

Advanced Deep Learning Architectures

COMP 5214 & ELEC 5680

Instructor: Dr. Qifeng Chen

<https://cqd.io>

Graph Neural Networks

Content

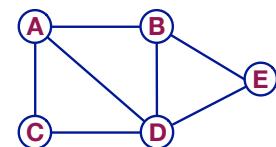
- **Local network neighborhoods:**
 - Describe aggregation strategies
 - Define computation graphs
- **Stacking multiple layers:**
 - Describe the model, parameters, training
 - How to fit the model?
 - Simple example for unsupervised and supervised training

Setup

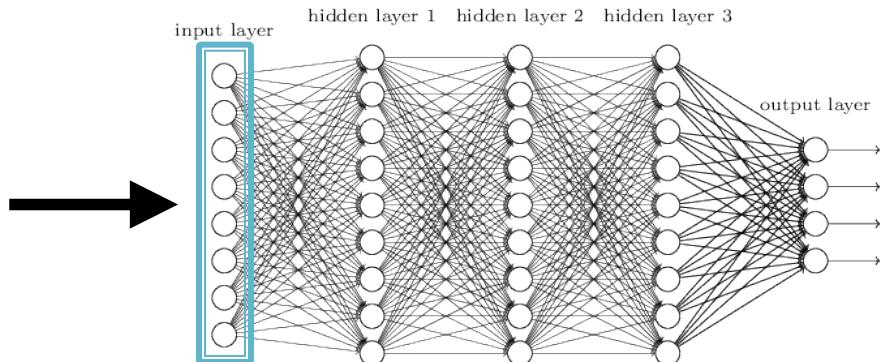
- Assume we have a graph G :
 - V is the **vertex set**
 - A is the **adjacency matrix** (assume binary)
 - $X \in \mathbb{R}^{m \times |V|}$ is a matrix of **node features**
 - v : a node in V ; $N(v)$: the set of neighbors of v .
 - **Node features:**
 - Social networks: User profile, User image
 - 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, ..., 1]

A Naïve Approach

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



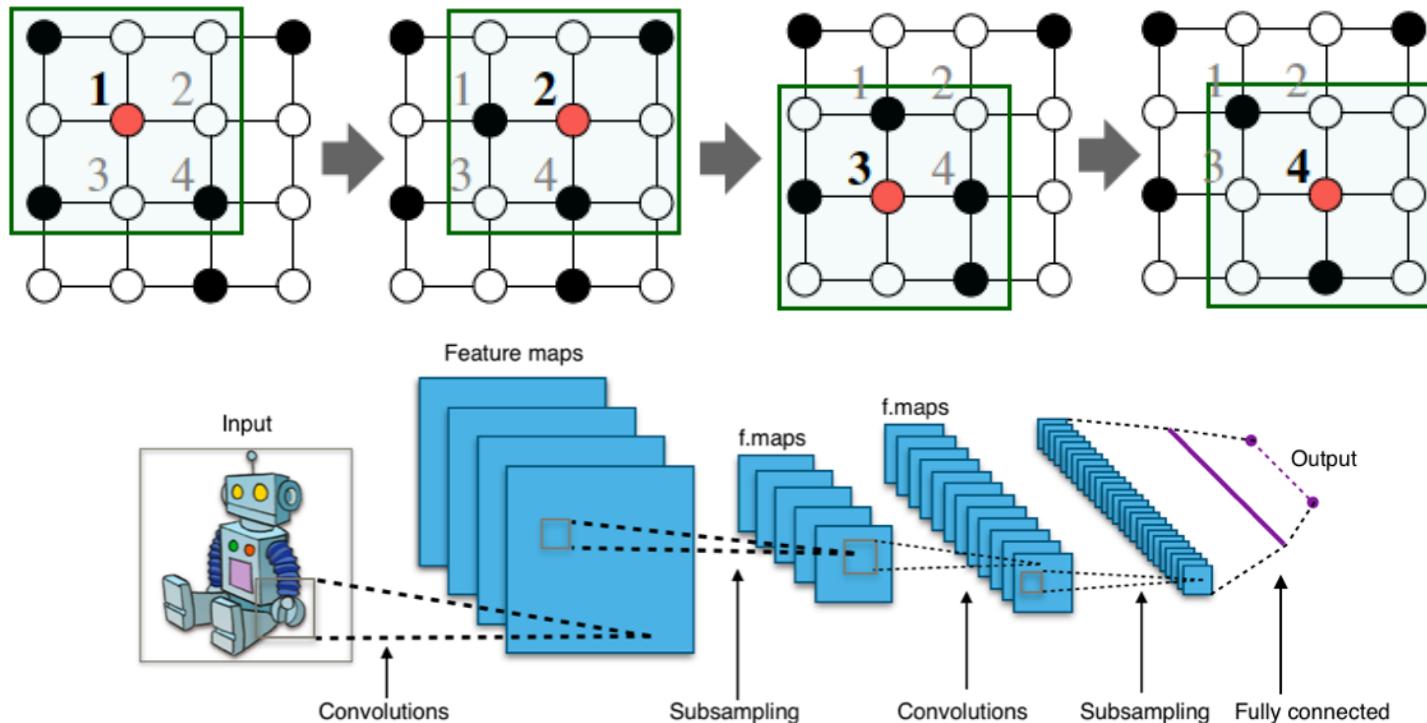
	A	B	C	D	E	Feat
A	0	1	1	1	0	1 0
B	1	0	0	1	1	0 0
C	1	0	0	1	0	0 1
D	1	1	1	0	1	1 1
E	0	1	0	1	0	1 0



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

Idea: Convolutional Networks

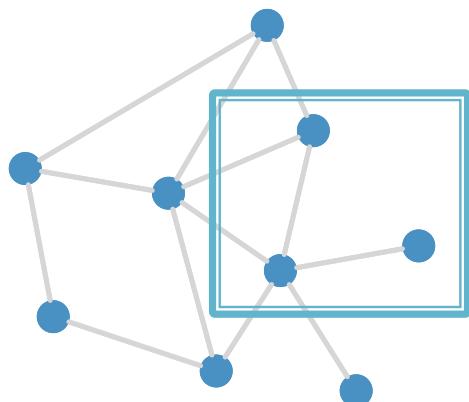
CNN on an image:



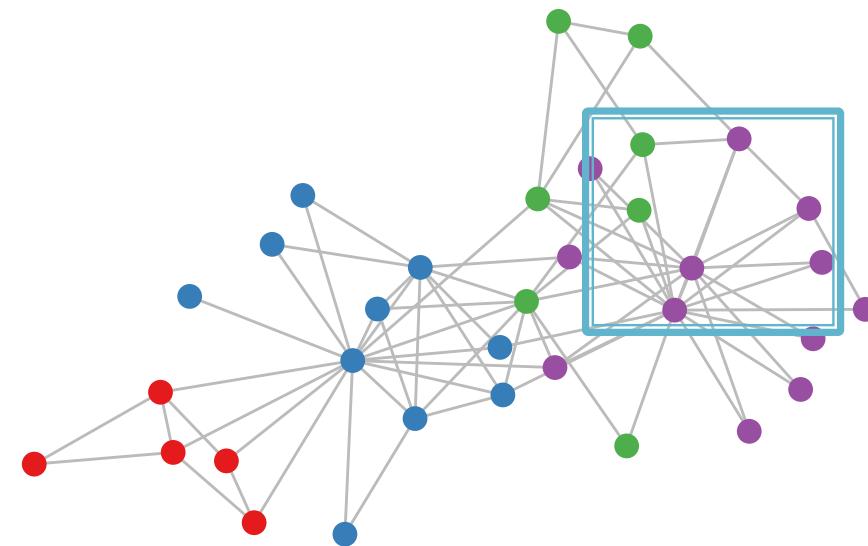
Goal is to generalize convolutions beyond simple lattices
Leverage node features/attributes (e.g., text, images)

Real-World Graphs

But our graphs look like this:



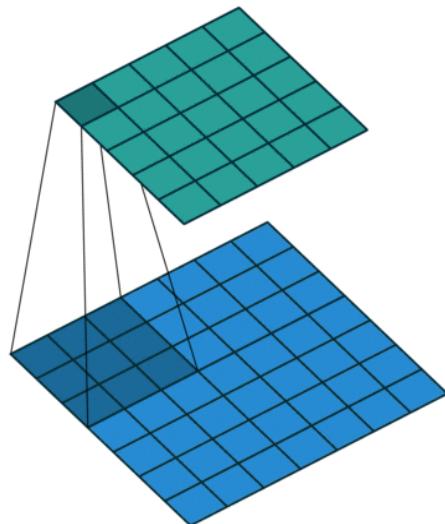
or this:



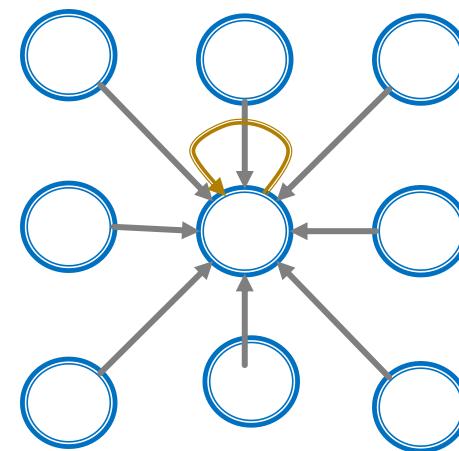
- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

From Images to Graphs

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



Image



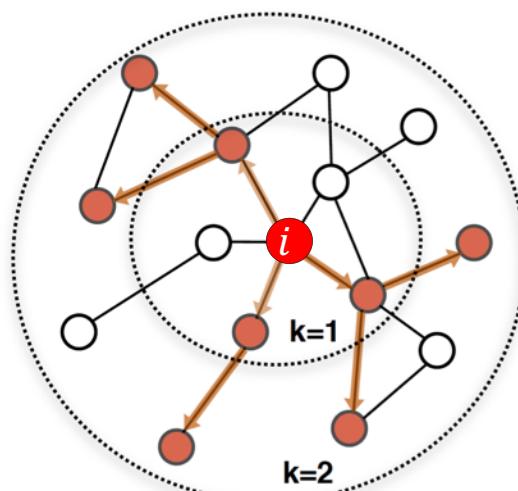
Graph

Idea: transform information at the neighbors and combine it:

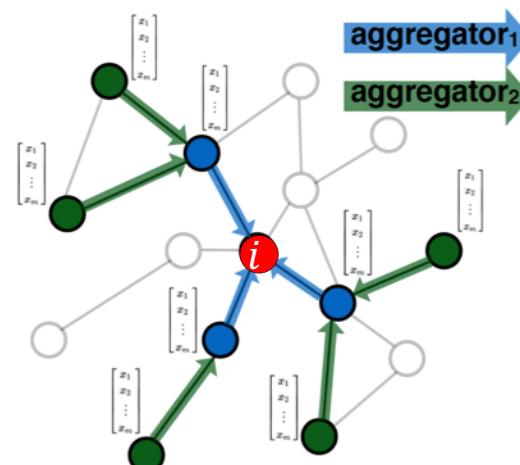
- Transform “messages” h_i from neighbors: $W_i h_i$
- Add them up: $\sum_i W_i h_i$

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

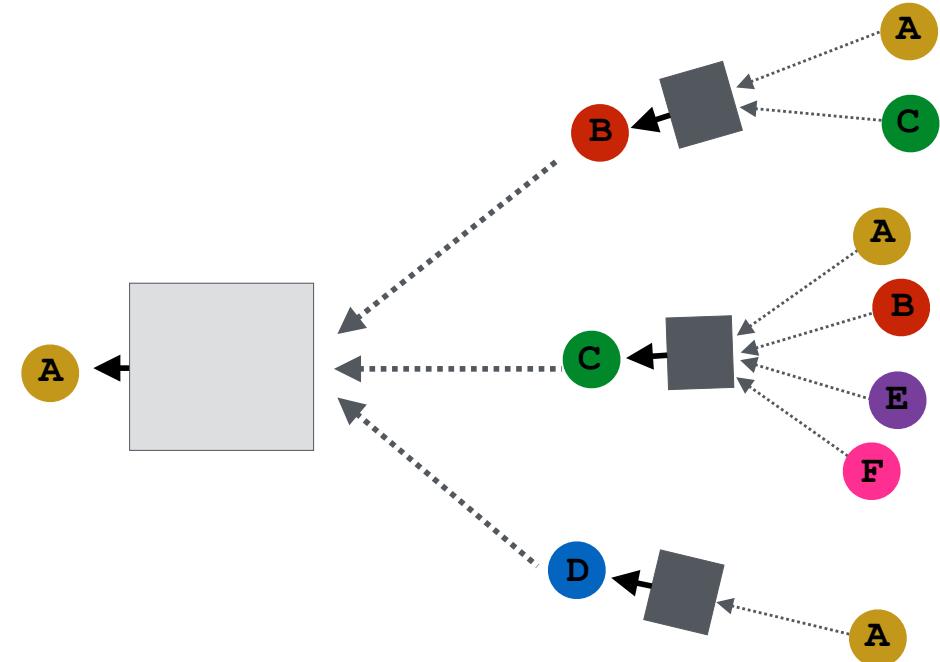
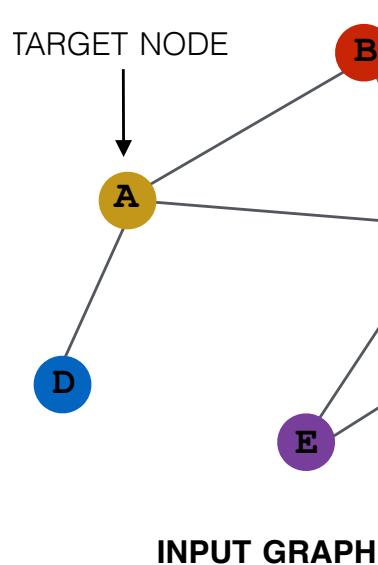


Propagate and transform information

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

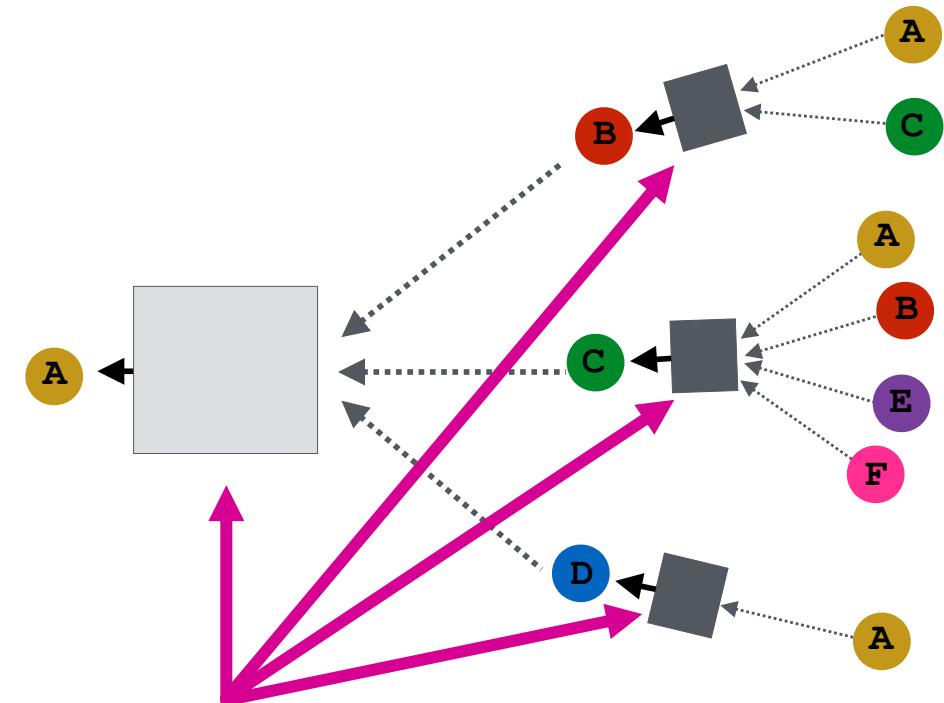
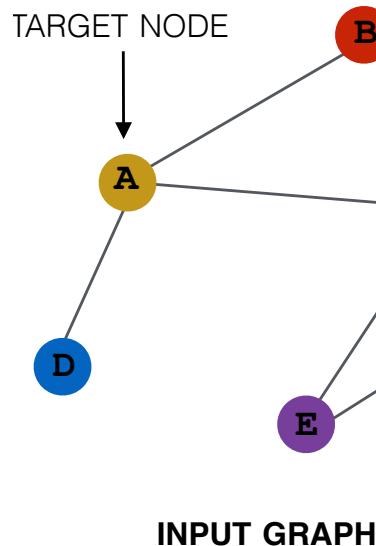
Idea: Aggregate Neighbors

