# Introduction to Graph Neural Networks

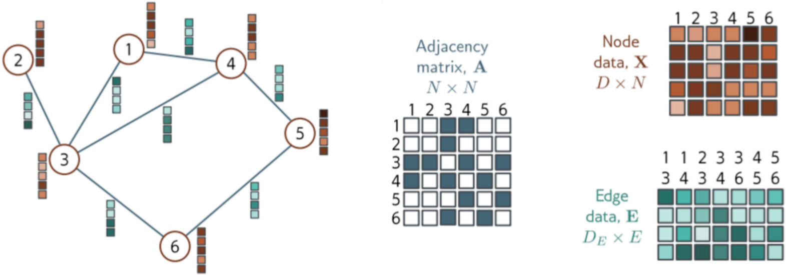
Mahdi Mastani

27 May 2025

# Graph Representation and Encoding

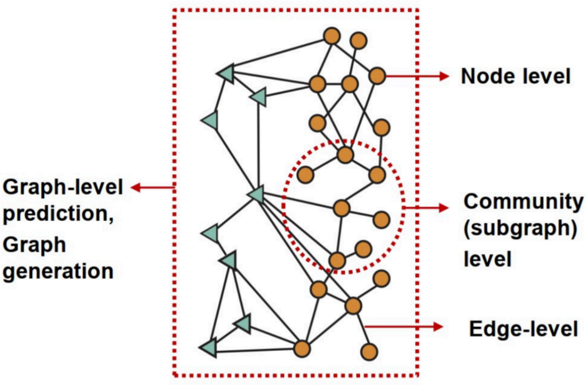
A graph consists of:

- **Graph Structure**
- **Node Embeddings**
- **Edge Embeddings**
- **Degree Matrix**



# Common Tasks for Graphs

- **Node Classification**
- **Link Prediction**
- **Graph Classification**
- **Community Detection**
- **Graph Generation**



# Key Properties of Graph Neural Networks

- **Generalization**: The ability to apply learned models to graphs of different sizes and topologies.
- **Scalability**: The architecture should be efficient enough to handle large graphs with millions of nodes and edges.
- **Permutation equivariance**: The model should produce the same output regardless of the ordering of the nodes and edges in the input graph.

# Permutation Equivariance in Graphs

- **Permutation matrix**: A permutation matrix $P \in \{0, 1\}^{n \times n}$ is a binary square matrix with exactly one entry of 1 in each row and column. It represents a reordering of elements.

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

- When position $(i, j)$ of the permutation matrix is set to **one**, it indicates that node $i$ will become node $j$ after the permutation.

# Indexing and Permutation Effects

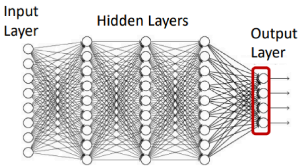
- Changing node indexing in a graph requires transforming the data accordingly.
- Pre-multiplying by $P$ reorders the **rows** (used for node features).
- Post-multiplying by $P^\top$ reorders the **columns** (used for graph structure).
- The operations to map between indexings:

$$X' = PX, \quad A' = PAP^\top$$

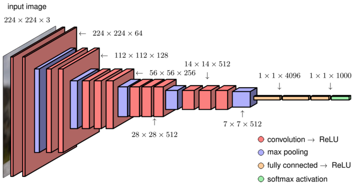
- **Conclusion**: Any graph processing model should remain invariant to these permutations:

$$P\hat{y}(X, A) = \hat{y}(X', A')$$

- **MLP**:

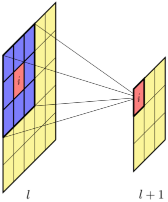

Input Layer
Hidden Layers
Output Layer

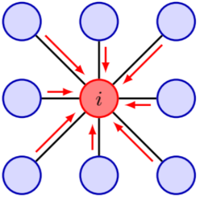
- **CNN**:


input image
224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution → ReLU
max pooling
fully connected → ReLU
softmax activation

**How CNNs Work**

- CNNs operate on grid-structured data (like images).
- Use local filters (kernels) to scan spatially arranged data.
- Employ weight sharing and local connectivity to capture local patterns.

