Graph Neural Networks Designs

CPSC483: Deep Learning on Graph-Structured Data

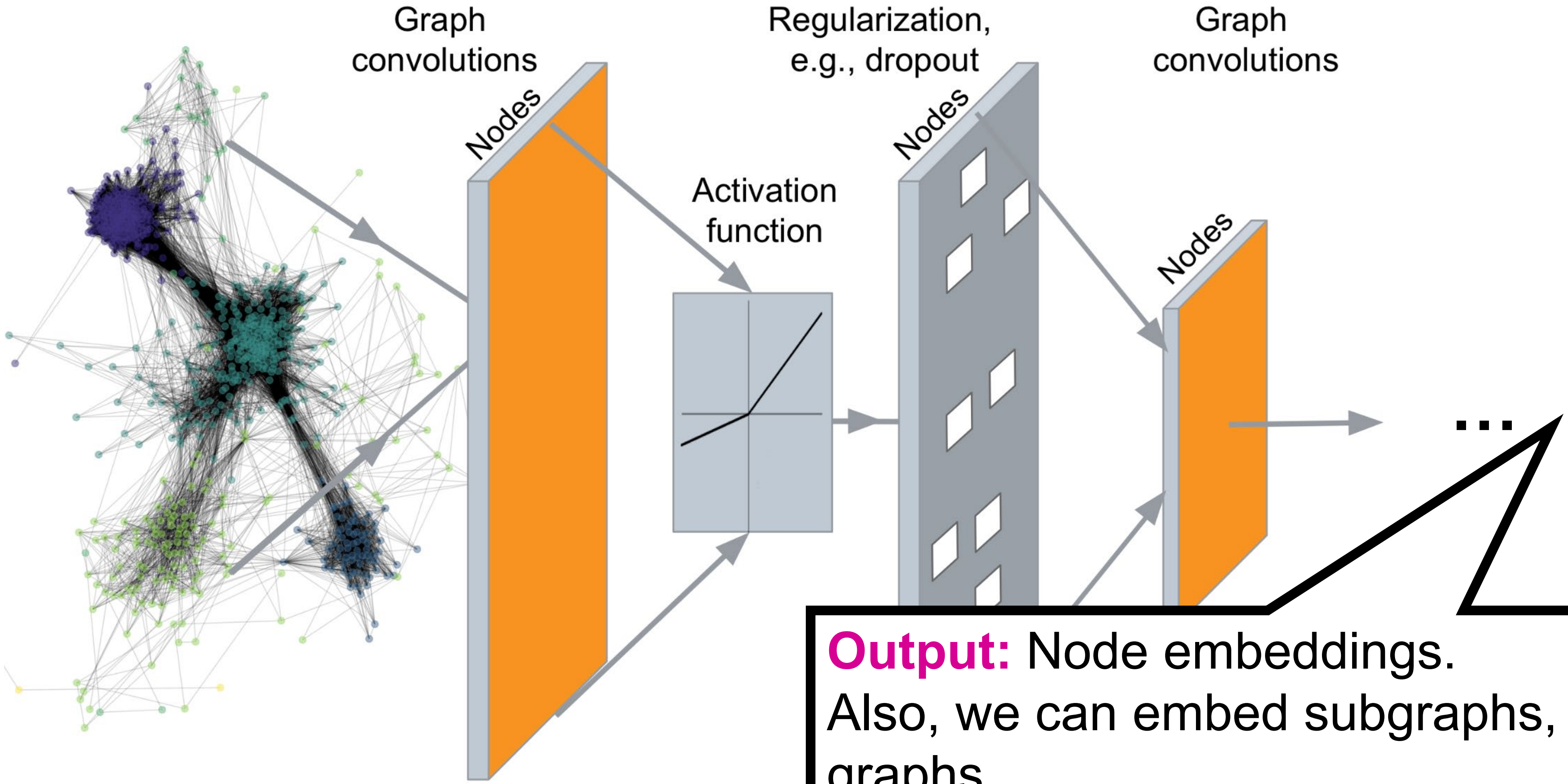
Rex Ying

Yale

Readings

- Readings are updated on the website (syllabus page)
- **Lecture 3 readings**:
  - [Graph representation learning: methods and application]
  - [Inductive Representation Learning for Large Graphs (GraphSAGE)]
- **Lecture 4 readings**:
  - [Semi-Supervised Classification with Graph Convolutional Networks]
  - [Principled Neighborhood Aggregation on Graph Nets]

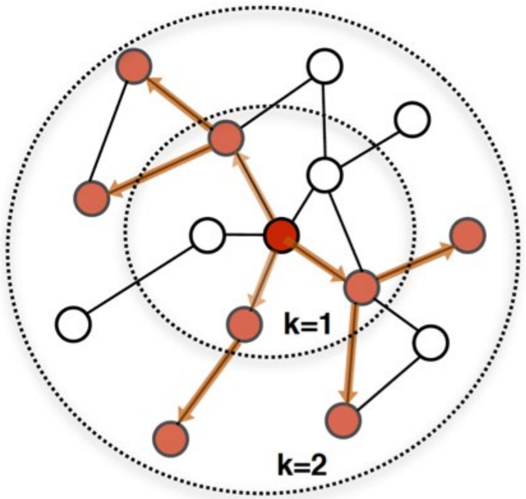
Recap: Deep Graph Encoders



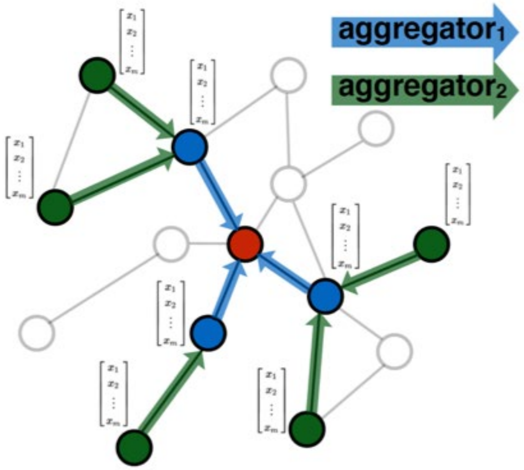
Output: Node embeddings.
Also, we can embed subgraphs, graphs

Recap: Graph Convolutional Networks

- Idea: Node's neighborhood defines a computation graph



Determine node computation graph

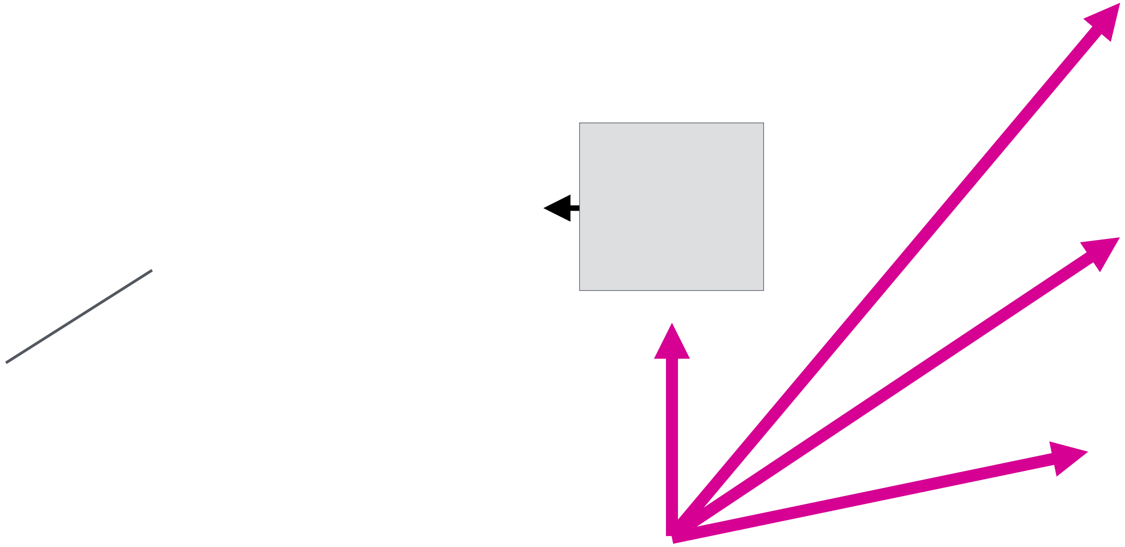


Propagate and transform information

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

Recap: Aggregate Neighbors (1)

