Bios 740- Chapter 5. Graph Neural Networks: GNN, GCN

Acknowledgement: Many thanks to Mr. Shuai Huang for preparing some of these slides, and to Drs. Shuiwang Ji, Qiao Liu, and Jiliang Tang for sharing their slides. I also drew on material from the Stanford CS224w lecture presentations, Dr. Liu's presentation, and Dr. Tang's AAAI lecture, as well as content generated by ChatGPT.

Content

Content

Graph-Structured Data

Graph-structured data is a type of data representation where **entities (nodes)** and their **relationships (edges)** are explicitly modeled as a graph. This structure captures the connections between data points, allowing for more effective analysis of relational patterns.

Examples:

- Social networks, citation networks, multi-agent systems
- Knowledge graphs
- Recommendation System
- Protein interaction networks
- Molecules
- Road maps
- Brain networks

Graphs are a general language for describing and analyzing entities with relations/interactions

Why Are Graph-structured Data Important?

Graphs capture **complex relationships and dependencies** between entities:

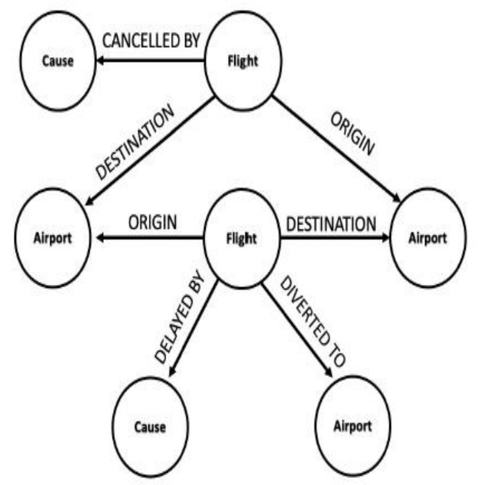
•**Interconnected entities influence each other** (e.g., in social networks, a person's behavior depends on their connections).

•**Knowledge is structured in relational forms** (e.g., in knowledge graphs, concepts are linked based on meaning and context).

•**Biological and medical data exhibit intricate interactions** (e.g., protein-protein interaction networks, brain connectivity graphs).

By modeling data as graphs, we can better understand structures, uncover hidden patterns, and improve AI-driven decision-making.

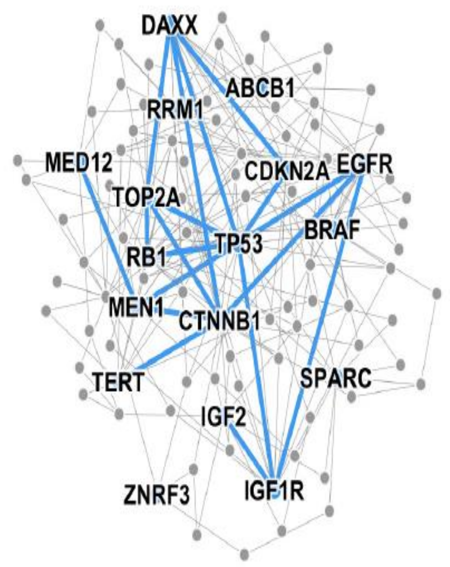
Graph-Structured Data is Everywhere



Event Graphs

Image credit: SalientNetworks

Computer Networks



Disease Pathways

Image credit: Medium

Social Networks

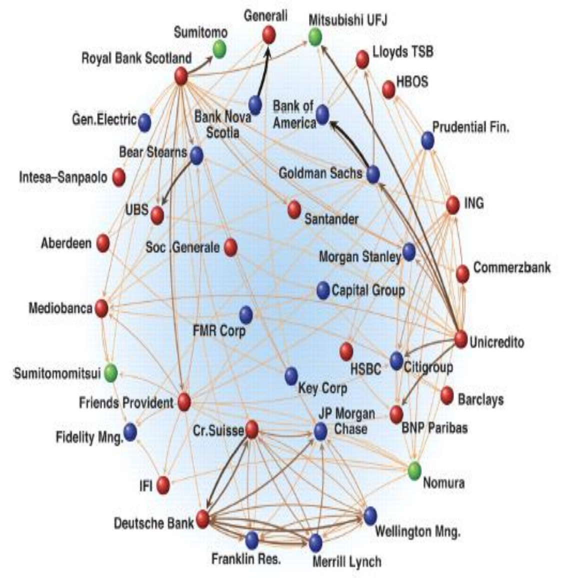


Image credit: Science

Economic Networks

Image credit: Lumen Learning

Communication Networks

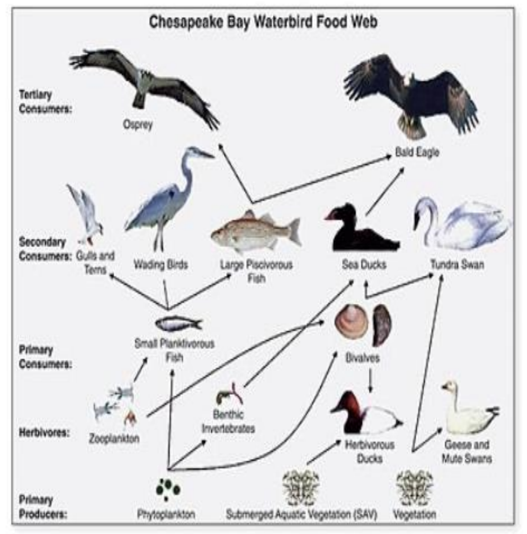


Image credit: Wikipedia

Food Webs

Image credit: Pinterest

Particle Networks

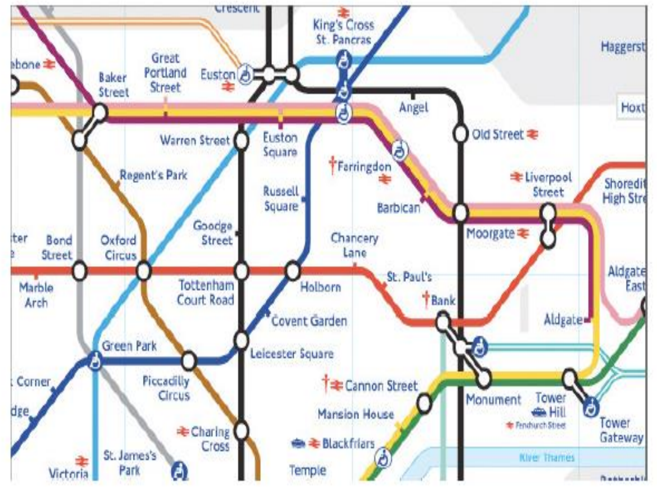
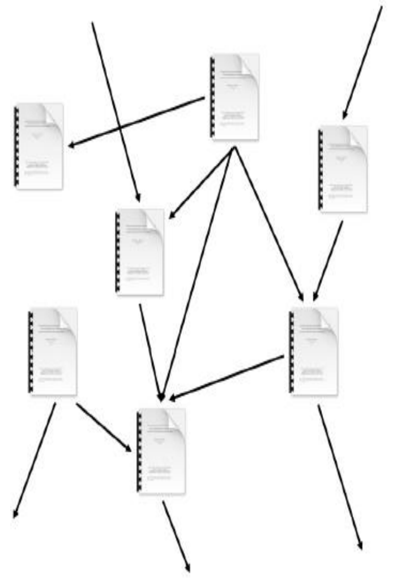


Image credit: visitlondon.com

Underground Networks



Citation Networks

Image credit: Missoula Current News

Internet

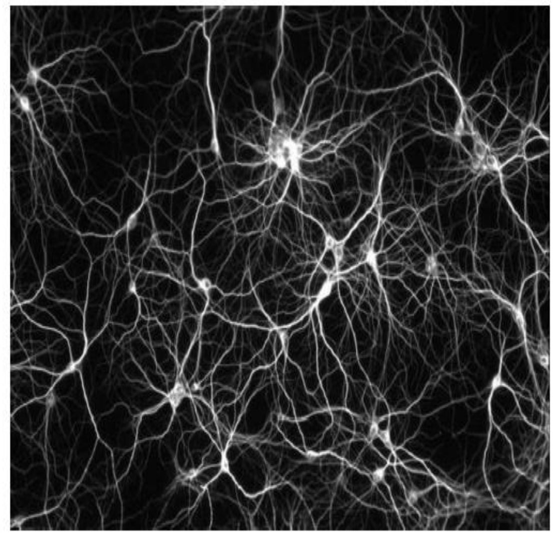


Image credit: The Conversation

Networks of Neurons

StanfordCS224w

Graph-Structured Data is Everywhere

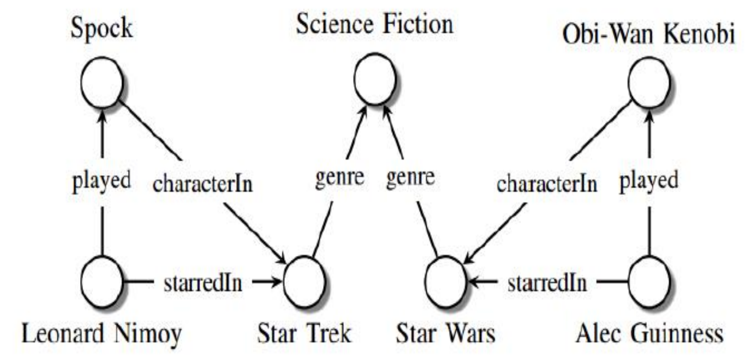


Image credit: Maximilian Nickel et al

Knowledge Graphs

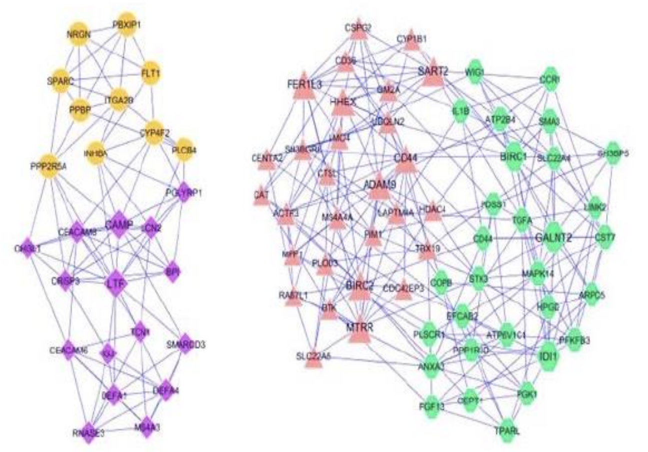


Image credit: ese.wustl.edu

Regulatory Networks

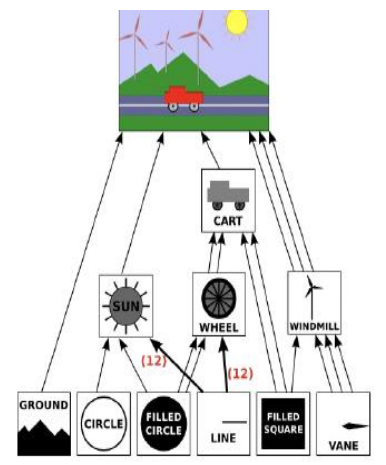


Image credit: math.hws.edu

Scene Graphs

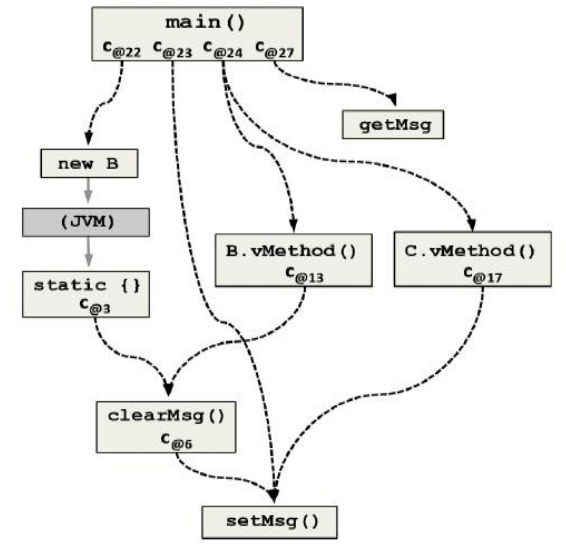
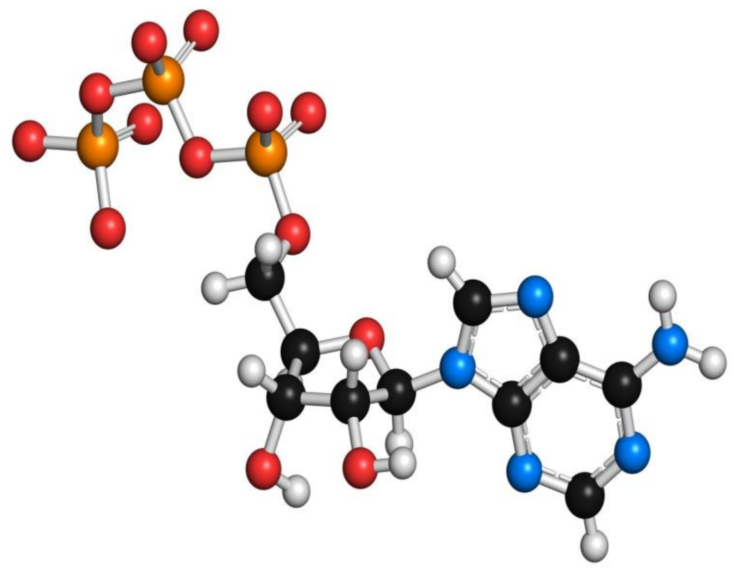


Image credit: ResearchGate

Code Graphs



Molecules

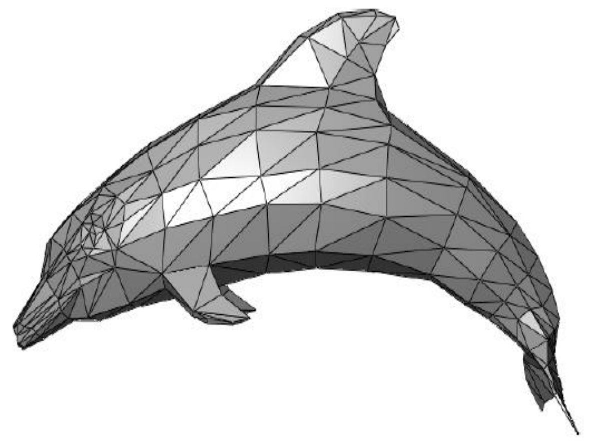
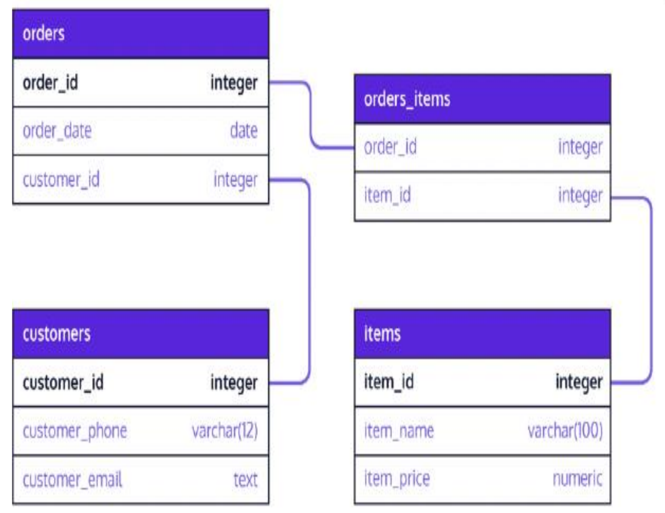
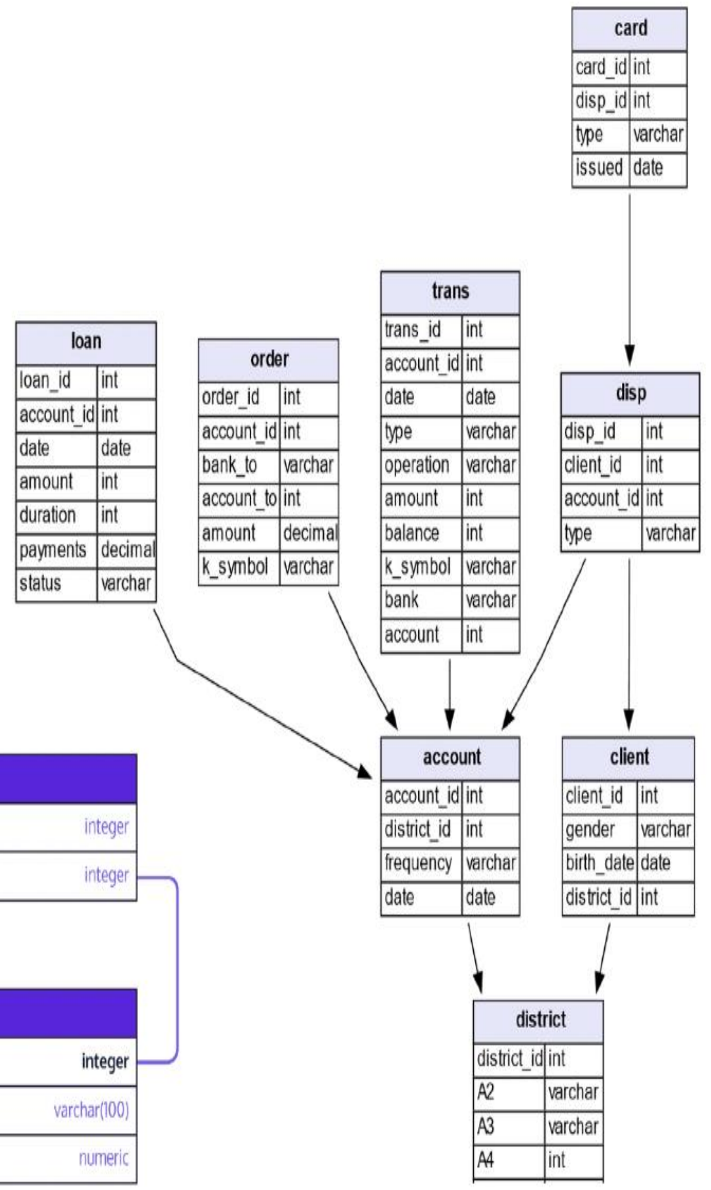


Image credit: Wikipedia

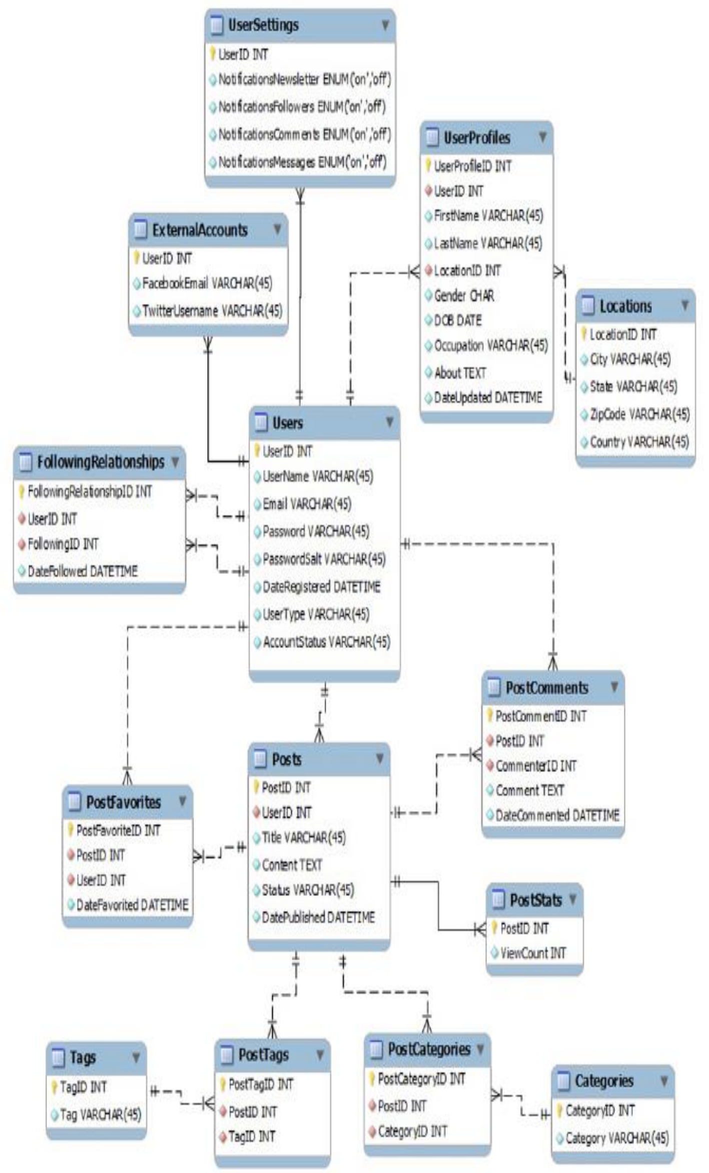
3D Shapes



Commerce



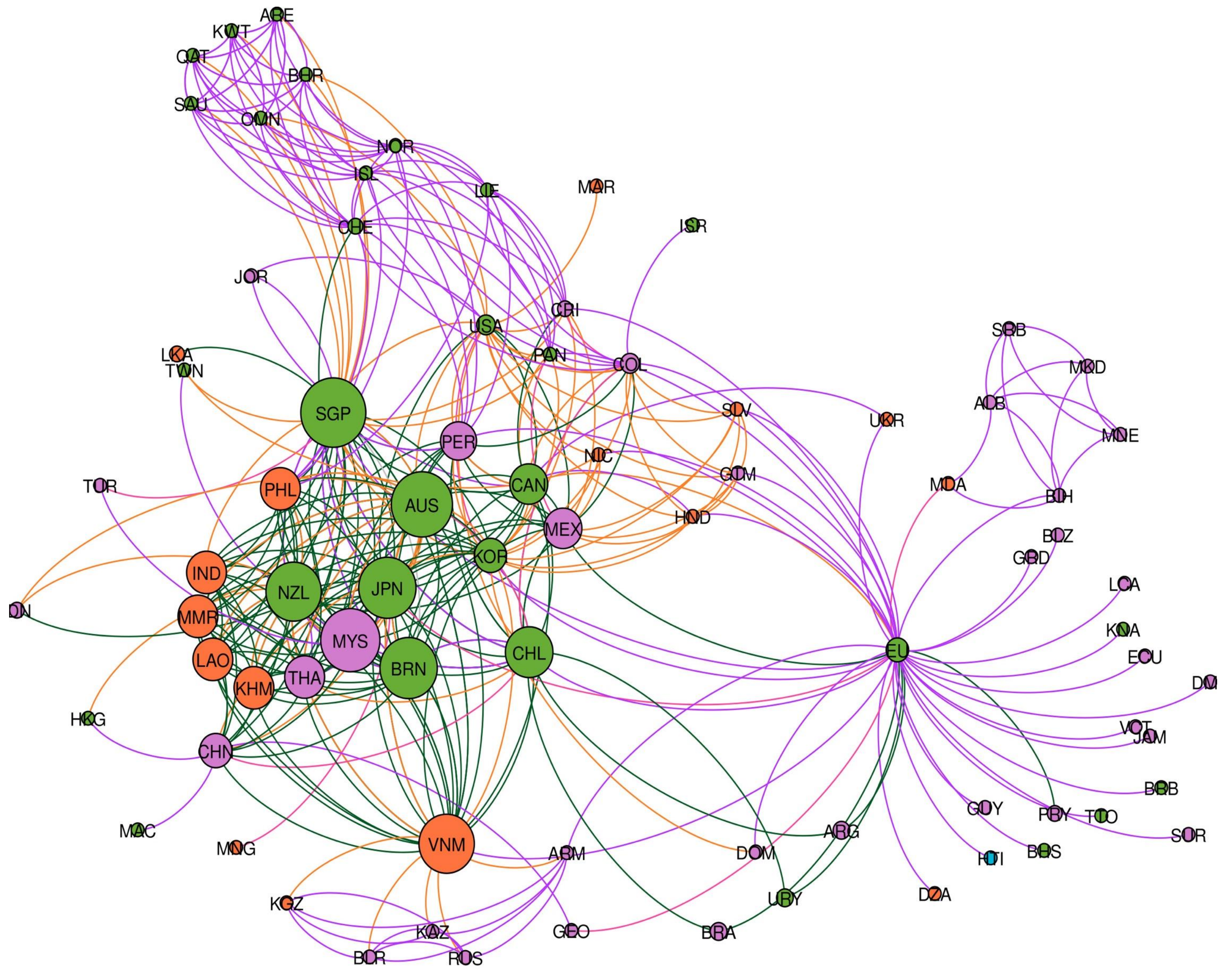
Finance

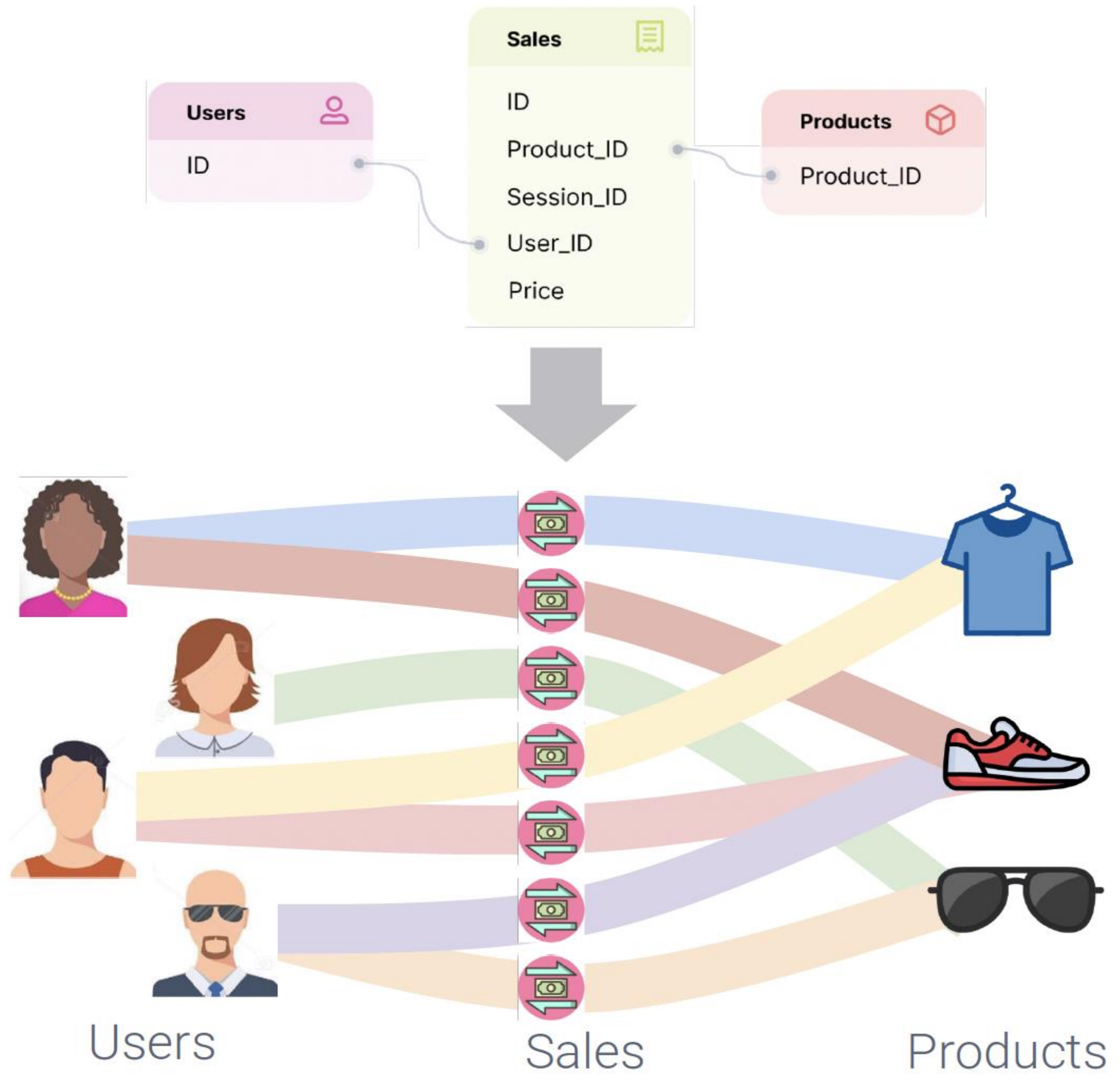


Social Media

StanfordCS224w

Graph-Structured Data is Everywhere





StanfordCS224w

https://iit.adelaide.edu.au/news/list/2021/09/16/the-topology-of-e-commerce-governance

Challenges

Graph-structured data pose significant challenges due to their irregularity, high dimensionality, and computational complexity. The major challenges include:

- ➢ Scalability and computational inefficiency
- ➢ Irregular and dynamic nature
- ➢ Data sparsity and missing values
- ➢ Complex relationships and non-Euclidean space
- ➢ Challenges in learning meaningful representations
- ➢ Privacy, security, and adversarial attacks

Scalability and Computational Inefficiency

- ❖ Large graphs (e.g., social networks, citation networks) contain billions of nodes and edges.
- ❖ Many traditional graph algorithms (e.g., PageRank, shortest path) have $O(n^2)$ or worse complexity.
- ❖ Memory and computational demands increase exponentially as graphs grow.

Example:

- ❑ Google's PageRank algorithm operates on a massive web graph (~60 billion pages).

Solution with GNNs:

- ➢ GraphSAGE: Uses node sampling and aggregation to reduce computation.
- ➢ FastGCN: Uses importance-based sampling for efficient learning.
- ➢ Distributed frameworks: Utilize tools like DGL and PyTorch Geometric for large-scale graphs.

Irregular and Dynamic Graph Structures

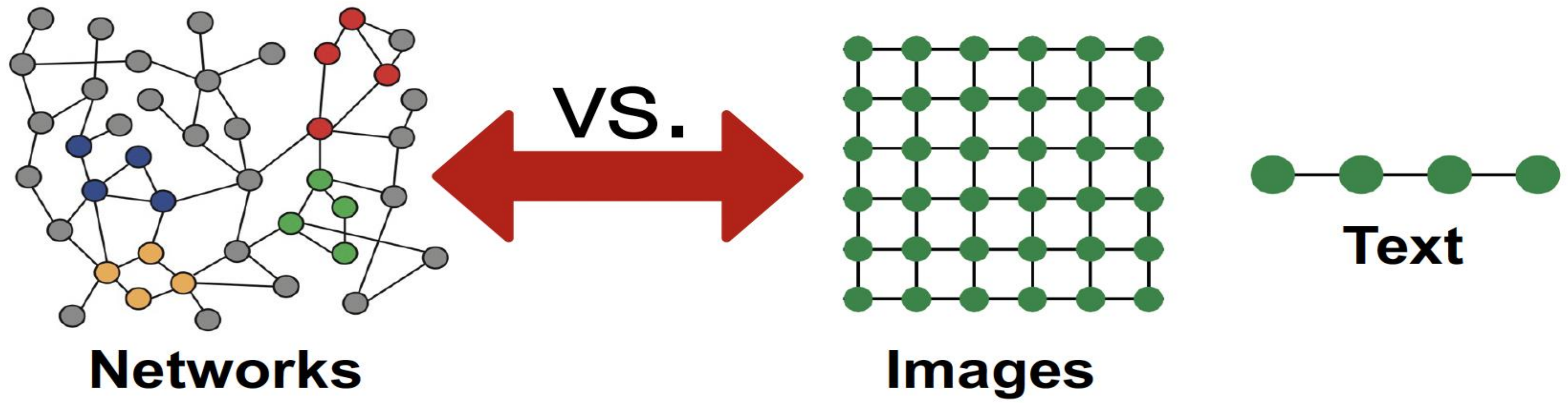
- ❖ Unlike images and sequences, graphs lack a fixed structure, making batch processing difficult.
- ❖ Nodes have varying numbers of neighbors, leading to inefficiencies in training.
- ❖ Many graphs evolve over time (e.g., Twitter networks, citation graphs).

Example:

- ❑ Social networks such as Facebook continuously update with new friendships and interactions.

Solution with GNNs:

- ➢ Temporal Graph Networks (TGNs): Adapt GNNs to evolving graph structures.
- ➢ Graph Attention Networks (GATs): Assign different importance to neighbors for better learning.
- ➢ Dynamic GNNs: Handle time-dependent graphs (e.g., financial fraud detection).



https://scitechdaily.com/scientists-discover-common-brain-network-for-psychiatric-illnesses/

Data Sparsity and Missing Values

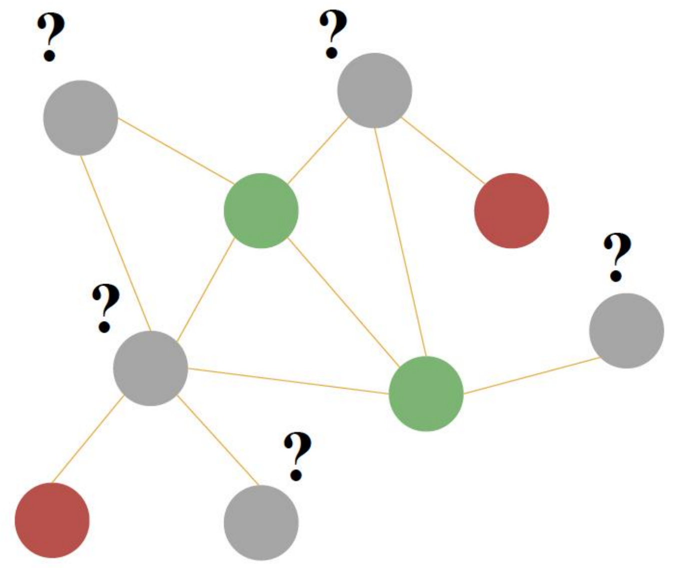
- ❖ Many real-world graphs have missing nodes, edges, or attributes.
- ❖ Data imbalance can lead to poor generalization in machine learning models.

Example:

- ❑ Recommender systems (e.g., Netflix, Amazon) have incomplete user-item interactions.
- ❑ Knowledge graphs have incomplete links and entities.

Solution with GNNs:

- ➢ Graph Autoencoders (GAEs): Predict missing edges and restore incomplete graphs.
- ➢ Self-supervised Learning: Leverages unlabeled data to enhance representations.
- ➢ Data Augmentation for Graphs: Synthesizes new nodes and edges to improve learning.



Non-Euclidean Nature of Graph Data

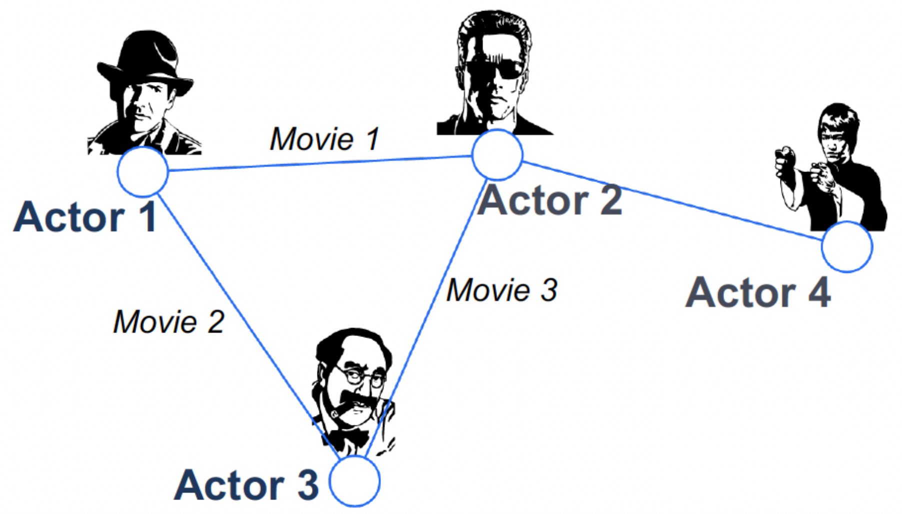
- ❖ Traditional ML assumes Euclidean space (e.g., CNNs for images, RNNs for text).
- ❖ Graphs exist in non-Euclidean space, making feature extraction difficult.

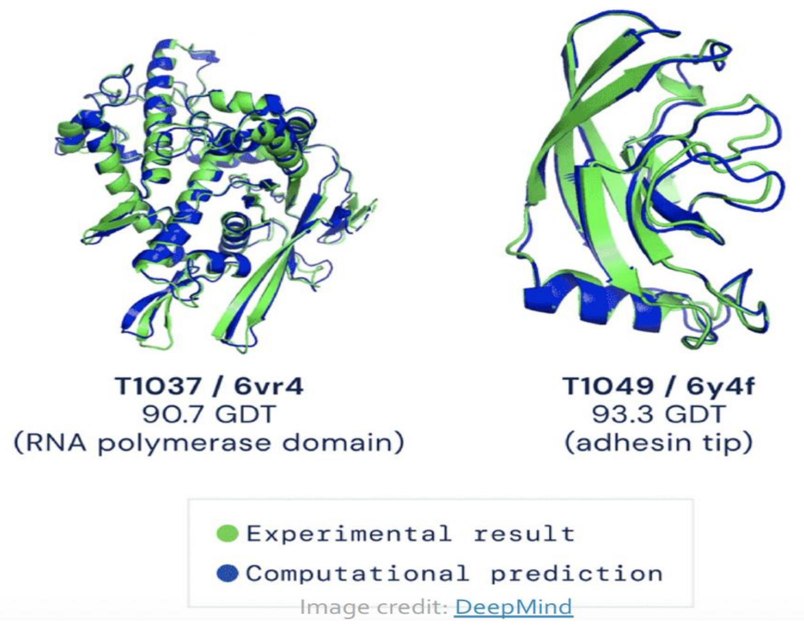
Example:

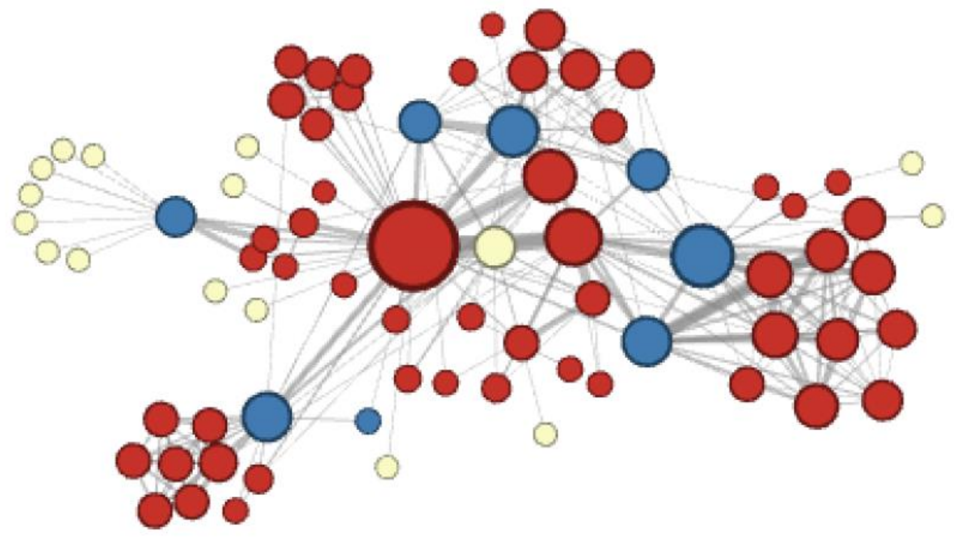
- ❑ Protein-protein interaction networks require specialized models beyond standard deep learning.

Solution with GNNs:

- ➢ Spectral GNNs: Use graph Fourier transforms to process signals on graphs.
- ➢ Spatial GNNs: Learn directly from local neighborhood structures.
- ➢ Hyperbolic GNNs: Embed graphs in hyperbolic space to improve distance preservation.







Challenges in learning meaningful representations

- ❖ Irregular and Non-Euclidean Structure.
- ❖ Capturing Long-range Dependencies
- ❖ Handling Heterogeneous Graphs (Multi-type Nodes and Edges)
- ❖ Lack of Labeled Data for Training

Examples:

- ❑ A **social network** where users have different numbers of connections.
- ❑ **Academic networks** contain researchers, papers, and institutions, each connected by different relationships.
- ❑ **Biomedical graphs** contain vast amounts of unlabeled molecular structures.

Solution with GNNs:

- ➢ **Graph Convolutional Networks (GCNs)** generalize convolution operations to graphs.
- ➢ **Graph Attention Networks (GATs)** assign different importance to different neighbors.
- ➢ **Residual connections** (e.g., Graph Residual Networks) help preserve information from earlier layers.
- ➢ **Jumping Knowledge Networks (JK-Nets)** allow the model to learn adaptive neighborhood aggregation.
- ➢ **Higher-order GNNs** consider information from multi-hop neighbors.
- ➢ **Heterogeneous GNNs (e.g., HAN, R-GCN)** learn different embeddings for different node and edge types.
- ➢ **Meta-path-based methods** extract structural patterns in heterogeneous graphs.
- ➢ **Self-supervised learning (SSL)** generates pseudo-labels from the graph itself (e.g., contrastive learning).
- ➢ **Semi-supervised GNNs (e.g., GCNs)** propagate labels from a few labeled nodes.

Security and Adversarial Attacks

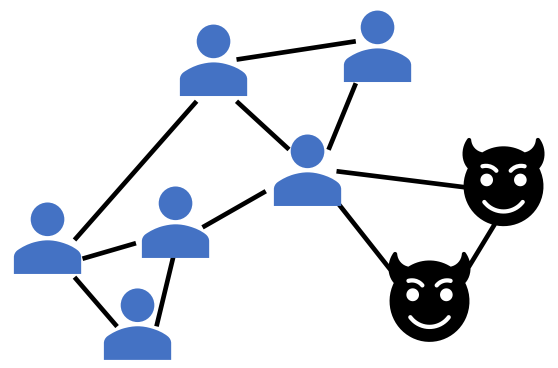
❖ Graphs are vulnerable to adversarial attacks where malicious modifications disrupt model predictions.
❖ Privacy concerns arise when handling sensitive data (e.g., financial transactions, medical records).

Example:

❑ - Fake social media accounts manipulate recommendation systems and misinformation spread.

Solution with GNNs:

➢ **Adversarially Robust GNNs:** Detect and mitigate fake node additions.
➢ **Differential Privacy for Graphs:** Ensures sensitive data is protected during training.
➢ **Graph Sanitization:** Removes malicious edges and nodes before training.



Financial Systems
- **Credit Card Fraud Detection**

Recommender Systems
- **Social Recommendation**
- **Product Recommendation**

....

Content

Homogeneous Graph

Key Characteristics of Homogeneous Graphs

- ❖ **Single Node Type:** All nodes in the graph belong to the same category.
- ❖ **Single Edge Type:** All edges represent the same kind of relationship between nodes.
- ❖ **Uniform Structure:** The graph follows a consistent connectivity pattern, making it easier to apply traditional graph-based algorithms.

Examples of Homogeneous Graphs

- ➢ **Social Networks (e.g., Facebook, Twitter, LinkedIn)**

Nodes: Users. **Edges**: "Friends" or "Follows" relationships between users.

- ➢ **Citation Networks (e.g., Google Scholar, ArXiv, PubMed)**

Nodes: Research papers. **Edges**: "Cites" relationships, where one paper references another.

- ➢ **Protein Interaction Networks (e.g., Biological Networks)**

Nodes: Proteins. **Edges**: "Interacts with" relationships, representing biological interactions between proteins.

How to build an effective graph?

- ❖ **Nodes (or vertices)** represent the fundamental entities in a graph. They can correspond to different objects depending on the problem domain.

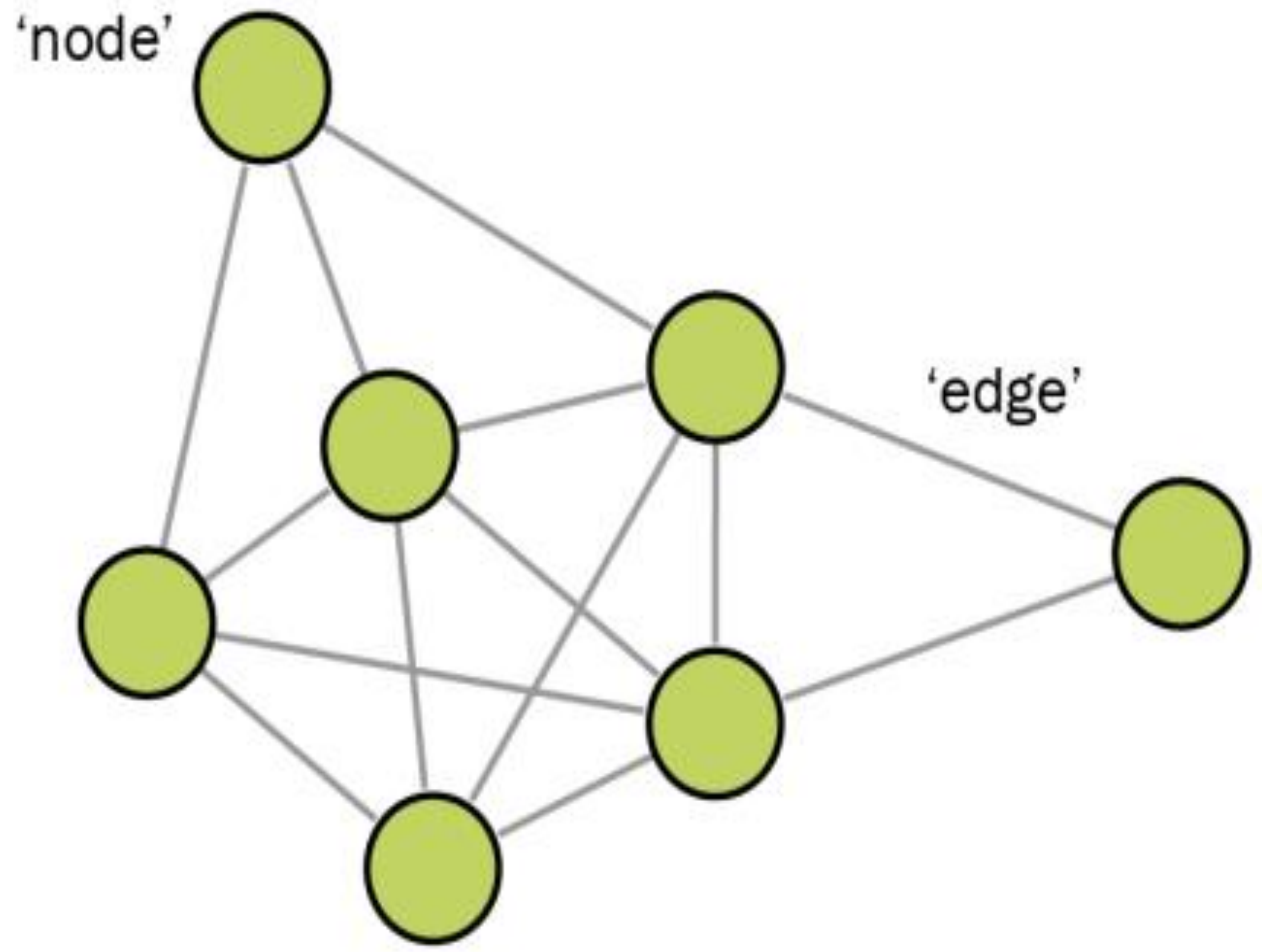
- ❖ **Edges (or links)** define relationships or interactions between nodes. Edges can be:
  - ➢ **Directed or undirected** (e.g., one-way vs. mutual friendships).
  - ➢ **Weighted or unweighted** (e.g., flight routes with different distances).
  - ➢ **Static or dynamic** (e.g., evolving relationships over time).

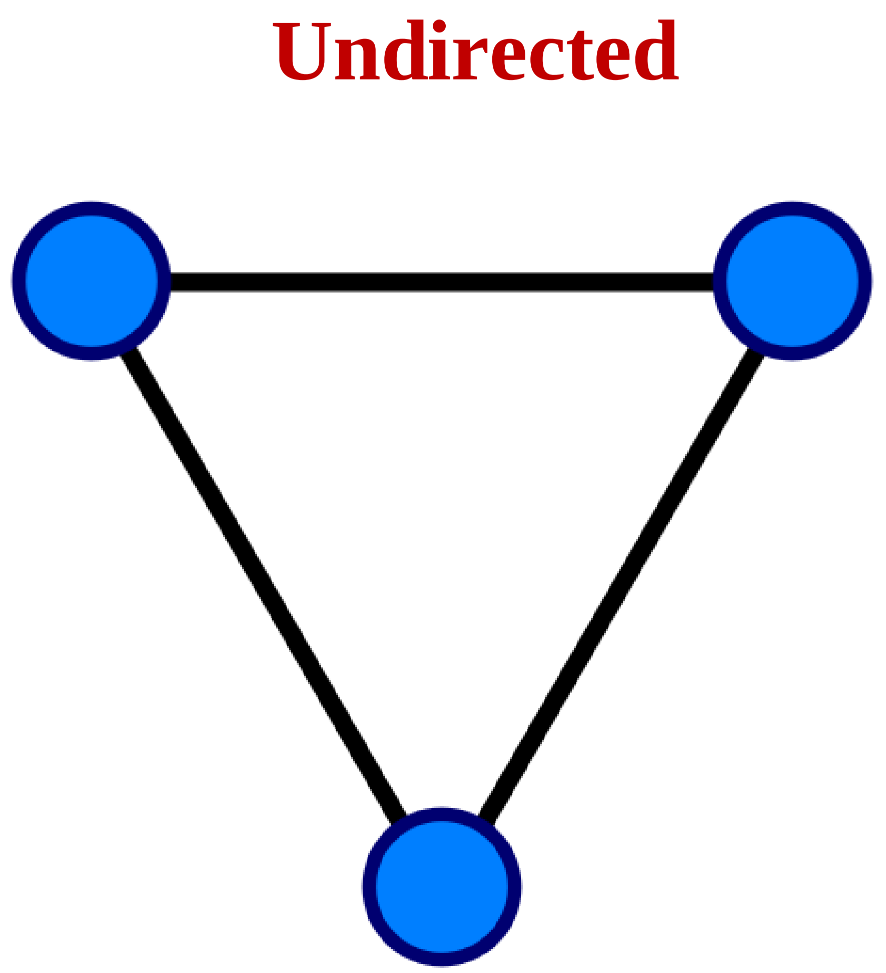
- ❖ **Choosing the Proper Network Representation.** The way we construct a graph determines our ability to extract meaningful insights. Different representations can lead to different outcomes.
- ➢ **Cases Where Representation is Unique and Unambiguous**
- ➢ **Cases Where Representation is Not Unique**

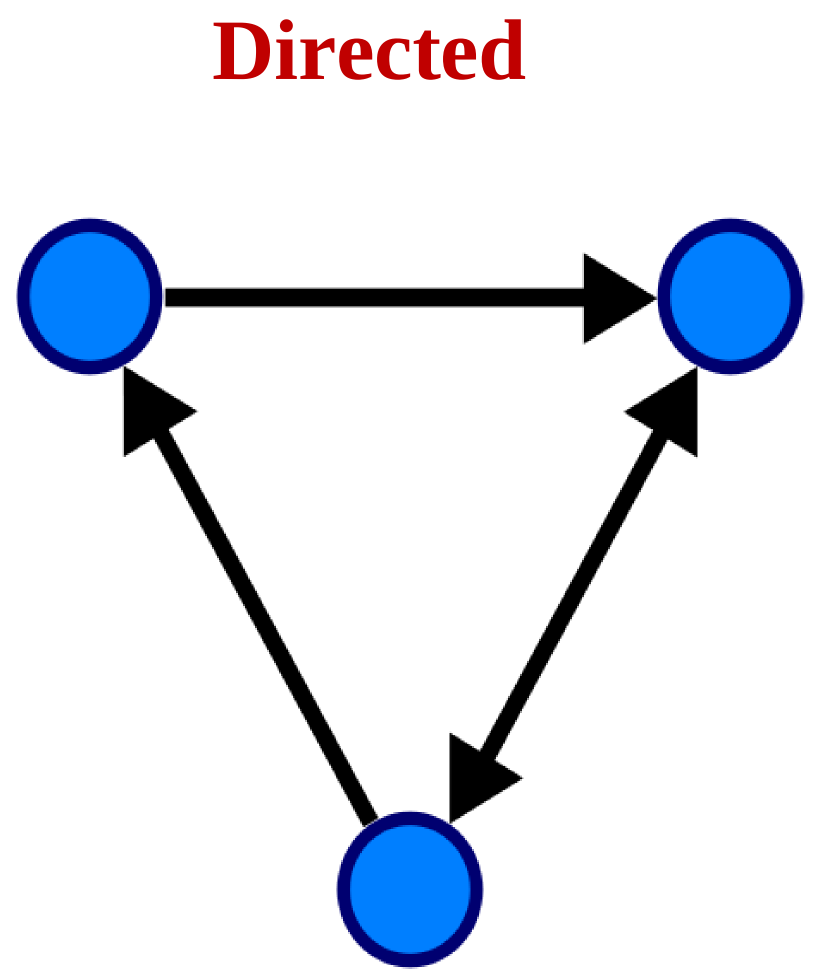
- ❖ **How the Choice of Links Affects the Questions You Can Study**
- ➢ The **way you define connections** (edges) influences the type of insights you can extract.
- ➢ If you **ignore certain relationships**, you may miss critical aspects of the data.
- ➢ If you **add unnecessary edges**, you might introduce noise and bias in analysis.

Graph Set-up

Graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E between these nodes. An edge going from node $u \in V$ to node $v \in V$ as $(u, v) \in E$.

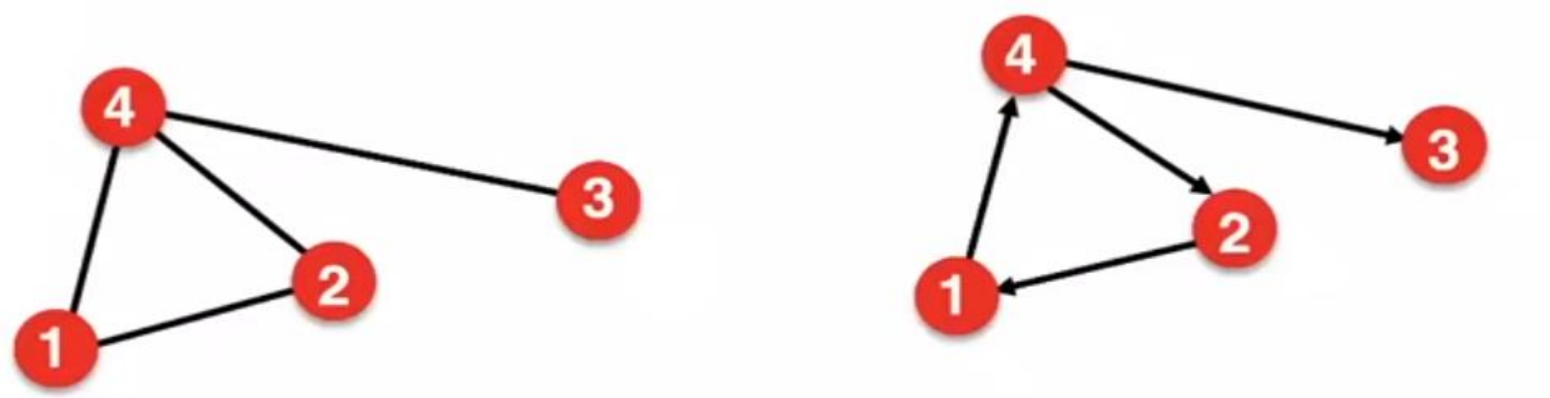






Adjacency Matrix

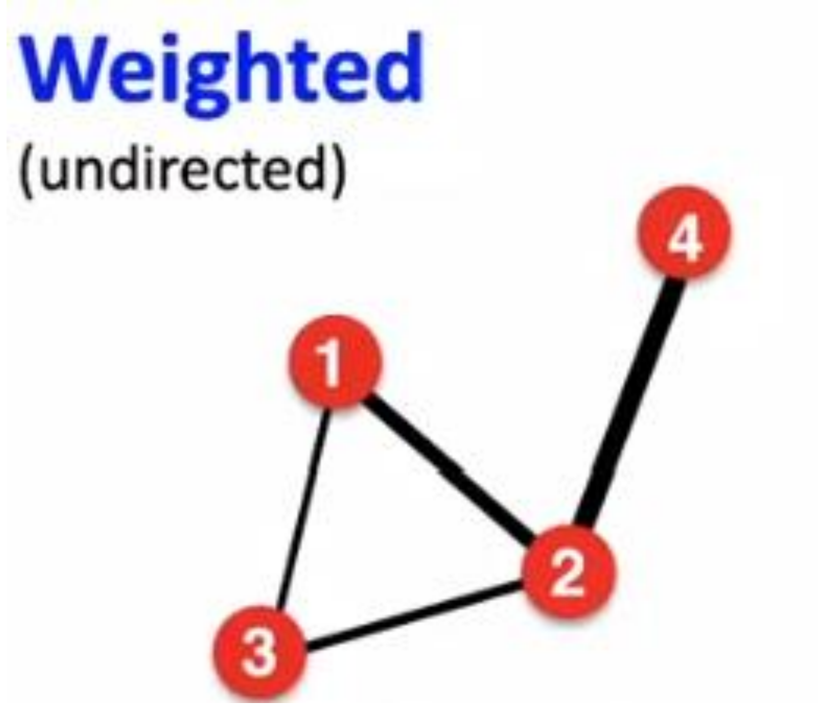
A convenient way to represent graphs is through an adjacency matrix $A \in \mathbb{R}^{|V|\times|V|}$. We order the nodes in the graph so that every node indexes a particular row and column in the adjacency matrix.

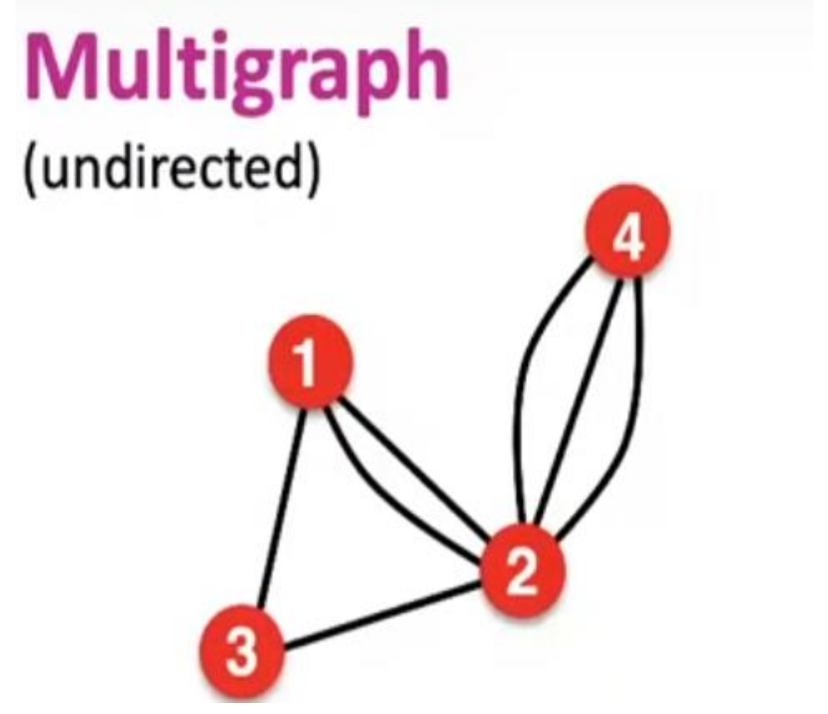


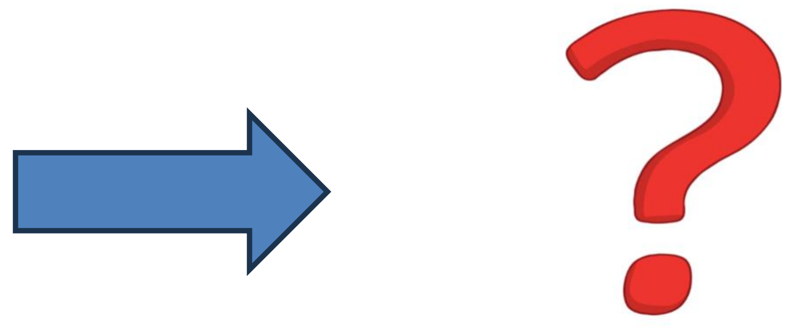
$A_{ij} = 1$ if there is a link from node i to node j

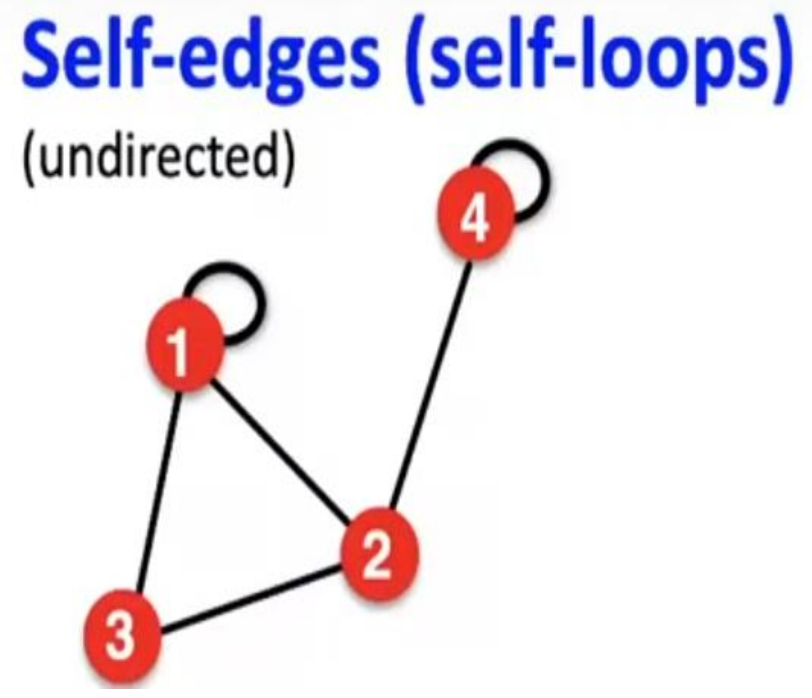
$A_{ij} = 0$ otherwise

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \qquad A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

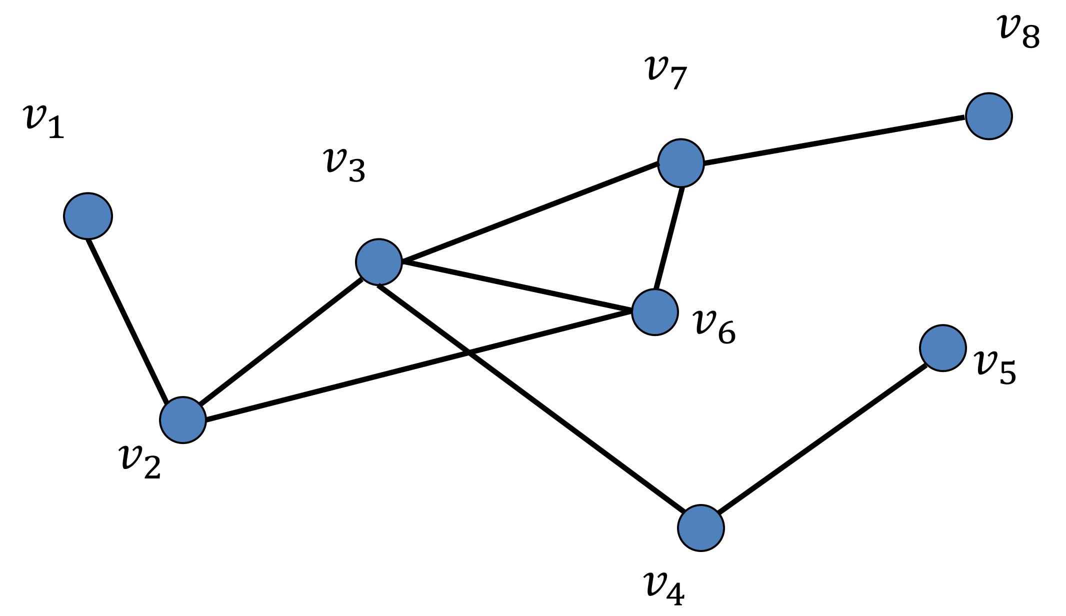








Graphs and Graph Signals

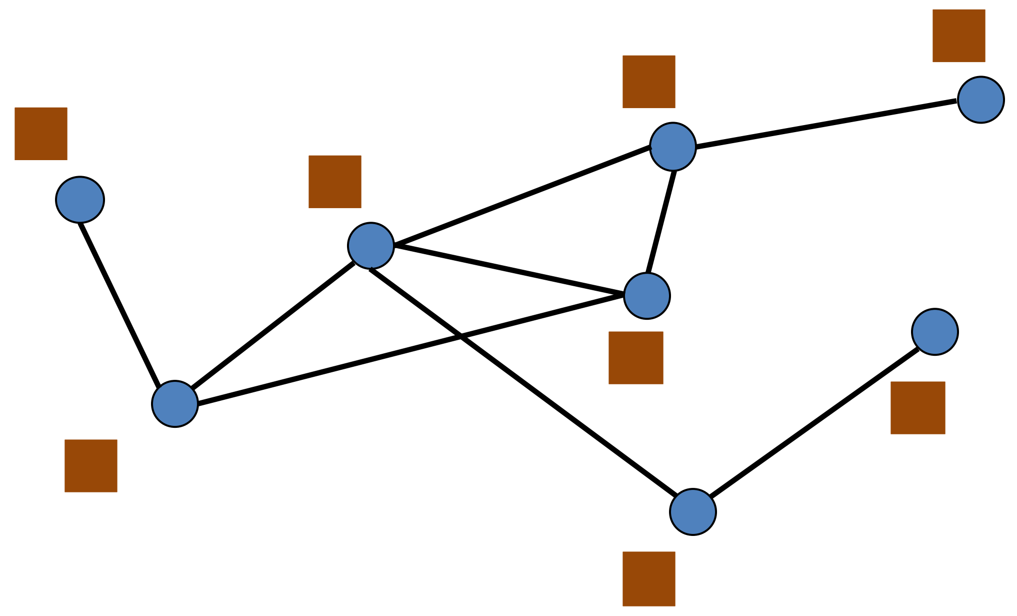


$$\mathcal{V} = \{v_1, \ldots, v_N\}$$

$$\mathcal{E} = \{e_1, \ldots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

Graphs and Graph Signals



$$\mathcal{V} = \{v_1, \ldots, v_N\}$$

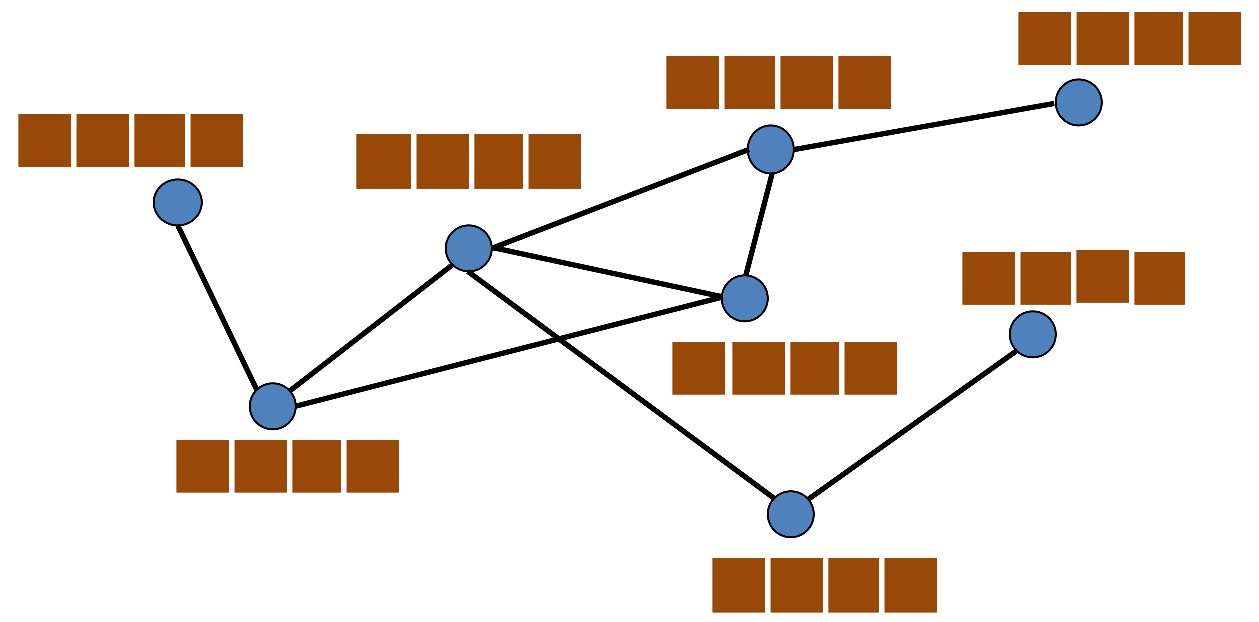
$$\mathcal{E} = \{e_1, \ldots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

Graph Signal: $f : \mathcal{V} \to \mathbb{R}^N$

$$\mathcal{V} \longrightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

Graphs and Graph Signals



Graph Signal:$f : \mathcal{V} \rightarrow \mathbb{R}^{N \times d}$

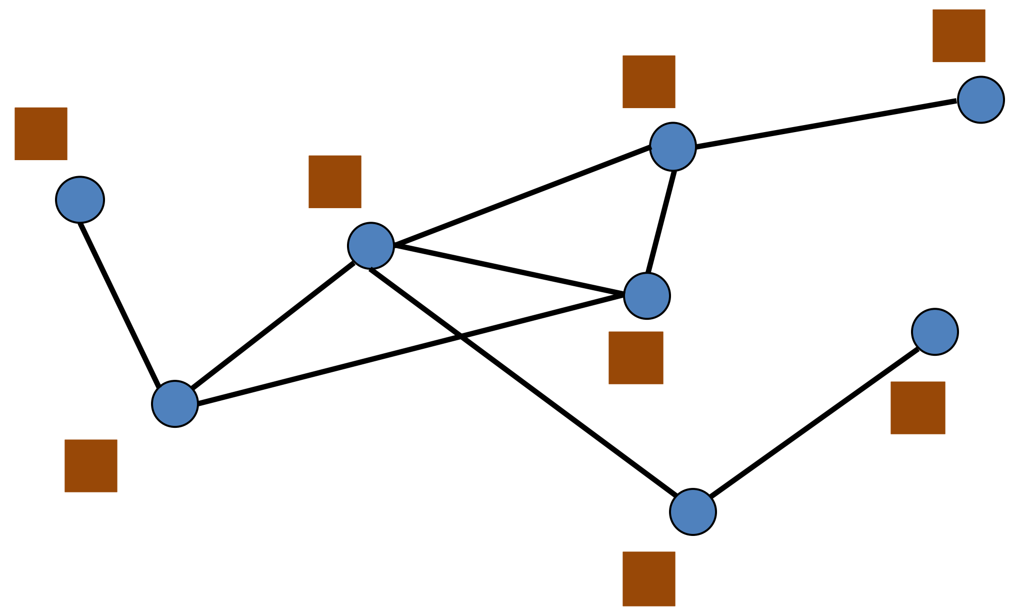
$$\mathcal{V} = \{v_1, \ldots, v_N\}$$

$$\mathcal{E} = \{e_1, \ldots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

$$\mathcal{V} \longrightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

Graphs and Graph Signals



Graph Signal: $f : \mathcal{V} \to \mathbb{R}^N$

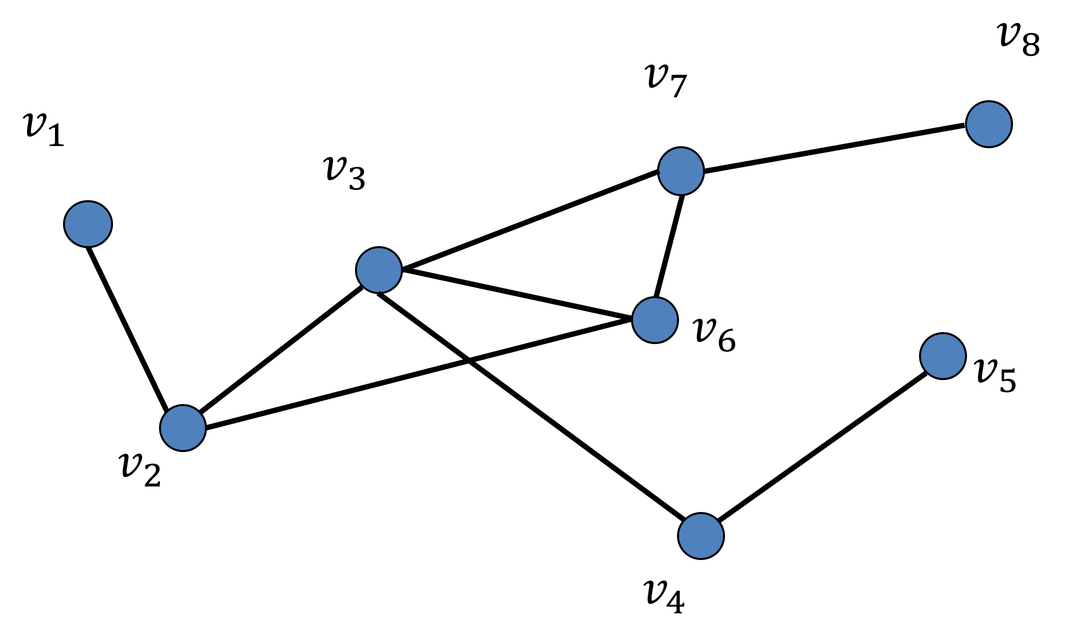
$$\mathcal{V} = \{v_1, \ldots, v_N\}$$

$$\mathcal{E} = \{e_1, \ldots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

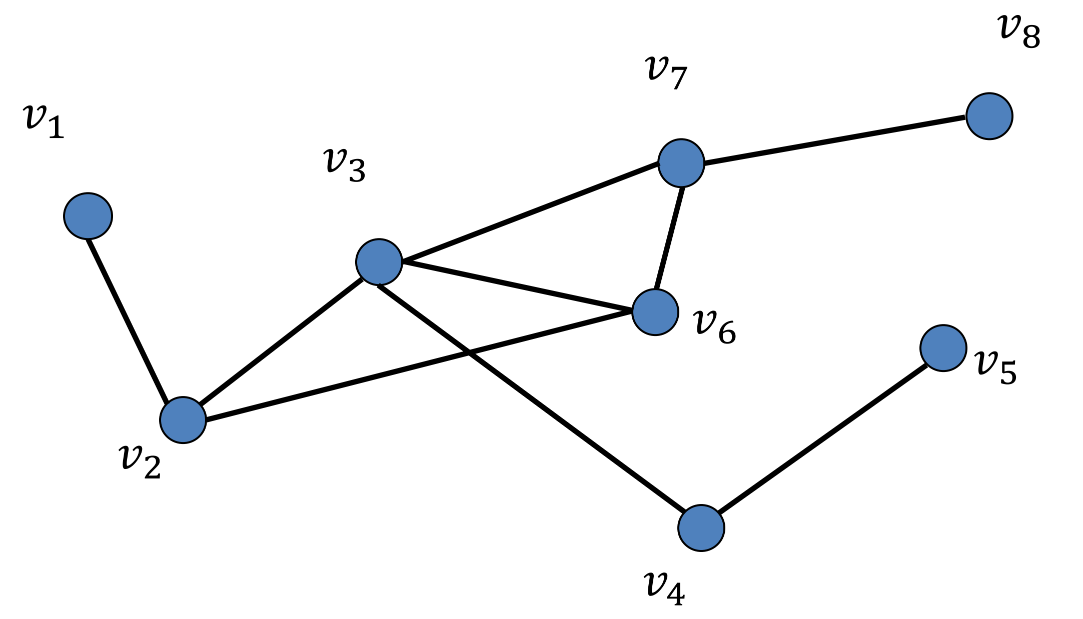
$$\mathcal{V} \longrightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

Matrix Representations of Graphs



Spectral graph theory. American Mathematical Soc.; 1997.