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



Idea: Aggregate Neighbors

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

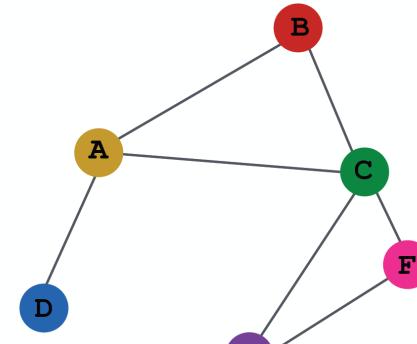


Neural networks

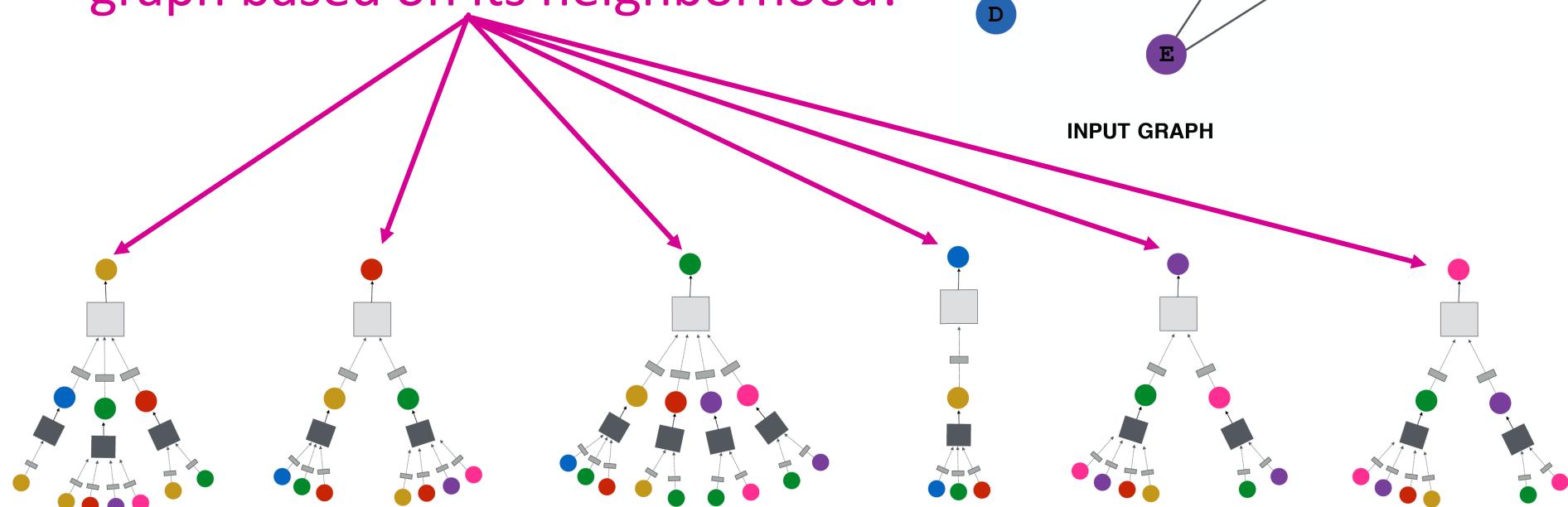
Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

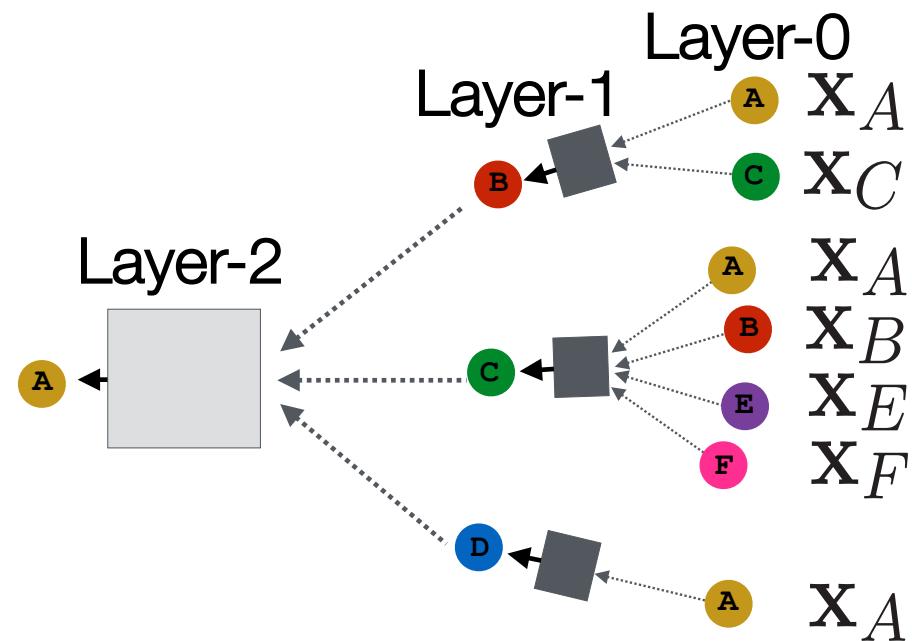
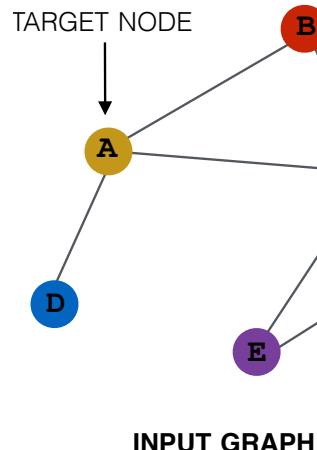


INPUT GRAPH



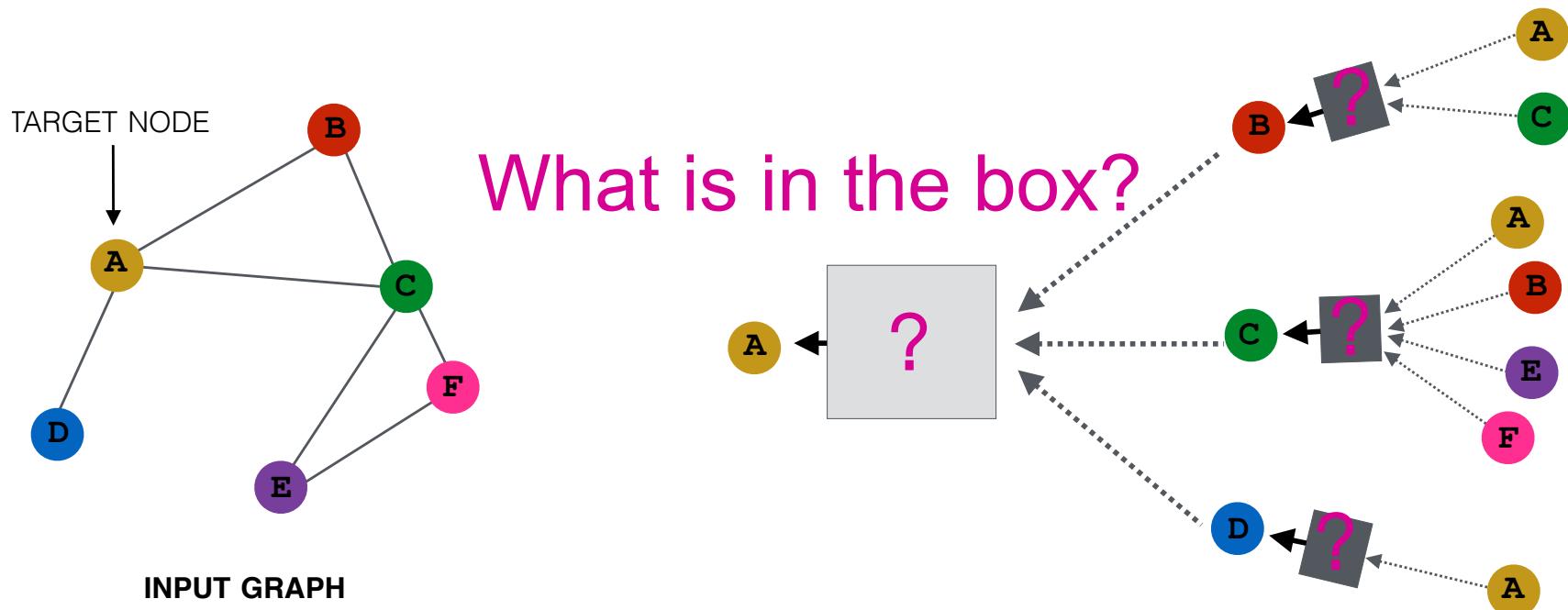
Deep Model: Many Layers

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



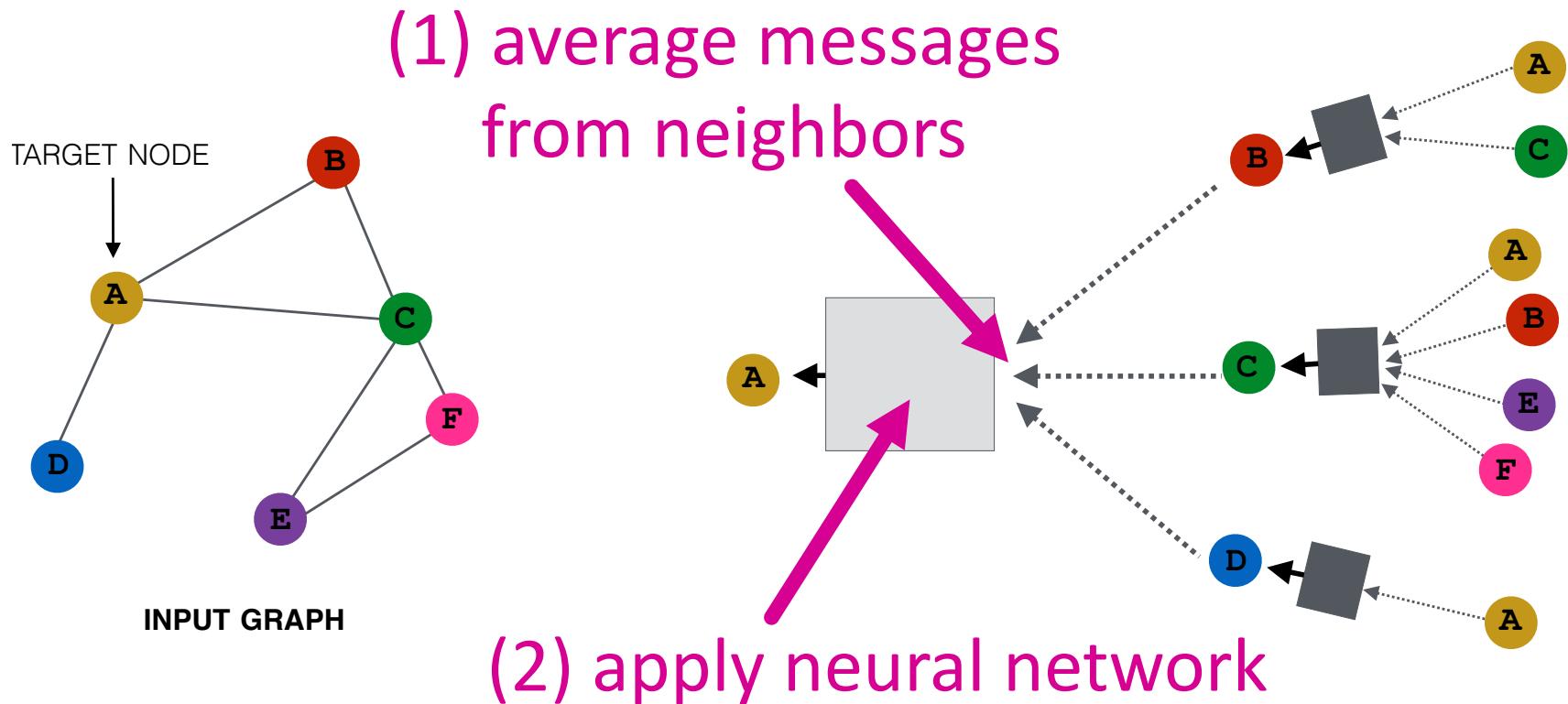
Neighborhood Aggregation

- **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers



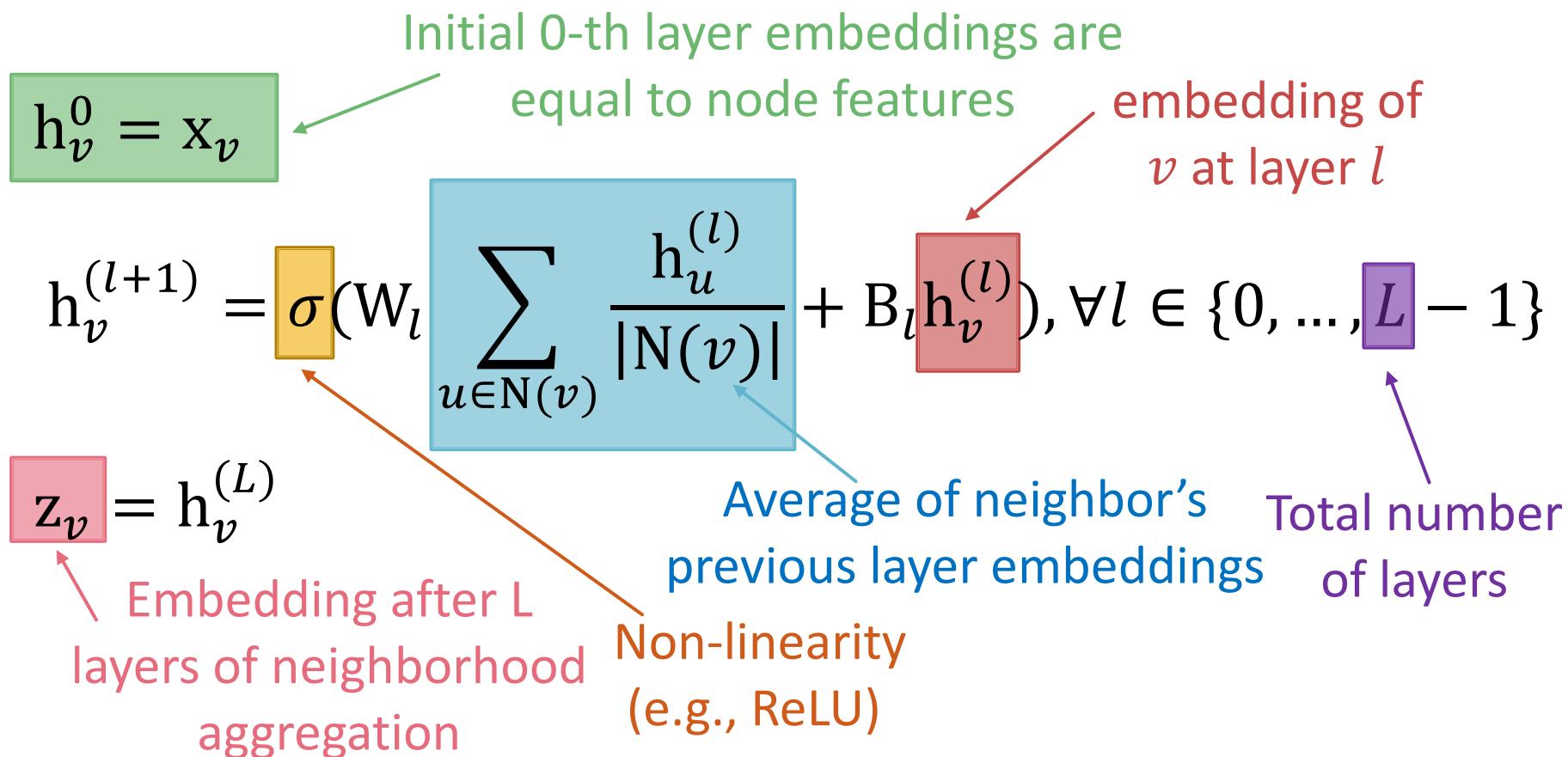
Neighborhood Aggregation

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



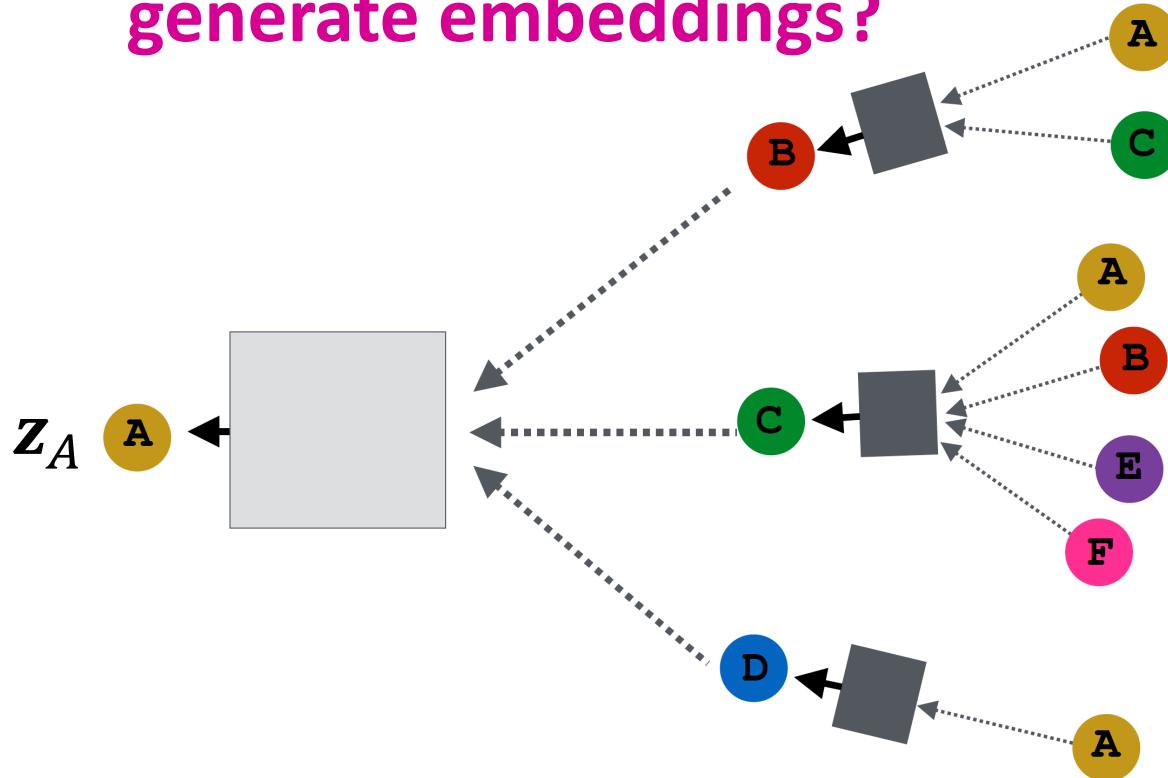
The Math: Deep Encoder

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



Training the Model

How do we train the model to generate embeddings?



