Leveraging Graph Structure for Graph Classification: An Examination of Graph Neural Networks

Bachelor Thesis
Faculty of Science, University of Bern

submitted by

Nûr Johanna Studer

from Bern, Switzerland

Supervision:

PD Dr. Kaspar Riesen
Anthony Gillioz
Institute of Computer Science (INF)
University of Bern, Switzerland

Abstract

This thesis explores the field of graph neural networks (GNN), a rapidly developing subclass of deep learning models, designed for graph data. We investigate the effectiveness of state-of-the-art GNN models in leveraging graph structure for graph classification tasks. Specifically, two models, Graph Isomorphism Network (GIN) and Deep Graph Convolutional Neural Network (DGCNN), are analysed on datasets with and without node features and compared to a multilayer perceptron (MLP) as a baseline. The results reveal that GIN consistently outperforms DGCNN. However, there are datasets where the MLP baseline outperforms both GNN models. Further, it is shown that the introduction of node degrees as node features in social datasets, generally enhances classification accuracy. Additionally, our results indicate that the performance consistency primarily depends on the dataset, rather than the model or the presence of node features. Lastly, we identify potential underutilisation of graph structure by GNN models, suggesting room for improvement. This study provides insights for future research on improving the exploitation of graph structure for graph classification tasks.

Contents

Chapter 1

Introduction

Artificial intelligence (AI) is the branch of computer science concerned with creating machines, that have the ability to think and act intelligently [1]. The intended intelligence can either be modelled after human behaviour or after the general concept of rationality, i.e., doing the "right thing" given the present information. To pass the test of acting humanly a machine would have to be able to communicate (natural language processing), store its knowledge (knowledge representation), use its knowledge to draw conclusions (automated reasoning), adapt to new circumstances and recognise patterns (machine learning), perceive objects (computer vision) and be able to move and manipulate objects (robotics) [1].

Machine learning (ML) as a subfield of AI focuses on the design and implementation of algorithms that allow machines to learn from and make decisions or predictions based on data, effectively improving through experience [2]. ML intersects the field of statistics, asking what principles rooted in statics, computation and information theory regulate all learning systems. Many AI developers have recognised that for various applications, letting a system learn from examples rather than programming it manually, is easier. This in combination with the present abundance of data and low-cost computation resources has led to ML becoming the preferred approach to many AI applications [2].

A core task in Machine Learning is *pattern recognition*, the classification of input values into different categories [3]. Some areas in which Pattern Recognition plays an important role include machine vision, computer-aided diagnosis, speech recognition and data mining. The classification abilities can either be based on supervised or unsupervised learning. In supervised learning, a machine learns from training data with class labels and generalises the present knowledge to unknown cases. However, labelled training data is not always available. In the case of unsupervised learning a machine is presented with training data and the goal is to recognise underlying patterns and group (or cluster) similar data points together [3].

Deep learning is a subset of machine learning and a form of representation learning [4]. The goal of representation learning is to learn the representations, that are necessary for feature detection or classification, from raw data. Deep-learning methods implement this by transforming the initial representation over multiple layers, using simple, non-linear modules. Deep learning is especially well-equipped to utilise the increasing availability of data and computation, due to its key aspect of these layers of modules not being hand-engineered but rather learned from data. Most applications of deep learning are implemented using *artificial neural networks (ANNs)* [4]. ANNs, modelled loosely after the human brain, are composed of interconnected nodes (or neurons) which take in input values, process them based on a non-linear activation function and give out an output value [5].

Geometric deep learning, is a term used for emerging techniques, that attempt to generalise deep learning methods to non-Euclidean data, like graphs [6]. A central part of geometric deep learning is extending the concepts of convolutional neural networks (CNNs) designed for grid-like data (like images) to more general types of structures [6]. *Graph representation learning* on the other hand is concerned with how to effectively represent graphs with vectors [7]. The goal is to encode elements of a graph in such a way, that information about the graph can later be reconstructed.

At the intersection of these two areas and an important part of both are *graph neural networks (GNNs)*. GNNs manage to leverage both node features and topological structure information in their learning mechanism [7]. They have become one of the fastest-growing areas of deep learning in the last decade [8]. A wide range of real-world data can inherently be modelled as graphs, which is why the question of how those graphs can be analysed with machine learning has been of increasing interest. Some examples include social networks, molecular structures, citation networks, knowledge graphs or transportation networks. Some applications of GNNs are recommendation systems, social influence prediction, molecular fingerprinting, chemical reaction prediction, protein interface prediction, text classification, visual reasoning and traffic forecasting [9]. GNN applications can be categorised into node-level tasks like node classification, edge-level tasks like link prediction and graph-level tasks like graph classification. For graph classification tasks, GNNs learn a graph-level representation from node and structural information [10]. Since most well-known GNN models are designed for node-level tasks and the research field is still relatively young, the effectiveness of GNNs designed for graph classification is a question open for investigation.

The objective of this thesis is to investigate the effectiveness of current state-of-the-art GNN models in exploiting graph structure for graph classification tasks. The hypothesis is that current GNN models are not utilising the graph structure to the fullest potential. To explore this hypothesis, two GNN models are chosen for further analysis. The performance of the models is examined on graphs with node features as well as on graphs with their node features removed, leaving only the structural information. Additionally, the models are compared to a baseline model, that only uses the node features to classify the graphs. Through this approach, it can be evaluated whether the GNN models utilise structural information by comparing their performance when only structural information is available to their performance when they have full access to both structural information and node features. Additionally, by using a baseline model, their performance can be compared to the performance without any structural information. On one hand, this gives more insight into the models' ability to utilise structural information and on the other hand, it allows a general statement about their performance. The results are then used to discuss the stated research question and hypothesis.

This thesis is organised as follows. Chapter 2 provides an introduction to the theoretical context, explaining the basics of ANNs and graph theory. In Chapter 3 an overview of GNNs is given, describing different approaches and models. Chapter 4 is dedicated to the experimental part of the thesis. The experiment setup as well as its execution is described, and results are presented and discussed. The final Chapter 5 wraps up the thesis with a conclusion, summarising the key findings and looking into possible future work.

Chapter 2

Theoretical Context

This chapter aims to embed the thesis in a theoretical context. In section 2.1 the basic elements of artificial neural networks are explained. Terms like neuron, activation function, and backpropagation are clarified and the architecture of a neural network in its simplest form is described. Section 2.2 serves as an introduction to basic graph theory, explaining what a graph is and how it can be represented. It then touches on the motivation for analysing graphs with machine learning.

2.1 Artificial Neural Networks

The introduction to artificial neural networks (ANNs) in this section follows the one given by Liu and Zhou [11]. Neural networks are of great importance in the field of machine learning. ANNs, consisting of many interconnected neurons, conceptually stem from their biological counterparts. To learn, a neural network is initialised with arbitrary weights and then iteratively adjusts them using backpropagation until the model reaches a high level of accuracy. Neural network research primarily focuses on refining learning methods using various algorithms and structures to improve the generalisation capabilities of models.

Neurons are the fundamental units of ANNs. Their basic structure is illustrated in Fig. 2.1(a). A neuron receives input values $x_0, x_1, ..., x_n$ either from the input layer or from the previous layer of neurons. Each input value has a corresponding weight w_i, which is initialised randomly but is adjusted during training. The neuron calculates the weighted sum y by multiplying each input value by its associated weight, summing the products, and adding a bias term b.

$$y = \left(\sum_{i=1}^{n} w_i x_i + b \right)$$

The net input is then passed through an activation function f, which maps the input to a value between 0 and 1 and represents the neuron's activation. Common examples of activation functions include the Sigmoid Function, the Tanh Function and the Rectified Linear Unit (ReLU). The neuron's output z is subsequently sent to the next layer of neurons or serves as the network's final output if it belongs to the output layer.

$$z = f(y) = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.1}$$

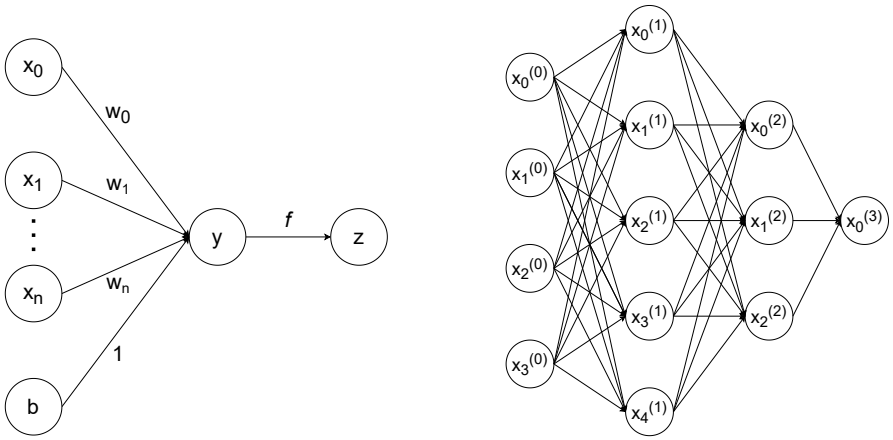
Backpropagation

Throughout the learning process, model parameters are typically optimised using backpropagation, which consists of a forward pass and a backward pass. During the forward pass, input data progresses through the network as described and an output is produced. This output is then compared to the target value, resulting in an error (also called loss). In the backward pass, the gradients of the loss relative to the weights and biases are calculated using the chain rule. The gradients represent the contribution of each weight and bias to the overall loss. With an optimisation algorithm like gradient descent, the parameters are updated based on their gradient to minimise the loss. The network undergoes iterations of forward and backward passes until the loss is effectively minimised. This enables the ANN to learn patterns and make predictions or classifications based on input data.

Neural Network Architectures

The simplest version of an ANN is a feedforward neural network (FNN). FNNs are characterised by their hierarchical structure, connecting each layer of neurons only to its neighbouring layers and not containing any loops. This type of network, contains an input layer, one or several hidden layers and an output layer. Typically, the layers are fully connected, meaning neurons are connected to all the neurons of the previous layer. A fully connected FNN is also referred to as a multilayer perceptron (MLP). An example architecture is shown in Fig. 2.1(b).

An important variation of a FNN is a convolutional neural network (CNN). CNNs, apart from containing fully connected layers, have some neurons that are only connected to a limited number of neurons from the previous layer. This way of preserving local connections is useful in areas like computer vision, where CNNs can detect local patterns within an image, improving capabilities like scene understanding and image processing [11].



(a) Neuron structure (b) Network architecture

Figure 2.1: Components of an ANN with the general structure of a neuron (a) and a possible architecture of an MLP (b).

2.2 Graphs

Our motivation for understanding graphs is two-fold. On one hand, ANNs as explained in section 2.1 can be seen as graphs themselves, with the neurons being nodes organised in layers with connections (or edges) going from one layer to the next. On the other hand, a wide range of data from real-world applications can inherently be modelled as graphs [9]. Some examples include social networks, molecular structures, citation networks, knowledge graphs or transportation networks. Due to this, the question of how these graphs can be analysed with machine learning has been of increasing interest.

