A Study on Graph Neural Networks and Graph Pooling

M.Sc. MATHEMATICS

2024-2026

A Study on Graph Neural Networks and Graph Pooling

DISSERTATION

Submitted to the University of Kerala in partial fulfillment of the requirements for the award of Master of Science in Mathematics

Programme Code : 620

Exam Code : 62023402

Course Code : MM 545

CERTIFICATE

This is to certify that the dissertation entitled **"A STUDY ON GRAPH NEURAL NETWORKS AND GRAPH POOLING"** is a record of studies carried out by **AKHILA K**, M.Sc. Mathematics student of the year **2024-2026** at **T.K.M. College of Arts and Science, Kollam**, under my guidance and submitted to the University of Kerala in partial fulfillment of the Degree of Master of Science in Mathematics.

Kollam
July 2026

Dr. ASWATHY M R
H.O.D. and Assistant Professor
Department of Mathematics
T.K.M. College of Arts and Science
Kollam

HEAD
Department of Mathematics
T.K.M. College of Arts and Science
Kollam - 691 005

Contents

INTRODUCTION

Graph Neural Networks (GNNs) have emerged as a powerful deep learning framework for processing graph-structured data. Unlike traditional neural networks, which are designed for Euclidean data such as images and text, GNNs can effectively capture the complex relationships among interconnected entities. They have found widespread applications in areas such as social network analysis, recommendation systems, molecular property prediction, knowledge graphs, and computational biology. In this Project, we introduce the graph neural network (GNN) frameworks for specifically Node-focused task. Specifically, we introduce two major components: 1) the graph filters, which refines the node features; and 2) the graph pooling, which aims to coarsen the graph and finally generate the graph-level representation. We categorize graph filters as spectral-based and spatial-based filters.We group graph pooling as flat graph pooling and hierarchical graph pooling and introduce representative methods for each group.

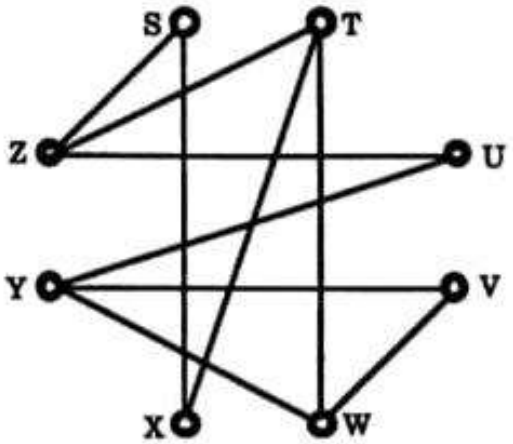
Chapter 1

Preliminaries

1.1 Graph

A graph $G = (V(G), E(G))$ consists of two finite sets:

- $V(G)$, the vertex set of the graph, often denoted by just V, which is a nonempty set of elements called **vertices**, and
- $E(G)$, the edge set of the graph, often denoted by just E, which is a possibly empty set of elements called **edges**, such that each edge $e \in E$ is assigned an unordered pair of vertices (u, v), called the **end vertices** of e.



Thus in the above figure , the vertex set is

$$V = \{S, T, U, V, W, X, Y, Z\}.$$

the edge set E has 10 edges and these edges are assigned the unordered pairs of vertices

$$(S, X),\ (S, Z),\ (T, W),\ (T, X),\ (T, Z),\ (U, Y),\ (U, Z),\ (V, W),\ (V, Y),\ (W, Y).$$

Vertices are also sometimes called points, nodes, or just dots. If e is an edge with end vertices u and v then e is said to join u and v. Note that the definition of a graph allows the possibility of the edge e having identical end vertices, i.e., it is possible to have a vertex u joined to itself by an edge such an edge is called a **loop**. We now give two examples to illustrate the above definitions.

Example 1. Let $G = (V, E)$ where

$$V = \{a, b, c, d, e\}, \qquad E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\},$$

and the ends of the edges are given by:

$$e_1 \leftrightarrow (a, b), \qquad e_2 \leftrightarrow (b, c), \qquad e_3 \leftrightarrow (c, c), \qquad e_4 \leftrightarrow (c, d),$$

$$e_5 \leftrightarrow (b, d), \qquad e_6 \leftrightarrow (d, e), \qquad e_7 \leftrightarrow (b, e), \qquad e_8 \leftrightarrow (b, e).$$

we can represent G Diagramatically as:

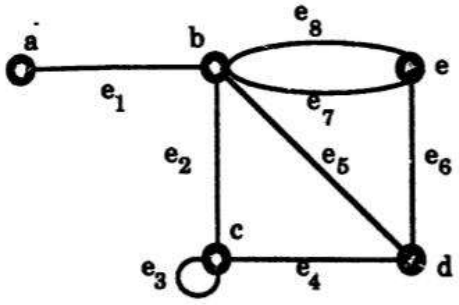


Figure 1.1: A Graph G wwith five vertices and eight edges

1.2 Graphs as Models

Problem 1 :Here is a simple example of a social media network graph

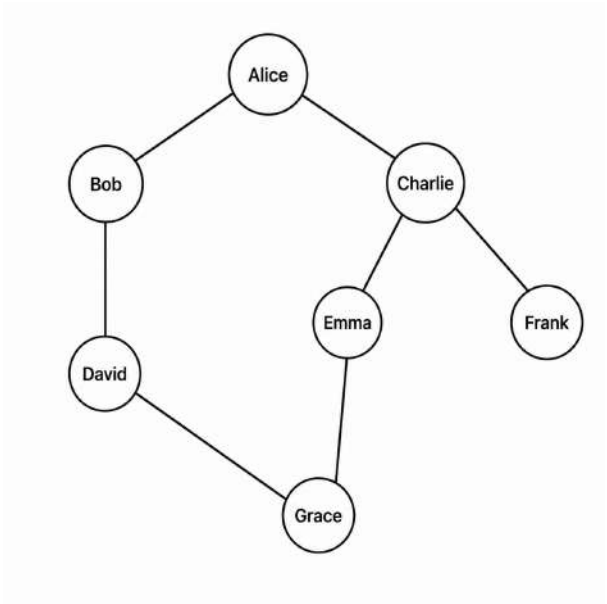


Figure 1.2: Social Media Network Graph

The Vertex set and Edge set are given by:

V ={Alice, Bob, Charlie, David, Emma, Frank, Grace}

E ={(Alice,Bob),(Alice,Charlie),(Bob,David),(Charlie,Emma),(Charlie,Frank),(David,Grace), (Emma,Grace)}

Problem 2 : Suppose that the graph of Figure 1.3 represents a network of telephone lines and poles. We are interested in the network's vulnerability to accidental disruption. We want to identify those lines and poles that must stay in service to avoid disconnecting the network.

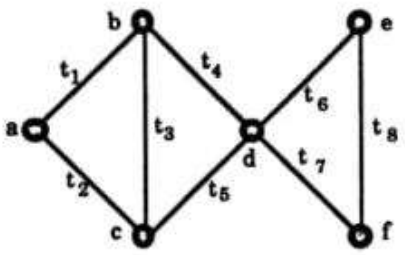


Figure 1.3: A network of telephone lines

There is no single line whose disruption (removal) will disconnect the graph (network), but the graph will become disconnected if we remove the two lines represented by the edges t_4 and t_5, for example. When it comes to poles, the network is more vulnerable since there is a single vertex, d, whose removal disconnects the graph. This illustrates the notions of edge connectivity and vertex connectivity of a graph. We may also want to find a smallest possible set of edges needed to connect the six vertices. There are several such minimal sets. One is

$$\{t_1, t_3, t_5, t_6, t_7\}.$$

1.3 More Definitions

Let G be a graph. If two (or more) edges of G have the same end vertices, then these edges are called **parallel**.

A vertex of G which is not the end of any edge is called **isolated**.

Two vertices which are joined by an edge are said to be **adjacent** or **neighbours**.

The set of all neighbours of a fixed vertex v of G is called the **neighbourhood set** of v and is denoted by $N(v)$.

Let v be a vertex of the graph G. The **degree** $d(v)$ (or $d_G(v)$ if we want to emphasize G) of v is the number of edges of G incident with v, counting each loop twice, i.e., it is the number of times v is an end vertex of an edge.

If $G = (V, E)$ and V has at least two elements (i.e., G has at least two vertices), then for any vertex v of G, $G - v$ denotes the subgraph of G with vertex set $V - \{v\}$ and whose edges are all those of G which are not incident with v, i.e., $G - v$ is obtained from G by removing v and all the edges of G which have v as an end. $G - v$ is referred to as a **vertex deleted subgraph**. If $G = (V, E)$ and e is an edge of G then $G - e$ denotes the subgraph of G having V as its vertex set and $E - \{e\}$ as its edge set, i.e., $G - e$ is obtained from G by removing the edge e (but not its endpoint(s)). $G - e$ is referred to as an **edge deleted subgraph**.

If $G = (V, E)$ and U is a proper subset of V then $G - U$ denotes the subgraph of G with vertex set $V - U$ and whose edges are all those of G which are not incident with any vertex in U.

If F is a subset of the edge set E then $G - F$ denotes the subgraph of G with vertex set V and edge set $E - F$, i.e., obtained by deleting all the edges in F, but not their endpoints. $G - U$ and $G - F$ are also referred to as **vertex deleted** and **edge deleted subgraphs** (respectively).

Hypergraph

A **hypergraph** is a generalization of a graph in which an edge can connect more than two vertices. A hypergraph is denoted by

$$H = (V, E)$$

where

- V is the set of vertices, and
- E is the set of hyperedges such that each hyperedge is a subset of V.

That is,

$$E = \{e_1, e_2, \ldots, e_m\}$$

where

$$e_i \subseteq V.$$

Example: Consider the hypergraph

$$H = (V, E)$$

where

$$V = \{A, B, C, D, E\}$$

and

$$E = \{e_1, e_2, e_3\}$$

such that

$$e_1 = \{A, B, C\},$$

$$e_2 = \{B, D\},$$

$$e_3 = \{C, D, E\}.$$

Here,

- hyperedge e_1 connects the vertices A, B, and C,
- hyperedge e_2 connects the vertices B and D,
- hyperedge e_3 connects the vertices C, D, and E.

1.4 Adjacency and Laplacian Matrix

Adjacency Matrix

Let $G = (V, E)$ be a graph with vertices

$$V = \{v_1, v_2, \ldots, v_n\}.$$

The **adjacency matrix** of G is an $n \times n$ matrix

$$A = [a_{ij}]$$

defined by

$$a_{ij} = \begin{cases} 1, & \text{if there is an edge between } v_i \text{ and } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

Example: Consider the graph with vertex set

$$V = \{P, Q, R, S\}$$

and edge set

$$E = \{(P, Q), (P, R), (Q, R), (R, S)\}.$$

The adjacency matrix of the graph is

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Laplacian Matrix

Let $G = (V, E)$ be an undirected graph.The Laplacian matrix of G is the matrix

$$L = [l_{ij}]$$

defined by

$$l_{ij} = \begin{cases} d(v_i), & \text{if } i = j, \\ -1, & \text{if } v_i \text{ is adjacent to } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

where $d(v_i)$ denotes the degree of the vertex v_i.

Example: For a simple example, take the graph with vertices $1, 2, 3, 4$ and edges

$$(1, 2),\ (1, 3),\ (2, 3),\ (2, 4);$$

For this graph:

- Vertex 1 has degree 2
- Vertex 2 has degree 3
- Vertex 3 has degree 2
- Vertex 4 has degree 1

then the Laplacian is

$$L = \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 2 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}.$$

Diagonal Degree Matrix

The **diagonal degree matrix** of a graph is a diagonal matrix in which each diagonal entry represents the degree of the corresponding vertex of the graph. For a graph with vertices $v_1, v_2, \ldots, v_n$, the diagonal degree matrix is denoted by D and is given by

$$D = \begin{bmatrix} d(v_1) & 0 & 0 & \cdots & 0 \\ 0 & d(v_2) & 0 & \cdots & 0 \\ 0 & 0 & d(v_3) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & d(v_n) \end{bmatrix}$$

where $d(v_i)$ is the degree of vertex v_i.