Matrix Representations of Graphs



Adjacency Matrix: $A[i,j] = 1$ if v_i is adjacent to v_j

$A[i,j] = 0$, otherwise

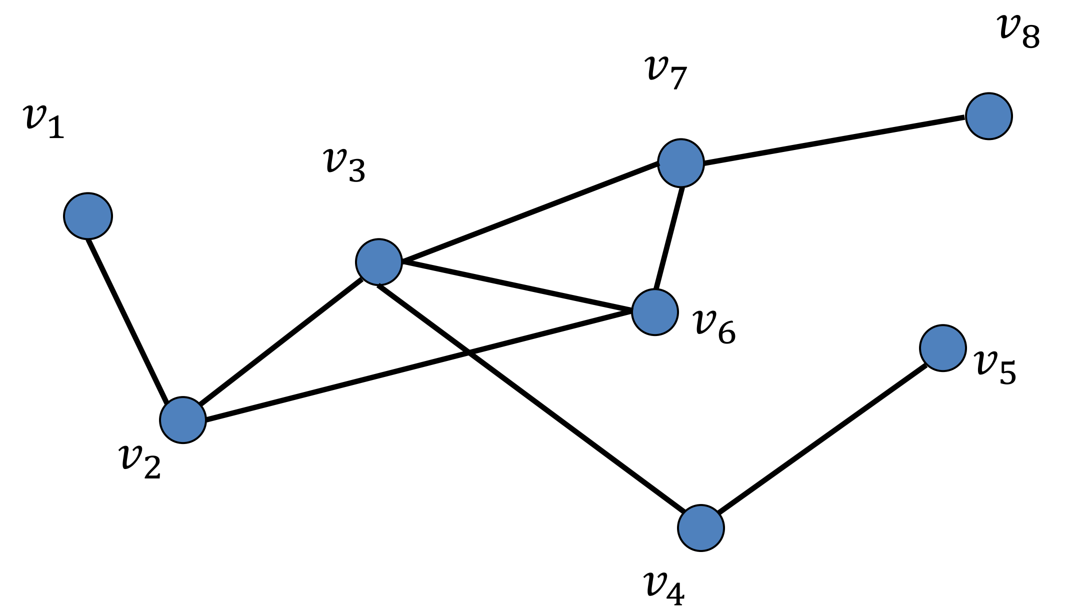
Adjacency Matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$\boldsymbol{A}$

Spectral graph theory. American Mathematical Soc.; 1997.

Matrix Representations of Graphs



Adjacency Matrix: $A[i,j] = 1$ if v_i is adjacent to v_j
$A[i,j] = 0$, otherwise

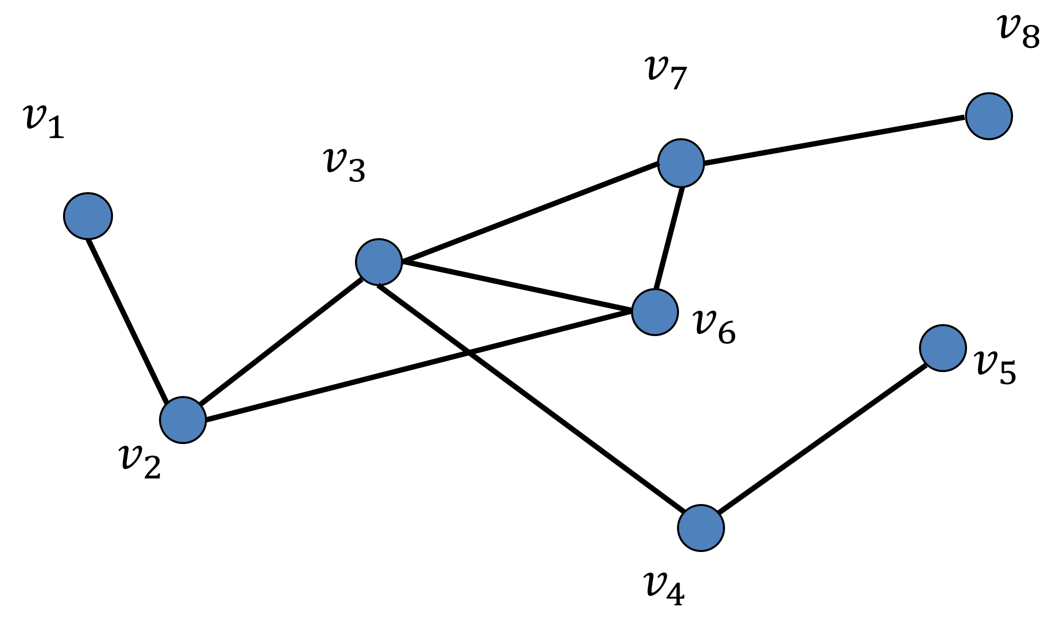
Degree Matrix: $\mathbf{D} = \mathrm{diag}(degree(v_1), \ldots, degree(v_N))$

Degree Matrix   Adjacency Matrix

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{\boldsymbol{D}} - \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}}_{\boldsymbol{A}}$$

Spectral graph theory. American Mathematical Soc.; 1997.

Matrix Representations of Graphs



Adjacency Matrix: $A[i,j] = 1$ if v_i is adjacent to v_j

$A[i,j] = 0$, otherwise

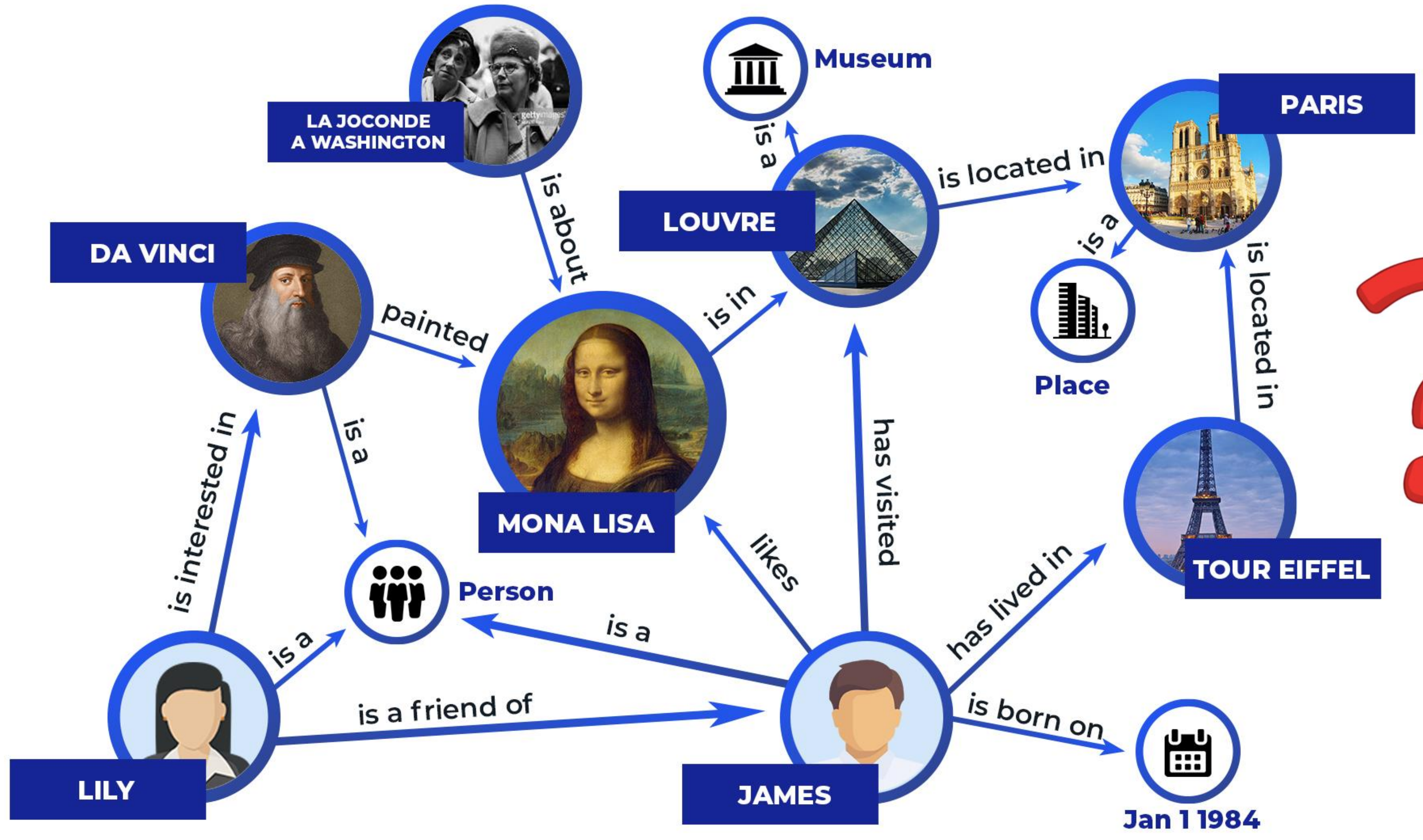
Degree Matrix: $\mathbf{D} = \mathrm{diag}(degree(v_1), \ldots, degree(v_N))$

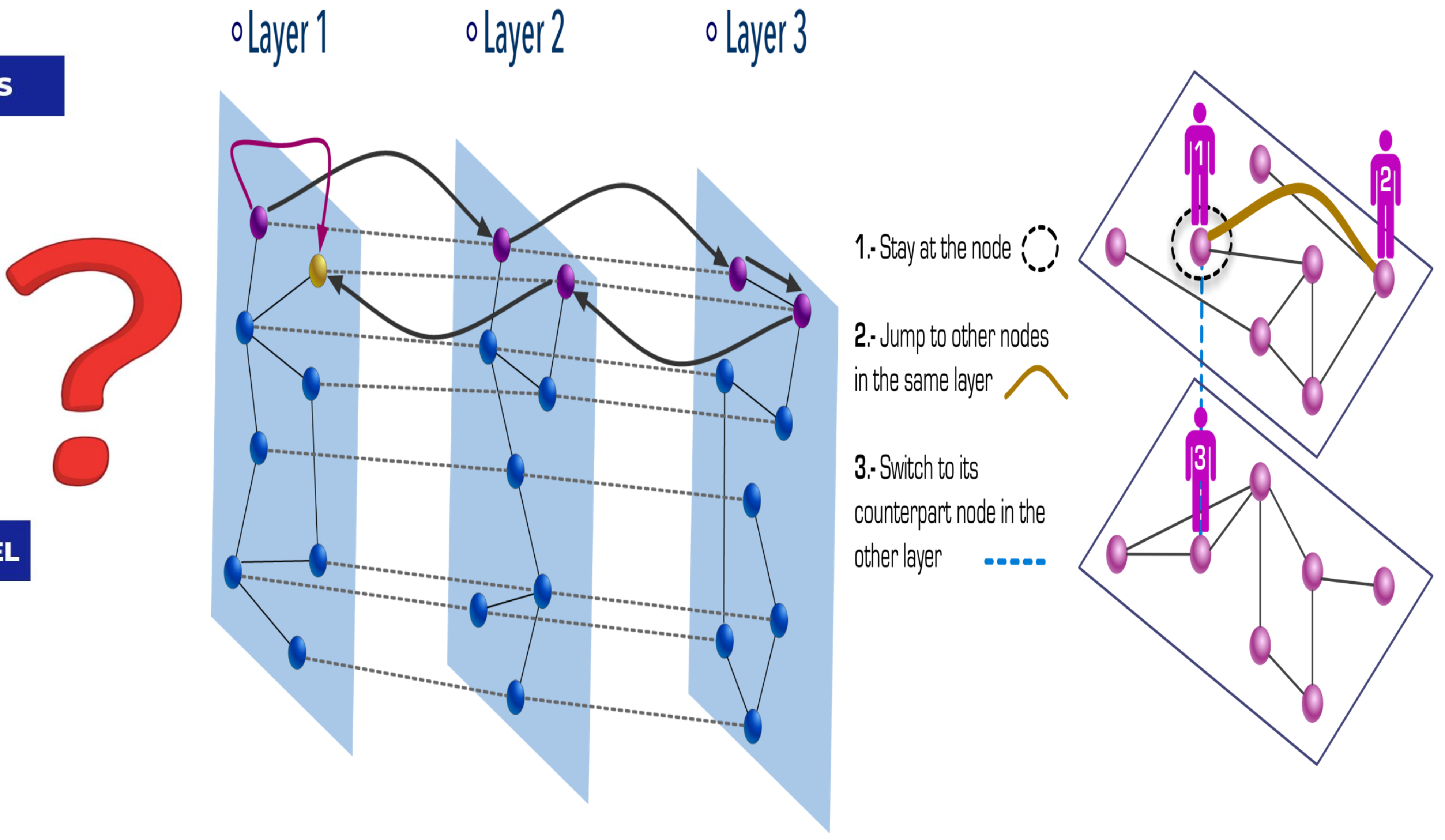
Degree Matrix | Adjacency Matrix | Laplacian Matrix

$$\underbrace{\begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&3&0&0&0&0&0&0\\ 0&0&4&0&0&0&0&0\\ 0&0&0&2&0&0&0&0\\ 0&0&0&0&2&0&0&0\\ 0&0&0&0&0&4&0&0\\ 0&0&0&0&0&0&3&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix}}_{\boldsymbol{D}} - \underbrace{\begin{pmatrix} 0&1&0&0&0&0&0&0\\ 1&0&1&0&0&1&0&0\\ 0&1&0&1&0&1&1&0\\ 0&0&1&0&1&0&0&0\\ 0&0&0&1&0&1&0&0\\ 0&1&1&0&1&0&1&0\\ 0&0&1&0&0&1&0&1\\ 0&0&0&0&0&0&1&0 \end{pmatrix}}_{\boldsymbol{A}} = \underbrace{\begin{pmatrix} 1&-1&0&0&0&0&0&0\\ -1&3&-1&0&0&-1&0&0\\ 0&-1&4&-1&0&-1&-1&0\\ 0&0&-1&2&-1&0&0&0\\ 0&0&0&-1&2&-1&0&0\\ 0&-1&-1&0&-1&4&-1&0\\ 0&0&-1&0&0&-1&3&-1\\ 0&0&0&0&0&0&-1&1 \end{pmatrix}}_{\boldsymbol{L}}$$

Spectral graph theory. American Mathematical Soc.; 1997.

How to Deal with Multi-relation?




Layer 1
Layer 2
Layer 3
1.- Stay at the node
2.- Jump to other nodes in the same layer
3.- Switch to its counterpart node in the other layer
1
2
3

Heterogeneous Graph

Key Characteristics of Heterogeneous Graphs

- ❖ **Multiple Node Types:** Nodes represent different entities, such as users, items, papers, or institutions.
- ❖ **Multiple Edge Types:** Different relationships exist between nodes, such as "authored by," "cites"
- ❖ **Rich Semantic Information:** The diverse relationships provide deeper insights than homogeneous graphs.

Examples of Heterogeneous Graphs

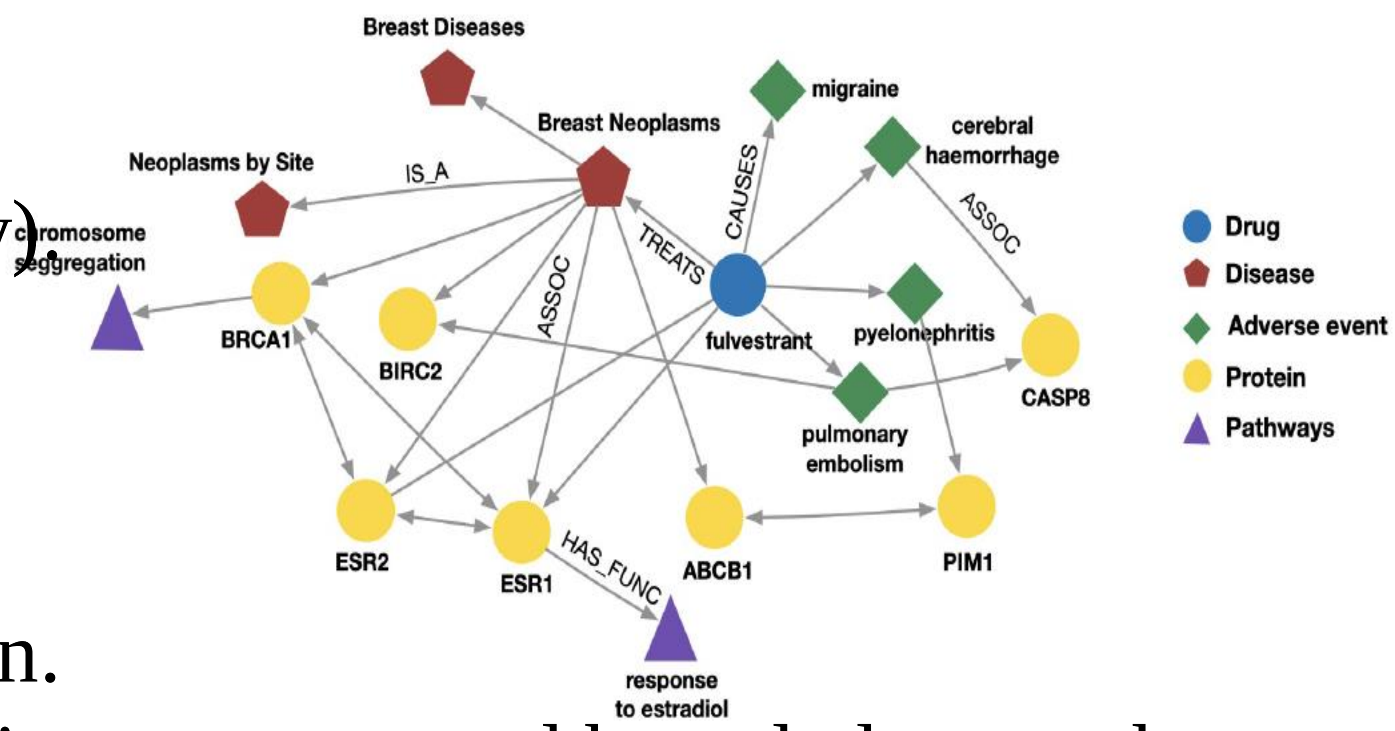
➢ **Academic Citation Network**

Nodes: Papers, authors, journals. **Edges:** "Cites" (paper-to-paper), "Authored by" (paper-to-author).

➢ **Knowledge Graphs (e.g., Google Knowledge Graph, Wikidata)**

Nodes: People, locations, organizations, events.

Edges: "Works at" (person-to-organization), "Located in" (place-to-country).



Why Are Heterogeneous Graphs Important?

- ❑ **More expressive** than homogeneous graphs, capturing richer information.
- ❑ **Essential for real-world applications** in social networks, recommendation systems, and knowledge graphs.
- ❑ **Enhance AI models** by incorporating multi-type relationships in representation learning.

Node, Edge, and Global Features

Node features represent **characteristics or attributes** of individual nodes for downstream tasks like **node classification, clustering, and link prediction**.

Common Types of Node Features

- ❖ **Categorical Features**: Node types (e.g., "user" or "product" in a recommendation system).
- ❖ **Numerical Features**: Values like age, price, or degree centrality.
- ❖ **Textual Features**: Descriptions, reviews, or labels in textual form.
- ❖ **Vectorized Embeddings**: Learned representations from NLP models or pre-trained embeddings.

Edge features define **relationships or interactions** between nodes for **link prediction and edge classification**.

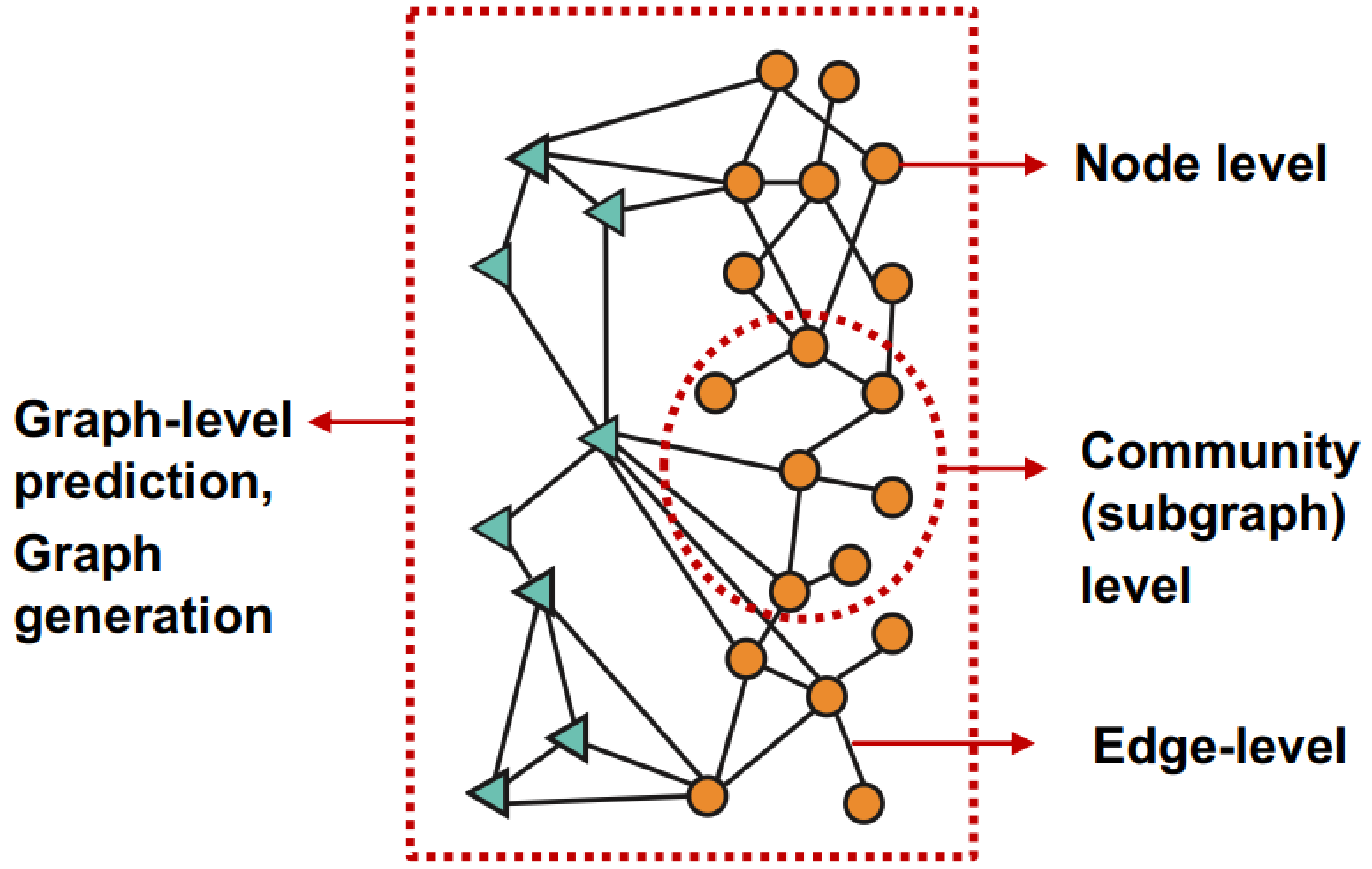
Common Types of Edge Features

- ➢ **Weight**: The strength or importance of a connection (e.g., frequency of interactions).
- ➢ **Type**: The kind of relationship (e.g., friendship, purchase, citation).
- ➢ **Timestamp**: When the connection was established (useful for dynamic graphs).
- ➢ **Directionality**: Whether the edge is directed or undirected.

Graph-Level Features: graphs have global properties or features that apply to the entire network.
Examples include:

  - ➢ **Graph Density** (How connected is the graph?).
  - ➢ **Average Clustering Coefficient** (Tendency of nodes to form clusters).
  - ➢ **Graph Size** (Number of nodes and edges).

Different Types of Task


Node level
Community (subgraph) level
Edge-level
Graph-level prediction, Graph generation

Graph-based machine learning involves multiple tasks categorized by the focus of analysis. The main categories of tasks include:

- ▶ **Node-Level Tasks:** Predicting properties of individual nodes.
- ▶ **Edge-Level Tasks:** Inferring relationships between node pairs.
- ▶ **Community-Level Tasks:** Detecting and analyzing groups of closely connected nodes.
- ▶ **Graph-Level Tasks:** Understanding global graph properties.

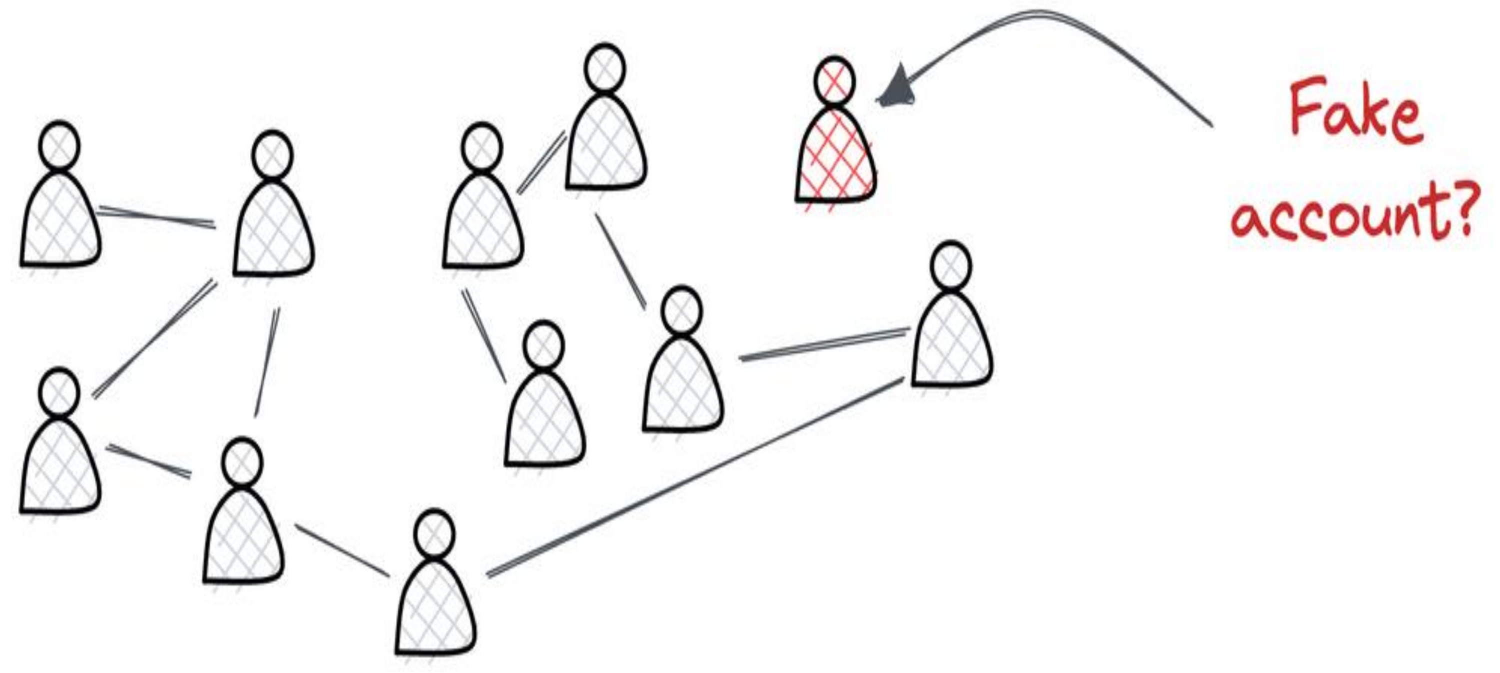
Node Level Task

Predict attributes or labels of individual nodes.

Common examples:

▶ Node Classification: Assign labels to nodes (e.g., fraud detection in financial networks).

▶ Node Regression: Predict continuous values (e.g., influence score in social networks).

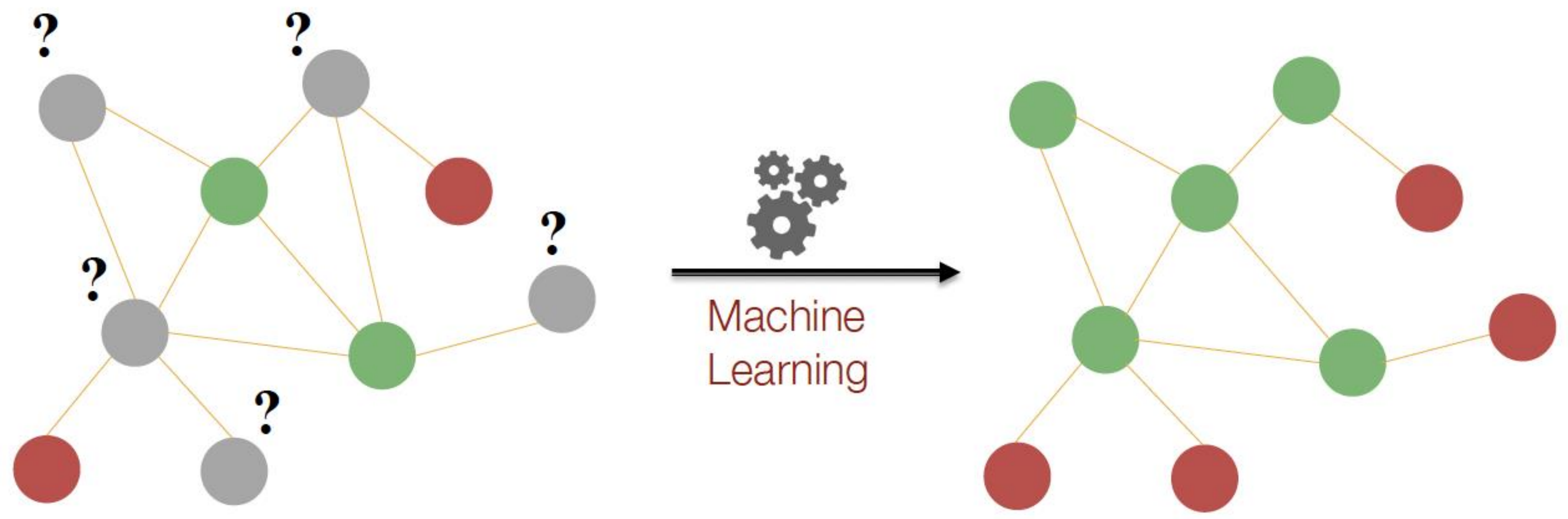


Applications:

▶ Social network analysis

▶ Protein function prediction

▶ Fraud detection



Node-level Network Structure and Graphlets

The **node-level network structure** examines the local properties of individual nodes within a graph. It helps in understanding how a particular node is positioned within the overall graph.

Key Properties:

- ❖ **Degree**: The number of direct connections a node has.
- ❖ **Clustering Coefficient**: Measures how well a node's neighbors are interconnected.
- ❖ **Centrality Measures**:
  - ➢ **Degree Centrality**: The number of direct links a node has.
  - ➢ **Betweenness Centrality**: Measures how often a node acts as a bridge.
  - ➢ **PageRank**: Determines the importance of a node based on link structures.

Applications:

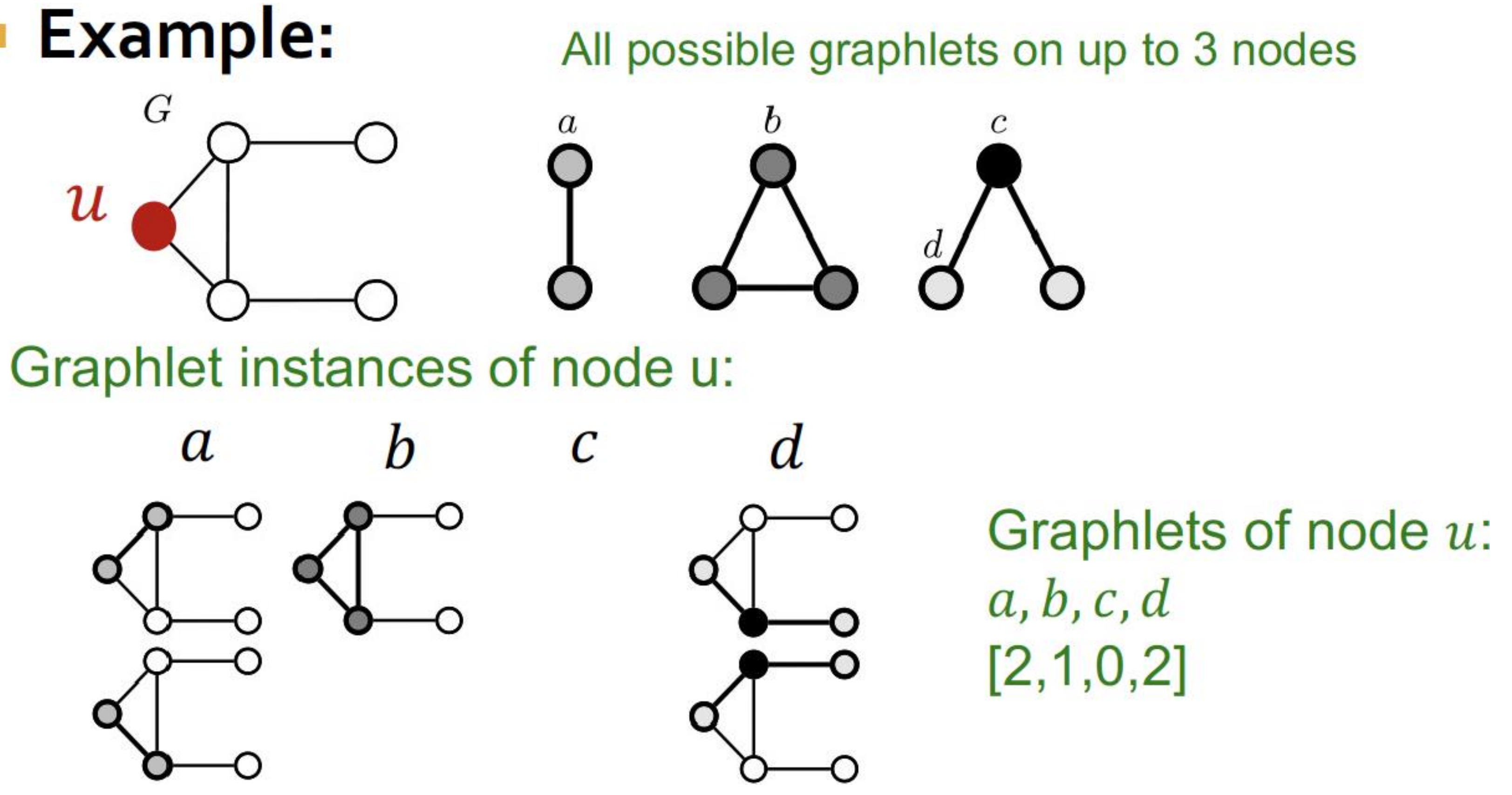
- ▶ **Social Networks:** Identifying influential users and community structures.
- ▶ **Biological Networks:** Analyzing protein interactions and genetic pathways.
- ▶ **Fraud Detection:** Finding anomalous transaction patterns.
- ▶ **Recommendation Systems:** Improving item similarity measures.

Graphlets are small, connected subgraphs used to analyze local graph structures.

Provide insights into:

- ▶ Structural patterns in networks.
- ▶ Node similarity based on shared subgraph structures.
- ▶ Network motif detection.
- ▶ Graphlets capture the building blocks of complex networks.

Example:


All possible graphlets on up to 3 nodes
G
u
a
b
c
d
Graphlet instances of node u:
a
b
c
d
Graphlets of node u:
a, b, c, d
[2,1,0,2]

AlphaFolder

AlphaFold is a deep learning-based model for protein structure prediction.

- It represents protein structures as spatial graphs, where:

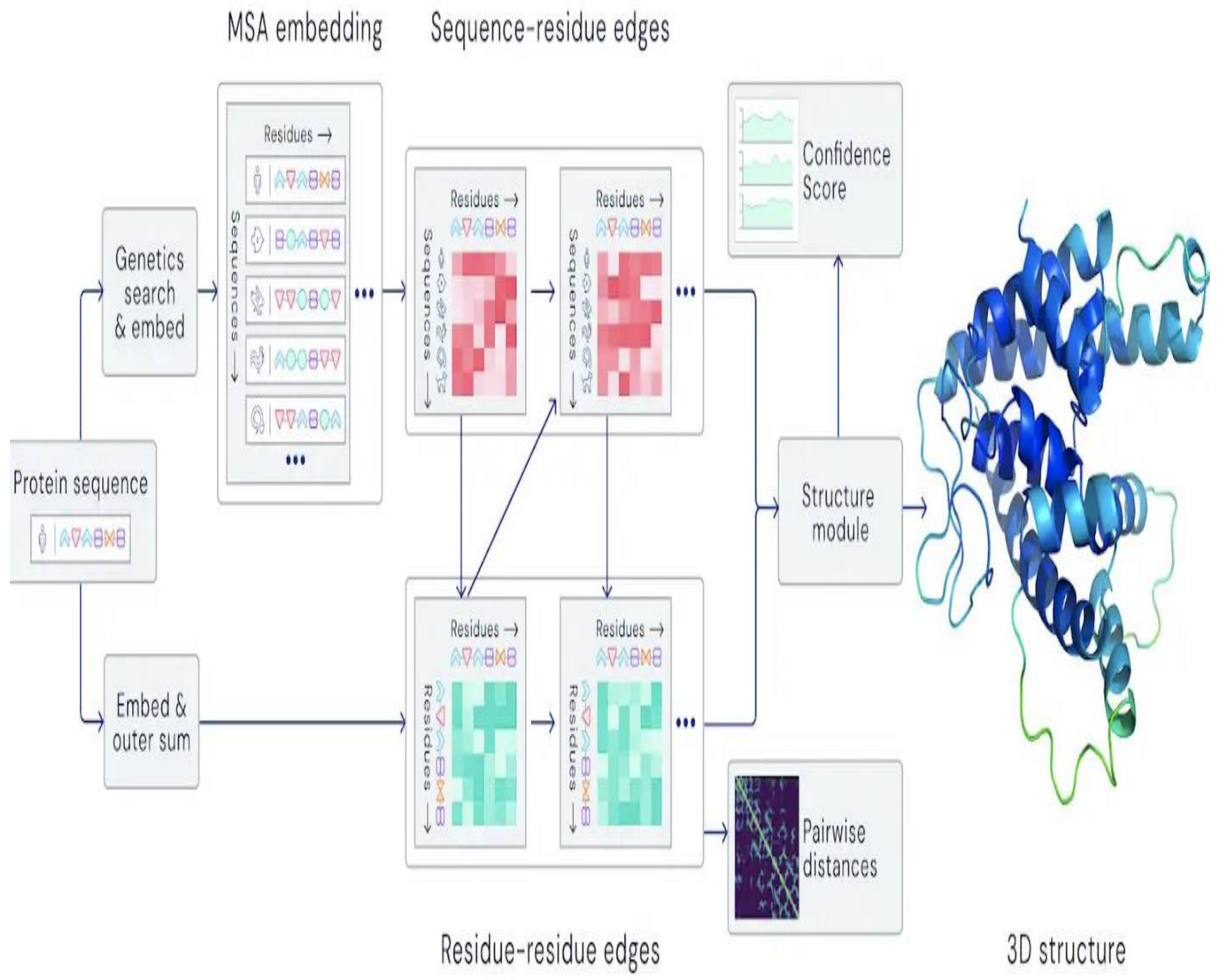
Nodes: Amino acids (residues) in a protein sequence.

Edges: Defined by spatial proximity between amino acids.

- These graphs capture important structural and functional relationships in proteins.

Captures long-range interactions between amino acids that influence folding.

- Helps predict tertiary structure from primary sequence.
- Facilitates learning of protein properties such as stability and function.



Applications:

- **Drug Discovery:** Identifying binding sites for drug molecules.
- **Protein Engineering:** Designing proteins with desired properties.
- **Disease Research:** Understanding mutations in protein structures.

Edge Level Task

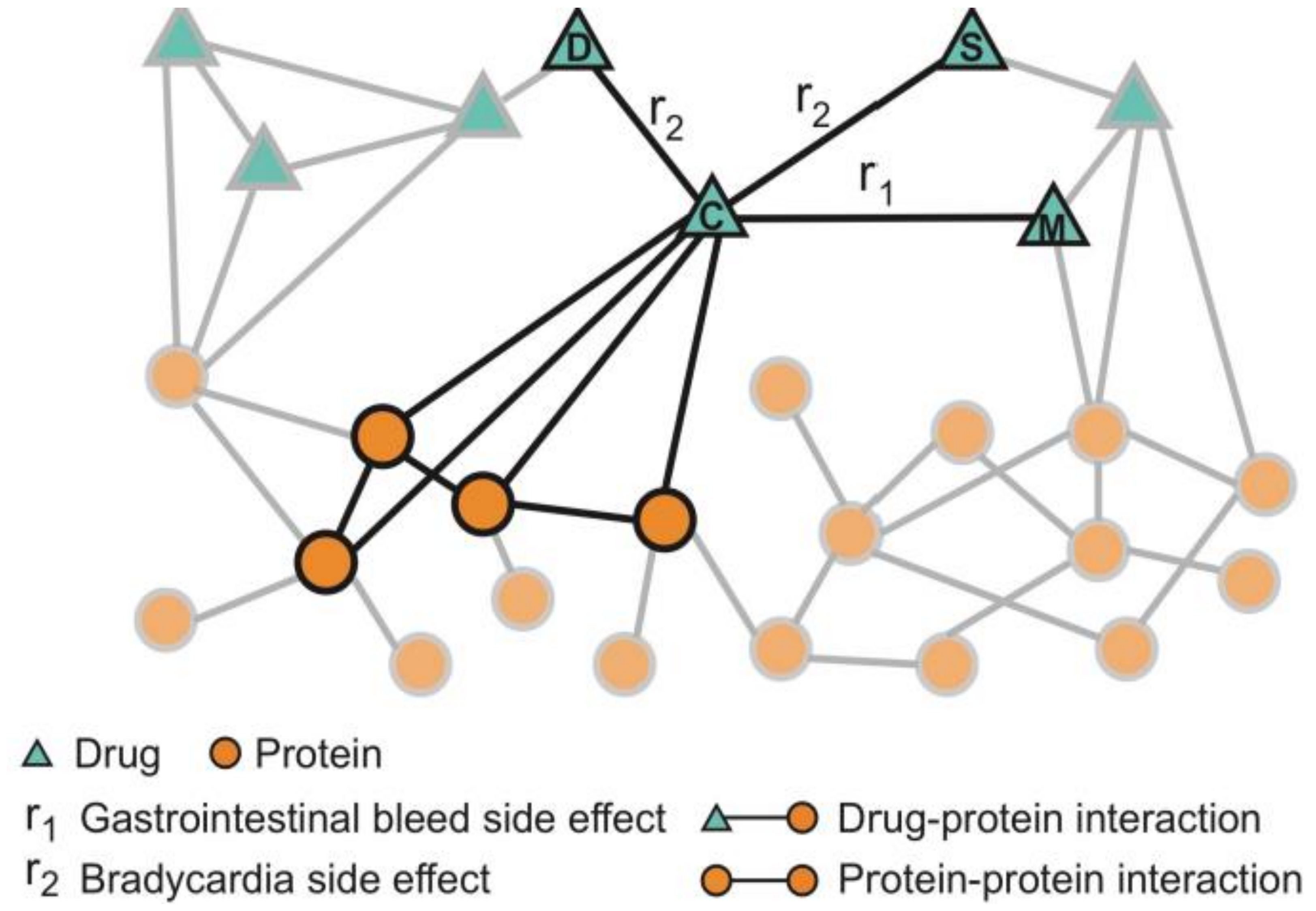
Predict properties or existence of edges.

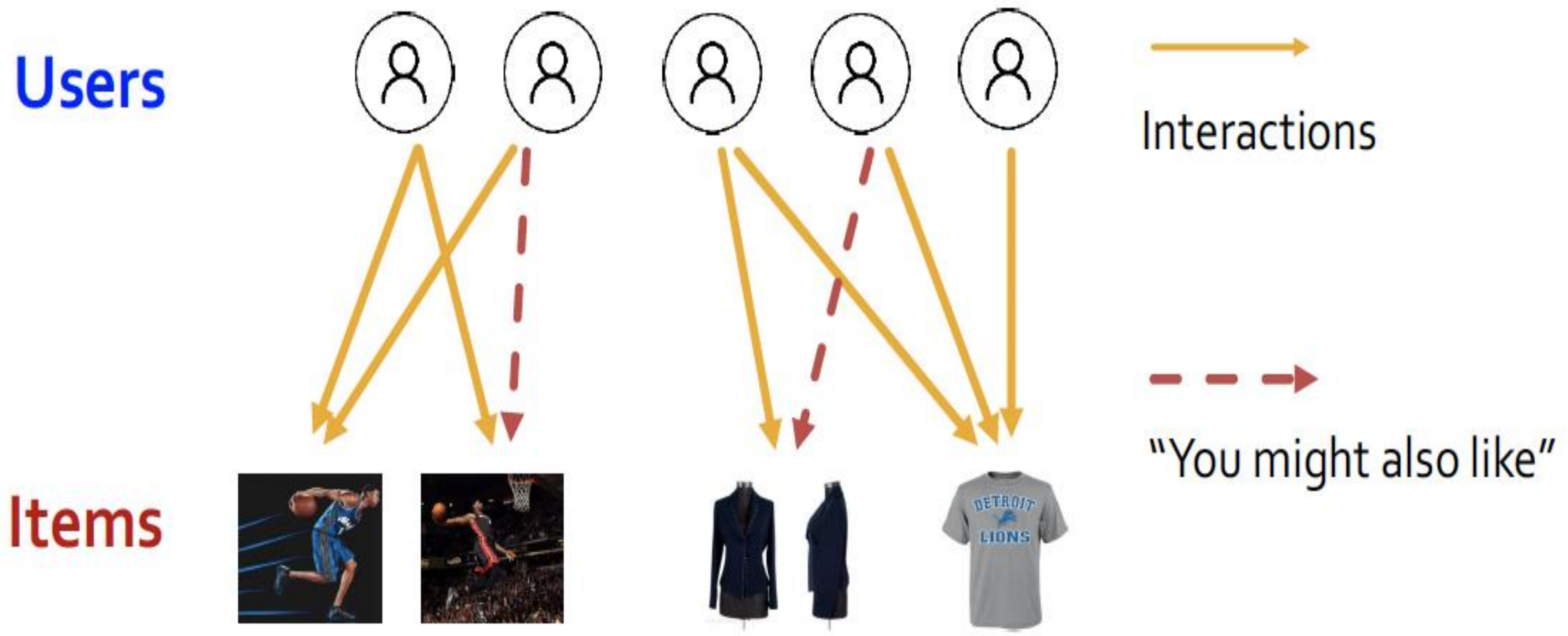
Common examples:

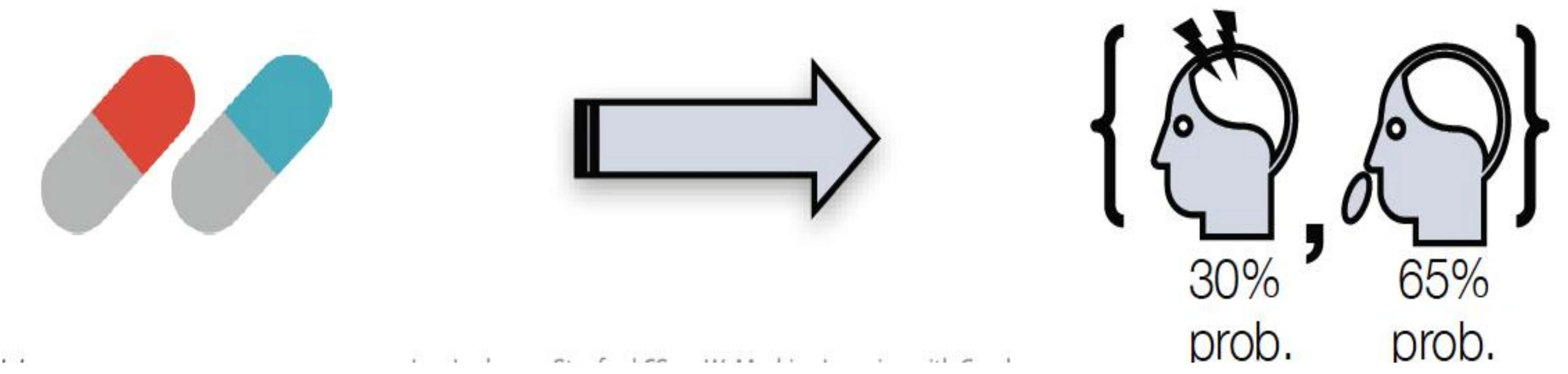
- ▶ Link Prediction: Determine if an edge should exist (e.g., friend recommendations on social media).
- ▶ Edge Classification: Categorize relationships (e.g., sentiment analysis in social interactions).

Applications:

- ▶ Recommendation systems
- ▶ Fraud detection in transactions
- ▶ Predicting drug-target interactions


D
S
C
M
r2
r2
r1
Drug
Protein
r1 Gastrointestinal bleed side effect
r2 Bradycardia side effect
Drug-protein interaction
Protein-protein interaction





Community Level Task

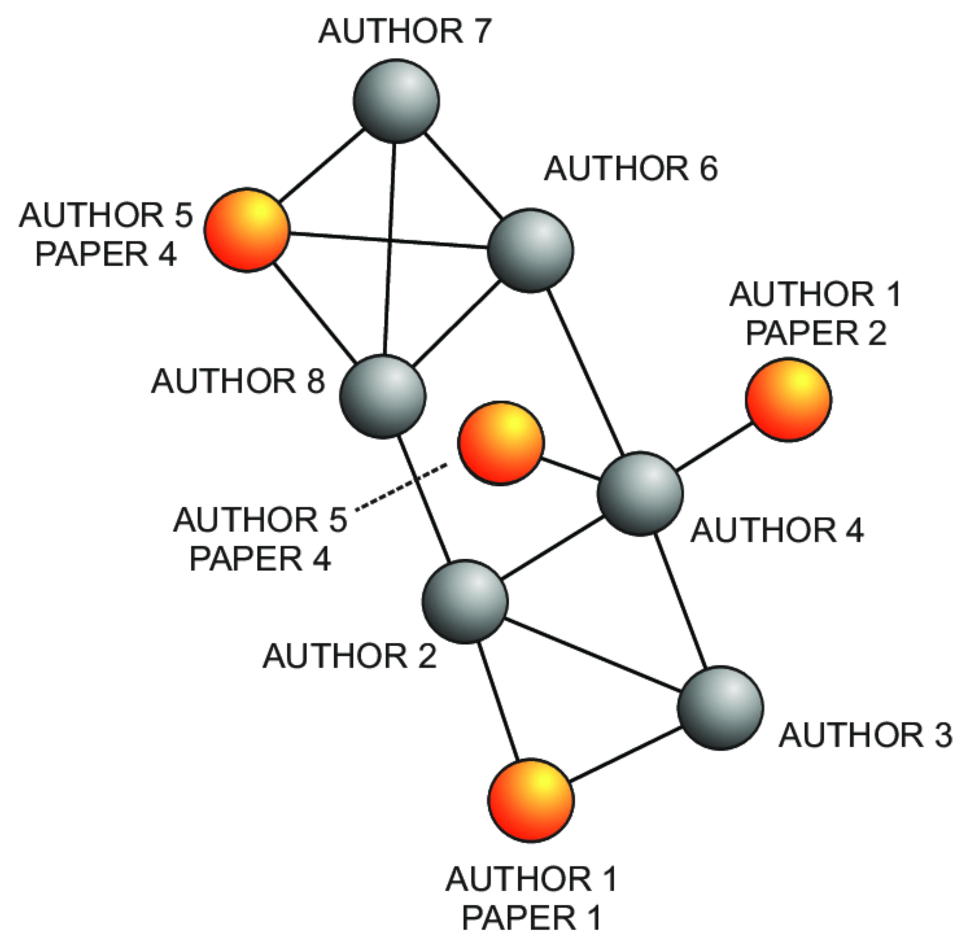
Identify groups of nodes with similar properties or high connectivity.

Common examples:

- Community Detection: Identifying clusters of related nodes (e.g., social network communities).
- Graph Partitioning: Dividing a graph into smaller, meaningful subgraphs.

Applications:

- Social network analysis (detecting influencer groups)
- Biological networks (identifying protein complexes)
- Market segmentation in business analytics



Graph Level Task

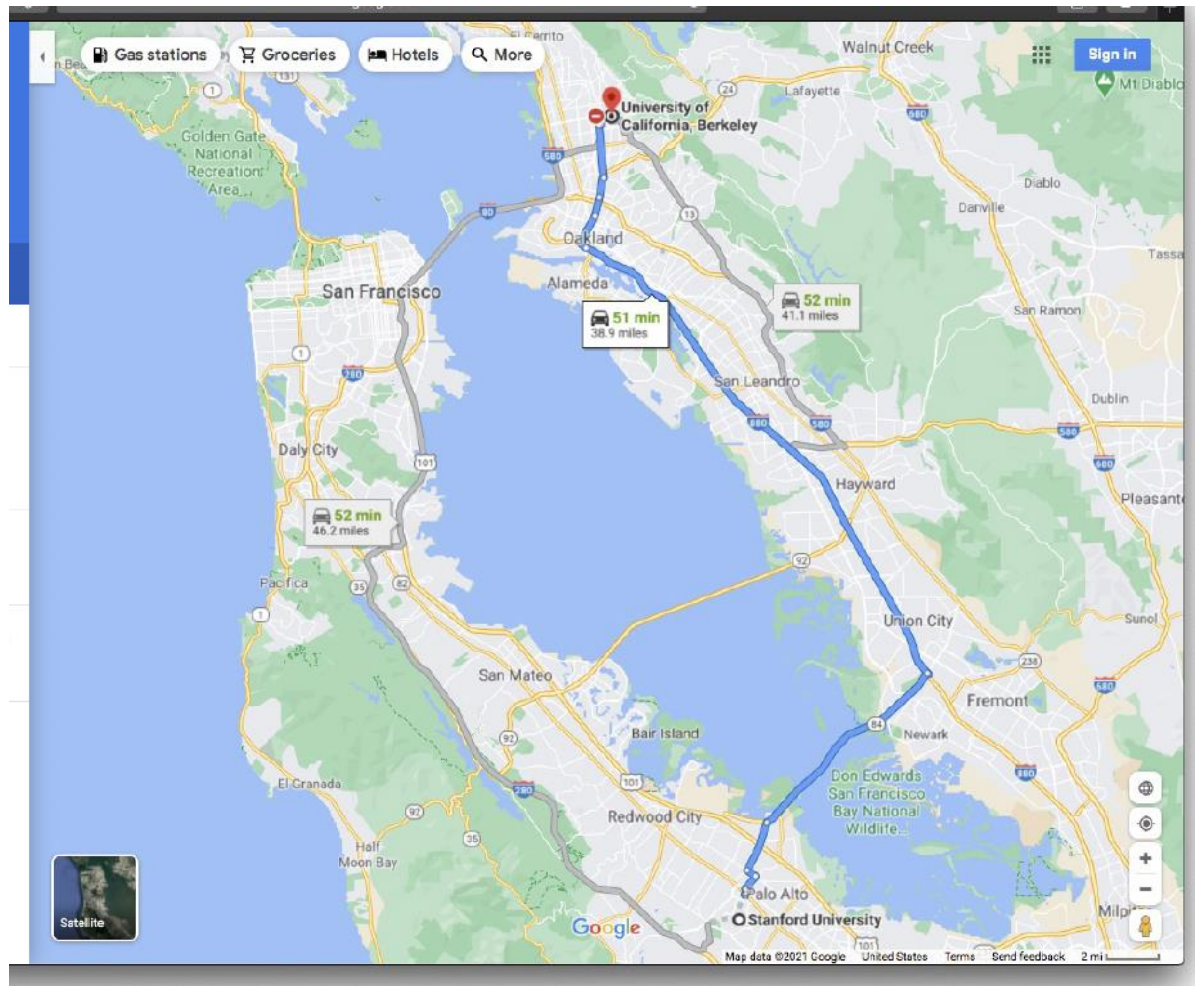
Analyze entire graphs (or a subset) to understand global properties.