Basic Graph Theory

A graph $G = (V, E)$ consists of a set of nodes V and a set of edges $E \subseteq V \times V$ connecting pairs of nodes. An edge between node $u \in V$ and node $v \in V$ is denoted as $(u, v) \in E$ [7]. A relevant property of graphs is, that they are non-Euclidean meaning they lack certain familiar properties like a standard coordinate system, shared measurement system, vector space structure, or consistent location-based patterns [6]. Another characteristic is that nodes are not organised in a specific order [12]. Two graphs which contain the same nodes and edges between said nodes are called isomorphic. As any function applied to a graph should not depend on the order of nodes, it follows that the function applied to two isomorphic graphs should lead to the same result. The described property is also referred to as permutation invariance [7]. This can be written as $f(PAP^T) = f(A)$ with A being the adjacency matrix of a graph, P being a permutation matrix and f being the applied function.

Representations of Graphs

The adjacency matrix $A \in \mathbb{R}^{|V|\times|V|}$ is one way to conveniently represent a graph [7]. For this, the nodes are arranged in a specific order, with each node corresponding to a particular row and column in the matrix. The presence of edges is represented as entries in the matrix.

$$A_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \text{ and } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

Additionally, graphs commonly have node-level labels or features associated with them [7]. These features can be represented using a matrix of real values, denoted by $X \in \mathbb{R}^{|V|\times m}$, where m represents the number of attributes or features associated with each node and the ordering of the nodes in the feature matrix X is consistent with the ordering of the nodes in the adjacency matrix.

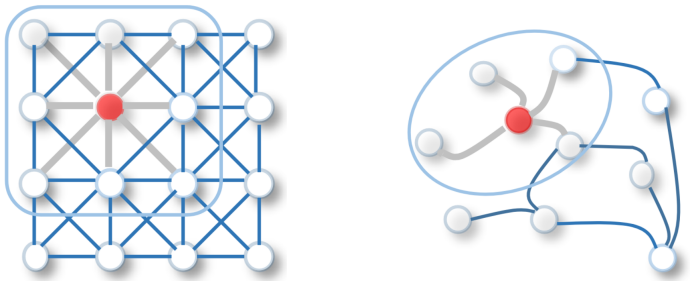
An extension of the adjacency matrix are Laplacian matrices, which are formed through various transformations of the adjacency matrix and possess useful mathematical properties [11]. The most basic form of a Laplacian matrix is the unnormalised Laplacian $L \in \mathbb{R}^{|V|\times|V|}$, defined as $L = D–A$ where D and A are degree and adjacency matrices respectively. The degree $d(v)$ of a node $v \in V$ is the number of edges connected to v (i.e., its number of neighbour nodes). The degree matrix $D \in \mathbb{R}^{|V|\times|V|}$ is a diagonal matrix, with the diagonal consisting of the nodes' degrees. A commonly used variant of the Laplacian matrix is the symmetric normalised Laplacian, defined as:

$$L_{norm} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \tag{2.2}$$

with $I \in \mathbb{R}^{|V|\times|V|}$ being the identity matrix [11].

Analysing Graphs with Machine Learning

The motivation for analysing graphs with machine learning stems from the advancement of deep neural networks, especially CNNs, on one hand and from graph representation learning on the other [9]. *Graph representation learning* is concerned with how to effectively represent graphs with vectors [7]. The goal is to encode a graph or its nodes or edges with a low-dimensional vector in such a way that it can later be decoded and used to reconstruct certain information about the original graph. Traditional approaches apart from GNNs include random-walk- and factorisation-based methods [7]. While Deep Learning has progressed far in the analysis of Euclidean data, the analysis of non-Euclidean data poses a new chal-



(a) 2D convolution (b) Graph convolution

Figure 2.2: 2D vs graph convolution [8].

lenge [8]. In addition to there being no order to nodes, graphs in a dataset might also vary in size, and nodes might vary by their number of neighbours. The attempt of generalising existing deep learning models to non-Euclidean domains including graphs is referred to as *geometric deep learning* [6]. As an example, CNNs have the ability to extract local spatial properties but can only process Euclidean data such as images which can be represented in Euclidean space as a 2D grid [9]. As shown in Fig. 2.2(a) for 2D convolution, an image can be viewed as a graph with nodes being pixels and neighbours being the pixels within the filter size [8]. The value of a pixel is calculated by taking the weighted average pixel value of the pixel and its neighbours. Similarly, graph convolution as shown in Fig. 2.2(b) can be modelled after 2D convolution by taking the weighted average of a node and its neighbour nodes. The difference is, that the neighbourhood of a node is unordered and can vary in size.

Chapter 3

Graph Neural Networks

This chapter serves as an introduction to graph neural networks (GNNs). In section 3.1 an overview of the different approaches to GNNs is given and the message-passing framework is introduced. In section 3.2 different GNN models are reviewed. First, the most well-known models are introduced, and an example GNN model is analysed to bridge the gap between graph convolution and MLPs. Later GNNs specifically designed for graph classification tasks are described in more detail.

3.1 Approaches

Wu et al. provide an overview of different GNN approaches and implementations [8]. They describe the research on GNNs as having become one of the fastest-growing areas of deep learning in the last decade. In an attempt to extend deep learning techniques like the convolution from CNNs to graphs, different approaches have been developed in parallel, with different frameworks trying to categorise them. Graph convolutional neural networks are sometimes divided by their initial motivation into spectral- and spatial-based approaches. The authors describe a general spectral-based approach as having its foundation in graph signal processing and using the Fourier transform. The graph convolution operation is defined in the Fourier domain, utilising the eigendecomposition of the normalised Laplacian matrix. A spatial-based approach, on the other hand, involves aggregating the feature information of a node's neighbour nodes and combining it with its own [8]. However, these two categories can overlap and there are also approaches which unify existing models under one general framework [10]. One popular framework categorises many GNN models as being Message Passing Neural Networks (MPNN) [13]. MPNNs are described as having a message-passing phase where each node is iteratively updated based on the aggregated information from its neighbour nodes, and a readout phase where node features are combined to produce a graph-level representation.

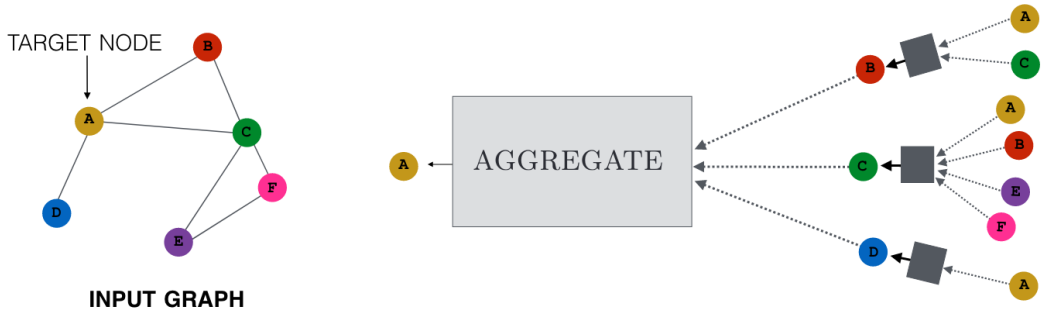


Figure 3.1: Aggregation of a single node's embedding in a two-layer message-passing model as a tree-like structure [7].

3.1.1 Message-Passing Framework

As described in more detail by Hamilton [7], in the message-passing framework a node $u \in V$ in a graph $G = (V, E)$ is initially represented with a hidden embedding h_u based on the node's input features. The node's hidden embedding $h_u^{(k)}$ changes based on the iteration k of the message-passing mechanism. The nodes belonging to u's neighbourhood are denoted with $N(u)$. The iterative updating of the node embedding can be described as follows:

$$h_u^{(k+1)} = \text{UPDATE}^{(k)}\left(h_u^{(k)}, \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in N(u)\})\right) \tag{3.1}$$

$$= \text{UPDATE}^{(k)}\left(h_u^{(k)}, m_{N(u)}^{(k)}\right) \tag{3.2}$$

UPDATE and AGGREGATE are differential functions and $m_{N(u)}^{(k)}$ is the process of aggregating the information from the neighbouring nodes described as a "message" (Fig. 3.1). The UPDATE function combines the message with the node's embedding to form the next embedding $h_u^{(k+1)}$ [7].

3.1.2 The three flavours of GNN layers

Bronstein et al. [12] further distinguish between three flavours of GNN layers with a containment hierarchy: convolutional $\subseteq$ attentional $\subseteq$ message-passing (Fig. 3.2). They describe the message-passing mechanism with ϕ being the updating function, $\bigoplus$ being the aggregation function and ψ being a function to transform the features from neighbour nodes. The convolutional type directly aggregates the features of the neighbourhood nodes with fixed weights. With c_{uv} being a constant that stands for the importance of a neighbourhood node x_v to the node x_u, h_u is calculated as:

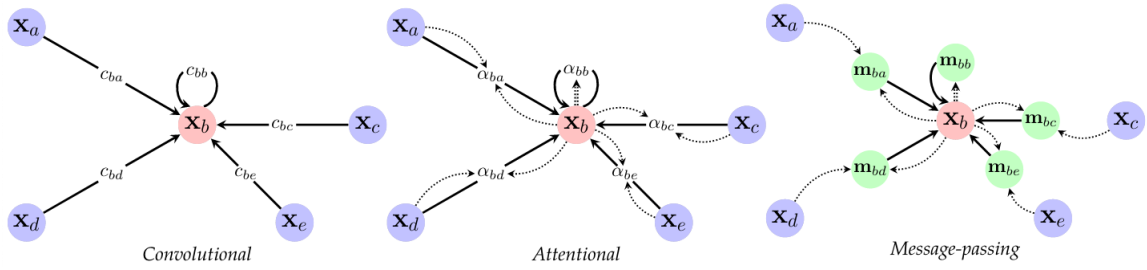


Figure 3.2: The three flavours of GNN layers [12].

$$h_u = \phi\left(x_u, \bigoplus_{v \in N_u} c_{uv}\psi(x_v)\right) \tag{3.3}$$

In the attentional type, a learnable self-attention mechanism a is used. With a computing the importance of x_v implicitly, the weights depend on the features and h_u is calculated as:

$$h_u = \phi\left(x_u, \bigoplus_{v \in N_u} a(x_u, x_v)\psi(x_v)\right) \tag{3.4}$$

The attentional layer can be seen as a convolutional layer by implementing the attention mechanism with a look-up table $a(x_u, x_v) = c_{uv}$. For the most general message-passing type, vectors are computed across edges. ψ is a learnable message function, calculating the vector or "message" that is sent from v to u and h_u is calculated as:

$$h_u = \phi\left(x_u, \bigoplus_{v \in N_u} \psi(x_u, x_v)\right) \tag{3.5}$$

Both convolutional and attentional layers can be understood as a form of message passing with $\psi(x_u, x_v) = c_{uv}\psi(x_v)$ and $\psi(x_u, x_v) = a(x_u, x_v)\psi(x_v)$ respectively. The authors note that while it is useful, that most GNNs can be expressed with message passing, the message-passing variant is typically harder to train and uses an impractically big amount of memory [12].

