

Graph Neural Networks

General Framework

Acknowledgement: Jure Leskovec, Stanford University

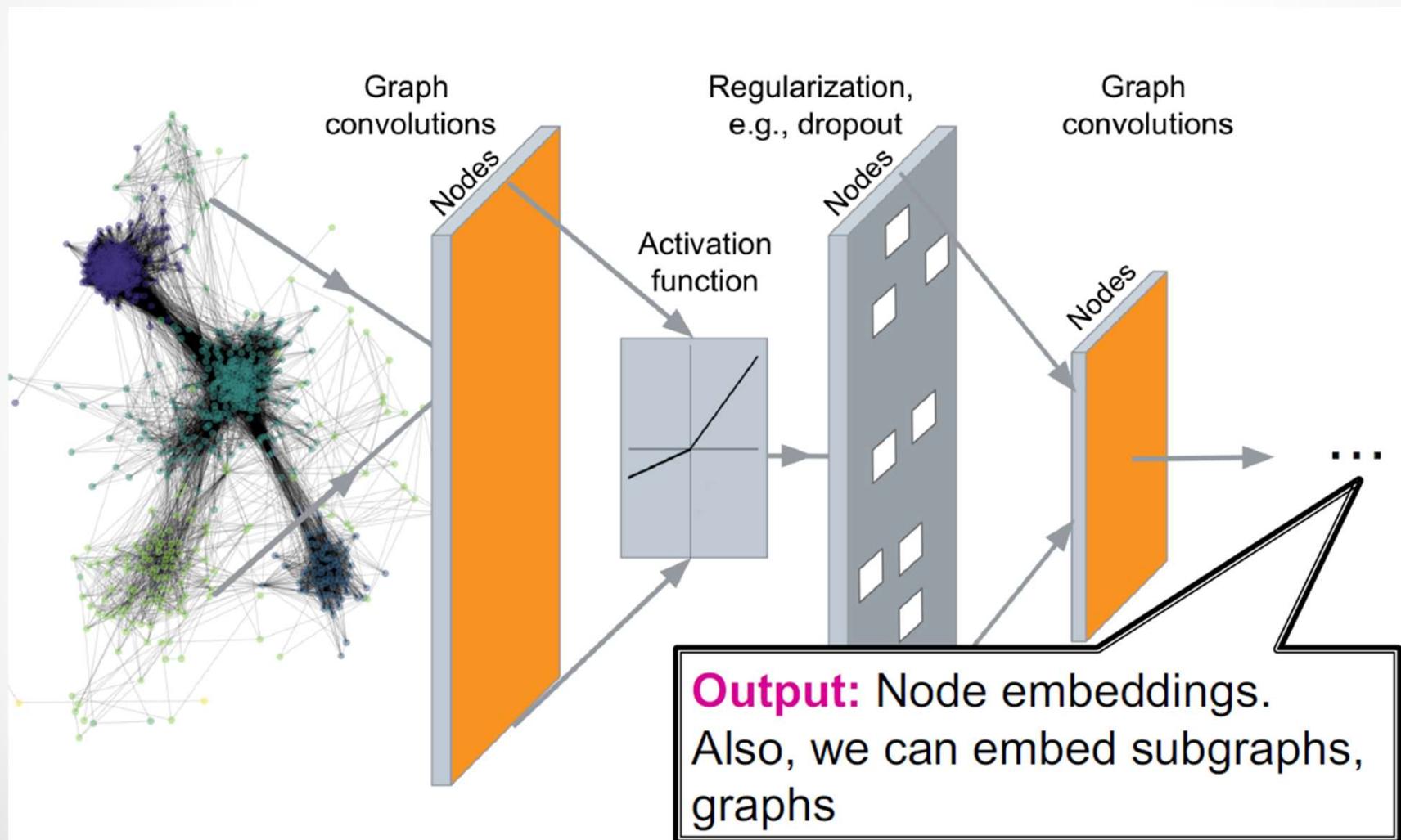
Overview

- GNN – General Framework
- A Single Layer of GNN
- Stacking Layers of a GNN
- Graph Augmentation for GNNs

Overview

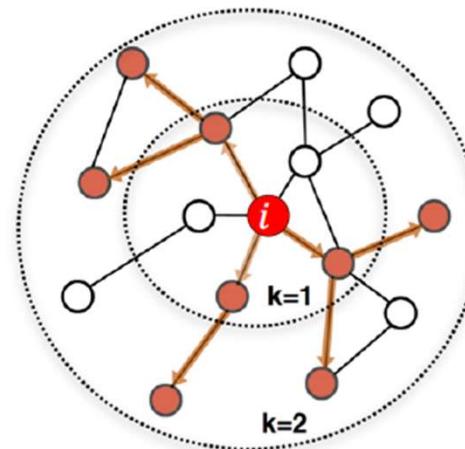
- GNN – General Framework
- A Single Layer of GNN
- Stacking Layers of a GNN
- Graph Augmentation for GNNs

Deep Graph Encoders

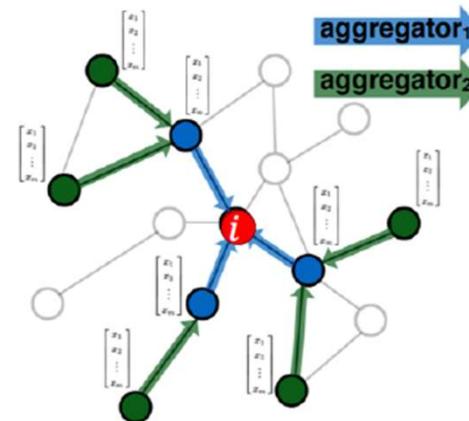


Graph Neural Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

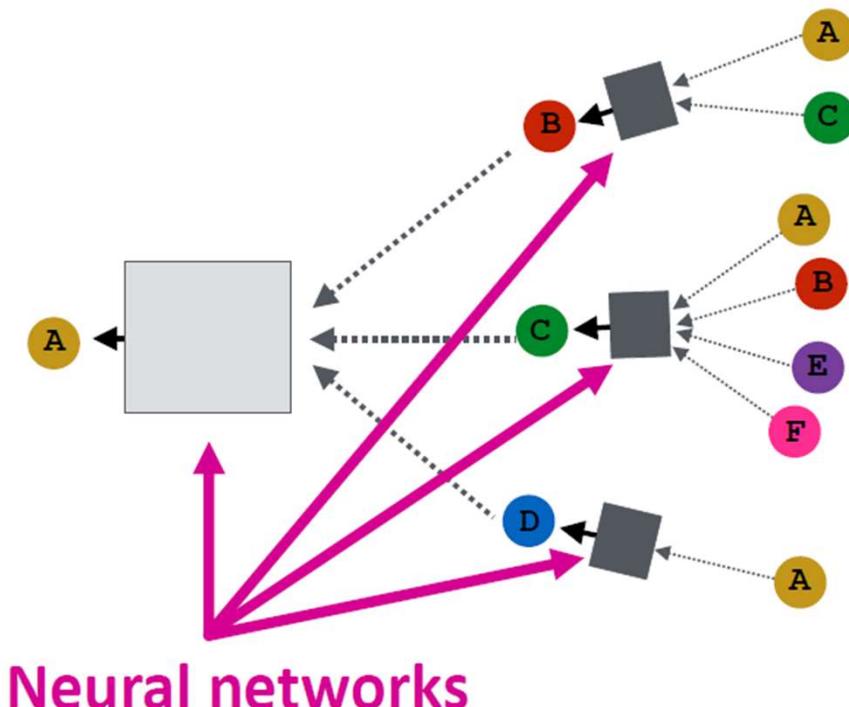
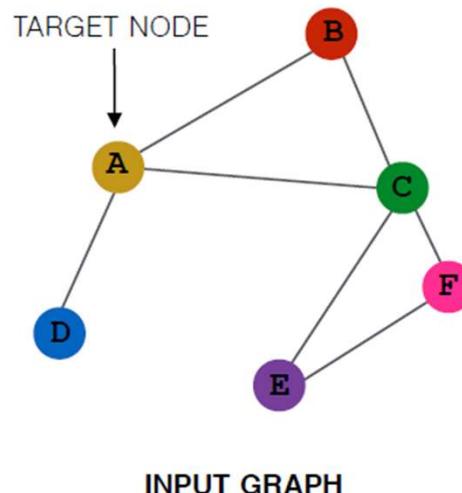


Propagate and transform information

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

Aggregate from Neighbors

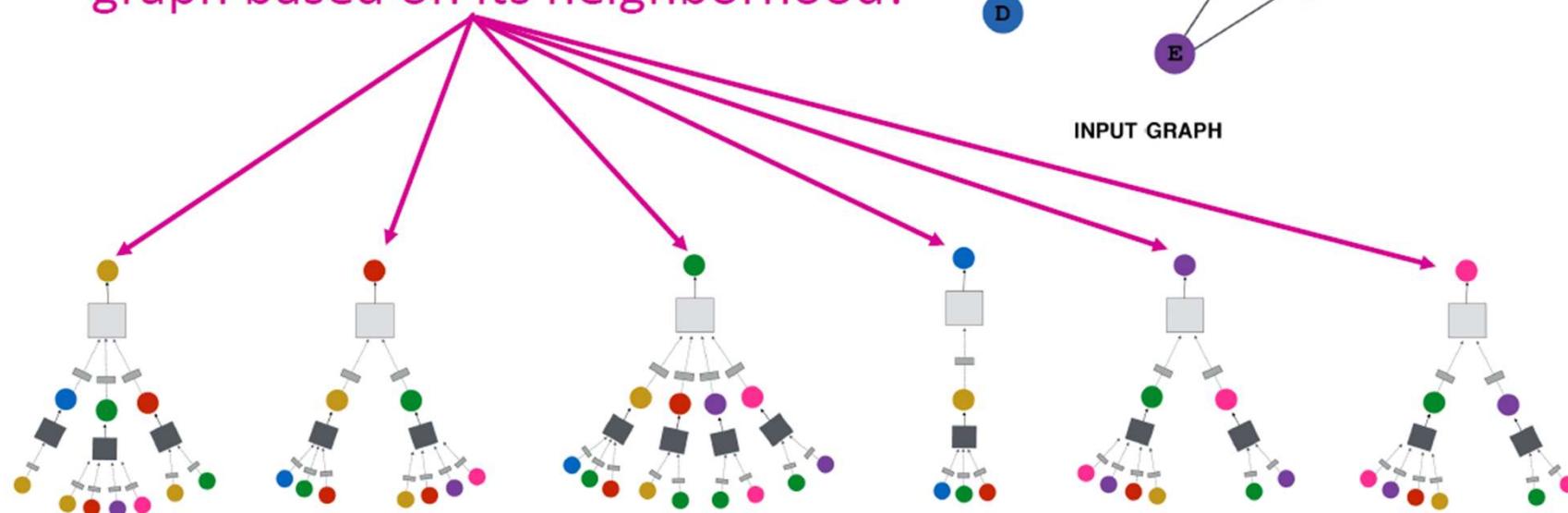
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



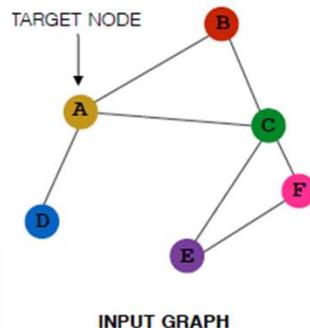
Aggregate from Neighbors

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

Every node defines a computation graph based on its neighborhood!

