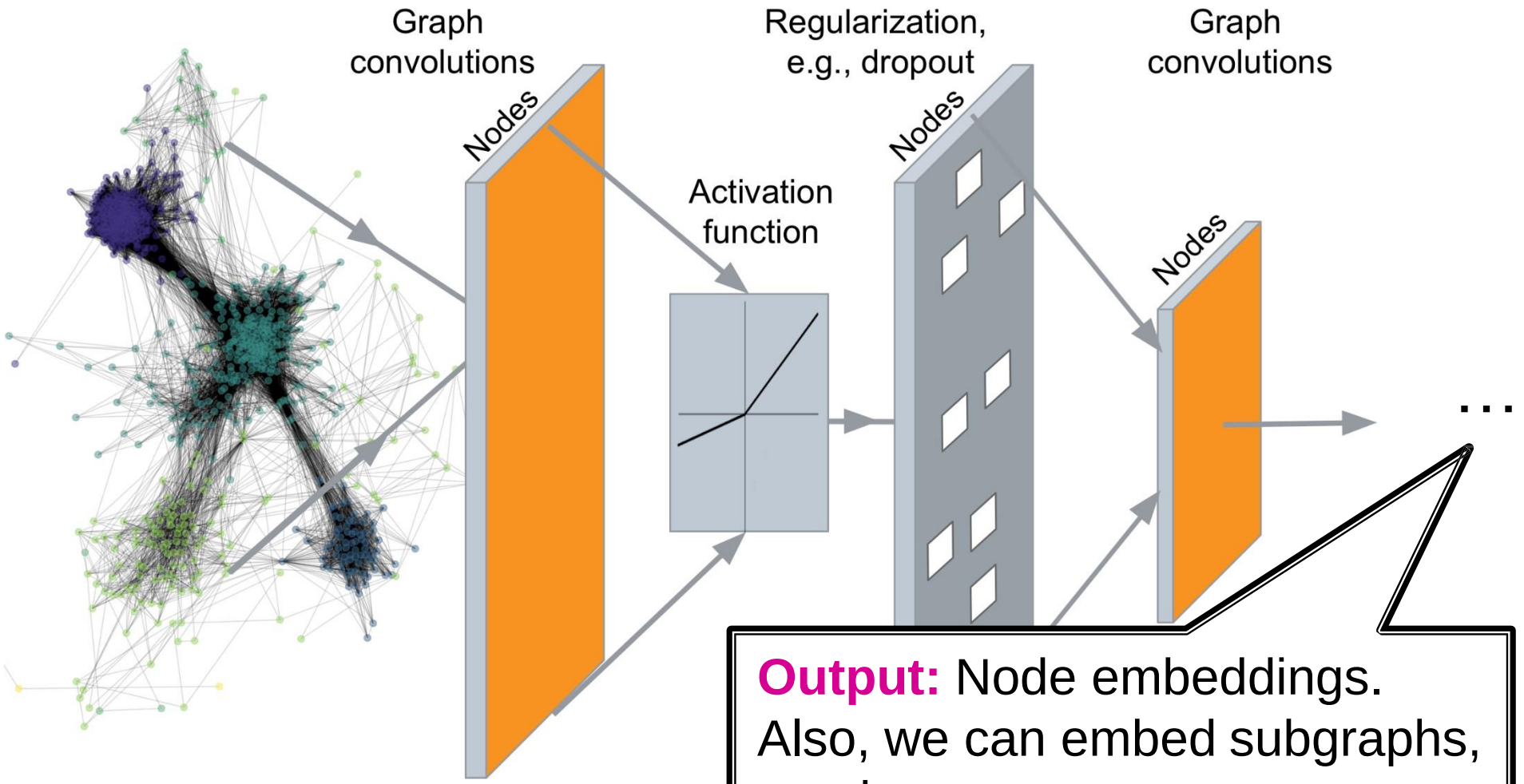
COMP4222 Machine Learning with Structured Data

Graph Neural Networks 3

Instructor: Yangqiu Song

Slides credits: Jure Laskovec

Recap: Deep Graph Encoders

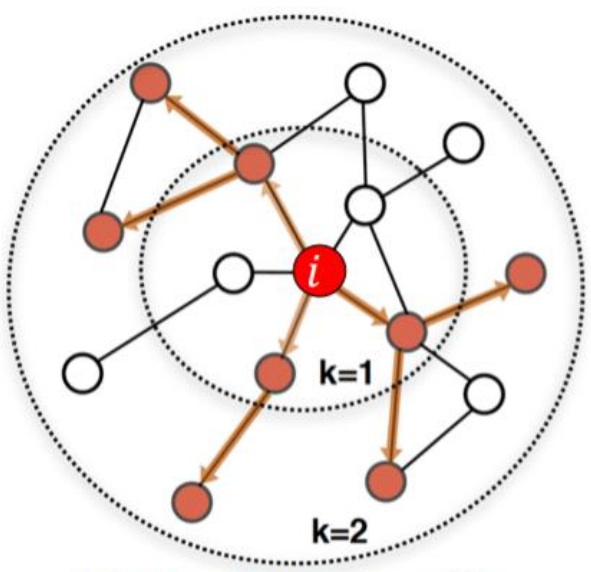


Output: Node embeddings.
Also, we can embed subgraphs, graphs

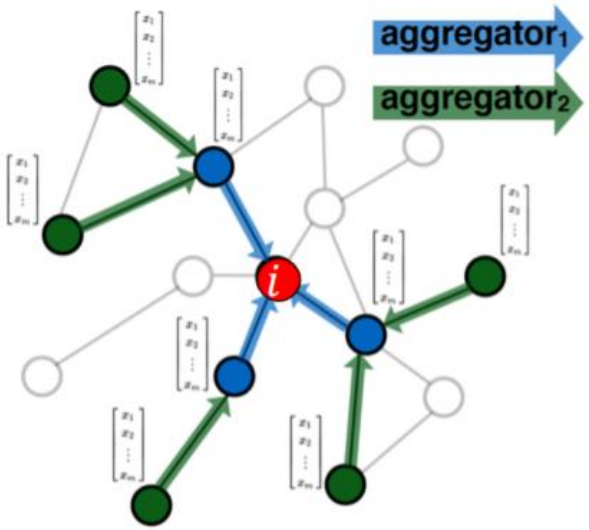
[Kipf and Welling, ICLR 2017]

Recap: Graph Convolutional Networks

- Idea: Node's neighborhood defines a computation graph



Determine node computation graph

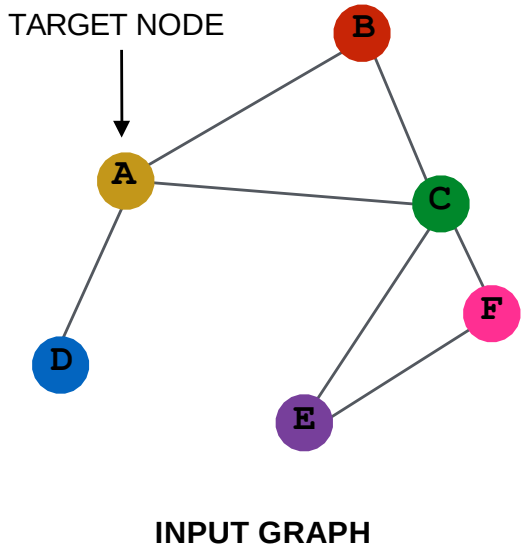


Propagate and transform information

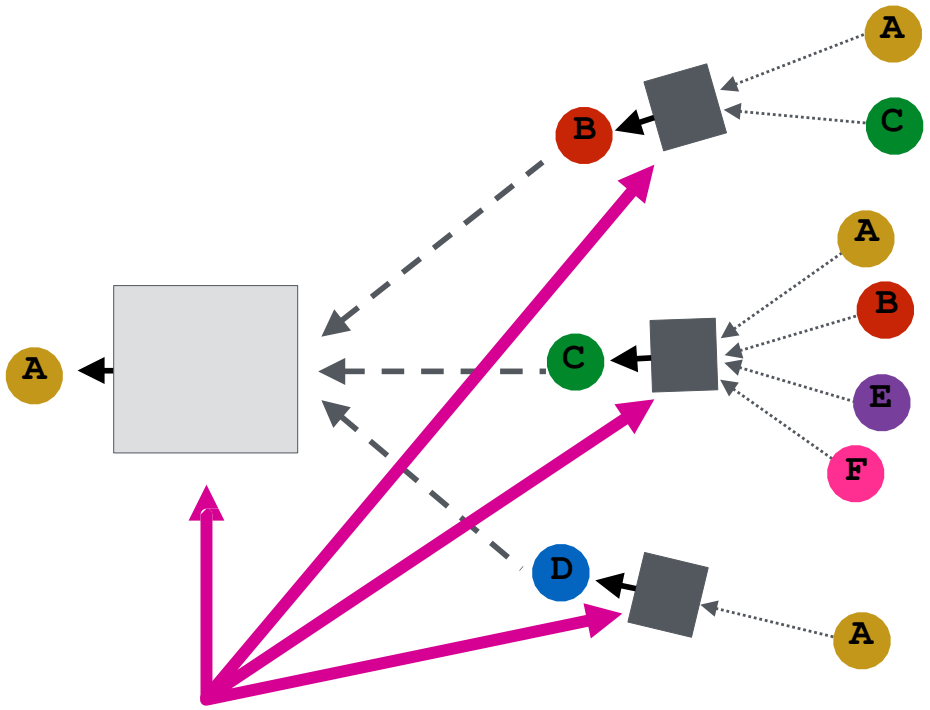
Learn how to propagate information across the graph to compute node features

Recap: Aggregate from Neighbors

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



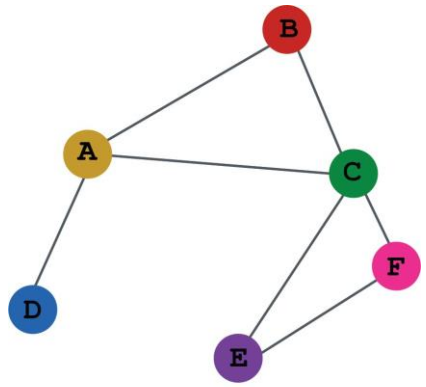
INPUT GRAPH



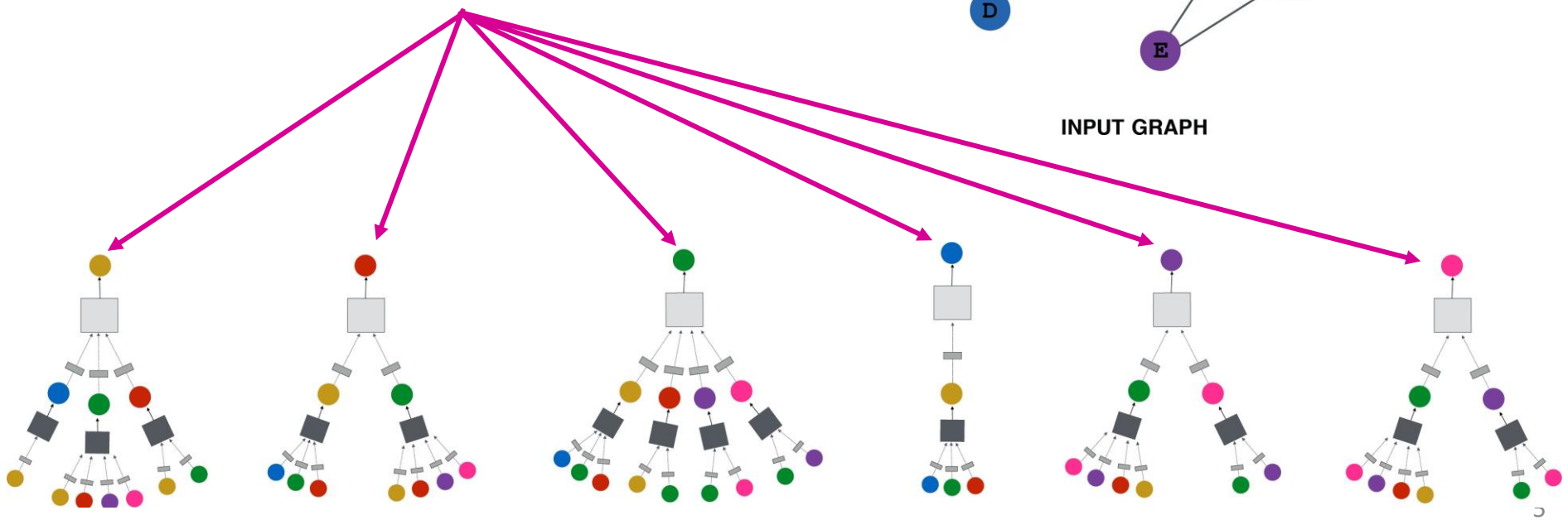
Neural networks

Recap: Aggregate Neighbors

- **Intuition:** Network neighborhood defines a computation graph
  - Every node defines a computation graph based on its neighborhood!



