BACHELOR'S DEGREE THESIS

# Degree in Statistics

**Title:** Application of Graph Neural Networks for biomedical data

**Author:** Mengru Ma

**Advisor:** Esteban Vegas Lozano

**Department:** Genetics, Microbiology and Statistics, University of Barcelona

**Academic year:** 2023-2024

UNIVERSITAT DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

## Abstract

The rapid advancement of biomedical data collection technology has enabled comprehensive data analysis, but traditional methods often struggle with complex and irregular data. While Convolutional Neural Networks (CNNs) have improved biomedical image analysis, they are less effective for non-Euclidean data like graphs. In contrast, Graph Neural Networks (GNNs) offer a promising solution by capturing complex relationships and topological structures in graph data.

This final work provides an overview of GNN applications and challenges in biomedical data analysis. Core GNN models, including Graph Convolutional Networks (GCN), Graph Attention Networks (GAT), and GraphSAGE, are explained and then implemented through two detailed case studies, demonstrating their feasibility and effectiveness.

Overall, this work highlights the potential of GNNs in bioinformatics, providing robust support for analyzing complex biomedical data and offering insights for future research and development in this field.

**Keywords**: Deep Learning in Biomedicine; Graph Neural Networks (GNNs); Biomedical Data Analysis; Machine Learning; Bioinformatics

## Resumen

El rápido desarrollo de las tecnologías de recopilación de datos biomédicos hace posible el análisis integral de datos, pero los métodos tradicionales a menudo tienen dificultades para manejar datos complejos e irregulares. Aunque las Redes Neuronales Convolucionales (CNNs) han mejorado el análisis de imágenes biomédicas, son menos efectivas para datos no euclidianos como los gráficos. Por el contrario, las Redes Neuronales de Gráficos (GNNs) ofrecen una solución innovadora al capturar relaciones complejas y estructuras topológicas en datos de gráficos.

Este trabajo final ofrece una descripción completa de las aplicaciones y retos de las GNN en el análisis de datos biomédicos. Los principales modelos de GNN, incluidos Graph Convolutional Networks (GCN), Graph Attention Networks (GAT) y GraphSAGE, se explican y luego se implementan a través de dos estudios de caso detallados, demostrando su viabilidad y eficacia.

En general, este trabajo destaca el potencial de las GNN en bioinformática, brindando un apoyo sólido para el análisis de datos biomédicos complejos y proporcionando información útil para futuras investigaciones y desarrollos en este campo.

**Palabras clave**: Aprendizaje profundo en biomedicina; Redes Neuronales de Gráficos (GNNs); Análisis de datos biomédicos; Aprendizaje automático; Bioinformática

## AMS Classification

- 05Cxx Graph theory
- 62P10 Applications to biology and medical sciences
- 68Txx Artificial intelligence
- 68T07 Artificial neural networks and deep learning

# Acknowledgements

Firstly, I would like to thank my tutor for providing me with this research topic, giving me the opportunity to delve into deep learning and Graph Neural Networks (GNNs), and continually offering suggestions and feedback that helped me improve throughout the process.

Secondly, I want to thank my friends who have been by my side during this period. I am especially grateful for your support and encouragement during my most stressful times.

Lastly, I want to express my gratitude to my university, especially the teachers who guided and taught me. Your guidance has helped me discover my passion, and I hope to continue contributing to this field in the future.

# Contents

# List of Figures

# List of Tables

# Glossary

| Term | Description |
|---|---|
| **Activation Function** | Function used to introduce non-linear properties, such as ReLU, Sigmoid, Tanh, etc. |
| **Adjacency Matrix** | A matrix representing graph structure, where $A_{ij} = 1$ if there is an edge between node i and node j. |
| **Aggregation Function** | Function used in GNNs to aggregate features from neighboring nodes, such as sum, mean, max, etc. |
| **Attention Mechanism** | Mechanism in GATs for assigning different weights to different neighboring nodes based on importance. |
| **Backpropagation** | Algorithm for updating neural network weights by computing gradients. |
| **Batch Normalization** | Technique for normalizing inputs of each layer to improve training speed and stability. |
| **Batch Size** | Number of training examples utilized in one iteration. |
| **Convolutional Neural Network (CNN)** | Neural network designed for processing grid-like data such as images. |
| **Degree Matrix** | Diagonal matrix where elements represent the degree of corresponding nodes. |
| **Dropout** | Regularization technique to prevent overfitting by randomly dropping neurons during training. |
| **Edge** | Connection between two nodes in a graph. |
| **Epoch** | One complete pass through the entire training dataset. |
| **Feature** | Attribute or property of a node or edge in a graph. |
| **Graph Attention Network (GAT)** | GNN variant using attention mechanisms to assign different weights to different neighbors. |
| **Graph Convolutional Network (GCN)** | GNN variant using convolution operations to aggregate node features. |
| **Graph Isomorphism Network (GIN)** | GNN variant designed to be as powerful as the Weisfeiler-Lehman graph isomorphism test. |
| **Graph Neural Network (GNN)** | Neural network designed for processing graph-structured data. |
| **GraphSAGE** | GNN variant using sampling and aggregation of neighbor features for inductive learning. |
| **Hidden Layer** | Layers in a neural network between the input and output layers. |

| | |
|---|---|
| **Inductive Learning** | Learning process where the model generalizes to unseen data or nodes. |
| **Learning Rate** | Hyperparameter that controls the step size during the optimization process. |
| **LeakyReLU** | Variant of ReLU activation function allowing a small gradient when the unit is inactive. |
| **Loss Function** | Function used to measure the difference between predicted and true values, such as MSE, cross-entropy. |
| **LSTM** | Long Short-Term Memory, a type of recurrent neural network capable of learning long-term dependencies. |
| **Message Passing** | Process in GNNs where nodes send and receive information to and from neighbors. |
| **Node** | Entity in a graph representing data points. |
| **Normalization** | Technique used to adjust the distribution of features in the input data, often used in neural networks to improve training performance. |
| **Optimization Algorithm** | Algorithm used to minimize the loss function, such as gradient descent, Adam, etc. |
| **Output Layer** | Final layer in a neural network producing the output prediction. |
| **Pooling** | Layer used for reducing the dimensionality of feature maps or graph representations in neural networks, including CNNs and GNNs, such as max pooling and average pooling. |
| **Protein-Protein Interaction (PPI)** | Interactions between protein molecules that affect their function and behavior in biological processes. |
| **Regularization** | Techniques used to prevent overfitting in a model, such as L2 regularization, dropout, etc. |
| **Transductive Learning** | Training model where the test set is known and fixed during training. |
| **Update Function** | Function in GNNs for updating node features with aggregated neighbor information. |
| **XAI (Explainable Artificial Intelligence)** | Techniques and methods in AI that try to provide clear and understandable explanations for the decision-making processes. |

# Notations

| Notations | Descriptions |
|---|---|
| $\mathbb{R}^m$ | m-dimensional Euclidean space |
| $a, \mathbf{a}, \mathbf{A}$ | Scalar, vector and matrix |
| $\theta, \Theta$ | Parameter and parameters set |
| $\mathbf{w}, \mathbf{W}$ | Weight vector and matrix |
| $\mathbf{b}$ | Bias term |
| $\mathbf{x}, \mathbf{X}$ | Input vector and matrix |
| $\mathbf{h}, \mathbf{H}$ | Hidden representation vector and matrix |
| $\mathbf{A}^T$ | Matrix transpose |
| $\mathbf{I}_N$ | Identity matrix of dimension $N$ |
| $\mathcal{N}(v)$ | Neighborhood set of node $v$ |
| $\mathbf{h}_v^l$ | Feature representation of node $v$ after the $l$-th layer. |
| $\sigma$ | Activation function |
| $\|$ | Vector concatenation |

# 1 Introduction

In the era of big data, with the rapid development of biomedical data collection technology, researchers can use more ways to conduct biomedical data analysis and research more comprehensively. Traditional methods of mining and analyzing biomedical data often fall short in efficiency when faced with a massive volume of data. This is because, on the one hand, biomedical big data features higher dimensions and more complex computations; on the other hand, the diverse types of biomedical data demand highly versatile processing capabilities from models due to their complex and variable nature. Recently, deep learning has made breakthrough progress in the field of biomedicine (Cao et al. 2018), thanks to its robust feature extraction capabilities (Hinton and Salakhutdinov 2006), making it exceptionally suited for the statistical analysis of biomedical big data and related issues.

Convolutional Neural Networks (CNNs) (LeCun et al. 1998), in particular, have seen widespread application in biomedical image analysis. Their role in extracting and analyzing biomedical image information, such as cell segmentation and classification (Esteva et al. 2017), as well as physiological and pathological image segmentation and detection (Hu et al. 2018), is becoming increasingly significant. However, traditional CNNs are primarily designed for Euclidean space data (Figure 1.1 Left), characterized by translational invariance, such as images, and are less effective for non-Euclidean data, like graph data.

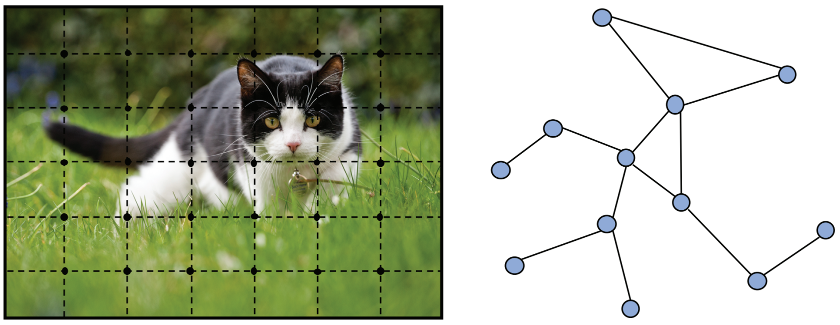


Figure 1.1: Image in Euclidean space (Left). Graph in non-Euclidean space (Right). Image taken from (Zhou et al. 2020)

Graph data naturally represent the complex relationships and topological structures between entities, which are crucial for understanding drug development, predicting protein functions through PPI (protein-protein interactions), molecular structures, and so on. It is against this backdrop that the development of Graph Neural Networks (GNNs) (Scarselli et al. 2008) offers a new perspective for analyzing biomedical data. By operating directly on graph data, GNNs can effectively capture complex data structures and relationships in non-Euclidean space (Figure 1.1 Right), providing strong support for multiple applications in the field of biomedicine.

Although the literature discussing various applications of GNNs is extensive, comprehensive summaries focusing on their applications and challenges in biomedical data analysis are relatively rare. This project aims to bridge this gap by providing a detailed review and summary of GNNs and their applications in biomedical data analysis, highlighting their strengths and potential in handling these specific types of data.

This thesis is structured as follows. In Chapter 2, we will introduce the fundamentals of Graph Neural Networks (GNNs), presenting the core GNN models such as Graph Convolutional Networks (GCN), Graph Attention Networks (GAT), and GraphSAGE, along with an exploration of GNN explainability. In Chapter 3, we will discuss the specific applications of GNNs within the biomedical domain. Following this, in Chapter 4 we will showcase two detailed case studies, including the data involved, analysis performed, and results obtained. And finally, we will conclude the survey in Chapter 5.

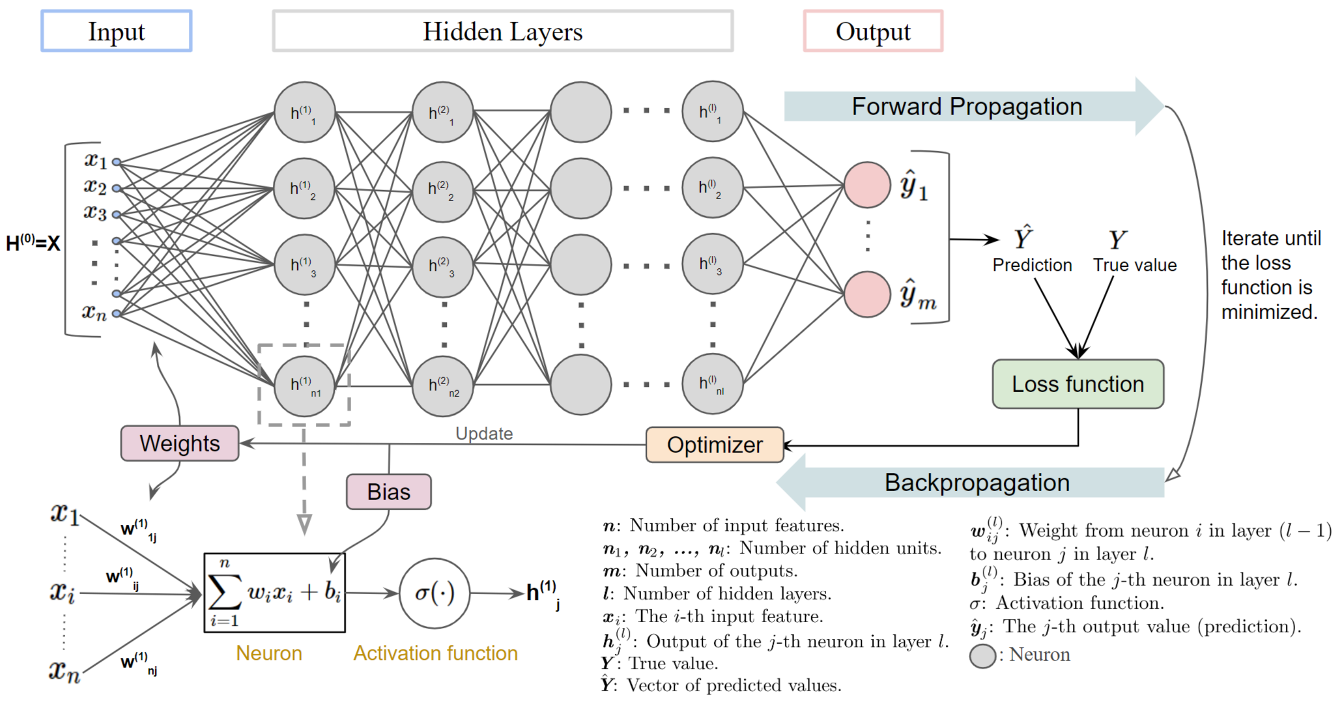
# 2 Foundations of Graph Neural Networks

This chapter provides a detailed introduction to the basic concepts of deep learning and the fundamental principles and common models of Graph Neural Networks (GNNs). Additionally, it delves into the explainability of GNNs, setting the stage for a comprehensive understanding of these advanced techniques.

## 2.1 Basic Concepts of Deep Learning

### 2.1.1 Basic Structure of Neural Networks

Neural networks are computational models that mimic the structure and function of the human brain. They consist of numerous simple processing units, called neurons, organized into layers and interconnected to form a complex network structure for data processing and analysis, as shown in the Figure 2.1.



Input
Hidden Layers
Output
Forward Propagation
Backpropagation
$H^{(0)}=X$
$x_1$ $x_2$ $x_3$ $x_n$
$\hat{y}_1$ $\hat{y}_m$
$\hat{Y}$ Prediction
$Y$ True value
Loss function
Iterate until the loss function is minimized.
Optimizer
Update
Weights
Bias
$\sum_{i=1}^{n} w_i x_i + b_i$
$\sigma(\cdot)$
$h^{(1)}_j$
Neuron
Activation function
$n$: Number of input features.
$n_1$, $n_2$, ..., $n_l$: Number of hidden units.
$m$: Number of outputs.
$l$: Number of hidden layers.
$x_i$: The $i$-th input feature.
$h_j^{(l)}$: Output of the $j$-th neuron in layer $l$.
$Y$: True value.
$\hat{Y}$: Vector of predicted values.
$w_{ij}^{(l)}$: Weight from neuron $i$ in layer $(l-1)$ to neuron $j$ in layer $l$.
$b_j^{(l)}$: Bias of the $j$-th neuron in layer $l$.
$\sigma$: Activation function.
$\hat{y}_j$: The $j$-th output value (prediction).
: Neuron

Figure 2.1: Structure and learning process of a neural network

**Neuron:**

A neuron is the basic unit of a neural network, similar to a biological neuron. Each neuron receives multiple inputs, performs computations, and generates an output. The basic structure of a neuron can be described as follows:

- **Inputs**: Each neuron receives multiple inputs, typically represented as a vector $\mathbf{x} = [x_1, x_2, \dots, x_n]$, with $\mathbf{x} \in \mathbb{R}^n$.

- **Weights**: Each input signal is associated with a weight $\mathbf{w} = [w_1, w_2, \dots, w_n]$, with $\mathbf{w} \in \mathbb{R}^n$, which adjusts the importance of the input signal.
- **Weighted Sum**: The neuron computes the weighted sum of the input signals $z = \sum_{i=1}^{n} w_i x_i + b$, where $b$ is the bias term.
- **Activation Function**: The weighted sum is passed through an activation function $\sigma(z)$ to produce the output of the neuron.

**Layer:**

A neural network is composed of multiple layers of neurons, with each layer containing several neurons. Layers are categorized based on their position and function:

- **Input Layer**: The layer that receives the raw data input. The number of neurons in the input layer equals the number of input features.
- **Hidden Layer**: Layers located between the input and output layers and the number and size of hidden layers can be chosen freely.
- **Output Layer**: The layer that produces the final prediction results. The number of neurons in the output layer depends on the specific task.

**Neural Network Training:**

Neural networks consist of layers and neurons interconnected to form a complete architecture, where each neuron in one layer is typically connected to several neurons in the next layer. Training a neural network involves adjusting the weights and biases to minimize prediction errors, through a process that involves several key concepts:

- **Forward Propagation**: Data flows from the input layer through each hidden layer to the output layer, with each layer computing the outputs of its neurons.
- **Loss Function**: Computes the error between the predicted values and the true values.
- **Cost function**: Computes the average of the loss functions over all training samples.
- **Backpropagation**: Based on the error, gradients are computed for each weight and bias, updating the parameters to minimize the cost function.
- **Epoch**: An epoch involves the complete propagation of the entire dataset through the neural network in both forward and backward directions. Since the dataset is typically too large to be processed in one go, it is divided into multiple smaller subsets called **Batches**.
- **Batch Size**: This term refers to the number of training examples included in a single batch.
- **Iterations**: Iterations are the total number of batches required to complete one full epoch of training. For example, suppose we have a dataset with 800 samples and we set the batch size to 100. To complete one epoch (processing all samples once), it would require

8 iterations.

**Input and Output:**

- **Input**: The input to a neural network is typically a multidimensional vector representing the features of the raw data. For example, the input for image data can consist of pixel values organized into three matrices — one for each of the Red, Green, and Blue (RGB) channels, while the input for text data can be word vectors.
- **Output**: The output of a neural network consists of one or more values representing the prediction results. The form of the output depends on the specific task. For example, in a classification task, the output can be class probabilities, while in a regression task, the output can be continuous values.

Above is a basic understanding of neural networks. Next, we will provide a detailed introduction to important components such as activation functions and optimization algorithms in the training of neural networks.

### 2.1.2 Activation Functions

Activation functions introduce non-linearity into deep learning models, enabling them to capture complex relationships between inputs and outputs. These functions, denoted as $\sigma$, are essential for the model's ability to extract meaningful features from data. In this section, we will discuss several widely used activation functions (Goodfellow, Bengio, and Courville 2016).

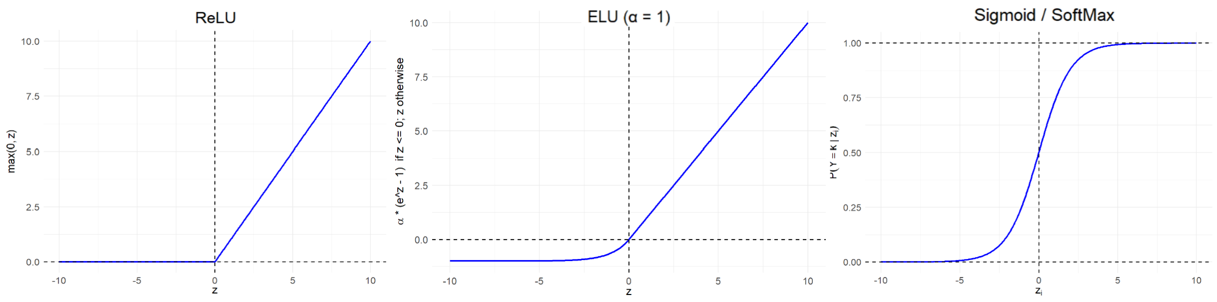


Figure 2.2: Activation Functions: Sigmoid/Softmax and ReLU. This plot was generated using R.

#### 1) Rectified linear unit (ReLU)

$$\sigma(z) = \max(0, z)$$

where $\sigma : \mathbb{R} \to \mathbb{R}$, $z \in \mathbb{R}$ is the input to the neuron. This activation function and its variants show superior performance in many cases and are the most popular activation function in deep learning so far (Figure.2.2). ReLU can also solve the gradient saturation problem and the calculation speed is much faster (Glorot, Bordes, and Bengio 2011).

However, ReLU has its limitations, notably the “dead neuron” problem, where neurons become inactive and cease to output anything other than zero because their inputs are always negative.

**2) Exponential Linear Unit (ELU)**

To addresses this problem, ELU offers a solution to the dead neuron issue associated with ReLU, as shown in Figure 2.2. It introduces a nonzero gradient when the input is negative, which helps keep the neurons active:

$$\mathrm{ELU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases}$$

where $\alpha \in \mathbb{R}$. ELU not only overcomes the problem of dead neurons but also tends to converge faster than ReLU during training by producing outputs with a mean closer to zero, which is beneficial for learning dynamics.

**3) Sigmoid**

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

where $\sigma : \mathbb{R} \to (0, 1)$, $z \in \mathbb{R}$ represents the weighted sum of inputs to the neuron.

A sigmoid function transforms variables to values ranging from 0 to 1 and is commonly used to produce a Bernoulli distribution, as shown in Figure 2.2. Hence, The sigmoid function is widely used in binary classification problems, where an output of 0 or 1 signifies the two distinct classes.

**4) Softmax**

The Softmax function is a generalization of the Sigmoid function for multi-class classification problems. It converts a vector of raw scores (logits) into probabilities. It is defined as:

$$\mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where $\sigma : \mathbb{R}^K \to (0, 1)^K$:

- $z_i \in \mathbb{R}$ is the $i$-th element of the input vector **z**.
- $\mathbf{z} = [z_1, z_2, \ldots, z_K] \in \mathbb{R}^K$ is the input vector containing $K$ elements (logits).
- $K$ is the number of classes.

**Comparison between the Sigmoid and Softmax:**

- Sigmoid Function: Used for binary classification problems, mapping a single scalar input to a probability between 0 and 1.
- Softmax Function: Used for multi-class classification problems, mapping an input vector to a probability distribution over multiple classes, with each probability between 0 and 1, and the sum of all probabilities equal to 1.

The visual representation of the Sigmoid function and the individual probabilities in the Softmax function are similar, as shown in the Figure 2.2.

### 2.1.3 Optimization Objective

The objective of optimization typically comprises both a **cost function** and a **regularization** component. The loss function $L(f(\mathbf{x}|\Theta), \mathbf{y})$ measures the error between the network's output $f(\mathbf{x}|\Theta)$ and the expected result $y$ for a single sample. The cost function $J(\Theta)$ is the average of the loss functions over all training samples. However, effective learning algorithms are characterized not only by their performance on training data but also by their generalization to unseen test data. Regularization, a suite of techniques aimed at minimizing test error and fighting overfitting, enforces penalties on model parameters to deter the model from becoming overly complex. Therefore, we introduce a commonly utilized cost function alongside the regularization term $\Omega(\Theta)$. Typically, the optimization objective is defined as:

$$\text{Optimization Objective} = J(\Theta) + \alpha\Omega(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}_i|\Theta), \mathbf{y}_i) + \alpha\Omega(\Theta)$$

Here,

- $\mathbf{x}_i \in \mathbb{R}^m$ is the input feature vector for the $i$-th sample, where $\mathbf{x}_i = [x_{i1}, x_{i2}, \cdots, x_{im}]$,
- $\mathbf{y}_i$ is the expected result or ground truth label for the $i$-th sample, which can be a single target value or a vector,
- $\Theta \in \mathbb{R}^p$ is the set of all parameters of the network that need to be trained, thus $p$ is the total number of parameters,
- $n$ is the total number of samples in the training dataset,
- $\alpha \in \mathbb{R}$ is a hyperparameter that controls the trade-off between the cost function and the regularization term.

#### 2.1.3.1 Loss function (Single Sample)

**1) Mean Squared Error (MSE)**

The loss function utilized is defined as follows:

$$L(f(\mathbf{x}|\Theta), y) = (y - f(\mathbf{x}|\Theta))^2$$

Here,

- $\mathbf{x} \in \mathbb{R}^m$ is the input feature vector for a single sample,
- $y \in \mathbb{R}$ is the target value for the input sample,
- $f : \mathbb{R}^m \to \mathbb{R}$ is the model mapping the input feature vector to the predicted value.

This loss function is often used in regression problems. Mean Absolute Error (MAE) is also commonly used in such problems.

**2) Cross-entropy Loss**

*i. Binary Cross-Entropy Loss:*

We usually use binary cross-entropy for a two-class classification problem. Its discrete form is given by:

$$L(p(\mathbf{x}), y) = -[y \log(p(y = 1|\mathbf{x})) + (1 - y)\log(1 - p(y = 1|\mathbf{x}))]$$

Here,

- $\mathbf{x} \in \mathbb{R}^m$ is the input feature vector for a single sample,
- $y \in \{0, 1\}$ is the true label for the input sample,
- $p : \mathbb{R}^m \to [0, 1]$ is the model mapping the input feature vector to a predicted probability,
- $p(y = 1|\mathbf{x})$ is the predicted probability that the sample belongs to class 1.

*ii. Multiclass Cross-Entropy Loss:*

In multi-class problems, each observation is no longer a binary label but a probability distribution representing the likelihood of the observation belonging to each class. The Multiclass Cross-Entropy Loss function is defined as follows:

$$L(p(\mathbf{x}), \mathbf{y}) = -\sum_{k=1}^{K} y_k \log(p(y = k|\mathbf{x}))$$

Here,

- $\mathbf{x} \in \mathbb{R}^m$ is the input feature vector for a single sample,

- $\mathbf{y} \in \mathbb{R}^K$ is a one-hot encoded vector representing the true category of the sample, where $\mathbf{y} = [y_1, y_2, \ldots, y_K]$ with $y_k$ being 1 if the sample belongs to class $k$, and 0 otherwise,
- $p : \mathbb{R}^m \to [0, 1]^K$ is the model mapping the input feature vector to a predicted probability distribution over $K$ classes,
- $p(y = k|\mathbf{x})$ is the model's predicted probability that the sample belongs to class $k$.

Selecting the right activation and loss functions depends on the type of machine learning problem. The Table 2.1 below summarizes the recommended functions for different problem types (Chollet 2021):

Table 2.1: Summary of Functions for Different Problem Types

| Problem Type | Last Layer Activation | Loss Function |
|---|---|---|
| Binary Classification | Sigmoid | Binary Crossentropy |
| Multi-class, Single-label | Softmax | Categorical Crossentropy |
| Multi-class, Multi-label | Sigmoid | Binary Crossentropy |
| Regression to Arbitrary Values | None | MSE |
| Regression to Values in [0, 1] | Sigmoid | MSE or Binary Crossentropy |

#### 2.1.3.2 Regularization term

Regularization techniques are a set of best practices that actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation (Chollet 2021). Here, we will introduce the most common regularization techniques found within the optimization objective, namely L1 and L2 regularization.