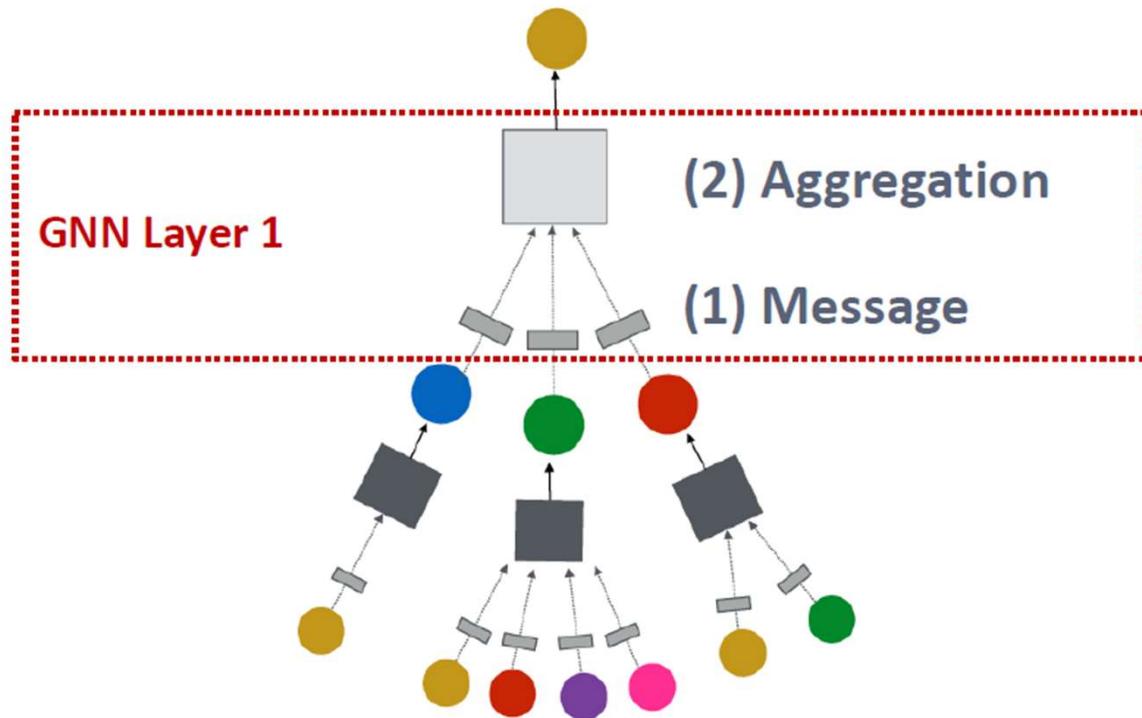


A General GNN Framework



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

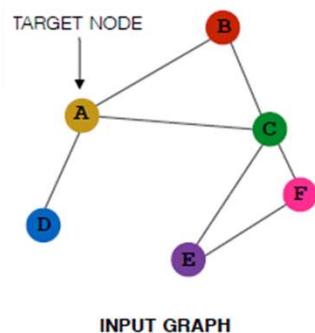
Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity

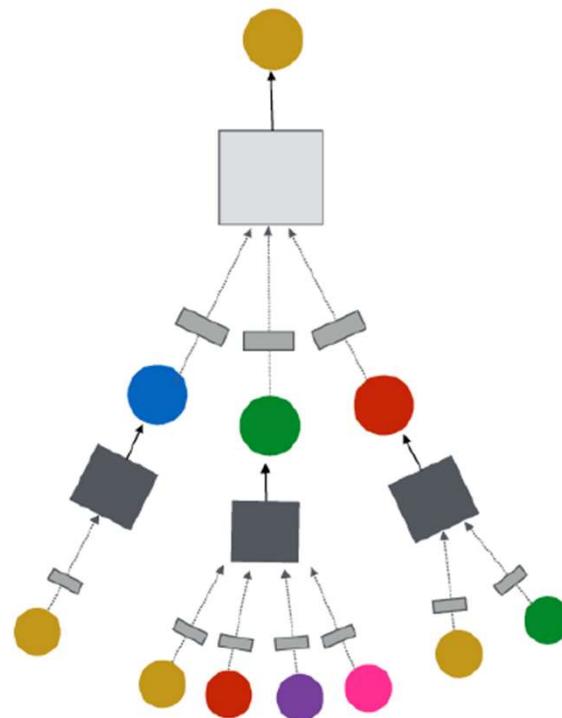
The diagram illustrates a three-layer GNN framework. It starts with an **INPUT GRAPH** containing nodes A, B, C, D, E, and F. Node A is labeled as the **TARGET NODE**. The graph is processed through two **GNN Layer**s. **GNN Layer 1** processes nodes A, B, C, D, E, and F. **GNN Layer 2** processes nodes A, B, C, D, E, and F. The final output is a yellow node. Skip connections are shown as dashed arrows from the input graph nodes to the output node.

A General GNN Framework



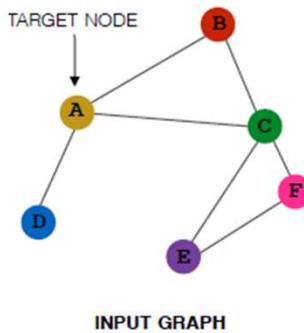
Idea: Raw input graph \neq computational graph

- Graph feature augmentation
- Graph structure augmentation

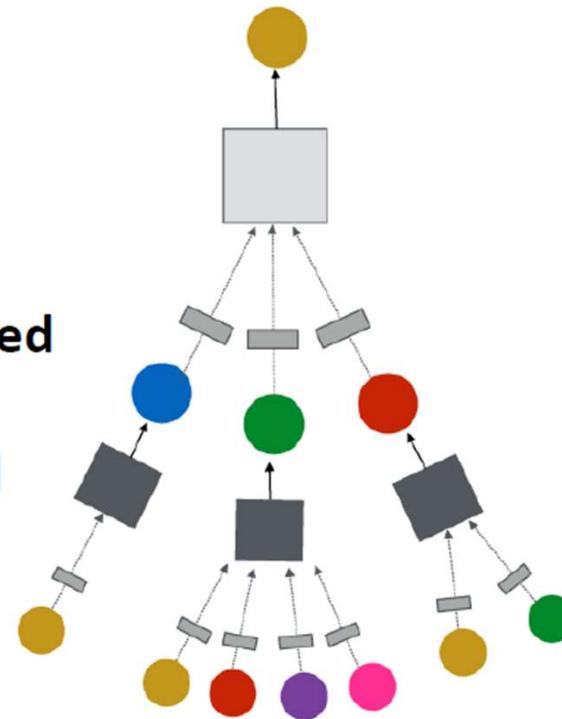


(4) Graph augmentation

A General GNN Framework



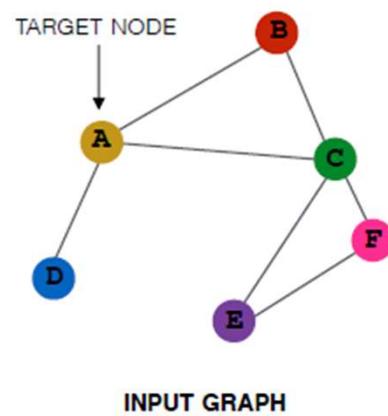
(5) Learning objective



How do we train a GNN

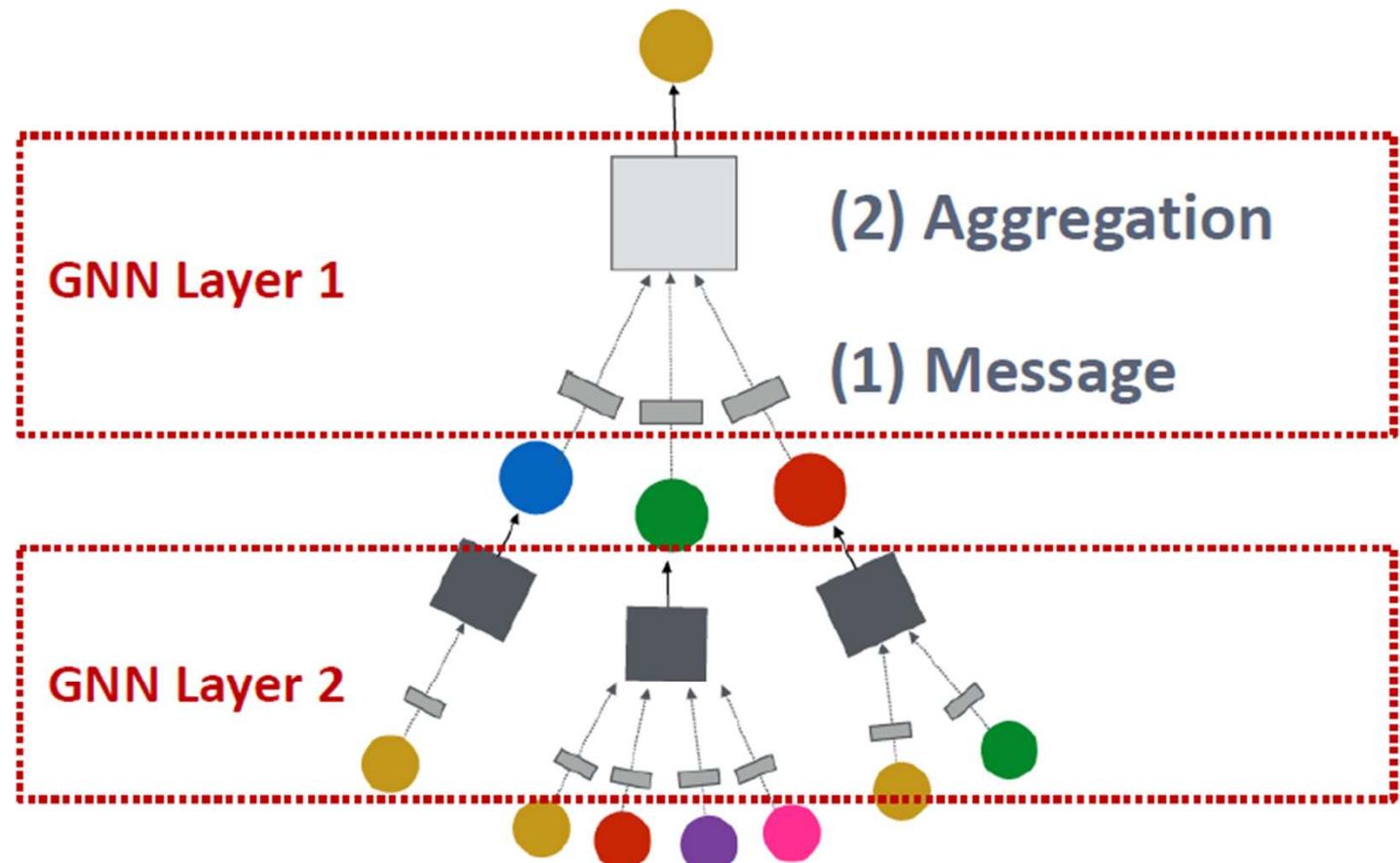
- Supervised/Unsupervised objectives
- Node/Edge/Graph level objectives

