we update the parameters using SGD, just like in other neural network architectures.

2.4.4 Limitations of Gradient Descent

Gradient descent relies on following the local slope of the loss function. While it is guaranteed to converge to a global minimum for convex functions, many real-world applications – such as training neural networks – involve non-convex loss functions. In these scenarios, gradient descent can converge to local minima or saddle points, preventing it from reaching a globally optimal solution. Descent is very sensitive to the learning rate: if it is set too high, each update can overshoot and lead to divergence; if it is set too low,

training proceeds slowly and is more likely to get stuck in bad local minima. Moreover, poor weight initialization can steer the learning process toward suboptimal areas of the parameter space, increasing the risk of ending up in saddle points or degenerate minima – regions where the loss is flat in some directions, so many nearby parameter settings yield similar loss values.

Another challenge comes from working with very high-dimensional problems, where there are more saddle points than simple local minima. These saddle points can slow or halt progress because the gradient can be zero without the model being at a true minimum. In deep networks, gradients can also become extremely small (vanish) or extremely large (explode) when backpropagated through many layers, causing instability or very slow training. On the computational side, calculating gradients over the entire dataset each time is expensive, so stochastic or mini-batch methods are usually preferred.

Gradient descent is called a first-order method because it only uses the gradients themselves. Although second-order methods that use the Hessian can, in theory, converge faster, they are often impractical for large models due to their high computational and memory requirements. Despite these drawbacks, gradient descent and its variants remain central to training deep models because they scale well and are supported by many techniques that help reduce these limitations.

2.5 Common GNN limitations

As described earlier, the message-passing framework in GNNs is constructed by stacking several layers, where each node representation $x_i \in \mathbb{R}^d$ is updated by aggregating information from its neighboring nodes. In most neural network architectures, particularly those used in computer vision, depth plays a crucial role, and it is common to see models with dozens or even hundreds of layers. However, in GNNs, architectures are often relatively shallow, with only a few layers. This limitation arises due to specific challenges that hinder the performance and scalability of deeper GNNs, including the issues of oversmoothing [17] and oversquashing [6].

Oversmoothing arises when, after several rounds of message passing, the features of different nodes become almost the same. Since each node repeatedly mixes its own information with that of its neighbors, these representations eventually converge to a single value, making it impossible to distinguish one node from another. The model can no longer capture the unique characteristics of each node, which undermines its ability to perform tasks like node classification and link prediction effectively.

Oversquashing occurs when the graph’s structure itself restricts how far information can travel. If only a small number of edges connect two parts of the graph, any signal passing between them can get “squashed” or lost, preventing effective communication between distant nodes. One way to reduce oversquashing is by “rewiring” the graph to

add extra edges, allowing information to flow more freely.

2.6 Experimental Setup

In this section, we present the datasets and describe the experimental environment used to evaluate our methods. Given that our results are primarily empirical, establishing a rigorous and reproducible experimental setup is crucial.

2.6.1 Datasets description

The data used in this project consists entirely of publicly available open datasets. The two primary sources are TUDataset [15] and the Open Graph Benchmark [9]. These datasets provide a diverse collection of graphs, including molecular structures, protein interactions, and social graphs, among others. Due to the high computational cost of running experiments, our study focuses on graph-level classification tasks to allow for an in-depth performance analysis. We also propose methods for regression and outline potential extensions for node-level tasks, although experimental results for these will not be included. We will conduct various experiments on three main datasets:

- **Mutagenicity dataset:** A chemical compound dataset that categorizes drugs into two classes: mutagen (1,936 instances) and non-mutagen (2,401 instances), which are fairly balanced. Each compound is represented as a graph, where nodes correspond to atoms that are one-hot encoded into a 14-dimensional feature vector, reflecting 14 distinct atom types (e.g., carbon, nitrogen, oxygen, etc.). Chemical bonds are modeled as edges that include additional attributes; however, these edge attributes are not considered in our study to simplify the GNN architectures.
- **NCI1 dataset:** A cheminformatics dataset that classifies chemical compounds based on graph representations of their molecular structures as either active or inactive in inhibiting lung cancer cell growth. As in the previous dataset, each molecule is represented as a graph where nodes represent atoms encoded as a 37-dimensional one-hot vector corresponding to 37 distinct atom types, and edges represent chemical bonds – without any additional edge features. The dataset is well balanced, containing 2,053 positive and 2,057 negative instances.
- **COLLAB dataset:** A social dataset comprising 5,000 graphs represents researchers’ collaboration networks. In each graph, nodes correspond to researchers and edges indicate collaborative relationships. Each graph is assigned one of three fields – high energy physics, condensed matter physics, or astrophysics – reflecting the researcher’s area of expertise. The dataset is not completely balanced, with distributions of 2,600, 775, and 1,625 graphs across the three fields. Although no intrinsic

node features are provided, a common practice is to one-hot encode the node degree for each node, resulting in 369 unique degree features.

These datasets are widely used as benchmark datasets for evaluating graph classification tasks. Table 1 summarizes general properties of the datasets.

Dataset	Number of graphs	Classes	Avg. Nodes	Avg. Edges	Node labels
Mutagenicity	4337	2	30.32	30.77	✓
NCI1	4110	2	29.9	32.3	✓
COLLAB	5000	3	74.49	2457.78	✗

Table 1: Properties of benchmark datasets.

2.6.2 Experimental Environment

All experiments were conducted using Python 3.10, with PyTorch and PyTorch Geometric used for implementing the graph neural networks (GNNs). The experiments were run on an NVIDIA GeForce RTX 3080 GPU.

In the absence of predefined train/test splits, we adopted a standard cross-validation approach to evaluate accuracy performance across methods. The data splits were fixed with a random seed of zero to ensure that all methods are compared under identical fold conditions.

3 Alignment Measurement

As described in the previous chapter, GNN models can map graphs to meaningful embeddings. These embeddings are then used by a downstream head, often a multilayer perceptron (MLP), to perform a specific machine learning task.

Intuitively, for a given task, a well-learned representation should place graphs with similar labels closer together in the embedding space, while graphs with different labels should be positioned farther apart. To evaluate this property, we propose two pseudometrics: one to quantify similarity between graphs in the embedding space, and another to measure similarity in the label space. With these two pseudometrics, we can quantify the alignment between the embedding space and the label space. For a specific GNN model and given task (i.e. a set of labeled graphs), we can assign an alignment score that measures how effectively the learned representations reflect the underlying label structure.

The primary goal of this score is to evaluate the performance of a sampled MPNN (i.e. with randomly initialized weights) without explicitly computing accuracy, which would require training an MLP head for the specific task. As such, there are two main requirements for this measure. First, it must be computationally efficient. The faster we can evaluate a sampled MPNN, the more quickly we can iterate and sample new models to identify the optimal one. Second, the score must be meaningful. Since the ultimate objective is to achieve strong performance on the target task, this score should exhibit a high correlation with the performance metric used for the dataset. In this work, accuracy is the primary performance metric under consideration.

3.1 Alignment Definition

In this section, we first present the mathematical definitions of the two pseudometrics and then introduce the alignment score for a given MPNN, applicable to both graph-level classification and regression tasks.

Definition 2 (Pseudometric on graphs) *A pseudometric on a set of graphs \mathcal{G} is a function $d : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}_+$ that satisfies the following properties for all $F, G, H \in \mathcal{G}$:*

$$d(G, H) \geq 0 \quad (\text{Non-negativity})$$

$$d(G, H) = d(H, G) \quad (\text{Symmetry})$$

$$d(G, H) \leq d(G, F) + d(F, H) \quad (\text{Triangle inequality})$$

$$d(G, G) = 0 \quad (\text{Identity})$$

It is worth noting that it is indeed a pseudometric rather than a metric, as we do not require separability – that is, $d(G, H) = 0$ does not necessarily imply $G \cong H$.

Definition 3 (Pseudometric space on graphs) A pseudometric space on graphs is a pair (\mathcal{G}, d) , where \mathcal{G} is a set of graphs and $d : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}_+$ is a pseudometric that satisfies the properties of non-negativity, symmetry, triangle inequality, and identity.

After presenting the general definition of pseudometrics on graphs, we define a specific pseudometric based on the embeddings produced by the MPNN model.

Definition 4 (MPNN Pseudometric) Assume we have a message passing neural network architecture $\phi_\theta : \mathcal{G} \rightarrow \mathbb{R}^d$, where $\theta \in \mathbb{R}^m$ represents the parameters of a fixed architecture, which computes a d -dimensional graph representation $\phi_\theta(G)$ for any $G \in \mathcal{G}$. Furthermore, assume that we have a metric $d : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$.

The pseudometric d_{MPNN} is defined as the function that measures the distance between two graphs $G, H \in \mathcal{G}$ by applying the metric d to their respective embeddings obtained via the message-passing neural network ϕ_θ . Formally, the MPNN pseudometric is defined as:

$$d_{MPNN}(G, H) = d(\phi_\theta(G), \phi_\theta(H)).$$

It is worth noting that d_{MPNN} is a pseudometric rather than a true metric due to the limited expressivity of message-passing neural networks. Specifically, MPNNs are known to map certain non-isomorphic graphs to the same embedding (i.e., $\phi_\theta(G) = \phi_\theta(H)$ for $G \not\cong H$), particularly in cases where the graph structures are indistinguishable by the Weisfeiler–Lehman graph isomorphism test [21]. As a result, $d_{MPNN}(G, H) = 0$ does not necessarily imply $G \cong H$.

Now that we have defined the pseudometric on the embedding space, we introduce the functional pseudometric on the label space, which measures the similarity between graphs based solely on their labels.

Definition 5 (Functional pseudometric) Let y_G be the target label of graph G in a given task. In classification, y_G is a categorical class (typically represented as an integer corresponding to a category), while y_G is a numerical value in regression. We assume the space for y_G is bounded. Then, the functional pseudometric space (\mathcal{G}, d_{func}) is obtained from $d_{func} : \mathcal{G} \times \mathcal{G} \rightarrow [0, 1]$ defined as:

$$d_{func}(G, H) := \begin{cases} 1_{y_G \neq y_H} & (\text{classification}) \\ \frac{|y_G - y_H|}{\sup_{I \in \mathcal{G}} y_I - \inf_{I \in \mathcal{G}} y_I} & (\text{regression}), \end{cases}$$

where $1_{y_G \neq y_H}$ is the indicator function that equals one if $y_G \neq y_H$, otherwise zero.

One can easily prove that (\mathcal{G}, d_{func}) satisfies the three requirements for a pseudometric space. In addition, (\mathcal{G}, d_{func}) is not a metric space in general, since there can be non-isomorphic graphs with identical target labels.

Definition 6 (Alignment between d_{MPNN} and d_{func}) Let \mathcal{D} be a labeled graph dataset, ϕ an MPNN, and k a positive integer. For each graph $G \in \mathcal{D}$, define its k -nearest neighbor set in the embedding space as the unique subset $\mathcal{S}_k(G) \subseteq \mathcal{D} \setminus \{G\}$ with $|\mathcal{S}_k(G)| = k$ such that

$$\max_{H \in \mathcal{S}_k(G)} d_{\text{MPNN}}(G, H) \leq d_{\text{MPNN}}(G, H') \quad \forall H' \in \mathcal{D} \setminus (\mathcal{S}_k(G) \cup \{G\})$$

We also define two terms:

$$A_k(G; \phi, \mathcal{D}) := \frac{1}{k} \sum_{H \in \mathcal{S}_k(G)} d_{\text{func}}(G, H)$$

and

$$B_k(G; \phi, \mathcal{D}) := \frac{1}{|\mathcal{D}| - k - 1} \sum_{H \in \mathcal{D} \setminus (\mathcal{S}_k(G) \cup \{G\})} d_{\text{func}}(G, H).$$

The alignment score between d_{MPNN} and d_{func} is then given by

$$ALI_k(\phi, \mathcal{D}) := \frac{1}{|\mathcal{D}|} \sum_{G \in \mathcal{D}} \left[-A_k(G; \phi, \mathcal{D}) + B_k(G; \phi, \mathcal{D}) \right].$$

Here, $A_k(G; \phi, \mathcal{D})$ is the average functional difference between G and its neighbors. Similarly, $B_k(G; \phi, \mathcal{D})$ is the average functional difference between G and its non-neighbors. Thus, if $A_k(G; \phi, \mathcal{D}) < B_k(G; \phi, \mathcal{D})$, it follows that the MPNN embeds G and functionally similar graphs close, and functionally dissimilar graphs distant. Finally, if $A_k(G; \phi, \mathcal{D}) < B_k(G; \phi, \mathcal{D})$ holds on average, i.e., $ALI_k(\phi, \mathcal{D})$ is positive, we say d_{MPNN} is aligned with d_{func} . In addition, the larger $ALI_k(\phi, \mathcal{D})$, the more we say d_{MPNN} is aligned with d_{func} .

This alignment score quantifies the extent to which the local label distribution around a graph in the embedding space differs from the global label distribution within the dataset. Specifically, for a graph $G \in \mathcal{D}$, $A_k(G; \phi, \mathcal{D})$ will be small if the label distribution among the k nearest neighbors of G is homogeneous. Conversely, $B_k(G; \phi, \mathcal{D})$ captures the global label distribution of G across the entire dataset \mathcal{D} , excluding its local neighborhood. An alignment score close to zero indicates that the local label distribution is nearly identical to the global distribution, suggesting that no significant local structure exists within the embedding space. In this case, the MPNN fails to encode the graphs into a meaningful structure that reflects the label information. This interpretation suggests that a well-performing MPNN should produce embeddings that achieve a high alignment score.

The parameter k determines the locality of the neighborhood: smaller values of k focus on a narrow, immediate neighborhood in the embedding space, while larger values capture a broader local context. If k is too small, there may not be enough meaningful neighboring graphs to compare, leading to a noisy and imprecise measure. Alternatively, if k is too large, our graph might be compared with graphs that are too distant, resulting in a less meaningful comparison.

3.2 Practical Implementation

As mentioned before, the alignment score would be used to assess the performance of a sampled MPNN. Therefore, the performance of the method relies highly on the computational speed of the score. To ensure efficiency, it is essential to vectorize these operations and leverage GPU acceleration to handle large datasets and computationally intensive tasks effectively.

The score is computed as follows. First, we compute the MPNN distance matrix for the entire dataset $\mathcal{D} = \{G_1, G_2, \dots, G_n\}$:

$$\mathbf{D}_{\text{MPNN}} = \begin{bmatrix} d_{\text{MPNN}}(G_1, G_1) & d_{\text{MPNN}}(G_1, G_2) & \cdots & d_{\text{MPNN}}(G_1, G_n) \\ d_{\text{MPNN}}(G_2, G_1) & d_{\text{MPNN}}(G_2, G_2) & \cdots & d_{\text{MPNN}}(G_2, G_n) \\ \vdots & \vdots & \ddots & \vdots \\ d_{\text{MPNN}}(G_n, G_1) & d_{\text{MPNN}}(G_n, G_2) & \cdots & d_{\text{MPNN}}(G_n, G_n) \end{bmatrix}.$$

In practice, for d , we used the l_2 distance; however, we also experimented with l_1 and cosine distances, and observed no notable differences in the results.