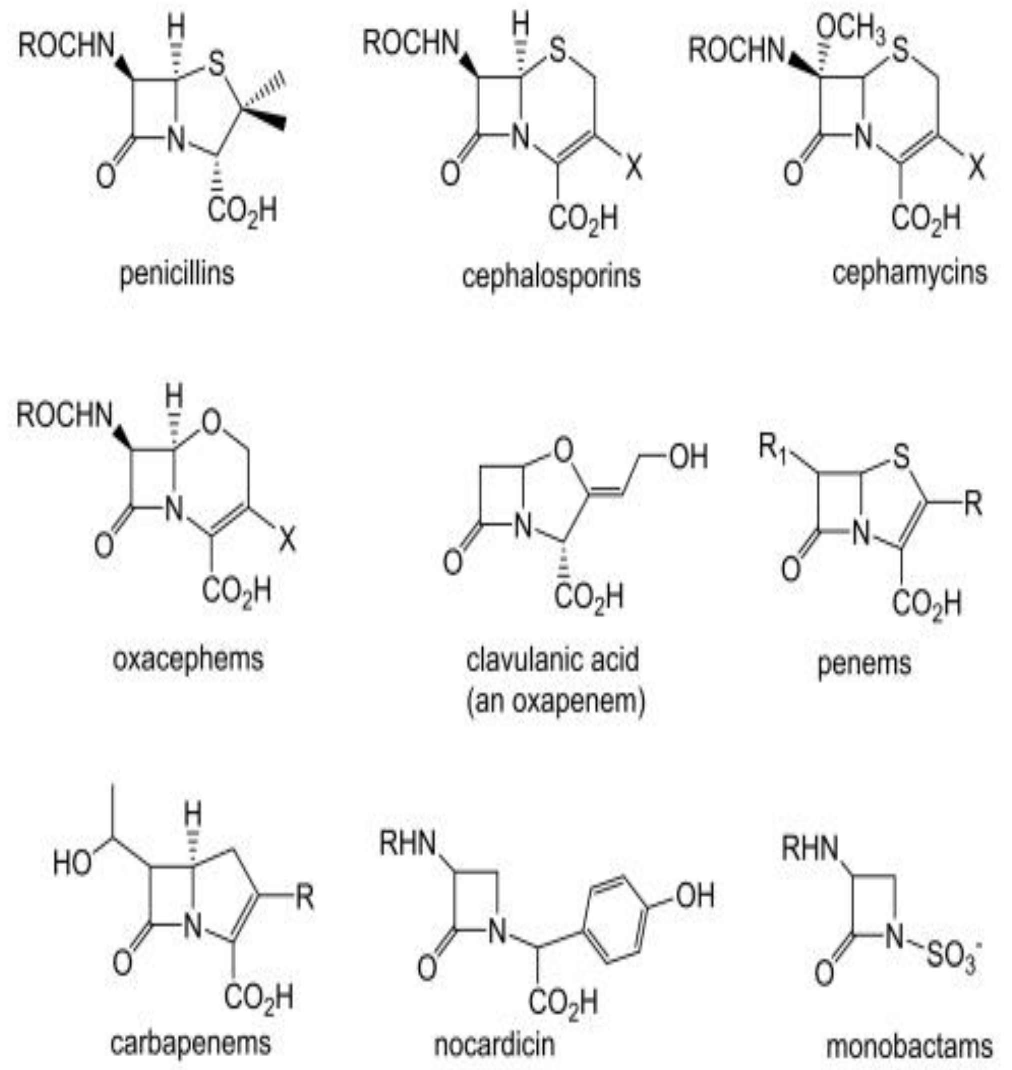
Common examples:

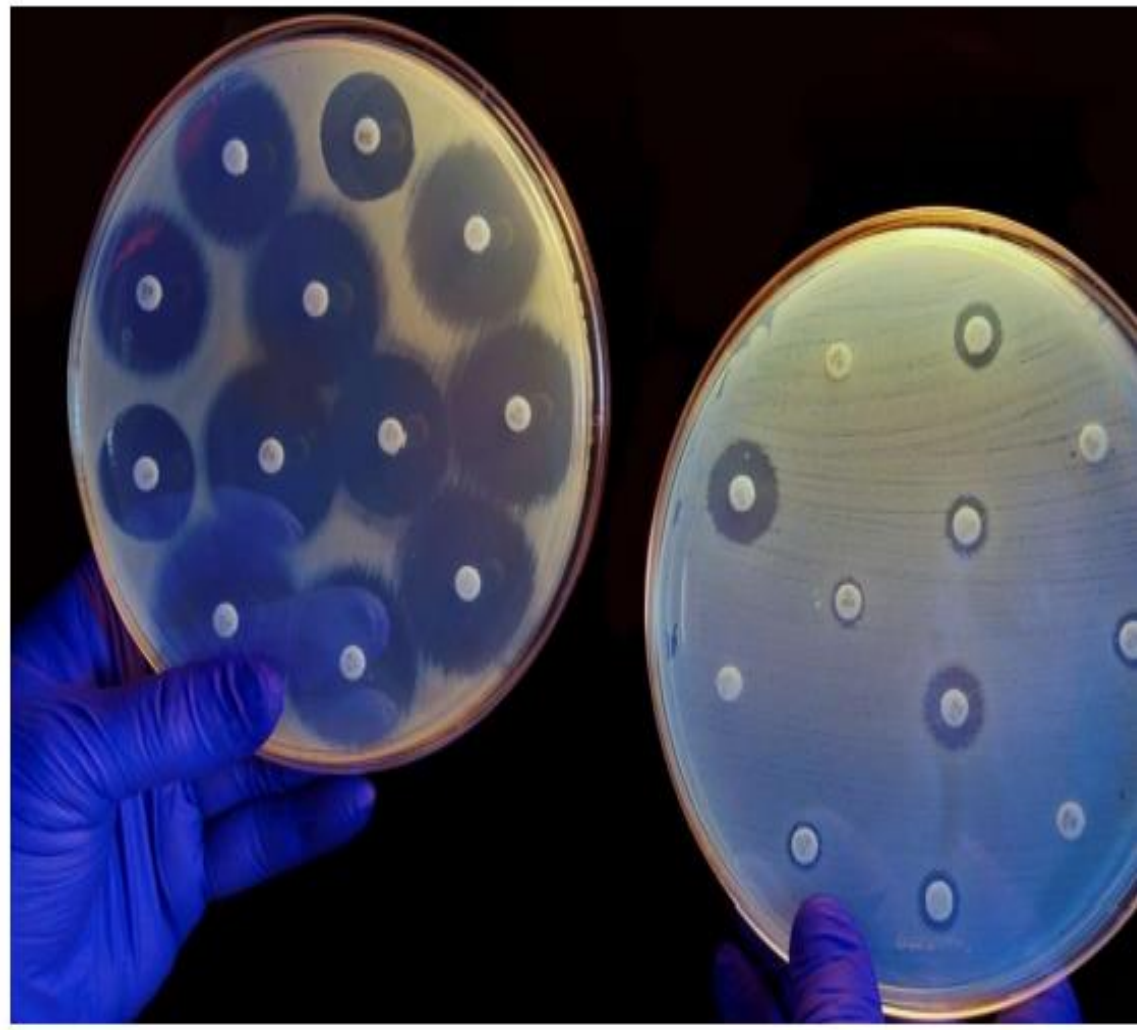
- Graph Classification: Assign labels to entire graphs (e.g., molecule toxicity prediction).
- Graph Clustering: Group similar graphs based on structure.

Applications:

- Chemical compound analysis
- Fake news detection
- Biological network analysis







Content

Graph Representation Learning

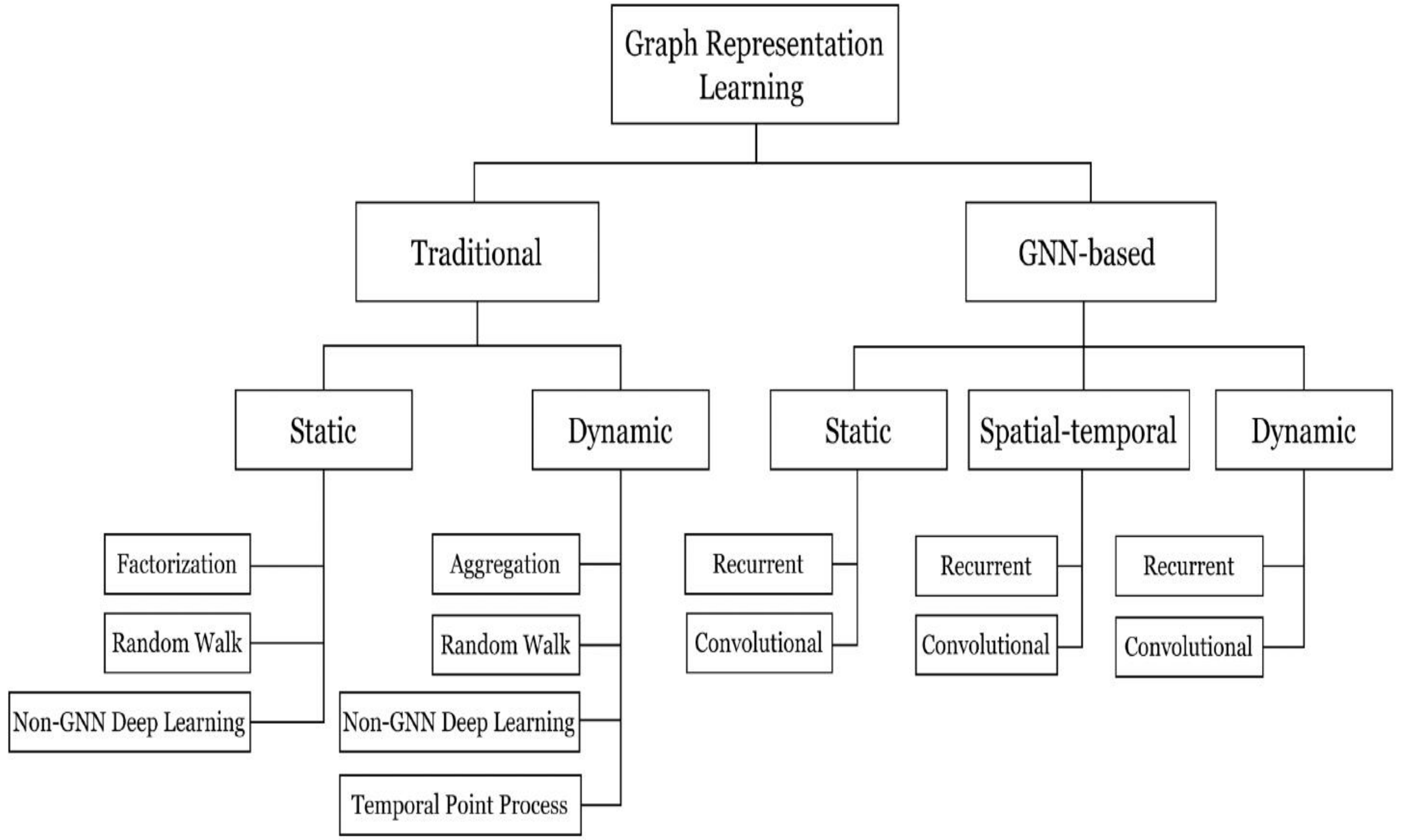
Graph representation learning methods transform **nodes, edges, or entire graphs** into continuous vector representations while preserving their **structural and attribute-based properties**. These embeddings make it possible to apply standard machine learning models to graph-based problems, such as **node classification, link prediction, and graph clustering**.

Traditional Graph Embedding Methods:

- **Matrix Factorization (Laplacian Eigenmaps, HOPE).**
- **Random Walk-Based (DeepWalk, node2vec).**
- **Deep Learning-Based (Graph Autoencoders).**

Graph Neural Networks (GNNs):

- **Spectral GNNs (GCN).**
- **Spatial GNNs (GraphSAGE, GAT).**
- **Dynamic GNNs (EvolveGCN).**



Categories of graph representation learning methods for non-heterogeneity-aware graphs (Khoshraftar and An, 2024).

Why Map Nodes to Embeddings

Goal: Represent nodes as dense vectors in a low-dimensional space while preserving network relationships.

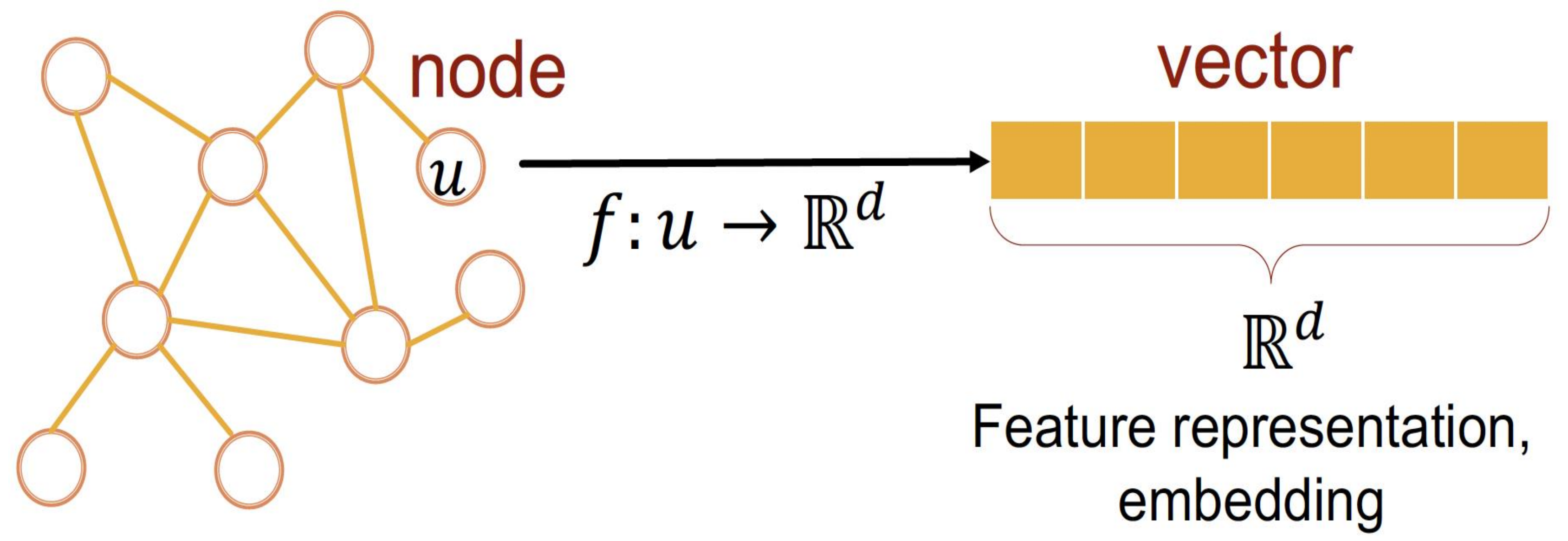
▶ The similarity between node embeddings should reflect their proximity in the original graph.

▶ Nodes with similar embeddings:

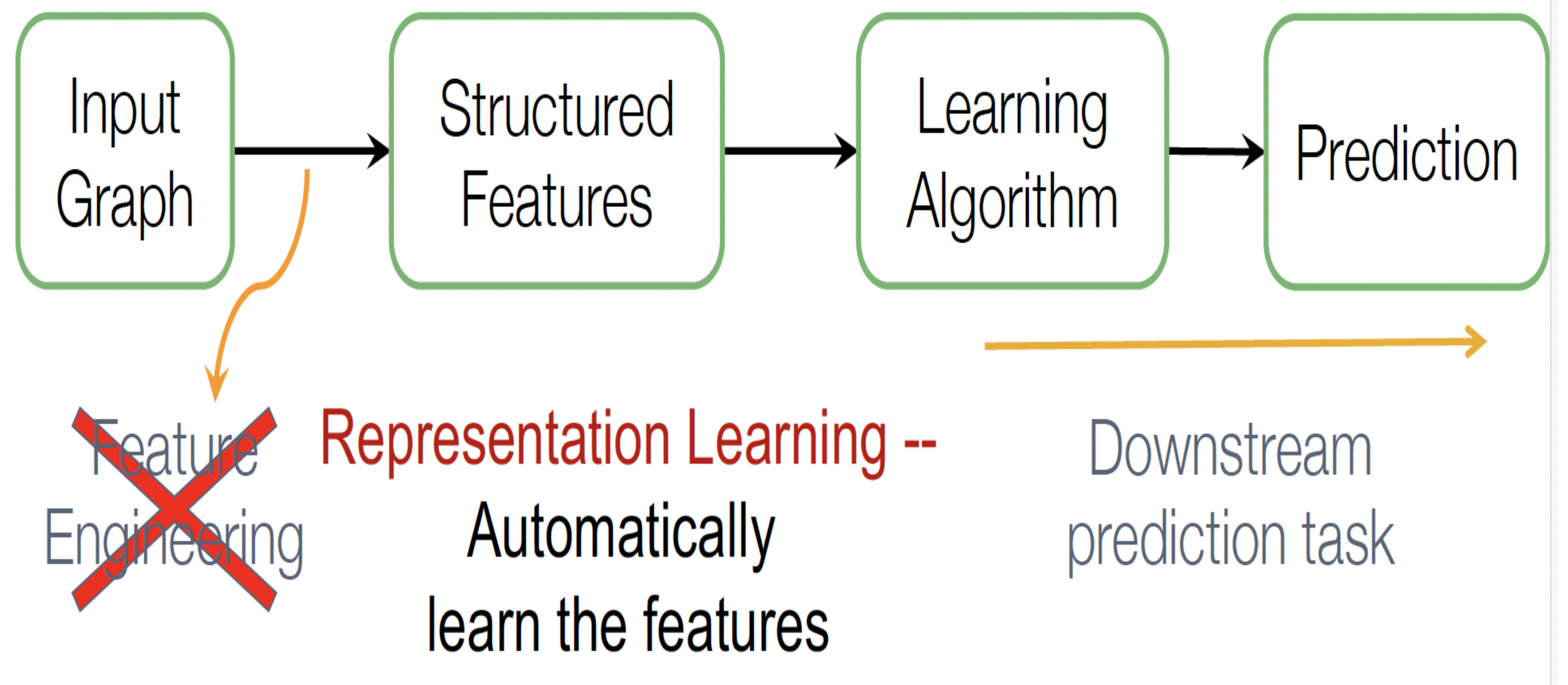
▶ Are structurally close (connected by an edge or via common neighbors).

▶ Share similar attributes (e.g., user preferences in a recommendation system).

▶ Encode network topology efficiently for various tasks.

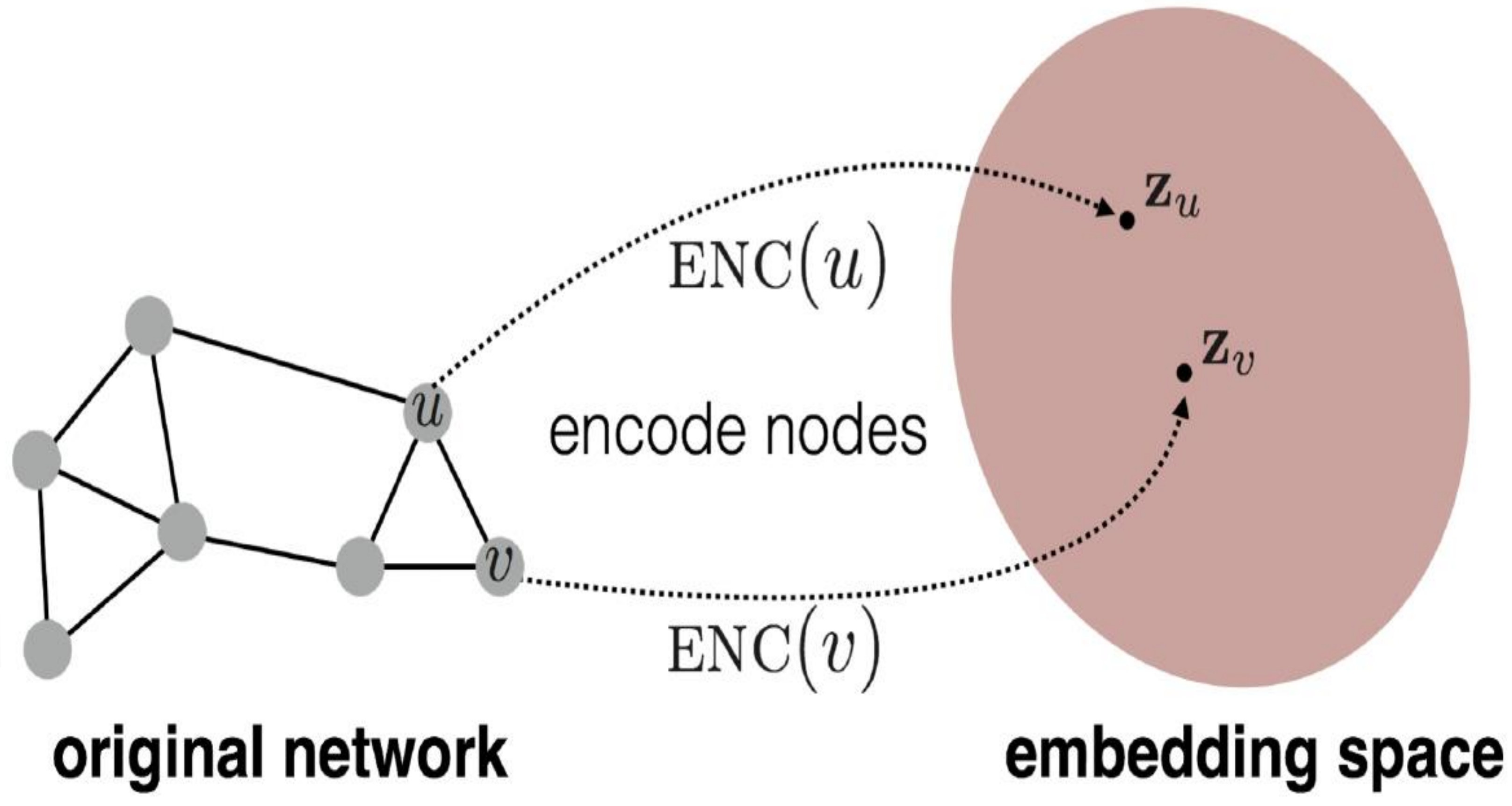

node
u
$f: u \rightarrow \mathbb{R}^d$
vector
$\mathbb{R}^d$
Feature representation, embedding

Project nodes into a latent space Geometric relations in this latent space correspond to relationships in the original graph


Input Graph
Structured Features
Learning Algorithm
Prediction
~~Feature Engineering~~
Representation Learning --
Automatically learn the features
Downstream prediction task

Tasks

- Node classification
- Link prediction
- Graph classification
- Anomalous node detection
- Clustering
- ....



An Encoder-Decoder Framework

Encoder maps each node $v \in \mathcal{V}$ to a vector embedding $z_v \in \mathbb{R}^d$

$$\text{ENC}(v): \mathcal{V} \rightarrow \mathbb{R}^d \qquad \text{ENC}(v) = Z[v]$$

$Z = (Z[v]) \in \mathbb{R}^{|\mathcal{V}| \times d}$ is a matrix containing the embedding vectors for all nodes. The encoder can also use node features or the local graph structure around each node as input to generate an embedding.

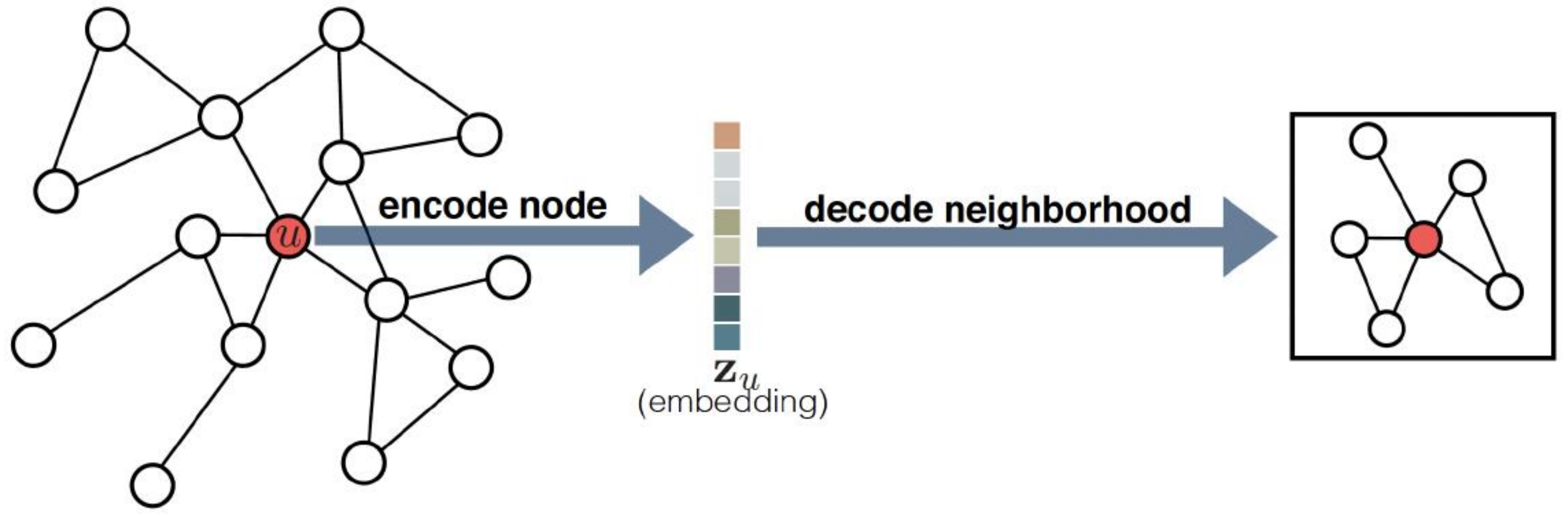
Decoder reconstructs certain graph statistics from the node embeddings that are generated by the encoder.

Pairwise decoders (similarity): DEC: $\mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^+$

Reconstruction of the relationship: optimize the encoder and decoder to minimize the reconstruction loss so that

$$\text{DEC}(\text{ENC}(u), \text{ENC}(v)) = \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \approx \mathbf{S}[u, v].$$

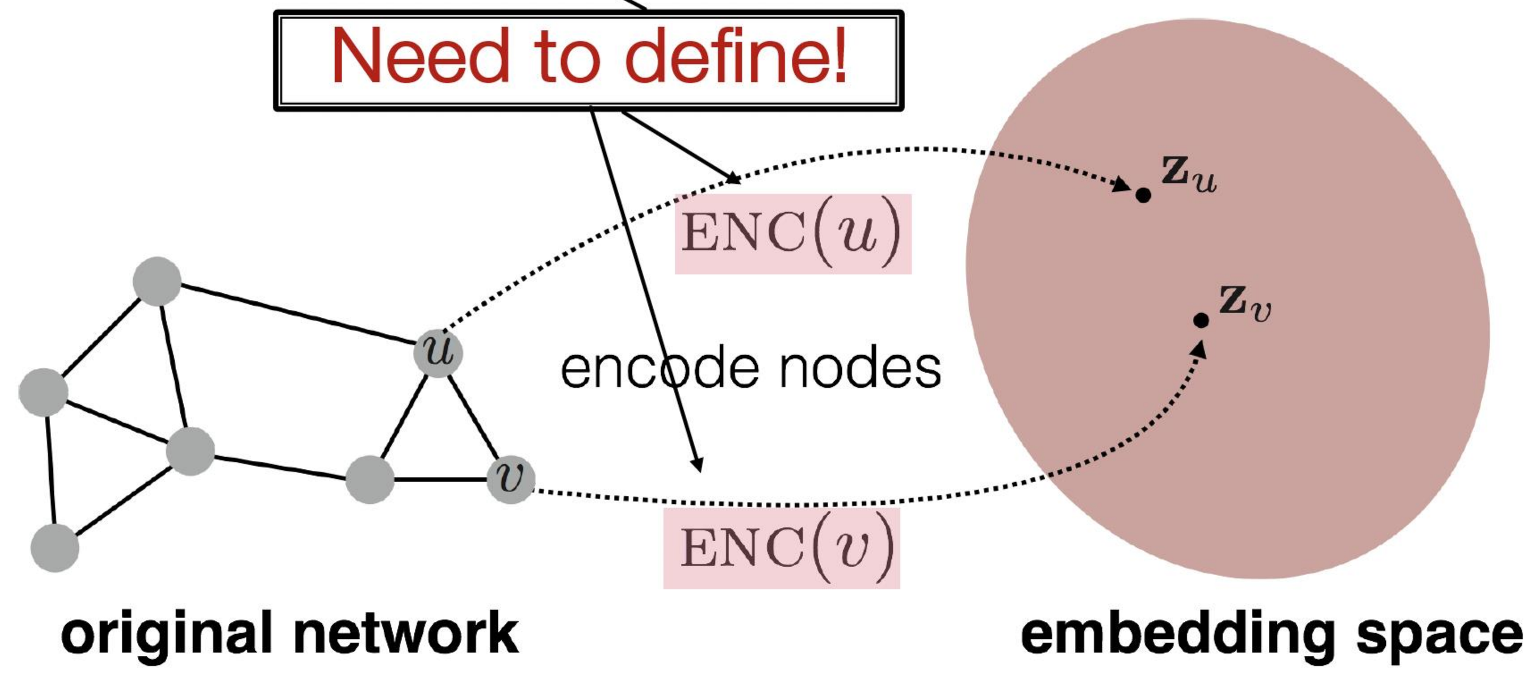
where $S[u, v]$ is **a graph-based similarity measure.**



Goal: $\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$

in the original network

Similarity of the embedding



Optimizing an Encoder-Decoder Model

Minimize an empirical reconstruction loss over a set of training node pairs

$$\mathcal{L} = \sum_{(u,v)\in \boxed{\mathcal{E}}} \ell\left(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v), \mathbf{S}[u,v]\right),$$

The overall objective is to train the encoder and the decoder so that pairwise node relationships can be effectively reconstructed.

Most approaches minimize the loss in using stochastic gradient descent, but there are certain instances when more specialized optimization methods can be used.

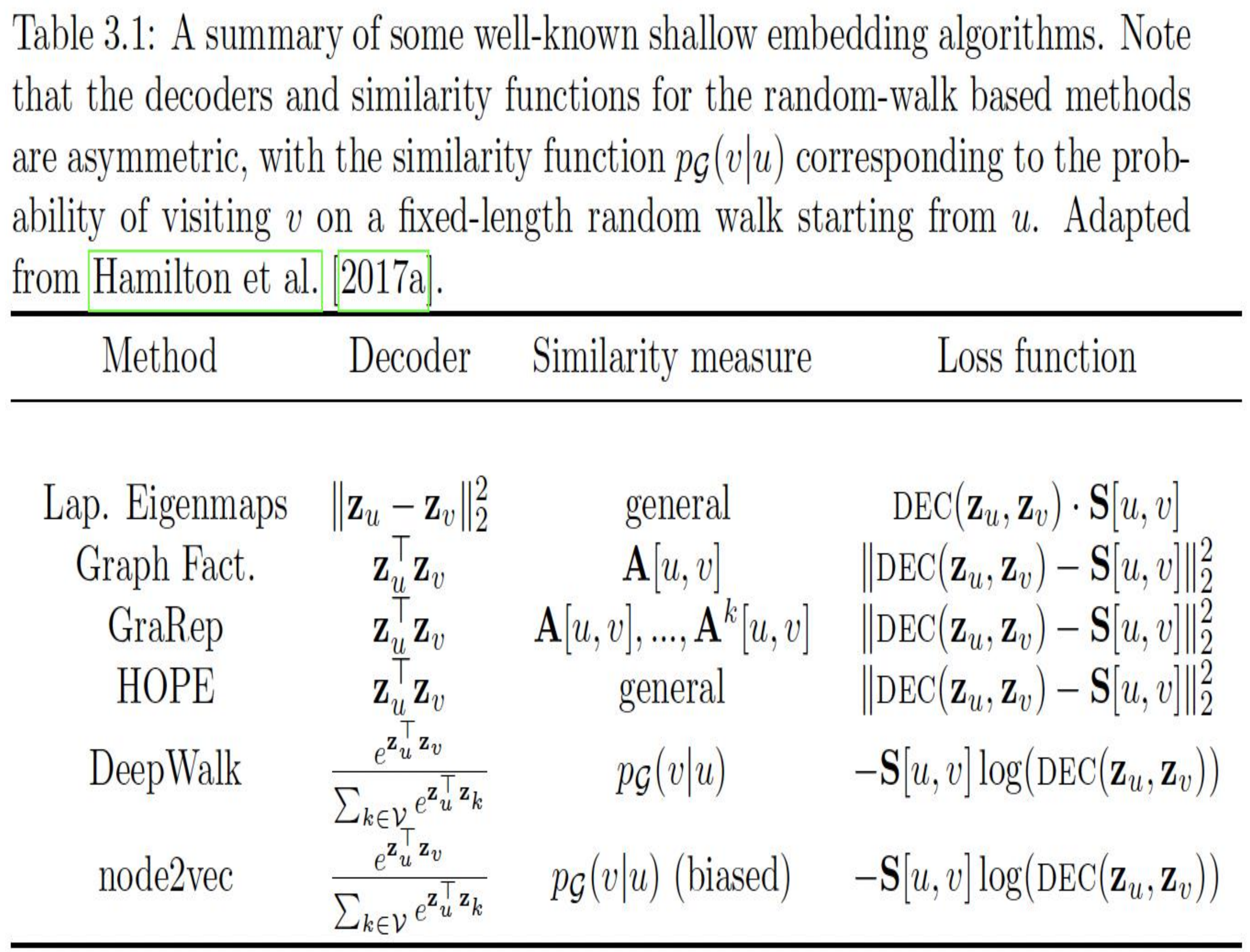
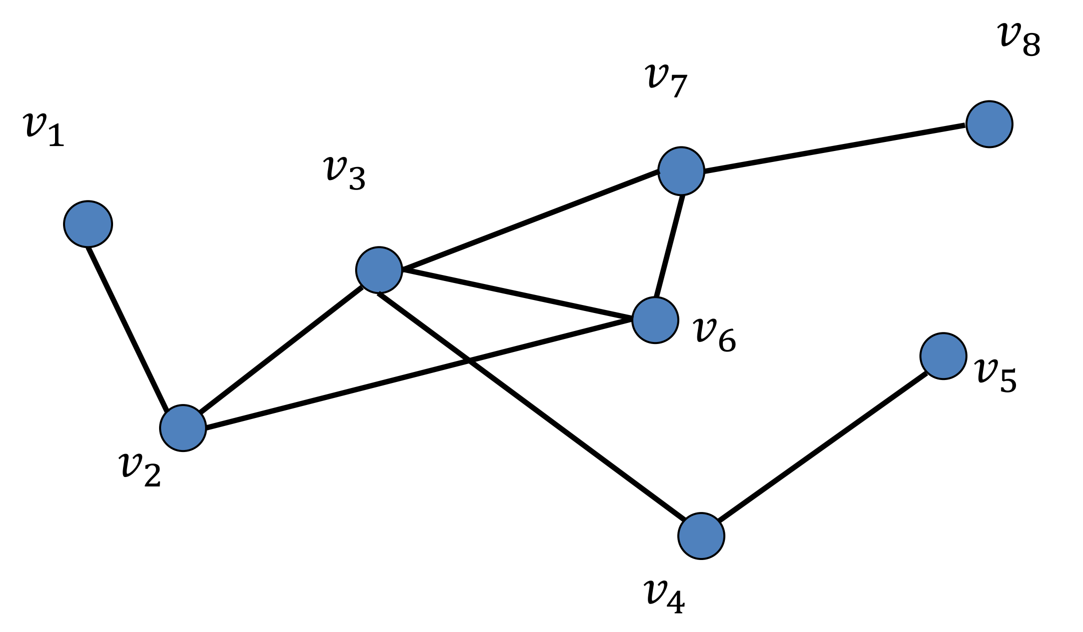


Table 3.1: A summary of some well-known shallow embedding algorithms. Note that the decoders and similarity functions for the random-walk based methods are asymmetric, with the similarity function $p_{\mathcal{G}}(v|u)$ corresponding to the probability of visiting v on a fixed-length random walk starting from u. Adapted from Hamilton et al. [2017a].

Method	Decoder	Similarity measure	Loss function
Lap. Eigenmaps	$\|\mathbf{z}_u - \mathbf{z}_v\|_2^2$	general	$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u,v]$
Graph Fact.	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u,v]$	$\|\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u,v]\|_2^2$
GraRep	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u,v], ..., \mathbf{A}^k[u,v]$	$\|\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u,v]\|_2^2$
HOPE	$\mathbf{z}_u^\top \mathbf{z}_v$	general	$\|\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u,v]\|_2^2$
DeepWalk	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k\in\mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v|u)$	$-\mathbf{S}[u,v]\log(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v))$
node2vec	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k\in\mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v|u)$ (biased)	$-\mathbf{S}[u,v]\log(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v))$

Matrix Representations of Graphs



Adjacency Matrix: $A[i,j] = 1$ if v_i is adjacent to v_j
$A[i,j] = 0$, otherwise

Degree Matrix: $\mathbf{D} = \mathrm{diag}(degree(v_1), \ldots, degree(v_N))$

Degree Matrix — Adjacency Matrix — Laplacian Matrix

$$\underbrace{\begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&3&0&0&0&0&0&0\\ 0&0&4&0&0&0&0&0\\ 0&0&0&2&0&0&0&0\\ 0&0&0&0&2&0&0&0\\ 0&0&0&0&0&4&0&0\\ 0&0&0&0&0&0&3&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix}}_{\boldsymbol{D}} - \underbrace{\begin{pmatrix} 0&1&0&0&0&0&0&0\\ 1&0&1&0&0&1&0&0\\ 0&1&0&1&0&1&1&0\\ 0&0&1&0&1&0&0&0\\ 0&0&0&1&0&1&0&0\\ 0&1&1&0&1&0&1&0\\ 0&0&1&0&0&1&0&1\\ 0&0&0&0&0&0&1&0 \end{pmatrix}}_{\boldsymbol{A}} = \underbrace{\begin{pmatrix} 1&-1&0&0&0&0&0&0\\ -1&3&-1&0&0&-1&0&0\\ 0&-1&4&-1&0&-1&-1&0\\ 0&0&-1&2&-1&0&0&0\\ 0&0&0&-1&2&-1&0&0\\ 0&-1&-1&0&-1&4&-1&0\\ 0&0&-1&0&0&-1&3&-1\\ 0&0&0&0&0&0&-1&1 \end{pmatrix}}_{\boldsymbol{L}}$$

Spectral graph theory. American Mathematical Soc.; 1997.

Laplacian Matrix as an Operator

Laplacian matrix is a difference operator:

$$\mathbf{h} = \mathbf{L}\mathbf{f} = (\mathbf{D} - \mathbf{A})\mathbf{f} = \mathbf{D}\mathbf{f} - \mathbf{A}\mathbf{f}$$

$$\mathbf{h}(i) = \sum_{v_j \in \mathcal{N}(v_i)} (\mathbf{f}(i) - \mathbf{f}(j))$$

Laplacian quadratic form:

$$\mathbf{f}^T \mathbf{L} \mathbf{f} = \frac{1}{2} \sum_{i,j=1}^{N} \mathbf{A}[i,j](\mathbf{f}(i) - \mathbf{f}(j))^2$$

“Smoothness” or “Frequency” of the signal f

The symmetric normalized Laplacian is

$$\mathbf{L}_{\text{sym}} = \mathbf{D}^{-\frac{1}{2}} \mathbf{L} \mathbf{D}^{-\frac{1}{2}},$$

The random walk Laplacian is

$$\mathbf{L}_{\text{RW}} = \mathbf{D}^{-1}\mathbf{L}$$

- L=D-A is symmetric and positive semi-definite.
- The geometric multiplicity of the 0 eigenvalue of the Laplacian L corresponds to the number of connected components in the graph.

Spectral Clustering methods:

- Select the k smallest nonzero eigenvectors.
- Construct the spectral embedding matrix (SEM).
- Perform clustering methods on SEM.

Eigen-decomposition of Laplacian Matrix

Laplacian matrix has a complete set of orthonormal eigenvectors:

$$\mathbf{L} = \underbrace{\begin{bmatrix} | & & | \\ \mathbf{u}_0 & \cdots & \mathbf{u}_{N-1} \\ | & & | \end{bmatrix}}_{\boldsymbol{U}} \underbrace{\begin{bmatrix} \lambda_0 & & 0 \\ & \ddots & \\ 0 & & \lambda_{N-1} \end{bmatrix}}_{\boldsymbol{\Lambda}} \underbrace{\begin{bmatrix} — & \mathbf{u}_0 & — \\ & \vdots & \\ — & \mathbf{u}_{N-1} & — \end{bmatrix}}_{\boldsymbol{U}^{\boldsymbol{T}}}$$

Eigenvalues are sorted non-decreasingly:

$$0 = \lambda_0 < \lambda_1 \leq \cdots \lambda_{N-1}$$

Factorization-based approaches

$$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) = \|\mathbf{z}_u - \mathbf{z}_v\|_2^2$$

$$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) = \mathbf{z}_u^\top \mathbf{z}_v.$$

$$\mathcal{L} = \sum \mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]$$

$$\mathcal{L} \approx \|\mathbf{Z}\mathbf{Z}^\top - \mathbf{S}\|_2^2$$

$\mathcal{L}$=2tr((D-S)ZZ’)

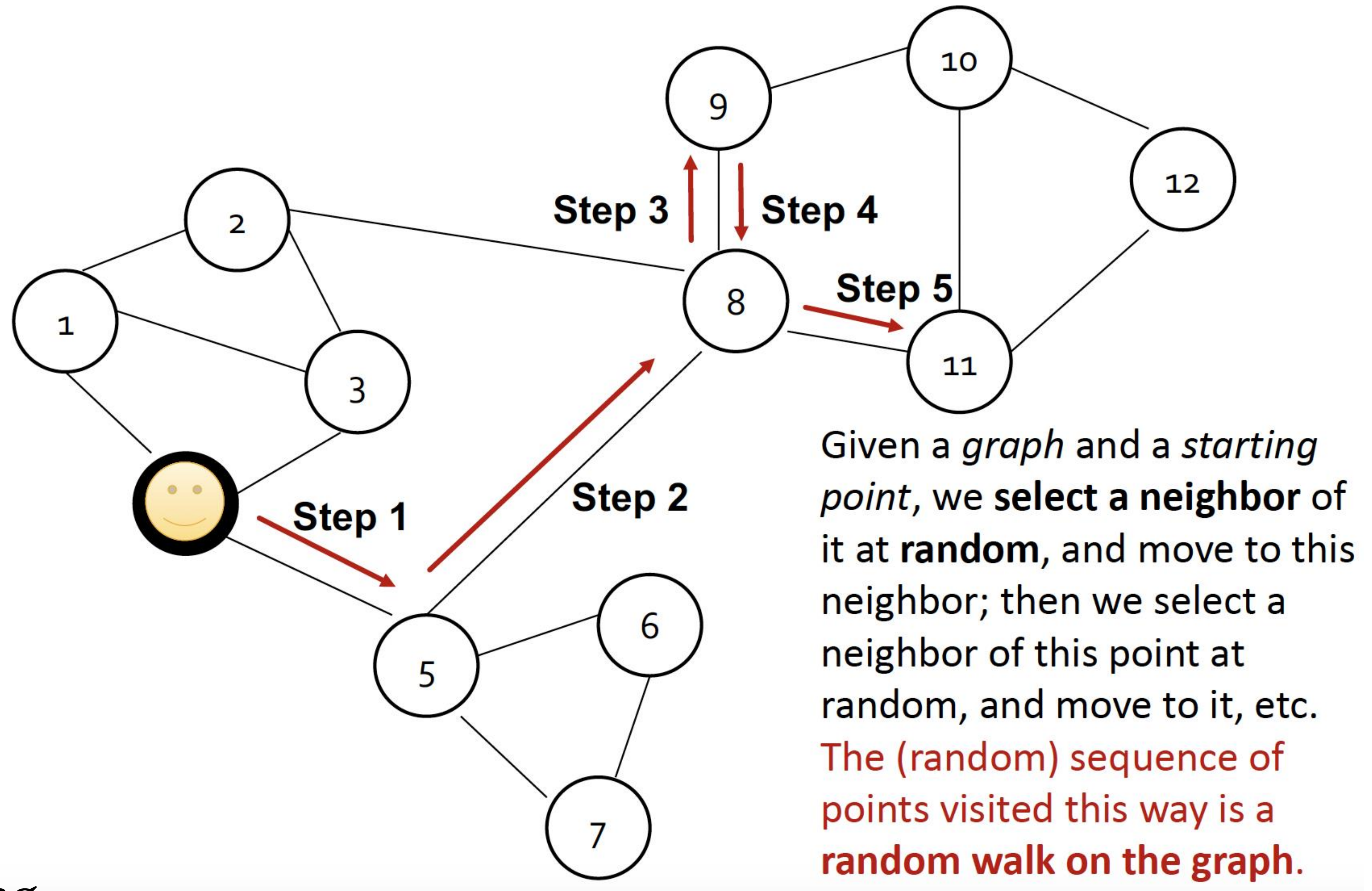
- If S=A is the Laplacian matrix, then the node embeddings that minimize the loss in the loss function are identical to the solution for spectral clustering.
- If we assume the embeddings are d-dimensional, then the optimal solution that minimizes the loss function is given by the d smallest eigenvectors of the Laplacian (excluding the eigenvector of all ones).

Random Walk Embeddings

Random Walks generate node sequences that capture both local and global structural features.

▶ **Motivation:** Node pairs that frequently co-occur in random walks are considered structurally similar.

▶ **Popular methods:** DeepWalk (Perozzi et al. 2014) and node2vec (Grover and Leskovec 2016).



Given a *graph* and a *starting point*, we **select a neighbor** of it at **random**, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc. The (random) sequence of points visited this way is a **random walk on the graph**.

Why Random Walks?

❖ **Multi-hop context:** A random walk naturally explores k-hop neighborhoods, capturing global structure.

❖ **Probabilistic exploration:** Each walk is governed by transition probabilities P(v |u).

❖ **Efficiency:** Only node pairs co-occurring in walks matter for training, reducing complexity compared to matrix factorization.

❖ **Flexibility:** Parameters like walk length (t), number of walks (R), and biases (p, q) in node2vec control exploration.

Core Idea of Random-Walk Embeddings

Walk Generation: For each node u, run R random walks of length t, forming sequences $[u, v_1, v_2, \dots, v_t]$.

Co-occurrence Collection: Within each walk, consider a context window w. Node pairs (u, v) appearing within w steps are recorded.

Train Model: Often use a Skip-Gram approach maximizing $P(v|u)$ for co-occurring pairs.

Output: Each node u is assigned an embedding $z_u \in \mathbb{R}^d$ reflecting its structural context.

$$\arg\min_{z} \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

$$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) \triangleq \frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{v_k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}} \approx p_{\mathcal{G},T}(v|u),$$

- **Walk length** (t): Longer walks capture more distant relationships but increase cost.
- **Number of walks** (R): More walks => more coverage and co-occurrences.
- **Window size** (w): Defines local context in a walk sequence.
- **Embedding dimension** (d): Higher d can capture more complex structure.
- p, q **in node2vec**: Balance BFS (local) vs. DFS (global) exploration.

$$\mathcal{L} = \underbrace{\boxed{\sum_{u \in V}}}_{\text{sum over all nodes } u} \underbrace{\boxed{\sum_{v \in N_R(u)}}}_{\text{sum over nodes } v \text{ seen on random walks starting from } u} - \log\Bigg(\underbrace{\boxed{\frac{\exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_n)}}}_{\text{predicted probability of } u \text{ and } v \text{ co-occuring on random walk}}\Bigg)$$

DeepWalk and node2vec

Key Steps (Perozzi et al., 2014):

- **Random Walk Generation**:

$$\text{Walk}(u) = [u, v_1, v_2, \ldots, v_t], \quad P(v_{i+1}|v_i) = \frac{1}{|\mathcal{N}(v_i)|} \tag{1}$$

  where $\mathcal{N}(v_i)$ are neighbors of v_i.

- **Skip-Gram Training**:

$$\max_z \sum_{(u,v)\in\text{co-occurrences}} \log P(v|u; z) \tag{2}$$

- **Window Size (w)**: Defines how far apart nodes can be in a walk to be considered co-occurring.
- Embeddings z_u learned via gradient-based optimization.

- Introduces parameters p (return parameter) and q (in-out parameter) to guide the transition probabilities.
- Transition probability:

$$P(v_{i+1} = x|v_i = y, v_{i-1} = z) \propto \alpha_{pq}(z, x) \cdot w_{yx}$$

- $\alpha_{pq}(z, x)$ controls BFS/DFS bias:

$$\alpha_{pq}(z, x) = \begin{cases} 1/p, & \text{if } d(z, x) = 0, \\ 1, & \text{if } d(z, x) = 1, \\ 1/q, & \text{if } d(z, x) = 2 \end{cases}$$

  where $d(z, x)$ is the shortest path distance between z and
- Skip-Gram objective as in DeepWalk.
- $w_{y,x}$ = edge weight between y and x (often 1 for unweighted graphs)

Skip-Gram with Negative Sampling

Motivation: In large graphs, computing the full softmax over all nodes is expensive. Negative sampling approximates this by only comparing each positive pair to a small set of negative pairs.

$$\log\left(\frac{\exp(\mathbf{z}_u^{\mathrm{T}}\mathbf{z}_v)}{\sum_{n\in V}\exp(\mathbf{z}_u^{\mathrm{T}}\mathbf{z}_n)}\right)$$

$$\approx \log\left(\sigma(\mathbf{z}_u^{\mathrm{T}}\mathbf{z}_v)\right) + \sum_{i=1}^{k}\log\left(\sigma(-\mathbf{z}_u^{\mathrm{T}}\mathbf{z}_{n_i})\right), \underbrace{n_i \sim P_V}_{\text{random distribution over nodes}}$$

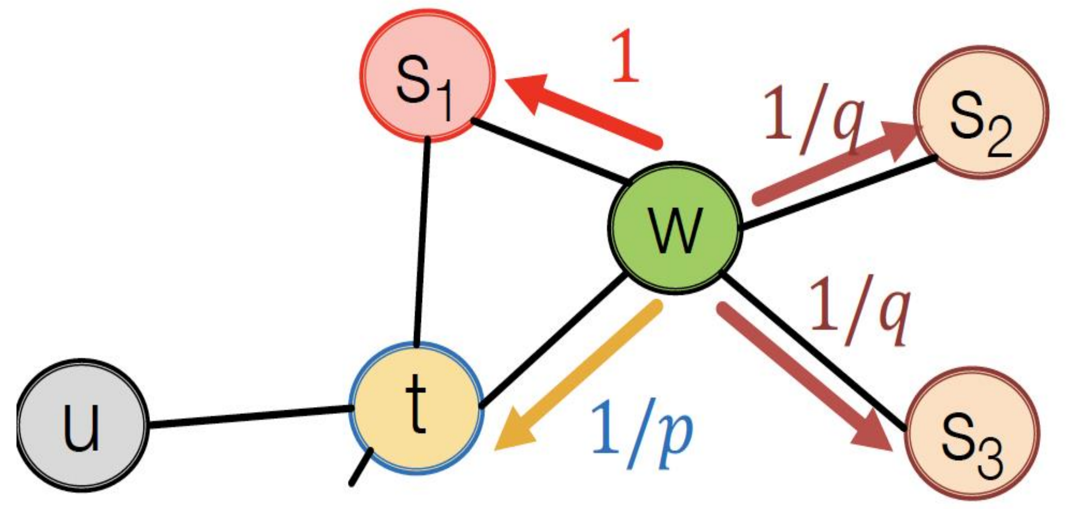
- Sample k negative nodes n_i each with prob. proportional to its degree.
- Two considerations for k (# negative samples):
  1. Higher k gives more robust estimates
  2. Higher k corresponds to higher bias on negative events

  In practice k =5-20.

BFS and DFS

- Walker came over edge (s_1, w) and is at w.

  How to set edge transition probabilities?



$w \rightarrow$

Target x	Prob.	Dist. (t, x)
t	$1/p$	0
s_1	1	1
s_2	$1/q$	2
s_3	$1/q$	2

Unnormalized transition prob. segmented based on distance from t

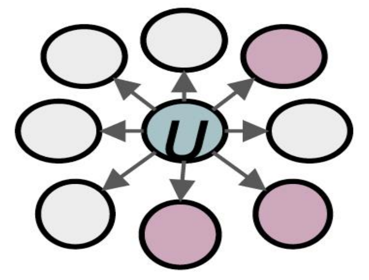
- **BFS-like** walk: Low value of p
- **DFS-like** walk: Low value of q

$N_R(u)$ are the nodes visited by the biased walk

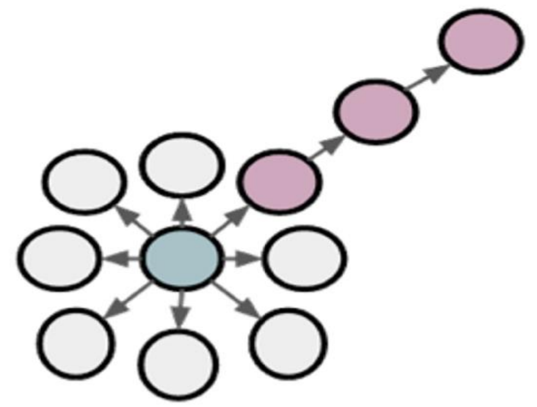
- **Return parameter p:**
  - Return back to the previous node
- **In-out parameter q:**
  - Moving outwards (DFS) vs. inwards (BFS) from the previous node
  - Intuitively, q is the "ratio" of BFS vs. DFS

Breadth-First Search (BFS)

▶ Explores neighbors of a node first, layer by layer.
▶ Naturally finds shortest paths in unweighted graphs.
▶ Focuses on the breadth of the graph around each node.



BFS:
$N_R(\cdot)$ will provide a micro-view of neighbourhood



DFS:
$N_R(\cdot)$ will provide a macro-view of neighbourhood

Depth-First Search (DFS)

▶ Proceeds along one path until no unvisited neighbors remain, then backtracks.
▶ Good for enumerating deep paths or checking connectivity.
▶ Focuses on the depth of the graph from each node.

Graph Autoencoder (GAE)

Definition: A Graph Autoencoder (GAE) is a neural network that learns low-dimensional node embeddings by reconstructing graph information (e.g., adjacency or node features).

Key Components:

1. **Encoder (GCN-based)**
   - Let $\mathbf{X} \in \mathbb{R}^{N \times d}$ be the node feature matrix for N nodes with d-dimensional features.
   - Let $\mathbf{A} \in \{0,1\}^{N \times N}$ be the adjacency matrix.
   - A simplified graph convolution layer:

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

   where $\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, and $\widetilde{\mathbf{D}}$ is the diagonal degree matrix of $\widetilde{\mathbf{A}}$.
   - After L layers, we obtain latent embeddings $\mathbf{Z} = \mathbf{H}^{(L)} \in \mathbb{R}^{N \times k}$.

2. **Decoder (Inner-Product Reconstruction)**

   $$\hat{\mathbf{A}} = \sigma\left(\mathbf{Z}\mathbf{Z}^\top\right).$$

   where σ is an elementwise nonlinearity (e.g., sigmoid).
   - $\hat{A}_{ij}$ estimates the existence probability of edge (i,j).

Objective: Minimize the reconstruction loss between the original graph and the reconstructed output.

- **Objective:**

  $$\min_{\theta} \quad \mathcal{L}_{\text{recon}},$$

  where θ are all learnable parameters of the encoder (and decoder).

- Often use **binary cross-entropy** on adjacency reconstruction:

  $$\mathcal{L}_{\text{recon}} = -\sum_{i=1}^{N}\sum_{j=1}^{N}\left[\mathbf{A}_{ij}\log\left(\hat{\mathbf{A}}_{ij}\right) + \left(1 - \mathbf{A}_{ij}\right)\log\left(1 - \hat{\mathbf{A}}_{ij}\right)\right].$$

t-SNE

- **t-SNE (t-Distributed Stochastic Neighbor Embedding)** is a powerful tool for visualizing high-dimensional data by preserving local neighborhood relationships.
- It transforms high-dimensional similarities into probability distributions and minimizes the KL divergence between these distributions.
- Careful tuning of hyperparameters (perplexity, learning rate, iterations) is crucial for meaningful visualizations.

For data points $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^D$:

- Define the conditional probability:

$$p_{j|i} = \frac{\exp\left(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2\right)}.$$

- σ_i is determined such that the **perplexity** of the distribution equals a user-defined value.
- The symmetric joint probability is:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}, \quad i \neq j,$$

with $p_{ii} = 0$, where N is the total number of points.

After mapping high-dimensional points $\mathbf{x}_i$ to low-dimensional points $\mathbf{y}_i \in \mathbb{R}^d$:

- Define the similarity using a Student's t-distribution (with one degree of freedom):

$$q_{ij} = \frac{\left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2\right)^{-1}},$$

for $i \neq j$, with $q_{ii} = 0$.

- The heavy-tailed t-distribution alleviates the **crowding problem** by allowing moderate distances to be represented more faithfully.

The goal is to minimize the Kullback-Leibler (KL) divergence between the high-dimensional and low-dimensional distributions:

$$\mathcal{C} = \sum_i \sum_j p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right)$$

$$\frac{\partial \mathcal{C}}{\partial \mathbf{y}_i} = 4 \sum_j (p_{ij} - q_{ij})\,(\mathbf{y}_i - \mathbf{y}_j)\,\left(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2\right)^{-1}$$

Combining Graph Autoencoders & t-SNE

Learn Node Embeddings:

$$\mathbf{Z} \in \mathbb{R}^{N \times k} = \text{GAE}(\mathbf{X}, \mathbf{A}).$$

Apply t-SNE:

$$\mathbf{Y} = \text{t-SNE}(\mathbf{Z}),$$

where $\mathbf{Y} \in \mathbb{R}^{N \times 2}$ or $\mathbb{R}^{N \times 3}$.

Interpretation: Points close in **Y**-space share similar graph structure/features in **Z**.

1. **Data Preparation:** Adjacency **A**, node features **X**.
2. **GAE Training:**
   - Encoder: GCN layers $\rightarrow \mathbf{Z}$.
   - Decoder: Inner product or MLP to reconstruct **A**.
   - Loss: Cross-entropy or MSE on adjacency.
3. **Embedding Extraction:** Take final **Z**.
4. **t-SNE:**
$$\mathbf{Y} = \text{t-SNE}(\mathbf{Z}).$$
5. **Visualization & Evaluation:**
   - Plot **Y** in 2D.
   - Color by labels or cluster IDs (e.g., KMeans).

Evaluation Metrics

- ❖ Graph embedding quality is often task-dependent.
- ❖ Common tasks: node classification, link prediction, clustering.
- ❖ Metrics should measure how well embeddings capture structural/semantic properties.

Node Classification Metrics

Typical Steps:

- ➢ Split nodes into train/test based on known labels.
- ➢ Use embeddings zu as features for a classifier (e.g., logistic regression, SVM).
- ➢ Evaluate accuracy on the held-out test set.

Common Metrics:

- ➢ Accuracy: #correct #total .
- ➢ F1-score: harmonic mean of precision and recall.
- ➢ Macro/Micro-F1: for multi-class settings.

Reconstruction Error:

- ▶ Factorization-based methods can measure how well the adjacency or proximity matrix is reconstructed.
- ▶ For example, $||A - ZZ^T||_F^2$ or variants.

Visualization:

- ▶ Use dimensionality reduction (e.g., t-SNE) on embeddings to see if clusters/communities are visually separable.
- ▶ Qualitative check: do similar nodes appear close in 2D/3D projection?

Karate Club Graph

The Karate Club Graph is a well-known dataset in network science, commonly used in community detection, graph clustering, and Graph Neural Networks (GNNs) research.

1. Origin & Background

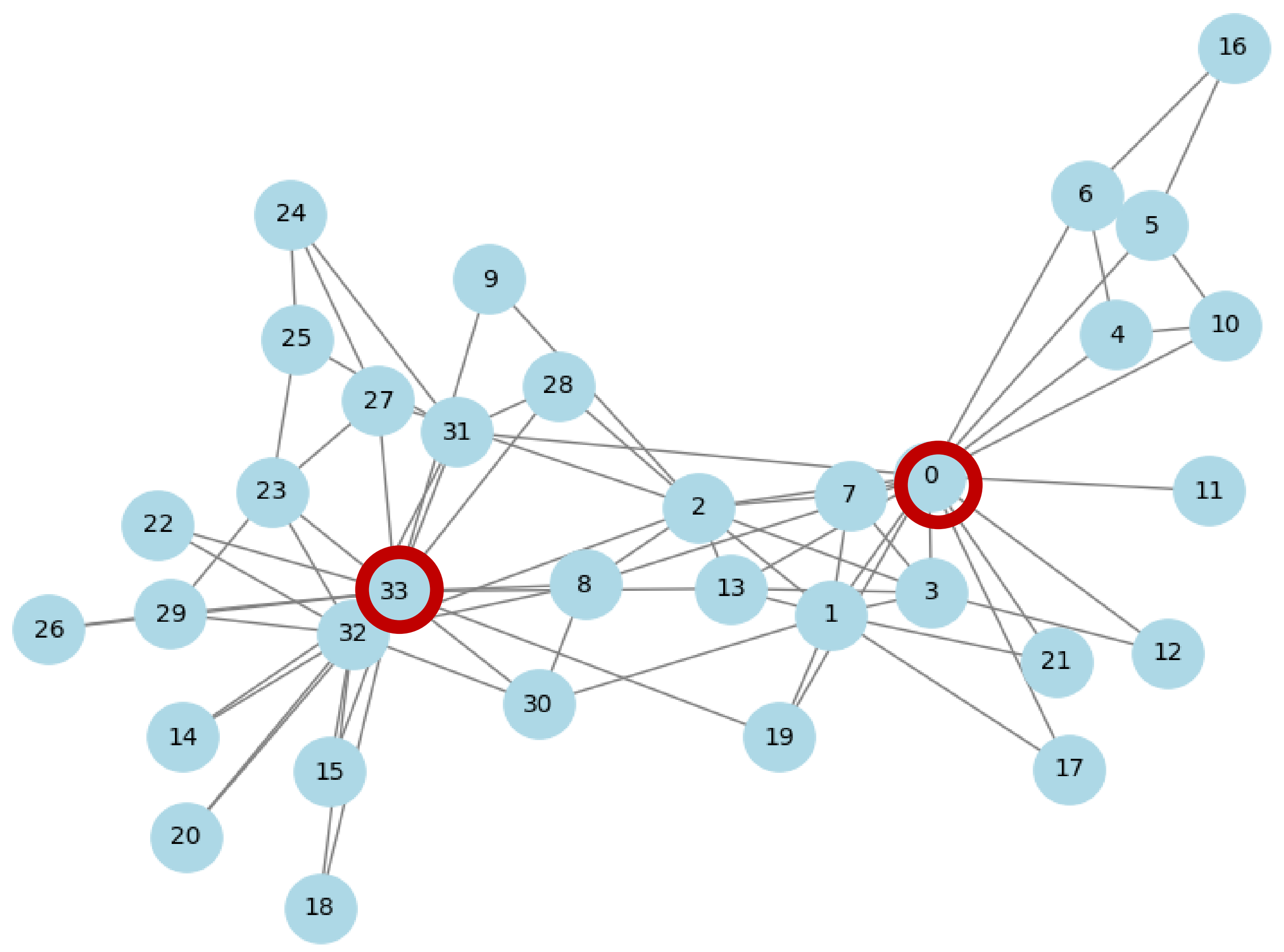
- The dataset was collected by **Wayne W. Zachary** in the 1970s.
- It represents the **social interactions** of **34 members** in a university **karate club** over a period of time.
- Due to internal conflicts, the club eventually **split into two groups**, forming **two communities**.

2. Structure of the Graph

- **Nodes (34):** Each node represents a **club member**.
- **Edges (78):** An edge between two nodes indicates that the corresponding members interacted **outside of the club**.
- **Two Communities:**
  - One group followed the instructor (Leader: Node 0)
  - Another group followed the club administrator (Leader: Node 33)

Feature	Description
Nodes (Members)	34
Edges (Interactions)	78
Communities	2 (Instructor's group vs. Administrator's group)
Real-World Labels	Yes (Ground truth available)
Common Uses	Community detection, Graph Embedding, GNN Research

Karate Club Graph



Four Graph Embedding Methods

1 Laplacian Eigenmaps (Matrix Factorization)

- Uses the **graph Laplacian matrix** for spectral decomposition.
- Captures **global structure**, but is **limited to linear projections**.

2 Random Walk-based Embeddings (DeepWalk-like)

- Simulates **random walks** on the graph.
- Uses **t-SNE** for dimensionality reduction.
- Inspired by **word embeddings** in NLP.

3 GNN-Based Embeddings (Graph Convolutional Networks**)**

- Uses **message passing** to learn node representations.
- Captures both **local and global** structure.

4 Deep Learning-Based (Graph Autoencoder-like using t-SNE)

- Further refines GNN embeddings for **better visualization**.
- Ensures **local neighborhood relationships** are well-preserved.

Sample Code

```
# Load the Karate Club Graph
G = nx.karate_club_graph()
# Convert NetworkX graph to PyTorch Geometric format
data = from_networkx(G)
# Define a simple Graph Neural Network (GNN) for learning
embeddings
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, output_dim)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return x
# Initialize random node features (no predefined features)
num_nodes = G.number_of_nodes()
input_dim = 5
hidden_dim = 8
output_dim = 2
x = torch.randn((num_nodes, input_dim))
# Initialize the GNN model
model = GCN(input_dim, hidden_dim, output_dim)
```

```
# 0. Forward pass to obtain node embeddings
node_embeddings = model(x, data.edge_index).detach().numpy()
# 1. Matrix Factorization (Laplacian Eigenmaps)
laplacian = nx.normalized_laplacian_matrix(G).toarray()
eigvals, eigvecs = np.linalg.eigh(laplacian)
laplacian_embeddings = eigvecs[:, 1:3]
# 2. Random Walk-Based (DeepWalk-like embeddings using t-SNE)
random_walk_embeddings =
TSNE(n_components=2).fit_transform(laplacian_embeddings)
# 3. Deep Learning-Based (Graph Autoencoder-like using t-SNE)
deep_learning_embeddings = TSNE(n_components=2,
perplexity=5).fit_transform(node_embeddings)

# Apply KMeans clustering to each embedding method
num_clusters = 2
kmeans_laplacian = KMeans(n_clusters=num_clusters,
                    random_state=42).fit(laplacian_embeddings)
kmeans_random_walk = KMeans(n_clusters=num_clusters,
                   random_state=42).fit(random_walk_embeddings)
kmeans_deep_learning = KMeans(n_clusters=num_clusters,
                  random_state=42).fit(deep_learning_embeddings)
kmeans_gnn = KMeans(n_clusters=num_clusters,
                        random_state=42).fit(node_embeddings)
```

File: graphEmbedding.ipynb

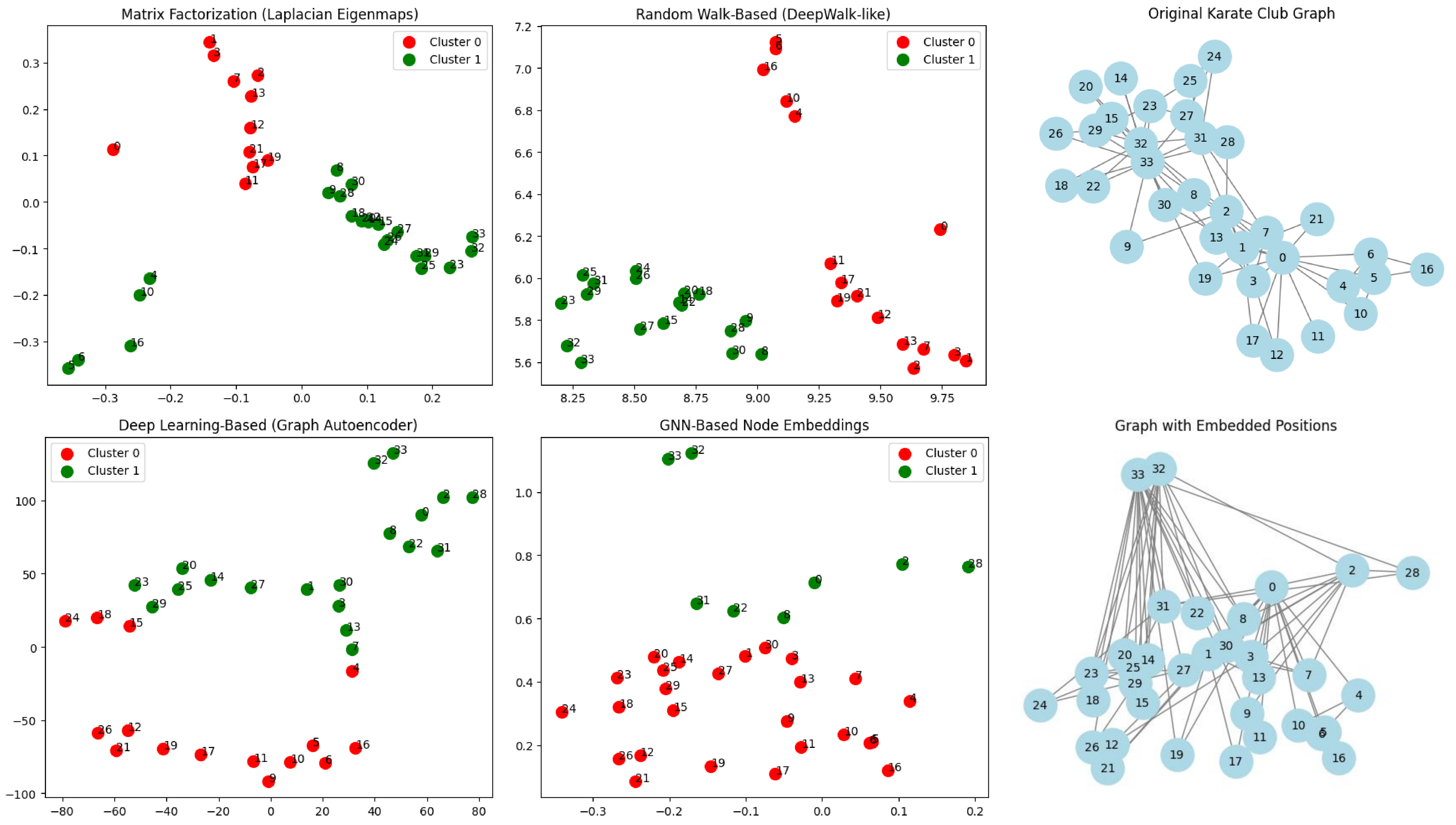
Clustering & Evaluation

- **Applying K-Means Clustering:** We cluster the node embeddings into two communities.
- **How Do We Evaluate Clustering?**

✓ Silhouette Score measures clustering quality.
✓ Higher score = better-defined clusters.