Need to define a loss function on the embeddings

Model Parameters

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

$$\begin{aligned} h_v^{(0)} &= x_v \\ h_v^{(l+1)} &= \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\} \\ z_v &= h_v^{(L)} \end{aligned}$$

Final node embedding

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

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

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

Matrix Formulation (1)

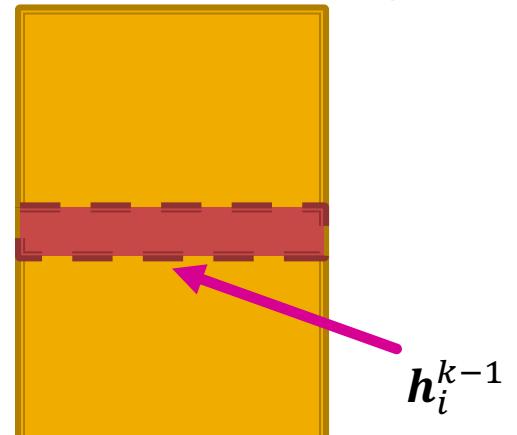
- Many aggregations can be performed efficiently by (sparse) matrix operations
- Let $H^{(l)} = [h_1^{(l)} \dots h_{|V|}^{(l)}]^T$
- Then: $\sum_{u \in N_v} h_u^{(l)} = A_{v,:} H^{(l)}$
- Let D be diagonal matrix where $D_{v,v} = \text{Deg}(v) = |N(v)|$
 - The inverse of D : D^{-1} is also diagonal:
$$D_{v,v}^{-1} = 1/|N(v)|$$
- Therefore,

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



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

Matrix of hidden embeddings H^{k-1}

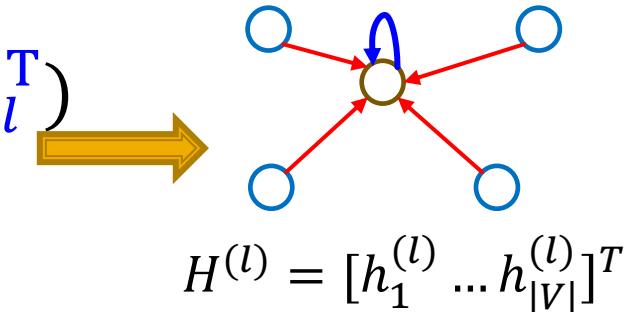


Matrix Formulation (2)

- Re-writing update function in matrix form:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W_l^T + H^{(l)}B_l^T)$$

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 matrix form, when aggregation function is complex

How to train a GNN

- Node embedding \mathbf{z}_v is a function of input graph
- **Supervised setting:** we want to minimize the loss \mathcal{L} (see also slide 15):

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

- \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

- “Similar” nodes have similar embeddings

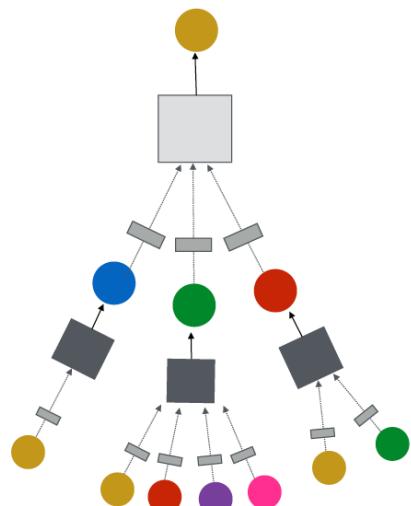
$$\mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

- Where $y_{u,v} = 1$ when node u and v are **similar**
- **CE** is the cross entropy (slide 16)
- **DEC** is the decoder such as inner product (lecture 4)
- **Node similarity** can be anything from lecture 3, e.g., a loss based on:
 - **Random walks** (node2vec, DeepWalk, struc2vec)
 - **Matrix factorization**
 - **Node proximity in the graph**

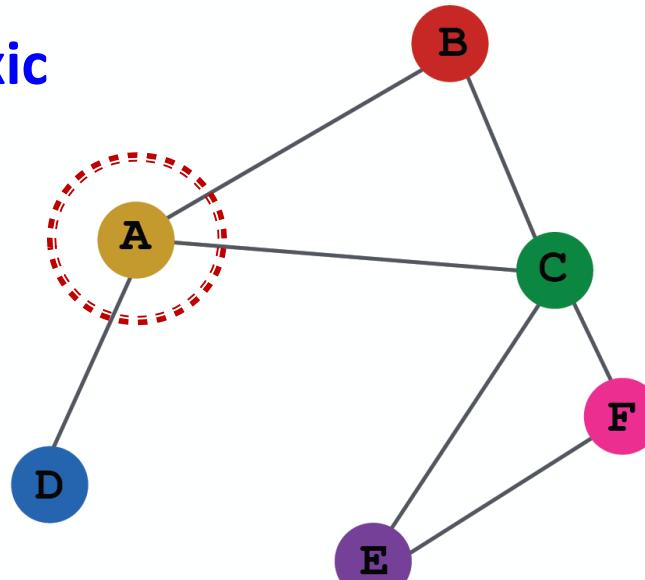
Supervised Training

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

Safe or toxic
drug?



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 (slide 16)

$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

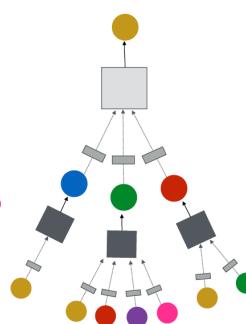
Encoder output: node embedding

Safe or toxic drug?

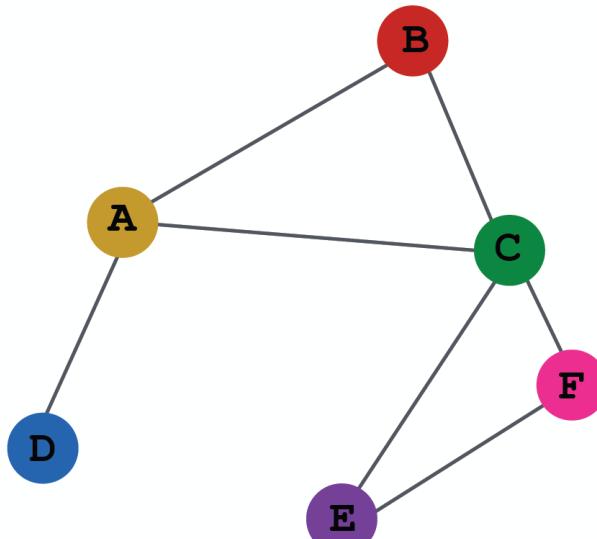
Node class

label

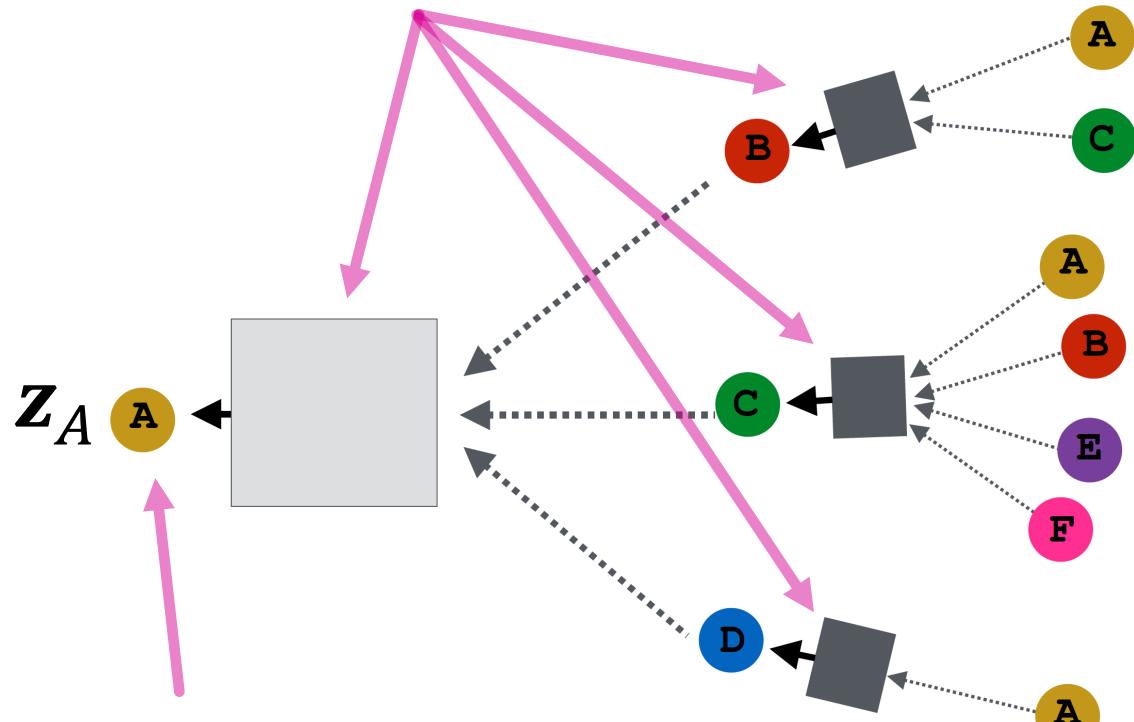
Classification weights



Model Design: Overview

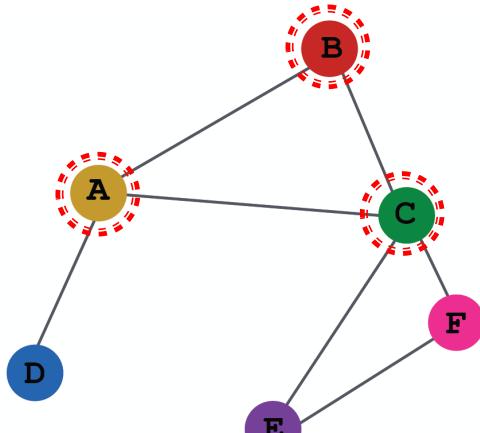


(1) Define a neighborhood aggregation function



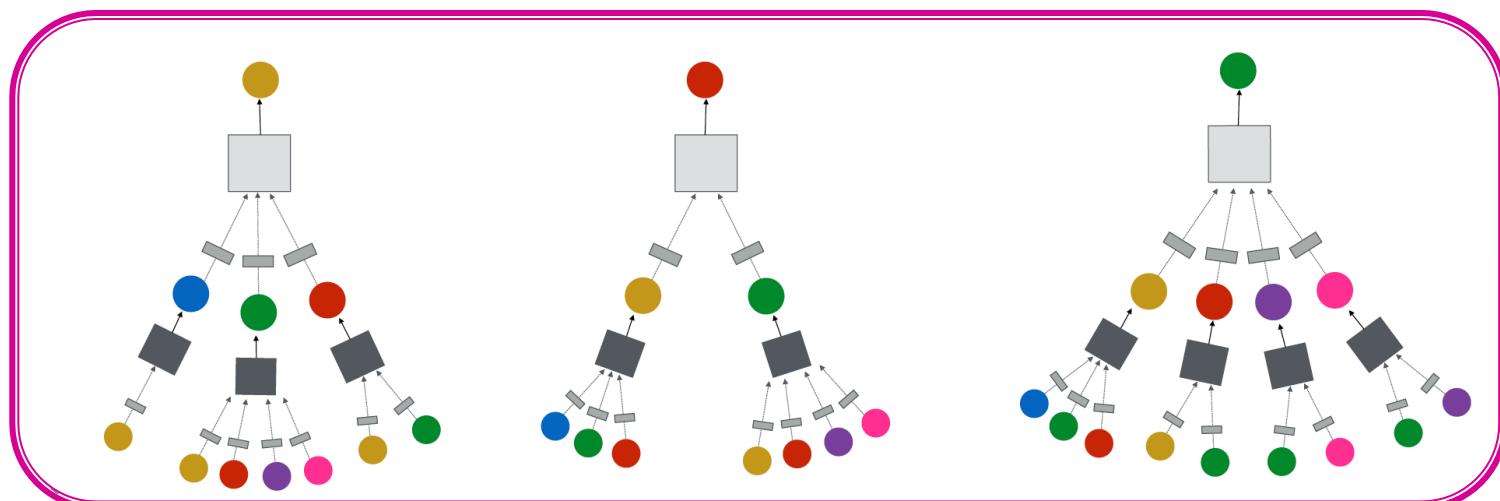
(2) Define a loss function on the embeddings

Model Design: Overview

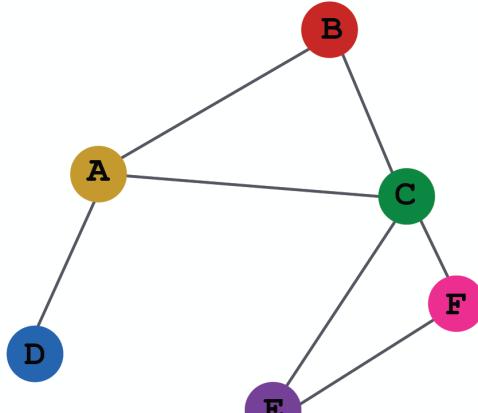


INPUT GRAPH

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



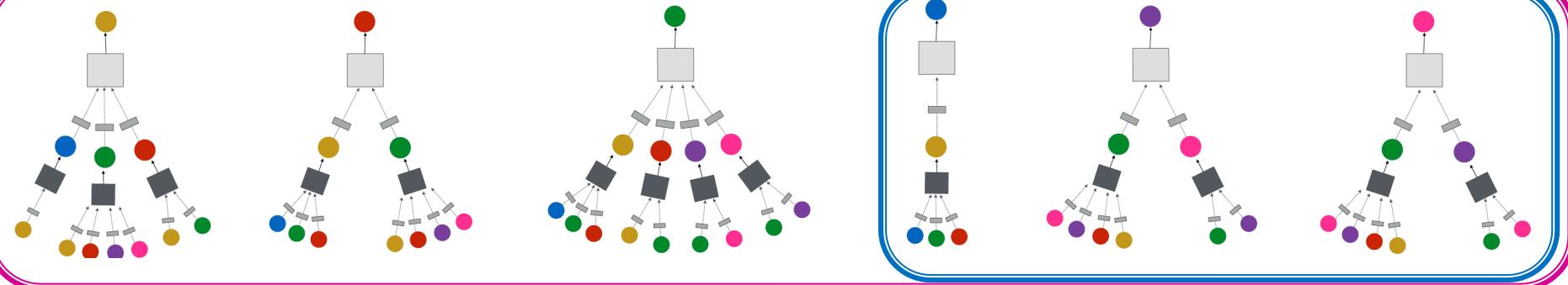
Model Design: Overview



INPUT GRAPH

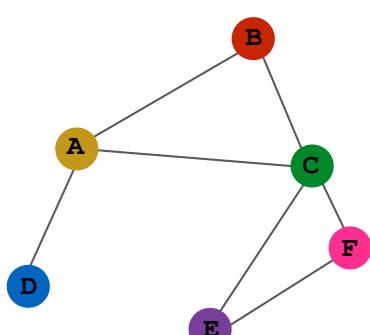
(4) Generate embeddings
for nodes as needed

Even for nodes we never
trained on!

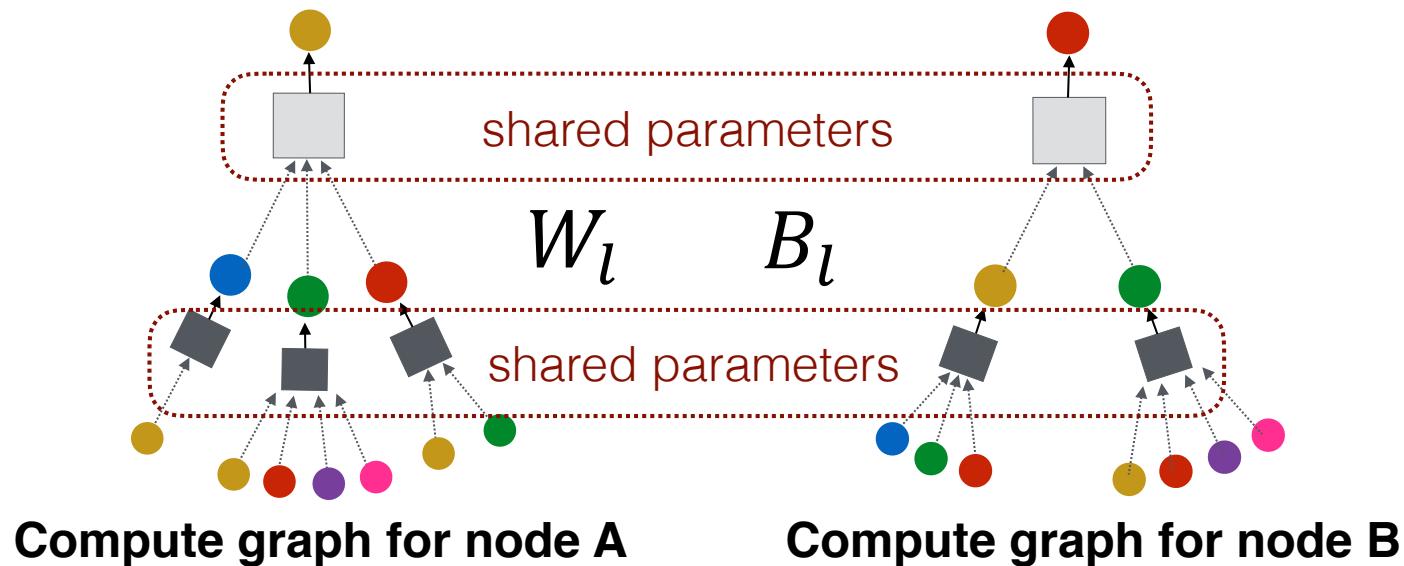


Inductive Capability

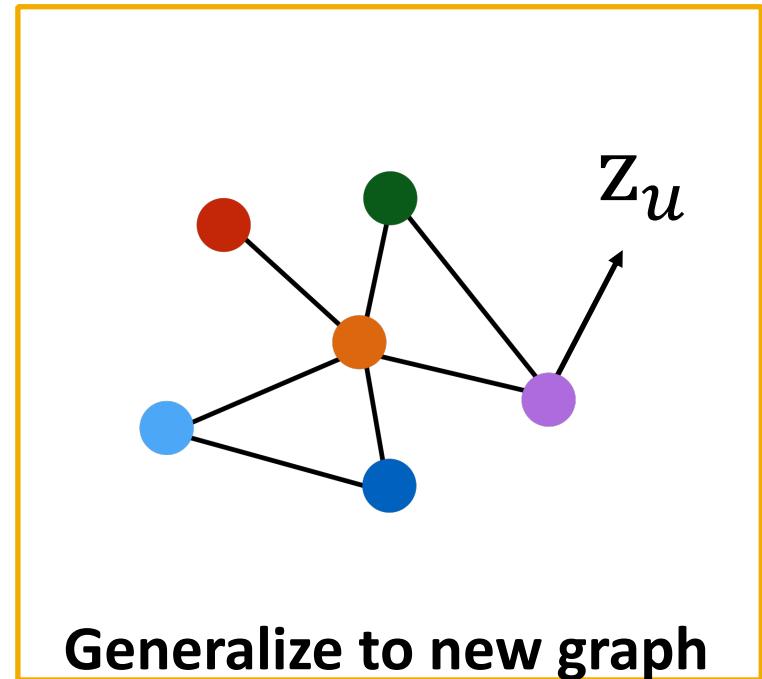
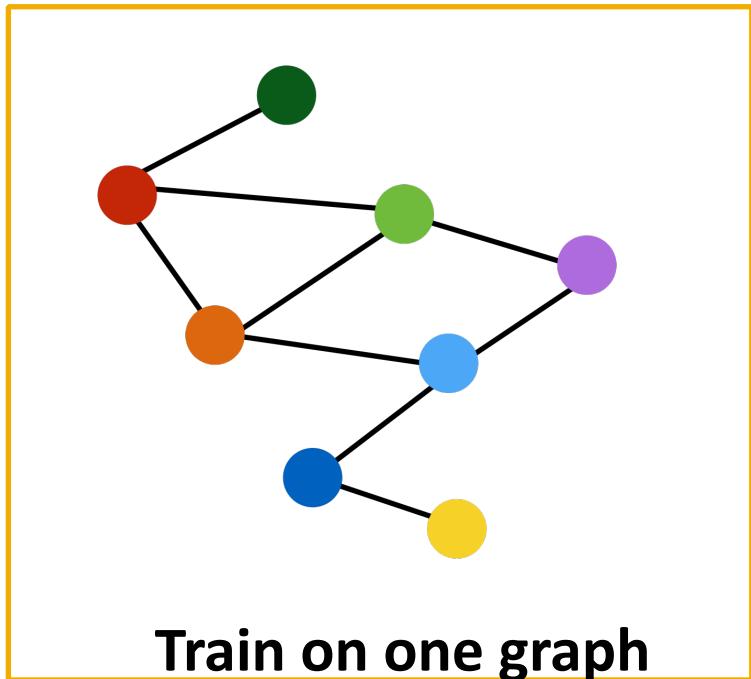
- The same aggregation parameters are shared for all nodes:
 - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes!**