- *L1 regularization* operates by adding the sum of the absolute values of the model parameters, typically focusing on the weights. This approach encourages the model to learn sparse parameter weights, effectively setting many of them to zero. This sparsity can act as a form of feature selection, helping the model to concentrate on the most significant features in the data. The L1 regularization can be defined as:

$$\Omega(\Theta) = \|\Theta\|_1 = \sum_{i=1}^{p} |w_i|$$

- *L2 regularization* functions by adding the sum of the squares of the weight parameters to the cost function, leading to smoother model weights where each element of the weight vector is as small as possible without becoming zero. L2 is especially effective in addressing multicollinearity in the model. The formula for L2 regularization is:

$$\Omega(\Theta) = \frac{1}{2}\|\Theta\|^2 = \frac{1}{2}\sum_{i=1}^{p} w_i^2$$

### 2.1.4 Optimizer

In deep learning, the solution to many problems is essentially to solve optimization-related problems. It is an algorithm that updates the network weight by training the neural network and minimizing the error. Here are some common optimization methods:

**Stochastic Gradient Descent (SGD)** (LeCun et al. 1998) is the foundational optimization method, which updates the model parameters by calculating the gradient of the cost function for a small batch of the dataset. It introduces randomness by using only a single or a small batch of samples to compute the gradient, which, despite adding noise, helps the model escape local minima.

*Formula*: $\Theta = \Theta - \eta \cdot \nabla_\Theta J(\Theta) = \Theta - \eta \cdot \frac{\partial J}{\partial \Theta}$, where $\Theta$ represents the model parameters (weights and biases), $\eta \in \mathbb{R}^+$ is the learning rate [1], $J$ is the cost function, and $\nabla_\Theta J(\Theta)$ or $\frac{\partial J}{\partial \Theta}$ is the gradient of the cost function with respect to $\Theta$.

**Momentum (SGD-M)** SGD can struggle in regions where the terrain is uneven, causing it to oscillate and make slow progress. By incorporating momentum, the algorithm can move more smoothly and quickly toward the optimum by reducing these oscillations, as shown in Figure 2.3. Momentum helps to speed up SGD by adding a portion of the previous update vector to the current one, thus smoothing out the path and reducing oscillations.

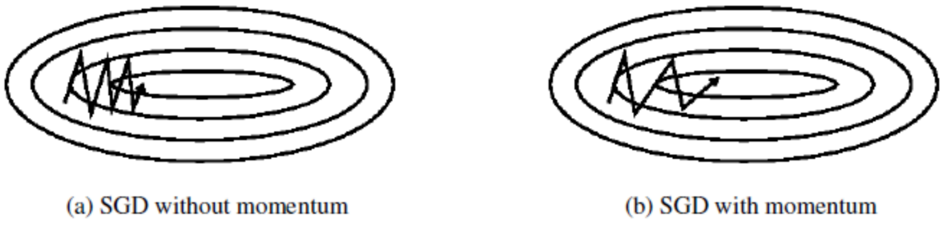


Figure 2.3: SGD with momentum

*Formula*: $v_t = \gamma \mathbf{v}^{(t-1)} + \eta \nabla_\Theta J(\Theta)$, then update $\Theta = \Theta - \eta \mathbf{v}^{(t)}$, Where $\mathbf{v}^{(t)}$ is the current velocity vector, and $\gamma \in [0, 1]$ is the momentum factor.

Building on SGD, **Adagrad** updates parameters according to the accumulation of squared gra-

[1]The learning rate is a crucial hyperparameter in machine learning algorithms that controls the step size of parameter updates during training. Proper selection of the learning rate affects the convergence speed and the model's ability to reach the optimal solution.

dients, which can converge rapidly with convex functions but performs worse in certain models (Goodfellow, Bengio, and Courville 2016). To address some of Adagrad's limitations, **RMSprop** was developed, which has emerged as a highly effective and widely adopted approach for parameter optimization. In December 2014, the **Adam** optimizer was proposed by scholars Kingma and Lei Ba (Kingma and Ba 2014), combining the advantages of both Adagrad and RMSprop optimization algorithms. Adam adapts the learning rate for each parameter by computing individual adaptive learning rates, making it particularly powerful and popular for a wide range of machine learning models.

## 2.2 Graph Theory

### 2.2.1 Basic Concepts

A graph is often denoted by $G = (V, E)$, where $V$ is the set of **nodes** (or vertices) and $E$ is the set of **edges** (or links). An edge ($e = u, v$) has two **endpoints** $u$ and $v$, which are said to be joined by $e$. In this case, $u$ is called a **neighbor** of $v$, or in other words, these two vertices are **adjacent**. Note that an edge can either be **directed** or **undirected**. The **degree** of vertex $v$, denoted by $d(v)$, is the number of edges connected with $v$ (Liu and Zhou 2022) (Figure.2.4).

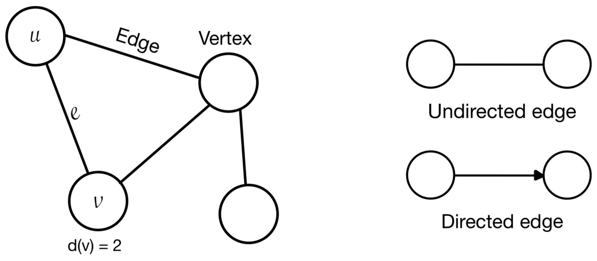


Figure 2.4: Graph theory

### 2.2.2 Algebra Representations of Graphs

- **Adjacency matrix:** for a simple graph $G = (V, E)$ with $n$-vertices, it can be described by an adjacency matrix $A \in \mathbb{R}^{n \times n}$, where

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

  It is obvious that such matrix is a symmetric matrix in the case where $G$ represents an undirected graph.

- **Degree matrix:** for a graph $G = (V, E)$ with $n$ vertices, its degree matrix $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix, where

$$D_{ii} = d(v_i).$$

## 2.3 GNN Models and Layers

When designing a GNN model, it's essential to follow a systematic process to ensure that the model is well-suited for the specific task at hand. The general design pipeline for a GNN model involves several crucial steps, from data preparation to model evaluation. The Figure 2.5 illustrates the overall design pipeline for a GNN model, providing a comprehensive overview of the key stages and components involved in building and deploying a GNN.

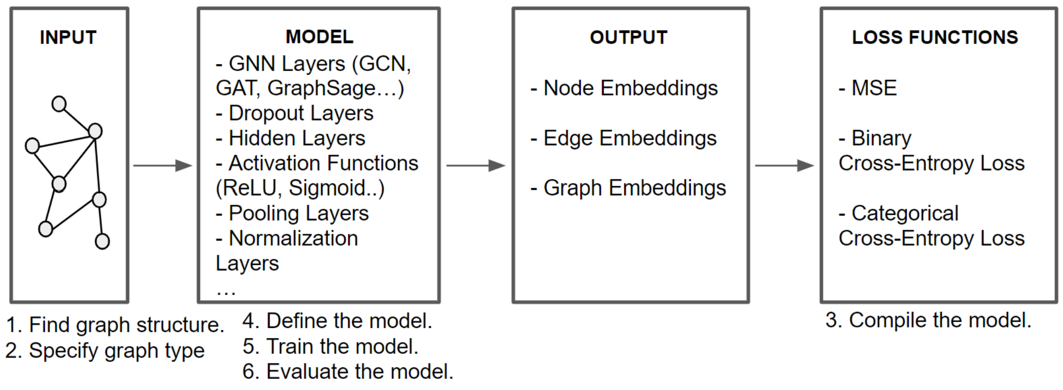


Figure 2.5: The general design pipeline for a GNN model.

1. **Find graph structure**: Determine the basic structure of the graph, including nodes, edges, and other relevant components.
2. **Specify graph type**: Identify the type of task (Figure 2.6) and problem (Table 2.1).
3. **Compile the model**: Compile the model by setting the loss function, optimizer, and metrics.
4. **Define the model**: Define the model by selecting layers, activation functions, and other necessary components.
5. **Train the model**: Optimize model parameters using training data. Compute the cost function through forward propagation and update the model parameters through backward propagation to minimize the cost function.
6. **Evaluate the model**: Evaluate the model's performance using a test set.

Previously, we introduced some basic concepts related to model compilation and training. Next, we will delve into the definition of the model, detailing its principles and layers.

### 2.3.1 Principles of GNN

The core concept behind GNNs is iterative information aggregation from neighboring nodes. In each iteration, a node updates its representation by aggregating information from its neighbors, capturing the local neighborhood's structural features. After multiple iterations, the representation of each node encodes information about the topology of the entire graph. Here are some key concepts and common formulas associated with GNNs:

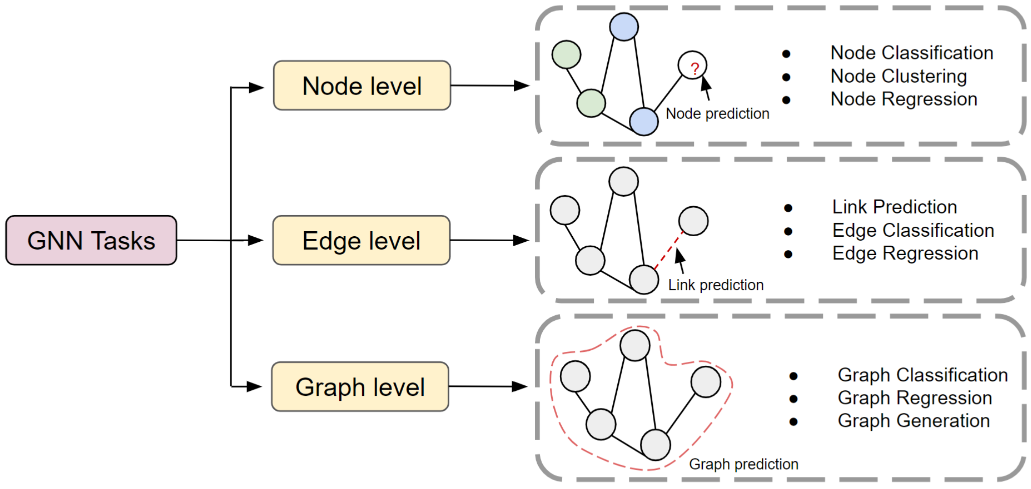


Figure 2.6: Graph learning tasks

**Message Passing:**

GNN typically operate on a message-passing mechanism, where nodes send and receive information to and from their neighbors. The information update for each node across iterations can be broken down into two steps: Aggregation and Update (Figure 2.7).

- **Aggregation:** each node aggregates features from its neighbors. The aggregation function could be sum, mean, max, etc., aimed at capturing the neighbors' information.
- **Update:** nodes combine their own features with the aggregated neighbor information to update their feature representation.

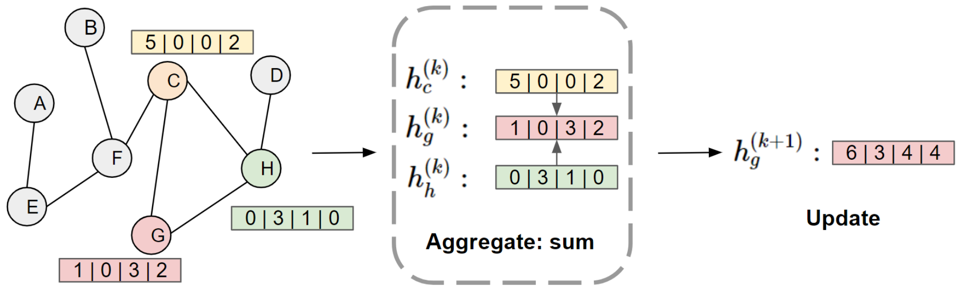

A
B
C
D
E
F
G
H
5 | 0 | 0 | 2
0 | 3 | 1 | 0
1 | 0 | 3 | 2
$h_c^{(k)}$ : 5 | 0 | 0 | 2
$h_g^{(k)}$ : 1 | 0 | 3 | 2
$h_h^{(k)}$ : 0 | 3 | 1 | 0
Aggregate: sum
$h_g^{(k+1)}$ : 6 | 3 | 4 | 4
Update

Figure 2.7: Message-passing mechanism

The feature update for a node can be represented by the following formula:

$$\mathbf{h}_v^{(k+1)} = f(\mathbf{h}_v^{(k)}, \text{Aggregate}(\{\mathbf{h}_u^{(k)} : u \in \mathcal{N}(v)\}))$$

Here,

- $\mathbf{h}_v^{(k+1)}$ is the feature representation of node $v$ after the $(k+1)$-th layer,
- $\mathbf{h}_u^{(k)}$ represents the features of neighbor nodes $u$ in the $k$-th layer,
- $\mathcal{N}(v)$ denotes the set of neighbor nodes of $v$, and $f$ is the update function.

After the initial layer (k=1), each node embedding explicitly incorporates information from its immediate 1-hop vicinity, accessible through a path of length 1 in the graph. Following the second layer (k=2), each node embedding contains information from its 2-hop neighborhood. In general, after k layers, the embedding of each node embodies information from its k-hop surroundings (Khemani et al. 2024).

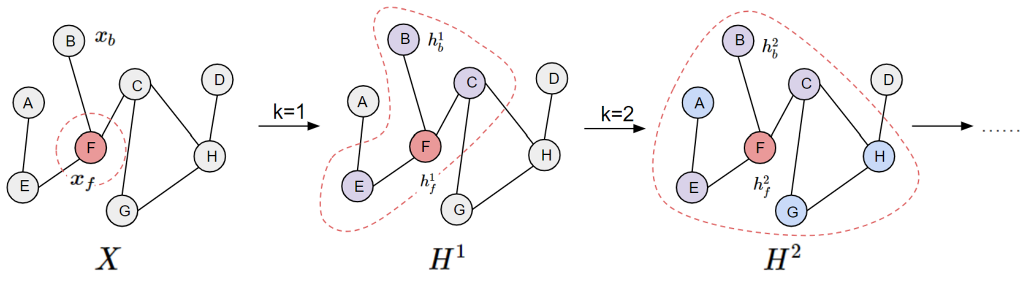


Figure 2.8: GNN iterations

It is obvious from the Figure 2.8 that during the iterative update process, the connection relationships remain unchanged, in other words, the adjacency matrix remains the same. Moreover, it becomes evident that nodes can capture most of the node features with just a few layers of a GNN. Hence, it is often unnecessary to employ many layers when using GNNs.

### 2.3.2 Graph Convolutional Network (GCN)

GCNs are a type of neural network designed to operate on graph-structured data. The classical GCN model, introduced by Kipf and Welling (2016), is one of the most well-known and widely used GCN architectures. Its update rule is as follows:

$$\mathbf{H}^{(k+1)} = \sigma\left(\widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(k)}\mathbf{W}^{(k)}\right)$$

Here,

- $\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ is the adjacency matrix of the undirected graph with added self-loops.
- $\mathbf{I}_N$ is the identity matrix.
- $\widetilde{\mathbf{D}}$ is the degree matrix of $\widetilde{\mathbf{A}}$.
- $\mathbf{W}^{(k)}$ is a layer-specific trainable weight matrix.
- $\sigma(\cdot)$ denotes an activation function, such as the ReLU(.) = max(0, .).
- $\mathbf{H}^{(k)} \in \mathbb{R}^{N\times D}$ is the matrix of activations in the $k$-th layer; $\mathbf{H}^{(0)} = \mathbf{X}$.

Although GCN have demonstrated significant potential, their limitations in handling specific tasks highlight the need for further advancements. A notable deficiency of GCNs is their inability to effectively perform inductive learning; they struggle with unseen nodes during testing. Furthermore, GCNs impose a uniform convolutional transformation across all neighbors, which restricts their ability to allocate differential weights to neighbors based on their importance or contextual relevance. This uniform approach limits the generalizability of a GCN model trained on one graph to effectively operate on another, distinctly structured graph, thus categorizing GCNs as primarily semi-supervised methods.

### 2.3.3 Graph Attention Networks (GAT)

To address the issue of uniform edge weights in GCN, Graph Attention Networks (GAT) (Velickovic et al. 2017) utilize masked self-attentional layers, which fundamentally innovate by enabling the assignment of different weights to various neighbors without the need for matrix operations or prior knowledge of the graph structure.

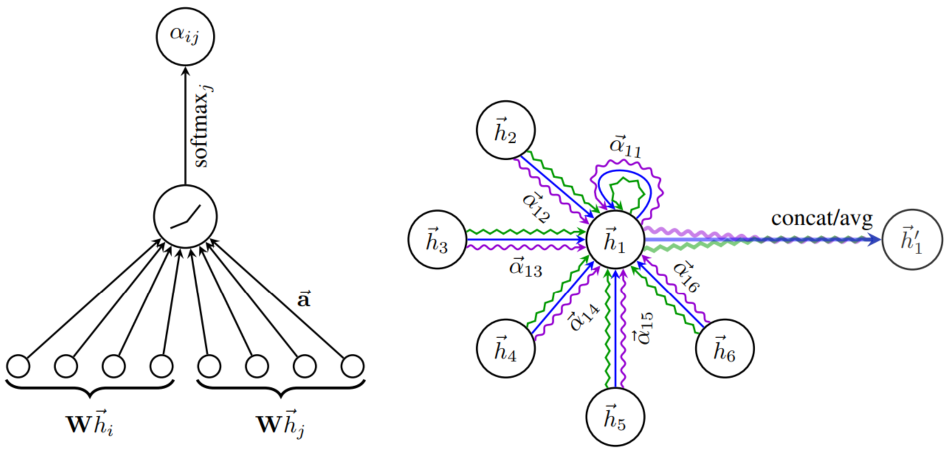


Figure 2.9: Left: The attention mechanism. Right: An illustration of multi-head attention (with K = 3 heads) by node 1 on its neighborhood.

In the GAT model, input the set $\mathbf{x} = \{x_1, x_2, \dots, x_N\}$ of node features, where each $x_i \in \mathbb{R}^F$ and $N$ is the number of nodes, $F$ is the number of features in each node. The layer produces a new set of node features, $\mathbf{h} = \{h_1, h_2, \dots, h_N\}$, where $h_i \in \mathbb{R}^{F'}$ as its output. The *attention coefficient* of the edge $(i, j)$ represented by $e_{ij}$, according to the equation:

$$e_{ij} = a(\mathbf{W}\mathbf{x}_i, \mathbf{W}\mathbf{x}_j)$$

where $\alpha$ represents a shared attentional mechanism, and $W \in \mathbb{R}^{F' \times F}$ is an initial shared linear transformation weight matrix. To ensure comparability of coefficients across nodes, a softmax function is used for normalization:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})}$$

Then, the coefficients computed by the attention mechanism (Figure 2.9 Left) may then be expressed as:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{x}_i \| \mathbf{W}\mathbf{x}_j]))}{\sum_{k \in N_i} \exp(\text{LeakyReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{x}_i \| \mathbf{W}\mathbf{x}_k]))}$$

where $\top$ indicates transposition, and $\|$ is the concatenation operation. A shared linear transformation weight matrix is utilized. The output features of each node are then calculated by the following equation:

$$\mathbf{h}_i = \sigma\left(\sum_{j \in N_i} \alpha_{ij} \mathbf{W}\mathbf{x}_j\right)$$

Multi-head attention diversifies the attention layer by implementing K independent attention mechanisms (Figure 2.9 Right), enhancing the stability of the self-attention learning process, with the final representation formulated as follows:

$$\mathbf{h}_i = \Big\|_{k=1}^{K} \sigma\left(\sum_{j \in N_i} \alpha_{ij}^k \mathbf{W}^k \mathbf{x}_j\right)$$

where $\|$ represents concatenation.

### 2.3.4 GraphSAGE

While both GCN and GAT are tailored for full-graph computation, which poses computational challenges when applied to large-scale graphs, GraphSAGE, proposed by Hamilton, Ying, and Leskovec (2017), offers a promising solution. GraphSAGE not only addresses the computational complexity but also effectively manages inductive learning tasks. The process comprises two primary steps: “sampling” and “aggregation”. Subsequently, the aggregated information, combined with the original node features, undergoes non-linear transformations, such as the ReLU activation function, to update the node representations.

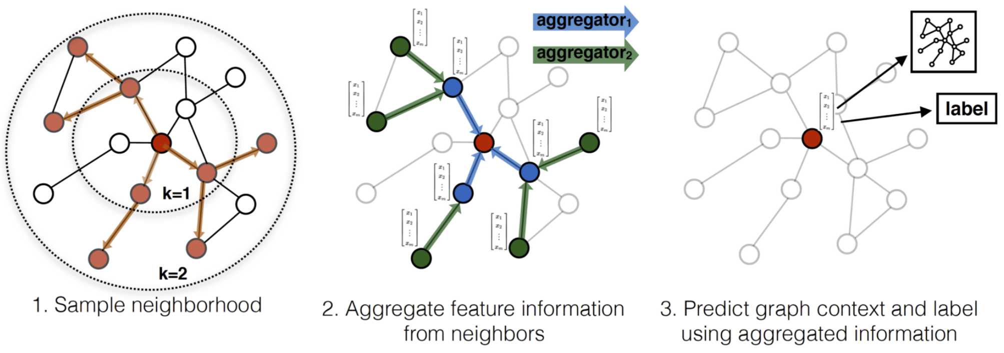

k=1
k=2
1. Sample neighborhood
aggregator1
aggregator2
2. Aggregate feature information from neighbors
label
3. Predict graph context and label using aggregated information

Figure 2.10: Visual illustration of the GraphSAGE sample and aggregate approach.

**1) Sampling:** GraphSAGE adopts a sampling strategy whereby, instead of utilizing all neighbors, a fixed number of neighboring nodes, denoted as $K$, are randomly selected from the neighborhood of each target node, as illustrated in Figure 2.10.

**2) Aggregation:** For each node, GraphSAGE defines an aggregation function to aggregate the information of neighbor nodes. This aggregation function can involve some operations, such as averaging, pooling (e.g., max pooling), etc. Below are some common types of Aggregators:

- Mean aggregator:

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \mathrm{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in N(v)\}))$$

- Max-Pooling Aggregator:

$$\mathrm{AGGREGATE}_k^{pool} = \max\left(\left\{\sigma\left(\mathbf{W}_{pool}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall i \in N(v)\right\}\right)$$

- LSTM Aggregator: Utilizes Long Short-Term Memory (LSTM) networks (Hochreiter and Schmidhuber 1997) to capture temporal dependencies in the neighbor node features.

**3) Update:** A general GraphSAGE update formula is:

$$\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \left(\mathbf{h}_v^{k-1} \| \mathrm{AGG}_k\left(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}\right)\right)\right)$$

Here,

- $\mathbf{h}_v^k$ is the updated feature representation of node $v$ at layer $k$.
- $\mathbf{h}_v^{k-1}$ is the feature representation of node $v$ from the previous layer $k-1$.
- $\mathbf{h}_u^{k-1}$ are the feature representations of the neighbors $u$ of node $v$ from layer $k-1$.
- $\mathcal{N}(v)$ is the set of neighbors for node $v$.
- $\mathrm{AGG}_k$ is the differentiable aggregator functions at layer $k$.
- $\mathbf{W}^k$ is the trainable weight matrix for layer $k$.
- $\sigma$ is a non-linear activation function.
- $\|$ represents the concatenation of the node's previous representation with the aggregated neighbors' features.

### 2.3.5 GNN Pooling

Pooling in GNNs is essential for summarizing node information and creating compact graph representations, especially for tasks like graph classification. By aggregating features from all nodes, pooling helps reduce the dimensionality of the graph data, efficiently combining node features into a cohesive graph-level representation. This process enables the creation of hierarchical representations, facilitating a multi-scale understanding of the graph (as illustrated in

Figure 4.2).

Common Pooling Methods:

- **Global Average Pooling**: $\mathbf{h}_{\text{graph}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{h}_i$
- **Global Max Pooling**: $\mathbf{h}_{\text{graph}} = \max_{i=1}^{N} \mathbf{h}_i$
- **Global Sum Pooling**: $\mathbf{h}_{\text{graph}} = \sum_{i=1}^{N} \mathbf{h}_i$

where $N$ is the total number of nodes and $\mathbf{h}_i$ is the feature vector of the $i$-th node.

### 2.3.6 Dropout Layer

Dropout is a regularization technique used to prevent overfitting in neural networks. It works by randomly deactivating a fraction of neurons during training, forcing the network to learn more robust features. The dropout rate, usually between 0.2 and 0.5, determines the fraction of neurons to be deactivated (Srivastava et al. 2014).

## 2.4 Explainable GNN

Machine learning (ML) algorithms employed in AI can be classified as either *white-box* or *black-box* (Vilone and Longo 2021). White-box models are easily understood and explained due to their simple structures and high transparency, such as decision trees and linear regression. However, not all models possess such clarity. Some black-box models, like deep neural networks, despite providing higher accuracy across various domains, feature complex hierarchical structures and extensive non-linear operations that even experts find challenging to comprehend. The transparency and explainability of models become particularly crucial when applied in high-risk areas such as biomedicine.

As mentioned in the LIME paper (Ribeiro, Singh, and Guestrin 2016), transparent and explainable decision-making processes increase trust—you are likely to trust your doctor because they can explain their diagnosis based on your symptoms. Similarly, if predictions made by deep learning models can be explained or validated in a way that humans can understand, these predictions appear more reliable and trustworthy. To address these challenges, **Explainable Artificial Intelligence (XAI)** has been introduced. This initiative involves developing a range of machine learning techniques that help human users understand, trust properly, and effectively manage the next generation of AI systems (Arrieta et al. 2020). Explainability is achieved through two primary methods: (i) intrinsic model—designing the model with a simple structure, and (ii) post-hoc approaches—applying explanations to understand the model's decision-making logic (A. and R. 2023).

### Intrinsic model

A model is considered intrinsically explainable if it is designed to be interpretable from the outset (Puthanveettil Madathil et al. 2024). These models are also regarded as white-box models, such as decision trees and linear regression.

### Post-hoc approaches

As the name indicates, post-hoc XAI tools are designed to address the opacity of black-box machine learning models after they have made predictions. Common explanatory methods are divided into local and global types; for instance, LIME is local, while SHAP (Lundberg and Lee 2017) provides both local and global explanations. These traditional explanatory methods are particularly effective in the context of CNNs. As shown in Figure 2.11, the example illustrates the LIME explanations for the top three predicted class labels for an input image (a). It is evident from the visualization how specific parts of the image correlate with each predicted class label. For example, the guitar neck significantly influences the prediction of "electric guitar" (b).

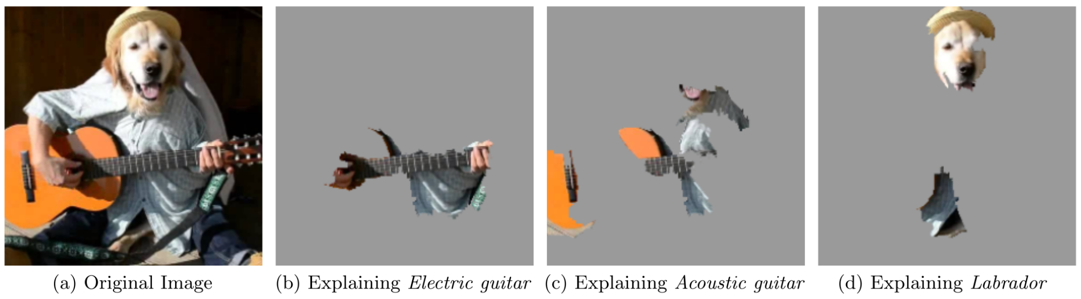

(a) Original Image
(b) Explaining *Electric guitar*
(c) Explaining *Acoustic guitar*
(d) Explaining *Labrador*

Figure 2.11: Explaining an image classification prediction made by Google's Inception neural network. The top 3 classes predicted are "Electric Guitar" (p = 0.32), "Acoustic guitar" (p = 0.24) and "Labrador" (p = 0.21). Image taken from (Lundberg and Lee 2017).

However, when it comes to GNNs, things become a bit more complicated. Unlike the highly structured grids handled by CNNs, the irregular nature of graph structures presents numerous challenges. For instance, while we can easily interpret the explanations of the aforementioned CNN models, providing similar node-level explanations for a graph is not as straightforward to visualize and explain.

For GNNs, current explainable methods offer different perspectives to understand the model from various aspects of the graph, such as which input edges are more important, which input nodes carry more weight, which node features are more significant, and what types of graph patterns maximize the predictive ability for a certain class. To comprehend these approaches, Yuan et al. (2022) have proposed a classification of different GNN explanation techniques. Based on the type of explanation, these techniques are primarily divided into two main categories: **instance-level** methods and **model-level** methods (Figure 2.12).

