

# Graph Neural Networks (cont)

COMP9312\_24T2



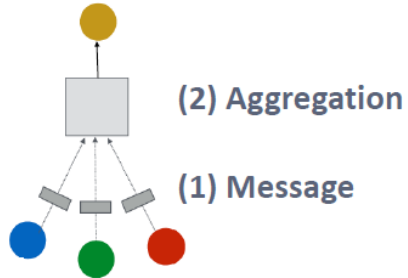
**UNSW**  
SYDNEY

Several slides are from Stanford CS224W: Machine Learning with Graphs

# GCN & GraphSage & GAT

# Graph Convolutional Networks (GCN)

A simplified version:

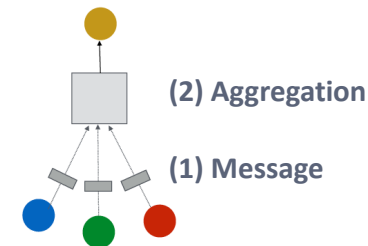
$$\mathbf{h}_v^{(l)} = \sigma \left( \underbrace{\sum_{u \in N(v)} \left( \underbrace{\mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}}_{\text{Message}} \right)}_{\text{Aggregation}} \right)$$


The diagram illustrates the simplified GCN operation. It shows a central node (yellow) receiving messages from three neighboring nodes (blue, green, red). The process is labeled (1) Message and (2) Aggregation.

The original one:  $H^k = D^{-1/2} A D^{1/2} H^{k-1}$

# 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

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

- **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

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

$$\text{AGG} = \underbrace{\sum_{u \in N(v)} \mathbf{h}_u^{(l-1)}}_{\text{Aggregation}} \underbrace{|N(v)|}_{\text{Message computation}}$$

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

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

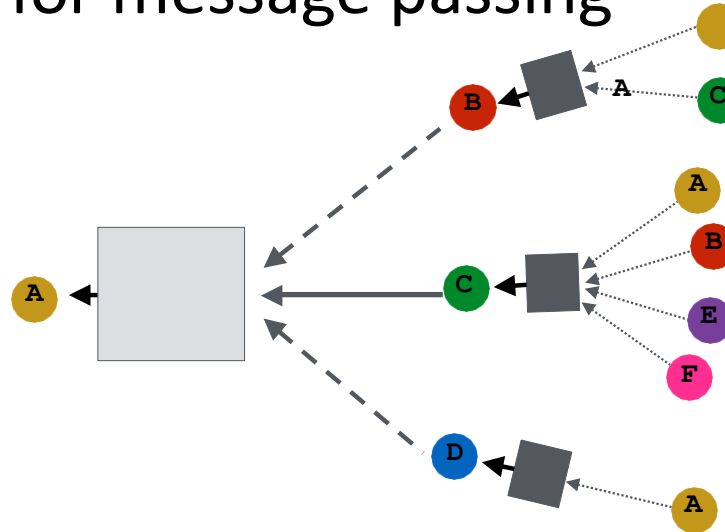
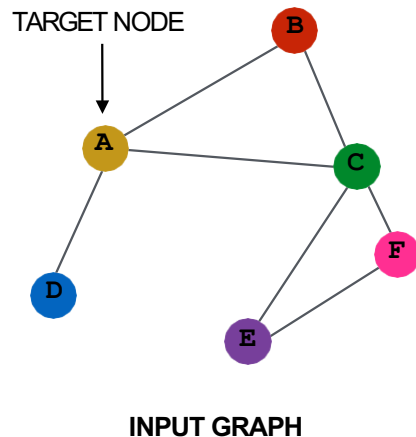
- **LSTM:** Apply LSTM to reshuffled of neighbors

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

# GraphSAGE: Neighborhood Sampling

## ■ Previously:

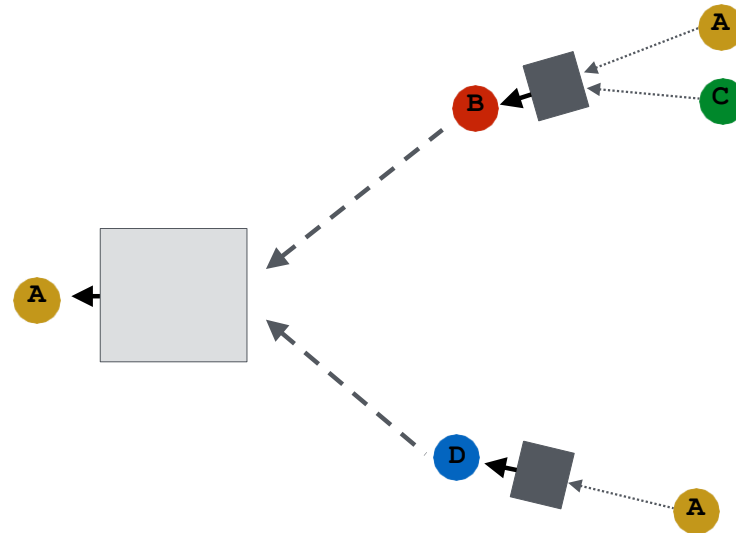
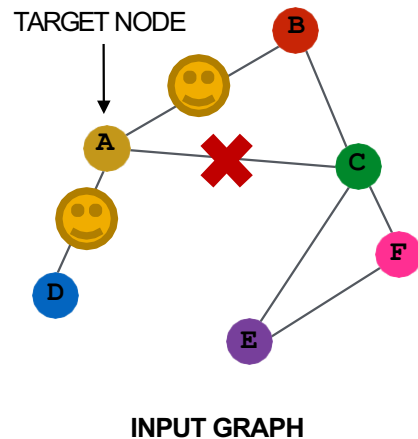
- All the nodes are used for message passing