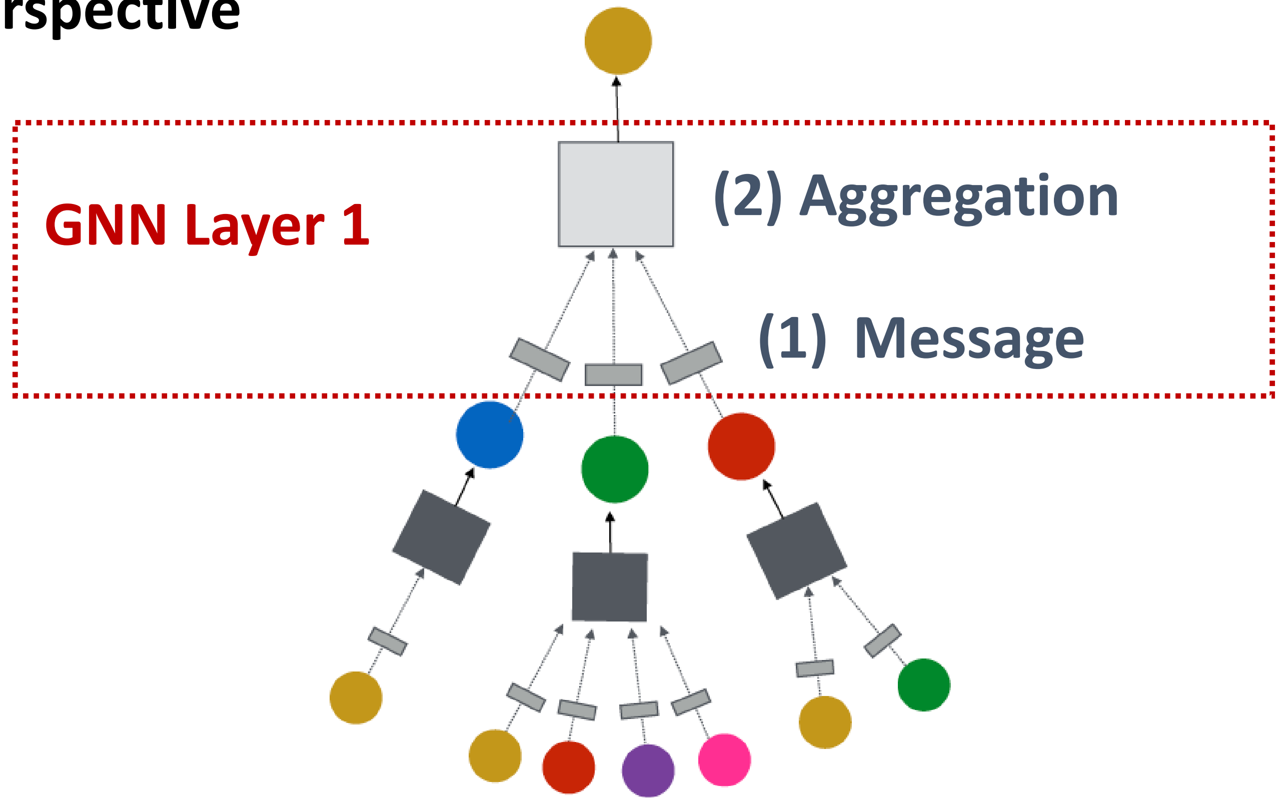
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



Neural networks

A General GNN Framework (1)

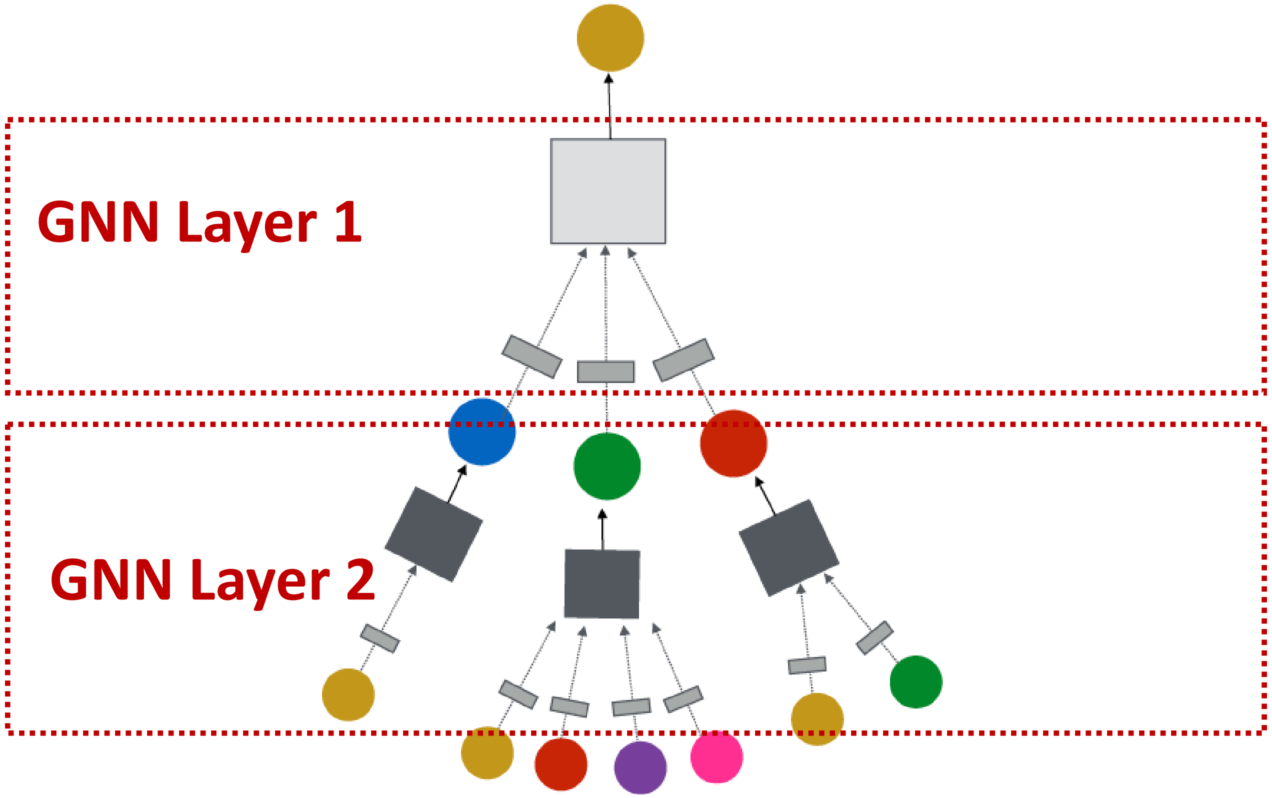
- **GNN Layer = Message + Aggregation**
  - **Different instantiations under this perspective**
  - **GCN, GraphSAGE, GAT, …**



A General GNN Framework (2)

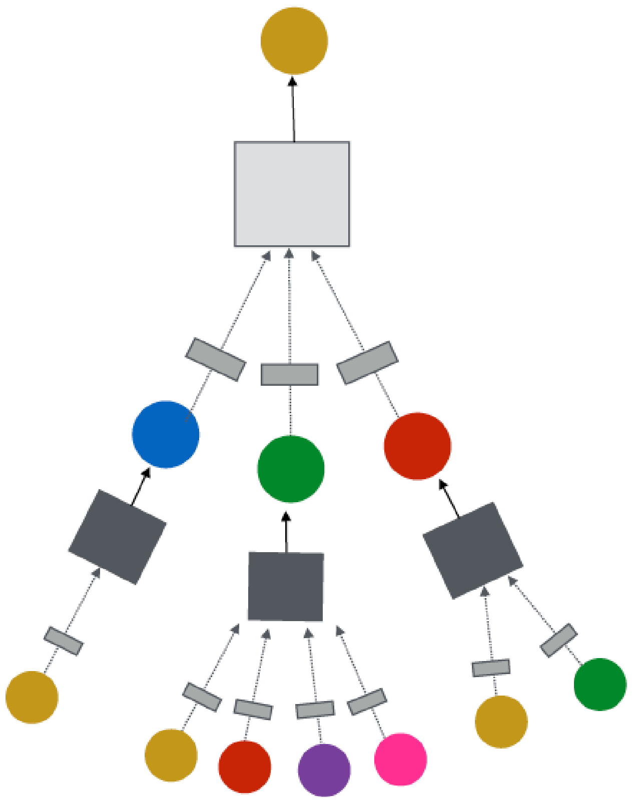
- **Connect GNN layers into a GNN**
  - **Stack layers sequentially**
  - **Ways of adding skip connections**

(3) Layer connectivity



A General GNN Framework (3)

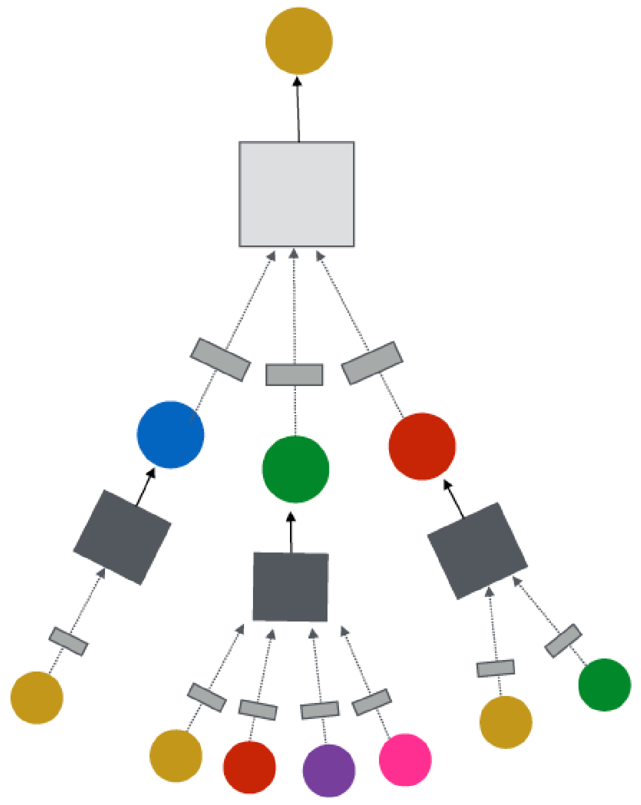
- **Idea: Raw input graph $\neq$ computational graph**
  - **Graph feature augmentation**
  - **Graph structure augmentation**