3.2 GNN Models

GNN models can be designed with different tasks in mind, which can be categorised as node-, edge- or graph-level tasks [9]. Node-level tasks focus on attributes of individual nodes and include node classification which aims to categorise nodes into classes, node regression which predicts continuous values for nodes and node clustering which is concerned with partitioning nodes into groups according to node-similarity. Edge-level tasks on the other hand deal with analysing and predicting relationships between nodes and include edge classification and link prediction. Lastly, graph-level tasks require learned graph representations and include graph classification, graph regression and graph matching [9].

3.2.1 Popular Models

It is important to note that most well-known models were primarily designed for and evaluated on node-level tasks, typically node classification. Some of the most cited GNN models[1] include the following. The Spectral Neural Network (**Spectral CNN**) [14] is an early model following a spectral approach. It was later extended by the Chebyshev Spectral CNN (**ChebNet**) [15], which makes an approximation using Chebyshev polynomials to reduce computational complexity. The Graph Convolutional Network (**GCN**) [16] further simplified the approach and is the most well-known model to date. While its original motivation comes from a spectral approach, it can also be considered a spatial method which aggregates feature information from neighbour nodes [8]. In terms of the three flavours of GNN layers, GCN is the typical example for convolutional layers, as described in Eq. (3.3) [12]. The following models have their basis in spatial approaches. **GraphSAGE** [17] addresses the issue that neighbourhoods of nodes can vary greatly in size, by sampling a fixed number of neighbour nodes to aggregate information. It suggests and evaluates three different types of aggregators. On the other hand, the key feature of the Graph Attention Network (**GAT**) [18], is using an attention mechanism to determine the relative weights between nodes as described in the attentional flavour of GNN layers in Eq. (3.4) [12].

3.2.2 Bridging the Gap between GNNs and MLPs

As an example, the GCN model is further analysed to connect the introduced knowledge about the MLP architecture from section 2.1 with what we have learned so far about GNN models. The GCN model defines the propagation rule for a

[1] See Appendix A, Table A.1 for an overview of citation count and original publishing year.

convolutional layer as follows [16]:

$$H^{(k+1)} = f\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k)}W^{(k)}\right) \tag{3.6}$$

With $\tilde{A} = A + I$ being the adjacency matrix with added self-connections and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ being the degree matrix with the added self-connections. $H^{(k)}$ is the matrix of activations (i.e., node embeddings) at the k^{th} layer, $W^{(k)}$ is a trainable weight matrix and f is the activation function [16]. The added self-connections, in terms of message passing, can be understood as the nodes being updated, not only based on the node embeddings of their neighbours, but also their own.

If we define a modified version of the symmetric normalised Laplacian (Eq. (2.2)) as:

$$L_{norm}^{mod} = \tilde{D}^{-\frac{1}{2}}(A + I)\tilde{D}^{-\frac{1}{2}} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \tag{3.7}$$

We can then rewrite Eq. (3.6) as:

$$H^{(k+1)} = f\left(L_{norm}^{mod}H^{(k)}W^{(k)}\right) \tag{3.8}$$

The previously defined output of a fully connected FNN layer z (Eq. (2.1)) can be written with a matrix-vector multiplication as:

$$z = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f\left(Wx + b\right) \tag{3.9}$$

with W being the trainable weight matrix, x being the vector of inputs, b being the bias vector and f being the activation function.

In this way, by comparing Eq. (3.8) and Eq. (3.9) the similarities between layers in an MLP and a GCN become more apparent, the main difference being the Laplacian matrix representing graph structural information in the GCN. To further illustrate the similarities of the structure of MLPs (as shown in Fig. 2.1(b)) with the structure of GNNs, Figure 3.3 shows how the graph structure itself determines the connections in a GNN rather than the graph only being an input. The GNN layers represent the calculation of node embeddings at the respective layer.

3.2.3 GNNs for Graph Classification

Wu et al. describe the general approaches to graph classification, comparing graph kernels to GNNs [8]. Historically, graph classification tasks have been solved with graph kernels. Graph kernels use kernel functions to measure the similarity be-

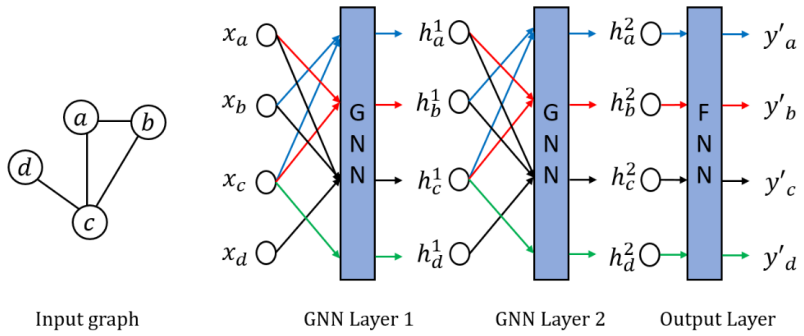


Figure 3.3: Structure of a two-layer GNN, with the graph structure determining the connections in the network [19].

tween graph pairs, allowing kernel-based algorithms like support vector machines to be used for supervised learning. Due to graph kernel methods suffering from a considerable computational load, GNNs can be an attractive and more efficient alternative for graph classification tasks.

The authors state that for graph classification, GNNs commonly employ convolutional layers to obtain node embeddings, graph pooling layers to down-sample and a readout layer to derive a fixed-size graph representation. Finally, an MLP and a softmax layer can be applied for the classification task (Fig. 3.4). The softmax layer produces a probability distribution over the possible classes, where the sum of the probabilities equals one [8]. In terms of pooling, the readout layer can also be considered direct or global pooling, which only considers node features, while in comparison hierarchical pooling considers structural information and is applied by layer [9]. Another related term, sometimes used interchangeably with pooling and readout function, is graph coarsening, which refers to reducing the number of nodes in a graph to less or even one final node [10]. A simple and computationally inexpensive way of pooling is mean, max and sum pooling [8]. By reducing the size of graph representations, pooling layers help avoid overfitting and reduce computational complexity.

Hierarchical Pooling

One of the earlier variants of hierarchical pooling is Edge-Conditioned Convolution **(ECC)** [20], which uses a recursive down-sampling method for pooling [9]. It weighs the node aggregation based on edge parameters [21]. A popular hierarchical pooling model is Differential Pooling **(DiffPool)** [22]. It uses a learnable cluster assignment matrix for every layer based on node and structural information [8]. DiffPool can be combined with any type of convolutional layer, but the original paper suggests combining it with GraphSAGE with the mean aggregator and incorporating a pool-

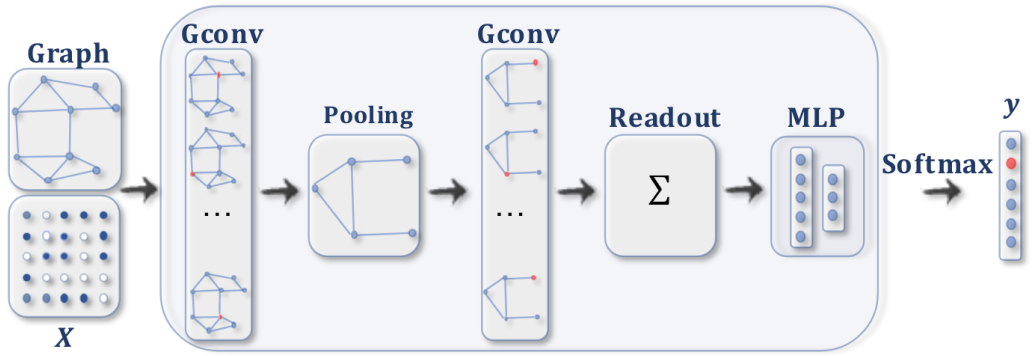


Figure 3.4: A general convolutional GNN with pooling, a readout layer and an MLP a softmax layer for final classification [8].

ing layer after every two convolutional layers [22]. The downside of DiffPool is its high computational complexity [8]. The Graph U-Nets model **(g-U-Nets)** [23] uses graph pooling (gPool) and its inverse un-pooling (gUnpool). The gPool layer uses scalar projection scores to select nodes, and form a smaller graph, while the gUnpool layer restores the original structure [23]. In comparison to DiffPool, g-U-Nets saves storage complexity by using a vector instead of a matrix per layer. However, the graph structure is not taken into account [9]. Self-Attention Graph Pooling **(SAGPool)** [24] on the other hand also uses node as well as structural information like DiffPool but uses a self-attention mechanism resulting in a lower time and storage complexity [9].

Sorting Methods and the Weisfeiler-Lehmann Algorithm

A method based on assigning labels to nodes and ordering them is PATCHY-SAN **(PSCN)** [25]. The labels are assigned as a score based solely on graph structure (e.g., node degree, centrality, or Weisfeiler-Lehmann (WL) colour). For each node, a fixed number of neighbour nodes are selected based on their label. Because of the uniform size, the result can then be passed to a traditional 1D CNN [8]. Similarly to PSCN the Deep Graph Convolutional Neural Network **(DGCNN)** [26] uses an assigned ordering of nodes by sorting nodes according to their structural role and applying 1D CNNs [9]. In comparison to PSCN, DGCNN sorts all nodes of a graph rather than sorting the respective neighbourhoods of nodes [10]. In the mechanism called SortPooling DGCNN first uses spatial convolutions on the unordered nodes to determine their structural importance before selecting the top-k nodes for a fixed-size representation [8]. As described by the original authors, the graph convolution of DGCNN uses a propagation matrix similar to GCN [26]. They draw the connection to the WL subtree kernel which uses the WL algorithm to extract

features for graph classification. The authors argue that their convolution is a closer approximation of the WL algorithm than GCN. The idea of the WL algorithm is to update a node's colour, i.e., its assigned label, based on a concatenated version of the node's colour with the colours of its one-hop neighbours. Ordering nodes by WL colour results in nodes with similar structural roles in different graphs being assigned a similar relative position. DGCNN aims to approximate WL colours and shows that the output of the graph convolution layers equates to a continuous version of WL colours and can thus be used for sorting [26]. The Graph Isomorphism Network **(GIN)** [27] was proposed based on further analysis of the relationship of the WL algorithm and GNNs. It was shown that GNNs cannot be better at distinguishing graph structures than the WL graph isomorphism test which is based on the same WL algorithm as the WL kernel [10]. GIN introduces a parameter ϵ, which can be learnable or a fixed scalar, as a weight for the target node in the aggregation step [27]. In addition, a readout function is proposed where summed node features are concatenated for each iteration. Combined with the use of an MLP it is shown that GIN reaches the maximum possible discriminative power a GNN can have [27]. See Appendix A, Table A.2 for an overview of citation count and original publishing year of the described GNN models.

Chapter 4

Experiment

This chapter describes the conducted experiment. Section 4.1 outlines the experiment setup. It explains the decision process for which GNN models were chosen, how the experiment was designed and what evaluation framework was used. Additionally, the used datasets and hyperparameters are described. In section 4.2 the execution of the experiment is described. The architectures of the used models are presented in detail and the code implementation is explained.