INPUT GRAPH



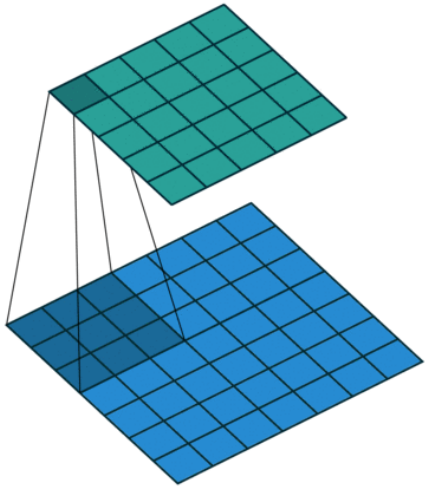
Why GNNs Generalize other NNs?

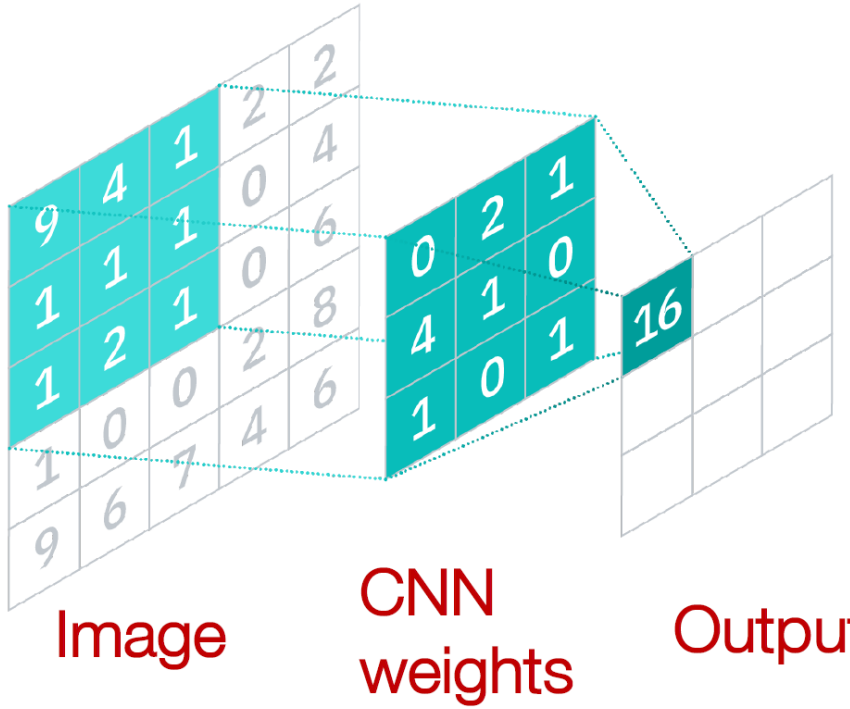
- Defined notions of permutation invariance and equivariance.

- How does GNNs compare to prominent architectures such as Convolutional Neural Nets, and Transformers?

CNN

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



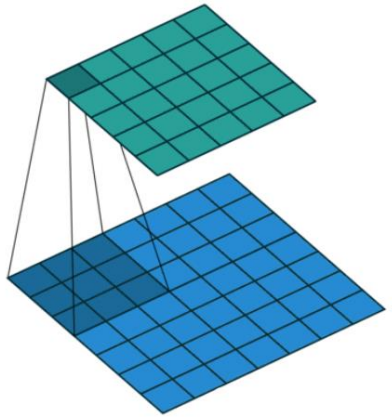


CNN formulation: $$\mathrm{h}_v^{(l+1)} = \sigma\left(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}\right), \quad \forall l \in \{0, \ldots, L-1\}$$

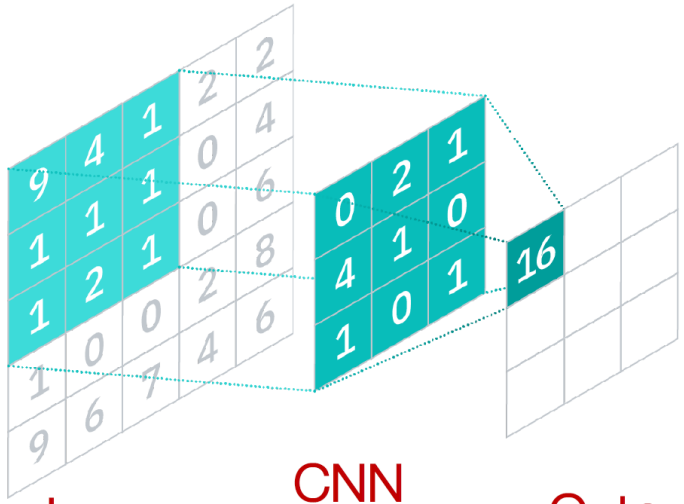
$N(v)$ represents the 8 neighbor pixels of v.

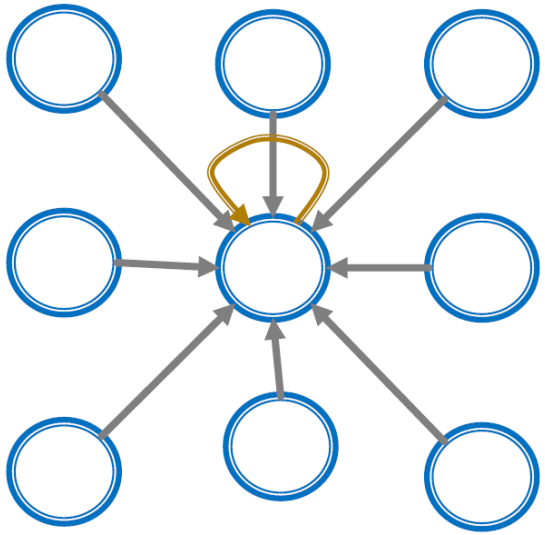
GNN vs. CNN

We can learn different weights for different "neighbor" on the image



Image





Graph

CNN formulation: $$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

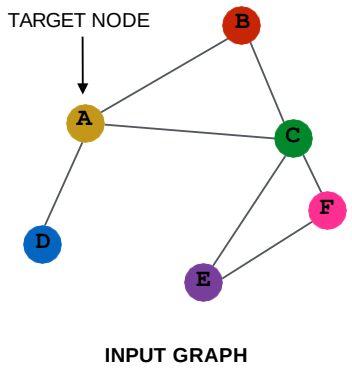
GNN formulation: $$h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

Rewrite: $$h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v)} W_l^u h_u^{(l)} + B_l h_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

GNN vs. CNN

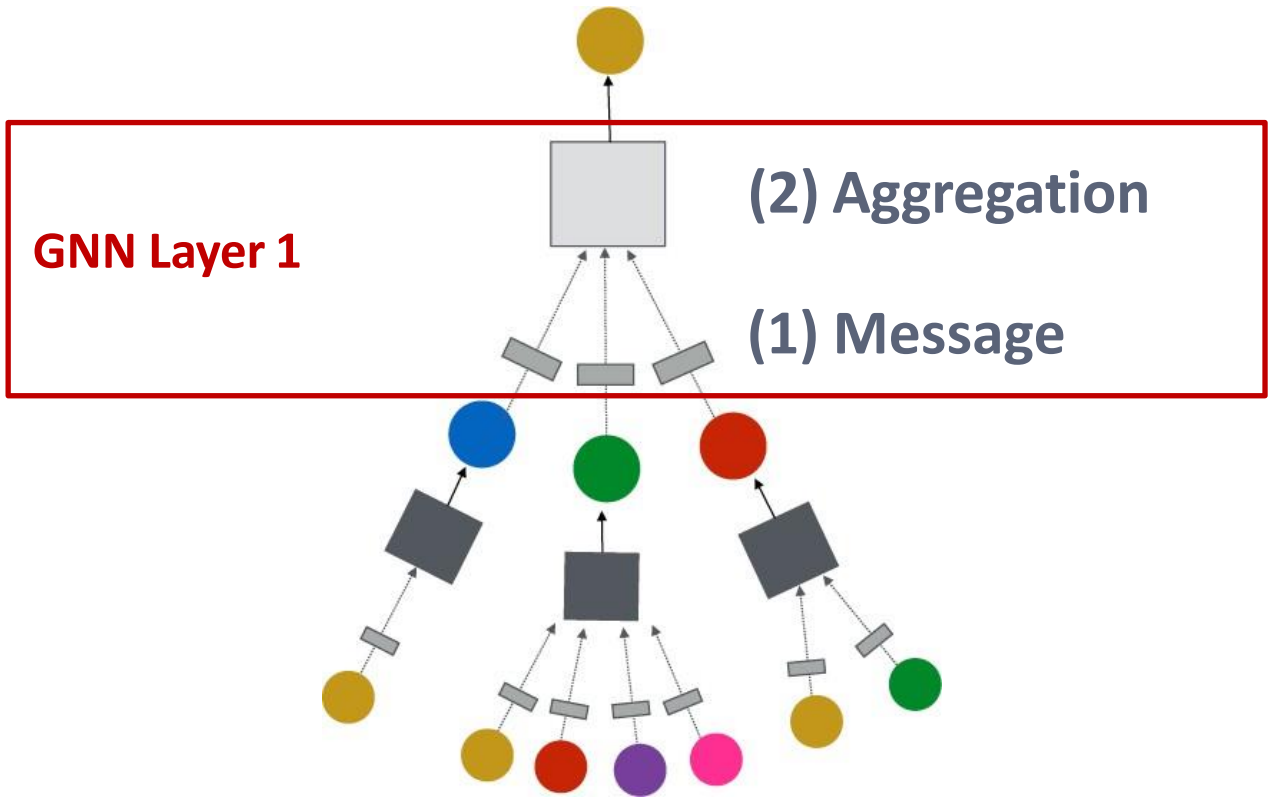
- CNN can be seen as a special GNN with fixed neighbor size and ordering:
  - The size of the filter is pre-defined for a CNN
  - The advantage of GNN is it processes arbitrary graphs with different degrees for each node
- CNN is not permutation equivariant
  - Switching the order of pixels will leads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

A General GNN Framework



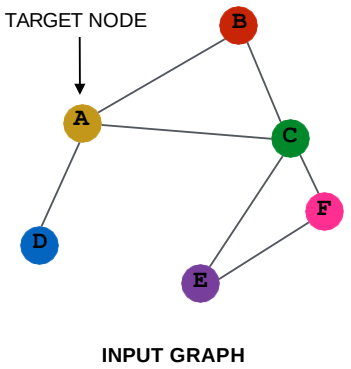
GNN Layer = Message + Aggregation

- **Different instantiations under this perspective**
- **GCN, GraphSAGE, GAT, …**



J.You, R.Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

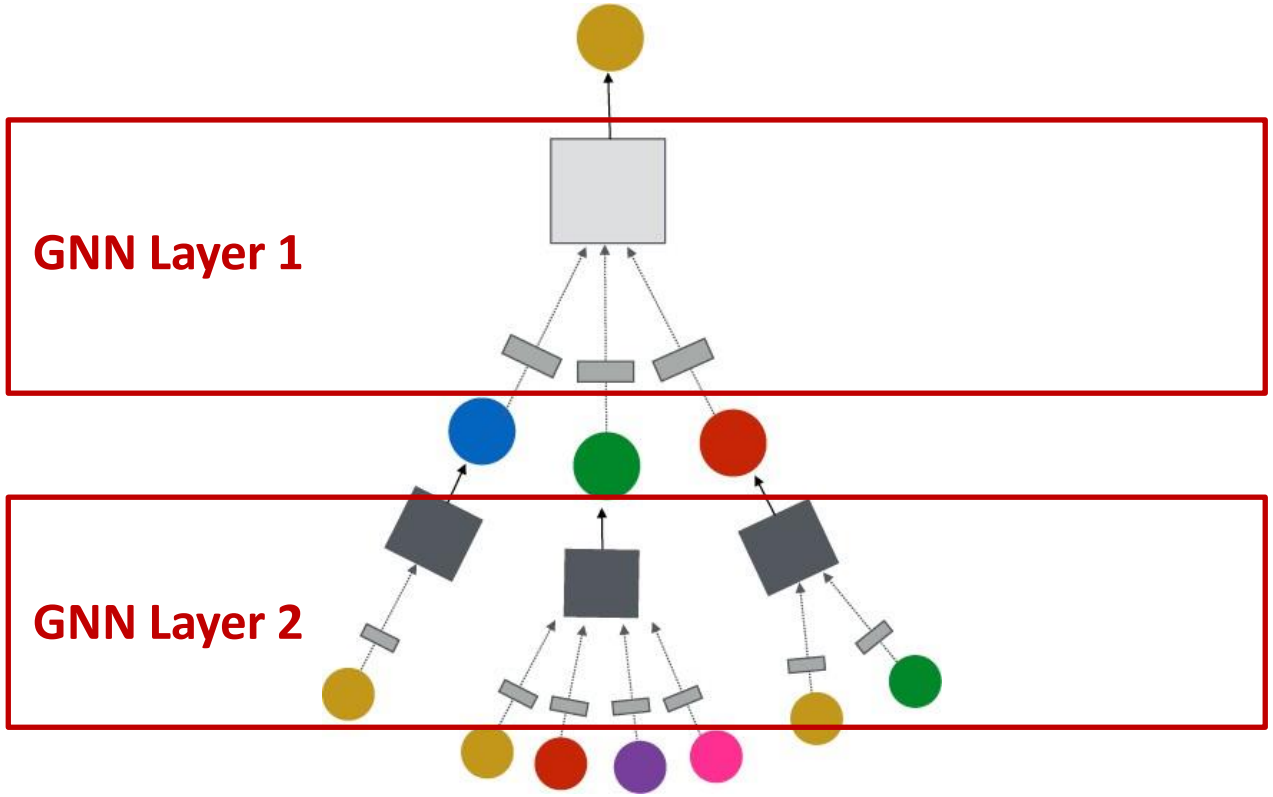
A General GNN Framework



Connect GNN layers into a GNN

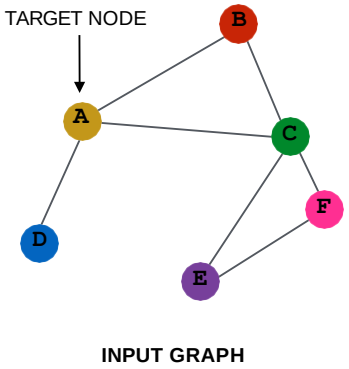
- **Stack layers sequentially**
- **Ways of adding skip connections**