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

# GraphSAGE: Neighborhood Sampling

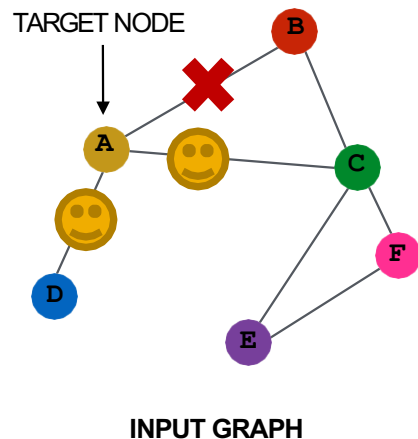
- For example, we can randomly choose 2 neighbors to pass messages in a given layer
  - Only nodes *B* and *D* will pass messages to *A*





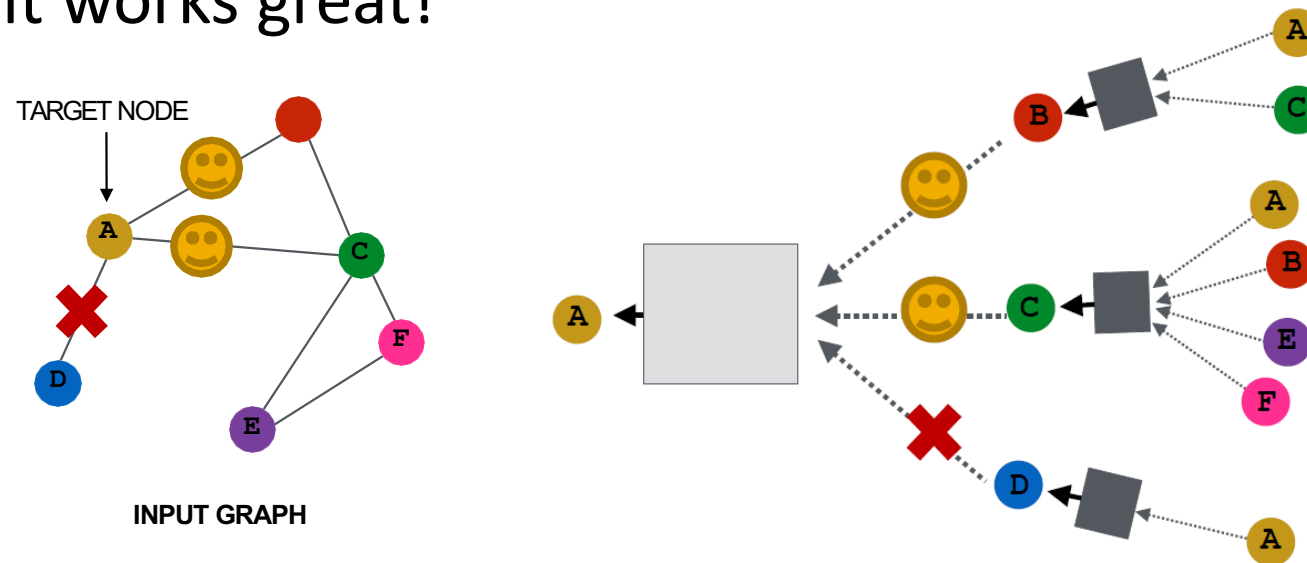
# GraphSAGE: Neighborhood Sampling

- In the next layer when we compute the embeddings, we can sample different neighbors
  - Only nodes *C* and *D* will pass messages to *A*



# GraphSAGE: Neighborhood Sampling

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



# GraphSAGE: $\ell_2$ Normalization

- $\ell_2$  Normalization:

- **Optional:** Apply  $\ell_2$  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  $\ell_2$  normalization, the embedding vectors have different scales ( $\ell_2$ -norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After  $\ell_2$  normalization, all vectors will have the same  $\ell_2$ -norm

# Graph Attention Networks (GAT)

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

## ■ 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  $v$

# Graph Attention Networks (GAT)

## ■ Graph Attention Networks

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

## Not all node's neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention**  $\alpha_{vu}$  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 (GAT)

Can we do better than simple neighborhood aggregation?

Can we let weighting factors  $\alpha_{vu}$  to be learned?