Next, using \mathbf{D}_{MPNN} , we compute a mask matrix \mathbf{M} based on whether each graph is in the neighborhood $\mathcal{S}_k(G_i)$ of G_i . For each $i = 1, \dots, n$, let $t_k(G_i)$ be the k^{th} smallest value in the i^{th} row of \mathbf{D}_{MPNN} . Then, the mask is defined by

$$\mathbf{M}_{i,j} = \begin{cases} 1, & \text{if } d_{\text{MPNN}}(G_i, G_j) \leq t_k(G_i), \\ 0, & \text{otherwise.} \end{cases}$$

Now we can now compute the vector $\mathbf{a} \in \mathbb{R}^n$ which represents the A_k term for each graph in the dataset. We apply the mask matrix \mathbf{M} to the matrix of functional distances \mathbf{D}_{func} , where $\mathbf{D}_{\text{func}}(i, j) = d_{\text{func}}(G_i, G_j)$. The i^{th} component of the resulting \mathbf{a} vector is then computed as:

$$\mathbf{a}_i = \frac{1}{k} \sum_{G_j \in \mathcal{S}_k(G_i)} d_{\text{func}}(G_i, G_j) = \frac{1}{k} \sum_j \mathbf{M}_{i,j} \cdot \mathbf{D}_{\text{func}}(i, j).$$

For the \mathbf{b} vector, we could follow a similar approach by defining a mask for the complement of the neighborhood $\mathcal{S}_k(G_i)$. The ALI score is then obtained by averaging the difference of the \mathbf{a} and \mathbf{b} vectors. This practical implementation does not fundamentally alter the asymptotic complexity; however, by vectorizing the operations and enabling GPU acceleration in Python, it significantly improves the runtime performance.

3.3 Limitations

The first limitation of this approach lies in the size of the dataset. As the dataset grows, both the memory requirements and computational cost of calculating the alignment score increase significantly. This is because the method involves computing pairwise distances for all graphs in the dataset, resulting in a complexity of $O(m^2)$, where m is the number of graphs. Consequently, for very large datasets, this process becomes resource-intensive. In practice, we have found that the alignment score can be computed efficiently in a single run for datasets with up to 20,000 graphs, beyond which computational constraints may arise.

A second limitation is that the alignment score depends on comparing graphs within the same label class. For the score to be meaningful, the dataset must contain a sufficient number of similar graphs sharing the same label, and it needs to be dense enough around each graph to allow for meaningful comparisons. Without enough labeled examples, particularly in smaller or imbalanced datasets, it becomes challenging to establish robust comparisons.

3.4 Evolution of the Alignment Score During Training

As previously discussed, the ALI score must exhibit a strong correlation with the final performance metric. In this section, we will analyze its behavior for trained GNNs and examine the evolution of the ALI score throughout the training process across various datasets. The correlation of the ALI score for sampled GNNs will be addressed in the next chapter.

For graph-level tasks, the MPNN first computes node embeddings through multiple message-passing layers. To obtain the final vector representation of the graph, a pooling operation – typically mean or sum pooling – is applied to aggregate the node embeddings into a single graph-level representation. To adapt the model for a specific task, an MLP head, usually consisting of two hidden layers, is trained on top of the graph representation.

In a traditional supervised training framework, the dataset is typically divided into two or three subsets: the training set, validation set, and test set. The training set, which constitutes the majority of the data, is used to train the model’s weights. The validation set is then employed to select the best performing model. Finally, the test set is used to evaluate the model’s performance on unseen data, providing a measure of its real-world effectiveness. The accuracy is calculated on the test set to assess this performance.

For the alignment score (ALI), our primary focus is on evaluating its predictive power for the final test accuracy. It is important to note that the ALI score is computed on the training set, as it measures how well the MPNN aligns its learned representations with the target labels within this subset. Additionally, analyzing the correlation between the ALI

score and the training accuracy provides valuable insights into how the alignment reflects the model’s learning dynamics and its potential to generalize effectively to unseen data.

Let the dataset \mathcal{D} consist of n labeled graphs:

$$\mathcal{D} = \{(G_1, y_1), (G_2, y_2), \dots, (G_n, y_n)\},$$

where G_i is a graph and $y_i \in \mathcal{Y}$ is its label.

The dataset is split into three disjoint subsets:

$$\mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}} \cup \mathcal{D}_{\text{test}} = \mathcal{D}.$$

- $\mathcal{D}_{\text{train}}$: Used to optimize the model parameters.
- \mathcal{D}_{val} : Used to select the best performing model.
- $\mathcal{D}_{\text{test}}$: Used to evaluate final performance.

An MPNN, denoted $\phi_{\theta}^{\text{MPNN}}$, computes a graph embedding $\phi_{\theta}^{\text{MPNN}}(G_i) = h_G \in \mathbb{R}^d$ for each graph $G_i \in \mathcal{D}$, where θ are the parameters of the MPNN. During training, these parameters evolve over n_{epochs} , and we denote the parameters at epoch l as $\theta^{(l)}$ for $l = 1, 2, \dots, n_{\text{epochs}}$.

The MLP head, denoted $f_{\vartheta}^{\text{MLP}}$, maps the graph embeddings to predictions. The parameters of the MLP, ϑ , are trained jointly with the MPNN. At epoch l , the label for graph $G_i \in \mathcal{D}_{\text{test}}$ is predicted as:

$$\hat{y}_i^{(l)} = f_{\vartheta^{(l)}}^{\text{MLP}}(\phi_{\theta^{(l)}}^{\text{MPNN}}(G_i)).$$

Then, the alignment score is computed on the train set on the epoch l and denoted as:

$$ALI_k^{(l)} = ALI_k(\phi_{\theta^{(l)}}^{\text{MPNN}}, \mathcal{D}_{\text{train}})$$

The accuracy at the epoch l is computed as:

$$\text{Acc}_{\text{test}}^{(l)} = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(G_i, y_i) \in \mathcal{D}_{\text{test}}} \mathbf{1}_{\hat{y}_i^{(l)} = y_i}$$

To assess the relationship between the alignment score and model accuracy, we compute the Pearson correlation coefficient. For two sequences $X = \{X_i\}_{i=1}^n$ and $Y = \{Y_i\}_{i=1}^n$ the Pearson correlation is defined as

$$\rho_{XY} = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2 \sum_{i=1}^n (Y_i - \bar{Y})^2}},$$

where $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$ and $\bar{Y} = \frac{1}{n} \sum_{i=1}^n Y_i$.

In our experiments, we compute two correlation coefficients. The first, ρ_{train} , is obtained by setting $X = [ALI_k^{(l)}]_{l=1}^{n_{\text{epochs}}}$ and $Y = [\text{Acc}_{\text{train}}^{(l)}]_{l=1}^{n_{\text{epochs}}}$, while the second, ρ_{test} , is computed using $Y = [\text{Acc}_{\text{test}}^{(l)}]_{l=1}^{n_{\text{epochs}}}$.

To examine the evolution of the alignment score during training, we trained GNN models using the process described earlier. The training setup is straightforward, employing a classical binary cross-entropy loss, a fixed learning rate of 10^{-4} , and 200 epochs. Figure 1 shows the evolution of the alignment score and accuracy during training on the Mutagenicity dataset with a 3-layer GCN model with a dimension of 128. As expected, both training and test accuracy increase over time, and the alignment score rises in parallel. Although the correlation between the alignment score and accuracy is not strictly maintained during the initial epochs – where test accuracy increases rapidly – the two metrics eventually exhibit an approximately linear relationship, indicating that training the model leads to higher alignment scores. Similar trends have been observed on the NCI1 and COLLAB datasets as well as with the GIN architecture. Overall, the alignment score appears to be a relevant measure that is indirectly optimized during the training of MPNNs.

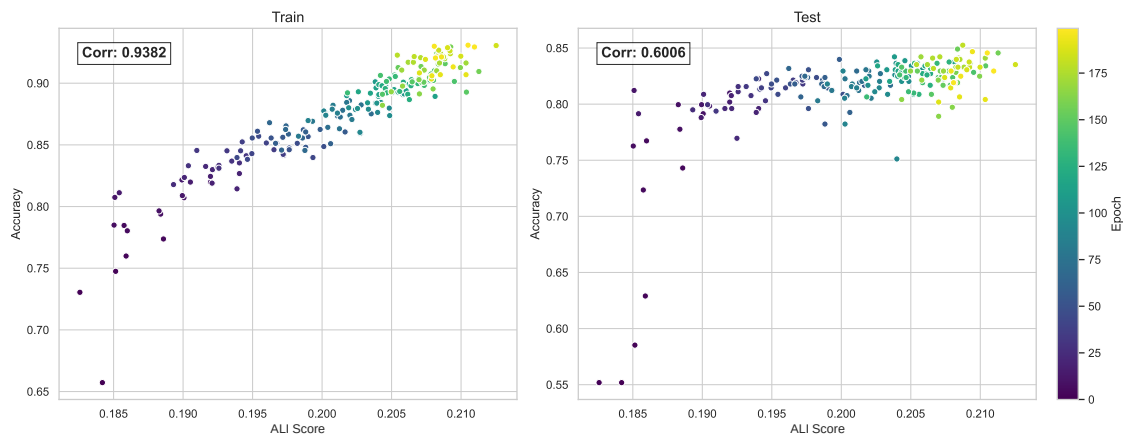


Figure 1: Correlation between ALI_k and accuracy with $k = 10$ during training on the Mutagenicity dataset.

4 Random Sampling

In the previous section, we analyzed the relationship between alignment score and accuracy in trained MPNNs, revealing that both metrics improve simultaneously during training. An intriguing question remains: does this relationship hold in reverse? Specifically, can the alignment score reliably predict the performance of a randomly sampled, untrained MPNN?

4.1 ALI_k -Acc Correlation

In this section, we will first analyze the correlation between the alignment score and accuracy across various datasets and architectures, considering different pooling and initialization methods. Additionally, we will study the distribution of the alignment scores and the corresponding accuracies of the sampled models to evaluate whether this approach can produce competitive models within a reasonable timeframe. We will also estimate the likelihood of obtaining such models.

4.1.1 Sampling Methodology

To thoroughly analyze the correlation in this section, we randomly sample GNN models and, for each model, train an MLP network to obtain a pair (ALI_k, Acc) . Training the MLP is necessary to compute the accuracy score. To ensure consistency, the same MLP architecture is used for all the sampled GNN architectures.

To generate meaningful data, we ensure that the sampled architectures and pooling methods are sufficiently diverse and representative. The sampling method proceeds as follows: we first generate all possible architecture combinations \mathcal{H} based on the hyperparameter variations outlined in Table 2. For each architecture $A \in \mathcal{H}$, we sample random weights for the GNN, compute embeddings for the training set, and calculate the ALI score for various values of k . We then train an MLP on the embeddings to obtain accuracy scores on both the training and test sets. A validation set is used to select the best epoch during training, and final performance is measured by test accuracy. To account for the stochastic nature of MLP training, we train multiple MLPs for each sampled GNN and average the accuracy results. This step ensures that the given accuracy is stable and reliable. Additionally, to obtain a more representative dataset, we repeat the sampling process multiple times for each architecture combination. Details of the sampling algorithm are provided in Algorithm 1.

Different weight sampling initialization methods have been explored in this work. While specific weight initialization is typically employed to stabilize and optimize training convergence during backpropagation, here it is treated as a factor that entirely determines

Hyperparameter	Values
GNN Type	GCN, GIN
Number of Layers	2, 3, 4
Layer Dimension	32, 64, 128, 256
Final Pooling	Sum, Mean
Weight Initialization	Normal, Uniform, Orthogonal
Activation Function	None, ReLU, Tanh, Sigmoid

Table 2: Hyperparameter combinations for GNN experiments.

the GNN’s initial weights and, consequently, its embeddings. Unlike the commonly used Xavier or Glorot initialization [7], which sets a specific variance based on the number of input and output units to improve gradient flow during backpropagation, we focus on three standard distributions for weight initialization: normal, uniform, and orthogonal. Specifically, for the normal and uniform distributions, each element of the weight matrices is independently drawn from the respective distribution, while the orthogonal matrices are sampled following the method detailed in Mezzadri [14]. Since our network does not include any bias terms, and provided that the mean of the distribution is set to zero, varying the variance only scales the resulting embeddings by a constant factor. This scaling does not affect the outcomes of our study since both the alignment score and the accuracy are invariant to uniform scaling of the embeddings. Therefore, the variance of the initialization distribution is fixed in our experiments to isolate the effects of the distribution type (normal, uniform, or orthogonal) on the results.

Algorithm 1: Sampling GNNs and Computing Alignment and Accuracy

Input: Hyperparameter combinations \mathcal{H} , dataset \mathcal{D} , number of MLPs per GNN n_{MLP} , number of samples per architecture n_{samples} , set of k values for ALI

Output: Sets $\mathcal{R}_{\text{train}}$ and $\mathcal{R}_{\text{test}}$ containing pairs $(\text{ALI}_k, \text{AvgAcc})$ for train and test respectively

Initialize result sets: $\mathcal{R}_{\text{train}} \leftarrow \emptyset, \mathcal{R}_{\text{test}} \leftarrow \emptyset;$

foreach *architecture* $A \in \mathcal{H}$ **do**

for $s = 1$ to n_{samples} **do**

 Initialize GNN ϕ with architecture A and random weights $\theta;$

 Compute embeddings: for each $G \in \mathcal{D}$, calculate $\phi_{\theta}(G);$

foreach k in set of k values **do**

 Compute ALI_k using embeddings $\phi_{\theta}(G)$ on $\mathcal{D}_{\text{train}};$

end

 Initialize empty lists: $\text{TrainAccs} \leftarrow [], \text{TestAccs} \leftarrow [];$

for $l = 1$ to n_{MLP} **do**

 Train an MLP f on $\mathcal{D}_{\text{train}}$ using $\phi_{\theta}(G)$ as input;

 Evaluate training accuracy $\text{Acc}_{\text{train}}$ and test accuracy $\text{Acc}_{\text{test}};$

 Append $\text{Acc}_{\text{train}}$ to TrainAccs and Acc_{test} to $\text{TestAccs};$

end

$\text{AvgAcc}_{\text{train}} \leftarrow \frac{1}{n_{\text{MLP}}} \sum_{l=1}^{n_{\text{MLP}}} \text{TrainAccs}[l];$

$\text{AvgAcc}_{\text{test}} \leftarrow \frac{1}{n_{\text{MLP}}} \sum_{l=1}^{n_{\text{MLP}}} \text{TestAccs}[l];$

foreach k in set of k values **do**

 Append $(\text{ALI}_k, \text{AvgAcc}_{\text{train}})$ to $\mathcal{R}_{\text{train}};$

 Append $(\text{ALI}_k, \text{AvgAcc}_{\text{test}})$ to $\mathcal{R}_{\text{test}};$

end

end

end

return $\mathcal{R}_{\text{train}}, \mathcal{R}_{\text{test}};$

4.1.2 Practical Parameter Choices

For the choice of parameters, we arbitrarily selected an intermediate MLP architecture consisting of two layers: one hidden layer with 64 units, which is considered a standard intermediate size for these types of datasets, and an output layer. The MLP employs common practices such as batch normalization, ReLU activation, and no dropout, ensuring a balance between simplicity and effective representation learning. These choices were made to standardize the MLP across experiments while minimizing additional hyperparameter tuning.