INPUT GRAPH



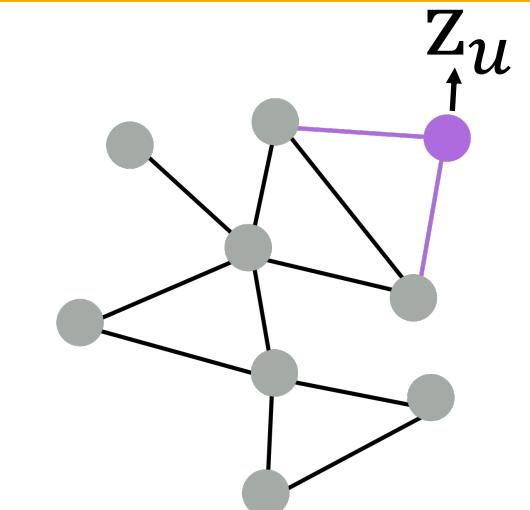
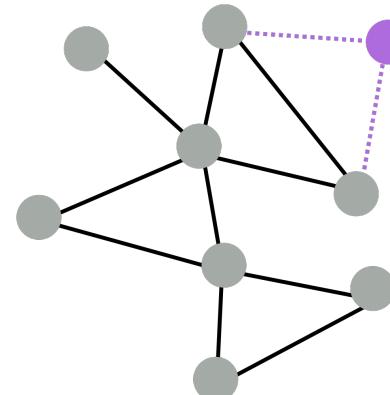
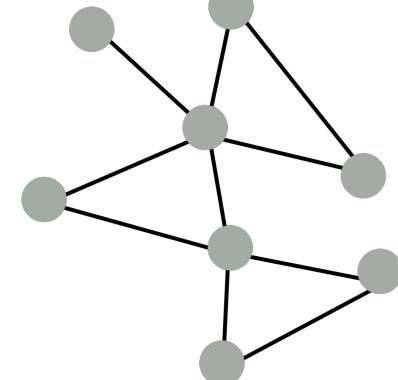
Inductive Capability: New Graphs



Inductive node embedding → Generalize to entirely unseen graphs

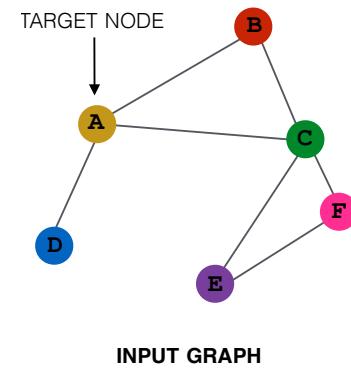
E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

Inductive Capability: New Nodes



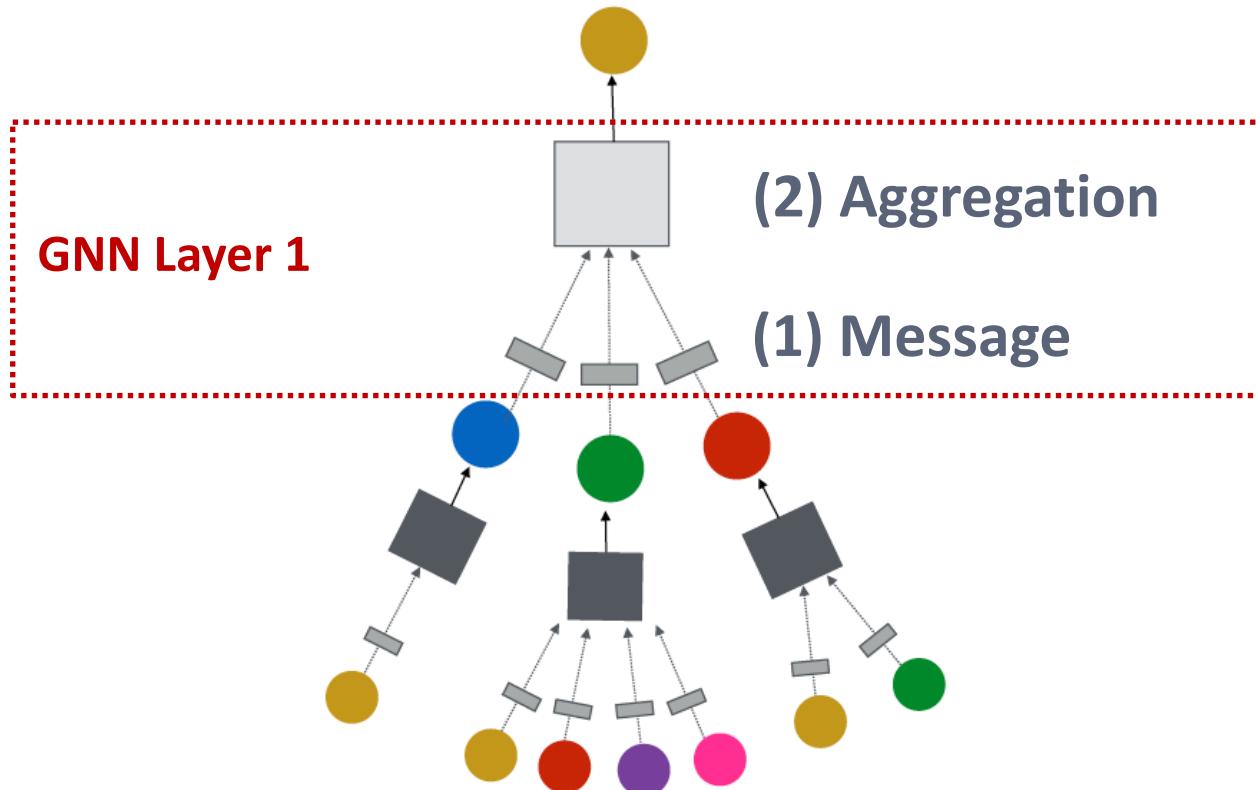
- Many application settings constantly encounter previously unseen nodes:
 - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings “on the fly”

A General GNN Framework (1)

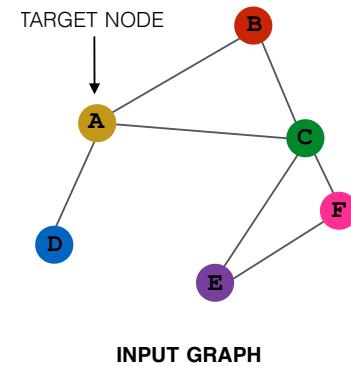


GNN Layer = Message + Aggregation

- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...



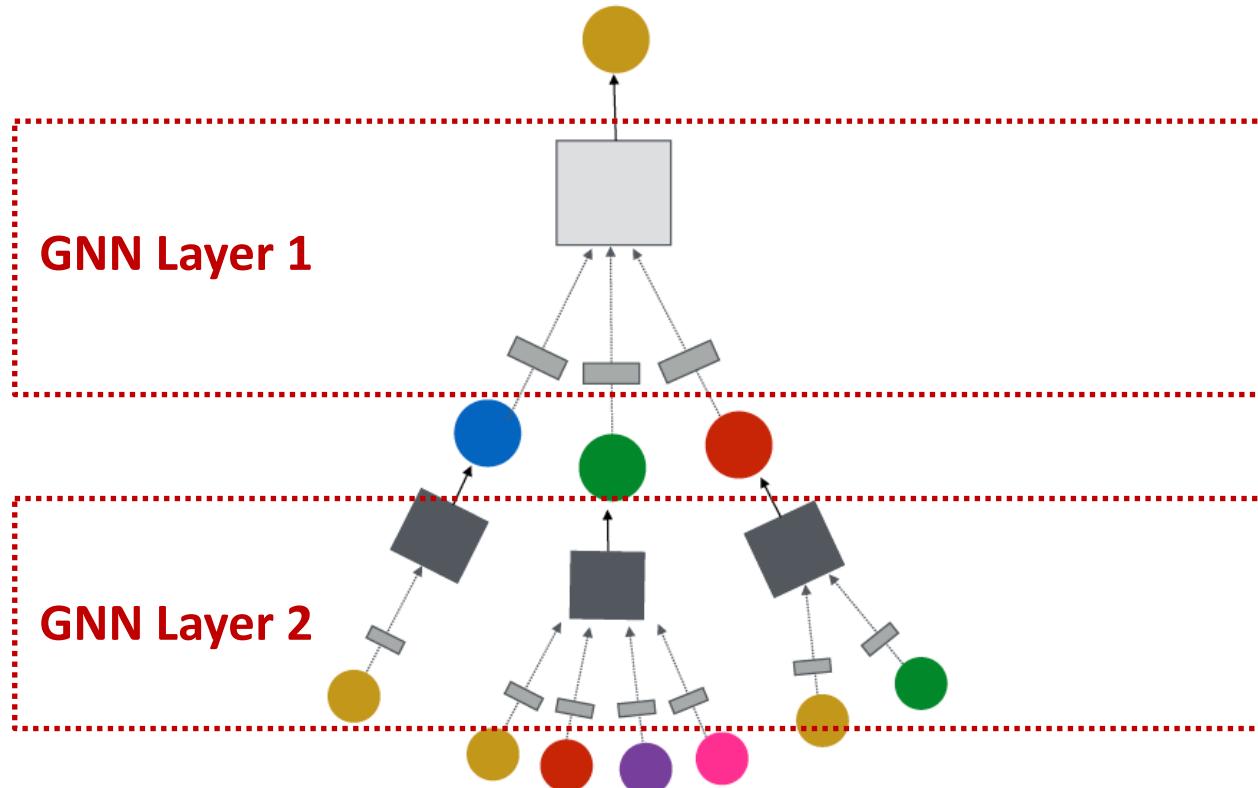
A General GNN Framework (2)



Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections

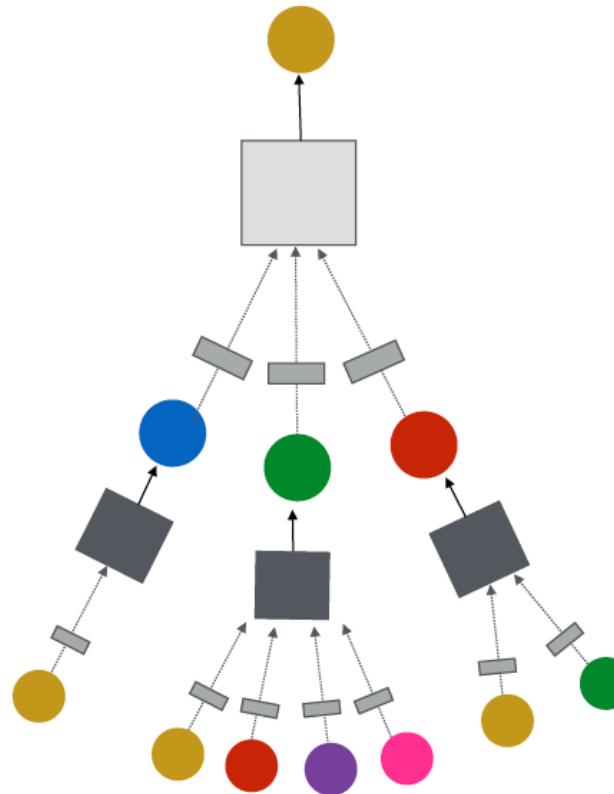
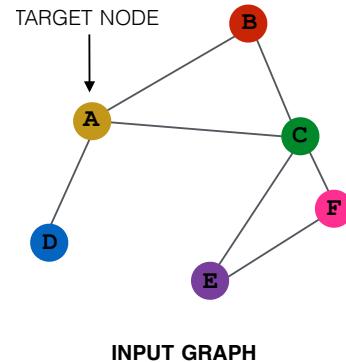
(3) Layer connectivity



A General GNN Framework (3)

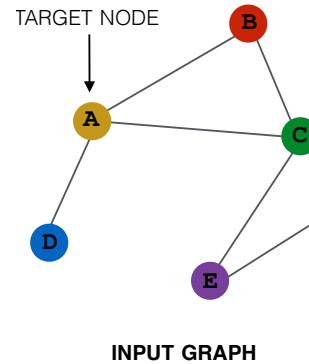
Idea: Raw input graph \neq computational graph

- Graph feature augmentation
- Graph structure augmentation



(4) Graph augmentation

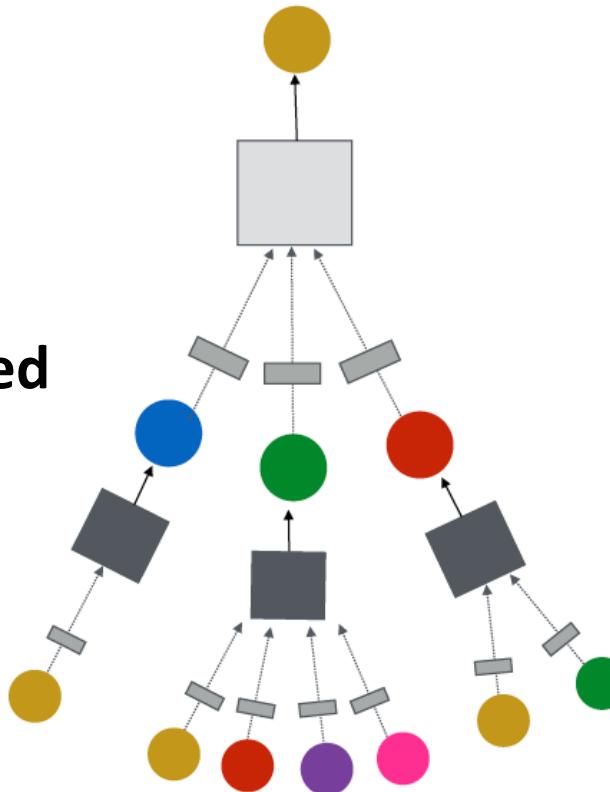
A General GNN Framework (4)



(5) Learning objective

How do we train a GNN

- **Supervised/Unsupervised objectives**
- **Node/Edge/Graph level objectives**



Stanford CS224W: A Single Layer of a GNN

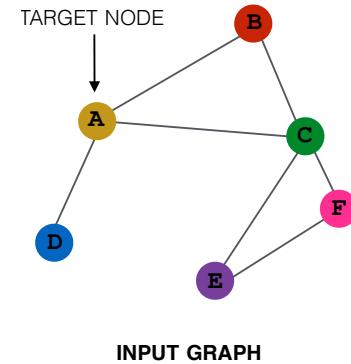
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>