A General GNN Framework



(3) Layer connectivity

(5) Learning objective

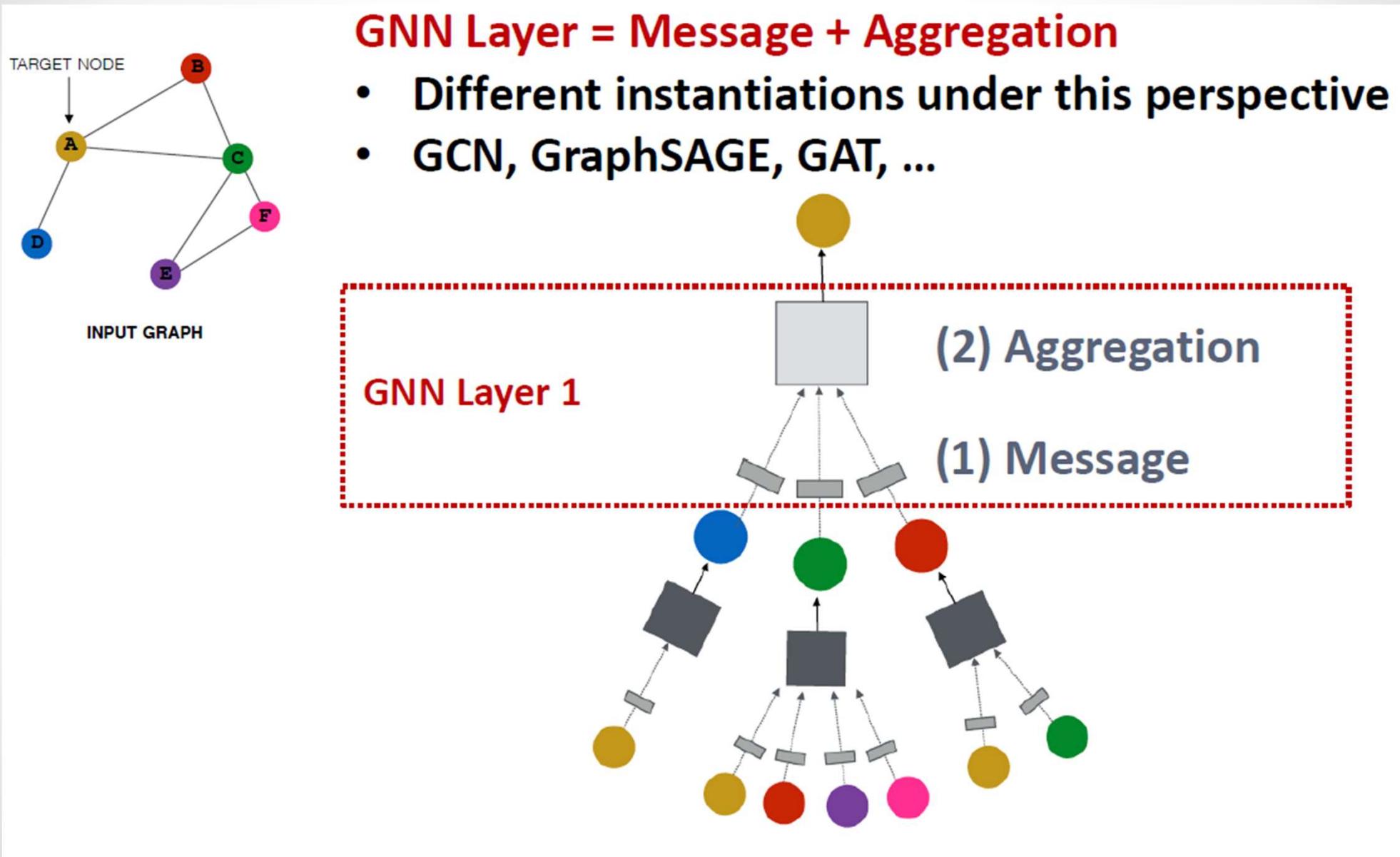


(4) Graph augmentation

Overview

- GNN – General Framework
- A Single Layer of GNN
- Stacking Layers of a GNN
- Graph Augmentation for GNNs

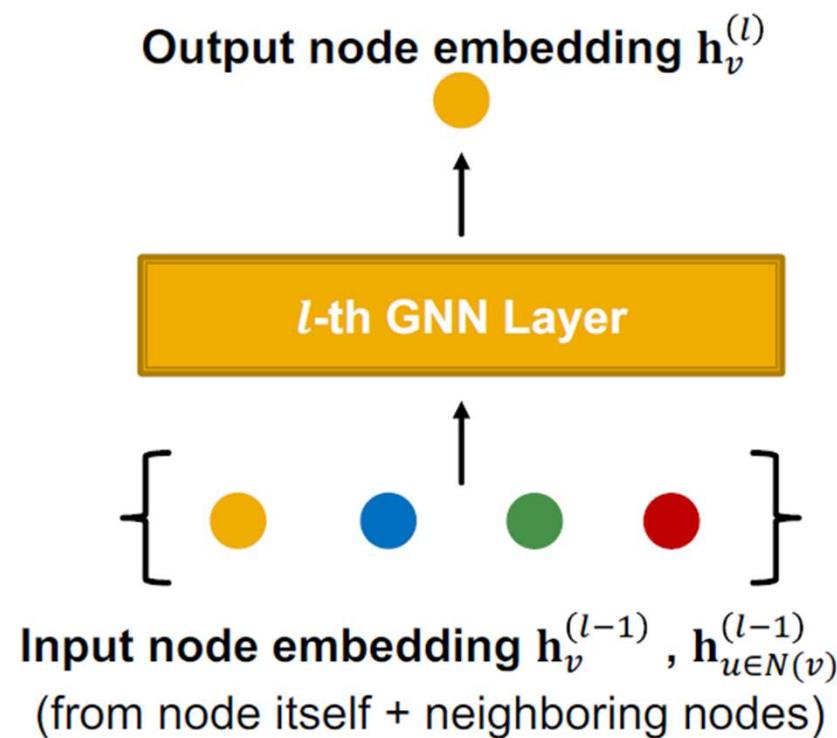
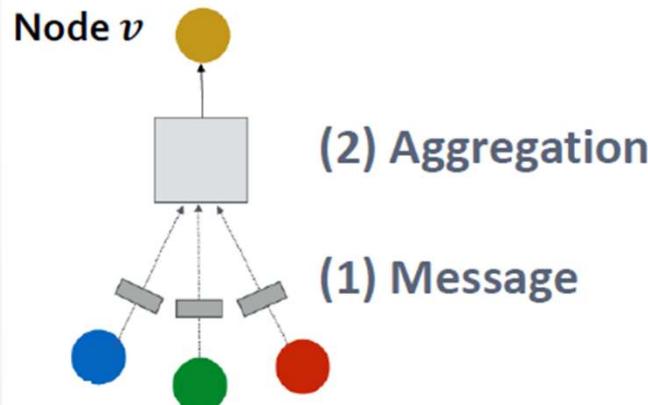
A GNN Layer



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

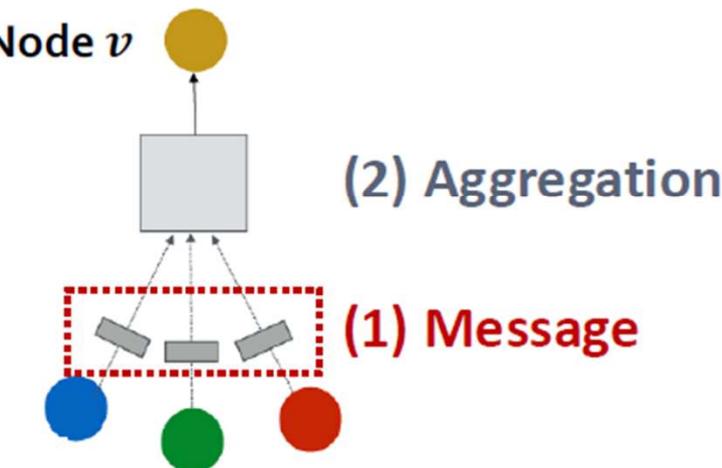
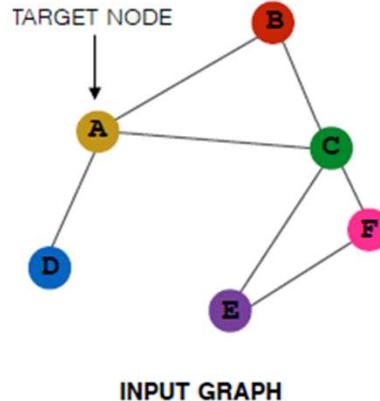


Set: Ordering of the message aggregation does not matter.
Aggregation function has to be order invariant.

