

Note to other teachers and users of these slides: We would be delighted if you found our material useful for giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: <http://cs224w.Stanford.edu>

Stanford CS224W: A General Perspective on Graph Neural Networks

CS224W: Machine Learning with Graphs

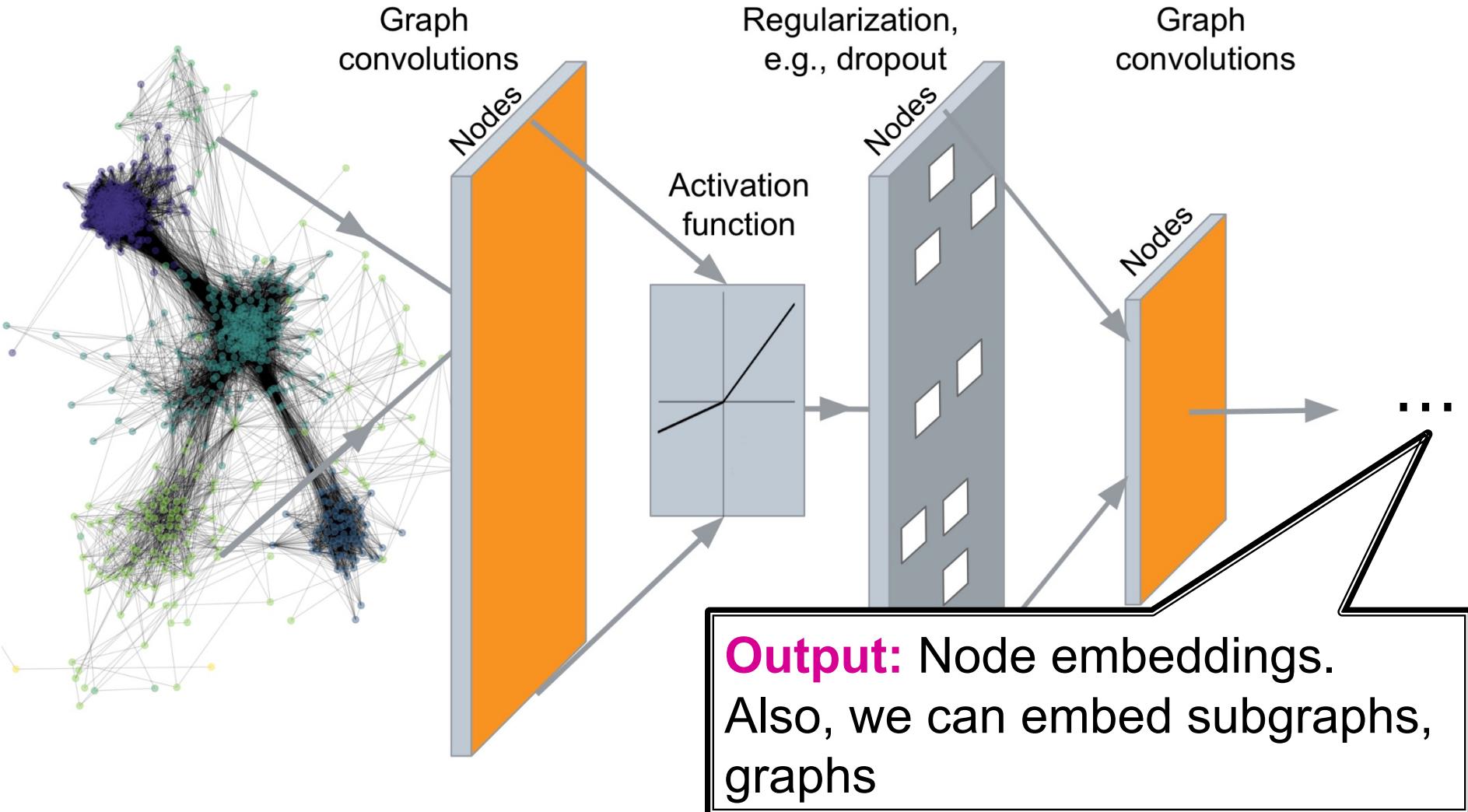
Jure Leskovec, Stanford University

Charilaos Kanatsoulis, Stanford University

<http://cs224w.stanford.edu>

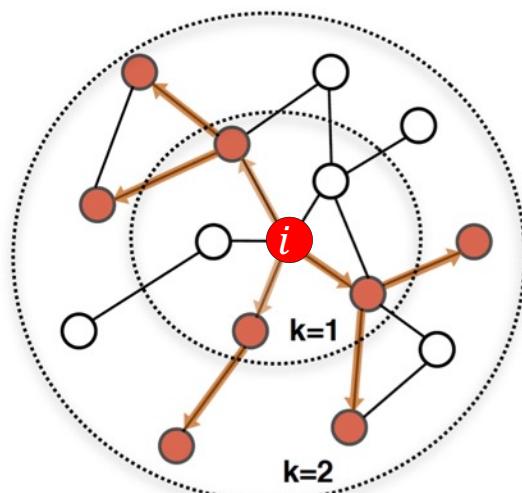


Recap: Deep Graph Encoders

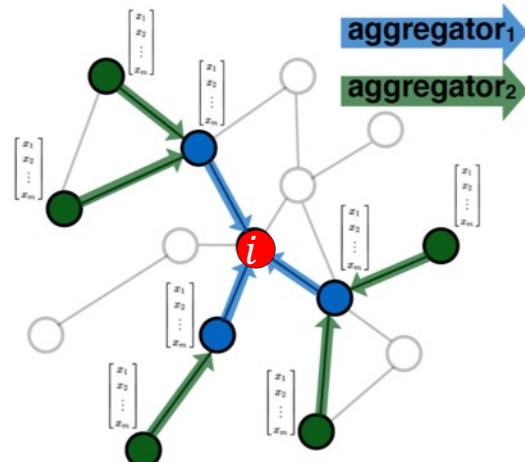


Recap: Graph Neural Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

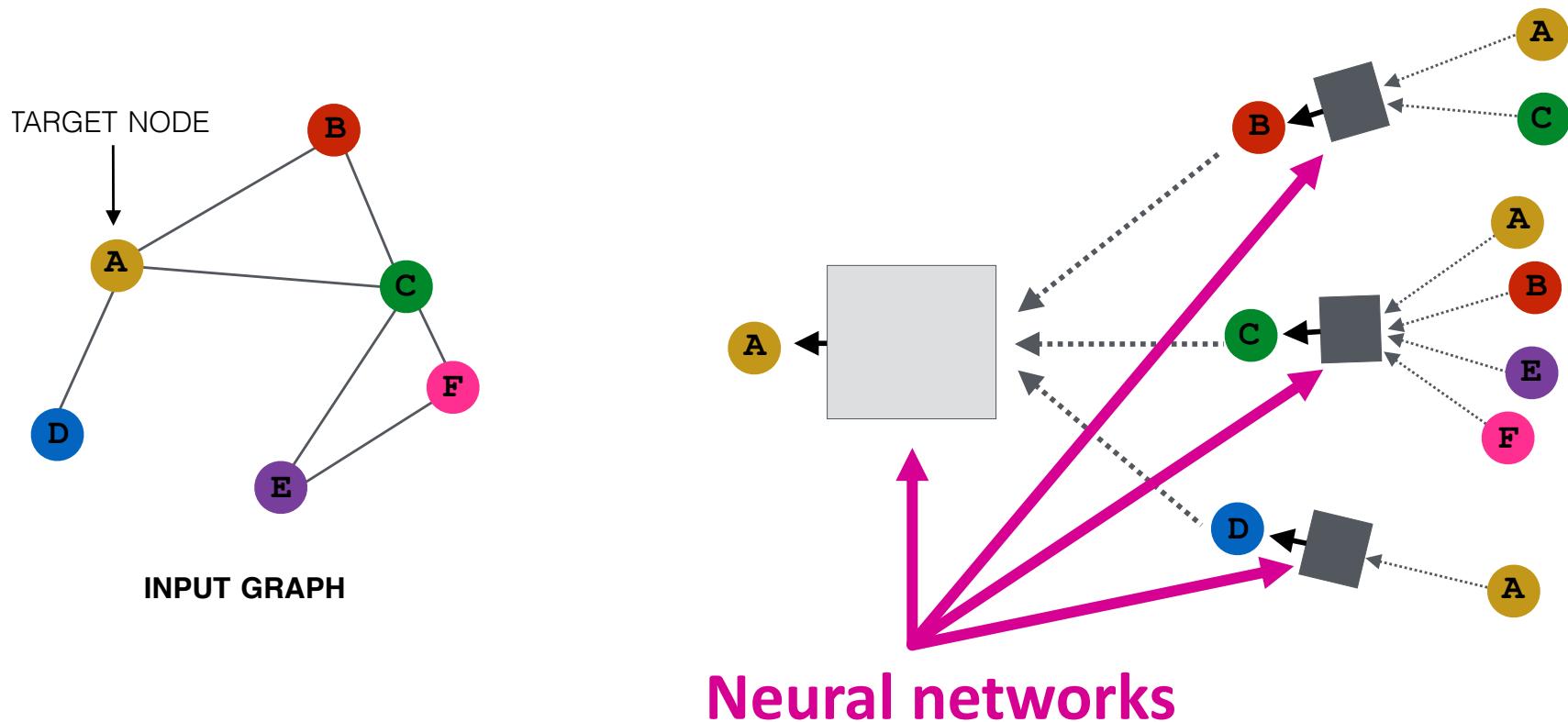


Propagate and transform information

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

Recap: Aggregate from Neighbors

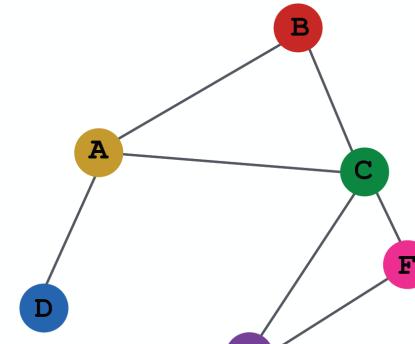
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