Method	Silhouette Score
Laplacian Eigenmaps	**0.4252**
Random Walk (DeepWalk-like)	**0.4867**
Graph Autoencoder (Deep Learning)	**0.4408**
GNN-Based Node Embeddings	**0.4810**

Figures



Matrix Factorization (Laplacian Eigenmaps)
Random Walk-Based (DeepWalk-like)
Original Karate Club Graph
Deep Learning-Based (Graph Autoencoder)
GNN-Based Node Embeddings
Graph with Embedded Positions
Cluster 0
Cluster 1

Enhancement

Change	**Impact**
Increased node feature dimension **(5 → 10)**	Allows GNN to learn richer representations.
Increased GNN hidden layer size **(8 → 16)**	Improves model capacity for **better feature extraction**.
Adjusted t-SNE perplexity=5 for GNN embeddings	Focuses on **local neighborhood structures**, leading to more refined visualization.

Code for Enhancement

```
# Load the Karate Club Graph
G = nx.karate_club_graph()
# Convert NetworkX graph to PyTorch Geometric format
data = from_networkx(G)
# Define a simple Graph Neural Network (GNN) for learning
embeddings
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, output_dim)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return x
# Initialize random node features (Karate Club has no
predefined features)
num_nodes = G.number_of_nodes()
input_dim = 10 # Number of features per node
hidden_dim = 16
output_dim = 2 # 2D embedding for visualization
x = torch.randn((num_nodes, input_dim))
# Initialize the GNN model
model = GCN(input_dim, hidden_dim, output_dim)
```

```
# 0. Forward pass to obtain node embeddings
node_embeddings = model(x, data.edge_index).detach().numpy()
# 1. Matrix Factorization (Laplacian Eigenmaps)
laplacian = nx.normalized_laplacian_matrix(G).toarray()
eigvals, eigvecs = np.linalg.eigh(laplacian)
laplacian_embeddings = eigvecs[:, 1:3]
# 2. Random Walk-Based (DeepWalk-like embeddings using t-SNE)
random_walk_embeddings =
TSNE(n_components=2).fit_transform(laplacian_embeddings)
# 3. Deep Learning-Based (Graph Autoencoder-like using t-SNE)
deep_learning_embeddings = TSNE(n_components=2,
perplexity=5).fit_transform(node_embeddings)

# Apply KMeans clustering to each embedding method
num_clusters = 2
kmeans_laplacian = KMeans(n_clusters=num_clusters,
                          random_state=42).fit(laplacian_embeddings)
kmeans_random_walk = KMeans(n_clusters=num_clusters,
                         random_state=42).fit(random_walk_embeddings)
kmeans_deep_learning = KMeans(n_clusters=num_clusters,
                        random_state=42).fit(deep_learning_embeddings)
kmeans_gnn = KMeans(n_clusters=num_clusters,
                            random_state=42).fit(node_embeddings)
```

File: graphEmbedding.ipynb

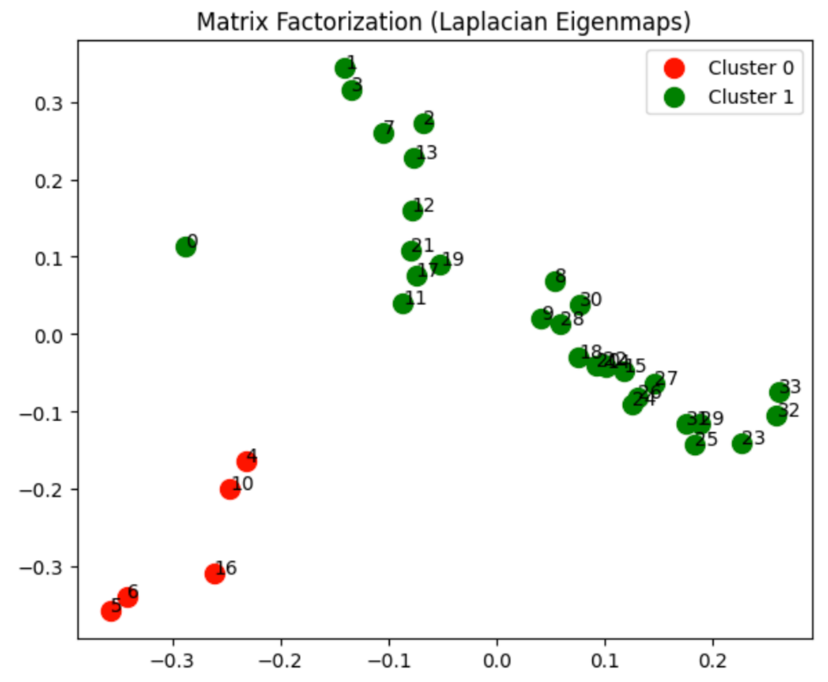
Clustering & Evaluation

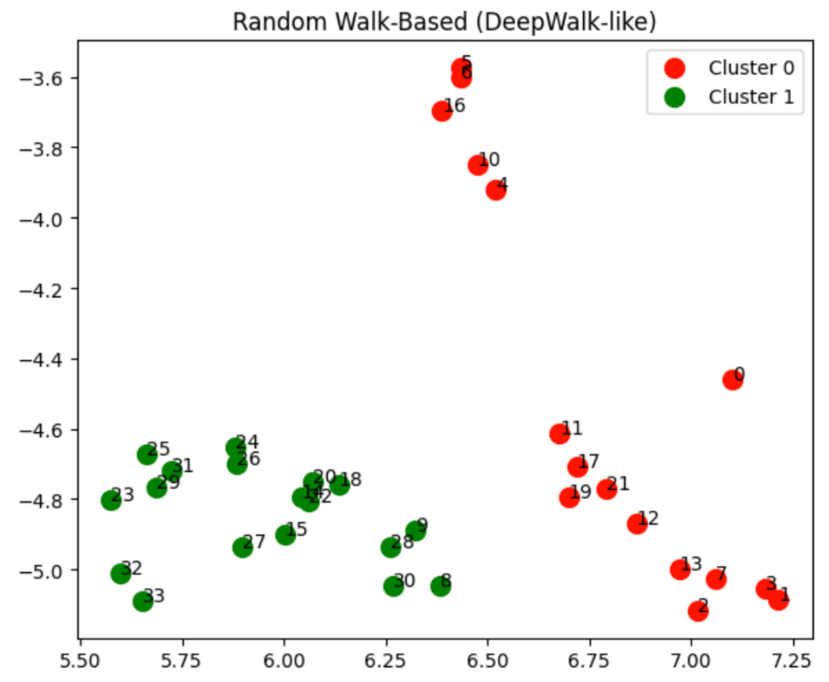
- **Applying K-Means Clustering:** We cluster the node embeddings into two communities.
- **How Do We Evaluate Clustering?**

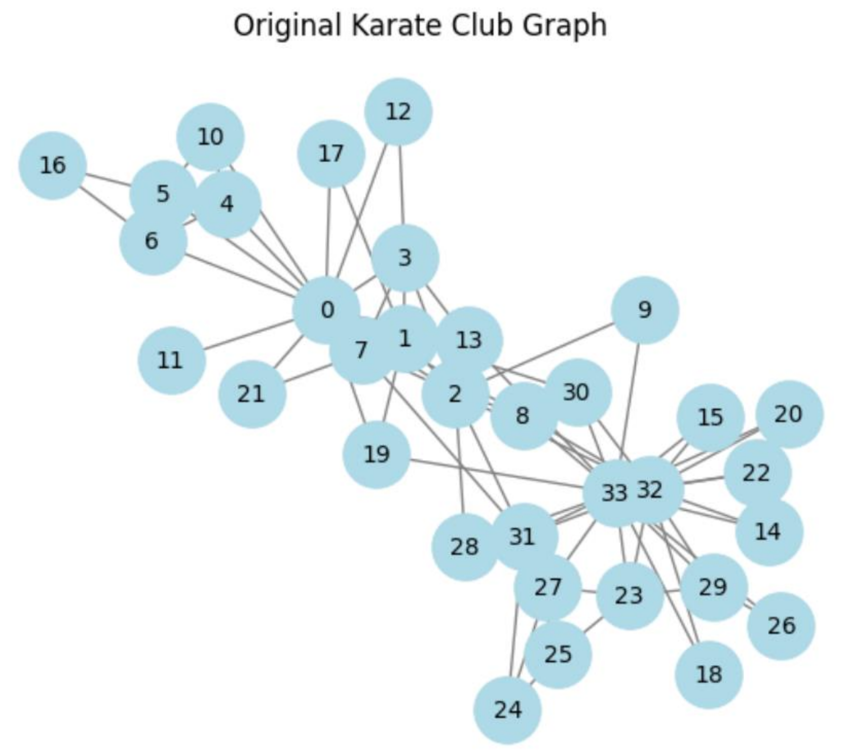
  ✓ Silhouette Score measures clustering quality.
  ✓ Higher score = better-defined clusters.

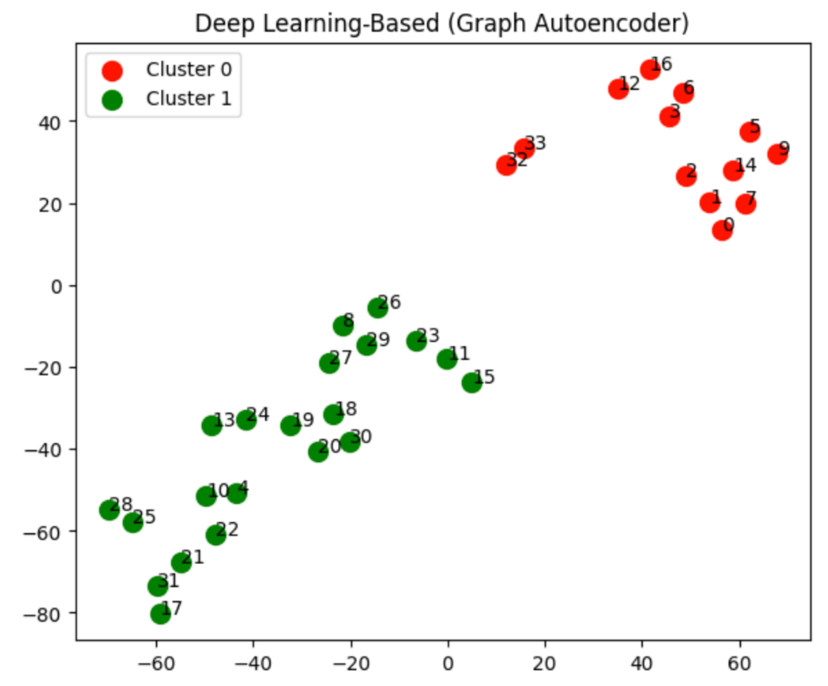
Method	Silhouette Score
Laplacian Eigenmaps	**0.5525**
Random Walk (DeepWalk-like)	**0.4889**
Graph Autoencoder (Deep Learning)	**0.6777**
GNN-Based Node Embeddings	**0.3688**

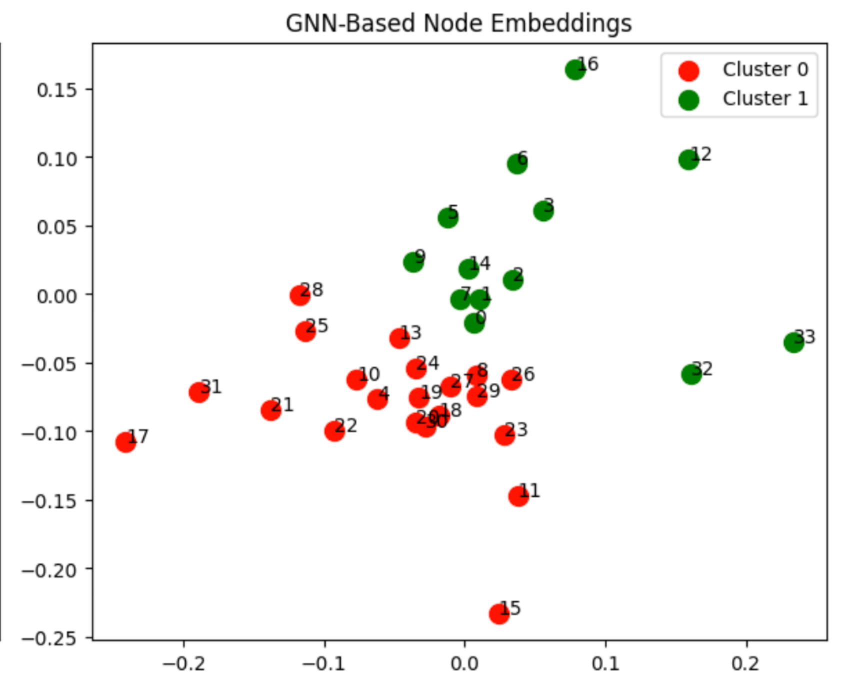
Figures

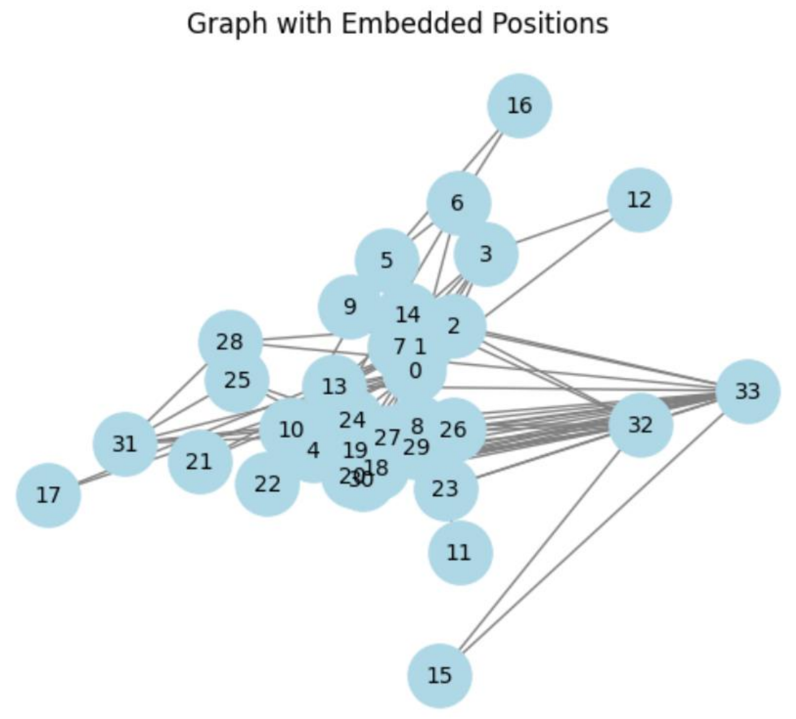

Matrix Factorization (Laplacian Eigenmaps)
Cluster 0
Cluster 1


Random Walk-Based (DeepWalk-like)
Cluster 0
Cluster 1


Original Karate Club Graph

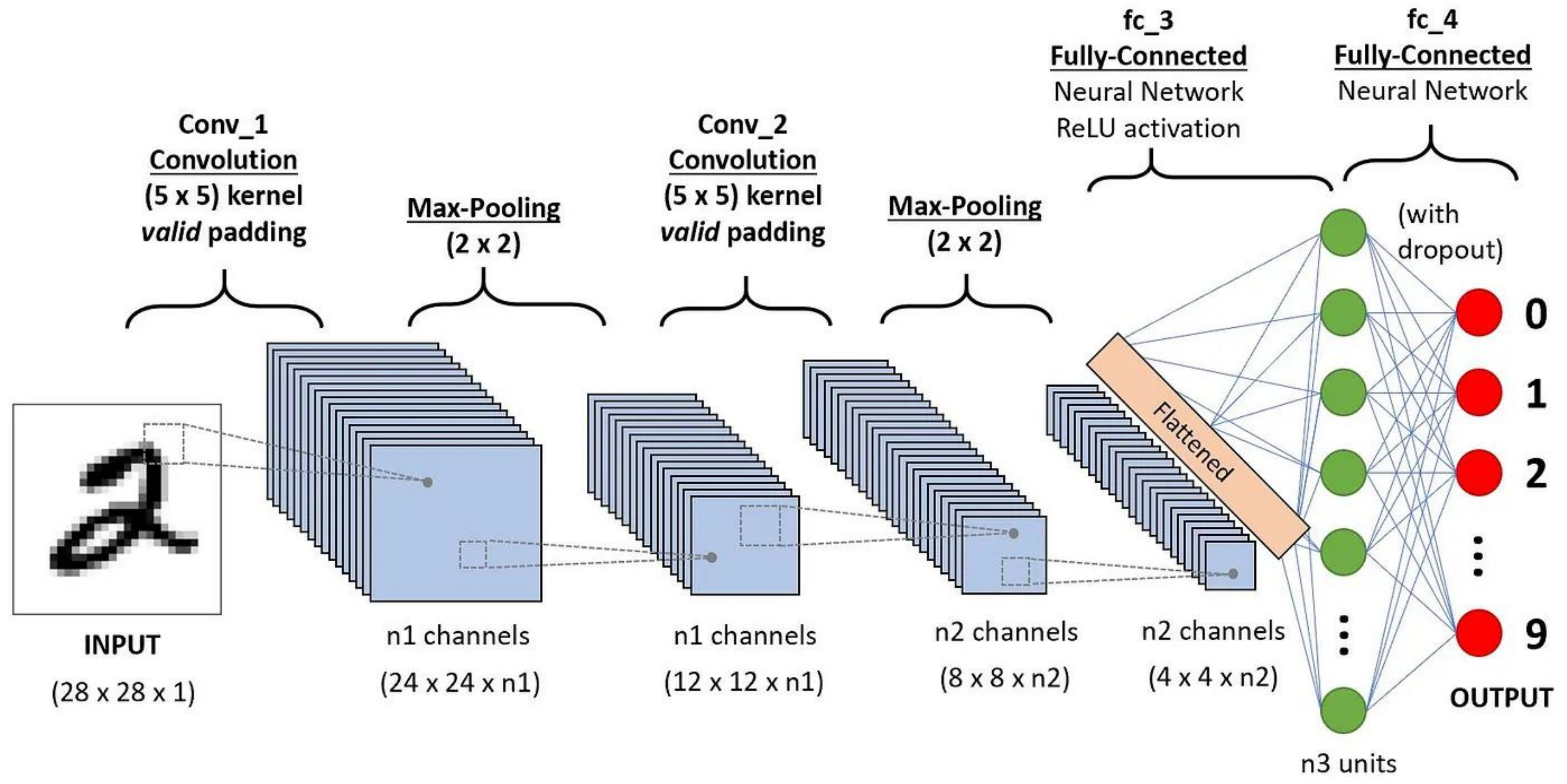

Deep Learning-Based (Graph Autoencoder)
Cluster 0
Cluster 1


GNN-Based Node Embeddings
Cluster 0
Cluster 1

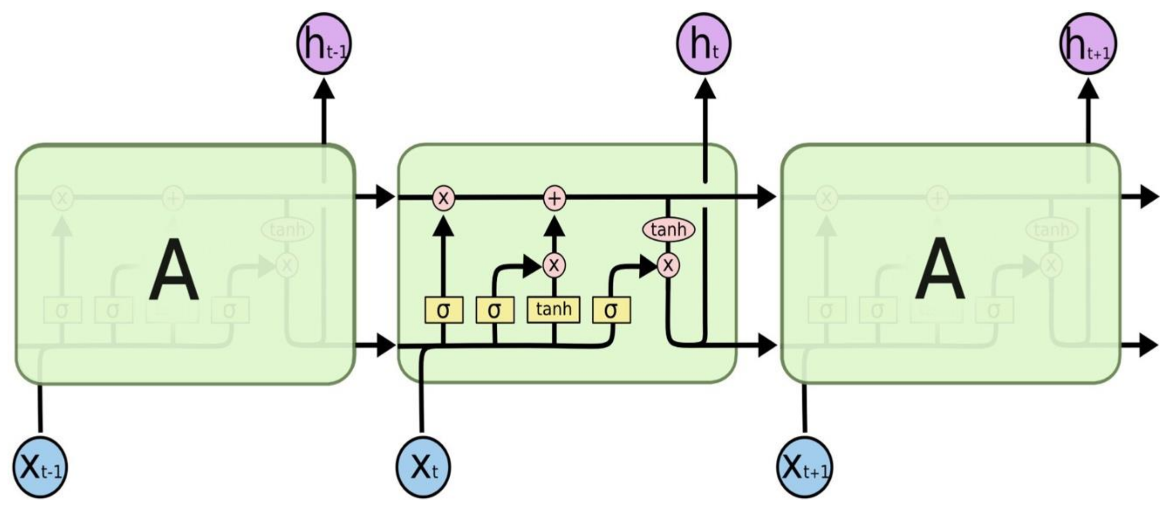

Graph with Embedded Positions

Content

CNN and RNN



Obama *"I stand here today humbled by the task before us, grateful for the trust you have bestowed, mindful of the sacrifices borne by our ancestors. I thank President Bush for his service to our nation, as well as the generosity and cooperation he has shown throughout this transition.*



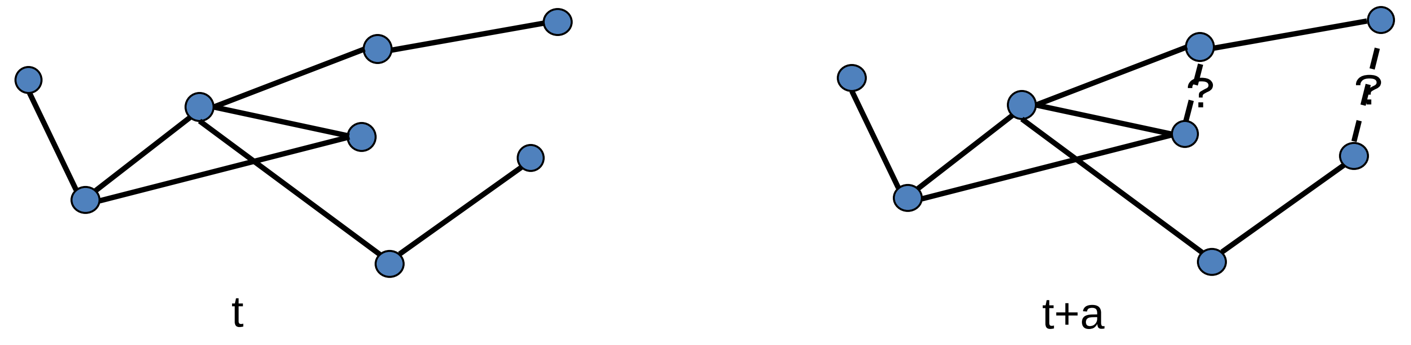
CNN/RNN Key Modules

Modules	CNN	RNN
Core Concept	Spatial feature extraction	Sequential dependency modeling
Kernel (Filters)	Yes (learnable filters)	No kernels, uses weight matrices
Pooling (Downsampling)	Yes	No
Multi-Scale Processing	Yes (hierarchical)	No
Weight Sharing	Yes	Partially (shared weights over time)
Handles Variable Input Length	No	Yes
Long-Term Dependencies	No	Yes (with LSTMs/GRUs)

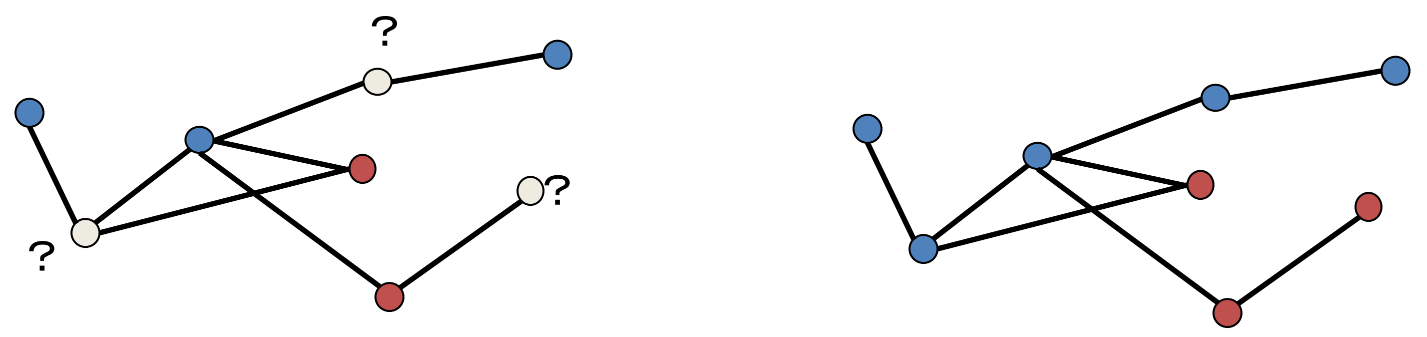
Node- and Graph-level Tasks

Node-level

Link Prediction

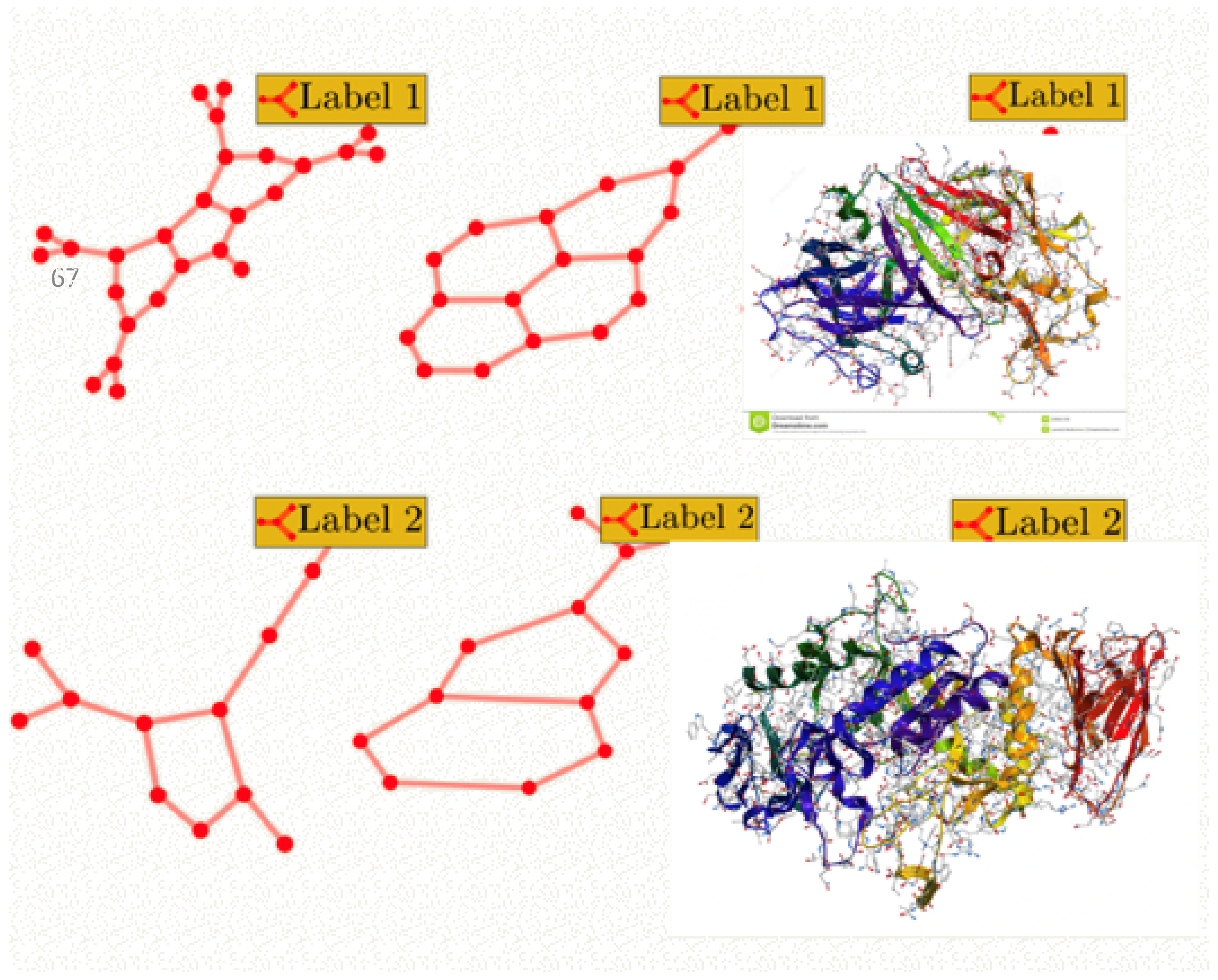


Node Classification

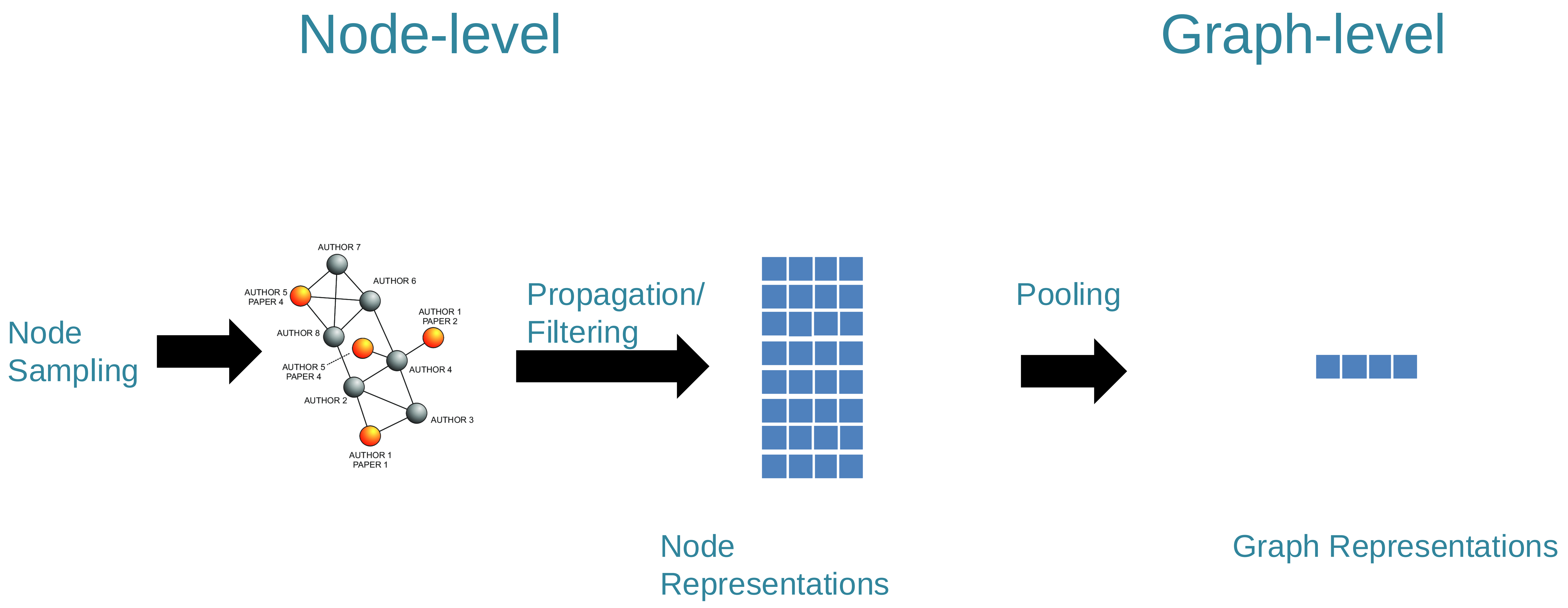


Graph-level

Graph Classification

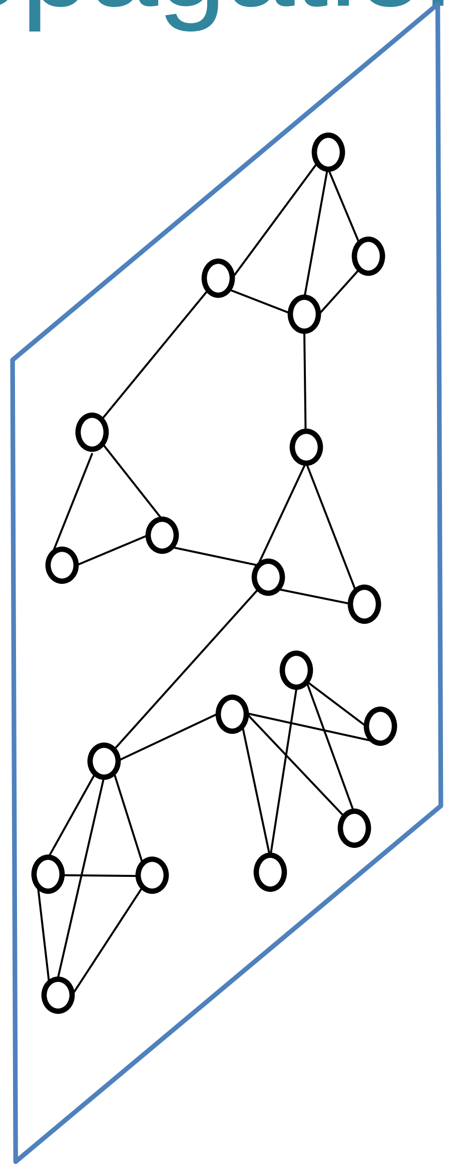


Propagation and Pooling



Group Propagation/Filtering

Graph Propagation



Node Representation Update:

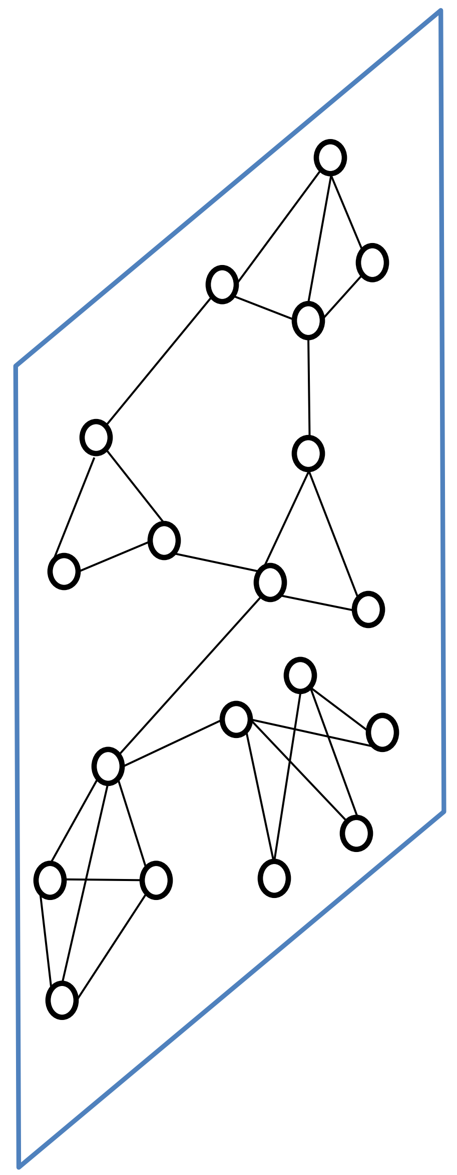
$$h_i^{(l+1)} = f\left(h_i^{(l)}, \sum_{j \in N(i)} g(i,j)\right),$$

where:

- $h_i^{(l)}$ represents the embedding of node i at layer l.
- $N(i)$ denotes the set of neighbors of node i.
- f and g are learnable functions.

Propagation

❖ A node's representation is influenced by its neighbors.
❖ By stacking multiple GNN layers, each node can capture information from multi-hop neighbors.



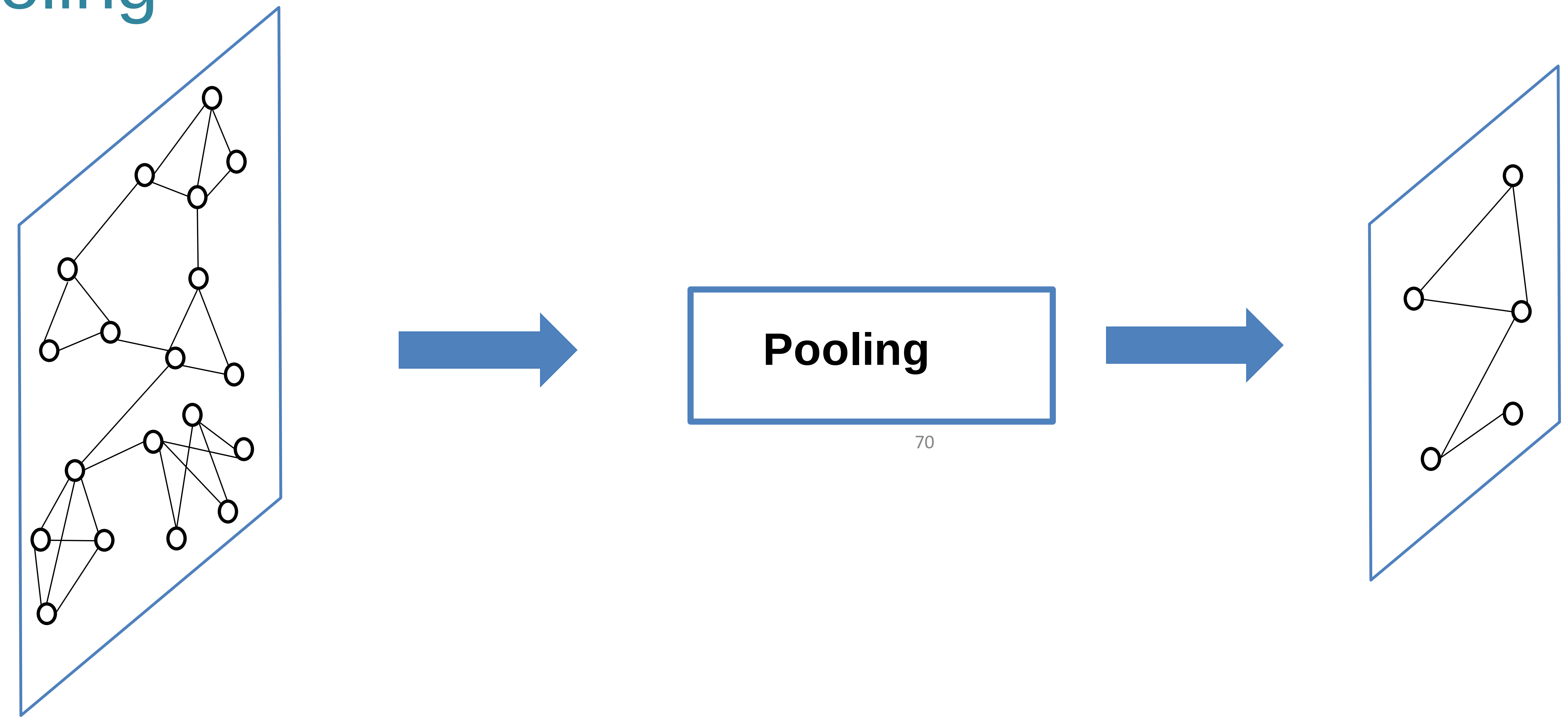
$\mathbf{A} \in \{0,1\}^{n \times n}, \mathbf{X} \in \mathbb{R}^{n \times d}$

$\mathbf{A} \in \{0,1\}^{n \times n}, \mathbf{X}_f \in \mathbb{R}^{n \times d_{new}}$

Propagation refines the node features

Group Pooling

Graph Pooling



$\mathbf{A} \in \{0,1\}^{n \times n}, \mathbf{X} \in \mathbb{R}^{n \times d}$

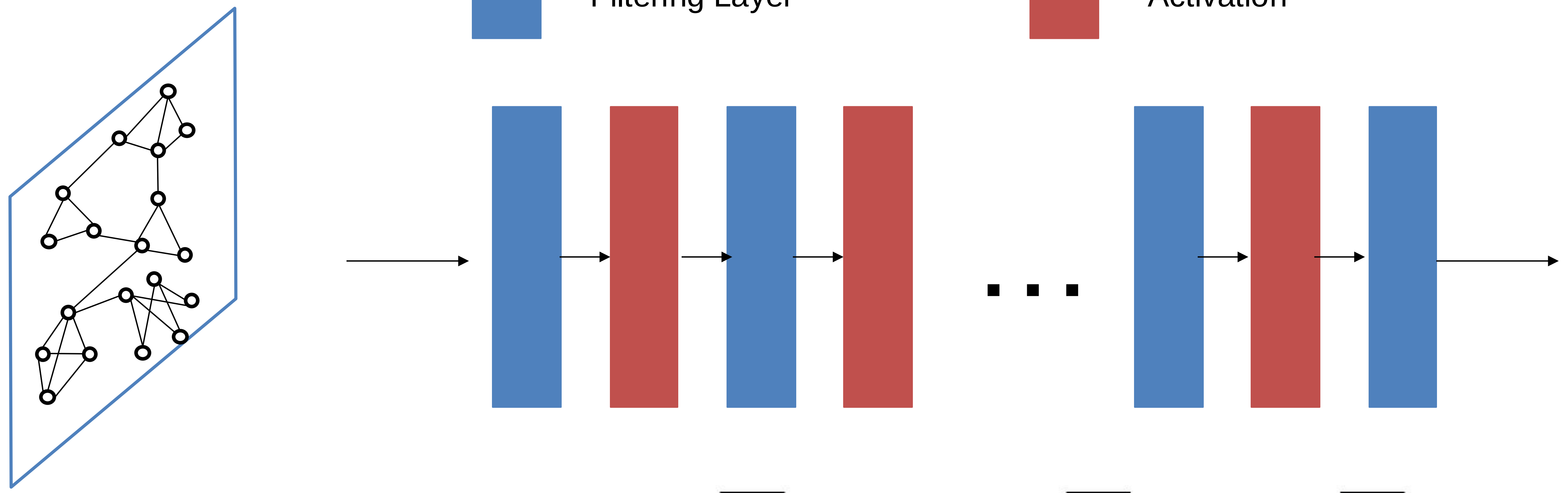
$\underline{\mathbf{A}_p \in \{0,1\}^{n_p \times n_p}, \mathbf{X}_p \in \mathbb{R}^{n_p \times d_{new}}, n_p < n}$

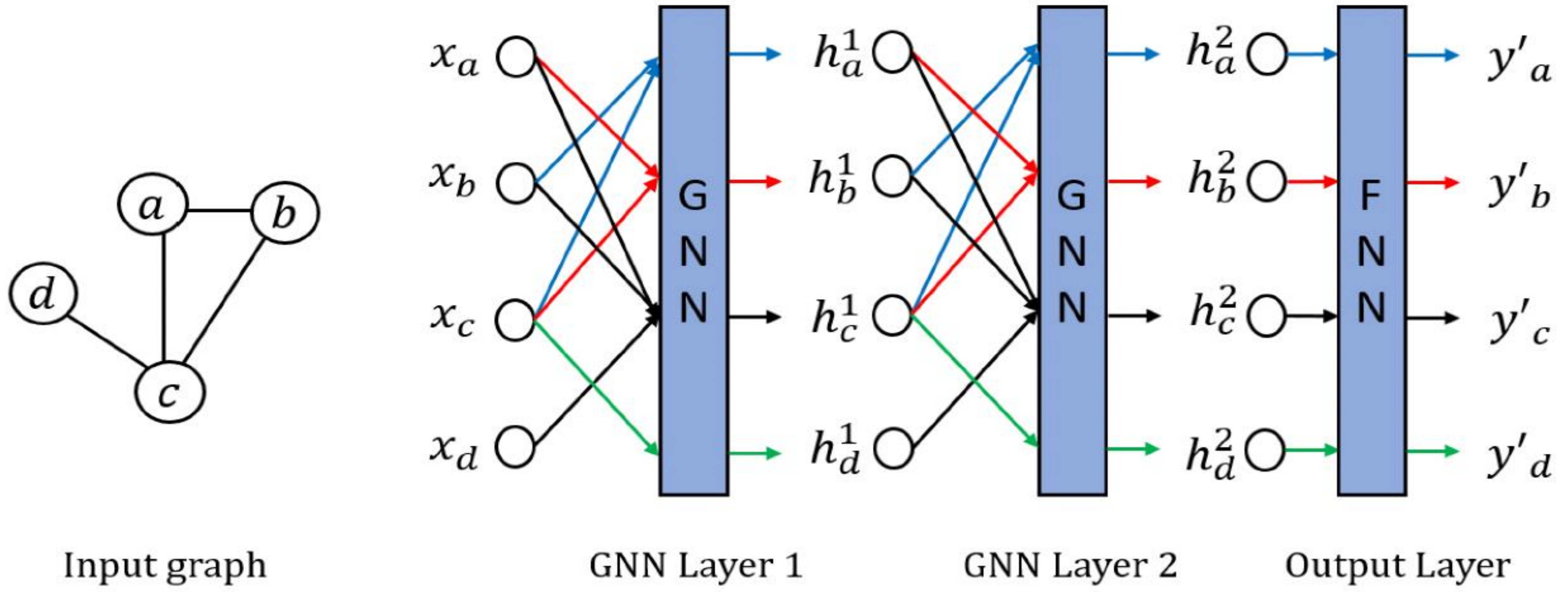
Graph pooling generates a smaller graph

General GNN Framework

For node-level tasks

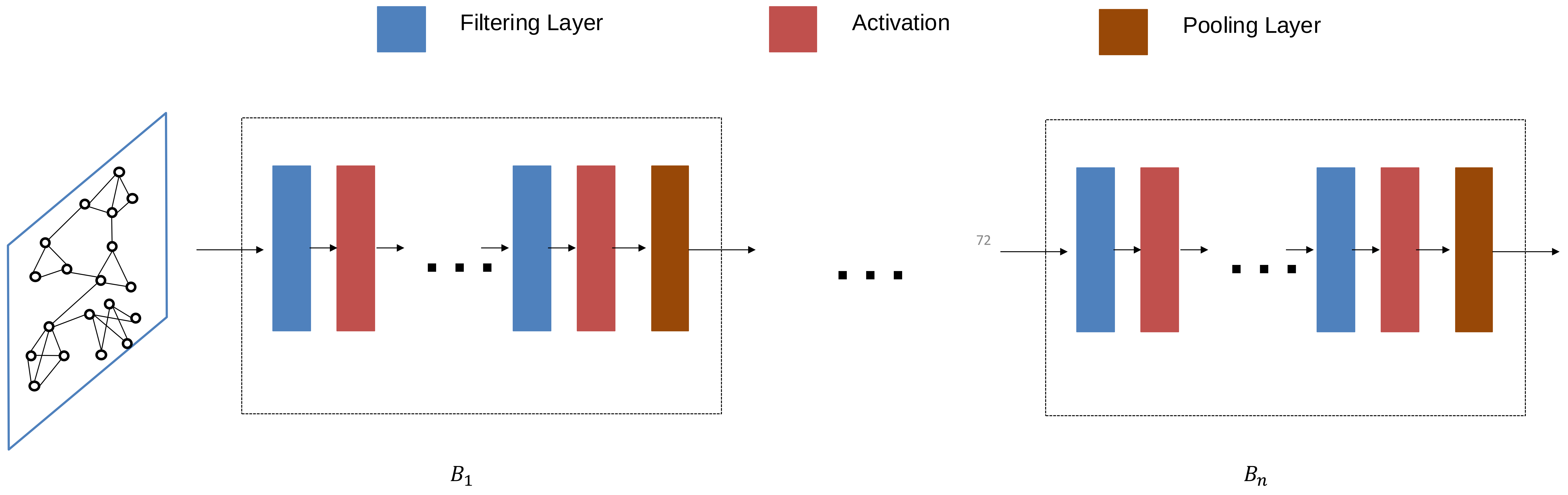
🟦 Filtering Layer 🟥 Activation



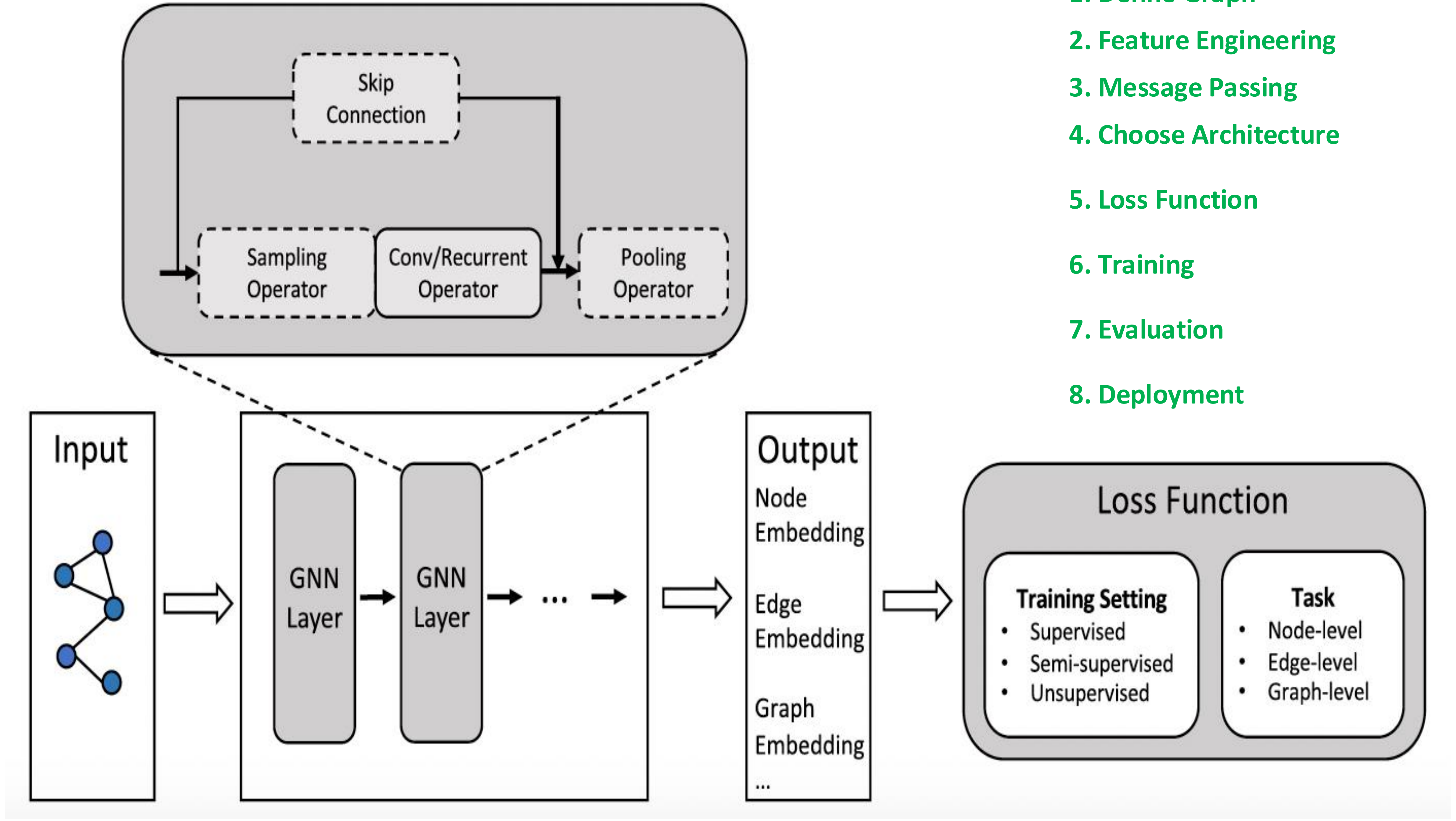


General GNN Framework

For graph-level tasks



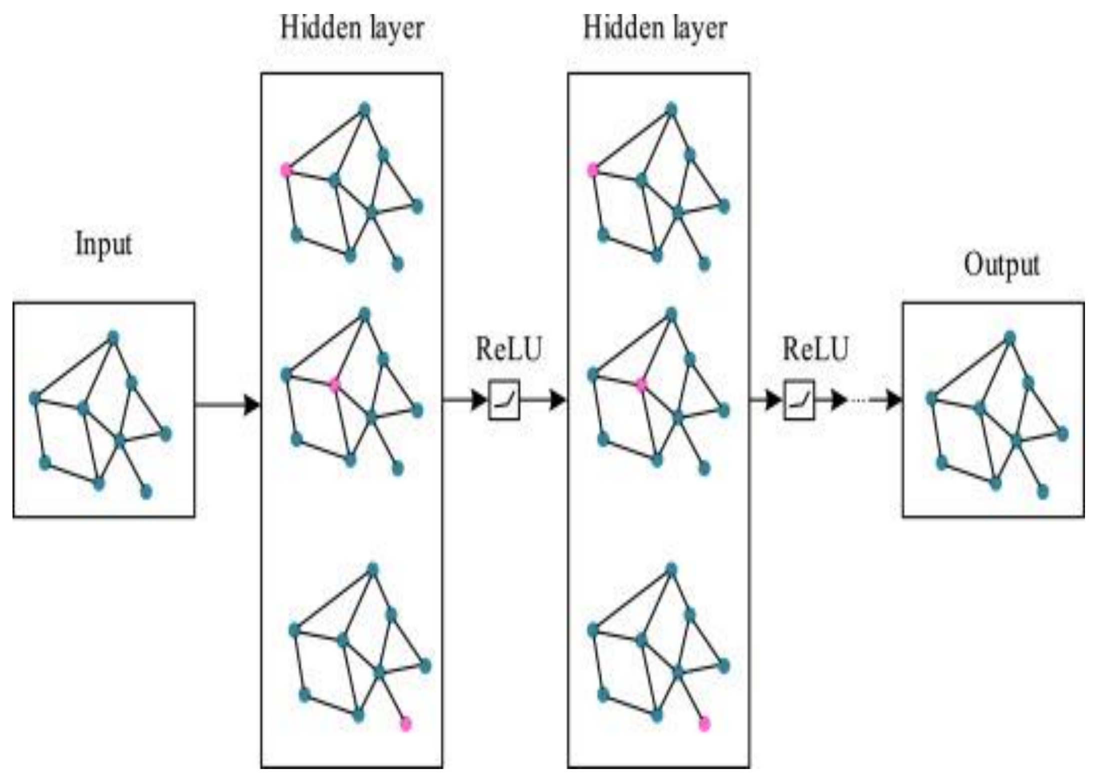
GNN Designs



Jie Zhou, et al. (2020). AI Open

Summary

Step	Task
1. Define Graph	Nodes, edges, features
2. Feature Engineering	Define node and edge features
3. Message Passing	Select aggregation method
4. Choose Architecture	GCN, GAT, GraphSAGE, etc.
5. Loss Function	Supervised (cross-entropy), unsupervised (contrastive)
6. Training	Use mini-batching and optimizers
7. Evaluation	Classification, link prediction, graph-level tasks
8. Deployment	Optimize for inference speed



Key Modules in Graph Neural Networks

GNNs process graph-structured data by propagating and aggregating information across nodes and edges.

Three key modules in GNNs:

❖ **Sampling Module**

Aims to **reduce the size** of each node's neighborhood, especially for **large graphs**, preventing the neighbor explosion problem.

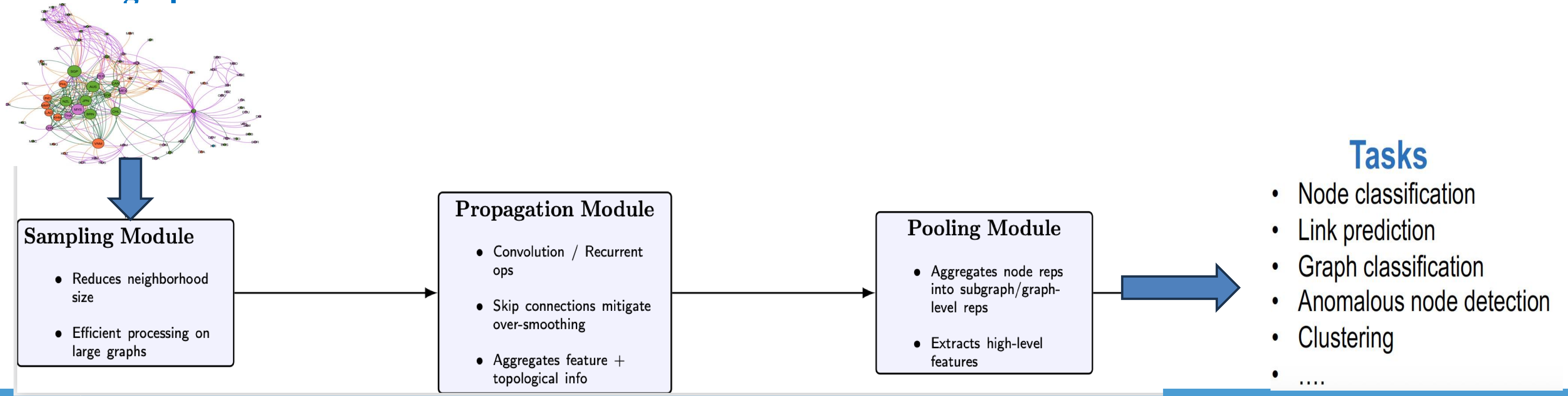
❖ **Propagation Module**

➢ Performs **message passing** via **convolutions** (e.g., GCNs) or **recurrent operators** (e.g., GRUs) on node features.

➢ Uses **skip connections** to mitigate over-smoothing and incorporate historical representations.

❖ **Pooling Module**

Aggregates node-level embeddings into **subgraph** or **graph-level** representations, extracting higher-level features needed for tasks like **graph classification**.



Sampling Module
Reduces neighborhood size
Efficient processing on large graphs
Propagation Module
Convolution / Recurrent ops
Skip connections mitigate over-smoothing
Aggregates feature + topological info
Pooling Module
Aggregates node reps into subgraph/graph-level reps
Extracts high-level features
Tasks
Node classification
Link prediction
Graph classification
Anomalous node detection
Clustering
....

The Sampling Module

Efficient Graph Processing via Sampling

- ▶ Direct propagation on large graphs is computationally infeasible.
- ▶ The Sampling Module reduces cost by selecting subsets of nodes or edges.

Key Challenge:

- ❖ **Neighbor Explosion:** The number of neighbors grows exponentially with depth. GNNs aggregate messages from each node's neighbors in the previous layer. Tracking back multiple layers can exponentially increase the neighbor set. Storing and processing all neighborhood information becomes intractable for large graphs.
- ❖ **Computational Efficiency:** Full neighbor aggregation is impractical for large graphs.
- ❖ **Memory Constraints:** Storing all neighborhood information for each node is infeasible.
- ❖ **Scalability:** Enables GNNs to handle large graphs effectively.

Common sampling techniques: Node Sampling; Layer Sampling; Subgraph Sampling.

Impact on Permutation Properties:

- ▶ Node-level predictions remain unchanged under node reordering.
- ▶ Node representations transform consistently when input ordering changes.

Impact on Task Performance:

- ▶ Preserve downstream performance in classification, link prediction, etc.
- ▶ Sampling strategies must capture essential structural information despite reduced neighborhood size.
- ▶ Aim for low variance while avoiding high computational costs.

Common Sampling Methods

Node Sampling: Selects a subset of nodes and their immediate neighbors.

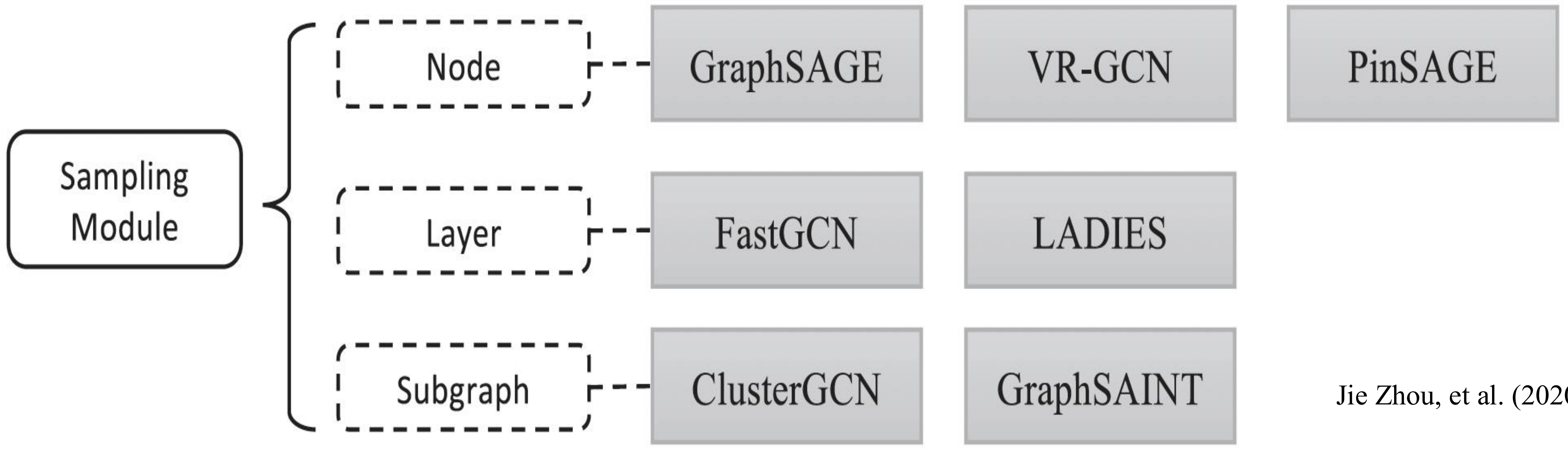
- Reduces computational complexity by limiting the number of participating nodes.
- Often used in algorithms like GraphSAGE.

Layer Sampling: It selects a fixed number of neighbors per layer.

- Controls exponential growth by restricting the number of aggregated neighbors.
- Balances efficiency and performance in large-scale graphs.

Subgraph Sampling: Extracts a subgraph based on connectivity patterns.

- Useful for mini-batch training by working on graph partitions.
- Preserves graph topology while reducing computation.



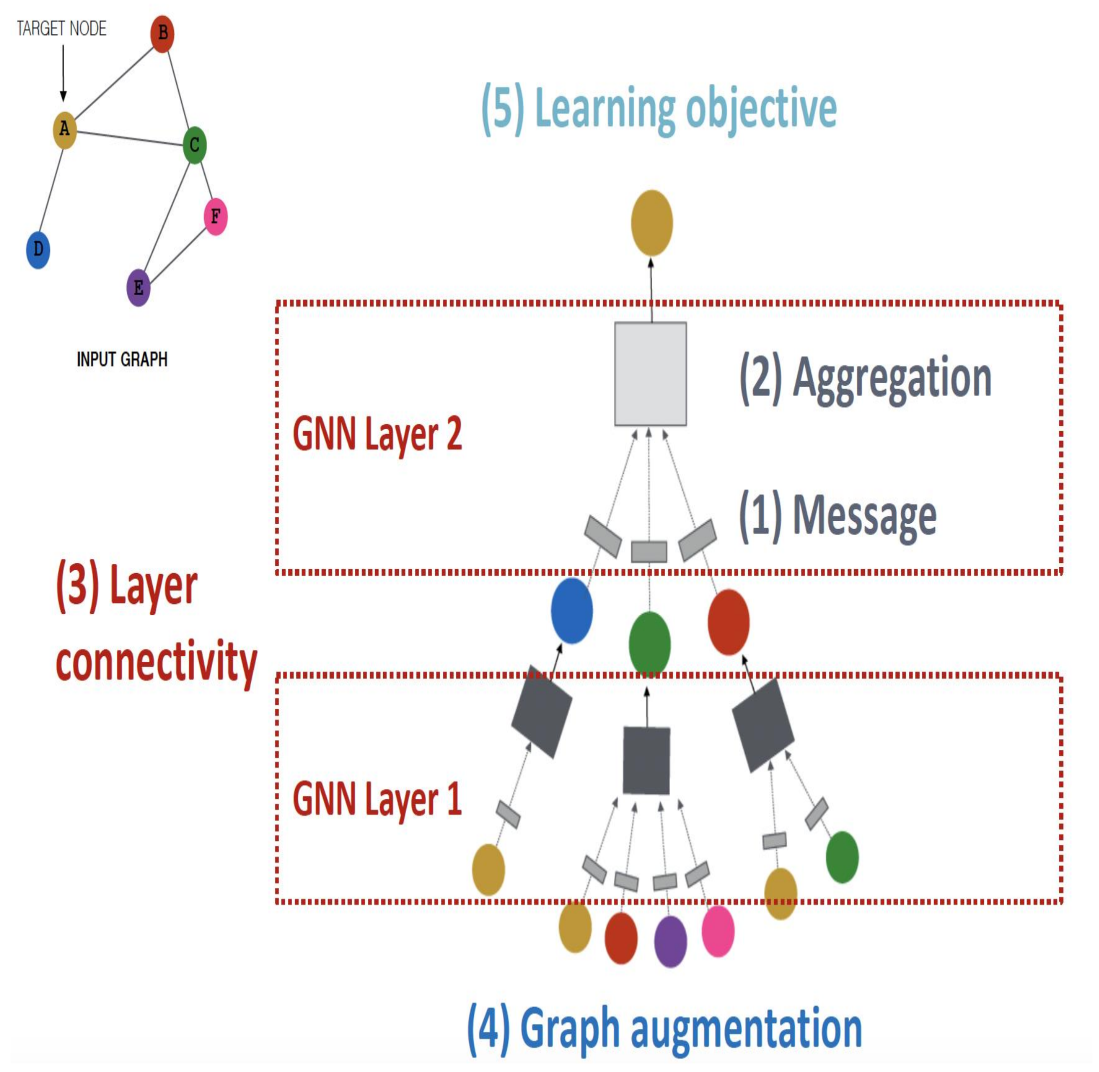
Jie Zhou, et al. (2020). AI Open

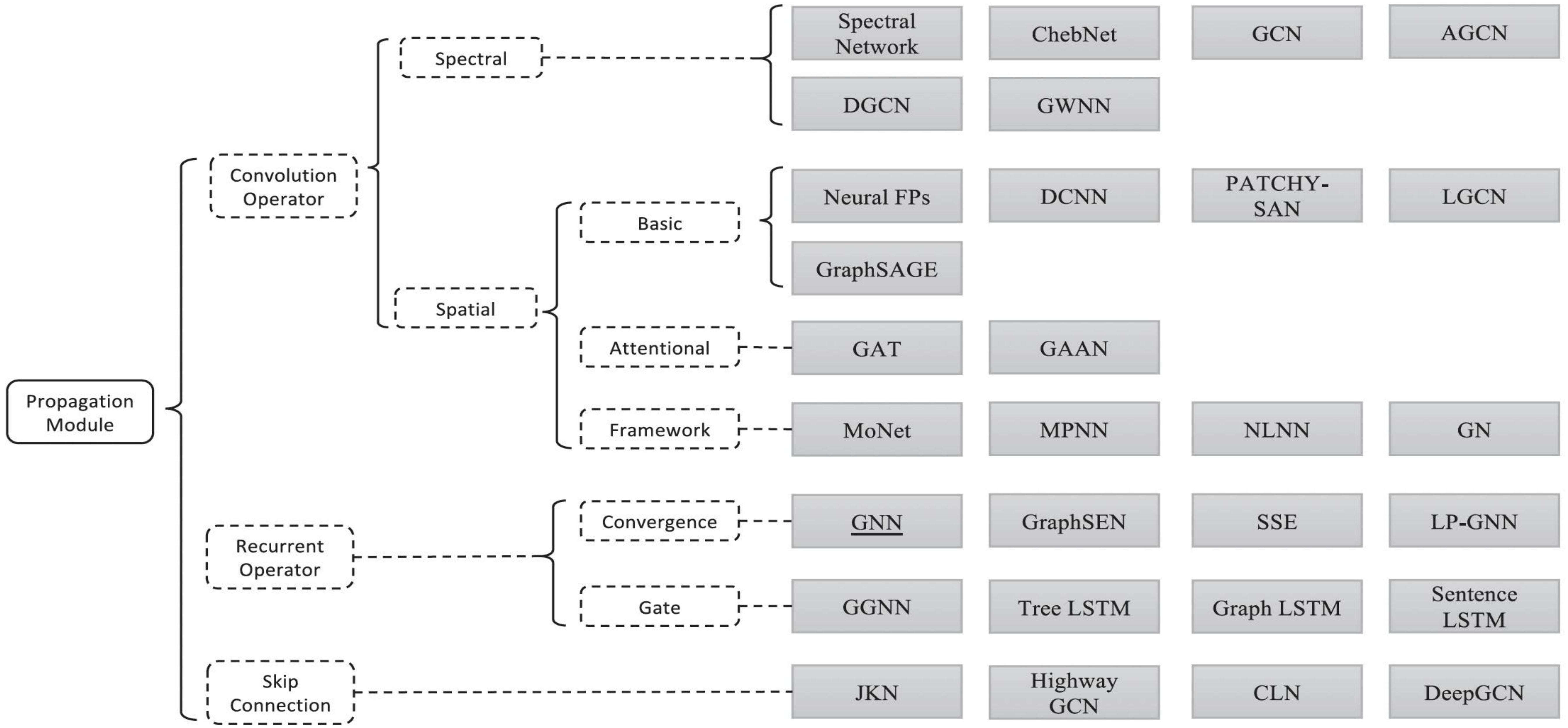
The Propagation Module

Facilitates message passing between nodes to integrate structural and feature information.