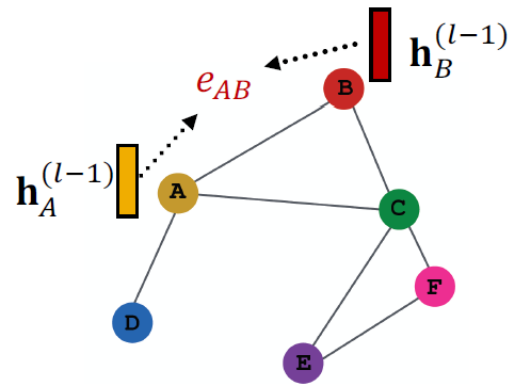
- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding  $\mathbf{h}_v^{(l)}$  of each node in the graph following an **attention strategy**:
  - Nodes attend over their neighborhoods' message
  - Implicitly specifying different weights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

# Attention Mechanism (1)

- Let  $\alpha_{vu}$  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**  $\alpha_{vu}$

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

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

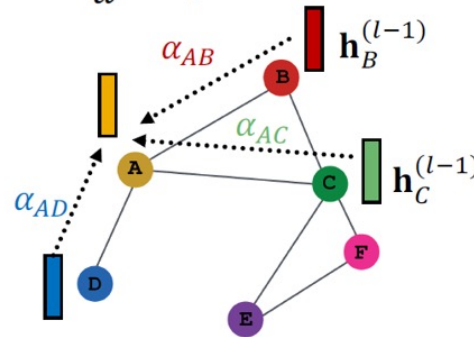
- **Weighted sum** based on the **final attention weight**

$\alpha_{vu}$

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

**Weighted sum using  $\alpha_{AB}$ ,  $\alpha_{AC}$ ,  $\alpha_{AD}$ :**

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



- Parameters of  $\alpha$  are trained jointly:

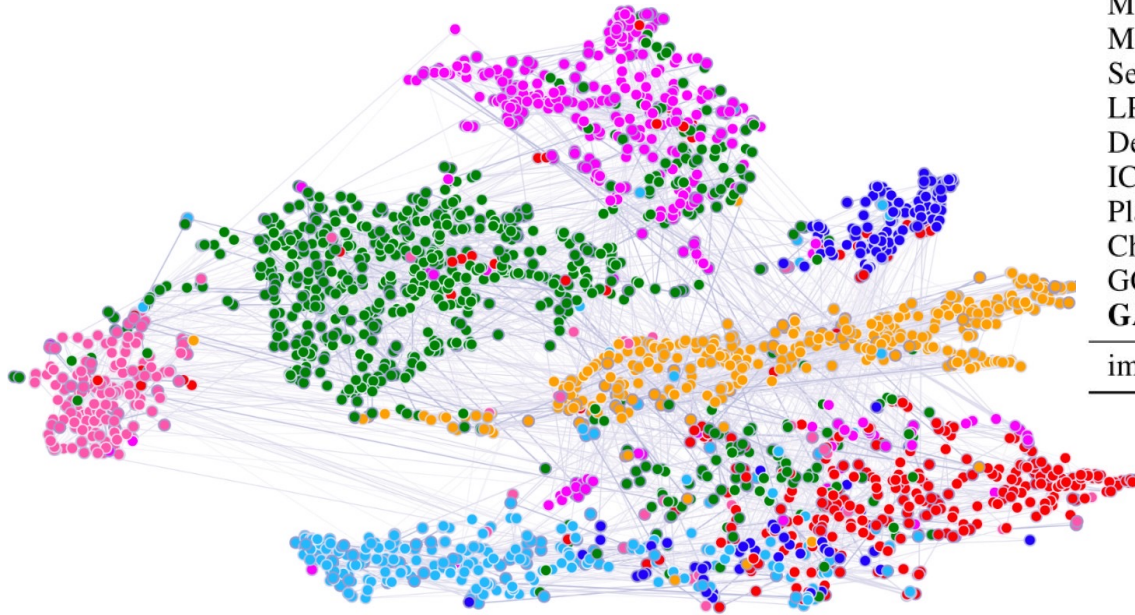
- Learn the parameters together with weight matrices (i.e., other parameter of the neural net  $\mathbf{W}^{(l)}$ ) in an end-to-end fashion



# Benefits of Attention Mechanism

- **Key benefit:** Allows for (implicitly) specifying **different importance values ( $\alpha_{vu}$ ) 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: 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%
<b>GAT</b>	<b>83.3%</b>
improvement w.r.t GCN	1.8%