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

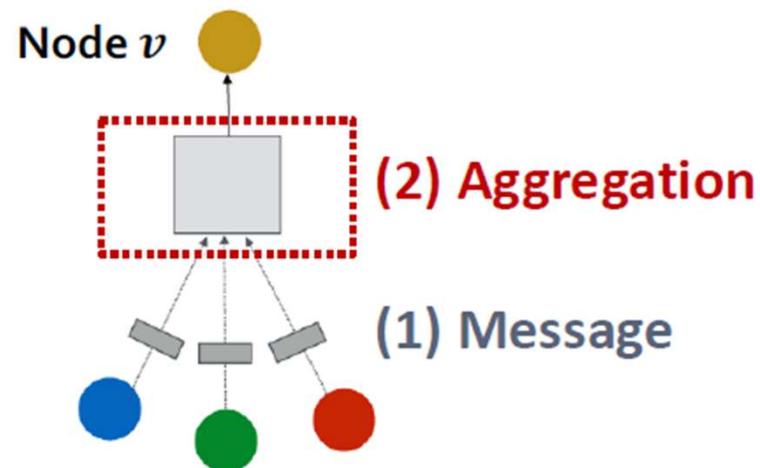
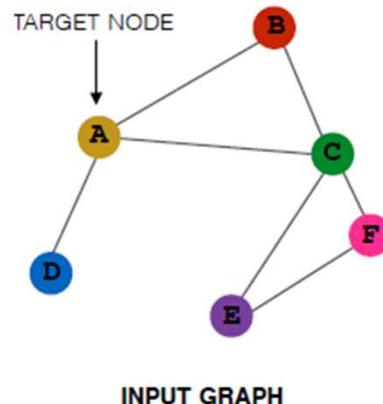
■ (2) Aggregation

- **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

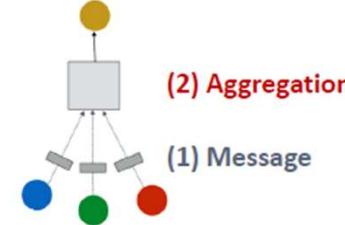
$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)} \left(\mathbf{h}_u^{(l-1)} \right), u \in \{N(v) \cup v\}$$

- **(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**



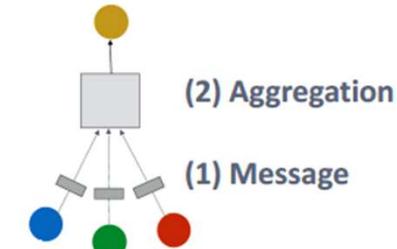
Classical GNN Layers: GCN

- (1) 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?

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

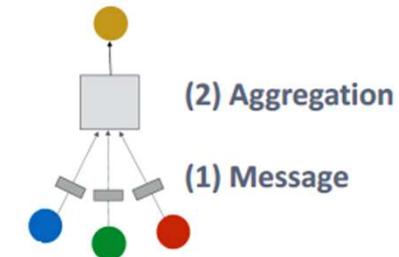


T. Kipf, M. Welling. Semi-Supervised Classification with Graph Convolutional Networks, ICLR 2017

Classical GNN Layers: GCN

■ (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

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

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

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

AggregationLSTM is not order invariant

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

■ (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

■ 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

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

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 $\mathbf{h}_v^{(l)}$ 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

Vaswani et al., NIPS 2017 ; Velickovic et al., ICLR 2018

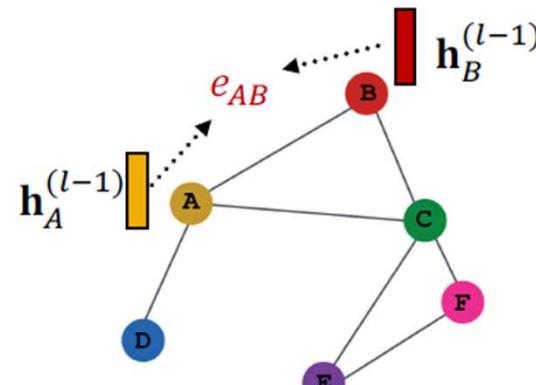
Attention Mechanism

- 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

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

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

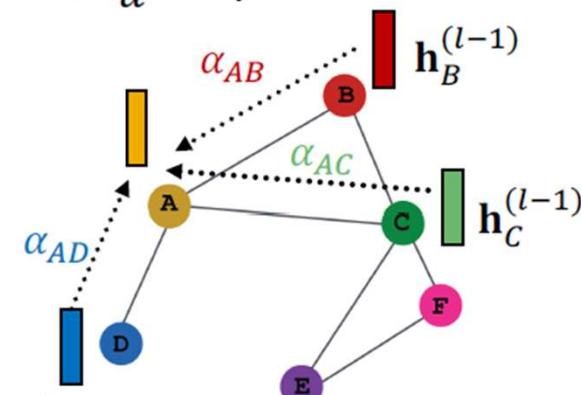
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

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

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

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

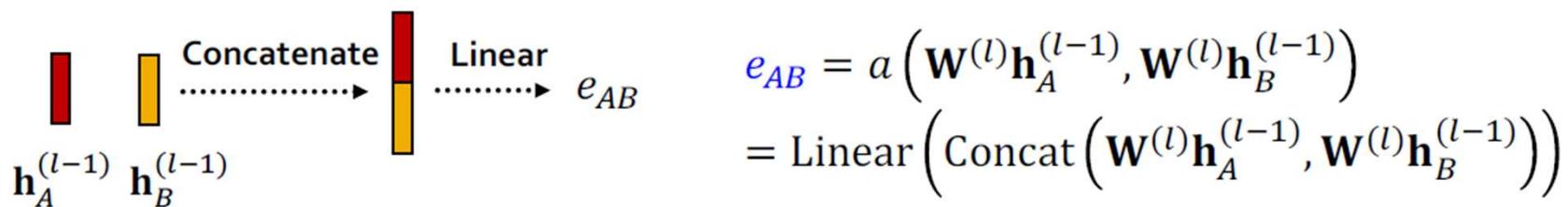
$$\mathbf{h}_A^{(l)} = \sigma(\alpha_{AB} \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)} + \alpha_{AC} \mathbf{W}^{(l)} \mathbf{h}_C^{(l-1)} + \alpha_{AD} \mathbf{W}^{(l)} \mathbf{h}_D^{(l-1)})$$



Attention Mechanism

■ What is the form of attention mechanism a ?

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



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

Note that e_{AB} and e_{BA} need not be equal

Attention Mechanism

- **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

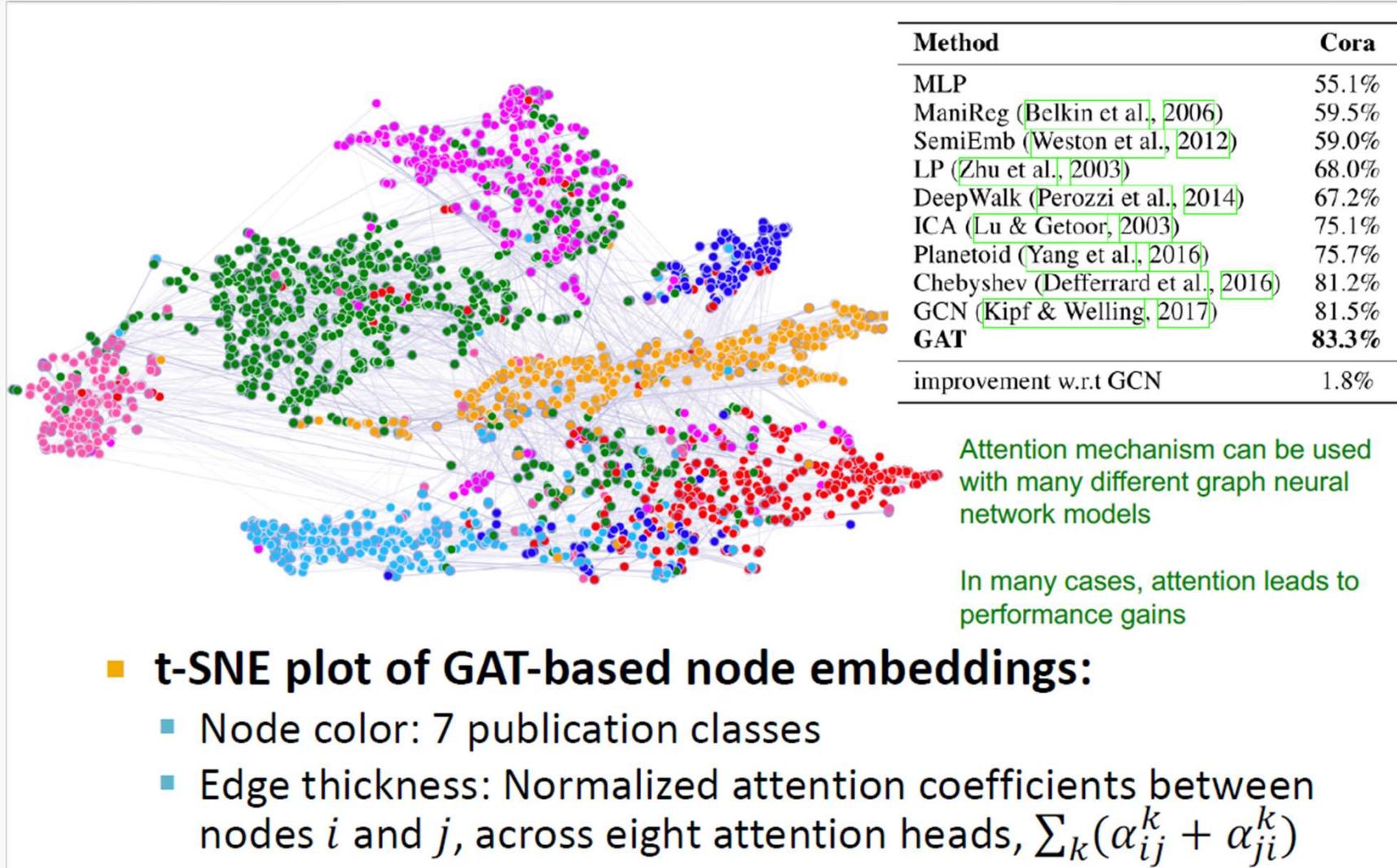
- $\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