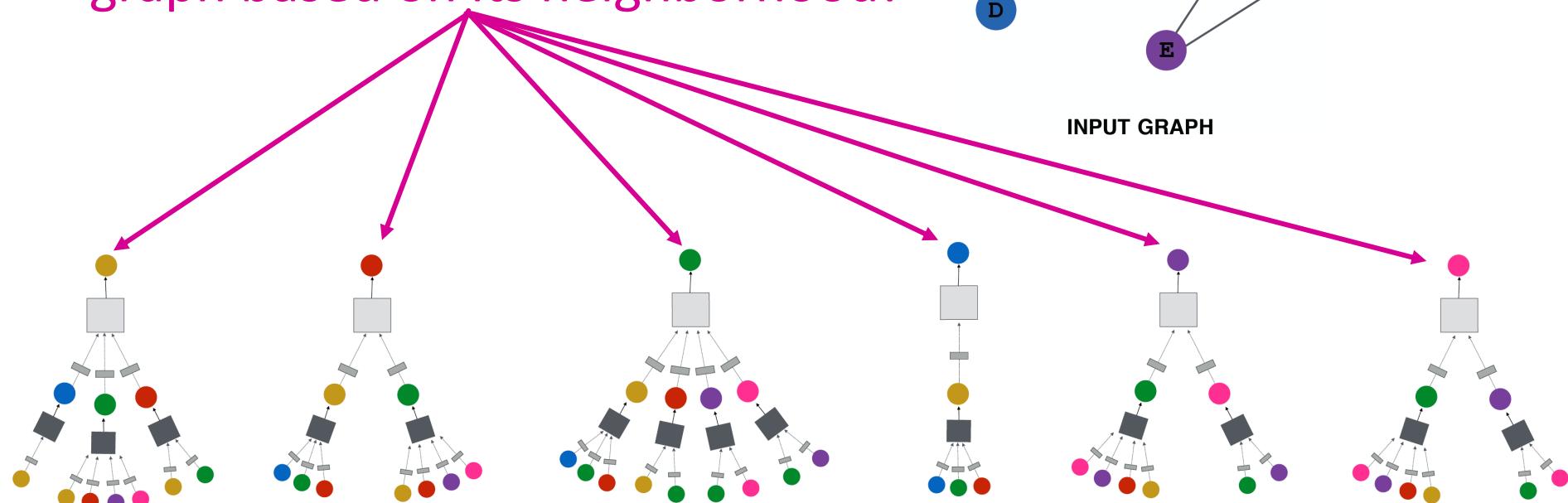
Recap: Aggregate Neighbors

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

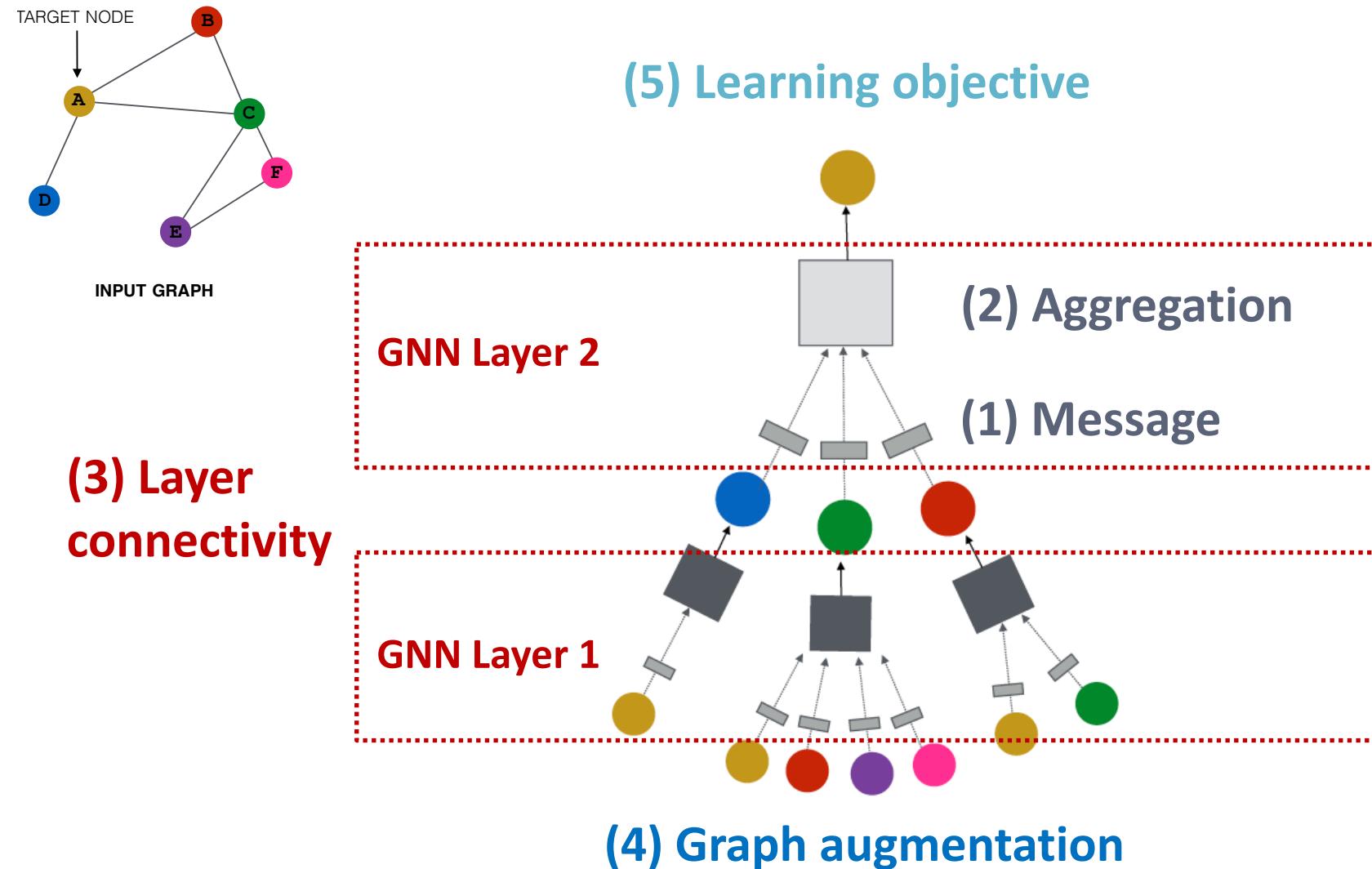
Every node defines a computation graph based on its neighborhood!



INPUT GRAPH



Today: A General GNN Framework



Stanford CS224W: A Single Layer of a GNN

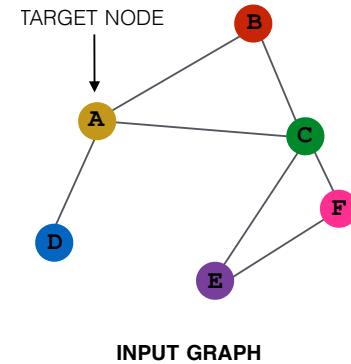
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>