For the number of sampled GNNs, we typically generated $n_{\text{samples}} = 10$ random initializations for each architecture. Additionally, for each sampled GNN, we trained $n_{\text{MLP}} = 5$ MLPs to account for variability in MLP training and ensure the results are robust to the randomness of initialization.

An additional observation we made during the experiments was that training the MLP consistently to achieve optimal performance was not straightforward. Due to the small size of the datasets, the variation from one training run to another was quite high. To address this, we employed a cosine learning rate scheduler with a warmup phase, which is known to stabilize the training process and improve convergence [13]. The learning rate $\eta(t)$ at step t is defined as follows:

$$\eta(t) = \begin{cases} \frac{t}{t_{\text{warmup}}} \cdot \eta_{\text{max}} & \text{if } t < t_{\text{warmup}} \\ \frac{\eta_{\text{max}}}{2} \left(1 + \cos \left(\pi \cdot n_{\text{cycles}} \cdot \frac{t - t_{\text{warmup}}}{t_{\text{total}} - t_{\text{warmup}}} \right) \right) & \text{otherwise} \end{cases}$$

Here, η_{max} is the maximum learning rate attained at the end of the warmup phase, t_{warmup} is the number of steps in the warmup phase, t_{total} is the total number of training steps, and n_{cycles} determines the number of cosine cycles applied throughout the training. The warmup phase ensures that the learning rate starts small and gradually increases, while the cosine schedule provides a smooth decay over time, enhancing the stability and convergence of the training process. A small representation of the evolution of the learning rate is plotted in Figure 2.

The number of training steps (or epochs) has been set to 200. The number of warmup steps to 20. Additionally, the learning rate has been set to 10^{-4} . For simplicity, and because additional cycles did not change the results much, the number of cycles has been set to only one.

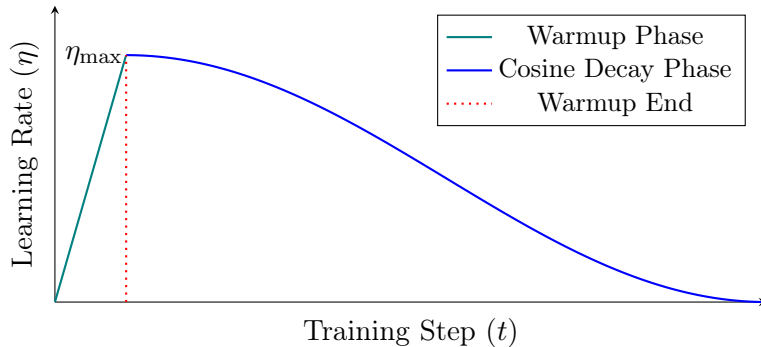


Figure 2: Learning rate schedule with warmup and cosine decay.

4.1.3 Silhouette Score Comparison

Other scores have been proposed in the literature to assess embedding quality. In the clustering field, the silhouette score is commonly used [16].

Definition 7 (Intra-cluster distance and nearest-cluster distance) *Let*

$$\mathcal{G} = \{(G_1, y_1), (G_2, y_2), \dots, (G_n, y_n)\}$$

be our dataset, where each G_i is a graph and $y_i \in \{1, 2, \dots, m\}$ indicates the cluster (or label) to which G_i belongs. Let C_k be the set of all graphs labeled with k . We then have m such clusters: $\{C_1, C_2, \dots, C_m\}$.

We use the previously defined $d_{\text{MPNN}}(G_i, G_j)$, which was introduced as the measure of distance between the MPNN embeddings of G_i and G_j .

- The intra-cluster distance $a(i)$ is the average distance from G_i to all other graphs in its own cluster C_{y_i} :

$$a(i) = \frac{1}{|C_{y_i}| - 1} \sum_{\substack{G_j \in C_{y_i} \\ j \neq i}} d_{\text{MPNN}}(G_i, G_j).$$

- The nearest-cluster distance $b(i)$ is the smallest average distance from G_i to any other cluster C_ℓ (where $\ell \neq y_i$):

$$b(i) = \min_{\ell \neq y_i} \left\{ \frac{1}{|C_\ell|} \sum_{G_j \in C_\ell} d_{\text{MPNN}}(G_i, G_j) \right\}.$$

These definitions tell us how tightly a graph G_i is clustered with others sharing its label, as well as how close it might be to a different cluster.

Definition 8 (Silhouette coefficient and silhouette score) *The silhouette coefficient $s(i)$ for a graph G_i is given by:*

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}},$$

where $a(i)$ and $b(i)$ are defined in the previous definition. This coefficient takes values in $[-1, 1]$, with 1 indicating that G_i is well-clustered in the MPNN embedding space, and -1 suggesting that it may fit better in a different cluster.

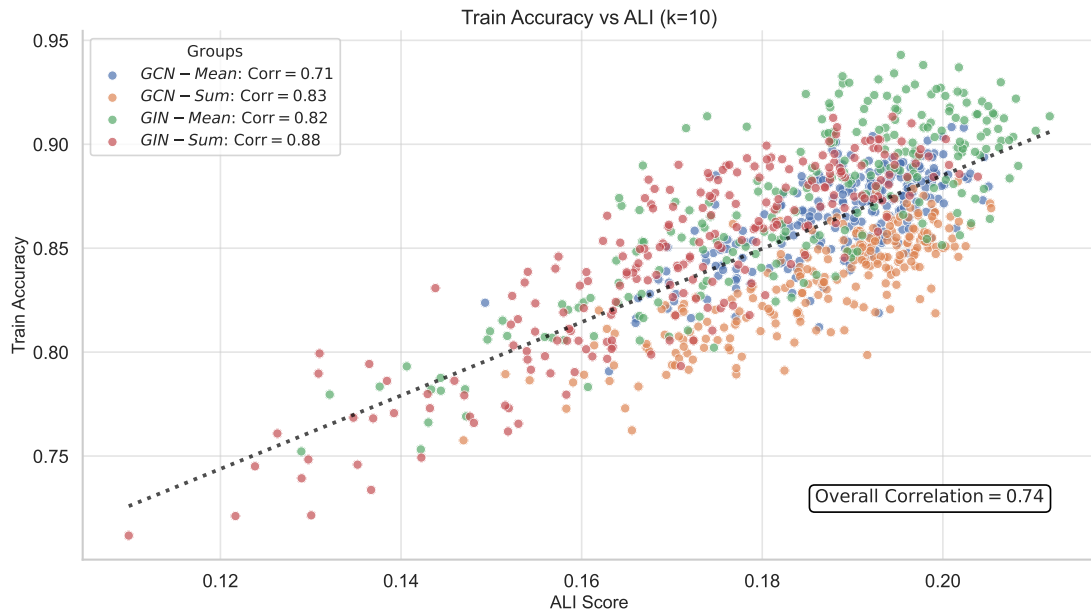
Finally, the silhouette score for the entire dataset is obtained by averaging $s(i)$ over all graphs:

$$\text{Silhouette Score} = \frac{1}{n} \sum_{i=1}^n s(i).$$

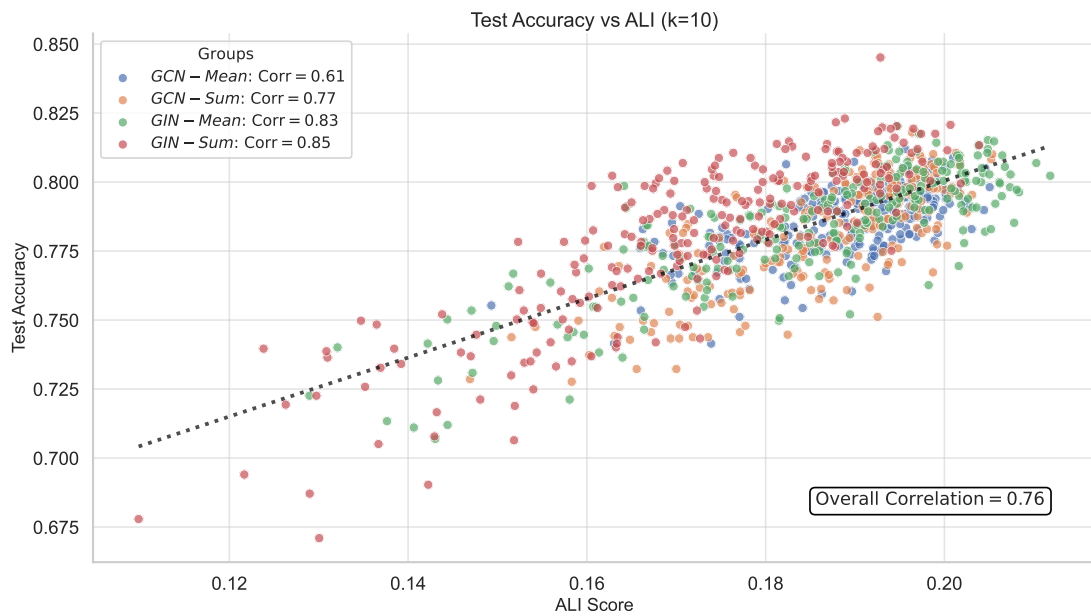
Unlike the previously defined alignment score, which measures local coherence, the silhouette score evaluates the global structure of the data. Achieving a high silhouette score requires well-separated clusters. The silhouette score will serve as a benchmark for comparison with the alignment score.

4.1.4 Correlation Analysis

To investigate the correlation between alignment and accuracy, sampling experiments were conducted across multiple datasets.



(a) Training Set



(b) Test Set

Figure 3: Correlation between ALI_k and accuracy with $k = 10$ on the Mutagenicity dataset.

Figure 3 presents a scatterplot illustrating the relationship between alignment scores (x-axis) and accuracy scores (y-axis) for various sampled GNNs with different architectures (according to Table 2) on the Mutagenicity dataset. Each point in the plot corresponds to a specific sampled GNN, and the points are organized into groups based on different architectural hyperparameter combinations. The plot reveals a strong overall correlation between alignment and accuracy. Moreover, within architecture subgroups, this relationship becomes even more pronounced, indicating that using the alignment score to guide model selection for a specific architecture can yield better performance. This observation aligns with the method’s intended design of sampling the weights of a given GNN architecture and retaining the best model based on alignment score.

Figure 4 shows the results of an experiment analogous to the one in Figure 3, but using the silhouette score instead of the alignment score. The plot does not display any clear correlation between the accuracy and the silhouette score. This observation suggests that the silhouette score may not serve as an effective descriptor for the quality of the embeddings. Additionally, the results indicate that the embedding space produced by the GNN has a local structure, where labels are homogeneous only in nearby regions without forming a consistent global pattern. Therefore, the alignment score is a better indicator, as it considers a specific number of neighbors (defined by the k value), providing a more accurate assessment of the embeddings’ quality.

It is worthwhile to examine how the number of neighbors k influences the correlation, as shown in Table 3. For the considered datasets, the optimal k value seems to be between 5 and 10. Additionally, the correlation decreases as k increases, with the COLLAB dataset showing no correlation at all for $k = 100$. This behavior supports the notion that the structure in the embedding space is more local than global. For all subsequent experiments, the alignment score will be computed using $k = 10$.

Dataset	$k = 5$	$k = 10$	$k = 25$	$k = 100$
Mutagenicity	0.78	0.76	0.72	0.62
NCI1	0.76	0.77	0.74	0.62
COLLAB	0.79	0.76	0.68	-0.05

Table 3: Correlation coefficients on the test set for different k values across datasets.

Figure 5 illustrates the distribution of ALI_k and accuracy for various weight matrix dimensions. The key observation is that lower-dimensional weight matrices exhibit higher variance in the results, whereas higher dimensions lead to more stable and improved outcomes. This behavior can be explained by the concentration of measure phenomenon. In high-dimensional spaces, random fluctuations tend to average out, resulting in most of the data concentrating around a central value. Consequently, larger matrices reduce the impact of randomness, yielding more consistent embeddings across different instances.

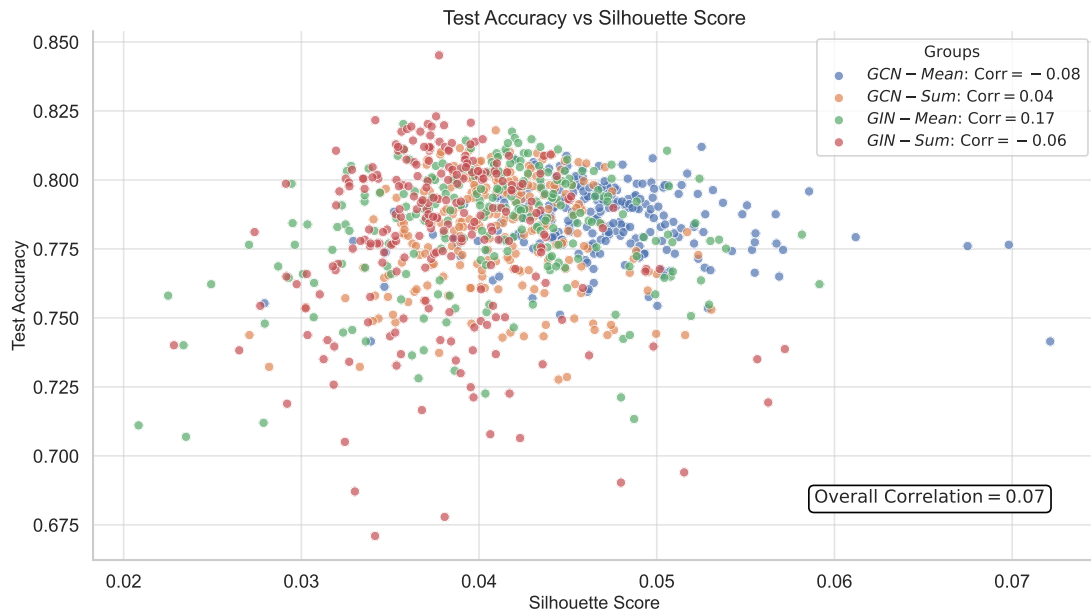
Higher dimensions lead to better results because they allow for a richer and more

detailed encoding of the graph data. In a higher-dimensional space, the model can capture more complex patterns and relationships. During training, the MLP head selects the most useful features from this richer encoding, which further improves performance.

This aligns with observations in Böker et al. [2], where increasing the hidden dimension has been shown to enhance the performance of untrained MPNNs.