4.1 Experiment Setup

The goal of the experiment is to investigate the effectiveness of current state-of-the-art GNNs in exploiting graph structure for graph classification tasks. The hypothesis is that current GNN models are not utilising the graph structure to the fullest potential. An important point of reference for this thesis is the paper titled, *A Fair Comparison of Graph Neural Networks for Graph Classification* by Errica et al. [21], which criticises the model evaluation practises of different works and offers a standardised evaluation framework. To explore our hypothesis, two widely-known GNN models, GIN [27] and DGCNN [26], are tested on three chemical datasets (MUTAG, NCI1 and PROTEINS) and two social datasets (IMDB-BINARY and IMDB-MULTI). The performance of the models is examined on graphs with node features as well as on graphs with their node features removed, leaving only the structural information. Additionally, the models are compared to a baseline MLP model, that only uses the node features to classify the graphs. The experiment was implemented using PyTorch Geometric [28] for its deep learning model implementations, PyTorch Lightning [29] for structure and readability and Optuna [30] for hyperparameter optimisation.

GNN Model Selection

For the selection of GNN models to evaluate, several criteria are taken into account:

- Citation count
- Code availability
- Differences in architecture
- Test results in the original paper
- Test results in the reference paper by Errica et al. [21]

The citation count is considered, as a reflection of the impact and recognition of the models in the research community. The availability of the original code as well as a PyG implementation is checked to ensure reproducibility. The differences in the models are examined to ensure there is some diversity in the investigated architectures. In the original papers, the overall performance of the models is considered as well as the availability of results on social and chemical datasets. Lastly, the results in the reference paper are reviewed. As the paper provides a fair and objective comparison of the performance of different models, it serves as a strong indicator of the suitability of the models for this thesis. By considering these criteria, the GIN [27] and DGCNN [26] models were chosen for further investigation.

Datasets

The used graph classification datasets[1] are some of the most used benchmark datasets and they are publicly available [31]. They include three chemical (MUTAG [32], NCI1 [33], PROTEINS [34]) and two social (IMDB-BINARY, IMDB-MULTI[35]) datasets. The chemical datasets are node-labelled, while the social datasets are not. The datasets pose binary classification tasks except for the IMDB-MULTI dataset which is a multi-class classification problem with three classes.

The MUTAG dataset represents nitroaromatic compounds, labelled as mutagenic or non-mutagenic Nodes represent atom types in a one-hot encoding. The NCI1 dataset contains graphs of chemical compounds screened for cancer activity, with one-hot encoded atom types. The PROTEINS dataset contains graphs of proteins, classified as enzymes or non-enzymes, with nodes representing amino acids. The IMDB-BINARY dataset includes graphs of movies, classified into action or romance genres, with nodes as actors and edges as collaborations. The IMDB-MULTI dataset follows the same concept as IMDB-BINARY but uses three genres instead.

[1] Dataset statistics in Appendix A Table A.3.

Experiment Design

The purpose of the experiment is to investigate how effective the two selected models are at exploiting the input graphs' structural information. Three different variants of the experiment are designed:

- 1. Experiment: graphs with node features classified with GNN.
- 2. Experiment: graphs without node features classified with GNN.
- 3. Experiment: node features classified with an MLP.

In the first experiment, the maximum amount of information is used. The node features are kept and by classifying the graphs with a GNN the structural information is used. In the case of chemical datasets, the node labels are one-hot encodings, resulting in vectors of 0/1 elements. For the social datasets, which are unlabelled (i.e., have no node features), one-hot encodings of the node degrees are used as their features. In the second experiment, the graphs are stripped of their node features by replacing them with a single value of one. By then classifying them with a GNN, only the structural is leveraged. In the third experiment, the graphs' structural information is discarded. Only the node features are used to classify the graphs with an MLP.

For the MLP, used as a baseline to compare the GNN performance to, the implementation by Errica et al. [21] is followed. For chemical datasets, the so-called Molecular Fingerprint technique is used, which uses global sum pooling followed by a single-layer MLP with ReLU. For social datasets, a single-layer MLP followed by global sum pooling and another single-layer MLP is used.

Evaluation Framework

As previously stated, Errica et al. criticise the model evaluation of several GNN model proposals. The authors focus their criticism on the usage of data splits for training, validation, and testing as well as the availability of code, especially for model selection and not only model assessment. For example, for the GIN [27] model the original paper explicitly states that the validation accuracy of a 10-fold cross-validation (CV) is reported. For DGCNN [26] the model was evaluated using a 10-time repetition of a 10-fold CV. The hyperparameters were tuned once using one random fold. However, the average of the 10 final scores was reported as the result instead of the overall average, which lowered the variance. The code for model selection is not provided from either GNN model.

Errica et al. propose an evaluation framework to use when comparing GNNs, and they also share their code[2] and all hyperparameters that are tuned and their respective values. Their process consists of using a 10-fold CV, where for each fold they use a holdout technique with a 90%/10% split to further split the training folds into training and validation sets. The data splits are stratified, meaning class proportions are represented accurately in the split sets. They optimise hyperparameters on the validation set and select the best model. The selected model is then retrained on the entire set of training folds and tested on the test set, repeating the process three times. However, in the retraining step, they again split the training set to use 10% for early stopping.

In our experiment, the proposed evaluation framework was slightly altered. A 10-fold CV and inner holdout technique with a 90%/10% split was also used, with splits being stratified. However, once the best model was selected it was directly tested on the test set. The 10-fold CV was additionally repeated 10 times. In Appendix B a visual representation of the procedure (Fig. B.1) and the algorithms for model assessment (Alg. 4) and model selection (Alg. 5) can be found.

Hyperparameters

The training mechanism has several parameters, that can be tuned to optimise the process [36]. These parameters are referred to as hyperparameters to distinguish them from basic parameters like weights and biases. They can be tuned manually, with grid or random search, or using Bayesian methods to predict the next hyperparameter configuration in a series of trials [36].

For the GIN model, the original authors mention tuning the number of hidden channels, the batch size and the dropout ratio [27]. Errica et al. [21] additionally tune the number of layers. For the DGCNN model, the original authors mentioned tuning only the learning rate [26] while Errica et al. additionally tune the number of layers and the number of hidden channels. In our experiment, the extended list of hyperparameters of Errica et al. was loosely followed. For the baseline MLP model, they tune batch size, number of hidden channels learning rate and weight decay. A full overview of the used values for each hyperparameter can be found in Appendix B Table B.1.

Witten et al. [36] provide detailed explanations for the hyperparameters that are set or tuned during the training process. Batch size refers to the number of graphs that are used for training a model in one iteration before the parameters are updated. The learning rate determines how much parameters are changed, after

[2]The code is available at: `https://github.com/diningphil/gnn-comparison`

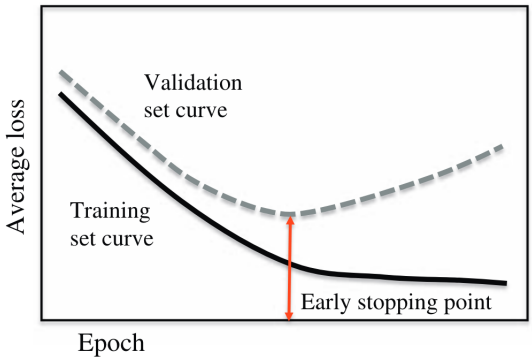


Figure 4.1: Typical curves for training with a validation set. Once the validation loss starts to increase, the training should be stopped [36].

each calculation of the loss. It can be a fixed value or can be adjusted over time with a learning rate scheduler (as is the case for the GIN model [27]). Weight decay is a regularisation technique that encourages the model to minimise weights. Dropout is a method that randomly removes some nodes during training to prevent overfitting. By creating a multitude of networks with some connections missing, the model cannot rely too heavily on specific features. Overfitting refers to a model performing well on training data but losing its ability to generalise to unseen data. One way to help avoid this is by using a validation set for early stopping. The validation loss is observed alongside the training loss and once the validation loss starts to increase while the training loss keeps decreasing, the training is stopped. Figure 4.1 illustrates at which point early stopping should occur [36]. Additional hyperparameters to set include the number of epochs, an epoch being one pass over the entire training set, and the number of trials for hyperparameter optimisation, meaning how many configurations to try.

For optimisation, PyTorchs implementation of the Adam algorithm [37] is used following the original implementations of both GIN and DGCNN as well as Errica et al. For hyperparameter tuning Optuna's TPE (Tree-structured Parzen Estimator) sampler[3] is used which is a Bayesian method. The chosen number of trials is oriented on the maximum number of hyperparameter combinations. A full list of the used hyperparameters can be found in Appendix B Table B.1.

[3]Documentation for the TPE sampler at: `https://optuna.readthedocs.io/en/stable/reference/samplers/generated/optuna.samplers.TPESampler.html`

4.2 Experiment Execution

4.2.1 Model Architecture

GIN

In the following the architecture of the GIN model is presented following the description of the original authors Xu et al. [27]. For the convolutional layers, the following propagation rule is proposed:

$$h_u^{(k+1)} = \text{MLP}^{(k+1)} \left((1 + \epsilon^{(k+1)}) \cdot h_u^{(k)} + \sum_{v \in N(u)} h_v^{(k)} \right) \tag{4.1}$$

with ϵ being either a learnable parameter or a fixed scalar representing the importance of a node's own features $h_u^{(k)}$ in comparison to those of its neighbour nodes. The weighted node features are added to the sum of the neighbour node features $h_v^{(k)} \mid v \in N(u)$. Finally, the aggregated features are passed to an MLP to create a more expressive feature representation. The reader is referred to Eq. (3.2) to observe that Eq. (4.1) represents a message-passing mechanism. For graph classification tasks where a readout layer is needed, the following function is proposed to obtain a graph representation:

$$h_G = \text{CONCAT} \Big(\text{READOUT} \Big(\{ h_u^{(k)} \mid u \in G \} \Big) \mid k = 0, 1, \ldots, K \Big) \tag{4.2}$$

It is suggested that for the READOUT function, all nodes of the same iteration are summed up. The implementation is further described as using five GNN layers including the input layer, and all MLPs having two layers (excluding the input layer according to the original code[4]). Additionally, batch normalisation is applied on every hidden layer and the final dense layer (i.e., single fully connected layer) has a dropout layer. From the original code, it can be gathered that ReLU was used as the activation function.

To summarise: After each convolutional layer, the node embeddings are summed up to get a representation of the graph at that particular iteration. In the final step, the graph representations from the different iterations are concatenated to form a final graph representation which reflects information from all layers. This can then be passed to a final classifier.

[4]The code is available at: `https://github.com/weihua916/powerful-gnns`

DGCNN

In the following, the architecture of the DGCNN model is presented following the description of the original authors Zhang et al. [26]. They describe the graph convolution as:

$$H^{(k+1)} = f\left(\tilde{D}^{-1}\tilde{A}H^{(k)}W^{(k)}\right) \tag{4.3}$$

with $\tilde{A} = A + I$ being the adjacency matrix with added self-connections and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ being the degree matrix with the added self-connections. $H^{(k)}$ is the feature matrix, $W^{(k)}$ is a trainable weight matrix and f is the activation function. The convolution is very similar to the convolution in the GCN model in Eq. (3.6), the difference being the normalisation with the modified degree matrix with $\tilde{D}^{-1}\tilde{A}$ rather than $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$. The DGCNN architecture is further described as containing the following layers, as illustrated in figure 4.2:

- 3 graph convolution layers with 32 output channels and tanh as activation function.
- 1 graph convolution layer with 1 output channel used for sorting and tanh as activation function.
- SortPooling with k so that 60% of graphs have more than k nodes.
- 1D convolution layer with 16 output channels and ReLU as activation function.
- MaxPooling layer with filter size 2 and step size 2.
- 1D convolution layer with 32 output channels, filter size 5, step size 1 and ReLU as activation function.
- Dense layer with 128 hidden channels, dropout rate 0.5 and ReLU as activation function.

