

ITCS 6156/8156 Fall 2024

Machine Learning

Graph Neural Network

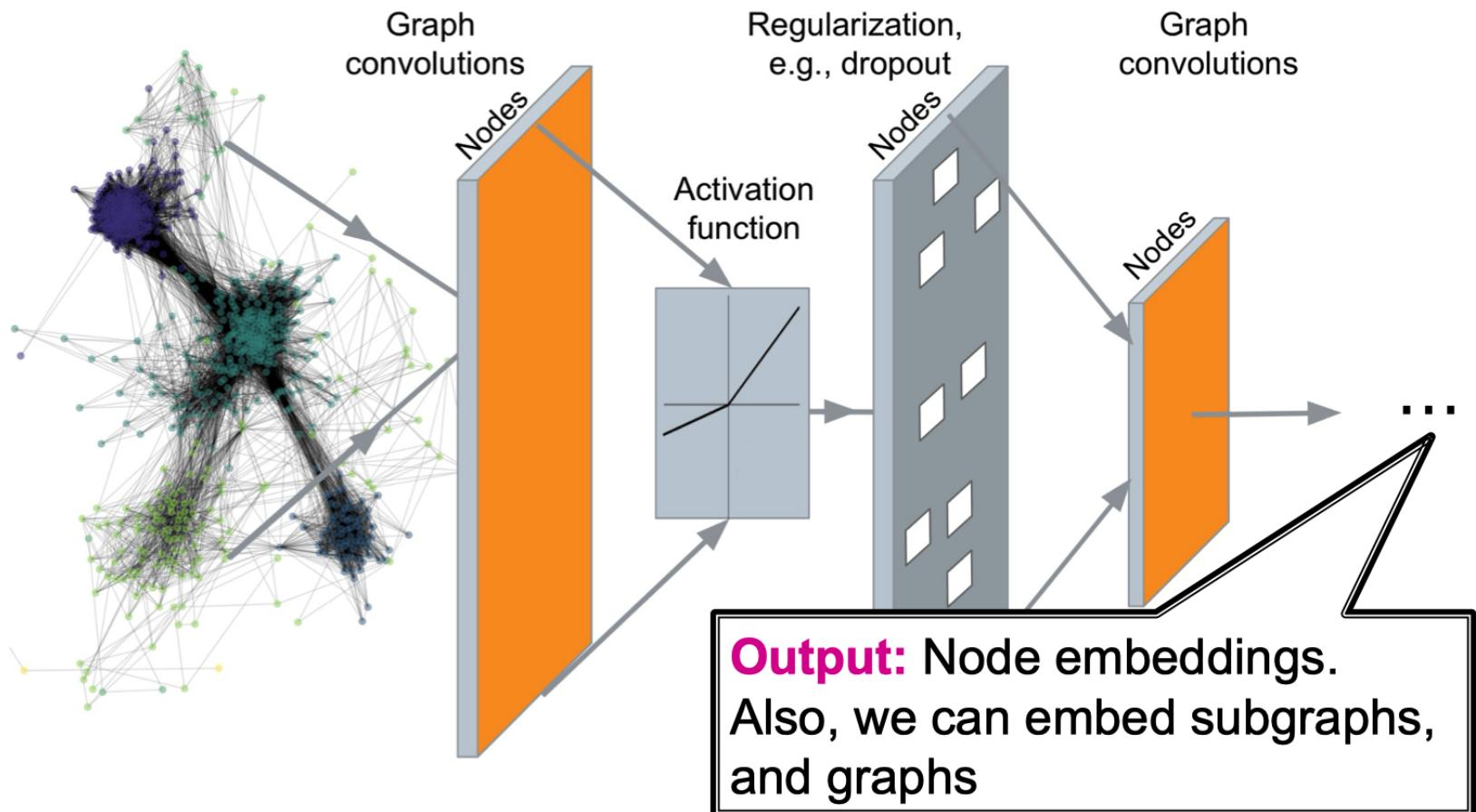
Instructor: Hongfei Xue

Email: hongfei.xue@charlotte.edu

Class Meeting: Tue & Thu, 4:00 PM – 5:15 PM, WWH 130



Deep Graph Encoders



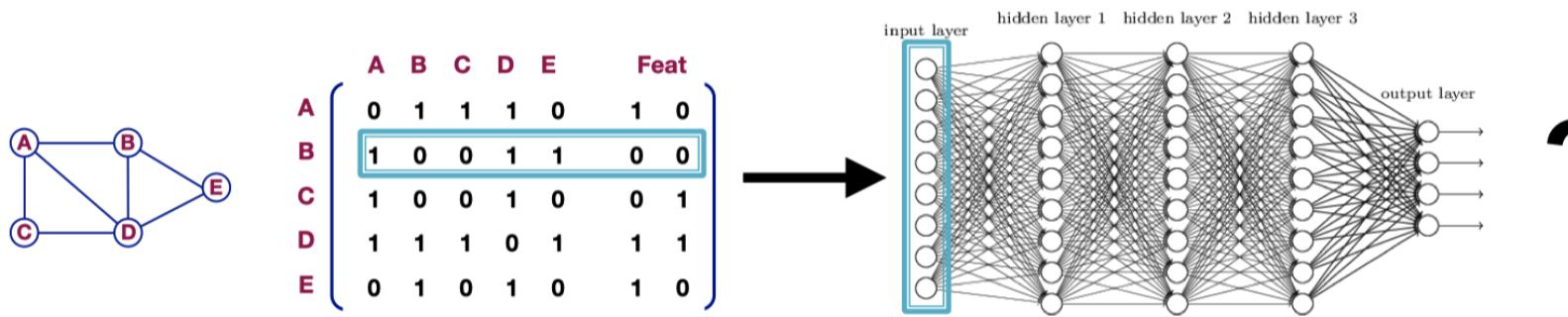
Setup

■ Assume we have a graph G :

- V is the **vertex set**
- A is the **adjacency matrix** (assume binary)
- $X \in \mathbb{R}^{|V| \times m}$ is a matrix of **node features**
- v : a node in V ; $N(v)$: the set of neighbors of v .
- **Node features:**
 - Social networks: User profile, User image
 - Biological networks: Gene expression profiles, gene functional information
 - When there is no node feature in the graph dataset:
 - Indicator vectors (one-hot encoding of a node)
 - Vector of constant 1: $[1, 1, \dots, 1]$

A Naive Approach

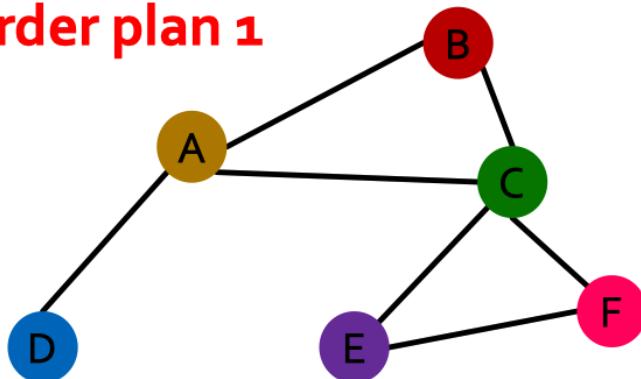
- Join adjacency matrix and features
- Feed them into a deep neural net:



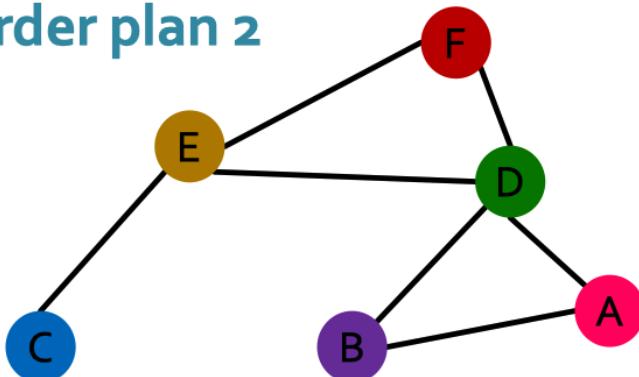
- Issues with this idea:
 - $O(|V|)$ parameters
 - Not applicable to graphs of different sizes
 - Sensitive to node ordering

Sensitive to Node Ordering

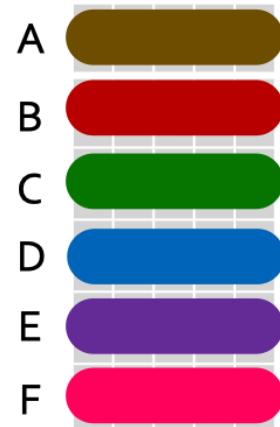
Order plan 1



Order plan 2



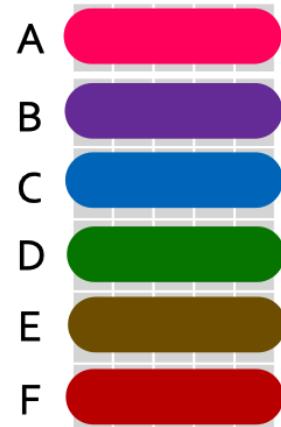
Node features X_1



Adjacency matrix A_1

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 1 | 0 | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 0 | 0 | 1 | 1 |
| C | 1 | 0 | 1 | 1 | 1 | 0 |
| D | 1 | 0 | 1 | 1 | 0 | 0 |
| E | 0 | 1 | 1 | 0 | 1 | 1 |
| F | 0 | 1 | 0 | 0 | 1 | 1 |

Node features X_2

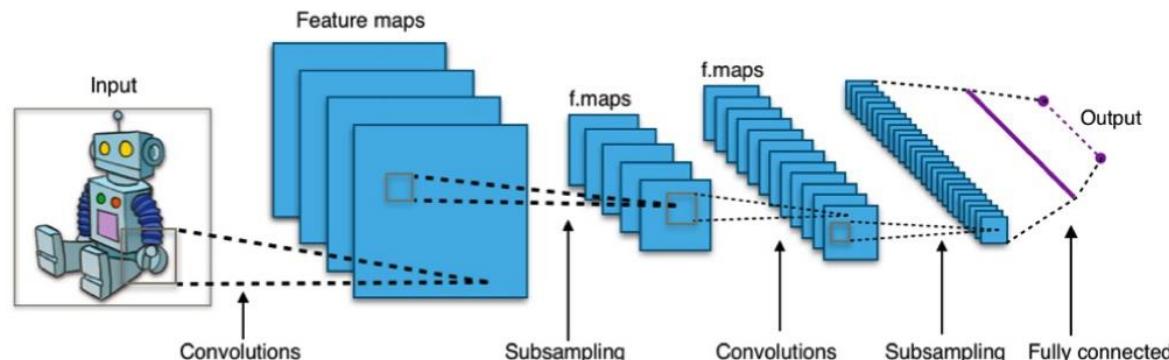
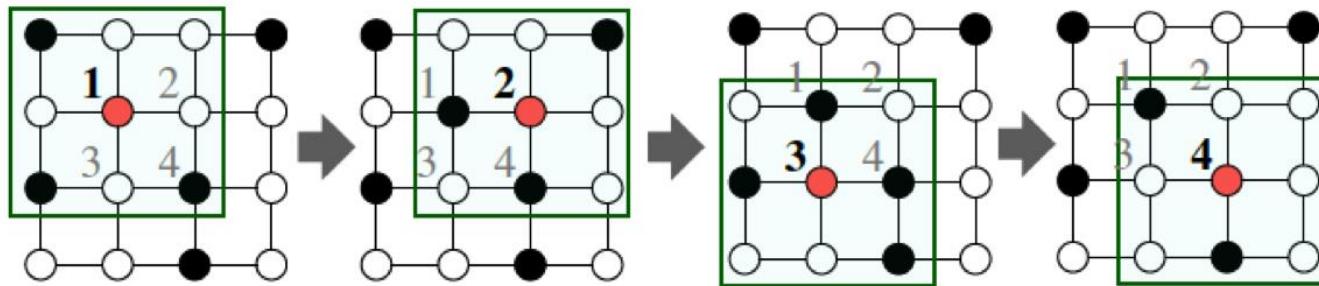


Adjacency matrix A_2

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 1 | 0 | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 0 | 0 | 1 | 1 |
| C | 1 | 0 | 1 | 1 | 1 | 0 |
| D | 1 | 0 | 1 | 1 | 0 | 0 |
| E | 0 | 1 | 1 | 0 | 1 | 1 |
| F | 0 | 1 | 0 | 0 | 1 | 1 |

Conv-based Solution?

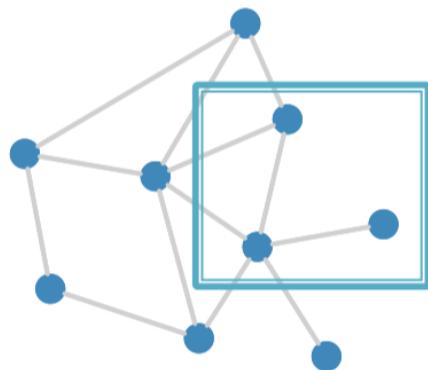
CNN on an image:



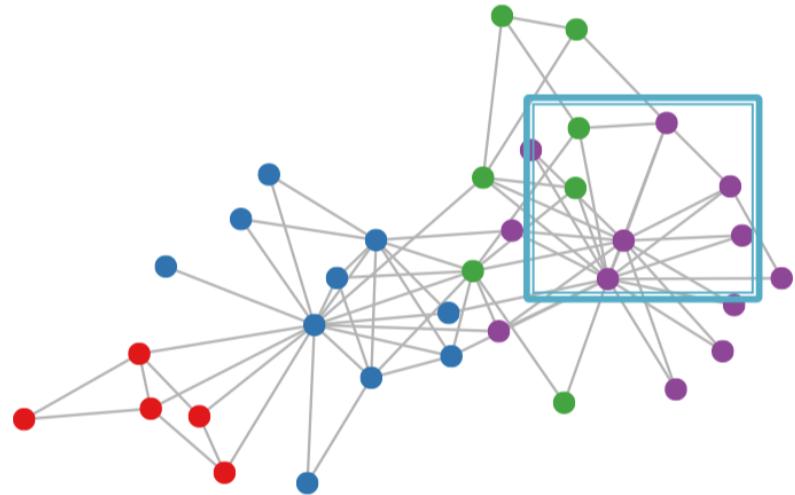
Goal is to generalize convolutions beyond simple lattices
Leverage node features/attributes (e.g., text, images)

Real-world Graphs

But our graphs look like this:



or this:

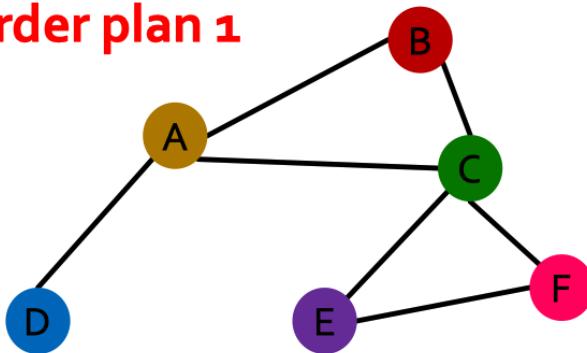


- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

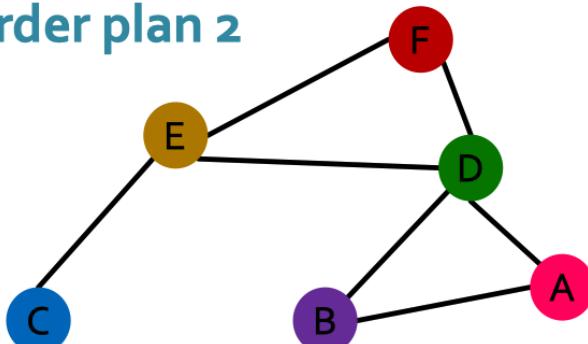
Permutation Invariance

- Graph does not have a canonical order of the nodes!

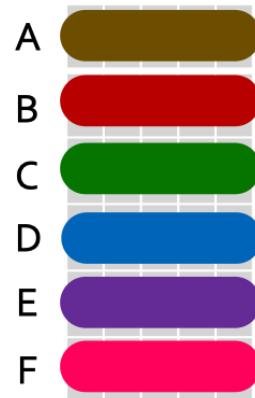
Order plan 1



Order plan 2



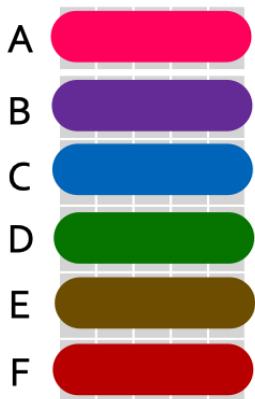
Node features X_1



Adjacency matrix A_1

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 1 | 0 | 1 | 0 | 0 | 0 |
| B | 0 | 1 | 0 | 0 | 1 | 0 |
| C | 1 | 0 | 1 | 1 | 0 | 0 |
| D | 0 | 0 | 1 | 1 | 0 | 0 |
| E | 0 | 1 | 0 | 0 | 1 | 0 |
| F | 0 | 0 | 0 | 0 | 0 | 1 |

Node features X_2



Adjacency matrix A_2

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 1 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 1 | 0 | 0 | 0 | 0 |
| C | 0 | 0 | 1 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 1 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 1 | 0 |
| F | 0 | 0 | 0 | 0 | 0 | 1 |

Graph representations should be the same for
Order plan 1 and Order plan 2.