Key operations:

- ❖ **Convolution Operators:** Aggregate neighbor information.
- ❖ **Recurrent Operators:** Maintain temporal dependencies in dynamic graphs (e.g., Graph GRU, Graph LSTM).
- ❖ **Skip Connections:** Mitigate over-smoothing by retaining historical representations.





Jie Zhou, et al. (2020). AI Open

Permutation Equivariance and Invariance

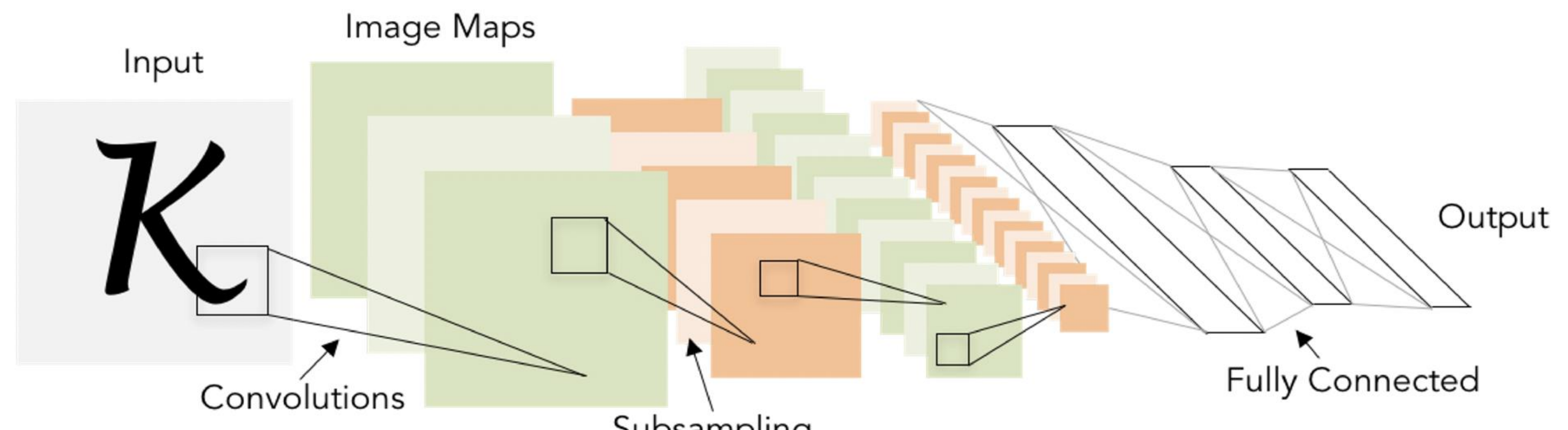
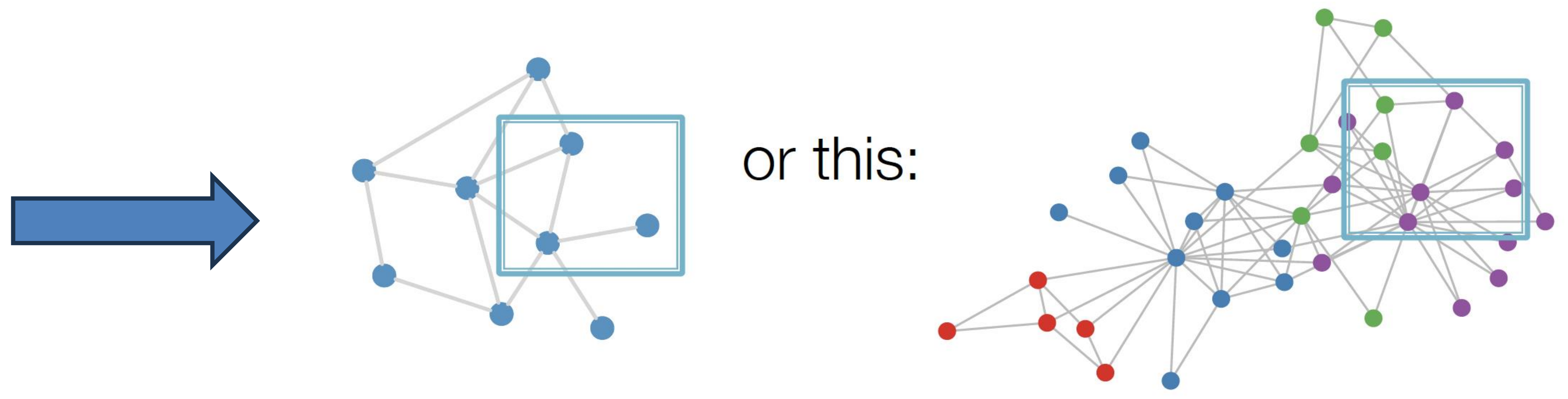


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1



Key Observation:

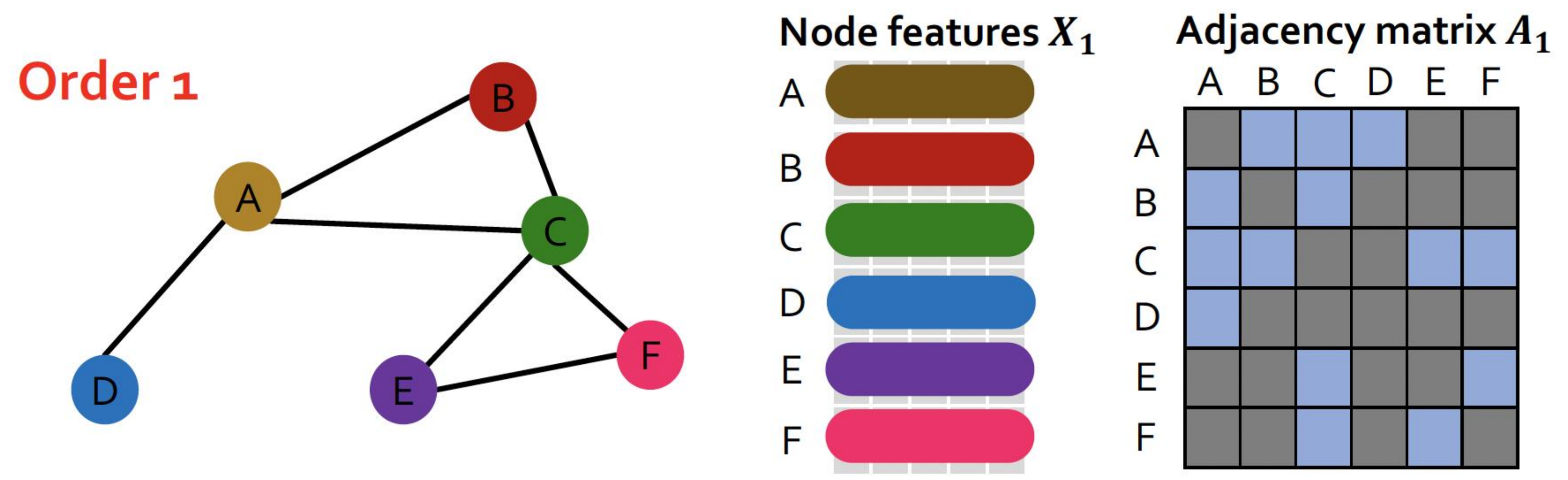
- A graph does not have a fixed, canonical ordering of its nodes.
- Any permutation of node indices can still represent the same underlying graph.

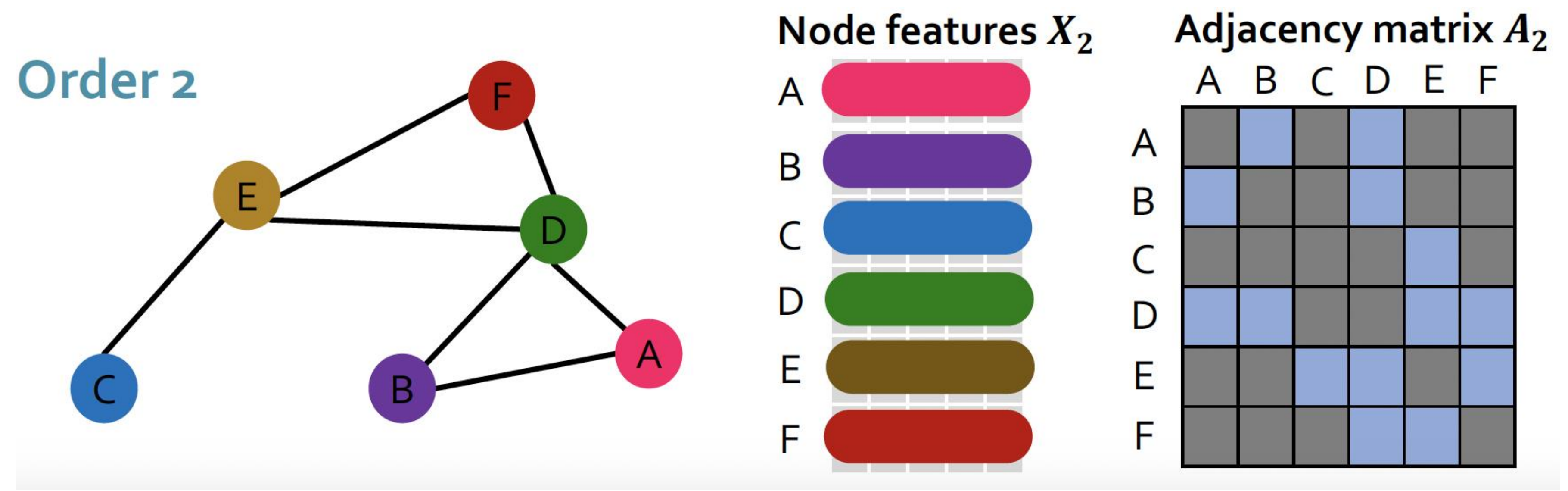
Implication:

- The labeling or numbering of nodes is arbitrary.
- Reorder node IDs without changing the graph's structure.

Permutation Equivariance (PE): Node embeddings maintain structure when node order changes.

Permutation Invariance (PI): Graph-level representations remain
unchanged under different node orderings.





Definition of PE and PI

Permutation Invariance:

- A function $f(\cdot)$ is *invariant* if permuting inputs does not change the output:

$$f(\pi(A, X)) = f(A, X), \quad \forall \pi \in S_n.$$

- Typical for **graph-level** tasks (e.g., entire graph classification).

Permutation on Graphs:

- Let P be an $n \times n$ permutation matrix.
- Then $A \mapsto PAP^\top, \quad X \mapsto PX$.

- $f(A, X) = 1^T X$: Permutation-**invariant**
  - Reason: $f(PAP^T, PX) = 1^T PX = 1^T X = f(A, X)$
- $f(A, X) = AX$: Permutation-**equivariant**
  - Reason: $f(PAP^T, PX) = PAP^T PX = PAX = Pf(A, X)$
- $f(A, X) = X$: Permutation-**equivariant**
  - Reason: $f(PAP^T, PX) = PX = Pf(A, X)$

Permutation Equivariance:

- A function $g(\cdot)$ is *equivariant* if the output is permuted in the same way:

$$g(\pi(A, X)) = \pi(g(A, X)), \quad \forall \pi \in S_n.$$

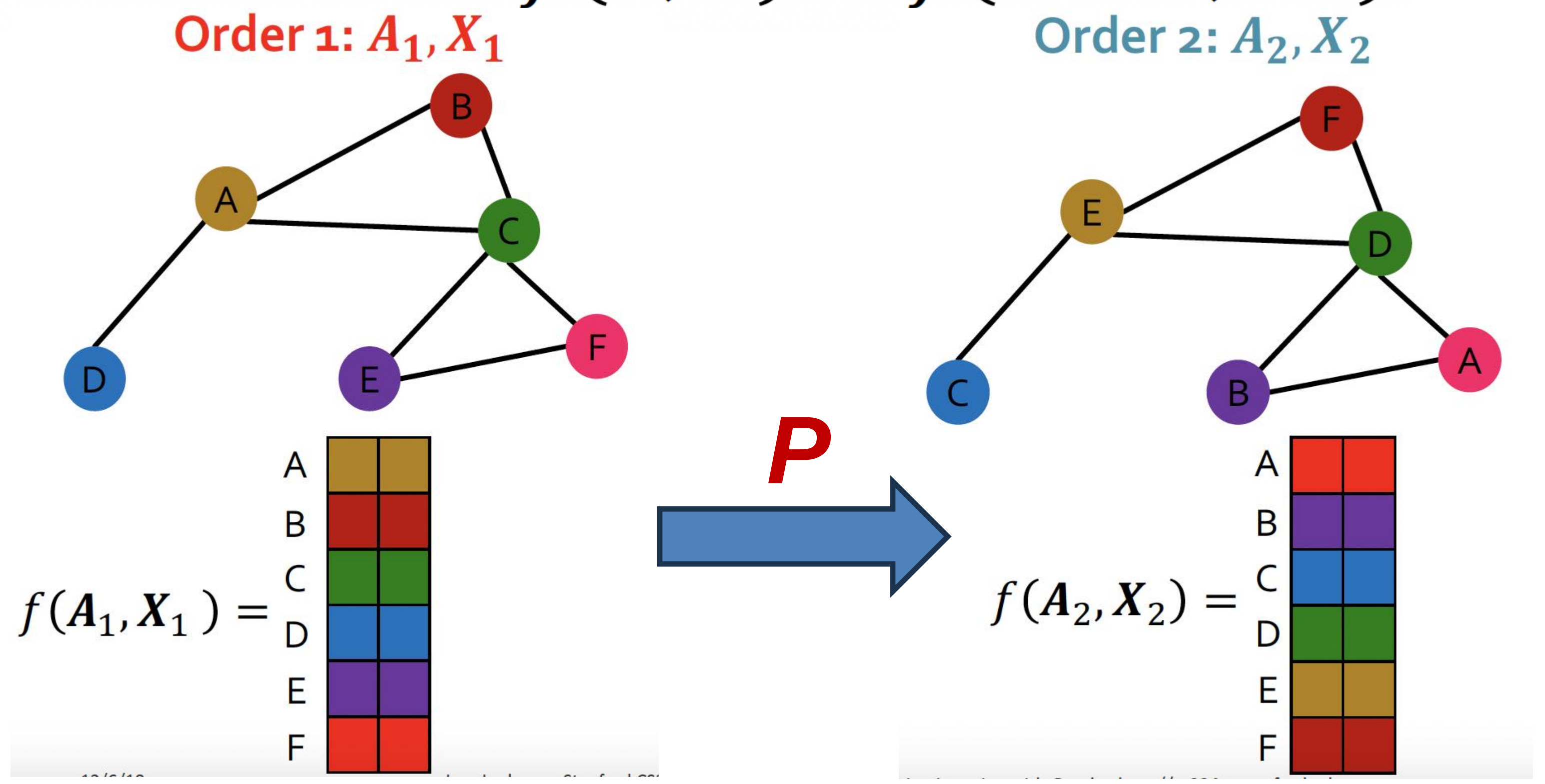
- Typical for **node-level** tasks (e.g., node embeddings, node classification).

- **Permutation-invariant**

$$f(A, X) = f(PAP^T, PX)$$

- **Permutation-equivariant**

$$Pf(A, X) = f(PAP^T, PX)$$

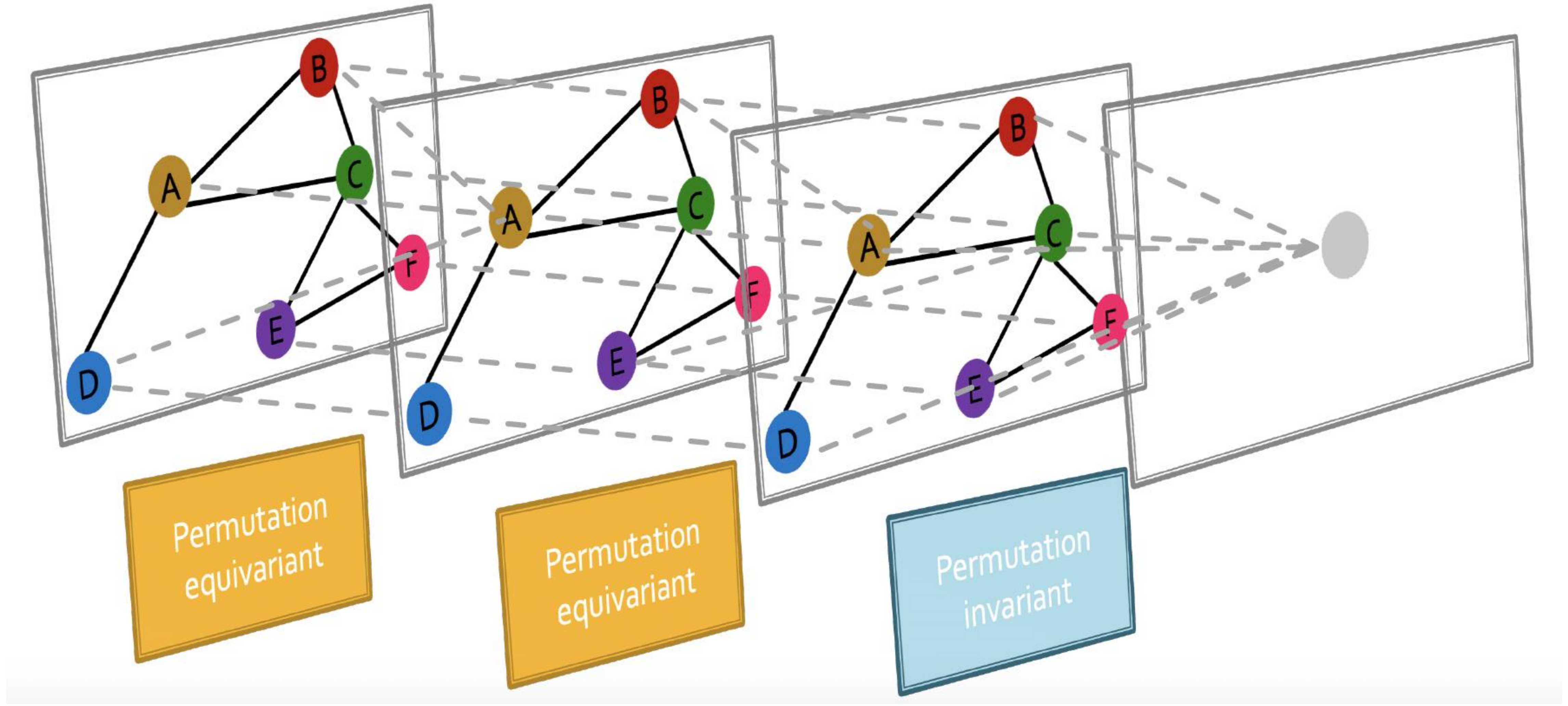


Designing GNN

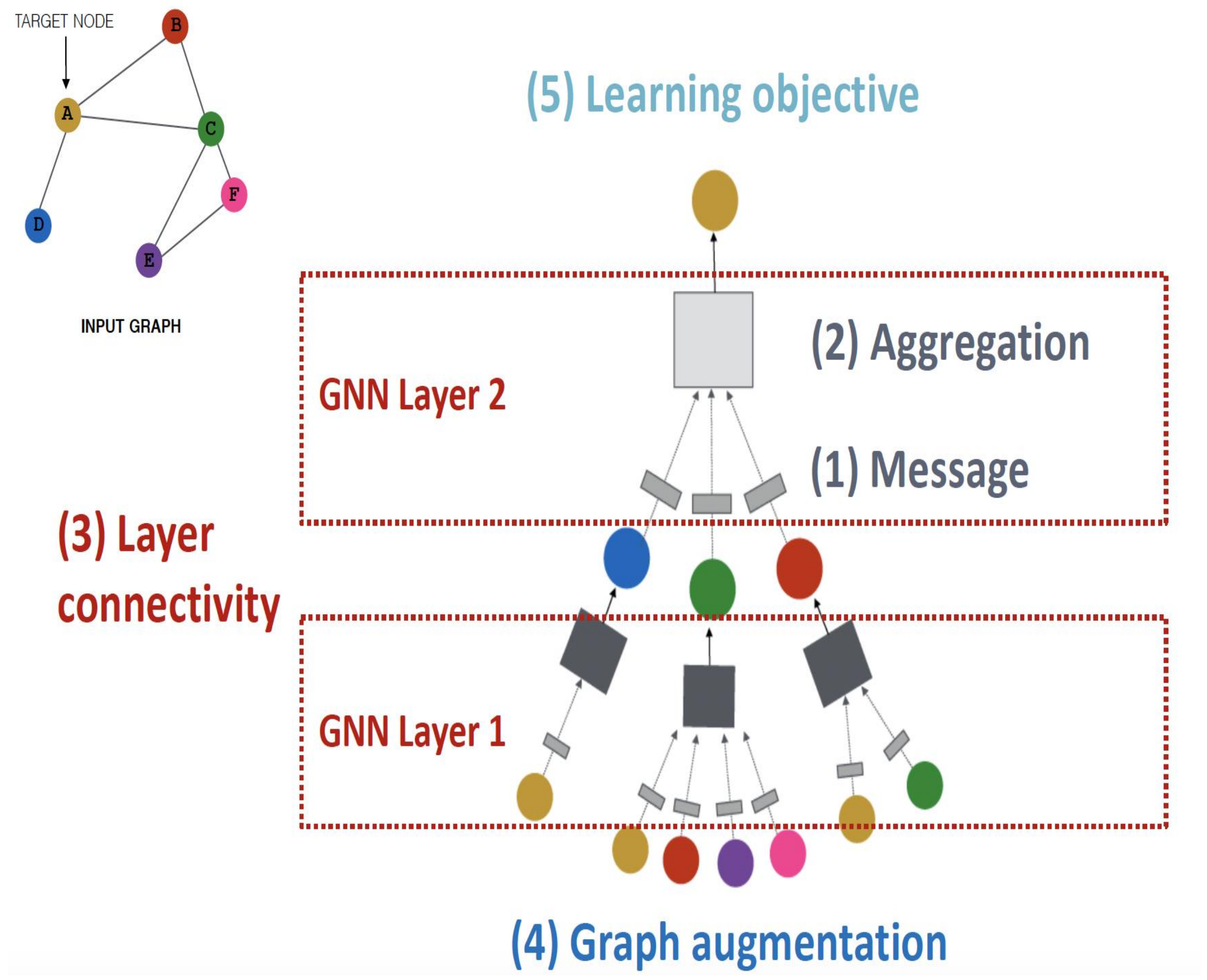
Designing GNN Layers must preserve or respect permutations at each update step. PI and PE are crucial for robust GNN models that handle node reorderings gracefully.

- ❖ **Sampling + Approximation:** Avoid violating permutation properties in large-scale graphs (random sampling, etc.).
- ❖ **Pooling Mechanisms:** Summation/average pooling ensures invariant graph-level outputs.
- ❖ **Challenges:** Hierarchical pooling, dynamic graphs, and advanced aggregator designs can complicate these properties.

GNN consist of multiple permutation equivariant / invariant functions.



A general GNN framework



The Pooling Module

Extracting High-Level Representations

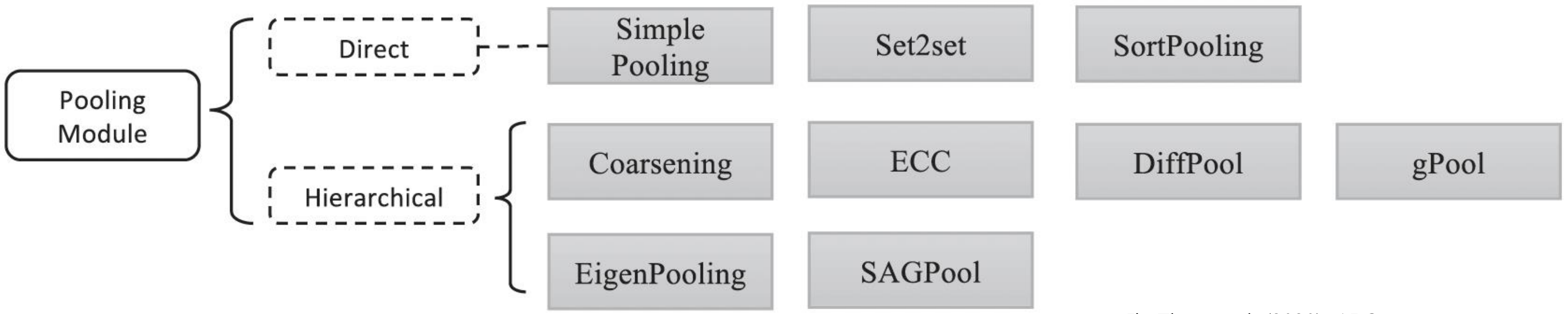
- ❖ Generates compact representations of subgraphs or entire graphs.
- ❖ Essential for tasks like graph classification and hierarchical learning.

Key pooling techniques:

- ❖ **Node Dropout Pooling:** Drops less informative nodes (e.g., Top-K pooling).
- ❖ **Cluster-based Pooling:** Merges similar nodes into clusters (e.g., DiffPool).
- ❖ **Attention-based Pooling:** Assigns weights to nodes based on learned importance.
- ❖ **Maintaining Permutation Invariance:** Ensures that graph representations remain unchanged.

Two main categories:

- ❖ **Direct (Readout) Pooling Modules:** Aggregate node embeddings into a single graph-level embedding in one step.
- ❖ **Hierarchical Pooling Modules:** Iteratively coarsen (or cluster) the graph, creating a hierarchy of smaller graphs or subgraphs.



Jie Zhou, et al. (2020). AI Open

GNN Training Framework

Training Approaches

- **Supervised Learning:** Uses labeled data to train GNNs for node/graph classification.
- **Semi-supervised Learning:** Uses both labeled and unlabeled data to improve training.
- **Unsupervised Learning:** Uses self-supervision (e.g., contrastive learning) to learn node embeddings.

Prediction Tasks in GNNs

- **Node-focused:** Predicts node labels (e.g., node classification) using an MLP or softmax layer.
- **Edge-focused:** Predicts relationships between nodes (e.g., link prediction) using similarity functions or MLPs.
- **Graph-focused:** Generates graph embeddings using pooling layers for tasks like graph classification.

Cross-Entropy Loss: Example

$$L = \sum_{i \in V_{train}} y_i \log(\sigma(h_i^T \theta)) + (1 - y_i) \log(1 - \sigma(h_i^T \theta))$$

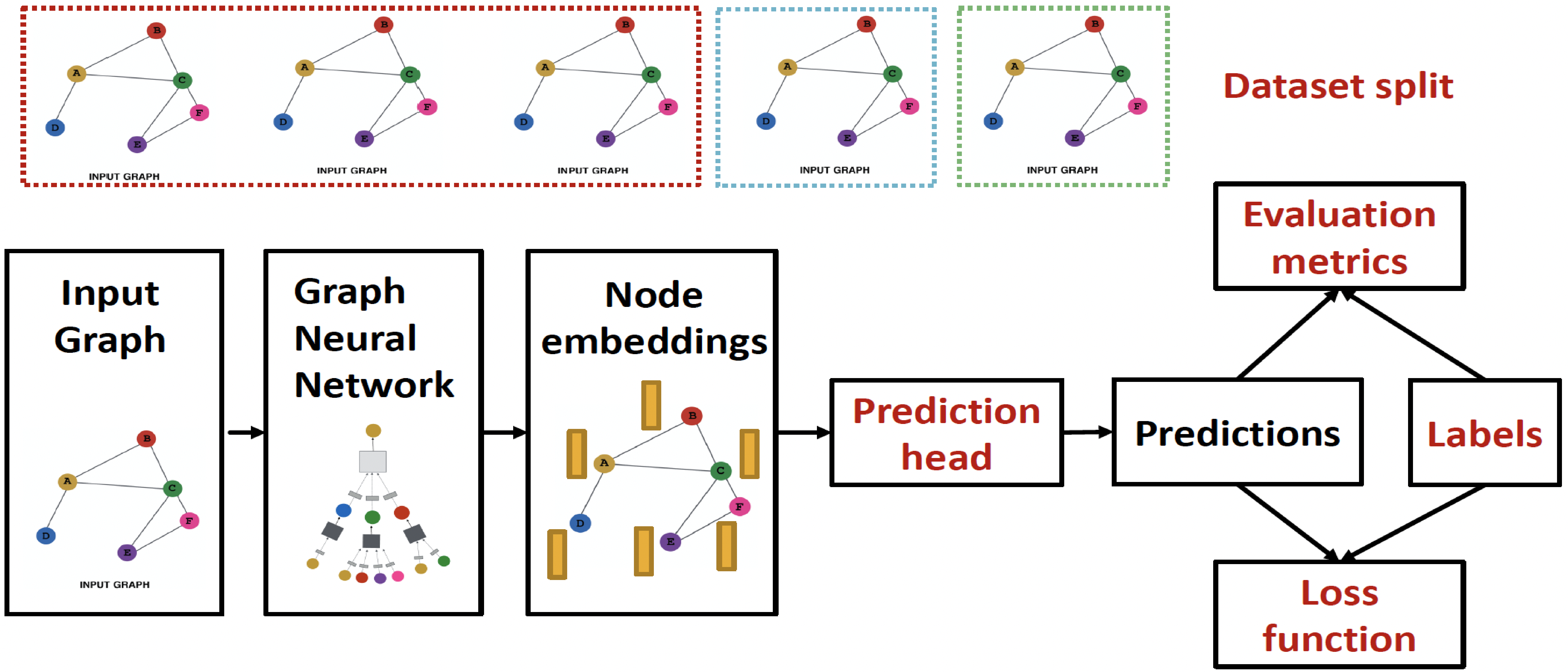
where:

- h_i is the node embedding at the last GNN layer.
- y_i is the true class label of node i.
- θ represents the classification weights.
- σ is the sigmoid function.

Types of Nodes in GNN Training

- **Training Nodes:** Used in loss computation.
- **Transductive Test Nodes:** Processed in GNN but not included in loss computation.
- **Inductive Test Nodes:** Not included in GNN computation or loss function.

GNN Training Pipeline



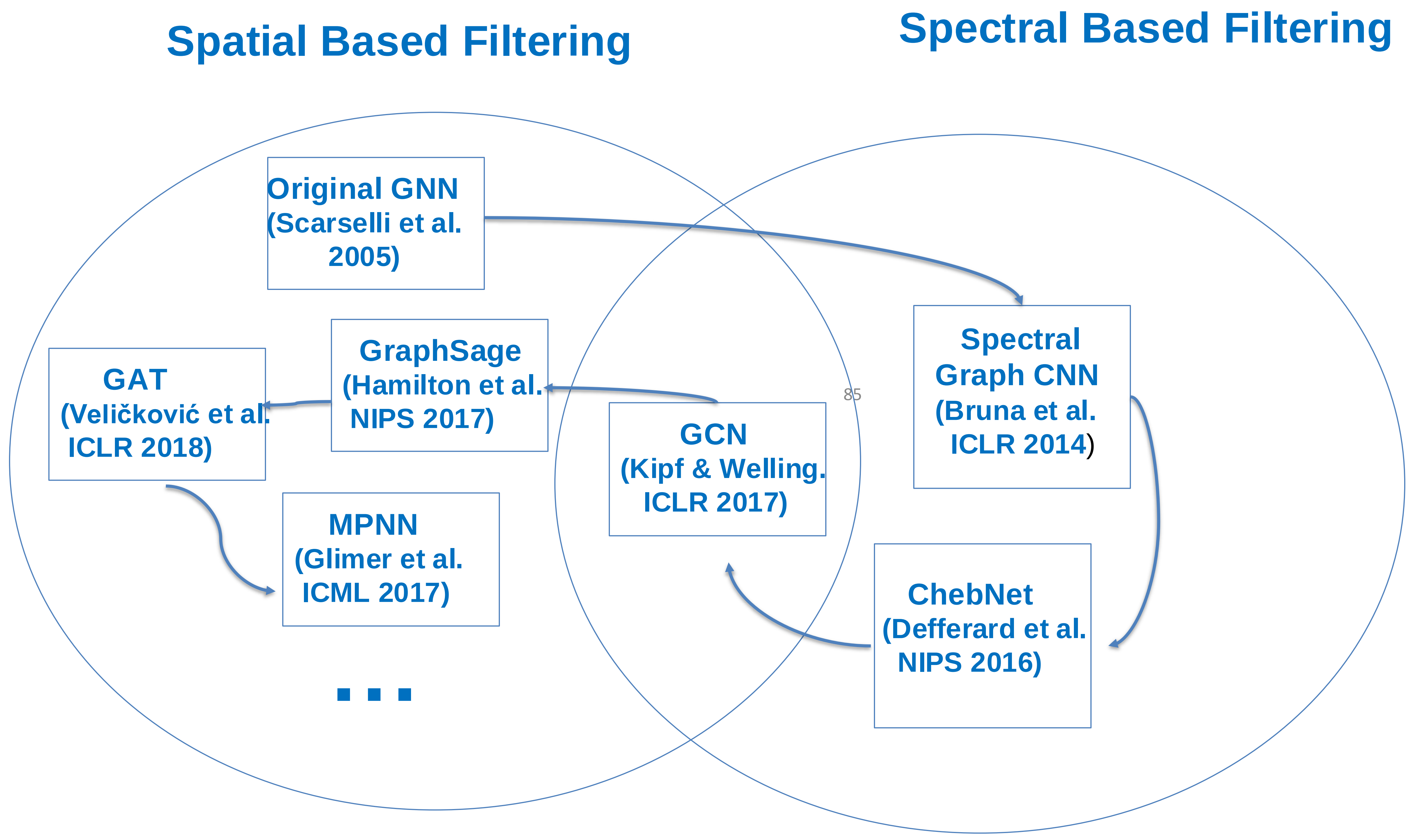
Implementation resources:

PyG provides core modules for this pipeline

GraphGym further implements the full pipeline to facilitate GNN design

Content

Spectral and Spatial GNN Framework

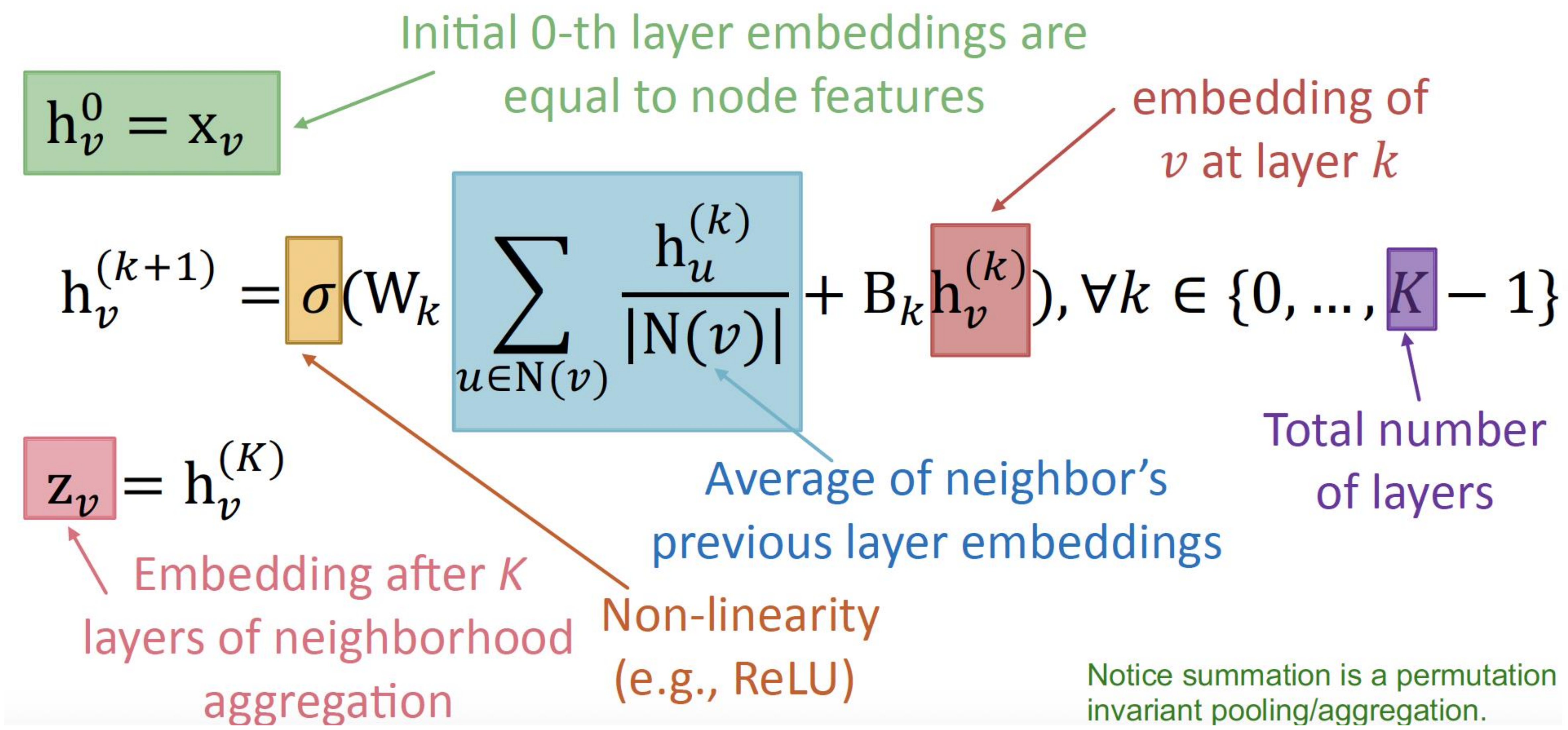

Spatial Based Filtering
Spectral Based Filtering
Original GNN (Scarselli et al. 2005)
GraphSage (Hamilton et al. NIPS 2017)
GAT (Veličković et al. ICLR 2018)
MPNN (Glimer et al. ICML 2017)
…
GCN (Kipf & Welling. ICLR 2017)
Spectral Graph CNN (Bruna et al. ICLR 2014)
ChebNet (Defferard et al. NIPS 2016)

Spatial GNN Framework

Key Concepts:

- ❖ Spatial approaches define convolutions directly on the graph using graph topology.
- ❖ Unlike spectral methods, these approaches operate in the node domain without eigen-decomposition.
- ❖ The challenge lies in handling variable neighborhood sizes and preserving local invariance.

General Spatial Convolution:



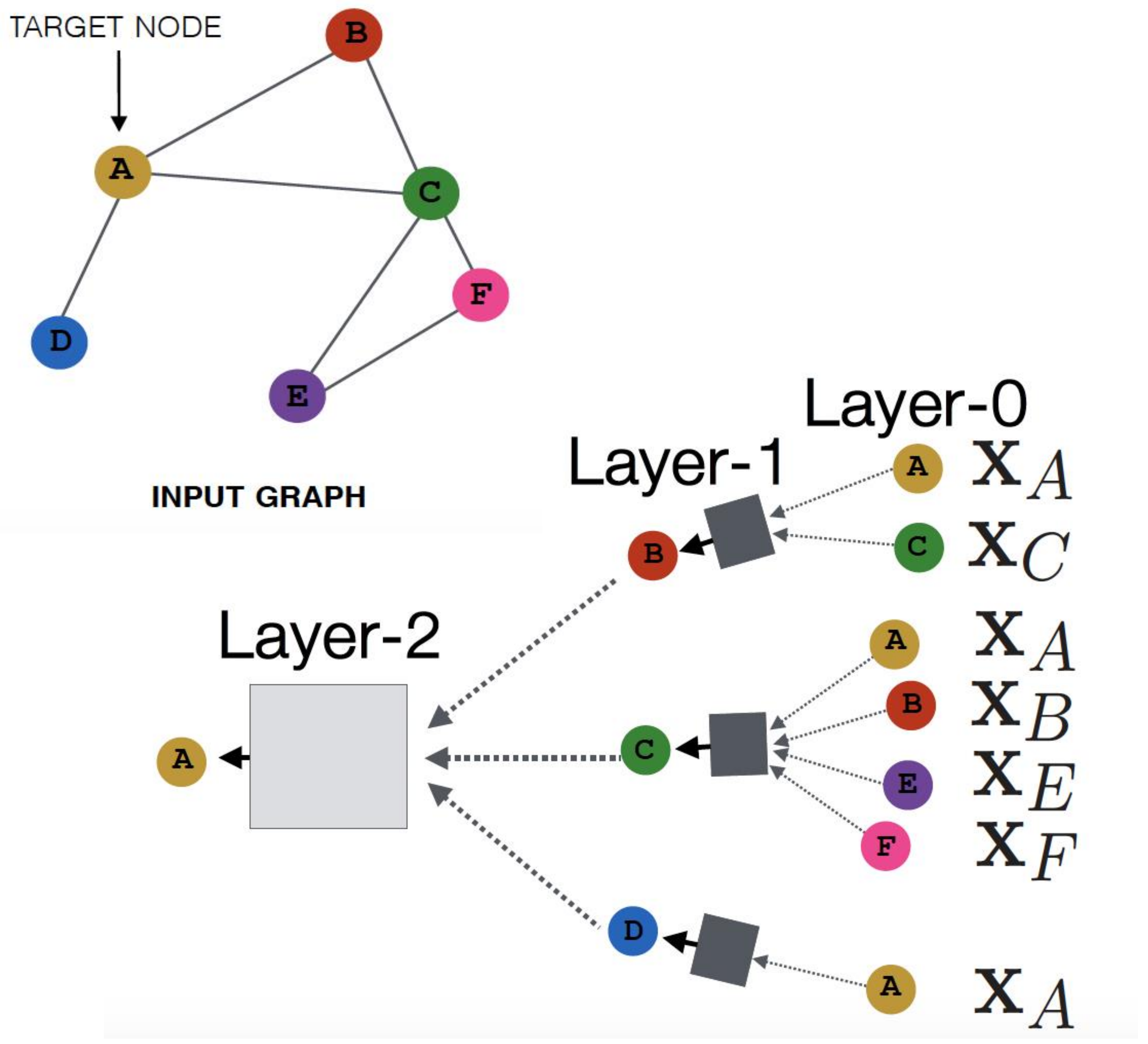
$$h_v^0 = x_v$$

$$h_v^{(k+1)} = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\right), \forall k \in \{0, \dots, K-1\}$$

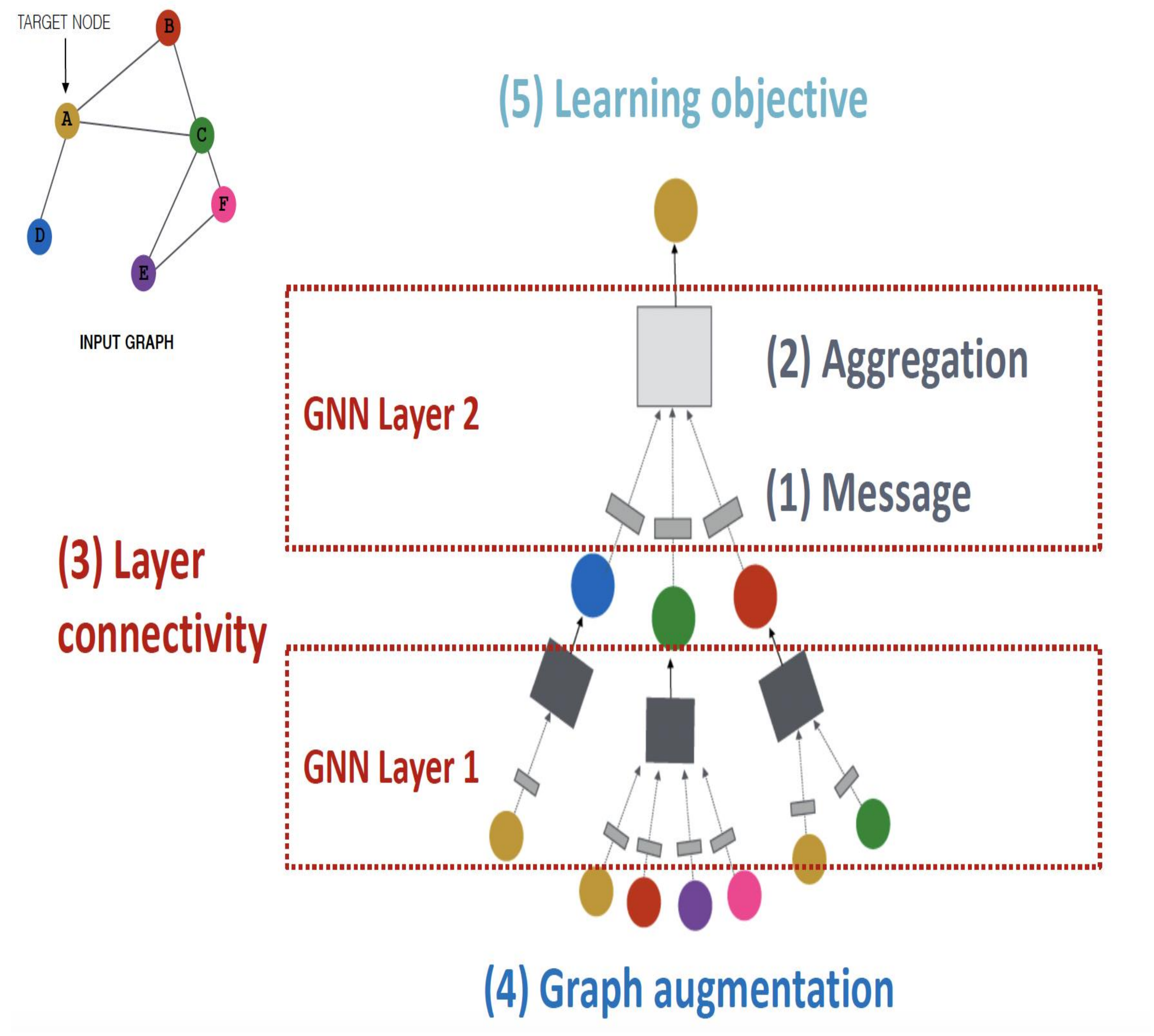
$$z_v = h_v^{(K)}$$

Initial 0-th layer embeddings are equal to node features
embedding of v at layer k
Total number of layers
Average of neighbor's previous layer embeddings
Embedding after K layers of neighborhood aggregation
Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.



Neural Message Passing



The defining feature of a GNN is that is uses a form of *neural message passing*.

During each iteration k, a hidden embedding $h_u^{(k)}$ for node u is updated according to the information **aggregated from its neighborhood $N(u)$**, which can be expressed as follows:

$$h_u^{(k+1)} = update^{(k)}\left(h_u^{(k)}, aggregate^{(k)}\left(\{h_v^{(k)}, \forall v \in N(u)\}\right)\right)$$

We often denote $m_{N(u)} = aggregate^{(k)}\left(\{h_v^{(k)}, \forall v \in N(u)\}\right)$ as the "**message**" aggregated from neighborhood. The initial embeddings at $k = 0$ are set to the input features for all nodes, i.e., $h_u^{(0)} = x_u$. After running K iterations of the GNN message passing, we can use the **output of the final layer to define the embeddings for each node**, i.e., $z_u = h_u^{(K)}, \forall u \in V$.

Neural Message Passing: Intuition

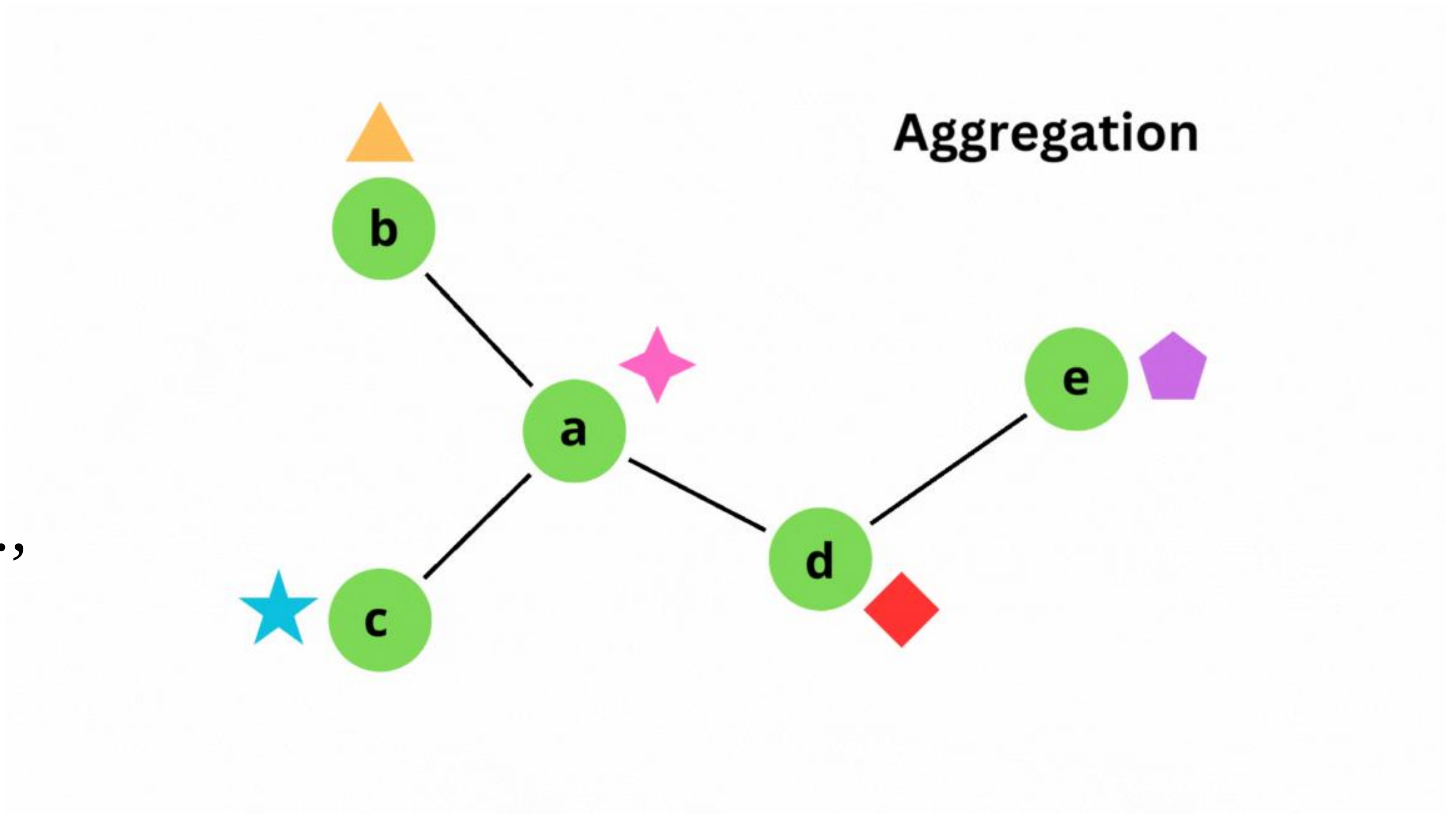
Intuition Behind Message-Passing Framework

- ❖ The core idea of message passing is simple:
  - ➢ At each iteration, every node aggregates information from its 1-hop neighbors.
  - ➢ As iterations progress, nodes encode information from progressively farther regions of the graph.
- ❖ This allows nodes to capture both local and global structures over time.

What Do Node Embeddings Encode?

Node embeddings contain two main types of information:

- **Structural Information**: Local connectivity patterns; Higher-order graph structures; the importance of a node based on its graph position (e.g., centrality measures).
- **Feature Information**: Numerical attributes (e.g., temperature, population density in spatial graphs); Categorical attributes (e.g., user preferences in recommendation systems); Learned representations from deep neural networks.



Why is Message Passing Powerful?

- ❖ Combines local and global information efficiently.
- ❖ Enables deep learning models to capture rich relational patterns.
- ❖ Supports various tasks like node classification, link prediction, and graph generation.

GNN: Basic Form

The basic GNN message passing is defined in **node-level**:

$$h_{\mathrm{u}}^{(k)} = \sigma\left(W_{self}^{(k)} h_u^{(k-1)} + W_{neigh}^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)}\right)$$

where W_{self}, W_{neigh} are trainable parameter and σ denotes an elementwise non-linearity such as ReLU. Alternatively, it can also be succinctly defined in **graph-level**:

$$H^{(t)} = \sigma(H^{(k-1)} W_{self}^{(k)} + A H^{(k-1)} W_{neigh}^{(k)})$$

The basic GNN message passing can be simplified by omitting the explicit update step:

$$h_u^{(k+1)} = aggregate\left(\{\, h_v^{(k)}, \forall v \in N(u) \cup \{u\}\}\right)$$

where now the aggregation is also taken over the node u itself. Adding self-loops is equivalent to sharing parameters between self and neighbor transformations.

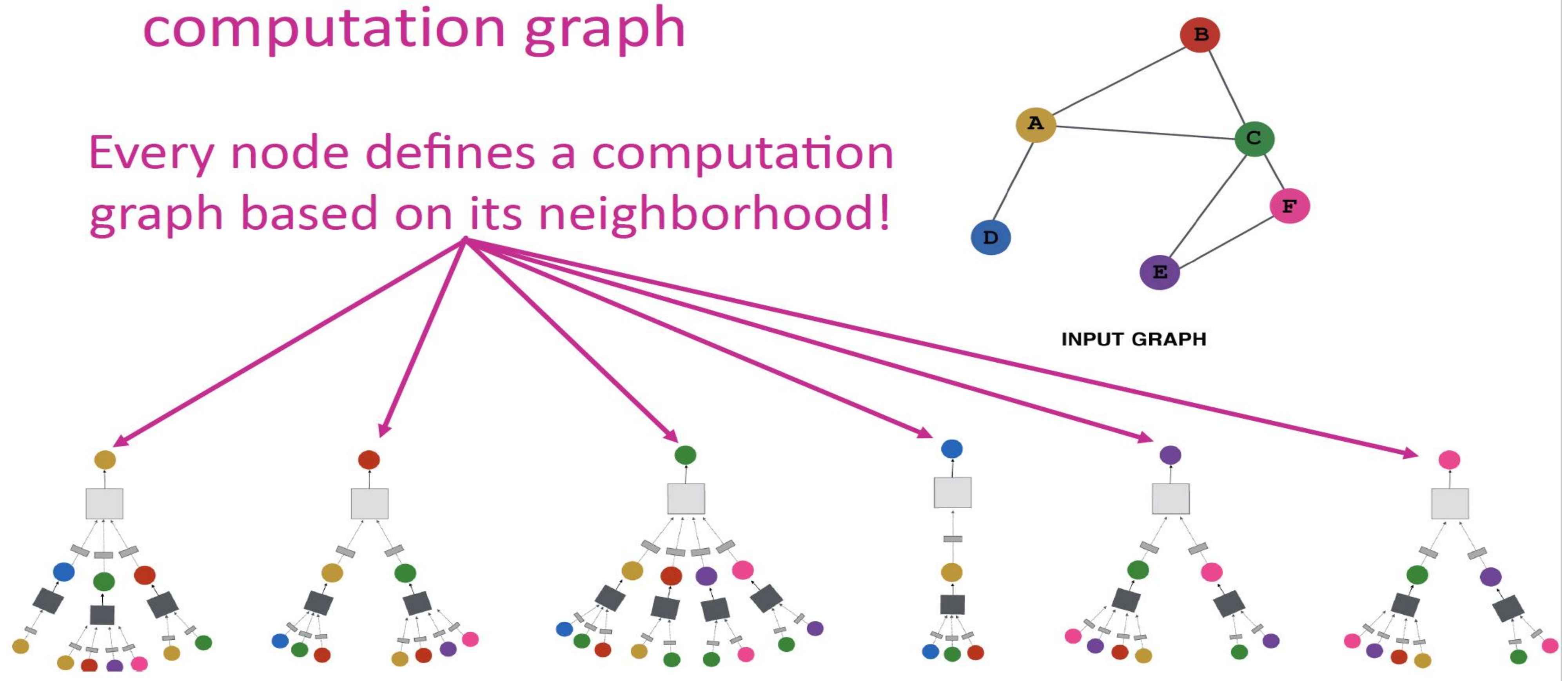
$$\mathbf{H}^{(t)} = \sigma\left((\mathbf{A} + \mathbf{I})\mathbf{H}^{(t-1)}\mathbf{W}^{(t)}\right)$$

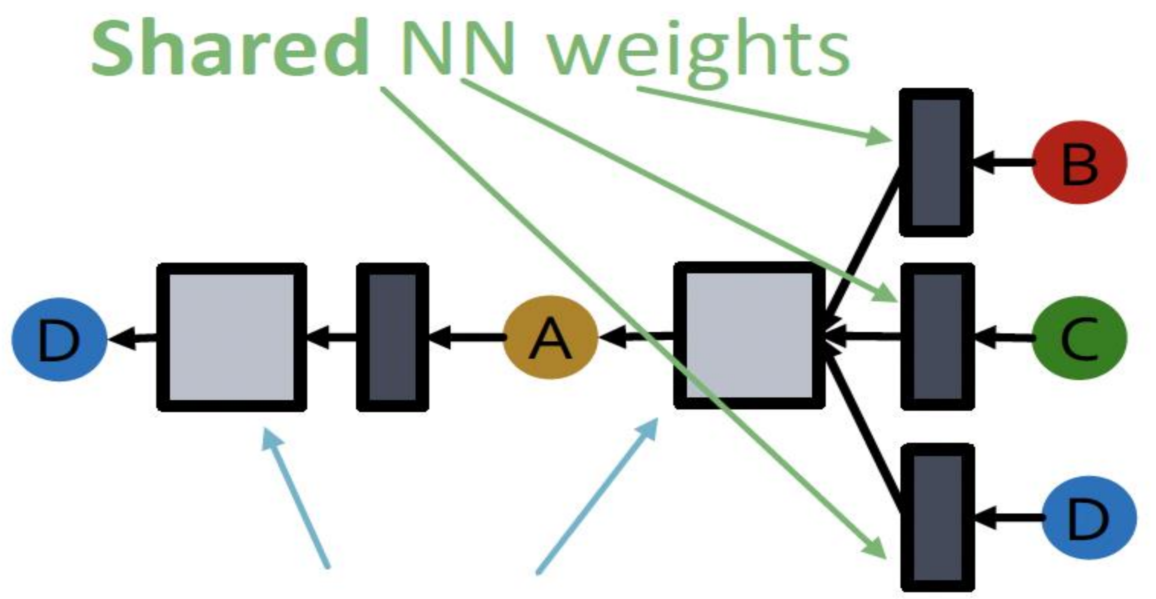
The self-loop GNN approach balances simplicity and efficiency but has some limitations. Self-loops make it harder to differentiate between node and neighbor information. Blurs the distinction between structural and feature information

Permutation Invariant and Equivariant

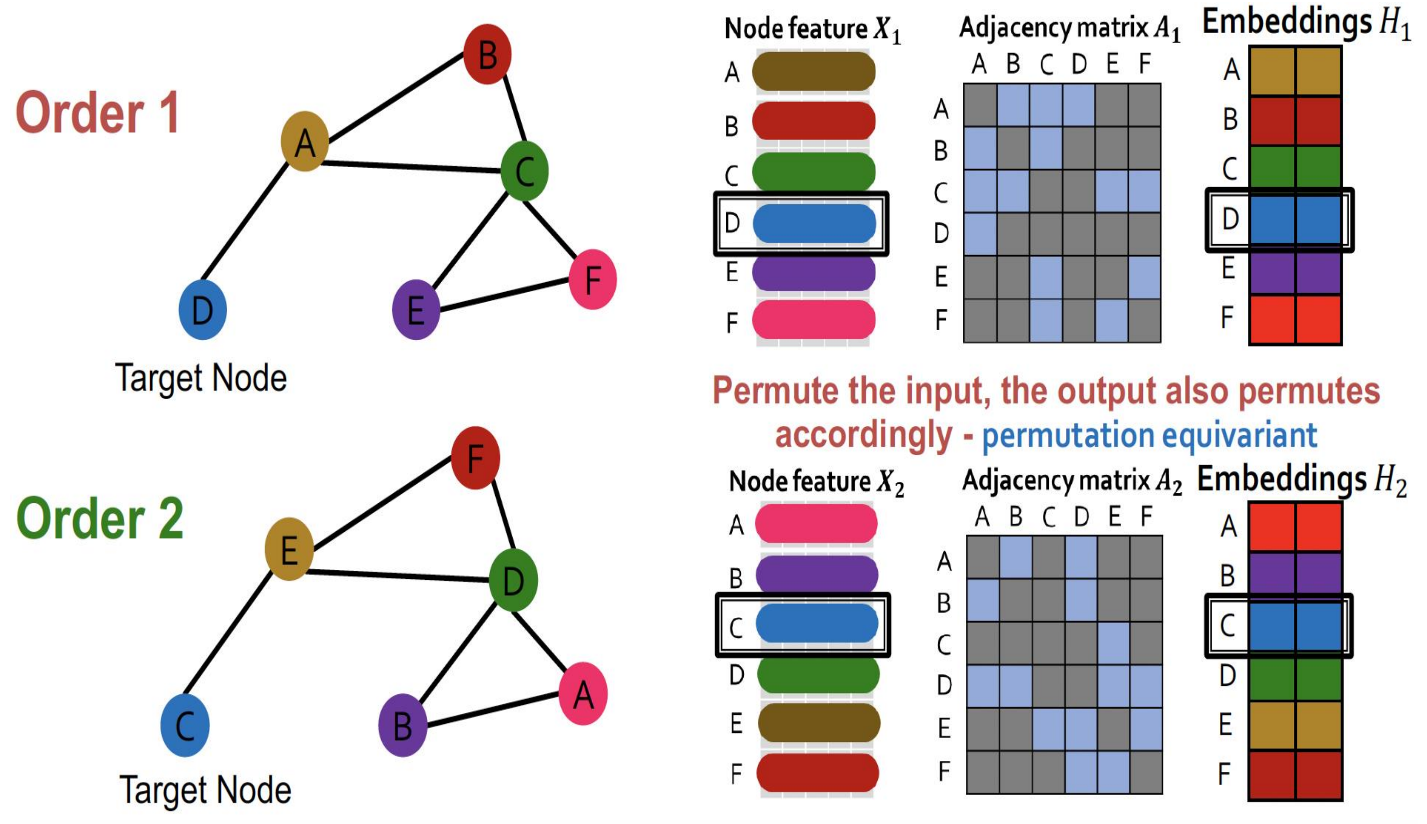
- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!

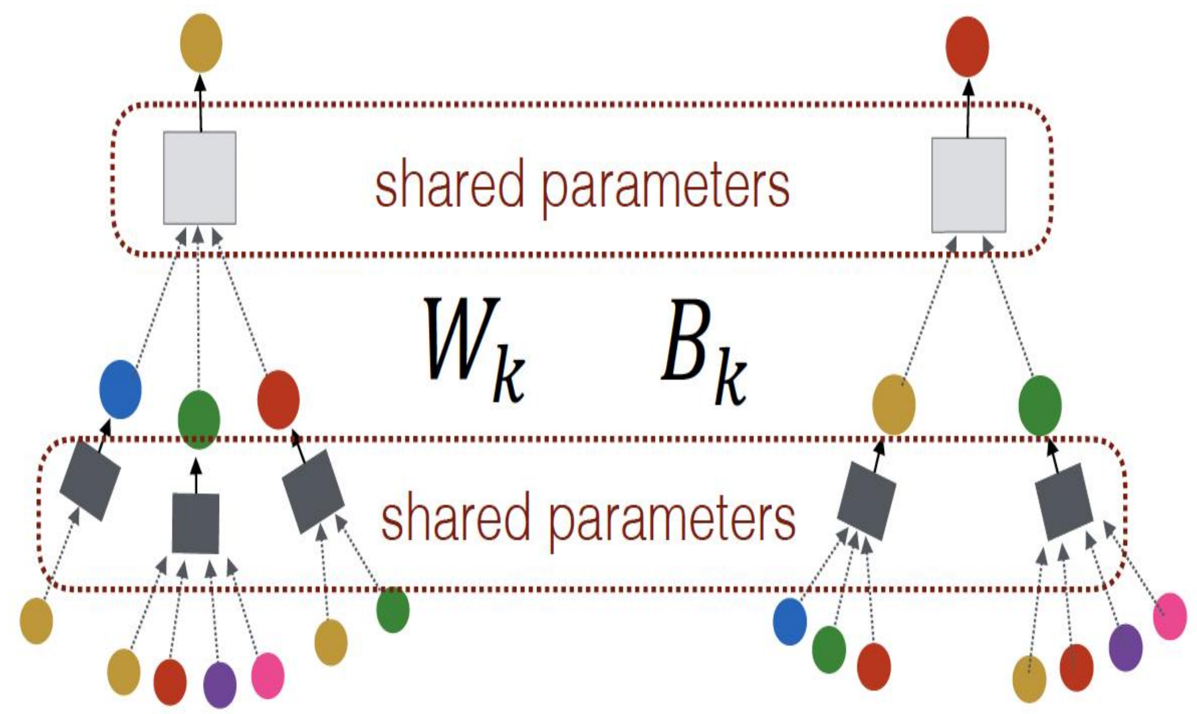




Average of neighbor's previous layer embeddings - **Permutation invariant**



Permute the input, the output also permutes accordingly - permutation equivariant



Compute graph for node A

Compute graph for node B

Neighborhood Normalization

A basic approach is summing neighbor embeddings, but summing neighbor embeddings can create large magnitude differences. Nodes with significantly different degrees may lead to instability and optimization challenges.

$$h_u^{(k+1)} = update^{(k)}\left(h_u^{(k)}, m\big(N(u)\big)\right)$$

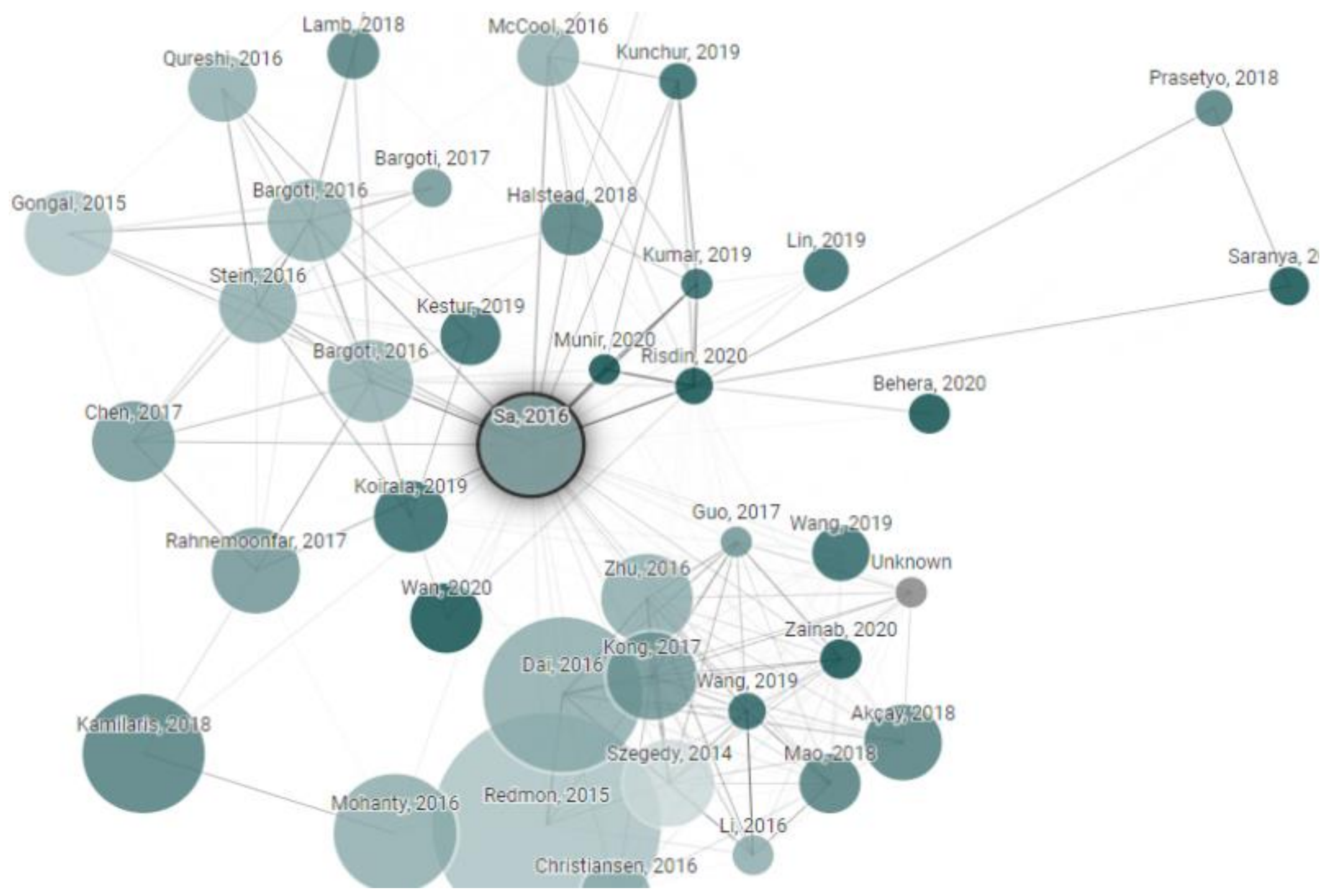
Example: A node with 100× more neighbors than another will have drastically different embedding scales. Leads to numerical instability and difficulties in optimization.