(a) Training set

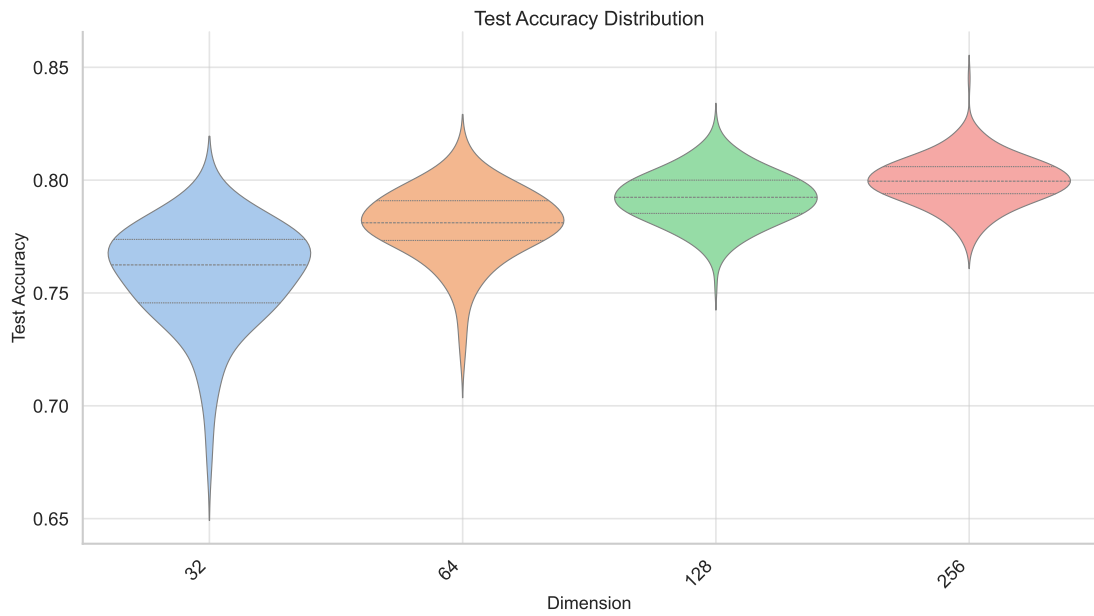


(b) Test set

Figure 4: Correlation between silhouette score and accuracy on the Mutagenicity dataset.



(a) ALI_k distribution on the Training Set



(b) Accuracy Distribution on the Test Set

Figure 5: Distribution by dimension with $k = 10$ on the Mutagenicity dataset.

4.2 Best Model Sampling and Baseline Comparison

Having established the correlation between the alignment score and accuracy, we now apply this insight to address our initial motivation by sampling weights, retaining the best one based on the alignment score, and evaluating how it performs against a conventionally trained GNN.

4.2.1 Methodology for Best Sampling

The sampling method is the same as described in the previous chapter; however, this time we retain the model with the highest alignment score. Typically, the sampling process is stopped after a predetermined number of models have been sampled. Alternatively, one can set a threshold for the ALI score and select the first model that exceeds this threshold. Once the best GNN model is identified, we train an MLP to evaluate its performance, following the same procedure as in the previous section.

A secondary method to select and evaluate the best model was also explored. In this variant, the top 10 models (based on alignment score) are retained, and an MLP head is trained on each. The final selection then depends on each model’s validation accuracy. Although this method combines both alignment and accuracy for model selection, it showed no substantial improvements in practice and was therefore not used.

A common cross-validation method was adopted, where the data are split into n folds, five in this case. The model is trained (or sampled) on $n - 1$ folds, and the last fold is split into a validation set and a test set to evaluate the model. This process is repeated for each fold, resulting in n different performance scores. These scores are then averaged to obtain the final performance estimate, while the variation among them provides an estimate of uncertainty.

4.2.2 Influence of Architecture and Sampling Method on Performance

The previously described method for sampling the best model was applied to different architectures and weight sampling strategies, and we will compare their performance here. The architectural combinations are those in Table 2, with a small change: only the ReLU function was used, since it performed significantly better across all architectures and datasets. For this experiment, 1000 models were sampled, and the best one was selected based on the ALI score with $k = 10$.

Figures 6, 7, 8 and 9 compare the weight sampling methods for different hidden dimensions across all datasets. Each model architecture that was sampled and tested is grouped according to its hidden dimension and weight sampling method. The average test accuracy is reported, along with its standard deviation indicated by error bars. No single sampling method stands out as significantly superior. Within each subgroup of the same hidden

dimension, the average and standard deviation of the test accuracy remain similar across the different sampling methods. Regarding the influence of the hidden dimension, larger dimensions again show better performance, which is consistent with earlier observations. This outcome is coherent with the idea that a larger hidden dimension can encode more information. Moreover, the standard deviation decreases with bigger dimensions, indicating more consistent performance. For the COLLAB dataset, results are divided between two figures: Figure 8 for the GCN architecture and Figure 9 for the GIN architecture. This separation is necessary because, unlike the NCI1 and Mutagenicity datasets, the GCN and GIN models behave very differently on COLLAB. For GCN, the performance is consistent with the trends observed on Mutagenicity and NCI1, but for GIN, it is significantly lower with a notably higher standard deviation. As we will see in Figure 13, this variability is closely related to the number of layers in the GIN architecture.

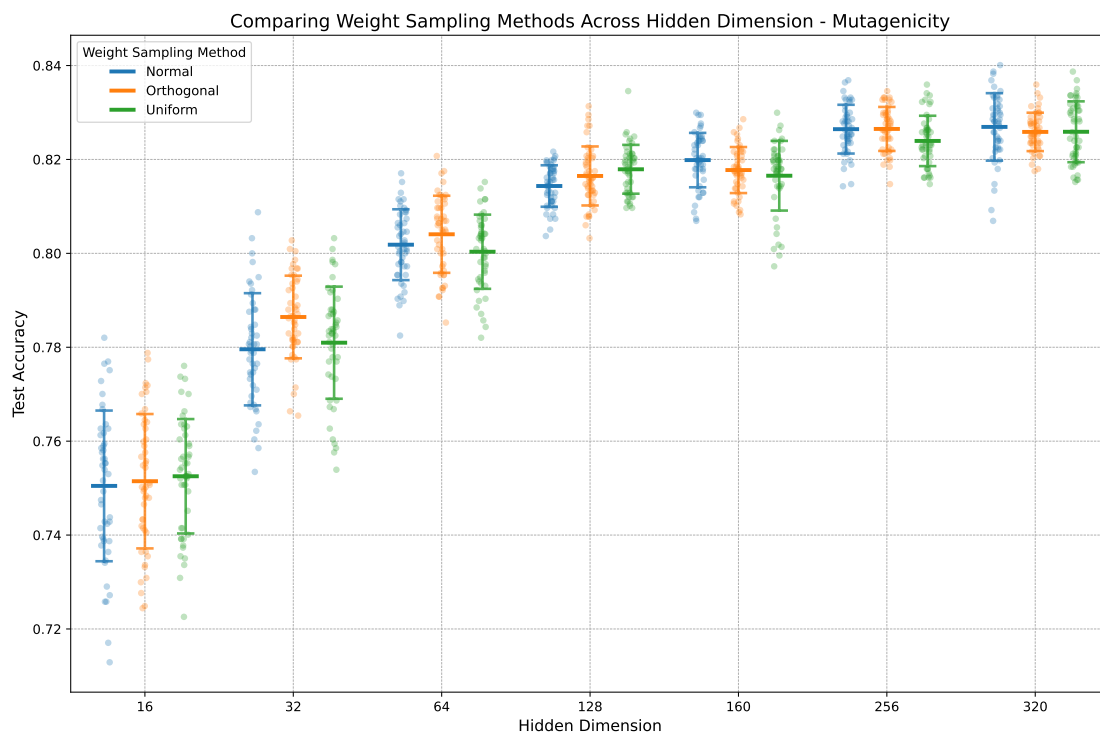


Figure 6: Accuracy distribution of different weight sampling techniques across hidden dimensions on the Mutagenicity dataset.

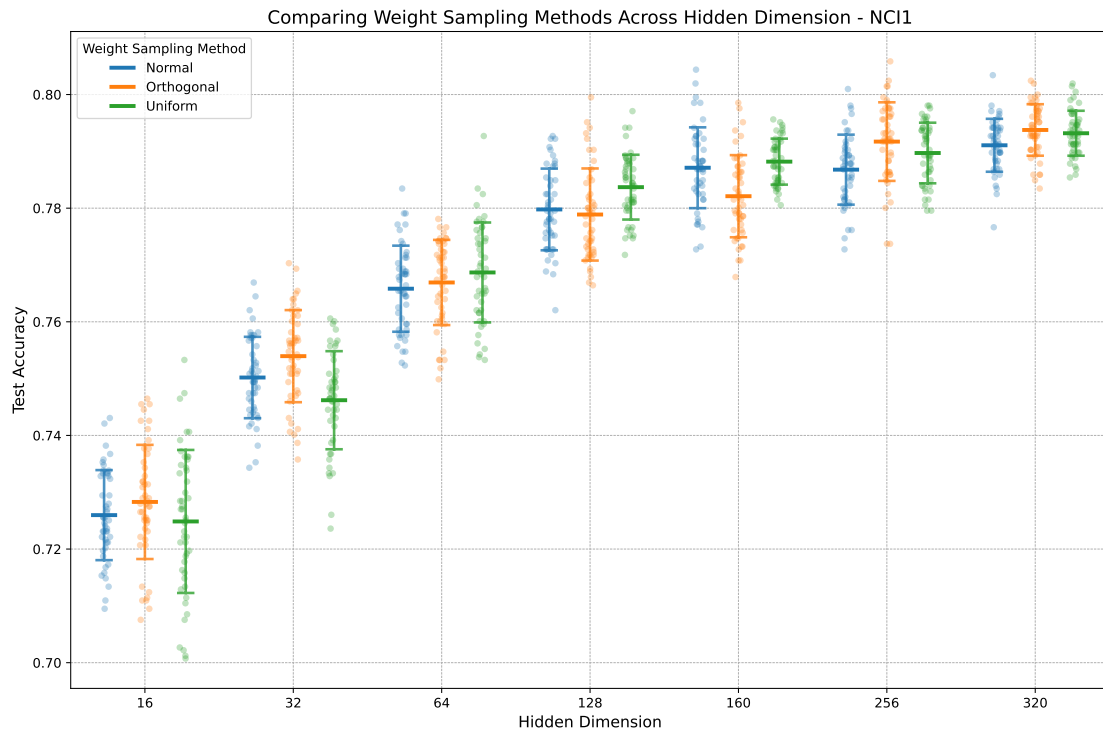


Figure 7: Accuracy distribution of different weight sampling techniques across hidden dimensions on the NCI1 dataset.

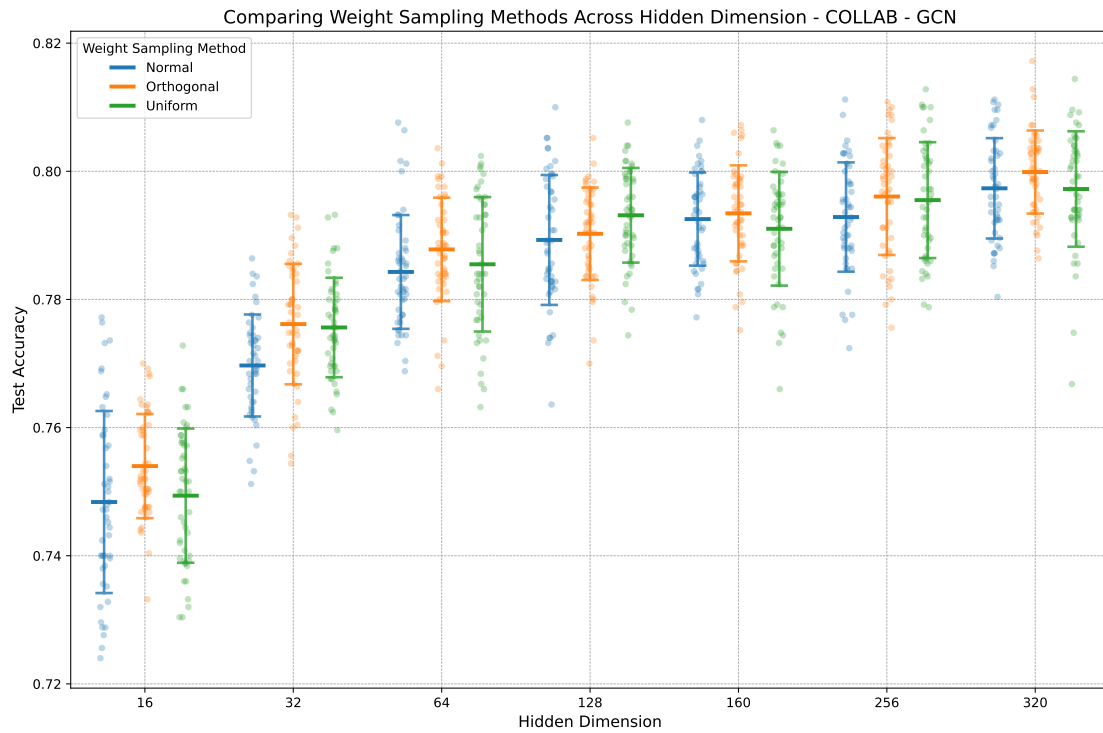


Figure 8: Accuracy distribution of different weight sampling techniques across hidden dimensions for GCN on the COLLAB dataset.

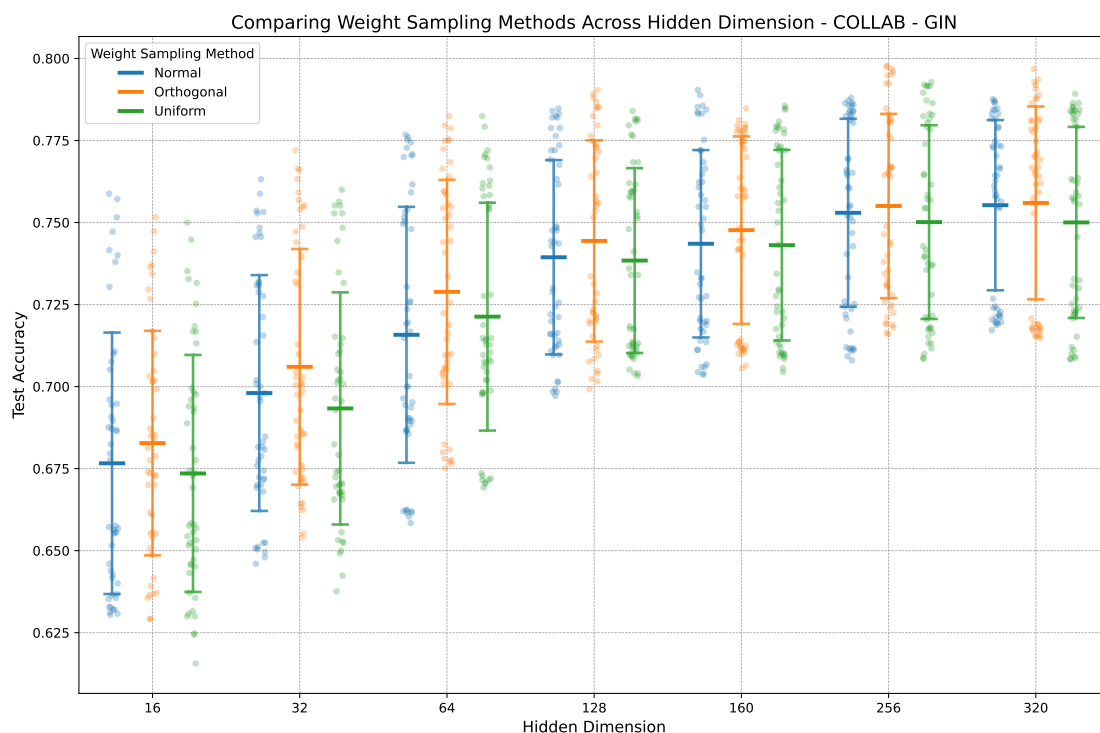


Figure 9: Accuracy distribution of different weight sampling techniques across hidden dimensions for GIN on the COLLAB dataset.