Example: Consider the graph with vertices v_1, v_2, v_3, v_4 having degrees:

$$d(v_1) = 2, \quad d(v_2) = 3, \quad d(v_3) = 2, \quad d(v_4) = 1$$

Then the diagonal degree matrix is

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Relation Between Laplacian Matrix, Degree Matrix, and Adjacency Matrix

For a graph G, the Laplacian matrix L is defined as

$$L = D - A$$

where:

- L = Laplacian matrix
- D = Diagonal degree matrix
- A = Adjacency matrix

Chapter 2

The General GNN Frameworks

2.1 Introduction To GNNs

Graph Neural Networks (GNNs) are a class of deep learning models specifically designed to process and analyze graph-structured data. In a GNN, the main prediction or graph learning tasks usually fall into three levels: node-level, edge-level, and graph-level. These tasks cover predicting labels or values for individual nodes, relationships between node pairs, or entire graphs respectively.

1. **Node-level tasks**

   Here the model predicts something about each node in the graph. Common examples are node classification, where each node gets a class label, and node regression, where each node gets a continuous value.

   Example: In a social network, predict whether a user belongs to a certain community or whether a molecule atom has a certain property.

2. **Edge-level tasks**

   Here the model predicts something about a pair of nodes, usually the existence or type of a connection between them. The most common example is link prediction, which asks whether two nodes should be connected or how strongly they are related.

Example: Recommend a friendship link in a social network or predict the bond type between two atoms in a molecule.

3. **Graph-level tasks**

   Here the model predicts one label or value for the whole graph. This includes graph classification and graph regression, where the output summarizes the entire structure using information from all nodes and edges. **Example:** Classify a molecule as toxic or non-toxic, or predict a numerical molecular property.

The key idea behind GNNs is to learn meaningful representations of nodes, edges, or entire graphs by iteratively aggregating information from neighboring nodes.Here the information refers to **Node features** and **Edge features**.

Let $G = (V, E)$ be a graph, where V is the set of nodes and E is the set of edges.

Node Features

For each node $v_i \in V$, the node feature is a vector

$$x_i \in \mathbb{R}^{d_n},$$

where

$$d_n = \text{number of features (or attributes) for each node.}$$

If there are $N = |V|$ nodes, all node features can be stacked into a matrix

$$X \in \mathbb{R}^{N \times d_n},$$

where,the i-th row is x_i^T.

Edge Features

For each edge $(v_i, v_j) \in E$, the edge feature is a vector

$$e_{ij} \in \mathbb{R}^{d_e},$$

where,

$$d_n = \text{number of features (or attributes) for each edge.}$$

If the graph has $M = |E|$ edges, all edge features can be collected into an edge-feature set or edge-feature matrix, depending on the implementation, with each edge carrying its own attribute vector in $\mathbb{R}^{d_e}$.

2.2 General Structure of a Graph Neural Network

The general structure of a Graph Neural Network (GNN) is similar to that of a traditional neural network. A GNN typically consists of three major components:

- Input Layer
- Hidden Layers
- Output Layer

Input Layer

The input layer receives the graph data. A graph is generally represented as

$$G = (V, E),$$

where V denotes the set of nodes and E denotes the set of edges. Each node in the graph is associated with a feature vector. The collection of all node features is represented by the feature matrix

$$H \in \mathbb{R}^{n \times d},$$

where:

- n is the number of nodes,
- d is the dimension of node features.

Hidden Layers

The hidden layers perform graph filtering or message passing operations. In each hidden layer, a node gathers information from its neighboring nodes and updates its representation.

Suppose:

- $H^{(k)}$ = node feature matrix at layer k
- $H^{(k+1)}$ = updated feature matrix at layer $k+1$

The transformation is:

$$H^{(k+1)} = \sigma\left(\text{AGGREGATE}(H^{(k)})W^{(k)} + b^{(k)}\right)$$

where:

- AGGREGATE = neighbor aggregation/filtering
- $W^{(k)}$ = learnable weight matrix
- $b^{(k)}$ = bias vector
- σ = activation function

In GNNs, the most commonly used activation function is ReLU.

$$\text{ReLU}(x) = \max(0, x)$$

so it outputs the input if it is positive and 0 if it is negative. The hidden layers help the network learn meaningful structural and feature representations from the graph.

Output Layer

After several GNN layers, we obtain:

$$H^{(K)}$$

where:

- K = final GNN layer
- $H^{(K)}$ = final node embeddings

Each node embedding produces a node prediction:

$$\hat{Y} = \text{softmax}\left(H^{(K)}W_o\right)$$

Here softmax is an activation function which converts raw output values into probabilities. After the GNN makes predictions,we find loss function that measures the error, and backward propagation updates the weights to reduce that error.

Loss Function

The loss function measures How far the predictions are from the true values. It gives a single numerical value called the loss. Smaller loss means better predictions.Commonly used Loss Functions are Mean Squared Error,Mean Absolute Error

Now we compute

$$\frac{\partial \mathcal{L}}{\partial W^{(k)}}$$

This is called the gradient of the loss with respect to the weight matrix $W^{(k)}$. This gradient tells how sensitive the loss is to changes in the weights. The magnitude of the gradient shows how strongly a weight affects the loss. If the gradient is large, even a small change in the weight can significantly change the loss value. This means that the corresponding weight has a strong influence on the prediction error. On the other hand, if the gradient is small, changing that weight has very little effect on the loss, so the weight is less important for the current prediction. The sign of the gradient determines the direction in which the weight should move. If the gradient is positive, increasing the weight increases the loss, so the weight should be decreased. If the gradient is negative, increasing the weight decreases the loss, so the weight should be increased. In this way, the network knows how to modify its parameters to improve predictions. After computing

the gradients, the weights are updated using gradient descent:

$$W^{(k)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(k)}} \rightarrow W^{(k)}$$

where η is the learning rate.

In practice, the learning rate is usually selected experimentally. The learning rate controls the size of the update step. If the learning rate is very small, the weights change only slightly during each update. Training becomes slow, and the network may take a long time to reduce the loss.If the learning rate is very large, the weights change too much in each step. By repeatedly updating the weights in the direction that reduces the loss, the GNN gradually learns better node representations and improves its performance over time.

2.3 Graph Filters

There are various perspectives to design graph filters, which can be roughly split into two categories: (1) spectral-based graph filters and (2) spatial-based graph filters.

The spatial-based graph filters explicitly leverage the graph structure (i.e., the connections between the nodes) to perform the feature refining process in the graph domain. In contrast, the spectral-based graph filters utilize spectral graph theory to design the filtering operation in the spectral domain. These two categories of graph filters are closely related. Especially, some of the spectral-based graph filters can be regarded as spatial-based filters. In this section, we introduce spectral-based graph filters and explain how some of the spectral-based graph filtersand then discuss spatial-based graph filters.

2.3.1 Spectral-based Graph Filters

The spectral-based graph filters are designed in the spectral domain of graph signals. We first introduce the graph spectral filtering and then describe how it can be adopted to design spectral-based graph filters.

Spectral Graph Filtering

The idea of the graph spectral filtering is to modulate the frequencies of a graph signal such that some of its frequency components are kept/amplified while others are removed/diminished. Hence, given a graph signal $f \in \mathbb{R}^N$, we need first to apply the Graph Fourier Transform (GFT) on it to obtain its graph Fourier coefficients, and then modulate these coefficients before reconstructing the signal in the spatial domain. For a signal $f \in \mathbb{R}^N$ defined on a graph G, its Graph Fourier Transform is defined as follows:

$$\hat{f} = U^\top f, \tag{2.1}$$

where U consists of the eigenvectors of the Laplacian matrix of G and $\hat{f}$ is the obtained graph Fourier coefficients for the signal f. These graph Fourier coefficients describe how each graph Fourier component contributes to the graph signal f. Specifically, the i-th element of $\hat{f}$ corresponds to the i-th graph Fourier component u_i with the frequency λ_i. Note that λ_i is the eigenvalue corresponding to u_i. To modulate the frequencies of the signal f, we filter the graph Fourier coefficients as follows:

$$\hat{f}'[i] = \hat{f}[i] \cdot \gamma(\lambda_i), \qquad \text{for } i = 1, \ldots, N. \tag{2.2}$$

where $\gamma(\lambda_i)$ is a function with the frequency λ_i as input which determines how the corresponding frequency component should be modulated. This process can be expressed in a matrix form as follows:

$$\hat{f}' = \gamma(\Lambda) \cdot \hat{f} = \gamma(\Lambda) \cdot U^\top f, \tag{2.3}$$

where Λ is a diagonal matrix consisting of the frequencies (eigenvalues of the Laplacian matrix) and $\gamma(\Lambda)$ is obtained by applying the function $\gamma(\cdot)$ to each element on the diagonal

of Λ. Formally, Λ and $\gamma(\Lambda)$ can be represented as follows:

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \lambda_N \end{pmatrix}, \qquad \gamma(\Lambda) = \begin{pmatrix} \gamma(\lambda_1) & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \gamma(\lambda_N) \end{pmatrix} \tag{2.4}$$

With the filtered coefficients, we can now reconstruct the signal to the graph domain using the Inverse Graph Fourier Transform (IGFT) as follows:

$$f' = U\hat{f}' = U \cdot \gamma(\Lambda) \cdot U^\top f, \tag{2.5}$$

where f' is the obtained filtered graph signal. The filtering process can be regarded as applying the operator $U \cdot \gamma(\Lambda) \cdot U^\top$ to the input graph signal. For convenience, we sometimes refer to the function $\gamma(\Lambda)$ as the *filter* since it controls how each frequency component of the graph signal f is filtered.

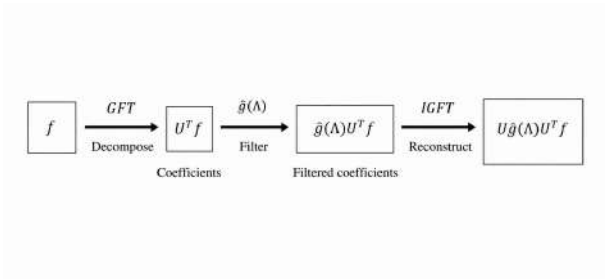


Figure 2.1: The Process of Spectral Filtering

We have introduced the graph spectral filtering operator, which can be used to filter certain frequencies in the input signal. For example, if we want to get a smooth signal after filtering, we can design a low-pass filter, where $\gamma(\lambda)$ is large when λ is small, and it is small when λ is large. In this way,the obtained filtered signal is smooth since it majorly contains the low-frequency part of the input signal. However, when utilizing the spectral-based filter as a graph filter in graph neural networks, we often do not know which frequencies are more important. Hence, just like the classical neural networks,

the graph filters can be learned in a data-driven way. Specifically, we can model $\gamma(\Lambda)$ with certain functions and then learn the parameters with the supervision from data. A natural attempt is to give full freedom when designing $\gamma(\cdot)$ (or a non-parametric model). In detail, the function $\gamma(\cdot)$ is defined as follows:

$$\gamma(\lambda_l) = \theta_l,$$

where θ_l is a parameter to be learned from the data. It can also be represented in a matrix form as follows:

$$\gamma(\Lambda) = \begin{pmatrix} \theta_1 & 0 & \cdots & 0 \\ 0 & \theta_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \theta_N \end{pmatrix}$$

However, there are some limitations with this kind of filter. First, the number of parameters to be learned is equal to the number of nodes N, which can be extremely large in real-world graphs. Hence, it requires lots of memory to store these parameters and also abundant data to fit them. Second, the filter $U \cdot \gamma(\Lambda) \cdot U^\top$ is likely to be a dense matrix. Therefore, the calculation of the i-th element of the output signal f' could relate to all the nodes in the graph. In other words, the operator is not spatially localized. Furthermore, the computational cost for this operator is quite expensive due to the eigen decomposition of the Laplacian matrix and the matrix multiplication between dense matrices when calculating $U \cdot \gamma(\Lambda) \cdot U^\top$. To address these issues, a polynomial filter operator, which we denoted as Poly-Filter. The function $\gamma(\Lambda)$can be modeled with a K-order truncated polynomial as follows:

$$\gamma(\lambda_l) = \sum_{k=0}^{K} \theta_k \lambda_l^k$$

In terms of the matrix form, it can be rewritten as below:

$$\gamma(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k$$

Clearly, the number of the parameters in above two equations is $K + 1$, which is not dependent on the number of nodes in the graph. Furthermore, we can show that $U \cdot \gamma(\Lambda) \cdot U^\top$ can be simplified to be a polynomial of the Laplacian matrix. It means that:

1. no eigendecomposition is needed; and

2. the polynomial parametrized filtering operator is spatially localized, i.e., the calculation of each element of the output f' only involves a small number of nodes in the graph.