Based on the original code[5] the filter and step size of the first 1D convolution layer is equivalent to the total latent dimension D:

$$D = (number\_of\_layers - 1) \cdot output\_channels + 1$$

The calculation of the input dimension of the dense depends on the output size of the last 1D convolution layer. This output size is in turn determined by the first 1D convolution layer and the max pooling layer.

[5]The code is available at: `https://github.com/muhanzhang/pytorch_DGCNN`

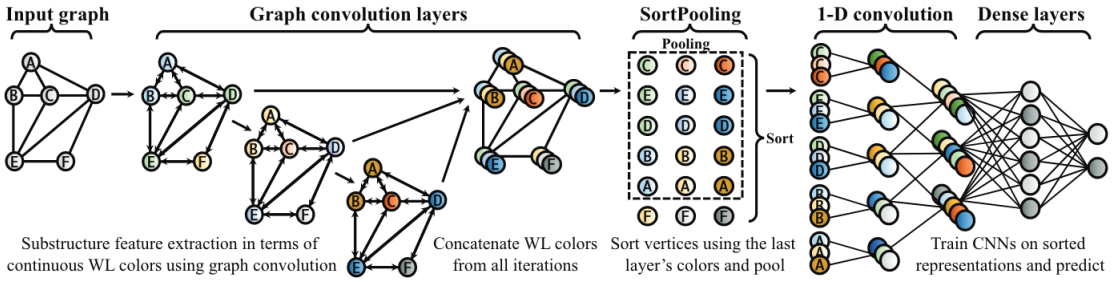


Figure 4.2: The structure of the DGCNN model, with node features visualised as colours. Graph convolution layers are followed by a SortPooling layer, traditional 1D convolution and a final dense layer [26].

- **1D convolution layer 1:** Starting from the first 1D convolution layer, the input size depends on the output size of the SortPooling layer, which is k. The original code also uses k as an approximation for the output size of the first 1D convolutional layer. So, let's denote the output size of the first 1D convolution layer as $O = k$.

- **MaxPool layer:** Next, the output size of the MaxPool layer can be calculated using the formula:

$$O' = (((W'–F')/S') + 1) = floor((O - 2)/2) + 1) = floor(O/2)$$

  Where the input size W' is equal to the output size O of the previous layer, and the filter size (F') and step size (S') are 2

- **1D convolution layer 2:** Finally, the output size of the second 1D convolution layer can be calculated similarly, with the additional multiplication with C'' for the number of output channels:

$$O' = (((W''–F'')/S'') + 1) \cdot C'' = (((O'–5)/1) + 1) \cdot 32 = (O'–4) \cdot 32$$

  Here the input size W'' is equal to the output size of the previous layer, in this case O'. The filter size (F') and step size (S') are 5 and 1 respectively. The number of output channels is 32.

- **Dense layer:** By inserting the previous formulas, we get the following input dimension of the dense layer O'':

$$O'' = (floor(O/2)–4) \cdot 32 = (floor(k/2)–4) \cdot 32 \tag{4.4}$$

4.2.2 Code

For the implementation[6] of the experiment, several machine learning libraries were used. PyTorch Geometric (PyG) [28] and PyTorch Lightning (PL) [29] are both libraries built on top of PyTorch [38]. PyG provides implemented GNN architectures and methods, as well as common benchmark datasets and tools for handling graph data. PL simplifies complex training pipelines by abstracting boilerplate code, thus making it more readable. It also manages training loops and data handling. Optuna [30] is a library designed for hyperparameter optimisation and provides an interface for managing and tracking optimisation trials.

Key Elements

The GNN architectures are implemented as PyTorch `Modules`. They are then wrapped by one general `PL.LightningModule` which initialises the specific GNN model based on user input and defines the training, validation and testing steps as well as the optimiser. The data is handled by a `PL.LightningDataModule` which ensures that the data is pre-processed and split into training, validation, and test sets for model training and evaluation. The hyperparameter optimisation is organised by an `optuna.Study` object which optimises based on an objective function. The `optuna.Trial` object is one single execution of the objective function. The objective function returns the validation accuracy for one hyperparameter configuration. The training is handled by the `PL.Trainer` object, which can be configured with several callback and logging functionalities. The used callbacks are early stopping, trial pruning and model checkpoint. The early stopping is set up to monitor validation loss. The loss was chosen because the loss curve is usually smoother than the accuracy curve, and at the point of early stopping the increase in loss is more distinct than the decrease in accuracy. The trial pruning monitors validation accuracy and prunes unpromising trials. The model checkpoint saves the parameters of the model at the time of maximum validation accuracy. The `main` function involves setting up arguments for running the script, and running the main training and evaluation loop for the specified number of repetitions and folds. The user can specify various parameters such as the model, dataset and experiment type as well as the number of folds, repetitions, epochs, and trials for hyperparameter optimisation.

[6]Full code available at: `https://github.com/C8XY66/GNN_TrainingFramework`

Implementation

Three elements of the code are discussed in more detail and illustrated with pseudocode to provide an overview of the learning pipeline: the `GraphDataModule` class, the `objective()` function, and the `main()` function.

An overview of the structure of the `GraphDataModule` class is provided in Algorithm 1. The `prepare_data()` method (line 2) is called once per run. It handles the data loading and any necessary node feature transformations. If the dataset is of the social type, it adds a one-hot encoding of node degrees as features (line 5). If the experiment type is without node features, it neutralises the features (line 9). The `setup_rep()` method (line 11) is called once per repetition. It shuffles the dataset deterministically based on a repetition-specific seed and splits it into stratified folds. The `setup()` method (line 15) is called once per fold. It makes a stratified split of the train folds into a train and validation set based on a fold-specific seed. The seeds are set up this way to ensure some level of randomness, but make it possible to continue a run from a specific repetition and fold or run folds in parallel on separate CPUs.

Algorithm 1 GraphDataModule Class

Require: DataSet Ds, DatasetType Dt, Experiment Ex, Folds F

```
 1: class GraphDataModule:
 2:     def prepare_data():
 3:         pre_transform ← None
 4:         if Dt = social then
 5:             pre_transform ← OneHotEncoding(degree)
 6:         end if
 7:         dataset ← TUDataset(Ds, pre_transform)
 8:         if Ex = WithoutNF then
 9:             neutralise_node_features(dataset)
10:         end if
11:     def setup_rep(rep):
12:         seed_r ← rep + 1
13:         dataset ← shuffle(Ds, seed_r)
14:         train_test_splits[ ] ← stratified_k_fold(dataset, F)
15:     def setup(fold):
16:         seed_f ← fold + 1
17:         train_set, test_set ← train_test_splits[fold]
18:         train_set, val_set ← stratified_split(train_set, ratio, seed_f)
19: end class
```

Next the `objective()` function of the `Study` object is illustrated with Algorithm 2. It first loads the config file (line 1) and initialises the hyperparameters with default values (line 2). It then checks the file for the hyperparameters to be tuned and suggests a hyperparameter configuration (line 4). The configuration is then used to initialise the GNNModel (line 6). Next a `Trainer` object is created (line 7). The hyperparameters are logged (line 8) and finally, the trainer is used to train the initialised model with the `trainer.fit()` method (line 9). The method primarily returns the validation accuracy, as this is the value that is to be maximised by the study object. However, the validation loss is used to break ties if two trials report the same validation accuracy.

Algorithm 2 Objective Function

Require: Model M, DataModule Dm, DatasetType Dt, Trial t, Epochs Ep

1: config ← load_config_file(M)
2: hyperparameters ← default_values
3: **for** param, values in config.items() **do**
4: hyperparameters[param] ← t.suggest(param, values)
5: **end for**
6: model ← GNNModel(M, Dt, k, hyperparameters)
7: trainer ← create_trainer(t, Ep)
8: trainer.log(hyperparameters)
9: val_acc, val_loss ← trainer.fit(model, Dm)
10: **return** val_acc, val_loss

Lastly the `main()` function as illustrated with Algorithm 3 is responsible for running the main experiment loop. Before the loop, the function sets up a folder to save all logs and results (line 1) and initialises a `GraphDataModule` with the name of the dataset and experiment type (line 2). It handles the data modules as described: `prepare_data()` (line 3) once per run, `setup_rep()` (line 5) once per repetition and `setup()` (line 8) once per fold. It also creates a log folder for every fold (line 7). Next, it creates a `Study` object for every fold (line 9) and uses the `study.optimise()` function to optimise the specified number of trials with an `objective()` function for each trial (line 10). The model of the best trial, at its best validation accuracy, is then accessed via its saved checkpoint (line 11). A `Trainer` object is initialised in training mode, meaning no callbacks are used (line 12). Finally, the trainer is used to test the best model with the `trainer.test()` method (line 13) and the test accuracy is saved to an SQLite database (line 14).

Algorithm 3 Main Function

Require: ModelName Mn, DataSet Ds, DatasetType Dt, Experiment Ex, Repetitions R, Folds F, Trials T, Epochs Ep

```
create_parent_dir()
datamodule ← GraphDataModule(Ds, Ex)
datamodule.prepare_data()
for r = 1 to R do
    datamodule.setup_rep(r, F)
    for f = 1 to F do
        create_sub_dir(r, f)
        datamodule.setup(f)
        study ← optuna.create_study()
        study.optimise(trial: objective(trial, Mn, Dm, Dt, Ep), T)
        best_model ← study.best_trial.checkpoint
        trainer ← create_trainer(testing)
        test_acc ← trainer.test(best_model)
        save_test_result(test_acc)
    end for
end for
```

For the GIN and DGCNN models, the architecture was implemented as described by the original authors and elaborated on in Sections 4.2.1 and 4.2.1 respectively. The full code for the GIN and DGCNN model implementation can be found in Appendix B. For the GIN model `GINConv` , PyGs implementation of GINs convolutional layer was used. For the DGCNN model `SortAggregation` , PyGs implementation of SortPooling was used. Additionally, for DGCNNs graph convolution layer the implementation by Errica et al. [21] was used. For the MLP baseline models, their implementation was followed. All calculations were performed on UBELIX[7], the HPC cluster at the University of Bern. An overview of the computation times can be found in Appendix B Table B.2.