Figures 10, 11, 12 and 13 present the performance comparison by number of layers across various hidden dimensions, reflecting the same experimental results as in the previous plots. For NCI1 and Mutagenicity (Figures 10 and 11), shallow models initially outperform deeper ones when the hidden dimension is small. However, as the hidden dimension increases, this gap narrows, and the different layer depths achieve nearly the same average performance. In contrast, for the COLLAB dataset (Figures 12, 13), a noticeable performance gap persists across all hidden dimension sizes, and it is even more pronounced for the GIN architecture. This large difference in performance across different numbers of layers also explains the high standard deviation observed in Figure 9. Consequently, COLLAB appears more sensitive to the depth of the model.

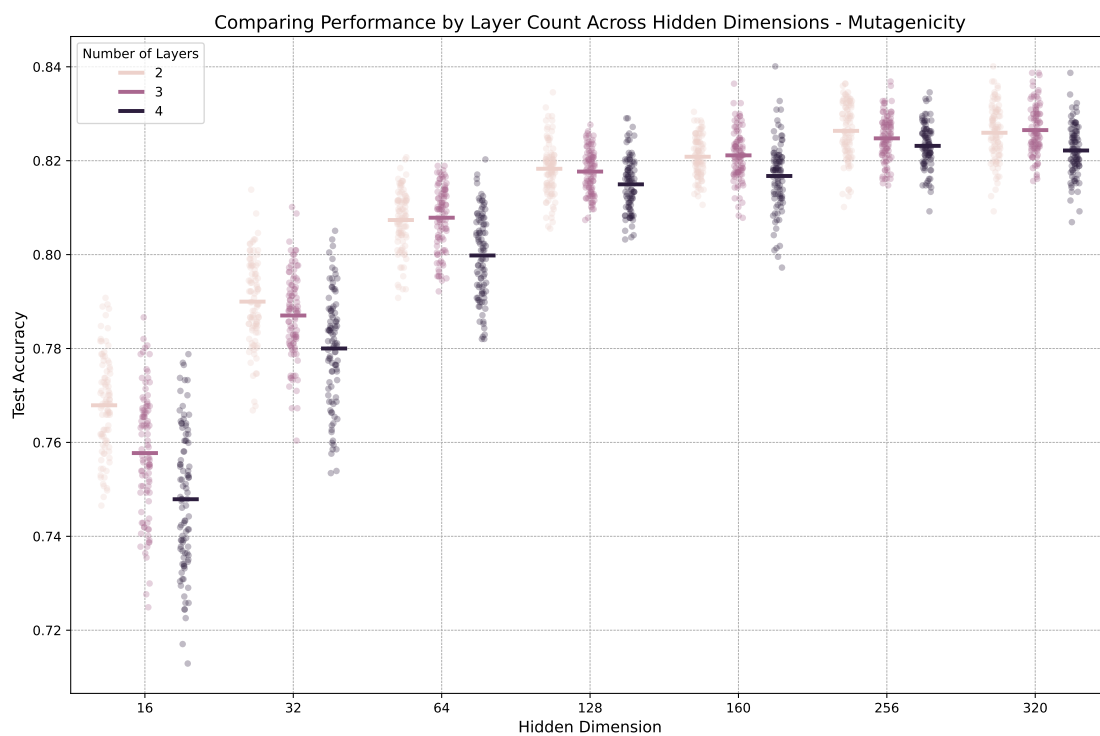


Figure 10: Accuracy distribution for different numbers of layers across hidden dimensions on the Mutagenicity dataset.

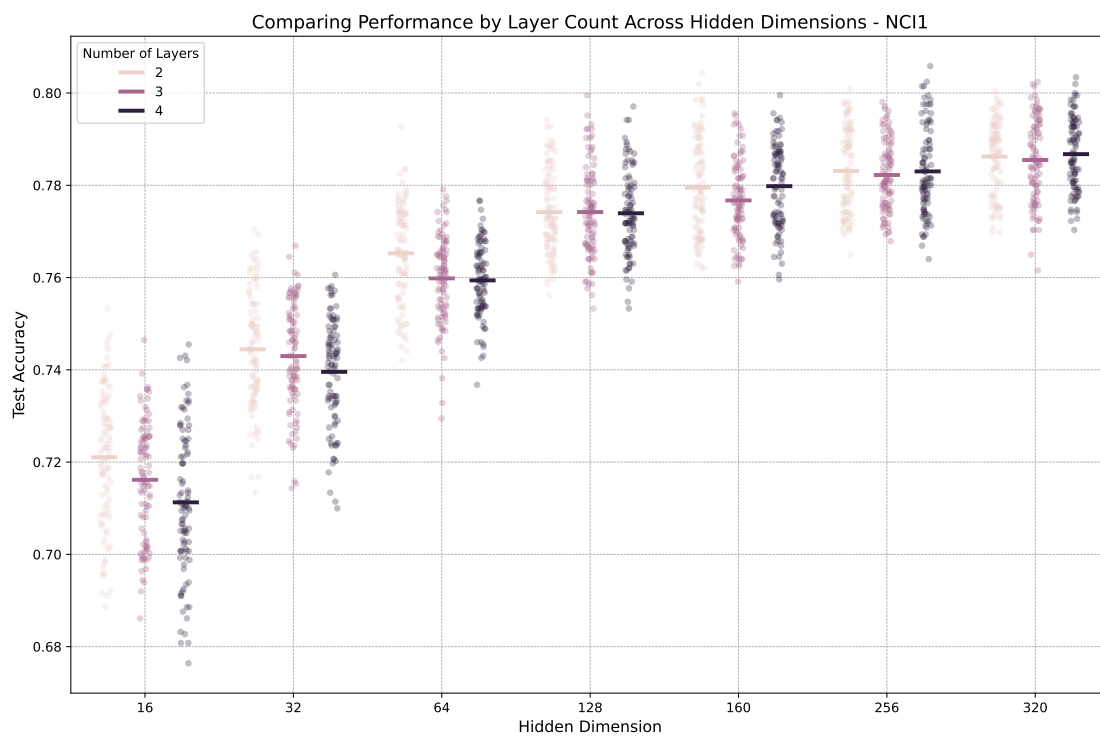


Figure 11: Accuracy distribution for different numbers of layers across hidden dimensions on the NCI1 dataset.

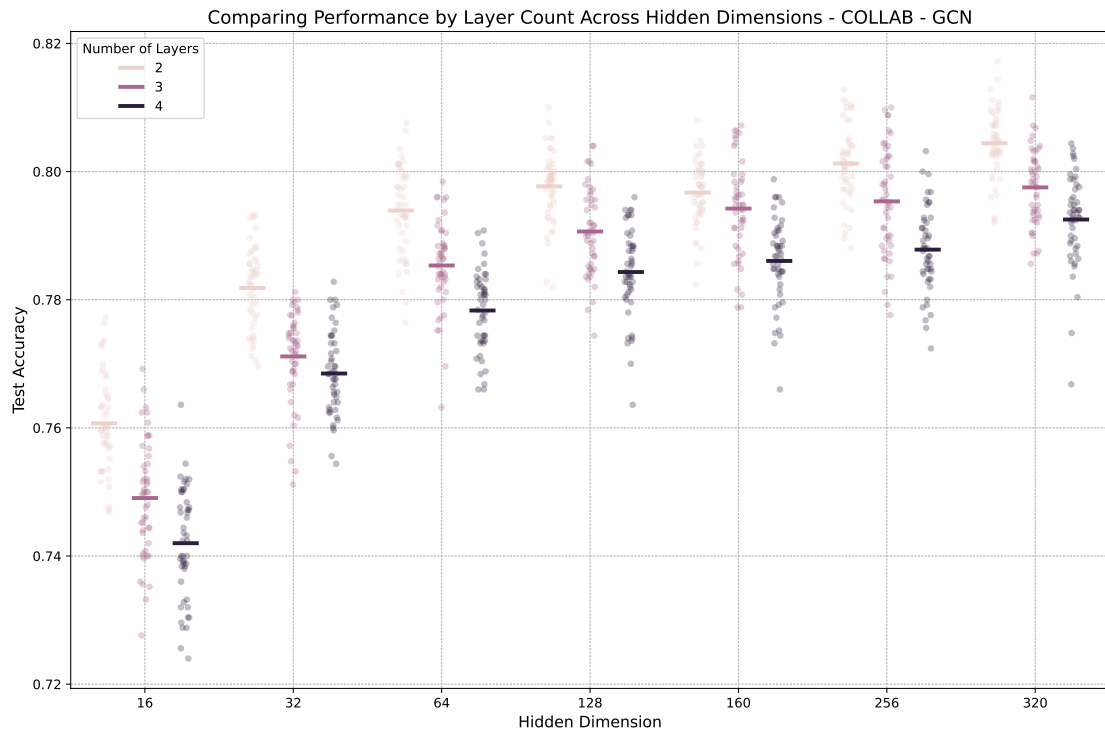


Figure 12: Accuracy distribution for different numbers of layers across hidden dimensions for GCN on the COLLAB dataset.

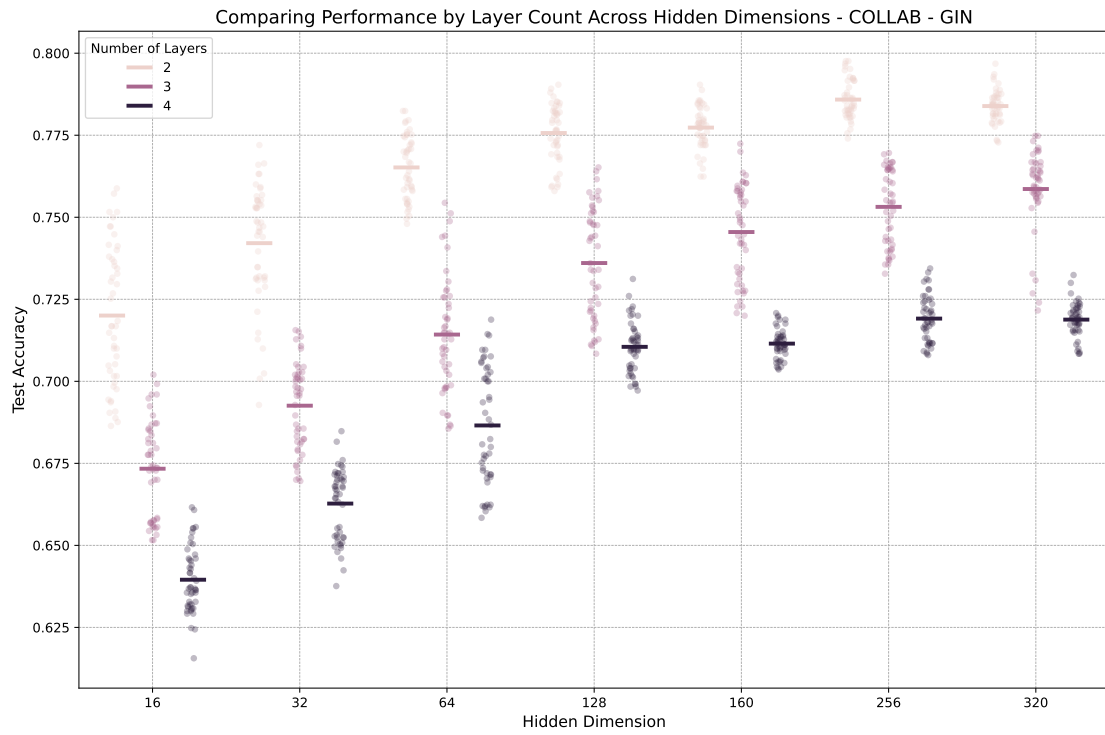


Figure 13: Accuracy distribution for different numbers of layers across hidden dimensions for GIN on the COLLAB dataset.

4.2.3 Stochastic Gradient Descent Baseline

The results of the sampling method are compared against a fully trained baseline, meaning both the MPNN and MLP weights are trained. For each hyperparameter combination, the corresponding architecture is trained using SGD for 400 epochs, applying the same cosine scheduler and warmup procedure as in the MLP training. The same evaluation method is employed, meaning we use a validation set to select the best epoch, followed by testing on a separate test set. Furthermore, the cross-validation process remains identical to that of the sampling method, using the same random seed to generate exactly the same folds, which ensures a fair comparison. Because of the cosine scheduler, varying the maximum learning rate within a reasonable range did not substantially affect the final performance, so it was fixed at 10^{-3} for all architectures. The scheduler also yielded higher performance than typically reported in literature benchmarks [5], but using it was deemed fair, as it was already employed for the MLP training in the sampling method. This difference should be considered when comparing performances.

4.2.4 Performance Comparison with Stochastic Gradient Descent

To evaluate the performance of traditional SGD-based training against the sampling method, a GNN model was fully trained for each architecture considered in the previous experiment. This enables a direct comparison of both methods for each architecture combination across all three datasets. To compare them exhaustively, we calculated the pairwise difference in test accuracy for each model, then averaged these differences over different hidden dimensions and layer counts. Figures 14, 15, 16 and 17 report the results, displaying the mean and standard deviation for each architecture subgroup.

The main observation is that, across all datasets, the performance gap between the SGD baseline and the sampling method decreases as the hidden dimension grows. In other words, while the fully trained model outperforms the sampling method at smaller dimensions, this difference narrows with larger dimensions. Indeed, the baseline proves more consistent across various dimensions. Although both methods benefit from increasing dimensionality, the incremental gains for the baseline become less pronounced at higher dimensions. Another notable finding, which aligns with the previous results, is that the sampling method becomes more competitive with the baseline when using shallower architectures, but this advantage diminishes with larger hidden dimensions. On the Mutagenicity dataset (Figure 5), for example, the sampling method becomes competitive starting at a dimension of 64 and appears to slightly outperform the baseline at 256 and 320. However, for NCI1 and COLLAB with a GCN architecture (Figures 15 and 16), the sampling approach still underperforms the baseline slightly, even at higher dimensions. Finally, for COLLAB with a GIN architecture (Figure 17), the sampling method cannot match the baseline’s performance, particularly in deeper configurations.

One possible explanation for the drop in performance observed in deeper architectures is that, because our sampled network remains random, it becomes increasingly difficult to propagate meaningful information through many layers. In contrast, stochastic gradient descent selectively propagates only the most relevant information. Additionally, the high performances observed for larger dimensions may be attributed to their ability to encode bigger and richer representations, which increases the likelihood of capturing meaningful features.