Permutation Invariance

What does it mean by “graph representation is same for two order plans”?

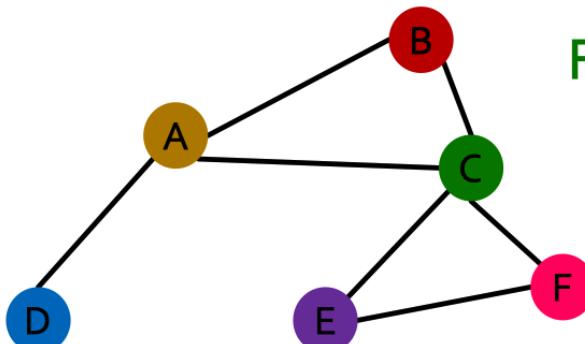
In other words, f maps a graph to a d -dim embedding

- Consider we learn a function f that maps a graph $G = (A, X)$ to a vector \mathbb{R}^d then

$$f(A_1, X_1) = f(A_2, X_2)$$

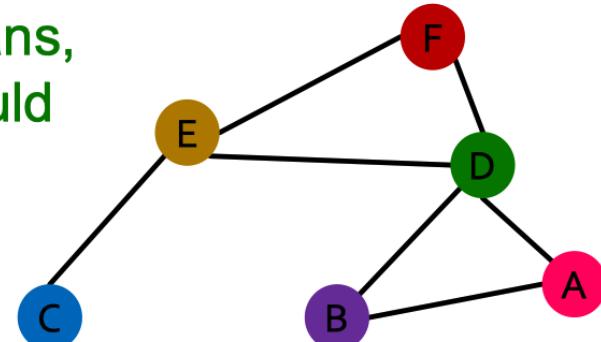
A is the adjacency matrix
 X is the node feature matrix

Order plan 1: A_1, X_1



For two order plans,
output of f should
be the same!

Order plan 2: A_2, X_2



Permutation Invariance

What does it mean by “graph representation is same for two order plans”?

- Consider we learn a function f that maps a graph $G = (A, X)$ to a vector \mathbb{R}^d .

A is the adjacency matrix
 X is the node feature matrix

- Then, if $f(A_i, X_i) = f(A_j, X_j)$ for any order plan i and j , we formally say f is a **permutation invariant function**.

For a graph with $|V|$ nodes, there are $|V|!$ different order plans.

... each node has a m -dim feature vector associated with it.

- Definition:** For any **graph** function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^d$, f is **permutation-invariant** if $f(A, X) = f(PAP^T, PX)$ for any permutation P .

d ... output embedding dimensionality of embedding the graph $G = (A, X)$

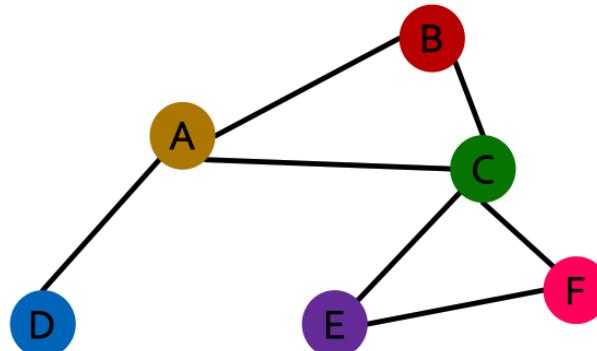
Permutation P : a shuffle of the node order
Example: (A,B,C)->(B,C,A)

Permutation Equivariance

For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{|V| \times d}$.

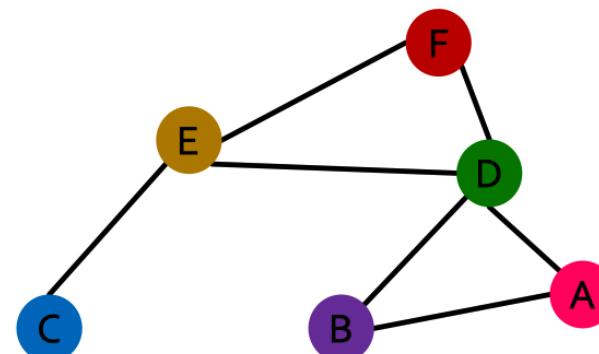
In other words, each node in V is mapped to a d -dim embedding.

Order plan 1: A_1, X_1



$$f(A_1, X_1) = \begin{matrix} & \text{A} & \text{B} \\ \text{A} & \text{brown} & \text{brown} \\ \text{B} & \text{red} & \text{red} \\ \text{C} & \text{green} & \text{green} \\ \text{D} & \text{blue} & \text{blue} \\ \text{E} & \text{purple} & \text{purple} \\ \text{F} & \text{red} & \text{red} \end{matrix}$$

Order plan 2: A_2, X_2



$$f(A_2, X_2) = \begin{matrix} & \text{A} & \text{B} \\ \text{A} & \text{red} & \text{red} \\ \text{B} & \text{purple} & \text{purple} \\ \text{C} & \text{blue} & \text{blue} \\ \text{D} & \text{green} & \text{green} \\ \text{E} & \text{brown} & \text{brown} \\ \text{F} & \text{red} & \text{red} \end{matrix}$$

For two order plans, the vector of node at the same position in the graph is the same!

Permutation Equivariance

For node representation:

- Consider we learn a function f that maps a graph $G = (A, X)$ to a matrix $\mathbb{R}^{|V| \times d}$
- If the output vector of a node at the same position in the graph remains unchanged for any order plan, we say f is **permutation equivariant**.
- **Definition:** For any **node** function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^{|V| \times d}$, f is **permutation-equivariant** if $Pf(A, X) = f(PAP^T, PX)$ for any permutation P .
 m ... each node has a m -dim feature vector associated with it.
 f maps each node in V to a d -dim embedding.

Invariance and Equivariance

- **Permutation-invariant**

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

Permute the input, the output
stays the same.
(map a graph to a vector)

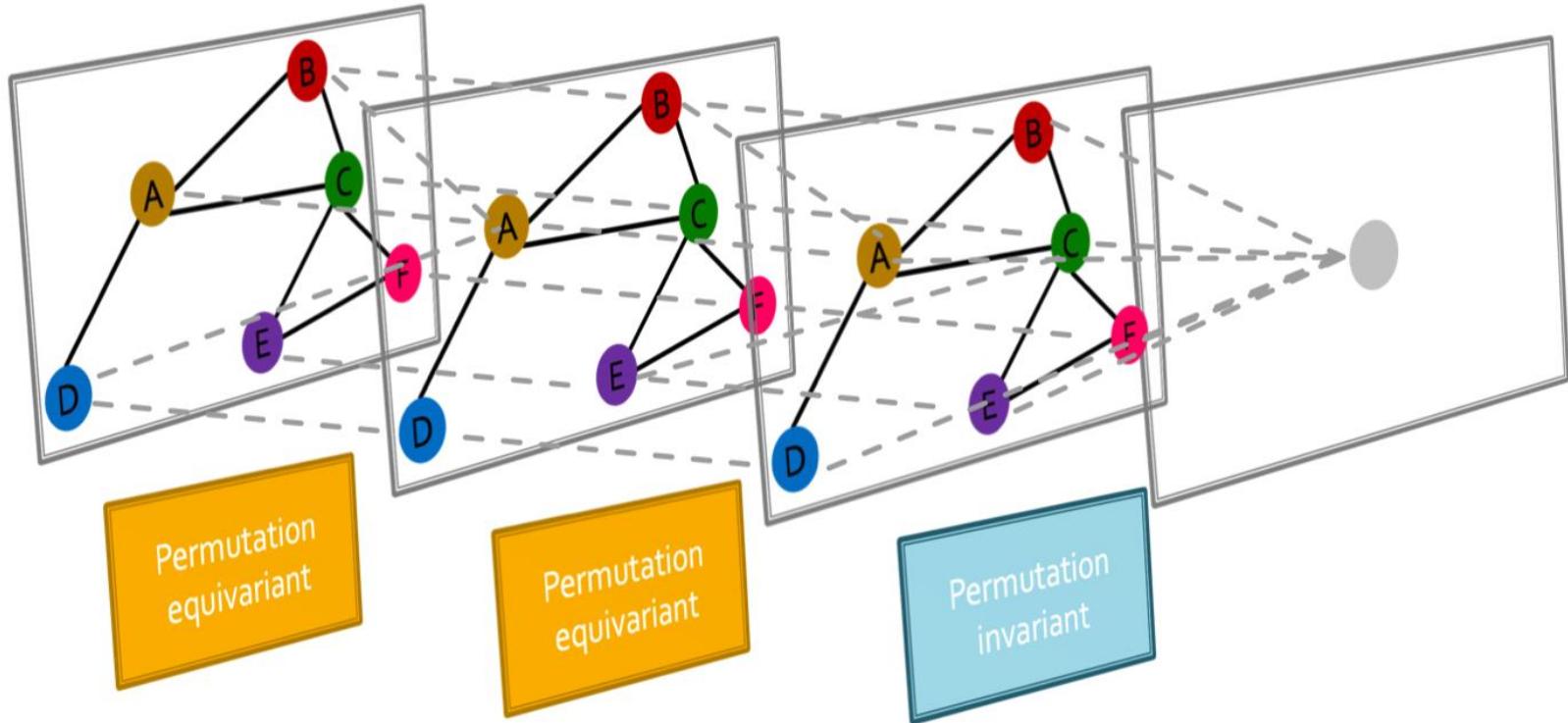
- **Permutation-equivariant**

$$\textcolor{red}{P}f(A, X) = f(PAP^T, PX)$$

Permute the input, output also
permutes accordingly.
(map a graph to a matrix)

Graph Neural Network

- Graph neural networks consist of multiple permutation equivariant / invariant functions.

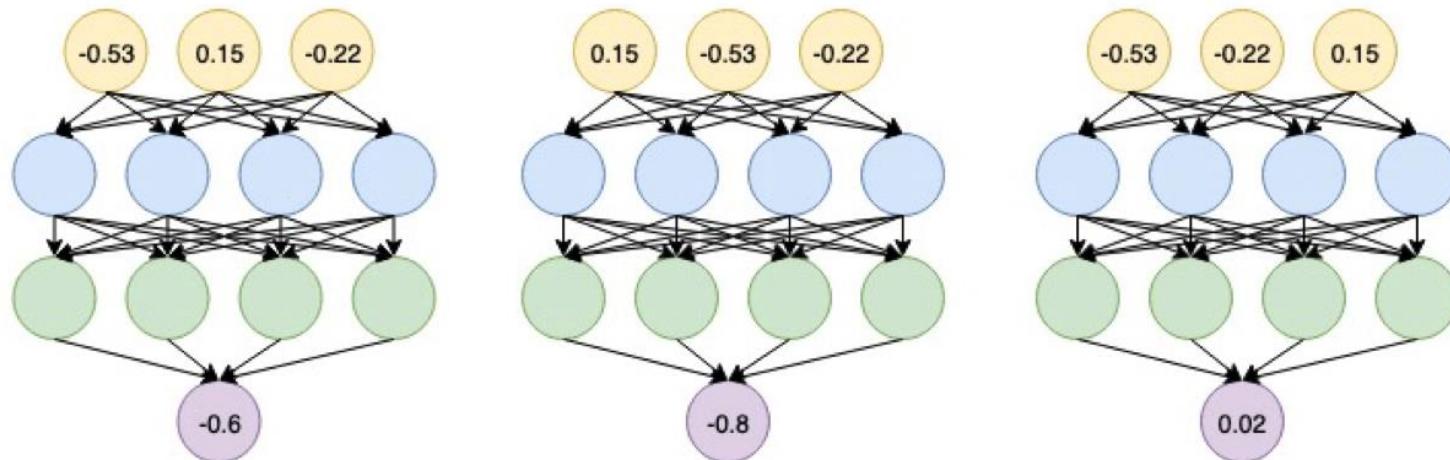


Graph Neural Network

**Are other neural network architectures, e.g.,
MLPs, permutation invariant / equivariant?**

■ **No.**

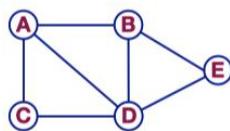
Switching the order of the
input leads to different
outputs!



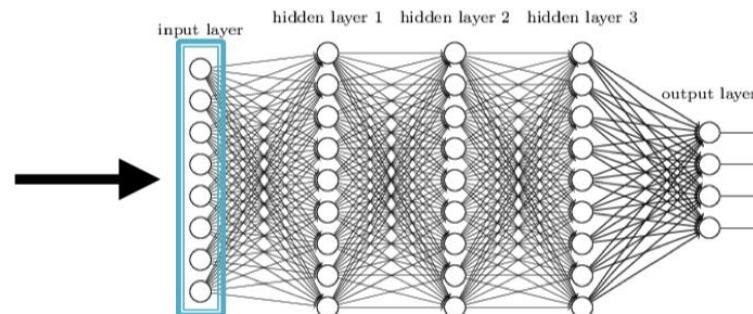
Graph Neural Network

Are other neural network architectures, e.g.,
MLPs, permutation invariant / equivariant?

- No.



| | A | B | C | D | E | Feat |
|---|---|---|---|---|---|------|
| A | 0 | 1 | 1 | 1 | 0 | 1 0 |
| B | 1 | 0 | 0 | 1 | 1 | 0 0 |
| C | 1 | 0 | 0 | 1 | 0 | 0 1 |
| D | 1 | 1 | 1 | 0 | 1 | 1 1 |
| E | 0 | 1 | 0 | 1 | 0 | 1 0 |

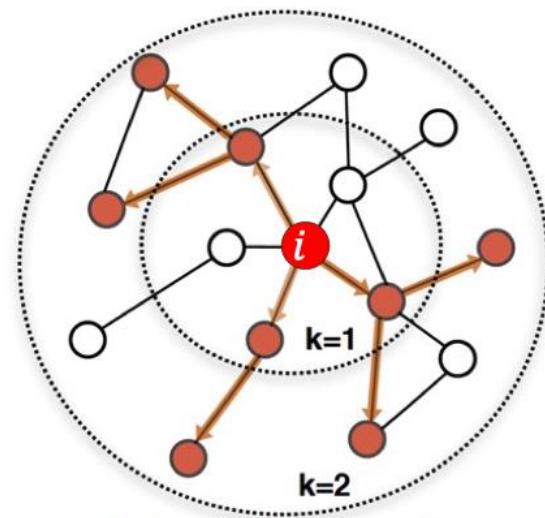


This explains why **the naïve MLP approach fails for graphs!**

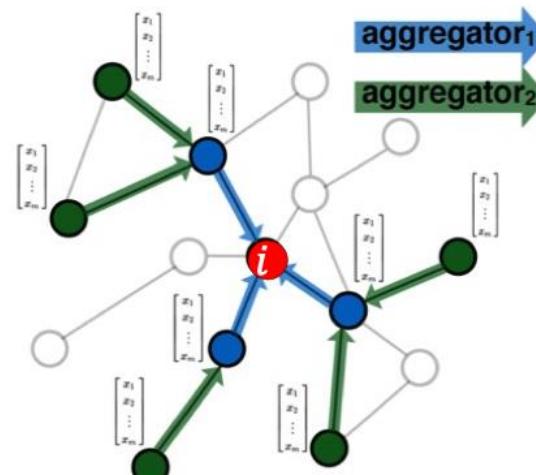
?