Each attention score has to be trained separately

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

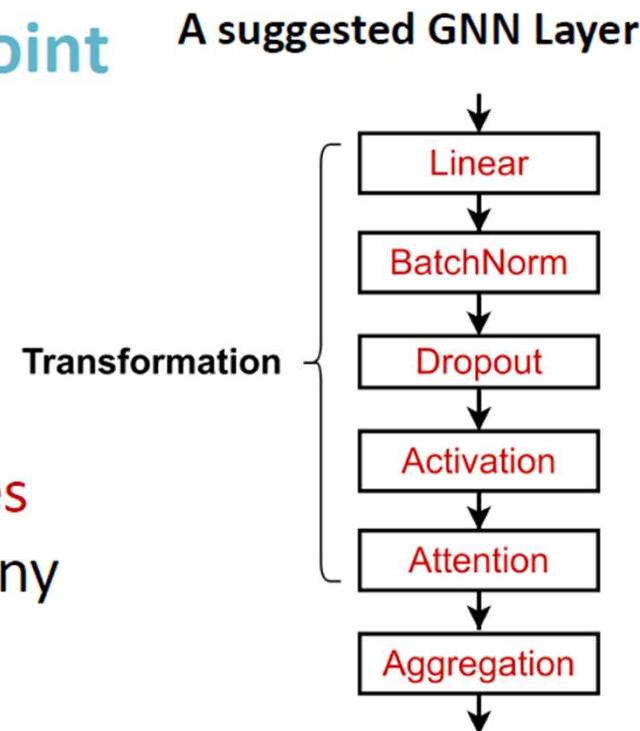
GAT Example: Cora Citation Net



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

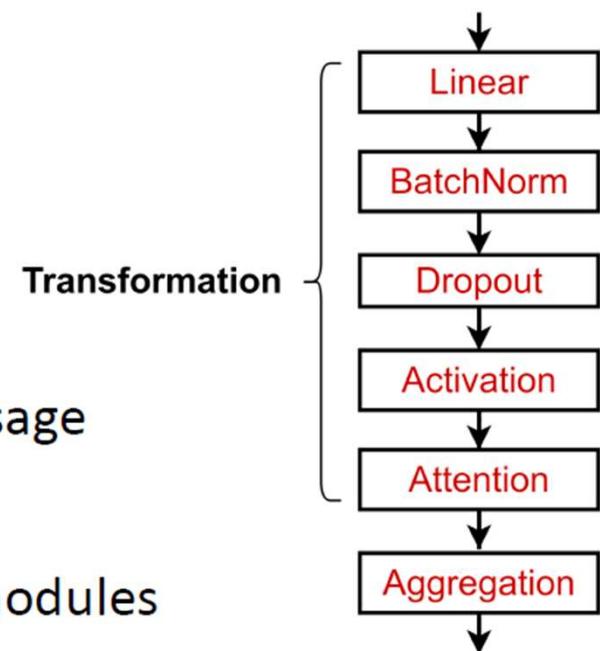
- **Attention/Gating:**

- Control the importance of a message

- **More:**

- Any other useful deep learning modules

A suggested GNN Layer



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

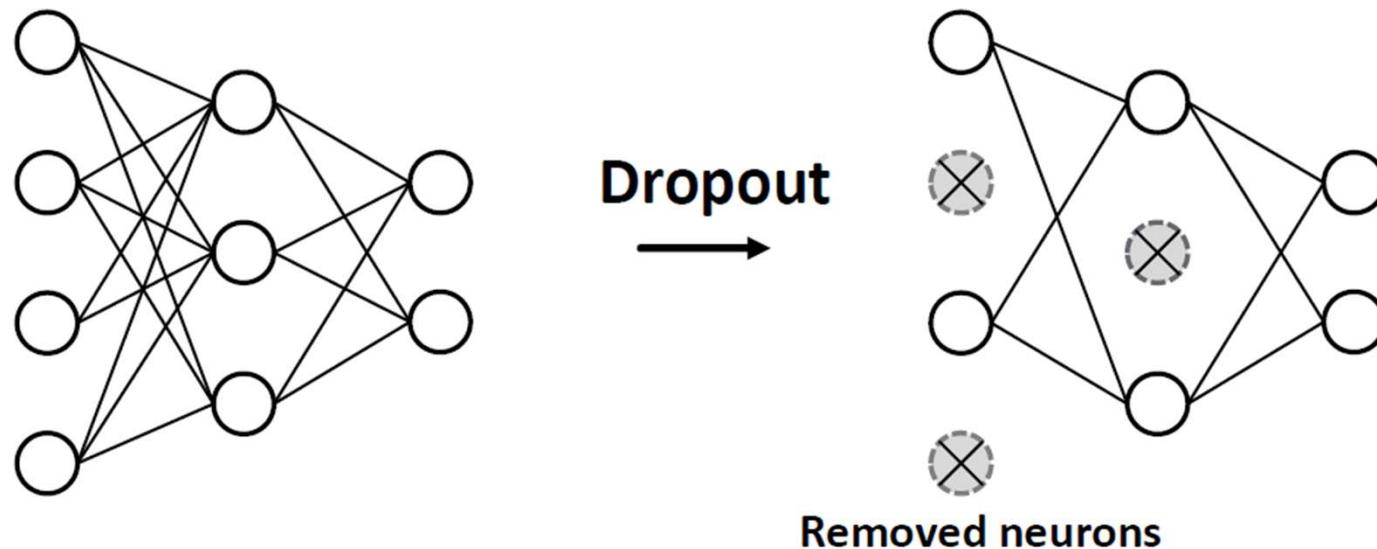
$$\mu_j = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_{i,j}$$
$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (\mathbf{x}_{i,j} - \mu_j)^2$$

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



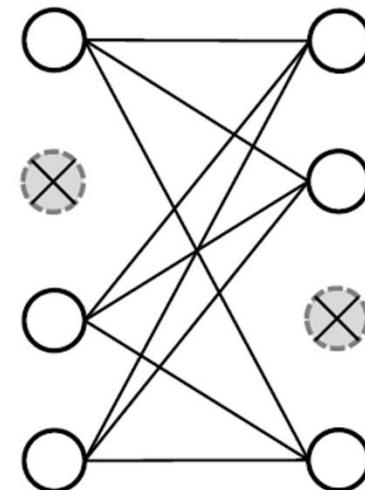
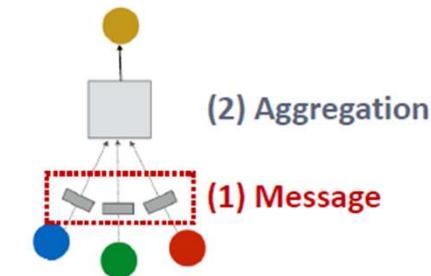
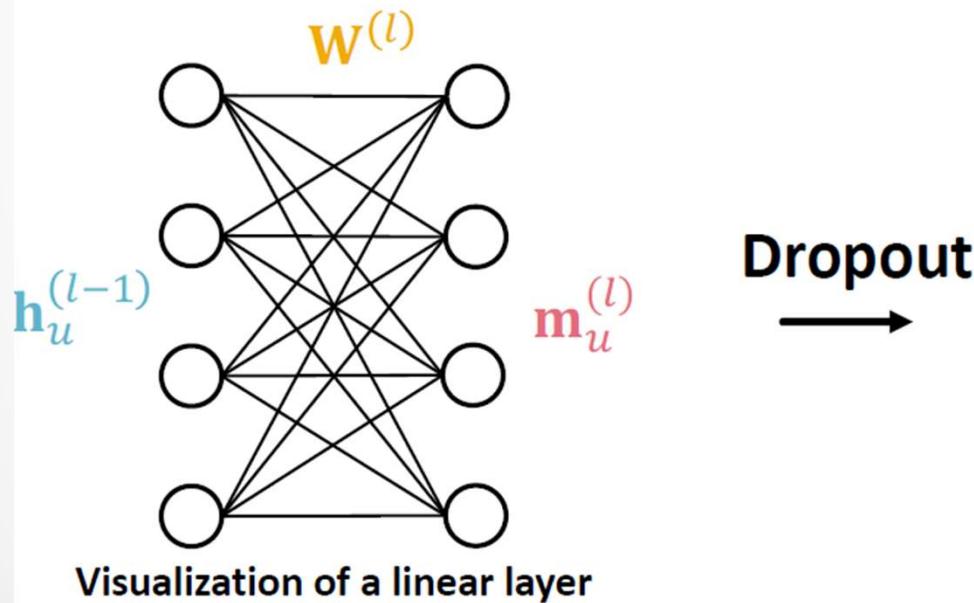
Overfitting occurs when a model fits too closely or even exactly to its training data, resulting in a model that can't make accurate predictions or conclusions from any data other than the training data

Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

Dropouts for GNN

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

- A simple message function with linear layer: $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}$



Activations (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

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

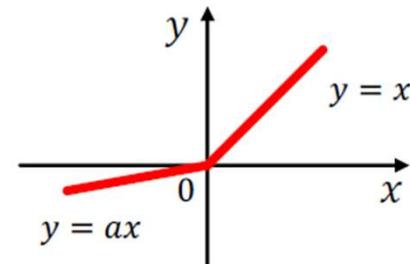
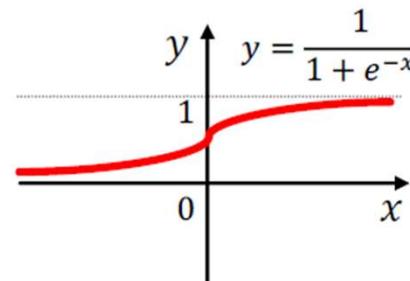
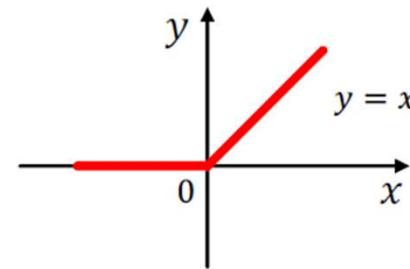
- Used only when you want to restrict the range of your embeddings