(4) Graph augmentation

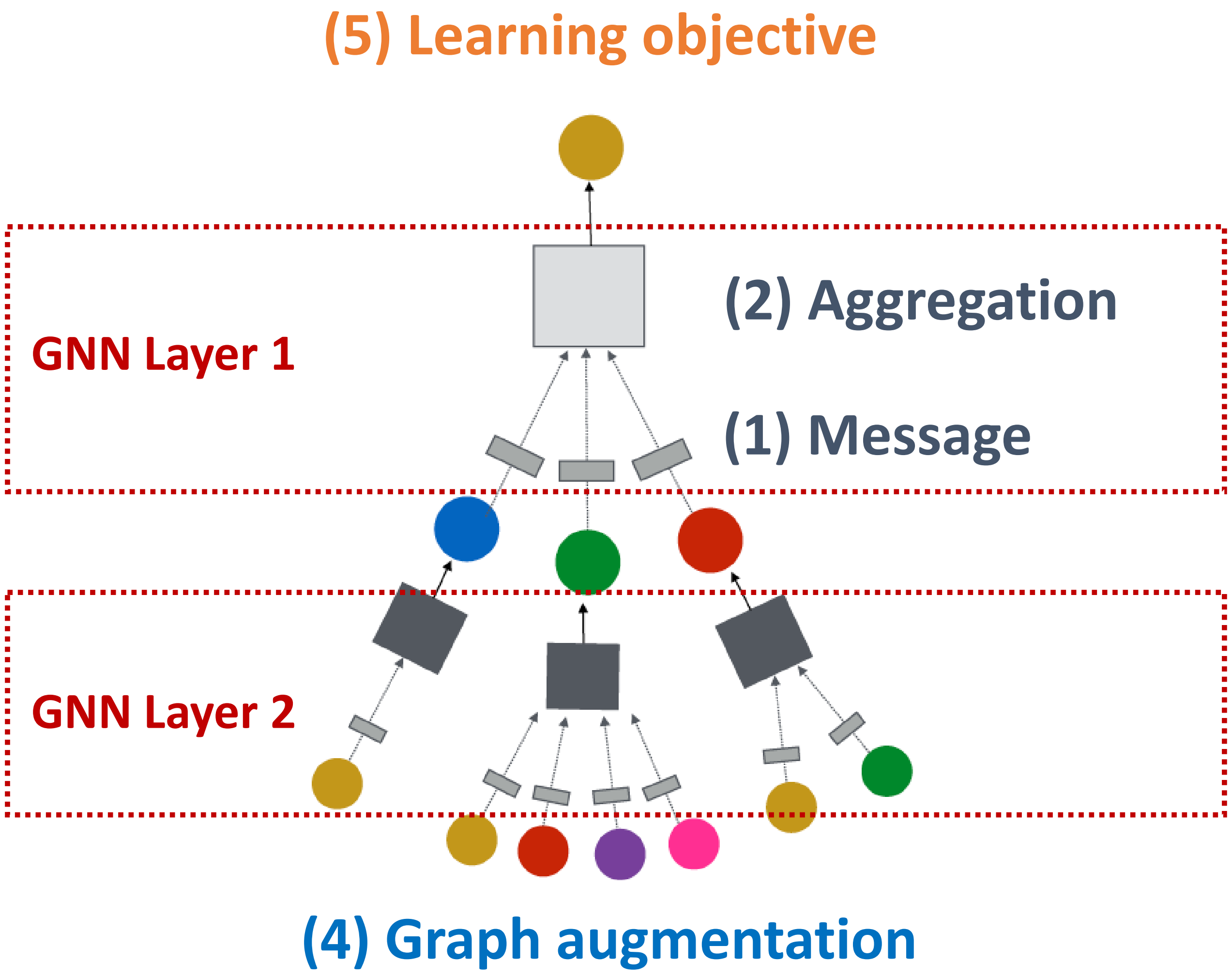
A General GNN Framework (4)

- **How do we train a GNN**
  - **Supervised/Unsupervised objectives**
  - **Node/Edge/Graph level objectives**



(4) Graph augmentation

A General GNN Framework (5)



Content

- **A Single Layer of a GNN**
- **Stacking Layers of a GNN**
- **Graph Manipulation in GNNs**

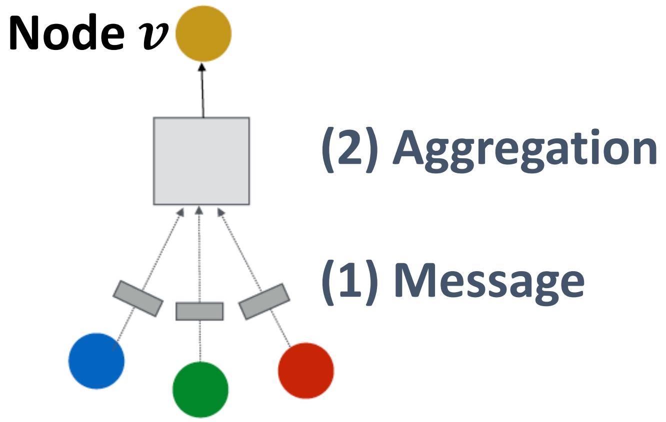
Outline of Today’s Lecture

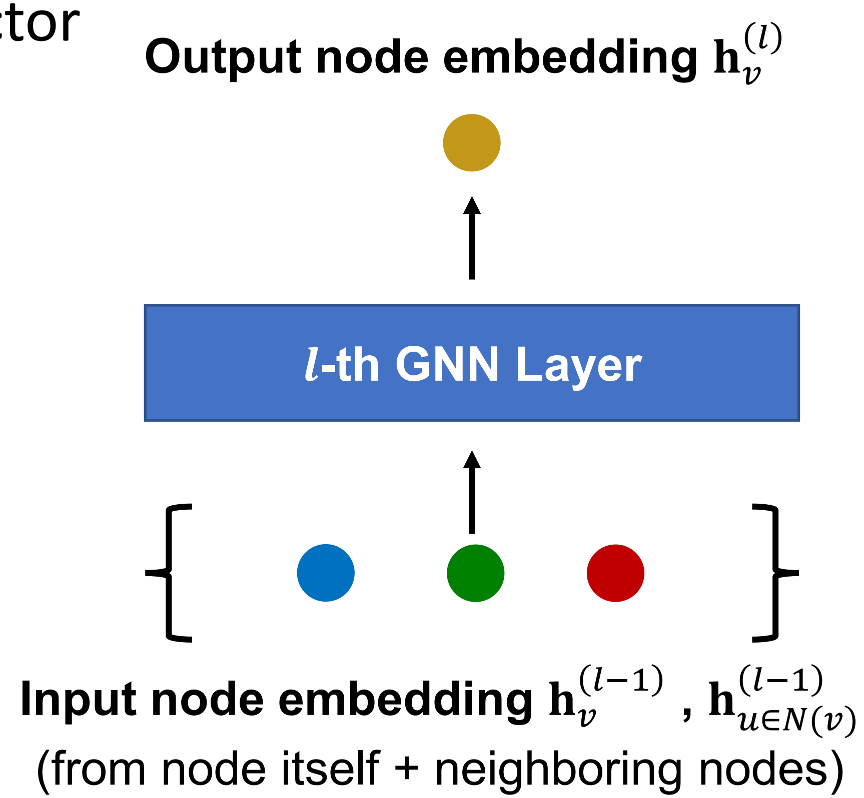
- **A Single Layer of a GNN**
- Stacking Layers of a GNN
- Graph Manipulation in GNNs

A Single layer of a GNN

A Single GNN Layer: Two Steps

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

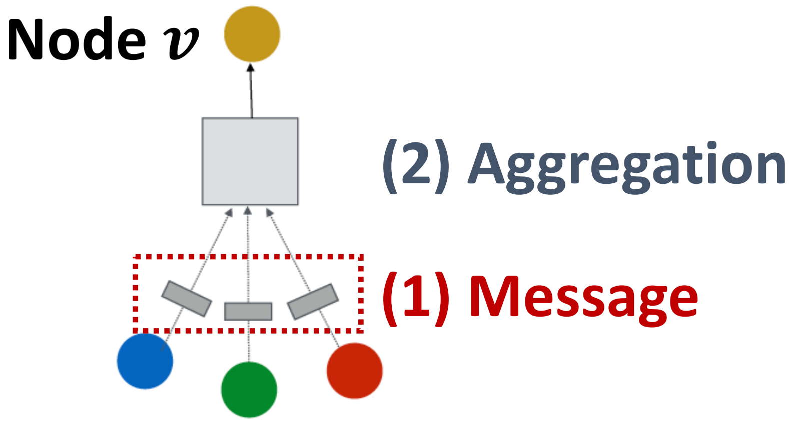




Message Computation

- **(1) Message computation**

  Message function
  - **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 $\mathbf{W}^{(l)}$