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

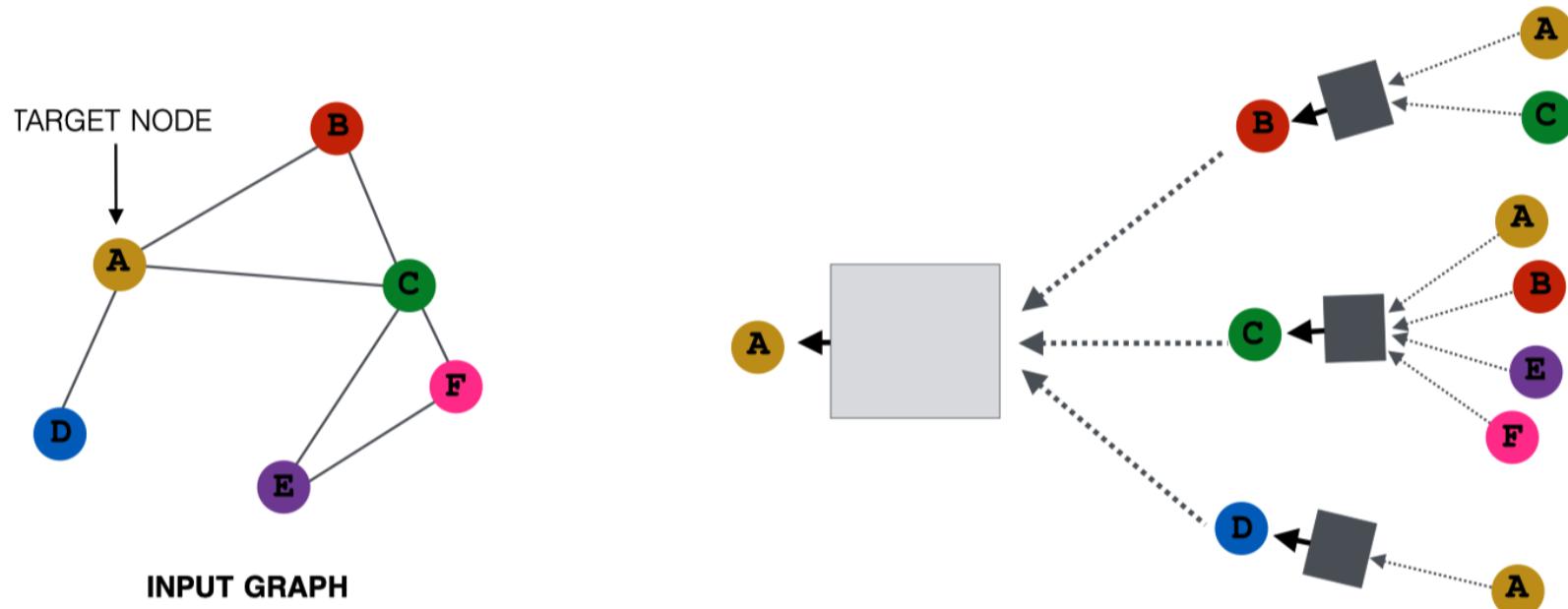


Propagate and transform information

Learn how to propagate information across the graph to compute node features

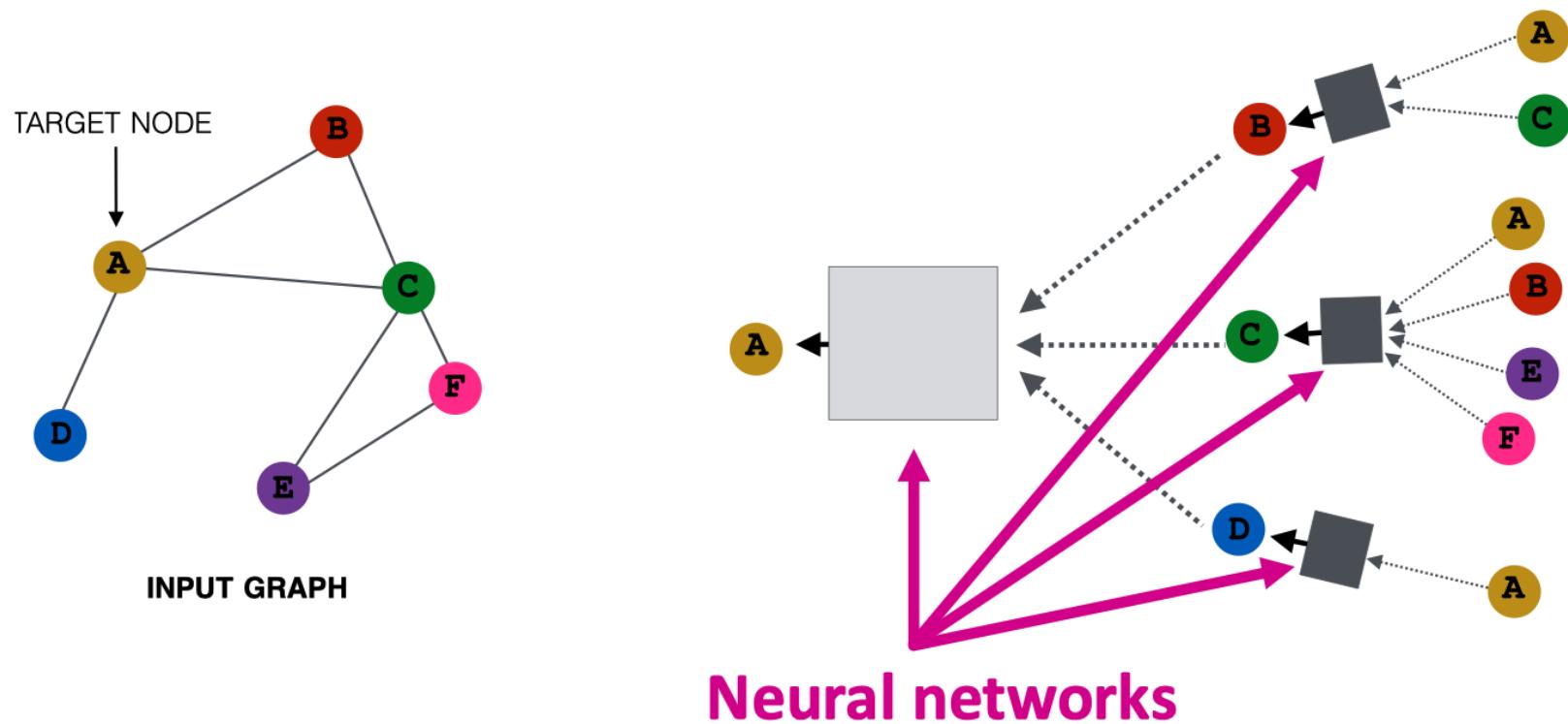
Aggregate Neighbors

- **Key idea:** Generate node embeddings based on **local network neighborhoods**



Aggregate Neighbors

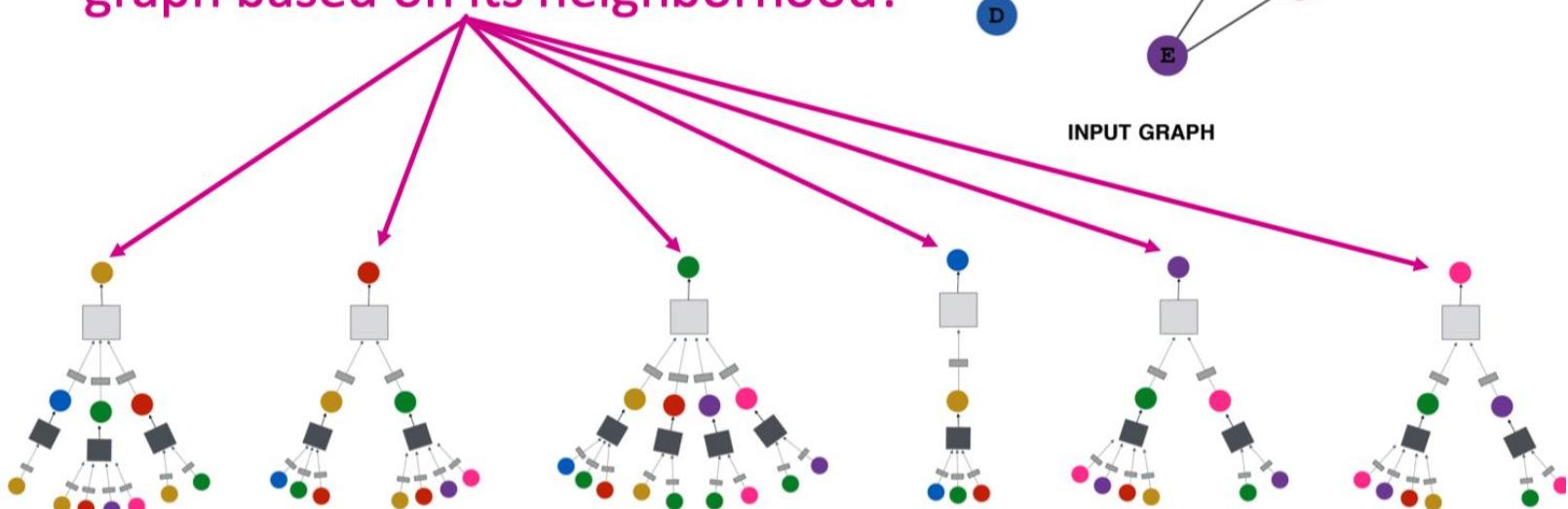
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



Aggregate Neighbors

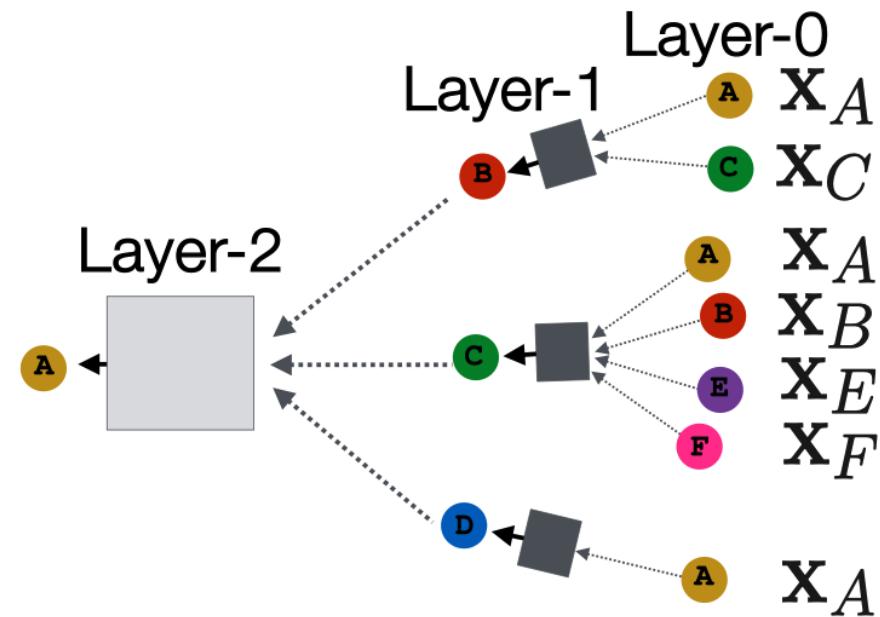
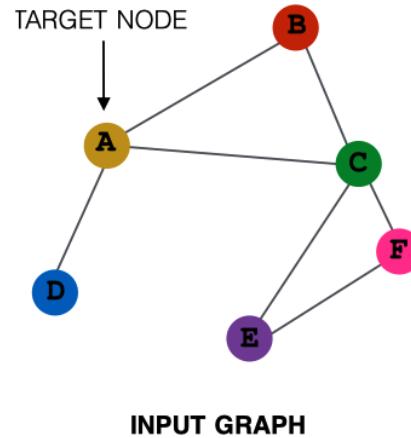
- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!



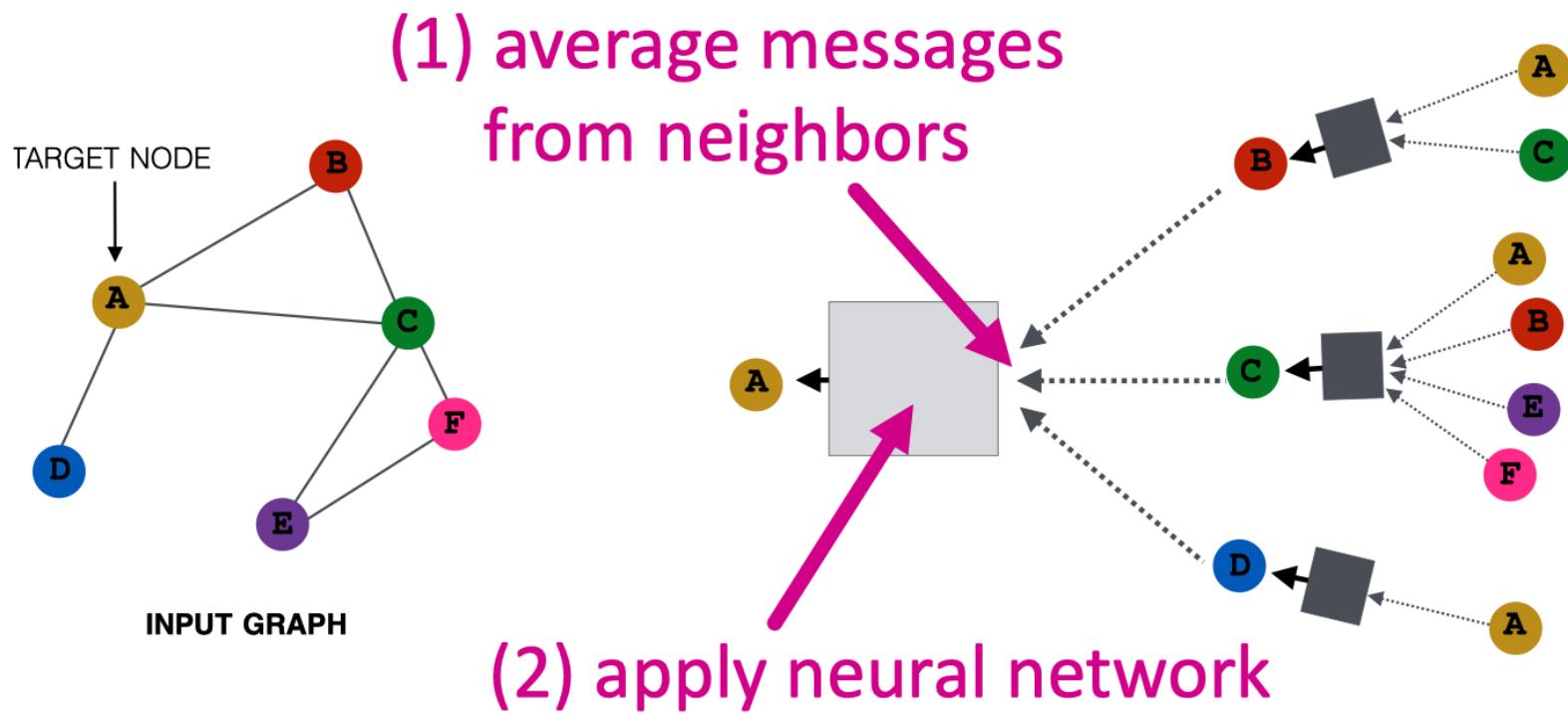
Many Layers

- Model can be **of arbitrary depth**:
 - Nodes have embeddings at each layer
 - Layer-0 embedding of node v is its input feature, x_v
 - Layer- k embedding gets information from nodes that are k hops away



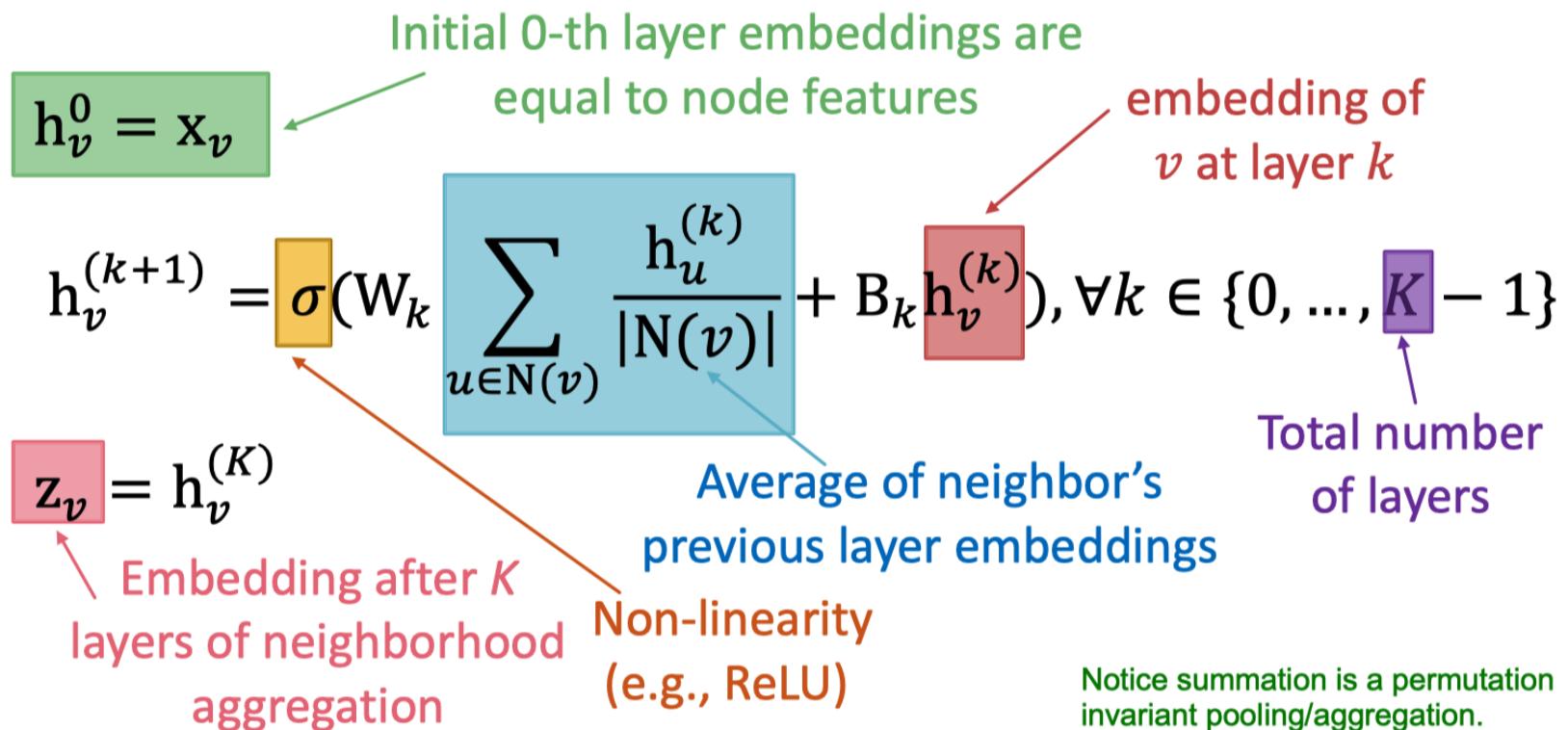
Neighborhood Aggregation

- **Basic approach:** Average information from neighbors and apply a neural network