- Parametric ReLU

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

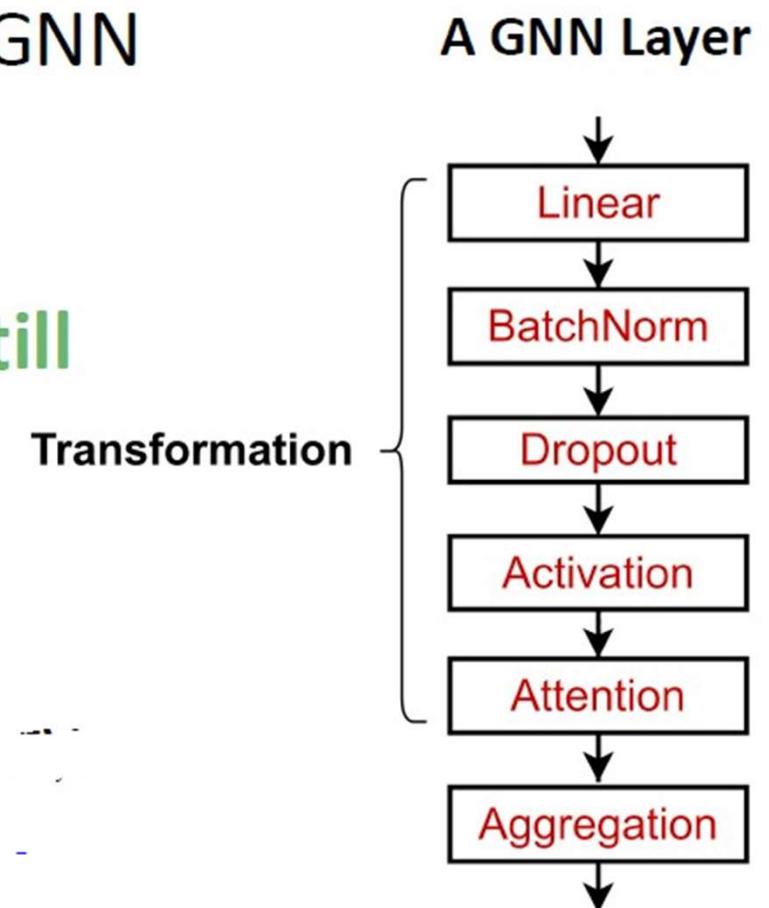
a_i is a trainable parameter

- Empirically performs better than ReLU



GNN Layer in Practice

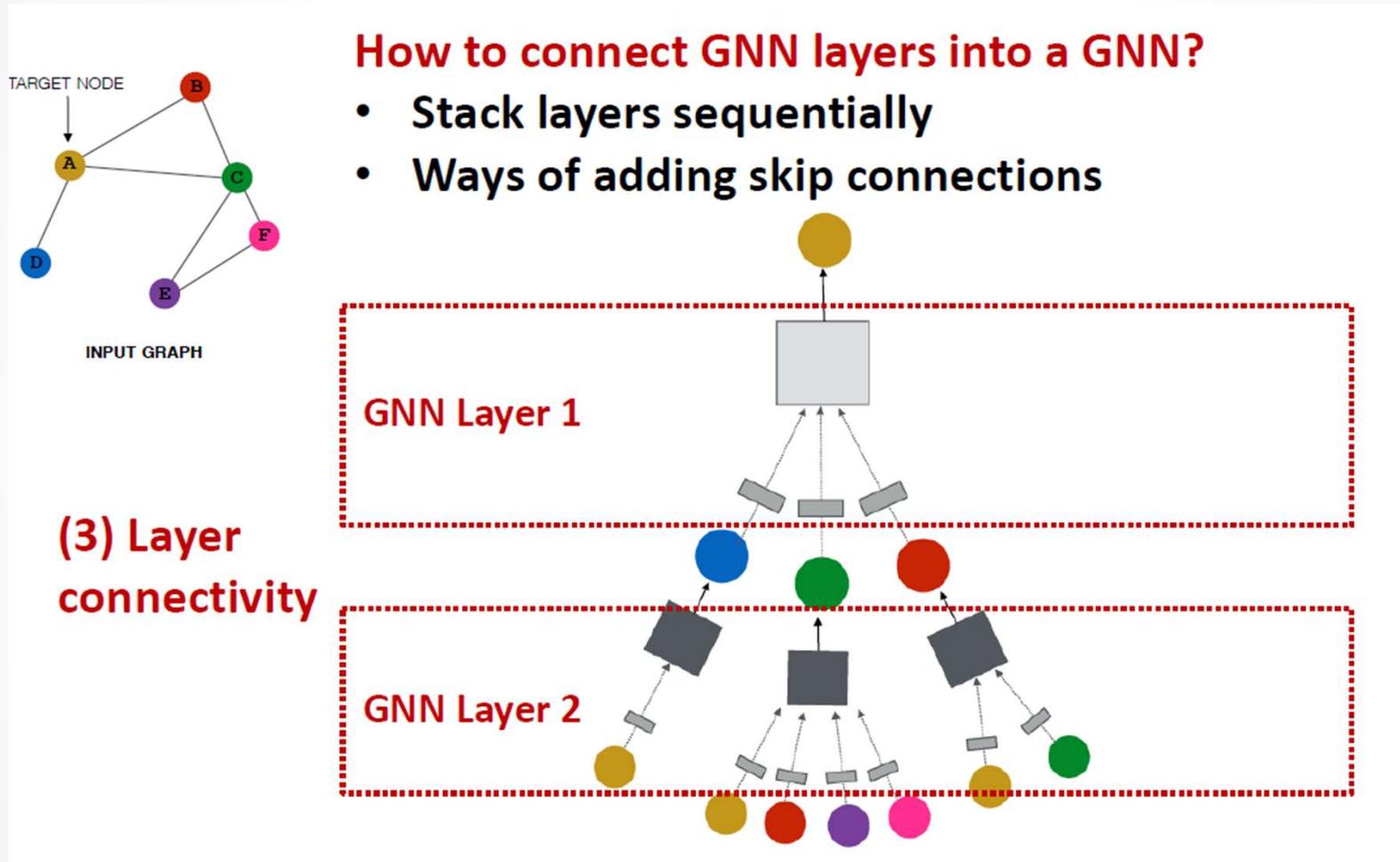
- **Summary:** Modern deep learning modules can be included into a GNN layer for better performance
- **Designing novel GNN layers is still an active research frontier!**



Overview

- GNN – General Framework
- A Single Layer of GNN
- Stacking Layers of a GNN
- Graph Augmentation for GNNs

Stacking GNN Layers

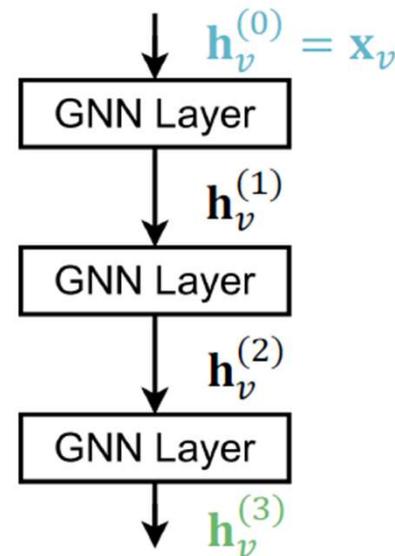


J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks,
NeurIPS 2020

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

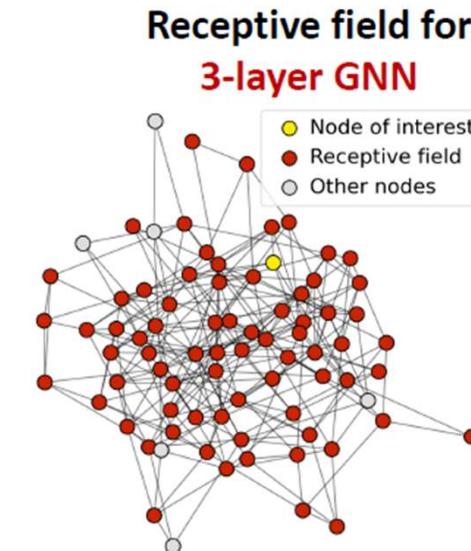
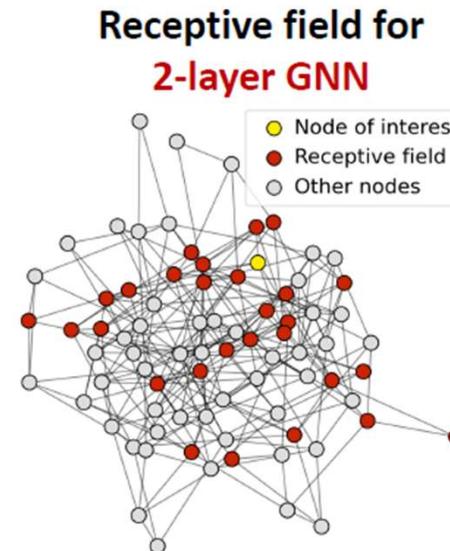
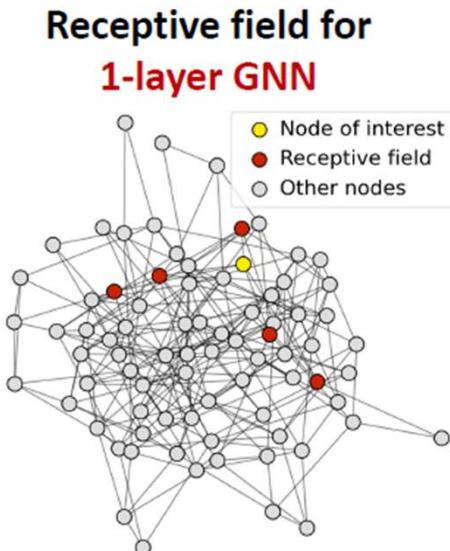


The Over-smoothing Problem

- **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

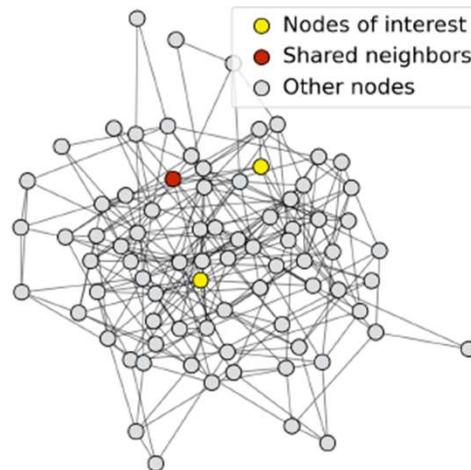
- **Receptive field:** the set of nodes that determine the embedding of a node of interest
- In a K -layer GNN, each node has a receptive field of K -hop neighborhood



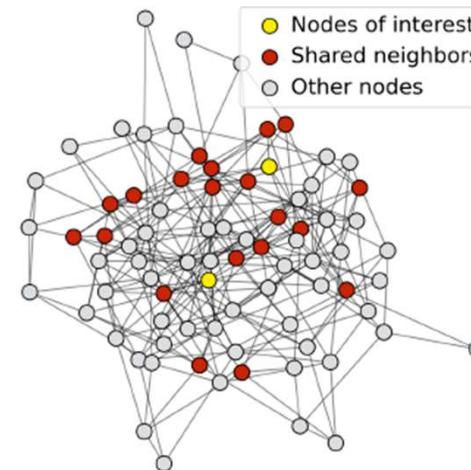
Receptive Field of a GNN

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

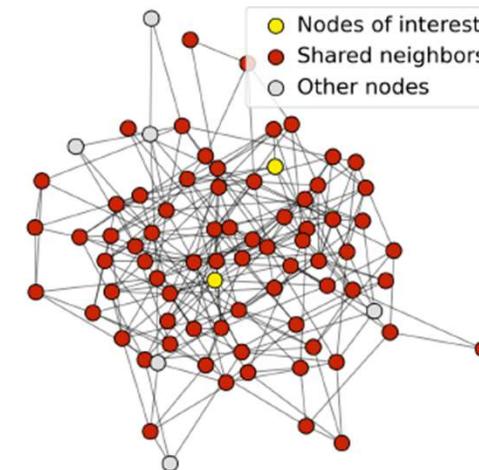
1-hop neighbor overlap
Only 1 node



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?