A straightforward solution is **degree-based normalization**:

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v}{|\mathcal{N}(u)|}$$

One solution to this problem is to normalize based upon the degrees of the nodes involved, which is called *symmetric normalization*:

$$m_{N(u)} = \sum_{v \in N(u)} \frac{h_v}{\sqrt{|N(u) \times |N(v)|}}$$



Graph convolutional networks (GCNs)

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right)$$

Deepset and Attention

Deepset/Set pooling

$$m_{N(u)} = MLP_\theta\Big(\sum_{v \in N(u)} MLP_\phi(h_v)\Big)$$

- Given a set of embeddings $\{x_1, x_2, \ldots, x_n\}$, a function f is permutation-invariant if:

$$f(x_1, x_2, \ldots, x_n) = f(x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)})$$

for any permutation σ. - Any permutation-invariant function can be approximated arbitrarily well by a function of the form:

$$f(X) = \rho\left(\sum_{i=1}^{n} \phi(x_i)\right)$$

where ϕ is a transformation applied to each element, and ρ is a post-aggregation transformation.

Another method would be to assign an *attention weight* on the importance to each neighbor so that the neighbor's influence can be weighted during the aggregation step, as proposed in the *Graph Attention Network (GAT):*

$$m_{N(u)} = \sum_{v \in N(u)} \alpha_{u,v} h_v$$

In principle any standard attention model at large can be used,

- The bilinear attention model, since the operation is linear in both h_u and h_v separately, and the weight matrix W makes it more expressive than simple dot-product similarity

$$\alpha_{u,v} = \frac{\exp(h_u^T W h_v)}{\sum_{v' \in N(u)} \exp(h_u^T W h_{v'})}$$

- The multi-head attention model, which is used in the Transformer architecture.

Adding attention is a useful strategy especially in cases where we have prior knowledge to indicate that some neighbors might be more informative than others.

Generalized Message Passing

As the last attempt to generalize the basic neural message passing framework, now we extend the approach beyond the node level, **leveraging edge and graph-level information** at each stage.

One more generalized message passing approach can be formulized according to the following equations:

$$h_{(u,v)}^{(k)} = update_{edge}\left(h_{(u,v)}^{(k-1)}, h_u^{(k-1)}, h_v^{(k-1)}, h_G^{(k-1)}\right)$$
$$m_{N(u)} = aggregate_{node}\left(\{h_{(u,v)}^{(k)}, \forall v \in N(u)\}\right)$$
$$h_u^{(k)} = update_{node}\left(h_u^{(k-1)}, m_{N(u)}, h_G^{(k-1)}\right)$$
$$h_G^{(k)} = update_{graph}(h_G^{(k-1)}, \{h_u^{(k)}, \forall u \in V\}, \{h_{(u,v)}^{(k)}, \forall (u,v) \in E\})$$

The important innovation in this framework is that we generate hidden embeddings not only for each node $h_v^{(k)}$, but also $h_{(u,v)}^{(k)}$ for each edge in the graph as well as an embedding $h_G^{(k)}$ that corresponds to the entire graph. This allows the message passing model to easily integrate edge and graph-level features and have enhanced performances compared to a standard basic GNN. Generating embeddings for edges and the entire graph also makes it trivial to **define loss functions based on the graph or edge-level classification tasks**.

Spectral GNN Framework

What are Spectral GNNs?

- Spectral GNNs use graph signal processing techniques to define convolution operators in the spectral domain.
- They leverage the eigen-decomposition of the graph Laplacian to transform signals to the frequency domain.

Eigen-decomposition of Normalized Laplacian Matrix

$$L = I_N - D^{-1/2} A D^{-1/2} = \underbrace{\begin{bmatrix} | & & | \\ \mathbf{u}_0 & \cdots & \mathbf{u}_{N-1} \\ | & & | \end{bmatrix}}_{\boldsymbol{U}} \underbrace{\begin{bmatrix} \lambda_0 & & 0 \\ & \ddots & \\ 0 & & \lambda_{N-1} \end{bmatrix}}_{\boldsymbol{\Lambda}} \underbrace{\begin{bmatrix} — & \mathbf{u}_0 & — \\ & \vdots & \\ — & \mathbf{u}_{N-1} & — \end{bmatrix}}_{\boldsymbol{U}^{\boldsymbol{T}}}$$

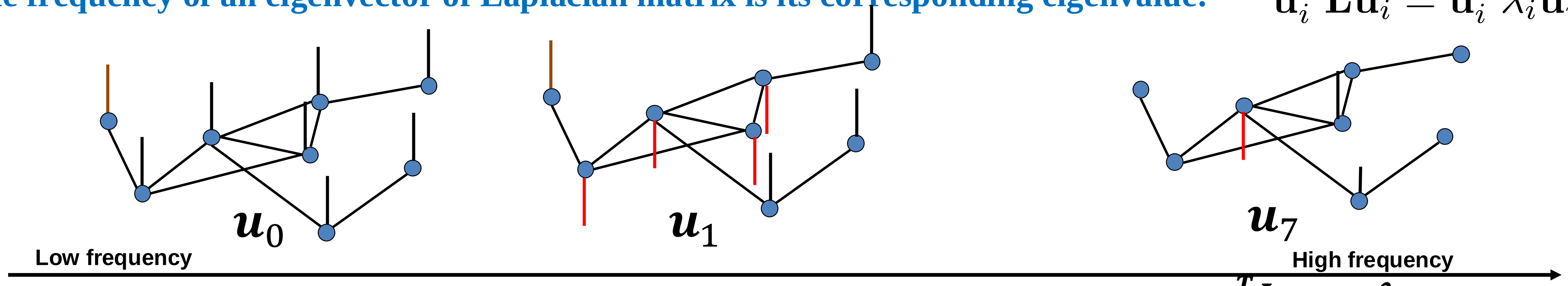
Eigenvalues are sorted non-decreasingly:

$$0 = \lambda_0 < \lambda_1 \leq \cdots \lambda_{N-1}$$

Frequency of the signal $\boldsymbol{u_i}$

$$\mathbf{u}_i^T \mathbf{L} \mathbf{u}_i = \mathbf{u}_i^T \lambda_i \mathbf{u}_i = \lambda_i$$

The frequency of an eigenvector of Laplacian matrix is its corresponding eigenvalue:



$$\boldsymbol{u}_0^T \boldsymbol{L} \boldsymbol{u}_0 = \boldsymbol{\lambda}_0 = 0 \qquad \boldsymbol{u}_1^T \boldsymbol{L} \boldsymbol{u}_1 = \boldsymbol{\lambda}_1 \qquad \boldsymbol{u}_7^T \boldsymbol{L} \boldsymbol{u}_7 = \boldsymbol{\lambda}_7$$

Graph Fourier Transform (GFT) and Inverse GFT

A signal f can be written as graph Fourier series:

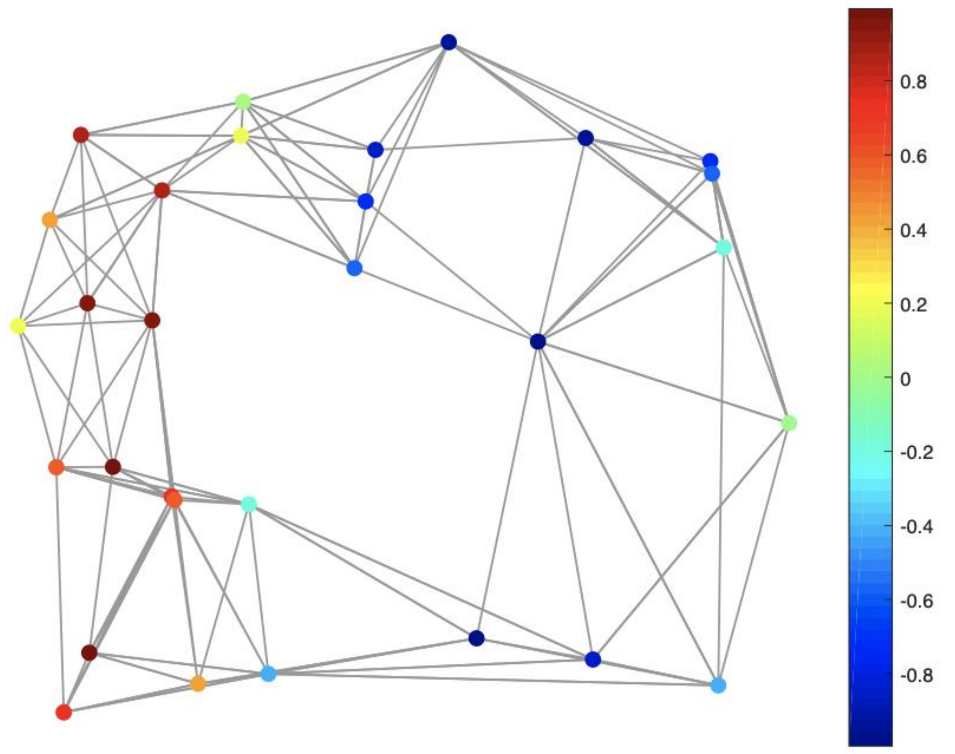
$$f = \sum_{i=0}^{N-1} \underbrace{\hat{f}_i}_{f^T u_i} \cdot u_i$$

$$\mathscr{F}(\mathbf{x}) = \mathbf{U}^T\mathbf{x},$$
$$\mathscr{F}^{-1}(\mathbf{x}) = \mathbf{U}\mathbf{x}.$$

$\boldsymbol{u_i}$: graph Fourier mode

$\boldsymbol{\lambda_i}$: frequency

$\boldsymbol{\hat{f}_i}$: graph Fourier coefficients



Spatial domain: f

$$\hat{f} = U^T f$$

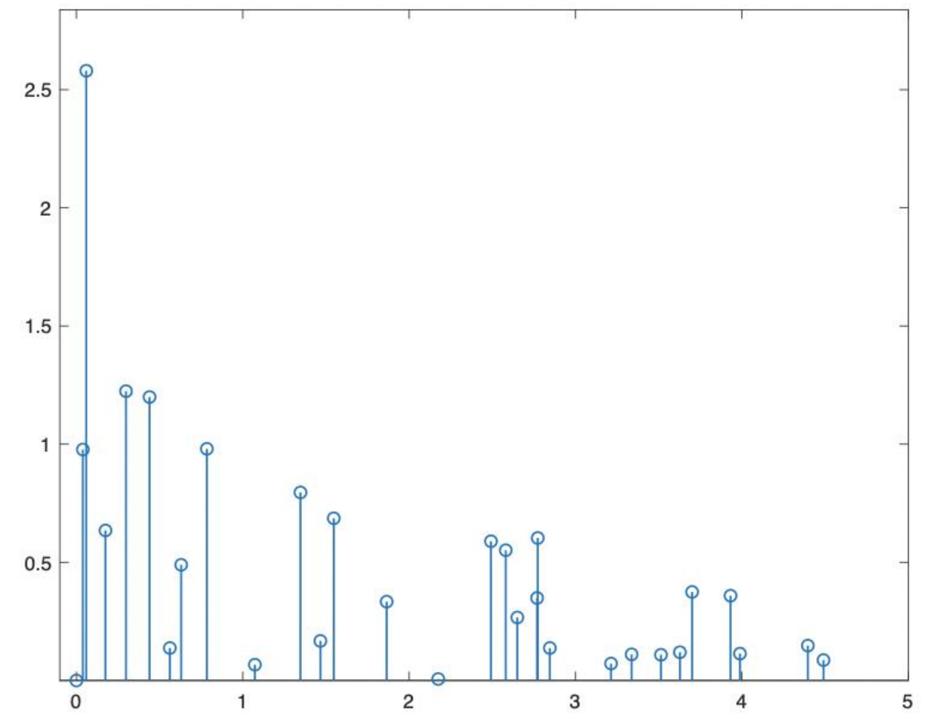
$$\xrightarrow{\hspace{6em}}$$

Decompose signal f

$$f = U\hat{f}$$

$$\xleftarrow{\hspace{6em}}$$

Reconstruct signal f



Spectral domain: $\hat{f}$

Graph Spectral Filtering

Recall:

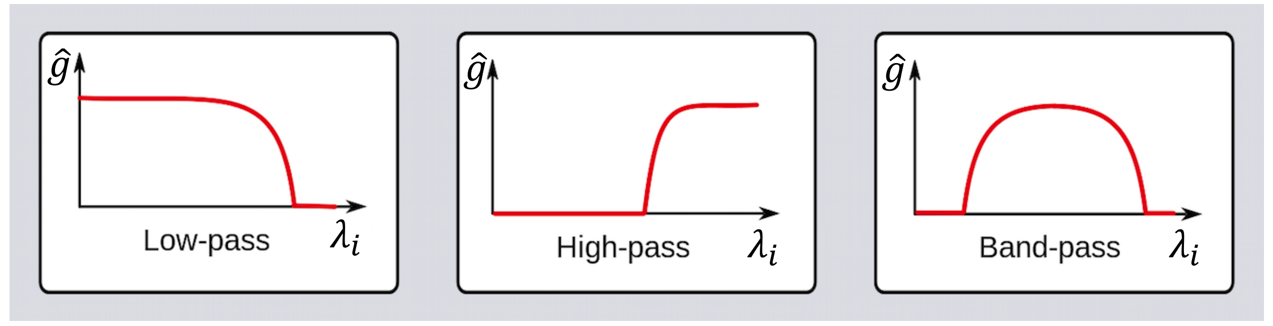
$$GFT: \hat{f} = U^T f \qquad IGFT: f = U\hat{f}$$

Filter a graph signal f:

$$\begin{tikzcd}
\boxed{f} \arrow[r, "GFT", "\text{Decompose}"'] & \boxed{u_i^T f} \arrow[r, "\hat{g}(\lambda_i)", "\text{Filter}"'] & \boxed{\hat{g}(\lambda_i) \cdot u_i^T f} \arrow[r, "IGFT", "\text{Reconstruct}"'] & \boxed{\sum_{i=0}^{N-1} \hat{g}(\lambda_i) \cdot u_i^T f \cdot u_i}
\end{tikzcd}$$

Coefficients **Filtered coefficients**

Example:



Filter $\hat{g}(\lambda_i)$: Modulating the frequency

Graph Spectral Filtering

$$\begin{tikzcd} \boxed{\boldsymbol{f}} \arrow[r, "\text{Filtering}"'] & \boxed{\hat{g}(L)\boldsymbol{f} = U\hat{g}(\Lambda)U^T\boldsymbol{f}} \end{tikzcd}$$

Filter a graph signal f:

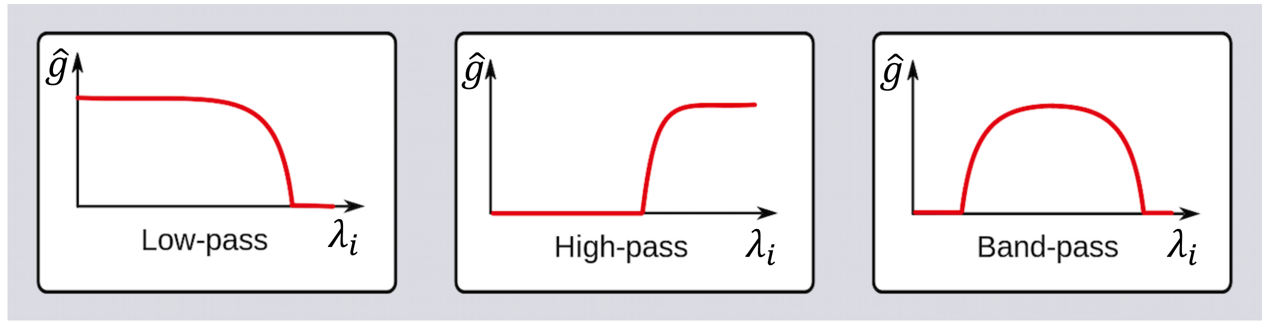
$$\begin{tikzcd} \boxed{\boldsymbol{f}} \arrow[r, "GFT", "\text{Decompose}"'] & \boxed{U^T\boldsymbol{f}} \arrow[r, "\hat{g}(\Lambda)", "\text{Filter}"'] & \boxed{\hat{g}(\Lambda)U^T f} \arrow[r, "IGFT", "\text{Reconstruct}"'] & \boxed{U\hat{g}(\Lambda)U^T\boldsymbol{f}} \end{tikzcd}$$

Coefficients

Filtered coefficients

$$\hat{g}(\Lambda) = \begin{bmatrix} \hat{g}(\lambda_0) & & 0 \\ & \ddots & \\ 0 & & \hat{g}(\lambda_{N-1}) \end{bmatrix}$$

Example:



Convolution Theorem and Approximation

$$g \star \mathbf{f} = \mathcal{F}^{-1}(\mathcal{F}(g) \odot \mathcal{F}(\mathbf{f})) = U(U^T g \odot U^T \mathbf{f})$$

$$g_w \star \mathbf{f} = U g_w U^T \mathbf{f}$$

Key Properties:

❖ Computation requires eigen-decomposition of the Laplacian.
❖ Filters are defined in the spectral domain, enabling flexible frequency-based operations.

Chebyshev Approximation:

- Hammond et al. (2011) introduced Chebyshev approximation.
- Avoids explicit eigen-decomposition.
- Approximates filters with Chebyshev polynomials:

$$g_w \star \mathbf{f} \approx \sum_{k=0}^{K} w_k T_k(\tilde{L})\mathbf{f},$$

  where $T_k(\cdot)$ are Chebyshev polynomials.
- Reduces computational complexity.

Graph Convolutional Networks:

- Simplifies Chebyshev approximation to first-order ($K = 1$):

$$g_w \star \mathbf{f} \approx w_0 \mathbf{f} + w_1(L - I_N)\mathbf{f}.$$

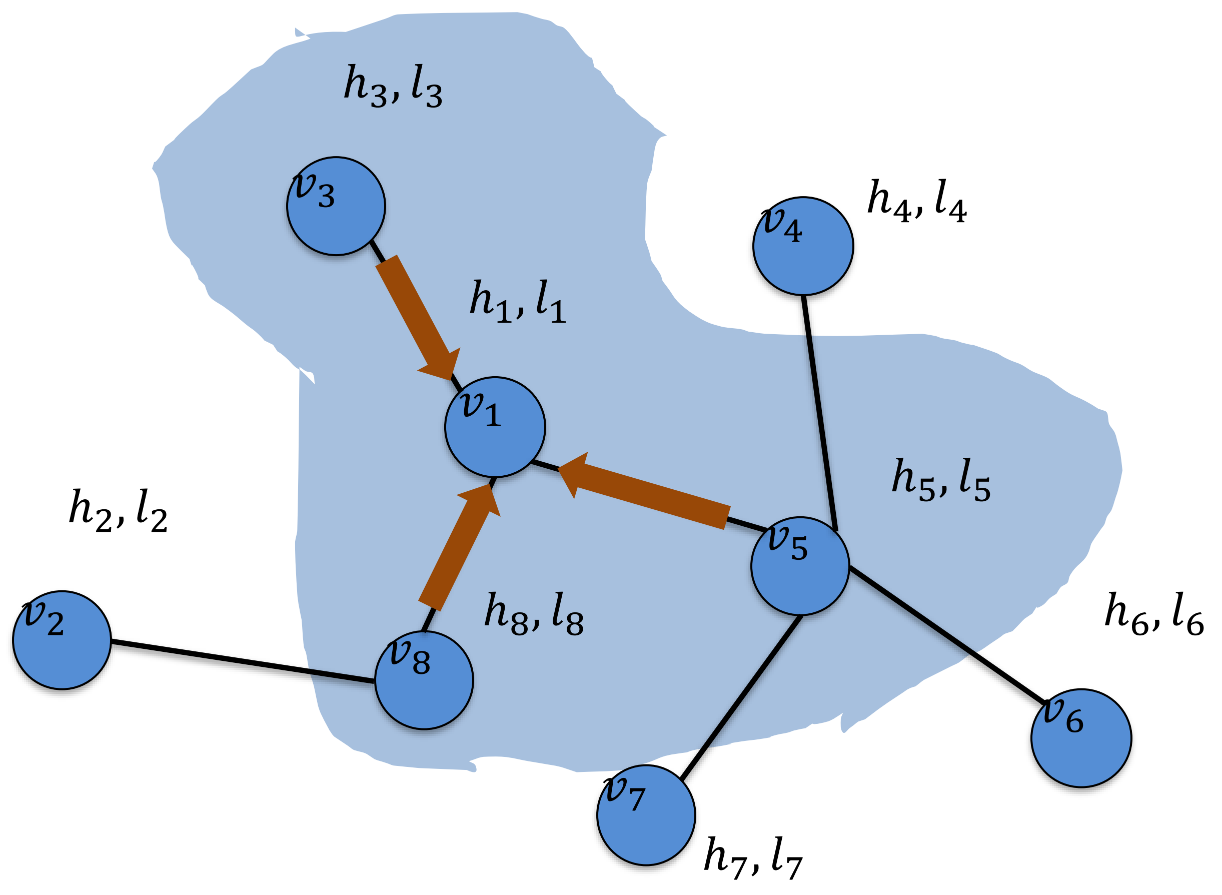
- Further simplifies to:

$$g_w \star \mathbf{f} \approx w(I_N + D^{-1/2} A D^{-1/2})\mathbf{f}.$$

- Final GCN propagation rule:

$$\tilde{A} = A + I_N, \quad \tilde{D}_{ii} = \sum_j \tilde{A}_{ij}. \qquad H^{(l+1)} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2} H^{(l)} W^{(l)}).$$

Graph Filtering in the First GNN paper



h_i: The hidden features

l_i: The input features

$$\boldsymbol{h}_{\boldsymbol{i}}^{(\boldsymbol{k}+\mathbf{1})} = \sum_{\boldsymbol{v}_{\boldsymbol{j}} \in \boldsymbol{N}(\boldsymbol{v}_{\boldsymbol{i}})} \boldsymbol{f}\left(\boldsymbol{l}_{\boldsymbol{i}}, \boldsymbol{h}_{\boldsymbol{j}}^{(\boldsymbol{k})}, \boldsymbol{l}_{\boldsymbol{j}}\right), \qquad \forall\, \boldsymbol{v}_{\boldsymbol{i}} \in \boldsymbol{V}.$$

$N(v_i)$: Neighbors of the node v_i.

$f(\cdot)$: Feedforward neural network.

Graph neural networks for ranking web pages. *WI*. IEEE, 2005.

Graph Spectral Filtering for GNN

How to design the filter?

Data-driven! Learn $\hat{g}(\Lambda)$ from data!

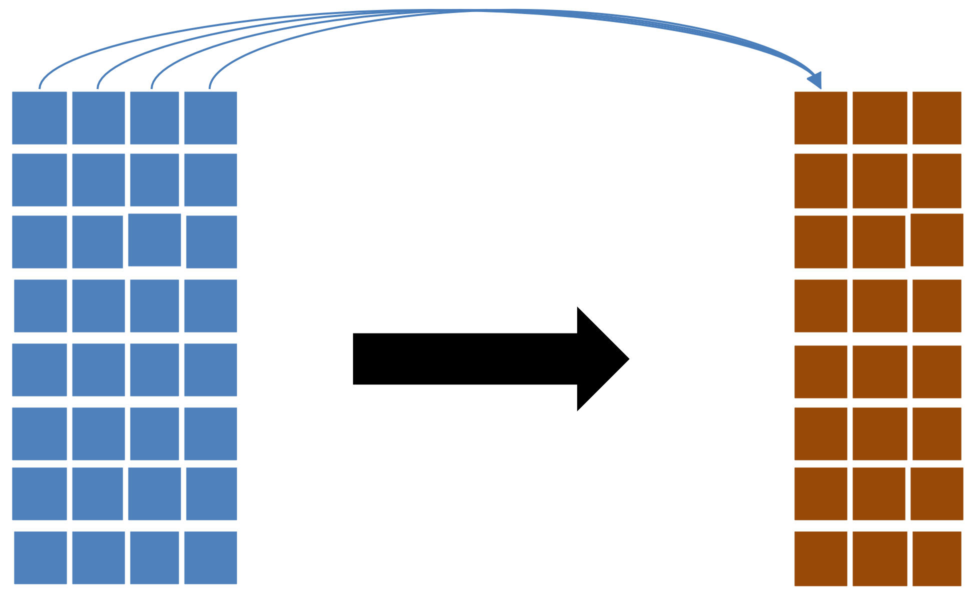
How to deal with multi-channel signals?

$$\boldsymbol{F}_{in} \in \mathbb{R}^{N\times d_1} \rightarrow \boldsymbol{F}_{out} \in \mathbb{R}^{N\times d_2}.$$

Each input channel contributes to each output channel

$$\boldsymbol{F}_{out}[:,i] = \sum_{j=1}^{d_1} \underline{\hat{g}_{ij}(\mathbf{L})\boldsymbol{F}_{in}[:,j]} \quad i = 1,\dots d_2$$

Filter each input channel



Learn $d_2 \times d_1$ filters

Graph Spectral Filtering for GNN

$\hat{g}(\Lambda)$: Non-parametric

$$\hat{g}(\Lambda) = \begin{bmatrix} \hat{g}(\lambda_0) & & 0 \\ & \ddots & \\ 0 & & \hat{g}(\lambda_{N-1}) \end{bmatrix}$$

$\hat{g}(\Lambda)$: Polynomial Parametrized

$$\hat{g}(\Lambda) = \begin{bmatrix} \sum_{k=0}^{K} \theta_k \lambda_1^k & & & \\ & \sum_{k=0}^{K} \theta_k \lambda_2^k & & \\ & & \ddots & \\ & & & \sum_{k=0}^{K} \theta_k \lambda_N^k \end{bmatrix}$$

$d_2 \times d_1 \times K$ parameters

$$\mathrm{U}\hat{g}(\Lambda)U^T f = \sum_{k=0}^{K} \theta_k L^k f$$

No eigen-decomposition needed

Polynomial Parametrized Filter: a Spatial View

$$U\hat{g}(\Lambda)U^T f(i) = \sum_{j=0}^{N}\sum_{k=0}^{K}\theta_k L_{i,j}^k f(j)$$

If the node v_j is more than K-hops away from node v_i, then,

$$\sum_{k=0}^{K}\theta_k L_{i,j}^k = 0$$

The filter is localized within K-hops neighbors in the spatial domain

Chebyshev Polynomials

The polynomials adopted have non-orthogonal basis $1, x, x^2, x^3, \ldots$

$$g(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots$$

Unstable under perturbation of coefficients

Chebyshev polynomials:

Recursive definition:

- $T_0(x) = 1; T_1(x) = x$
- $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$

The Chebyshev polynomials $\{T_k\}$ form an orthogonal basis for the Hilbert space $L^2([-1,1], \frac{dy}{\sqrt{1-y^2}})$.

Chebyshev Polynomials

The polynomials adopted have non-orthogonal basis $1, x, x^2, x^3, \ldots$

$$g(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots$$

Unstable under perturbation of coefficients

Chebyshev polynomials:

Recursive definition:

- $T_0(x) = 1; T_1(x) = x$
- $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$

The Chebyshev polynomials $\{T_k\}$ form an orthogonal basis for the Hilbert space $L^2([-1,1], \frac{dy}{\sqrt{1-y^2}})$.

$$g(x) = \theta_0 T_0(x) + \theta_1 T_1(x) + \theta_2 T_2(x) + \cdots$$

ChebNet

Parametrize $\hat{g}(\Lambda)$ with Chebyshev polynomials

$$\hat{g}(\Lambda) = \sum_{k=0}^{K} \theta_k T_k(\tilde{\Lambda}), with\ \tilde{\Lambda} = \frac{2\Lambda}{\lambda_{max}} - 1$$

$d_2 \times d_1 \times K$ parameters

$\mathrm{U}\hat{g}(\Lambda)U^T f = \sum_{k=0}^{K} \theta_k T_k(\tilde{L})\, f$, with $\tilde{L} = \frac{2L}{\lambda_{max}} - I$

No eigen-decomposition needed

Stable under perturbation of coefficients

GCN: Simplified ChebNet

Use Chebyshev polynomials with $K = 1$ and assume $\lambda_{max} = 2$

$$\hat{g}(\Lambda) = \theta_0 + \theta_1(\Lambda - I)$$

Further constrain $\theta = \theta_0 = -\theta_1$

$$\hat{g}(\Lambda) = \theta(2I - \Lambda)$$

$$U\hat{g}(\Lambda)U^T f = \theta(2I - L)f = \theta\left(I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}\right)f$$

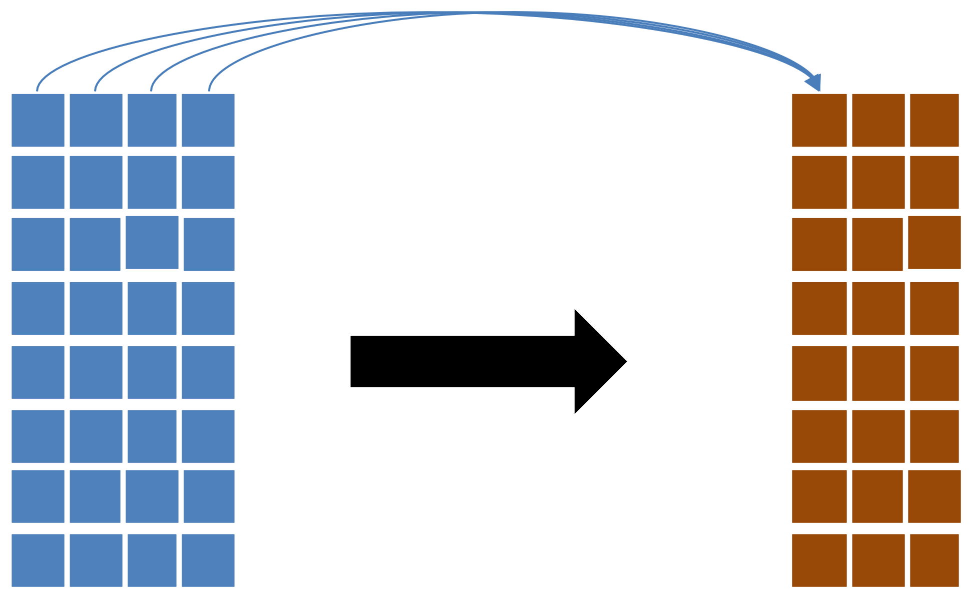
Apply a renormalization trick

$$U\hat{g}(\Lambda)U^T f = \theta\left(\widetilde{D}^{-\frac{1}{2}}\tilde{A}\widetilde{D}^{-\frac{1}{2}}\right)f, with\ \hat{A} = A + I$$

GCN for Multi-channel Signal

Recall:

$$F_{out}[:,i] = \sum_{j=1}^{d_1} \underbrace{\hat{g}_{ij}(\mathbf{L}) F_{in}[:,j]}_{\text{Filter each input channel}} \quad i = 1, \dots d_2$$



For GCN:

$$F_{out}[:,i] = \sum_{j=1}^{d_1} \underbrace{\theta_{ji}(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}})}_{\text{GCN filter}} F_{in}[:,j] \quad i = 1, \dots d_2$$

In matrix form:

$$F_{out} = (\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}) F_{in} \Theta \text{ with } \Theta \in \mathbb{R}^{d_1 \times d_2} \text{ and } \Theta[j,i] = \theta_{ji}$$

A Spatial View of GCN Filter

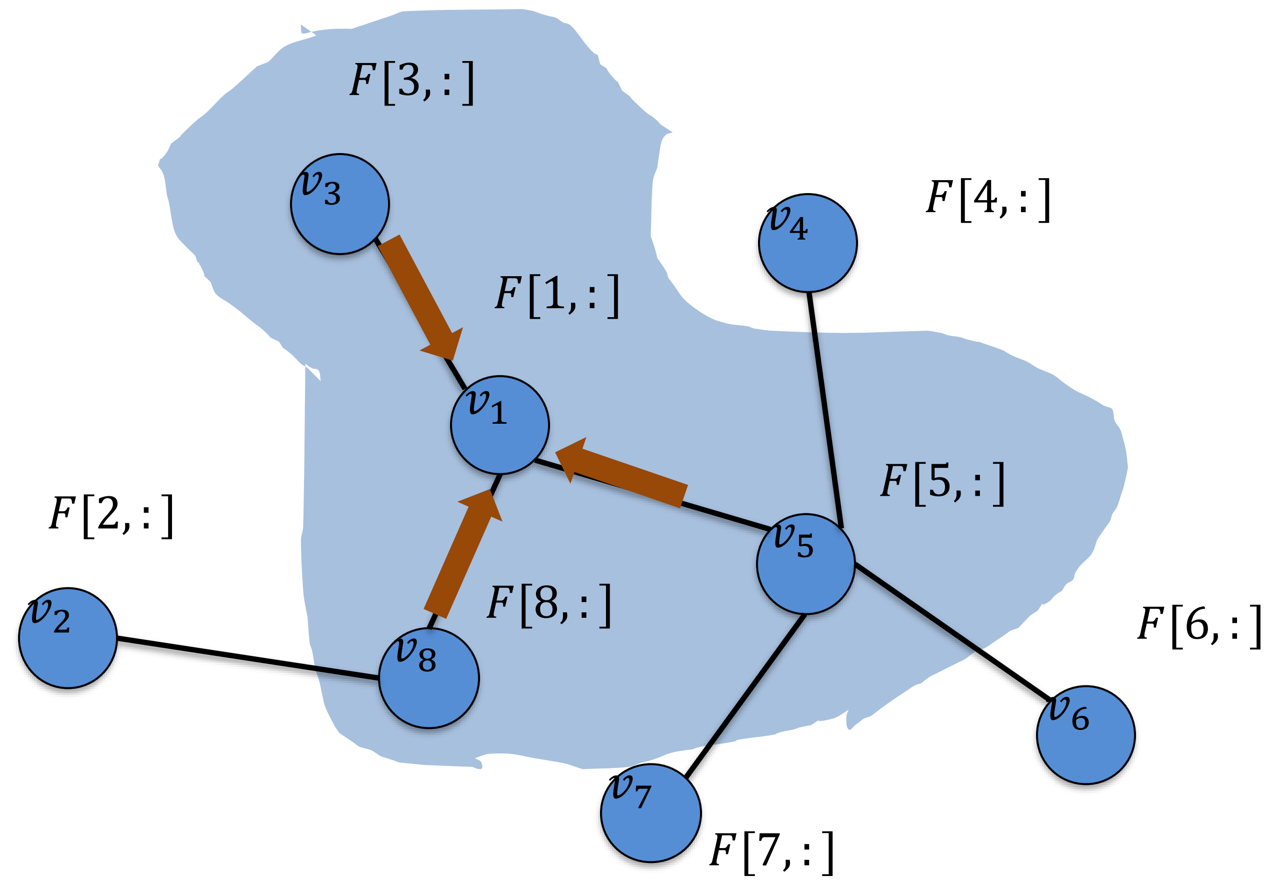
Denote $C = \widetilde{D}^{-\frac{1}{2}}\tilde{A}\widetilde{D}^{-\frac{1}{2}}$

- Then $F_{out} = CF_{in}\Theta$
- For node v_i, $F_{out}[i,:] = \sum_j C[i,j]F_{in}[j,:]\Theta$

Observe that:

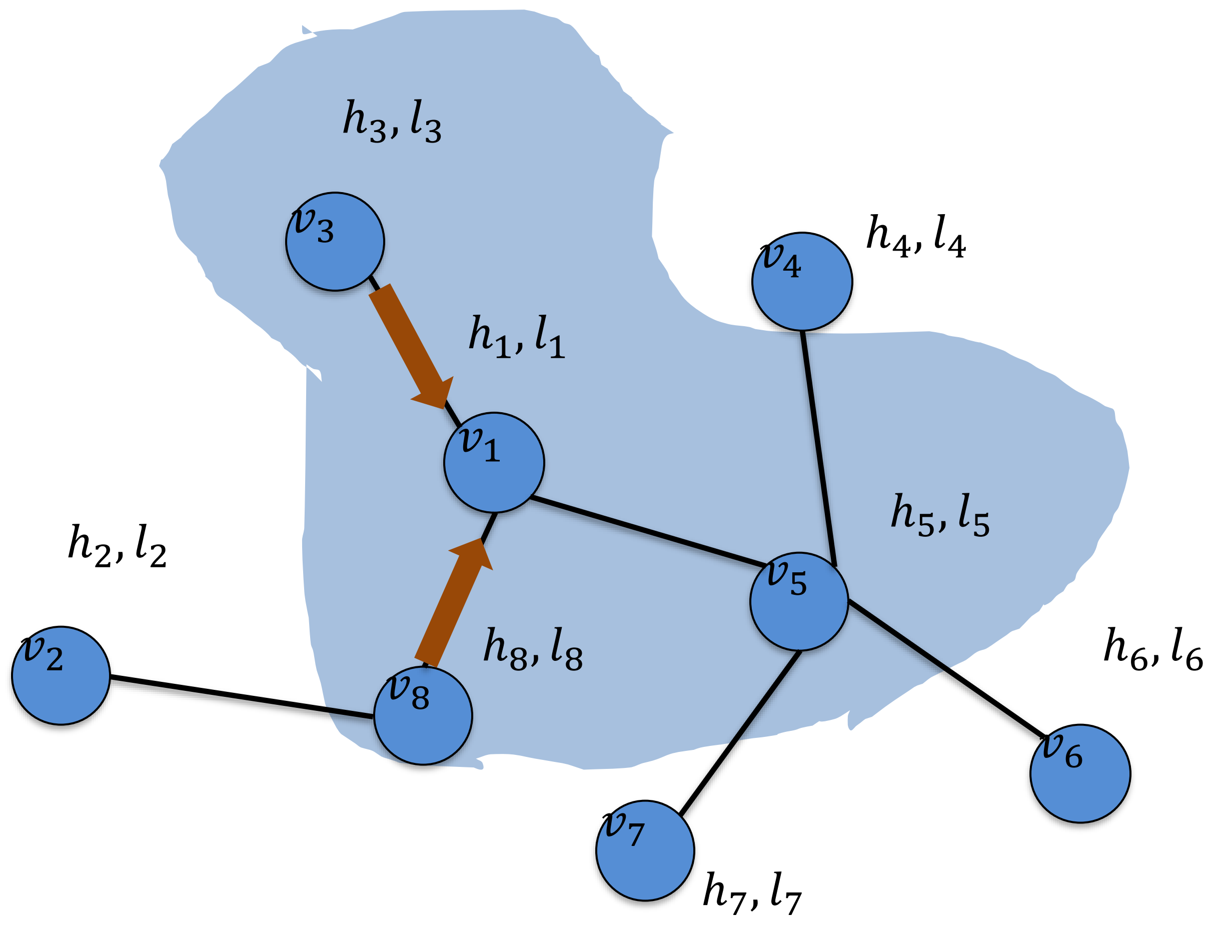
- $C[i,j] = 0$ for $v_j \notin N(v_i) \cup \{v_i\}$

Hence,



$$\mathbf{F}_{out}[i,:] = \underbrace{\sum_{v_j \in \mathcal{N}(v_i)\cup\{v_i\}}}_{\text{Aggregation}} C[i,j]\underbrace{\mathbf{F}_{in}[j,:]\Theta}_{\text{Feature transformation}}$$

Filter in GraphSage



Neighbor Sampling

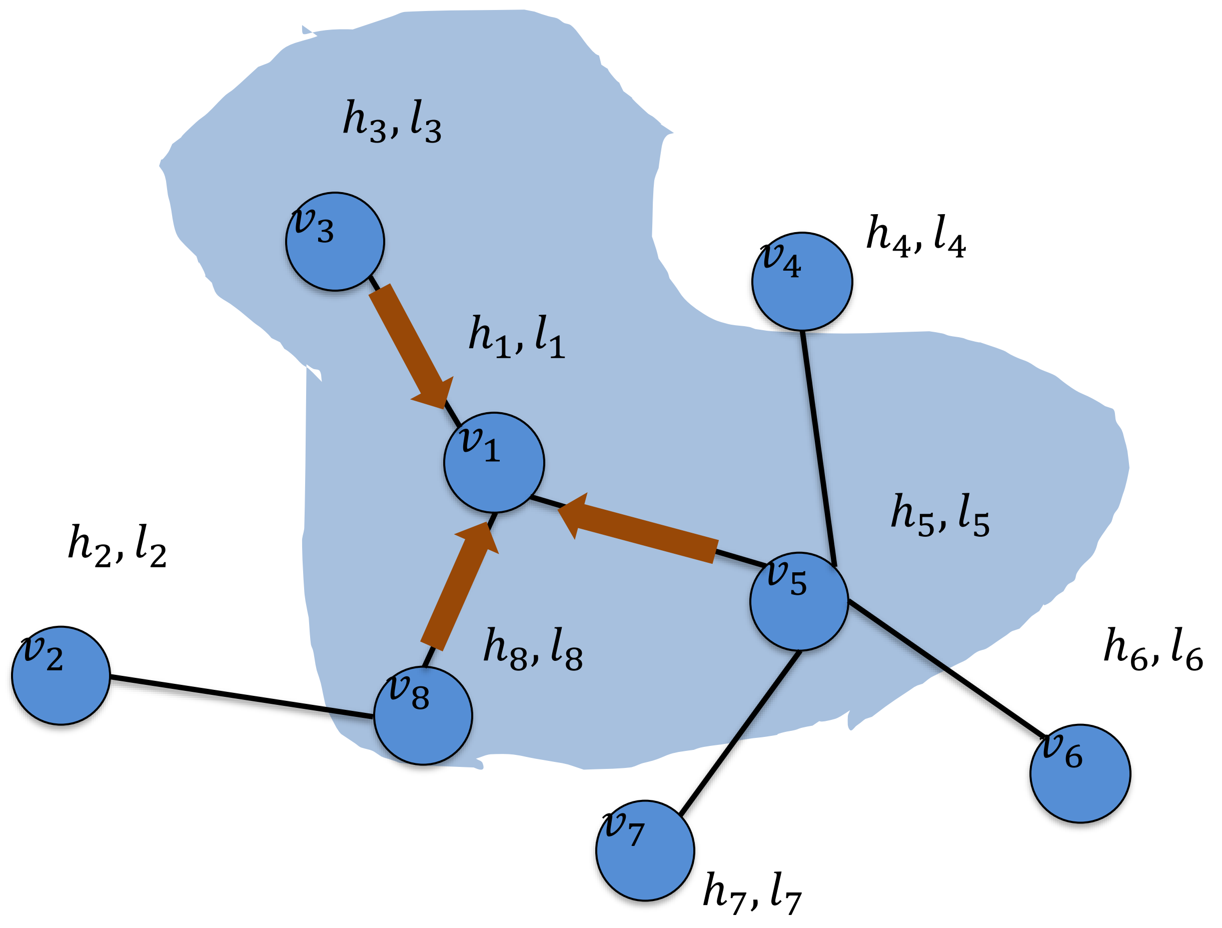
$$\mathcal{N}(v_i) \rightarrow \mathcal{N}_s(v_i)$$

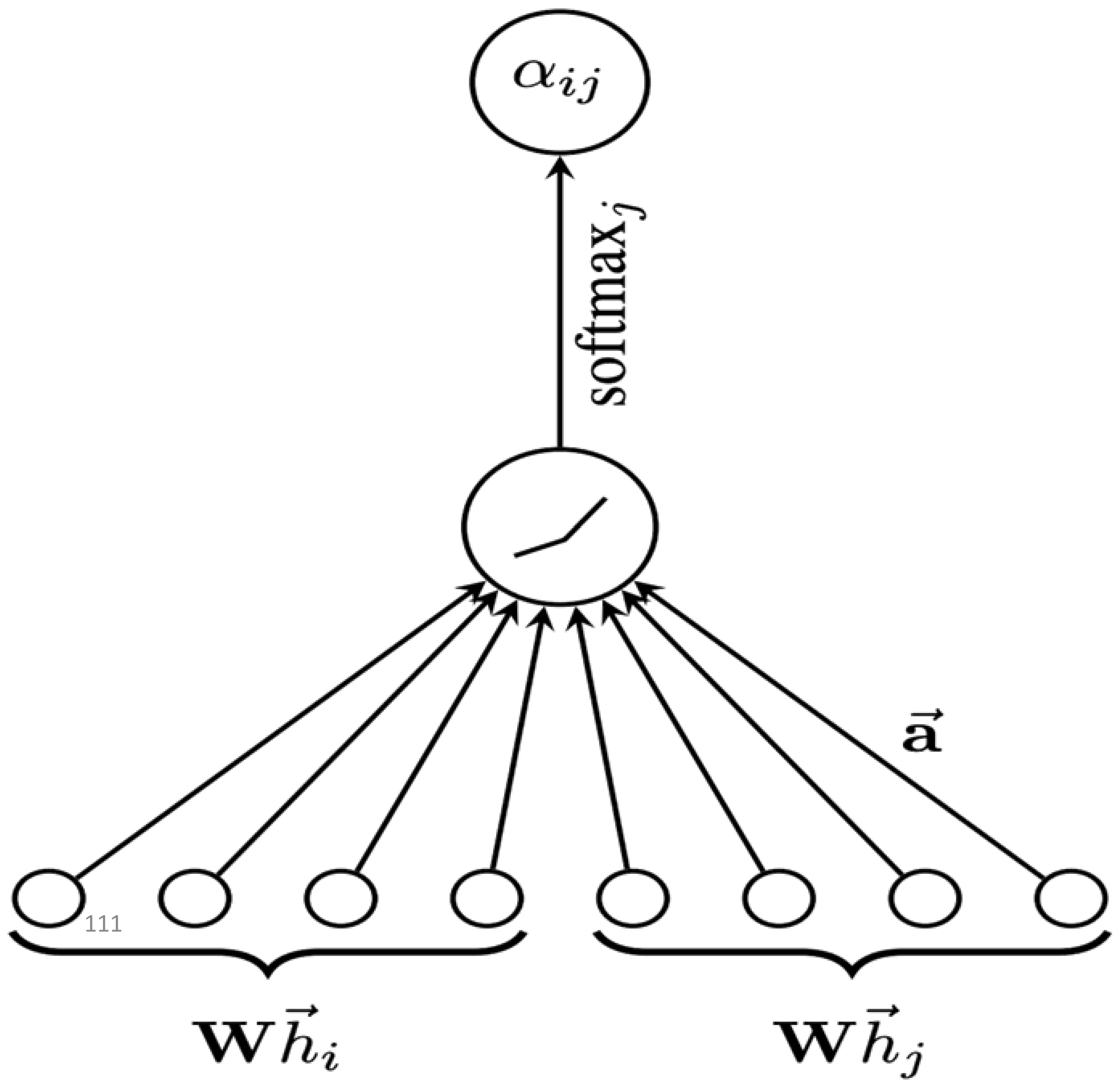
Aggregation

$$\mathbf{h}^{(k+1)}_{\mathcal{N}_s(v_i)} = \underline{AGG}(\{\mathbf{h}^{(k)}_i, v_j \in \mathcal{N}_s(v_i)\})$$

$$\mathbf{h}^{(k+1)}_i = \sigma(\Theta[\mathbf{h}^{(k)}_i, \mathbf{h}^{(k+1)}_{\mathcal{N}_s(v_i)}])$$

Filter in GAT

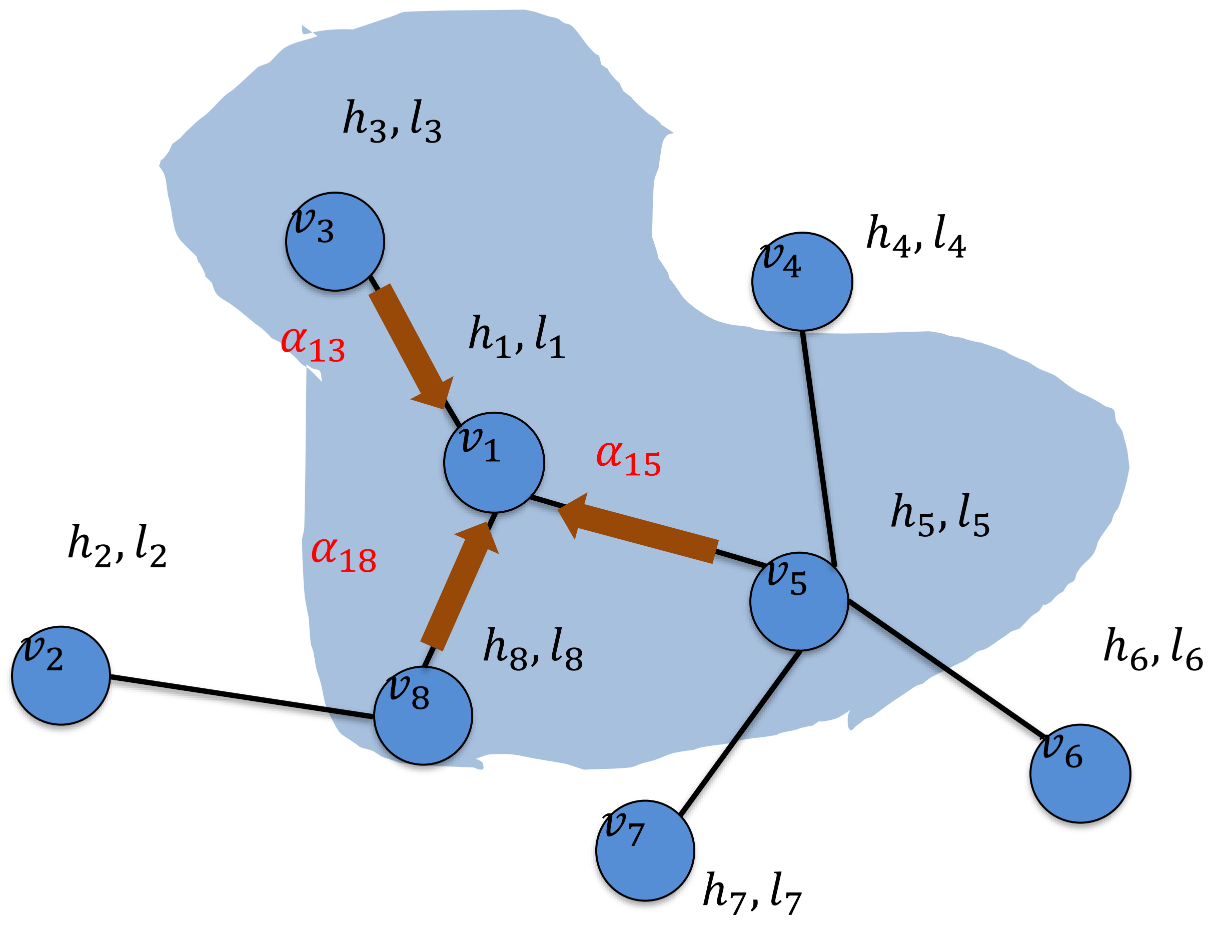


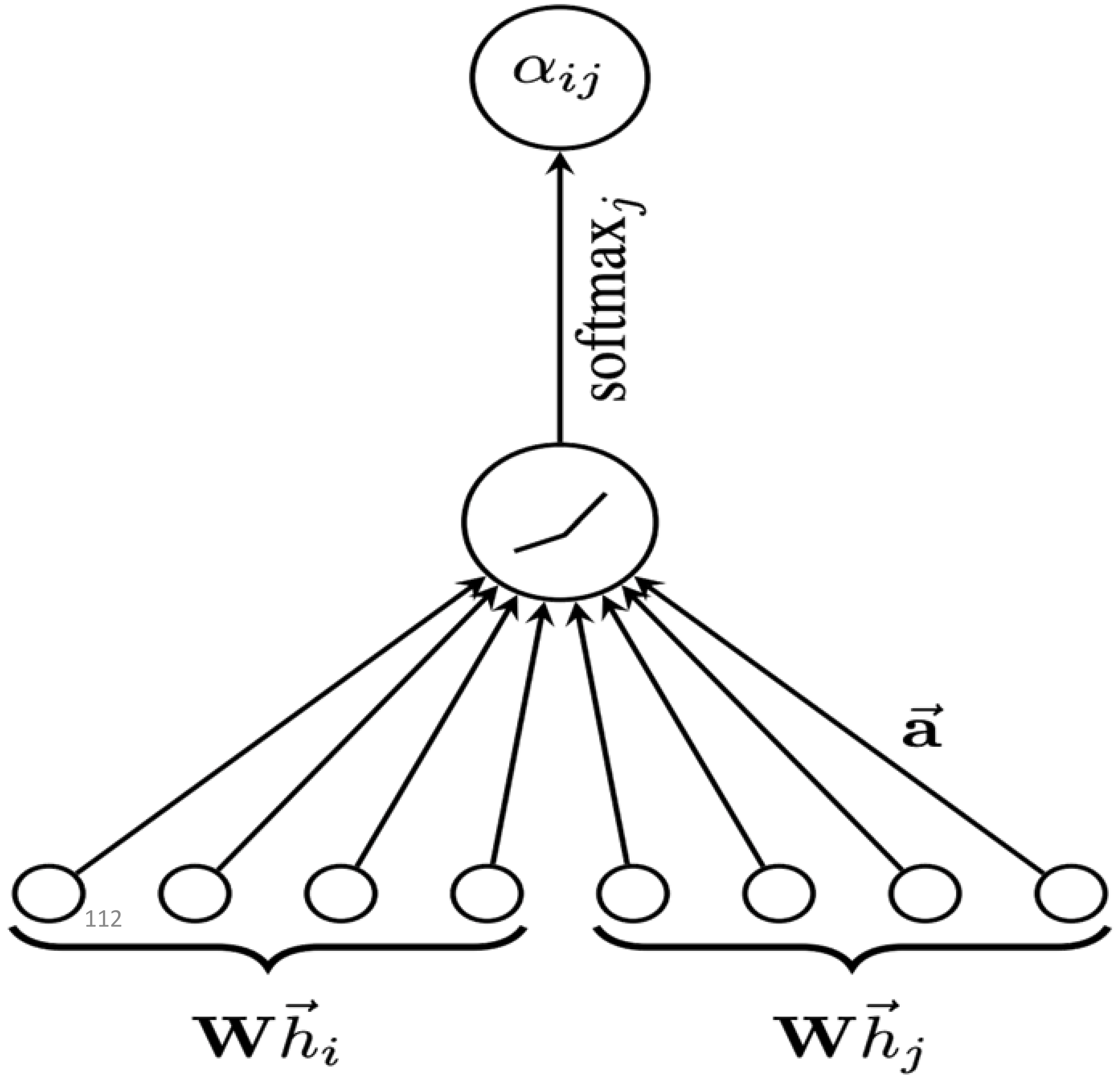


$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)}$$

Graph Attention Networks. ICLR 2018.

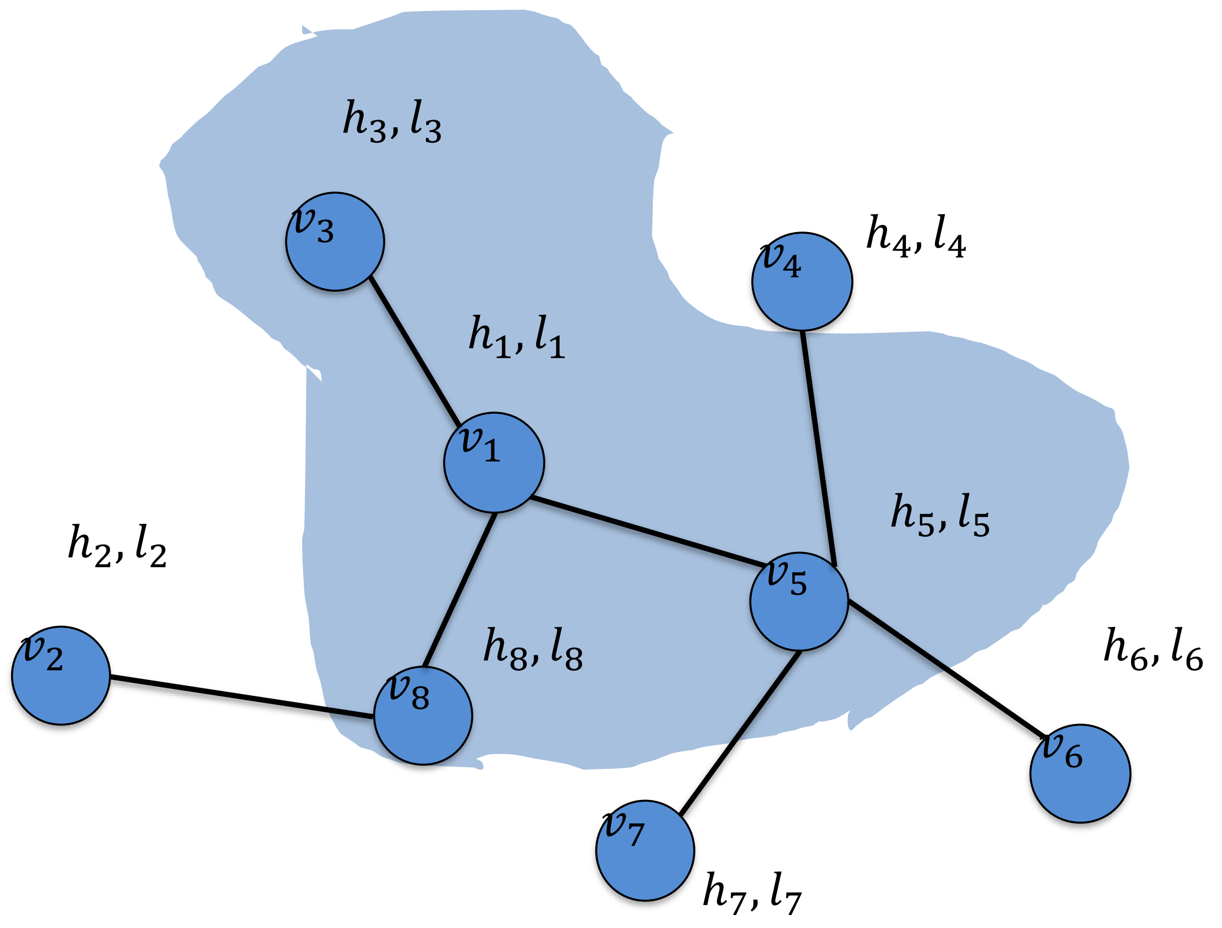
Filter in GAT





$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)}$$

Filter in MPNN



Message Passing

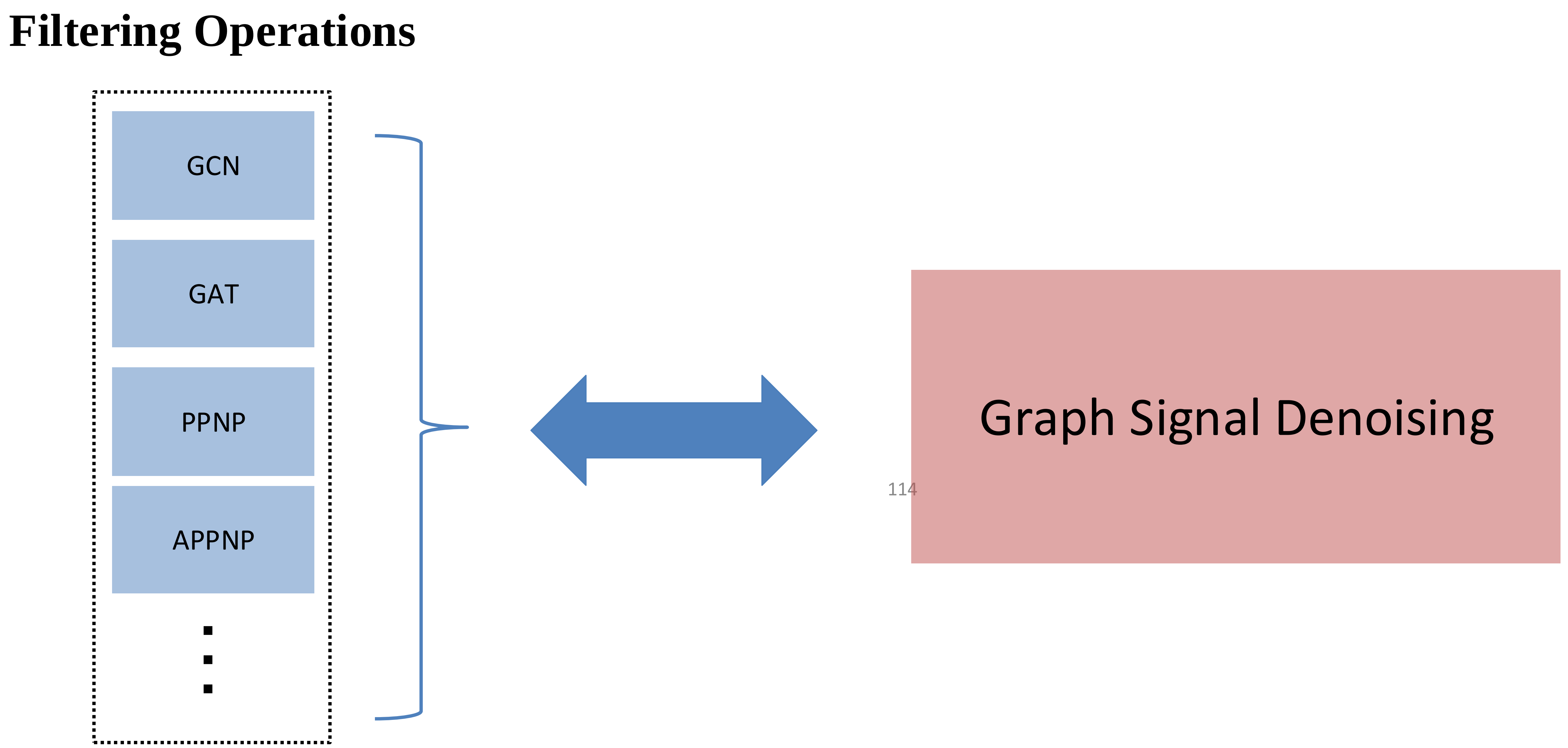
$$m_i^{(k+1)} = \sum_{v_j \in N(v_i)} M_k\left(h_i^{(k)}, h_j^{(k)}, e_{ij}\right)$$

Feature Updating

$$h_i^{(k+1)} = U_k\left(h_i^{(k)}, m_i^{(k+1)}\right)$$

$M_k(.)$ and $U_k(.)$ are functions to be designed

UGNN: A Unified Framework



$$\arg\min_{\mathbf{X}_f} \mathcal{L}\left(\mathbf{X}_f\right) = \|\mathbf{X}_f - \mathbf{X}'\|_F^2 + c \cdot \frac{1}{2} \sum_{i \in \mathcal{V}} \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \|\mathbf{X}_f[i,:] - \mathbf{X}_f[j,:]\|_2^2$$

A Unified View on Graph Neural Networks as Graph Signal Denosing. arXiv, 2020.

Example: Cora Dataset

The Cora dataset is a benchmark dataset used in graph-based machine learning research.

It represents a citation network, where:

- Nodes: Research papers
- Edges: Citation relationships (A → B means A cites B)
- Features: Bag-of-words representation of paper content
- Labels: Paper categories (7 different topics)

Real-world Relevance:

✅ Helps in developing models for document classification.

✅ Can be extended to recommendation systems, knowledge graphs, and social network analysis.

Key Statistics of the Cora Dataset:

- Number of Nodes: 2,708 (papers)
- Number of Edges: 5,429 (citations)
- Number of Features: 1,433 (word-based features)
- Number of Classes: 7 (e.g., AI, databases, NLP, etc.)

GNN Node Classification

Goal: Compare different GNN models for node classification on the Cora dataset.