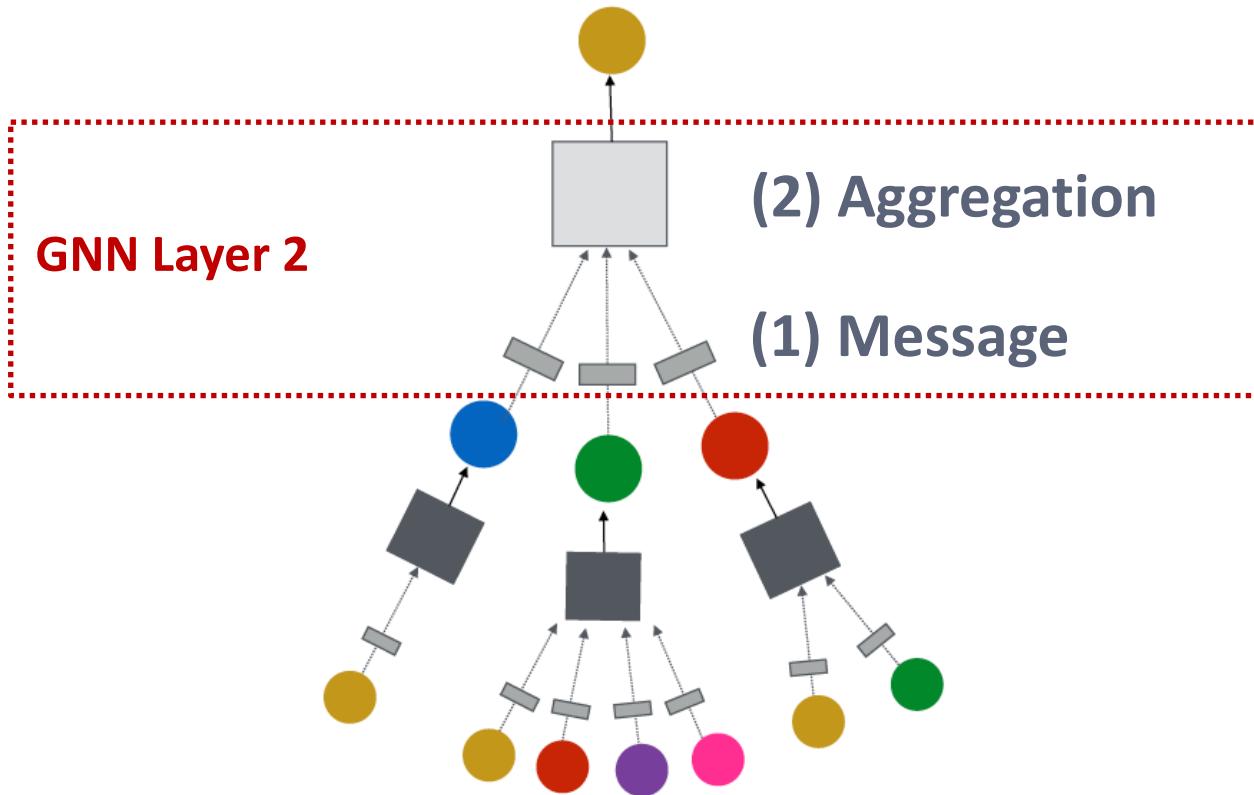


A GNN Layer



GNN Layer = Message + Aggregation

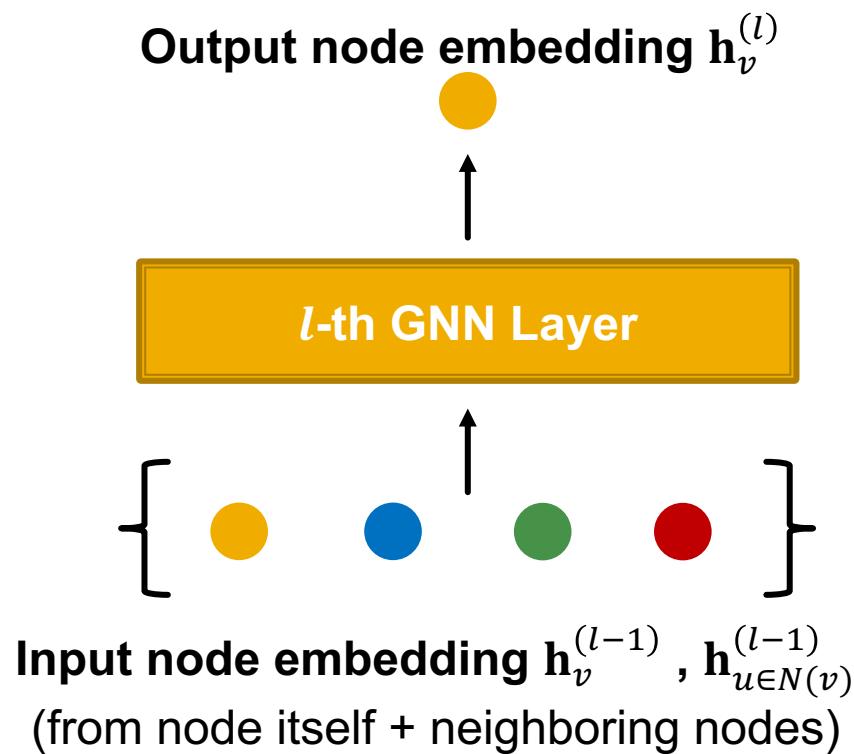
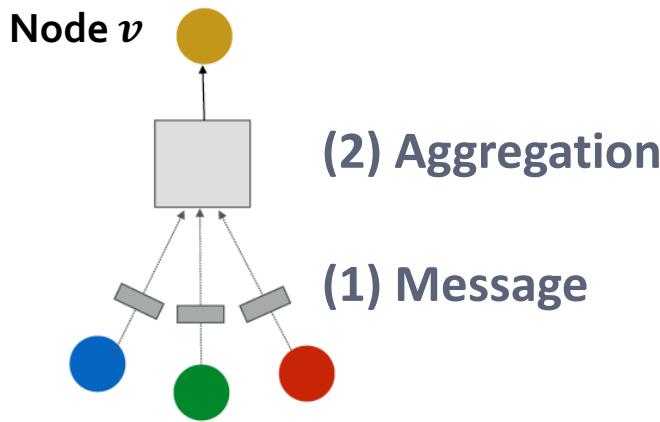
- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...



A Single GNN Layer

■ Idea of a GNN Layer:

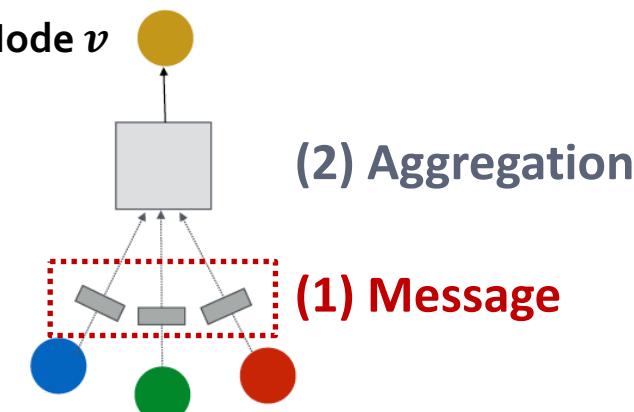
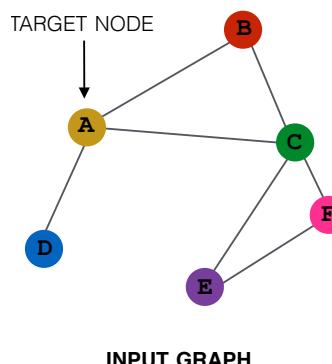
- Compress a set of vectors into a single vector
- Two-step process:
 - (1) Message
 - (2) Aggregation



Message Computation

■ (1) Message computation

- **Message function:** $\mathbf{m}_u^{(l)} = \text{MSG}^{(l)}(\mathbf{h}_u^{(l-1)})$
- **Intuition:** Each node will create a message, which will be sent to other nodes later
- **Example:** A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$
 - Multiply node features with weight matrix $\mathbf{W}^{(l)}$



Message Aggregation

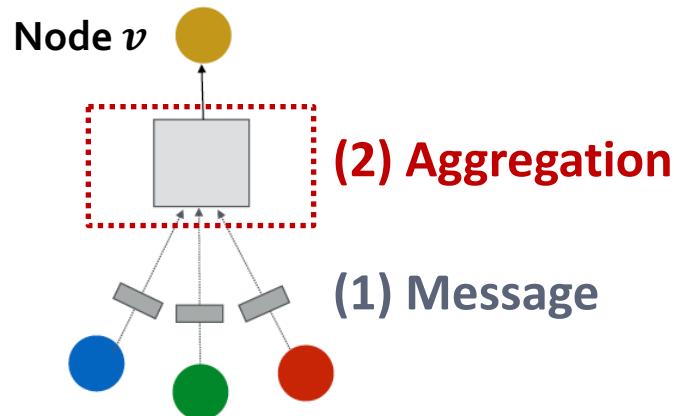
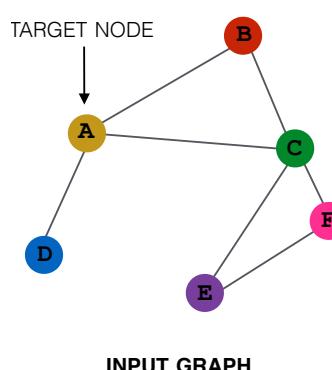
■ (2) Aggregation

- **Intuition:** Node v will aggregate the messages from its neighbors u :

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

- **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

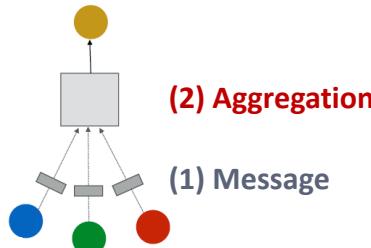
$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$

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



Activation (Non-linearity)

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

- Rectified linear unit (ReLU)

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

- Most commonly used

- Sigmoid

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

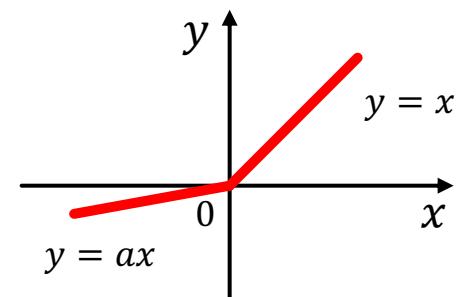
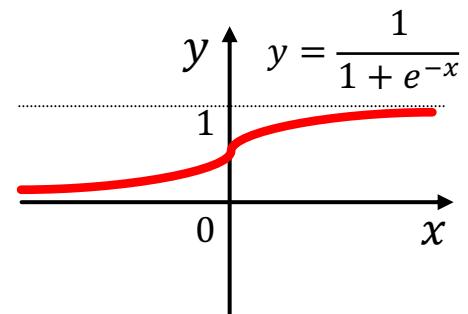
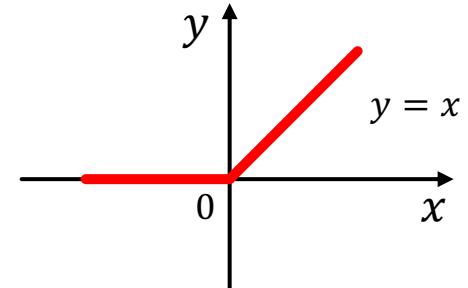
- Used only when you want to restrict the range of your embeddings

- Parametric ReLU

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

a_i is a trainable parameter

- Empirically performs better than ReLU



Classical GNN Layers: GCN (1)

- (1) Graph Convolutional Networks (GCN)

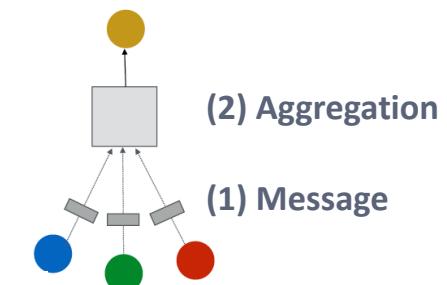
$$\mathbf{h}_v^{(l)} = \sigma \left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

- How to write this as Message + Aggregation?