Neighborhood Aggregation

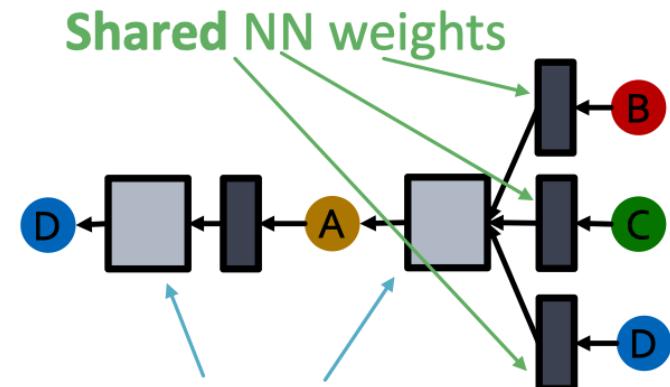
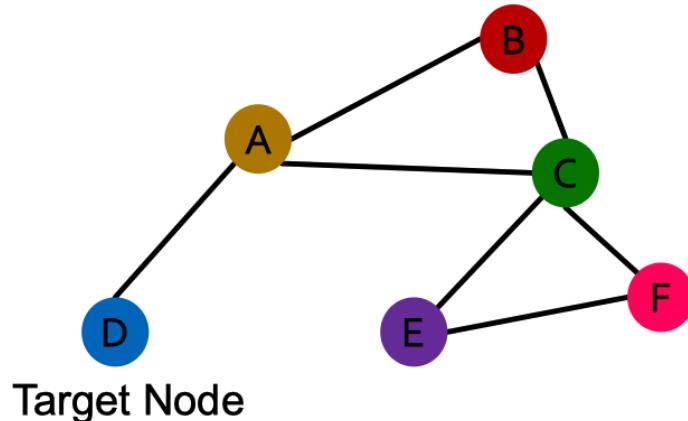
- **Basic approach:** Average neighbor messages and apply a neural network



GCN: Invariance and Equivariance

What are the **invariance and **equivariance** properties for a GCN?**

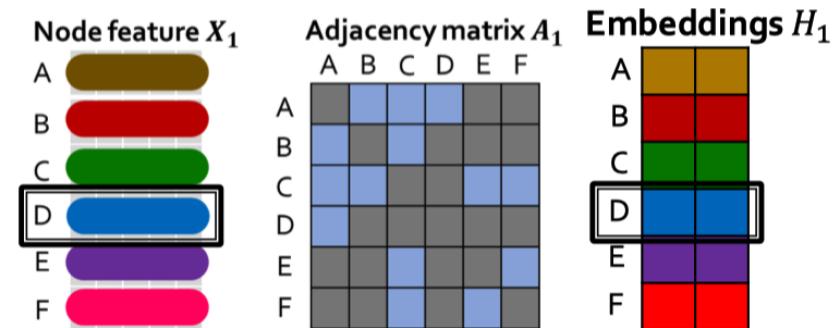
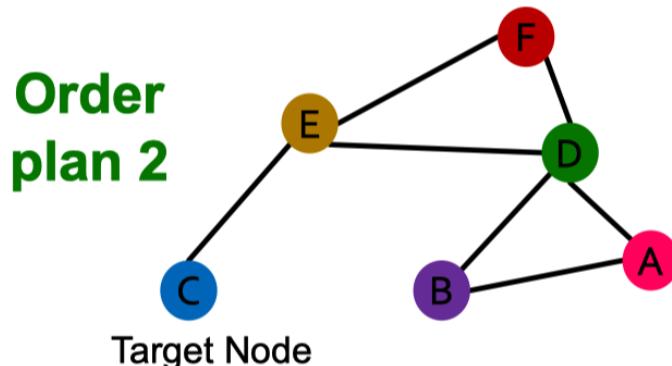
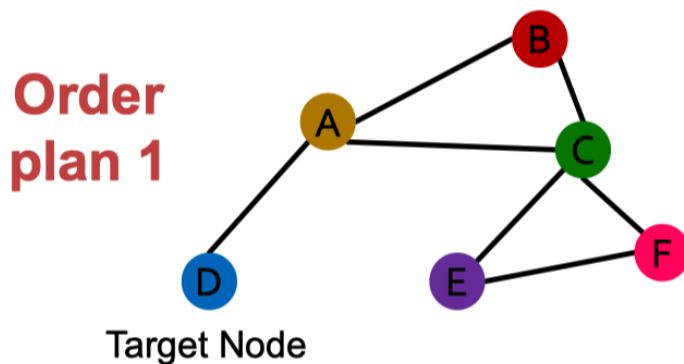
- **Given a node**, the GCN that computes its embedding is **permutation invariant**



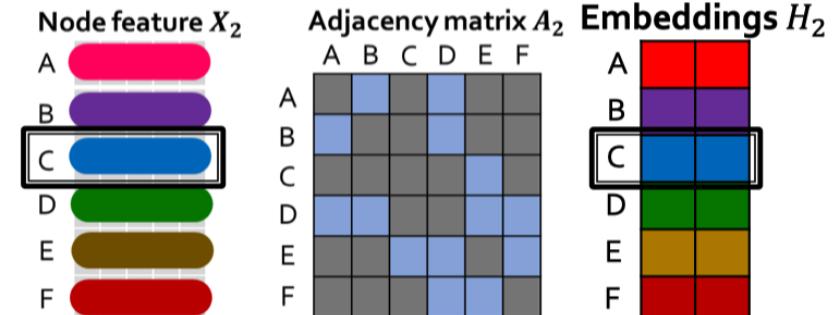
Average of neighbor's previous layer embeddings - Permutation invariant

GCN: Invariance and Equivariance

- Considering all nodes in a graph, GCN computation is **permutation equivariant**



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



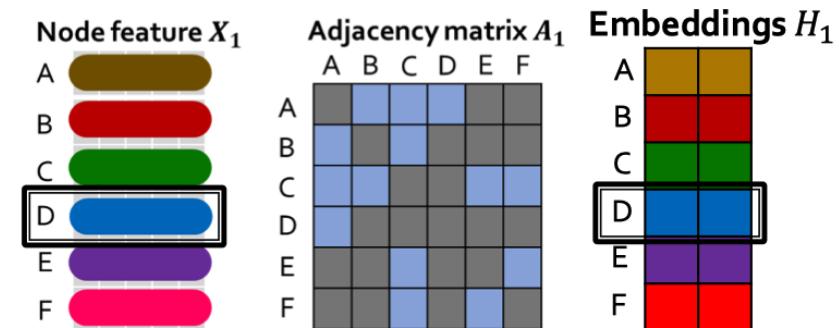
GCN: Invariance and Equivariance

- Considering all nodes in a graph, GCN computation is **permutation equivariant**

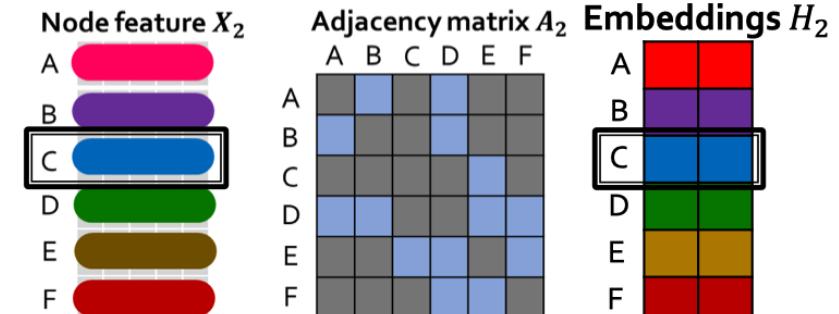
Detailed reasoning:

1. The rows of **input node features** and **output embeddings** are aligned
2. We know computing the embedding of a **given node** with GCN is **invariant**.
3. So, after permutation, the **location of a given node in the input node feature matrix** is changed, and the **the output embedding of a given node stays the same (the colors of node feature and embedding are matched)**

This is **permutation equivariant**



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



Model Parameters

Trainable weight matrices
(i.e., what we learn)

$$\begin{aligned} h_v^{(0)} &= x_v \\ h_v^{(k+1)} &= \sigma(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}), \forall k \in \{0..K-1\} \\ z_v &= h_v^{(K)} \end{aligned}$$

Final node embedding

We can feed these **embeddings into any loss function** and run SGD to **train the weight parameters**

h_v^k : the hidden representation of node v at layer k

- W_k : weight matrix for neighborhood aggregation
- B_k : weight matrix for transforming hidden vector of self

Matrix Formulation

- Many aggregations can be performed efficiently by (sparse) matrix operations

- Let $H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$

Matrix of hidden embeddings $H^{(k)}$

- Then: $\sum_{u \in N(v)} h_u^{(k)} = A_{v,:} H^{(k)}$

- Let D be diagonal matrix where

$$D_{v,v} = \text{Deg}(v) = |N(v)|$$

- The inverse of D : D^{-1} is also diagonal:

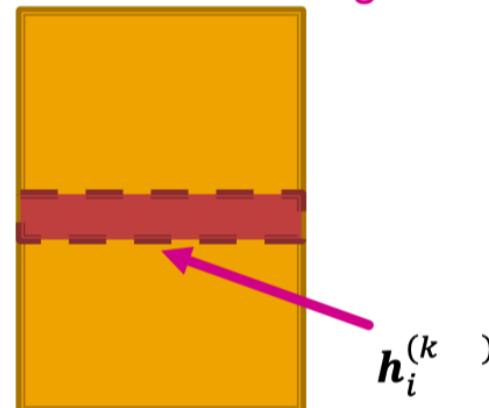
$$D_{v,v}^{-1} = 1/|N(v)|$$

- Therefore,

$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|}$$



$$H^{(k+1)} = D^{-1} A H^{(k)}$$

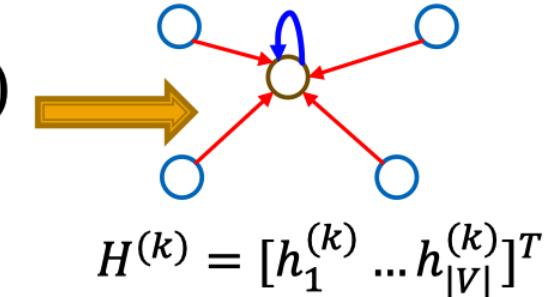


Matrix Formulation

- Re-writing update function in matrix form:

$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^T + H^{(k)}B_k^T)$$

where $\tilde{A} = D^{-1}A$



- Red: neighborhood aggregation
- Blue: self transformation
- In practice, this implies that efficient sparse matrix multiplication can be used (\tilde{A} is sparse)
- **Note:** not all GNNs can be expressed in a simple matrix form, when aggregation function is complex

Training

- Node embedding \mathbf{z}_v is a function of input graph
- **Supervised setting:** We want to minimize loss \mathcal{L} :

$$\min_{\Theta} \mathcal{L}(\mathbf{y}, f_{\Theta}(\mathbf{z}_v))$$

- \mathbf{y} : node label
- \mathcal{L} could be L2 if \mathbf{y} is real number, or cross entropy if \mathbf{y} is categorical
- **Unsupervised setting:**
 - No node label available
 - **Use the graph structure as the supervision!**

Unsupervised Training

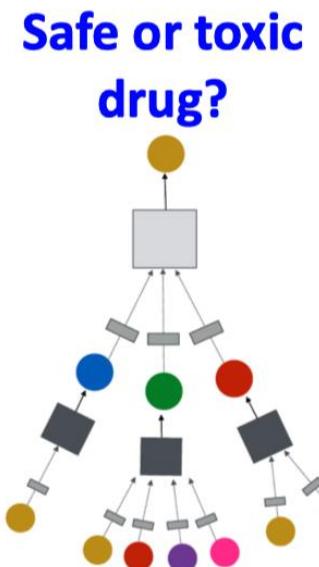
- One possible idea: “Similar” nodes have similar embeddings:

$$\min_{\Theta} \mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

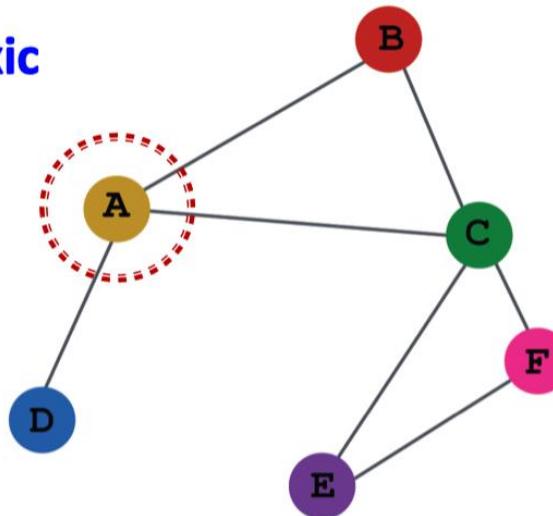
- where $y_{u,v} = 1$ when node u and v are similar
 - $z_u = f_{\Theta}(u)$ and $\text{DEC}(\cdot, \cdot)$ is the dot product
- CE is the cross entropy loss:
 - $\text{CE}(y, f(x)) = - \sum_{i=1}^C (y_i \log f_{\Theta}(x)_i)$
 - y_i and $f_{\Theta}(x)_i$ are the actual and predicted values of the i -th class.
 - Intuition: the lower the loss, the closer the prediction is to one-hot
- Node similarity can be anything from Lecture 2, e.g., a loss based on:
 - Random walks (node2vec, DeepWalk, struc2vec)
 - Matrix factorization

Supervised Training

Directly train the model for a supervised task
(e.g., **node classification**)



Safe or toxic drug?

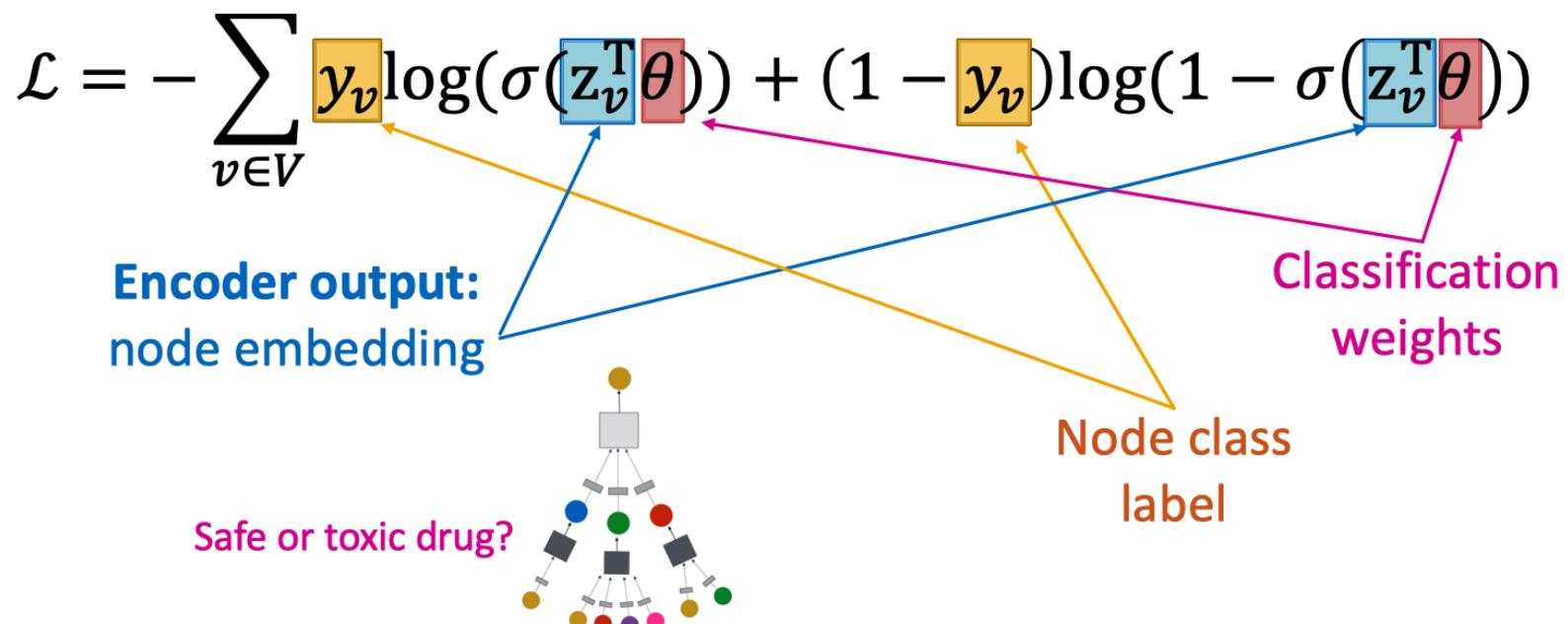


E.g., a drug-drug interaction network

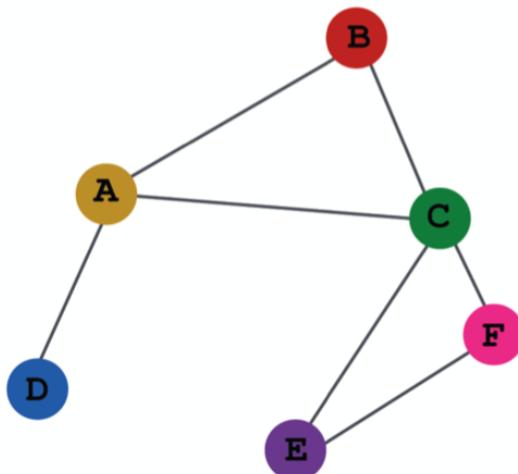
Supervised Training

Directly train the model for a supervised task
(e.g., **node classification**)