(3) Layer connectivity



J.You, R.Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

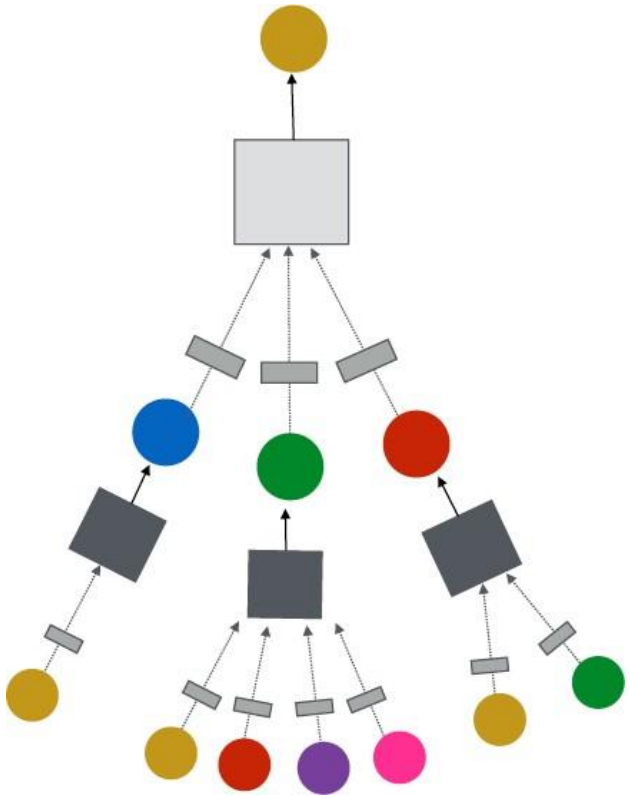
A General GNN Framework



INPUT GRAPH

Idea: Raw input graph ≠ computational graph

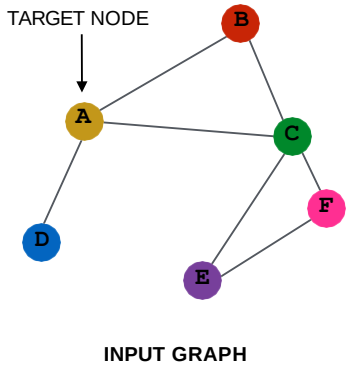
- **Graph feature augmentation**
- **Graph structure augmentation**



(4) Graph augmentation

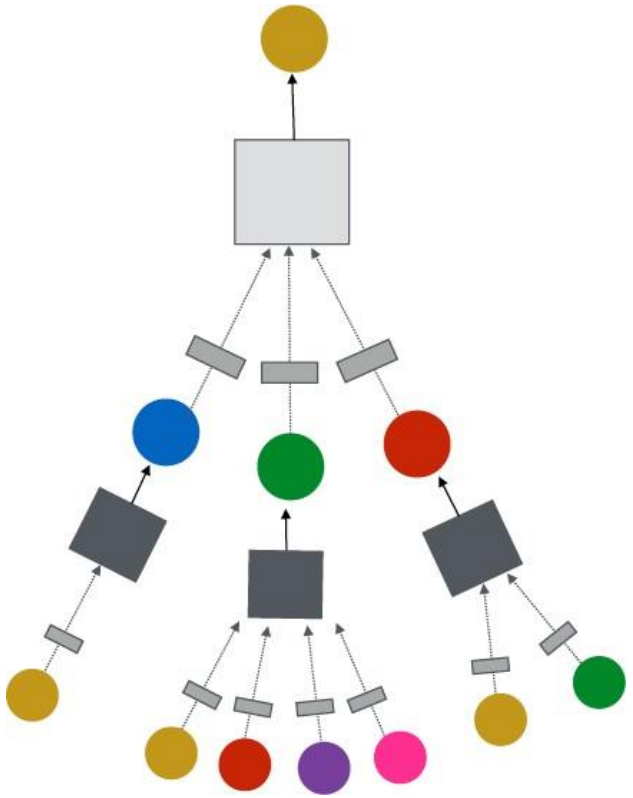
J.You, R.Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

A General GNN Framework



INPUT GRAPH

(5) Learning objective

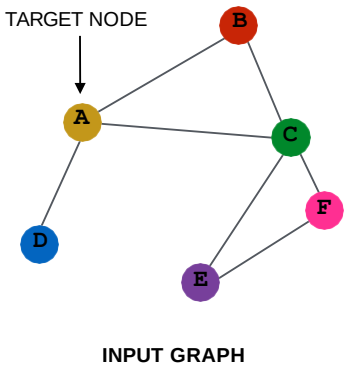


How do we train a GNN

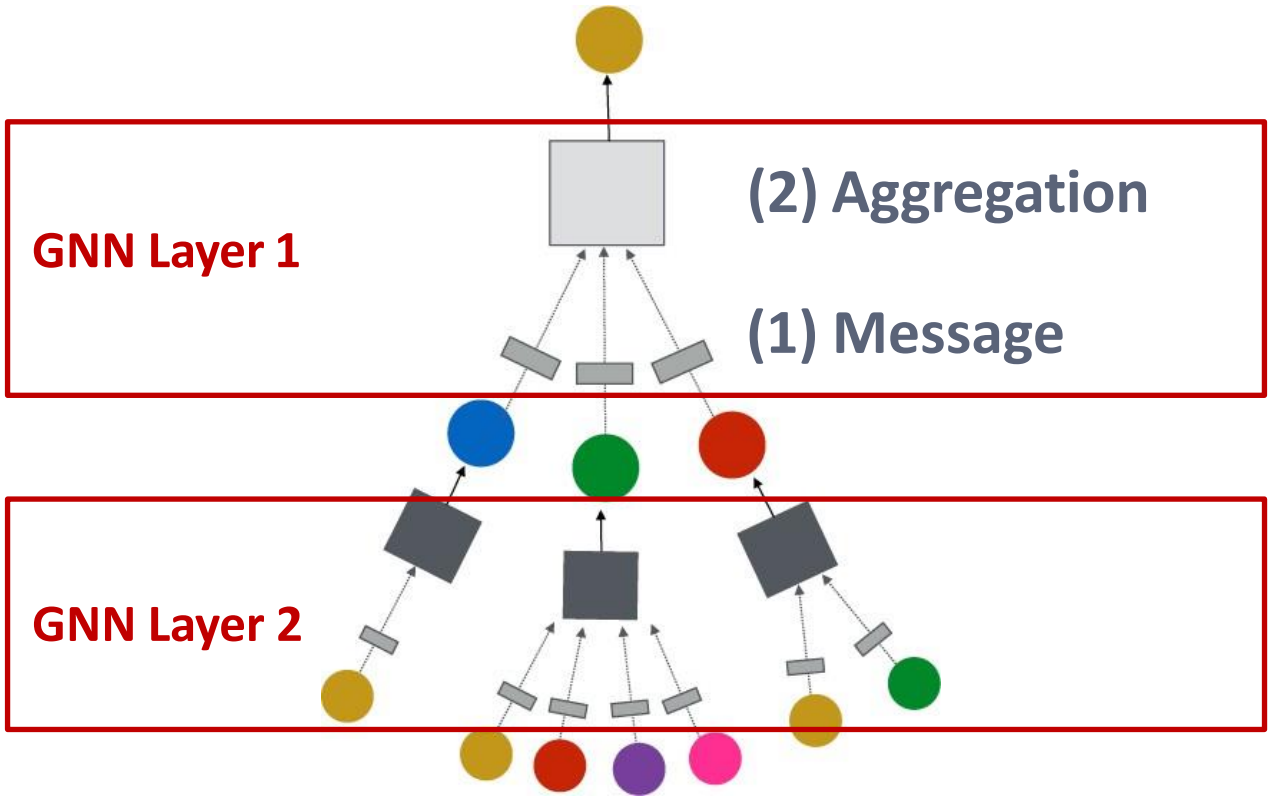
- **Supervised/Unsupervised objectives**
- **Node/Edge/Graph level objectives**

J.You, R.Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

A General GNN Framework



(5) Learning objective



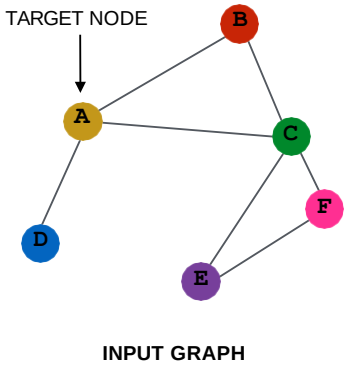
(3) Layer connectivity

(4) Graph augmentation

J.You, R.Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

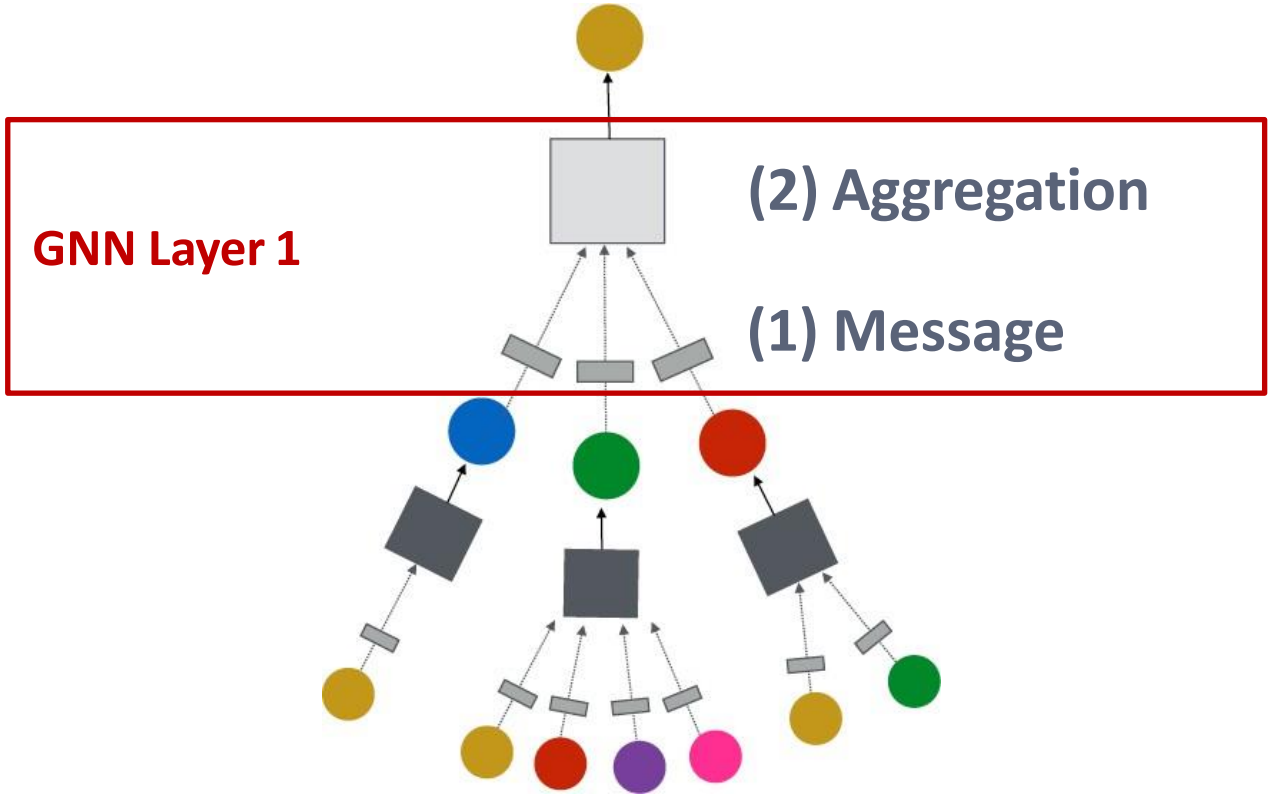
A Single GNN Layer

A GNN Layer



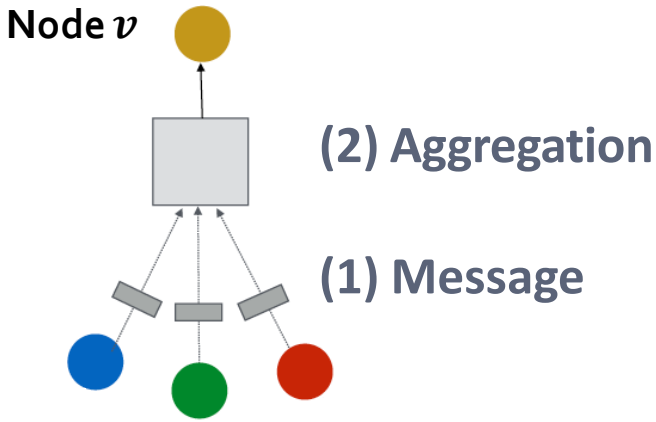
GNN Layer = Message + Aggregation

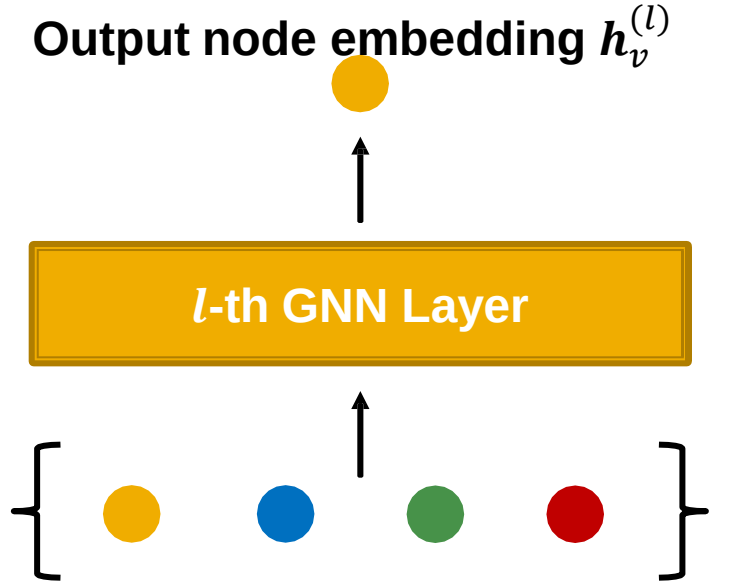
- **Different instantiations under this perspective**
- **GCN, GraphSAGE, GAT, …**