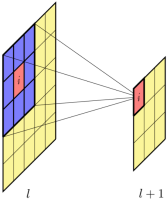


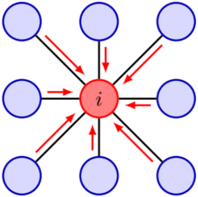



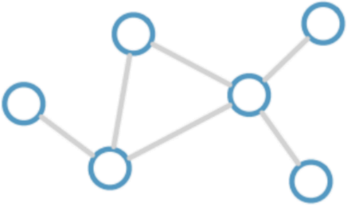

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_j \mathbf{W}_j^{(l)} \mathbf{h}_j^{(l)}\right)$$

# Why CNNs Are Not Suitable for Graphs

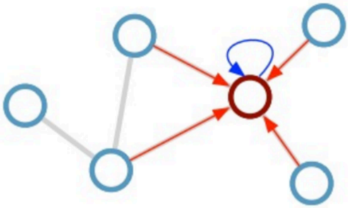
- Graphs are non-Euclidean: no fixed node order or grid structure.
- Nodes may have varying numbers of neighbors.





**Undirected Graph**



**Update Rule for Node Embedding**

- **Step 1: Aggregate neighbors**
- **Step 2: Add self-loop**

**Update rule:** $$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j \in \mathcal{N}_i} \frac{1}{c_{ij}} \mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right)$$

# Simple Message-Passing Neural Network

**Algorithm 1** Simple message-passing neural network

**Require:** Undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Initial node embeddings $\{\mathbf{h}_n^{(0)} = \mathbf{x}_n\}$
Aggregate$(\cdot)$ function
Update$(\cdot, \cdot)$ function

**Ensure:** Final node embeddings $\{\mathbf{h}_n^{(L)}\}$

1: // Iterative message-passing
2: **for** $l \in \{0, \ldots, L-1\}$ **do**
3: $\quad \mathbf{z}_n^{(l)} \leftarrow \text{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right)$
4: $\quad \mathbf{h}_n^{(l+1)} \leftarrow \text{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right)$
5: **end for**
6: **return** $\{\mathbf{h}_n^{(L)}\}$

# Aggregator and Update Functions in GNNs

**Aggregator Function:**

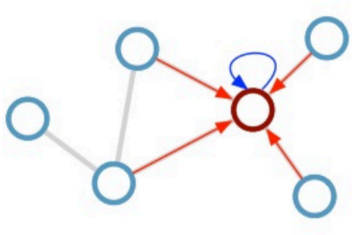
- Aggregator must be **permutation invariant**.
- Options:
  - **Sum**: Adds up neighbor features; sensitive to node degree.
  - **Mean**: Computes the average of neighbor features.
  - **Max**: Captures the most prominent signal per feature dimension.

**Update Function:**

- Update function should preserve or enhance node representations.
- Typically a neural network (e.g., MLP or linear layer).
- Can include residual connections or batch normalization.

**Step 1: Neighborhood Aggregation**

$$\textbf{Update rule:} \quad \mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j \in \mathcal{N}_i} \frac{1}{c_{ij}} \mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right)$$

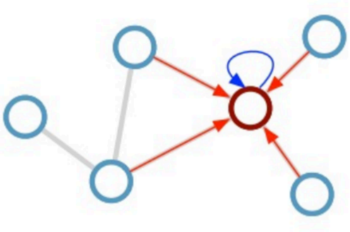


# Building GCN Step-by-Step

if we set $\mathbf{W}_0^l = \mathbf{W}_1^l = \mathbf{W}^l$ (Shared weight matrix):

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}^{(l)} + \sum_{j\in\mathcal{N}_i} \frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}^{(l)}\right)$$

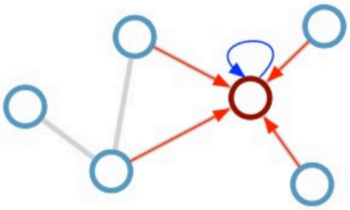
- how about $c_{ij}$ ?