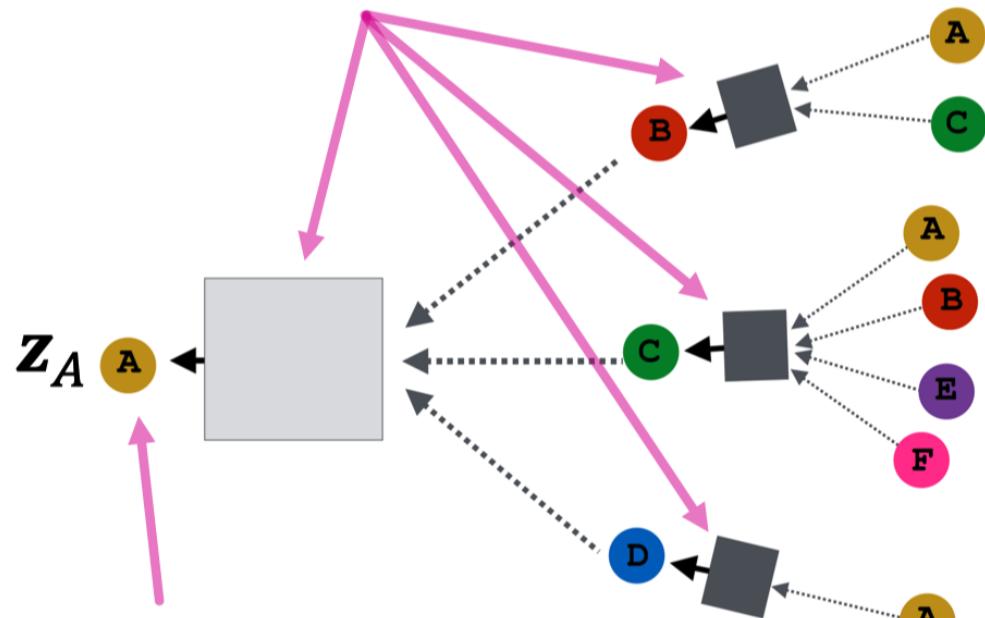
- Use cross entropy loss



Model Design: Overview

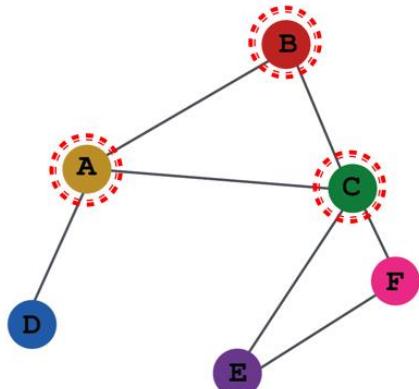


(1) Define a neighborhood aggregation function



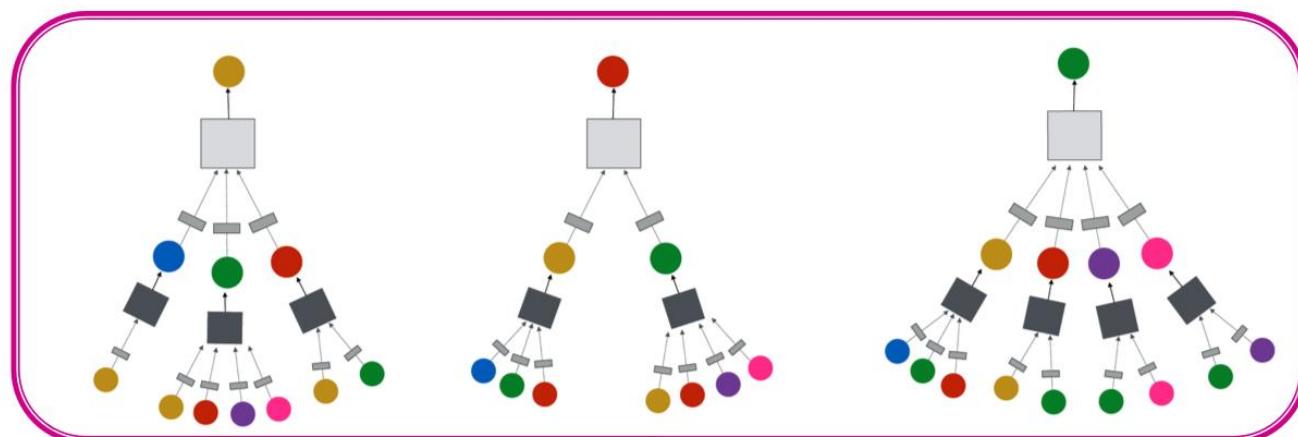
(2) Define a loss function on the embeddings

Model Design: Overview

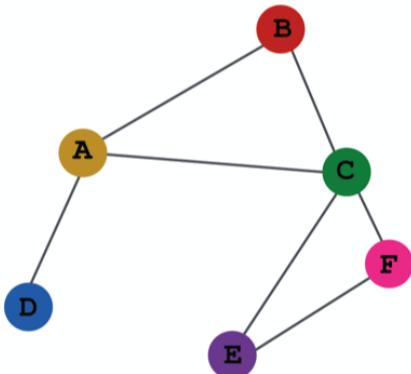


INPUT GRAPH

**(3) Train on a set of nodes, i.e.,
a batch of compute graphs**



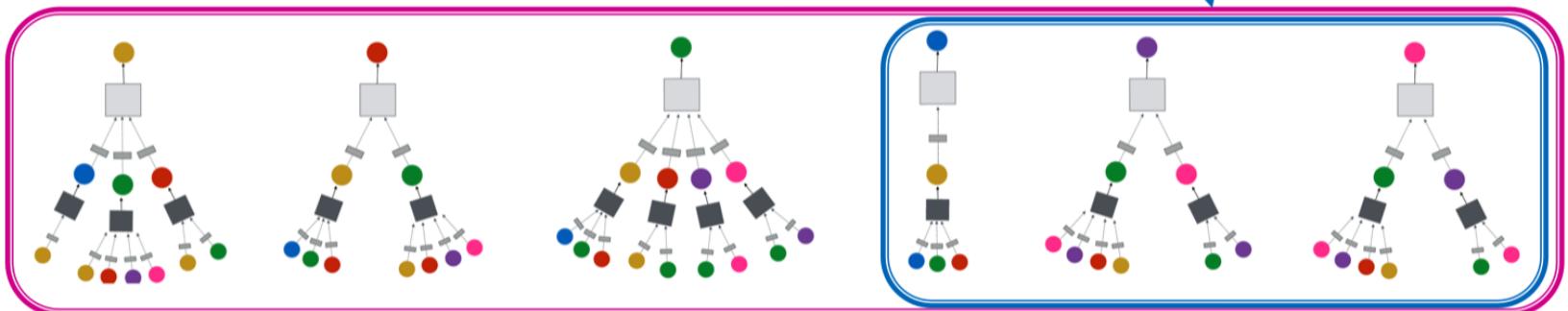
Model Design: Overview



INPUT GRAPH

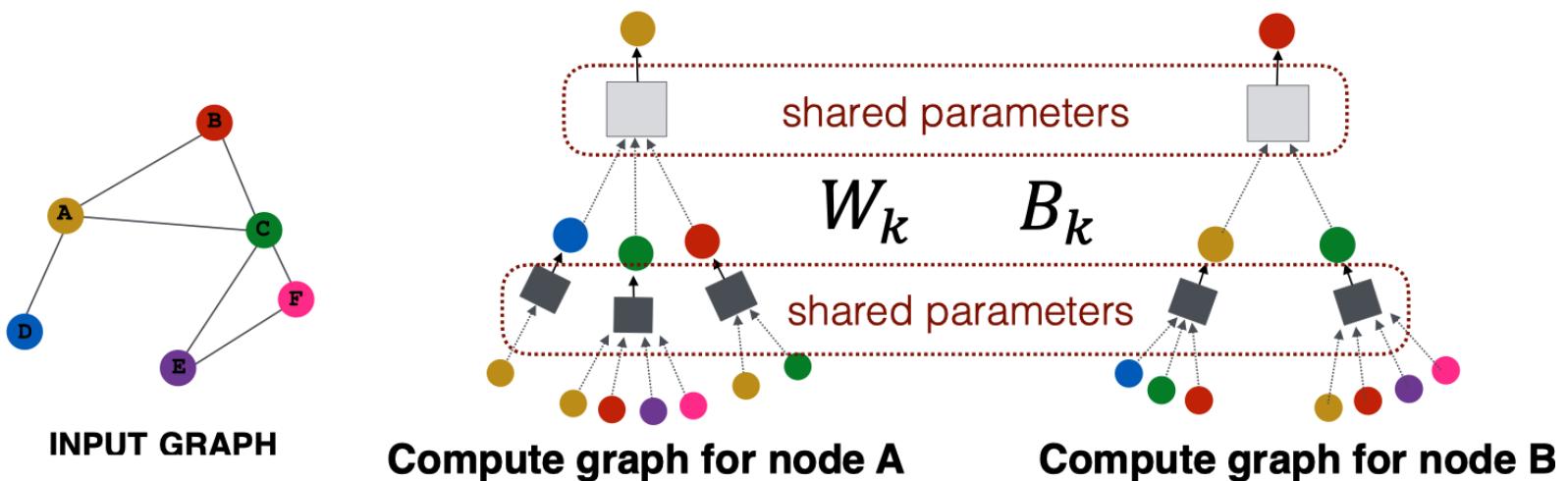
**(4) Generate embeddings
for nodes as needed**

**Even for nodes we never
trained on!**

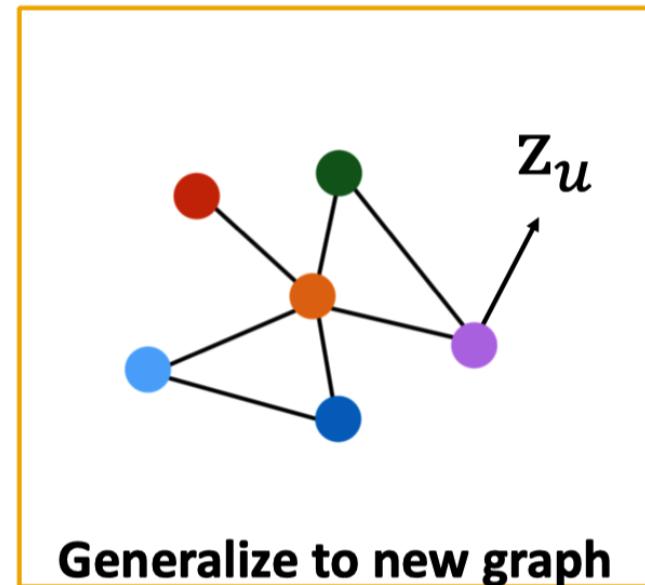
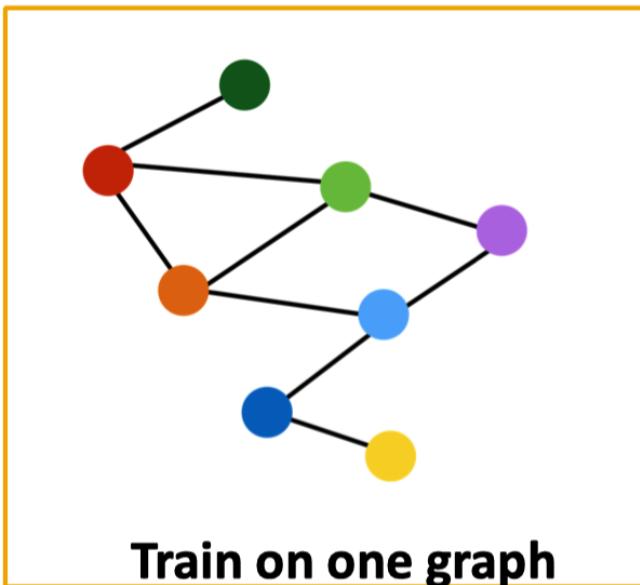


Model Design: Overview

- The same aggregation parameters are shared for all nodes:
 - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes!**



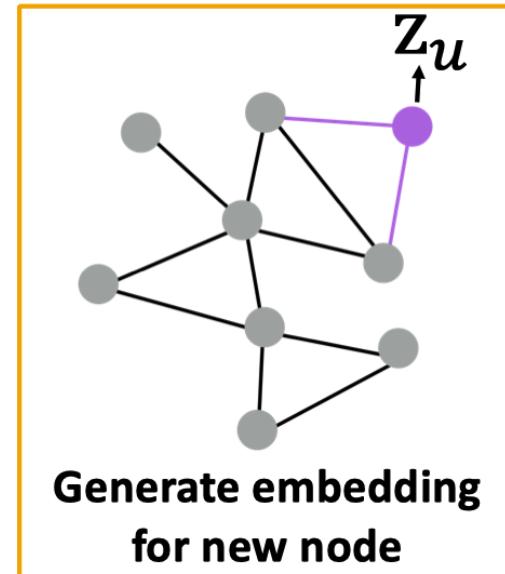
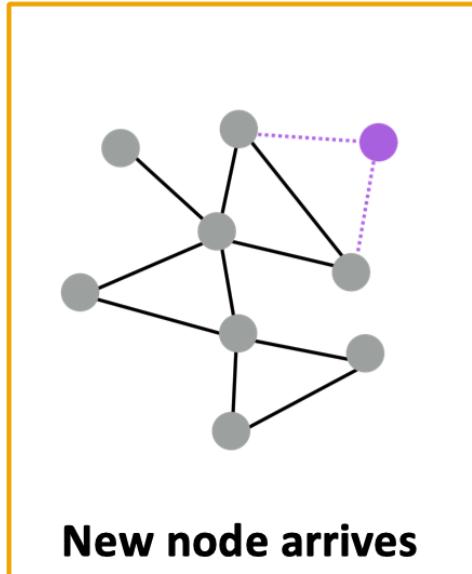
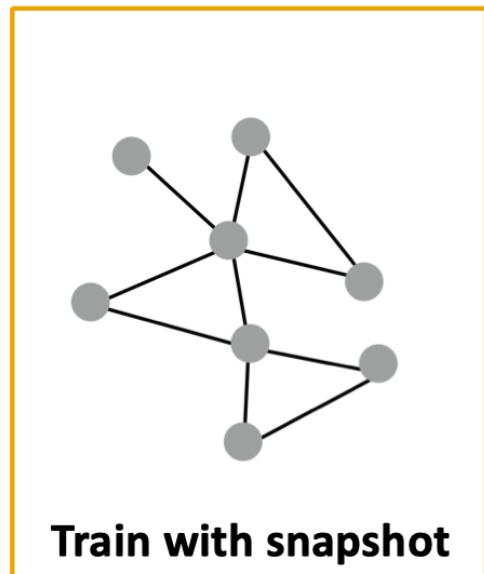
Inductive Capability: New Graphs



Inductive node embedding \rightarrow Generalize to entirely unseen graphs

E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

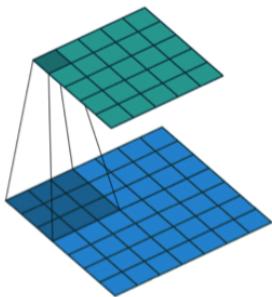
Inductive Capability: New Nodes



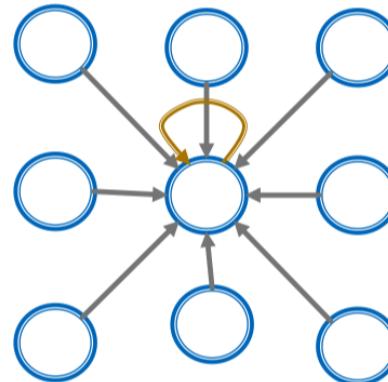
- Many application settings constantly encounter previously unseen nodes:
 - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings “on the fly”

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

$$\text{GNN formulation: } h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

Key difference: We can learn different W_l^u for different “neighbor” u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



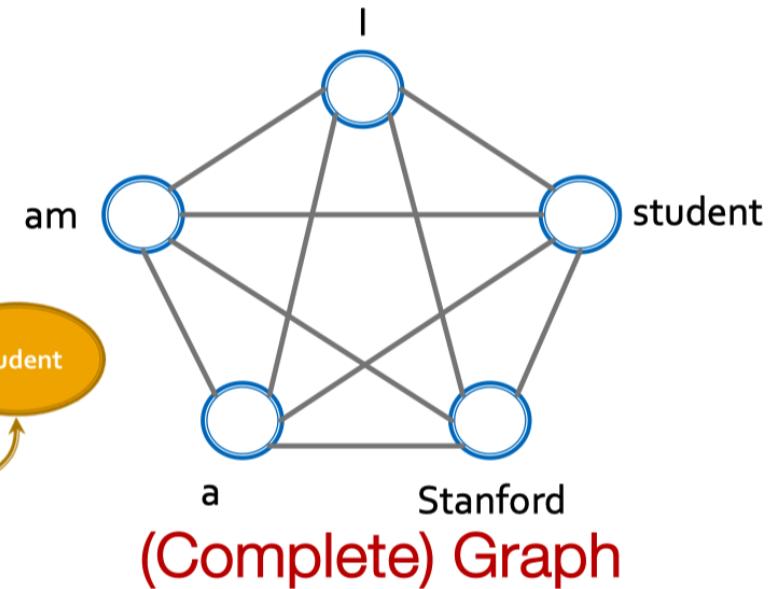
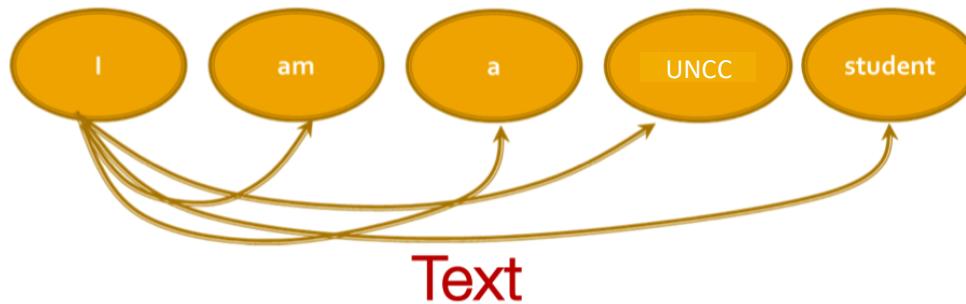
- CNN can be seen as a special GNN with fixed neighbor size and ordering:
 - The size of the filter is pre-defined for a CNN.
 - The advantage of GNN is it processes arbitrary graphs with different degrees for each node.
- CNN is not permutation invariant/equivariant.
 - Switching the order of pixels leads to different outputs.