Next, we first show that the Poly-Filter operator can be formulated as a polynomial of the Laplacian matrix and then understand it from a spatial perspective. By applying this Poly-Filter operator on f, according to Eq. (2.5), we can get the output f' as follows:

$$\begin{aligned} f' &= U \cdot \gamma(\Lambda) \cdot U^\top f \\ &= U \cdot \sum_{k=0}^{K} \theta_k \Lambda^k \cdot U^\top f \\ &= \sum_{k=0}^{K} \theta_k \, U \cdot \Lambda^k \cdot U^\top f \end{aligned}$$

2.3.2 Spatial-based Graph Filters

Spatial graph filtering processes graph data directly in the graph domain by updating each node using information from its neighboring nodes. In this approach, every node collects and combines features from the nodes connected to it through the graph structure. Unlike spectral methods, spatial filtering does not require transforming the graph signal into another domain such as the Fourier domain. Instead, the computation is performed locally on the graph itself. A spatial graph filter is a function that computes a node's feature vector by combining its own features with the features of its neighbors according to the graph connectivity.

$$h_v^{(l+1)} = \phi\left(h_v^{(l)}, \mathrm{AGG}\left\{h_u^{(l)} : u \in \mathcal{N}(v)\right\}\right)$$

where:

- $h_v^{(l)}$ is the feature vector of node v at layer l,
- $\mathcal{N}(v)$ denotes the neighborhood of node v,
- AGG represents an aggregation function such as sum, mean, or max,
- ϕ denotes a **learnable transformation**(Transformation containing trainable parameters such as weight matrices and biases that are learned during training.)

The above equation shows that the representation of node v at layer $l+1$ is obtained by combining its current feature vector with the aggregated information from its neighboring nodes.

Chapter 3

Graph Pooling

The graph filters refine the node features without changing the graph struc- ture. After the graph filter operation, each node in the graph has a new feature representation. Typically, the graph filter operations are sufficient for node- focused tasks that take advantage of the node representations. However, for graph-focused tasks, a representation of the entire graph is desired. To obtain such representation, we need to summarize the information from the nodes. There are two main kinds of information that are important for generating the graph representation is the node features, and the other is the graph structure. The graph representation is expected to preserve both the node fea- ture information and the graph structure information. Similar to the classical convolutional neural networks, graph pooling layers are proposed to generate graph-level representations. The early designs of graph pooling layers are usually flat. In other words, they generate the graph-level representation directly from the node representations in a single step. For example, the average pooling layers and max-pooling layers can be adapted to graph neural networks by applying them to each feature. Later on, hierarchical graph pooling designs have been developed to summa- rize the graph information by coarsening the original graph step by step. In the hierarchical graph pooling design, there are often several graph pooling layers, each of which follows a stack of several filters.

3.1 Problem Formulation

Notions. Let $G = (V, E)$ denote a graph with node set V and edge set E. Node features are denoted as $X \in \mathbb{R}^{n \times d}$, where n is the number of nodes, and d is the dimension of node features. The adjacency matrix is defined as $A \in \{0, 1\}^{n \times n}$. $A[i, j] = 1$ if there exists an edge between node v_i and node v_j, otherwise, $A[i, j] = 0$.

Graph Pooling. Let a graph pooling operator be defined as any function POOL that maps a graph G to a new pooled graph $G' = (V', E')$:

$$G' = \text{POOL}(G), \tag{3.1}$$

where $|V'| < |V|$.[1] The primary goal of graph pooling is to reduce the number of nodes in a graph while preserving the semantic information of the graph.

3.2 Approaches for Graph Pooling

Graph pooling can be roughly divided into flat pooling and hierarchical pooling according to its role in graph-level rep- resentation learning. The former directly generates graph-level representations in a single step ($|V'| = 1$), while the latter coarsens the graph gradually into a smaller sized graph ($|V'| > 1$).

3.2.1 Global Pooling(Flat Pooling)

The global graph pooling operators are employed as readout functions to transform the graph into a single low-dimensional dense vector.In Global pooling layers, there is no new graph but a single node being generated.In practical usage, global pooling generally has wider applications than hierarchical pooling. Based on complexity and representation capability, representative methods of global pooling operators can be categorized into four categories, including simple functions, grouping/cascading, attention, and learnable

[1]The pooled graph contains fewer nodes than the original graph.

functions. Simple functions refer to simple permutation-invariant functions,), and Grouping/Cascading refers to a class of graph representation methods in grouping or cascading nodes. Attention refers to the method of weighted summation of node representations, with the weights typically filled by attention coefficients. Learnable functions are parametric approaches, particularly utilizing neural networks.

Simple permutation-invariant functions

Simple permutation-invariant functions are graph pooling methods that aggregate node features without considering how the nodes are connected. In other words, these methods produce the same output regardless of the order in which the nodes are arranged. The three most common permutation-invariant functions are:

- **Sum Pooling:**

$$\mathbf{h}_G = \sum_{i=1}^{N} \mathbf{h}_i \tag{3.2}$$

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

- **Mean Pooling:**

$$\mathbf{h}_G = \frac{1}{N} \sum_{i=1}^{N} \mathbf{h}_i \tag{3.3}$$

  This computes the average of all node feature vectors.

- **Max Pooling:**

$$\mathbf{h}_G^{(j)} = \max_{i=1,\ldots,N} \mathbf{h}_i^{(j)}, \qquad j = 1, \ldots, d \tag{3.4}$$

  where d is the feature dimension, and the maximum is taken independently for each feature dimension.

Among these, **Sum** and **Mean** pooling were the earliest global pooling techniques introduced in graph neural networks. They are computationally efficient and straightforward to implement, making them popular baseline methods for generating a graph-level representation. However, because they ignore the graph's structural connectivity, they may fail

to capture important topological information required for more complex graph learning tasks.

3.2.2 Hierarchical Pooling

Hierarchical pooling methods aim to preserve the hierarchical graph's structural information by iteratively coarsening the graph into a new graph of smaller size. The hierarchical pooling can be roughly classified into node clustering pooling,node drop pooling, and other pooling according to the manner in which it coarsens a graph. The main difference between the first two types of methods is that node clustering pooling generates new nodes for the coarsened graph, whereas node drop pooling retains nodes from the original graph. Note that hierarchical pooling methods still technically employ flat pooling methods to obtain the graph-level representation of the coarsened graph.

Node Clustering Pooling

Based on the assumption that each node is a member of a potentially significant substructure, clustering pooling transfers nodes from the input graph to nodes of the coarsened graph. The nodes in the coarsened graph can also be referred to as **supernodes**, and the coarsened graph can also be referred to as a **hypergraph**, since each node represents a substructure of the original graph, where clustering pooling operators can benefit from existing community detection and graph clustering algorithms. A real-life application analogous to this assumption can be found in molecular structures, where functional groups are considered as communities of atoms, and molecules are assemblies of several functional groups. In addition to discovering the alignment between nodes and supernodes, clustering pooling requires learning the representation of supernodes and determining the links between supernodes. The last two tasks can be summarized as learning hypergraphs, whose central task is node clustering. Formally, a generic computational flow for clustering pooling can be described in the following steps. **Step 1:**

Node Clustering Given a potential hypergraph G_{hyper} with node set V_{hyper}, we assume

$$|V_{\text{hyper}}| < |V|$$

and define a surjection

$$f_{\text{clus}} : V \to V_{\text{hyper}}$$

also known as the **vertex mapping function** or **clustering function**, in which each node in V corresponds to at least one node in V_{hyper}. The variation in the clustering function f_{clus} results in significant differences in clustering pooling. Furthermore, permutation invariance of the clustering functions is required for clustering pooling permutation invariance. An explicit or implicit cluster assignment matrix

$$S \in \mathbb{R}^{|V| \times |V_{\text{hyper}}|}$$

with each row representing a node and each column representing a supernode, can be used to describe the outcome of node clustering.

remark 3.2.1 *A **cluster-assignment matrix** is a rectangular binary (or nonnegative) matrix that encodes which data point belongs to which cluster. Each row corresponds to a data point and each column corresponds to a cluster. The entry*

$$A_{ij} = \begin{cases} 1, & \textit{if data point } i \textit{ is assigned to cluster } j, \\ 0, & \textit{otherwise.} \end{cases}$$

More generally, A_{ij} may take a positive value representing a weighted assignment. An

example of a cluster-assignment matrix is

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

where 5 data points are assigned to 3 clusters.

The pooling ratio r is defined as the ratio of the number of clusters to the number of nodes:

$$r = \frac{|V_{\text{hyper}}|}{|V|}$$

The pooling ratio is a hyperparameter in both deterministic algorithms and parametric networks, since it determines the size of the hypergraph, which affects the computational complexity of the algorithm and the amount of preserved information.

Step 2: Learning Hypergraph(Graph Coarsening)

Step 1 specifies which supernodes are included in the hypergraph, but it does not determine what features the supernodes possess or how they are linked to one another. Hence, the hypergraph created in the previous step is revised to include the following two components. **Step 2.1: Learning the Representation of Supernodes.**

This step can be considered as a local readout operation, which can use any readout func-tion One of the classical methods is to use the cluster assignment matrix to conduct a weighted summation where Z^{l+1} is the representation matrix of supernodes that has not yet performed message passing.

$$Z^{l+1} = S^T H^l, \qquad Z^{l+1} \in \mathbb{R}^{N_{l+1} \times d}$$

Step 2.2: Learning the Hyperedges.

Intuitively, if two clusters are adjacent, then at least one node in each cluster is adjacent

Models	CAM Generator	Graph Coarsening
DiffPool	$\mathrm{C} = \text{softmax}\left(\mathrm{GNN}_{\text{pool}}(X, A)\right)$	$\begin{cases} X' = C^T \cdot \mathrm{GNN}_{\text{emb}}(X, A) \\ A' = C^T AC \end{cases}$
NMF	$\begin{cases} A \approx UV \\ C = V^T \end{cases}$	$\mathrm{X'} = \mathrm{C}^T X, \qquad A' = C^T AC$
LaPool	$\begin{cases} h_i = \left\| \sum_{j \in \mathcal{N}(v_i)} A_{ij}(x_i - x_j) \right\|_2 \\ V_e = \{v_i \in V \mid \forall v_j,\ h_i - A_{ij} s_j > 0\} \\ C = \text{sparsemax}\left(p \frac{XX^T}{\|X\| \|X_c\|}\right) \end{cases}$	$\begin{cases} X' = \mathrm{MLP}(C^T X) \\ A' = C^T AC \end{cases}$
MinCut	C=MLP(X)	$\begin{cases} X' = C^T X, \quad \hat{A} = C^T \tilde{A} C \\ A' = \hat{A} - I\,\mathrm{diag}(\hat{A}) \end{cases}$
StructPool	C:Minimize E(C)	$\mathrm{X'}=\mathrm{C}^T X, \qquad A' = C^T AC$
MemPool	$\begin{cases} C_{ij} = \frac{(1 + \|x_i - k_j\|^2/\tau)^{-\frac{\tau+1}{2}}}{\sum_j (1 + \|x_i - k_j\|^2/\tau)^{-\frac{\tau+1}{2}}} \\ C = \text{softmax}\left(\Gamma\left(\Big\|_{m=0}^{\lvert m \rvert} C_i\right)\right) \end{cases}$	$\mathrm{X'}=\mathrm{MLP}(\mathrm{C}^T X)$
HAP	$\begin{cases} T = \mathrm{GCont}(X) \\ C_{ij} = \sigma\left(p^T [h_{u_m}, h_{u_l}]\right) \\ C = \text{softmax}(C) \end{cases}$	$\begin{cases} X' = \mathrm{MLP}(C^T X), \quad \hat{A} = C^T AC \\ A' = \text{Gumbel-SoftMax}(\hat{A}) \end{cases}$
SEP	C:Minimize $\mathrm{H}^T(\mathcal{G})$	$\mathrm{X'}=\mathrm{C}^T X, \qquad A' = C^T AC$

Notations: $X' \in \mathbb{R}^{c \times d}$ $and A' \in \mathbb{R}^{c \times c}$are the adjacency matrix and feature matrix for the new graph, respectively; $\tilde{A} \in \mathbb{R}^{n \times n}$ is the adjacency matrix with self-loop; $L \in \mathbb{R}^{n \times n}$ is the graph Laplacian; $I_k \in \mathbb{R}^{c \times c}$ is the identity matrix; $k \in \mathbb{R}^d$ is a memory key vector; $p \in \mathbb{R}^{1 \times 2d}$ is a trainable vector; τ is the degree of freedom of the Student's t-distribution, i.e., temperature; c is the number of clusters; $|m|$ is the number of heads; Γ is an $[1 \times 1]$ convolutional operator; $\|$ is the concatenation operator; GCont is an auto-learned global graph content; and Gumbel-SoftMax achieves soft sampling for neighborhood relationships to decrease the edge density.