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**

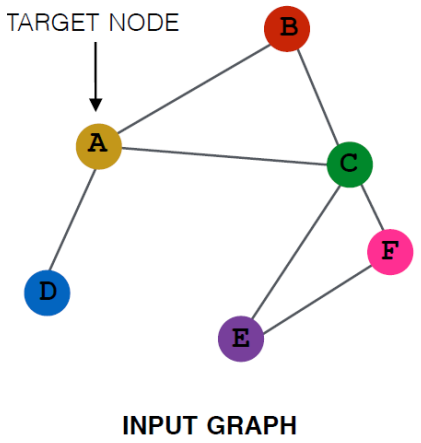


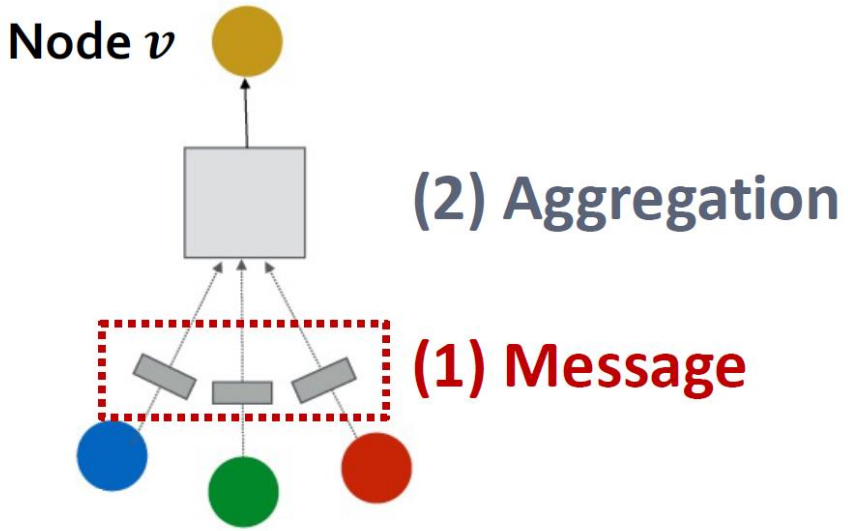


Input node embedding $h_{u \in N(v)}^{(l-1)}$, $h_v^{(l-1)}$
(from node itself + neighboring nodes)

Message Computation

- **(1) Message computation**
  - **Message function**: $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right)$
    - **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 $\mathbf{W}^{(l)}$





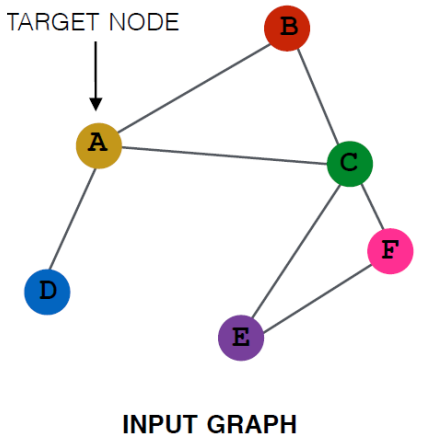
Message Aggregation

- **(2) Aggregation**
  - **Intuition:** Each node will aggregate the messages from node v's neighbors

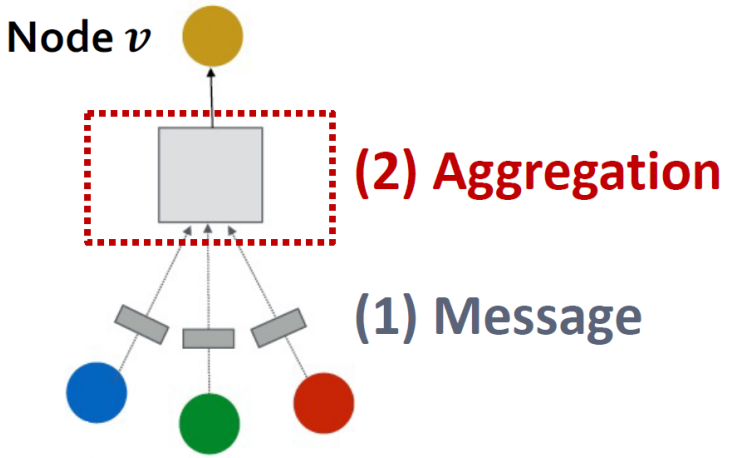
$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)$$

  - **Example:** $\mathrm{Sum}(\cdot)$, $\mathrm{Mean}(\cdot)$ or $\mathrm{Max}(\cdot)$ aggregator

$$\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$$



INPUT GRAPH



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**

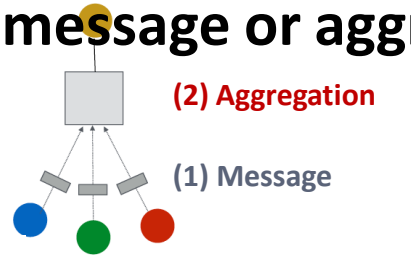
Then aggregate from node itself

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

First aggregate from neighbors

A Single GNN Layer

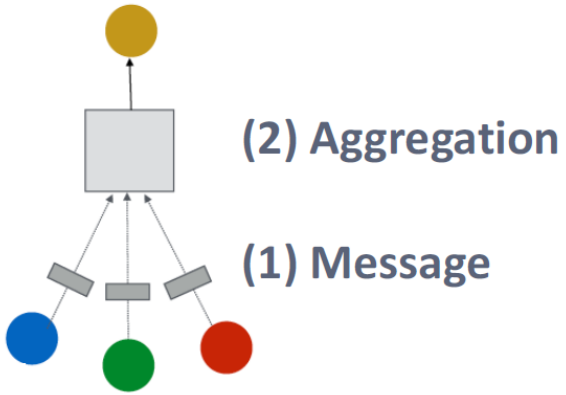
- **Putting things together:**
  - **(1) Message**: each node computes a mess $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$
  - **(2) Aggregation**: aggregate messages from neighb $\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}, \mathbf{m}_v^{(l)}\right)$
  - **Nonlinearity (activation):** Adds expressiveness
    - Often written as $\sigma(\cdot)$: $\mathrm{ReLU}(\cdot)$, $\mathrm{Sigmoid}(\cdot)$, ...
    - 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

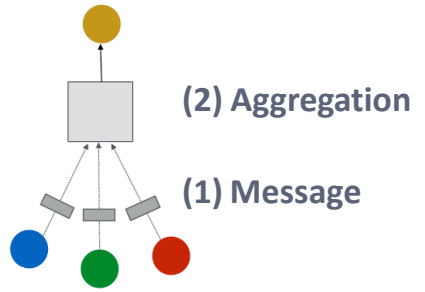
$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

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

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

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

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

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

Aggregation — Message computation

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

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

Aggregation — Message 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

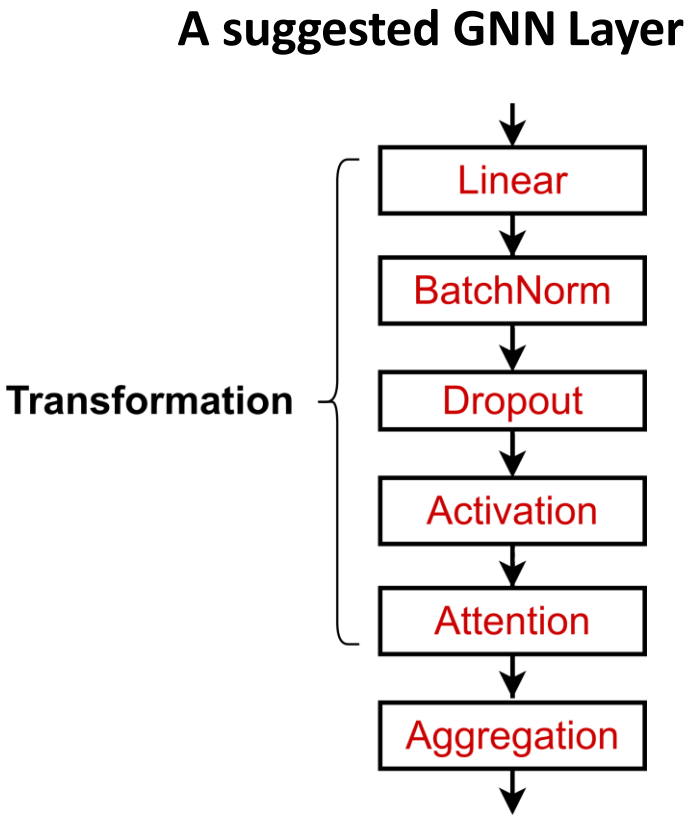
- **L2 normalization:**
  - **Optional:** Apply L2 normalization to $\mathbf{h}_v^{(l)}$ at every layer

$$\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\left\|\mathbf{h}_v^{(l)}\right\|_2} \; \forall v \in V \text{ where } \|u\|_2 = \sqrt{\sum_i u_i^2} \; (\ell_2\text{-norm})$$

- Without L2 normalization, the embedding vectors have different scales (L2 -norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After L2 normalization, all vectors will have the same L2 norm

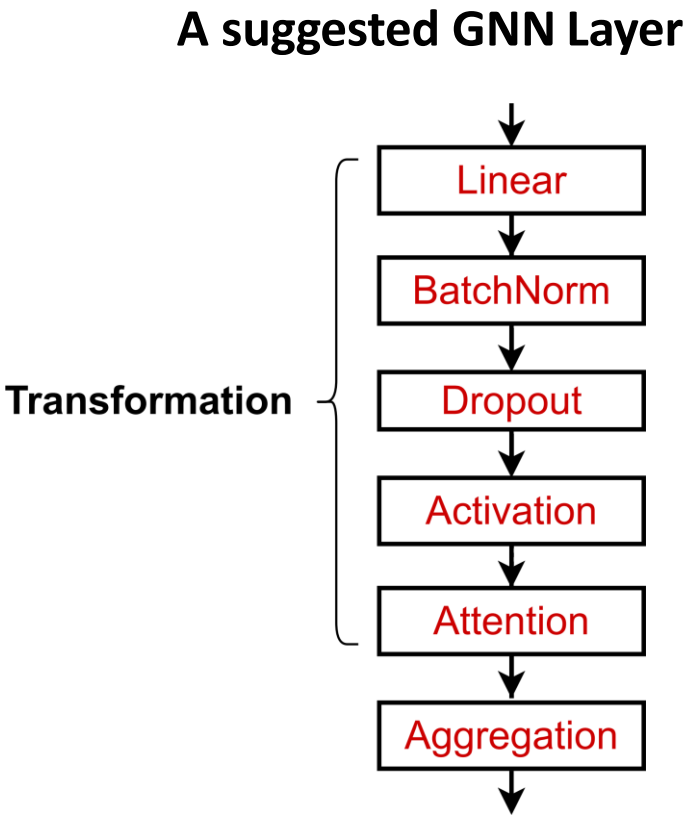
GNN Layer in Practice

- **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



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

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

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

$$\boldsymbol{\mu}_j = \frac{1}{N}\sum_{i=1}^{N} \mathbf{X}_{i,j}$$

$$\boldsymbol{\sigma}_j^2 = \frac{1}{N}\sum_{i=1}^{N} (\mathbf{X}_{i,j} - \boldsymbol{\mu}_j)^2$$