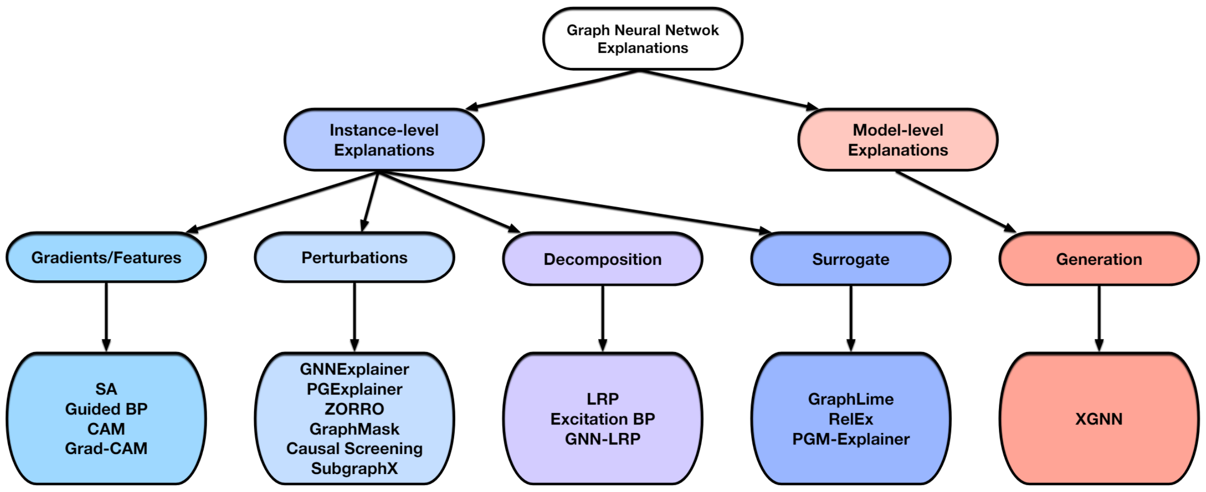


Figure 2.12: An overview of the proposed taxonomy.

Instance-level methods provide explanations specific to each input graph by identifying important features. These methods are divided into four categories based on how they generate importance scores: gradients/features-based, perturbation-based, decomposition, and surrogate methods.

- **Gradients/features-based methods** use gradients or feature values to assess feature importance.
- **Perturbation-based methods** evaluate how changes in input affect predictions to determine feature importance.
- **Decomposition methods** break down prediction scores to the last hidden layer and trace these back to input features.
- **Surrogate methods** sample data from a given example's neighbors and use simple models like decision trees to explain predictions.

Model-level methods explain graph neural networks globally without focusing on specific inputs. They provide general insights into how models behave overall, such as through XGNN (Yuan et al. 2020), which generates and uses graph patterns to explain model predictions.

Both approaches offer different insights: instance-level methods provide detailed, specific explanations, while model-level methods offer broader, more general understandings. Validating and interpreting these models requires human supervision, especially for instance-level methods, which need more detailed examination. Model-level methods, though less demanding in supervision, might produce less interpretable explanations for humans. Together, these methods enhance the overall understanding of graph models.

Next, we will focus on two common XAI methods: SA and GNNExplainer.

**SA** (Baldassarre and Azizpour 2019) uses the square of the gradient as an importance score for different input features, which can be directly calculated through backpropagation. This process is similar to network training but targets input features instead of model parameters, allowing for a direct measure of how changes in inputs affect the outputs. These inputs could be nodes, edges, or features of a graph. It assumes that a higher absolute gradient value indicates a more important input feature. Although simple and effective, this approach has limitations. Firstly, SA only reflects the sensitivity between inputs and outputs, not their true importance. Additionally, in the model's saturation regions where input changes have minimal impact on the output, gradients may fail to accurately represent input contributions.

**GNNExplainer** (Ying et al. 2019) is a perturbation-based method that provides interpretable explanations for predictions made by any GNN-based model on graph-based machine learning tasks. It employs *soft masks* to learn the importance of edges and node features. These masks are optimized by maximizing the mutual information between the predictions of the original graph and a modified graph, where the modifications are guided by the masks. This process allows for the identification of crucial input features, as different masks are generated to explore various aspects of input influence. The generated masks are applied to the input graph through *element-wise multiplication*, producing a new graph that highlights important input features. This new graph is then fed into the trained GNN to evaluate the effectiveness of the masks and refine the mask generation algorithm.

While both SA and GNNExplainer have been instrumental in providing insights into the inner workings of GNNs, they are not without their limitations.

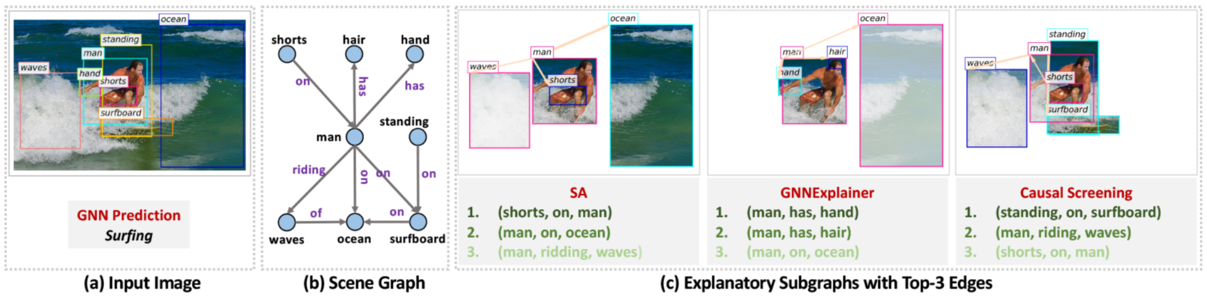


Figure 2.13: An example of explaining the classification of a scene graph. Image taken from (X. Wang et al. 2020).

- **Confounding Associations**: Consider the example illustrated in Figure 2.13, where the scene type of a scene graph is predicted as Surfing. Both SA and GNNExplainer employ gradients and masks, respectively, to attribute importance to various parts of the graph. However, these methods often highlight edges with large gradients or significant masks, such as (shorts, on, man) or (man, has, hand). These associations, while highly correlated with the prediction, do not necessarily cause it. In other words, the model identifies correlations that do not truly explain the underlying causation. For example, “shorts” and “man” may be correlated with surfing scenes but do not directly cause the prediction. Instead, the actual causal relationship might involve edges like (standing, on, surfboard), which are more directly indicative of surfing.

- **Redundancy**: Another significant issue is redundancy. In GNNs, the interconnected nature of the graph means that the gradient-like signals of edges are influenced and even scaled by their connected neighbors. This interconnectedness can result in redundant edges being highlighted in the top explanations. For instance, edges such as (man, on, ocean) and (man, riding, waves) might both appear as important due to their high gradient signals. However, these edges essentially convey the same information and contribute to redundancy in the explanation. This redundancy obscures unique and potentially more informative edges, leading to a less clear understanding of the model’s decision-making process.

To address these shortcomings, various new methods have been proposed by researchers. For example, X. Wang et al. (2020) developed Causal Screening, which focuses on identifying causal relationships within the graph. This approach provides more accurate and reliable explanations for the predictions made by GNNs, as shown in Figure 2.13 (c).

## 2.5 GNN Frameworks

In this section, we will briefly introduce several common frameworks for implementing Graph Neural Networks (GNNs). These frameworks provide a rich set of tools and libraries that support the construction and training of various GNN models, helping researchers and engineers efficiently analyze and apply graph data. Each framework has its strengths in functionality, performance, and support, and the choice of the appropriate framework depends on specific application requirements and technical preferences.

Table 2.2: Overview of Common GNN Frameworks

| **Framework** | **Backend** | **Key Features** | **URL** |
|---|---|---|---|
| PyTorch Geometric (PyG) | PyTorch | Efficient graph data processing and manipulation, multiple GNN layers and operations, strong community support and documentation | PyG |

| **Framework** | **Backend** | **Key Features** | **URL** |
|---|---|---|---|
| Deep Graph Library (DGL) | PyTorch, TensorFlow, MXNet | Supports multiple deep learning frameworks, efficient graph data processing and modeling capabilities, extensive GNN models and examples | DGL |
| TensorFlow GNN (TF-GNN) | TensorFlow | Integrated with TensorFlow's powerful capabilities, provides flexible API interfaces, supports various GNN models and operations | TF-GNN |
| Graph Nets | TensorFlow, Sonnet | Provides a general GNN framework, supports multiple graph operations and models, easy to integrate and extend | Graph Nets |
| StellarGraph | Keras, TensorFlow | Based on Keras and TensorFlow, offers various pre-built GNN models and examples, focuses on node classification, link prediction, and graph classification tasks | StellarGraph |
| Spektral | Keras, TensorFlow | Provides multiple graph neural network layers and tools, suitable for quickly building and training GNN models | Spektral |

Through these frameworks, researchers can choose the appropriate tools to build and train GNN models according to their needs, achieving efficient graph data analysis and applications. The advantages and features of these frameworks make them highly flexible and effective in handling complex graph-structured data.

# 3 Application Domains of GNN in Biomedicine

Biomedical data are typically characterized by high complexity and interconnectivity. Traditional machine learning methods are often designed for data with regular grid structures, such as two-dimensional medical images and one-dimensional mRNA sequences, which makes it challenging for them to fully account for the diverse structural characteristics and rich interactions present in biomedical data. The advent of GNNs provides a new solution to this problem. GNNs can effectively extract and utilize the information embedded in these biomedical networks, thereby advancing biomedical research.

In this chapter, we will explore the fundamental principles and the diverse applications of GNNs in the field of biomedical sciences. To organize our discussion, we categorize the applications into several key areas (Zhang et al. 2021): **Drug Discovery and Development, Medical Diagnosis and Analysis, and Disease Association Prediction**. Each category represents a critical domain where GNNs are making significant contributions by leveraging their ability to model complex, interconnected data.

## Principles of GNN Applications in Biomedicine

Biomedical data with graph structures can be modeled in two ways: **structure-based modeling (molecule-based modeling)** and **biological network-based modeling** (Zhang et al. 2021).

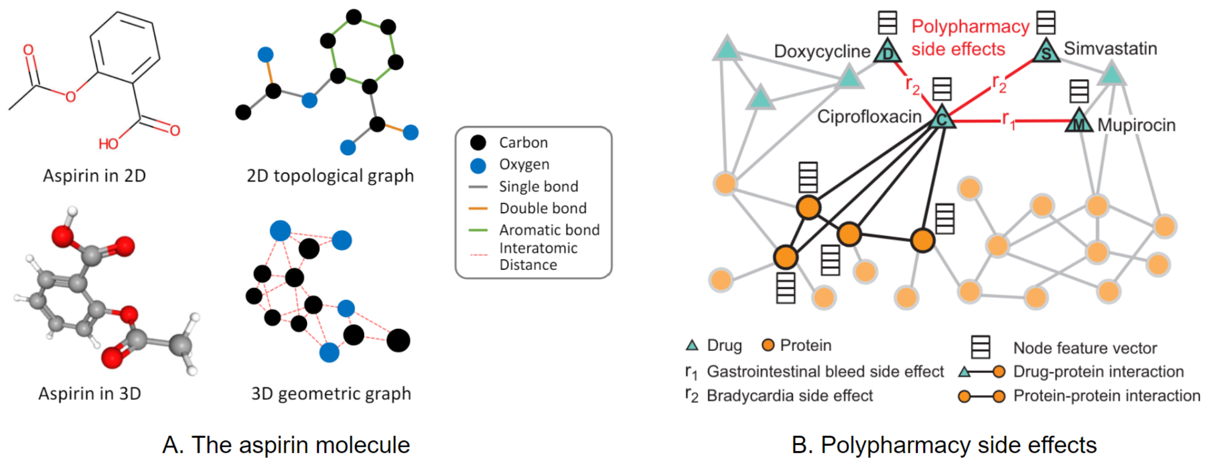


Figure 3.1: (A) Molecular structure-based modeling (Y. Wang, Li, and Barati Farimani 2023). (B) Biological network-based modeling (Zitnik, Agrawal, and Leskovec 2018).

In molecule-based modeling, atoms or functional chemical substructures are used as nodes, and bonds are used as edges to construct a molecular graph, as shown in Figure 3.1 (A). Molecule-based GNNs have a wide range of applications in the biomedical field, such as molecular property prediction, compound classification, and designing new molecules in drug development.

In biological network-based modeling, various biological entities are used as nodes, such as proteins, diseases, drugs, and RNA, as shown in Figure 3.1 (B). The edges between nodes represent known associations between these entities, thereby generating a relational network. This type of network can be used to enhance the analysis and research in bioinformatics.

## 3.1 Drug Discovery and Development

The process of drug development and discovery is intricate and expensive, involving stages from fundamental scientific research to new drug discovery, clinical trials, and market introduction, as illustrated in Figure 3.2. The primary aim of this process is to identify effective therapeutic agents while ensuring their safety and efficacy. The integration of Graph Neural Networks (GNNs) into this process has brought revolutionary improvements, especially in the areas of data analysis and pattern recognition. This section will discuss the critical roles that GNNs can play at various stages of drug discovery and development, including target identification, drug screening and optimization, and clinical trials.

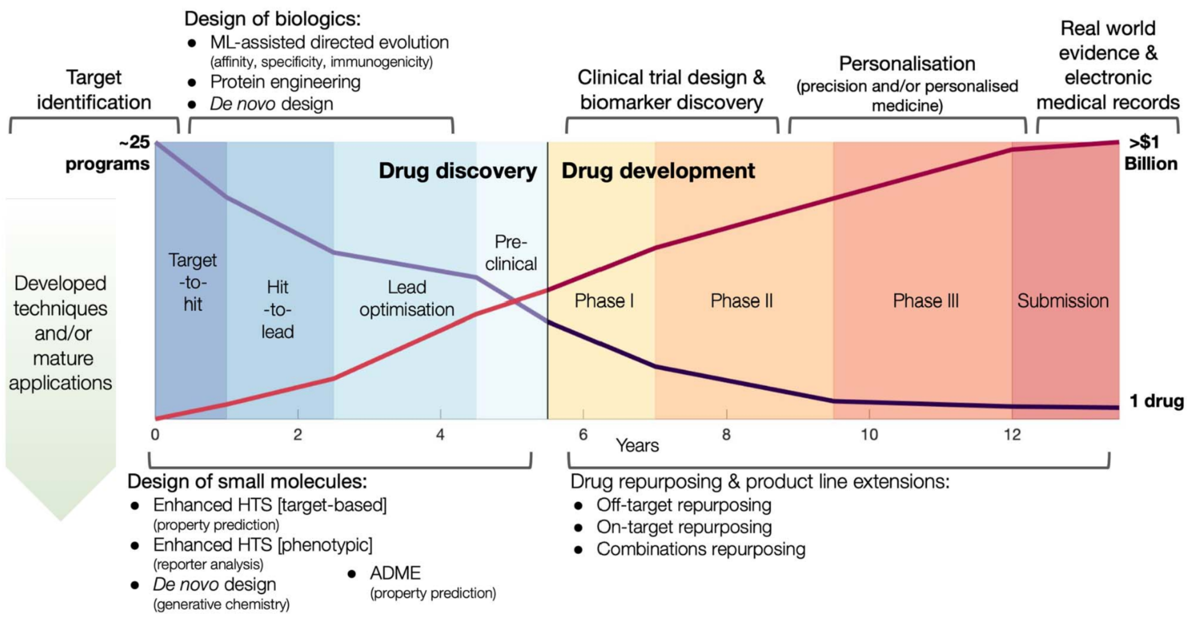


Figure 3.2: The process of drug discovery and development. Image taken from (Gaudelet et al. 2021).

### 3.1.1 Target Identification

In drug development, targets typically refer to molecules such as proteins, receptors, enzymes, or genes that play a crucial role in the progression of a disease. These targets are instrumental in the formation, progression, or symptom manifestation of diseases. Drugs interact specifically with these targets to modulate or inhibit their biological activity, thereby treating or managing the disease.

For example, if a specific protein is critical in the growth and spread of a cancer, that protein can be targeted by a drug. The drug would be specifically designed to bind to this protein, thereby inhibiting its activity and preventing the proliferation of cancer cells.

During this process, GNNs can assist researchers in predicting protein functions and identifying and validating potential drug targets by analyzing complex data such as protein structures, functionalities, and protein-protein interactions (PPI).

**Protein Structure-Based Methods**

Proteins execute their biological functions by folding into specific three-dimensional structures determined by their amino acid sequences. Gligorijevic et al. (2019) have proposed modeling protein structures as graphs to predict protein functions. In this model, proteins are viewed as graphs where nodes represent amino acids, and edges represent physical bonds or spatial proximities between these amino acids. GNNs can identify key features from the protein structure, such as active sites, that directly influence the protein's catalytic activity.

**Protein-Protein Interaction Network-Based Methods**

Ioannidis, Marques, and Giannakis (2019) have used a multi-relational graph approach based on Protein-Protein Interaction (PPI) networks for semi-supervised learning to predict protein functions. This method focuses on understanding how proteins interact within a larger system to infer their functions. In PPI networks, each node typically represents a protein, and the node attributes may include the amino acid sequence, protein domains, subcellular location, and gene expression profiles. These features help describe the biological characteristics and functions of proteins.

### 3.1.2 Drug Screening and Optimization

Once potential drug targets have been identified, researchers proceed with high-throughput screening (HTS), selecting candidate drugs that effectively bind to the targets from thousands of compounds. This is followed by post-screening data analysis, where GNNs are utilized to predict molecular compound attributes and drug-target interactions.

**Molecular Property Prediction**

Molecular property prediction is a crucial step in drug screening and optimization, primarily used to identify compounds with therapeutic potential early on, assess their safety and toxicity, optimize pharmacokinetic properties, and enhance the efficiency of the research and development process (Wieder et al. 2020). By predicting these properties, researchers can screen for the most promising drug candidates before experimental validation, saving time and costs while increasing the likelihood of successful drug development. The application of these predictive tools helps ensure that drugs have the desired biological activity and minimized risk of adverse effects.

#### Drug–Target Interaction Prediction

Drug-target interaction prediction is crucial in the drug discovery and development process, involving the prediction of interactions between small molecule drugs and their biological targets (typically proteins). While traditional methods have struggled with inefficiencies and a lack of biological explanation, the introduction of GNN has significantly enhanced predictive performance.

Feng et al. (2018) pioneered the use of Graph Convolutional Networks (GCN) to predict the real-valued interaction strengths between drugs and proteins, notably improving the prediction accuracy for cold targets. In their study, a form of **structure-based modeling**, drug molecules and target proteins are modeled as graphs, with nodes representing atoms of drugs or amino acids of proteins, and edges denoting chemical bonds or spatial proximities. The GCN processes these graphs to learn and extract deep features, which are then processed through fully connected layers to predict interaction strengths. Subsequent studies have adopted similar approaches to enhance predictive outcomes (Gao et al. 2018; Nguyen et al. 2021).

In contrast, research like Zhao et al. (2021) has improved model accuracy and biological interpretability by integrating drug-protein relationship networks with deep learning technologies. This **network-based modeling** approach represents entire drugs or proteins as nodes and potential or known interactions between them as edges, modeling the entire drug-target interaction network to analyze interaction patterns.

### 3.1.3 Clinical Trials

#### Drug-Drug Interactions Prediction

Drug-drug interactions (DDIs) are crucial in clinical medication, especially for patients who require multiple drug treatments, such as those with chronic diseases or the elderly. DDIs can affect the absorption, distribution, metabolism, and excretion (ADME) of drugs, potentially altering their efficacy or causing adverse reactions. Therefore, accurate prediction and management of DDIs are essential for ensuring medication safety. For example, Zitnik, Agrawal, and Leskovec (2018) successfully predicted potential side effects between drugs by analyzing a multimodal heterogeneous network composed of protein-protein interactions, drug-target interactions, and drug-drug interactions, where different types of edges represent various side effects, as shown in Figure 3.1 (B).

## 3.2 Medical Diagnosis and Analysis

Medical diagnosis often relies on the interpretation of medical images, such as MRI and CT scans, by healthcare professionals. This process can be time-consuming and prone to human error due to variability in individual experience. To address these challenges, advanced computational methods, including deep learning technologies, have been increasingly integrated into the medical field.

Convolutional Neural Networks (CNNs) have shown promise in medical image analysis, effectively detecting anomalies in chest X-rays and identifying malignant cells in histopathology slides (Singh et al. 2018; Dabeer, Khan, and Islam 2019). However, while CNNs excel at analyzing local pixel patterns, they may not fully capture the complex global interactions in medical data. In contrast, GNNs, with their ability to model complex dependencies and interactions, are particularly useful for data that can be represented as graphs, such as brain connectivity analysis, electrophysiological data analysis, and so on.

### 3.2.1 Brain connectivity analysis

Brain connectivity analysis is crucial for understanding and diagnosing various neurological disorders. Functional Magnetic Resonance Imaging (fMRI) is a core data type in this field, providing critical insights into brain function by recording brain activity. And GNNs can offer unique advantages in analyzing such data, as they effectively model the complex interactions and connections between brain regions.

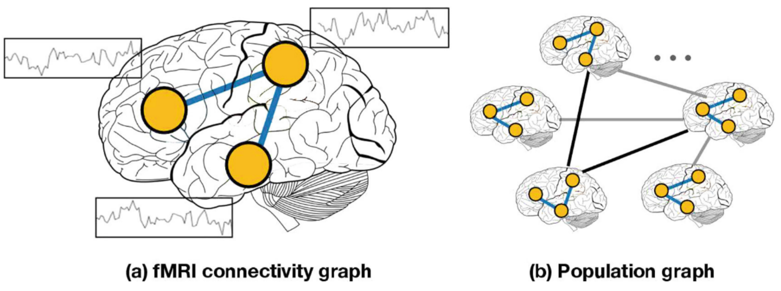


Figure 3.3: (a) fMRI connectivity graph at the individual level. (b) Population graph. Image taken from (Rakhimberdina and Murata 2020).

**Modeling Method:**

The methods are mainly divided into individual graph models and population graph models (Ahmedt-Aristizabal et al. 2021), as illustrated in Figure 3.3. The Table 3.1 below also details their differences:

Table 3.1: Individual and Population Level Graph Models

| Aspect | Individual Level | Population Level |
|---|---|---|
| **Nodes** | ROI (brain regions of interest) | Subjects |
| **Features** | Physiological or statistical properties of the brain region, such as ROI coordinates | RSFC (Resting State Functional Connectivity), phenotypic features (age, gender, city, handedness) |
| **Edges** | Pearson correlation, partial correlation | Phenotypic similarity |
| **Task** | Graph classification | Node classification |

#### Autism Spectrum Disorder (ASD)

In ASD studies, both individual graph models and population graph models are widely used. For individual graph models, researchers construct a brain connectivity graph for each subject and use GNNs to classify these graphs as either ASD or Healthy Control (HC). Population graph models classify subjects by constructing graphs based on similarities between subjects, analyzing overall group characteristics to identify ASD.

#### Analysis of Other Brain Disorders

Similar methods can be applied to other brain disorders such as schizophrenia, major depressive disorder, and bipolar disorder. In these cases, the specifics of implementation and graph models may vary. For instance, schizophrenia studies might focus on identifying functional connectivity anomalies in specific brain regions, while major depressive disorder studies might target dysfunctions in regions associated with emotional regulation. Bipolar disorder research could integrate both functional and structural brain data for a comprehensive analysis. Each case tailors the graph structure and analysis method to the specific characteristics and research goals of the disorder.

### 3.2.2 Electrical Signal Analysis

Electrical signal analysis is a crucial field within medical signal processing, primarily involving the processing and analysis of electroencephalograms (EEG), intracranial electroencephalograms (iEEG), and electrocardiograms (ECG). These electrical signals provide critical information about the functional state of organs such as the brain and heart. By leveraging GNNs, we can effectively capture the spatial and temporal relationships within these signals, thereby enhancing the accuracy of classification and detection tasks.

Graph Neural Network-based electrical signal analysis can be applied to the detection and classification of various diseases, such as epilepsy by analyzing EEG signals to detect seizures (Mathur and Chakka 2020), emotional and mental states by identifying different emotional states (e.g., happiness, sadness) through EEG signals (Jang, Moon, and Lee 2018), and cardiac anomalies by detecting arrhythmias and other heart conditions using ECG signals (H. Wang et al. 2020).

**Modeling Method:**

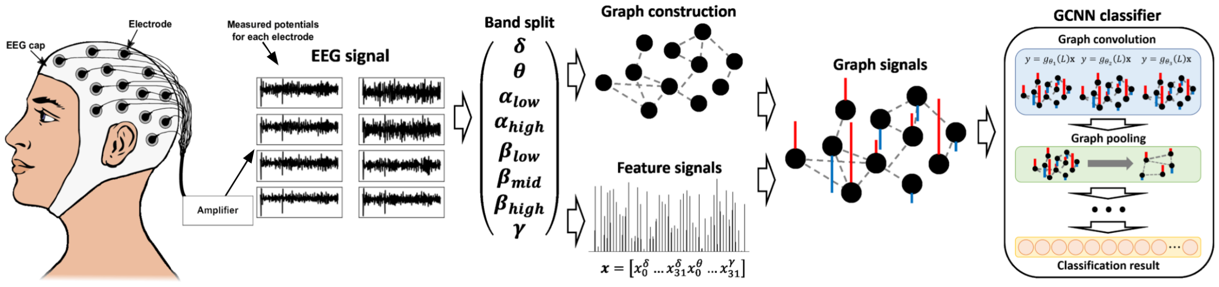


Figure 3.4: Example of process of classifying EEG signals using GNNs. Image adapted from (Nagel 2019) and (Jang, Moon, and Lee 2018).

- **Nodes**: Each node represents an electrode that records local electrical signals (e.g., in EEG analysis, each electrode is placed on the scalp at different positions to capture brain activity signals from various regions, as shown in Figure 3.4).
- **Node Features**: Node features are extracted from the time-series signals recorded by the electrodes. These features can include both time-domain features and frequency-domain features [2].
- **Edges**: Edges represent the relationships between nodes (electrodes), typically based on their spatial proximity or functional connectivity.

**Classification Tasks**:

- **Node-Level Classification**: Each node is independently classified to predict the state of each electrode. (e.g., Independently classify each electrode to predict whether it detects seizure activity.)
- **Graph-Level Classification**: The entire graph is classified as a whole to predict the state of the entire EEG recording. (e.g., Aggregate the node-level classification results. If any electrode's classification result indicates seizure activity, the entire graph is considered to have seizure activity.)

[2]Time-domain features are extracted directly from the signal's time series, such as mean, standard deviation, and peak value. Frequency-domain features, derived from band splitting, include power spectral density and band energy, such as the energy of $\delta$, $\theta$, $\alpha$, $\beta$, and $\gamma$ waves.

### 3.2.3 Image Segmentation

Although Convolutional Neural Networks (CNNs) have already achieved significant success and become the mainstream method in the field of medical image segmentation, GNNs still have substantial room for development due to their ability to handle complex graph-structured data. By dividing an image into multiple interconnected regions or pixel blocks, where each region is treated as a node with features such as pixel intensity and texture, and edges represent the relationships between regions, GNNs can capture the complex spatial structure and contextual information within the image. This capability makes GNNs particularly effective in tasks such as the precise segmentation of organs or lesions like brain tumors and lung nodules, significantly improving segmentation accuracy and robustness (Mohammadi and Allali 2024).

### 3.2.4 Multimodal Fusion

In multimodal medical image analysis, GNNs are widely used to integrate rich information from different modalities (such as CT, MRI, and PET). By representing multimodal image data as graph structures, where node features include values from each modality and edges represent associations either between different modalities or within the same modality, GNNs can effectively integrate and exploit complementary information from multimodal data, enhancing the accuracy of disease detection and diagnosis. This approach is particularly important in the analysis of complex diseases like cancer, where multimodal imaging provides a more comprehensive and accurate diagnosis. By leveraging multimodal fusion, GNNs can better capture the relationships between different imaging modalities, providing a more holistic understanding and diagnosis of diseases.