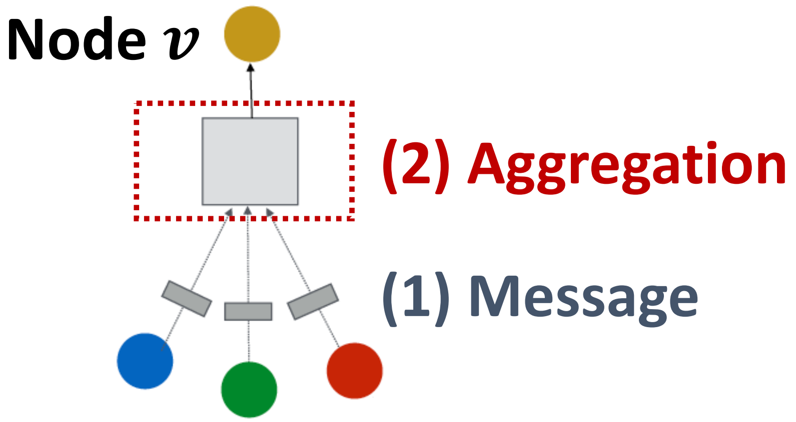
Message Aggregation

- **(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

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



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

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

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

$$\mathbf{h}_v^{(l)} = \mathrm{CONCAT}\left(\underbrace{\mathrm{AGG}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)}_{\text{First aggregate from neighbors}}, \overbrace{\mathbf{m}_v^{(l)}}^{\text{Then aggregate from node itself}}\right)$$

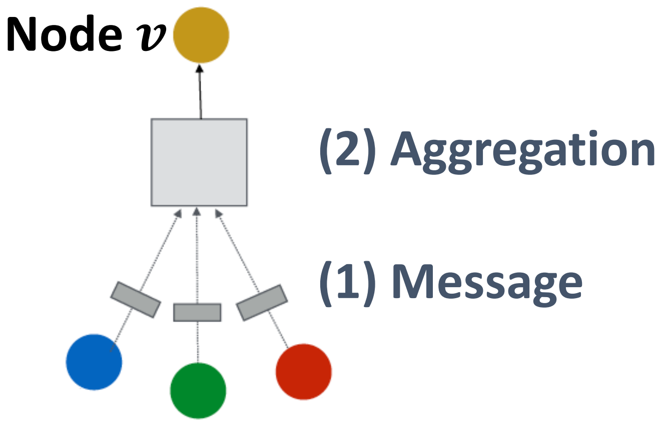
A Single GNN Layer: Message and Aggregation

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

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

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

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



  - **Nonlinearity (activation):** Adds expressiveness
    - Often written as $\sigma(\cdot)$: $\mathrm{ReLU}(\cdot)$, $\mathrm{Sigmoid}(\cdot)$, …
    - Can be added to **message or aggregation**

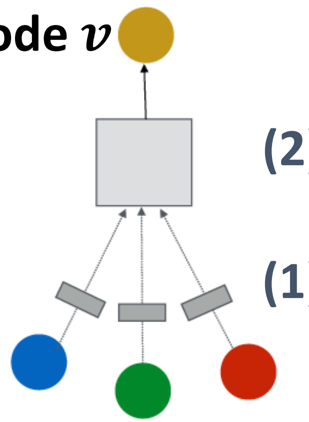
Classical GNN Layers: GCN (1)

- **(1) Graph Convolutional Networks (GCN)**

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

- **How to write this as Message + Aggregation?**

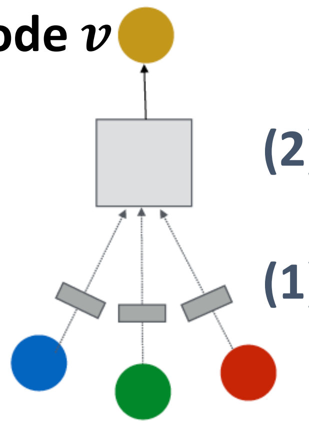
$$\mathbf{h}_v^{(l)} = \sigma\left(\underbrace{\sum_{u \in N(v)} \mathbf{W}^{(l)}}_{\text{Aggregation}} \overbrace{\frac{\mathbf{h}_u^{(l-1)}}{\sqrt{|N(u)||N(v)|}}}^{\text{Message}}\right)$$



Classical GNN Layers: GCN (2)

- **Graph Convolutional Networks (GCN)**

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



- **Message:**
  - Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{\sqrt{|N(u)||N(v)|}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}$
- **Aggregation:**
  - **Sum** over messages from neighbors, then apply activation
  - $\mathbf{h}_v^{(l)} = \sigma\left(\mathrm{Sum}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)\right)$

Classical GNN Layers: GraphSAGE

- **GraphSAGE**

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

- **How to write this as Message + Aggregation?**
  - **Message** is computed within the $\mathrm{AGG}(\cdot)$
  - **Two-stage aggregation**
    - **Stage 1:** Aggregate from node neighbors

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

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

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

GraphSAGE Neighborhood Aggregation

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

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

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

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

- **LSTM:** Apply LSTM to reshuffled of neighbors (not order invariant)

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

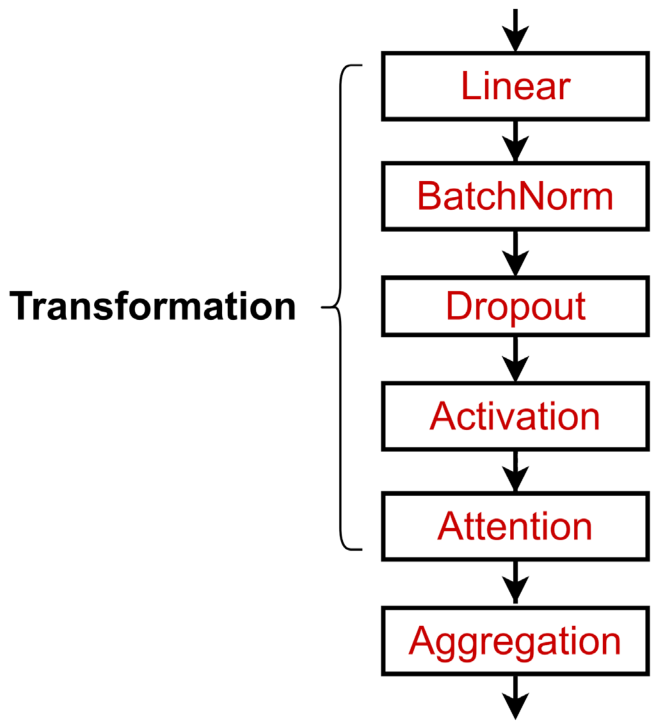
GraphSAGE: L2 Normalization

- ℓ_2 **Normalization:**
  - **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
  - $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\left\|\mathbf{h}_v^{(l)}\right\|_2} \ \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2-norm)
  - Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement
  - **Benefit**: more efficient retrieval (finding the nearest embedding vector) through locality sensitive hashing (**LSH**)

GNN Layer in Practice (1)

- **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

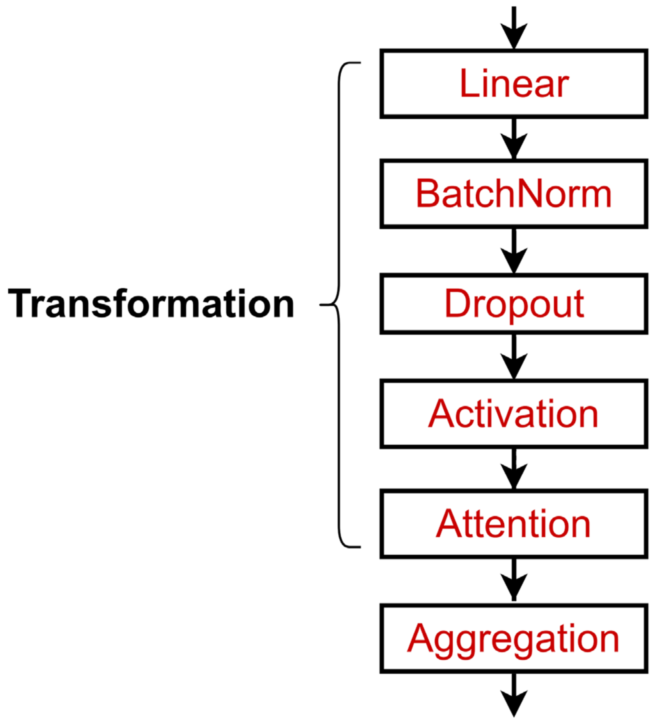
A suggested GNN Layer



GNN Layer in Practice Overview

- **Overview: many modern deep learning modules can be incorporated into a GNN layer**
  - **Batch Normalization**
    - Normalize embeddings in a minibatch
    - Stabilize neural network training
  - **Layer Normalization (alternative to BN)**
    - Normalize individual output embeddings
  - **Dropout**
    - Prevent overfitting
  - **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:
$\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{D}$