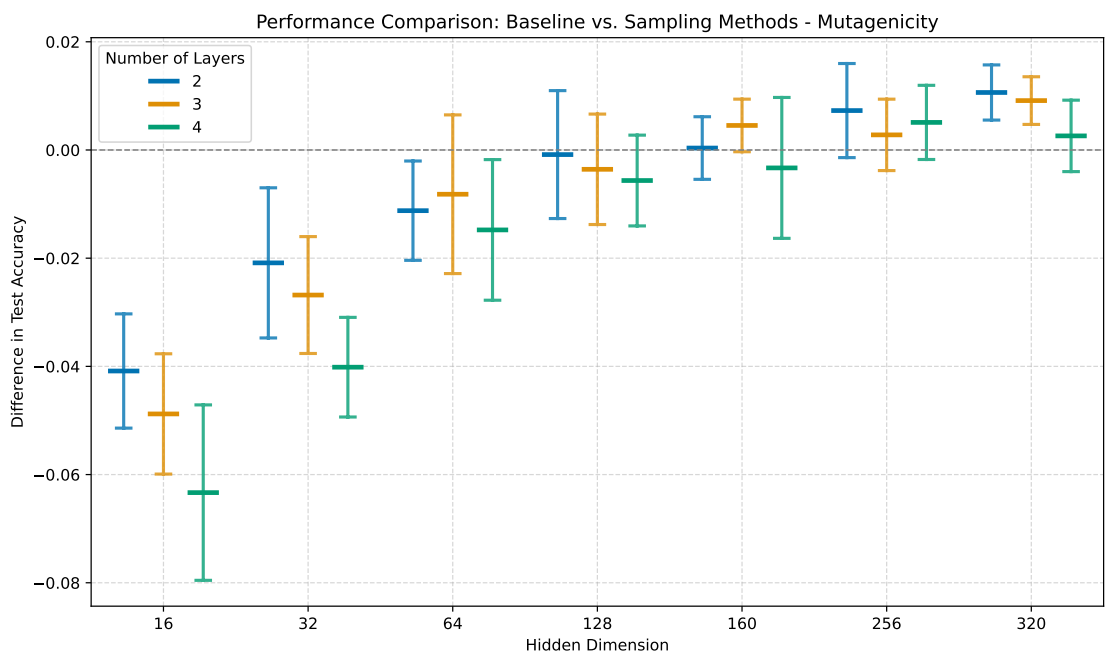


Figure 14: Pairwise accuracy difference between baseline and sampling methods across hidden dimensions and layers on the Mutagenicity dataset.

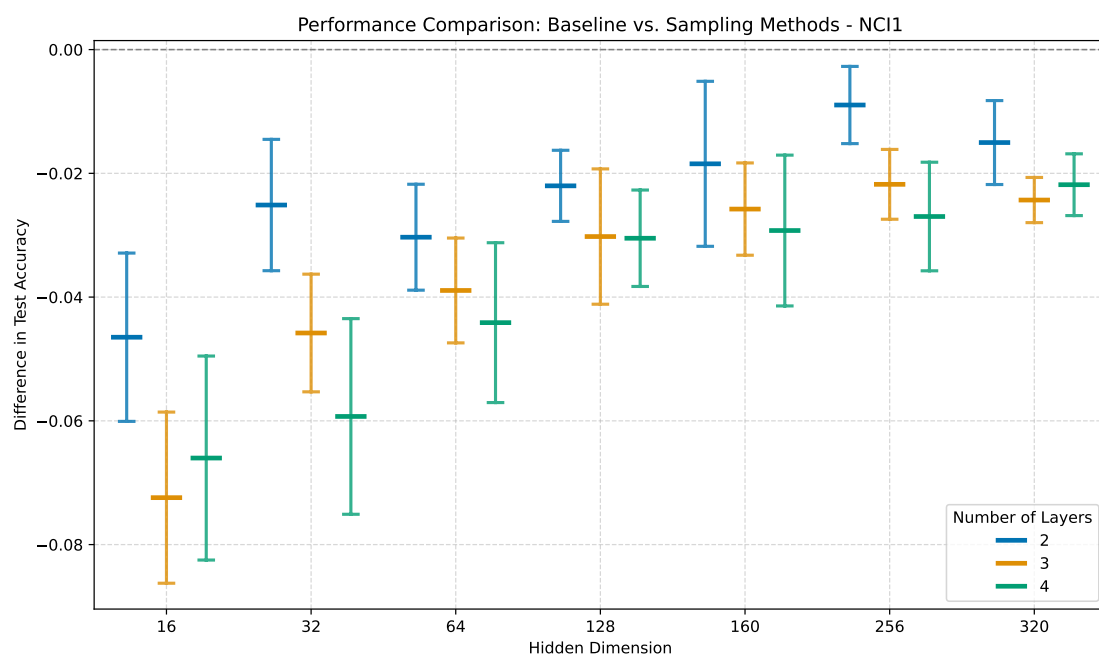


Figure 15: Pairwise accuracy difference between baseline and sampling methods across hidden dimensions and layers on the NCI1 dataset.

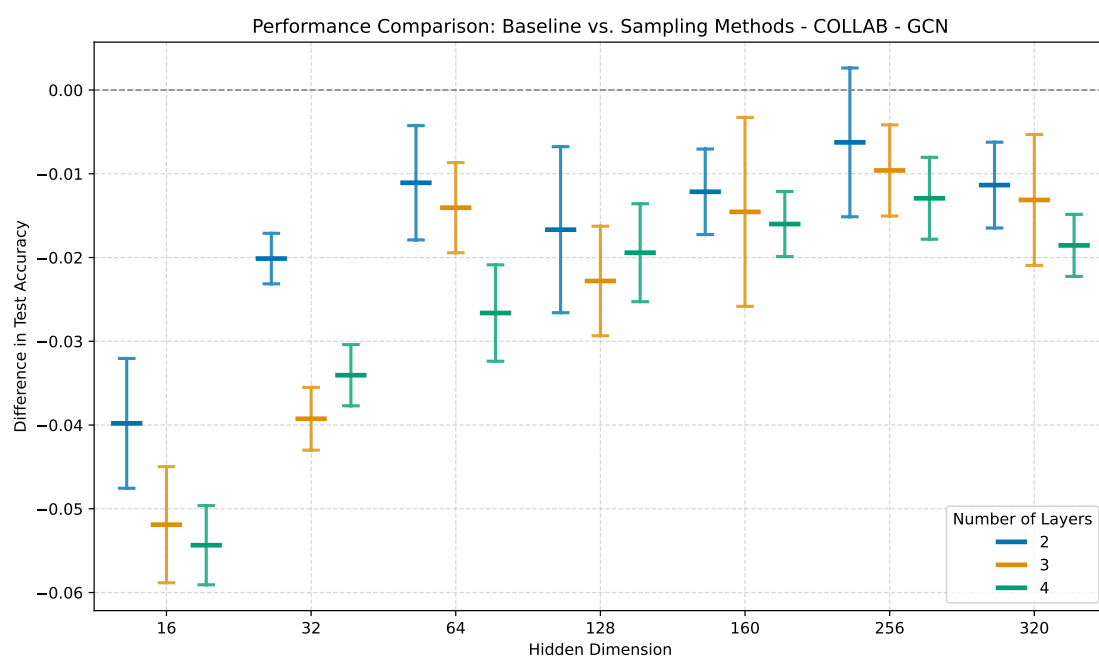


Figure 16: Pairwise accuracy difference between baseline and sampling methods across hidden dimensions and layers for GCN on the COLLAB dataset.

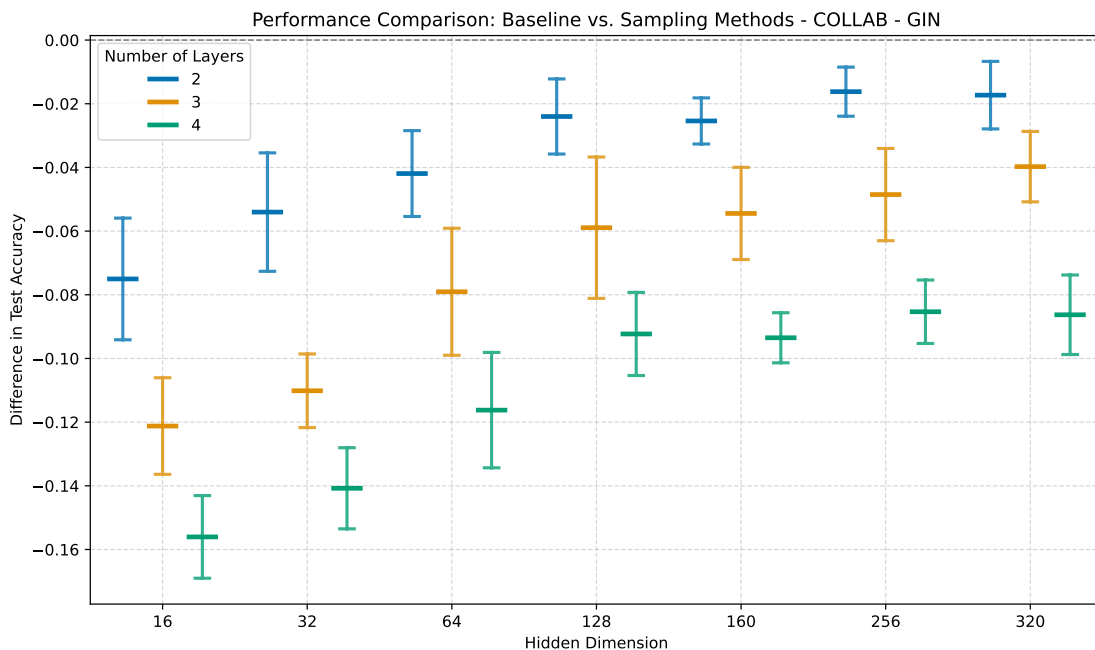


Figure 17: Pairwise accuracy difference between baseline and sampling methods across hidden dimensions and layers for GIN on the COLLAB dataset.

4.2.5 Training Time Analysis

An important factor is the time required to train a model. Typically, with stochastic gradient descent, as model complexity increases, both the training duration and the memory usage for storing parameter gradients also rise. For the random sampling method, the most computationally intensive step is sampling the MPNNs before selecting the best model. On our computer, using the Mutagenicity dataset, sampling and evaluating a single MPNN takes about one-twentieth of a second for the smallest models and roughly one-tenth of a second for the largest architectures. This increase is due to both extended message passing and additional alignment computations for larger embeddings. Similar sampling times are observed for the NCI1 dataset, as the graphs and overall dataset are of comparable size, while for the COLLAB dataset, sampling takes between half a second and one second because message passing requires more time on larger graphs with significantly more edges. The MLP training is very fast, as it is shallow and only trains on the embeddings; for 200 epochs, it roughly takes five seconds.

The overall runtime of the method depends heavily on the number of iterations. We chose 1000 iterations, which results in a total execution time roughly equivalent to training the SGD-based model for the same architecture for 400 epochs. It would be beneficial to further analyze how accuracy evolves with the number of iterations; a preliminary experiment on this aspect is presented in the next chapter.

5 Random Ensemble Sampling

The random sampling method has demonstrated promising and competitive results compared to the SGD baseline, yet it still underperforms on some datasets, especially with deeper architectures. To address these issues, we propose a novel approach that combines the previous sampling method with ensemble techniques. Ensemble methods combine multiple weak or base learners to form a single, more robust model by ensuring that each learner is as independent as possible – typically by training them on distinct subsets of the training data. This strategy helps reduce overfitting and variance, ultimately leading to improved accuracy and stability on unseen data. In our approach, we will combine several small, independently sampled GNN models into a larger ensemble, with the aim of achieving superior performance.

5.1 Ensemble-Sampled MPNNs

In this section, we motivate and detail our proposed ensemble sampling approach. As shown in Figure 5, smaller sampled networks show higher variance in terms of performance, whereas larger networks generally achieve better performance. This observation suggests that the inherent performance variability of smaller networks can be fully exploited by sampling a large number of them. By selecting those that perform best, we can combine these top models into a single, larger ensemble that integrates the diverse strengths of the individual networks while achieving improved overall performance.

5.1.1 Details on the Ensemble Method Algorithm

The proposed method samples p independent small GNNs and combines their representations into a single, larger ensemble. Let each small network be denoted by ϕ_i , where $i \in \{1, 2, \dots, p\}$. For a given graph G , each ϕ_i produces an embedding vector $h_{G,i}$. These embeddings are concatenated to form the final graph embedding:

$$h_G = [h_{G,1} \parallel h_{G,2} \parallel \dots \parallel h_{G,p}],$$

where \parallel indicates the concatenation operator.

This concatenation can also be interpreted as creating a single, sparse network. Specifically, each small network ϕ_i has its own set of weight matrices $\{W_i^{(l)}\}_l$ for each layer l . To merge these matrices into one larger network, they are placed along the diagonal of a

block-diagonal matrix $\mathbf{W}^{(l)}$, defined as

$$\mathbf{W}^{(l)} = \begin{bmatrix} W_1^{(l)} & 0 & 0 & \cdots & 0 \\ 0 & W_2^{(l)} & 0 & \cdots & 0 \\ 0 & 0 & W_3^{(l)} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & W_p^{(l)} \end{bmatrix},$$

where:

- p is the total number of sampled small GNNs,
- $W_i^{(l)}$ is the weight matrix of the i th network at layer l ,
- $\mathbf{W}^{(l)}$ is the resulting block-diagonal matrix for layer l .

It should be noted that in a GIN architecture, the MLP component within each layer is defined using two weight matrices; however, this detail does not affect the aggregation process.

This block-diagonal structure preserves the independence of each network’s parameters while unifying them into a single architecture. As a result, the ensemble retains the diversity of the individual small networks, yet it operates as one GNN with sparse weight connections. Importantly, instead of requiring p times the inference time to compute p separate embeddings, this unified structure achieves nearly the same inference time as a single network, since message propagation is performed only once across the entire ensemble.

For the final classification or regression stage, rather than training p separate MLPs – one for each individual embedding – and then aggregating their predictions via voting, we propose training a single MLP on the combined, enriched graph representations from the p small networks. This approach captures the diverse information from each model while significantly reducing computational costs.

The implementation of the ensemble sampling is detailed in Algorithm 2. The algorithm operates in two main phases. In the first phase, we iteratively sample random MPNN models and compute their alignment scores, maintaining a list of the top p performers. Once sampling is complete, these top models are merged into a single ensemble GNN by constructing block-diagonal weight matrices for each layer. In the second phase, each graph in the dataset is processed through the merged network to obtain a concatenated embedding. These embeddings, together with their corresponding labels, are then used to train a MLP, whose test accuracy serves as the final performance metric for the ensemble method.

Algorithm 2: Ensemble Sampling Algorithm

Input: $n_{\text{iterations}}$, p , dataset \mathcal{D}

Output: Acc_{test}

Initialize an empty list: $\mathcal{M}_{\text{top}} = []$

for $i \leftarrow 1$ **to** $n_{\text{iterations}}$ **do**

 Sample random weights to create a new GNN ϕ

 Compute alignment score $\text{ALI}(\phi)$

if $\text{ALI}(\phi)$ is better than at least one model in \mathcal{M}_{top} **then**

if $|\mathcal{M}_{\text{top}}| = p$ **then**

 Remove the model with the worst alignment score from \mathcal{M}_{top}

end

 Insert ϕ into \mathcal{M}_{top}

end

end

Build the merged GNN Φ :

Let L be the number of layers in each GNN.