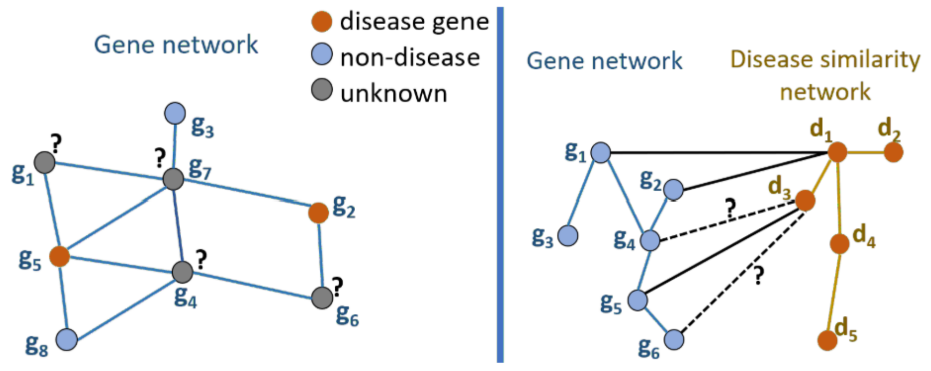
## 3.3 Disease Association Prediction

Disease association prediction is crucial in bioinformatics for identifying connections between diseases and biological entities like genes and RNA. Traditional methods, while somewhat effective, often struggle with the complex structures in biological networks, whereas GNNs offer a powerful alternative.

#### Disease Gene Prediction

Disease gene prediction helps to understand how biomolecules interact within the context of diseases and identifies genes that may be linked to specific conditions. Ata et al. (2021) have proposed two potential predictive frameworks: one based on node classification and the other on link prediction, as illustrated in Figure 3.5.

- **Node Classification**: By analyzing the neighborhood and connection patterns of nodes



(a) Node classification. (b) Link prediction.

Figure 3.5: Tasks in disease gene prediction.

within the network, GNNs can predict the disease status of unlabeled genes, identifying key features that indicate disease association.

- **Link Prediction**: This method is used to predict potential unknown associations between genes and diseases, where the weight of an edge represents the strength or credibility of the association. Cinaglia and Cannataro (2023) have demonstrated how GNN-based link prediction can analyze gene-disease networks, highlighting the potential of GNNs to improve the precision and interpretability of predictions.

Utilizing well-trained GNN models, it is possible to predict and discover new gene-disease associations, which are crucial for the early diagnosis of diseases and the development of new treatments.

**RNA–Disease Association Prediction**

RNA-disease association prediction is a critical area of research in bioinformatics and systems biology, aiming to elucidate the interactions between RNA molecules and specific diseases. Identifying these associations is crucial for understanding the molecular mechanisms of diseases, early diagnosis, and treatment.

Recently, GNNs have been extensively applied in this research field. In these models, RNA molecules and diseases are considered as nodes within a network, while the edges between nodes represent known or potential associations. This network framework allows GNNs to effectively perform link prediction to reveal possible RNA-disease associations. (Lu et al. 2018; Pan and Shen 2019; Yang and Lei 2021; Momanyi et al. 2024)

GNNs have a broad range of applications in biomedicine, including but not limited to drug discovery, disease association prediction, and medical diagnosis and analysis. Beyond these areas, GNNs also hold significant potential for future research and development. In summary, GNNs promise substantial advancements in biomedical research and healthcare, with many more applications yet to be explored.

# 4 Model Implementation

In the previous chapter, we discussed some common applications of GNNs in the field of biomedicine. In this section, we will implement two examples of predicting protein functions using GNNs: protein classification and protein-protein interactions (PPI). For more detailed results, please refer to the appendix.

## 4.1 Protein Classification

Proteins are fundamental molecules that play critical roles in nearly all biological processes. Among the various types of proteins, enzymes (Robinson 2015) are particularly significant. Enzymes are specialized proteins that act as catalysts, speeding up chemical reactions in cells without being consumed in the process. They are essential for numerous cellular functions, including metabolism, DNA replication, and signal transduction. Understanding which proteins function as enzymes is crucial for insights into biological mechanisms and for applications in biotechnology, medicine, and pharmaceuticals.

The primary objective of this task is to use GNNs to classify these protein graphs, categorizing the proteins as either enzymes or non-enzymes. This is a **graph-level classification** task **based on molecular modeling**, and it is a **binary classification** problem. For this task, we utilize the **Spektral** framework, built on TensorFlow/Keras, which supports various types of graph convolutions and pooling layers.

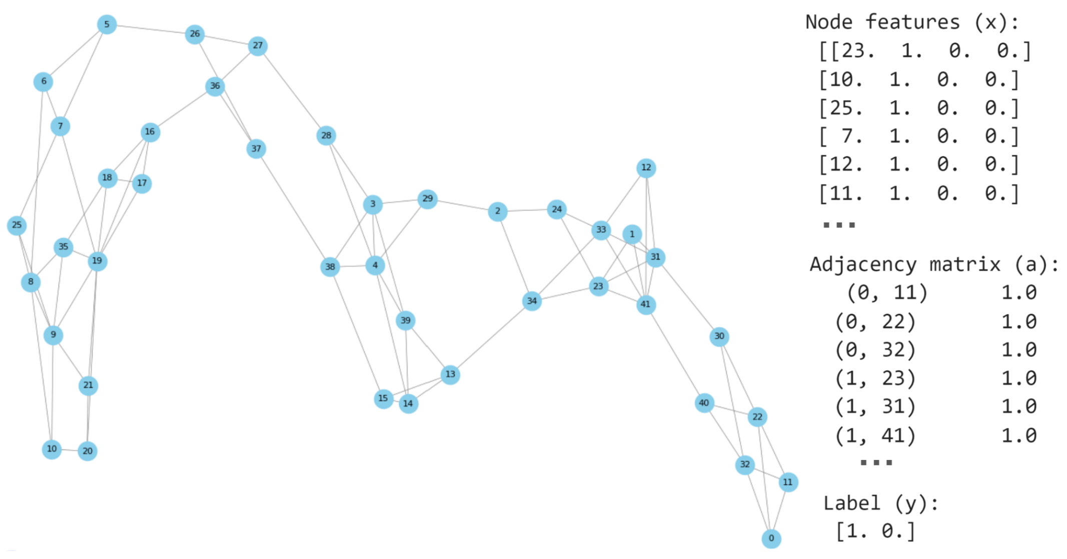
### 4.1.1 Data

The dataset *proteins* is sourced from the paper "Protein function prediction via graph kernels" (Borgwardt et al. 2005) and is part of the benchmark datasets provided by the Technical University of Dortmund's TUDataset collection (More details can be found here). This dataset has been preprocessed for simplicity and ease of use.

Detailed Information:

- **Number of Graphs**: There are 1113 graphs, each representing a different protein.
- **Nodes**: Each node represents an amino acid. On average, each graph contains 39.06 nodes.
- **Node Features**: Each node has 4 features, which likely represent the physicochemical properties of the amino acids.
- **Edges**: Two nodes are connected by an edge if the distance between them is less than 6 Angstroms. On average, each graph has 72.82 edges.

- **Labels**: Each graph is labeled to indicate whether it is an enzyme or not, using a binary classification (0 or 1). The labels are represented using one-hot encoding.



```
Node features (x):
 [[23.  1.  0.  0.]
 [10.  1.  0.  0.]
 [25.  1.  0.  0.]
 [ 7.  1.  0.  0.]
 [12.  1.  0.  0.]
 [11.  1.  0.  0.]
 ...

Adjacency matrix (a):
   (0, 11)      1.0
  (0, 22)       1.0
  (0, 32)       1.0
  (1, 23)       1.0
  (1, 31)       1.0
  (1, 41)       1.0
    ...

 Label (y):
  [1. 0.]
```

Figure 4.1: An example of a protein graph, illustrating the graph structure, the node feature matrix, the adjacency matrix, and the label.

### 4.1.2 Modeling and Training

#### 4.1.2.1 Model Definition

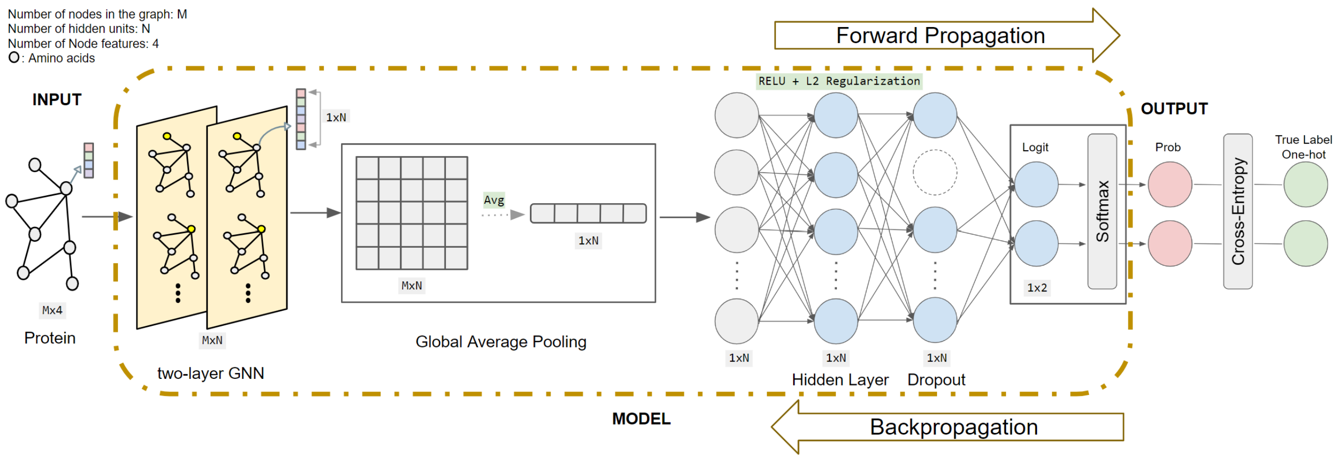


Figure 4.2: Overall Model Architecture and Workflow.

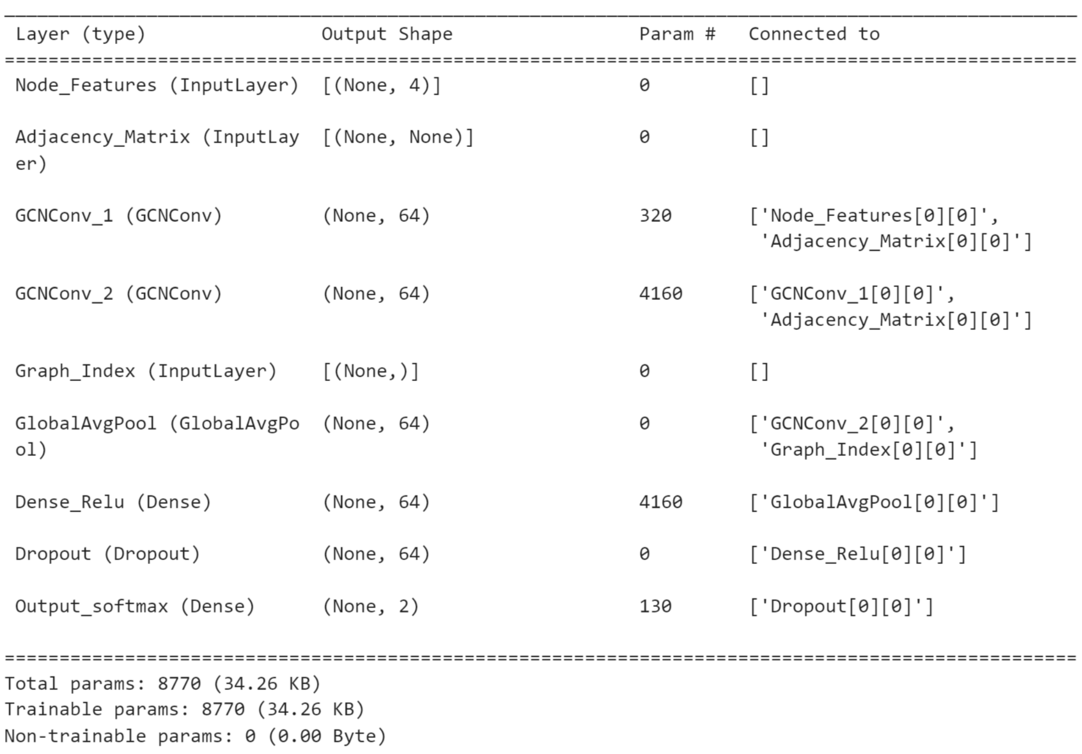
The model architecture consists of several key components designed for protein classification. The input layer accepts graphs representing proteins, where each node corresponds to an amino acid and initially has 4 features.

Following the input layer, there are two GNN layers, which can be Graph Convolutional Network (GCN), Graph Isomorphism Network (GIN) (Xu et al. 2018), or GraphSAGE-Mean layers. These layers transform the initial node features into new feature representations by aggregating information from the local neighborhood of each node. We will experiment with each of these methods.

After the GNN layers, a global average pooling layer is applied. This layer computes the average of the node features across the entire graph, resulting in a single global feature vector for each protein.

The global feature vector is then processed through a hidden layer with ReLU activation and L2 regularization to enhance model generalization. To further prevent overfitting, a dropout layer is included, which randomly drops a fraction of the hidden units during training.

The final output is produced by a dense (Logit) layer, which reduces the dimensionality to match the number of output classes. A softmax activation function is applied to convert the logits into probability distributions over the two classes (enzyme or non-enzyme).



```
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 Node_Features (InputLayer)  [(None, 4)]                  0         []

 Adjacency_Matrix (InputLay  [(None, None)]               0         []
 er)

 GCNConv_1 (GCNConv)         (None, 64)                   320       ['Node_Features[0][0]',
                                                                     'Adjacency_Matrix[0][0]']

 GCNConv_2 (GCNConv)         (None, 64)                   4160      ['GCNConv_1[0][0]',
                                                                     'Adjacency_Matrix[0][0]']

 Graph_Index (InputLayer)    [(None,)]                    0         []

 GlobalAvgPool (GlobalAvgPo  (None, 64)                   0         ['GCNConv_2[0][0]',
 ol)                                                                 'Graph_Index[0][0]']

 Dense_Relu (Dense)          (None, 64)                   4160      ['GlobalAvgPool[0][0]']

 Dropout (Dropout)           (None, 64)                   0         ['Dense_Relu[0][0]']

 Output_softmax (Dense)      (None, 2)                    130       ['Dropout[0][0]']

==================================================================================================
Total params: 8770 (34.26 KB)
Trainable params: 8770 (34.26 KB)
Non-trainable params: 0 (0.00 Byte)
```

Figure 4.3: GCN Model Summary

In order to provide a more concrete example, we present the detailed architecture of the GCN model. Figure 4.3 shows the layers, connections, and corresponding parameter counts within the GCN model used in our experiments.

#### 4.1.2.2 Compilation of the Model

The model employs categorical cross-entropy as the loss function, which is suitable for comparing the predicted probabilities against the true one-hot encoded labels. Even though the task is a binary classification problem, categorical_crossentropy is chosen over binary_crossentropy due to the one-hot encoding format of the labels. For the optimization of the model, the Adam optimizer is used. Furthermore, the model's performance is evaluated using accuracy as the metric, which directly reflects the proportion of correctly classified instances.

#### 4.1.2.3 Training and Evaluating

Given the moderate size of the dataset and significant variability in model performance across different training-test splits, we have chosen **k-fold cross-validation** as our primary evaluation method. Initially, 10% of the data is randomly selected to serve as a fixed test set. The remaining 90% is subjected to k-fold cross-validation. This strategy allows us to determine the most effective epoch by identifying which of the 100 training epochs achieves the highest accuracy.

Furthermore, we continuously experiment with various hyperparameters during each training cycle, including learning rates, the number of hidden units, and the number of folds, to optimize the model's performance. After identifying the optimal hyperparameters, we re-train the final model on the entire 90% training portion of the dataset. Parameters are meticulously optimized using the backpropagation algorithm throughout the training process. The model's performance is then assessed on the designated test set. This comprehensive approach ensures a robust model evaluation, effectively mitigating the effects of data variability and maximizing the potential of the available data.

### 4.1.3 Results

We evaluated three different models: GCN, GIN, and GraphSAGE-Mean. Each model was trained and validated using K-fold cross-validation. The performance of each model on the validation and test sets is summarized in Table 4.1. The hyperparameters used for each model are listed in Table 4.2.

Table 4.1: Best Epoch Model Performance on Validation and Test Sets

| Model | Validation Accuracy | Val Acc SD | Validation Loss | Val Loss SD | Test Accuracy | Test Loss |
|---|---|---|---|---|---|---|
| GCN | 0.75 | 0.03 | 0.55 | 0.03 | 0.77 | 0.52 |
| GIN | 0.75 | 0.05 | 0.55 | 0.07 | 0.74 | 0.55 |
| GraphSAGE | 0.73 | 0.04 | 0.56 | 0.03 | 0.77 | 0.52 |

*Note: "Val Acc SD" and "Loss SD" are standard deviations of accuracy and loss, respectively, across multiple folds at the best epoch. Results may vary due to various training factors.*

Table 4.2: Hyperparameters and Parameters for Each Model

| Model | Learning Rate | Hidden Units | Dropout Rate | K-fold | Epochs | Number of Parameters |
|---|---|---|---|---|---|---|
| GCN | 0.001 | 64 | 0.3 | 7 | 97 | 8770 |
| GIN | 0.001 | 64 | 0.4 | 7 | 98 | 8772 |
| GraphSAGE | 0.001 | 32 | 0.4 | 7 | 92 | 3490 |

*Note: Learning rates tested include 0.1, 0.01, 0.001. Hidden units options are 32, 64, 128. Dropout rates range from 0.2 to 0.5. K-fold cross-validation is conducted with 5 to 10 folds, and epochs range from 1 to 100.*

The results demonstrate that the three models—GCN, GIN, and GraphSAGE-Mean—show very similar performance on this dataset, both in terms of validation and test accuracy and loss. This suggests that the choice of the specific GNN layer (GCN, GIN, or GraphSAGE-Mean) does not significantly impact the performance for this particular protein classification task.

### 4.1.4 Explainability

We utilize GNNExplainer (Ying et al. 2019), a tool specifically designed to explain graph neural networks, to identify the most important node features and edges contributing to our model's predictions. For this analysis, we randomly select a protein from our dataset and analyze its prediction.

GNNExplainer works by learning a mask for the input graph. This mask highlights the key node features and edges that influence the model's predictions. The mask values are typically transformed through a sigmoid function, resulting in values ranging from [0, 1]. To identify the most significant edges or features, these values are binarized based on a chosen threshold: values above the threshold are set to 1 (indicating high importance), while those below are set to 0 (indicating lower importance). This binarization process allows us to clearly distinguish between crucial and non-crucial elements in the graph, providing a focused explanation of the model's predictions.

To further illustrate the application of GNNExplainer, we randomly selected a graph from the test set and used the GraphSAGE model for analysis.

### Important Edges

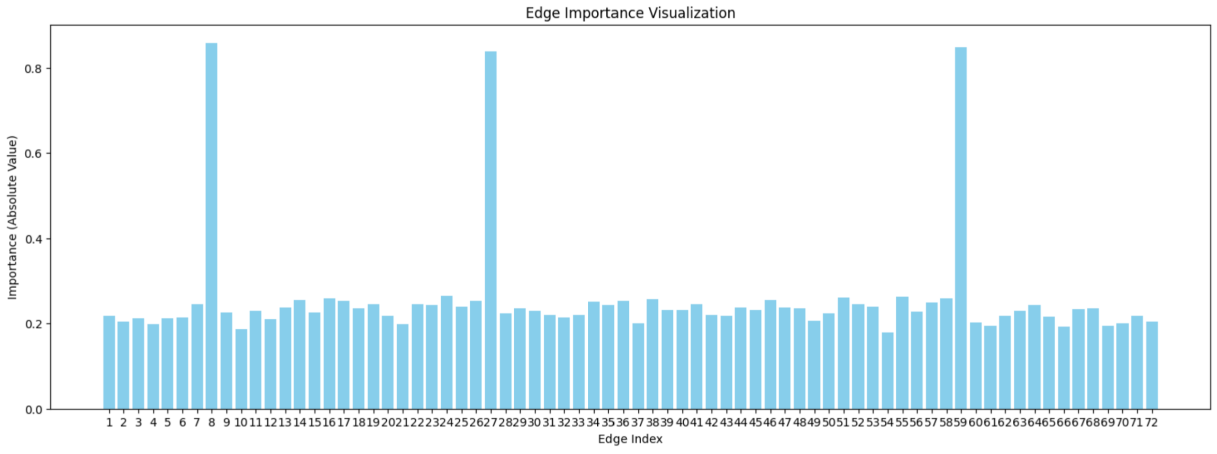
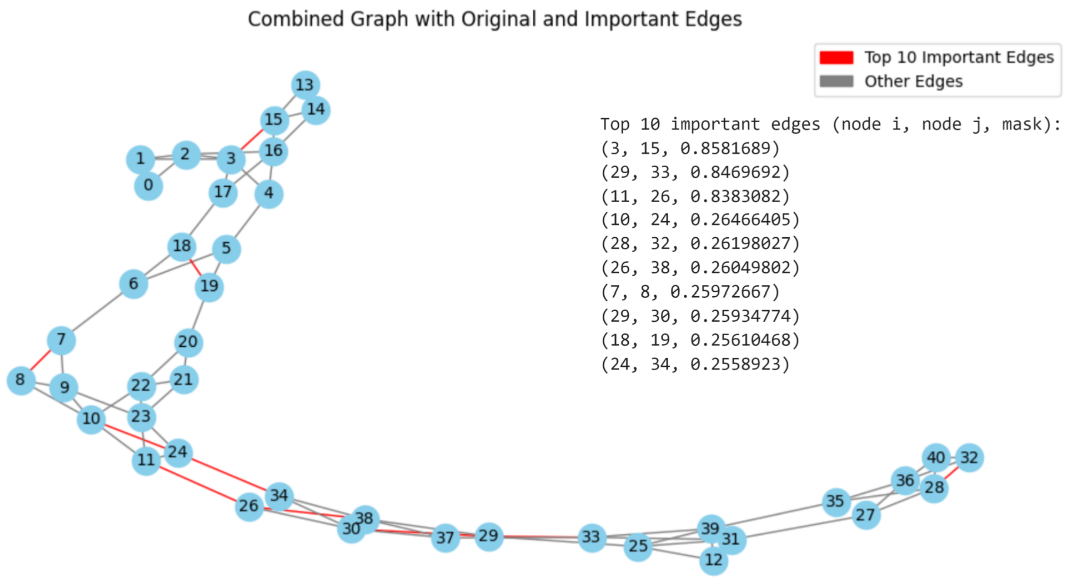


Figure 4.4: Edge Importance Visualization

The edge importance visualization, as shown in Figure 4.4, indicates that most edges have importance values ranging from 0.2 to 0.25. However, there are three edges that stand out with importance values exceeding 0.8, approaching 1.0. These high values suggest that these edges are extremely significant for the model's prediction.

Due to the presence of these few highly important edges, setting a fixed threshold for edge importance is challenging. If the threshold is set too low, many less important edges will be included, potentially cluttering the explanation. Conversely, a high threshold might exclude edges that are still relatively important. Therefore, instead of using a fixed threshold, we have chosen to highlight the top 10 most important edges. As illustrated in Figure 4.5, these top 10 edges are marked in red within the protein graph, allowing for a clear visual representation of the subgraph that has the most substantial impact on the model's prediction.



```
Top 10 important edges (node i, node j, mask):
(3, 15, 0.8581689)
(29, 33, 0.8469692)
(11, 26, 0.8383082)
(10, 24, 0.26466405)
(28, 32, 0.26198027)
(26, 38, 0.26049802)
(7, 8, 0.25972667)
(29, 30, 0.25934774)
(18, 19, 0.25610468)
(24, 34, 0.2558923)
```

Figure 4.5: Combined Graph with Original and Important Edges

#### Important Node Features

GNNExplainer not only allows us to assess the importance of edges but also enables the evaluation of node feature importance. The results show that the mask values for the four features of the selected protein graph are approximately equal: 0.2570, 0.2507, 0.2478, and 0.2542. This indicates that no single feature stands out as particularly crucial, suggesting that each feature contributes similarly to the model's prediction, with none being highly significant in isolation.

## 4.2 Protein-Protein Interactions (PPI)

Protein-Protein Interaction (PPI) is a vital process within biological systems, playing a crucial role not only in drug development but also in regulating gene expression, cell signaling, and maintaining cell cycle and apoptosis. These interactions enable cells to respond to external signals and maintain normal function and health, which is essential for understanding disease mechanisms and developing therapeutic strategies.

The primary objective of this task is to employ GNNs to predict protein functions. This is a **node-level classification** task **based on network modeling**, and it represents a **multi-class, multi-label** problem. For this task, we utilize the **PyG** (PyTorch Geometric) framework, which provides comprehensive tools for building and training GNN models.

### 4.2.1 Data

The dataset *PPI* is sourced from the paper "Predicting Multicellular Function through Multi-layer Tissue Networks" (Zitnik and Leskovec 2017).

Detailed Information:

- **Number of Graphs**: There are 24 graphs, each corresponding to different human tissues.
- **Nodes**: Each node represents a protein. On average, each graph contains 2,371 nodes, totaling 56,944 nodes across the dataset.
- **Node Features**: Each node has 50 features, which include positional gene sets, motif sets, and immunological characteristics.
- **Edges**: Each edge represents an interaction between two proteins. The entire dataset contains a total of 1,587,264 edges, with an average of approximately 66,136 edges per graph.
- **Labels**: The dataset uses Gene Ontology (GO)[3] for labeling, comprising 121 different

[3]Gene Ontology (GO) terms (Ashburner et al. 2000) are used to describe protein functions, providing a standardized vocabulary to define the roles of proteins in molecular functions, biological processes, and cellular components.

labels. The labels are not encoded in a one-hot manner.

Here is the division of the dataset into training, validation, and test sets.

```
Train Dataset:                  Validation Dataset:            Test Dataset:
=================               =================              =================
Number of graphs: 20            Number of graphs: 2            Number of graphs: 2
Total number of nodes: 44906    Total number of nodes: 6514    Total number of nodes: 5524
Total number of edges: 1226368  Total number of edges: 198920  Total number of edges: 161976
Node feature dimension: 50      Node feature dimension: 50     Node feature dimension: 50
Number of classes: 121          Number of classes: 121         Number of classes: 121
```

Figure 4.6: Summary of Datasets.