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

Step 1:
Compute the mean and variance over N embeddings

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

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

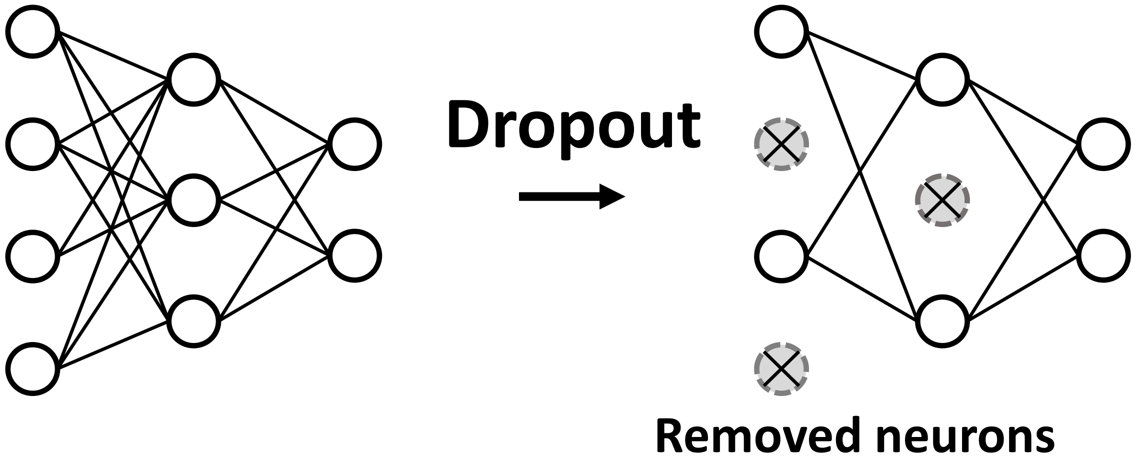
Step 2:
Normalize the feature using computed mean and variance

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

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

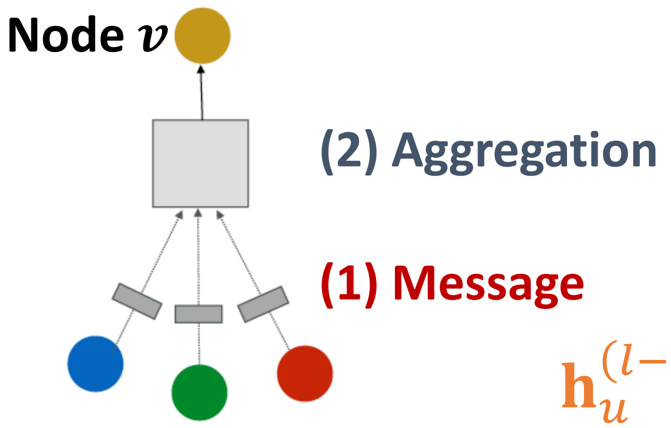
Dropout

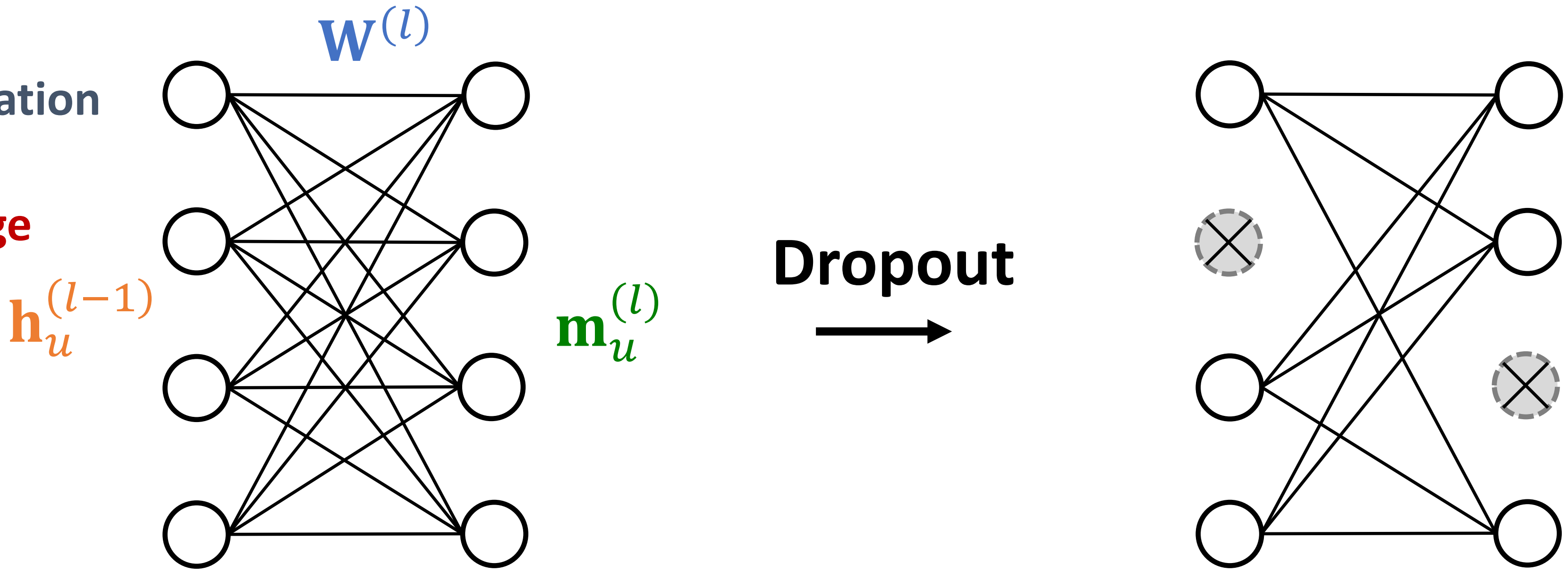
- **Goal**: Regularize a neural net to prevent overfitting.
  - Dropout ratio: percentage of dimensions that are dropped out
- **Idea**:
  - **During training**: with some probability p, randomly set neurons to zero (turn off)
  - **During testing:** Use all the neurons for computation
  - In PyTorch, use model.train() and model.eval() to toggle



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

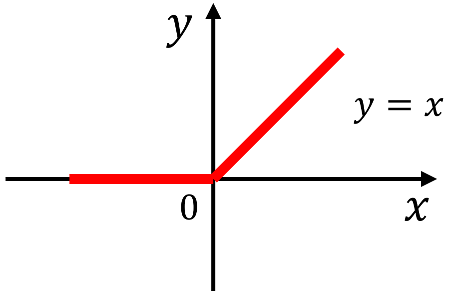
Activation (Non-linearity)