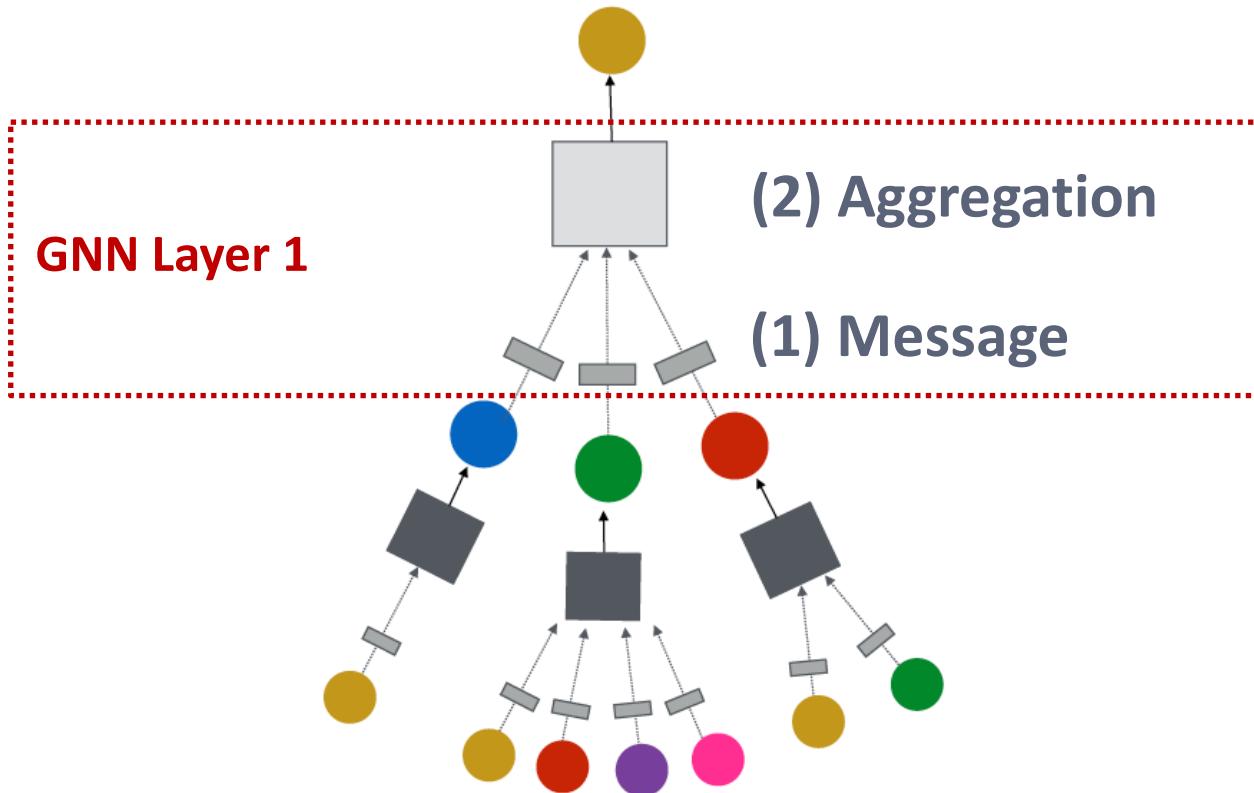


A GNN Layer



GNN Layer = Message + Aggregation

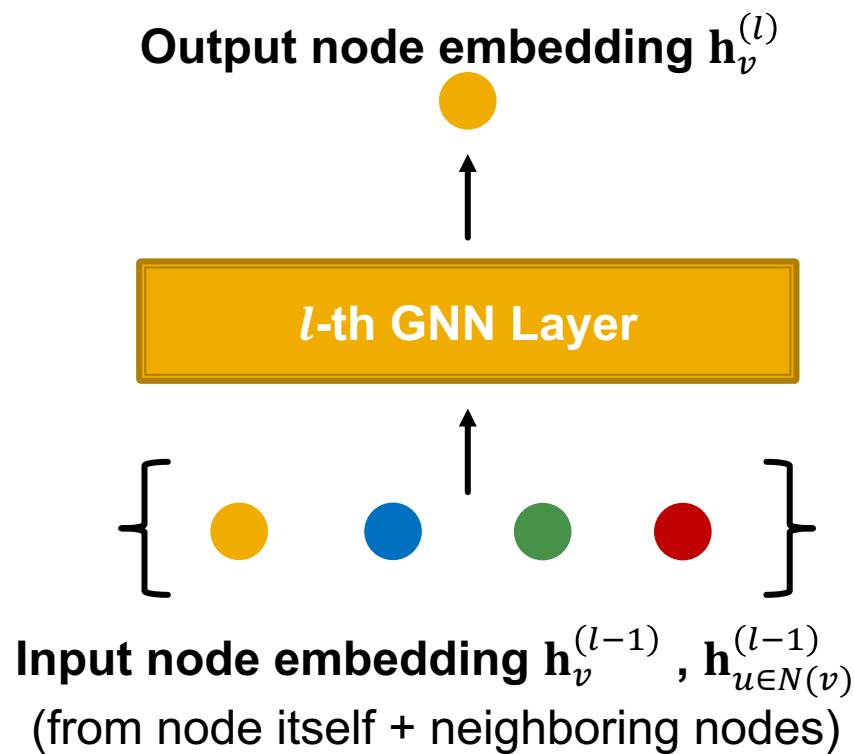
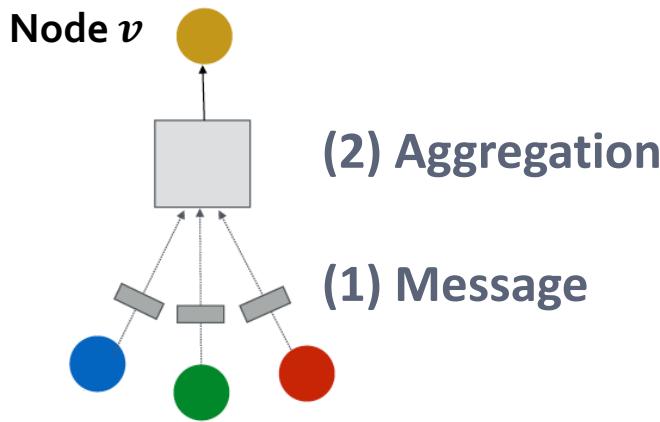
- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...



A Single GNN Layer

■ Idea of a GNN Layer:

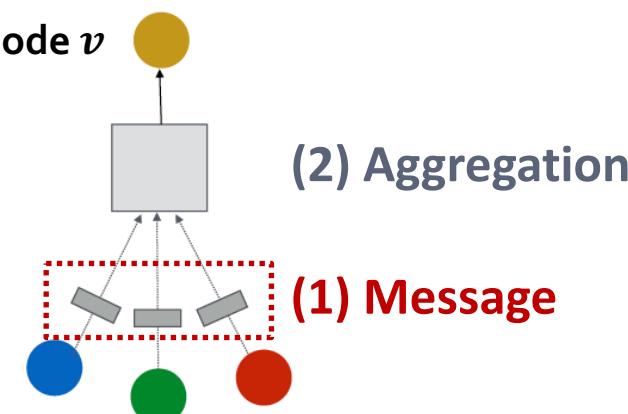
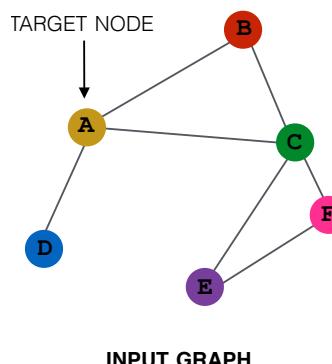
- Compress a set of vectors into a single vector
- Two step process:
 - (1) Message
 - (2) Aggregation



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

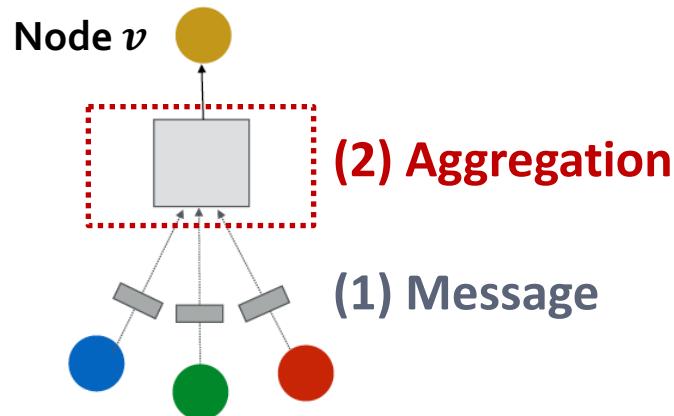
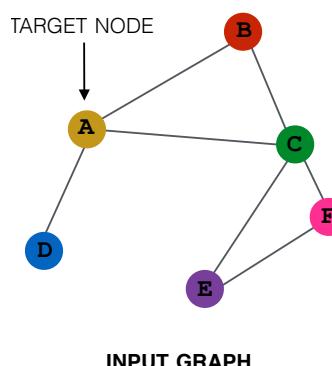
■ (2) Aggregation

- **Intuition:** Each node will aggregate the messages from node v 's neighbors

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

- **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

- **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

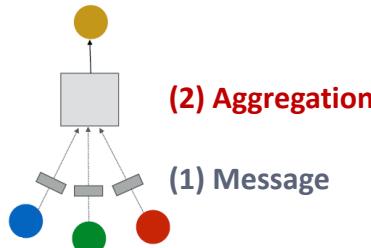
$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$

- **(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 $\sigma(\cdot)$: ReLU(\cdot), Sigmoid(\cdot) , ...
- Can be added to **message or aggregation**



Classical GNN Layers: GCN (1)

- (1) Graph Convolutional Networks (GCN)

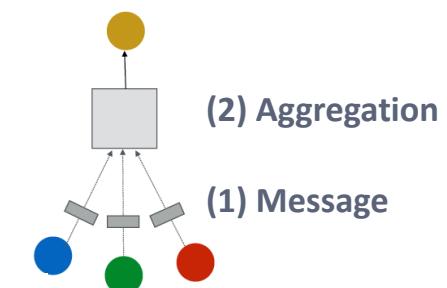
$$\mathbf{h}_v^{(l)} = \sigma \left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

- How to write this as Message + Aggregation?

Message

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

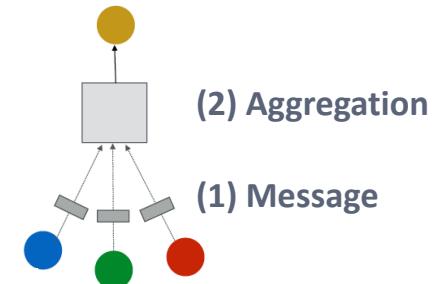
Aggregation



Classical GNN Layers: GCN (2)

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

Classical GNN Layers: GraphSAGE

- (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 Neighbor Aggregation

- **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: L₂ Normalization

■ ℓ_2 Normalization:

- **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
- $$\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\|\mathbf{h}_v^{(l)}\|_2} \quad \forall v \in V \text{ where } \|u\|_2 = \sqrt{\sum_i u_i^2} \text{ (\ell}_2\text{-norm)}$$
- 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

Classical GNN Layers: GAT (1)

■ (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

Classical GNN Layers: GAT (2)

■ (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.

Graph Attention Networks

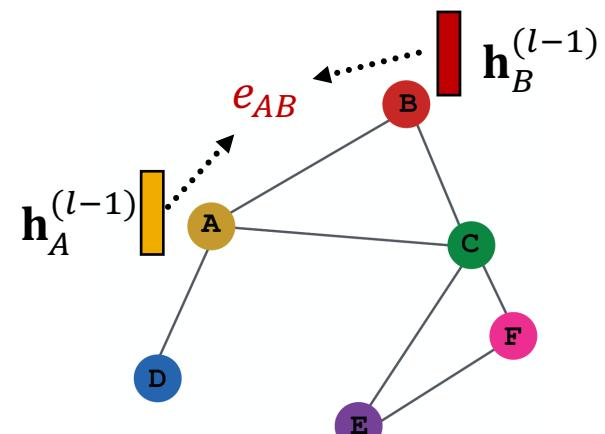
Can we do better than simple neighborhood aggregation?

Can we let weighting factors α_{vu} to 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

Attention Mechanism (1)

- 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



$$e_{AB} = a(\mathbf{W}^{(l)} \mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)})$$

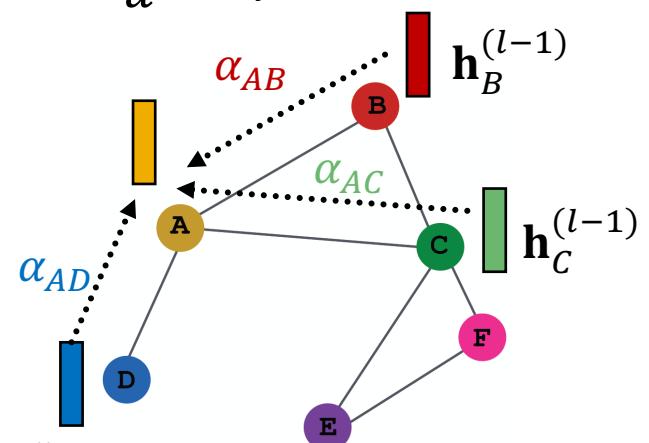
Attention Mechanism (2)

- **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} :

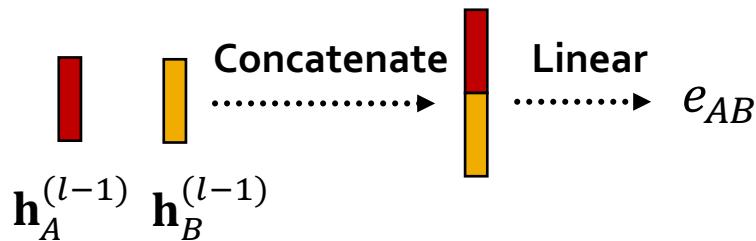
$$\begin{aligned}\mathbf{h}_A^{(l)} = \sigma(&\alpha_{AB} \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)} + \alpha_{AC} \mathbf{W}^{(l)} \mathbf{h}_C^{(l-1)} + \\ &\alpha_{AD} \mathbf{W}^{(l)} \mathbf{h}_D^{(l-1)})\end{aligned}$$



Attention Mechanism (3)

■ 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)



$$\begin{aligned} e_{AB} &= a \left(\mathbf{W}^{(l)} \mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)} \right) \\ &= \text{Linear} \left(\text{Concat} \left(\mathbf{W}^{(l)} \mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)} \right) \right) \end{aligned}$$

- 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

Attention Mechanism (4)

- **Multi-head attention:** Stabilizes the learning process of attention mechanism
 - Create **multiple attention scores** (each replica with a different set of parameters):

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

$$\mathbf{h}_v^{(l)}[2] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^2 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

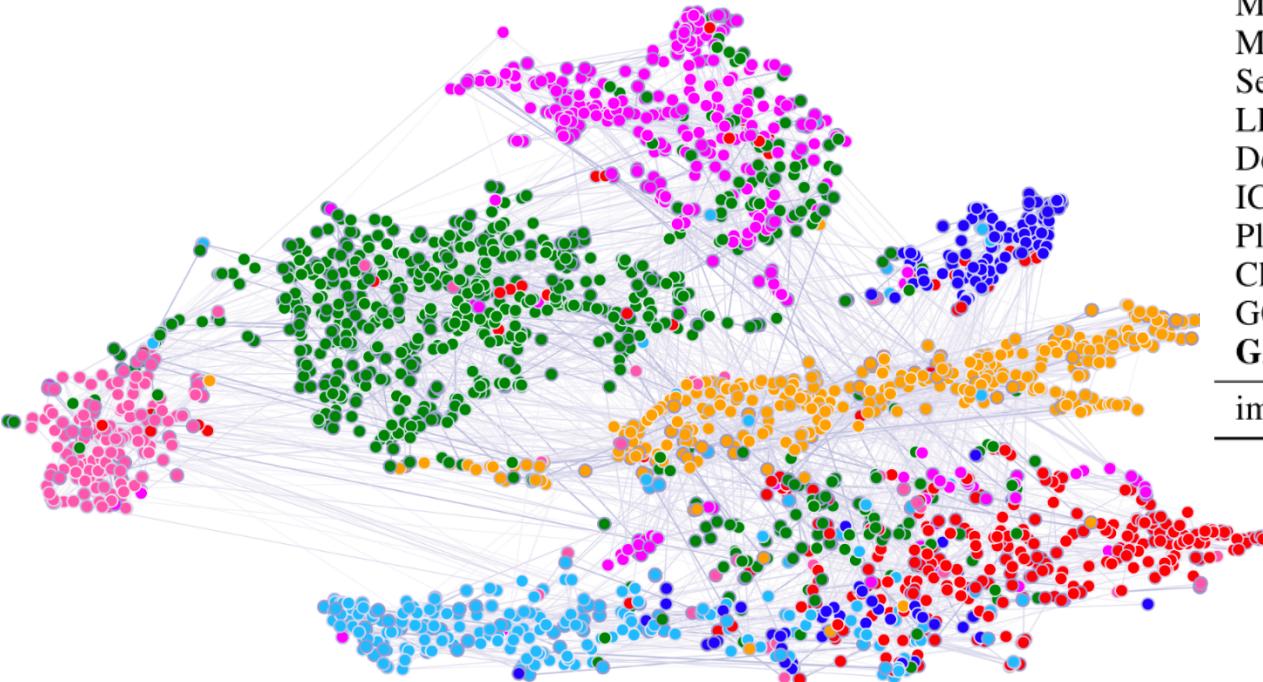
$$\mathbf{h}_v^{(l)}[3] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^3 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

- **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

GAT Example: Cora Citation Net



Method	Cora
MLP	55.1%
ManiReg (Belkin et al., 2006)	59.5%
SemiEmb (Weston et al., 2012)	59.0%
LP (Zhu et al., 2003)	68.0%
DeepWalk (Perozzi et al., 2014)	67.2%
ICA (Lu & Getoor, 2003)	75.1%
Planetoid (Yang et al., 2016)	75.7%
Chebyshev (Defferrard et al., 2016)	81.2%
GCN (Kipf & Welling, 2017)	81.5%
GAT	83.3%
improvement w.r.t GCN	1.8%

Attention mechanism can be used with many different graph neural network models

In many cases, attention leads to performance gains

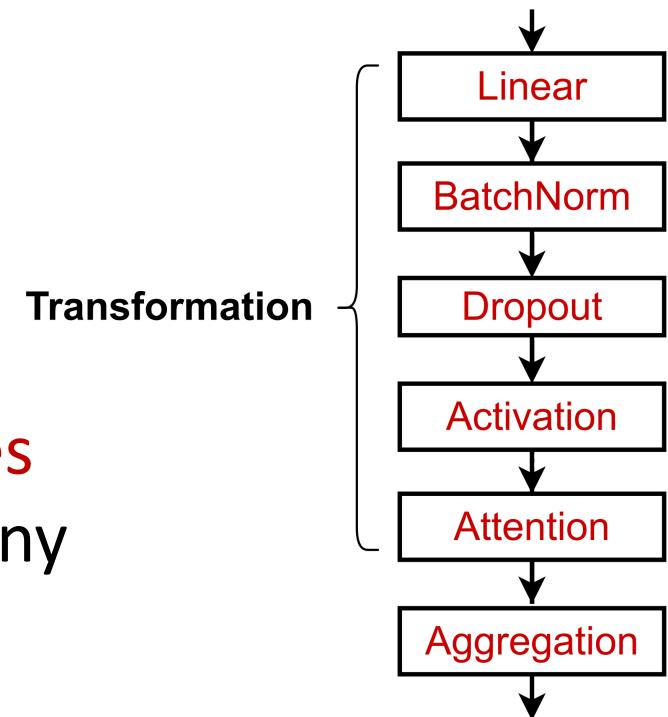
- **t-SNE plot of GAT-based node embeddings:**
 - Node color: 7 publication classes
 - Edge thickness: Normalized attention coefficients between nodes i and j , across eight attention heads, $\sum_k (\alpha_{ij}^k + \alpha_{ji}^k)$