Message

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

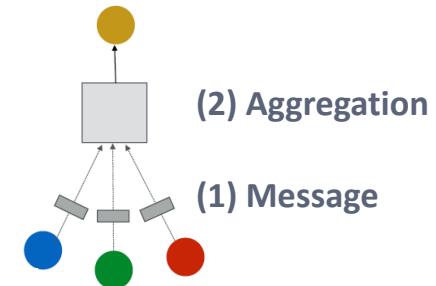
Aggregation



Classical GNN Layers: GCN (2)

■ (1) Graph Convolutional Networks (GCN)

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



■ Message:

- Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{|N(v)|} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}$

Normalized by node degree
(In the GCN paper they use a slightly different normalization)

■ Aggregation:

- Sum over messages from neighbors, then apply activation
- $\mathbf{h}_v^{(l)} = \sigma \left(\text{Sum} \left(\{\mathbf{m}_u^{(l)}, u \in N(v)\} \right) \right)$

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

Classical GNN Layers: GraphSAGE

- (2) GraphSAGE

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

- 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

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

- Message is computed within the $\text{AGG}(\cdot)$
- How to write this as Message + Aggregation?

GraphSAGE Neighbor Aggregation

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

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

AggregationMessage computation

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

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

AggregationMessage computation

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

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

Aggregation

GraphSAGE: L₂ Normalization

■ ℓ_2 Normalization:

- **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
- $$\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\|\mathbf{h}_v^{(l)}\|_2} \quad \forall v \in V \text{ where } \|u\|_2 = \sqrt{\sum_i u_i^2} \text{ (\ell}_2\text{-norm)}$$
- Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2 -norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After ℓ_2 normalization, all vectors will have the same ℓ_2 -norm

Classical GNN Layers: GAT (1)

■ (3) Graph Attention Networks

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

Attention weights

■ In GCN / GraphSAGE

- $\alpha_{vu} = \frac{1}{|N(v)|}$ is the **weighting factor (importance)** of node u 's message to node v
- $\Rightarrow \alpha_{vu}$ is defined **explicitly** based on the **structural properties** of the graph (node degree)
- \Rightarrow All neighbors $u \in N(v)$ are **equally important** to node v

Classical GNN Layers: GAT (1)

■ (3) Graph Attention Networks

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

Attention weights

■ In GCN / GraphSAGE

- $\alpha_{vu} = \frac{1}{|N(v)|}$ is the **weighting factor (importance)** of node u 's message to node v
- $\Rightarrow \alpha_{vu}$ is defined **explicitly** based on the **structural properties** of the graph (node degree)
- \Rightarrow All neighbors $u \in N(v)$ are **equally important** to node v

Classical GNN Layers: GAT (2)

■ (3) Graph Attention Networks

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

Attention weights

Not all node's neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention** α_{vu} focuses on the important parts of the input data and fades out the rest.
 - **Idea:** the NN should devote more computing power on that small but important part of the data.
 - Which part of the data is more important depends on the context and is learned through training.

Graph Attention Networks

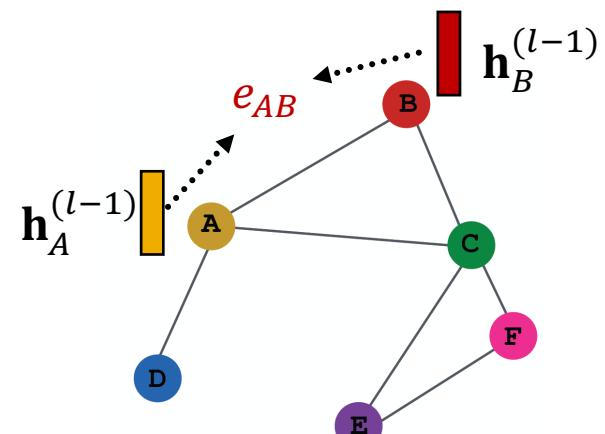
Can we do better than simple neighborhood aggregation?

Can weighting factors α_{vu} be learned?

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

Attention Mechanism (1)

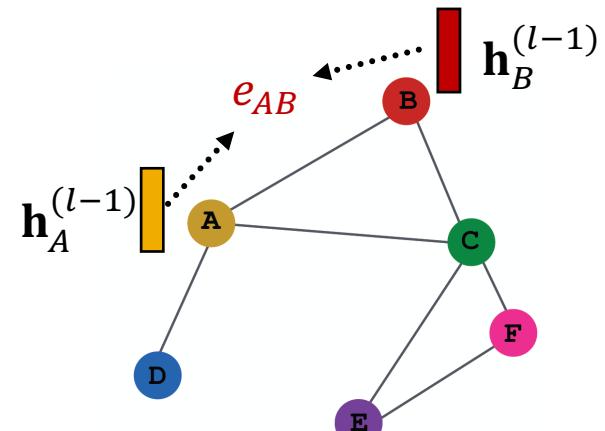
- Let α_{vu} be computed as a byproduct of an **attention mechanism** a :
 - (1) Let a compute **attention coefficients** e_{vu} across pairs of nodes u, v based on their messages:
$$e_{vu} = a(\mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_v^{(l-1)})$$
 - e_{vu} indicates the importance of u 's message to node v



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

Attention Mechanism (1)

- Let α_{vu} be computed as a byproduct of an **attention mechanism** a :
 - (1) Let a compute **attention coefficients** e_{vu} across pairs of nodes u, v based on their messages:
$$e_{vu} = a(\mathbf{m}_u^{(l)}, \mathbf{m}_v^{(l)})$$
 - e_{vu} indicates the importance of u 's message to node v



$$e_{AB} = a(\mathbf{m}_A^{(l)}, \mathbf{m}_B^{(l)})$$

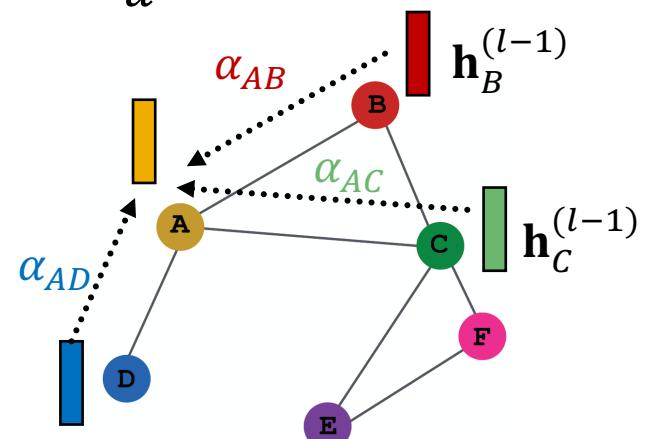
Attention Mechanism (2)

- **Normalize** e_{vu} into the **final attention weight** α_{vu}
 - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$
- **Weighted sum** based on the **final attention weight** α_{vu} :

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

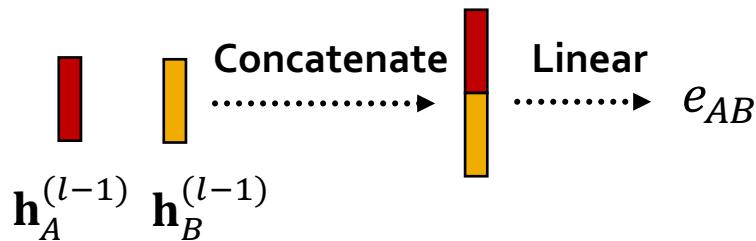
Weighted sum using α_{AB} , α_{AC} , α_{AD} :

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



Attention Mechanism (3)

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



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

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

Attention Mechanism (4)

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

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

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

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

- **Outputs are aggregated:**
 - By concatenation or summation
 - $\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

Stanford CS224W: GNN Layers in Practice

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

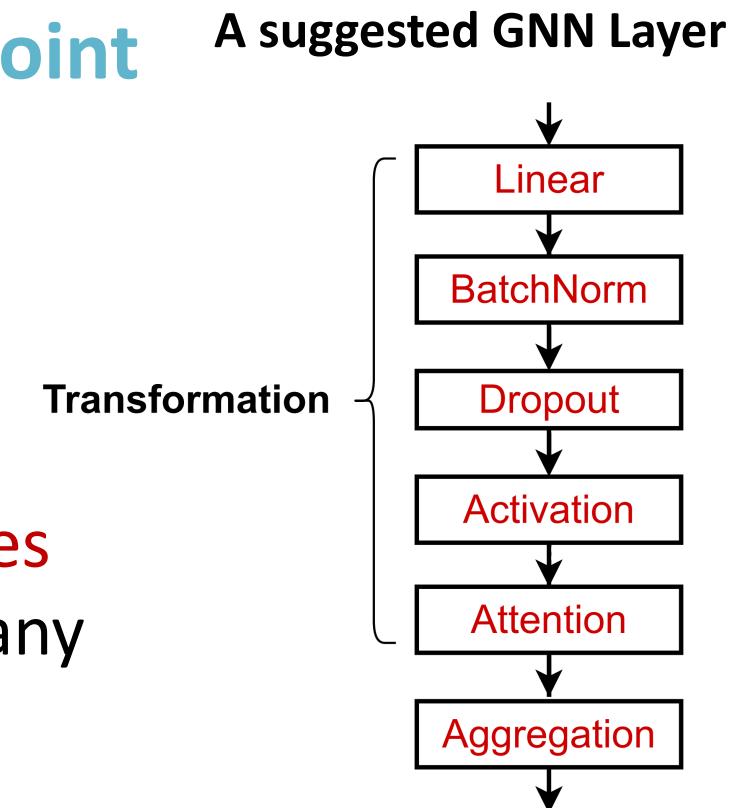
<http://cs224w.stanford.edu>



GNN Layer in Practice

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

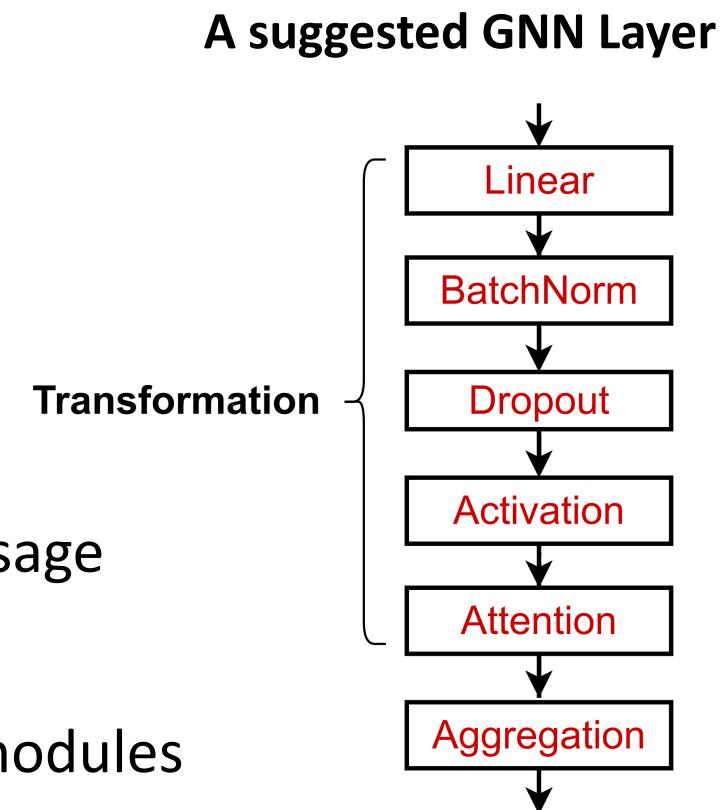
- We can often get better performance by considering a general GNN layer design
- Concretely, we can include modern deep learning modules that proved to be useful in many domains