Table 3.1: Summary of representative node clustering pooling methods.

to a node in the other cluster. Similarly, the transformation of a node adjacency matrix yields the adjacency matrix of a supernode. Hence, the hypergraph adjacency matrix A^{l+1} can be calculated using the original graph adjacency matrix A^l and the cluster assignment matrix S, as shown below:

$$A^{l+1} = S^T A^l S$$

More graph coarsening methods are given in the table 3.1 .

Node Drop Pooling

Node drop pooling exploits learnable scoring functions to delete nodes with comparatively lower significance scores. For a thorough analysis of node drop pooling, we propose a universal and modularized framework, which consists of three disjoint modules:

1. **Score Generator.** Given an input graph, the score generator calculates significance scores for each node.

2. **Node Selector.** The node selector selects the nodes with top-k significance scores.

3. **Graph Coarsening.** With the selected nodes, a new graph coarsened from the original one is obtained by learning a new feature matrix and an adjacency matrix.

The process can be formulated as follows:

$$S^{(l)} = \underbrace{\text{SCORE}\big(X^{(l)}, A^{(l)}\big)}_{\text{Score Generator}}$$

$$\text{idx}^{(l+1)} = \underbrace{\text{TOP}_k\big(S^{(l)}\big)}_{\text{Node Selector}}$$

$$X^{(l+1)}, A^{(l+1)} = \underbrace{\text{COARSEN}\Big(X^{(l)}, A^{(l)}, S^{(l)}, \text{idx}^{(l+1)}\Big)}_{\text{Graph Coarsening}}$$

where functions

$$\text{SCORE}, \quad \text{TOP}_k, \quad \text{and} \quad \text{COARSEN}$$

are specially designed by each method for the score generator, node selector, and graph coarsening modules, respectively.

$$S^{(l)} \in \mathbb{R}^{n \times 1}$$

indicates the significance scores. The operator

$$\text{TOP}_k$$

ranks the values and returns the indices of the largest k values in $S^{(l)}$. The index set

$$\text{idx}^{(l+1)}$$

indicates the reserved node indices for the new graph.

Models	Score Generator	Node Selector	Graph Coarsening
TopKPool	$S = Xp/\|\|p\|\|_2$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot \sigma(S_{\text{idx}})$ $A' = A_{\text{idx,idx}}$
SAGPool	$S = \text{GNN}(X, A)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$
AttPool	$S = \text{softmax}(XW)$	$\text{idx} = \text{TOP}_k(S)$	$X' = A_{\text{idx}}(X \odot S)$ $A' = A_{\text{idx}} A A_{\text{idx}}^T$
ASAP	$S = LEConv(X^e, A)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X^e_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx}} A A_{\text{idx}}^T$
HGP-SL	$S = \|\|(I - D^{-1}A)X\|\|_1$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $\hat{A} = A_{\text{idx,idx}}$
VIPool	$P = \frac{1}{t}\sum_{h=1}^{t}(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}})^h W^h \text{MLP}(X)$ $S = \sigma(\text{MLP}(\text{MLP}(X, P)))$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = \text{softmax}(A_{\text{idx}}) A \text{softmax}(A_{\text{idx}})^T$
GSAPool	$S_1 = \text{GNN}(X, A)$ $S_2 = \sigma(\text{MLP}(X))$ $S = \alpha S_1 + (1-\alpha) S_2$	$\text{idx} = \text{TOP}_k(S)$	$X' = (AXW)_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$
PANPool	$Z_\mu = \sum_{n=0}^{L} e^{-\mu} \sum_{j=1}^{N} g(i,j,n)$ $M = Z^{-1} \sum_{n=0}^{L} e^{-\mu \frac{n}{L}} A^n$ $S = Xp + \beta\, \text{diag}(M)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot \sigma(S_{\text{idx}})$ $A' = A_{\text{idx,idx}}$
CGIPool	$S_p = \text{GNN}_p(X, A)$ $S_f = \text{GNN}_f(X, A)$ $S = \sigma(S_p - S_f)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$
TAPool	$S_t = \text{softmax}\left(\frac{1}{n}((XX^T) \odot (\tilde{D}^{-1}\tilde{A}))1_n\right)$ $S_g = \text{softmax}(\tilde{D}^{-1}\tilde{A}Xp)$ $S = S_t + S_g$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$

Notations: $X' \in \mathbb{R}^{k \times d}$ and $A' \in \{0,1\}^{k \times k}$ are the feature matrix and adjacency matrix of the new graph, respectively. $D \in \mathbb{R}^{n \times n}$ is the degree matrix of A. $\bar{A}^h \in \mathbb{R}^{n \times n}$ is the matrix obtained by removing the diagonal values corresponding to the h-hop circles. $\bar{D}^h \in \mathbb{R}^{n \times n}$ is the corresponding degree matrix of $\bar{A}^h$. $W \in \mathbb{R}^{d \times 1}$ and $W_b \in \mathbb{R}^{d \times d}$ are the learnable weight matrices. $p \in \mathbb{R}^d$ and $a \in \mathbb{R}^{1 \times 2d}$ are the trainable projection vectors. I is the identity matrix. λ is a trade-off parameter. α is a user-defined hyperparameter. $\mathbf{1}_n \in \mathbb{R}^n$ is a vector whose elements are all equal to 1. $M \in \mathbb{R}^{n \times k}$ is a masking matrix. O is the matrix used for measuring the overlap between node neighbors. σ is the activation function (e.g., tanh). $\odot$ denotes the broadcasted element-wise product. MLP denotes a multi-layer perceptron. SEL_k is the algorithm used for selecting nodes one by one.

Table 3.2: Comparison of Graph Pooling Methods

3.3 Other Pooling

Apart from node drop and node clustering pooling methods, there also exist some other graph pooling methods. For example, *EdgePool* and *HyperDrop* pool the input graph from the edge view, which maintains the connectivity of the input graph through edge contractions based on an adaptive edge scoring design. *MuchPool* combines node clustering pooling and node drop pooling to capture different characteristics of a graph. *PAS-Pooling Architecture Search* proposes searching for adaptive pooling architectures via neural architecture search.

Chapter 4

APPLICATIONS

In general, the applications of GNNs can be grouped into two major scenarios:

1. Structural Scenarios
2. Non-structural Scenarios

4.1 Structural Scenarios

Structural scenarios refer to applications where the data naturally contains explicit relational structures represented in the form of graphs. These scenarios arise in both scientific research and industrial applications. In scientific research, GNNs are widely used in:

- Graph mining,
- Modeling physical systems,
- Modeling chemical and biological systems.

In industrial applications, GNNs are commonly applied to:

- Knowledge graphs,
- Traffic networks,
- Recommendation systems,

- Social networks.

In these applications, nodes represent entities while edges represent relationships or interactions between entities. GNNs effectively learn representations by exploiting both node features and graph structures.

4.1.1 Graph Mining

GNNs are used in graph mining tasks such as graph matching. In graph matching, GNNs learn graph structures and compare similarities between graphs efficiently by embedding similar graphs into nearby positions in a latent feature space, reducing the computational complexity of traditional methods.

4.1.2 Physics

Graph Neural Networks (GNNs) are widely used in physical systems to model objects and their interactions. In these systems, objects are represented as nodes and interactions between them are represented as edges. GNNs help in learning physical laws and predicting future states of the system.

4.1.3 Chemistry and Biology

- **Molecular Fingerprints:** Molecules can be represented as graphs where atoms are nodes and chemical bonds are edges. GNNs learn better molecular fingerprints than traditional hand-crafted methods, improving drug discovery and molecule searching.
- **Chemical Reaction Prediction:** GNNs help predict the products of chemical reactions by modeling molecular structures and transformations.
- **Protein Interface Prediction:** Proteins are modeled as graphs with amino acid residues as nodes. GNNs are used to identify interacting protein regions and improve protein interaction prediction.

- **Biomedical Engineering:** GNNs are applied to protein-protein interaction networks for tasks such as breast cancer subtype classification and prediction of drug side effects.

4.1.4 Knowledge Graph

A Knowledge Graph (KG) represents real-world entities and the relationships between them. GNNs are used in KGs to learn meaningful embeddings, predict missing links, and perform reasoning over connected entities. Since knowledge graphs are heterogeneous graphs with different types of entities and relations, GNNs help capture both graph structure and logical relationships, improving applications such as question answering and information retrieval.

Generative Models

Generative models for graphs are used to learn and generate real-world graph structures such as social networks, chemical molecules, and knowledge graphs. By using deep learning and GNNs, these models can learn the underlying distribution of graphs and create new realistic graph data.

GNNs are used in traffic networks to predict traffic conditions by modeling roads and their connections as graphs. They capture both spatial and temporal dependencies in dynamic traffic systems. Combining GNNs with methods such as LSTMs, temporal convolutions, and attention mechanisms improves traffic flow prediction and transportation management.

GNNs are used in recommendation systems by modeling users and items as graphs. They learn user and item embeddings from interaction data to improve recommendation accuracy. GNN-based methods are especially useful in large-scale platforms for predicting user preferences efficiently.

4.2 Non-structural Scenarios

Non-structural scenarios refer to applications where the relational structure is implicit or not directly available. These scenarios generally include:

4.2.1 Image processing in Computer Vision

Graph Neural Networks (GNNs) are used in image processing by representing images as graphs, where pixels, regions, or objects are treated as nodes and their relationships are treated as edges. This allows GNNs to capture both spatial and semantic relationships within an image. GNNs are applied in tasks such as image classification, object detection, visual reasoning, and semantic segmentation. They are especially useful for handling non-grid image structures and learning long-range dependencies between different parts of an image.

4.2.2 Text processing

In GNN-based text processing, words, sentences, or documents are represented as nodes in a graph. Edges are created based on relationships such as word co-occurrence, semantic similarity, syntactic dependency, or sentence connections.This representation helps GNNs capture contextual and semantic relationships between words more effectively than traditional text representations.

Conclusion

In conclusion, this project studied Graph Neural Networks (GNNs) with focus on graph filtering, pooling techniques, and their applications in real-world problems. The study showed how GNNs effectively learn from graph-structured data by capturing relationships between connected nodes. Filtering and pooling methods helped improve feature extraction and reduce graph complexity for better performance. Applications of GNNs in social networks, healthcare, recommendation systems, and transportation highlighted their importance in modern artificial intelligence. Future research can focus on developing more scalable, efficient, and interpretable GNN models for handling large and dynamic graphs.

Bibliography

1. Yao and Tang, Jiliang.*Deep Learning on Graphs.*Cambridge University Press, 2021.

2. Jie Zhou,Ganqu Cui,Shengding Hu,Zhengyan Zhang,Cheng Yang,Zhiyuan Liu,Lifeng Wang,Changcheng Li,Maosong Sun.*Graph neural networks: A review of methods and applications*,2020

3. Chuang Liu, Yibing Zhan, Jia Wu, Chang Li, Bo Du, Wenbin Hu, Tongliang Liu, and Dacheng Tao.*Graph Pooling for Graph Neural Networks: Progress, Challenges, and Opportunities*, 2022.