GNN Layer in Practice

- In practice, these classic GNN layers are a great starting point

- We can often get better performance by considering a general GNN layer design
- Concretely, we can include modern deep learning modules that proved to be useful in many domains

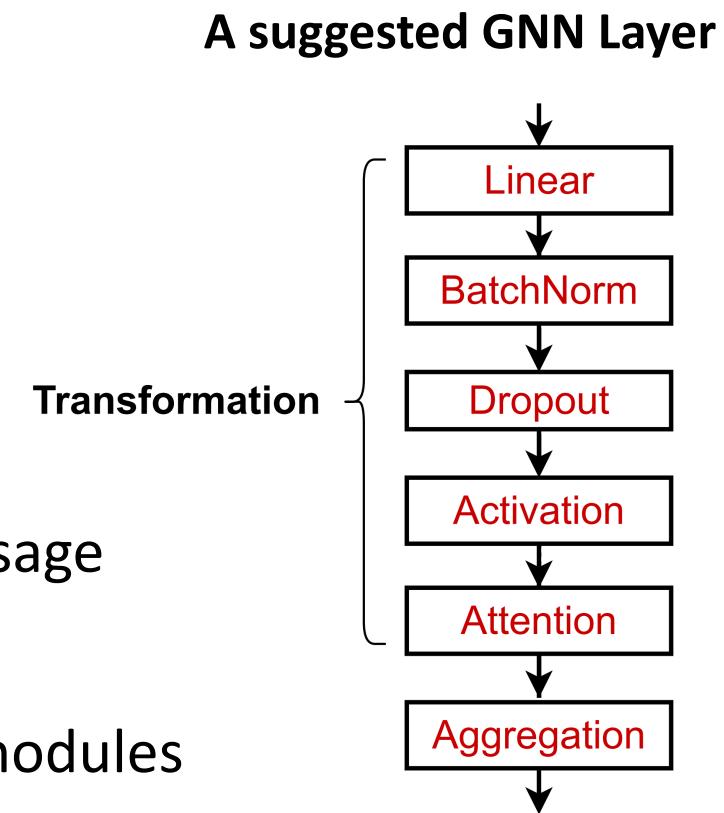
A suggested GNN Layer



GNN Layer in Practice

- Many modern deep learning modules can be incorporated into a GNN layer

- Batch Normalization:
 - Stabilize neural network training
- Dropout:
 - Prevent overfitting
- Attention/Gating:
 - Control the importance of a message
- More:
 - Any other useful deep learning modules



Batch Normalization

- **Goal:** Stabilize neural networks training
- **Idea:** Given a batch of inputs (node embeddings)
 - Re-center the node embeddings into zero mean
 - Re-scale the variance into unit variance

Input: $\mathbf{X} \in \mathbb{R}^{N \times D}$
 N node embeddings

Trainable Parameters:
 $\gamma, \beta \in \mathbb{R}^D$

Output: $\mathbf{Y} \in \mathbb{R}^{N \times D}$
Normalized node embeddings

Step 1:
**Compute the
mean and variance
over N embeddings**

$$\mu_j = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_{i,j}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (\mathbf{x}_{i,j} - \mu_j)^2$$

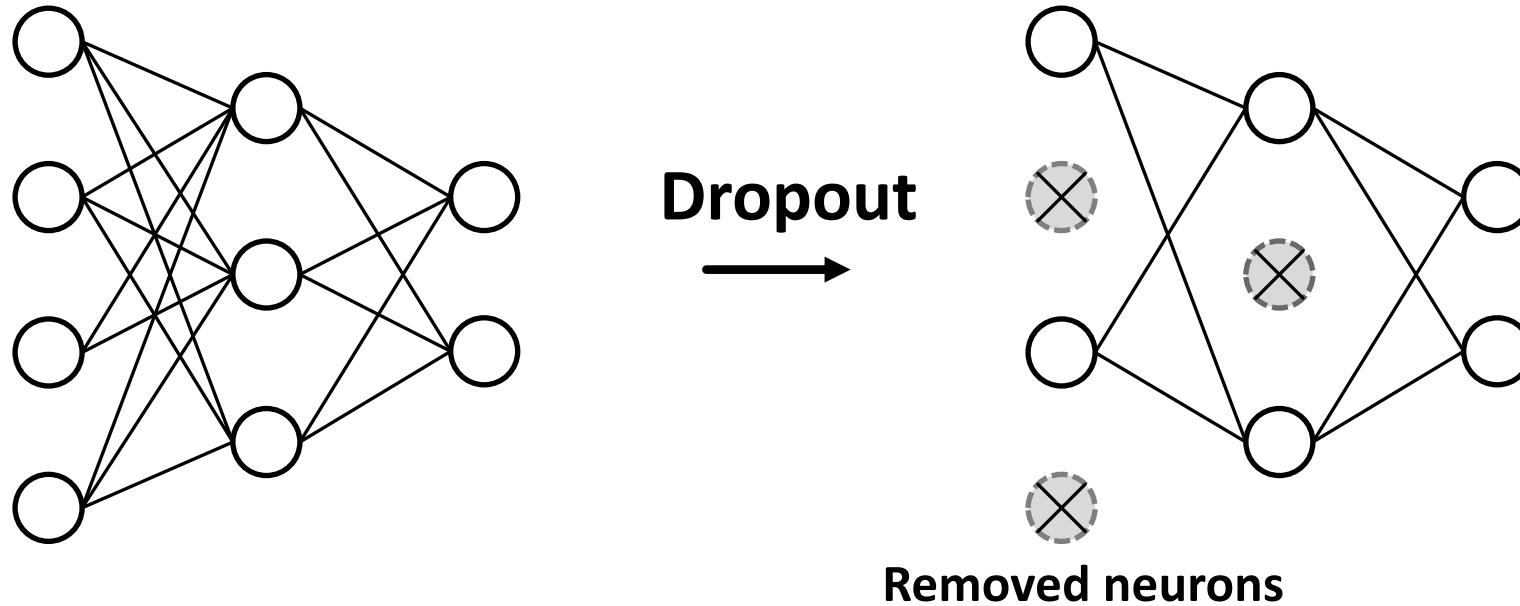
Step 2:
**Normalize the feature
using computed mean
and variance**

$$\hat{\mathbf{x}}_{i,j} = \frac{\mathbf{x}_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$\mathbf{y}_{i,j} = \gamma_j \hat{\mathbf{x}}_{i,j} + \beta_j$$

Dropout

- **Goal:** Regularize a neural net to prevent overfitting.
- **Idea:**
 - **During training:** with some probability p , randomly set neurons to zero (turn off)
 - **During testing:** Use all the neurons for computation

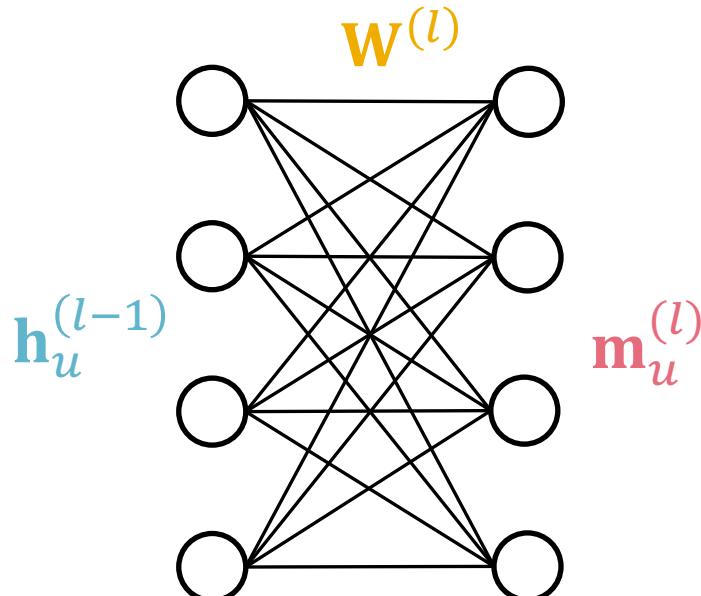


Dropout for GNNs

- In GNN, Dropout is applied to **the linear layer in the message function**

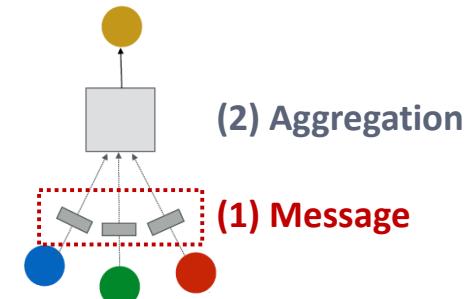
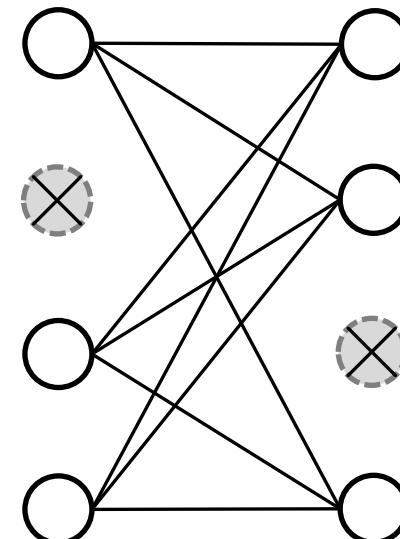
- A simple message function with linear layer:

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



Visualization of a linear layer

Dropout
→



Activation (Non-linearity)

Apply activation to i -th dimension of embedding \mathbf{x}

- Rectified linear unit (ReLU)

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

- Most commonly used

- Sigmoid

$$\sigma(\mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{x}_i}}$$

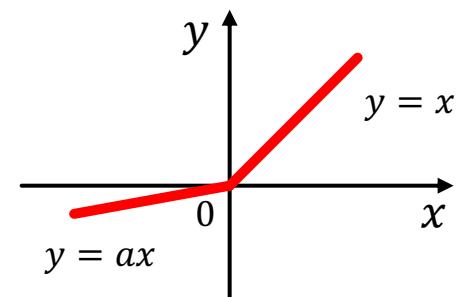
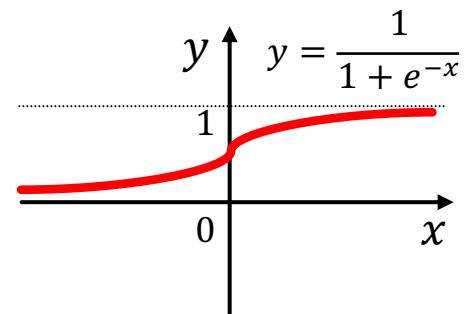
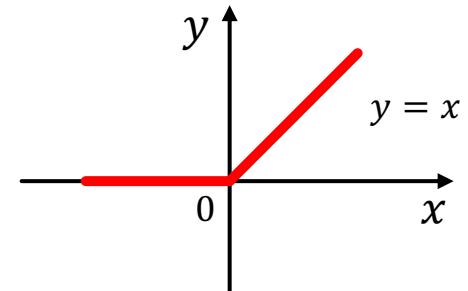
- Used only when you want to restrict the range of your embeddings

- Parametric ReLU

$$\text{PReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0) + a_i \min(\mathbf{x}_i, 0)$$

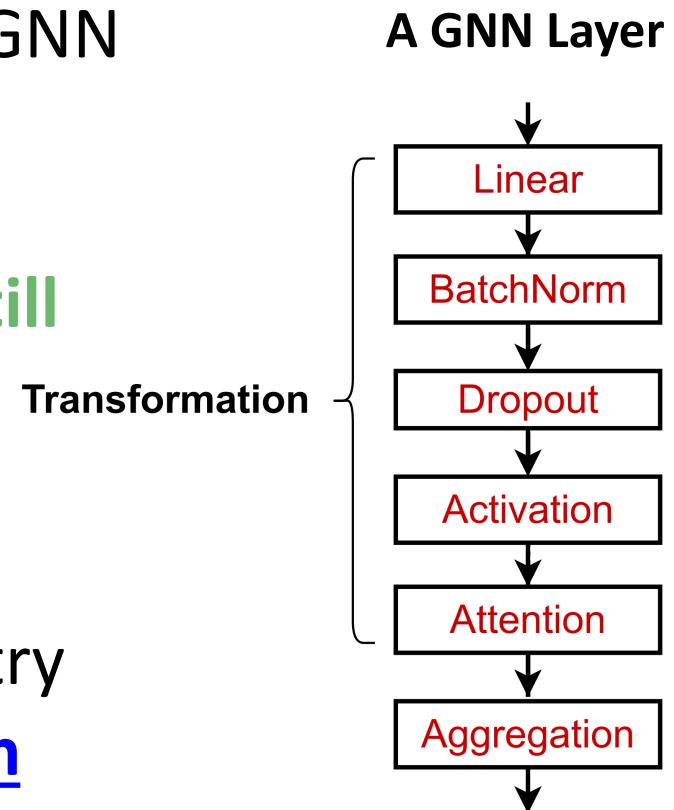
a_i is a trainable parameter

- Empirically performs better than ReLU



GNN Layer in Practice

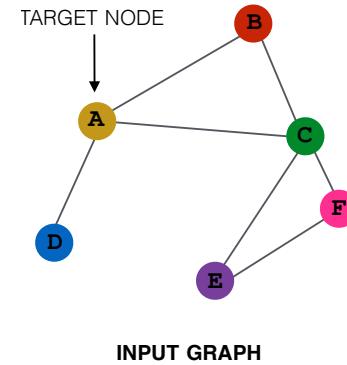
- **Summary:** Modern deep learning modules can be included into a GNN layer for better performance
- **Designing novel GNN layers is still an active research frontier!**
- **Suggested resources:** You can explore diverse GNN designs or try out your own ideas in [GraphGym](#)