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

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

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

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

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

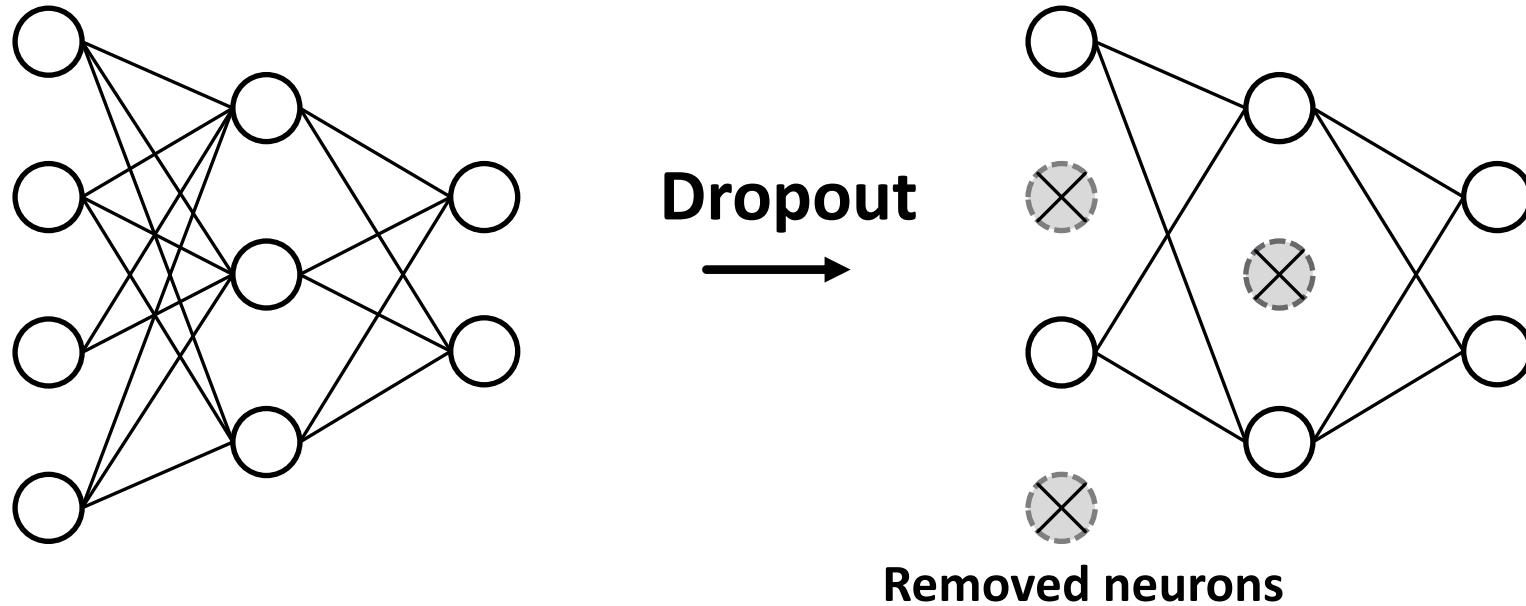
Step 2:
**Normalize the feature
using computed mean
and variance**

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

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

Dropout

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

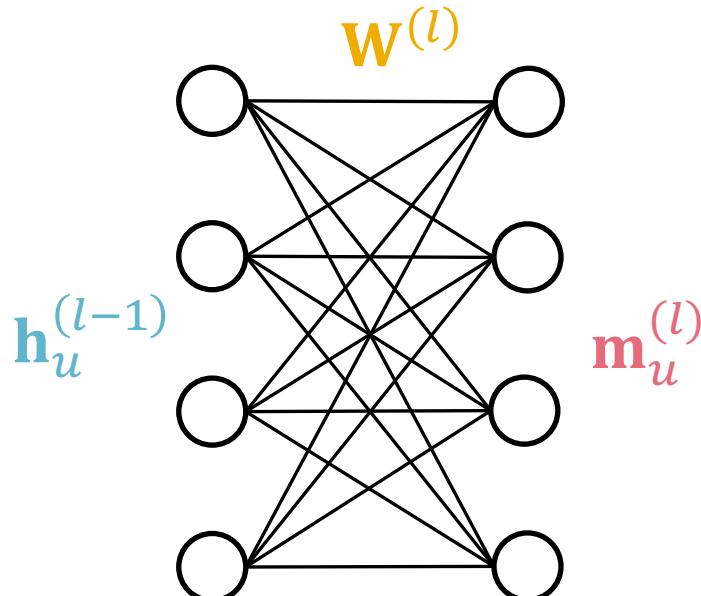


Dropout for GNNs

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

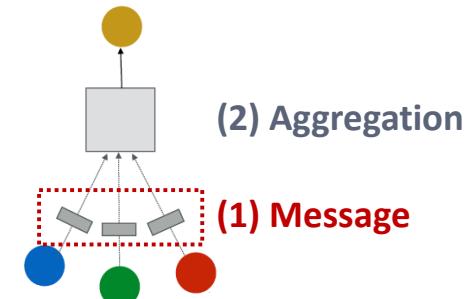
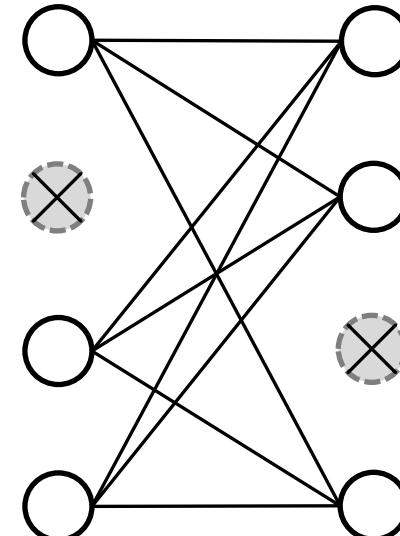
- A simple message function with linear layer:

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



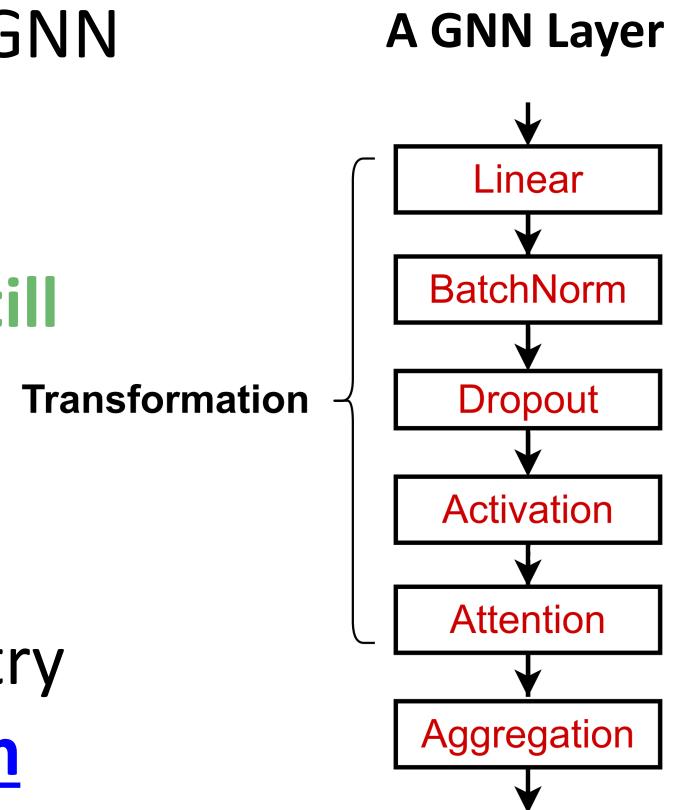
Visualization of a linear layer

Dropout
→



GNN Layer in Practice

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



Stanford CS224W: Stacking Layers of a GNN

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

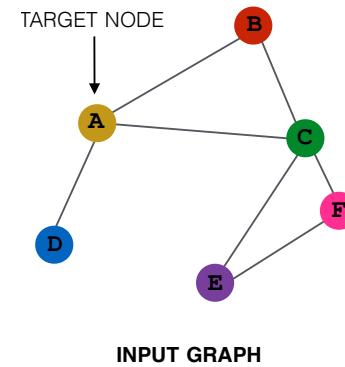
<http://cs224w.stanford.edu>



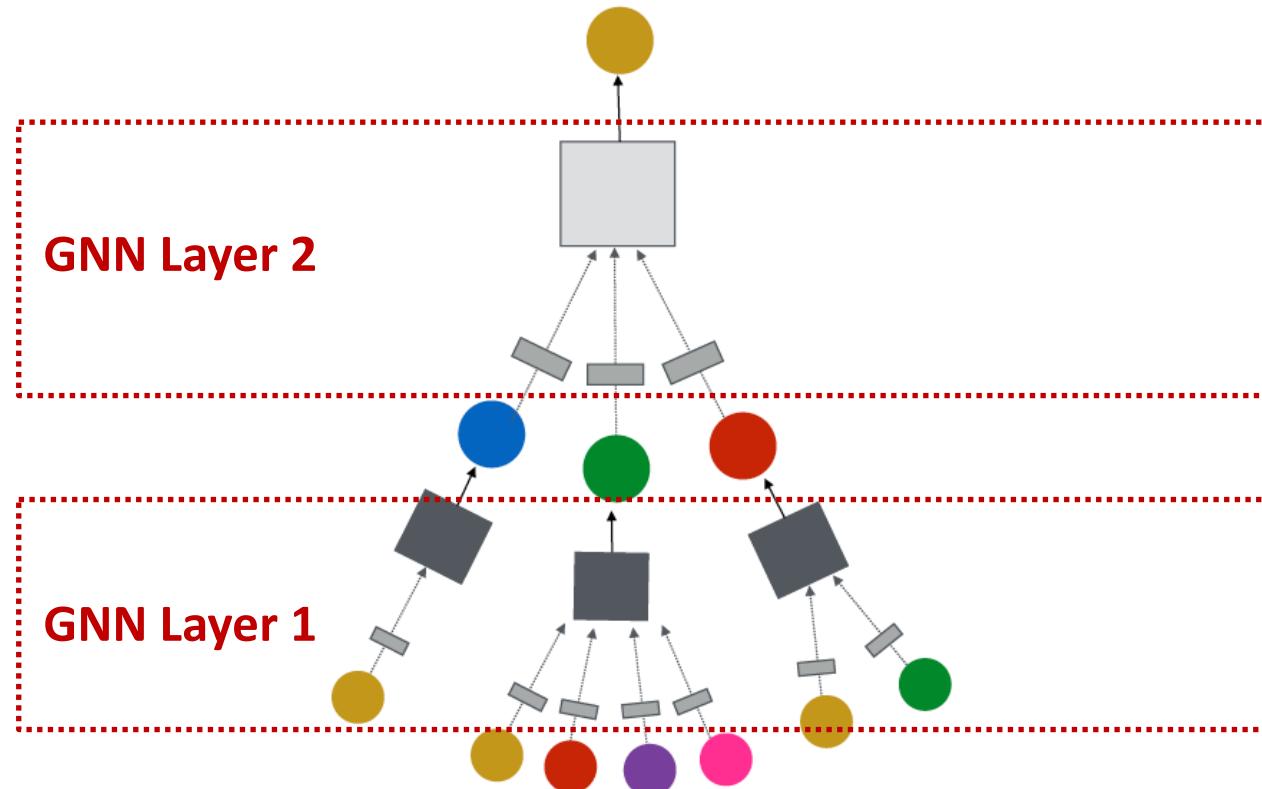
Stacking GNN Layers

How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections

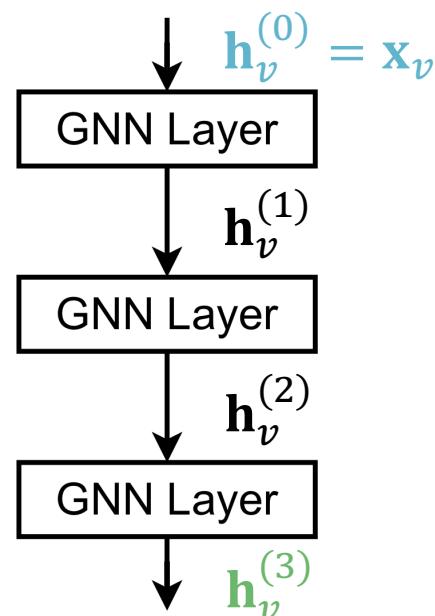


(3) Layer connectivity



Stacking GNN Layers

- **How to construct a Graph Neural Network?**
 - **The standard way:** Stack GNN layers sequentially
 - **Input:** Initial raw node feature \mathbf{x}_v
 - **Output:** Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers



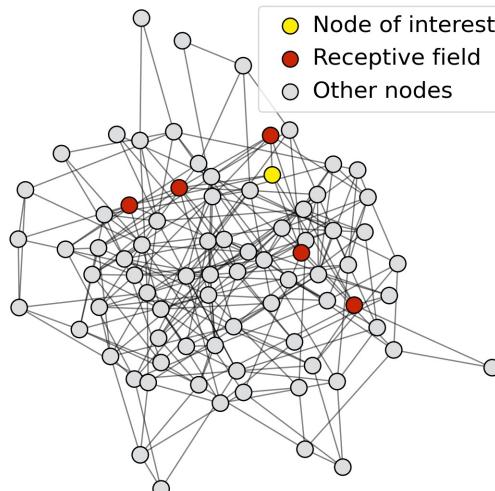
The Over-smoothing Problem

- **The issue of stacking many GNN layers**
 - GNN suffers from **the over-smoothing problem**
- **The over-smoothing problem:** all the node embeddings converge to the same value
 - This is bad because we **want to use node embeddings to differentiate nodes**
- **Why does the over-smoothing problem happen?**

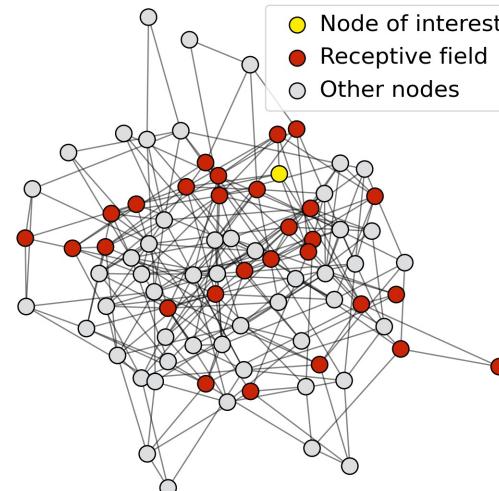
Receptive Field of a GNN

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

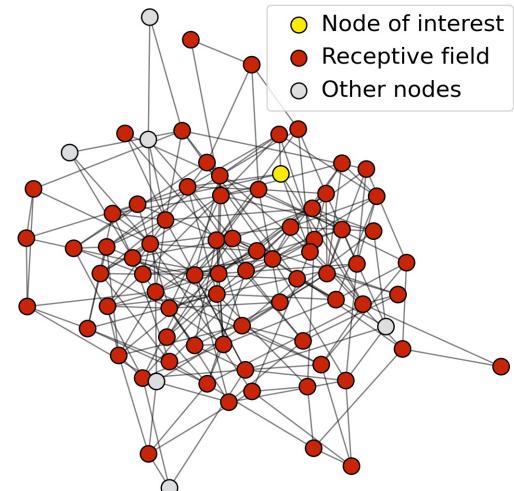
Receptive field for
1-layer GNN



Receptive field for
2-layer GNN



Receptive field for
3-layer GNN

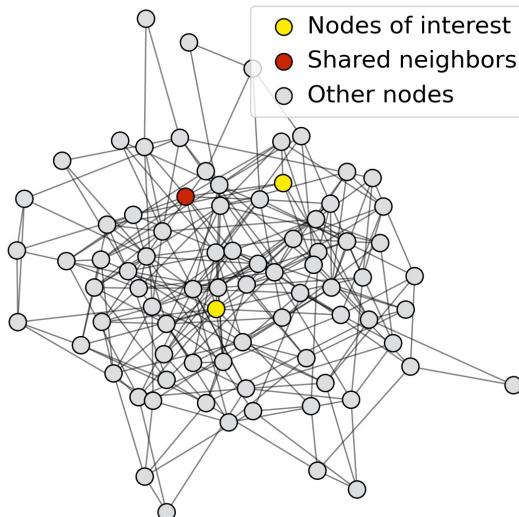


Receptive Field of a GNN

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

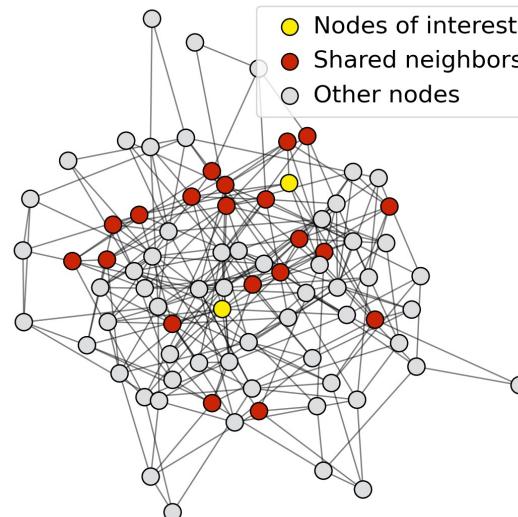
1-hop neighbor overlap

Only 1 node



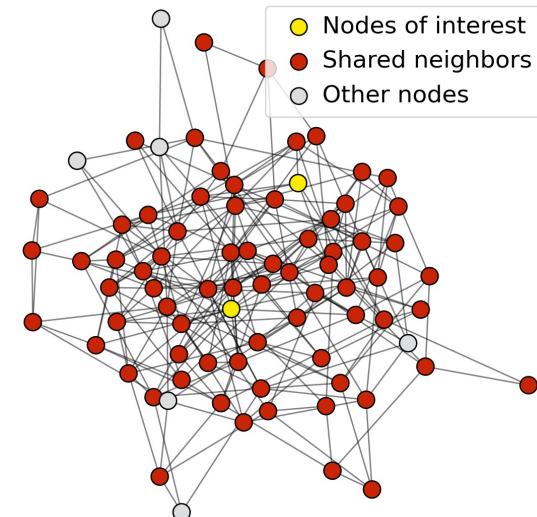
2-hop neighbor overlap

About 20 nodes



3-hop neighbor overlap

Almost all the nodes!