[7]UBELIX: `http://www.id.unibe.ch/hpc`

	Model	**MUTAG**	**PROTEINS**	**NCI1**	**IMDB-B**	**IMDB-M**
Experiment	GIN_wNF	84.0 ± 8.3	74.2 ± 4.2	$\mathbf{80.7 \pm 2.2}$	$\mathbf{73.4 \pm 3.8}$	49.7 ± 3.8
	GIN_woNF	$\mathbf{85.9 \pm 8.1}$	72.5 ± 4.3	75.1 ± 2.5	71.9 ± 4.5	48.7 ± 3.9
	DGCNN_wNF	80.5 ± 9.5	72.7 ± 3.7	67.0 ± 2.3	71.6 ± 5.1	47.8 ± 3.3
	DGCNN_woNF	80.8 ± 9.7	71.7 ± 3.8	62.4 ± 2.4	50.9 ± 4.1	36.4 ± 3.2
	MLP_wNF	80.4 ± 9.7	$\mathbf{75.1 \pm 4.6}$	69.5 ± 2.2	72.1 ± 4.9	$\mathbf{50.5 \pm 3.8}$
	MLP_woNF	81.7 ± 10.2	71.1 ± 3.9	62.3 ± 2.2	50.1 ± 4.4	36.0 ± 3.5
Errica et al.	GIN_wNF	-	73.3 ± 4.0	$\mathbf{80.0 \pm 1.4}$	$\mathbf{71.2 \pm 3.9}$	48.5 ± 3.3
	GIN_woNF	-	-	-	66.8 ± 3.9	42.2 ± 4.6
	DGCNN_wNF	-	72.9 ± 3.5	76.4 ± 1.7	69.2 ± 3.0	45.6 ± 3.4
	DGCNN_woNF	-	-	-	53.3 ± 5.0	38.6 ± 2.0
	MLP_wNF	-	$\mathbf{75.8 \pm 3.7}$	69.8 ± 2.2	70.8 ± 5.0	$\mathbf{49.1 \pm 3.5}$
	MLP_woNF	-	-	-	50.7 ± 2.4	36.1 ± 3.0
Orig.	GIN_wNF	89.4 ± 5.6	76.2 ± 2.8	82.7 ± 1.6	75.1 ± 5.1	52.3 ± 2.8
	DGCNN_wNF	85.8 ± 1.7	75.5 ± 0.9	74.4 ± 0.5	70.0 ± 0.9	47.8 ± 0.9

Table 4.1: Results on chemical and social datasets with mean accuracy and standard deviation as percentages. Best performance within a group in bold.

4.3 Results and Discussion

Our experiments evaluated the effectiveness of the GIN [27] and DGCNN [26] models for graph classification tasks, both with and without node features (wNF/woNF). In addition, the performance of the models was compared to the classification capabilities of an MLP, which did not utilise graph structure. The performance was defined as a model's test accuracy. The results were compared to those from the original papers, and those reported by Errica et al. [21]. Table 4.1 shows the compared results, test accuracies are given as percentages.

Our results show, that GIN consistently outperformed DGCNN, scoring higher test accuracies on all datasets with and without node features. The MLP baseline managed to outperform the GNNs on the PROTEINS and the IMDB-MULTI dataset. The biggest difference in performance was on the NCI1 dataset where GIN scored 80.7 ± 2.2 while DGCN scored 67.0 ± 2.3. The overall best accuracy was scored on the MUTAG dataset where GIN reached 84.0 ± 8.3, DGCNN reached 80.5 ± 9.5 and MLP reached 80.4 ± 9.7. Noticeably, the standard deviation (SD) was higher on the MUTAG dataset than on the others. Additionally, each of the three models scored lower on graphs with node features than without. However, one should note that MUTAG is by far the smallest dataset with 188 graphs, the second-smallest being IMDB-BINARY with 1000 graphs (App. A, Tab. A.3). With

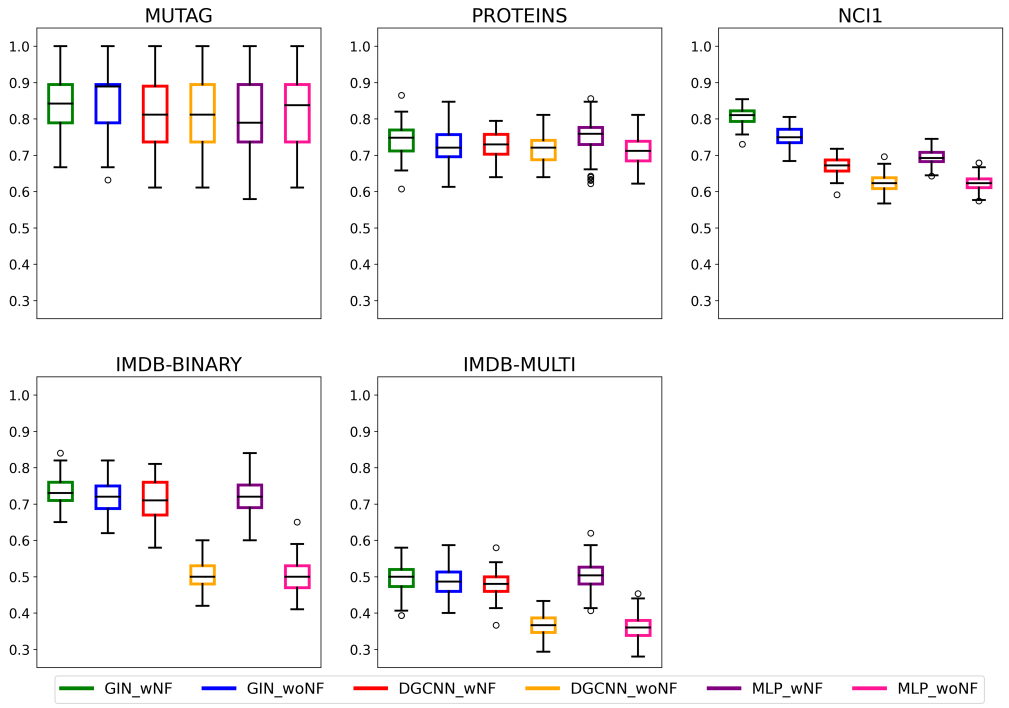


Figure 4.3: Box plots comparing the models' median test accuracies (horizontal line). The box represents the interquartile range (IQR). The whiskers extend up to 1.5 times the IQR, with outliers marked as circles.

our evaluation framework, the MUTAG dataset only had roughly 19 graphs in its test sets and roughly 2 in its validation sets. This could account for the bigger SDs and might lead to unstable results. On the IMDB-BINARY dataset, what stands out is that DGCNN and MLP lost their ability to effectively classify the graphs, both reaching an accuracy of around 50%. This equates to a random choice for a binary classification tasks. On the IMDB-MULTI dataset, which distinguishes between three classes, a similar trend could be observed. GIN on the other hand reached much more similar accuracies with and without node features on both of the social datasets. Figure 4.3 shows a box plot of our median test accuracies. The y-axis is fixed to allow a visual comparison of the models across all datasets. The boxes represent the interquartile ranges (IQRs), with the lower and upper edges indicating the 25th and 75th percentiles, respectively. The whiskers show variability outside the lower and upper quartiles, extending to the minimum and maximum within 1.5 times the IQR. Outliers are marked by circles. The box plots show where the majority of the data points lie. By comparing them, we can evaluate the performance consistency of the models. Smaller boxes and whiskers represent a more consistent performance. Our results suggest that performance consistency mostly depended on the dataset rather than the model or the presence of node features.

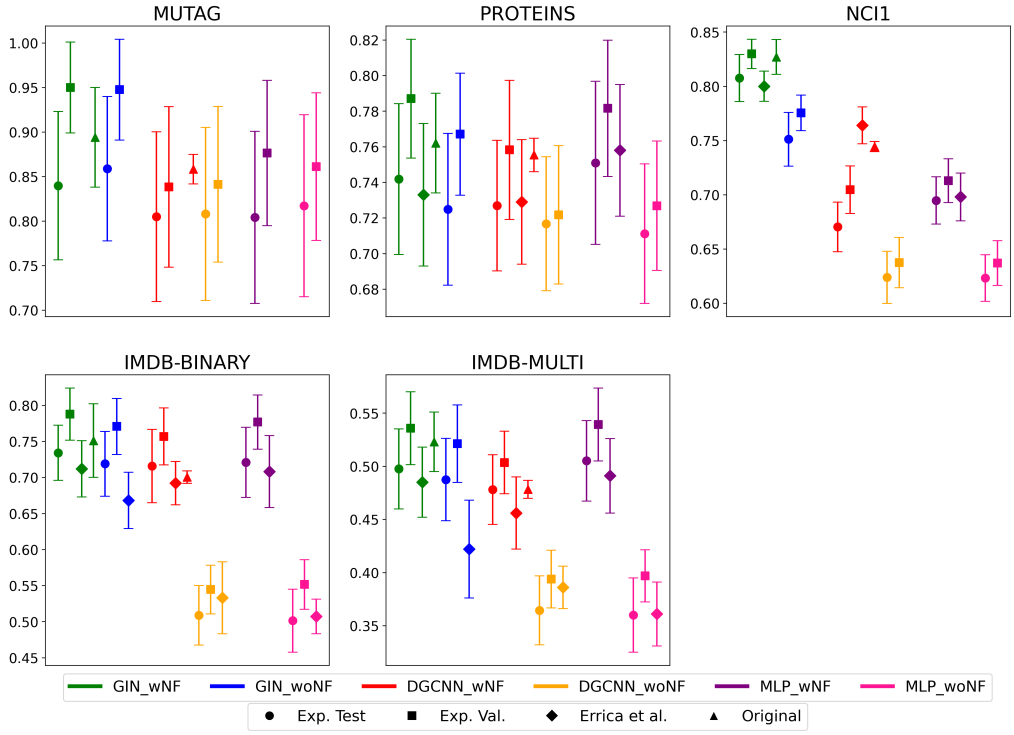


Figure 4.4: Mean accuracy and standard deviation of testing and validation, plotted with the results from Errica et al. [21] and the respective original sources of the models [27] [26].

Comparing our results with those presented by Errica et al. [21], the models overall showed similar performance, with each dataset having the same top-performer model for both sets of results. On chemical datasets, our results were higher for GIN and lower for DGCNN and MLP. On social datasets, our results were higher for all models on graphs with node features. Overall, on datasets with node features, our results never deviated more than 2.4% from their reported mean, except for DGCNN's performance on the NCI1 dataset. Our experiments showed an accuracy of 67.0 ± 2.3 on the NCI1 dataset by DGCNN while Errica et al., reported an accuracy of 76.4 ± 1.7. The difference in the performance of DGCNN might be explained by slight differences in the implementation. Errica et al. for example used a concatenation of the outputs from the graph convolution layer rather than just using the last channel as the original source did, which we followed. They also deviated from the original GIN model by only using a 1-layer MLP instead of a 2-layer MLP (excluding the input layer). Regarding the SDs, Figure 4.4 shows that in general our differences between datasets matched those of Errica et al., e.g., PROTEINS leading to larger SDs than NCI1. We can also see that in our implementation, the pattern of results on IMDB-MULTI more closely matched the one on

IMDB-BINARY. The range of accuracies is lower, because of IMDB-MULTI posing a multi-class classification problem, instead of a binary one. The results of Errica et al. show their implementation of GIN_woNF being an outlier, while the other results also match the pattern on IMDB-BINARY. One thing that we evaluated, which Errica et al. did not, was the performance of the MLP baseline on chemical datasets without node features. The performance was clearly above random levels, reaching an accuracy of 71.1 ± 3.9 on the PROTEINS dataset.