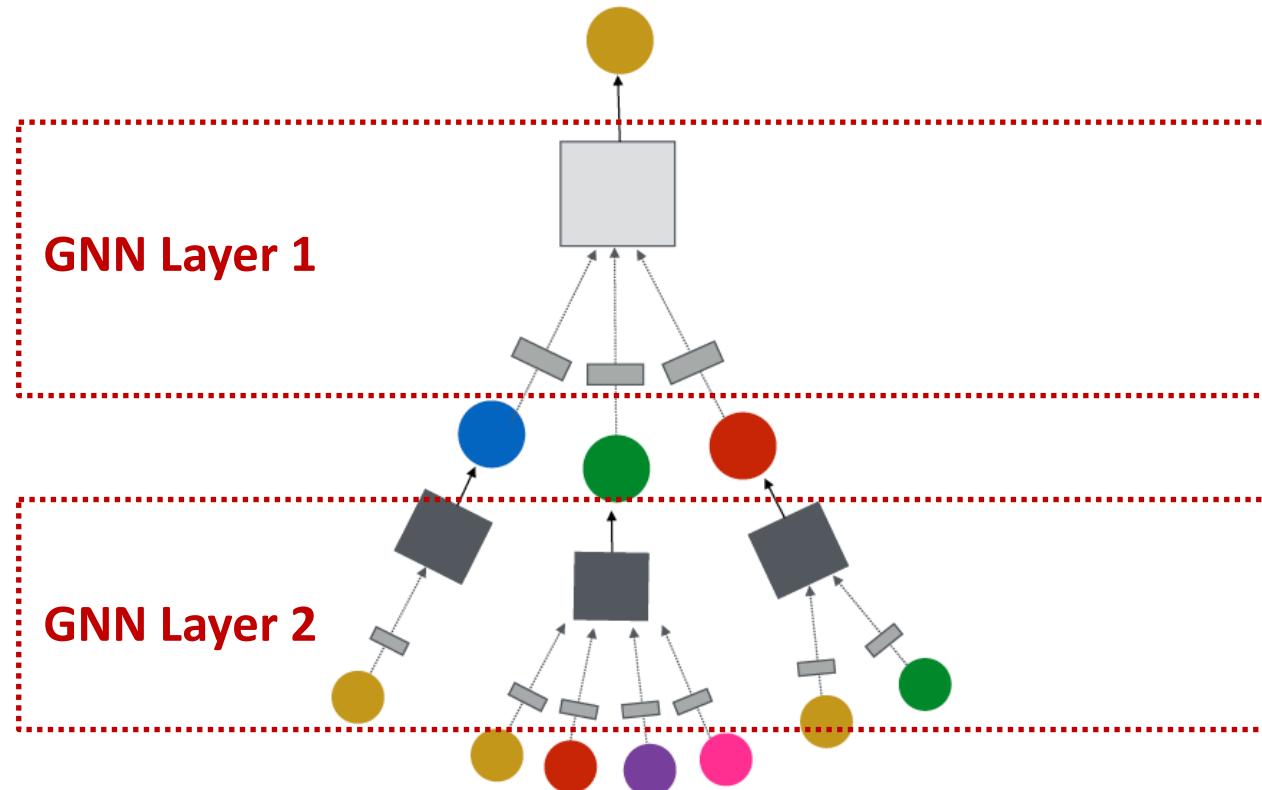
Stacking GNN Layers

How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections

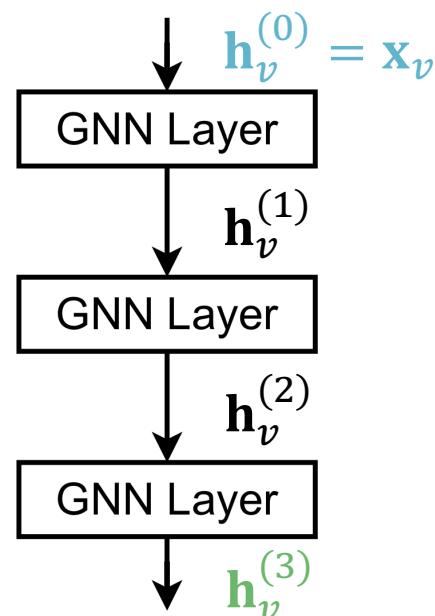


(3) Layer connectivity



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



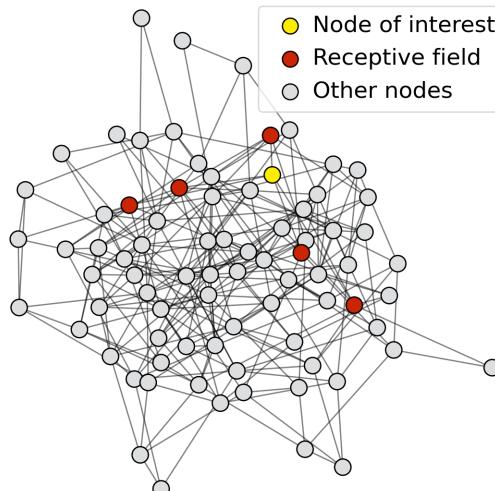
The Over-smoothing Problem

- **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?**

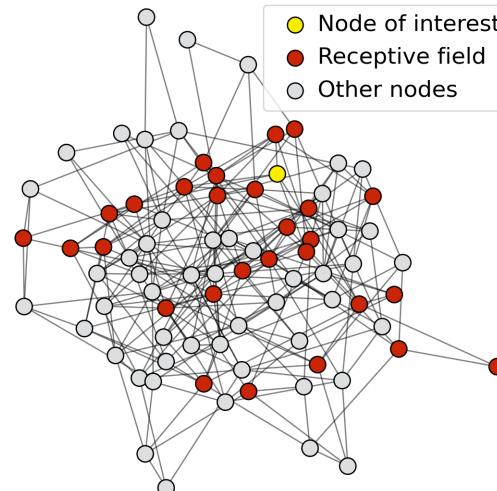
Receptive Field of a GNN

- **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

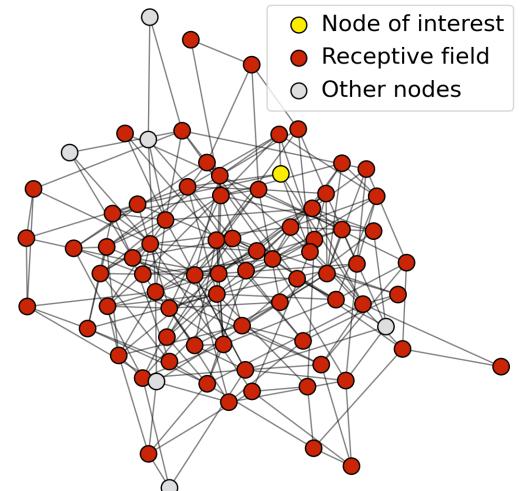
Receptive field for
1-layer GNN



Receptive field for
2-layer GNN



Receptive field for
3-layer GNN

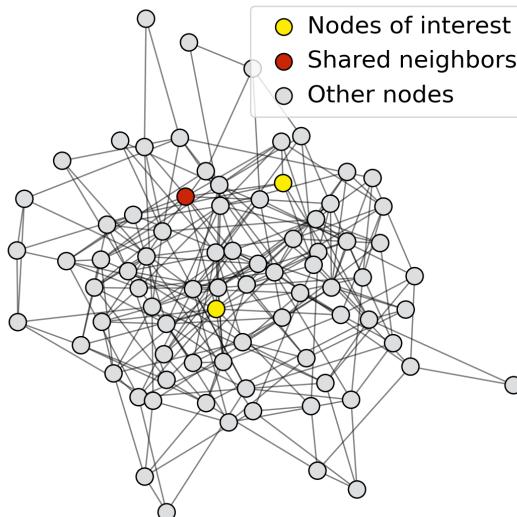


Receptive Field of a GNN

- **Receptive field overlap for two nodes**
 - **The shared neighbors quickly grows** when we increase the number of hops (num of GNN layers)

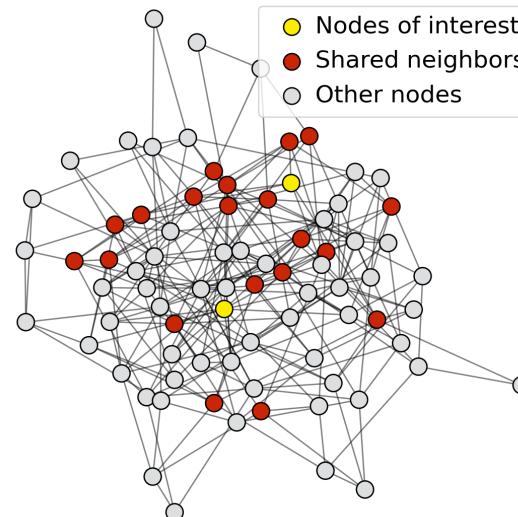
1-hop neighbor overlap

Only 1 node



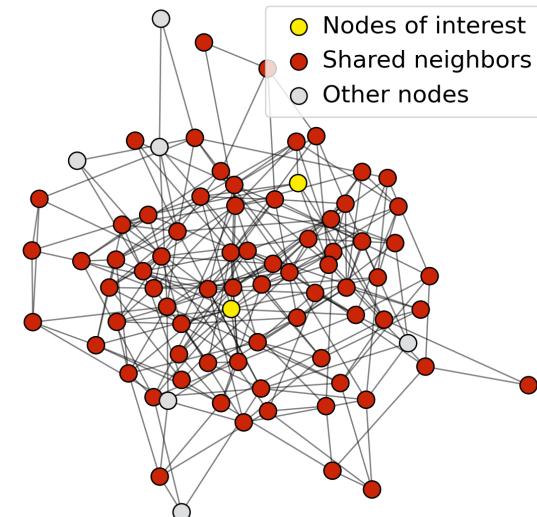
2-hop neighbor overlap

About 20 nodes



3-hop neighbor overlap

Almost all the nodes!



Receptive Field & Over-smoothing

- We can explain over-smoothing via the notion of receptive field
 - We knew 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?

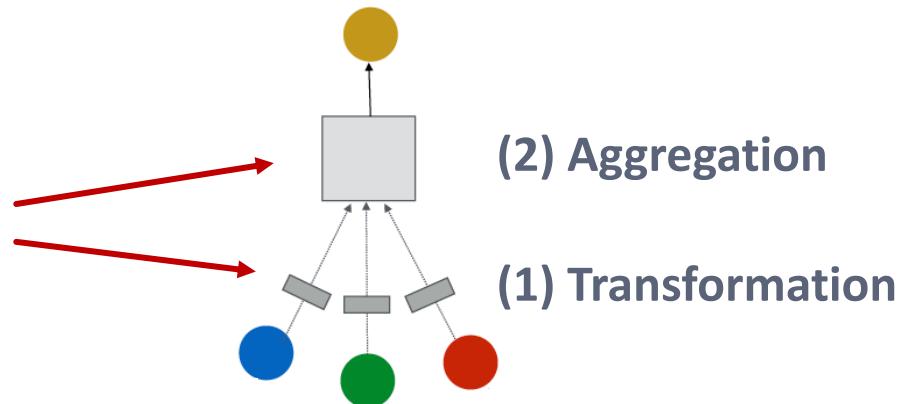
Design GNN Layer Connectivity

- **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

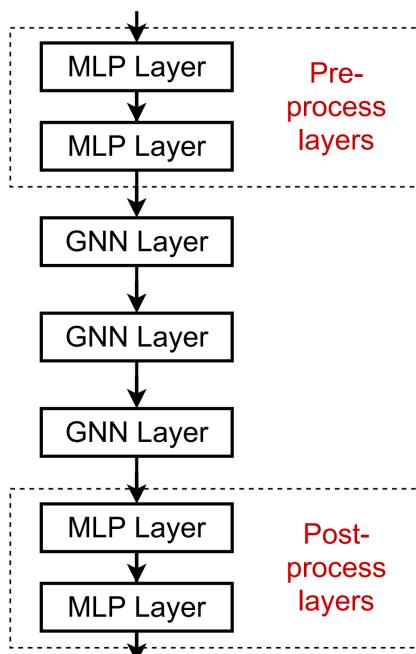
- How to make a shallow GNN more expressive?
- Solution 1: Increase the expressive power within each GNN layer
 - In our previous examples, each transformation or aggregation function only include one linear layer
 - We can make aggregation / transformation become a deep neural network!

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**



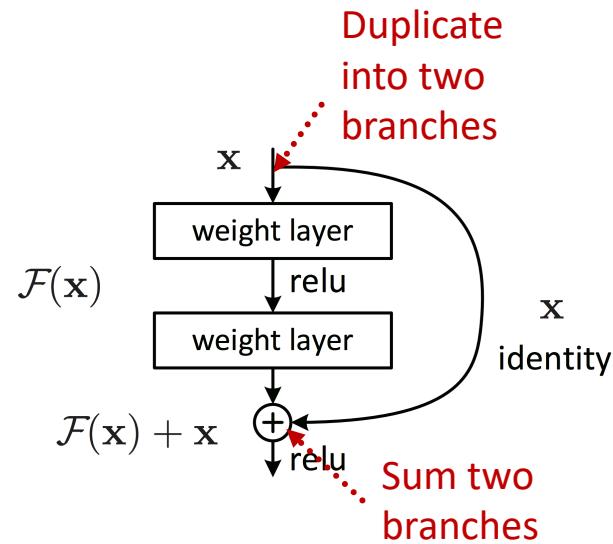
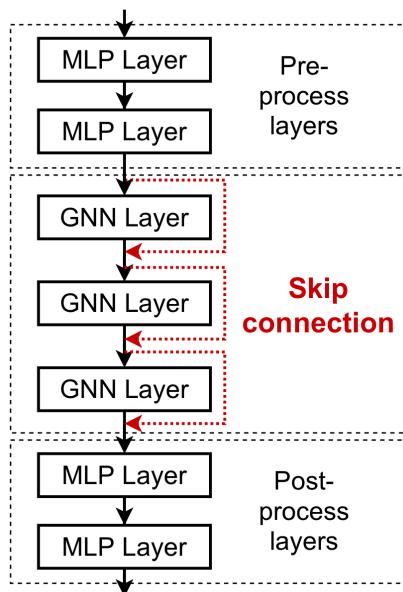
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

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:

$$\mathcal{F}(x)$$

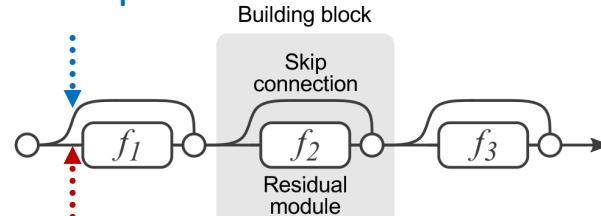
After adding shortcuts:

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

Idea 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**

Path 2: skip this module

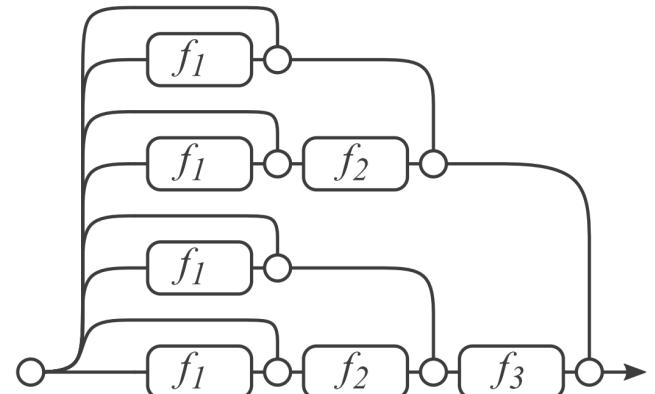


Path 1: include this module

(a) Conventional 3-block residual network

All the possible paths:

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



(b) Unraveled view of (a)

Veit et al. Residual Networks Behave Like Ensembles of Relatively Shallow Networks, ArXiv 2016

Example: GCN with Skip Connections

- A standard GCN layer

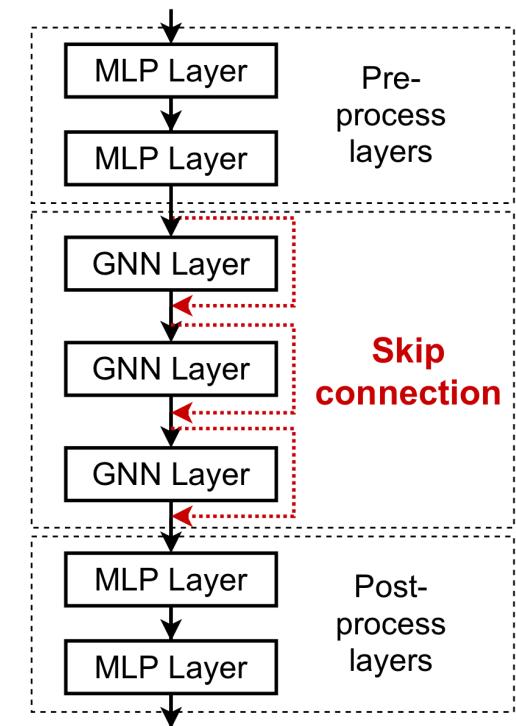
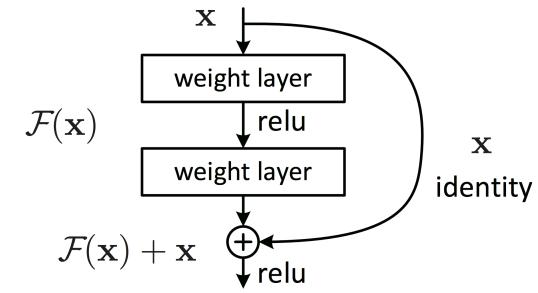
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

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

- A GCN layer with skip connection

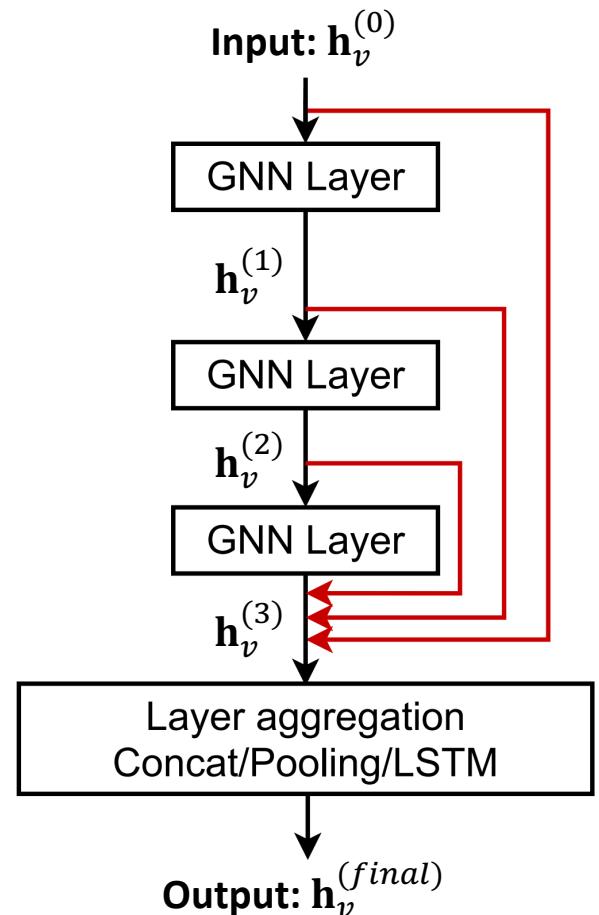
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)} \right)$$

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

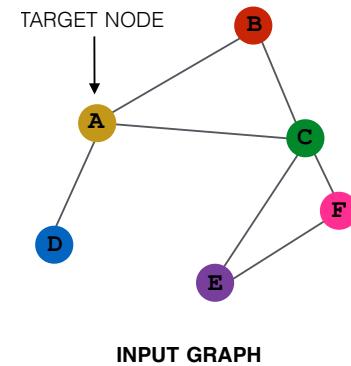


Other Options of Skip Connections

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

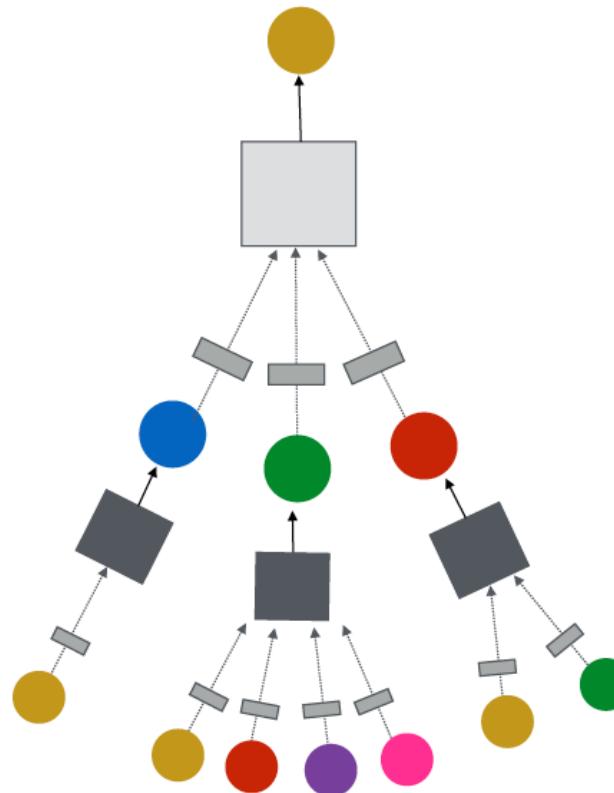


General GNN Framework



Idea: Raw input graph \neq computational graph

- Graph feature augmentation
- Graph structure manipulation



(4) Graph manipulation

Why Manipulate Graphs

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

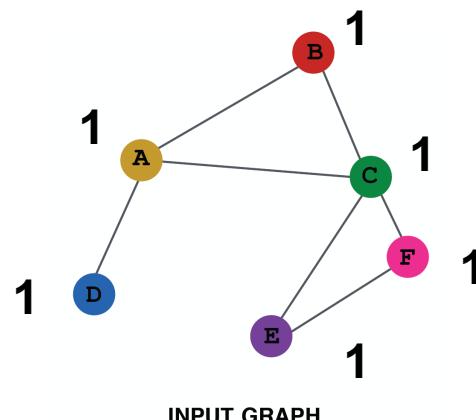
Graph Manipulation Approaches

- **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**
 - Will cover later in lecture: Scaling up GNNs

Feature Augmentation on Graphs

Why do we need feature augmentation?