Receptive Field & Over-smoothing

- We can explain over-smoothing via the notion of the receptive field
 - We know the embedding of a node is determined by its receptive field
 - If two nodes have highly-overlapped receptive fields, then their embeddings are highly similar
 - Stack many GNN layers → nodes will have highly-overlapped receptive fields → node embeddings will be highly similar → suffer from the over-smoothing problem
- Next: how do we overcome over-smoothing problem?

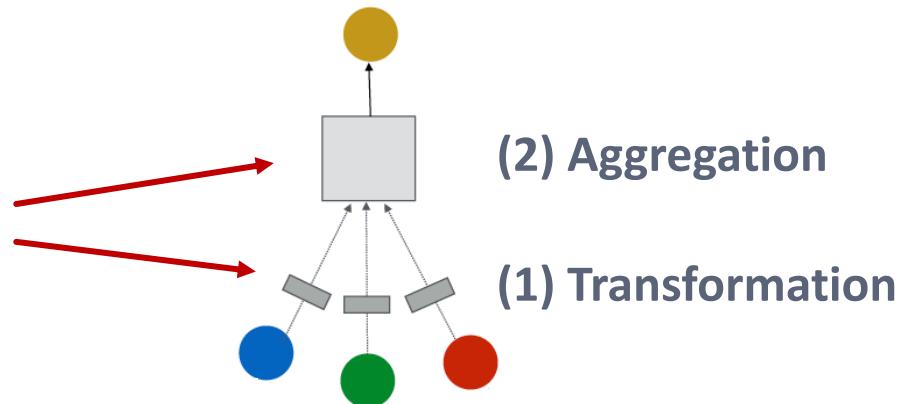
Design GNN Layer Connectivity

- **What do we learn from the over-smoothing problem?**
- **Lesson 1: Be cautious when adding GNN layers**
 - Unlike neural networks in other domains (CNN for image classification), **adding more GNN layers do not always help**
 - **Step 1:** Analyze the necessary receptive field to solve your problem. E.g., by computing the diameter of the graph
 - **Step 2:** Set number of GNN layers L to be a bit more than the receptive field we like. **Do not set L to be unnecessarily large!**
- **Question:** How to enhance the expressive power of a GNN, **if the number of GNN layers is small?**

Expressive Power for Shallow GNNs

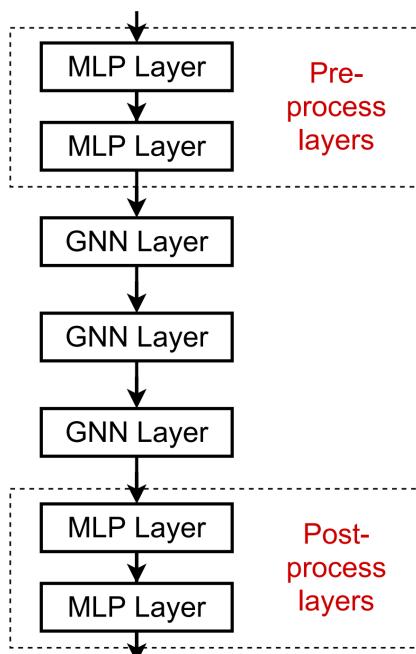
- How to make a shallow GNN more expressive?
- Solution 1: Increase the expressive power within each GNN layer
 - In our previous examples, each transformation or aggregation function only include one linear layer
 - We can make aggregation / transformation become a deep neural network!

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



Expressive Power for Shallow GNNs

- How to make a shallow GNN more expressive?
- Solution 2: Add layers that do not pass messages
 - A GNN does not necessarily only contain GNN layers
 - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**



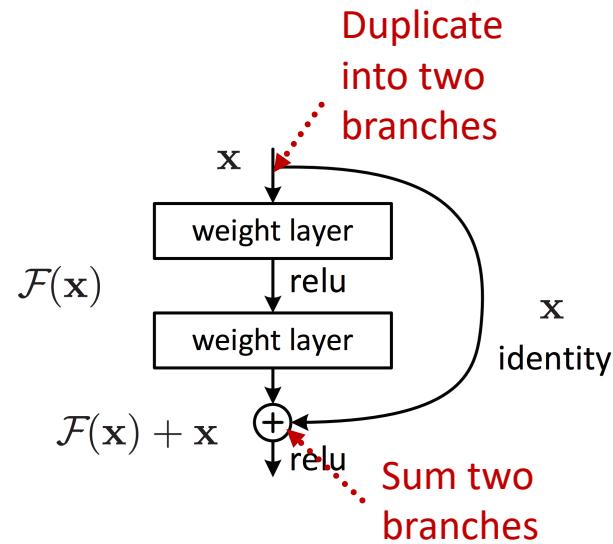
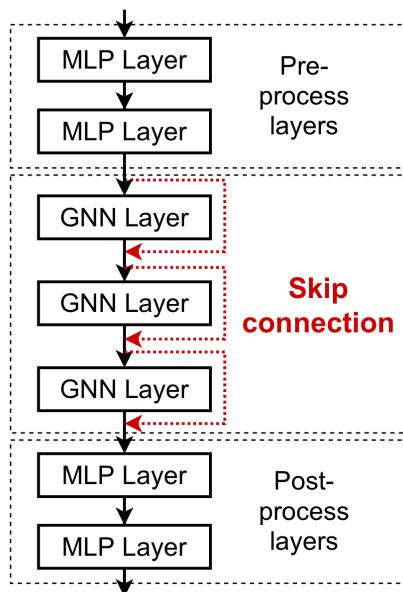
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

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

In practice, adding these layers works great!

Design GNN Layer Connectivity

- What if my problem still requires many GNN layers?
- Lesson 2: Add skip connections in GNNs
 - Observation from over-smoothing: Node embeddings in earlier GNN layers can sometimes better differentiate nodes
 - Solution: We can increase the impact of earlier layers on the final node embeddings, **by adding shortcuts in GNN**



Idea of skip connections:

Before adding shortcuts:

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

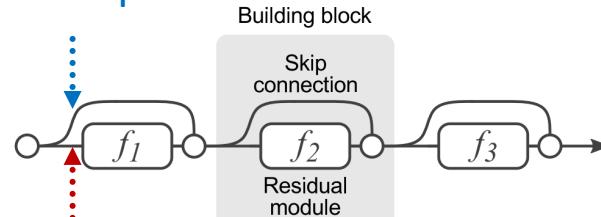
After adding shortcuts:

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

Idea of Skip Connections

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

Path 2: skip this module

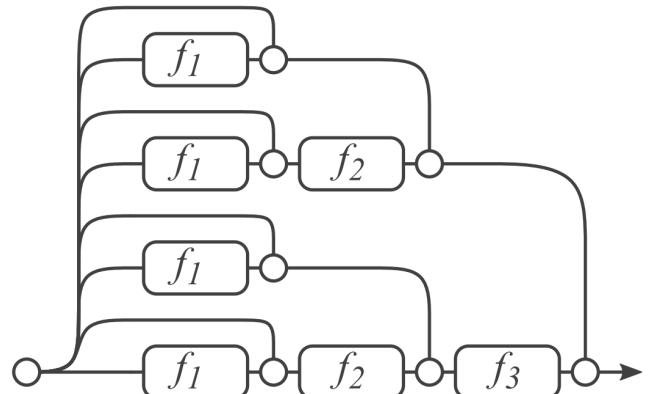


Path 1: include this module

(a) Conventional 3-block residual network

All the possible paths:

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



(b) Unraveled view of (a)

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

Example: GCN with Skip Connections

- A standard GCN layer

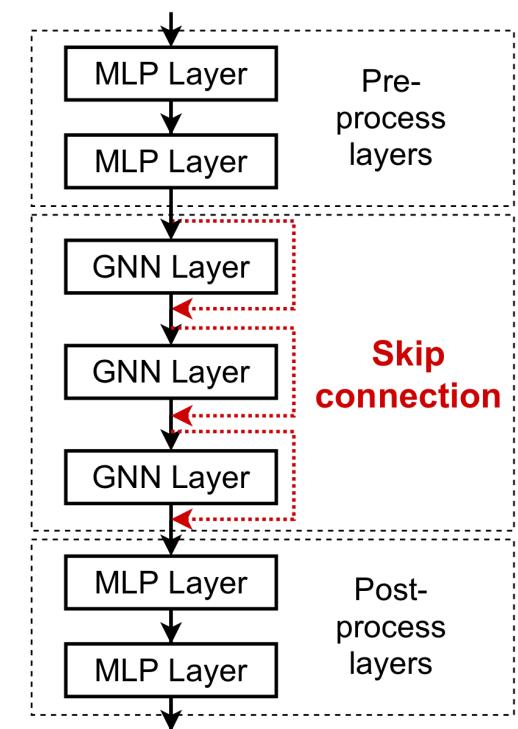
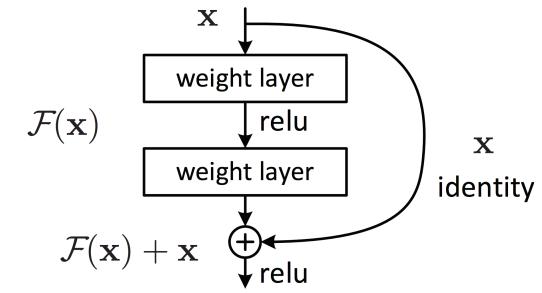
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