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

- **Rectified linear unit (ReLU)**

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

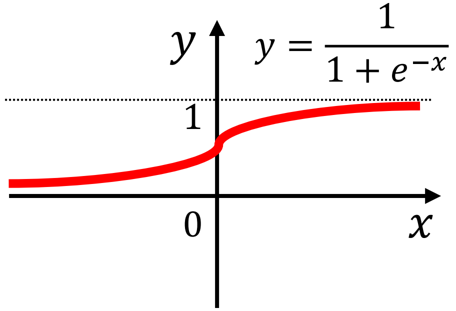
  - Commonly used



- **Sigmoid**

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

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

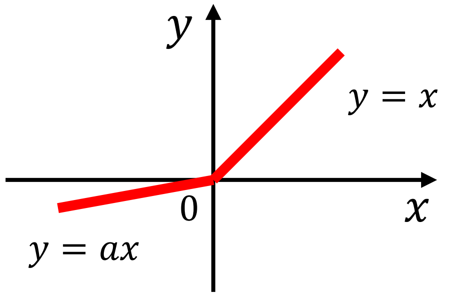


- **Parametric ReLU**

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

  a_i is a **trainable parameter**

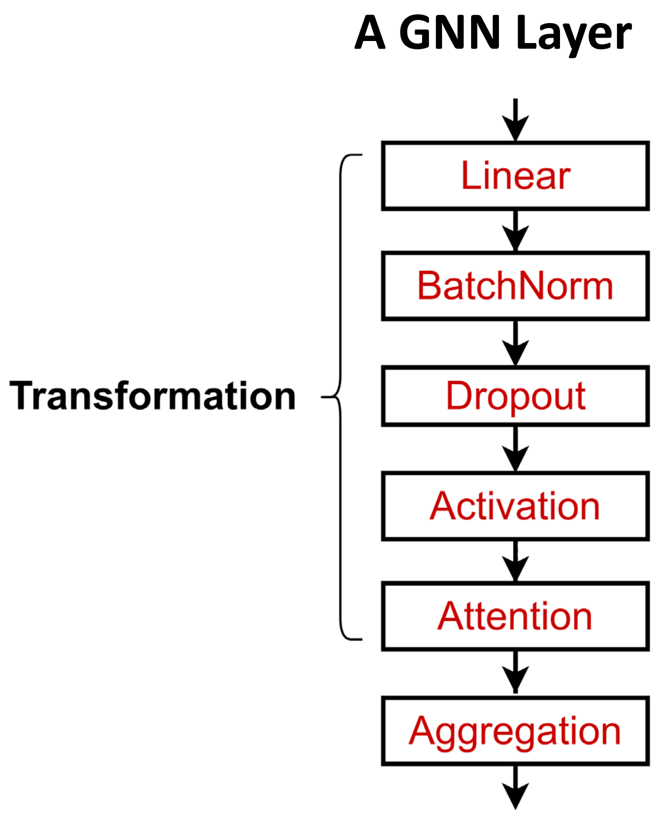
  - Sometimes performs better than ReLU



- See [Pytorch documentation](#) for more non-linearity functions

GNN Layer in Practice

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



Outline of Today's Lecture

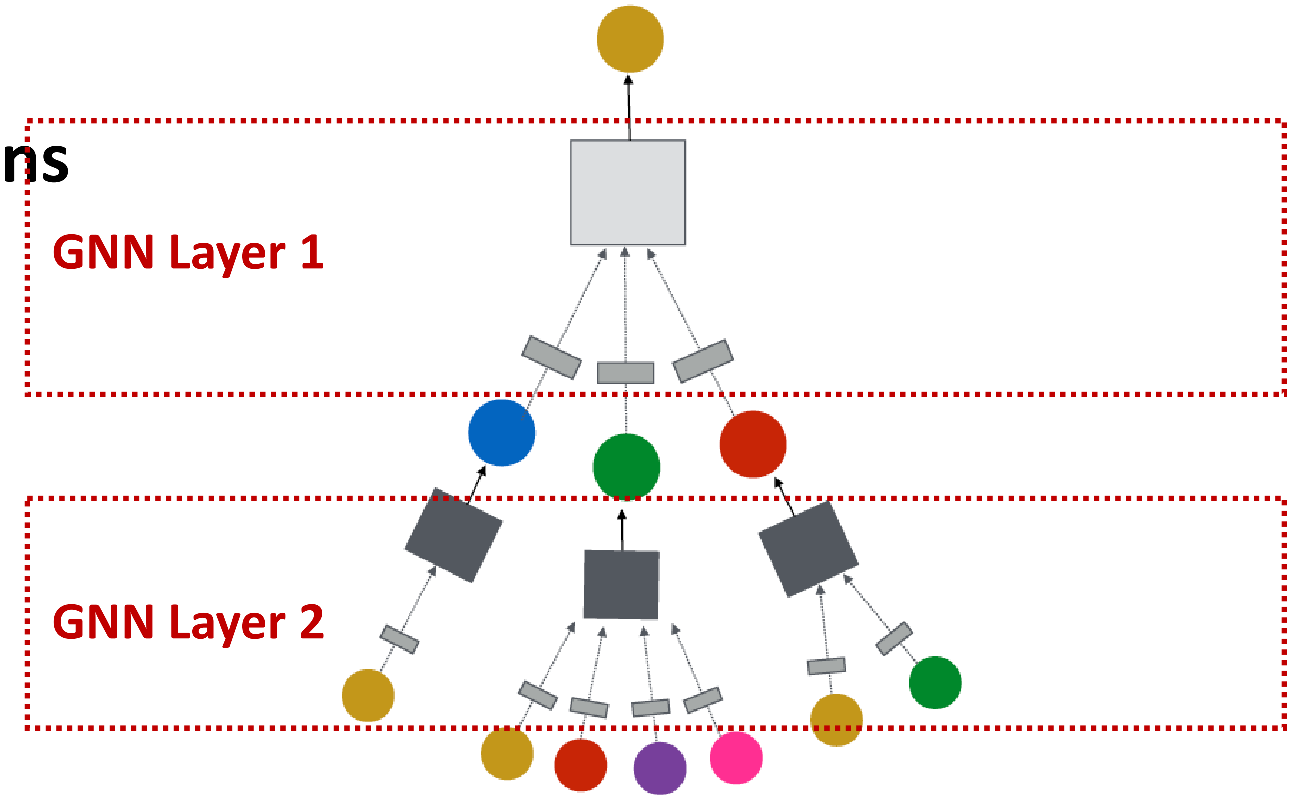
- A Single Layer of a GNN
- **Stacking Layers of a GNN**
- Graph Manipulation in GNNs

Stacking Layers of a GNN

Stacking GNN Layers (1)

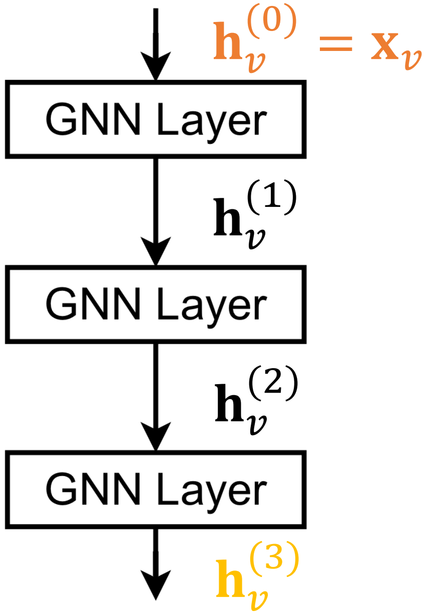
- **How to connect GNN layers into a GNN?**
- **Stack layers sequentially**
- **Ways of adding skip connections**

(3) Layer connectivity



Stacking GNN Layers (2)

- **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



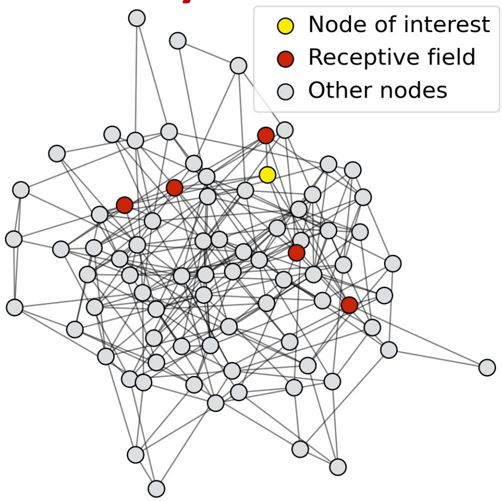
The Over-smoothing Problem

- **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?**

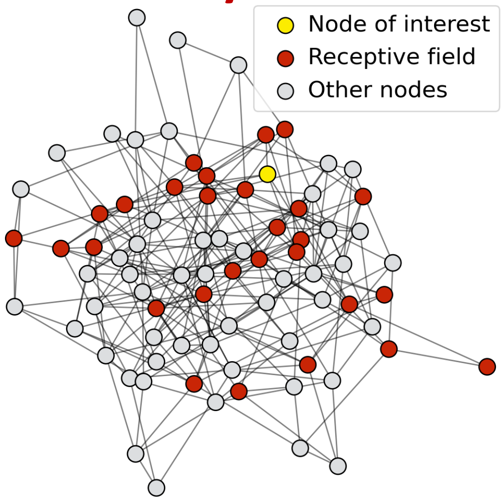
Receptive Field of a GNN (1)

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

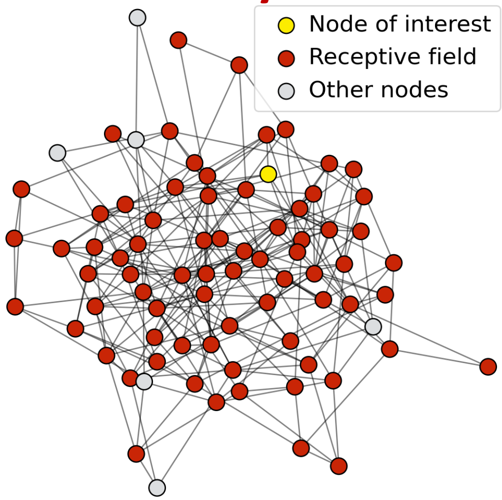
Receptive field for
1-layer GNN



Receptive field for
2-layer GNN



Receptive field for
3-layer GNN



Receptive Field of a GNN (2)

- **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

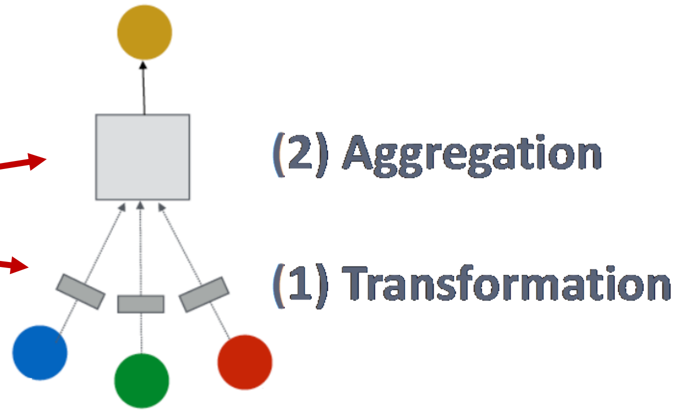
- **What do we learn from the over-smoothing problem?**
- **Lesson 1: Be cautious when adding GNN layers**
  - Unlike neural networks in other domains (CNN for image classification), **adding more GNN layers do not always help**
  - **Step 1: Analyze the necessary receptive field** to solve your problem. 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. Tune it as a hyper-parameter.

- **Question:** How to enhance the expressive power of a GNN, if the number of GNN layers is small?

Expressive Power for Shallow GNNs (1)

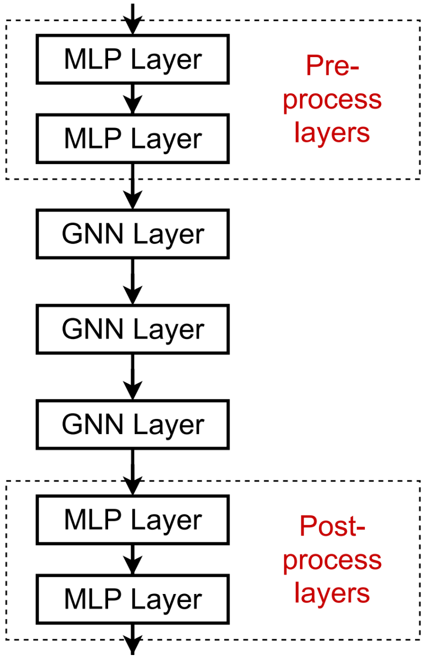
- **How to make GNN more expressive without more message passing layers?**
- **Solution 1:** Increase the expressive power **within each GNN layer**
  - In our previous examples, each transformation or aggregation function only include one linear layer
  - We can **make aggregation / transformation become a deep neural network**!