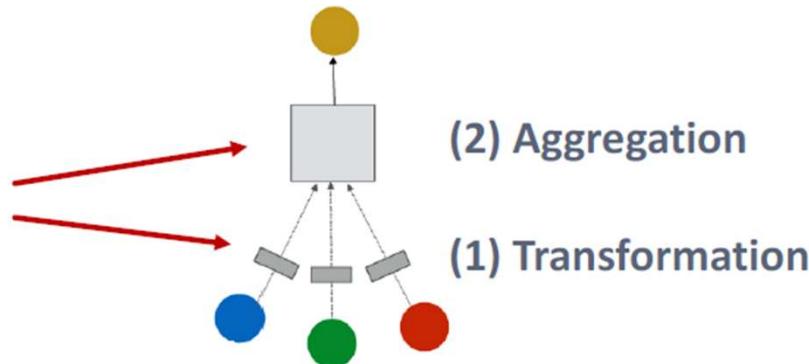
Designing GNN Layer Connectivity

- **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

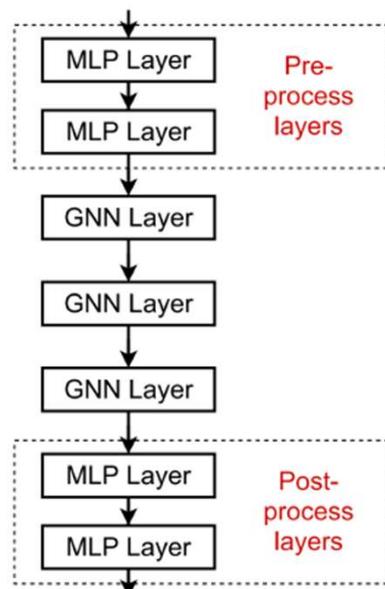
- How to make a shallow GNN more expressive?
- Solution 1: Increase the expressive power **within each GNN layer**
 - In our previous examples, each transformation or aggregation function only include one linear layer
 - We can **make aggregation / transformation become a deep neural network!**

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**



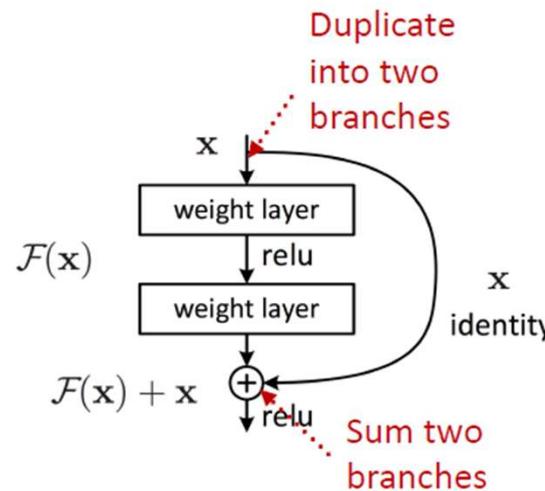
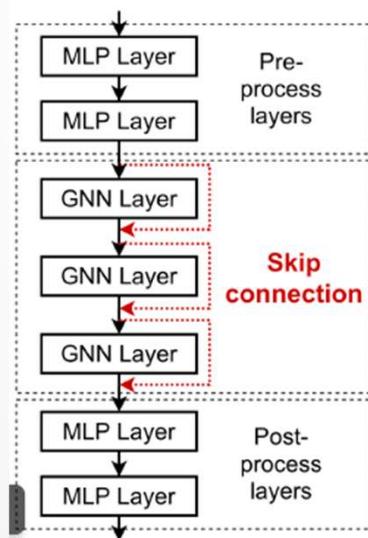
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

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



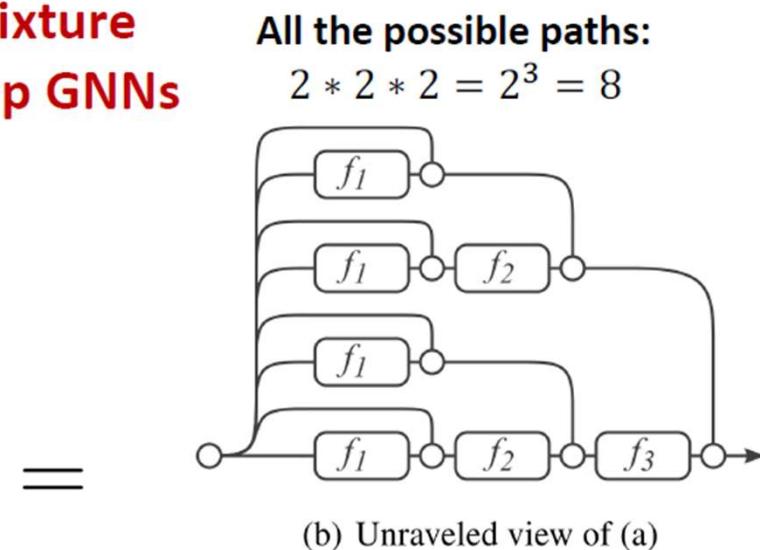
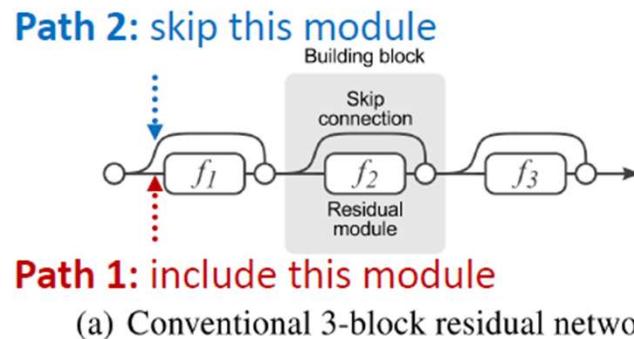
Idea of skip connections:
Before adding shortcuts:
 $\mathcal{F}(x)$
After adding shortcuts:
 $\mathcal{F}(x) + x$

Jure Leskovec, Stanford CS221W: Machine Learning with Graphs. <http://cs221w.stanford.edu>

He et al. Deep Residual Learning for Image Recognition,
CVPR 2015

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**



Veit et al. [Residual Networks Behave Like Ensembles of Relatively Shallow Networks](#), ArXiv 2016

Example: GCN with Skip Connections

- A standard GCN layer

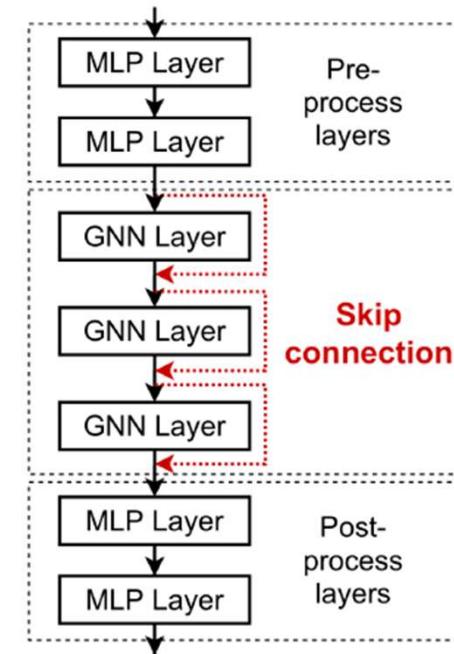
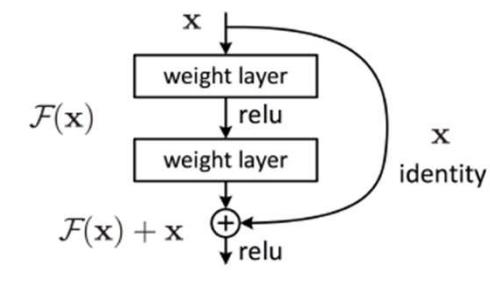
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

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

- A GCN layer with skip connection

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

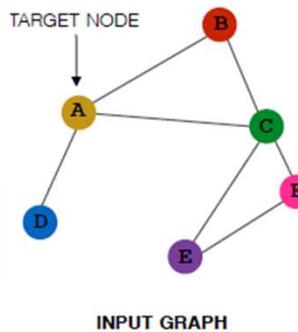
$F(\mathbf{x})$ + \mathbf{x}



Overview

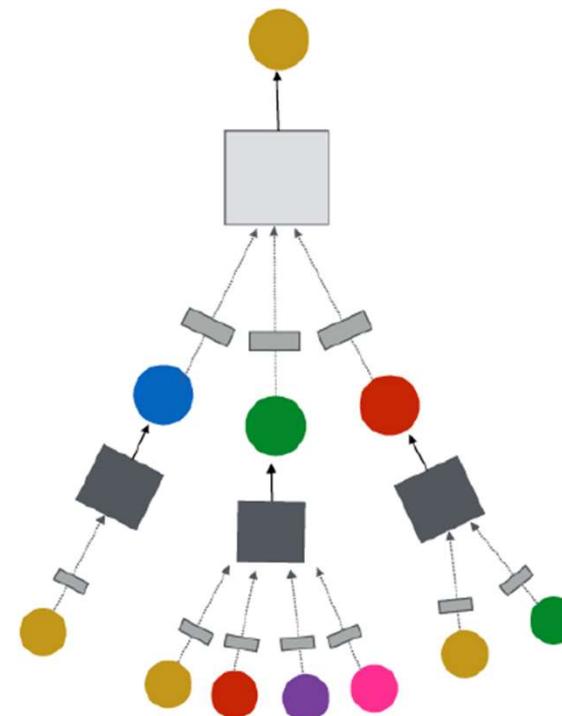
- GNN – General Framework
- A Single Layer of GNN
- Stacking Layers of a GNN
- Graph Augmentation for GNNs

General GNN Framework



Idea: Raw input graph \neq 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

Why Augment 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

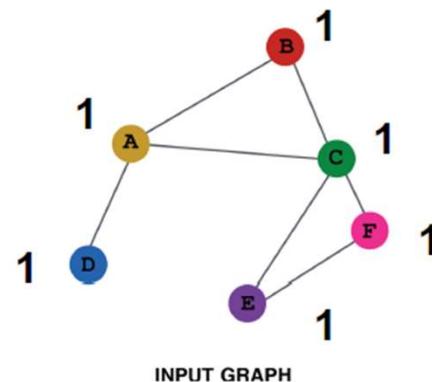
Graph Augmentation Approaches

- **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**

Feature Augmentation on Graphs

Why do we need feature augmentation?

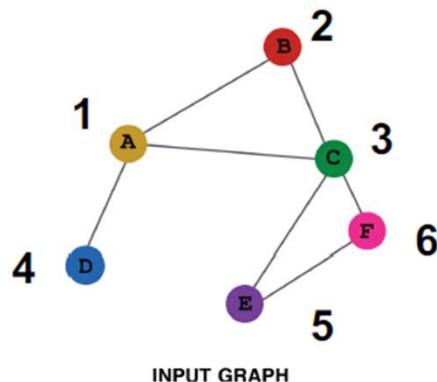
- (1) Input graph does not have node features
 - This is common when we only have the adj. matrix
- Standard approaches:
- a) Assign constant values to nodes



Feature Augmentation on Graphs

Why do we need feature augmentation?