Initialize Φ .

for $l \leftarrow 1$ **to** L **do**

 Construct a block-diagonal matrix $\mathbf{W}^{(l)}$ from the l^{th} layer weights of all models $(W_1^{(l)}, W_2^{(l)}, \dots, W_p^{(l)})$ in \mathcal{M}_{top}

 Set the l^{th} layer of Φ to $\mathbf{W}^{(l)}$

end

Compute embeddings and train MLP:

foreach $G \in \mathcal{D}$ **do**

 Compute the graph embedding $h_G = \Phi(G)$

 Store the resulting embedding h_G along with its corresponding label

end

Train a MLP using the set of embeddings $\{h_G\}$ and their associated labels

Evaluate the trained MLP on the test set to compute the test accuracy Acc_{test}

return Acc_{test}

5.1.2 Results for the Ensemble Method

To evaluate the performance of the ensemble sampling method, we used the same benchmark as before, a five-fold cross-validation on each dataset. For each fold, 1000 smaller models were sampled, and the ensemble model was constructed by aggregating the top ten performers. We set the hidden dimensions for the individual submodels to 16 and 32. This results in ensemble models with overall dimensions of 160 and 320, respectively, ensuring that the final model size remains reasonable. For the remaining hyperparameters, we adopted the configurations from Table 2, using the ReLU activation function.

As observed previously, the choice of weight sampling method does not significantly affect the results. Figure 18 compares the performance of each weight sampling method on the Mutagenicity dataset. As before, each model architecture is grouped according to its hidden dimensions, with the mean and standard deviation reported. The results for the NCI1 and COLLAB datasets are nearly identical, indicating no significant differences.

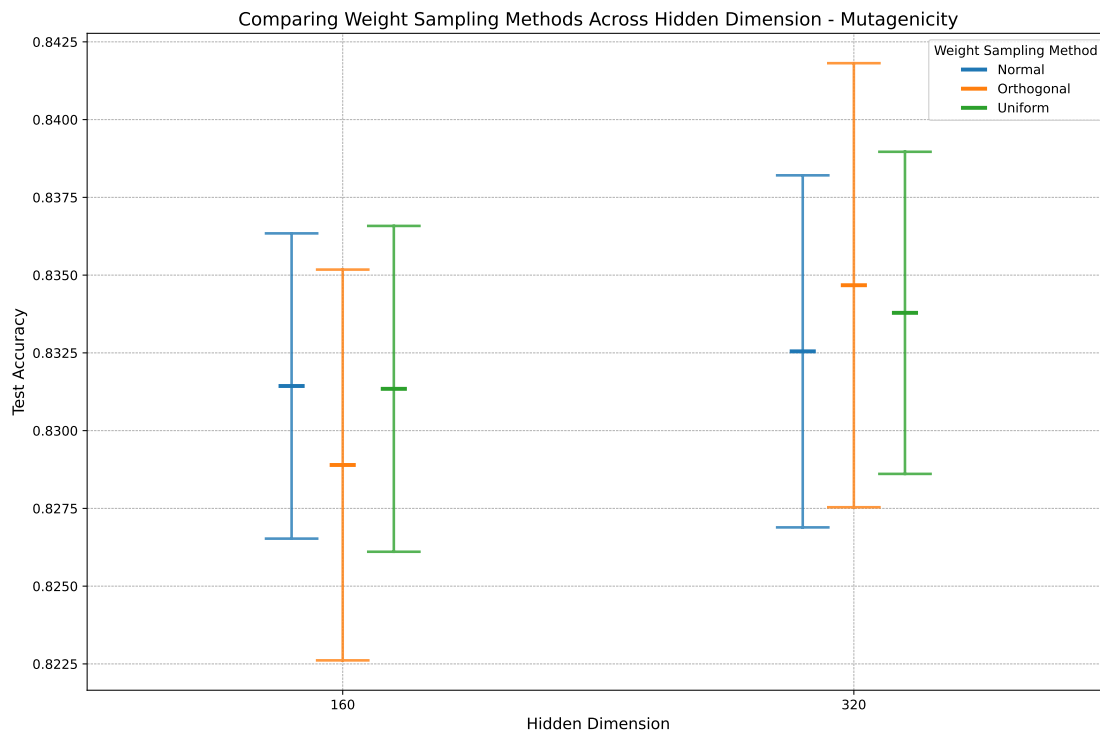


Figure 18: Accuracy distribution of different weight sampling techniques across hidden dimensions on the Mutagenicity dataset.

To compare this ensemble sampling method against the SGD baseline, we have used the same previously trained baseline for each architecture combination. It is important to note that for each hidden dimension, the ensemble method is compared against a baseline whose weight matrices have the same dimensions but no sparsity constraints. Since the ensemble model employs highly sparse weight matrices, it has significantly fewer parameters.

Figures 19, 20, 21 and 22 compare performance using the same pairwise differences in test accuracy as in the previous chapter. Overall, there is a noticeable improvement compared to the earlier random sampling method, although shallower models still tend to be more competitive relative to the baseline. For the Mutagenicity dataset, in Figure 19, the ensemble method performs significantly better across all hidden dimensions and layer counts. In particular, the ensemble model with an overall hidden dimension of 160, previously matching the baseline, now outperforms it. For the NCI1 dataset (Figure 15), the 2-layer ensemble model now equals the performance of the SGD baseline, while deeper architectures remain slightly behind yet are clearly competitive. For the COLLAB

dataset (Figures 21 and 22), the difference between the GCN and GIN architectures persists. Although the ensemble method is competitive with the SGD baseline for the GCN architecture, only the 2-layer GIN model can challenge the baseline, with deeper GIN architectures continuing to lag behind, as seen with the random sampling method. Overall, while the general trends remain consistent, the ensemble method has improved results by approximately 0.5–1% compared to the random sampling approach.

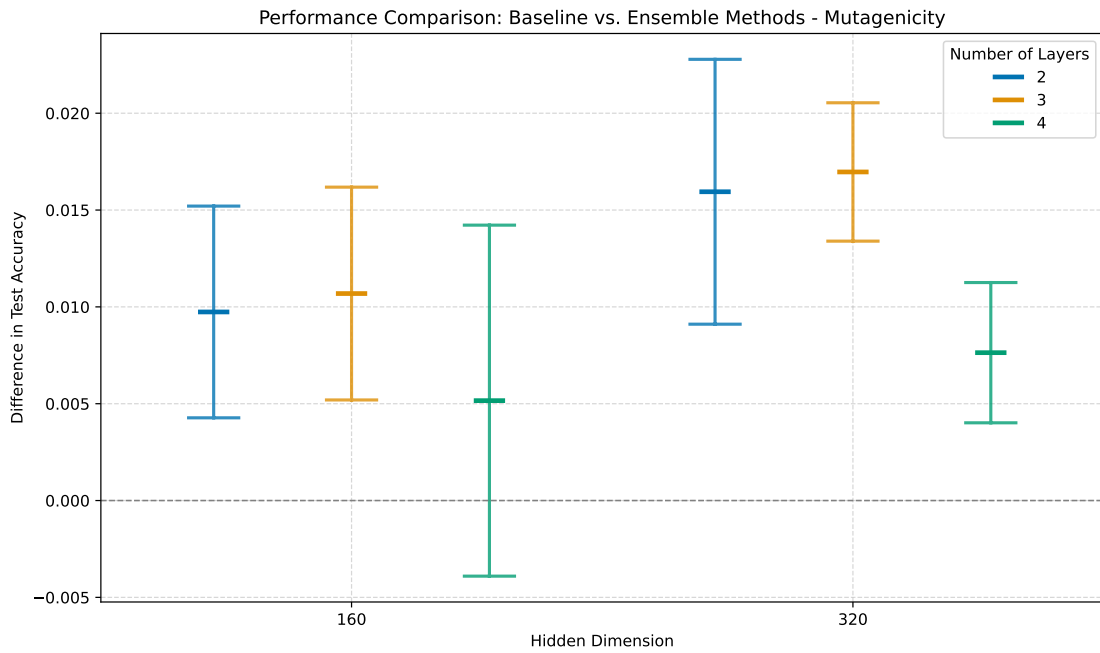


Figure 19: Pairwise accuracy difference between baseline and ensemble sampling methods across hidden dimensions and layers on the Mutagenicity dataset.

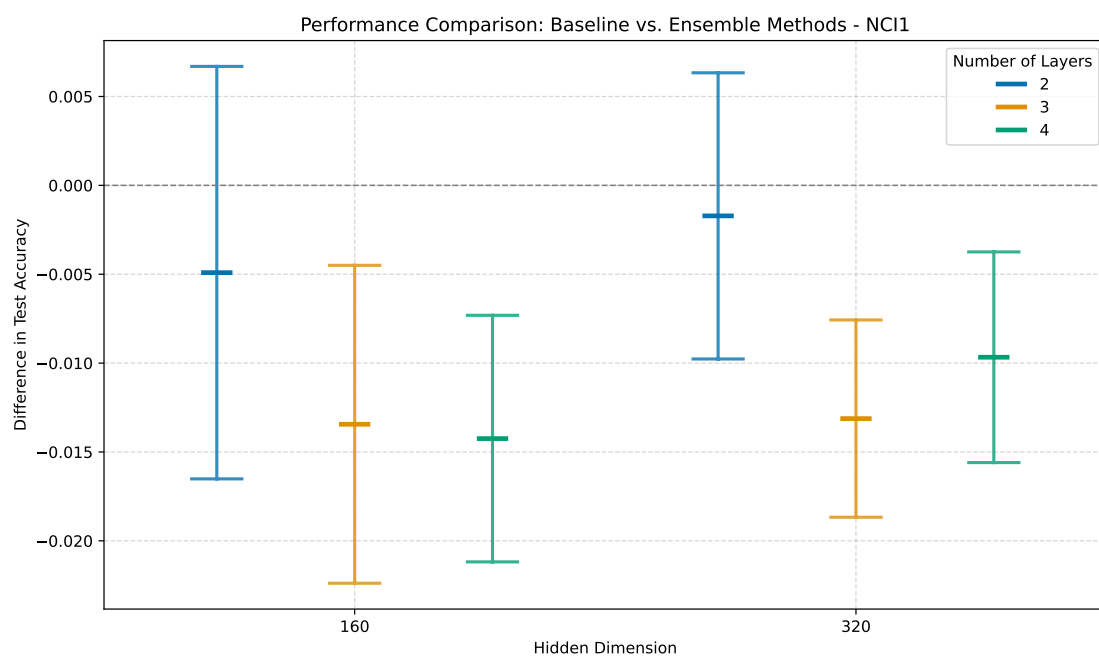


Figure 20: Pairwise accuracy difference between baseline and ensemble sampling methods across hidden dimensions and layers on the NCI1 dataset.

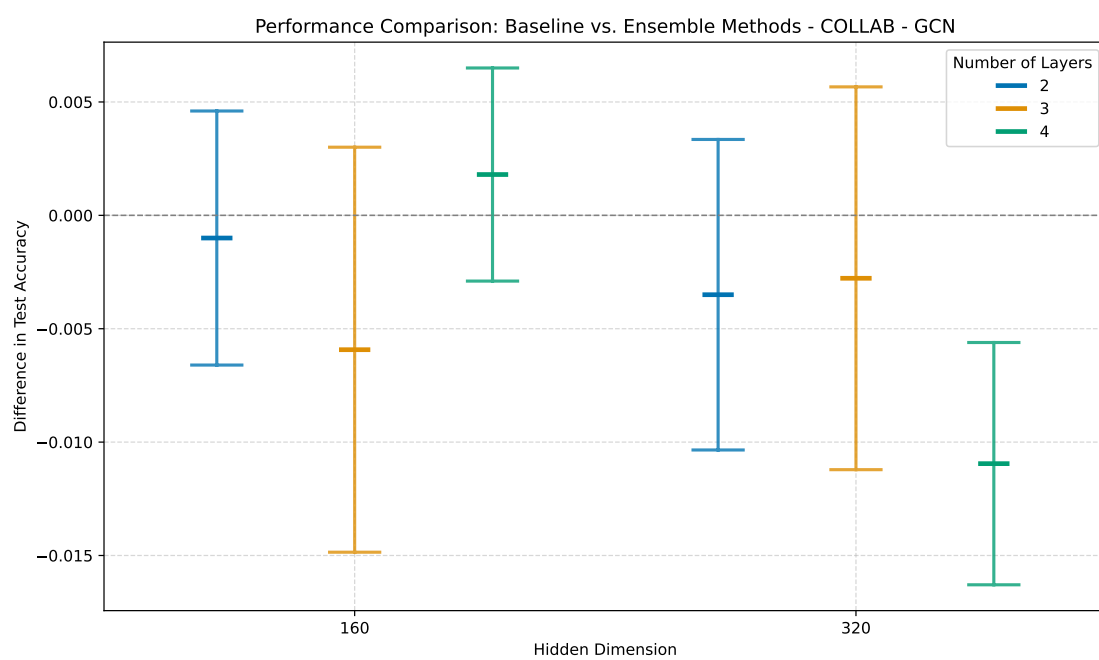


Figure 21: Pairwise accuracy difference between baseline and ensemble sampling methods across hidden dimensions and layers for GCN on the COLLAB dataset.

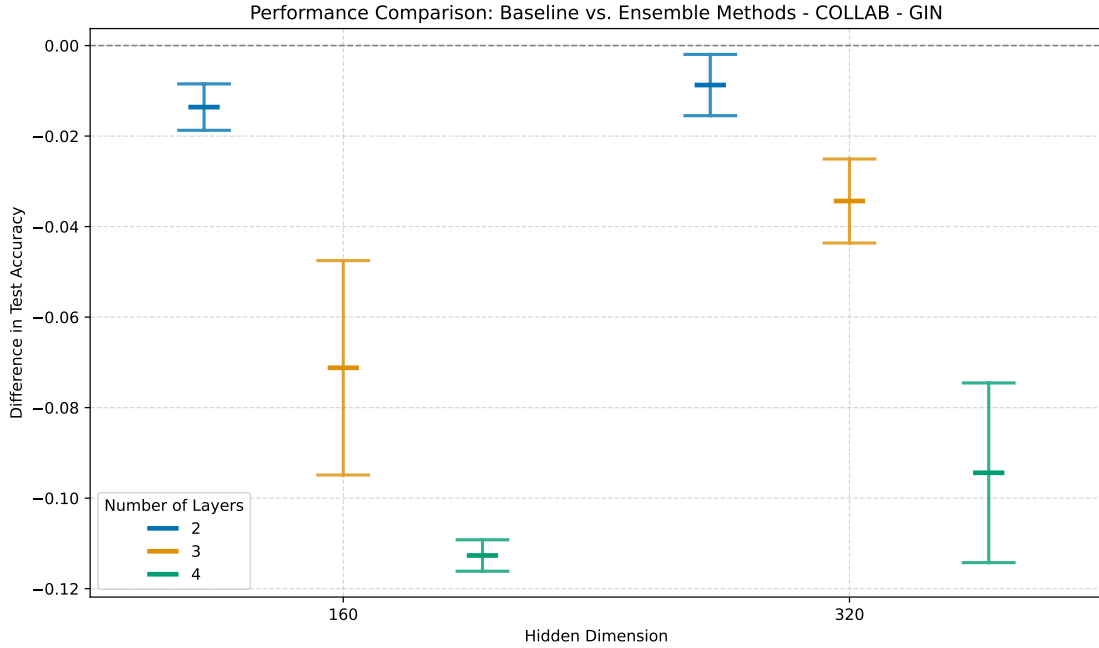


Figure 22: Pairwise accuracy difference between baseline and ensemble sampling methods across hidden dimensions and layers for GIN on the COLLAB dataset.