Key difference: We can learn different W_l^u for different “neighbor” u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: $\{(-1,-1), (-1,0), (-1,1), \dots, (1,1)\}$

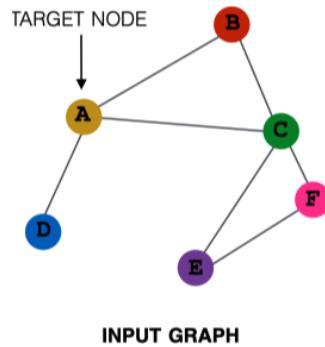
GNN vs. Transformer

Transformer layer can be seen as a special GNN that runs on a fully-connected “word” graph!

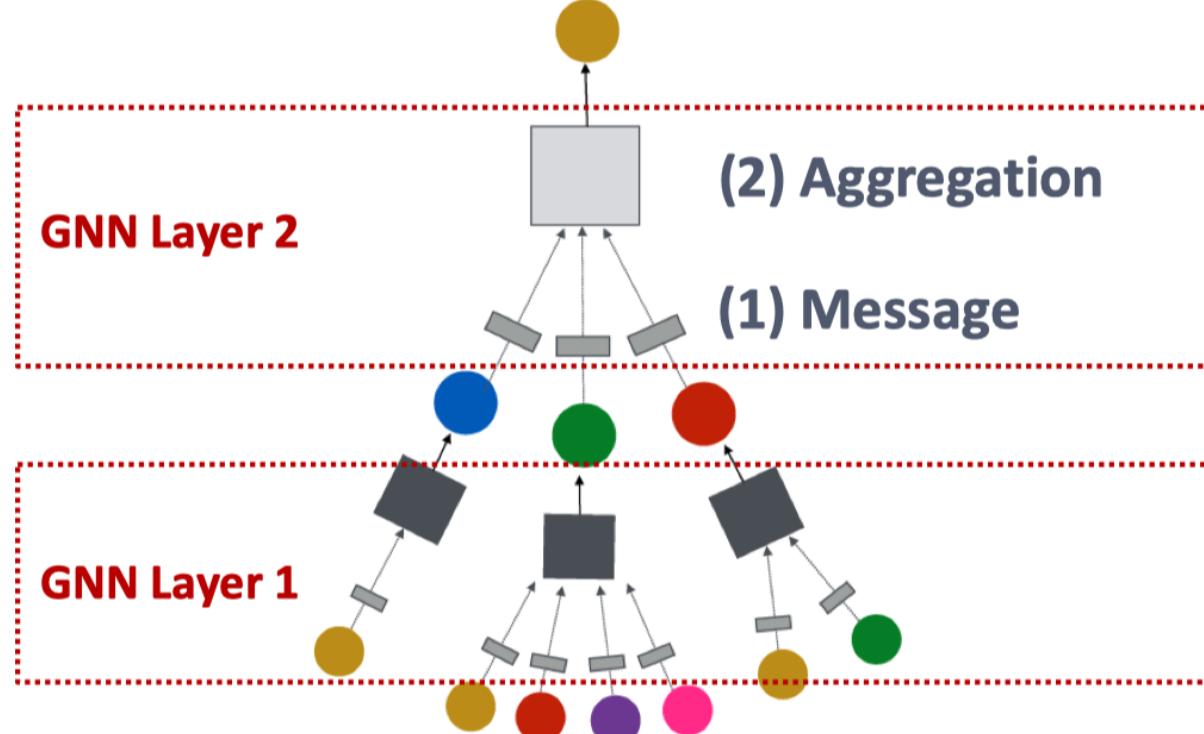
Since each word attends to **all the other words**, **the computation graph** of a transformer layer is identical to that of a GNN on the **fully-connected “word” graph**.



A More General GNN Framework



(3) Layer connectivity



Message Computation

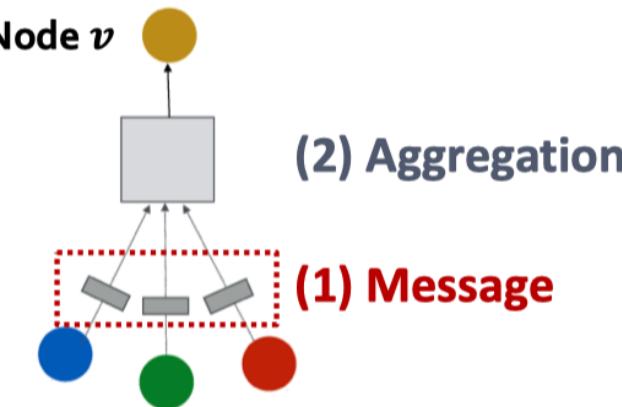
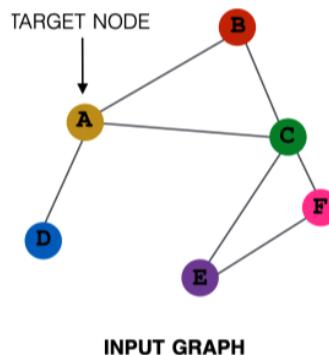
■ (1) Message computation

- **Message function:** $\mathbf{m}_u^{(l)} = \text{MSG}^{(l)}(\mathbf{h}_u^{(l-1)})$

- **Intuition:** Each node will create a message, which will be sent to other nodes later

- **Example:** A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$

- Multiply node features with weight matrix $\mathbf{W}^{(l)}$



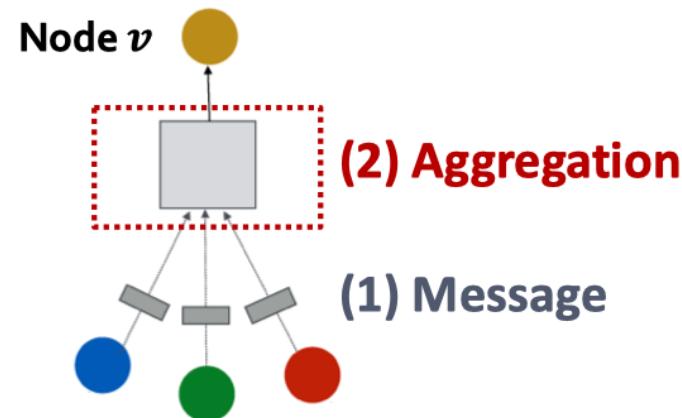
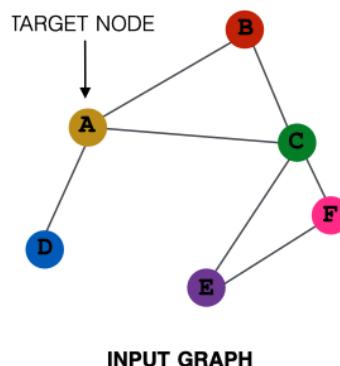
Message Aggregation

■ (2) Aggregation

- **Intuition:** Node v will aggregate the messages from its neighbors u :

$$\mathbf{h}_v^{(l)} = \text{AGG}^{(l)} \left(\left\{ \mathbf{m}_u^{(l)}, u \in N(v) \right\} \right)$$

- **Example:** Sum(\cdot), Mean(\cdot) or Max(\cdot) aggregator
 - $\mathbf{h}_v^{(l)} = \text{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$



Message Aggregation

- **Issue:** Information from node v itself **could get lost**
 - Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$
- **Solution:** Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$
 - **(1) Message:** compute message from node v itself
 - Usually, a **different message computation** will be performed



$$\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}$$



$$\mathbf{m}_v^{(l)} = \mathbf{B}^{(l)} \mathbf{h}_v^{(l-1)}$$

- **(2) Aggregation:** After aggregating from neighbors, we can aggregate the message from node v itself
 - Via **concatenation or summation**

$$\mathbf{h}_v^{(l)} = \text{CONCAT} \left(\text{AGG} \left(\left\{ \mathbf{m}_u^{(l)}, u \in N(v) \right\} \right), \boxed{\mathbf{m}_v^{(l)}} \right)$$

First aggregate from neighbors

Then aggregate from node itself

A Single GNN Layer

■ Putting things together:

- **(1) Message:** each node computes a message

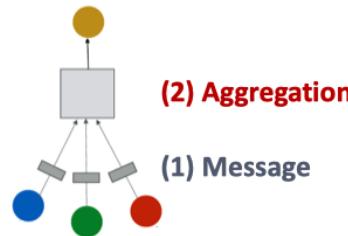
$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)} \left(\mathbf{h}_u^{(l-1)} \right), u \in \{N(v) \cup v\}$$

- **(2) Aggregation:** aggregate messages from neighbors

$$\mathbf{h}_v^{(l)} = \text{AGG}^{(l)} \left(\left\{ \mathbf{m}_u^{(l)}, u \in N(v) \right\}, \mathbf{m}_v^{(l)} \right)$$

- **Nonlinearity (activation):** Adds expressiveness

- Often written as $\sigma(\cdot)$. Examples: ReLU(\cdot), Sigmoid(\cdot) , ...
 - Can be added to message or aggregation



GCN Model

■ (1) Graph Convolutional Networks (GCN)

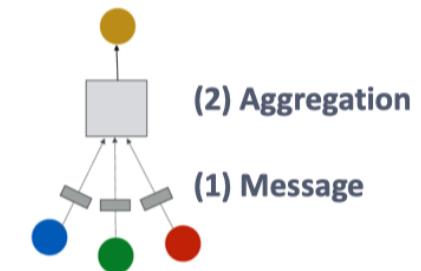
$$\mathbf{h}_v^{(l)} = \sigma \left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

■ How to write this as Message + Aggregation?

Message

$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

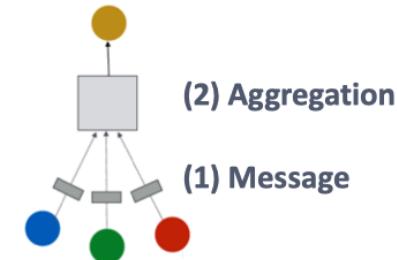
Aggregation



GCN Model

■ (1) Graph Convolutional Networks (GCN)

$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$



■ Message:

- Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{|N(v)|} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}$

Normalized by node degree
(In the GCN paper they use a slightly different normalization)

■ Aggregation:

- Sum over messages from neighbors, then apply activation
- $\mathbf{h}_v^{(l)} = \sigma \left(\text{Sum} \left(\left\{ \mathbf{m}_u^{(l)}, u \in N(v) \right\} \right) \right)$

In GCN the input graph is assumed to have self-edges that are included in the summation.

GraphSAGE Model

■ (2) GraphSAGE

$$\mathbf{h}_v^{(l)} = \sigma \left(\mathbf{W}^{(l)} \cdot \text{CONCAT} \left(\mathbf{h}_v^{(l-1)}, \text{AGG} \left(\left\{ \mathbf{h}_u^{(l-1)}, \forall u \in N(v) \right\} \right) \right) \right)$$

■ How to write this as Message + Aggregation?

- **Message** is computed within the $\text{AGG}(\cdot)$

- **Two-stage aggregation**

- **Stage 1:** Aggregate from node neighbors

$$\mathbf{h}_{N(v)}^{(l)} \leftarrow \text{AGG} \left(\left\{ \mathbf{h}_u^{(l-1)}, \forall u \in N(v) \right\} \right)$$

- **Stage 2:** Further aggregate over the node itself

$$\mathbf{h}_v^{(l)} \leftarrow \sigma \left(\mathbf{W}^{(l)} \cdot \text{CONCAT}(\mathbf{h}_v^{(l-1)}, \mathbf{h}_{N(v)}^{(l)}) \right)$$

GraphSAGE Model

- **Mean:** Take a weighted average of neighbors

$$\text{AGG} = \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}$$

AggregationMessage computation

- **Pool:** Transform neighbor vectors and apply symmetric vector function $\text{Mean}(\cdot)$ or $\text{Max}(\cdot)$

$$\text{AGG} = \text{Mean}(\{\text{MLP}(\mathbf{h}_u^{(l-1)}), \forall u \in N(v)\})$$

AggregationMessage computation

- **LSTM:** Apply LSTM to reshuffled of neighbors

$$\text{AGG} = \text{LSTM}([\mathbf{h}_u^{(l-1)}, \forall u \in \pi(N(v))])$$

Aggregation

GraphSAGE Model

■ ℓ_2 Normalization:

- **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
- $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\|\mathbf{h}_v^{(l)}\|_2}$ $\forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2 -norm)
- Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2 -norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After ℓ_2 normalization, all vectors will have the same ℓ_2 -norm

GAT Model

■ (3) Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Attention weights

■ In GCN / GraphSAGE

- $\alpha_{vu} = \frac{1}{|N(v)|}$ is the **weighting factor (importance)** of node u 's message to node v
- $\Rightarrow \alpha_{vu}$ is defined **explicitly** based on the **structural properties** of the graph (node degree)
- \Rightarrow All neighbors $u \in N(v)$ are **equally important** to node v

GAT Model

■ (3) Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Attention weights

Not all node's neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention** α_{vu} focuses on the important parts of the input data and fades out the rest.
 - **Idea:** the NN should devote more computing power on that small but important part of the data.
 - Which part of the data is more important depends on the context and is learned through training.

GAT Model

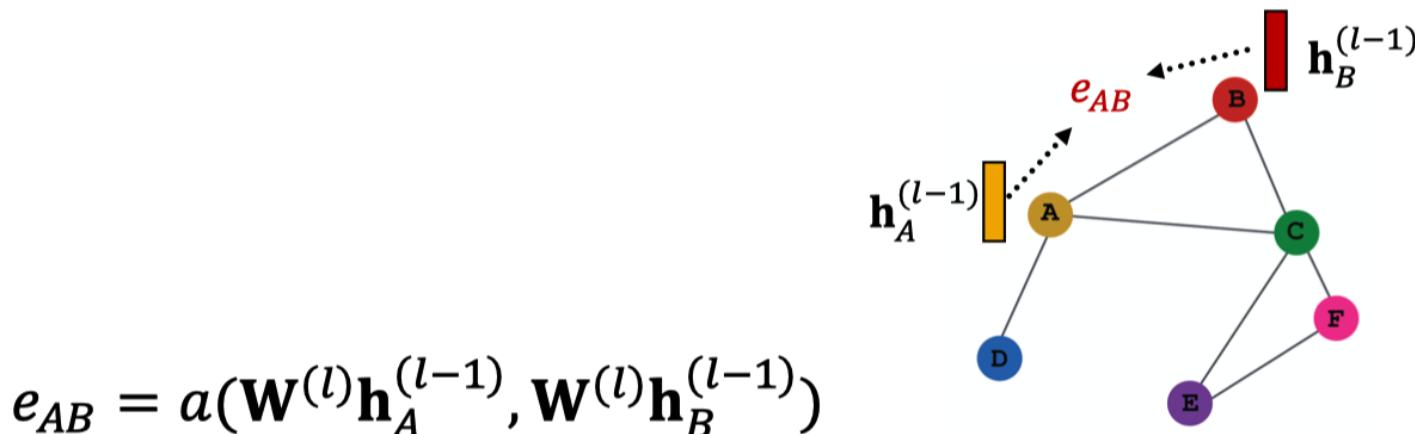
Can we do better than simple neighborhood aggregation?