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



Expressive Power for Shallow GNNs (2)

- **How to make GNN more expressive without more message passing layers?**
- **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**



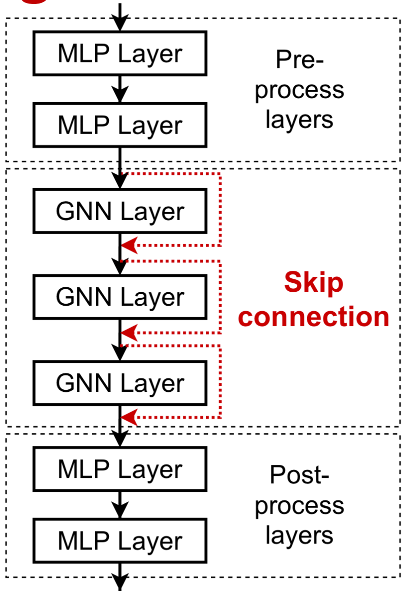
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

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

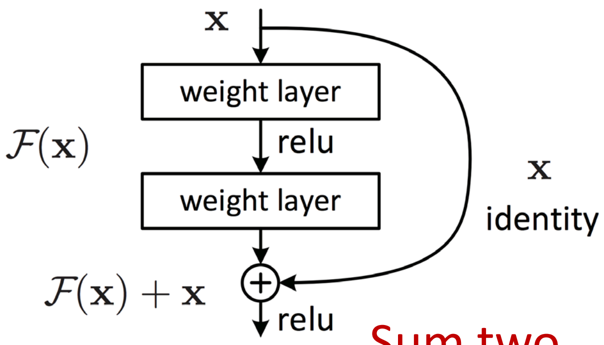
In practice, adding these layers works great!

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**



Duplicate into two branches



Sum two branches

Idea of skip connections:

Before adding shortcuts:

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

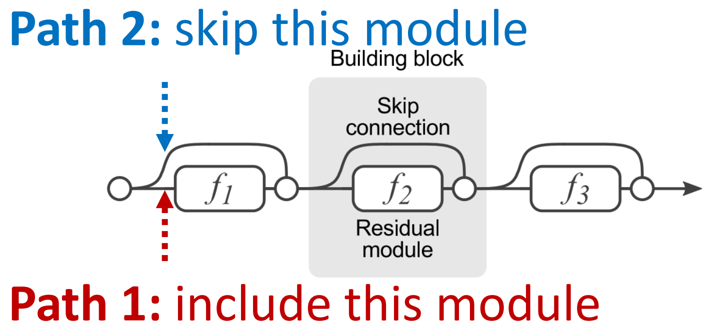
After adding shortcuts:

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

(also called **residual stream** in the context of Transformer)

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**

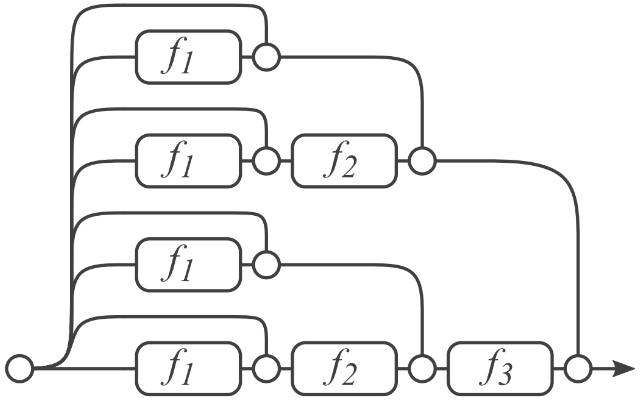


(a) Conventional 3-block residual network

=

All the possible paths:

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



(b) Unraveled view of (a)

Example: GCN with Skip Connections

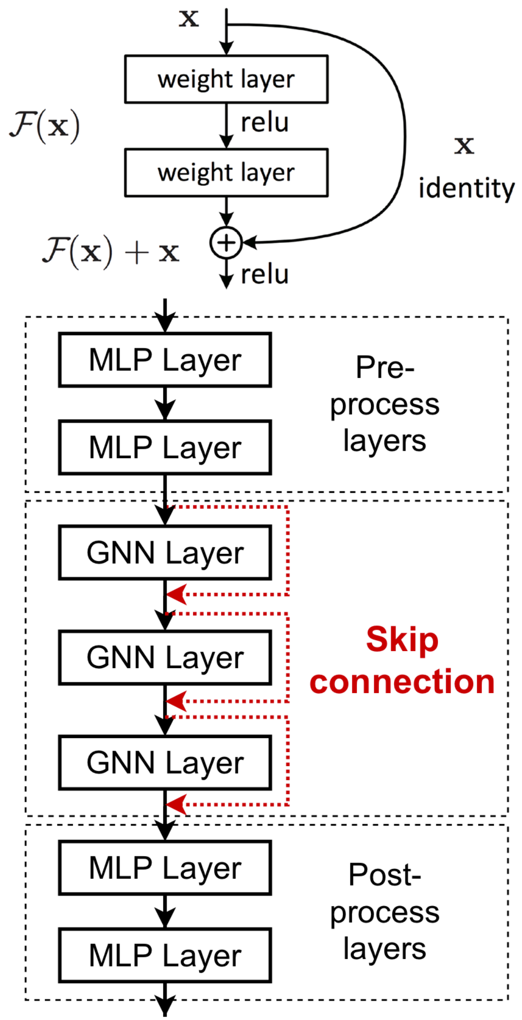
- **A standard GCN layer**

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

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

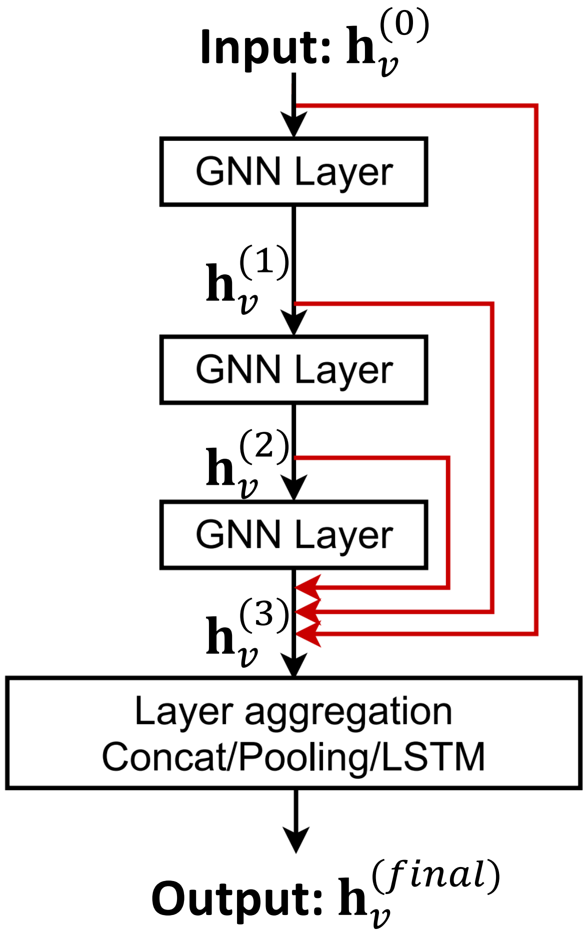
- **A GCN layer with skip connection**

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



Other Options of Skip Connections

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



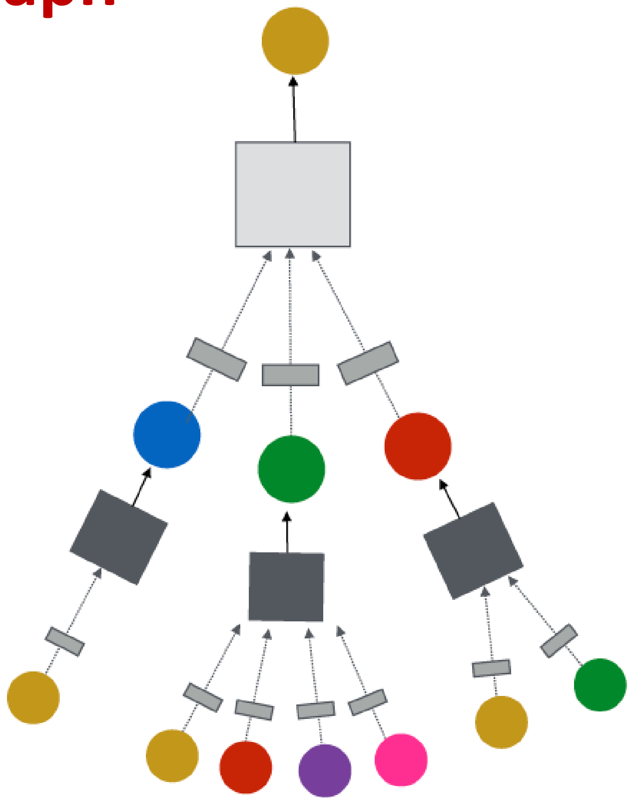
Outline of Today's Lecture

- A Single Layer of a GNN
- Stacking Layers of a GNN
- **Graph Manipulation in GNNs**