A Study on Graph Neural Networks and Graph Pooling

M.Sc. MATHEMATICS

2024-2026

A Study on Graph Neural Networks and Graph Pooling

DISSERTATION

Submitted to the University of Kerala in partial fulfillment of the requirements for the award of Master of Science in Mathematics

Programme Code : 620

Exam Code : 62023402

Course Code : MM 545

Contents

INTRODUCTION

Graph Neural Networks (GNNs) have emerged as a powerful deep learning framework for processing graph-structured data. Unlike traditional neural networks, which are designed for Euclidean data such as images and text, GNNs can effectively capture the complex relationships among interconnected entities. They have found widespread applications in areas such as social network analysis, recommendation systems, molecular property prediction, knowledge graphs, and computational biology. In this Project, we introduce the graph neural network (GNN) frameworks for specifically Node-focused task. Specifically, we introduce two major components: 1) the graph filters, which refines the node features; and 2) the graph pooling, which aims to coarsen the graph and finally generate the graph-level representation. We categorize graph filters as spectral-based and spatial-based filters.We group graph pooling as flat graph pooling and hierarchical graph pooling and introduce representative methods for each group.

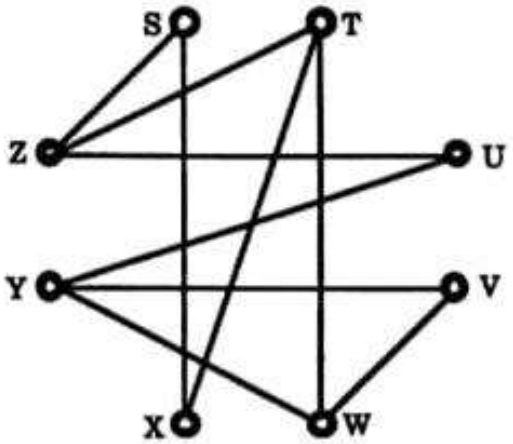
Chapter 1

Preliminaries

1.1 Graph

A graph $G = (V(G), E(G))$ consists of two finite sets:

- $V(G)$, the vertex set of the graph, often denoted by just V, which is a nonempty set of elements called **vertices**, and
- $E(G)$, the edge set of the graph, often denoted by just E, which is a possibly empty set of elements called **edges**, such that each edge $e \in E$ is assigned an unordered pair of vertices (u, v), called the **end vertices** of e.



Thus in the above figure , the vertex set is

$$V = \{S, T, U, V, W, X, Y, Z\}.$$

the edge set E has 10 edges and these edges are assigned the unordered pairs of vertices

$$(S, X),\ (S, Z),\ (T, W),\ (T, X),\ (T, Z),\ (U, Y),\ (U, Z),\ (V, W),\ (V, Y),\ (W, Y).$$

Vertices are also sometimes called points, nodes, or just dots. If e is an edge with end vertices u and v then e is said to join u and v. Note that the definition of a graph allows the possibility of the edge e having identical end vertices, i.e., it is possible to have a vertex u joined to itself by an edge such an edge is called a **loop**. We now give two examples to illustrate the above definitions.

Example 1. Let $G = (V, E)$ where

$$V = \{a, b, c, d, e\}, \qquad E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\},$$

and the ends of the edges are given by:

$$e_1 \leftrightarrow (a, b), \qquad e_2 \leftrightarrow (b, c), \qquad e_3 \leftrightarrow (c, c), \qquad e_4 \leftrightarrow (c, d),$$

$$e_5 \leftrightarrow (b, d), \qquad e_6 \leftrightarrow (d, e), \qquad e_7 \leftrightarrow (b, e), \qquad e_8 \leftrightarrow (b, e).$$

we can represent G Diagramatically as:

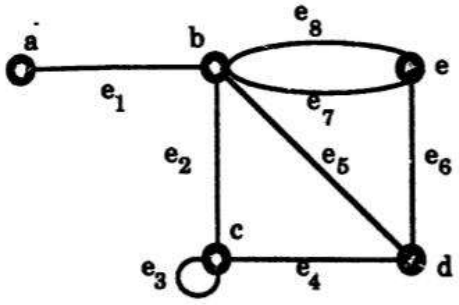


Figure 1.1: A Graph G wwith five vertices and eight edges

1.2 Graphs as Models

Problem 1 :Here is a simple example of a social media network graph

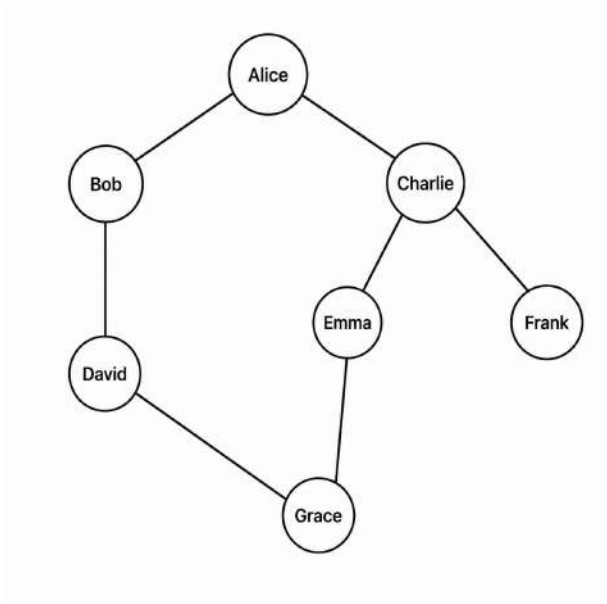


Figure 1.2: Social Media Network Graph

The Vertex set and Edge set are given by:

V ={Alice, Bob, Charlie, David, Emma, Frank, Grace}

E ={(Alice,Bob),(Alice,Charlie),(Bob,David),(Charlie,Emma),(Charlie,Frank),(David,Grace), (Emma,Grace)}

Problem 2 : Suppose that the graph of Figure 1.3 represents a network of telephone lines and poles. We are interested in the network's vulnerability to accidental disruption. We want to identify those lines and poles that must stay in service to avoid disconnecting the network.

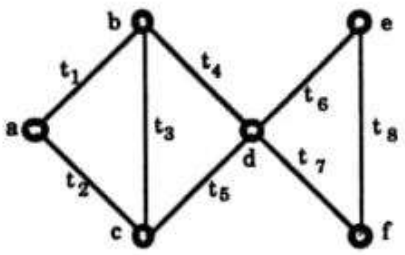


Figure 1.3: A network of telephone lines

There is no single line whose disruption (removal) will disconnect the graph (network), but the graph will become disconnected if we remove the two lines represented by the edges t_4 and t_5, for example. When it comes to poles, the network is more vulnerable since there is a single vertex, d, whose removal disconnects the graph. This illustrates the notions of edge connectivity and vertex connectivity of a graph. We may also want to find a smallest possible set of edges needed to connect the six vertices. There are several such minimal sets. One is

$$\{t_1, t_3, t_5, t_6, t_7\}.$$

1.3 More Definitions

Let G be a graph. If two (or more) edges of G have the same end vertices, then these edges are called **parallel**.

A vertex of G which is not the end of any edge is called **isolated**.

Two vertices which are joined by an edge are said to be **adjacent** or **neighbours**.

The set of all neighbours of a fixed vertex v of G is called the **neighbourhood set** of v and is denoted by $N(v)$.

Let v be a vertex of the graph G. The **degree** $d(v)$ (or $d_G(v)$ if we want to emphasize G) of v is the number of edges of G incident with v, counting each loop twice, i.e., it is the number of times v is an end vertex of an edge.

If $G = (V, E)$ and V has at least two elements (i.e., G has at least two vertices), then for any vertex v of G, $G - v$ denotes the subgraph of G with vertex set $V - \{v\}$ and whose edges are all those of G which are not incident with v, i.e., $G - v$ is obtained from G by removing v and all the edges of G which have v as an end. $G - v$ is referred to as a **vertex deleted subgraph**. If $G = (V, E)$ and e is an edge of G then $G - e$ denotes the subgraph of G having V as its vertex set and $E - \{e\}$ as its edge set, i.e., $G - e$ is obtained from G by removing the edge e (but not its endpoint(s)). $G - e$ is referred to as an **edge deleted subgraph**.

If $G = (V, E)$ and U is a proper subset of V then $G - U$ denotes the subgraph of G with vertex set $V - U$ and whose edges are all those of G which are not incident with any vertex in U.

If F is a subset of the edge set E then $G - F$ denotes the subgraph of G with vertex set V and edge set $E - F$, i.e., obtained by deleting all the edges in F, but not their endpoints. $G - U$ and $G - F$ are also referred to as **vertex deleted** and **edge deleted subgraphs** (respectively).

Hypergraph

A **hypergraph** is a generalization of a graph in which an edge can connect more than two vertices. A hypergraph is denoted by

$$H = (V, E)$$

where

- V is the set of vertices, and
- E is the set of hyperedges such that each hyperedge is a subset of V.

That is,

$$E = \{e_1, e_2, \ldots, e_m\}$$

where

$$e_i \subseteq V.$$

Example: Consider the hypergraph

$$H = (V, E)$$

where

$$V = \{A, B, C, D, E\}$$

and

$$E = \{e_1, e_2, e_3\}$$

such that

$$e_1 = \{A, B, C\},$$

$$e_2 = \{B, D\},$$

$$e_3 = \{C, D, E\}.$$

Here,

- hyperedge e_1 connects the vertices A, B, and C,
- hyperedge e_2 connects the vertices B and D,
- hyperedge e_3 connects the vertices C, D, and E.

1.4 Adjacency and Laplacian Matrix

Adjacency Matrix

Let $G = (V, E)$ be a graph with vertices

$$V = \{v_1, v_2, \ldots, v_n\}.$$

The **adjacency matrix** of G is an $n \times n$ matrix

$$A = [a_{ij}]$$

defined by

$$a_{ij} = \begin{cases} 1, & \text{if there is an edge between } v_i \text{ and } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

Example: Consider the graph with vertex set

$$V = \{P, Q, R, S\}$$

and edge set

$$E = \{(P, Q), (P, R), (Q, R), (R, S)\}.$$

The adjacency matrix of the graph is

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Laplacian Matrix

Let $G = (V, E)$ be an undirected graph.The Laplacian matrix of G is the matrix

$$L = [l_{ij}]$$

defined by

$$l_{ij} = \begin{cases} d(v_i), & \text{if } i = j, \\ -1, & \text{if } v_i \text{ is adjacent to } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

where $d(v_i)$ denotes the degree of the vertex v_i.

Example: For a simple example, take the graph with vertices $1, 2, 3, 4$ and edges

$$(1, 2),\ (1, 3),\ (2, 3),\ (2, 4);$$

For this graph:

- Vertex 1 has degree 2
- Vertex 2 has degree 3
- Vertex 3 has degree 2
- Vertex 4 has degree 1

then the Laplacian is

$$L = \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 2 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}.$$

Diagonal Degree Matrix

The **diagonal degree matrix** of a graph is a diagonal matrix in which each diagonal entry represents the degree of the corresponding vertex of the graph. For a graph with vertices $v_1, v_2, \ldots, v_n$, the diagonal degree matrix is denoted by D and is given by

$$D = \begin{bmatrix} d(v_1) & 0 & 0 & \cdots & 0 \\ 0 & d(v_2) & 0 & \cdots & 0 \\ 0 & 0 & d(v_3) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & d(v_n) \end{bmatrix}$$

where $d(v_i)$ is the degree of vertex v_i.