Step 2:
Normalize the feature using computed mean and variance

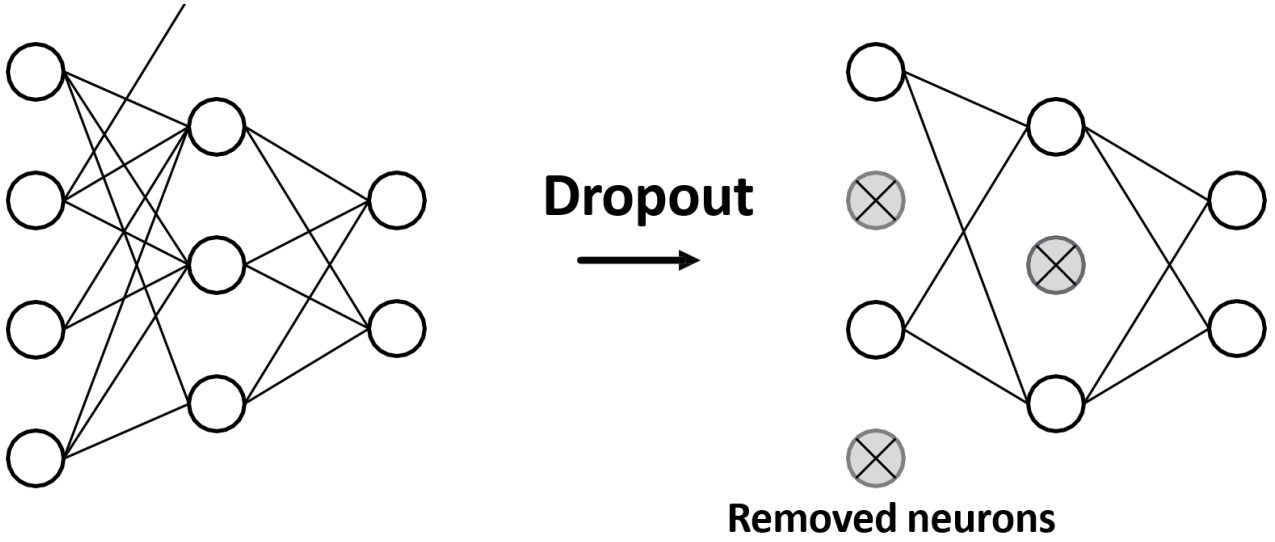
$$\widehat{\mathbf{X}}_{i,j} = \frac{\mathbf{X}_{i,j} - \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{\sigma}_j^2 + \epsilon}}$$

$$\mathbf{Y}_{i,j} = \boldsymbol{\gamma}_j \widehat{\mathbf{X}}_{i,j} + \boldsymbol{\beta}_j$$

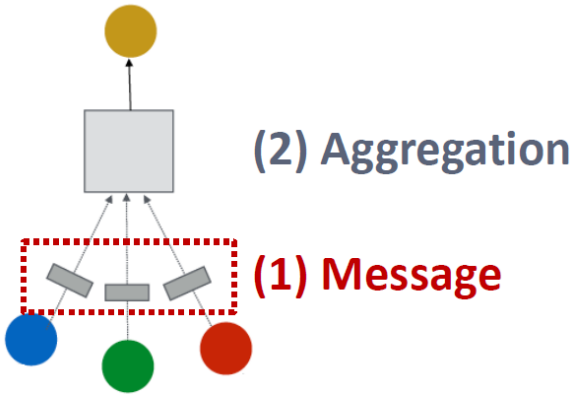
S. Loffe, C.Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Dropout

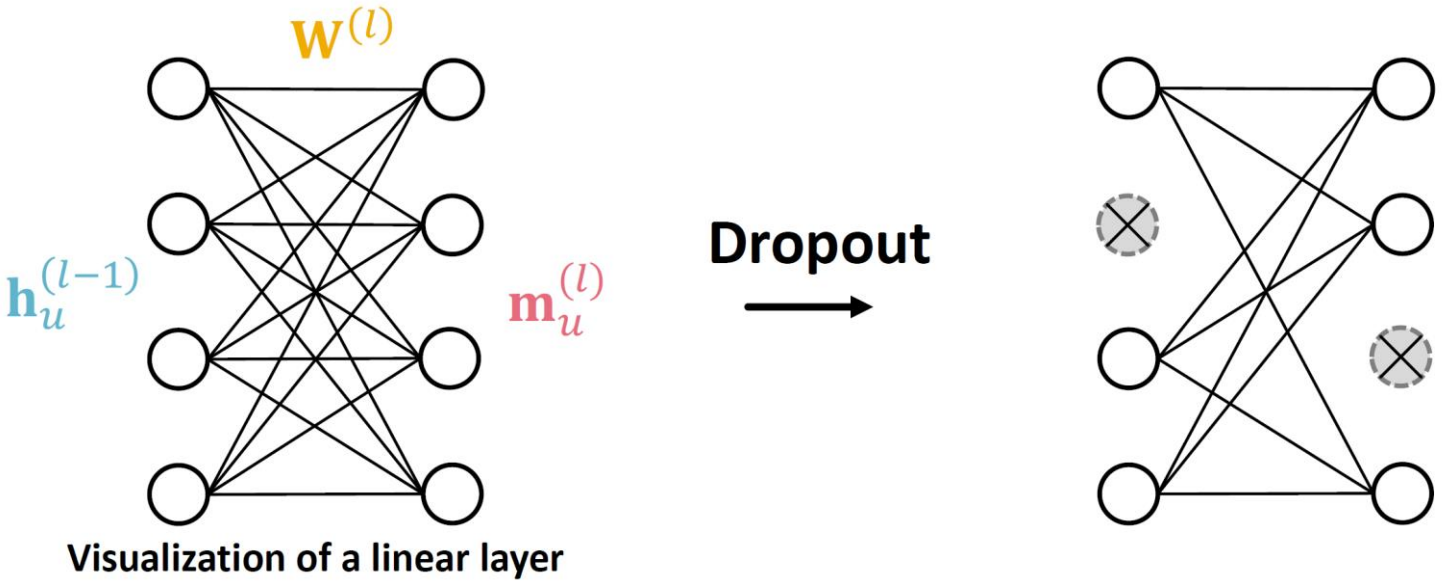
- **Goal**: Regularize a neural net to prevent overfitting.
- **Idea**:
  - **During training**: with some probability p, randomly set neurons to zero (turn off)
  - **During testing:** Use all the neurons for computation



Dropout for GNNs



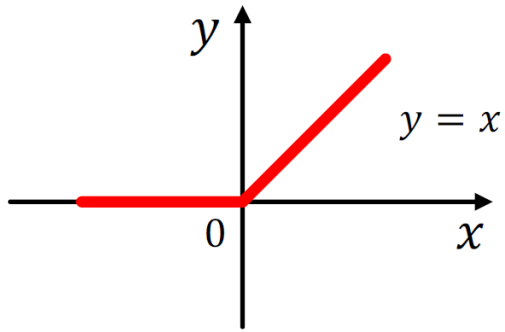
- In GNN, Dropout is applied to **the linear layer in the message function**
- **A simple message function with linear layer** $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$

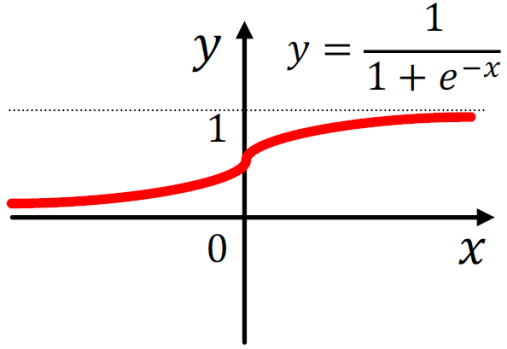


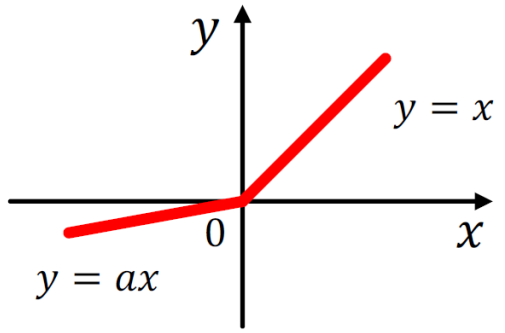
Visualization of a linear layer

Activation (Non-linearity)

- Apply activation to $\boldsymbol{i}$-th dimension of embedding $\mathbf{x}$
  - **Rectified linear unit (ReLU)** $\mathrm{ReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0)$
    - 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** $\mathrm{PReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0) + \alpha_i \min(\mathbf{x}_i, 0)$
    - α_i is a trainable parameter
    - So-called leaky ReLU
      - Small slope for negative values instead of a flat slope
    - Empirically performs better than ReLU







GNN Layer in Practice

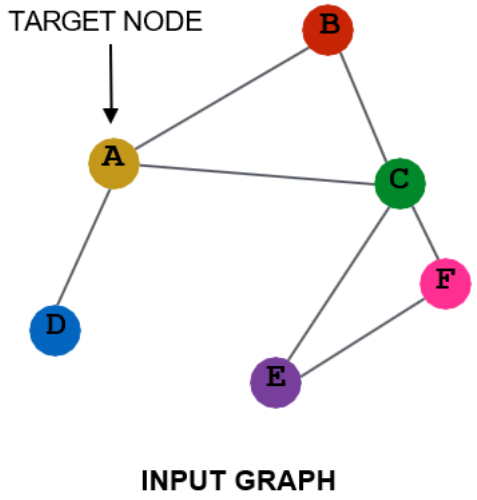
- **Summary:** Modern deep learning modules can be included into a GNN layer for better performance

- **Designing novel GNN layers is still an active research frontier!**

More than One Layers

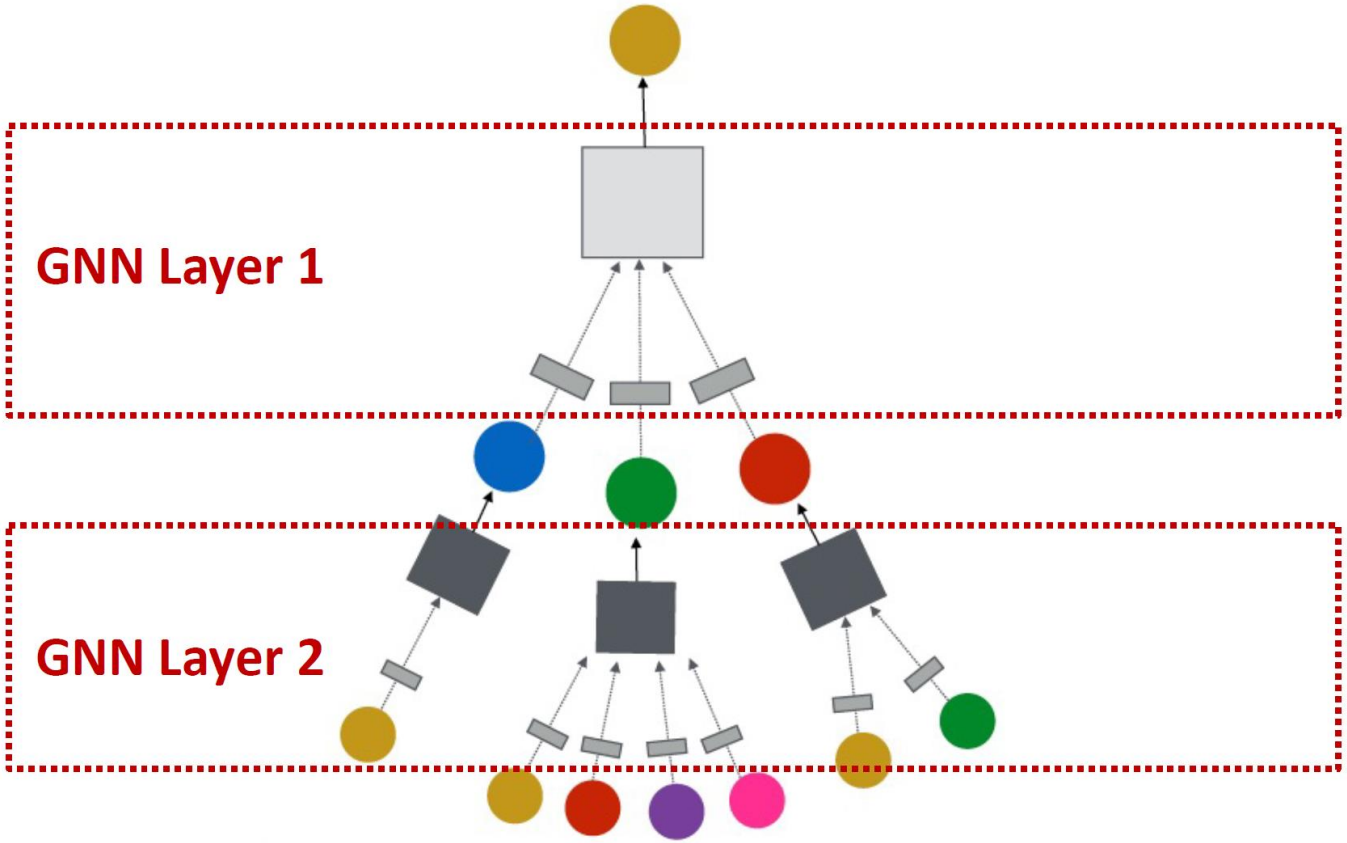
Stacking GNN Layers

- How to connect GNN layers into a GNN?
  - Stack layers sequentially
  - Ways of adding skip connections