### 4.2.2 Modeling and Training

#### 4.2.2.1 Model Definition

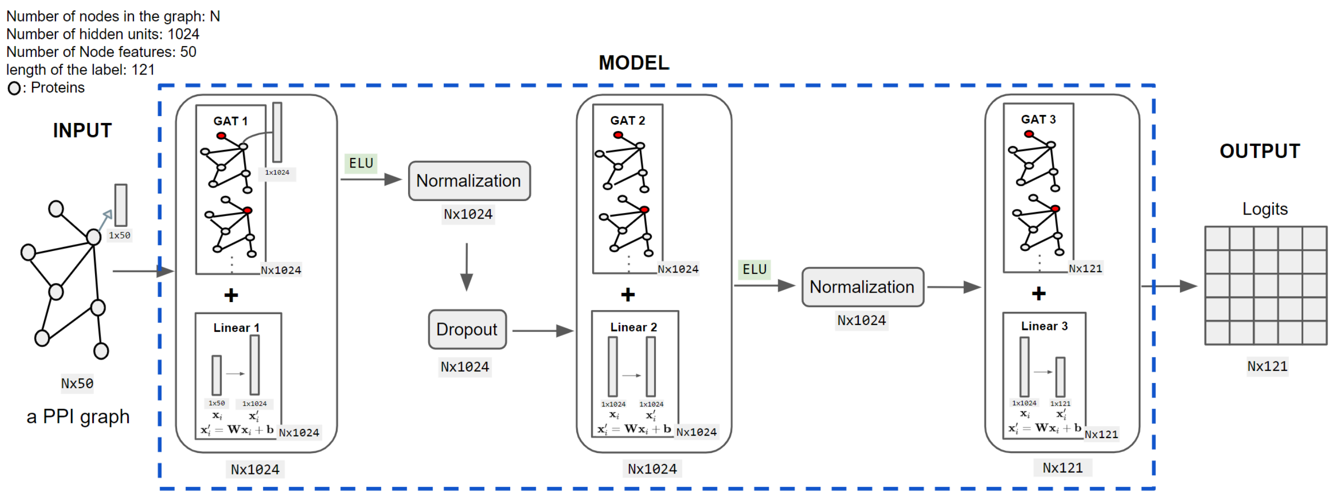


Figure 4.7: Overall Model Architecture

This model employs a Graph Attention Network (GAT) framework designed to process data with complex relationships. By combining graph attention layers with linear transformation layers in each layer, the model optimizes and merges features layer by layer. Each layer enhances the information exchange between nodes and improves the generalizability of features through activation functions and regularization techniques. The detailed process is illustrated in the accompanying Figure 4.7. The model structure is as follows:

In the first layer, GAT and linear transformation are simultaneously applied to the input features. This layer's GAT is configured with four attention heads and defaults to concatenating features. Both the GAT and linear layers expand the input features from 50 to 1024 dimensions (256 dimensions per head across four heads), then their outputs are added together to merge information. This design allows each node to extract rich contextual information from its neighbors while retaining essential original features. Subsequently, the ELU activation function introduces non-linearity, batch normalization standardizes feature distribution, and a dropout layer reduces the risk of overfitting, thus enhancing the model's stability and generalizability.

The second layer continues the fusion strategy of GAT and linear layers, similar in structure to the first layer. This layer aims to further refine and deepen the feature representation between nodes to better capture and understand deeper inter-node relationships. In this layer, the number of hidden units and attention heads remains constant to ensure feature dimension stability. The use of ELU activation and batch normalization again helps maintain data stability while enhancing the model's ability to recognize complex data patterns.

In the final layer of the model, the GAT is adjusted to output dimensions of 121, corresponding to the number of labels in a multi-label classification. Although four attention heads are retained, the layer opts for averaging instead of concatenating the outputs from each head to control the output dimensions and prepare for final predictions. The linear transformation layer adjusts its output dimensions accordingly to suit the classification task. The output of this layer is the logits used directly for the classification task, with no further activation function applied to facilitate direct use in loss function calculation.

```
+---------------------+-------------------------+----------------+-----------+
| Layer               | Input Shape             | Output Shape   | #Param    |
|---------------------+-------------------------+----------------+-----------|
| GAT                 | [1767, 50], [2, 32318]  | [1767, 121]    | 2,832,570 |
| ├─(gat1)GATConv     | [1767, 50], [2, 32318]  | [1767, 1024]   | 54,272    |
| ├─(lin1)Linear      | [1767, 50]              | [1767, 1024]   | 52,224    |
| ├─(bn1)BatchNorm1d  | [1767, 1024]            | [1767, 1024]   | 2,048     |
| ├─(dropout1)Dropout | [1767, 1024]            | [1767, 1024]   | --        |
| ├─(gat2)GATConv     | [1767, 1024], [2, 32318] | [1767, 1024]  | 1,051,648 |
| ├─(lin2)Linear      | [1767, 1024]            | [1767, 1024]   | 1,049,600 |
| ├─(bn2)BatchNorm1d  | [1767, 1024]            | [1767, 1024]   | 2,048     |
| ├─(gat3)GATConv     | [1767, 1024], [2, 32318] | [1767, 121]   | 496,705   |
| ├─(lin3)Linear      | [1767, 1024]            | [1767, 121]    | 124,025   |
+---------------------+-------------------------+----------------+-----------+
```

Figure 4.8: Summary of GAT Model Layer Transformations and Parameters

To demonstrate the model's functionality, we examine the first graph in the training set. This example highlights the changes in dimensionality and the number of parameters across each model layer. Figure 4.8 illustrates these transformations and details, providing clear insights into how each layer processes and refines node features.

#### 4.2.2.2 Compilation of the Model

In the model configuration, as specified in Table 2.1, we utilize the **Binary Cross-Entropy** with Logits Loss function, which is specifically designed for binary and multi-label classification tasks. This function incorporates a sigmoid layer within the loss calculation, facilitating the handling of predictions in logits form. This integration offers numerical stability during training and enhances performance by directly optimizing the model's output for probability thresholds. Regarding the optimizer, the **Adam** optimizer is chosen to efficiently manage weight updates during the training process.

The **F1 score** is employed as a primary performance metric, particularly valuable in binary and multi-label classification tasks. The F1 score is the harmonic mean of precision and recall, providing a balanced measure of the model's accuracy and its ability to handle positive class predictions.

To comprehensively evaluate the model's performance across all labels, we employ the **micro-average F1**[4] score calculation method. The micro-average approach accumulates the True Positives (TP), False Positives (FP), and False Negatives (FN) across all labels, and then computes the Precision and Recall based on these aggregated statistics to determine the F1 score. The formula for the F1 score is as follows:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Where:

$$\text{Precision} = \frac{\text{Total TP}}{\text{Total TP} + \text{Total FP}}, \text{Recall} = \frac{\text{Total TP}}{\text{Total TP} + \text{Total FN}}$$

#### 4.2.2.3 Training and Evaluating

Model training is completed through multiple epochs. In each epoch, the training data is divided into 20 batches (batch size of 1), and the validation data is divided into 1 batch (batch size of 2). For each batch, the model first performs forward propagation to compute predictions. Then, the loss function is used to calculate prediction errors. Next, backpropagation is used to compute gradients, and finally, the optimizer updates the model parameters based on these gradients.

During the training process, we employed an early stopping mechanism, which is primarily used to prevent overfitting and ensure the generalization ability of the model. In this instance, the mechanism operates by monitoring the model's F1 score on the validation set: if the F1 score does not improve by at least 0.001 over 10 consecutive training epochs, the training will be prematurely halted. This strategy not only helps save computational resources but also ensures that the model performs optimally in practical applications.

[4]Another method, called the **Macro-average F1 score**, calculates the F1 score independently for each label and then takes the average.

Then, we saved the trained model for evaluation. The test data is divided into 1 batch with a batch size of 2 for the evaluation process.

### 4.2.3 Results

Here is a refined and supplemented version of the performance analysis:

Table 4.3: Performance Metrics Summary

| Metric | Training | Validation | Testing |
|---|---|---|---|
| **Loss** | 0.0118 | 0.0529 | 0.0280 |
| **F1 Score** | 0.9934 | 0.9823 | 0.9890 |

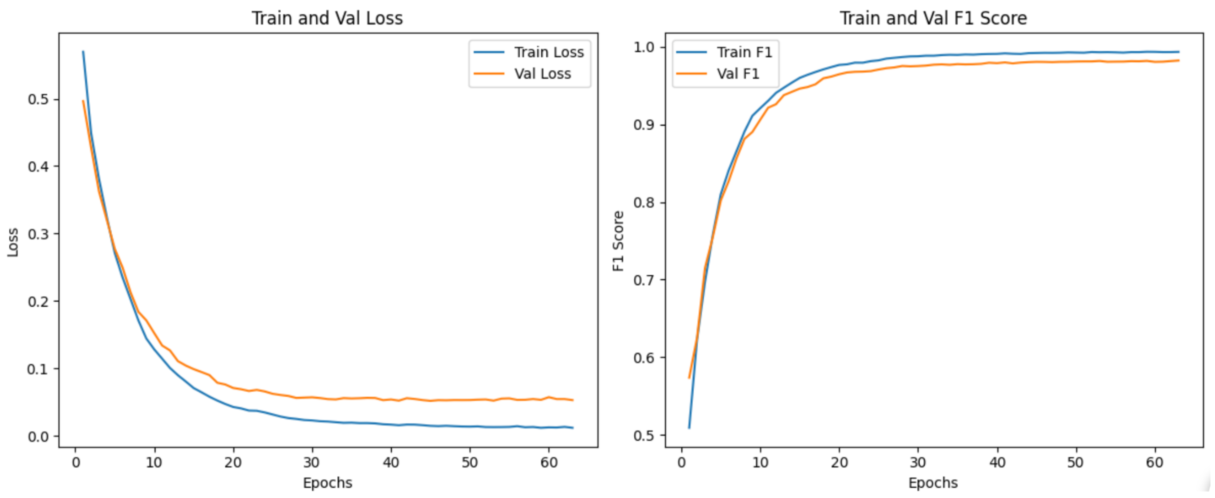


Figure 4.9: Training and Validation Loss and F1 Score Over Epochs

The metrics table, along with Figure 4.9, concisely demonstrates the model's performance across the training, validation, and testing phases:

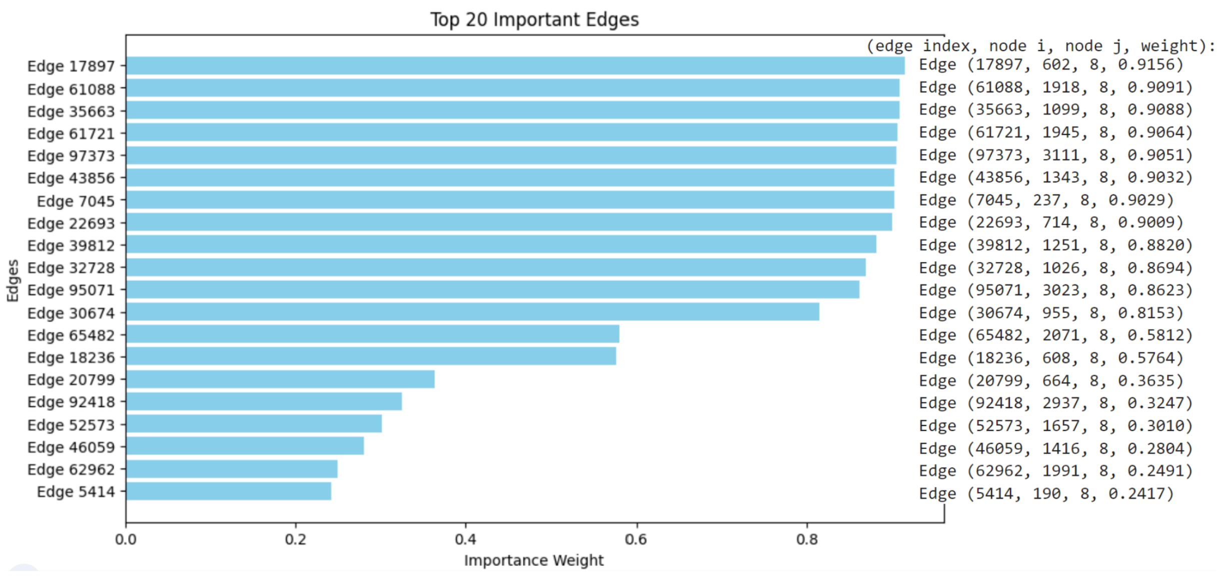
- **Loss Trends**: The training phase shows the lowest loss at 0.0118, indicative of effective initial learning. The validation loss, although slightly higher at 0.0529, suggests adequate generalization capabilities. The test loss stands at 0.0280, confirming consistent performance on unseen data.
- **F1 Score Trends**: The model achieves high F1 scores across all datasets, with the training phase nearly perfect at 0.9934. The validation score slightly drops to 0.9823 but maintains a high level in testing at 0.9890, emphasizing the model's robustness and precision.

These trends indicate that the model quickly adapts to the necessary features, evidenced by the rapid initial decrease and subsequent stabilization of loss shown in the Figure 4.9. Importantly, there is no sign of overfitting, as the model demonstrates a stable performance curve across different datasets, showcasing its ability to generalize well and perform consistently in varied environments.

### 4.2.4 Explainability

We also employ GNNExplainer, a tool designed to interpret graph neural networks, which helps us identify the most influential node features and edges that contribute to our model's predictions. For this analysis, we focus on a single node within a graph from the test set, examining which features and connections most significantly impact its predicted outcome.

#### Important Edges



```
(edge index, node i, node j, weight):
Edge (17897, 602, 8, 0.9156)
Edge (61088, 1918, 8, 0.9091)
Edge (35663, 1099, 8, 0.9088)
Edge (61721, 1945, 8, 0.9064)
Edge (97373, 3111, 8, 0.9051)
Edge (43856, 1343, 8, 0.9032)
Edge (7045, 237, 8, 0.9029)
Edge (22693, 714, 8, 0.9009)
Edge (39812, 1251, 8, 0.8820)
Edge (32728, 1026, 8, 0.8694)
Edge (95071, 3023, 8, 0.8623)
Edge (30674, 955, 8, 0.8153)
Edge (65482, 2071, 8, 0.5812)
Edge (18236, 608, 8, 0.5764)
Edge (20799, 664, 8, 0.3635)
Edge (92418, 2937, 8, 0.3247)
Edge (52573, 1657, 8, 0.3010)
Edge (46059, 1416, 8, 0.2804)
Edge (62962, 1991, 8, 0.2491)
Edge (5414, 190, 8, 0.2417)
```

Figure 4.10: Top 20 Important Edges

In this analysis, we focus on a specific node, node 8, from the test set, utilizing GNNExplainer to reveal the 20 most influential edges affecting its prediction outcome. As shown in Figure 4.10, these edges directly connect to node 8, indicating the model's dependency on these specific connections for making predictions. The figure shows that the distribution of edge weights indicates that the top 12 edges all have weights above 0.8, demonstrating their significant impact on the model's prediction results. However, the weights of the last 6 edges are all below 0.4, suggesting their relatively minor influence on the model.

Figure 4.11 presents these 20 important edges and their connected nodes in a more visually intuitive manner (due to the size limitations of the graph, only important edges and their connecting nodes are displayed). From this figure, we can clearly see that all edges identified as important are directly connected to node 8, further corroborating the model's potential reliance on features related to node 8 or its connection patterns. This phenomenon may suggest a strong dependency

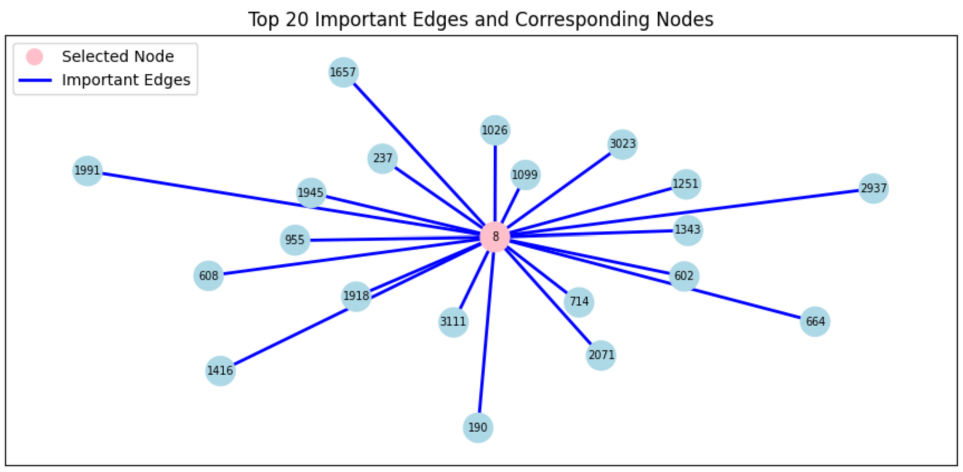


Figure 4.11: Top 20 Important Edges and Corresponding Nodes

of the model on specific node features, significantly impacting prediction outcomes, or it might indicate that the model's predictions are heavily reliant on the node itself and its neighboring nodes. Of course, these are just possibilities, and different models may exhibit different explanatory results.

### Important Node Features

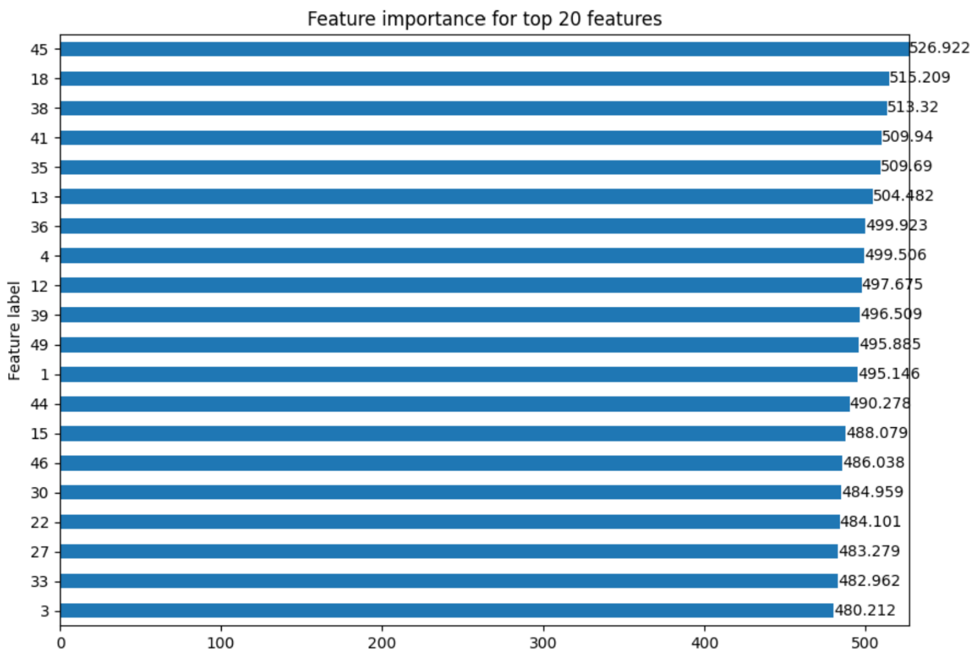


Figure 4.12: Feature Importance for Top 20 Features

Figure 4.12 displays the importance scores for the top 20 features in the model. Each node possesses 50 features, including positional gene sets, motif sets, and immunological characteristics. From the chart, it is evident that the importance scores of these features are very close, ranging from 480 to 530, suggesting that they almost equally influence the model's prediction output. This tight distribution of scores may indicate a broad dependency of the model on various features, with no single feature predominating. The decision-making process of the model likely requires the integration of multiple biological signals and characteristics.

# 5 Conclusion

Graph Neural Networks (GNNs), as a branch of deep learning, excel in processing non-Euclidean and graph-structured data across various tasks. This project presents a comprehensive review of GNNs and their advancements in biomedicine from multiple perspectives. According to specific applications for various omics data, we categorize and discuss related studies in three key areas: drug discovery and development, medical diagnosis and analysis, and disease association prediction. Additionally, we implemented the application of GNNs in predicting protein functions for drug development, focusing on two representative tasks based on different levels of structural information: node classification and graph classification. Our practical experiments demonstrate the effectiveness and explainability of GNNs in several biomedical research domains.

Despite the excellent results GNNs have achieved in many biomedical tasks, they still face challenges, particularly in the complexity of data preparation and preprocessing, as well as in achieving explainability and measuring the explained results. There is still a long road ahead in overcoming these challenges. However, we firmly believe that with continuous advancements, GNNs are expected to become increasingly integral in addressing complex biological questions and driving innovations in the field of biomedicine.

# 6 Reflections and Future Work

This project has been a significant and challenging undertaking. Despite having some background in machine learning and bioinformatics, focusing on a specialized and relatively new branch of deep learning like Graph Neural Networks (GNNs) posed considerable difficulties. However, these challenges have also been a source of immense learning and growth for me.

Firstly, throughout this project, I transitioned from understanding the basics of machine learning to delving into neural network concepts. Starting from Convolutional Neural Networks (CNNs) and gradually moving to GNNs, I gained a thorough understanding of how to apply these models to specific tasks. Secondly, my programming skills have been expanded and enhanced. Previously, my programming experience was primarily in R. For this project, I chose to use the more widely adopted Python, which required learning a new language and its libraries. Additionally, I learned to use GitHub to find and understand some open-source GNN code. Thirdly, writing the thesis itself was a significant learning experience. Since there is a lack of books on GNNs available on the market, I had a large amount of literature to review. Using RMarkdown for documentation, I encountered numerous errors and challenges, but each issue was an opportunity to learn and improve my ability to write in LaTeX.

Despite overcoming many obstacles, some challenges remain unresolved. One of the biggest challenges in implementing GNNs was data preprocessing. The tasks I worked on were based on pre-processed data provided by other researchers. Thus, I realized that I lacked the technical knowledge on how to convert raw data (such as images or text) into a form suitable for GNNs (including nodes, edges, adjacency matrices, etc.). For instance, in Figure 2.13, while I theoretically understand how to construct a graph, I am unsure of the practical steps involved in transforming an image (a) into a graph structure (b). Whether this process is done manually or involves specific automated techniques is still a question for me. Addressing this gap will be a focus of my future research, where I will explore the methods and technologies for preprocessing raw data for GNNs.

Looking ahead, I am also interested in exploring the application of GNNs in other domains, such as financial risk, transportation, and more. These fields present unique challenges and opportunities for leveraging the strengths of GNNs to uncover new insights and solutions.

In conclusion, this project has not only expanded my technical knowledge and skills but also prepared me for future research and professional challenges in the field of bioinformatics and machine learning. I look forward to continuing my journey in this exciting and rapidly evolving area of study.

# 7 Bibliography

A., Saranya, and Subhashini R. 2023. "A Systematic Review of Explainable Artificial Intelligence Models and Applications: Recent Developments and Future Trends." *Decision Analytics Journal* 7: 100230. https://doi.org/https://doi.org/10.1016/j.dajour.2023.100230.

Ahmedt-Aristizabal, David, Mohammad Ali Armin, Simon Denman, Clinton Fookes, and Lars Petersson. 2021. "Graph-Based Deep Learning for Medical Diagnosis and Analysis: Past, Present and Future." *Sensors* 21 (14): 4758.

Arrieta, Alejandro Barredo, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, et al. 2020. "Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges Toward Responsible AI." *Information Fusion* 58: 82–115.

Ashburner, Michael, Catherine A Ball, Judith A Blake, David Botstein, Heather Butler, J Michael Cherry, Allan P Davis, et al. 2000. "Gene Ontology: Tool for the Unification of Biology." *Nature Genetics* 25 (1): 25–29.

Ata, Sezin Kircali, Min Wu, Yuan Fang, Le Ou-Yang, Chee Keong Kwoh, and Xiao-Li Li. 2021. "Recent Advances in Network-Based Methods for Disease Gene Prediction." *Briefings in Bioinformatics* 22 (4): bbaa303.

Baldassarre, Federico, and Hossein Azizpour. 2019. "Explainability Techniques for Graph Convolutional Networks." *arXiv Preprint arXiv:1905.13686*.

Borgwardt, Karsten M, Cheng Soon Ong, Stefan Schönauer, SVN Vishwanathan, Alex J Smola, and Hans-Peter Kriegel. 2005. "Protein Function Prediction via Graph Kernels." *Bioinformatics* 21 (suppl_1): i47–56.

Cao, Chensi, Feng Liu, Hai Tan, Deshou Song, Wenjie Shu, Weizhong Li, Yiming Zhou, Xiaochen Bo, and Zhi Xie. 2018. "Deep Learning and Its Applications in Biomedicine." *Genomics, Proteomics and Bioinformatics* 16 (1): 17–32.

Chollet, Francois. 2021. *Deep Learning with Python*. Simon; Schuster.

Cinaglia, Pietro, and Mario Cannataro. 2023. "Identifying Candidate Gene–Disease Associations via Graph Neural Networks." *Entropy* 25 (6): 909.

Dabeer, Sumaiya, Maha Mohammed Khan, and Saiful Islam. 2019. "Cancer Diagnosis in Histopathological Image: CNN Based Approach." *Informatics in Medicine Unlocked* 16: 100231. https://doi.org/https://doi.org/10.1016/j.imu.2019.100231.

Esteva, Andre, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. "Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks." *Nature* 542 (7639): 115–18.

Feng, Qingyuan, Evgenia Dueva, Artem Cherkasov, and Martin Ester. 2018. "Padme: A Deep Learning-Based Framework for Drug-Target Interaction Prediction." *arXiv Preprint arXiv:1807.09741*.

Gao, Kyle Yingkai, Achille Fokoue, Heng Luo, Arun Iyengar, Sanjoy Dey, Ping Zhang, et al.

2018. “Interpretable Drug Target Prediction Using Deep Neural Representation.” In *IJCAI*, 2018:3371–77.

Gaudelet, Thomas, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, et al. 2021. “Utilizing Graph Machine Learning Within Drug Discovery and Development.” *Briefings in Bioinformatics* 22 (6): bbab159.

Gligorijevic, Vladimir, P Douglas Renfrew, Tomasz Kosciolek, Julia Koehler Leman, Kyunghyun Cho, Tommi Vatanen, Daniel Berenberg, et al. 2019. “Structure-Based Function Prediction Using Graph Convolutional Networks. bioRxiv.”

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. 2011. “Deep Sparse Rectifier Neural Networks.” In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 315–23. JMLR Workshop; Conference Proceedings.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT press.

Hamilton, Will, Zhitao Ying, and Jure Leskovec. 2017. “Inductive Representation Learning on Large Graphs.” *Advances in Neural Information Processing Systems* 30.

Hinton, Geoffrey E, and Ruslan R Salakhutdinov. 2006. “Reducing the Dimensionality of Data with Neural Networks.” *Science* 313 (5786): 504–7.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. “Long Short-Term Memory.” *Neural Computation* 9 (8): 1735–80.

Hu, Zilong, Jinshan Tang, Ziming Wang, Kai Zhang, Ling Zhang, and Qingling Sun. 2018. “Deep Learning for Image-Based Cancer Detection and Diagnosis- a Survey.” *Pattern Recognition* 83: 134–49.

Ioannidis, Vassilis N., Antonio G. Marques, and Georgios B. Giannakis. 2019. “Graph Neural Networks for Predicting Protein Functions.” In *2019 IEEE 8th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, 221–25. https://doi.org/10.1109/CAMSAP45676.2019.9022646.

Jang, Soobeom, Seong-Eun Moon, and Jong-Seok Lee. 2018. “EEG-Based Video Identification Using Graph Signal Modeling and Graph Convolutional Neural Network.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 3066–70. IEEE.

Khemani, Bharti, Shruti Patil, Ketan Kotecha, and Sudeep Tanwar. 2024. “A Review of Graph Neural Networks: Concepts, Architectures, Techniques, Challenges, Datasets, Applications, and Future Directions.” *Journal of Big Data* 11 (1): 18.

Kingma, Diederik P, and Jimmy Ba. 2014. “Adam: A Method for Stochastic Optimization.” *arXiv Preprint arXiv:1412.6980*.

Kipf, Thomas N, and Max Welling. 2016. “Semi-Supervised Classification with Graph Convolutional Networks.” *arXiv Preprint arXiv:1609.02907*.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. “Gradient-Based Learning Applied to Document Recognition.” *Proceedings of the IEEE* 86 (11): 2278–2324.

Liu, Zhiyuan, and Jie Zhou. 2022. *Introduction to Graph Neural Networks*. Springer Nature.

Lu, Chengqian, Mengyun Yang, Feng Luo, Fang-Xiang Wu, Min Li, Yi Pan, Yaohang Li, and Jianxin Wang. 2018. "Prediction of lncRNA–Disease Associations Based on Inductive Matrix Completion." *Bioinformatics* 34 (19): 3357–64.

Lundberg, Scott M, and Su-In Lee. 2017. "A Unified Approach to Interpreting Model Predictions." *Advances in Neural Information Processing Systems* 30.

Mathur, Priyanka, and Vijay Kumar Chakka. 2020. "Graph Signal Processing of EEG Signals for Detection of Epilepsy." In *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*, 839–43. IEEE.

Mohammadi, Sina, and Mohamed Allali. 2024. "Advancing Brain Tumor Segmentation with Spectral–Spatial Graph Neural Networks." *Applied Sciences* 14 (8): 3424.