Example: Consider the graph with vertices v_1, v_2, v_3, v_4 having degrees:

$$d(v_1) = 2, \quad d(v_2) = 3, \quad d(v_3) = 2, \quad d(v_4) = 1$$

Then the diagonal degree matrix is

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Relation Between Laplacian Matrix, Degree Matrix, and Adjacency Matrix

For a graph G, the Laplacian matrix L is defined as

$$L = D - A$$

where:

- L = Laplacian matrix
- D = Diagonal degree matrix
- A = Adjacency matrix

Chapter 2

The General GNN Frameworks

2.1 Introduction To GNNs

Graph Neural Networks (GNNs) are a class of deep learning models specifically designed to process and analyze graph-structured data. In a GNN, the main prediction or graph learning tasks usually fall into three levels: node-level, edge-level, and graph-level. These tasks cover predicting labels or values for individual nodes, relationships between node pairs, or entire graphs respectively.

1. **Node-level tasks**

   Here the model predicts something about each node in the graph. Common examples are node classification, where each node gets a class label, and node regression, where each node gets a continuous value.

   Example: In a social network, predict whether a user belongs to a certain community or whether a molecule atom has a certain property.

2. **Edge-level tasks**

   Here the model predicts something about a pair of nodes, usually the existence or type of a connection between them. The most common example is link prediction, which asks whether two nodes should be connected or how strongly they are related.

Example: Recommend a friendship link in a social network or predict the bond type between two atoms in a molecule.

3. **Graph-level tasks**

   Here the model predicts one label or value for the whole graph. This includes graph classification and graph regression, where the output summarizes the entire structure using information from all nodes and edges. **Example:** Classify a molecule as toxic or non-toxic, or predict a numerical molecular property.

The key idea behind GNNs is to learn meaningful representations of nodes, edges, or entire graphs by iteratively aggregating information from neighboring nodes.Here the information refers to **Node features** and **Edge features**.

Let $G = (V, E)$ be a graph, where V is the set of nodes and E is the set of edges.

Node Features

For each node $v_i \in V$, the node feature is a vector

$$x_i \in \mathbb{R}^{d_n},$$

where

$$d_n = \text{number of features (or attributes) for each node.}$$

If there are $N = |V|$ nodes, all node features can be stacked into a matrix

$$X \in \mathbb{R}^{N \times d_n},$$

where,the i-th row is x_i^T.

Edge Features

For each edge $(v_i, v_j) \in E$, the edge feature is a vector

$$e_{ij} \in \mathbb{R}^{d_e},$$

where,

$$d_n = \text{number of features (or attributes) for each edge.}$$

If the graph has $M = |E|$ edges, all edge features can be collected into an edge-feature set or edge-feature matrix, depending on the implementation, with each edge carrying its own attribute vector in $\mathbb{R}^{d_e}$.

2.2 General Structure of a Graph Neural Network

The general structure of a Graph Neural Network (GNN) is similar to that of a traditional neural network. A GNN typically consists of three major components:

- Input Layer
- Hidden Layers
- Output Layer

Input Layer

The input layer receives the graph data. A graph is generally represented as

$$G = (V, E),$$

where V denotes the set of nodes and E denotes the set of edges. Each node in the graph is associated with a feature vector. The collection of all node features is represented by the feature matrix

$$H \in \mathbb{R}^{n \times d},$$

where:

- n is the number of nodes,
- d is the dimension of node features.

Hidden Layers

The hidden layers perform graph filtering or message passing operations. In each hidden layer, a node gathers information from its neighboring nodes and updates its representation.

Suppose:

- $H^{(k)}$ = node feature matrix at layer k
- $H^{(k+1)}$ = updated feature matrix at layer $k+1$

The transformation is:

$$H^{(k+1)} = \sigma\left(\text{AGGREGATE}(H^{(k)})W^{(k)} + b^{(k)}\right)$$

where:

- AGGREGATE = neighbor aggregation/filtering
- $W^{(k)}$ = learnable weight matrix
- $b^{(k)}$ = bias vector
- σ = activation function

In GNNs, the most commonly used activation function is ReLU.

$$\text{ReLU}(x) = \max(0, x)$$

so it outputs the input if it is positive and 0 if it is negative. The hidden layers help the network learn meaningful structural and feature representations from the graph.

Output Layer

After several GNN layers, we obtain:

$$H^{(K)}$$

where:

- K = final GNN layer
- $H^{(K)}$ = final node embeddings

Each node embedding produces a node prediction:

$$\hat{Y} = \text{softmax}\left(H^{(K)}W_o\right)$$

Here softmax is an activation function which converts raw output values into probabilities. After the GNN makes predictions,we find loss function that measures the error, and backward propagation updates the weights to reduce that error.

Loss Function

The loss function measures How far the predictions are from the true values. It gives a single numerical value called the loss. Smaller loss means better predictions.Commonly used Loss Functions are Mean Squared Error,Mean Absolute Error

Now we compute

$$\frac{\partial \mathcal{L}}{\partial W^{(k)}}$$

This is called the gradient of the loss with respect to the weight matrix $W^{(k)}$. This gradient tells how sensitive the loss is to changes in the weights. The magnitude of the gradient shows how strongly a weight affects the loss. If the gradient is large, even a small change in the weight can significantly change the loss value. This means that the corresponding weight has a strong influence on the prediction error. On the other hand, if the gradient is small, changing that weight has very little effect on the loss, so the weight is less important for the current prediction. The sign of the gradient determines the direction in which the weight should move. If the gradient is positive, increasing the weight increases the loss, so the weight should be decreased. If the gradient is negative, increasing the weight decreases the loss, so the weight should be increased. In this way, the network knows how to modify its parameters to improve predictions. After computing

the gradients, the weights are updated using gradient descent:

$$W^{(k)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(k)}} \rightarrow W^{(k)}$$

where η is the learning rate.

In practice, the learning rate is usually selected experimentally. The learning rate controls the size of the update step. If the learning rate is very small, the weights change only slightly during each update. Training becomes slow, and the network may take a long time to reduce the loss.If the learning rate is very large, the weights change too much in each step. By repeatedly updating the weights in the direction that reduces the loss, the GNN gradually learns better node representations and improves its performance over time.

2.3 Graph Filters

There are various perspectives to design graph filters, which can be roughly split into two categories: (1) spectral-based graph filters and (2) spatial-based graph filters.

The spatial-based graph filters explicitly leverage the graph structure (i.e., the connections between the nodes) to perform the feature refining process in the graph domain. In contrast, the spectral-based graph filters utilize spectral graph theory to design the filtering operation in the spectral domain. These two categories of graph filters are closely related. Especially, some of the spectral-based graph filters can be regarded as spatial-based filters. In this section, we introduce spectral-based graph filters and explain how some of the spectral-based graph filtersand then discuss spatial-based graph filters.

2.3.1 Spectral-based Graph Filters

The spectral-based graph filters are designed in the spectral domain of graph signals. We first introduce the graph spectral filtering and then describe how it can be adopted to design spectral-based graph filters.

Spectral Graph Filtering

The idea of the graph spectral filtering is to modulate the frequencies of a graph signal such that some of its frequency components are kept/amplified while others are removed/diminished. Hence, given a graph signal $f \in \mathbb{R}^N$, we need first to apply the Graph Fourier Transform (GFT) on it to obtain its graph Fourier coefficients, and then modulate these coefficients before reconstructing the signal in the spatial domain. For a signal $f \in \mathbb{R}^N$ defined on a graph G, its Graph Fourier Transform is defined as follows:

$$\hat{f} = U^\top f, \tag{2.1}$$

where U consists of the eigenvectors of the Laplacian matrix of G and $\hat{f}$ is the obtained graph Fourier coefficients for the signal f. These graph Fourier coefficients describe how each graph Fourier component contributes to the graph signal f. Specifically, the i-th element of $\hat{f}$ corresponds to the i-th graph Fourier component u_i with the frequency λ_i. Note that λ_i is the eigenvalue corresponding to u_i. To modulate the frequencies of the signal f, we filter the graph Fourier coefficients as follows:

$$\hat{f}'[i] = \hat{f}[i] \cdot \gamma(\lambda_i), \qquad \text{for } i = 1, \ldots, N. \tag{2.2}$$

where $\gamma(\lambda_i)$ is a function with the frequency λ_i as input which determines how the corresponding frequency component should be modulated. This process can be expressed in a matrix form as follows:

$$\hat{f}' = \gamma(\Lambda) \cdot \hat{f} = \gamma(\Lambda) \cdot U^\top f, \tag{2.3}$$

where Λ is a diagonal matrix consisting of the frequencies (eigenvalues of the Laplacian matrix) and $\gamma(\Lambda)$ is obtained by applying the function $\gamma(\cdot)$ to each element on the diagonal

of Λ. Formally, Λ and $\gamma(\Lambda)$ can be represented as follows:

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \lambda_N \end{pmatrix}, \qquad \gamma(\Lambda) = \begin{pmatrix} \gamma(\lambda_1) & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \gamma(\lambda_N) \end{pmatrix} \tag{2.4}$$

With the filtered coefficients, we can now reconstruct the signal to the graph domain using the Inverse Graph Fourier Transform (IGFT) as follows:

$$f' = U\hat{f}' = U \cdot \gamma(\Lambda) \cdot U^\top f, \tag{2.5}$$

where f' is the obtained filtered graph signal. The filtering process can be regarded as applying the operator $U \cdot \gamma(\Lambda) \cdot U^\top$ to the input graph signal. For convenience, we sometimes refer to the function $\gamma(\Lambda)$ as the *filter* since it controls how each frequency component of the graph signal f is filtered.

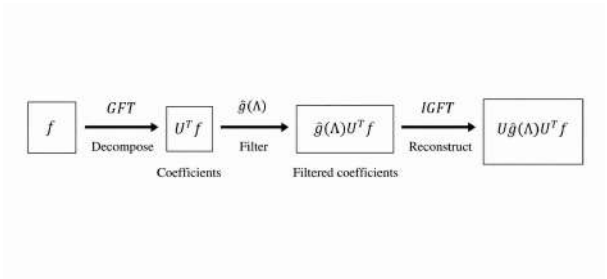


Figure 2.1: The Process of Spectral Filtering

We have introduced the graph spectral filtering operator, which can be used to filter certain frequencies in the input signal. For example, if we want to get a smooth signal after filtering, we can design a low-pass filter, where $\gamma(\lambda)$ is large when λ is small, and it is small when λ is large. In this way,the obtained filtered signal is smooth since it majorly contains the low-frequency part of the input signal. However, when utilizing the spectral-based filter as a graph filter in graph neural networks, we often do not know which frequencies are more important. Hence, just like the classical neural networks,

the graph filters can be learned in a data-driven way. Specifically, we can model $\gamma(\Lambda)$ with certain functions and then learn the parameters with the supervision from data. A natural attempt is to give full freedom when designing $\gamma(\cdot)$ (or a non-parametric model). In detail, the function $\gamma(\cdot)$ is defined as follows:

$$\gamma(\lambda_l) = \theta_l,$$

where θ_l is a parameter to be learned from the data. It can also be represented in a matrix form as follows:

$$\gamma(\Lambda) = \begin{pmatrix} \theta_1 & 0 & \cdots & 0 \\ 0 & \theta_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \theta_N \end{pmatrix}$$

However, there are some limitations with this kind of filter. First, the number of parameters to be learned is equal to the number of nodes N, which can be extremely large in real-world graphs. Hence, it requires lots of memory to store these parameters and also abundant data to fit them. Second, the filter $U \cdot \gamma(\Lambda) \cdot U^\top$ is likely to be a dense matrix. Therefore, the calculation of the i-th element of the output signal f' could relate to all the nodes in the graph. In other words, the operator is not spatially localized. Furthermore, the computational cost for this operator is quite expensive due to the eigen decomposition of the Laplacian matrix and the matrix multiplication between dense matrices when calculating $U \cdot \gamma(\Lambda) \cdot U^\top$. To address these issues, a polynomial filter operator, which we denoted as Poly-Filter. The function $\gamma(\Lambda)$can be modeled with a K-order truncated polynomial as follows:

$$\gamma(\lambda_l) = \sum_{k=0}^{K} \theta_k \lambda_l^k$$

In terms of the matrix form, it can be rewritten as below:

$$\gamma(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k$$

Clearly, the number of the parameters in above two equations is $K + 1$, which is not dependent on the number of nodes in the graph. Furthermore, we can show that $U \cdot \gamma(\Lambda) \cdot U^\top$ can be simplified to be a polynomial of the Laplacian matrix. It means that:

1. no eigendecomposition is needed; and

2. the polynomial parametrized filtering operator is spatially localized, i.e., the calculation of each element of the output f' only involves a small number of nodes in the graph.