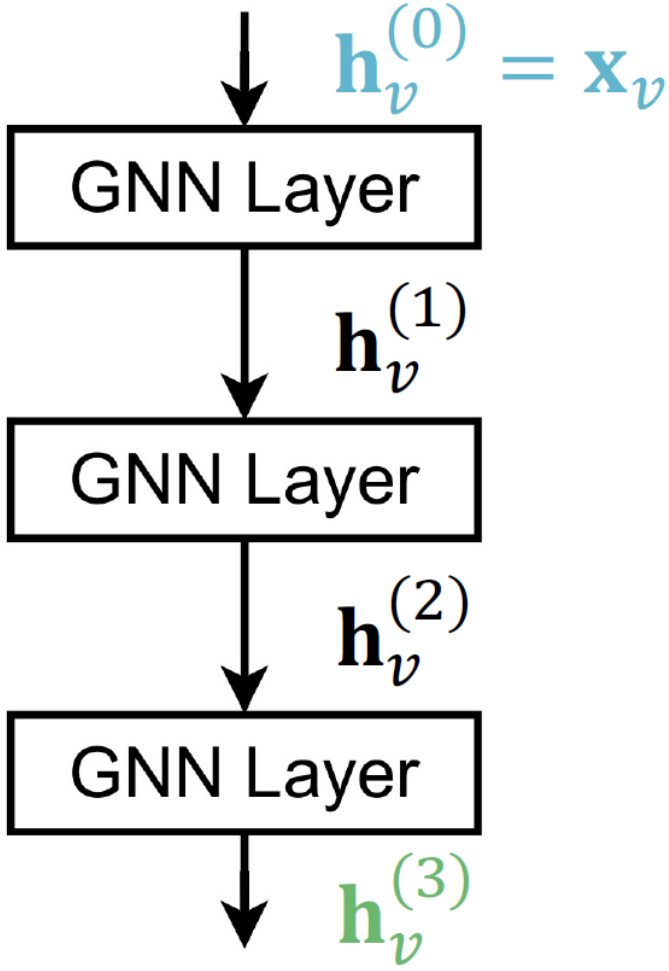
INPUT GRAPH

(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 $\mathbf{h}_v^{(0)} = \mathbf{x}_v$
  - **Output:** Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers

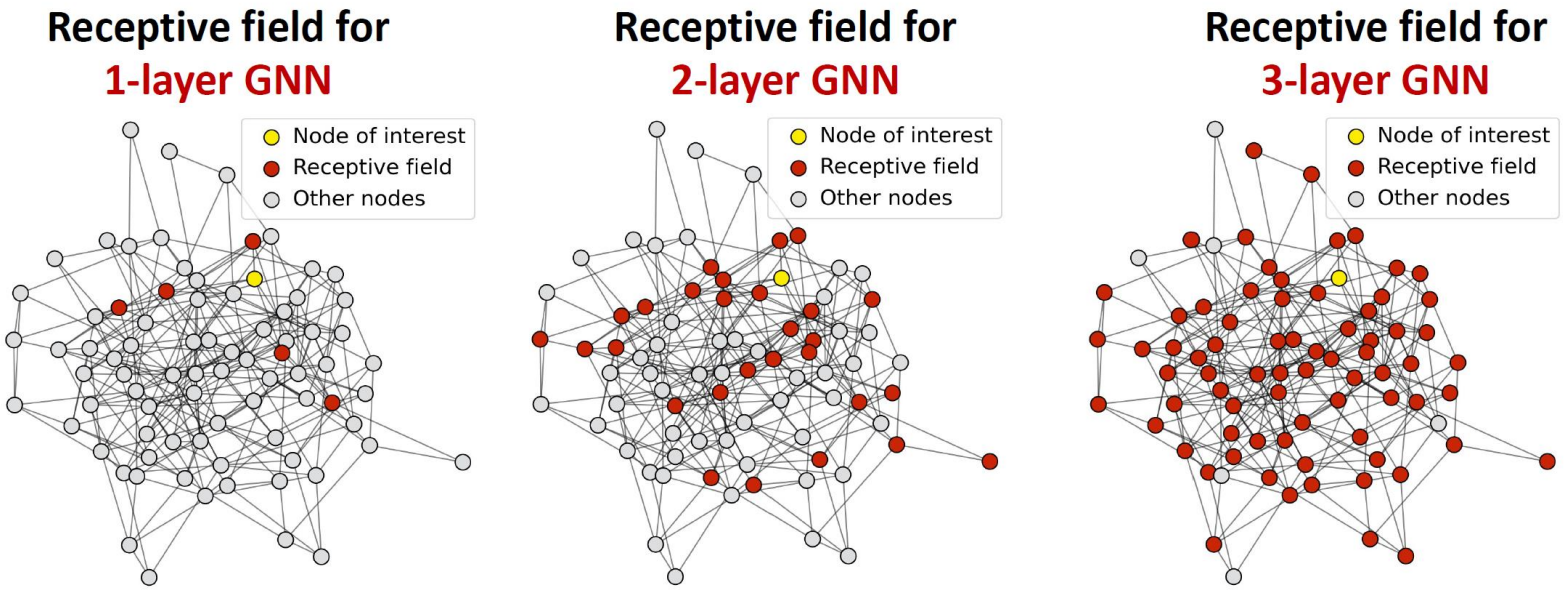


The Over-smoothing Problem

- **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**
  - **The shared neighbors quickly grows** when we increase the number of hops (num of GNN layers)



Receptive Field and Oversmoothing

- **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?

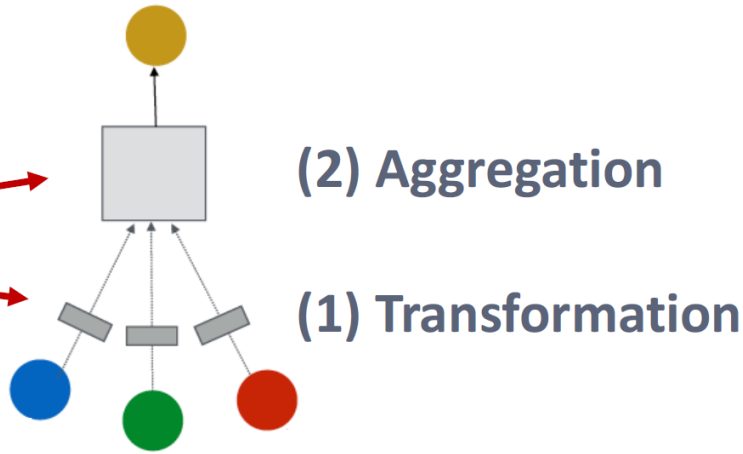
Design GNN Layer Connectivity

- **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 of Shallow GNNs

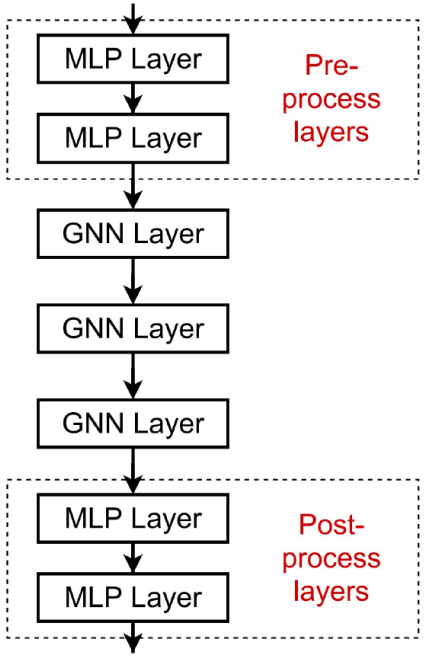
- **How to make a shallow GNN more expressive?**
- **Solution 1:** Increase the expressive power **within each GNN layer**
  - In our previous examples, each transformation or aggregation function only include one linear layer
  - We can **make aggregation / transformation become a deep neural network**!

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



Expressive Power of 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**



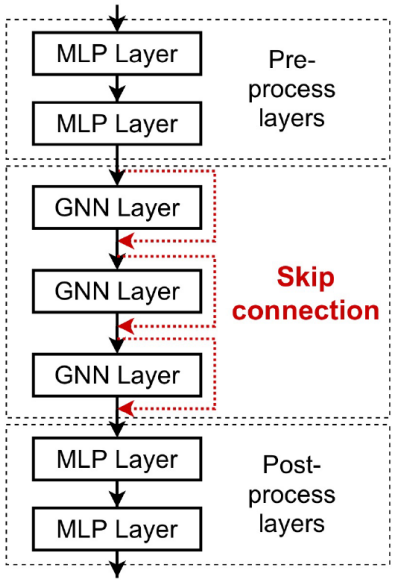
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

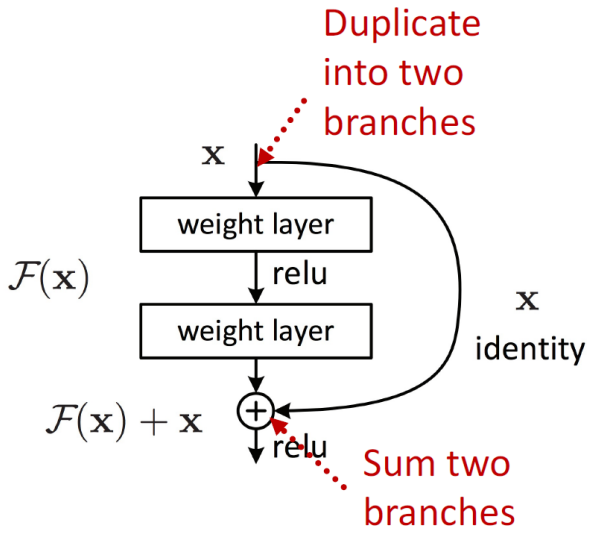
Post-processing layers: Important when reasoning / transformation over node embeddings are needed
E.g., graph classification, knowledge graphs

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:

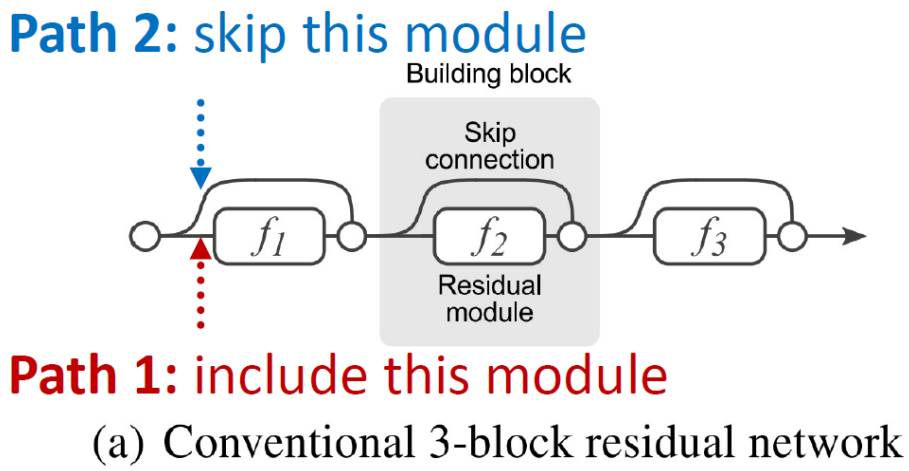
$$\boldsymbol{F}(\mathbf{x})$$

After adding shortcuts:

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

Idea of Skip Connections

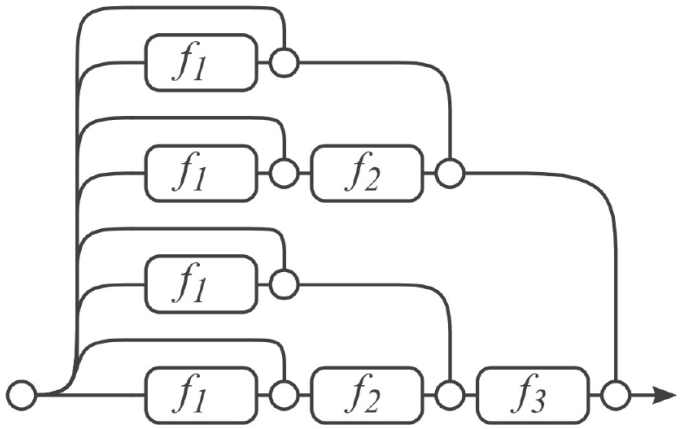
- **Why do skip connections work?**
- **Intuition:** Skip connections create **a mixture of models**
  - N skip connections → 2^N possible paths
  - Each path could have up to N modules
  - We automatically get **a mixture of shallow GNNs and deep GNNs**



(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**

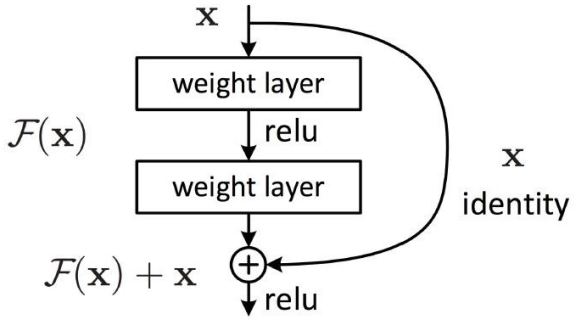
$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