Our implementation of the models consistently reached lower results than those reported by the original sources. In Figure 4.4 our validation accuracy is plotted alongside our test accuracy to compare them to the results by Errica et al. and the original source. In the case of the GIN model, the original source reported the validation accuracy rather than the test accuracy (Sec. 4.1). However, rather than matching our validation accuracy, their reported results lie in between our test and validation accuracies. In the case of DGCNN, it is important to note that the average of the 10-time repetition of the CV was reported, rather than the overall average, which accounts for the small SDs (Sec. 4.1). Regarding the original results being higher than the ones achieved by us, there is no obvious explanation. However, the model is more complex to implement and steps like the calculation of the input dimension from Section 4.2.1 were not documented in detail.

Chapter 5

Conclusions and Future Work

In this thesis, an examination of graph neural networks (GNNs) was performed. A thorough introduction to GNNs was given, starting with the theoretical background before introducing different approaches and GNN models. The models were both unified under the framework of message passing and divided according to the task they were designed for. In addition, the thought process behind the experimental setup, as well as its execution, was shared in detail.

From our experiment, we learned the following key points. The GIN model outperforms the DGCNN model on the used datasets. There are both chemical and social datasets (PROTEINS and IMDB-MULTI), where an MLP baseline model can outperform GNN models, specifically designed for graph classification tasks. On social datasets, the addition of node degrees as node features generally benefits the performance of the model. However, while the DGCNN and MLP models need the added degrees to have any classification abilities, GIN manages to score a similar performance with and without node features. The performance consistency primarily depends on the dataset, rather than the model, or the presence of node features. By adding experiments using the MLP baseline on chemical datasets without node features, we extend the question of how the performance of GNN models should be assessed and what they should be compared to.

In addition, by being able to mostly reproduce the results by Errica et al. [21] we show that their attempt at setting up a uniform evaluation framework for GNNs has been successful. However, while they state that they followed the original papers (over the original code if there were discrepancies), they made alterations to the theoretical specifications without explanation. In addition, more information about how they chose the additional hyperparameters to use for optimisation would have been interesting. By choosing to add the MUTAG dataset specifically for its difference in size, we saw that the evaluation framework, or at least its execution, might need to be altered for smaller datasets. A starting point would be to use a 5-fold

CV instead of a 10-fold CV to increase the validation and test set size. Additionally, the model from the best trial could be retrained on the entire training set, however, in contrast to Errica et al., instead of using a validation set for early stopping, the number of epochs from the best trial could be used for the retraining. Regarding further changes to the evaluation framework, an inner CV could be used for hyperparameter optimisation, instead of the implemented holdout technique. This could reduce the variance; however, a substantial amount of additional computing power would have to be available.

Regarding the question of whether GNNs leverage graph structure for classification tasks and the hypothesis that they do not to their fullest potential, no final answer was found. However, the claimed success of the evaluated models, DGCNN in particular, was pulled further into question. During the process of trying to answer our research question, a new question arose, specifically as posed by Errica et al. [21]: If GNN performance is similar to our structure-agnostic baseline, should we conclude that the GNN models are not exploiting graph structure adequately, or should we ask if the task does not need topological information to be solved effectively? They explain that this question can easily be answered by domain-specific human expertise. However, in light of the MLP performing adequately on chemical datasets without node features, it is still interesting to see what such a "simple" model can achieve with only the information of the number of nodes available. It serves as a reminder that MLPs, despite their simplicity, can model complex functions, given enough neurons and layers.

The question of depth also arises in the context of GNNs. Most GNN models have relatively shallow structures because adding more layers would lead to over-smoothing. However, there is a very recent attempt at adapting the success of deep CNNs for GNNs. Zhou et al. [39] propose a novel Deep Graph Convolutional Neural Network (DGCNNII), which is up to 32 layers deep and based on the DGCNN model. They introduce a trick to eliminate over-smoothing and report superior results on most chemical datasets. However, while they do report using a 10-time repetition of a 10-fold CV and describe their used hyperparameters, they do not provide their code. This unfortunately led to a failed attempt at implementing their proposed architecture. There are also other lesser-known and some newer GNN models, which report good results in classification tasks. An extensive overview is provided in Appendix C Table C.1 and C.2.

In the future, there will hopefully be more rigorous comparisons of GNN models, including a bigger range of models. As this research area is still in its infancy, there is reasonable hope that the research will become more standardised in the future, leading to more credible comparisons and reproducible proposals.

Appendix A

GNN Models and Datasets

Model	Citations	Year
GCN [16]	23'362	2016
GraphSAGE [17]	10'007	2017
GAT [18]	9'657	2017
ChebNet [15]	6'981	2016
Spectral CNN [14]	4'742	2013

Table A.1: Citation count of popular GNN models. The year refers to the original year of publishing (source: Google Scholar 09.05.2023).

Model	Citations	Year
GIN [27]	4'637	2018
PSCN [25]	2'170	2016
DGCNN [26]	1'244	2018
DiffPool [22]	1'194	2018
ECC [20]	1'173	2017
SAGPool [24]	803	2019
g-U-Nets [23]	797	2019

Table A.2: Citation count of GNN models for graph classification tasks. The year refers to the original year of publishing (source: Google Scholar 09.05.2023).

Dataset	# Graphs	# Classes	# Node labels
MUTAG	188	2	7
PROTEINS	1'113	2	3
NCI1	4'220	2	37
IMDB-BINARY	1'000	2	-
IMDB-MULTI	1'500	3	-

Table A.3: Statistics of the used datasets (`https://chrsmrrs.github.io/datasets/docs/datasets/`). If the graphs have no labels, the node degrees can be added as labels.

Appendix B

Experiment

Evaluation Framework

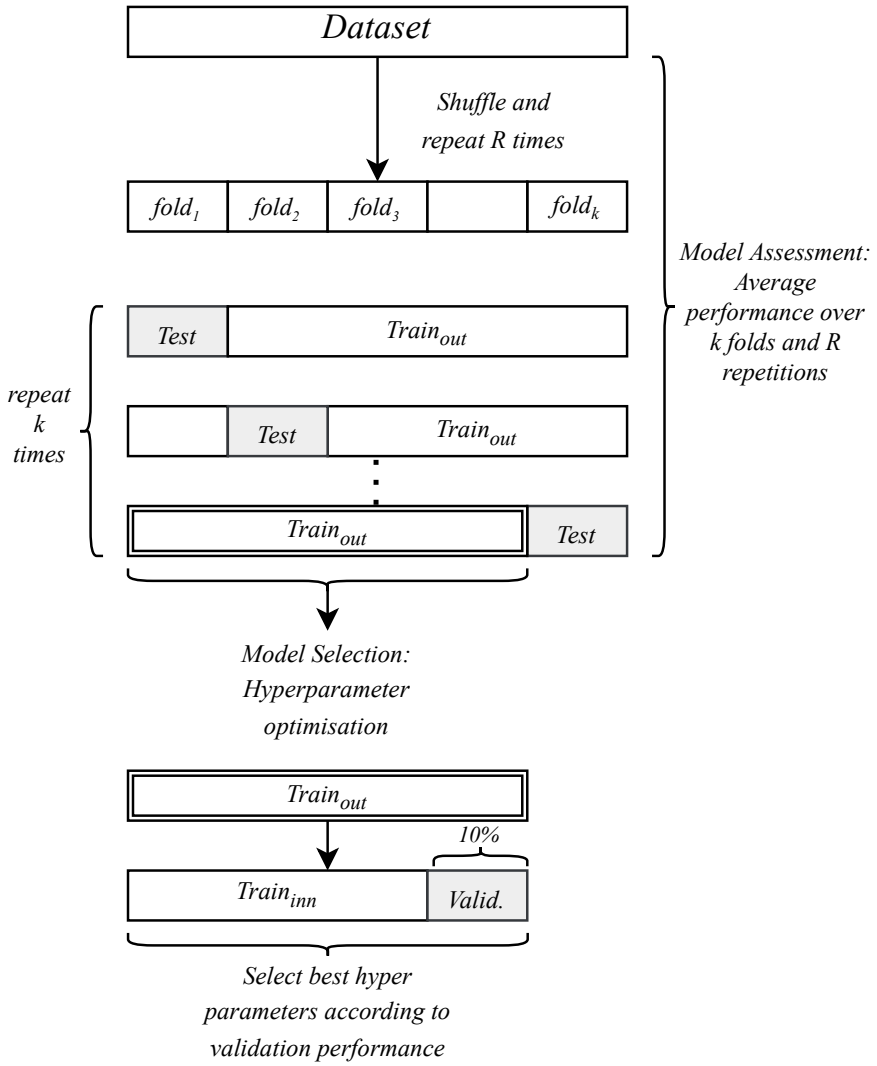


Figure B.1: Visual representation of the evaluation framework.

Algorithm 4 Model Assessment (k-fold CV)

Require: Dataset D, set of configurations Θ, number of repetitions R

for $r \leftarrow 1$ to R **do**
  Shuffle D
  Split D into k folds $F_1, \ldots, F_k$
  for $i \leftarrow 1$ to k **do**
    $train_k, test_k \leftarrow (\bigcup_{j \neq i} F_j), F_i$
    $best_{model} \leftarrow \text{Select}(train_k, \Theta)$
    $perf_k \leftarrow \text{Eval}(best_{model}, test_k)$
  end for
  $perf_r \leftarrow \sum_{i=1}^{k} perf_i / k$
end for
$perf \leftarrow \sum_{r=1}^{R} perf_r / R$
return $perf$

Algorithm 5 Model Selection

Require: $train_k$, Θ

$valid \leftarrow^{R} 10\% * train_k$
$train \leftarrow train_k \setminus valid$
$best_{model} \leftarrow$ None
$best_{perf} \leftarrow -\infty$
for all $\theta \in \Theta$ **do**
  $model \leftarrow \text{Train}(train, \theta)$
  $perf \leftarrow \text{Eval}(model, valid)$
  if $perf > best_{perf}$ **then**
    $best_{perf} \leftarrow perf$
    $best_{model} \leftarrow model$
  end if
end for
return $best_{model}$

In Algorithm 4, "Select" refers to Algorithm 5. "Train" and "Eval" represent the training and testing phases respectively. After each model selection, the best model is used to evaluate the external test fold. Performances are averaged across the k folds and R repetitions of the CV.

Hyperparameters

Model	Layers	Batch Size	Learn. Rate	Hidden Ch.	Weight Decay	Dropout
MLP	-	32	1.00E-01	32	1.00E-02	-
		128	1.00E-03	128	1.00E-03	
			1.00E-06	256	1.00E-04	
GIN	5	32	1.00E-02	32	-	0.0
		128		64		0.5
DGCNN	3	50	1.00E-04	32	-	0.5
	4		1.00E-05	64		
	5					

Model	Agg.	Epochs	Patience	Trials	Optimiser	LR Scheduler
MLP	sum	1000	100	50	ADAM	-
GIN	sum	1000	100	50	ADAM	step: 50, gamma: 0.5
DGCNN	mean	1000	100	50	ADAM	-

Table B.1: Hyperparameter configurations for model selection.