# Building GCN Step-by-Step

use Kipf normalization $c_{ij} = \sqrt{d_i d_j}$

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}^{(l)} + \sum_{j\in\mathcal{N}_i} \frac{1}{\sqrt{d_i d_j}}\mathbf{h}_j^{(l)}\mathbf{W}^{(l)}\right)$$



# Translate to Graph Input

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}^{(l)} + \sum_{j\in\mathcal{N}_i} \frac{1}{\sqrt{d_i d_j}}\mathbf{h}_j^{(l)}\mathbf{W}^{(l)}\right)$$

Matrix form:

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

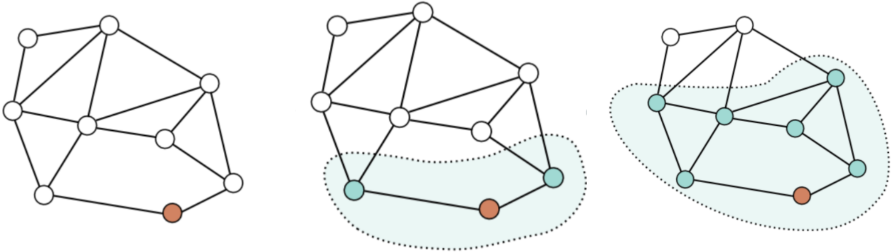
- If set $\hat{A} = \mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ we have:

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

# L-layer GCN

L-layer Graph convolutional networks (GCNs):

$$
\begin{aligned}
\mathbf{H}^{(1)} &= \mathbf{F}(\mathbf{X}, \mathbf{A}, \mathbf{W}^{(1)}) = \sigma\left(\hat{A}\mathbf{X}\mathbf{W}^{(1)}\right) \\
\mathbf{H}^{(2)} &= \mathbf{F}(\mathbf{H}^{(1)}, \mathbf{A}, \mathbf{W}^{(2)}) = \sigma\left(\hat{A}\mathbf{H}^{(1)}\mathbf{W}^{(2)}\right) \\
\vdots &= \vdots \\
\mathbf{H}^{(L)} &= \mathbf{F}(\mathbf{H}^{(L-1)}, \mathbf{A}, \mathbf{W}^{(L)}) = \sigma\left(\hat{A}\mathbf{H}^{(L-1)}\mathbf{W}^{(L)}\right)
\end{aligned}
$$



# From Fixed to Learnable Coefficients

- So far, we discussed using fixed normalization coefficients $\frac{1}{c_{ij}}$, such as:
  - Uniform (unweighted average)
  - Degree-based normalization (e.g., $\frac{1}{\sqrt{d_i d_j}}$)
- However, these do not adapt based on node features or context.
- Can we make these coefficients **learnable** instead?

# Graph Attention Layer - Overview

**Goal:** Compute hidden representations for each node by attending over its neighbors using self-attention.

**Key Properties:**

- Efficient and parallelizable across node-neighbor pairs.
- Supports nodes with varying degrees using adaptive neighbor weights.

- Input features: $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n\}, \quad \vec{h}_i \in \mathbb{R}^{\mathbf{F}}$
- Shared linear transformation: $\mathbf{W} \in \mathbb{R}^{\mathbf{F}' \times \mathbf{F}}$

# Self-Attention Mechanism in GAT

**Step 1: Linear Transformation**

$$\vec{h}'_i = \mathbf{W}\vec{h}_i \quad \forall i \in \mathcal{V}$$

**Step 2: Compute Attention Coefficients**

$$s_{ij} = a(\vec{h}'_i, \vec{h}'_j)$$

Where $a : \mathbb{R}^{\mathbf{F}'} \times \mathbb{R}^{\mathbf{F}'} \to \mathbb{R}$

**Popular Choices for Attention Scoring:**

- **Dot Product:** $a(\vec{h}'_i, \vec{h}'_j) = (\vec{h}'_i)^\top \vec{h}'_j$
- **Additive:**

$$a(\vec{h}'_i, \vec{h}'_j) = \mathrm{LeakyReLU}(\mathbf{a}^\top[\vec{h}'_i \| \vec{h}'_j])$$

# Attention Score Matrix **S**

- The matrix $\mathbf{S} \in \mathbb{R}^{n \times n}$ contains raw attention scores: $s_{ij} = a(\vec{h}'_i, \vec{h}'_j)$
- These scores indicate the importance of node $j$ to node $i$ based on transformed features.