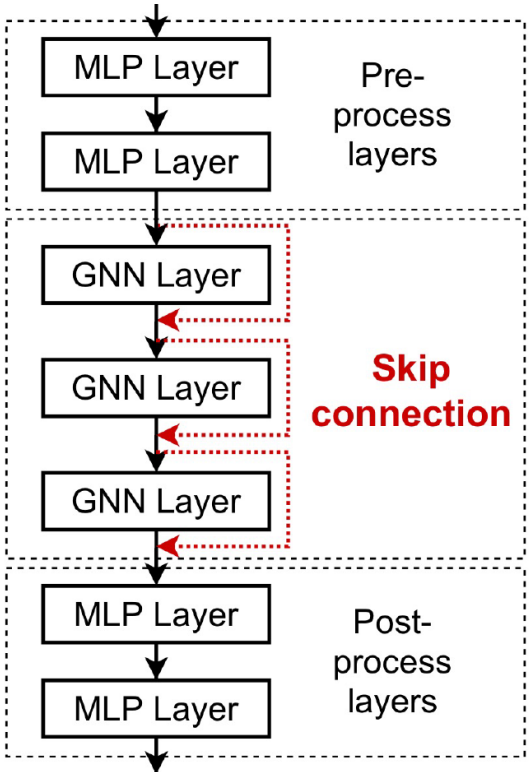
This is our $F(\mathbf{x})$

- **A GCN layer with skip connection**

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

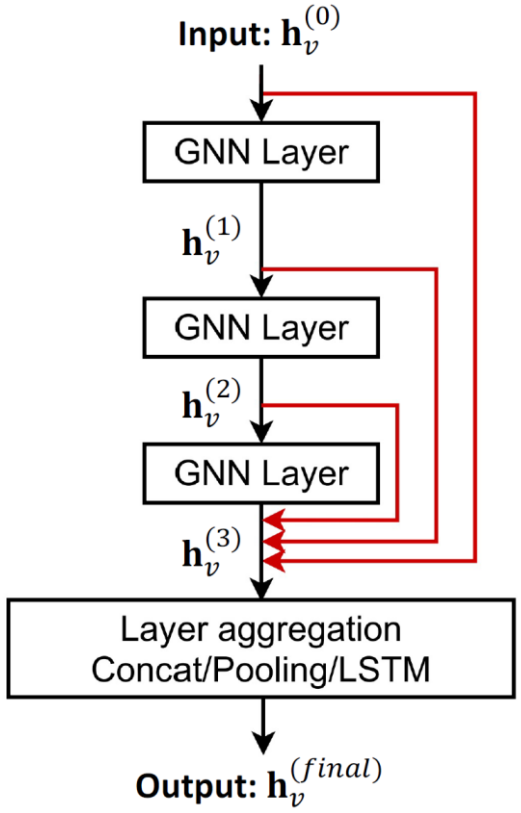
$F(\mathbf{x})$ + $\mathbf{x}$





Other Options of Skip Conncetions

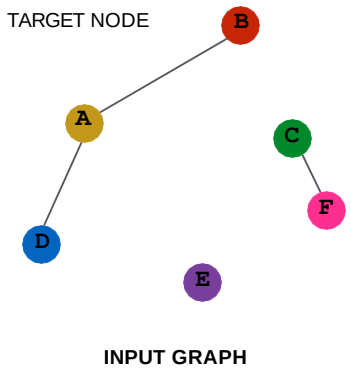
- **Other options:** Directly skip to the last layer
  - The final layer directly **aggregates from the all the node embeddings** in the previous layers

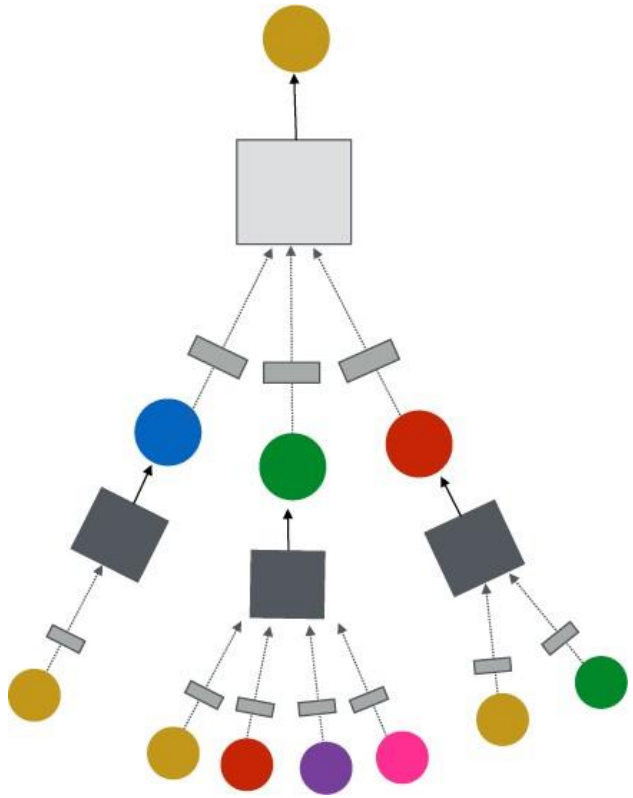


Graph Manipulation in GNNs

General GNN Framework

- Idea: Raw input graph ≠ computational graph
  - **Graph feature augmentation**
  - **Graph structure manipulation**





(4) Graph manipulation

Why Manipulate Graphs

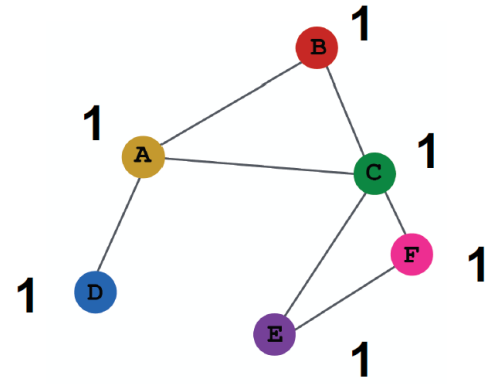
- Our assumption so far has been **Raw input graph = computational graph**
- **Reasons for breaking this assumption**
  - **Feature level:**
    - The input graph **lacks features** → feature augmentation
  - **Structure level:**
    - The graph is **too sparse** → inefficient message passing
    - The 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

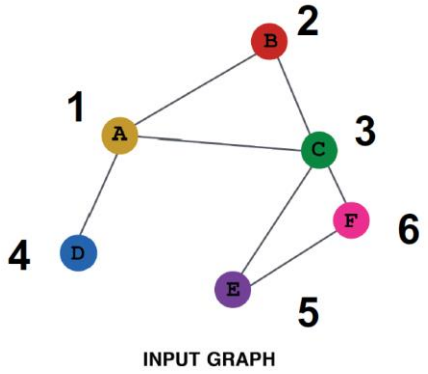
Graph Manipulation Approaches

- **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**
  - **b) Assign unique IDs to nodes**
    - These IDs are converted into **one-hot vectors**





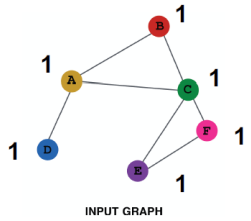
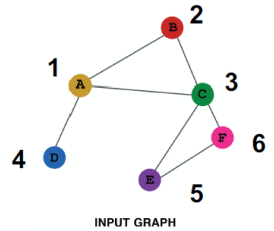
One-hot vector for node with ID=5

ID = 5

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

Total number of IDs = 6

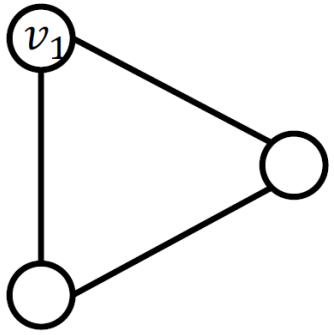
Feature Augmentation: Constant vs. One-hot

	Constant node feature	**One-hot node feature**
		
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**. $O \quad V$ dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

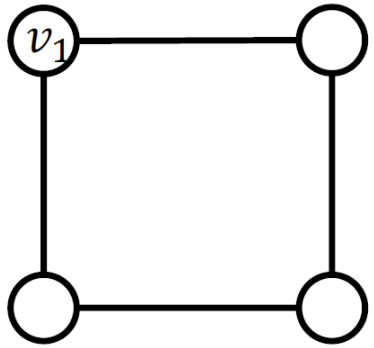
Feature Augmentation on Graphs

- 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



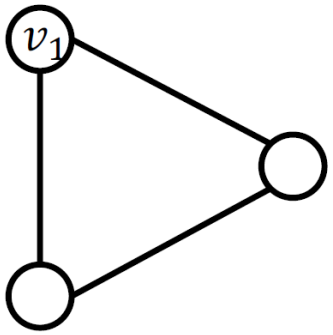
v_1 resides in a cycle with length 4



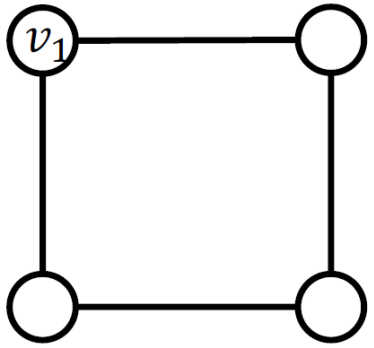
Feature Augmentation on Graphs

- **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

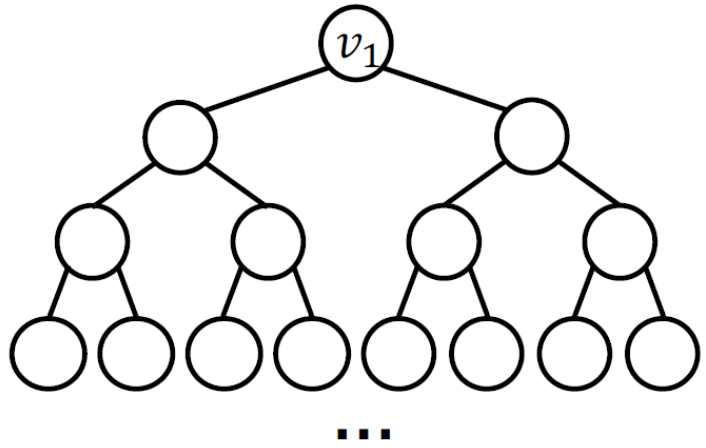
v_1 resides in a cycle with length 3



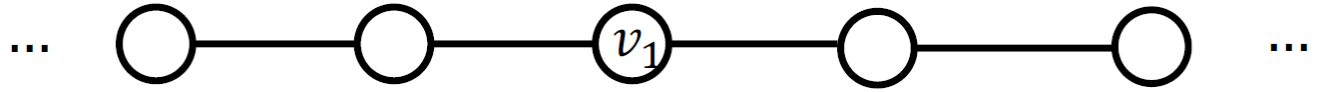
v_1 resides in a cycle with length 4



The computational graphs for node v_1 are always the same

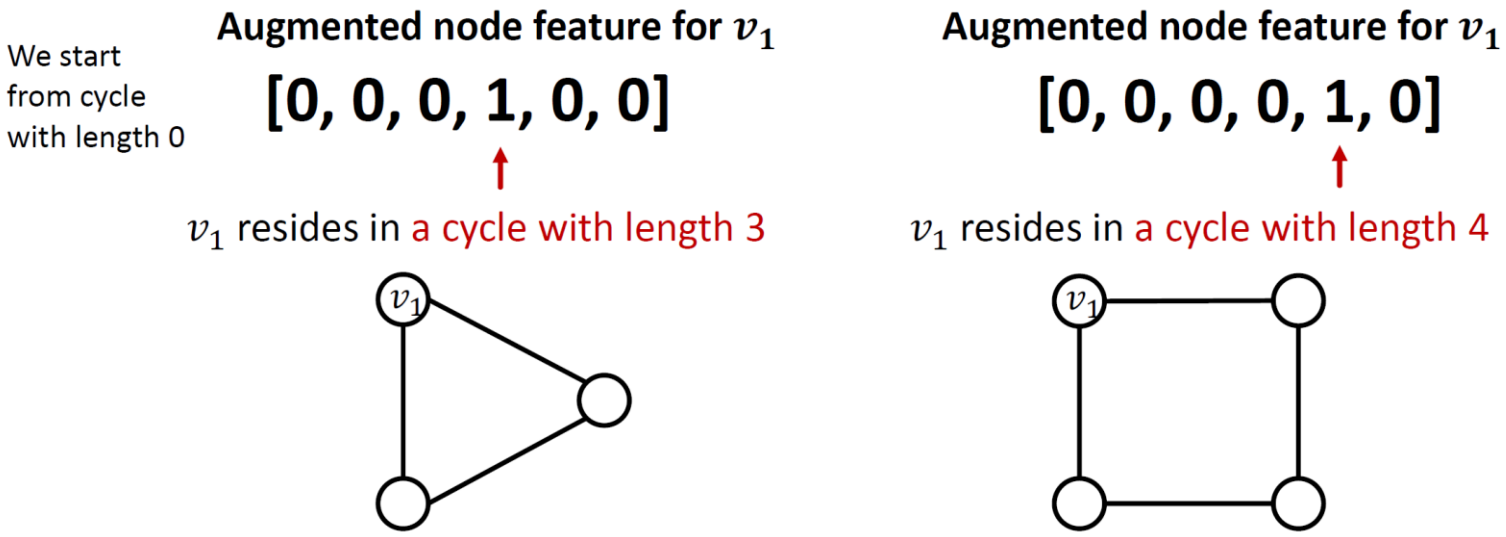


v_1 resides in a cycle with infinite length



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



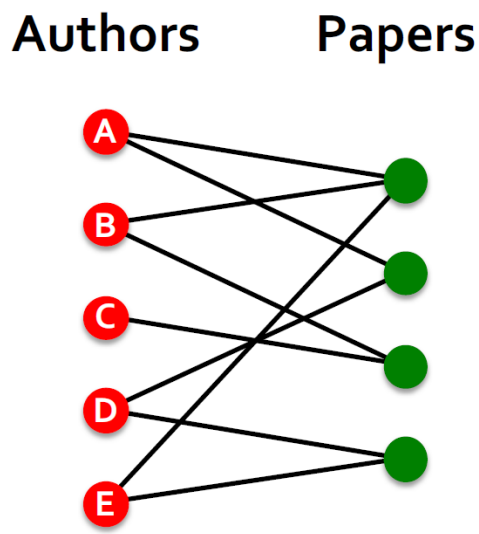
Feature Augmentation on Graphs

- Why do we need feature augmentation?
- (2) Certain structures are hard to learn by GNN
- Other commonly used augmented features:
  - **PageRank**
  - **Centrality**
  - **…**
  - Any feature we have introduced in Lecture 2 can be used!