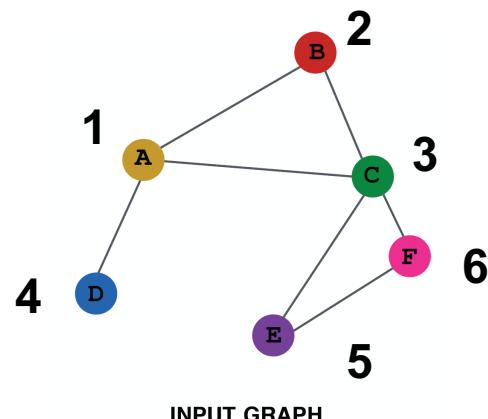
- **(1) Input graph does not have node features**
 - This is common when we only have the adj. matrix
- **Standard approaches:**
- **a) Assign constant values to nodes**



Feature Augmentation on Graphs

Why do we need feature augmentation?

- (1) Input graph does not have node features
 - This is common when we only have the adj. matrix
- Standard approaches:
 - b) Assign unique IDs to nodes
 - These IDs are converted into one-hot vectors



One-hot vector for node with ID=5

ID = 5
↓
[0, 0, 0, 0, 1, 0]

Total number of IDs = 6

Feature Augmentation on Graphs

■ Feature augmentation: **constant** vs. **one-hot**

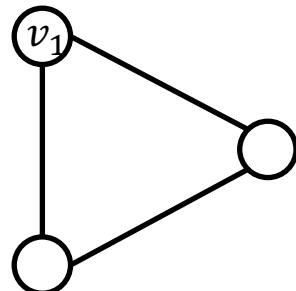
	Constant node feature	One-hot node feature
	<p>Constant node feature</p> <p>INPUT GRAPH</p>	<p>One-hot node feature</p> <p>INPUT GRAPH</p>
Expressive power	Medium. All the nodes are identical, but GNN can still learn from the graph structure	High. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	High. Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN	Low. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs
Computational cost	Low. Only 1 dimensional feature	High. $O(V)$ dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

Feature Augmentation on Graphs

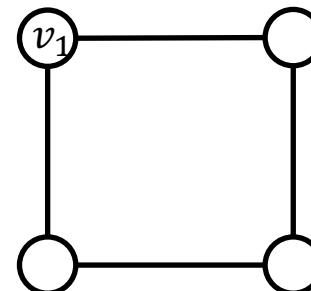
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- Example: Cycle count feature
 - Can GNN learn the length of a cycle that v_1 resides in?
 - Unfortunately, no

v_1 resides in a cycle with length 3



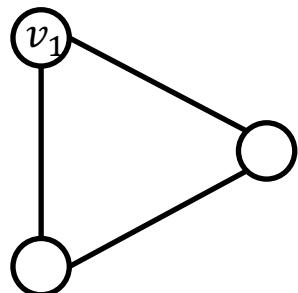
v_1 resides in a cycle with length 4



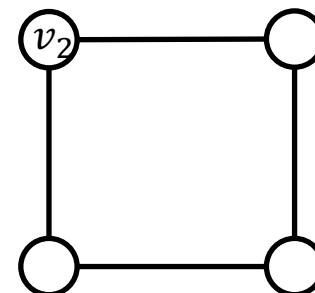
Feature Augmentation on Graphs

- v_1 cannot differentiate which graph it resides in
 - Because all the nodes in the graph have degree of 2
 - The computational graphs will be the same binary tree

v_1 resides in a cycle with length 3



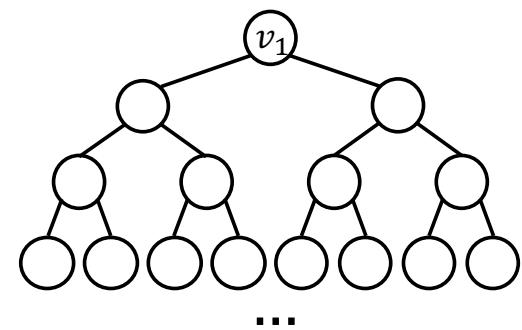
v_1 resides in a cycle with length 4



v_1 resides in a cycle with infinite length



The computational graphs for node v_1 are always the same



Feature Augmentation on Graphs

Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- Solution:
 - We can use *cycle count* as augmented node features

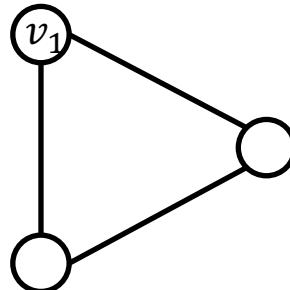
We start
from cycle
with length 0

Augmented node feature for v_1

[0, 0, 0, 1, 0, 0]



v_1 resides in a cycle with length 3

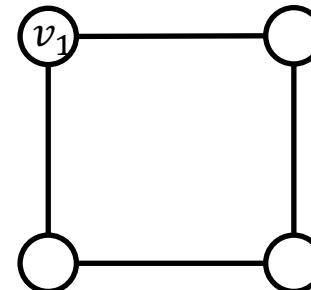


Augmented node feature for v_1

[0, 0, 0, 0, 1, 0]



v_1 resides in a cycle with length 4



Feature Augmentation on Graphs

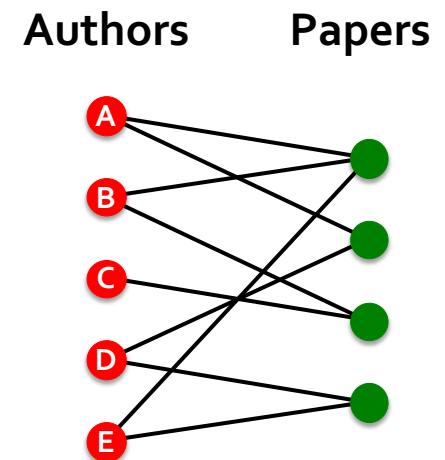
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- Other commonly used augmented features:
 - Clustering coefficient
 - PageRank
 - Centrality
 - ...
- Any feature we have introduced in Lecture 2 can be used!

Add Virtual Nodes / Edges

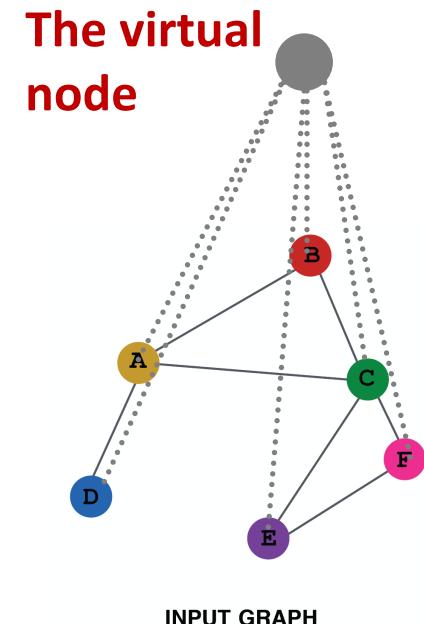
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
 - **Common approach:** Connect 2-hop neighbors via virtual edges
 - **Intuition:** Instead of using adj. matrix A for GNN computation, use $A + A^2$

- **Use cases:** Bipartite graphs
 - Author-to-papers (they authored)
 - 2-hop virtual edges make an author-author collaboration graph



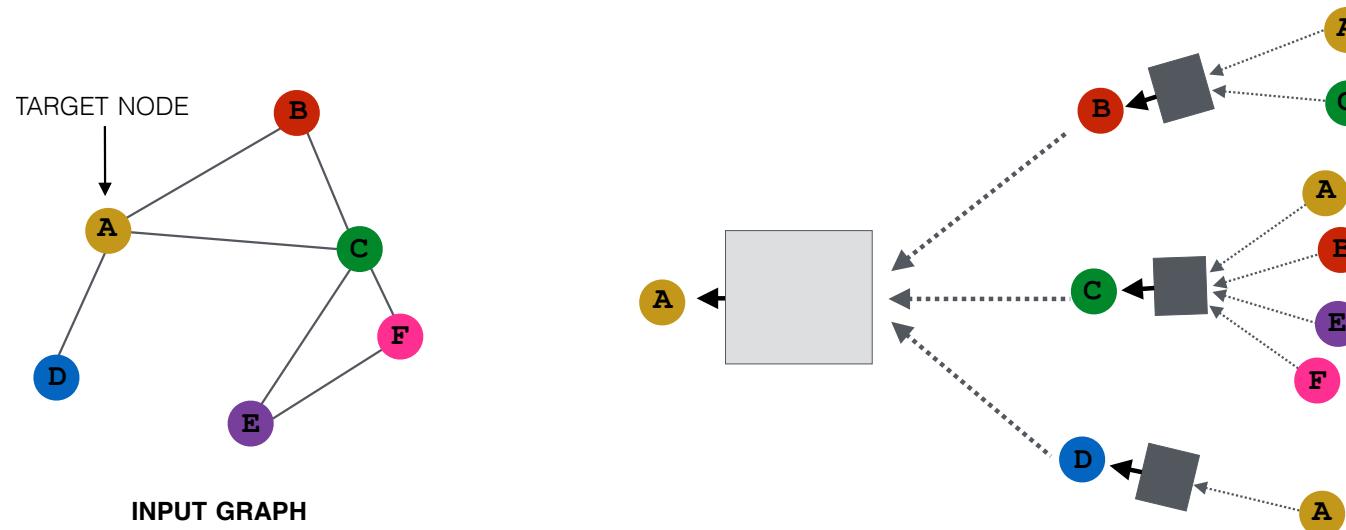
Add Virtual Nodes / Edges

- **Motivation:** Augment sparse graphs
- **(2) Add virtual nodes**
 - The virtual node will connect to all the nodes in the graph
 - Suppose in a sparse graph, two nodes have shortest path distance of 10
 - After adding the virtual node, **all the nodes will have a distance of 2**
 - Node A – Virtual node – Node B
 - **Benefits:** Greatly **improves message passing in sparse graphs**



Node Neighborhood Sampling

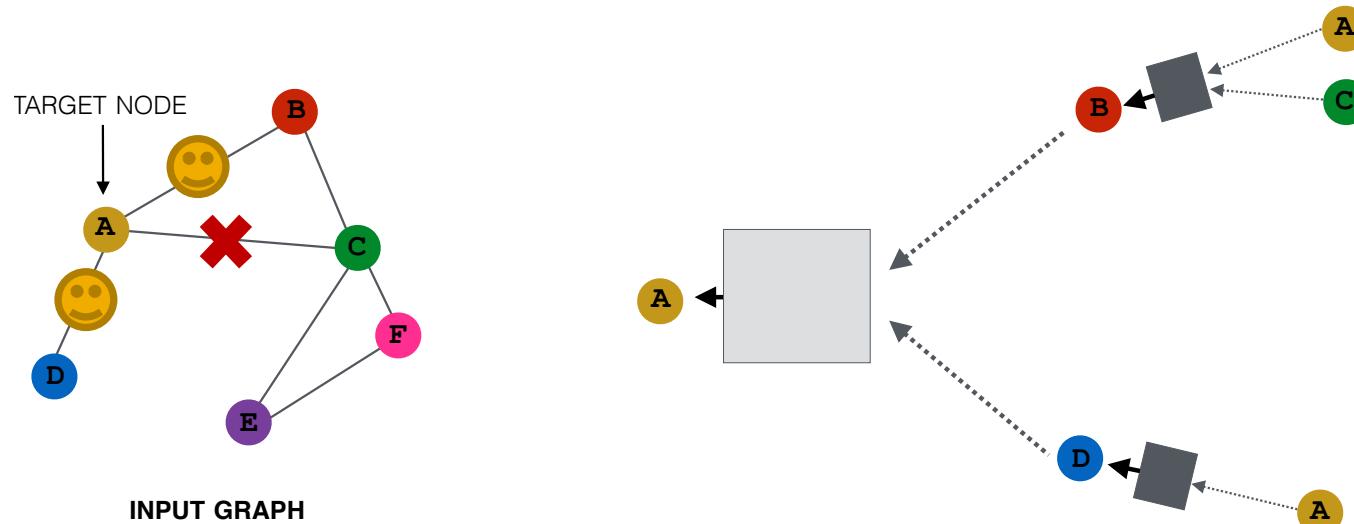
- Previously:
 - All the nodes are used for message passing



- New idea: (Randomly) sample a node's neighborhood for message passing

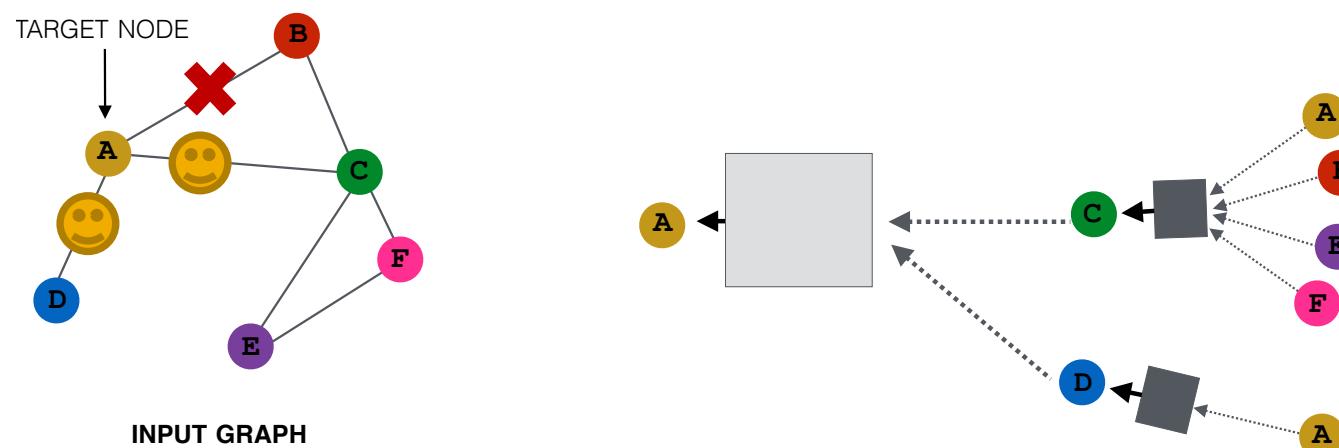
Neighborhood Sampling Example

- For example, we can randomly choose 2 neighbors to pass messages
 - Only nodes B and D will pass message to A



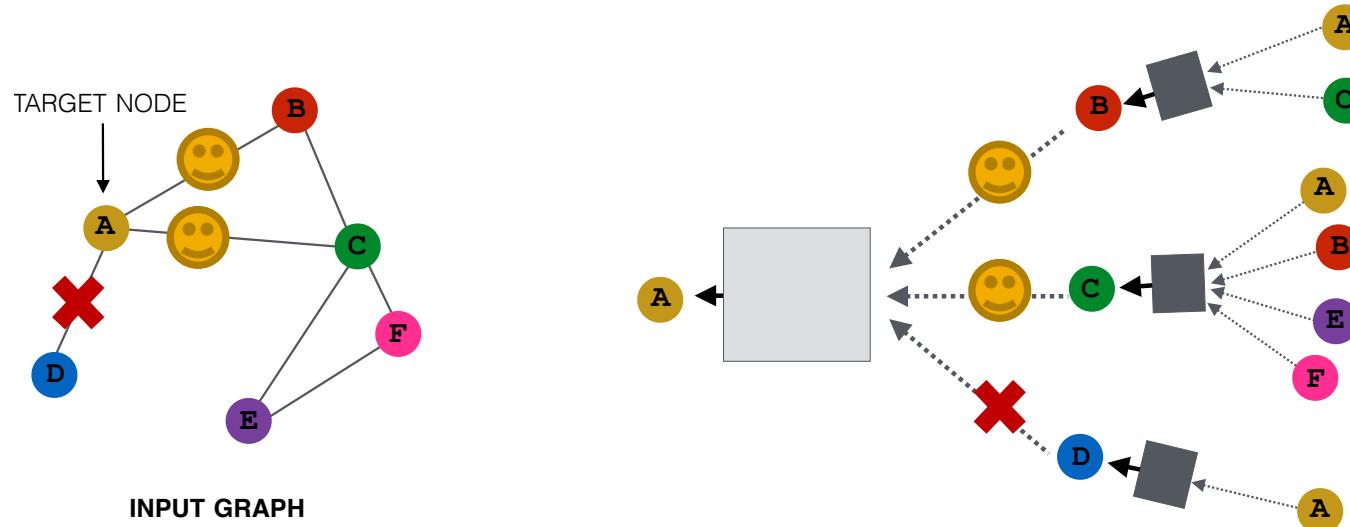
Neighborhood Sampling Example

- Next time when we compute the embeddings, we can sample different neighbors
 - Only nodes C and D will pass message to A



Neighborhood Sampling Example

- In expectation, we can get embeddings similar to the case where all the neighbors are used
 - Benefits: greatly reduce computational cost
 - And in practice it works great!



Summary of the lecture

- **Recap:** A general perspective for GNNs
 - **GNN Layer:**
 - Transformation + Aggregation
 - Classic GNN layers: GCN, GraphSAGE, GAT
 - **Layer connectivity:**
 - Deciding number of layers
 - Skip connections
 - **Graph Manipulation:**
 - Feature augmentation
 - Structure manipulation
- **Next:** GNN objectives, GNN in practice