Momanyi, Biffon Manyura, Yu-Wei Zhou, Bakanina Kissanga Grace-Mercure, Sebu Aboma Temesgen, Ahmad Basharat, Lin Ning, Lixia Tang, Hui Gao, Hao Lin, and Hua Tang. 2024. "SAGESDA: Multi-GraphSAGE Networks for Predicting SnoRNA-Disease Associations." *Current Research in Structural Biology* 7: 100122. https://doi.org/https://doi.org/10.1016/j.crstbi.2023.100122.

Nagel, Sebastian. 2019. "Towards a Home-Use BCI: Fast Asynchronous Control and Robust Non-Control State Detection." PhD thesis. https://doi.org/10.15496/publikation-37739.

Nguyen, Thin, Hang Le, Thomas P Quinn, Tri Nguyen, Thuc Duy Le, and Svetha Venkatesh. 2021. "GraphDTA: Predicting Drug–Target Binding Affinity with Graph Neural Networks." *Bioinformatics* 37 (8): 1140–47.

Pan, Xiaoyong, and Hong-Bin Shen. 2019. "Inferring Disease-Associated microRNAs Using Semi-Supervised Multi-Label Graph Convolutional Networks." *Iscience* 20: 265–77.

Puthanveettil Madathil, Abhilash, Xichun Luo, Qi Liu, Charles Walker, Rajeshkumar Madarkar, Yukui Cai, Zhanqiang Liu, Wenlong Chang, and Yi Qin. 2024. "Intrinsic and Post-Hoc XAI Approaches for Fingerprint Identification and Response Prediction in Smart Manufacturing Processes." *Journal of Intelligent Manufacturing*, 1–22.

Rakhimberdina, Zarina, and Tsuyoshi Murata. 2020. "Linear Graph Convolutional Model for Diagnosing Brain Disorders." In *Complex Networks and Their Applications VIII: Volume 2 Proceedings of the Eighth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2019 8*, 815–26. Springer.

Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. 2016. "" Why Should i Trust You?" Explaining the Predictions of Any Classifier." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–44.

Robinson, Peter K. 2015. "Enzymes: Principles and Biotechnological Applications." *Essays in Biochemistry* 59: 1.

Scarselli, Franco, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. "The Graph Neural Network Model." *IEEE Transactions on Neural Networks* 20 (1): 61–80.

Singh, Ramandeep, Mannudeep K Kalra, Chayanin Nitiwarangkul, John A Patti, Fatemeh

Homayounieh, Atul Padole, Pooja Rao, et al. 2018. "Deep Learning in Chest Radiography: Detection of Findings and Presence of Change." *PloS One* 13 (10): e0204155.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *The Journal of Machine Learning Research* 15 (1): 1929–58.

Velickovic, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. "Graph Attention Networks." *Stat* 1050 (20): 10–48550.

Vilone, Giulia, and Luca Longo. 2021. "Classification of Explainable Artificial Intelligence Methods Through Their Output Formats." *Machine Learning and Knowledge Extraction* 3 (3): 615–61.

Wang, Hongmei, Wei Zhao, Zhenqi Li, Dongya Jia, Cong Yan, Jing Hu, Jiansheng Fang, and Ming Yang. 2020. "A Weighted Graph Attention Network Based Method for Multi-Label Classification of Electrocardiogram Abnormalities." In *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, 418–21. IEEE.

Wang, Xiang, Yingxin Wu, An Zhang, Xiangnan He, and Tat-seng Chua. 2020. "Causal Screening to Interpret Graph Neural Networks.(2020)." In *URL Https://Openreview. Net/Forum*.

Wang, Yuyang, Zijie Li, and Amir Barati Farimani. 2023. "Graph Neural Networks for Molecules." In *Machine Learning in Molecular Sciences*, 21–66. Springer.

Wieder, Oliver, Stefan Kohlbacher, Mélaine Kuenemann, Arthur Garon, Pierre Ducrot, Thomas Seidel, and Thierry Langer. 2020. "A Compact Review of Molecular Property Prediction with Graph Neural Networks." *Drug Discovery Today: Technologies* 37: 1–12. https://doi.org/https://doi.org/10.1016/j.ddtec.2020.11.009.

Xu, Keyulu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. "How Powerful Are Graph Neural Networks?" *arXiv Preprint arXiv:1810.00826*.

Yang, Jing, and Xiujuan Lei. 2021. "Predicting circRNA-Disease Associations Based on Autoencoder and Graph Embedding." *Information Sciences* 571: 323–36. https://doi.org/https://doi.org/10.1016/j.ins.2021.04.073.

Ying, Zhitao, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. "Gnnexplainer: Generating Explanations for Graph Neural Networks." *Advances in Neural Information Processing Systems* 32.

Yuan, Hao, Jiliang Tang, Xia Hu, and Shuiwang Ji. 2020. "Xgnn: Towards Model-Level Explanations of Graph Neural Networks." In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 430–38.

Yuan, Hao, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2022. "Explainability in Graph Neural Networks: A Taxonomic Survey." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45 (5): 5782–99.

Zhang, Xiao-Meng, Li Liang, Lin Liu, and Ming-Jing Tang. 2021. "Graph Neural Networks and Their Current Applications in Bioinformatics." *Frontiers in Genetics* 12: 690049.

Zhao, Tianyi, Yang Hu, Linda R Valsdottir, Tianyi Zang, and Jiajie Peng. 2021. "Identifying

Drug–Target Interactions Based on Graph Convolutional Network and Deep Neural Network." *Briefings in Bioinformatics* 22 (2): 2141–50.

Zhou, Jie, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. "Graph Neural Networks: A Review of Methods and Applications." *AI Open* 1: 57–81.

Zitnik, Marinka, Monica Agrawal, and Jure Leskovec. 2018. "Modeling Polypharmacy Side Effects with Graph Convolutional Networks." *Bioinformatics* 34 (13): i457–66.

Zitnik, Marinka, and Jure Leskovec. 2017. "Predicting Multicellular Function Through Multi-Layer Tissue Networks." *Bioinformatics* 33 (14): i190–98.

# 8 Appendix A: Supplementary Figures

## Task 1: Protein Classification

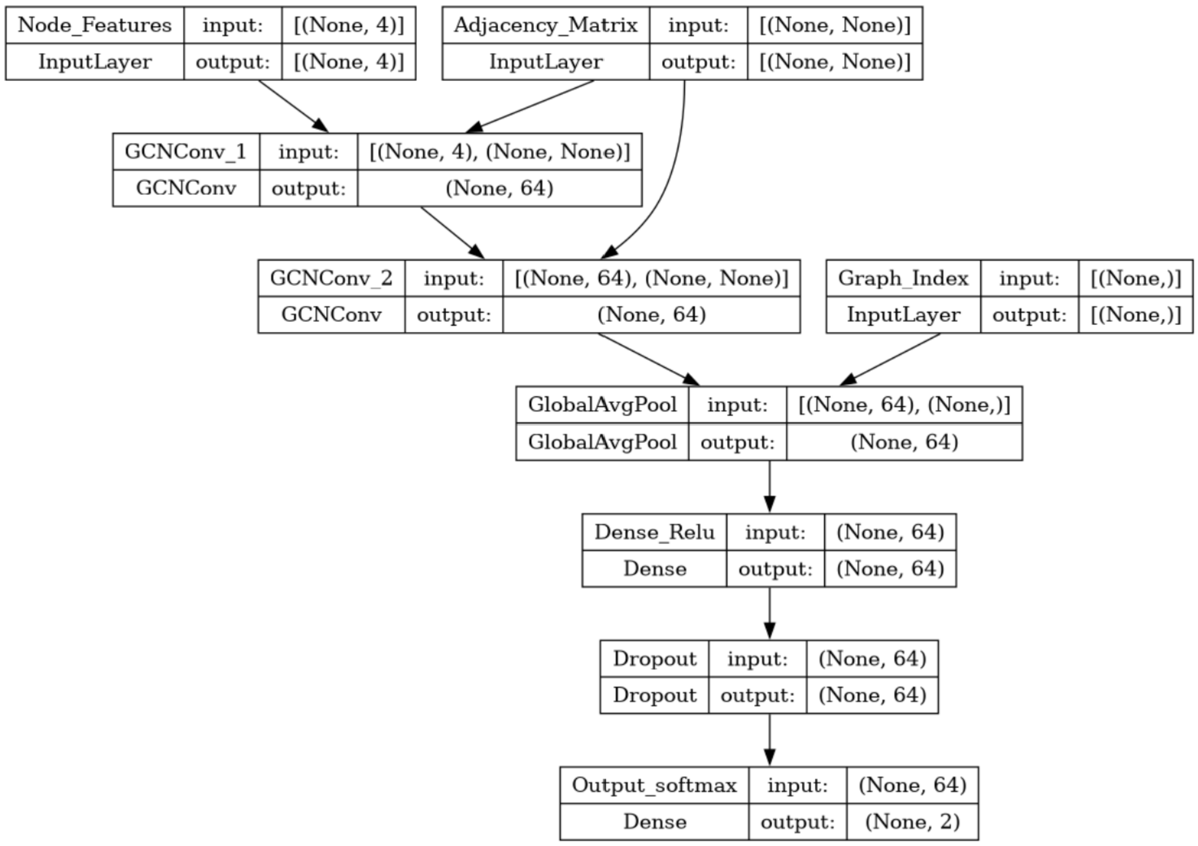
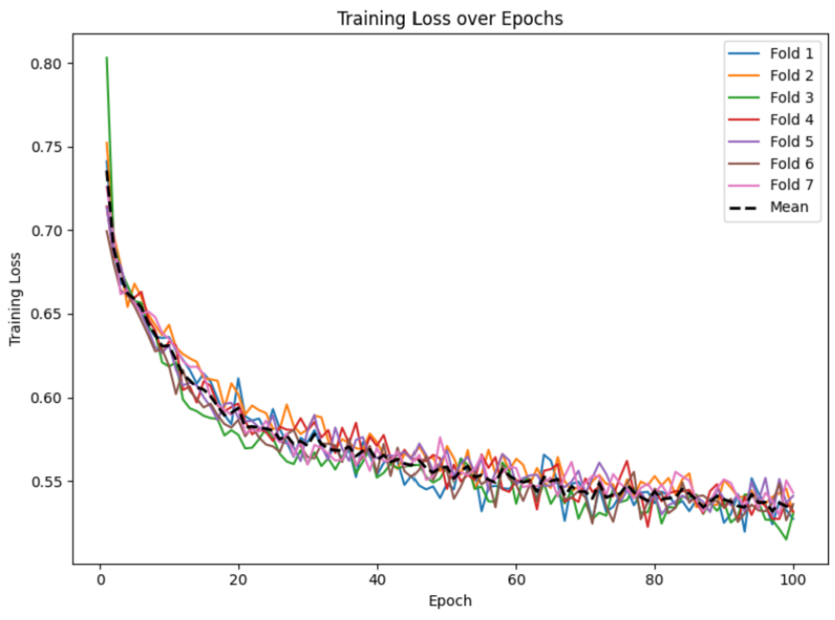

Node_Features | input: | [(None, 4)]
InputLayer | output: | [(None, 4)]
Adjacency_Matrix | input: | [(None, None)]
InputLayer | output: | [(None, None)]
GCNConv_1 | input: | [(None, 4), (None, None)]
GCNConv | output: | (None, 64)
GCNConv_2 | input: | [(None, 64), (None, None)]
GCNConv | output: | (None, 64)
Graph_Index | input: | [(None,)]
InputLayer | output: | [(None,)]
GlobalAvgPool | input: | [(None, 64), (None,)]
GlobalAvgPool | output: | (None, 64)
Dense_Relu | input: | (None, 64)
Dense | output: | (None, 64)
Dropout | input: | (None, 64)
Dropout | output: | (None, 64)
Output_softmax | input: | (None, 64)
Dense | output: | (None, 2)

Figure 8.1: GCN Model Architecture



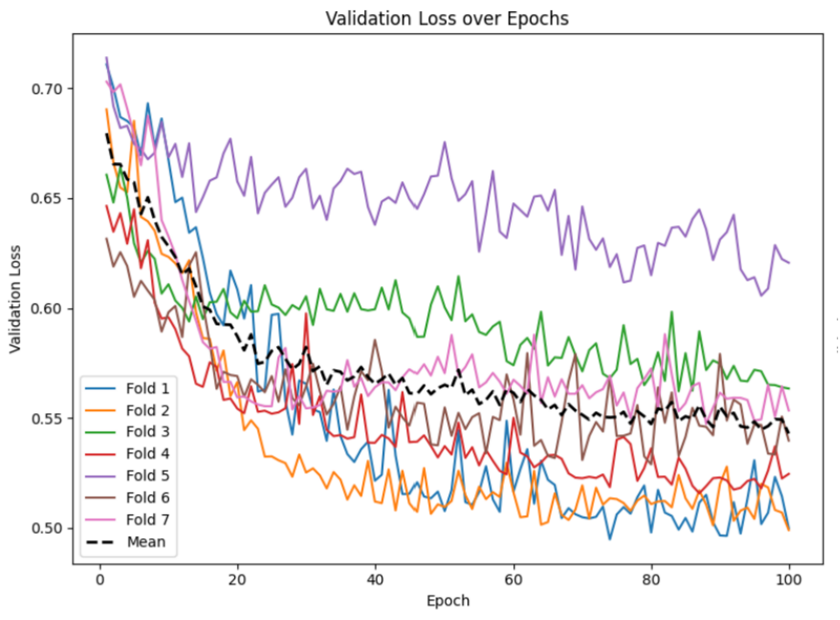
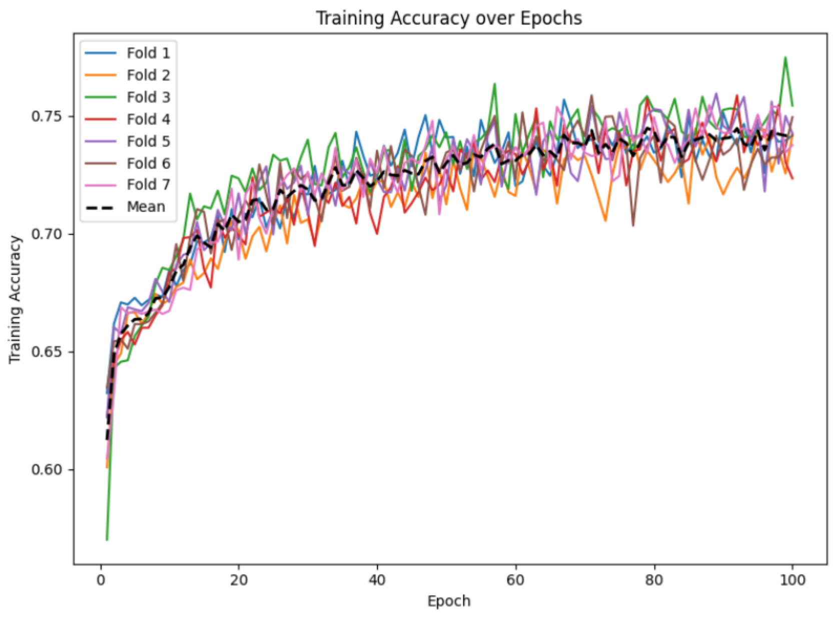


Figure 8.2: Training and Validation Loss over Epochs for Different Folds (GCN Model)



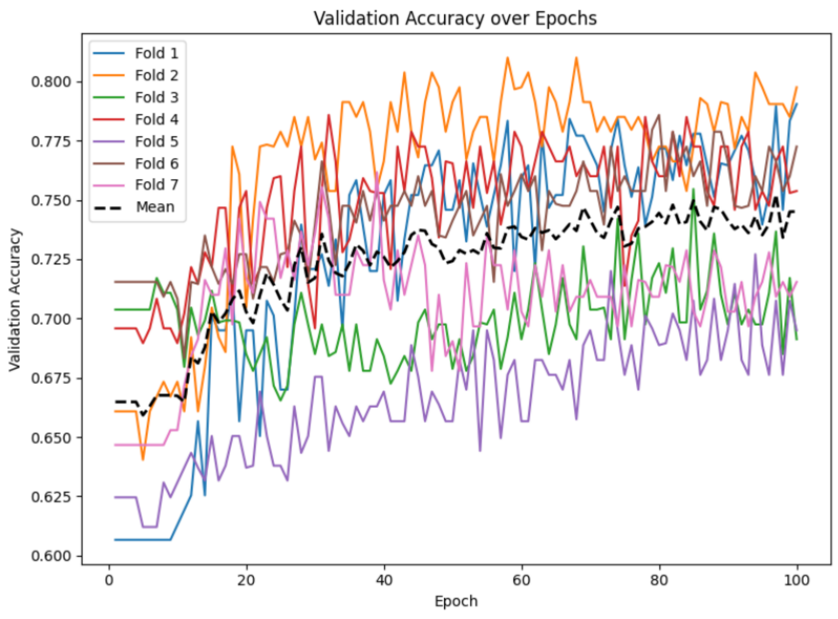


Figure 8.3: Training and Validation Accuracy over Epochs for Different Folds (GCN Model)

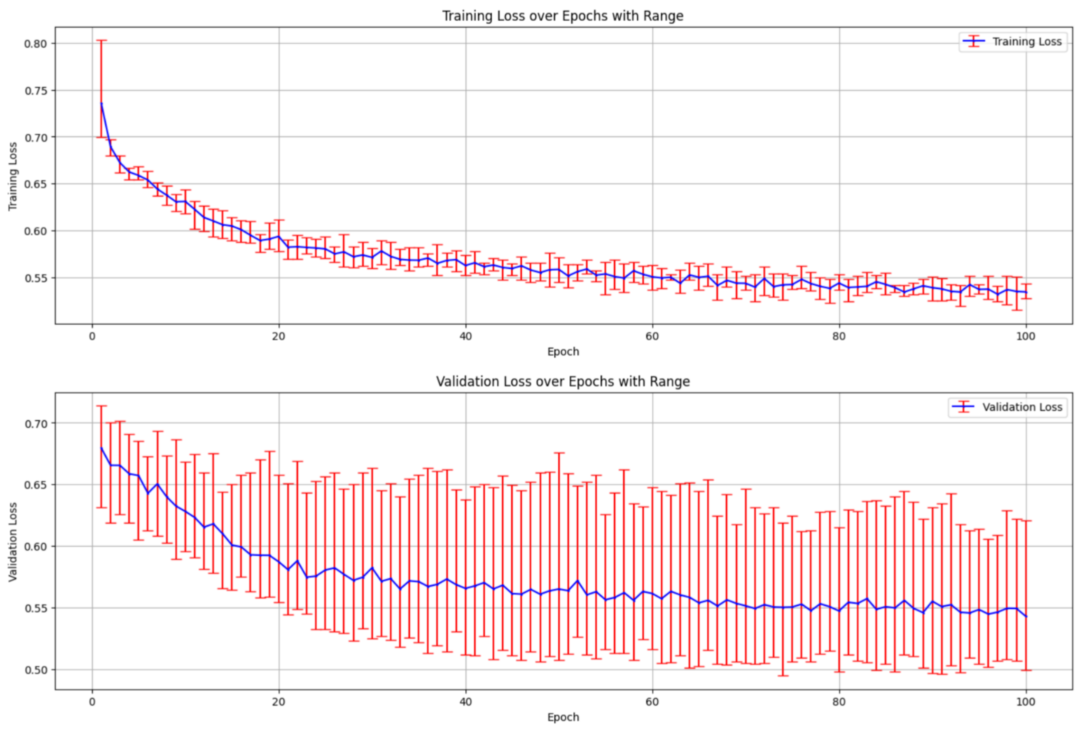


Figure 8.4: Training and Validation Loss over Epochs with Range (GCN Model)

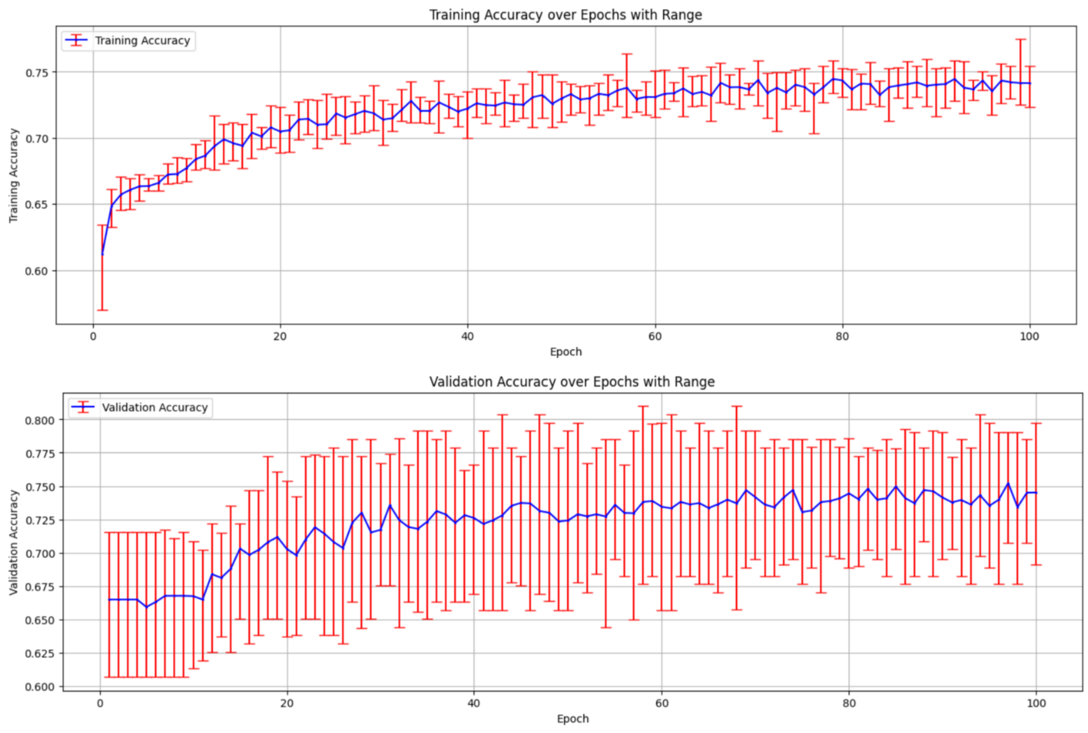
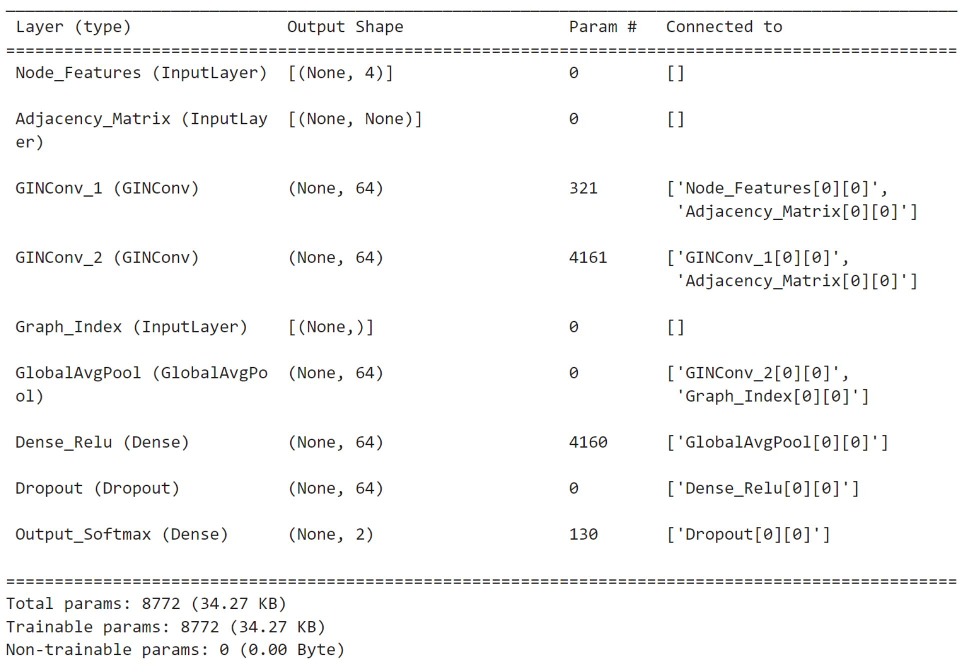
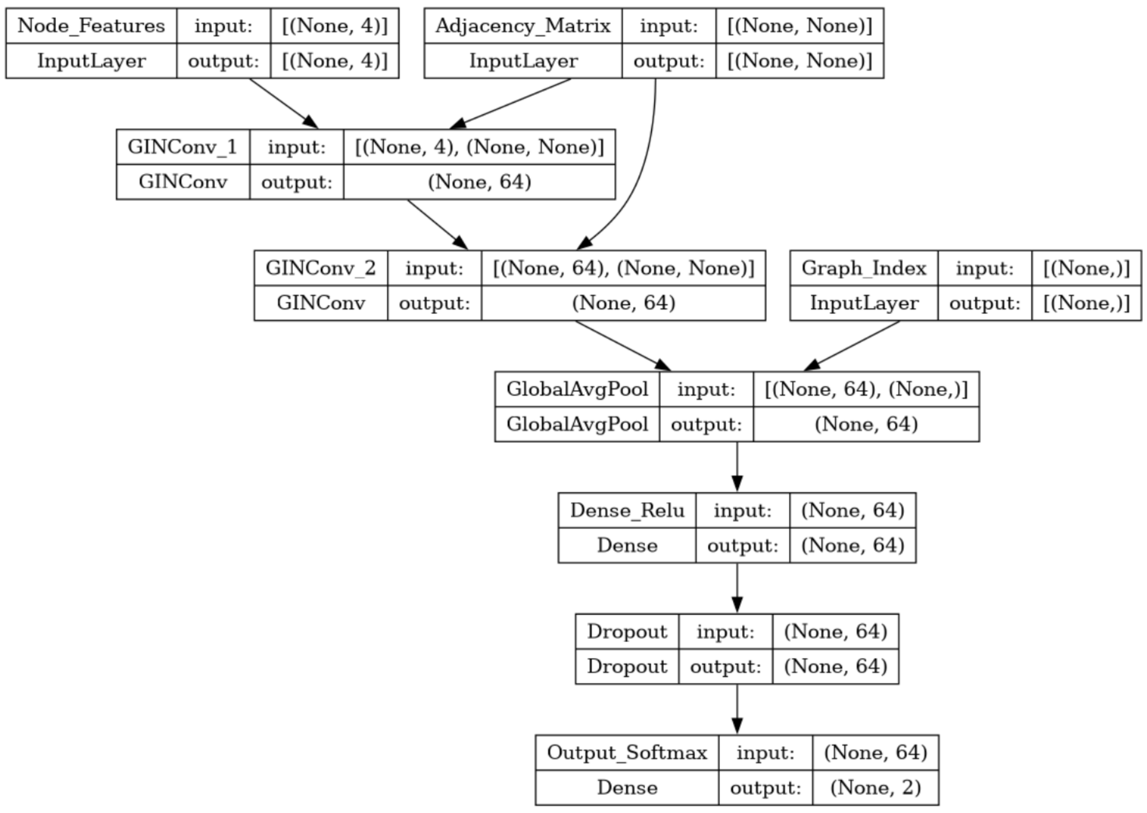


Figure 8.5: Training and Validation Accuracy over Epochs with Range (GCN Model)



```
_____________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
=============================================================================================
 Node_Features (InputLayer)  [(None, 4)]                  0         []

 Adjacency_Matrix (InputLay  [(None, None)]               0         []
 er)

 GINConv_1 (GINConv)         (None, 64)                   321       ['Node_Features[0][0]',
                                                                     'Adjacency_Matrix[0][0]']

 GINConv_2 (GINConv)         (None, 64)                   4161      ['GINConv_1[0][0]',
                                                                     'Adjacency_Matrix[0][0]']

 Graph_Index (InputLayer)    [(None,)]                    0         []

 GlobalAvgPool (GlobalAvgPo  (None, 64)                   0         ['GINConv_2[0][0]',
 ol)                                                                 'Graph_Index[0][0]']

 Dense_Relu (Dense)          (None, 64)                   4160      ['GlobalAvgPool[0][0]']

 Dropout (Dropout)           (None, 64)                   0         ['Dense_Relu[0][0]']

 Output_Softmax (Dense)      (None, 2)                    130       ['Dropout[0][0]']

=============================================================================================
Total params: 8772 (34.27 KB)
Trainable params: 8772 (34.27 KB)
Non-trainable params: 0 (0.00 Byte)
```