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

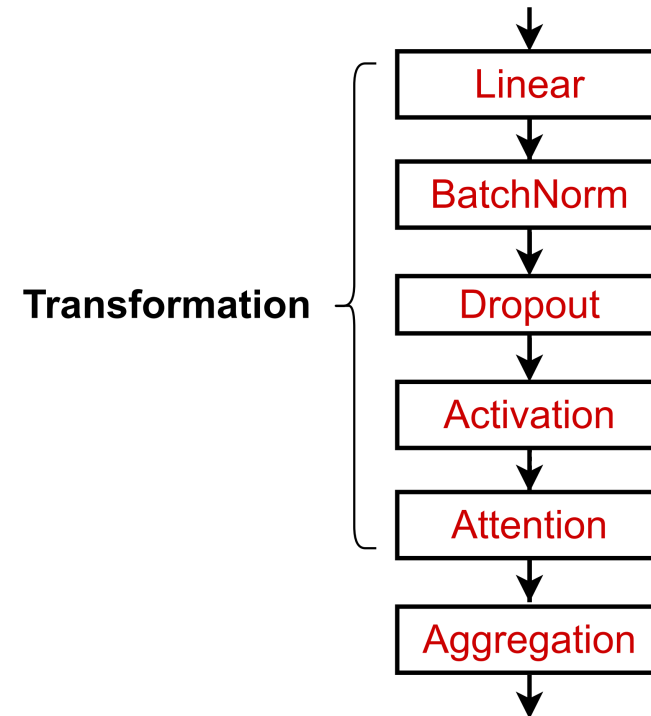
In many cases, attention leads to performance gains

- **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

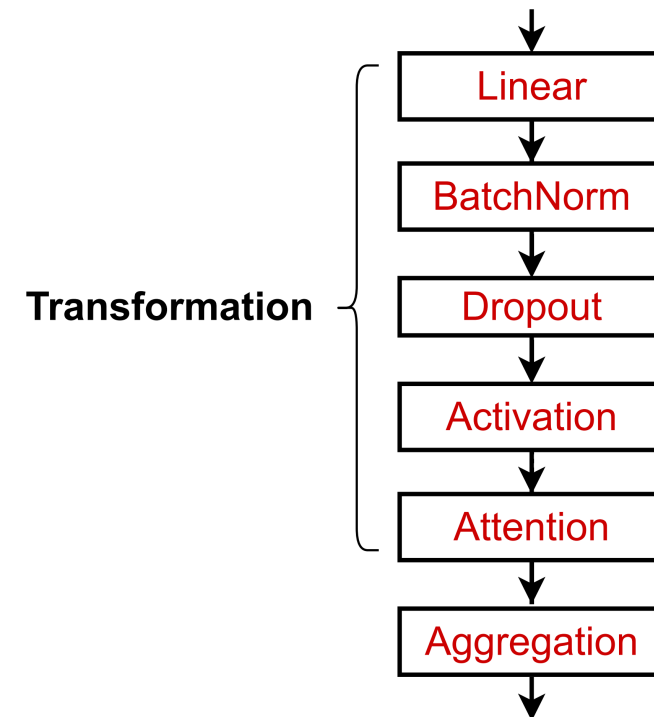


# GNN Layer in Practice

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

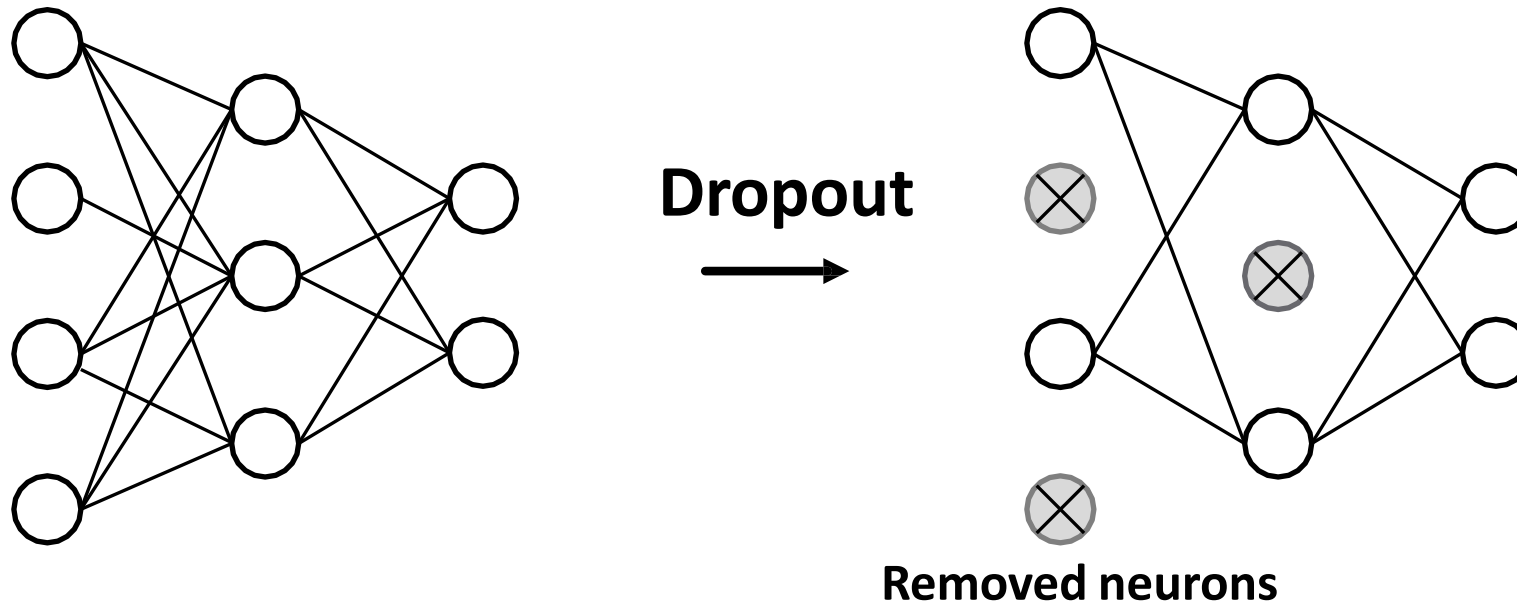
- **Batch Normalization:**
  - Stabilize neural network training
- **Dropout:**
  - Prevent overfitting
- **Attention/Gating:**
  - Control the importance of a message
- **More:**
  - Any other useful deep learning modules