Graph Manipulation in GNNs

Graph GNN Framework

- **Idea: Raw input graph $\neq$ computational graph**
  - **Graph feature augmentation**
  - **Graph structure manipulation**



(4) Graph manipulation

Why Manipulating Graphs

Our assumption so far has been

- **Raw input graph = computational graph**
  - **Feature level**
    - The input graph **lacks features**
  - **Structure level:**
    - The graph is **too sparse** → insufficient 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 **unlikely that the input graph happens to be the optimal computation graph** for embeddings

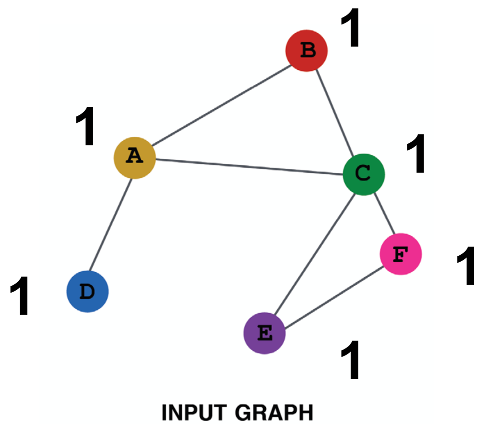
Graph Manipulation Approaches

- **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

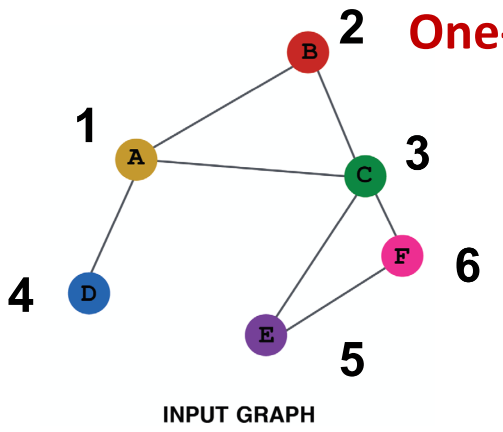
Feature Augmentation on Graphs (1)

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 (one hot)



One-hot vector for node with ID=5

ID = 5

$$\underbrace{[0, 0, 0, 0, 1, 0]}_{\text{Total number of IDs = 6}}$$

Feature Augmentation on Graphs (2)

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

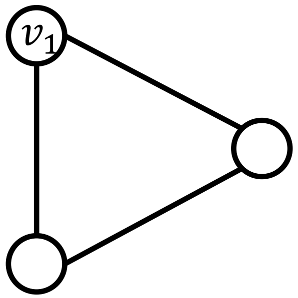
	Constant node feature	**One-hot node feature**
Expressive power	**Medium**. All the nodes are identical, but GNN can still learn from the graph structure	**High**. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High**. Simple to 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**. $O(\lvert V \rvert)$ 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 (3)

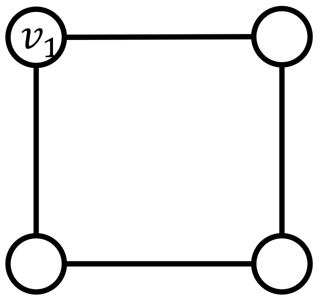
Why do we need feature augmentation?

- **(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** (we will talk more about it in Lecture 9)

v_1 resides in a cycle with length 3



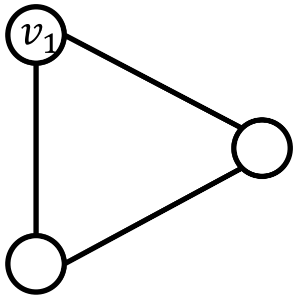
v_1 resides in a cycle with length 4



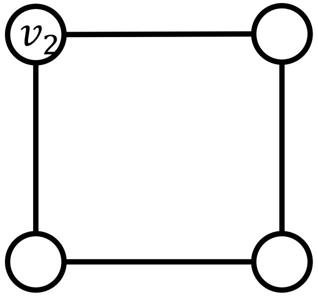
Feature Augmentation on Graphs (4)

- **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

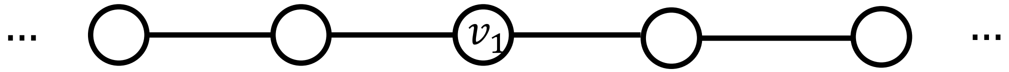
v_1 resides in **a cycle with length 3**



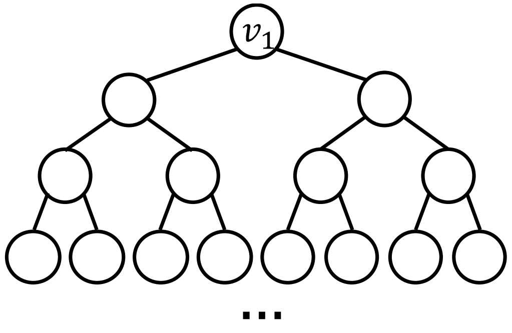
v_1 resides in **a cycle with length 4**



There exists **no cycle** that includes v_1



The computational graphs for node v_1 are always the same



Feature Augmentation on Graphs (5)

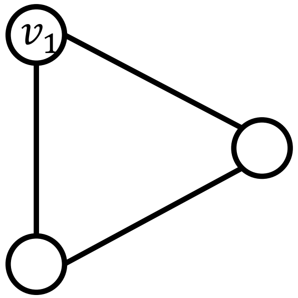
- **Solution:**
  - We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for v_1

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

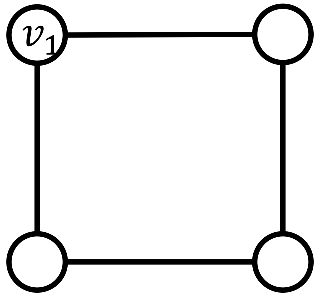
v_1 resides in **a cycle with length 3**



Augmented node feature for v_1

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

v_1 resides in **a cycle with length 4**

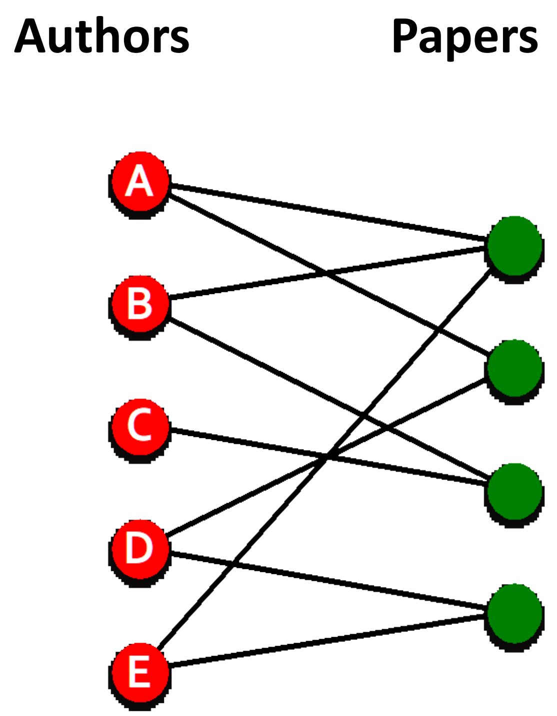


Feature Augmentation on Graphs (6)

- Other commonly used augmented features:
  - **Clustering coefficient**
  - **PageRank**
  - **Centrality**
  - **…**
- Any **feature we have introduced in Lecture 2 can be used!**

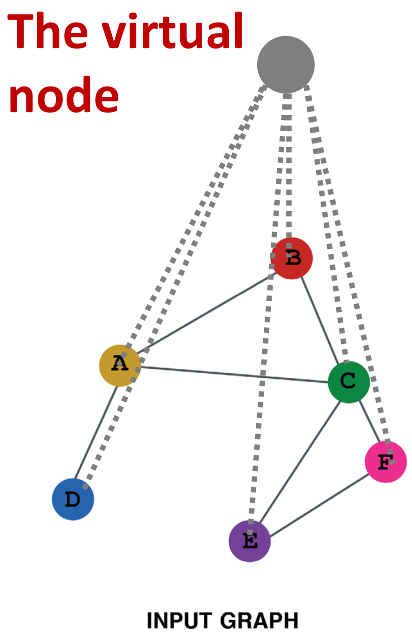
Add Virtual Nodes / Edges (1)

- **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 (2)

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



Node Neighborhood Sampling

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

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

Neighborhood Sampling Example (1)

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

Neighborhood Sampling Example (3)

- 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!
  - We will talk more about scalable GNNs in Lecture 6

Summary of the Lecture

- **Recap: A general perspective for GNNs**
  - **GNN Layer**:
    - Transformation + Aggregation
    - Classic GNN layers: GCN, GraphSAGE, GAT
  - **Layer connectivity**:
    - Deciding number of layers
    - Skip connections
  - **Graph Manipulation:**
    - Feature augmentation
    - Structure manipulation
- Having understood the basics of GNNs, we can now explore advanced architectures, tasks and applications!