To compare the performance of the baseline, the random sampling method, and the ensemble method, we report accuracies and standard deviations on three datasets in Table 4. The table presents results for a two-layer MPNN with sum pooling on both GCN and GIN architectures. Since no significant differences were observed among the weight sampling methods, we arbitrarily selected the normal sampling approach. The best results for each architecture are highlighted in bold. The findings are consistent with our earlier observations. Overall, both sampling methods are competitive, with the possible exception of the GIN models on the COLLAB dataset. However, the ensemble method appears to improve the performance of the random sampling approach, while requiring similar or lower computational power and producing significantly sparser networks.

The performance of the ensemble sampling method might be explained by two main factors. First, using independent models helps reduce the risk of mixing information during the message-passing step, a situation that can occur with fully dense weight matrices. Second, by adopting a *divide and conquer* approach and sampling different parts of the larger ensemble model, we gain greater control over its overall composition. In this scenario, combining a diverse set of submodels increases the likelihood of capturing a broader range of patterns, thereby enhancing the chance of achieving better performance.

Architecture	Method	Mutagenicity	NCI1	COLLAB
GCN (160)	SGD baseline	82.08 \pm 1.04	81.31 \pm 2.28	81.00 \pm 0.78
	Random sampling	81.98 \pm 1.63	78.19 \pm 1.46	79.32 \pm 1.25
	Ensemble sampling	82.94 \pm 1.10	79.71 \pm 1.27	80.64 \pm 1.48
GIN (160)	SGD baseline	82.44 \pm 1.71	79.36 \pm 2.08	80.24 \pm 0.64
	Random sampling	82.67 \pm 1.21	79.22 \pm 1.85	77.20 \pm 1.51
	Ensemble sampling	83.50 \pm 0.86	80.53 \pm 1.43	78.44 \pm 1.20
GCN (320)	SGD baseline	81.80 \pm 1.43	80.97 \pm 1.55	81.44 \pm 1.58
	Random sampling	83.31 \pm 1.19	78.97 \pm 1.34	80.32 \pm 1.07
	Ensemble sampling	83.64 \pm 1.06	80.34 \pm 1.71	81.20 \pm 1.06
GIN (320)	SGD baseline	82.35 \pm 2.04	81.02 \pm 1.49	80.60 \pm 1.31
	Random sampling	82.94 \pm 0.99	78.78 \pm 2.21	78.16 \pm 1.18
	Ensemble sampling	83.31 \pm 1.12	81.07 \pm 1.50	77.84 \pm 1.65

Table 4: Test accuracy (mean \pm std) for baseline, random sampling, and ensemble methods.

5.2 Comparison of Performance over Time

To compare the time efficiency and performance of the ensemble method against the SGD baseline, we analyze how the accuracy of both methods evolves over training time. In the ensemble method, the standard approach is to sample small MPNNs and, once sampling is complete, train a final MLP on the embeddings to evaluate performance. However, to monitor performance over time, we need to record the accuracy at each update: whenever a new model is added to the top p list, we train an MLP on the current ensemble (comprising the current top p models) and note its accuracy. Although the MLP training time is included in the overall computation, at each update time we only account for the training time of the latest MLP used to obtain the current accuracy, rather than summing the training times of all previous MLPs. Since a single run of this process may not provide sufficient precision, we repeat the experiment several times and average the recorded accuracies over time. This procedure serves as a Monte Carlo simulation to estimate the expected accuracy of the ensemble method at each time.

For the baseline, the process is more straightforward because we only need to report the test accuracy over time. At any given time (or equivalently, at a specific epoch), we determine the test accuracy by selecting the best-performing epoch so far using the validation set. In other words, the reported test accuracy is that of the best validation epoch up to the current time. As with the ensemble sampling method, we repeat this process several times to obtain a more reliable estimate.

Figure 23 presents the results of the experiment on the NCI1 dataset. Due to the extensive computational time required, the experiment was conducted on a single split of the dataset and evaluated on one architecture, a three-layer GIN with a hidden dimension

of 160. For the ensemble method, the process was repeated 300 times, and similarly, the SGD baseline was trained 300 times. We first observe that the sampling method starts with an average accuracy of about 79%, and using ensemble sampling, it quickly increases to reach a final accuracy of approximately 81.5% after 40 seconds. The performance curve begins at around 6 seconds, which corresponds to the time required to train the first MLP for evaluating the initial ensemble after sampling the first 10 small models. In contrast, although the SGD baseline eventually converges to a similar final accuracy, it does so more slowly, taking around 50 seconds to reach the same level as the ensemble method.

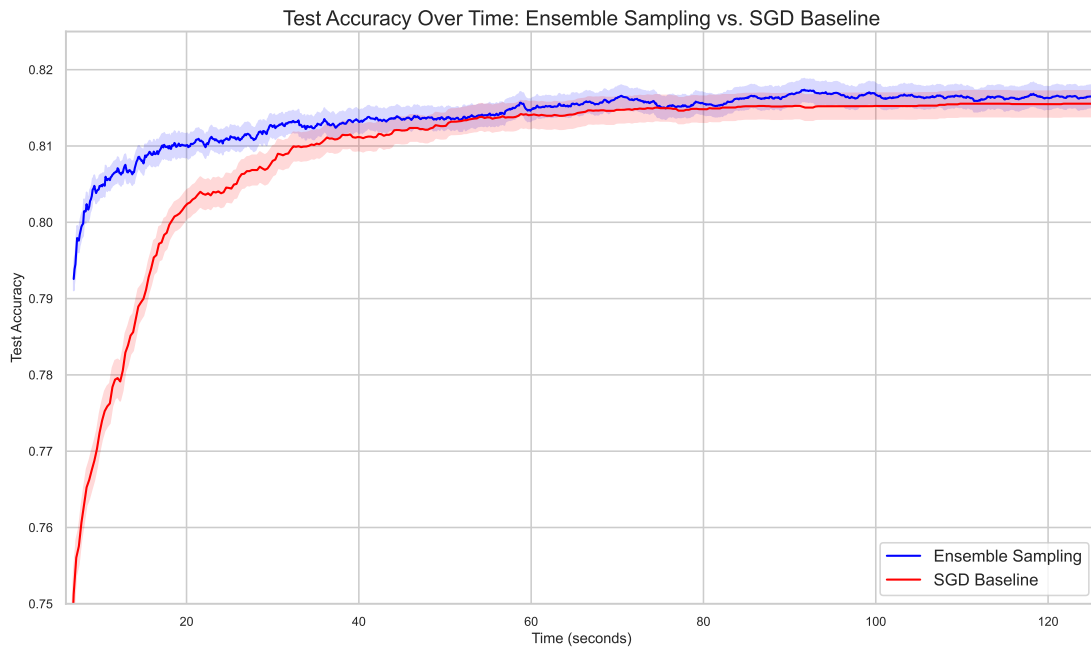


Figure 23: Accuracy comparison over time between SGD baseline and ensemble sampling methods on the NCI1 dataset.

While increasing the learning rate might seem like a straightforward way to speed up baseline training, the process is not simple. A higher learning rate tends to cause more unstable training and does not lead to faster convergence to a good test accuracy; in fact, it often results in a lower final accuracy. While this experiment supports the time efficiency of the ensemble method, further experiments on additional architectures and datasets are necessary to draw more definitive conclusions.

Apart from the benefits of increased sparsity and improved accuracy, another notable advantage of this ensemble method over the previous random sampling approach is its faster sampling step. As noted in the previous chapter, sampling is quicker for smaller networks. Consequently, for the same number of iterations, the ensemble sampling method runs significantly faster.

6 Conclusion

In this thesis, we developed novel methods based on randomly sampling the weights of graph neural networks. We demonstrated that these sampling methods can perform competitively with the classical stochastic gradient descent approach, and in some cases even exceed its performance.

We first introduced the alignment score as a means of assessing the quality of graph embeddings for tasks such as classification or regression. This score proved to be highly correlated with the model’s final performance, particularly test accuracy. The alignment score not only increases when an MPNN is trained using SGD, but it also serves as an effective predictor of final performance. Its computational efficiency makes it especially useful in sampling-based methods. Building on this, we proposed a random sampling method as an alternative to SGD. By iteratively sampling randomly initialized MPNNs, we can select the best-performing model based on the alignment score. This approach demonstrated competitive accuracy and time efficiency. In our extensive study, we evaluated the method across various architectures, pooling methods, and weight sampling strategies. In addition, we found that MPNNs with larger hidden dimensions are particularly effective at embedding graph inputs into meaningful vector representations. Finally, we introduced an extension to the random sampling method, the ensemble sampling method. Instead of sampling the weights for an entire network, we independently sample smaller MPNN models and then aggregate them into a larger, sparse model. This ensemble approach not only outperforms the previous random sampling method in terms of performance, but it also offers greater time efficiency during the sampling process and results in a significantly lower number of parameters due to its high sparsity.

Additional investigation could be pursued in several areas. Although the alignment score has proven effective for selecting the best-performing models, its scalability to large datasets is limited. One promising direction is to compute the alignment score on a subset of the dataset or to employ approximate methods such as approximate nearest neighbor (ANN) algorithms. Furthermore, for highly imbalanced datasets where accuracy is not the primary metric, the alignment score may need to be adapted accordingly. While we have conducted preliminary studies on time efficiency and the impact of iteration count on performance, extending this analysis to a broader range of datasets and architectures would be valuable. Another potential improvement is to incorporate node feature extraction before passing the data to the MPNN. High-dimensional input node representations may contain irrelevant features that introduce noise, thereby negatively affecting the final graph representation. More selective encoding of these features, as proposed in Bui et al. [4], could mitigate this issue. Finally, although the method for regression tasks has been theoretically introduced, experimental verification remains to be done. Additionally, the approach could be extended to node-level tasks.

References

- [1] Peter W. Battaglia et al. “Relational inductive biases, deep learning, and graph networks”. In: *CoRR* abs/1806.01261 (2018). arXiv: 1806.01261. URL: <http://arxiv.org/abs/1806.01261>.
- [2] Jan Böker et al. “Fine-grained Expressivity of Graph Neural Networks”. In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. Ed. by Alice Oh et al. 2023. URL: http://papers.nips.cc/paper%5C_files/paper/2023/hash/9200d97ca2bf3a26db7b591844014f00-Abstract-Conference.html.
- [3] Michael M. Bronstein et al. “Geometric Deep Learning: Going beyond Euclidean data”. In: *IEEE Signal Process. Mag.* 34.4 (2017), pp. 18–42. DOI: 10.1109/MSP.2017.2693418. URL: <https://doi.org/10.1109/MSP.2017.2693418>.
- [4] Thu Bui et al. “Random Propagations in GNNs”. In: *UniReps: 2nd Edition of the Workshop on Unifying Representations in Neural Models*. 2024. URL: <https://openreview.net/forum?id=kyPpQwMx3T>.
- [5] Federico Errica et al. “A Fair Comparison of Graph Neural Networks for Graph Classification”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: <https://openreview.net/forum?id=HygDF6NFPB>.
- [6] Francesco Di Giovanni et al. “On Over-Squashing in Message Passing Neural Networks: The Impact of Width, Depth, and Topology”. In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by Andreas Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 7865–7885. URL: <https://proceedings.mlr.press/v202/di-giovanni23a.html>.
- [7] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*. Ed. by Yee Whye Teh and D. Mike Titterton. Vol. 9. JMLR Proceedings. JMLR.org, 2010, pp. 249–256. URL: <http://proceedings.mlr.press/v9/glorot10a.html>.
- [8] 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. 2005, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

- [9] Weihua Hu et al. “Open Graph Benchmark: Datasets for Machine Learning on Graphs”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020. URL: <https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html>.
- [10] Tianjin Huang et al. “You Can Have Better Graph Neural Networks by Not Training Weights at All: Finding Untrained GNNs Tickets”. In: *Learning on Graphs Conference, LoG 2022, 9-12 December 2022, Virtual Event*. Ed. by Bastian Rieck and Razvan Pascanu. Vol. 198. Proceedings of Machine Learning Research. PMLR, 2022, p. 8. URL: <https://proceedings.mlr.press/v198/huang22a.html>.
- [11] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: <https://openreview.net/forum?id=SJU4ayYgl>.
- [12] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. “Self-Attention Graph Pooling”. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 3734–3743. URL: <http://proceedings.mlr.press/v97/lee19c.html>.
- [13] Ilya Loshchilov and Frank Hutter. “SGDR: Stochastic Gradient Descent with Warm Restarts”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: <https://openreview.net/forum?id=Skq89Scxx>.
- [14] Francesco Mezzadri. “How to generate random matrices from the classical compact groups”. In: *arXiv: Mathematical Physics* (2006). URL: <https://api.semanticscholar.org/CorpusID:6036276>.
- [15] Christopher Morris et al. “TUDataset: A collection of benchmark datasets for learning with graphs”. In: *CoRR* abs/2007.08663 (2020). arXiv: 2007.08663. URL: <https://arxiv.org/abs/2007.08663>.
- [16] Peter J. Rousseeuw. “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: [https://doi.org/10.1016/0377-0427\(87\)90125-7](https://doi.org/10.1016/0377-0427(87)90125-7). URL: <https://www.sciencedirect.com/science/article/pii/0377042787901257>.
- [17] T. Konstantin Rusch, Michael M. Bronstein, and Siddhartha Mishra. “A Survey on Oversmoothing in Graph Neural Networks”. In: *CoRR* abs/2303.10993 (2023). DOI: 10.48550/ARXIV.2303.10993. arXiv: 2303.10993. URL: <https://doi.org/10.48550/arXiv.2303.10993>.

- [18] Franco Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.
- [19] Petar Velickovic et al. “Graph Attention Networks”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: <https://openreview.net/forum?id=rJXMpikCZ>.
- [20] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Trans. Neural Networks Learn. Syst.* 32.1 (2021), pp. 4–24. DOI: 10.1109/TNNLS.2020.2978386. URL: <https://doi.org/10.1109/TNNLS.2020.2978386>.
- [21] Keyulu Xu et al. “How Powerful are Graph Neural Networks?” In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: <https://openreview.net/forum?id=ryGs6iA5Km>.
- [22] Zhitao Ying et al. “Hierarchical Graph Representation Learning with Differentiable Pooling”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 4805–4815. URL: <https://proceedings.neurips.cc/paper/2018/hash/e77dbaf6759253c7c6d0efc5690369c7-Abstract.html>.