So:

$$\mathbf{H}_{new} = \sigma(S.\mathbf{H}')$$

**Why not apply S directly?**

- **S** is a matrix of **unnormalized scores** — directly using it can lead to unstable and unbounded outputs.
- If we set $s'_{ij} = \frac{\exp(s_{ij})}{\sum_{i=1}^{n} \exp(s_{ik})}$ will have:

$$\mathbf{H}_{new} = \sigma(S'.\mathbf{H}')$$

# Attention Score Matrix $\mathbf{S}$

- It does not respect the graph structure — it includes all node-to-node interactions unless masked.
- It may cause unrelated nodes to influence each other.
- Mask scores outside neighborhood $\mathcal{N}_i$

$$\mathbf{s_{ij}}' = \frac{\exp(s_{ij})}{\sum_{k \in \mathcal{N}_i \cup i} \exp(s_{ik})}$$

- Resulting matrix $\boldsymbol{S}$ is row-stochastic (i.e., values sum to 1 per row)

# Masked Attention in Matrix Form

**Final Attention Mechanism with Graph Structure:**

- M = A+I
- Softmax over masked positions requires attention scores to be set to $-\infty$ for excluded elements
- Zero values in M set to $-\infty$
- Apply masking before softmax:

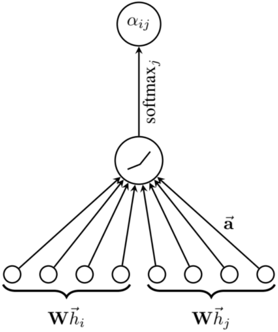
$$\tilde{\mathbf{S}} = \text{softmax}\left(\mathbf{S} \odot \mathbf{M}\right)$$

- $\odot$: element-wise multiplication (masking)

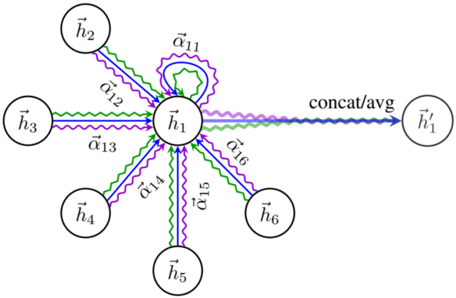
**Final Update Rule:**

$$\mathbf{H}_{\text{new}} = \sigma\left(\tilde{\mathbf{S}}\mathbf{H}'\right)$$

# GAT



$\alpha_{ij}$
softmax$_j$
$\vec{\mathbf{a}}$
$\mathbf{W}\vec{h}_i$
$\mathbf{W}\vec{h}_j$

$\vec{h}_1$
$\vec{h}_2$
$\vec{h}_3$
$\vec{h}_4$
$\vec{h}_5$
$\vec{h}_6$
$\vec{h}'_1$
$\vec{\alpha}_{11}$
$\vec{\alpha}_{12}$
$\vec{\alpha}_{13}$
$\vec{\alpha}_{14}$
$\vec{\alpha}_{15}$
$\vec{\alpha}_{16}$
concat/avg

# Training GNNs: Supervised and Unsupervised Settings

**What if we don't have any labels? (Unsupervised Learning)**

- Use node features and graph structure to learn useful representations.

- **One possible idea:** *"Similar" nodes should have similar embeddings.*

$$\min_{W} \mathcal{L} = \sum_{u,v} \mathrm{CE}(y_{u,v}, \langle \vec{h}_v, \vec{h}_u \rangle)$$

- $y_{u,v} = 1$ if node $u$ and $v$ are **similar**
- $\langle \vec{h}_v, \vec{h}_u \rangle$: **similarity of embeddings**

- **Node Similarity** can be:
  - edges
  - Random walk distance

# Supervised GNN Training: Node Classification

**Task:** Predict a label $y_i$ for each node $i \in \mathcal{V}$

**Approach:**

- Use GNN to compute node embeddings $\vec{h}_i$
- Apply a softmax classifier on each embedding

**Loss Function: Cross-Entropy**

$$\mathcal{L} = -\sum_{i \in \mathcal{V}_{\text{labeled}}} \sum_{c=1}^{C} y_{ic} \log \hat{y}_{ic}$$

where $\hat{y}_{ic} = \text{softmax}(\mathbf{W}\vec{h}_i)$

# Supervised GNN Training: Graph Classification

**Task:** Predict a label for the entire graph $G$

**Approach:**

- Compute node embeddings $\vec{h}_v$
- Aggregate (e.g., mean, sum, attention) to form graph embedding $\vec{h}_G$
- Apply a classifier on $\vec{h}_G$