Figure 8.6: GIN Model Summary



Node_Features | input: | [(None, 4)]
InputLayer | output: | [(None, 4)]
Adjacency_Matrix | input: | [(None, None)]
InputLayer | output: | [(None, None)]
GINConv_1 | input: | [(None, 4), (None, None)]
GINConv | output: | (None, 64)
GINConv_2 | input: | [(None, 64), (None, None)]
GINConv | output: | (None, 64)
Graph_Index | input: | [(None,)]
InputLayer | output: | [(None,)]
GlobalAvgPool | input: | [(None, 64), (None,)]
GlobalAvgPool | output: | (None, 64)
Dense_Relu | input: | (None, 64)
Dense | output: | (None, 64)
Dropout | input: | (None, 64)
Dropout | output: | (None, 64)
Output_Softmax | input: | (None, 64)
Dense | output: | (None, 2)

Figure 8.7: GIN Model Architecture

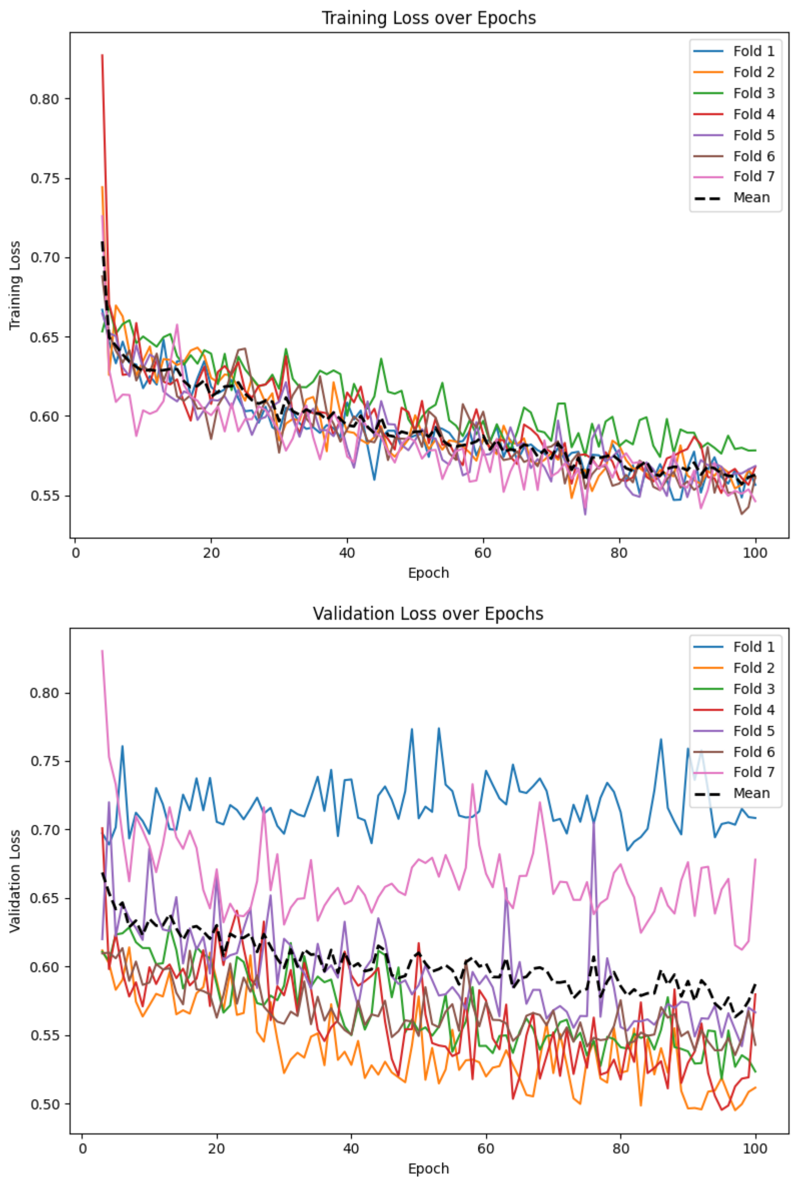
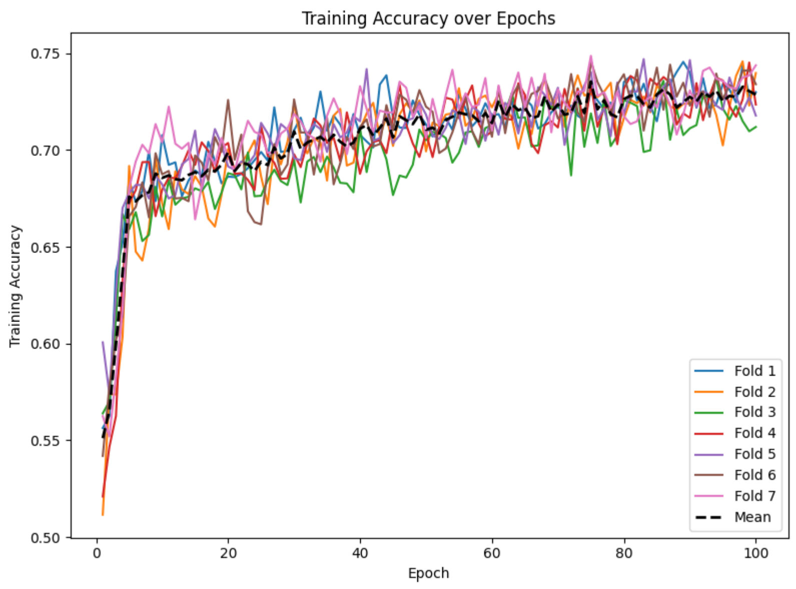


Figure 8.8: Training and Validation Loss over Epochs for Different Folds (GIN Model)



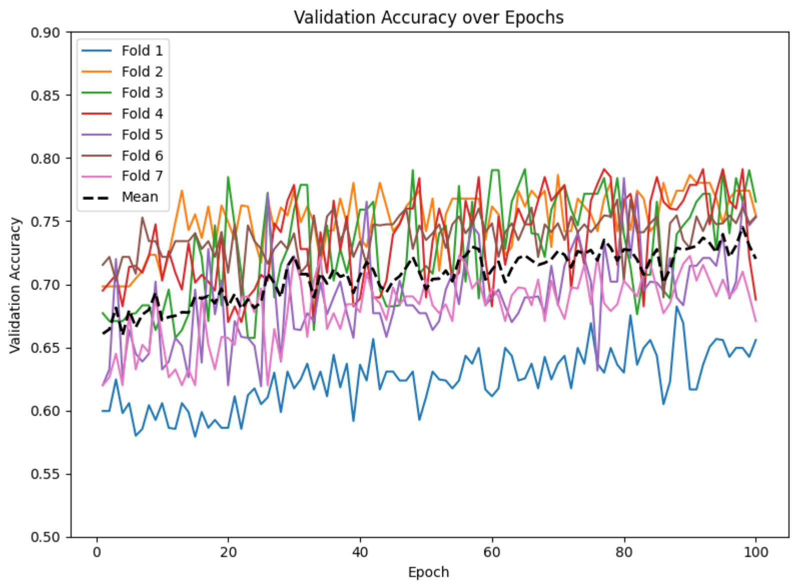


Figure 8.9: Training and Validation Accuracy over Epochs for Different Folds (GIN Model)

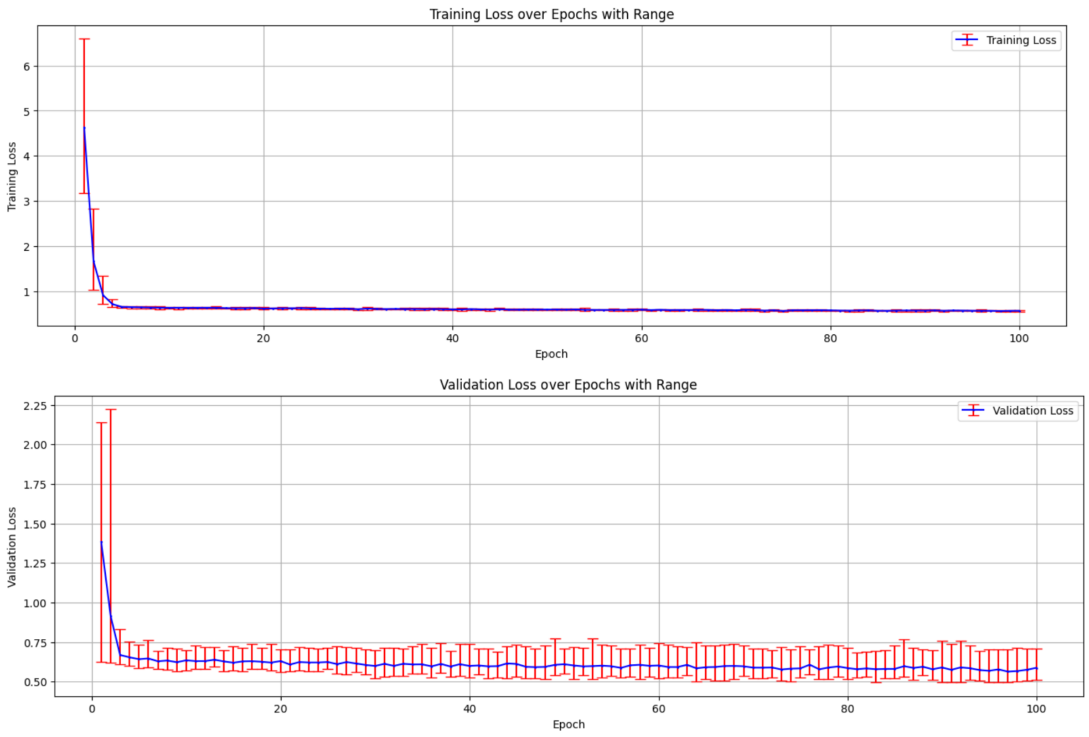


Figure 8.10: Training and Validation Loss over Epochs with Range (GIN Model)

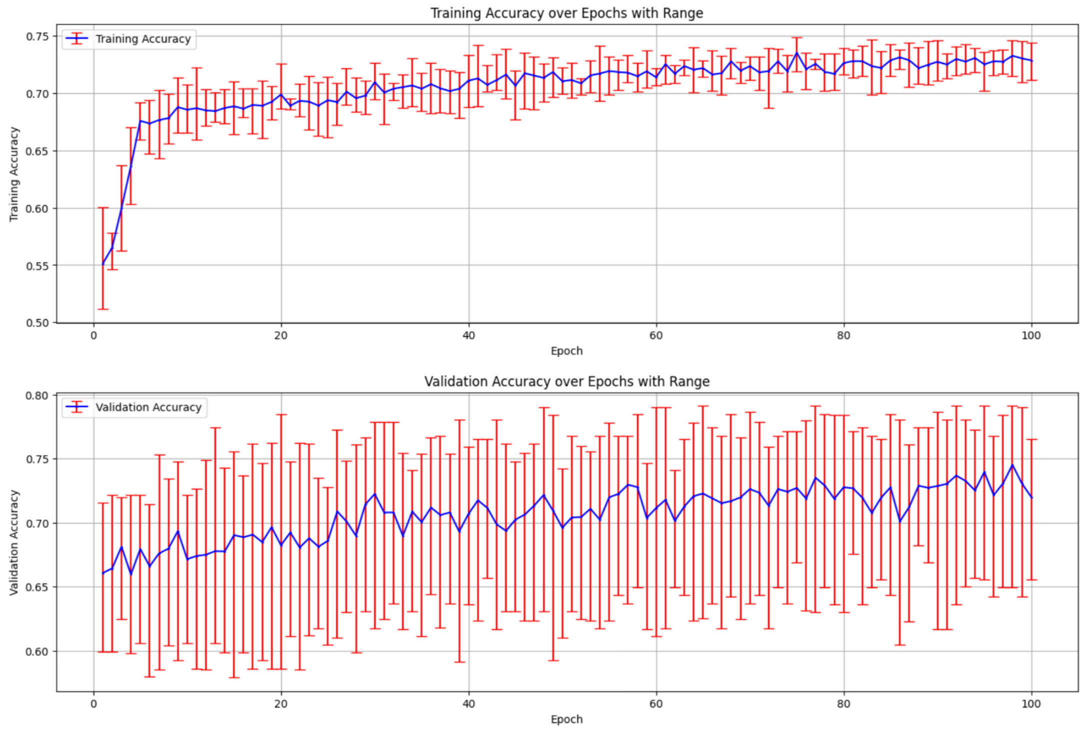
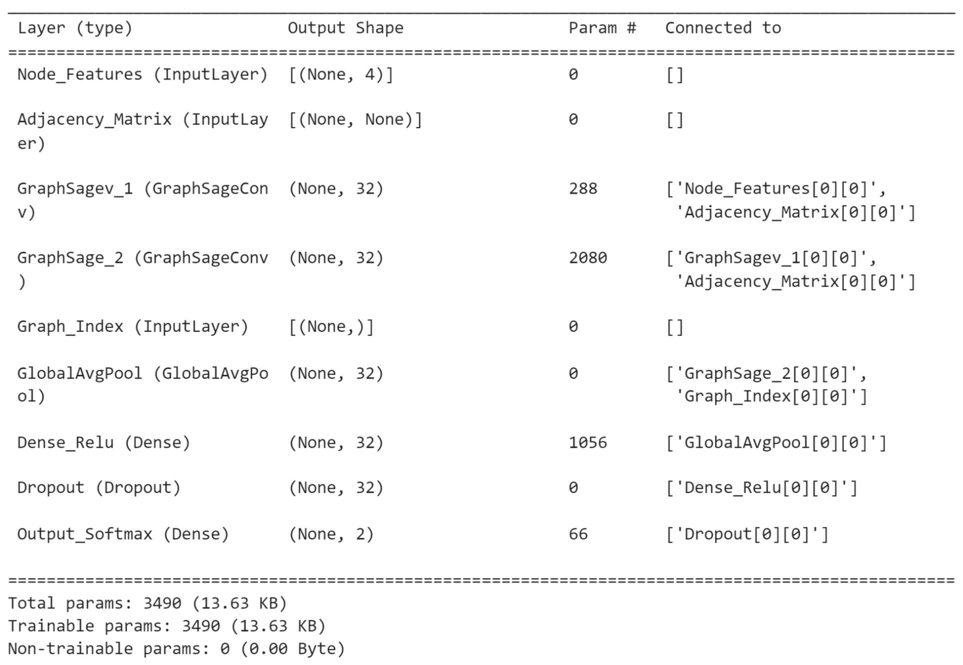
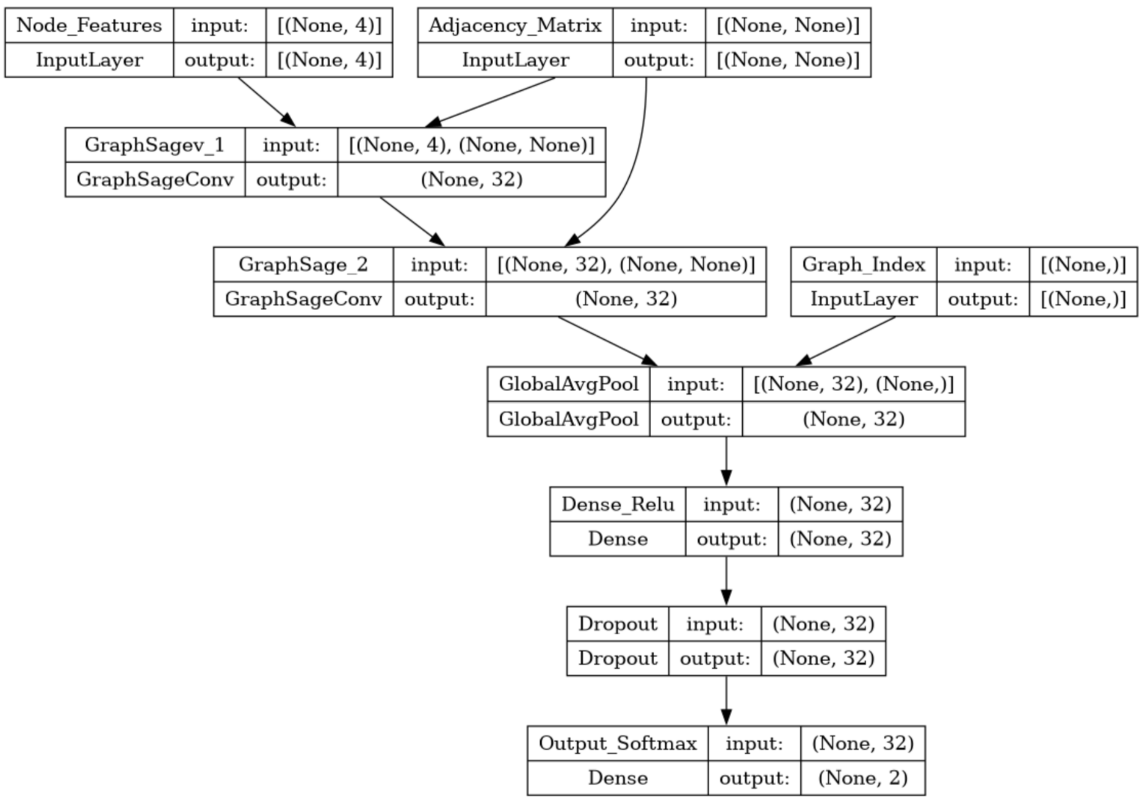


Figure 8.11: Training and Validation Accuracy over Epochs with Range (GIN Model)



```
_________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
=================================================================================================
 Node_Features (InputLayer)  [(None, 4)]                  0         []

 Adjacency_Matrix (InputLay  [(None, None)]               0         []
 er)

 GraphSagev_1 (GraphSageCon  (None, 32)                   288       ['Node_Features[0][0]',
 v)                                                                  'Adjacency_Matrix[0][0]']

 GraphSage_2 (GraphSageConv  (None, 32)                   2080      ['GraphSagev_1[0][0]',
 )                                                                   'Adjacency_Matrix[0][0]']

 Graph_Index (InputLayer)    [(None,)]                    0         []

 GlobalAvgPool (GlobalAvgPo  (None, 32)                   0         ['GraphSage_2[0][0]',
 ol)                                                                 'Graph_Index[0][0]']

 Dense_Relu (Dense)          (None, 32)                   1056      ['GlobalAvgPool[0][0]']

 Dropout (Dropout)           (None, 32)                   0         ['Dense_Relu[0][0]']

 Output_Softmax (Dense)      (None, 2)                    66        ['Dropout[0][0]']

=================================================================================================
Total params: 3490 (13.63 KB)
Trainable params: 3490 (13.63 KB)
Non-trainable params: 0 (0.00 Byte)
```

Figure 8.12: GraphSage Model Summary



Node_Features | input: | [(None, 4)]
InputLayer | output: | [(None, 4)]
Adjacency_Matrix | input: | [(None, None)]
InputLayer | output: | [(None, None)]
GraphSagev_1 | input: | [(None, 4), (None, None)]
GraphSageConv | output: | (None, 32)
GraphSage_2 | input: | [(None, 32), (None, None)]
GraphSageConv | output: | (None, 32)
Graph_Index | input: | [(None,)]
InputLayer | output: | [(None,)]
GlobalAvgPool | input: | [(None, 32), (None,)]
GlobalAvgPool | output: | (None, 32)
Dense_Relu | input: | (None, 32)
Dense | output: | (None, 32)
Dropout | input: | (None, 32)
Dropout | output: | (None, 32)
Output_Softmax | input: | (None, 32)
Dense | output: | (None, 2)

Figure 8.13: GraphSage Model Architecture

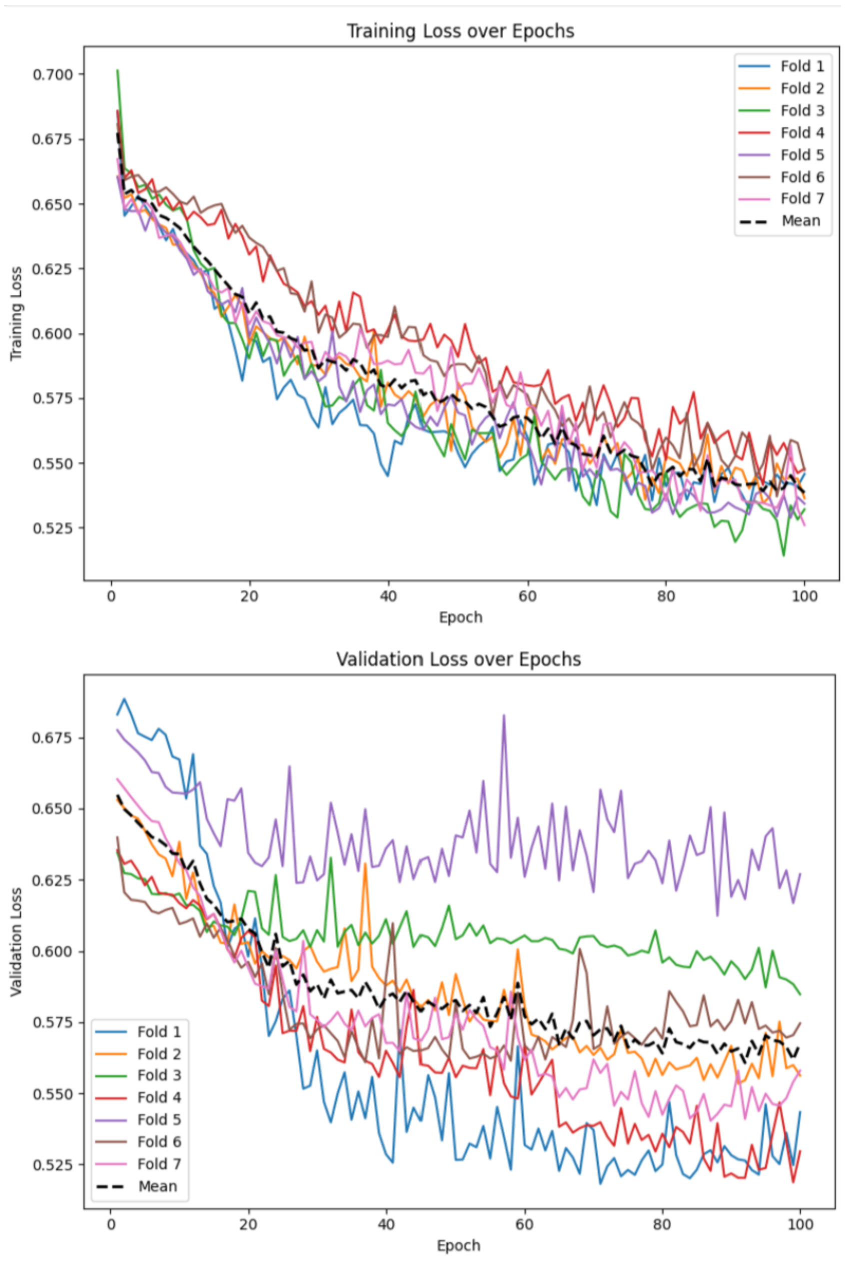
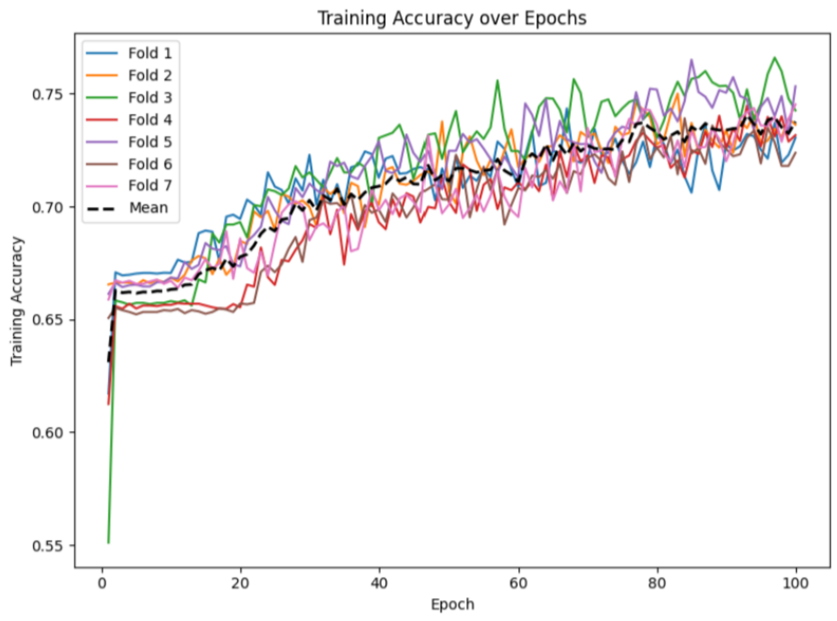


Figure 8.14: Training and Validation Loss over Epochs for Different Folds (GraphSage Model)



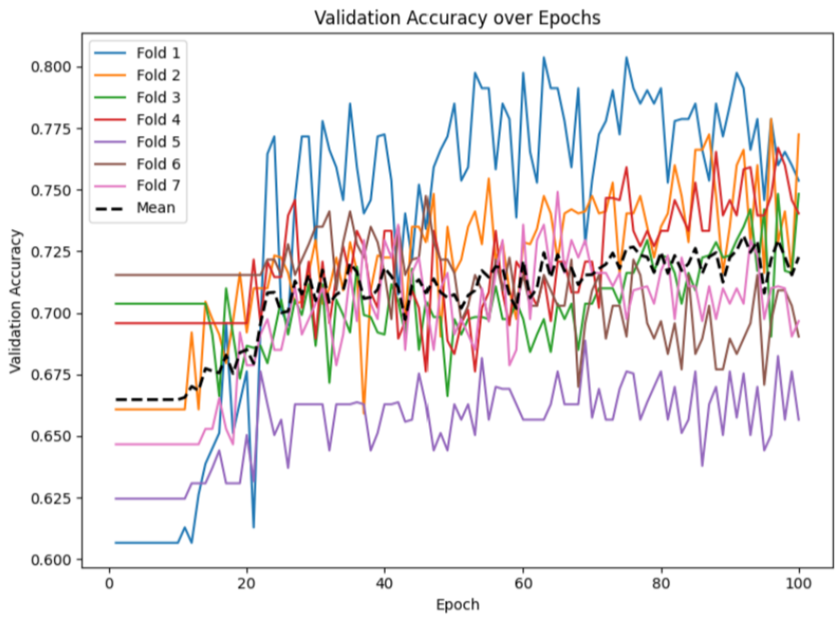


Figure 8.15: Training and Validation Accuracy over Epochs for Different Folds (GraphSage Model)