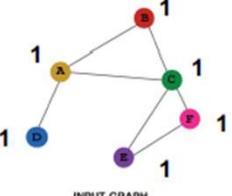
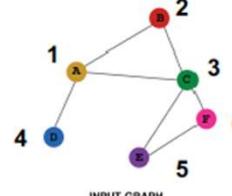
- **(1) Input graph does not have node features**
 - This is common when we only have the adj. matrix
- **Standard approaches:**
- **b) Assign unique IDs to nodes**
 - These IDs are converted into **one-hot vectors**



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**

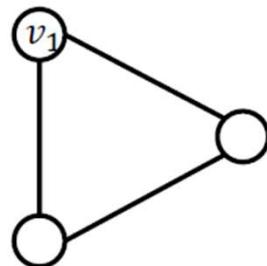
	Constant node feature 	One-hot node feature 
Expressive power	Medium. All the nodes are identical, but GNN can still learn from the graph structure	High. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	High. Simple to generalize to 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

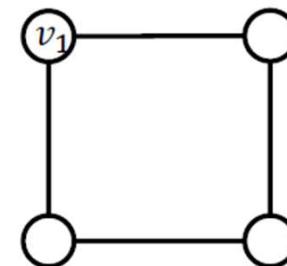
Why do we need feature augmentation?

- (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



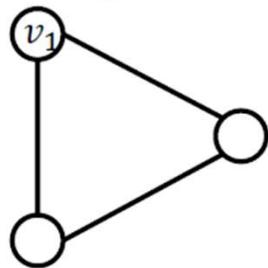
v_1 resides in a cycle with length 4



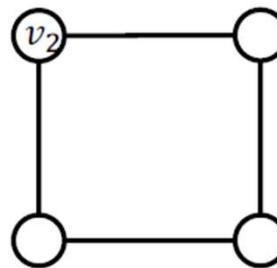
Feature Augmentation on Graphs

- 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

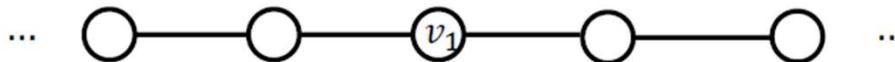
v_1 resides in a cycle
with length 3



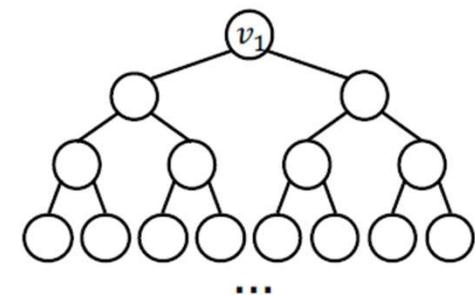
v_1 resides in a cycle
with length 4



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

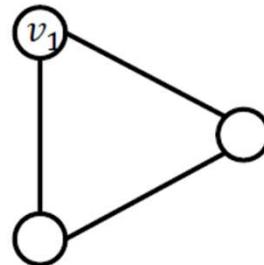
We start
from cycle
with length 0

Augmented node feature for v_1

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



v_1 resides in a cycle with length 3

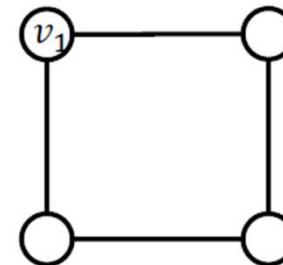


Augmented node feature for v_1

[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?

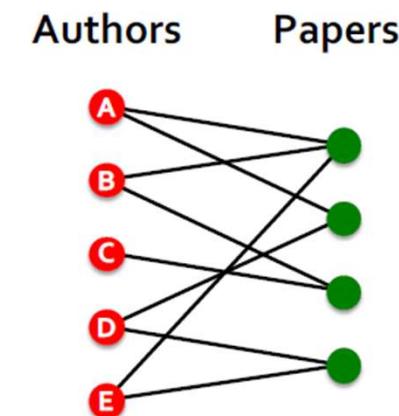
- (2) Certain structures are hard to learn by GNN
- Other commonly used augmented features:
 - Clustering coefficient
 - PageRank
 - Centrality
 - ...

Any features we have studied in this class can be used

Add Virtual Nodes/Edges

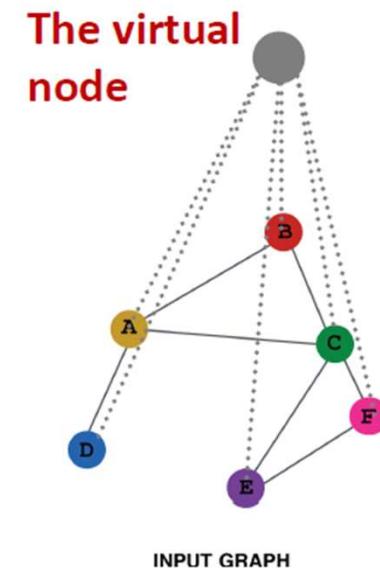
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
 - **Common approach:** Connect 2-hop neighbors via virtual edges
 - **Intuition:** Instead of using adj. matrix A for GNN computation, use $A + A^2$

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



Add Virtual Nodes/Edges

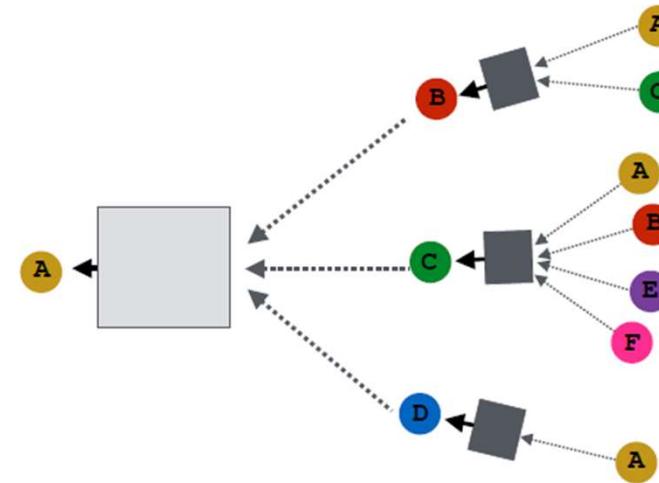
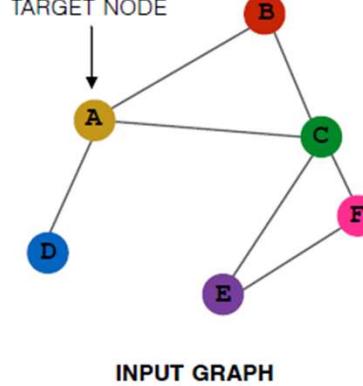
- **Motivation:** Augment sparse graphs
- **(2) Add virtual nodes**
 - The virtual node will connect to all the nodes in the graph
 - Suppose in a sparse graph, two nodes have shortest path distance of 10
 - After adding the virtual node, **all the nodes will have a distance of 2**
 - Node A – Virtual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



Node Neighborhood Sampling

- Previously:

- All the nodes are used for message passing

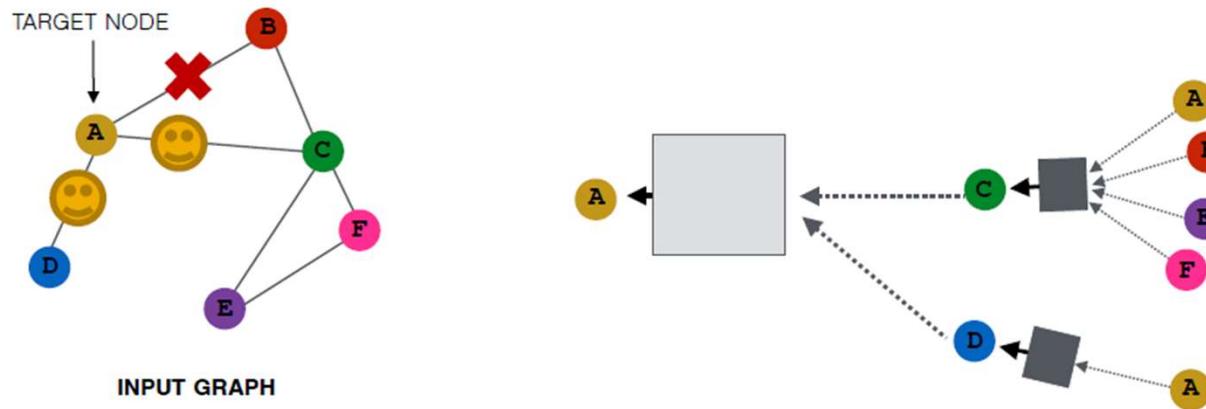


- New idea: (Randomly) sample a node's neighborhood for message passing

Hamilton et al. Inductive Representation Learning on Large Graphs,
NeurIPS 2017

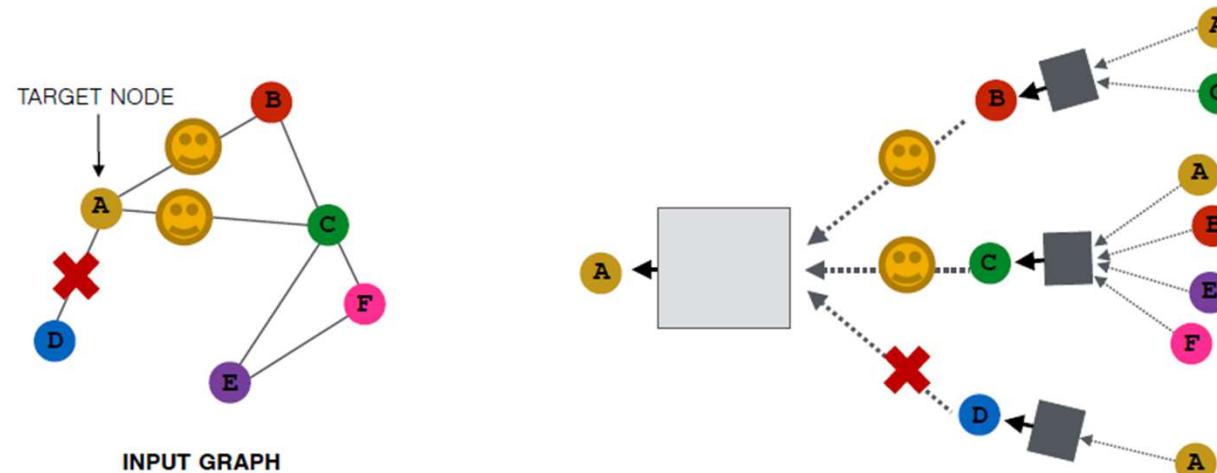
Neighborhood Sampling Example

- 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!



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

References

- Social Network Analysis Tanmoy Chakraborty. Chapter 9