Next, we first show that the Poly-Filter operator can be formulated as a polynomial of the Laplacian matrix and then understand it from a spatial perspective. By applying this Poly-Filter operator on f, according to Eq. (2.5), we can get the output f' as follows:

$$\begin{aligned} f' &= U \cdot \gamma(\Lambda) \cdot U^\top f \\ &= U \cdot \sum_{k=0}^{K} \theta_k \Lambda^k \cdot U^\top f \\ &= \sum_{k=0}^{K} \theta_k \, U \cdot \Lambda^k \cdot U^\top f \end{aligned}$$

2.3.2 Spatial-based Graph Filters

Spatial graph filtering processes graph data directly in the graph domain by updating each node using information from its neighboring nodes. In this approach, every node collects and combines features from the nodes connected to it through the graph structure. Unlike spectral methods, spatial filtering does not require transforming the graph signal into another domain such as the Fourier domain. Instead, the computation is performed locally on the graph itself. A spatial graph filter is a function that computes a node's feature vector by combining its own features with the features of its neighbors according to the graph connectivity.

$$h_v^{(l+1)} = \phi\left(h_v^{(l)}, \mathrm{AGG}\left\{h_u^{(l)} : u \in \mathcal{N}(v)\right\}\right)$$

where:

- $h_v^{(l)}$ is the feature vector of node v at layer l,
- $\mathcal{N}(v)$ denotes the neighborhood of node v,
- AGG represents an aggregation function such as sum, mean, or max,
- ϕ denotes a **learnable transformation**(Transformation containing trainable parameters such as weight matrices and biases that are learned during training.)

The above equation shows that the representation of node v at layer $l+1$ is obtained by combining its current feature vector with the aggregated information from its neighboring nodes.

Chapter 3

Graph Pooling

The graph filters refine the node features without changing the graph struc- ture. After the graph filter operation, each node in the graph has a new feature representation. Typically, the graph filter operations are sufficient for node- focused tasks that take advantage of the node representations. However, for graph-focused tasks, a representation of the entire graph is desired. To obtain such representation, we need to summarize the information from the nodes. There are two main kinds of information that are important for generating the graph representation is the node features, and the other is the graph structure. The graph representation is expected to preserve both the node fea- ture information and the graph structure information. Similar to the classical convolutional neural networks, graph pooling layers are proposed to generate graph-level representations. The early designs of graph pooling layers are usually flat. In other words, they generate the graph-level representation directly from the node representations in a single step. For example, the average pooling layers and max-pooling layers can be adapted to graph neural networks by applying them to each feature. Later on, hierarchical graph pooling designs have been developed to summa- rize the graph information by coarsening the original graph step by step. In the hierarchical graph pooling design, there are often several graph pooling layers, each of which follows a stack of several filters.

3.1 Problem Formulation

Notions. Let $G = (V, E)$ denote a graph with node set V and edge set E. Node features are denoted as $X \in \mathbb{R}^{n \times d}$, where n is the number of nodes, and d is the dimension of node features. The adjacency matrix is defined as $A \in \{0, 1\}^{n \times n}$. $A[i, j] = 1$ if there exists an edge between node v_i and node v_j, otherwise, $A[i, j] = 0$.

Graph Pooling. Let a graph pooling operator be defined as any function POOL that maps a graph G to a new pooled graph $G' = (V', E')$:

$$G' = \text{POOL}(G), \tag{3.1}$$

where $|V'| < |V|$.[1] The primary goal of graph pooling is to reduce the number of nodes in a graph while preserving the semantic information of the graph.

3.2 Approaches for Graph Pooling

Graph pooling can be roughly divided into flat pooling and hierarchical pooling according to its role in graph-level rep- resentation learning. The former directly generates graph-level representations in a single step ($|V'| = 1$), while the latter coarsens the graph gradually into a smaller sized graph ($|V'| > 1$).

3.2.1 Global Pooling(Flat Pooling)

The global graph pooling operators are employed as readout functions to transform the graph into a single low-dimensional dense vector.In Global pooling layers, there is no new graph but a single node being generated.In practical usage, global pooling generally has wider applications than hierarchical pooling. Based on complexity and representation capability, representative methods of global pooling operators can be categorized into four categories, including simple functions, grouping/cascading, attention, and learnable

[1]The pooled graph contains fewer nodes than the original graph.

functions. Simple functions refer to simple permutation-invariant functions,), and Grouping/Cascading refers to a class of graph representation methods in grouping or cascading nodes. Attention refers to the method of weighted summation of node representations, with the weights typically filled by attention coefficients. Learnable functions are parametric approaches, particularly utilizing neural networks.

Simple permutation-invariant functions

Simple permutation-invariant functions are graph pooling methods that aggregate node features without considering how the nodes are connected. In other words, these methods produce the same output regardless of the order in which the nodes are arranged. The three most common permutation-invariant functions are:

- **Sum Pooling:**

$$\mathbf{h}_G = \sum_{i=1}^{N} \mathbf{h}_i \tag{3.2}$$

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

- **Mean Pooling:**

$$\mathbf{h}_G = \frac{1}{N} \sum_{i=1}^{N} \mathbf{h}_i \tag{3.3}$$

  This computes the average of all node feature vectors.

- **Max Pooling:**

$$\mathbf{h}_G^{(j)} = \max_{i=1,\ldots,N} \mathbf{h}_i^{(j)}, \qquad j = 1, \ldots, d \tag{3.4}$$

  where d is the feature dimension, and the maximum is taken independently for each feature dimension.

Among these, **Sum** and **Mean** pooling were the earliest global pooling techniques introduced in graph neural networks. They are computationally efficient and straightforward to implement, making them popular baseline methods for generating a graph-level representation. However, because they ignore the graph's structural connectivity, they may fail

to capture important topological information required for more complex graph learning tasks.

3.2.2 Hierarchical Pooling

Hierarchical pooling methods aim to preserve the hierarchical graph's structural information by iteratively coarsening the graph into a new graph of smaller size. The hierarchical pooling can be roughly classified into node clustering pooling,node drop pooling, and other pooling according to the manner in which it coarsens a graph. The main difference between the first two types of methods is that node clustering pooling generates new nodes for the coarsened graph, whereas node drop pooling retains nodes from the original graph. Note that hierarchical pooling methods still technically employ flat pooling methods to obtain the graph-level representation of the coarsened graph.

Node Clustering Pooling

Based on the assumption that each node is a member of a potentially significant substructure, clustering pooling transfers nodes from the input graph to nodes of the coarsened graph. The nodes in the coarsened graph can also be referred to as **supernodes**, and the coarsened graph can also be referred to as a **hypergraph**, since each node represents a substructure of the original graph, where clustering pooling operators can benefit from existing community detection and graph clustering algorithms. A real-life application analogous to this assumption can be found in molecular structures, where functional groups are considered as communities of atoms, and molecules are assemblies of several functional groups. In addition to discovering the alignment between nodes and supernodes, clustering pooling requires learning the representation of supernodes and determining the links between supernodes. The last two tasks can be summarized as learning hypergraphs, whose central task is node clustering. Formally, a generic computational flow for clustering pooling can be described in the following steps. **Step 1:**

Node Clustering Given a potential hypergraph G_{hyper} with node set V_{hyper}, we assume

$$|V_{\text{hyper}}| < |V|$$

and define a surjection

$$f_{\text{clus}} : V \to V_{\text{hyper}}$$

also known as the **vertex mapping function** or **clustering function**, in which each node in V corresponds to at least one node in V_{hyper}. The variation in the clustering function f_{clus} results in significant differences in clustering pooling. Furthermore, permutation invariance of the clustering functions is required for clustering pooling permutation invariance. An explicit or implicit cluster assignment matrix

$$S \in \mathbb{R}^{|V| \times |V_{\text{hyper}}|}$$

with each row representing a node and each column representing a supernode, can be used to describe the outcome of node clustering.

remark 3.2.1 *A **cluster-assignment matrix** is a rectangular binary (or nonnegative) matrix that encodes which data point belongs to which cluster. Each row corresponds to a data point and each column corresponds to a cluster. The entry*

$$A_{ij} = \begin{cases} 1, & \textit{if data point } i \textit{ is assigned to cluster } j, \\ 0, & \textit{otherwise.} \end{cases}$$

More generally, A_{ij} may take a positive value representing a weighted assignment. An

example of a cluster-assignment matrix is

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

where 5 data points are assigned to 3 clusters.

The pooling ratio r is defined as the ratio of the number of clusters to the number of nodes:

$$r = \frac{|V_{\text{hyper}}|}{|V|}$$

The pooling ratio is a hyperparameter in both deterministic algorithms and parametric networks, since it determines the size of the hypergraph, which affects the computational complexity of the algorithm and the amount of preserved information.

Step 2: Learning Hypergraph(Graph Coarsening)

Step 1 specifies which supernodes are included in the hypergraph, but it does not determine what features the supernodes possess or how they are linked to one another. Hence, the hypergraph created in the previous step is revised to include the following two components. **Step 2.1: Learning the Representation of Supernodes.**

This step can be considered as a local readout operation, which can use any readout func-tion One of the classical methods is to use the cluster assignment matrix to conduct a weighted summation where Z^{l+1} is the representation matrix of supernodes that has not yet performed message passing.

$$Z^{l+1} = S^T H^l, \qquad Z^{l+1} \in \mathbb{R}^{N_{l+1} \times d}$$

Step 2.2: Learning the Hyperedges.

Intuitively, if two clusters are adjacent, then at least one node in each cluster is adjacent

Models	CAM Generator	Graph Coarsening
DiffPool	$\mathrm{C} = \text{softmax}\left(\mathrm{GNN}_{\text{pool}}(X, A)\right)$	$\begin{cases} X' = C^T \cdot \mathrm{GNN}_{\text{emb}}(X, A) \\ A' = C^T AC \end{cases}$
NMF	$\begin{cases} A \approx UV \\ C = V^T \end{cases}$	$\mathrm{X'} = \mathrm{C}^T X, \qquad A' = C^T AC$
LaPool	$\begin{cases} h_i = \left\| \sum_{j \in \mathcal{N}(v_i)} A_{ij}(x_i - x_j) \right\|_2 \\ V_e = \{v_i \in V \mid \forall v_j,\ h_i - A_{ij} s_j > 0\} \\ C = \text{sparsemax}\left(p \frac{XX^T}{\|X\| \|X_c\|}\right) \end{cases}$	$\begin{cases} X' = \mathrm{MLP}(C^T X) \\ A' = C^T AC \end{cases}$
MinCut	C=MLP(X)	$\begin{cases} X' = C^T X, \quad \hat{A} = C^T \tilde{A} C \\ A' = \hat{A} - I\,\mathrm{diag}(\hat{A}) \end{cases}$
StructPool	C:Minimize E(C)	$\mathrm{X'}=\mathrm{C}^T X, \qquad A' = C^T AC$
MemPool	$\begin{cases} C_{ij} = \frac{(1 + \|x_i - k_j\|^2/\tau)^{-\frac{\tau+1}{2}}}{\sum_j (1 + \|x_i - k_j\|^2/\tau)^{-\frac{\tau+1}{2}}} \\ C = \text{softmax}\left(\Gamma\left(\Big\|_{m=0}^{\lvert m \rvert} C_i\right)\right) \end{cases}$	$\mathrm{X'}=\mathrm{MLP}(\mathrm{C}^T X)$
HAP	$\begin{cases} T = \mathrm{GCont}(X) \\ C_{ij} = \sigma\left(p^T [h_{u_m}, h_{u_l}]\right) \\ C = \text{softmax}(C) \end{cases}$	$\begin{cases} X' = \mathrm{MLP}(C^T X), \quad \hat{A} = C^T AC \\ A' = \text{Gumbel-SoftMax}(\hat{A}) \end{cases}$
SEP	C:Minimize $\mathrm{H}^T(\mathcal{G})$	$\mathrm{X'}=\mathrm{C}^T X, \qquad A' = C^T AC$

Notations: $X' \in \mathbb{R}^{c \times d}$ $and A' \in \mathbb{R}^{c \times c}$are the adjacency matrix and feature matrix for the new graph, respectively; $\tilde{A} \in \mathbb{R}^{n \times n}$ is the adjacency matrix with self-loop; $L \in \mathbb{R}^{n \times n}$ is the graph Laplacian; $I_k \in \mathbb{R}^{c \times c}$ is the identity matrix; $k \in \mathbb{R}^d$ is a memory key vector; $p \in \mathbb{R}^{1 \times 2d}$ is a trainable vector; τ is the degree of freedom of the Student's t-distribution, i.e., temperature; c is the number of clusters; $|m|$ is the number of heads; Γ is an $[1 \times 1]$ convolutional operator; $\|$ is the concatenation operator; GCont is an auto-learned global graph content; and Gumbel-SoftMax achieves soft sampling for neighborhood relationships to decrease the edge density.