A suggested GNN Layer



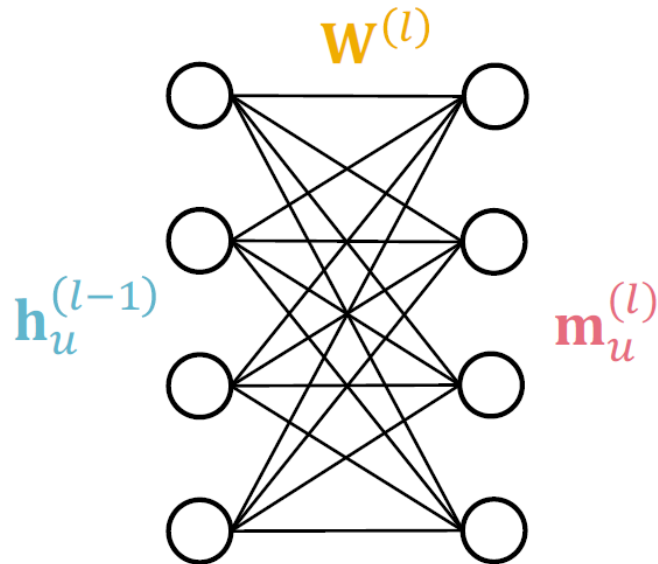
# Dropout

- **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



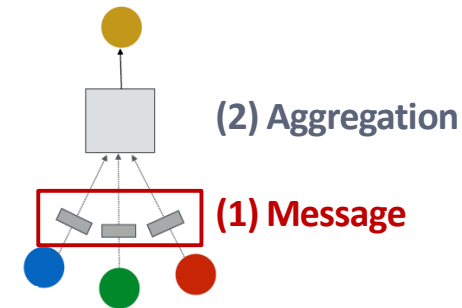
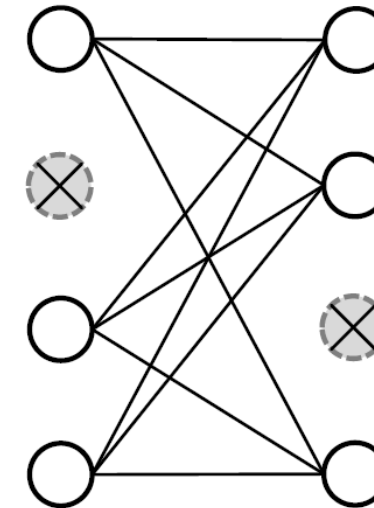
# Dropout for GNNs

- In GNN, Dropout is applied to **the linear layer in the message function**
- A simple message function with linear layer:



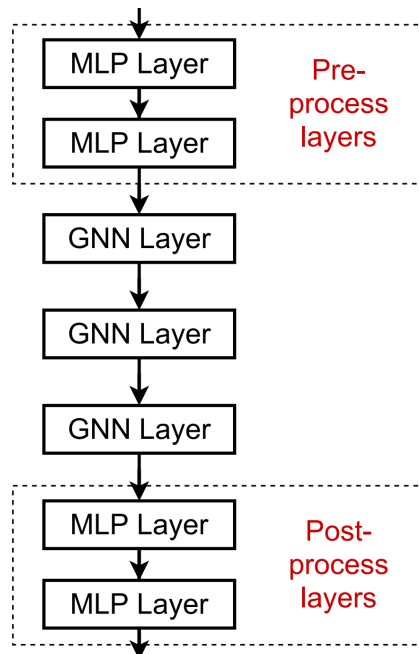
Visualization of a linear layer

Dropout  
→



# Expressive Power for Shallow GNNS

- **How to make a shallow GNN more expressive?**
- **Solution:** 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