Can weighting factors α_{vu} be learned?

- **Goal:** Specify arbitrary importance to different neighbors of each node in the graph
- **Idea:** Compute embedding $h_v^{(l)}$ of each node in the graph following an **attention strategy**:
 - Nodes attend over their neighborhoods' message
 - Implicitly specifying different weights to different nodes in a neighborhood

GAT Model

- Let α_{vu} be computed as a byproduct of an **attention mechanism a** :
 - (1) Let a compute **attention coefficients e_{vu}** across pairs of nodes u, v based on their messages:
$$e_{vu} = a(\mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_v^{(l-1)})$$
 - e_{vu} indicates the importance of u 's message to node v



GAT Model

- **Normalize** e_{vu} into the **final attention weight** α_{vu}

- Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:

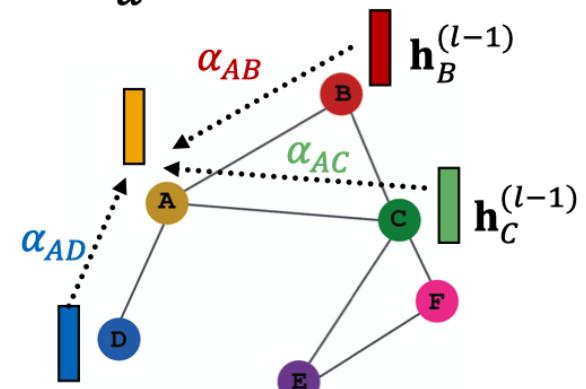
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** α_{vu} :

$$\mathbf{h}_v^{(l)} = \sigma(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Weighted sum using α_{AB} , α_{AC} , α_{AD} :

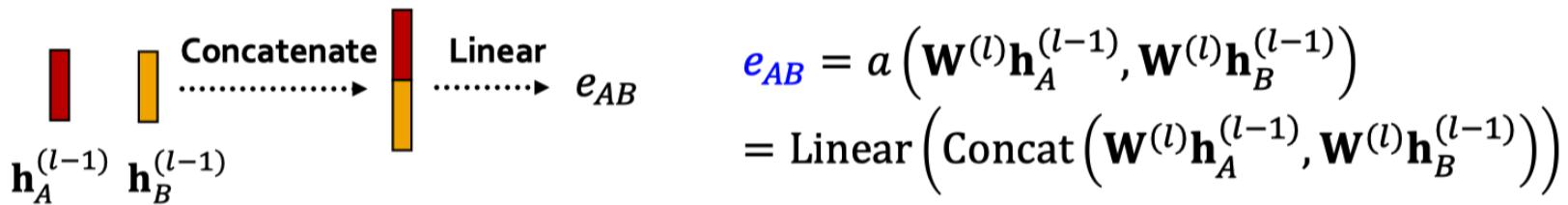
$$\mathbf{h}_A^{(l)} = \sigma(\alpha_{AB} \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)} + \alpha_{AC} \mathbf{W}^{(l)} \mathbf{h}_C^{(l-1)} + \alpha_{AD} \mathbf{W}^{(l)} \mathbf{h}_D^{(l-1)})$$



GAT Model

■ What is the form of attention mechanism a ?

- The approach is agnostic to the choice of a
 - E.g., use a simple single-layer neural network
 - a have trainable parameters (weights in the Linear layer)



- Parameters of a are trained jointly:
 - Learn the parameters together with weight matrices (i.e., other parameter of the neural net $\mathbf{W}^{(l)}$) in an end-to-end fashion

GAT Model

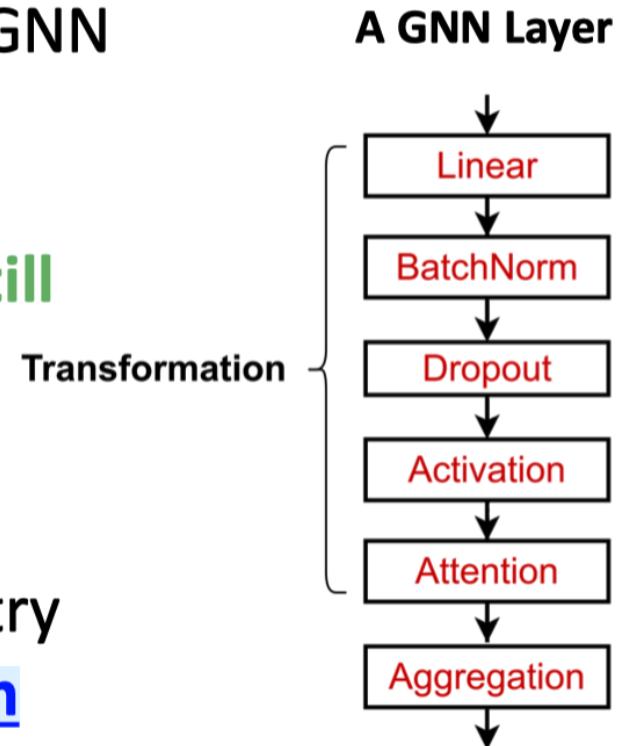
- **Multi-head attention:** Stabilizes the learning process of attention mechanism
 - Create multiple attention scores (each replica with a different set of parameters):
$$\mathbf{h}_v^{(l)}[1] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^1 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$
$$\mathbf{h}_v^{(l)}[2] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^2 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$
$$\mathbf{h}_v^{(l)}[3] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^3 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$
 - Outputs are aggregated:
 - By concatenation or summation
 - $\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

Benefits of Attention Mechanism

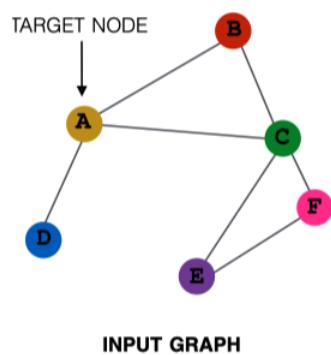
- **Key benefit:** Allows for (implicitly) specifying **different importance values (α_{vu}) to different neighbors**
- **Computationally efficient:**
 - Computation of attentional coefficients can be parallelized across all edges of the graph
 - Aggregation may be parallelized across all nodes
- **Storage efficient:**
 - Sparse matrix operations do not require more than $O(V + E)$ entries to be stored
 - **Fixed** number of parameters, irrespective of graph size
- **Localized:**
 - Only **attends over local network neighborhoods**
- **Inductive capability:**
 - It is a shared *edge-wise* mechanism
 - It does not depend on the global graph structure

GNN Layer in Practice

- **Summary:** Modern deep learning modules can be included into a GNN layer for better performance
- **Designing novel GNN layers is still an active research frontier!**
- **Suggested resources:** You can explore diverse GNN designs or try out your own ideas in [GraphGym](#)



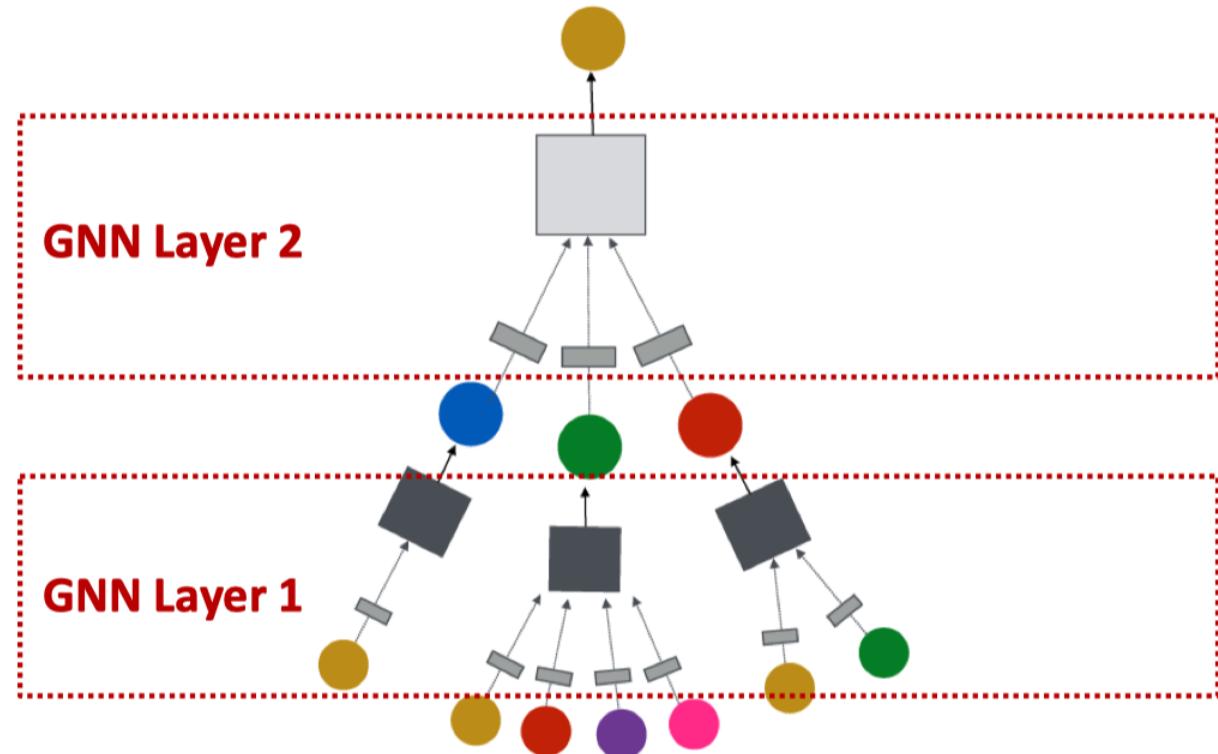
Stacking GNN Layers



(3) Layer connectivity

How to connect GNN layers into a GNN?

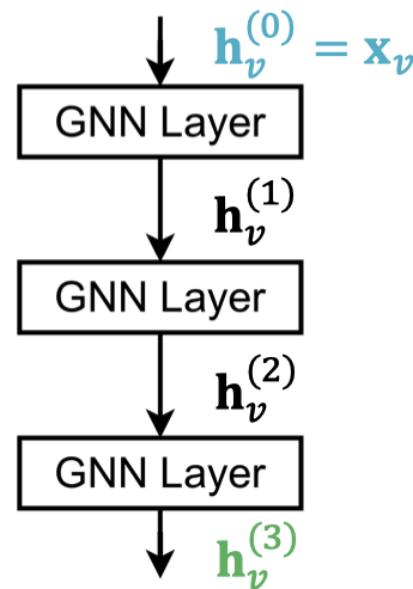
- Stack layers sequentially
- Ways of adding skip connections



Stacking GNN Layers

■ How to construct a Graph Neural Network?

- The standard way: Stack GNN layers sequentially
- Input: Initial raw node feature \mathbf{x}_v
- Output: Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers

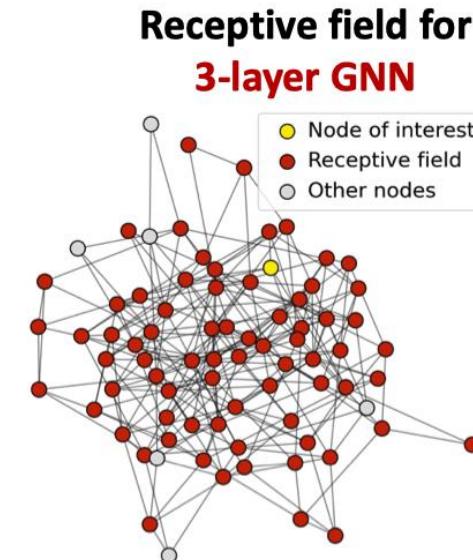
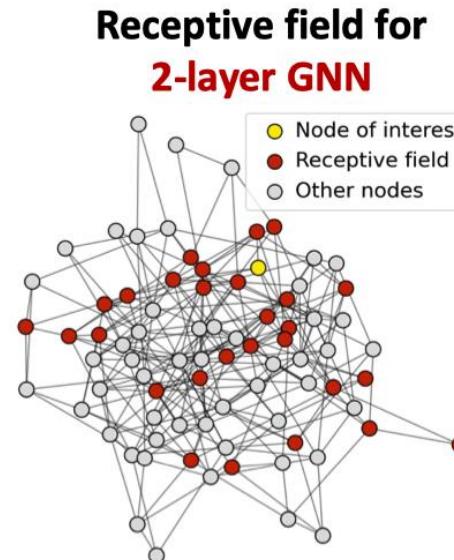
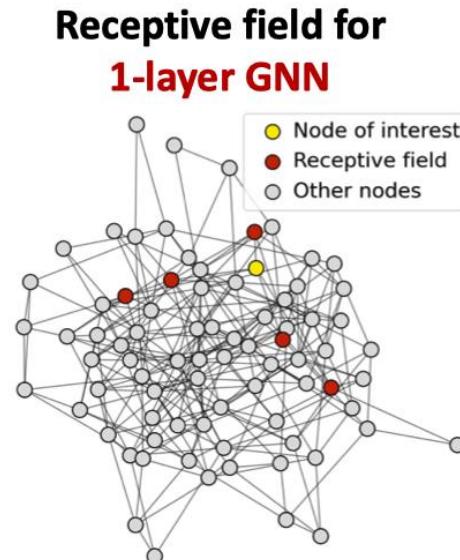


The Over-smoothing Problem

- **The issue of stacking many GNN layers**
 - GNN suffers from **the over-smoothing problem**
- **The over-smoothing problem: all the node embeddings converge to the same value**
 - This is bad because we **want to use node embeddings to differentiate nodes**
- **Why does the over-smoothing problem happen?**

Receptive Field of a GNN

- **Receptive field:** the set of nodes that determine the embedding of a node of interest
 - In a K -layer GNN, each node has a receptive field of K -hop neighborhood



Receptive Field & Over-smoothing

- We can explain over-smoothing via the notion of the receptive field
 - We know the embedding of a node is determined by its receptive field
 - If two nodes have highly-overlapped receptive fields, then their embeddings are highly similar
 - Stack many GNN layers → nodes will have highly-overlapped receptive fields → node embeddings will be highly similar → suffer from the over-smoothing problem
- Next: how do we overcome over-smoothing problem?

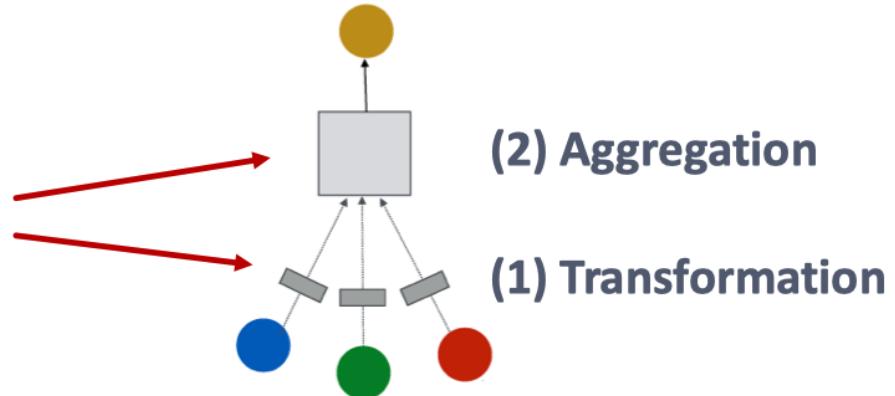
Design GNN Layer Connectivity

- **What do we learn from the over-smoothing problem?**
- **Lesson 1: Be cautious when adding GNN layers**
 - Unlike neural networks in other domains (CNN for image classification), **adding more GNN layers do not always help**
 - **Step 1:** Analyze the necessary receptive field to solve your problem. E.g., by computing the diameter of the graph
 - **Step 2:** Set number of GNN layers L to be a bit more than the receptive field we like. **Do not set L to be unnecessarily large!**
- **Question:** How to enhance the expressive power of a GNN, **if the number of GNN layers is small?**

Expressive Power for Shallow GNNs

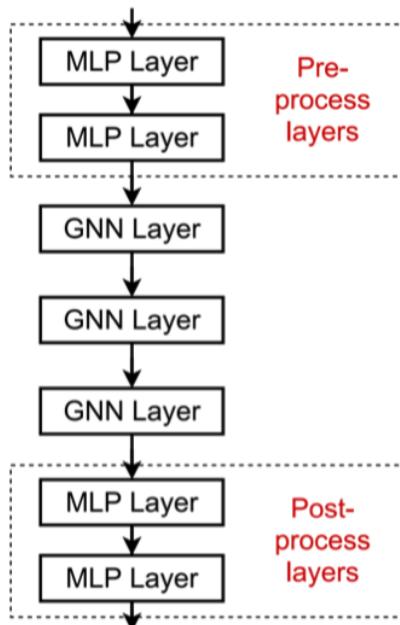
- **How to make a shallow GNN more expressive?**
- **Solution 1:** Increase the expressive power **within each GNN layer**
 - In our previous examples, each transformation or aggregation function only include one linear layer
 - We can **make aggregation / transformation become a deep neural network!**

If needed, each box could include a **3-layer MLP**



Expressive Power for Shallow GNNs

- **How to make a shallow GNN more expressive?**
- **Solution 2:** Add layers that do not pass messages
 - A GNN does not necessarily only contain GNN layers
 - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**



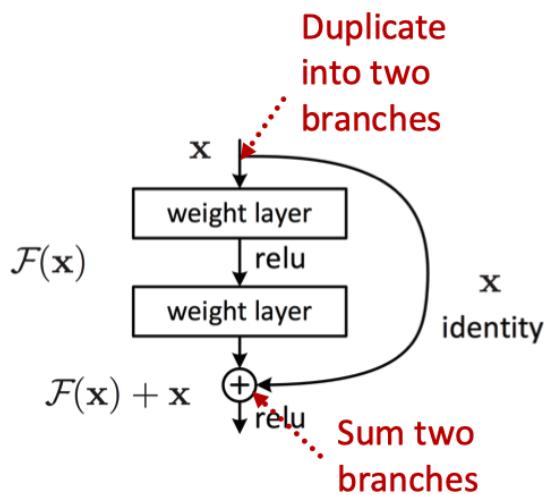
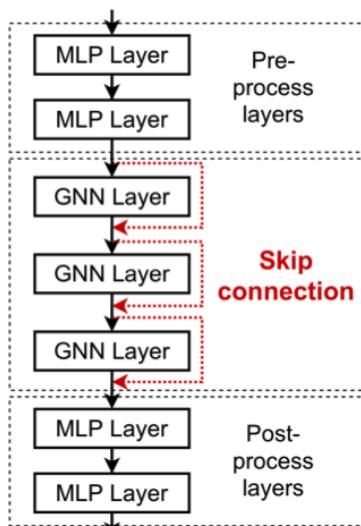
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

Post-processing layers: Important when reasoning / transformation over node embeddings are needed
E.g., graph classification, knowledge graphs

In practice, adding these layers works great!