Add Virtual Nodes/Edges

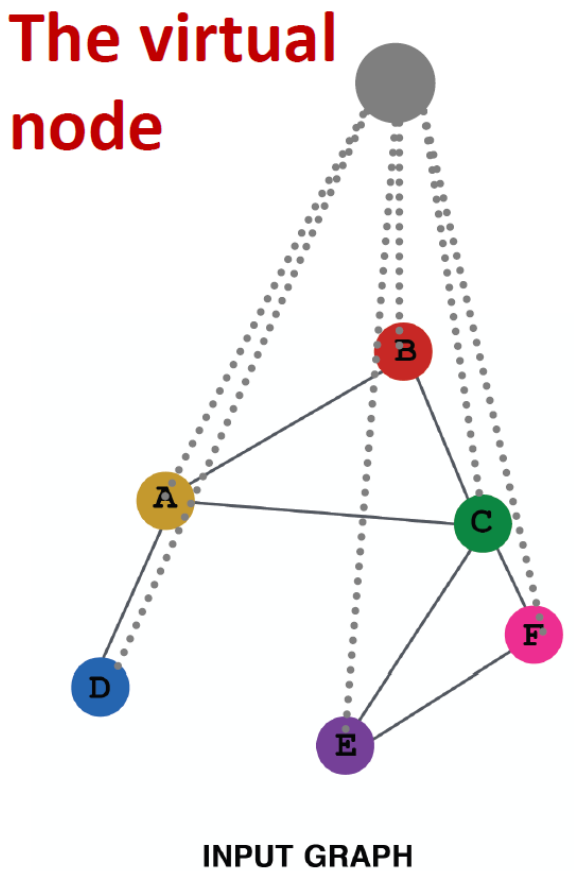
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
  - **Common approach:** Connect 2-hop neighbors via virtual edges
  - **Intuition:** Instead of using adjacency matrix A for GNN computation, use $A + A^2$

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



Add Virtual Nodes/Edges

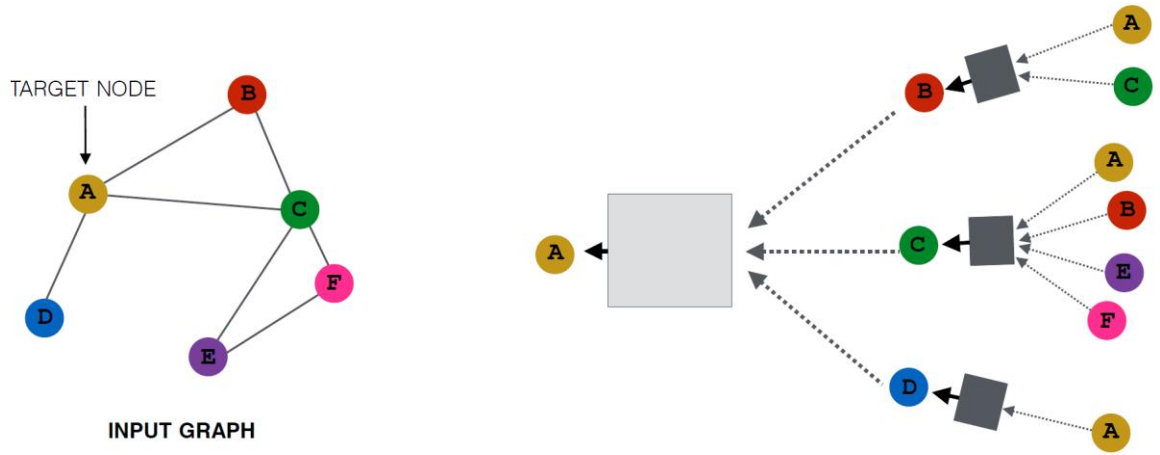
- **Motivation:** Augment sparse graphs
- **(2) Add virtual nodes**
- The virtual node will connect to all the nodes in the graph
- Suppose in a sparse graph, two nodes have shortest path distance of 10
- After adding the virtual node, **all the nodes will have a distance of 2**
  - **Node A – Virtual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



Xin Liu, Jiayang Cheng, Yangqiu Song, Xin Jiang: Boosting Graph Structure Learning with Dummy Nodes. ICML 2022: 13704-13716

Node Neighborhood Sampling

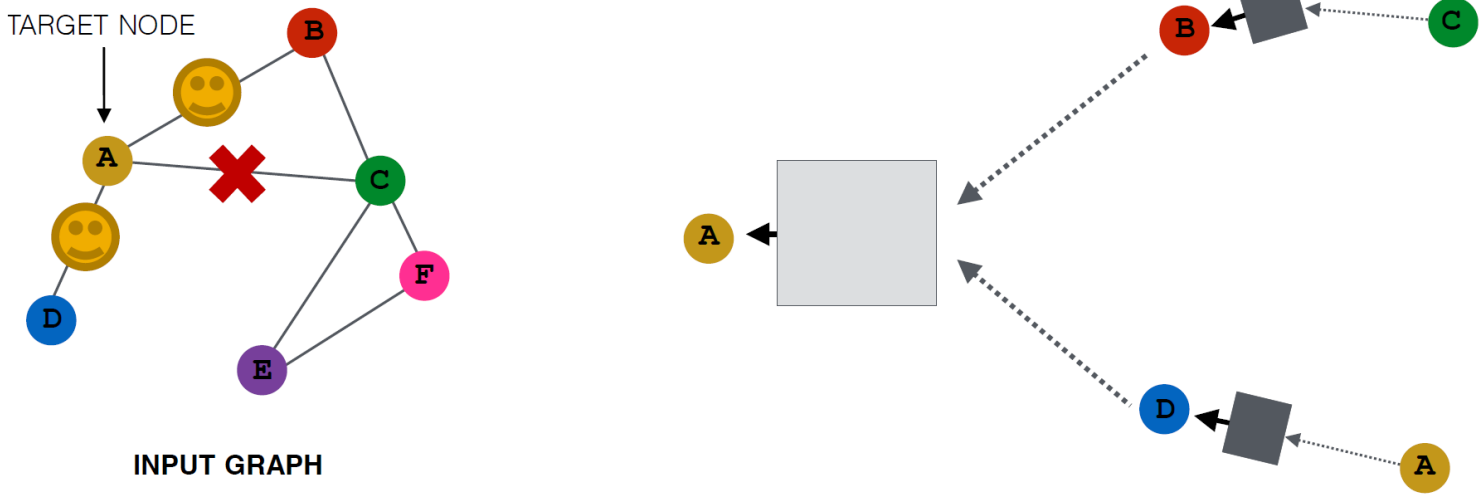
- **Previously**
  - All the nodes are used for message passing



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

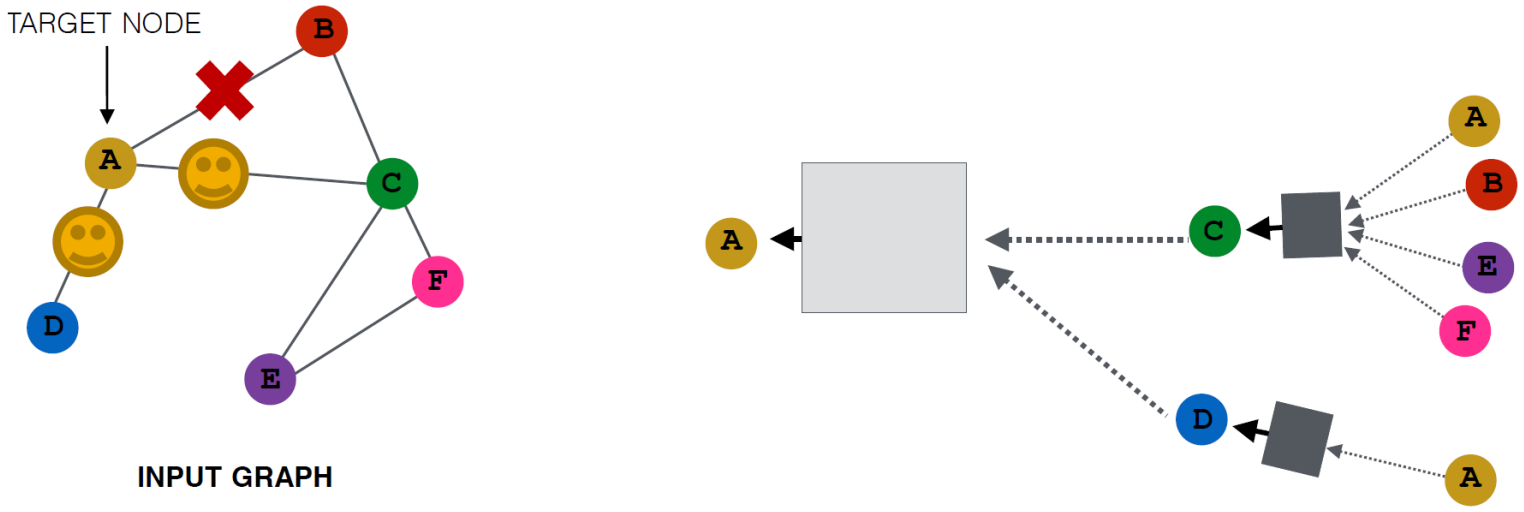
Neighborhood Sample Example

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



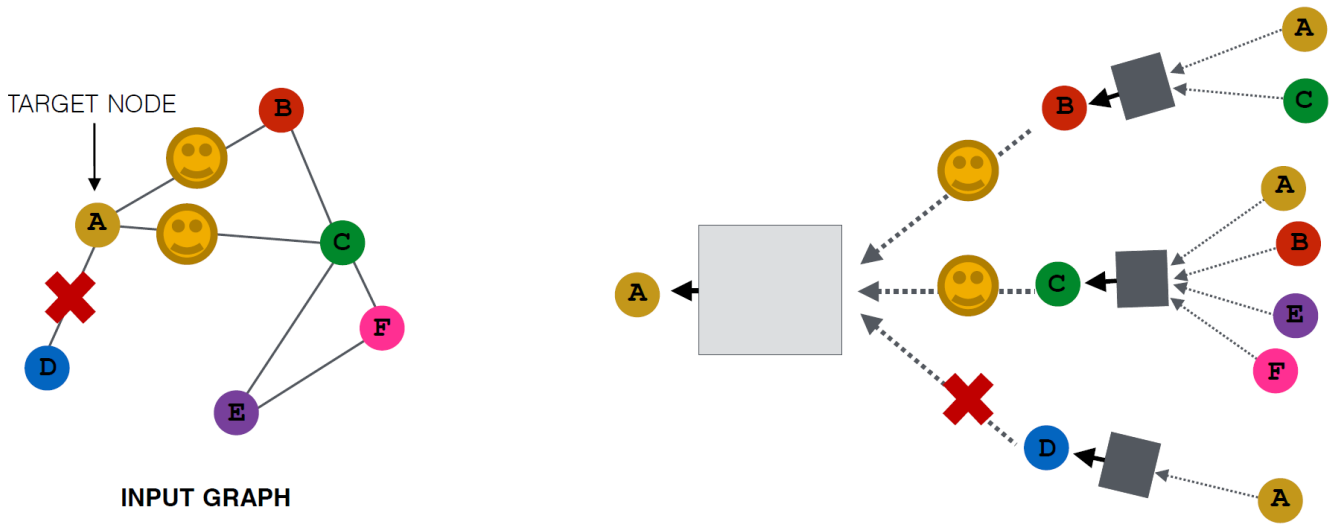
Neighborhood Sample Example

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



Neighborhood Sample 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!



Ying et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems, KDD 2018

Summary

- **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