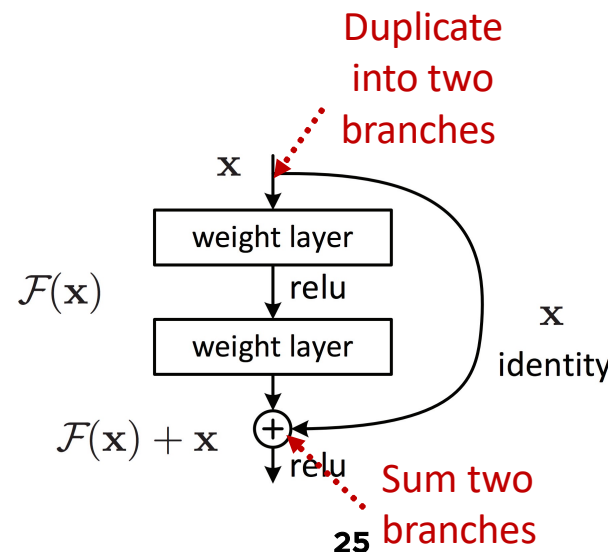
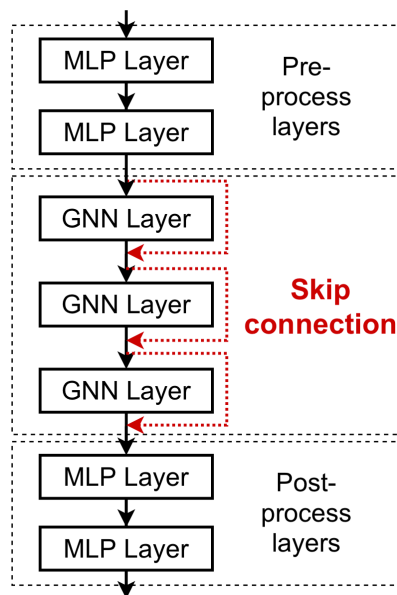
**Post-processing layers:** Important when reasoning / transformation over node embeddings are needed

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: 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:

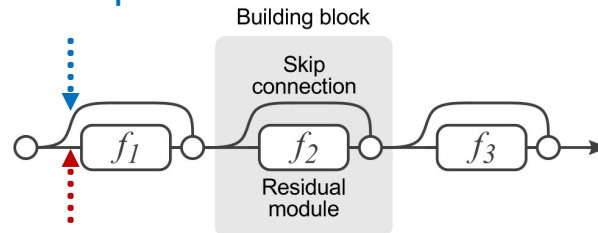
$$\mathcal{F}(x) + x$$



# 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**

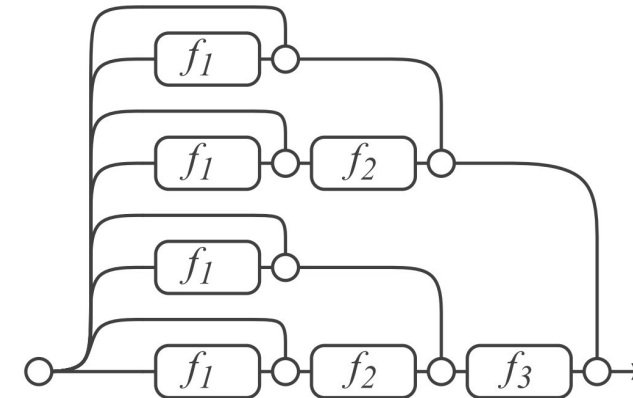
Path 2: skip this module



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)

# GCN with Skip Connections

- A standard GCN layer

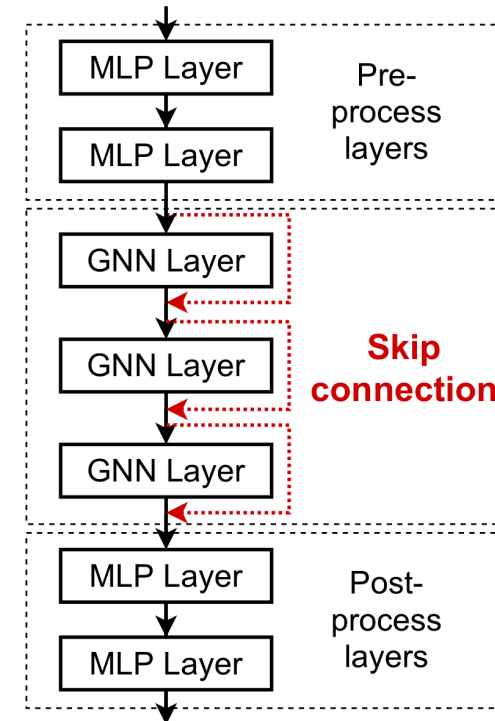
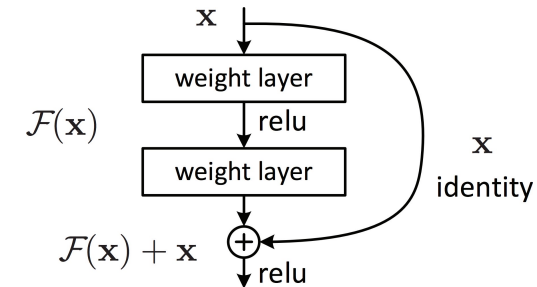
$$\mathbf{h}_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