Table 3.1: Summary of representative node clustering pooling methods.

to a node in the other cluster. Similarly, the transformation of a node adjacency matrix yields the adjacency matrix of a supernode. Hence, the hypergraph adjacency matrix A^{l+1} can be calculated using the original graph adjacency matrix A^l and the cluster assignment matrix S, as shown below:

$$A^{l+1} = S^T A^l S$$

More graph coarsening methods are given in the table 3.1 .

Node Drop Pooling

Node drop pooling exploits learnable scoring functions to delete nodes with comparatively lower significance scores. For a thorough analysis of node drop pooling, we propose a universal and modularized framework, which consists of three disjoint modules:

1. **Score Generator.** Given an input graph, the score generator calculates significance scores for each node.

2. **Node Selector.** The node selector selects the nodes with top-k significance scores.

3. **Graph Coarsening.** With the selected nodes, a new graph coarsened from the original one is obtained by learning a new feature matrix and an adjacency matrix.

The process can be formulated as follows:

$$S^{(l)} = \underbrace{\text{SCORE}\big(X^{(l)}, A^{(l)}\big)}_{\text{Score Generator}}$$

$$\text{idx}^{(l+1)} = \underbrace{\text{TOP}_k\big(S^{(l)}\big)}_{\text{Node Selector}}$$

$$X^{(l+1)}, A^{(l+1)} = \underbrace{\text{COARSEN}\Big(X^{(l)}, A^{(l)}, S^{(l)}, \text{idx}^{(l+1)}\Big)}_{\text{Graph Coarsening}}$$

where functions

$$\text{SCORE}, \quad \text{TOP}_k, \quad \text{and} \quad \text{COARSEN}$$

are specially designed by each method for the score generator, node selector, and graph coarsening modules, respectively.

$$S^{(l)} \in \mathbb{R}^{n \times 1}$$

indicates the significance scores. The operator

$$\text{TOP}_k$$

ranks the values and returns the indices of the largest k values in $S^{(l)}$. The index set

$$\text{idx}^{(l+1)}$$

indicates the reserved node indices for the new graph.

Models	Score Generator	Node Selector	Graph Coarsening
TopKPool	$S = Xp/\|\|p\|\|_2$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot \sigma(S_{\text{idx}})$ $A' = A_{\text{idx,idx}}$
SAGPool	$S = \text{GNN}(X, A)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$
AttPool	$S = \text{softmax}(XW)$	$\text{idx} = \text{TOP}_k(S)$	$X' = A_{\text{idx}}(X \odot S)$ $A' = A_{\text{idx}} A A_{\text{idx}}^T$
ASAP	$S = LEConv(X^e, A)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X^e_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx}} A A_{\text{idx}}^T$
HGP-SL	$S = \|\|(I - D^{-1}A)X\|\|_1$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $\hat{A} = A_{\text{idx,idx}}$
VIPool	$P = \frac{1}{t}\sum_{h=1}^{t}(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}})^h W^h \text{MLP}(X)$ $S = \sigma(\text{MLP}(\text{MLP}(X, P)))$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = \text{softmax}(A_{\text{idx}}) A \text{softmax}(A_{\text{idx}})^T$
GSAPool	$S_1 = \text{GNN}(X, A)$ $S_2 = \sigma(\text{MLP}(X))$ $S = \alpha S_1 + (1-\alpha) S_2$	$\text{idx} = \text{TOP}_k(S)$	$X' = (AXW)_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$
PANPool	$Z_\mu = \sum_{n=0}^{L} e^{-\mu} \sum_{j=1}^{N} g(i,j,n)$ $M = Z^{-1} \sum_{n=0}^{L} e^{-\mu \frac{n}{L}} A^n$ $S = Xp + \beta\, \text{diag}(M)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot \sigma(S_{\text{idx}})$ $A' = A_{\text{idx,idx}}$
CGIPool	$S_p = \text{GNN}_p(X, A)$ $S_f = \text{GNN}_f(X, A)$ $S = \sigma(S_p - S_f)$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$
TAPool	$S_t = \text{softmax}\left(\frac{1}{n}((XX^T) \odot (\tilde{D}^{-1}\tilde{A}))1_n\right)$ $S_g = \text{softmax}(\tilde{D}^{-1}\tilde{A}Xp)$ $S = S_t + S_g$	$\text{idx} = \text{TOP}_k(S)$	$X' = X_{\text{idx}} \odot S_{\text{idx}}$ $A' = A_{\text{idx,idx}}$

Notations: $X' \in \mathbb{R}^{k \times d}$ and $A' \in \{0,1\}^{k \times k}$ are the feature matrix and adjacency matrix of the new graph, respectively. $D \in \mathbb{R}^{n \times n}$ is the degree matrix of A. $\bar{A}^h \in \mathbb{R}^{n \times n}$ is the matrix obtained by removing the diagonal values corresponding to the h-hop circles. $\bar{D}^h \in \mathbb{R}^{n \times n}$ is the corresponding degree matrix of $\bar{A}^h$. $W \in \mathbb{R}^{d \times 1}$ and $W_b \in \mathbb{R}^{d \times d}$ are the learnable weight matrices. $p \in \mathbb{R}^d$ and $a \in \mathbb{R}^{1 \times 2d}$ are the trainable projection vectors. I is the identity matrix. λ is a trade-off parameter. α is a user-defined hyperparameter. $\mathbf{1}_n \in \mathbb{R}^n$ is a vector whose elements are all equal to 1. $M \in \mathbb{R}^{n \times k}$ is a masking matrix. O is the matrix used for measuring the overlap between node neighbors. σ is the activation function (e.g., tanh). $\odot$ denotes the broadcasted element-wise product. MLP denotes a multi-layer perceptron. SEL_k is the algorithm used for selecting nodes one by one.

Table 3.2: Comparison of Graph Pooling Methods

3.3 Other Pooling

Apart from node drop and node clustering pooling methods, there also exist some other graph pooling methods. For example, *EdgePool* and *HyperDrop* pool the input graph from the edge view, which maintains the connectivity of the input graph through edge contractions based on an adaptive edge scoring design. *MuchPool* combines node clustering pooling and node drop pooling to capture different characteristics of a graph. *PAS-Pooling Architecture Search* proposes searching for adaptive pooling architectures via neural architecture search.

Chapter 4

APPLICATIONS

In general, the applications of GNNs can be grouped into two major scenarios:

1. Structural Scenarios
2. Non-structural Scenarios

4.1 Structural Scenarios

Structural scenarios refer to applications where the data naturally contains explicit relational structures represented in the form of graphs. These scenarios arise in both scientific research and industrial applications. In scientific research, GNNs are widely used in:

- Graph mining,
- Modeling physical systems,
- Modeling chemical and biological systems.

In industrial applications, GNNs are commonly applied to:

- Knowledge graphs,
- Traffic networks,
- Recommendation systems,

- Social networks.

In these applications, nodes represent entities while edges represent relationships or interactions between entities. GNNs effectively learn representations by exploiting both node features and graph structures.

4.1.1 Graph Mining

GNNs are used in graph mining tasks such as graph matching. In graph matching, GNNs learn graph structures and compare similarities between graphs efficiently by embedding similar graphs into nearby positions in a latent feature space, reducing the computational complexity of traditional methods.

4.1.2 Physics

Graph Neural Networks (GNNs) are widely used in physical systems to model objects and their interactions. In these systems, objects are represented as nodes and interactions between them are represented as edges. GNNs help in learning physical laws and predicting future states of the system.

4.1.3 Chemistry and Biology

- **Molecular Fingerprints:** Molecules can be represented as graphs where atoms are nodes and chemical bonds are edges. GNNs learn better molecular fingerprints than traditional hand-crafted methods, improving drug discovery and molecule searching.
- **Chemical Reaction Prediction:** GNNs help predict the products of chemical reactions by modeling molecular structures and transformations.
- **Protein Interface Prediction:** Proteins are modeled as graphs with amino acid residues as nodes. GNNs are used to identify interacting protein regions and improve protein interaction prediction.

- **Biomedical Engineering:** GNNs are applied to protein-protein interaction networks for tasks such as breast cancer subtype classification and prediction of drug side effects.

4.1.4 Knowledge Graph

A Knowledge Graph (KG) represents real-world entities and the relationships between them. GNNs are used in KGs to learn meaningful embeddings, predict missing links, and perform reasoning over connected entities. Since knowledge graphs are heterogeneous graphs with different types of entities and relations, GNNs help capture both graph structure and logical relationships, improving applications such as question answering and information retrieval.

Generative Models

Generative models for graphs are used to learn and generate real-world graph structures such as social networks, chemical molecules, and knowledge graphs. By using deep learning and GNNs, these models can learn the underlying distribution of graphs and create new realistic graph data.

GNNs are used in traffic networks to predict traffic conditions by modeling roads and their connections as graphs. They capture both spatial and temporal dependencies in dynamic traffic systems. Combining GNNs with methods such as LSTMs, temporal convolutions, and attention mechanisms improves traffic flow prediction and transportation management.

GNNs are used in recommendation systems by modeling users and items as graphs. They learn user and item embeddings from interaction data to improve recommendation accuracy. GNN-based methods are especially useful in large-scale platforms for predicting user preferences efficiently.

4.2 Non-structural Scenarios

Non-structural scenarios refer to applications where the relational structure is implicit or not directly available. These scenarios generally include:

4.2.1 Image processing in Computer Vision

Graph Neural Networks (GNNs) are used in image processing by representing images as graphs, where pixels, regions, or objects are treated as nodes and their relationships are treated as edges. This allows GNNs to capture both spatial and semantic relationships within an image. GNNs are applied in tasks such as image classification, object detection, visual reasoning, and semantic segmentation. They are especially useful for handling non-grid image structures and learning long-range dependencies between different parts of an image.

4.2.2 Text processing

In GNN-based text processing, words, sentences, or documents are represented as nodes in a graph. Edges are created based on relationships such as word co-occurrence, semantic similarity, syntactic dependency, or sentence connections.This representation helps GNNs capture contextual and semantic relationships between words more effectively than traditional text representations.

Conclusion

In conclusion, this project studied Graph Neural Networks (GNNs) with focus on graph filtering, pooling techniques, and their applications in real-world problems. The study showed how GNNs effectively learn from graph-structured data by capturing relationships between connected nodes. Filtering and pooling methods helped improve feature extraction and reduce graph complexity for better performance. Applications of GNNs in social networks, healthcare, recommendation systems, and transportation highlighted their importance in modern artificial intelligence. Future research can focus on developing more scalable, efficient, and interpretable GNN models for handling large and dynamic graphs.

Bibliography

1. Yao and Tang, Jiliang.*Deep Learning on Graphs.*Cambridge University Press, 2021.

2. Jie Zhou,Ganqu Cui,Shengding Hu,Zhengyan Zhang,Cheng Yang,Zhiyuan Liu,Lifeng Wang,Changcheng Li,Maosong Sun.*Graph neural networks: A review of methods and applications*,2020

3. Chuang Liu, Yibing Zhan, Jia Wu, Chang Li, Bo Du, Wenbin Hu, Tongliang Liu, and Dacheng Tao.*Graph Pooling for Graph Neural Networks: Progress, Challenges, and Opportunities*, 2022.