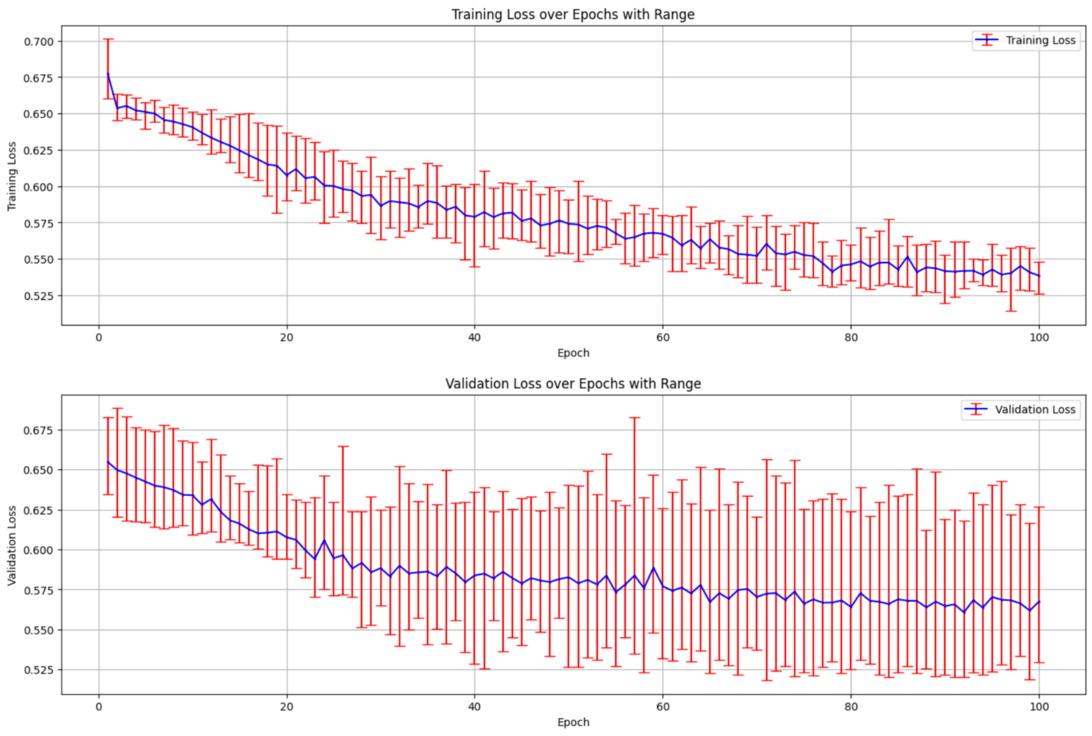


Figure 8.16: Training and Validation Loss over Epochs with Range (GraphSage Model)

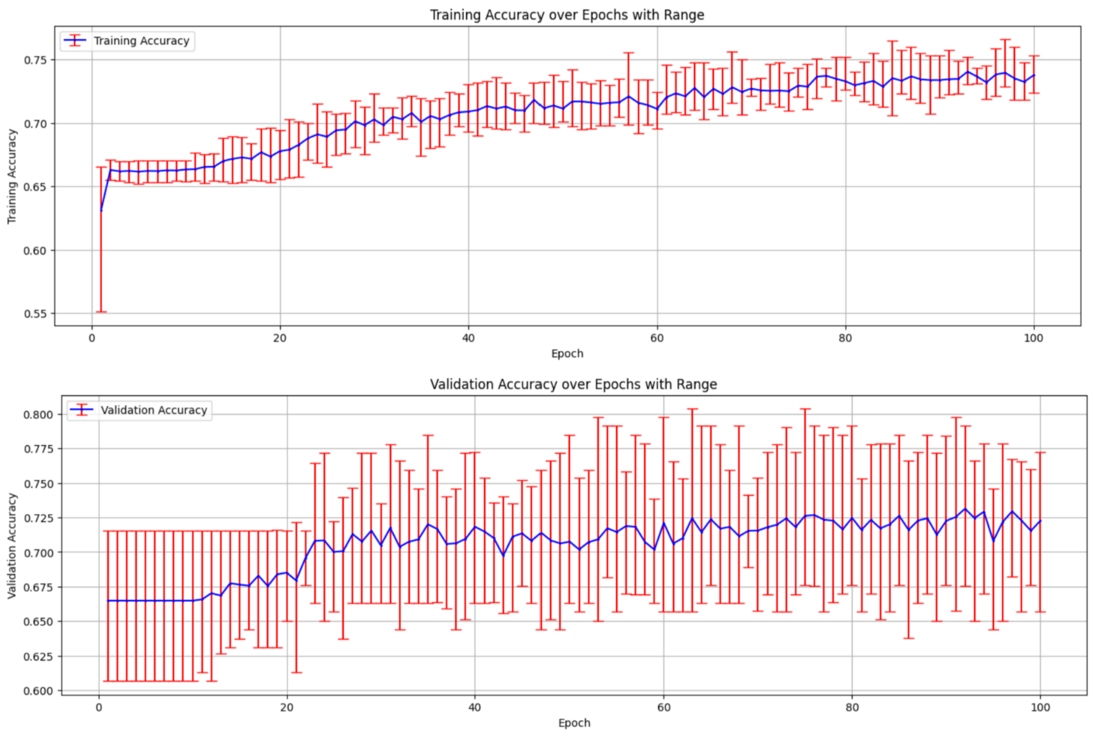


Figure 8.17: Training and Validation Accuracy over Epochs with Range (GraphSage Model)

# 9 Appendix B: Source Code

## 9.1 Code for Task 1: Protein Classification

```
# Packages and libraries
!pip install spektral
!pip install pydot graphviz

import pydot
import numpy as np
import tensorflow as tf
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import random
import scipy.sparse as sp

from tensorflow.keras.utils import plot_model
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.metrics import categorical_accuracy
from tensorflow.keras.optimizers import Adam

from spektral.layers import GCNConv, GlobalSumPool, GINConv,
                            GraphSageConv, GlobalAvgPool
from spektral.datasets import TUDataset
from spektral.transforms import GCNFilter, AdjToSpTensor
from spektral.transforms.normalize_adj import NormalizeAdj
from spektral.data import DisjointLoader
from spektral.models import GeneralGNN, GNNExplainer

from sklearn.preprocessing import StandardScaler

##########################################################################
# Load data and data description
##########################################################################
```

```python
dataset = TUDataset("PROTEINS")

# number of graphs
print(len(dataset))

# Example: the first graph
graph = dataset[0]

# Node features, adj matrix, label
x = graph.x
a = graph.a
y = graph.y

print("Node features (x):\n", x)
print("\nAdjacency matrix (a):\n", a)
print("\nLabel (y):\n", y)

# Plot the first graph
# Check if the adjacency matrix is a scipy sparse matrix
if sp.issparse(graph.a):
    adj_matrix_dense = graph.a.todense()
else:
    adj_matrix_dense = graph.a

G = nx.from_numpy_array(adj_matrix_dense)
plt.figure(figsize=(12, 8))
nx.draw(G,
       with_labels=True,
       node_color='skyblue',
       edge_color='gray',
       node_size=300,
       font_size=8,
       width=0.5)
plt.title("Graph Visualization")
plt.show()

##########################################################################
# GCN MODEL
```

```
###########################################################################

# Config ##################################################################
learning_rate = 0.001  # Learning rate
channels = 64  # Hidden units
layers = 2  # GCN layers
initial_epochs = 100  # Initial number of training epochs for cross-validation
batch_size = 32  # Batch size
k_folds = 7  # Number of folds for cross-validation
drop_rate = 0.3 # Dropout

np.random.seed(123)
tf.random.set_seed(123)
random.seed(123)

# Data preprocessing #####################################################
dataset = TUDataset("PROTEINS")
dataset.apply(GCNFilter())

# Train/test split
idxs = np.random.permutation(len(dataset))
split = int(0.9 * len(dataset))
idx_tr, idx_te = np.split(idxs, [split])
dataset_tr, dataset_te = dataset[idx_tr], dataset[idx_te]

# Build model ############################################################

# Parameters
N = max(g.n_nodes for g in dataset)
F = dataset.n_node_features  # Dimension of node features
n_out = dataset.n_labels  # Dimension of the target

# Build the model
def build_model():
    # Define inputs
    x_in = Input(shape=(F,), name="Node_Features")
    a_in = Input(shape=(None,), sparse=True, name="Adjacency_Matrix")
    i_in = Input(shape=(), dtype=tf.int32, name="Graph_Index")
    # Define the GIN model using the functional API
    x = GCNConv(channels,
                activation="relu",
                name="GCNConv_1")([x_in, a_in])

    for i in range(1, layers):
        x = GCNConv(channels,
                    activation="relu",
                    name=f"GCNConv_{i+1}")([x, a_in])

    x = GlobalAvgPool(name="GlobalAvgPool")([x, i_in])
    x = Dense(channels,
              activation="relu",
              kernel_regularizer=tf.keras.regularizers.l2(5e-4),
              name="Dense_Relu")(x)
    x = Dropout(drop_rate,name="Dropout")(x)
    output = Dense(n_out,
                   activation="softmax",
                   name="Output_softmax")(x)

    model = Model(inputs=[x_in, a_in, i_in], outputs=output)
    return model

# Build and plot the model
model = build_model()
model.summary()
plot_model(model,
           to_file='GCN_model.png',
           show_shapes=True,
           show_layer_names=True)

# Cross-validation #######################################################
idxs = np.random.permutation(len(dataset_tr))
split_size = len(dataset_tr) // k_folds

# Variables to store training results
all_train_losses = np.zeros((k_folds, initial_epochs))
all_train_accs = np.zeros((k_folds, initial_epochs))
```

```python
all_val_losses = np.zeros((k_folds, initial_epochs))
all_val_accs = np.zeros((k_folds, initial_epochs))

for fold in range(k_folds):
    print(f"Fold {fold+1}/{k_folds}")
    start, end = fold * split_size, (fold + 1) * split_size
    idx_va = idxs[start:end]
    idx_tr_fold = np.concatenate([idxs[:start], idxs[end:]])

    dataset_tr_fold, dataset_va_fold = dataset[idx_tr_fold], dataset[idx_va]

    loader_tr = DisjointLoader(dataset_tr_fold,
                               batch_size=batch_size,
                               epochs=initial_epochs,
                               shuffle=True)
    loader_va = DisjointLoader(dataset_va_fold,
                                batch_size=batch_size,
                                epochs=1,
                                shuffle=False)

    # Check if loaders are not empty
    assert len(dataset_tr_fold) > 0, "Training dataset is empty!"
    assert len(dataset_va_fold) > 0, "Validation dataset is empty!"

    model = build_model()
    optimizer = Adam(learning_rate)
    loss_fn = CategoricalCrossentropy()

    # Fit model ##########################################################
    @tf.function(input_signature=loader_tr.tf_signature(),
                 experimental_relax_shapes=True)
    def train_step(inputs, target):
        with tf.GradientTape() as tape:
            predictions = model(inputs, training=True)
            loss = loss_fn(target, predictions) + sum(model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        acc = tf.reduce_mean(categorical_accuracy(target, predictions))
        return loss, acc, gradients

    @tf.function(input_signature=loader_va.tf_signature(),
                 experimental_relax_shapes=True)
    def val_step(inputs, target):
        predictions = model(inputs, training=False)
        loss = loss_fn(target, predictions) + sum(model.losses)
        acc = tf.reduce_mean(categorical_accuracy(target, predictions))
        return loss, acc

    epoch = step = 0
    results = []
    val_results = []

    for batch in loader_tr:
        step += 1
        loss, acc, gradients = train_step(*batch)

        results.append((loss, acc))

        if step == loader_tr.steps_per_epoch:
            step = 0
            epoch += 1

            train_loss, train_acc = np.mean(results, 0)
            all_train_losses[fold, epoch-1] = train_loss
            all_train_accs[fold, epoch-1] = train_acc

            # Reinitialize the validation loader for each epoch
            # (without shuffle)
            loader_va = DisjointLoader(dataset_va_fold,
                                      batch_size=batch_size,
                                      epochs=1,
                                      shuffle=False)
            val_results = []
            for batch in loader_va:
                val_loss, val_acc = val_step(*batch)
                val_results.append((val_loss, val_acc))
            if val_results:
                val_loss, val_acc = np.mean(val_results, 0)
                all_val_losses[fold, epoch-1] = val_loss
                all_val_accs[fold, epoch-1] = val_acc
                # print(f"Ep. {epoch} - Loss: {train_loss:.4f}.
                # Acc: {train_acc:.4f}. Validation Loss: {val_loss:.4f}.
                # Validation Acc: {val_acc:.4f}")
            else:
                # print(f"Ep. {epoch} - Loss: {train_loss:.4f}.
                # Acc: {train_acc:.4f}. No validation data available.")

            results = []

# Plot training losses and accuracies
epochs = np.arange(1, initial_epochs + 1)

plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
for fold in range(k_folds):
    plt.plot(epochs,
             all_train_losses[fold],
             label=f'Fold {fold+1}')
mean_train_losses = np.mean(all_train_losses, axis=0)
plt.plot(epochs,
         mean_train_losses,
         label='Mean',
         color='black',
         linewidth=2,
         linestyle='--')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss over Epochs')
plt.legend()

plt.subplot(1, 2, 2)
for fold in range(k_folds):
    plt.plot(epochs, all_train_accs[fold], label=f'Fold {fold+1}')
mean_train_accs = np.mean(all_train_accs, axis=0)
```

```
plt.plot(epochs,
         mean_train_accs,
         label='Mean',
         color='black',
         linewidth=2,
         linestyle='--')
plt.xlabel('Epoch')
plt.ylabel('Training Accuracy')
plt.title('Training Accuracy over Epochs')
plt.legend()

plt.tight_layout()
plt.savefig('training_curves.png')
plt.show()

# Plot validation losses and accuracies
plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
for fold in range(k_folds):
    plt.plot(epochs, all_val_losses[fold], label=f'Fold {fold+1}')
mean_val_losses = np.mean(all_val_losses, axis=0)
plt.plot(epochs,
         mean_val_losses,
         label='Mean',
         color='black',
         linewidth=2,
         linestyle='--')
plt.xlabel('Epoch')
plt.ylabel('Validation Loss')
plt.title('Validation Loss over Epochs')
plt.legend()

plt.subplot(1, 2, 2)
for fold in range(k_folds):
    plt.plot(epochs, all_val_accs[fold], label=f'Fold {fold+1}')
mean_val_accs = np.mean(all_val_accs, axis=0)
plt.plot(epochs,
         mean_val_accs,
         label='Mean',
         color='black',
         linewidth=2,
         linestyle='--')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.title('Validation Accuracy over Epochs')
plt.legend()

plt.tight_layout()
plt.savefig('validation_curves.png')
plt.show()

def plot_with_error_bars(epochs,
                         data,
                         title,
                         y_label,
                         color='b',
                         ecolor='r',
                         figsize=(17, 5)):
    """
    Plots mean with range as error bars for given data across epochs.

    Parameters:
    - epochs: Array of epoch indices.
    - data: Multidimensional array where each row is data
            from a different fold.
    - title: Title for the plot.
    - y_label: Label for the Y-axis.
    - color: Line and marker color.
    - ecolor: Error bar color.
    - figsize: Size of the figure.
    """
    min_values = np.min(data, axis=0)
    max_values = np.max(data, axis=0)
    mean_values = np.mean(data, axis=0)

    plt.figure(figsize=figsize)
    plt.errorbar(epochs, mean_values,
                 [mean_values - min_values, max_values - mean_values],
                 fmt='o', linestyle='-', color='b', ecolor="r",
                 capsize=5, markersize=1, label=y_label)

    plt.xlabel('Epoch')
    plt.ylabel(y_label)
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()

epochs = np.arange(1, initial_epochs + 1)

# Plot mean and error
plot_with_error_bars(epochs,
                     all_train_losses,
                     'Training Loss over Epochs with Range',
                     'Training Loss')
plot_with_error_bars(epochs,
                     all_val_losses,
                     'Validation Loss over Epochs with Range',
                     'Validation Loss')
plot_with_error_bars(epochs,
                     all_train_accs,
                     'Training Accuracy over Epochs with Range',
                     'Training Accuracy')
plot_with_error_bars(epochs,
                     all_val_accs,
                     'Validation Accuracy over Epochs with Range',
                     'Validation Accuracy')

# best epoch
best_epoch = np.argmax(mean_val_accs) + 1
# Add 1 because epoch indexing starts from 1
best_val_loss = mean_val_losses[best_epoch - 1]
best_val_acc = mean_val_accs[best_epoch - 1]
```

```
# Calculate the standard deviation of validation loss and accuracy
# for the best epoch
std_val_loss = np.std(all_val_losses[:, best_epoch - 1])
std_val_acc = np.std(all_val_accs[:, best_epoch - 1])

# print(f"Best epoch: {best_epoch}, Best validation loss: {best_val_loss:.4f}
# (SD: {std_val_loss:.4f}), Best validation accuracy: {best_val_acc:.4f}
# (SD: {std_val_acc:.4f})")

# Re-train the model with the best parameters on the full training set
print("Re-training model on the full training set with the best parameters")

# Use the entire training dataset for training
loader_full_tr = DisjointLoader(dataset_tr,
                                batch_size=batch_size,
                                epochs=best_epoch)

model = build_model()
optimizer = Adam(learning_rate)
loss_fn = CategoricalCrossentropy()

epoch = step = 0
results = []

for batch in loader_full_tr:
    step += 1
    loss, acc, gradients = train_step(*batch)

    results.append((loss, acc))

    if step == loader_full_tr.steps_per_epoch:
        step = 0
        epoch += 1

        train_loss, train_acc = np.mean(results, 0)
        print(f"Ep. {epoch} - Loss: {train_loss:.4f}. Acc: {train_acc:.4f}")
        results = []
```

```
# Evaluate on the test set ###############################################
print("Final evaluation on the test set")
loader_te = DisjointLoader(dataset_te, batch_size=batch_size, epochs=1)
test_results = [val_step(*batch) for batch in loader_te]
if test_results:
    test_loss, test_acc = np.mean(test_results, 0)
    print(f"Final Test Loss: {test_loss:.4f}. Final Test Acc: {test_acc:.4f}")
else:
    print("No test data available.")
```

*Note: This section only demonstrates the complete code for the GCN model. The structures of the GIN and GraphSAGE-Mean models are almost identical, with differences only in the hyperparameters.*

## 9.2 Code for Task 2: Protein-Protein Interactions (PPI)

```
# Packages and Libraries
!pip install torch-geometric
!pip install tabulate
!pip install networkx matplotlib

import os.path as osp
import matplotlib.pyplot as plt

import torch
import torch.nn.functional as F
from torch.nn import Module, Linear, Dropout, BatchNorm1d

import torch_geometric.transforms as T
from torch_geometric.datasets import PPI
from torch_geometric.loader import DataLoader
from torch_geometric.nn import GATConv
from torch_geometric.explain import Explainer, GNNExplainer
from torch_geometric.nn import summary
from torch_geometric.utils import from_scipy_sparse_matrix
```

```python
import scipy.sparse as sp
import scipy.sparse
from sklearn.metrics import f1_score
from IPython.display import Image, display
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.lines as mlines
import networkx as nx

##Load data ##############################################################
path = '/tmp/PPI'
train_dataset = PPI(path, split='train')
val_dataset = PPI(path, split='val')
test_dataset = PPI(path, split='test')
train_loader = DataLoader(train_dataset, batch_size=1, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=2, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=2, shuffle=False)

# Data Description #######################################################
def get_graph_shapes(dataset, name):
    print(f"{name} Dataset:")
    print("=================")
    print(f"Number of graphs: {len(dataset)}")
    total_nodes = sum([data.num_nodes for data in dataset])
    total_edges = sum([data.num_edges for data in dataset])
    num_features = dataset[0].num_features
    num_classes = dataset[0].y.size(1) if dataset[0].y.ndim > 1
    else len(torch.unique(dataset[0].y))
    print(f"Total number of nodes: {total_nodes}")
    print(f"Total number of edges: {total_edges}")
    print(f"Node feature dimension: {num_features}")
    print(f"Number of classes: {num_classes}")
    print("=================\n")
    for i, data in enumerate(dataset):
        print(f"Graph {i + 1}:")
        print(f"  Number of nodes: {data.num_nodes}")
        print(f"  Number of edges: {data.num_edges}")
        print(f"  Node features shape: {data.x.shape}")
        print(f"  Edge index shape: {data.edge_index.shape}")
        print(f"  Labels shape: {data.y.shape}")
        print()

# print informations
get_graph_shapes(train_dataset, "Train")
get_graph_shapes(val_dataset, "Validation")
get_graph_shapes(test_dataset, "Test")

# Details of the first graph
def dataset_statistics(dataset, name):
    first_data = dataset[0]
    print(f"First graph information:")
    print("=========================")
    print(f"Number of nodes: {first_data.num_nodes}")
    print(f"Number of edges: {first_data.num_edges}")
    print(f"Node features: {first_data.x}")
    print(f"Edge index: {first_data.edge_index}")
    print(f"Node labels: {first_data.y}")
    print("\n")

dataset_statistics(train_dataset, "Train")

# Hyperparameters ########################################################
class Config:
    def __init__(self):
        self.in_features = train_dataset.num_features
        self.n_hidden = 256
        self.n_classes = train_dataset.num_classes
        self.n_heads = 4
        self.dropout = 0.2
        self.learning_rate = 0.005
        self.epochs = 100
        self.patience = 10
        self.delta = 0.001
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

config = Config()
```

```
# GAT Model ##########################################################

class GAT(Module):
    def __init__(self, config):
        super().__init__()
        self.gat1 = GATConv(config.in_features,
                            config.n_hidden,
                            heads=config.n_heads)
        self.lin1 = torch.nn.Linear(config.in_features,
                                    config.n_hidden * config.n_heads)
        self.bn1 = BatchNorm1d(config.n_hidden * config.n_heads)
        self.dropout1 = Dropout(p=config.dropout)

        self.gat2 = GATConv(config.n_hidden * config.n_heads,
                            config.n_hidden,
                            heads=config.n_heads)
        self.lin2 = torch.nn.Linear(config.n_hidden * config.n_heads,
                                    config.n_hidden * config.n_heads)
        self.bn2 = BatchNorm1d(config.n_hidden * config.n_heads)

        self.gat3 = GATConv(config.n_hidden * config.n_heads,
                            heads=config.n_heads,
                            concat=False)
        self.lin3 = torch.nn.Linear(config.n_hidden * config.n_heads,
                                    config.n_classes)

    def forward(self, x, edge_index):
        x = F.elu(self.gat1(x, edge_index) + self.lin1(x))
        x = self.bn1(x)
        x = self.dropout1(x)

        x = F.elu(self.gat2(x, edge_index) + self.lin2(x))
        x = self.bn2(x)

        x = self.gat3(x, edge_index) + self.lin3(x)
        return x

config = Config()
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GAT(config).to(config.device)
loss_op = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate)

# Summary
fixed_index = 0  # the first graph
data = train_dataset[fixed_index].to(device)
data = data.to(device)
model_summary = summary(model, data.x, data.edge_index)
print(model_summary)


# Train ##################################################################
def train():
    model.train()

    total_loss = 0
    ys, preds = [], []
    for data in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        out = model(data.x, data.edge_index)
        loss = loss_op(out, data.y)
        total_loss += loss.item() * data.num_graphs
        ys.append(data.y.cpu())
        preds.append((out > 0).float().cpu())
        loss.backward()
        optimizer.step()

    y, pred = torch.cat(ys, dim=0).numpy(), torch.cat(preds, dim=0).numpy()
    f1 = f1_score(y, pred, average='micro') if pred.sum() > 0 else 0
    return total_loss / len(train_loader.dataset), f1


@torch.no_grad()
def evaluate(loader):
    model.eval()
    total_loss = 0
    ys, preds = [], []
    for data in loader:
        ys.append(data.y.cpu())
        out = model(data.x.to(device), data.edge_index.to(device))
        preds.append((out > 0).float().cpu())
        loss = loss_op(out, data.y.to(device))
        total_loss += loss.item() * data.num_graphs

    y, pred = torch.cat(ys, dim=0).numpy(), torch.cat(preds, dim=0).numpy()
    f1 = f1_score(y, pred, average='micro') if pred.sum() > 0 else 0
    return total_loss / len(loader.dataset), f1

# Early Stopping
class EarlyStopping:
    def __init__(self, patience, delta):
        self.patience = patience
        self.delta = delta
        self.best_score = None
        self.counter = 0
        self.early_stop = False

    def __call__(self, val_f1, model):
        score = val_f1
        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(model)
        elif score < self.best_score + self.delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(model)
            self.counter = 0

    def save_checkpoint(self, model):
        torch.save(model.state_dict(), 'checkpoint.pt')
```

```
early_stopping = EarlyStopping(patience=config.patience, delta=config.delta)


# Store loss and F1
train_losses, train_f1s = [], []
val_losses, val_f1s = [], []

for epoch in range(1, 101):
    train_loss, train_f1 = train()
    val_loss, val_f1 = evaluate(val_loader)
    train_losses.append(train_loss)
    train_f1s.append(train_f1)
    val_losses.append(val_loss)
    val_f1s.append(val_f1)
    # print(f'Epoch: {epoch:03d}, Train Loss: {train_loss:.4f},
    # Train F1: {train_f1:.4f}, Val Loss: {val_loss:.4f},
    # Val F1: {val_f1:.4f}')

    # conditions of
    early_stopping(val_f1, model)
    if early_stopping.early_stop:
        print("Early stopping")
        break

model.load_state_dict(torch.load('checkpoint.pt'))

# Evaluate test ##########################################################
test_loss, test_f1 = evaluate(test_loader)
print(f'Test Loss: {test_loss:.4f}, Test F1: {test_f1:.4f}')

# Visualization
epochs = range(1, len(train_losses) + 1)

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(epochs, train_losses, label='Train Loss')
plt.plot(epochs, val_losses, label='Val Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Train and Val Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(epochs, train_f1s, label='Train F1')
plt.plot(epochs, val_f1s, label='Val F1')
plt.xlabel('Epochs')
plt.ylabel('F1 Score')
plt.title('Train and Val F1 Score')
plt.legend()

plt.tight_layout()
plt.show()

# Explainability ##########################################################

# Set a random seed for reproducibility
torch.manual_seed(68)

gnn_explainer = GNNExplainer(epochs=200)
explainer = Explainer(
    model=model,
    algorithm=gnn_explainer,
    explanation_type='model',
    node_mask_type='attributes',
    edge_mask_type='object',
    model_config=dict(
        mode='multiclass_classification',
        task_level='node',
        return_type='log_probs',
    ),
)

# Select the node index of test set to interpret
data = next(iter(test_loader)).to(device)
node_idx = 8
```

```
# explanation: mask
explanation = explainer(data.x, data.edge_index, index=node_idx)

# Top 20 Important features #####################
path = 'feature_importance.png'
explanation.visualize_feature_importance(path, top_k=20)
display(Image(filename=path))

# Important Edge ######################
edge_mask = explanation.edge_mask.cpu().detach().numpy()

# Retrieve the indices and weights of the top 20 important edges
top_20_indices = edge_mask.argsort()[-20:][::-1]
top_20_edges = [(i, edge_mask[i]) for i in top_20_indices]

# Get the connection information for the edges
edge_index = data.edge_index.cpu().numpy()
top_20_edge_nodes = [(i, edge_index[0, i],
                      edge_index[1, i],
                      edge_mask[i]) for i in top_20_indices]

# Print the top 20 important edges, their weights, and connection points
print("Explanation of important edges (edge index, node i, node j, weight):")
for edge_idx, src, dst, weight in top_20_edge_nodes:
    print(f"Edge ({edge_idx}, {src}, {dst}, {weight:.4f})")

# Prepare data to plot a horizontal bar chart
edges = [f"Edge {i}" for i in top_20_indices]
weights = [edge_mask[i] for i in top_20_indices]

# Plot a horizontal bar chart
plt.figure(figsize=(10, 6))
plt.barh(edges, weights, color='skyblue')
plt.xlabel('Importance Weight')
plt.ylabel('Edges')
plt.title('Top 20 Important Edges')
plt.gca().invert_yaxis()
plt.show()
```

```python
# Create graph
G = nx.Graph()

# Top 20 important nodes
for edge_idx, src, dst, weight in top_20_edge_nodes:
    G.add_edge(src, dst, weight=weight)
target_node = node_idx

# graph setting
pos = nx.spring_layout(G)
plt.figure(figsize=(11, 5))
node_colors = ['pink' if node == target_node else 'lightblue' for node in G.nodes()]
nx.draw_networkx_nodes(G, pos, node_size=350, node_color=node_colors)
edges = nx.draw_networkx_edges(G, pos,
                               edgelist=G.edges(data=True),
                               width=2, edge_color='blue')
nx.draw_networkx_labels(G, pos, font_size=7, font_color='black')

# Legend
selected_node = mlines.Line2D([], [],
                              color='pink',
                              marker='o',
                              markersize=10,
                              label='Selected Node',
                              linestyle='None')
important_edges = mlines.Line2D([], [],
                              color='blue',
                              label='Important Edges',
                              linewidth=2)
plt.legend(handles=[selected_node, important_edges], loc='upper left')

plt.title('Top 20 Important Edges and Corresponding Nodes')
plt.show()
```