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

- A GCN layer with skip connection

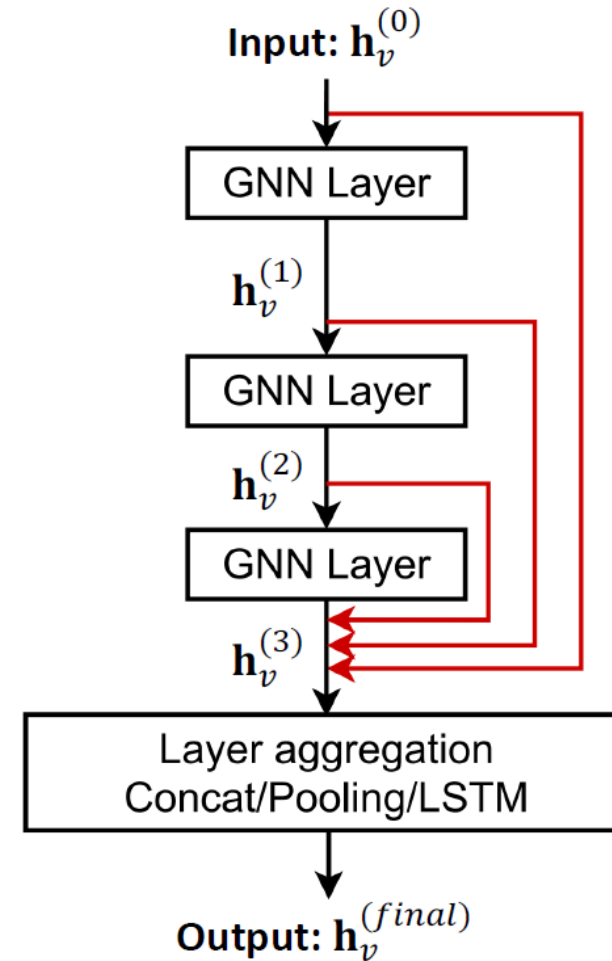
$$\mathbf{h}_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)} \right)$$

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



# Other 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 Augmentation (Optional)

# Why Augment Graphs

## Problems in training a GNN

- **Features:**

- The input graph **lacks features**

- **Graph structure:**

- 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

# Graph Augmentation Approaches

- **Graph Feature augmentation**

- The input graph **lacks features** → **feature augmentation**

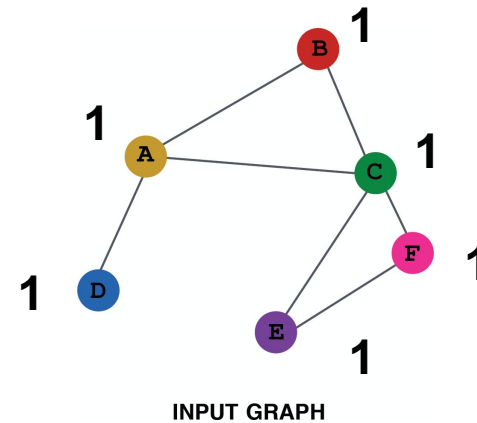
- **Graph Structure augmentation**

- 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**

# Features Augmentation on Graphs

When might 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**



# Features Augmentation on Graphs

When might we need feature augmentation?

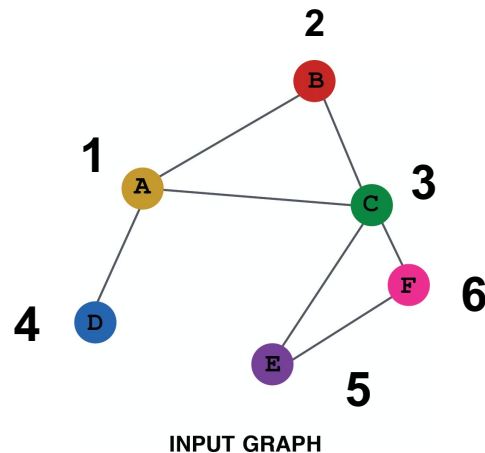
- **(1) Input graph does not have node features**

- 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

One-hot vector for node with ID=5

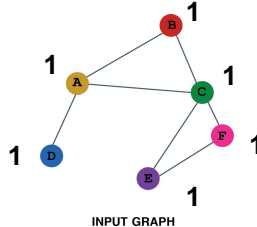
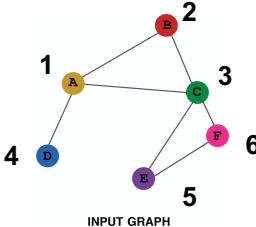
↓ ID = 5