**Loss Function: Cross-Entropy (for classification)**

$$\mathcal{L} = -\sum_{G \in \mathcal{D}} \sum_{c=1}^{C} y_{Gc} \log \hat{y}_{Gc}$$

where $\hat{y}_{Gc} = \text{softmax}(\mathbf{W}\vec{h}_G)$

# Supervised GNN Training: Link Prediction

**Task:** Predict whether an edge exists between a node pair $(u, v)$

**Approach:**

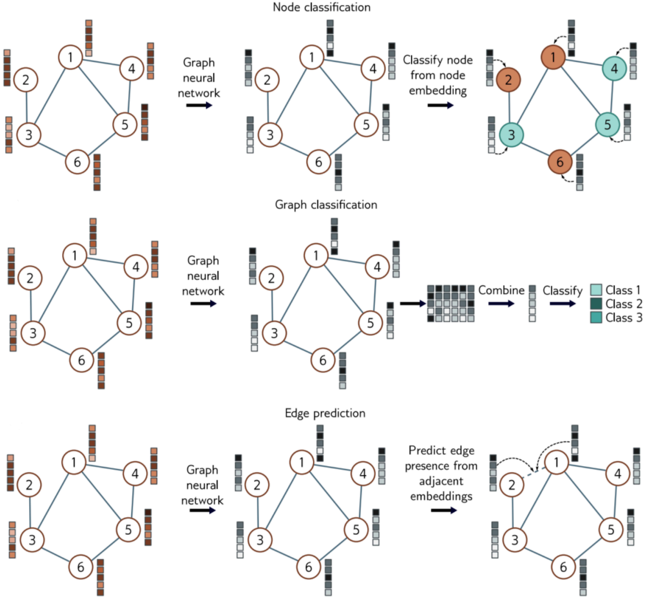
- Use GNN to compute embeddings $\vec{h}_u$, $\vec{h}_v$
- Predict link score using dot product or MLP:

$$\hat{y}_{uv} = \sigma(\vec{h}_u^\top \vec{h}_v) \quad \text{or} \quad \mathrm{MLP}([\vec{h}_u \| \vec{h}_v])$$

**Loss Function: Binary Cross-Entropy**

$$\mathcal{L} = -\sum_{(u,v)} y_{uv} \log \hat{y}_{uv} + (1 - y_{uv}) \log(1 - \hat{y}_{uv})$$

# Visualization of Tasks



# Semi-Supervised Learning

**Definition:** Learning from a dataset that contains both **labeled** and **unlabeled** examples.

**Occurs When:**

- Labels are expensive or time-consuming to obtain.
- Large amounts of raw (unlabeled) data are available.

**Objective:**

- Use unlabeled data to improve generalization.
- Learn representations that respect both labels and data structure.

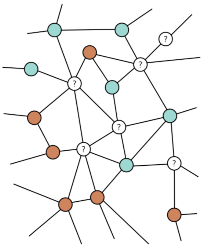
# Semi-Supervised Learning in Graphs

**Problem Setup:**

- A graph $G = (\mathcal{V}, \mathcal{E})$ with node features.
- Only a subset of nodes $\mathcal{V}_L \subset \mathcal{V}$ are labeled.

**Key Idea:**

- Use both graph structure and node features to propagate labels.
- Unlabeled nodes benefit from neighboring labeled information via message passing.



# Inductive vs. Transductive Learning in GNNs

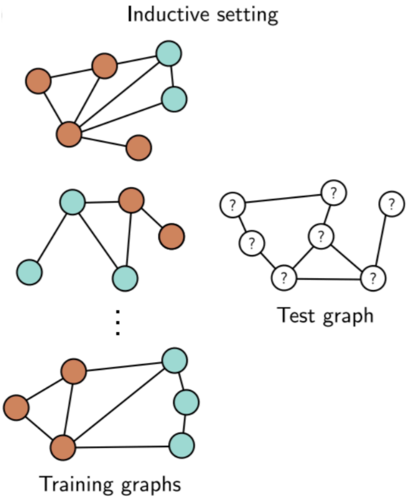
**Inductive Learning:**

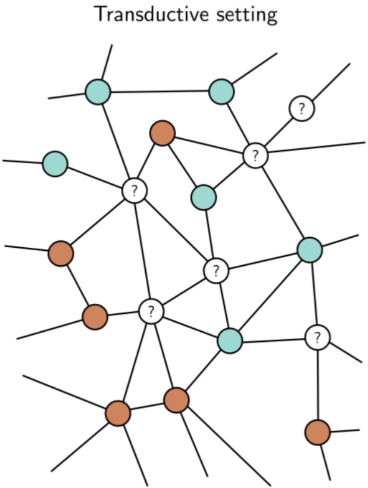
- Learns a general rule from labeled training data that maps inputs to outputs.
- Once trained, the model can be applied to **new, unseen data**.
- This is the default in most machine learning settings.