Code Implementation

```
class GINMLPModel(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.mlp = nn.Sequential(
                nn.Linear(in_c, out_c), nn.BatchNorm1d(out_c), nn.ReLU(),
                nn.Linear(out_c, out_c), nn.BatchNorm1d(out_c), nn.ReLU(),
                nn.Linear(out_c, out_c))

    def forward(self, x):
        x = self.mlp(x)
        return x

class GINModel(nn.Module):
    def __init__(self, in_c, out_c, hid_c,
        num_layers, dropout, train_eps):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.layers = nn.ModuleList()
        # GRAPH CONVOLUTION
        self.layers.append(GINConv(GINMLPModel(in_c, hid_c), train_eps))
        self.layers.append(nn.BatchNorm1d(hid_c))
        for _ in range(num_layers - 2):
            self.layers.append(GINConv(GINMLPModel(hid_c, hid_c), train_eps))
            self.layers.append(nn.BatchNorm1d(hid_c))
        self.layers.append(GINConv(GINMLPModel(hid_c, hid_c), train_eps))
        # DENSE LAYER
        self.fc1 = nn.Linear(num_layers * hid_c, hid_c)
        self.fc2 = nn.Linear(hid_c, out_c)

    def forward(self, x, edge_index, batch):
        summed_layer_outputs = []
        for i in range(0, len(self.layers), 2):
            x = self.layers[i](x, edge_index)
            if i + 1 < len(self.layers):
                x = F.relu(self.layers[i + 1](x))
        summed_layer_outputs.append(global_add_pool(x, batch))
        x = torch.cat(summed_layer_outputs, dim=-1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Python implementation of the `GINModel`. Type hints, keyword arguments and comments were removed and variable names were shortened.

```
class DGCNNModel(nn.Module):
    def __init__(self, in_c, out_c, hidden_c, num_layers, dropout, k):
        super().__init__()
        self.k = k
        self.dropout = nn.Dropout(dropout)
        self.graph_conv_layers = nn.ModuleList()
        # GRAPH CONVOLUTION
        self.graph_conv_layers.append(DGCNNConv(in_c, hidden_c))
        for _ in range(num_layers - 2):
            self.graph_conv_layers.append(DGCNNConv(hidden_c, hidden_c))
            self.graph_conv_layers.append(DGCNNConv(hidden_c, 1))
        # SORT POOL
        self.sort_pool = SortAggregation(self.k)
        # 1-D CONVOLUTION
        self.tot_lat_dim = (num_layers - 1) * hidden_c + 1
        self.conv1D_1 = nn.Conv1d(1, 16, self.tot_lat_dim, self.tot_lat_dim)
        self.max_pool = nn.MaxPool1d(2, 2)
        self.conv1D_2 = nn.Conv1d(16, 32, 5, 1)
        # DENSE LAYER
        dense_input_dim = (int(self.k / 2) - 4) * 32
        self.fc1 = nn.Linear(dense_input_dim, 128)
        self.fc2 = nn.Linear(128, out_c)

    def forward(self, x, edge_index, batch):
        for layer in self.graph_conv_layers:
            x = torch.tanh(layer(x, edge_index))

        x = self.sort_pool(x, batch)  # SortPool
        padding_size = self.k * self.tot_lat_dim - x.size(1)
        if padding_size > 0:
            padding = torch.zeros((x.size(0), padding_size), device=x.device)
            x = torch.cat((x, padding), dim=1)

        x = torch.unsqueeze(x, dim=1)
        x = F.relu(self.conv1D_1(x))
        x = self.max_pool(x)
        x = F.relu(self.conv1D_2(x))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Python implementation of the `DGCNNModel`. Type hints, keyword arguments and comments were removed and variable names were shortened.

Computation Time

	Model	**MUTAG**	**PROTEINS**	**NCI1**	**IMDB-B**	**IMDB-M**
Experiment	GIN_wNF	00 : 31	02 : 21	04 : 06	01 : 47	01 : 14
	GIN_woNF	00 : 35	01 : 52	04 : 39	01 : 57	02 : 23
	DGCNN_wNF	00 : 32	02 : 49	09 : 23	02 : 03	02 : 16
	DGCNN_woNF	00 : 39	04 : 48	12 : 09	03 : 41	05 : 00
	MLP_wNF	00 : 12	00 : 37	01 : 42	00 : 54	00 : 57
	MLP_woNF	00 : 09	00 : 30	02 : 00	00 : 48	00 : 48

Table B.2: Computation time for dataset per fold (in format hh:mm). For total computation time: value $\cdot$ 100.

Appendix C

Conclusion

Overview of GNN Models for Graph Classification

Model	MUTAG	PTC	PROTEINS	D&D	NCI1	ENZYMES
GIN [27]	89.4 ± 5.6	64.6 ± 7.0	76.2 ± 2.8	-	82.7 ± 1.7	-
DGCNN [26]	85.83 ± 1.66	58.59 ± 2.47	75.54 ± 0.94	79.37 ± 0.94	74.44 ± 0.47	-
PSCN [25]	92.63 ± 4.21	60.00 ± 4.82	75.89 ± 2.76	77.12 ± 2.41	78.59 ± 1.89	-
DiffPool [22]	-	-	76.25	80.64	-	**62.53**
ECC [20]	89.44	-	-	73.65	**83.80**	50.00
SAGPOOL [24]	-	-	70.04 ± 1.47	76.19 ± 0.94	74.18 ± 1.20	-
g-U-Nets [23]	-	-	77.68	82.43	-	-
DEMO-Net [40]	81.4	57.2	70.8	-	-	27.7
DGCNNII [39]	**94.44**	$\mathbf{76.47 \pm 2.94}$	$\mathbf{82.88 \pm 0.83}$	$\mathbf{83.33 \pm 1.29}$	80.32 ± 0.76	-
S2S-N2N-PP [41]	89.86 ± 1.10	64.54 ± 1.10	76.61 ± 0.50	-	83.72 ± 0.40	-
MA-GCNN [42]	93.89 ± 5.24	71.76 ± 6.33	79.35 ± 1.74	81.48 ± 1.03	81.77 ± 2.36	-
EigenPooling [43]	-	-	76.60	78.60	77.00	-
struc2vec [44]	88.28	-	-	82.22	83.72	61.10
NEST [45]	91.85 ± 1.57	67.42 ± 1.83	76.54 ± 0.26	78.11 ± 0.36	81.59 ± 0.46	-
GCAPS-CNN [46]	-	66.01 ± 5.91	76.40 ± 4.17	77.62 ± 4.99	82.72 ± 2.38	61.83
CapsGNN [47]	86.67 ± 6.88	-	76.28 ± 3.63	75.38 ± 4.17	78.35 ± 1.55	54.67

Table C.1: Results on chemical datasets with mean accuracy and standard deviation as percentages. Best performance within a group in bold.

Model	COLLAB	IMDB-BINARY	IMDB-MULTI	REDDIT-BINARY	REDDIT-MULTI
GIN [27]	$\mathbf{80.2 \pm 1.9}$	$\mathbf{75.1 \pm 5.1}$	$\mathbf{52.3 \pm 2.8}$	$\mathbf{92.4 \pm 2.5}$	$\mathbf{57.5 \pm 1.5}$
DGCNN [26]	73.76 ± 0.49	70.03 ± 0.86	47.83 ± 0.85	-	-
DiffPool [22]	75.48	-	-	-	-
g-U-Nets [23]	77.56	-	-	-	-
GCAPS-CNN [46]	77.71 ± 2.51	71.69 ± 3.40	48.50 ± 4.10	87.61 ± 2.51	50.10 ± 1.72
CapsGNN [47]	79.62 ± 0.91	73.10 ± 4.83	50.27 ± 2.65	-	52.88 ± 1.48

Table C.2: Results on social datasets with mean accuracy and standard deviation as percentages. Best performance within a group in bold.

Bibliography

[1] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[2] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[3] Anil K Jain, Robert P. W. Duin, and Jianchang Mao. Statistical pattern recognition: A review. *IEEE Transactions on pattern analysis and machine intelligence*, 22(1):4–37, 2000.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[5] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[6] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[7] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[8] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[9] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[10] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):249–270, 2020.

[11] Zhiyuan Liu and Jie Zhou. Introduction to graph neural networks. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(2):1–127, 2020.

[12] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[13] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[14] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[15] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29, 2016.

[16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[18] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

[19] Shima Khoshraftar and Aijun An. A survey on graph representation learning methods. *arXiv preprint arXiv:2204.01855*, 2022.

[20] Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3693–3702, 2017.

[21] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. *arXiv preprint arXiv:1912.09893*, 2019.

[22] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

[23] Hongyang Gao and Shuiwang Ji. Graph u-nets. In *international conference on machine learning*, pages 2083–2092. PMLR, 2019.

[24] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In *International conference on machine learning*, pages 3734–3743. PMLR, 2019.

[25] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023. PMLR, 2016.

[26] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[27] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

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

[29] William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019.

[30] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[31] Kristian Kersting, Nils M Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels. 2016.

[32] Asim Kumar Debnath, Rosa L Lopez de Compadre, Gargi Debnath, Alan J Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34(2):786–797, 1991.

[33] Nikil Wale, Ian A Watson, and George Karypis. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14:347–375, 2008.

[34] Karsten M Borgwardt, Cheng Soon Ong, Stefan Schönauer, SVN Vishwanathan, Alex J Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl_1):i47–i56, 2005.

[35] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1365–1374, 2015.

[36] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2016.

[37] P Kingma Diederik and LB Jimmy. Adam: a method for stochastic optimization (2014). *arXiv preprint arXiv:1412.6980*, 2014.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[39] Yuchen Zhou, Hongtao Huo, Zhiwen Hou, and Fanliang Bu. A deep graph convolutional neural network architecture for graph classification. *Plos one*, 18(3):e0279604, 2023.

[40] Jun Wu, Jingrui He, and Jiejun Xu. Net: Degree-specific graph neural networks for node and graph classification. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 406–415, 2019.

[41] Aynaz Taheri, Kevin Gimpel, and Tanya Berger-Wolf. Learning graph representations with recurrent neural network autoencoders. *KDD Deep Learning Day*, 2018.

[42] Hao Peng, Jianxin Li, Qiran Gong, Yuanxin Ning, Senzhang Wang, and Lifang He. Motif-matching based subgraph-level attentional convolutional network for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 5387–5394, 2020.

[43] Yao Ma, Suhang Wang, Charu C Aggarwal, and Jiliang Tang. Graph convolutional networks with eigenpooling. In *Proceedings of the 25th ACM SIGKDD*

international conference on knowledge discovery & data mining, pages 723–731, 2019.

[44] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[45] Carl Yang, Mengxiong Liu, Vincent W Zheng, and Jiawei Han. Node, motif and subgraph: Leveraging network functional blocks through structural convolution. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 47–52. IEEE, 2018.

[46] Saurabh Verma and Zhi-Li Zhang. Graph capsule convolutional neural networks. *arXiv preprint arXiv:1805.08090*, 2018.

[47] Zhang Xinyi and Lihui Chen. Capsule graph neural network. In *International conference on learning representations*, 2019.