We explore a variety of spatial and spectral GNN architectures:

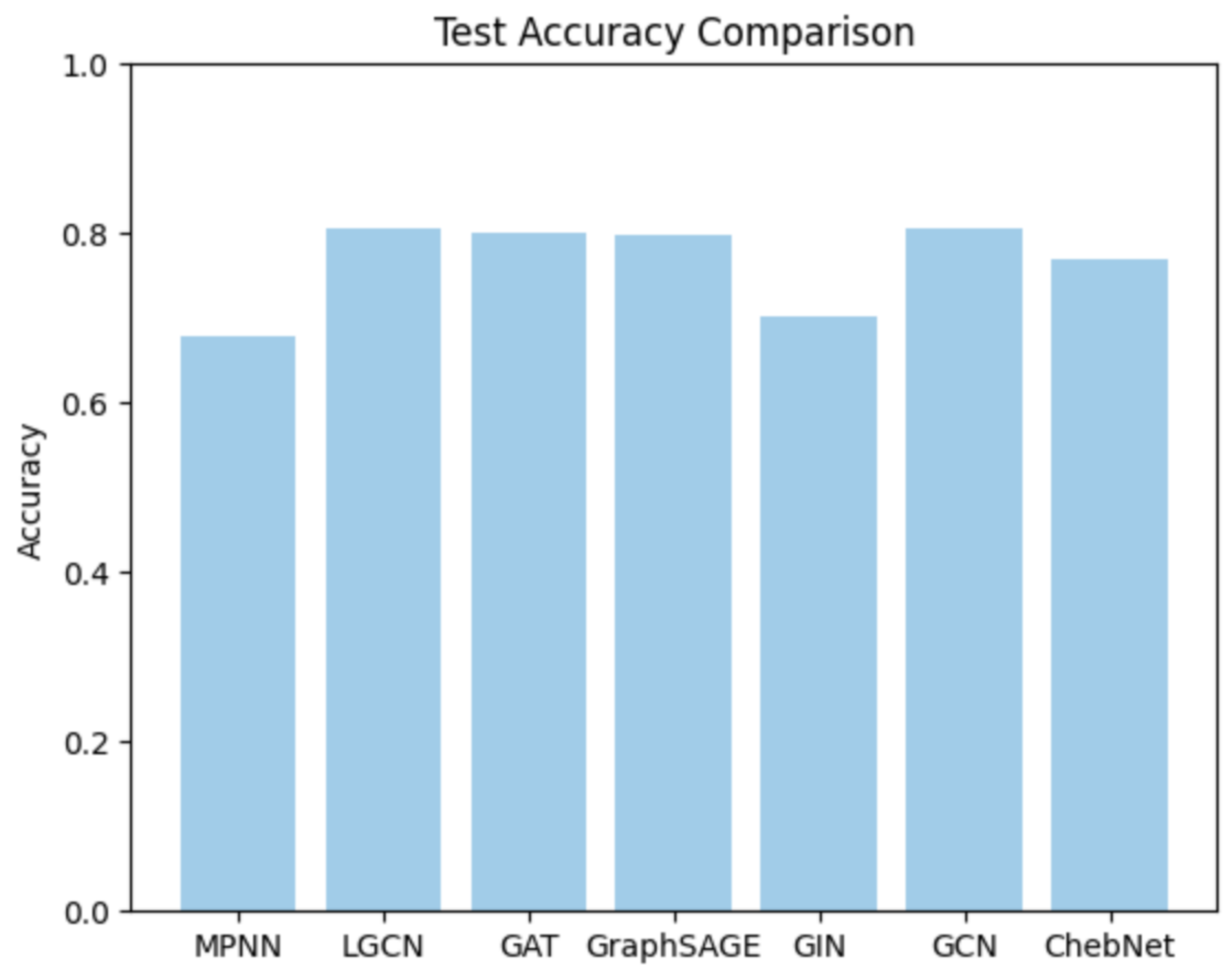
1. **Spatial-based Models:**
   - ✅ MPNN (Message Passing Neural Network)
   - ✅ LGCN (Learnable GCN)
   - ✅ GAT (Graph Attention Network)
   - ✅ GraphSAGE (Sample-based Aggregation GNN)
   - ✅ GIN (Graph Isomorphism Network)
2. **Spectral-based Models:**
   - ✅ GCN (Graph Convolutional Network)
   - ✅ ChebNet (Chebyshev Polynomial-based GNN)

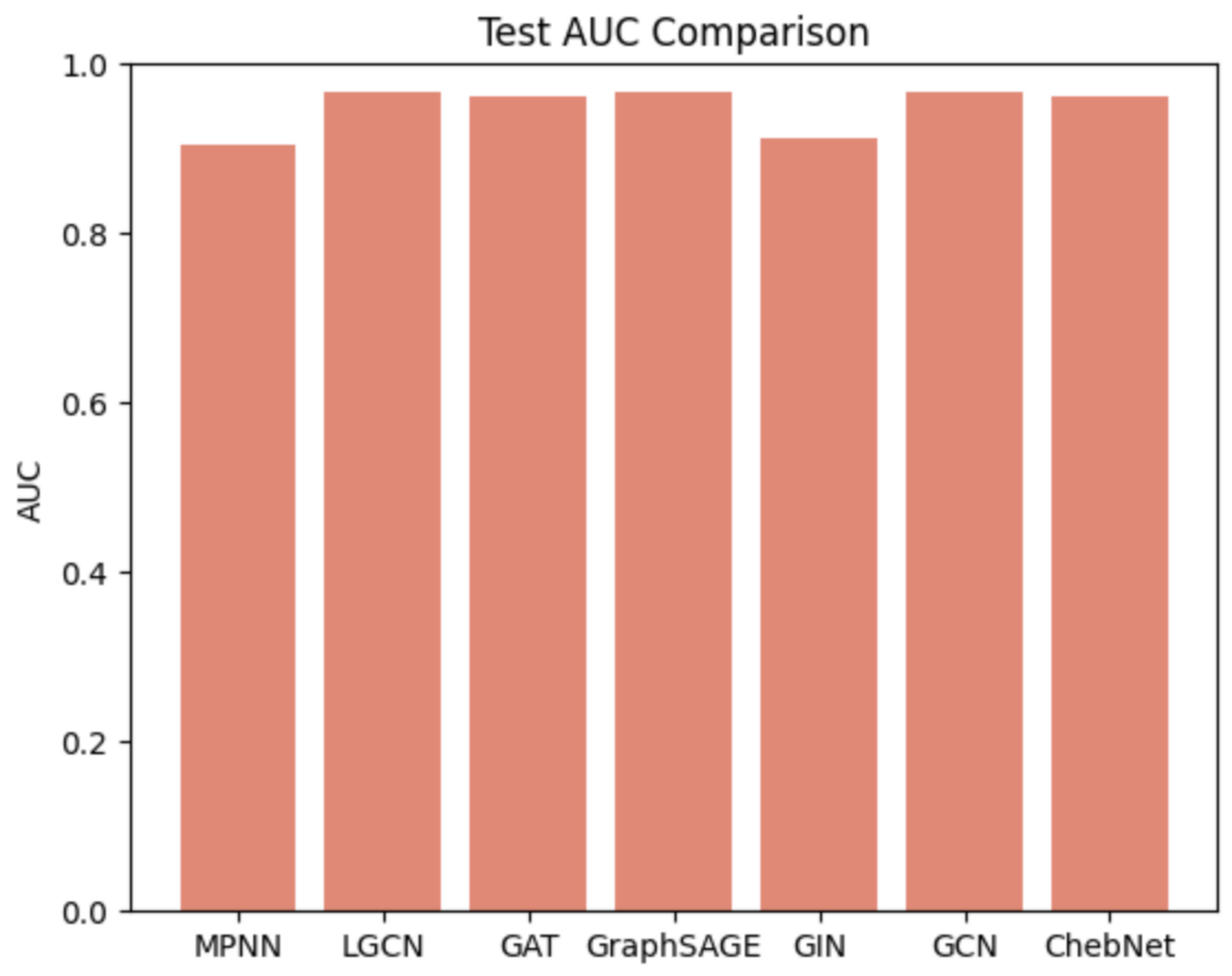
Evaluation Metrics:

1. Accuracy: Fraction of correctly classified nodes.
2. AUC Score: Multi-class AUC using one-vs-rest approach.
3. t-SNE Visualization: Low-dimensional embeddings of node representations.

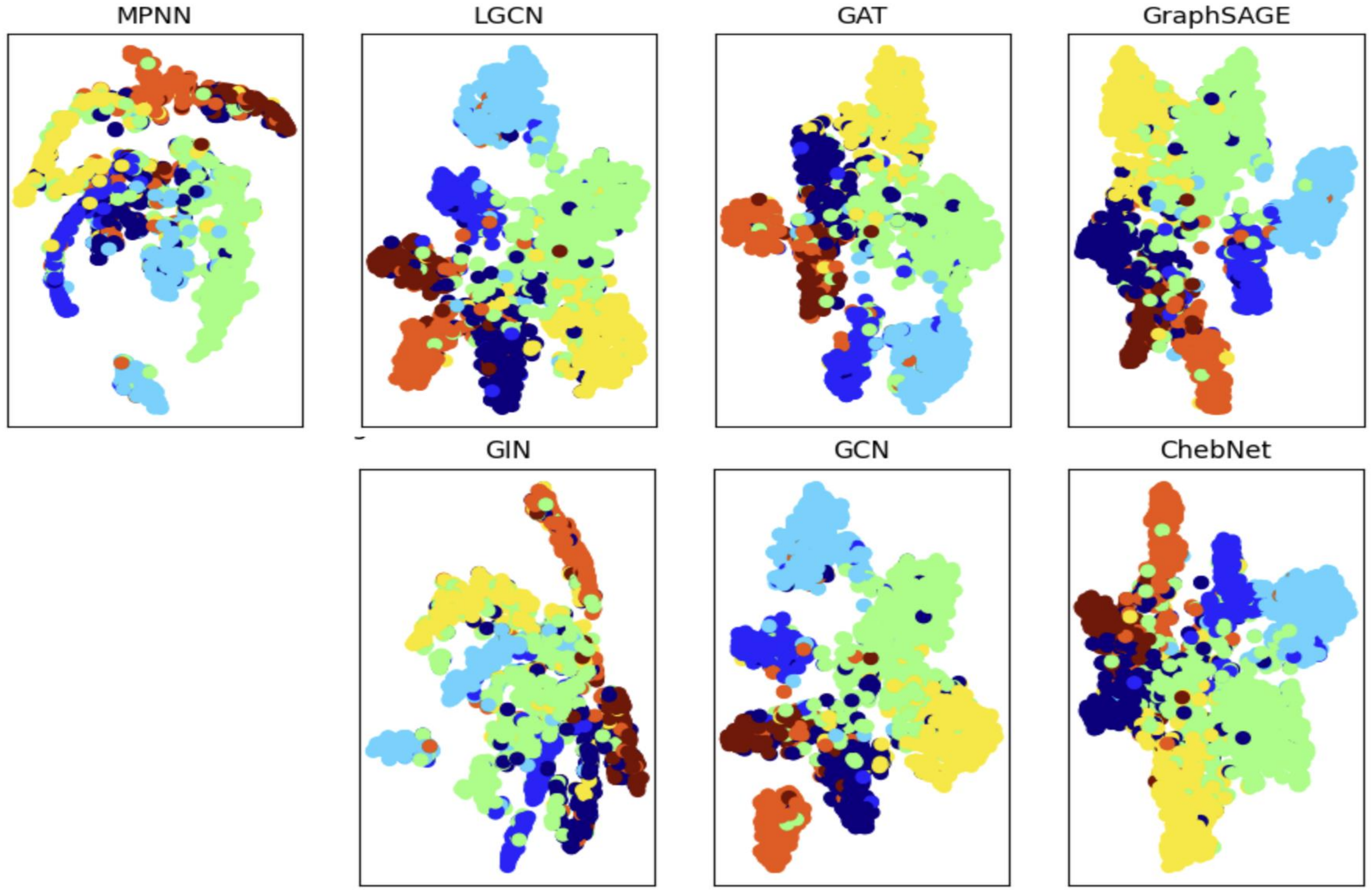
GNN Node Classification - Results

Metric	MPNN	LGCN	GAT	GraphSAGE	GIN	GCN	ChebNet
Test Accuracy	0.678	**0.807**	0.801	0.798	0.701	**0.807**	0.770
Test AUC	0.904	**0.967**	0.962	**0.967**	0.913	0.966	0.961



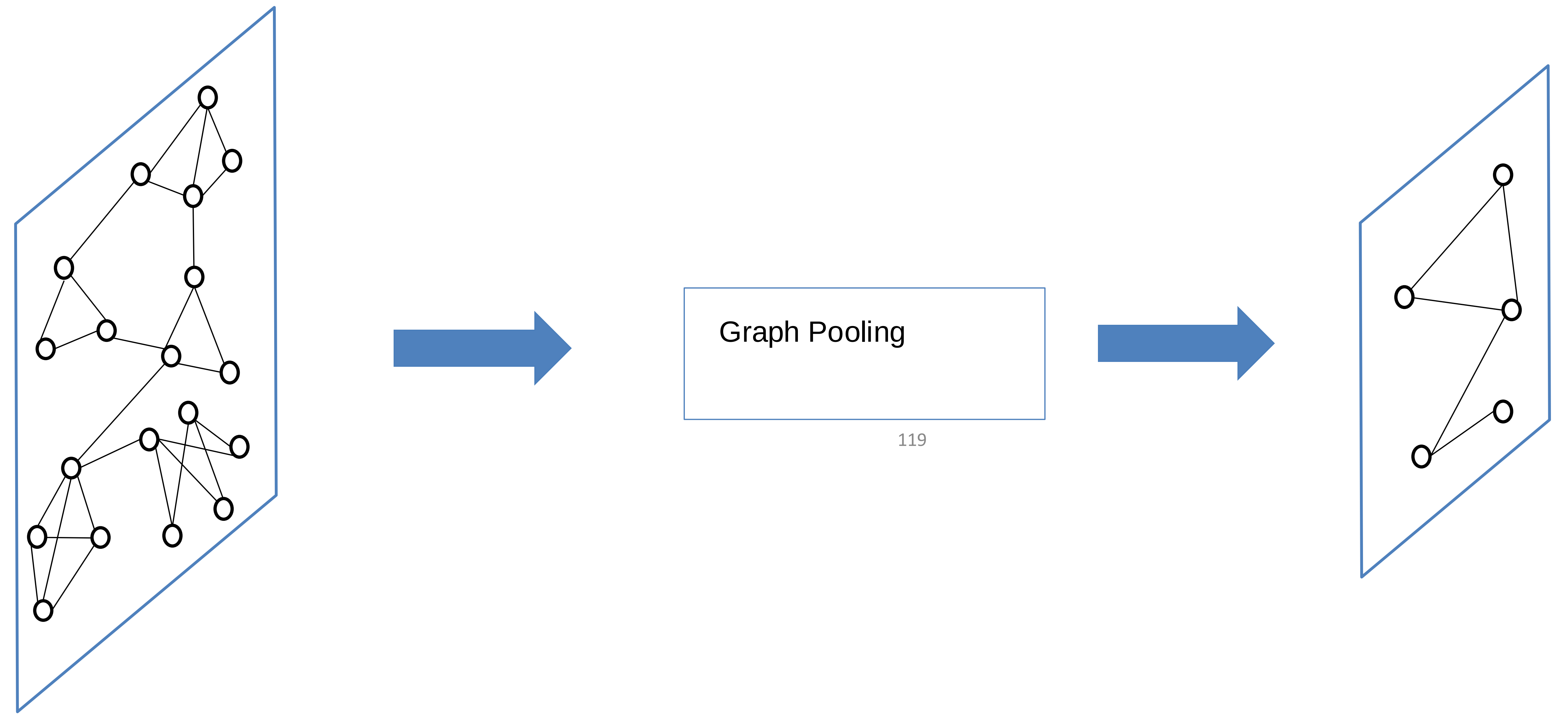


GNN Node Classification - Results



t-SNE Visualizations of Node Embeddings (color is based on true class labels)

Graph Pooling Operation



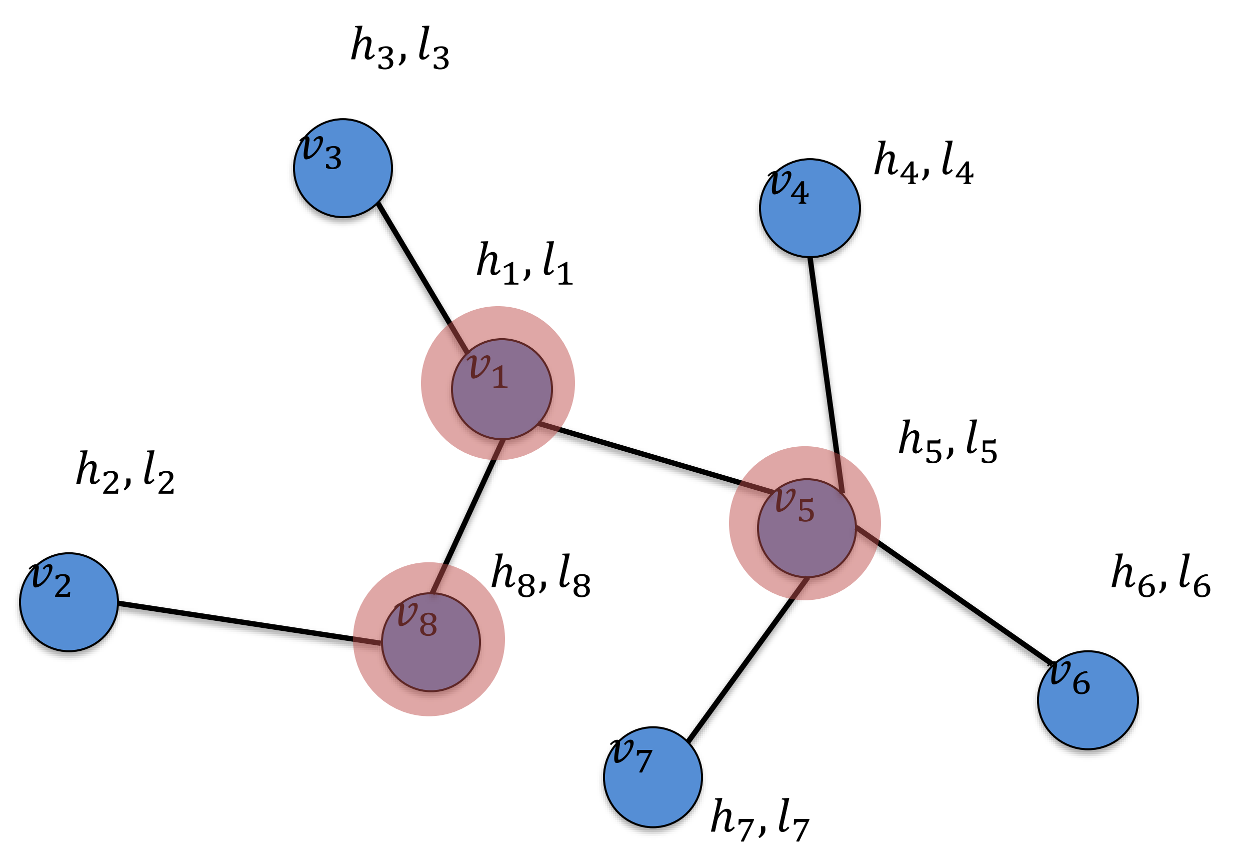
$$\mathbf{A} \in \{0,1\}^{n \times n}, \mathbf{X} \in \mathbb{R}^{n \times d}$$

$$\mathbf{A}_p \in \{0,1\}^{n_p \times n_p}, \mathbf{X}_p \in \mathbb{R}^{n_p \times d_{new}}, n_p < n$$

GILLINGS SCHOOL OF GLOBAL PUBLIC HEALTH

gPool

Downsample by selecting the most importance nodes



$$\mathbf{A} \in \{0,1\}^{n \times n}, \mathbf{H} \in \mathbb{R}^{n \times d}$$

$$\Downarrow$$

$$\mathbf{A}_p \in \{0,1\}^{n_p \times n_p}, \mathbf{H}_p \in \mathbb{R}^{n_p \times d_{new}}, n_p < n$$

Importance Measure

$$\boldsymbol{v_i} \rightarrow \boldsymbol{y_i} \qquad \boldsymbol{y_i} = \frac{\boldsymbol{h_i^T p}}{||\boldsymbol{p}||}$$

Select top the n_p nodes

$$\boldsymbol{idx} = \boldsymbol{rank}(\boldsymbol{y}, \boldsymbol{n_p})$$

Generate A_p and intermediate H_{inter}

$$\boldsymbol{A_p} = \boldsymbol{A}[\boldsymbol{idx}, \boldsymbol{idx}]$$

$$\boldsymbol{H_{inter}} = \boldsymbol{H}[\boldsymbol{idx}, :]$$

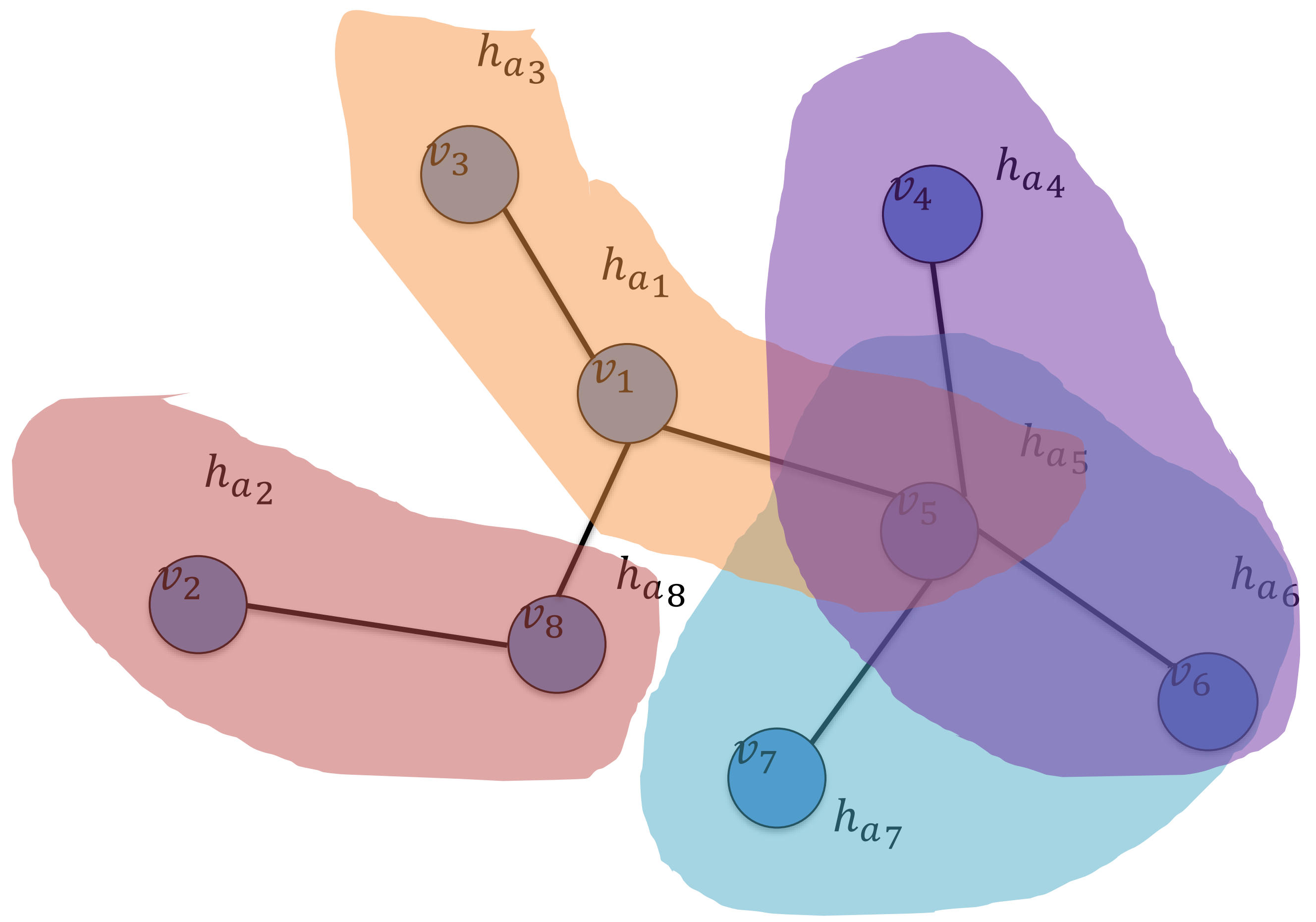
Generate H_p

$$\tilde{\boldsymbol{y}} = \boldsymbol{sigmoid}(\boldsymbol{y}[\boldsymbol{idx}])$$

$$H_p = H_{inter} \odot \tilde{y}$$

DiffPool

Downsample by clustering the nodes using GNN



$$\mathbf{A} \in \{0,1\}^{n\times n}, \mathbf{H} \in \mathbb{R}^{n\times d}$$

$$\Downarrow$$

$$\mathbf{A}_p \in \{0,1\}^{n_p\times n_p}, \mathbf{H}_p \in \mathbb{R}^{n_p\times d_{new}}, n_p < n$$

2 filters

Filter1:
Generate a soft-assign matrix

$$\mathbf{A} \in \{0,1\}^{n\times n}, \mathbf{H} \in \mathbb{R}^{n\times d}$$

$$\Downarrow$$

$$\mathbf{A} \in \{0,1\}^{n\times n}, \mathbf{H}_a \in \mathbb{R}^{n\times n_p}$$

Filter2:
Generate new features

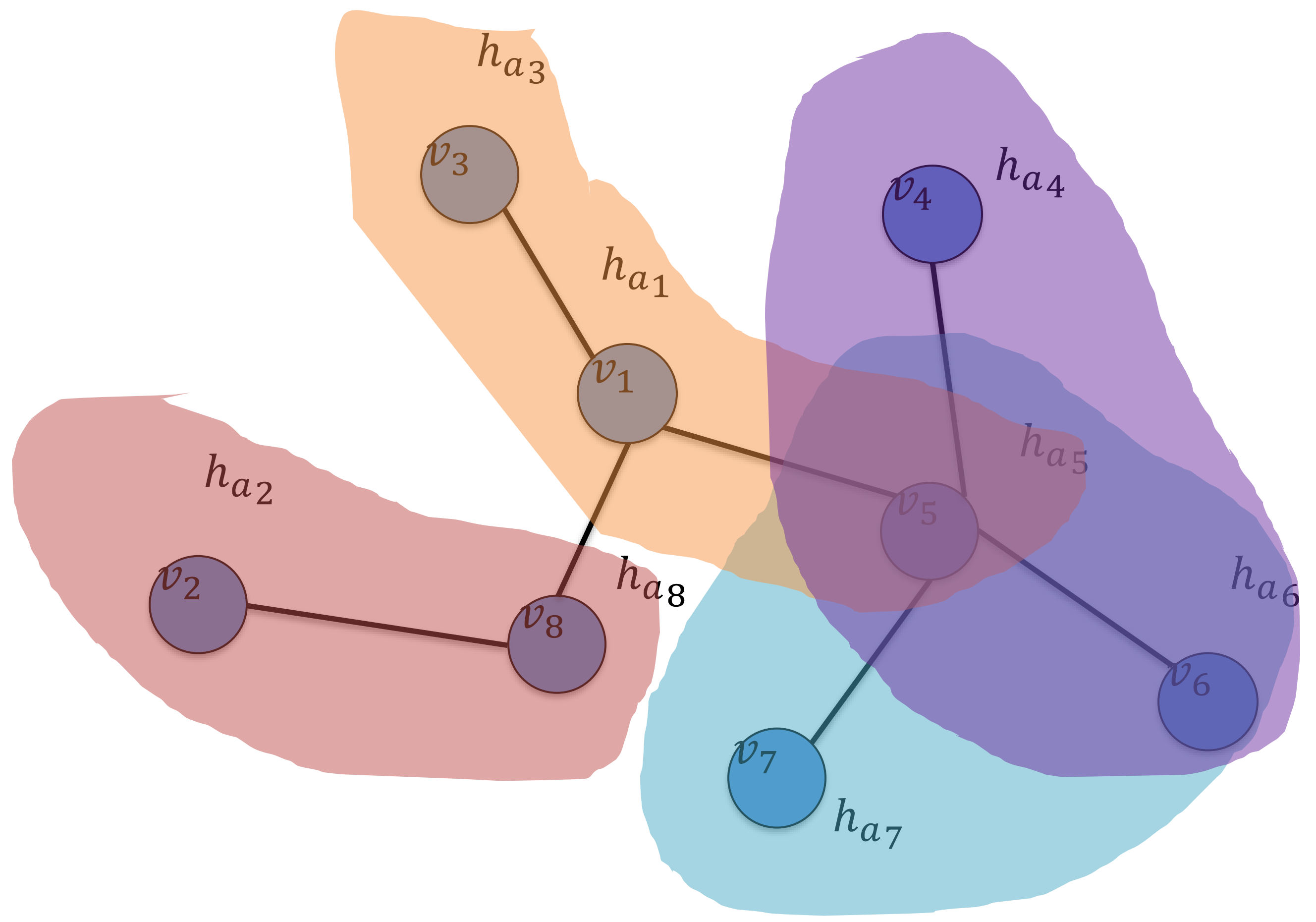
$$\mathbf{A} \in \{0,1\}^{n\times n}, \mathbf{H} \in \mathbb{R}^{n\times d}$$

$$\Downarrow$$

$$\mathbf{A} \in \{0,1\}^{n\times n}, \mathbf{H}_f \in \mathbb{R}^{n\times d_{new}}$$

DiffPool

Downsample by clustering the nodes using GNN



$$\mathbf{A} \in \{0,1\}^{n\times n}, \mathbf{H} \in \mathbb{R}^{n\times d}$$

$$\downarrow$$

$$\mathbf{A}_p \in \{0,1\}^{n_p\times n_p}, \mathbf{H}_p \in \mathbb{R}^{n_p\times d_{new}}, n_p < n$$

Generated soft-assign matrix $\mathbf{H}_a \in \mathbb{R}^{n\times n_p}$

Generated new features $\mathbf{H}_f \in \mathbb{R}^{n\times d_{new}}$

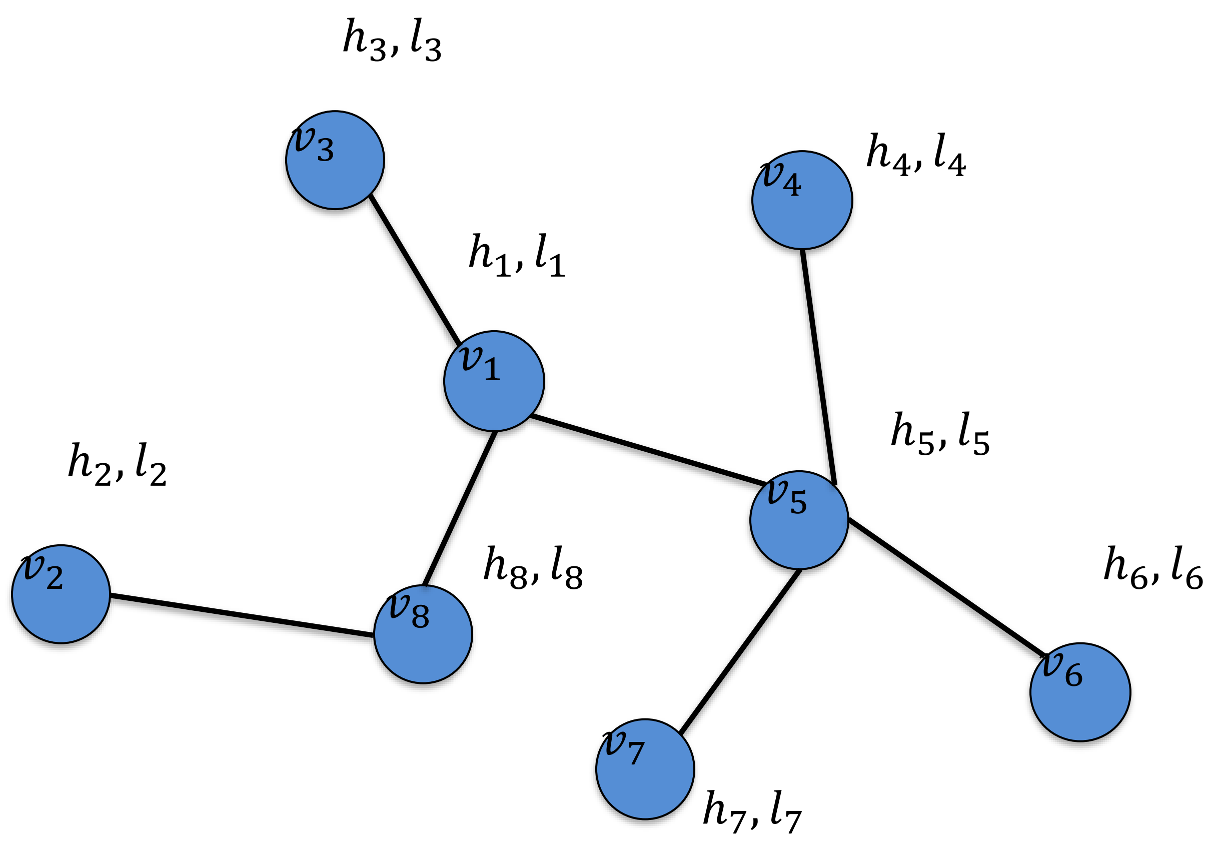
Generate A_p

$$\mathbf{A}_p = \mathbf{H}_a^T \mathbf{A} \mathbf{H}_a$$

Generate H_p

$$\mathbf{H}_p = \mathbf{H}_a^T \mathbf{H}_f$$

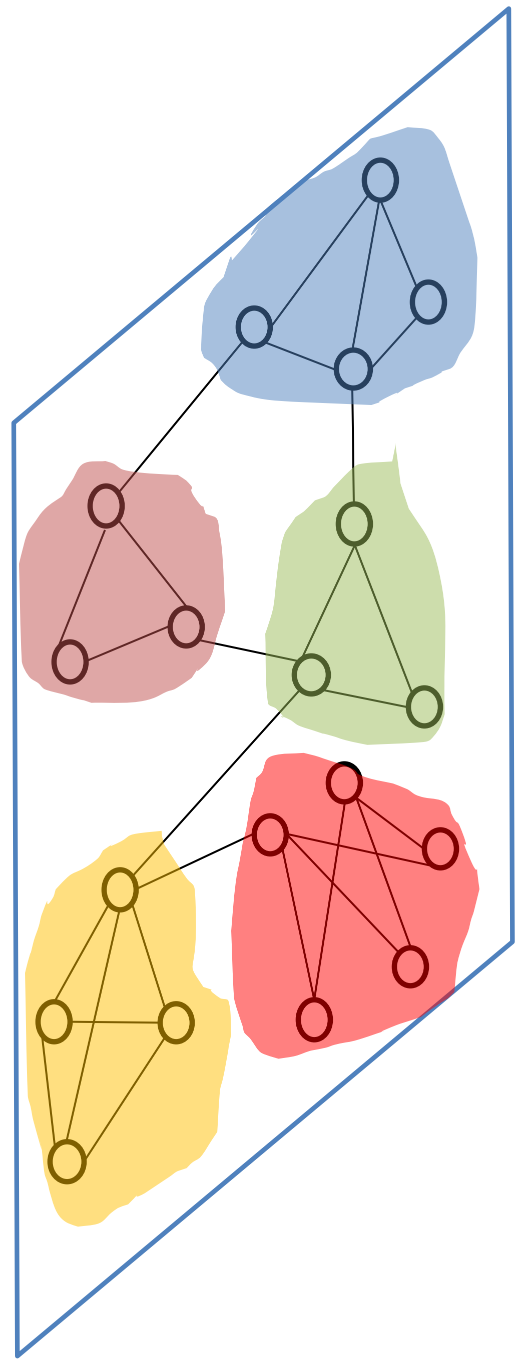
Eigenpooling



$$\mathbf{A} \in \{0,1\}^{n \times n}, \mathbf{H} \in \mathbb{R}^{n \times d}$$

$$\downarrow$$

$$\mathbf{A}_p \in \{0,1\}^{n_p \times n_p}, \mathbf{H}_p \in \mathbb{R}^{n_p \times d_{new}}, n_p < n$$



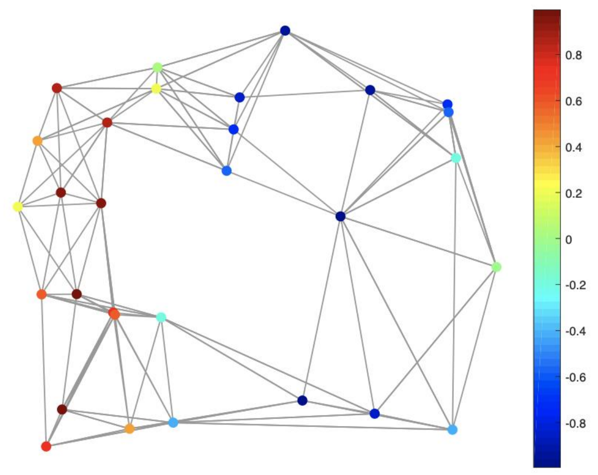
Learn A_p using clustering methods

Focus on learning better H_p

Capture both feature and graph structure

Going Back to Graph Spectral Theory

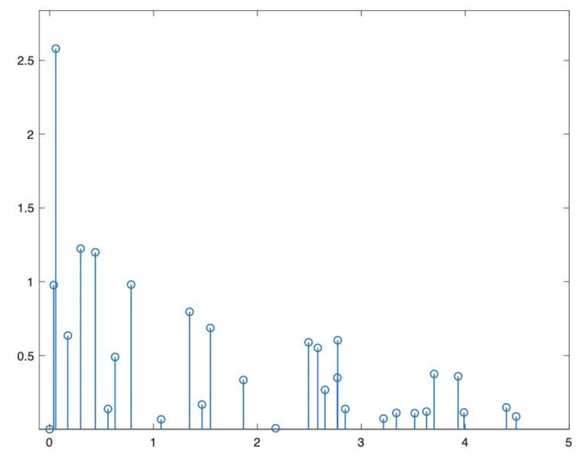
Recall:



Spatial domain: $\boldsymbol{f}$

$$\hat{\boldsymbol{f}} = \boldsymbol{U}^T \boldsymbol{f}$$

Decompose signal f

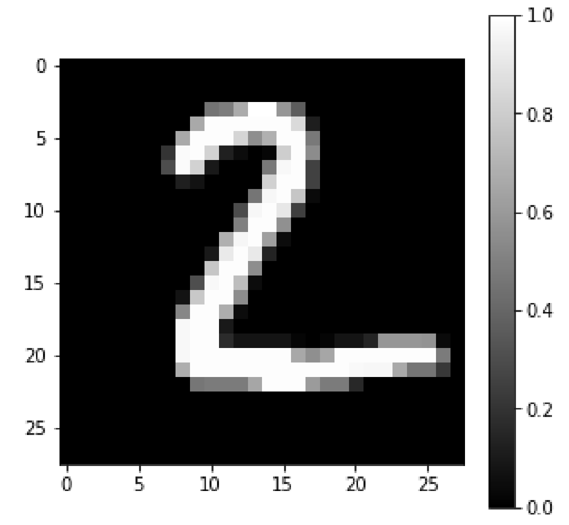


Spectral domain: $\hat{\boldsymbol{f}}$

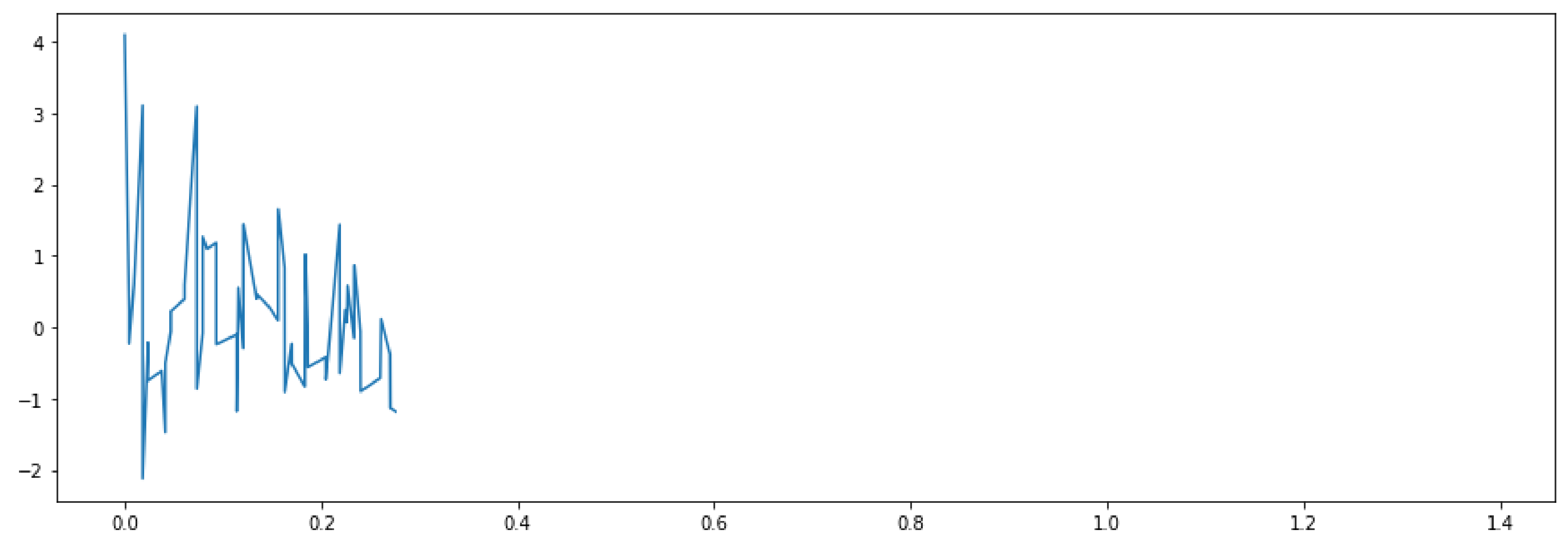
$$\mathbf{f} = \hat{f}_0 u_0 + \hat{f}_1 u_1 + \ldots \hat{f}_{N-1} u_{N-1}$$

Going Back to Graph Spectral Theory

Do we need all the coefficients to reconstruct a “good” signal?

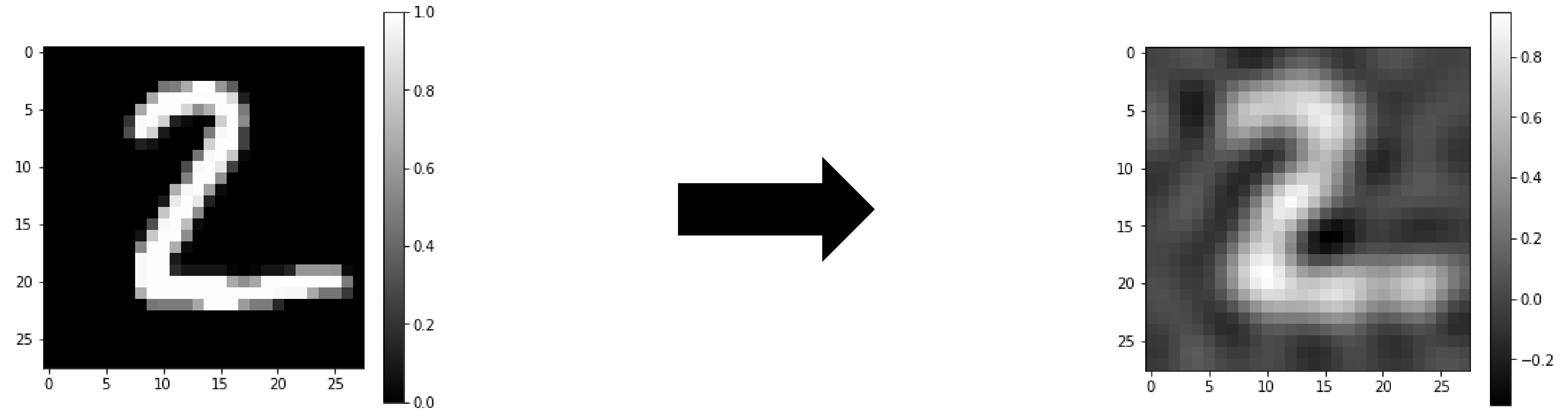


125

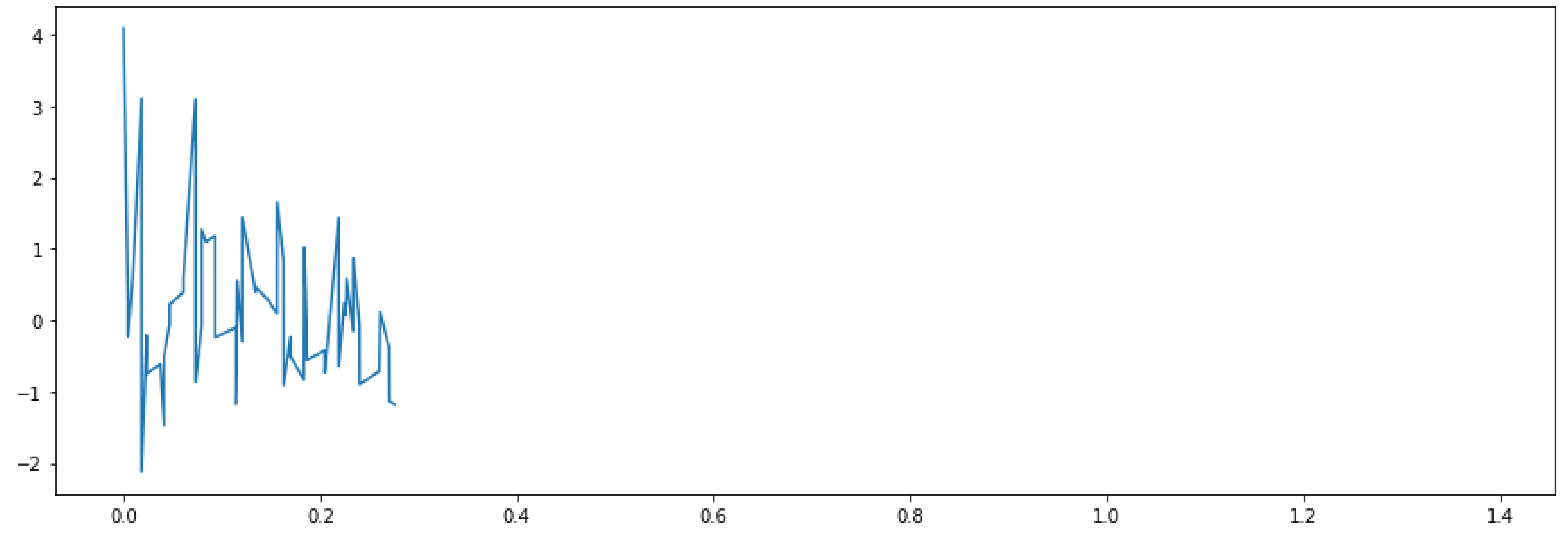


Going Back to Graph Spectral Theory

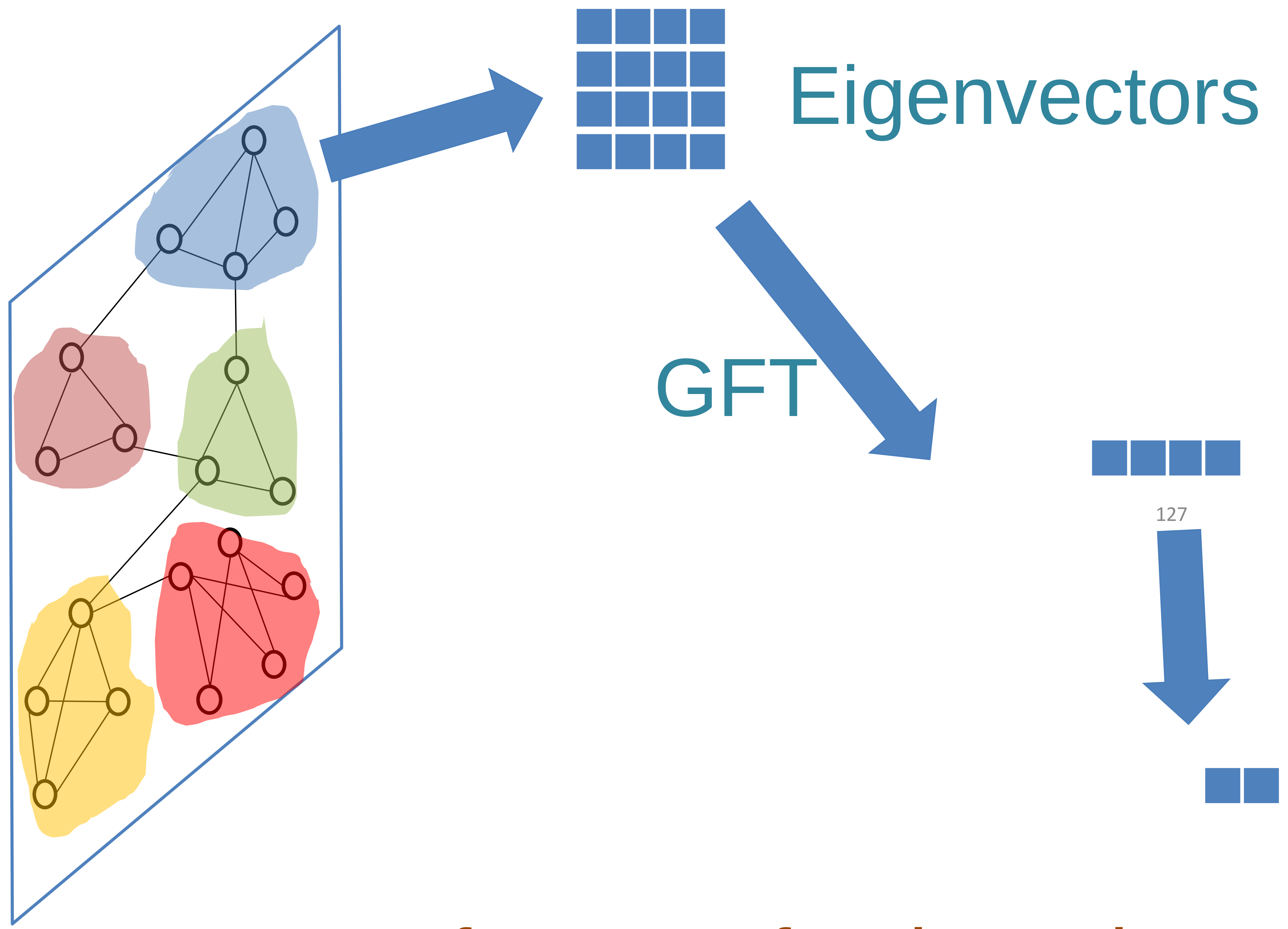
Do we need all the coefficients to reconstruct a "good" signal?



126



Eigenpooling: Truncated Fourier Coefficients



Eigenvectors (Fourier Modes) of the subgrap

Fourier coefficients

Truncated Fourier coefficient

New features for the subgraph (a node in the smaller graph

MUTAG Dataset

MUTAG is a dataset of **molecular graphs**, where:

- ✅ Nodes: Atoms in molecules
- ✅ Edges: Chemical bonds between atoms
- ✅ Labels: Mutagenic vs. non-mutagenic molecules (Binary classification)

Key Statistics of the MUTAG Dataset:

- Number of Graphs: 188
- Average Nodes per Graph: ~17
- Average Edges per Graph: ~19.8
- Number of Features: Based on atom properties
- Number of Classes: 2 (Mutagenic vs. Non-Mutagenic)

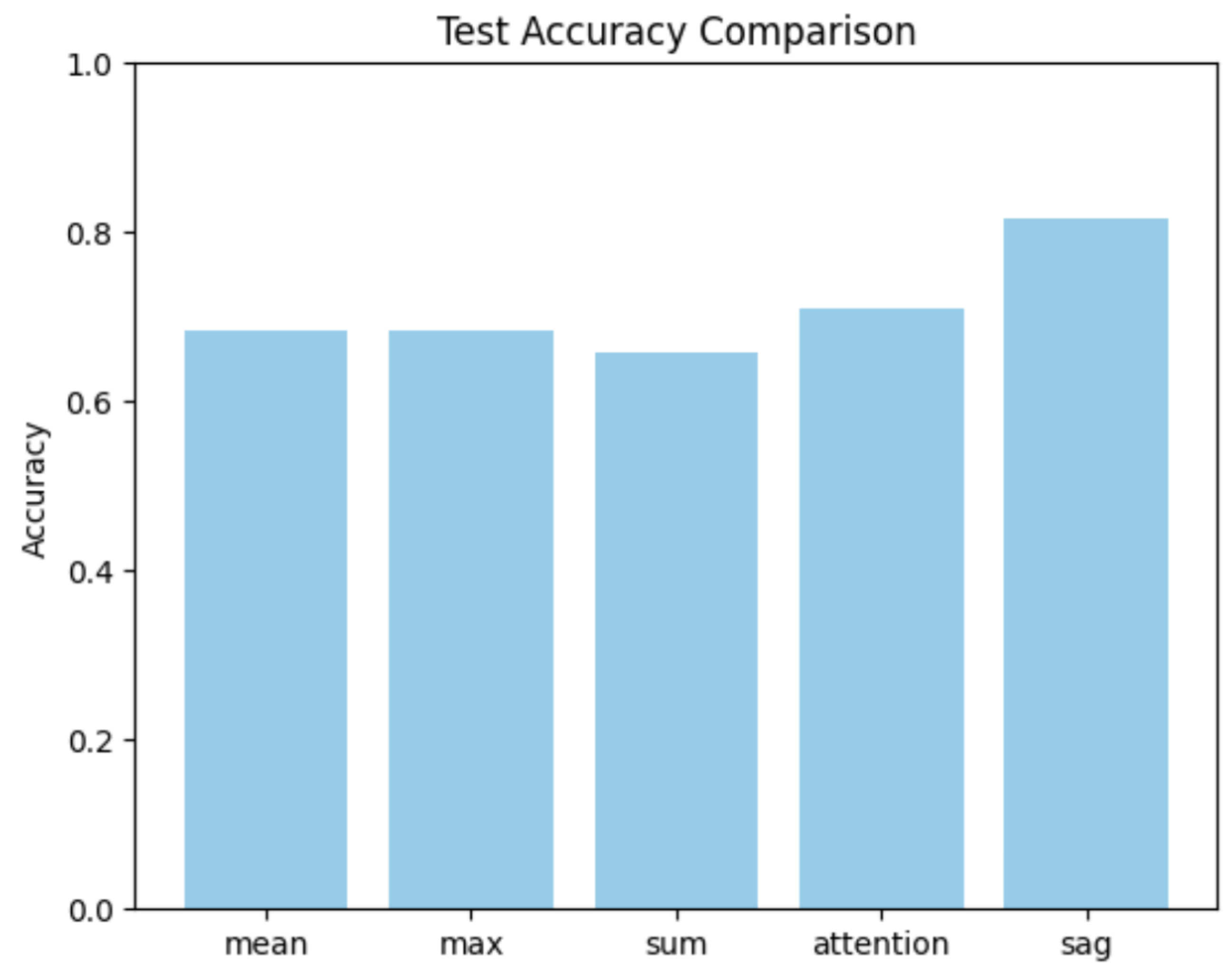
GNN Graph Pooling and Classification

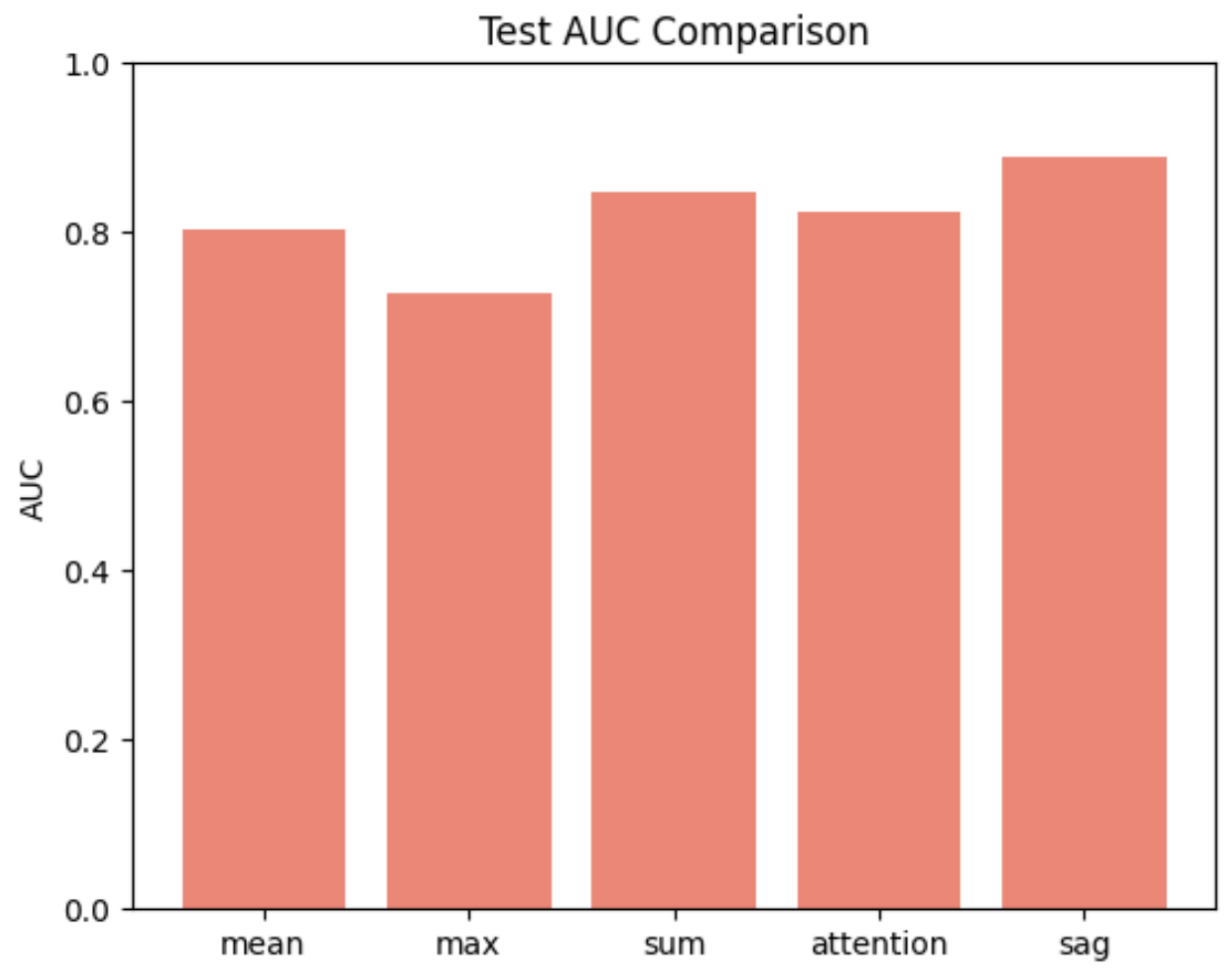
- **Goal:** Compare GNN-based graph classification models using different graph pooling techniques.
- **Model structure overview:**

  GCN → Activation → Graph Pooling (**graph embeddings**) → Fully Connected → Softmax outputs
- **Graph Pooling methods compared:**
  - ✓ Mean Pooling
  - ✓ Max Pooling
  - ✓ Sum Pooling
  - ✓ Attention Pooling
  - ✓ SAGPooling (Hierachical)
- **Evaluation Metrics:**
  - ✓ Test Accuracy
  - ✓ AUC Score
  - ✓ t-SNE Embeddings (Based on graph embeddings after pooling)
  - ✓ ROC Curves
  - ✓ Confusion Matrices

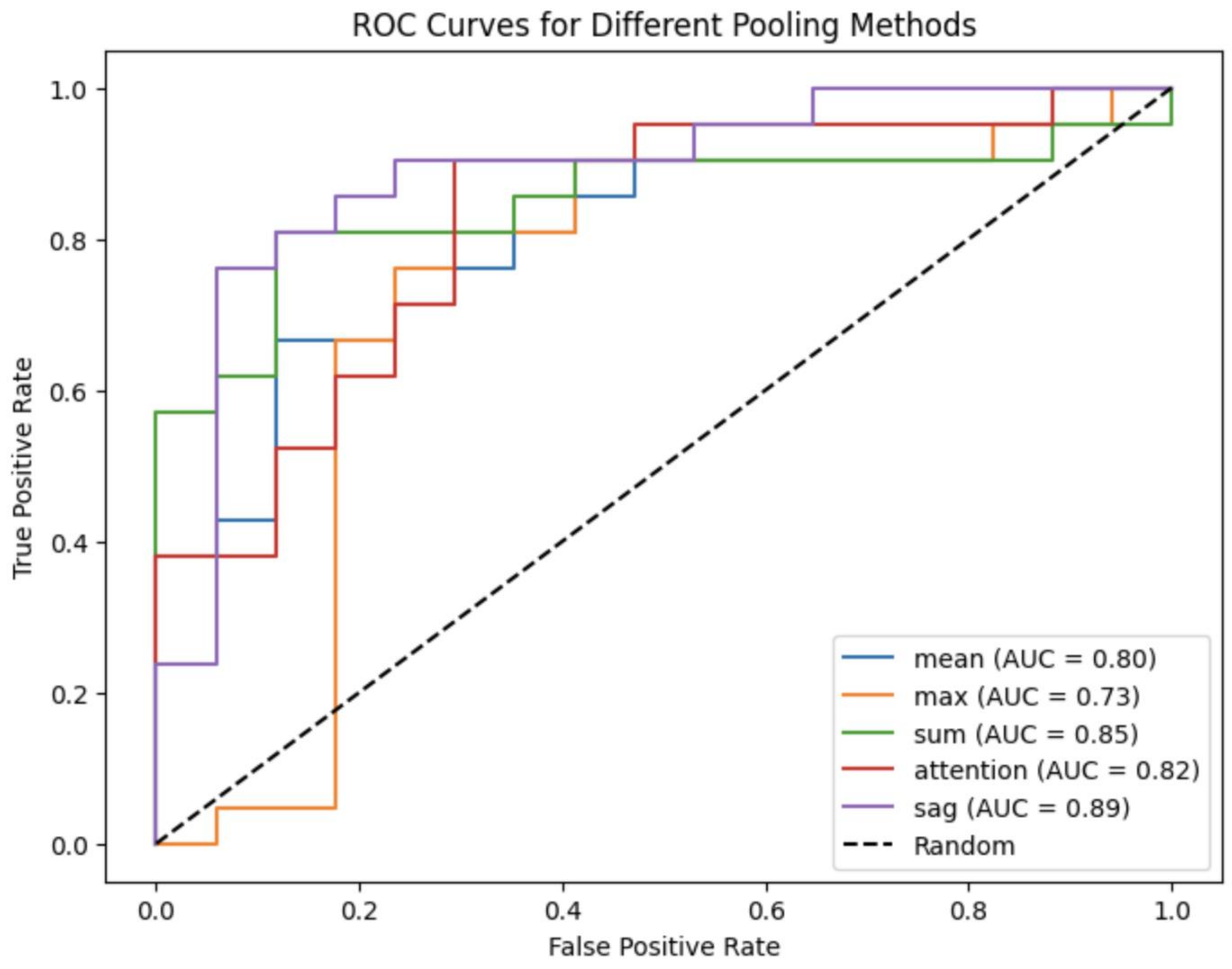
GNN Graph Pooling and Classification - Results

Pooling methods	mean	max	sum	attention	sag
Test Accuracy	0.684211	0.684211	0.657895	0.710526	**0.815789**
Test AUC	0.803922	0.725490	0.848739	0.826331	**0.885154**



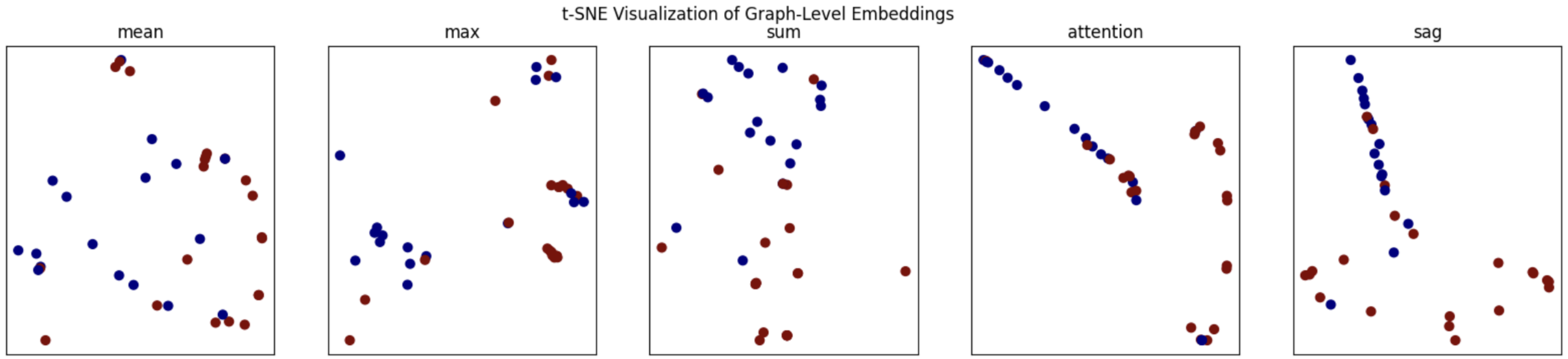


GNN Graph Pooling and Classification - Results



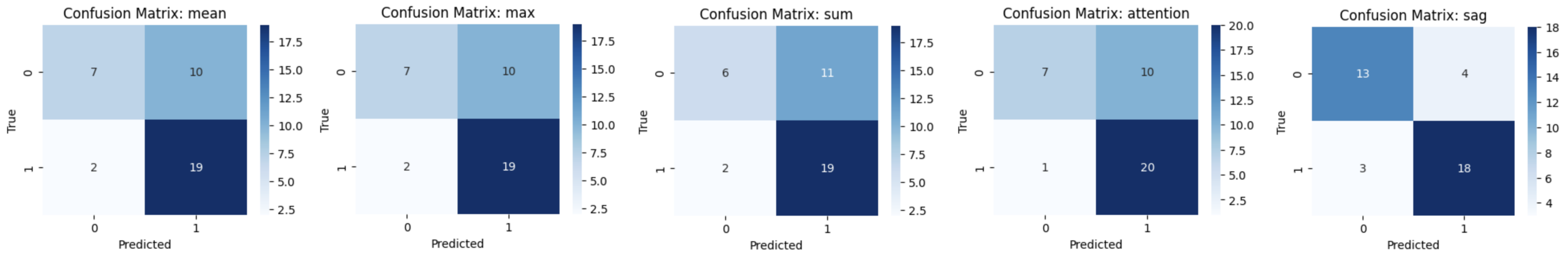
ROC Curves for Different Pooling Methods

GNN Graph Pooling and Classification - Results



t-SNE Visualizations of Graph Embeddings from Different Pooling Methods

GNN Graph Pooling and Classification - Results



Confusion Matrix for Different Pooling Methods

Content

Application to “classical” network problems

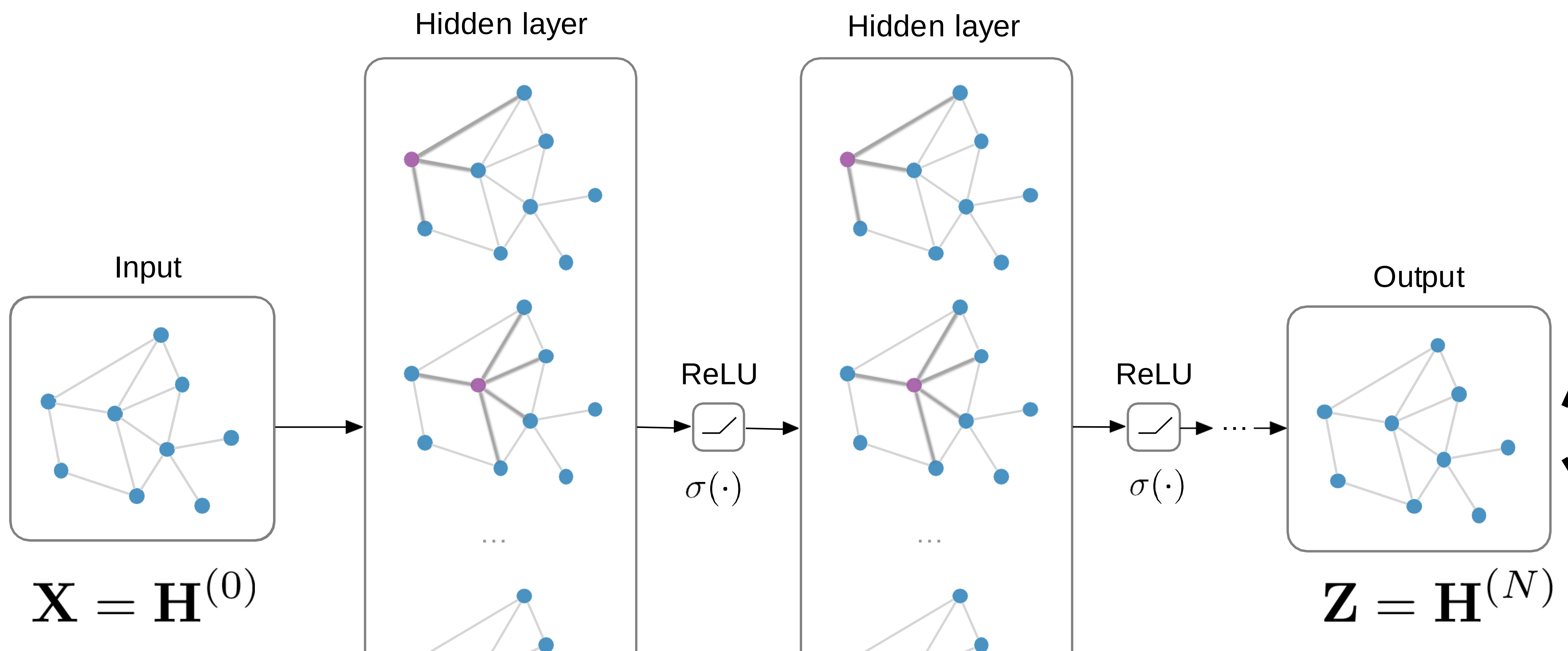
Node classification

Graph classification

Link prediction

One fits all: Classification and link prediction with GNNs/GCNs

Input: Feature matrix $\mathbf{X} \in \mathbb{R}^{N \times E}$, preprocessed adjacency matrix $\hat{\mathbf{A}}$



$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right)$$

Node classification:

$$\text{softmax}(\mathbf{z_n})$$

e.g. Kipf & Welling (ICLR 2017)

Graph classification:

e.g. Duvenaud et al. (NIPS 2015)

Link prediction:

$$p(A_{ij}) = \sigma(\mathbf{z_i}^{\mathrm{T}}\mathbf{z_j})$$

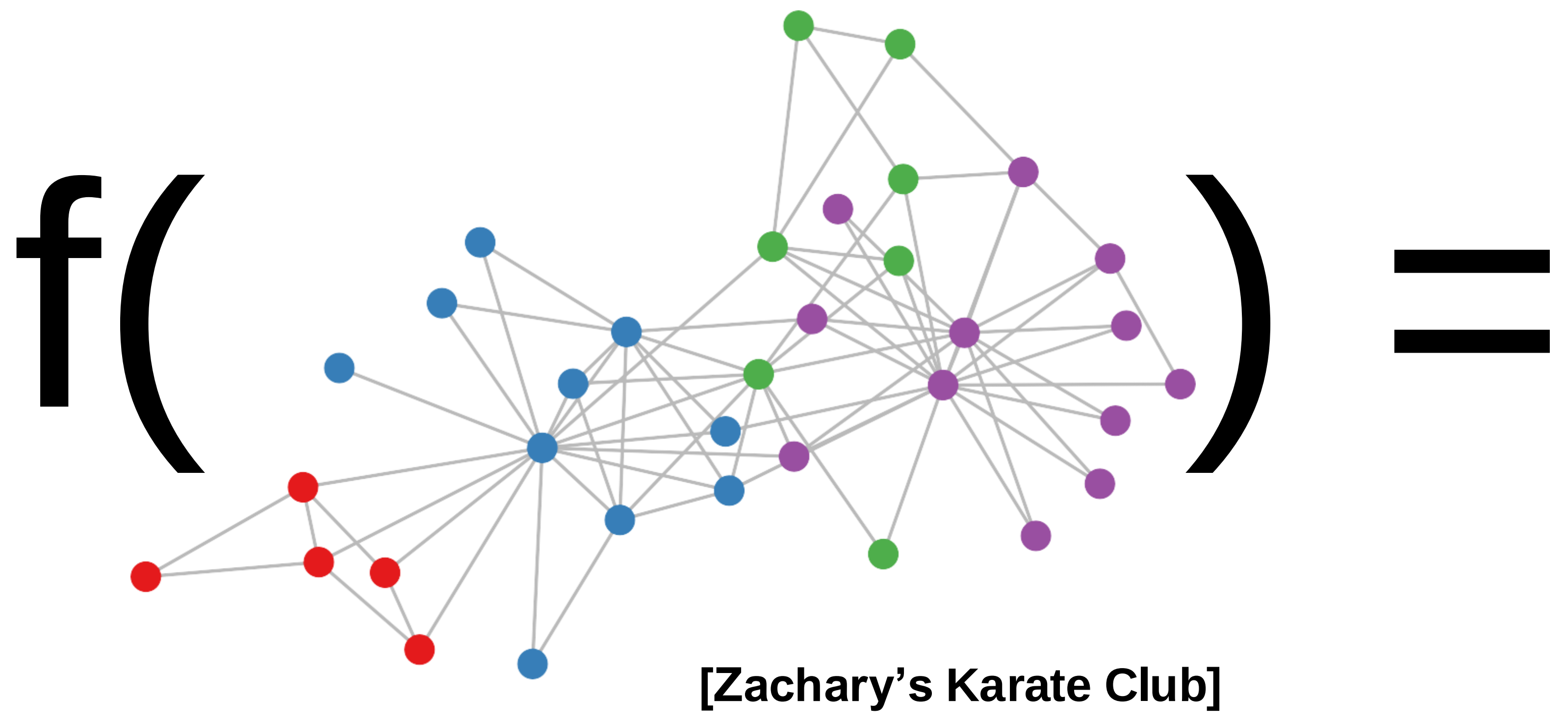
Kipf & Welling (NIPS BDL 2016)

"Graph Auto-Encoders"

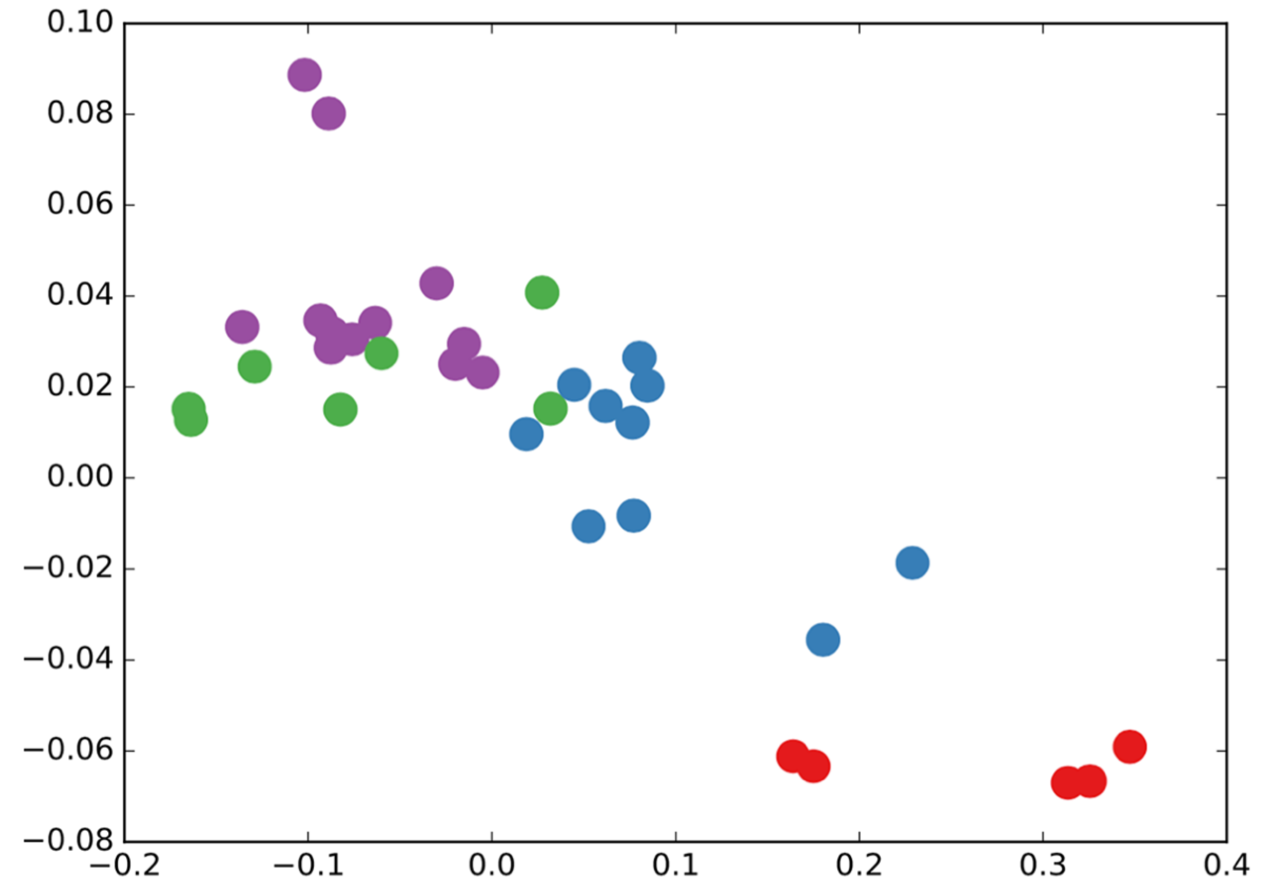
What do learned representations look like?