**Transductive Learning:**

- Considers both labeled and unlabeled data **simultaneously** during training.
- Does not learn a reusable rule — instead directly infers labels for the current test nodes.
- Can exploit patterns in unlabeled data, but must be retrained if new data are added.

# Inductive vs. Transductive Learning in GNNs



# Training Large Graphs in Batches

**Challenge:** Large graphs may not fit into memory, making full-graph training impractical.

**Mini-batch Training Strategies:**

- **Layer-wise sampling:** Sample fixed-size sets of neighbors per GNN layer.
- **Graph partitioning:** Cluster the original graph into disjoint subsets of nodes.

# Graph Partitioning

**Goal:** Break down a large graph into smaller, more manageable subgraphs for mini-batch training.

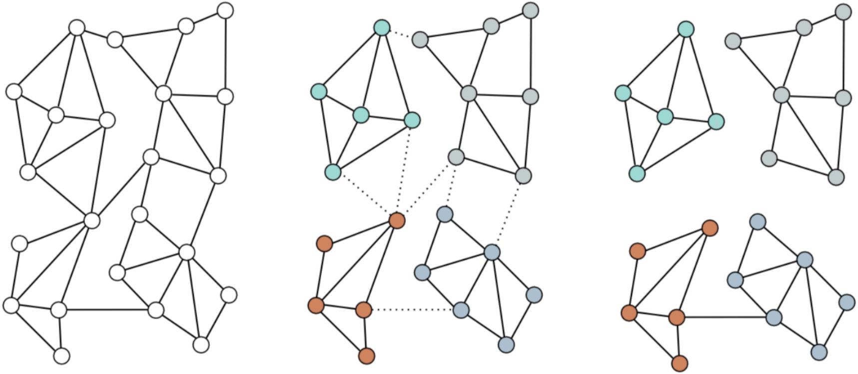
**How it works:**

- Cluster the original graph into disjoint subsets of nodes.
- Each subset becomes a smaller subgraph (a "partition") with many internal edges.

**Mini-batch Strategy:**

- Treat each partition as a separate training batch.
- Optionally combine multiple partitions in a batch, reintroducing inter-partition edges if needed.

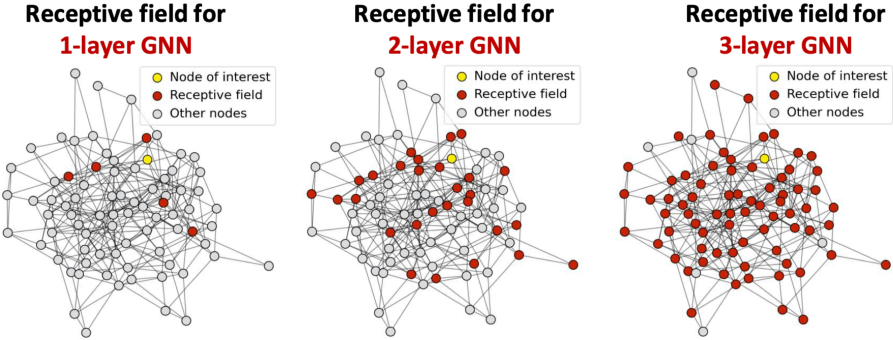
# Graph Partitioning



# Over-Smoothing in GNNs

**Problem:**

- After multiple layers of message passing, node embeddings tend to become very similar.
- This phenomenon is called **over-smoothing**.
- It limits the expressive power of deep GNNs.



# Over-Smoothing in GNNs

**Why is it bad?**

- Node features lose discriminative power.
- The model can no longer distinguish between nodes with different labels or roles.
- This **limits the depth** of the network.

**How to mitigate it:**

- **Residual connections:** Preserve original features and stabilize training.
- **Jumping Knowledge connections:** Let the output layer aggregate features from all earlier layers.