Design GNN Layer Connectivity

- What if my problem still requires many GNN layers?
- Lesson 2: Add skip connections in GNNs
 - Observation from over-smoothing: Node embeddings in earlier GNN layers can sometimes better differentiate nodes
 - Solution: We can increase the impact of earlier layers on the final node embeddings, by adding shortcuts in GNN



Idea of skip connections:
Before adding shortcuts:
 $\mathcal{F}(x)$
After adding shortcuts:
 $\mathcal{F}(x) + x$

Another Explanation of Skip Connections

■ Why do skip connections work?

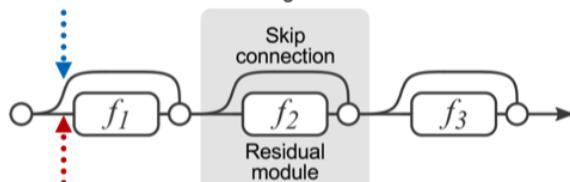
- **Intuition:** Skip connections create **a mixture of models**
- N skip connections $\rightarrow 2^N$ possible paths
- Each path could have up to N modules
- We automatically get **a mixture of shallow GNNs and deep GNNs**

All the possible paths:

$$2 * 2 * 2 = 2^3 = 8$$

Path 2: skip this module

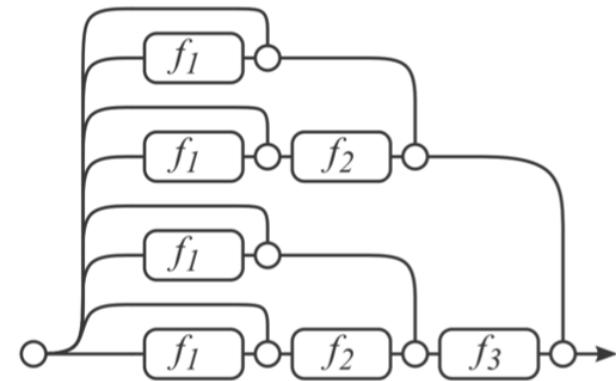
Building block



Path 1: include this module

(a) Conventional 3-block residual network

=



(b) Unraveled view of (a)

GCN with Skip Connections

■ A standard GCN layer

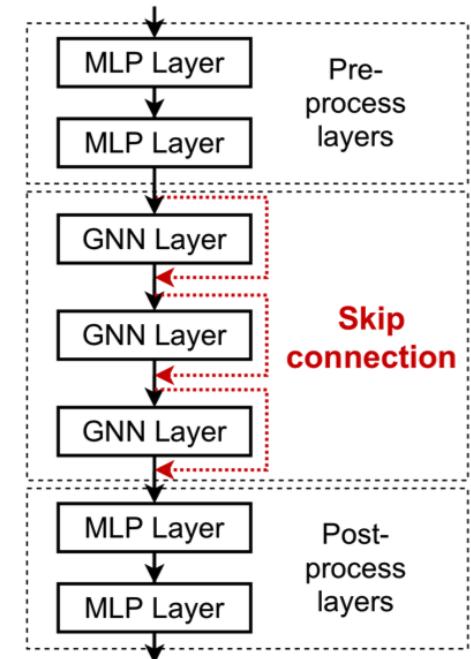
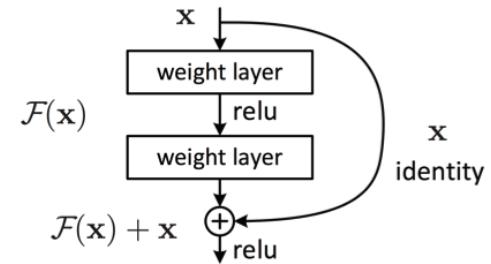
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

This is our $F(\mathbf{x})$

■ A GCN layer with skip connection

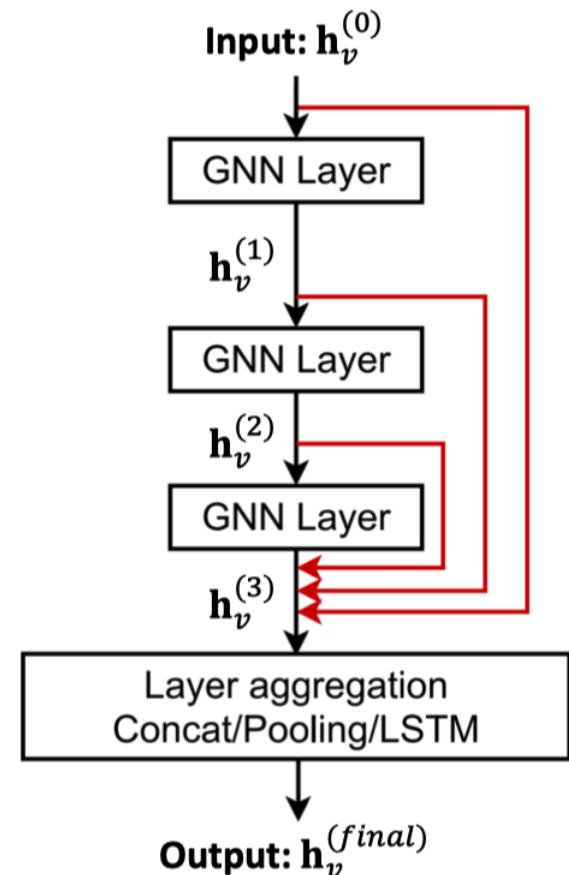
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)} \right)$$

$F(\mathbf{x})$ + \mathbf{x}



GCN with Skip Connections

- **Other options:** Directly skip to the last layer
 - The final layer directly **aggregates from the all the node embeddings** in the previous layers



Graph Manipulation

Our assumption so far has been

- **Raw input graph = computational graph**

Reasons for breaking this assumption

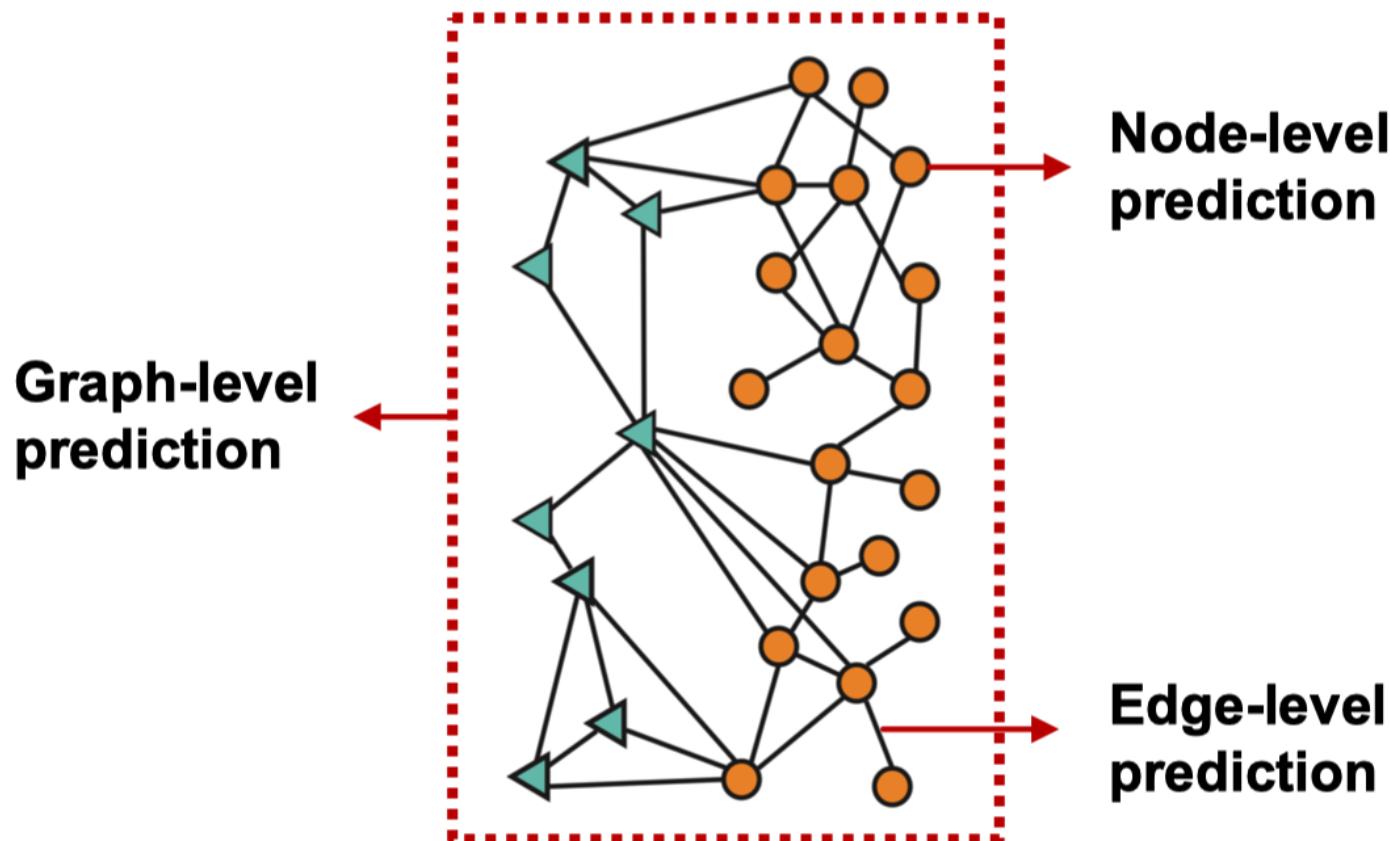
- **Feature level:**
 - The input graph **lacks features** → feature augmentation
- **Structure level:**
 - The graph is **too sparse** → inefficient message passing
 - The graph is **too dense** → message passing is too costly
 - The graph is **too large** → cannot fit the computational graph into a GPU
- It's just **unlikely that the input graph happens to be the optimal computation graph** for embeddings

Graph Manipulation

- **Graph Feature manipulation**
 - The input graph **lacks features** → **feature augmentation**
- **Graph Structure manipulation**
 - The graph is **too sparse** → **Add virtual nodes / edges**
 - The graph is **too dense** → **Sample neighbors when doing message passing**
 - The graph is **too large** → **Sample subgraphs to compute embeddings**

GNN Prediction Head

- Idea: Different task levels require different prediction heads

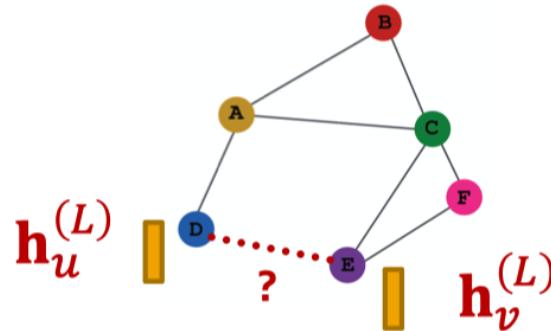


Node-level Prediction

- **Node-level prediction:** We can directly make prediction using node embeddings!
- After GNN computation, we have **d -dim node embeddings:** $\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\}$
- Suppose we want to make **k -way prediction**
 - Classification: classify among k categories
 - Regression: regress on k targets
- $\hat{\mathbf{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)} \mathbf{h}_v^{(L)}$
 - $\mathbf{W}^{(H)} \in \mathbb{R}^{k \times d}$: We map node embeddings from $\mathbf{h}_v^{(L)} \in \mathbb{R}^d$ to $\hat{\mathbf{y}}_v \in \mathbb{R}^k$ so that we can compute the loss

Edge-level Prediction

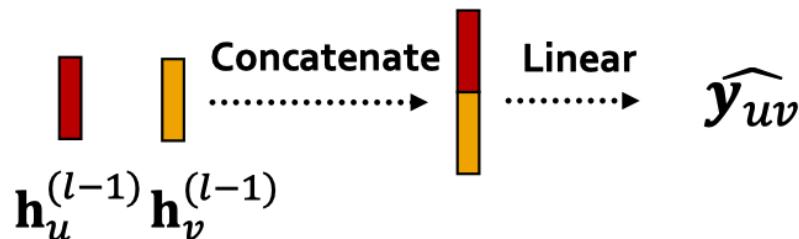
- **Edge-level prediction:** Make prediction using pairs of node embeddings
- Suppose we want to make k -way prediction
- $\hat{y}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



- What are the options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$?

Edge-level Prediction

- Options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- (1) Concatenation + Linear
 - We have seen this in graph attention



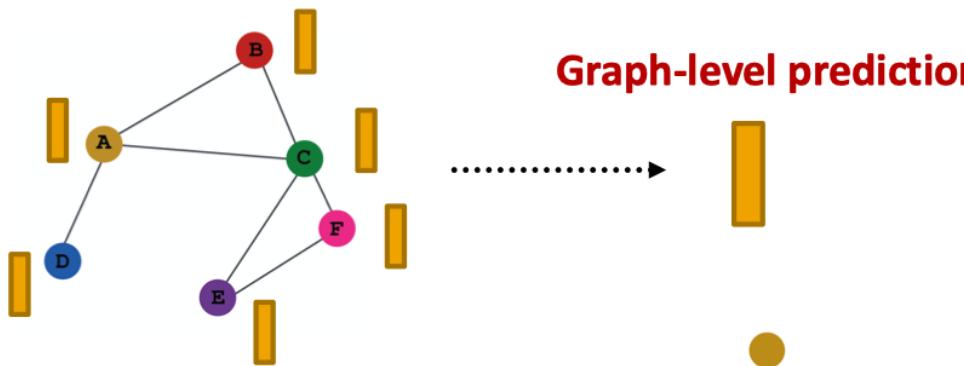
- $\hat{y}_{uv} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
- Here $\text{Linear}(\cdot)$ will map **2d-dimensional** embeddings (since we concatenated embeddings) to **k-dim** embeddings (k -way prediction)

Edge-level Prediction

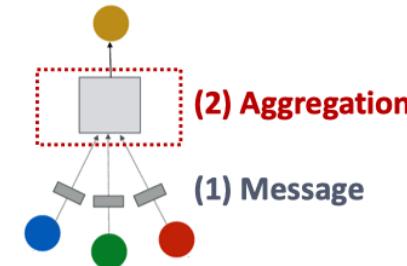
- Options for Head_{edge}($\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}$):
- **(2) Dot product**
 - $\hat{y}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
 - This approach only applies to 1-way prediction (e.g., link prediction: predict the existence of an edge)
 - Applying to k -way prediction:
 - Similar to multi-head attention: $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(k)}$ trainable
$$\hat{y}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$
$$\dots$$
$$\hat{y}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$
 - $$\hat{y}_{uv} = \text{Concat}(\hat{y}_{uv}^{(1)}, \dots, \hat{y}_{uv}^{(k)}) \in \mathbb{R}^k$$

Graph-level Prediction

- **Graph-level prediction:** Make prediction using all the node embeddings in our graph
- Suppose we want to make k -way prediction
- $\hat{\mathbf{y}}_G = \text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$



- $\text{Head}_{\text{graph}}(\cdot)$ is similar to $\text{AGG}(\cdot)$ in a GNN layer!



Graph-level Prediction

- Options for $\text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$
- **(1) Global mean pooling**

$$\hat{\mathbf{y}}_G = \text{Mean}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(2) Global max pooling**

$$\hat{\mathbf{y}}_G = \text{Max}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(3) Global sum pooling**

$$\hat{\mathbf{y}}_G = \text{Sum}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- These options work great for small graphs
- **Can we do better for large graphs?**

Questions?