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

⏟  
Total number of IDs = 6



# Features Augmentation on Graphs

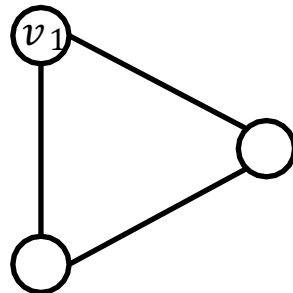
	<b>Constant node feature</b> 	<b>One-hot node feature</b> 
<b>Expressive power</b>	<b>Medium.</b> All the nodes are identical, but <b>GNN can still learn from the graph structure</b>	<b>High.</b> Each node has a unique ID, so <b>node-specific information can be stored</b>
<b>Inductive learning (Generalize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

# Features Augmentation on Graphs

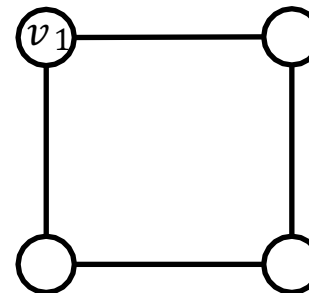
When might we need feature augmentation?

- **(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



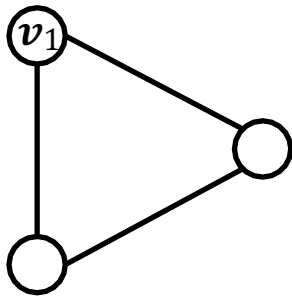
$v_1$  resides in a cycle with length 4



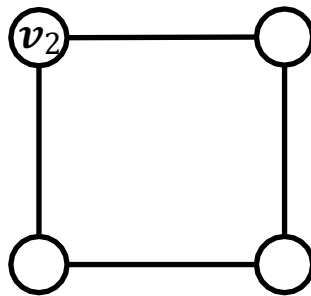
# Features Augmentation on Graphs

- $v_1$  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

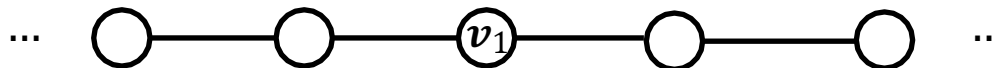
$v_1$  resides in a cycle with length 3



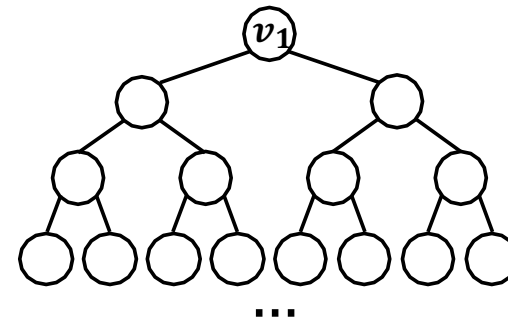
$v_1$  resides in a cycle with length 4



$v_1$  resides in a cycle with infinite length



The computational graphs for node  $v_1$  are always the same



# Features Augmentation on Graphs

When might we need feature augmentation?

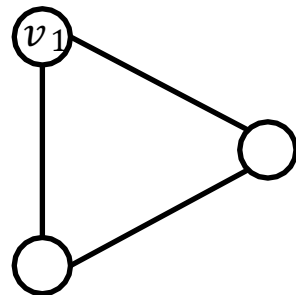
- **(2) Certain structures are hard to learn by GNN**
- **Solution:** We can use **cycle count** as augmented node features

We start  
from cycle  
with length 0

Augmented node feature for  $v_1$   
 $[0, 0, 0, 1, 0, 0]$

↑

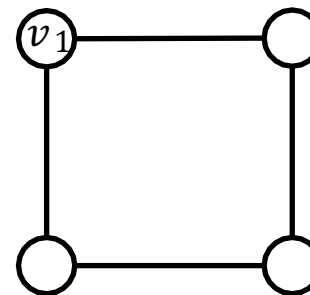
$v_1$  resides in a cycle with length 3



Augmented node feature for  $v_1$   
 $[0, 0, 0, 0, 1, 0]$

↑

$v_1$  resides in a cycle with length 4



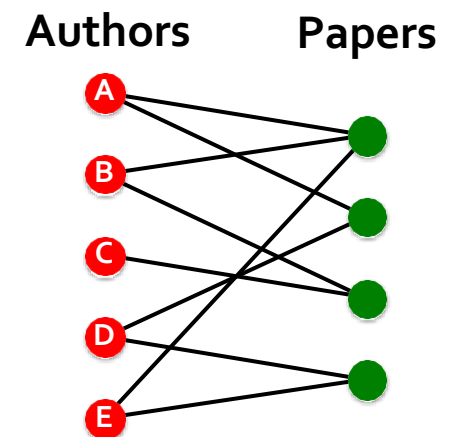
# Features Augmentation on Graphs

When might we need feature augmentation?

- **(2) Certain structures are hard to learn by GNN**
- Other commonly used augmented features:
  - Node degree
  - Clustering coefficient
  - Centrality
  - ...

# 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

The virtual node

- **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 two**
      - Node A – Virtual node – Node B
  - **Benefits:** Greatly **improves message passing in sparse graphs**