Forward pass through **untrained** 3-layer GCN model

Parameters initialized randomly



2-dim output per node

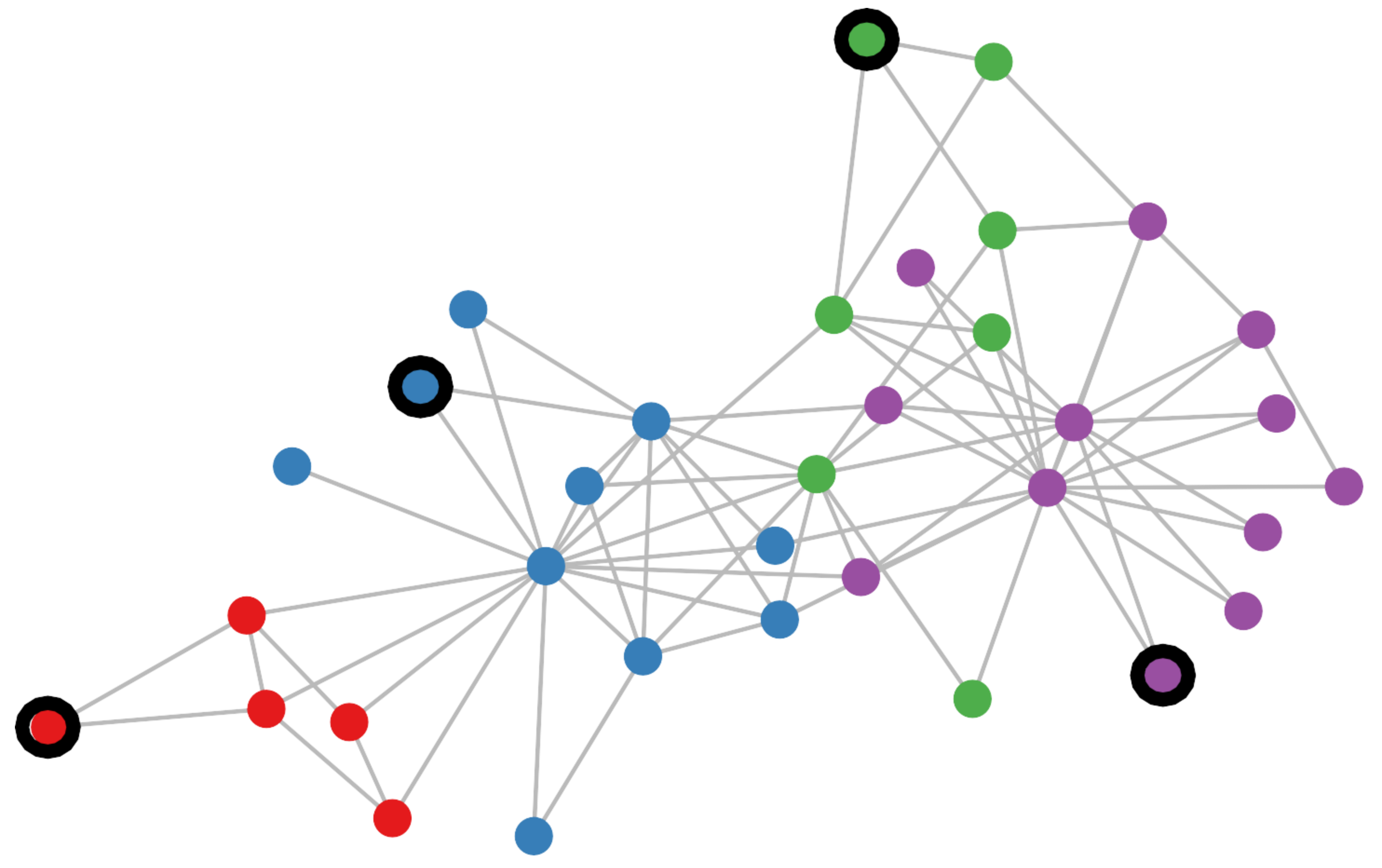


Semi-supervised classification on graphs

Setting:

Some nodes are labeled (black circle)
All other nodes are unlabeled

Task:
Predict node label of unlabeled nodes

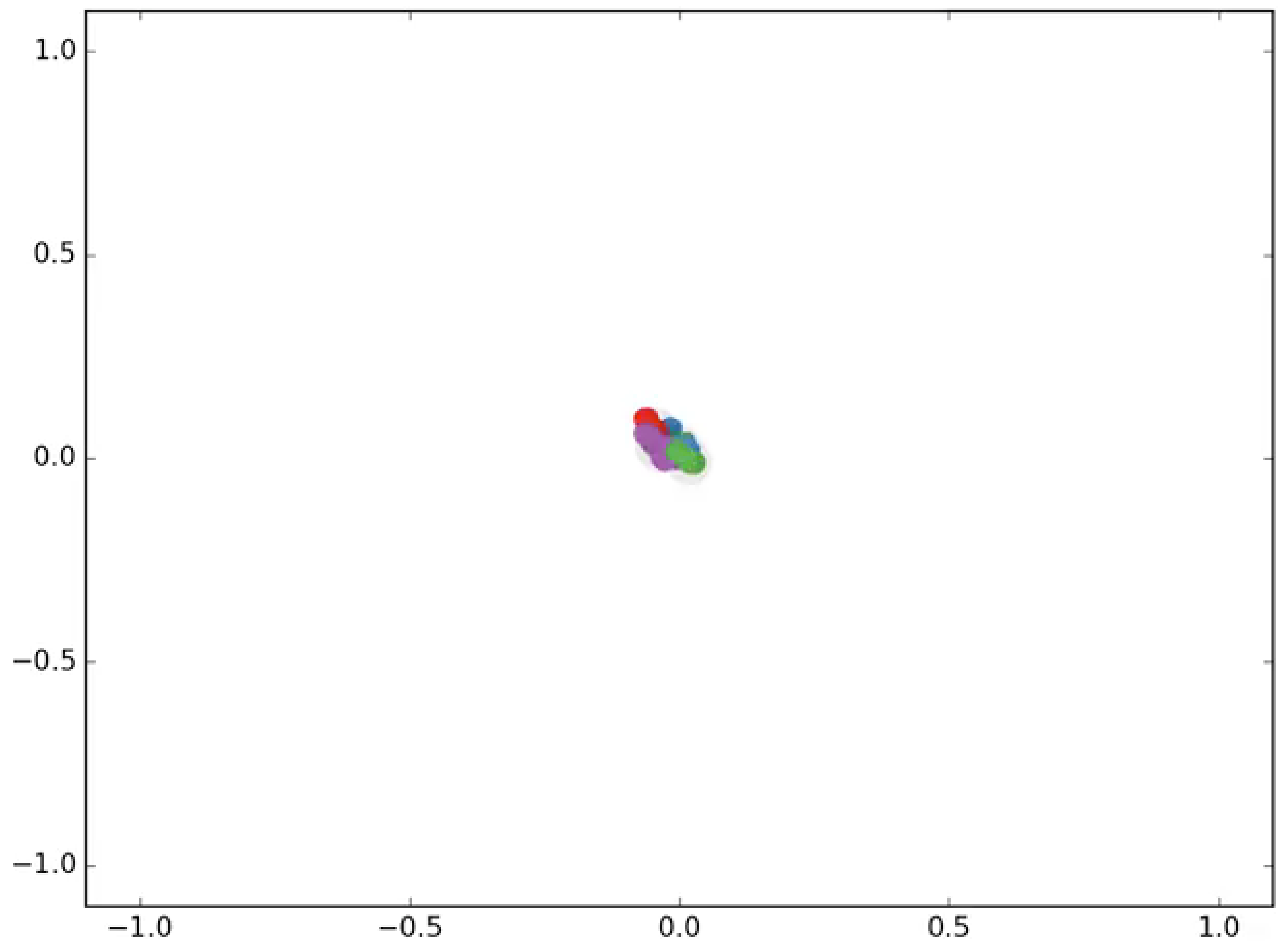


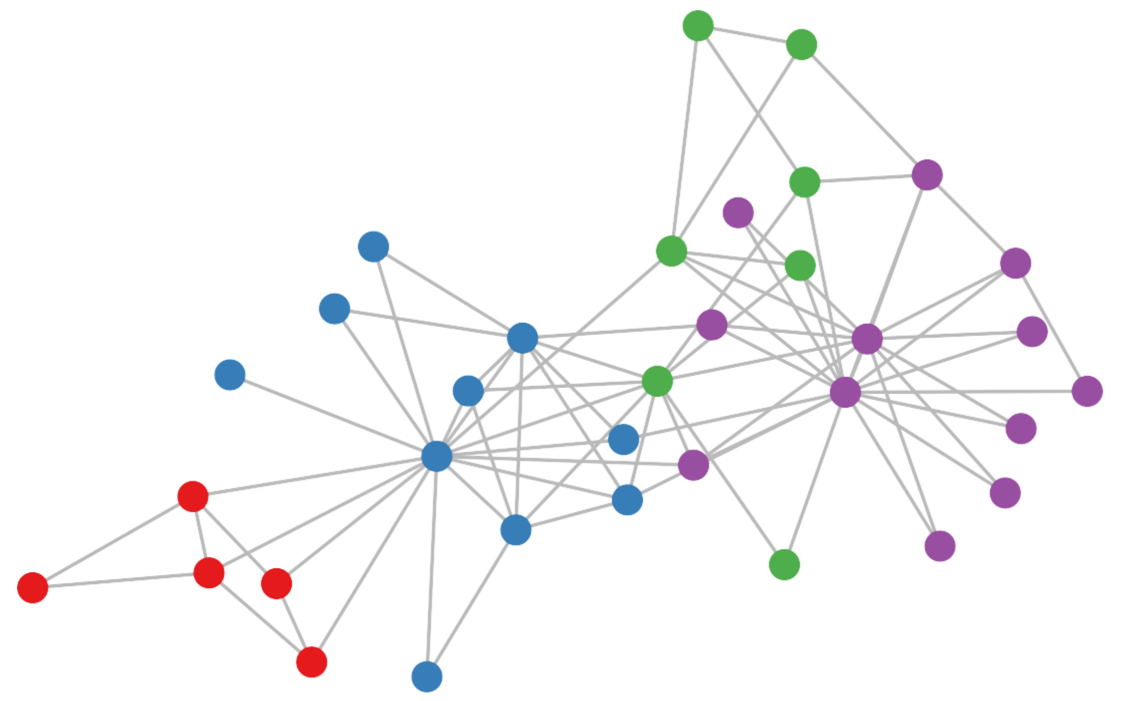
Evaluate loss on labeled nodes only:

$$\mathcal{L} = -\sum_{l \in \mathcal{Y}_L} \sum_{f=1}^{F} Y_{lf} \ln Z_{lf}$$

$\mathcal{Y}_L$	set of labeled node indices
$\mathbf{Y}$	label matrix
$\mathbf{Z}$	GCN output (after softmax)

Toy example (semi-supervised learning)

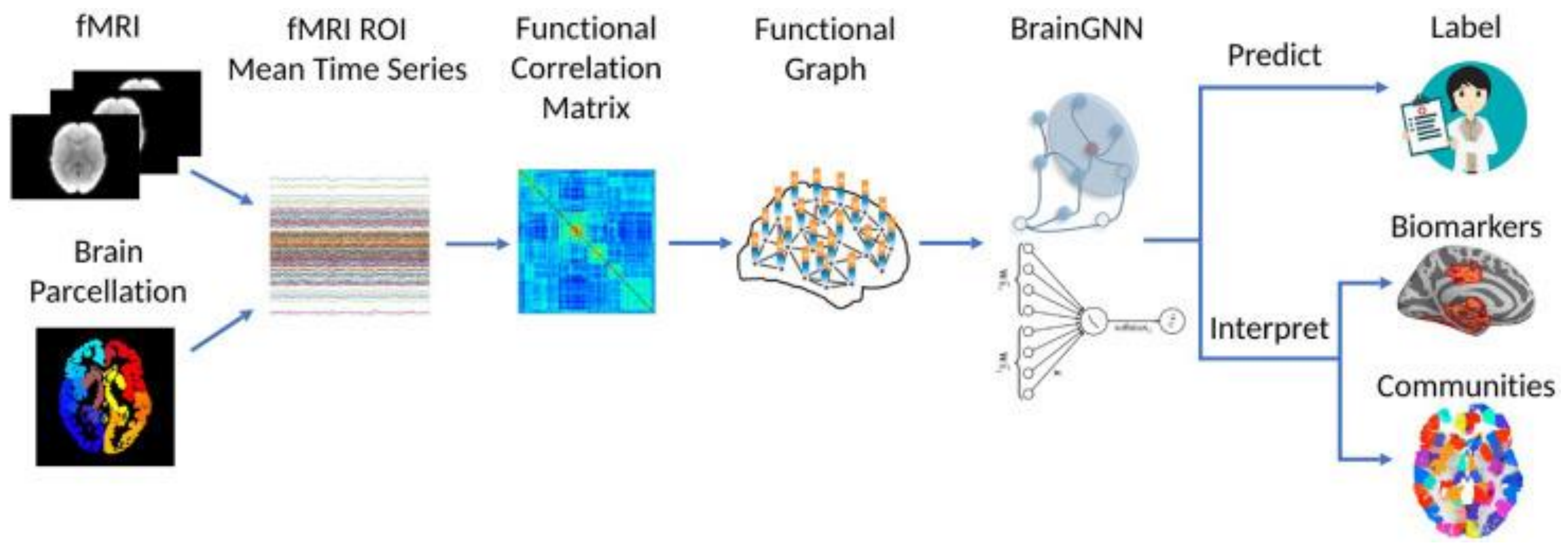


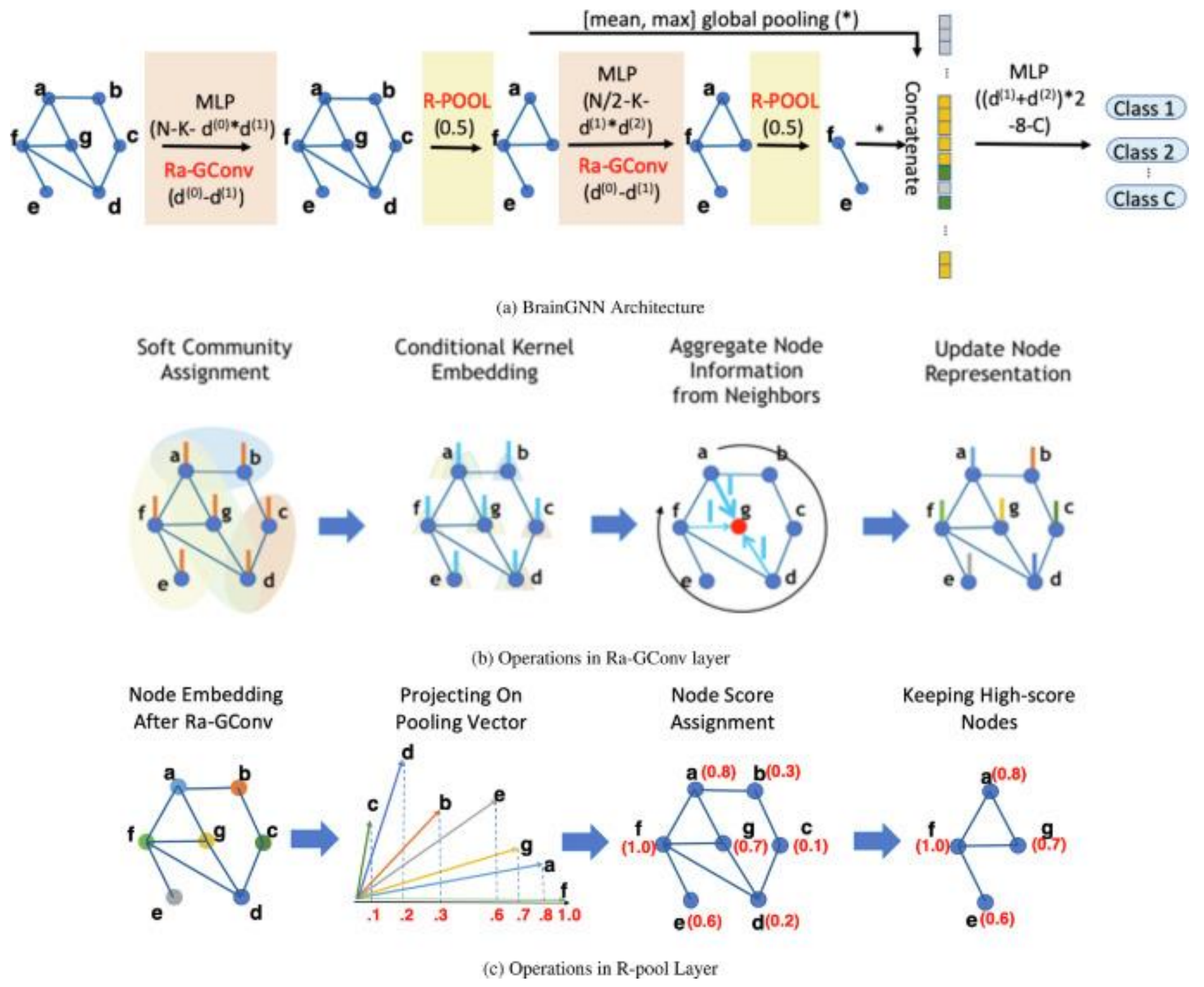


Video also available here:
http://tkipf.github.io/graph-convolutional-networks

Application 1: BrainGNN

Interpretable Brain Graph Neural Network for fMRI Analysis




[mean, max] global pooling (*)
MLP (N-K- d(0)*d(1))
Ra-GConv (d(0)-d(1))
R-POOL (0.5)
MLP (N/2-K- d(1)*d(2))
Ra-GConv (d(0)-d(1))
R-POOL (0.5)
Concatenate
MLP ((d(1)+d(2))*2 -8-C)
Class 1
Class 2
Class C
(a) BrainGNN Architecture
Soft Community Assignment
Conditional Kernel Embedding
Aggregate Node Information from Neighbors
Update Node Representation
(b) Operations in Ra-GConv layer
Node Embedding After Ra-GConv
Projecting On Pooling Vector
Node Score Assignment
Keeping High-score Nodes
(c) Operations in R-pool Layer

1. **Neurological Biomarker Discovery**: Identify key brain regions associated with neurological disorders or cognitive tasks.
2. **fMRI-Based Classification**: Classify Autism Spectrum Disorder (ASD) patients vs. Healthy Controls (HC) and decode cognitive states from fMRI data.
3. **Graph-Based Representation of the Brain**: Model brain regions as nodes and functional connections as edges, using fMRI data.

Li, X., Zhou, Y., Dvornek, N., Zhang, M., Gao, S., Zhuang, J., Scheinost, D., Staib, L.H., Ventola, P. and Duncan, J.S., 2021. Braingnn: Interpretable brain graph neural network for fmri analysis. Medical Image Analysis, 74, p.102233.

Application 1: BrainGNN

Graph Neural Network (GNN) Architecture:

- ROI-Aware Graph Convolutional Layer (Ra-GConv): Embeds nodes while considering brain region identities.
- ROI-Selection Pooling Layer (R-Pool): Selects the most relevant brain regions for classification.
- Regularization Losses:

a) Unit Loss: Ensures model stability.

b) Top-K Pooling (TPK) Loss: Encourages selection of the most informative brain regions.

c) Group-Level Consistency (GLC) Loss: Balances individual vs. group-level biomarker detection.

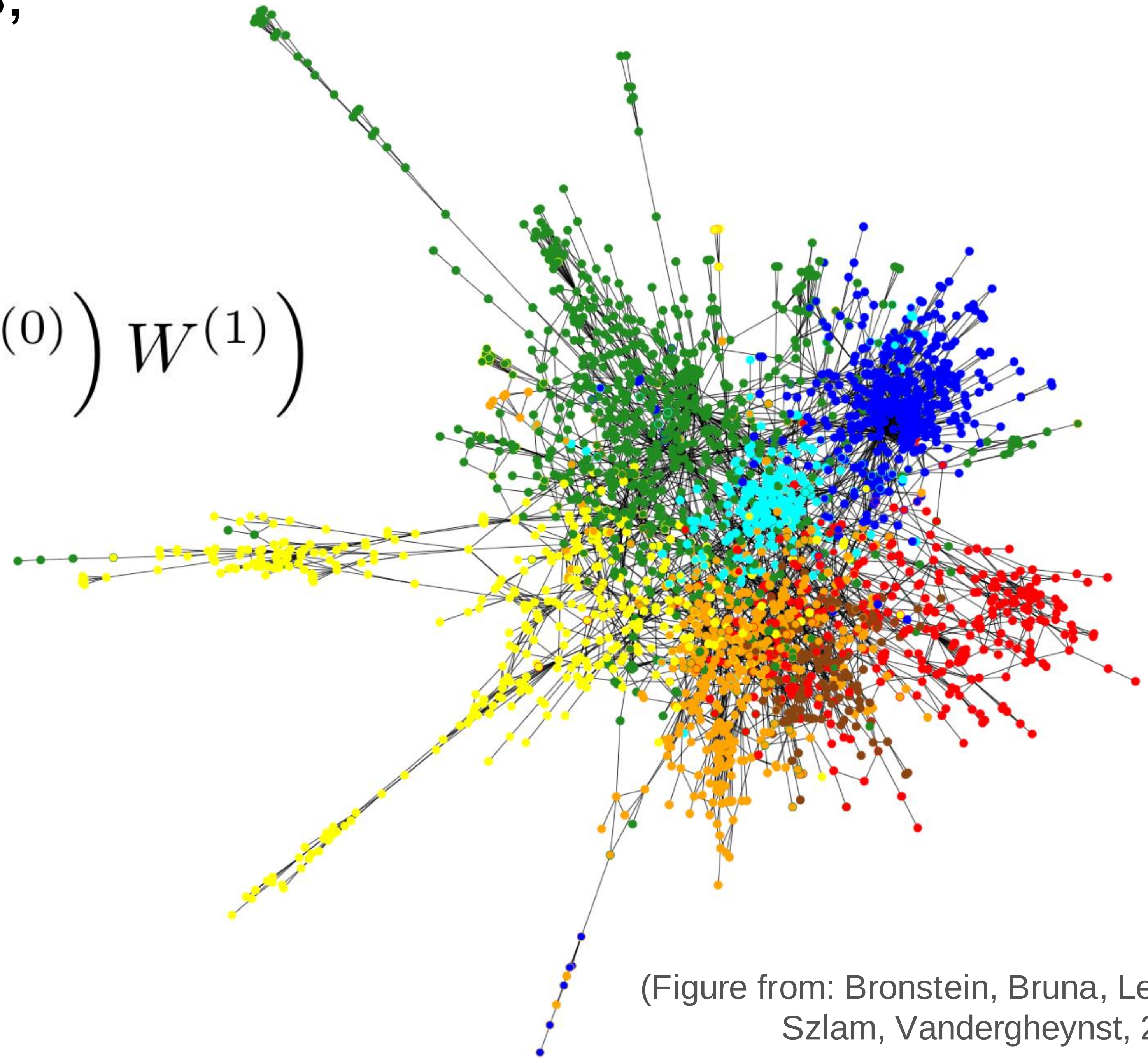
Li, X., Zhou, Y., Dvornek, N., Zhang, M., Gao, S., Zhuang, J., Scheinost, D., Staib, L.H., Ventola, P. and Duncan, J.S., 2021. Braingnn: Interpretable brain graph neural network for fmri analysis. Medical Image Analysis, 74, p.102233.

Application 2: Classification on citation networks

Input: Citation networks (nodes are papers, edges are citation links, optionally bag-of-words features on nodes)

Target: Paper category (e.g. stat.ML, cs.LG, …)

Model: 2-layer GCN $Z = f(X, A) = \mathrm{softmax}\Big(\hat{A}\ \mathrm{ReLU}\Big(\hat{A}XW^{(0)}\Big)\, W^{(1)}\Big)$



(Figure from: Bronstein, Bruna, LeCun, Szlam, Vandergheynst, 2016)

Kipf & Welling, Semi-Supervised Classification with Graph Convolutional Networks, ICLR 2017

Application 2: Classification on citation networks

Input: Citation networks (nodes are papers, edges are citation links, optionally bag-of-words features on nodes)

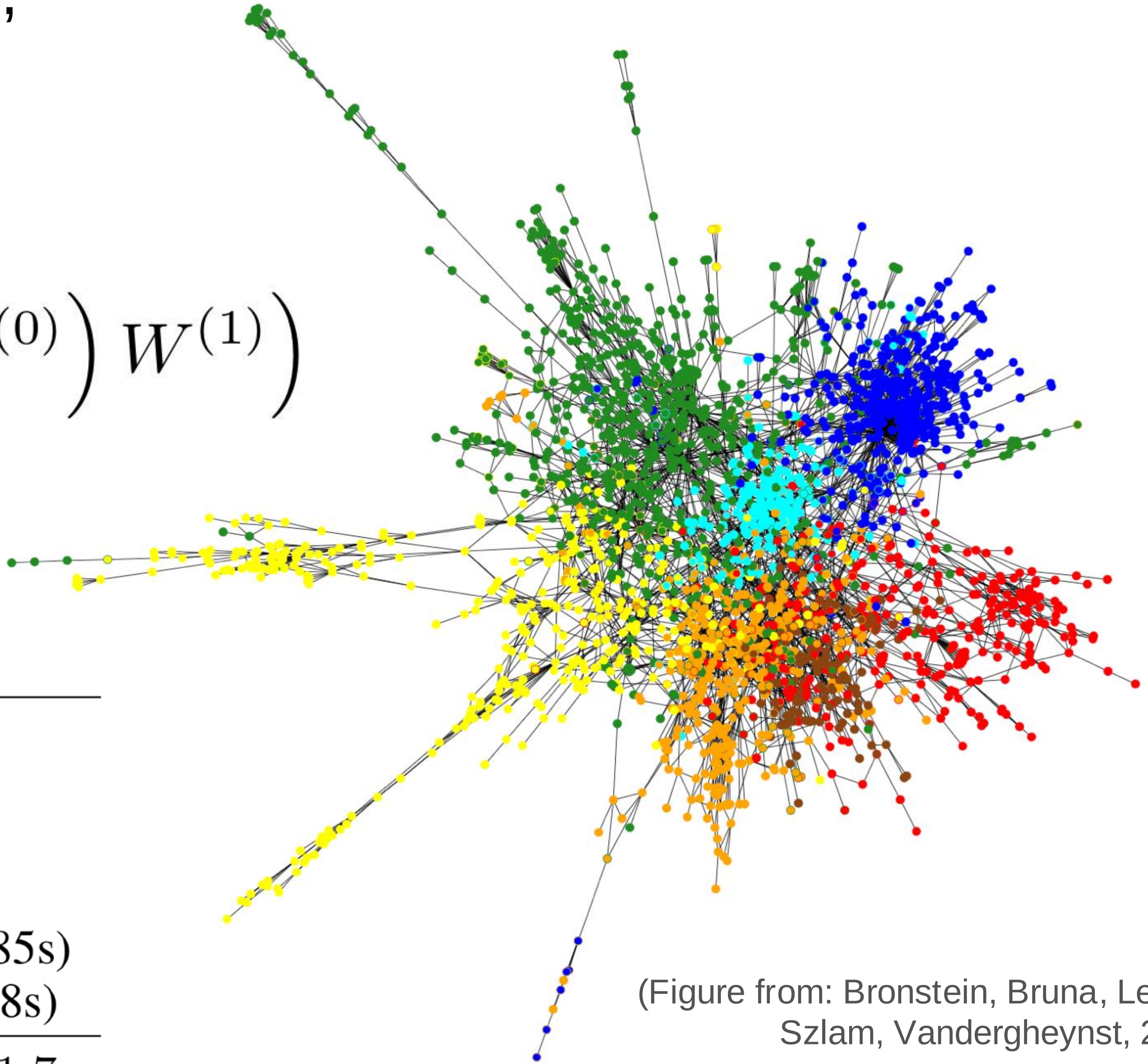
Target: Paper category (e.g. stat.ML, cs.LG, …)

Model: 2-layer GCN $Z = f(X, A) = \mathrm{softmax}\left(\hat{A}\ \mathrm{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right)$

Classification results (accuracy)

no input features

Method	Citeseer	Cora	Pubmed	NELL
ManiReg [3]	60.1	59.5	70.7	21.8
SemiEmb [24]	59.6	59.0	71.1	26.7
LP [27]	45.3	68.0	63.0	26.5
DeepWalk [18]	43.2	67.2	65.3	58.1
Planetoid* [25]	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN (this paper)	**70.3** (7s)	**81.5** (4s)	**79.0** (38s)	**66.0** (48s)
GCN (rand. splits)	67.9 ± 0.5	80.1 ± 0.5	78.9 ± 0.7	58.4 ± 1.7



(Figure from: Bronstein, Bruna, LeCun, Szlam, Vandergheynst, 2016)

Kipf & Welling, Semi-Supervised Classification with Graph Convolutional Networks, ICLR 2017

Application 3: Cancer Drug Response Prediction

DeepCDR is an end-to-end deep learning framework for predicting cancer drug response. DeepCDR contains a novel UGCN architecture for representing topological information of drugs.

- Drug can be represented as $\{\mathcal{G}_i = (\mathbf{X}_i, \mathbf{A}_i)|_{i=1}^{M}\}$ where $\mathbf{X}_i \in \mathbb{R}^{N_i \times C}$ and $\mathbf{A}_i \in \mathbb{R}^{N_i \times N_i}$ is the feature matrix and adjacent matrix of ith drug

- Each atom in a drug was represented as a 75-dimensional feature vector, including chemical and topological properties (atom type, degree and hybridization)

- We extended the original GCN (Kipf et al., *ICLR*, 2017) for processing drugs with variable sizes and structures by introducing complementary drug

- Layer-wise operation of UGCN is derived as

$$\mathbf{H}_i^{(l+1,\alpha)} = \sigma(((\tilde{\mathbf{D}}_i + \mathbf{D}_i^B)^{-\frac{1}{2}}\tilde{\mathbf{A}}_i(\tilde{\mathbf{D}}_i + \mathbf{D}_i^B)^{-\frac{1}{2}}\mathbf{H}_i^{(l,\alpha)} + (\tilde{\mathbf{D}}_i + \mathbf{D}_i^B)^{-\frac{1}{2}}\mathbf{B}_i(\tilde{\mathbf{D}}_i^c + \mathbf{D}_i^{B^T})^{-\frac{1}{2}}\mathbf{H}_i^{(l,\beta)})\mathbf{\Theta}^{(l)})$$

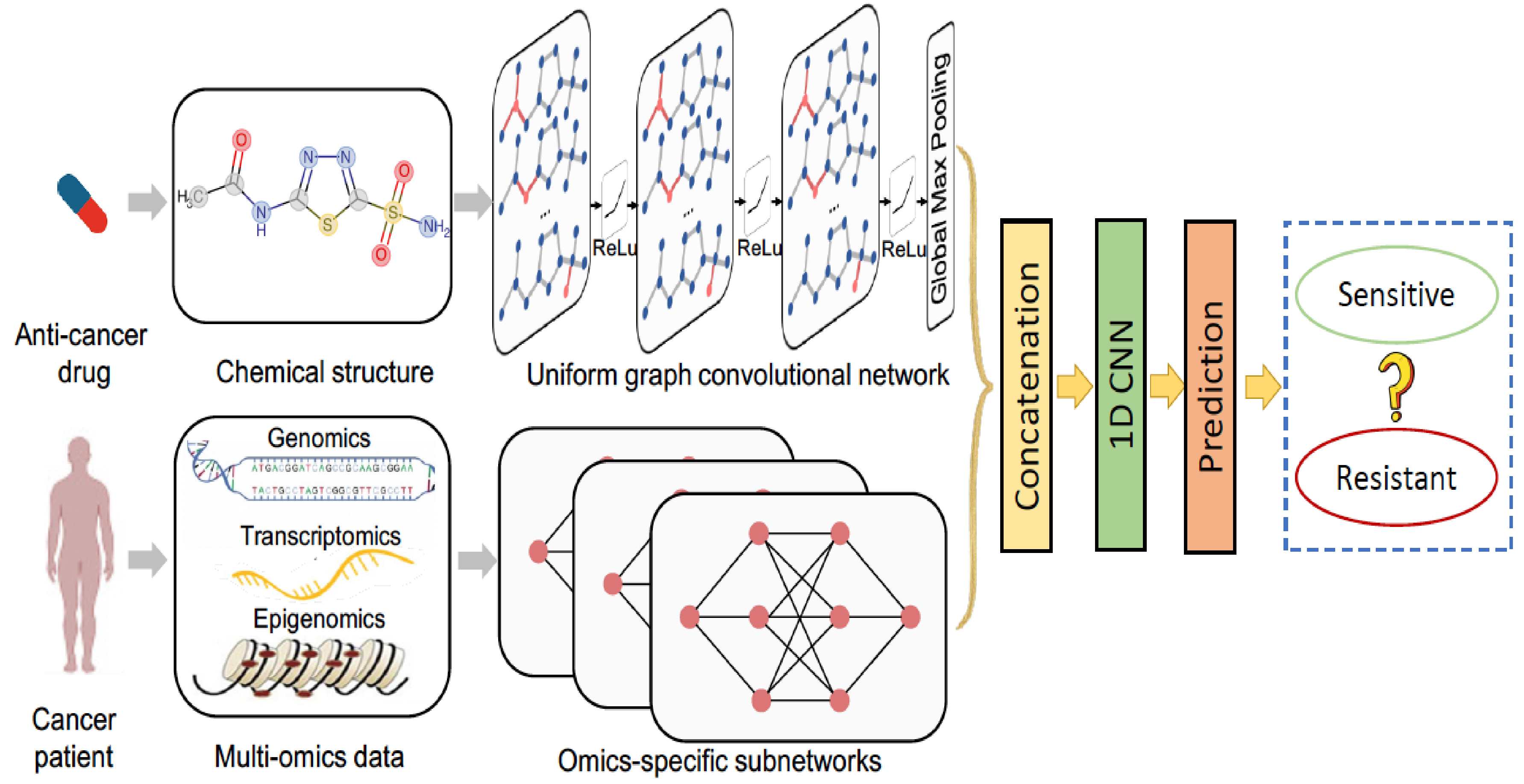
$$\mathbf{H}_i^{(l+1,\beta)} = \sigma(((\tilde{\mathbf{D}}_i^c + \mathbf{D}_i^{B^T})^{-\frac{1}{2}}\mathbf{B}_i^T(\tilde{\mathbf{D}}_i + \mathbf{D}_i^B)^{-\frac{1}{2}}\mathbf{H}_i^{(l,\alpha)} + (\tilde{\mathbf{D}}_i^c + \mathbf{D}_i^{B^T})^{-\frac{1}{2}}\tilde{\mathbf{A}}_i^c(\tilde{\mathbf{D}}_i^c + \mathbf{D}_i^{B^T})^{-\frac{1}{2}}\mathbf{H}_i^{(l,\beta)})\mathbf{\Theta}^{(l)})$$

- Drugs with different size will be embedded into a fixed dimensional vector (default: 100)

Liu, Qiao, Zhiqiang Hu, Rui Jiang, and Mu Zhou. "DeepCDR: a hybrid graph convolutional network for predicting cancer drug response." *Bioinformatics* 36, no. Supplement_2 (2020): i911-i918.

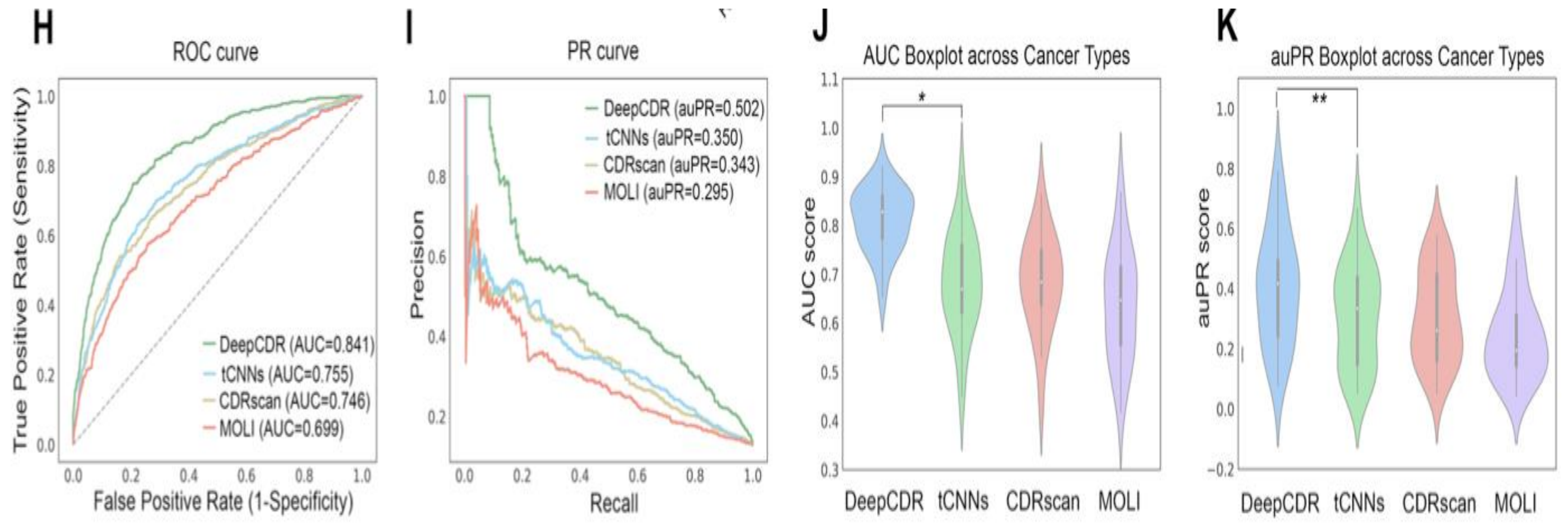
DeepCDR

DeepCDR takes chemical structures of drugs and multi-omics data of cancer cell line as input and outputs the sensitivity of drug.



Methods	Pearson's correlation	Spearman's correlation	RMSE
Ridge Regression	0.780	0.731	2.368
Random Forest	0.809	0.767	2.270
MOLI	0.813±0.007	0.782±0.005	2.282±0.008
CDRscan	0.871±0.004	0.852±0.003	1.982±0.005
tCNNs	0.885±0.008	0.862±0.006	1.782±0.006
DeepCDR	**0.923±0.006**	**0.903±0.004**	**1.058±0.006**

- Drug structure data (.MOL) were from Pubchem database (Kim et al., 2018)
- Cancer cell lines data (genomic mutation, gene expression and DNA methylation) were downloaded from Cancer Cell Line Encyclopedia (CCLE) (Barretina et al. 2012)
- Drug sensitivity data were obtained from Genomics of Drug Sensitivity in Cancer (GDSC) (Iorio et al. 2016)
- We finally collected a dataset containing 107446 instances across 561 cancer cell lines and 223 drugs
- Each cell line corresponds to a TCGA cancer type.



Content

Theoretical Properties

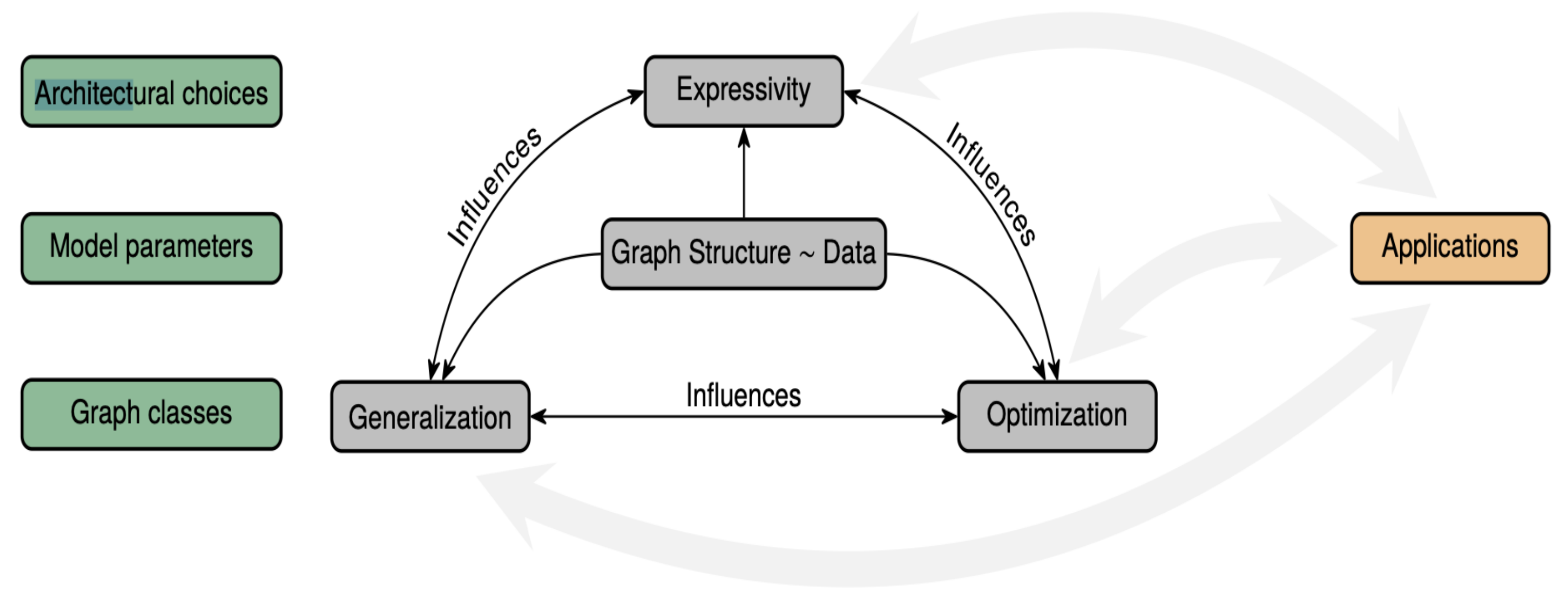


Figure 1: Interactions of the four challenges within graph machine learning: Fine-grained *expressivity*, *generalization*, *optimization*, *applications*, and their interactions. The green boxes *architectural choices* (hyperparameter and other design choices like normalization layers), *model parameters*, and *graph classes* (different types of graphs) represent aspects of all four challenges.

❖ **Expressivity:** What graph structures can a GNN distinguish?
– Traditional results relate GNNs to the 1-WL test, but finer geometric notions are needed.

❖ **Approximation:** Under what conditions can GNNs approximate continuous, permutation-invariant functions?
– Universal approximation results require a careful treatment of the topology of graph space.

❖ **Generalization:** How well do GNNs perform on unseen graphs?
– Existing VC-dimension based bounds are loose and do not fully capture the influence of architectural choices and graph structure.

Future Directions

❖ Develop fine-grained expressivity results that quantify not only if two graphs are distinguishable, but how similar they are.
❖ Derive uniform approximation bounds for GNNs using a refined topology on graph space.
❖ Establish tighter generalization bounds that incorporate architectural choices and graph geometry.
❖ Explore the interplay between expressivity, optimization, and generalization to inform the design of more robust GNN architectures.

Morris, Christopher, Fabrizio Frasca, Nadav Dym, Haggai Maron, Ismail Ilkan Ceylan, Ron Levie, Derek Lim, Michael M. Bronstein, Martin Grohe, and Stefanie Jegelka. "Position: Future Directions in the Theory of Graph Machine Learning." In *Forty-first International Conference on Machine Learning*.

Deepset

A function f transforming a set $X = \{x_1, \ldots, x_M\}$ into Y should be:

- **Permutation invariant:** The output does not change under reordering.

$$f(\{x_1, \ldots, x_M\}) = f(\{x_{\pi(1)}, \ldots, x_{\pi(M)}\})$$

  for any permutation π.

- **Permutation equivariant:** The output follows the permutation.

$$f([x_{\pi(1)}, \ldots, x_{\pi(M)}]) = [f_{\pi(1)}(x), \ldots, f_{\pi(M)}(x)]$$

Theorem. A function $f(X)$ is invariant to the permutation of instances in X if and only if it can be decomposed as:

$$f(X) = \rho\left(\sum_{x \in X} \phi(x)\right)$$

where ϕ and ρ are suitable transformations.

The standard neural network layer is represented as:

$$f_\Theta(\mathbf{x}) = \sigma(\Theta \mathbf{x})$$

where $\Theta \in \mathbb{R}^{M \times M}$ is the weight matrix.
Theorem. A function $f_\Theta : \mathbb{R}^M \to \mathbb{R}^M$ is permutation equivariant if:

$$\Theta = \lambda I + \gamma(11^T)$$

where:

- I is the identity matrix,
- $1 = [1, \ldots, 1]^T$,
- $\lambda, \gamma \in \mathbb{R}$.

de Finetti's theorem states that any exchangeable model can be factored as

$$p(X|\alpha, M_0) = \int d\theta \left[\prod_{m=1}^{M} p(x_m|\theta)\right] p(\theta|\alpha, M_0).$$

where θ is a latent feature and α, M_0 are hyper-parameters of the prior.

For Exponential Family with Conjugate Priors:

$$p(X|\alpha, M_0) = \exp\left(h\left(\alpha + \sum_m \phi(x_m), M_0 + M\right) - h(\alpha, M_0)\right)$$

PINE

Let $\mathbf{f}$ *be a continuous real-valued function defined on a compact set with the following form*

$$\mathbf{f}(\underbrace{\mathbf{x}_{1,1}, \mathbf{x}_{1,2}, \cdots, \mathbf{x}_{1,N_1}}_{G_1}, \underbrace{\mathbf{x}_{2,1}, \cdots, \mathbf{x}_{2,N_2}}_{G_2}, \cdots, \underbrace{\mathbf{x}_{K,1}, \cdots, \mathbf{x}_{K,N_K}}_{G_K}),$$

where $\mathbf{x}_{k,n} \in \mathbb{R}^{M_k}$. *If function* $\mathbf{f}$ *is partial permutation invariant,* then the PINE framework provides a **Core Representation Theorem as**

$$\mathbf{f}(\cdot) = \mathbf{h}\left(\sum_{n=1}^{N_1} \mathbf{g}_1(\mathbf{x}_{1,n}), \sum_{n=1}^{N_2} \mathbf{g}_2(\mathbf{x}_{2,n}), \ldots, \sum_{n=1}^{N_K} \mathbf{g}_K(\mathbf{x}_{K,n})\right) + o(1)$$

which requires $h(\cdot)$ and $g(\cdot)$ to ensure permutation invariant :

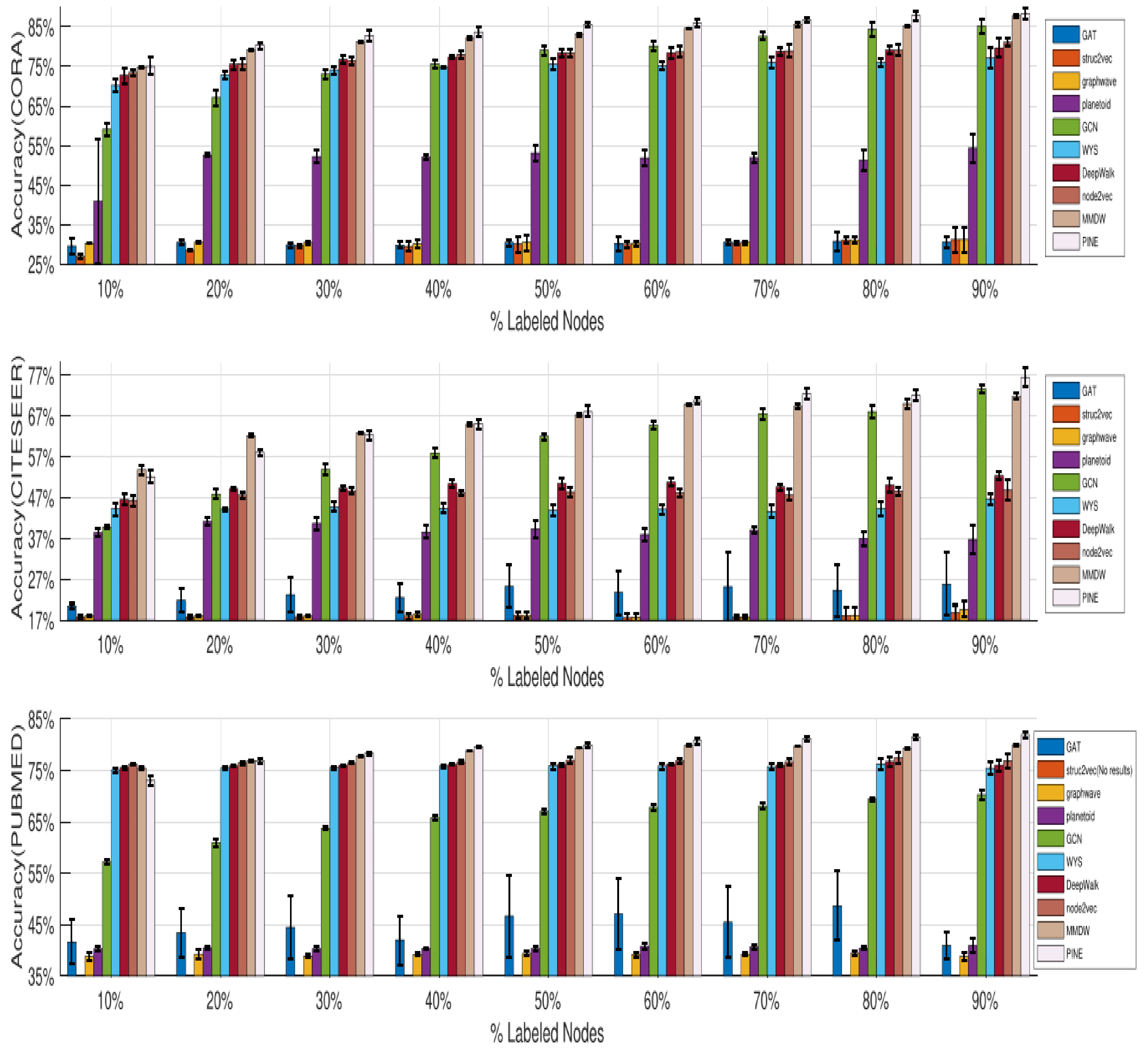
Then, PINE provides specific parameters for $h(\cdot)$ and $g(\cdot)$, which can be trained as follows:

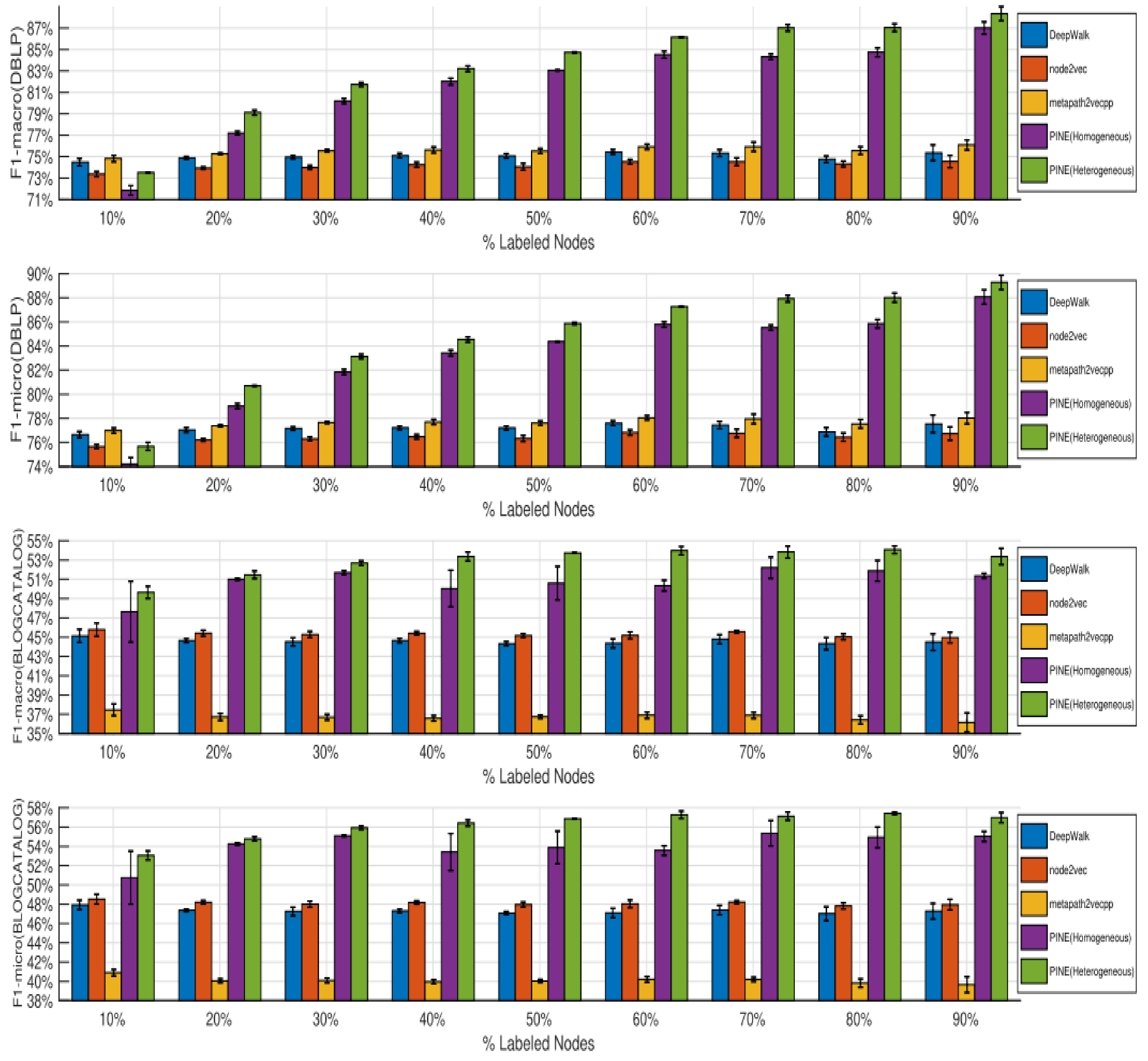
$$\mathbf{h}\left([\mathbf{z}_1^\top, \ldots, \mathbf{z}_K^\top]^\top =: \bar{\mathbf{z}} \mid c, W, b\right) = c^\top \sigma(W\bar{\mathbf{z}} + b)$$

$$\mathbf{g}\left(\mathbf{x} \mid T, u, \{a_t, b_t'\}_{t=1}^{T}\right) = \begin{bmatrix} \sigma((u \otimes a_1)\mathbf{x} + b_1') \\ \sigma((u \otimes a_2)\mathbf{x} + b_2') \\ \vdots \\ \sigma((u \otimes a_T)\mathbf{x} + b_T') \end{bmatrix}$$

Evaluation

Accuracy (%) of multi-class classification in homogeneous and heterogeneous graphs





References

Bessadok, A., Mahjoub, M. A., & Rekik, I. (2022). Graph neural networks in network neuroscience. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *45*(5), 5833-5848.

Corso, G., Stark, H., Jegelka, S., Jaakkola, T., & Barzilay, R. (2024). Graph neural networks. *Nature Reviews Methods Primers*, *4*(1), 17.

Hamilton, W. L. *Graph Representation Learning*, 2023. Synthesis Lectures in Artificial Intelligence and Machine Learning. Vol 14, No.3.

Gui, Shupeng, Xiangliang Zhang, Pan Zhong, Shuang Qiu, Mingrui Wu, Jieping Ye, Zhengdao Wang, and Ji Liu. "Pine: Universal deep embedding for graph nodes via partial permutation invariant set functions." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, no. 2 (2021): 770-782.

Khoshraftar, S., & An, A. (2024). A survey on graph representation learning methods. *ACM Transactions on Intelligent Systems and Technology*, *15*(1), 1-55.

Liu, Q., Hu, J., Xiao, Y., Zhao, X., Gao, J., Wang, W., ... & Tang, J. (2024). Multimodal recommender systems: A survey. *ACM Computing Surveys*, *57*(2), 1-17.

Ma, Y., & Tang, J. (2021). *Deep learning on graphs*. Cambridge University Press.

Veličković, P. (2023). Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, *79*, 102538.

Wu, S., Sun, F., Zhang, W., Xie, X., & Cui, B. (2022). Graph neural networks in recommender systems: a survey. *ACM Computing Surveys*, *55*(5), 1-37.

Wu, L., Cui, P., Pei, J., and Zhao, L. (2022). Graph Neural Networks: Foundations, Frontiers, and Applications. Springer.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, *32*(1), 4-24.

Xie, Y., Xu, Z., Zhang, J., Wang, Z., & Ji, S. (2022). Self-supervised learning of graph neural networks: A unified review. *IEEE transactions on pattern analysis and machine intelligence*, *45*(2), 2412-2429.

Ye, Z., Kumar, Y. J., Sing, G. O., Song, F., & Wang, J. (2022). A comprehensive survey of graph neural networks for knowledge graphs. *IEEE Access*, *10*, 75729-75741.

Yuan, H., Yu, H., Gui, S., & Ji, S. (2022). Explainability in graph neural networks: A taxonomic survey. *IEEE TPAMI*, *45*(5), 5782-5799.

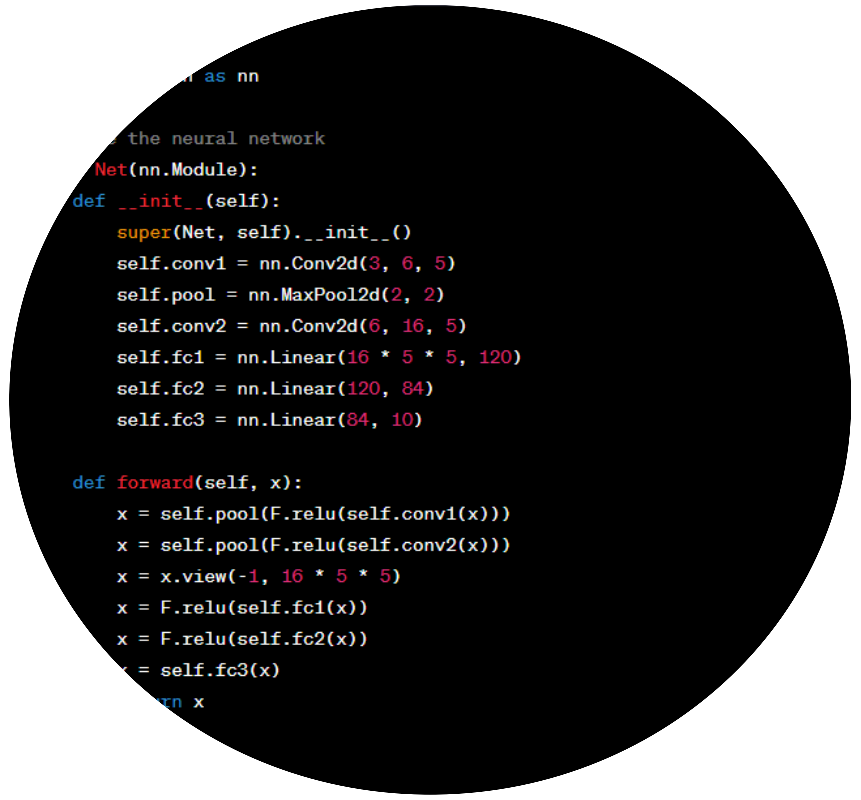
Zaheer, Manzil, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R. Salakhutdinov, and Alexander J. Smola. "Deep sets." *Advances in neural information processing systems* 30 (2017).

Zhang, S., Tong, H., Xu, J., & Maciejewski, R. (2019). Graph convolutional networks: a comprehensive review. *Computational Social Networks*, *6*(1), 1-23.

Zhang, Z., Cui, P., & Zhu, W. (2020). Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *34*(1), 249-270. X

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

How to succeed in this course?



```
    as nn

  the neural network
 Net(nn.Module):
def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(3, 6, 5)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
      = self.fc3(x)
      rn x
```

Practice

Explore

Visualize

Discuss

Ask