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

- A GCN layer with skip connection

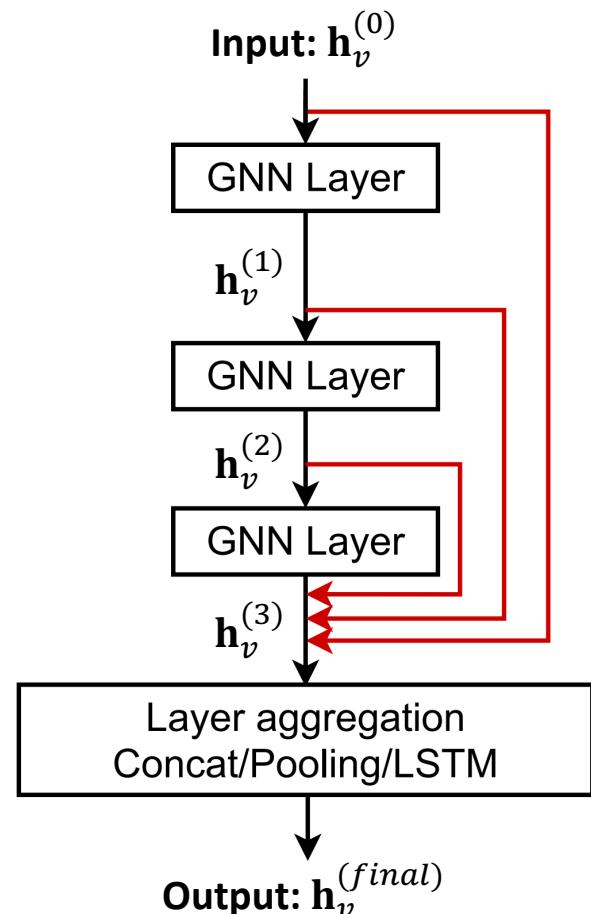
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)} \right)$$

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



Other Options of Skip Connections

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



Stanford CS224W: **Graph Manipulation in GNNs**

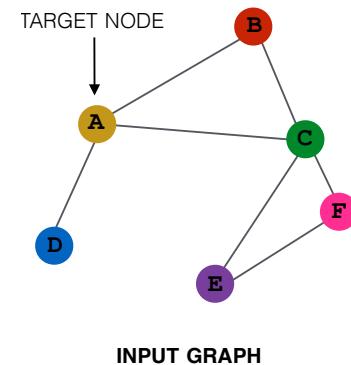
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>

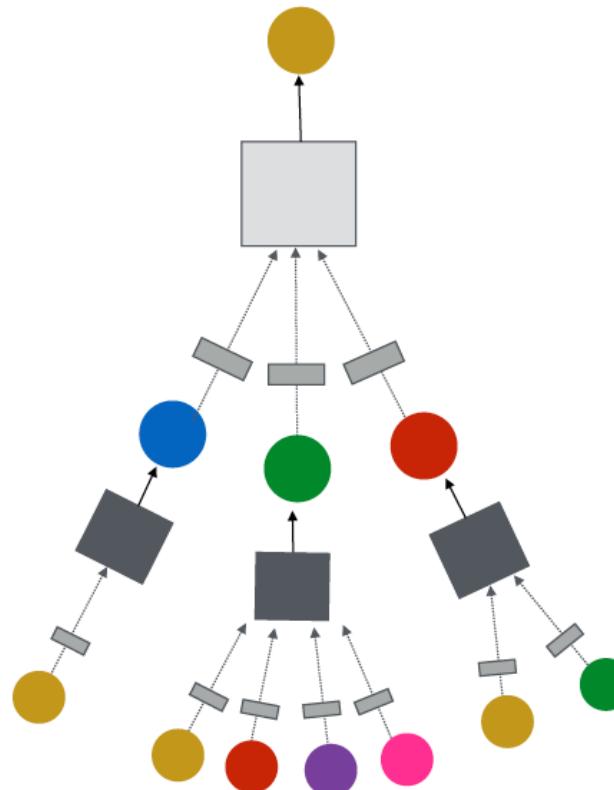


General GNN Framework



Idea: Raw input graph \neq computational graph

- Graph feature augmentation
- Graph structure manipulation



(4) Graph manipulation

Why Manipulate Graphs

Our assumption so far has been

- Raw input graph = computational graph

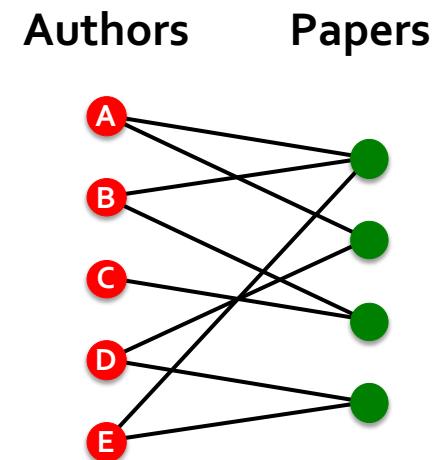
Reasons for breaking this assumption

- Feature level:
 - The input graph **lacks features** → feature augmentation
- Structure level:
 - The graph is **too sparse** → inefficient message passing
 - The graph is **too dense** → message passing is too costly
 - The graph is **too large** → cannot fit the computational graph into a GPU
- It's just **unlikely that the input graph happens to be the optimal computation graph** for embeddings

Add Virtual Nodes / Edges

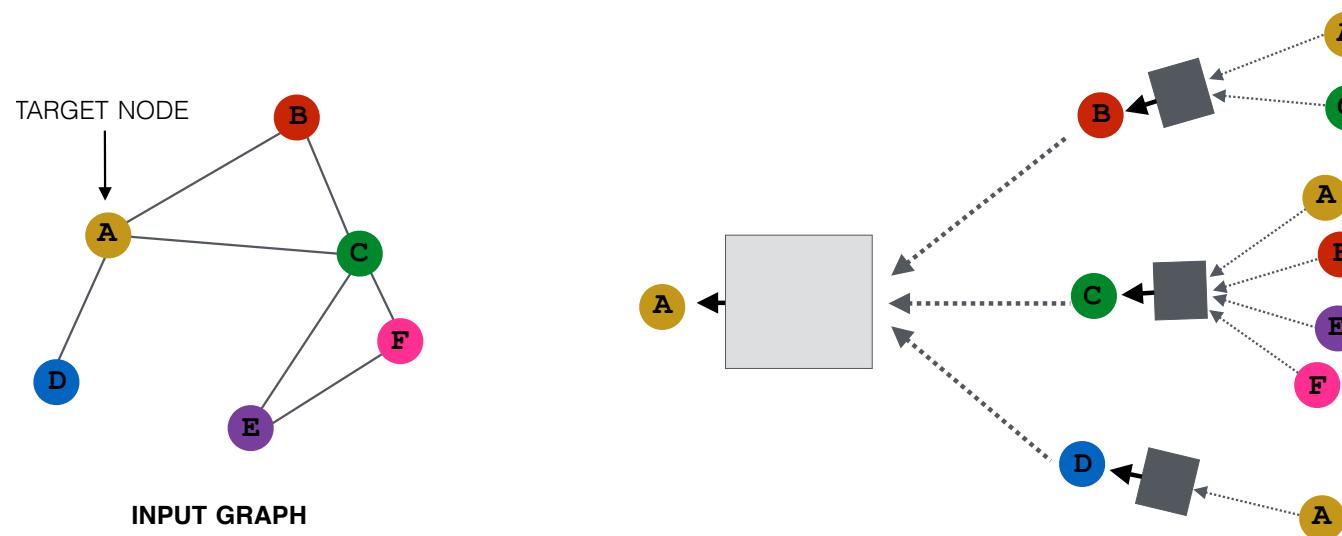
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
 - **Common approach:** Connect 2-hop neighbors via virtual edges
 - **Intuition:** Instead of using adj. matrix A for GNN computation, use $A + A^2$

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



Node Neighborhood Sampling

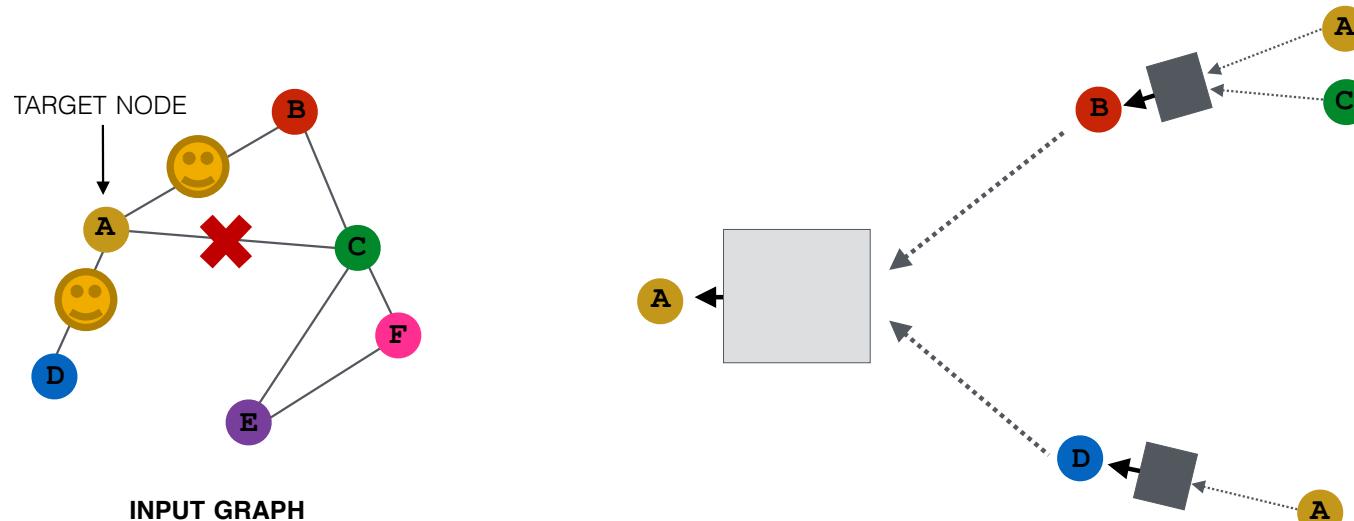
- Our approach so far:
 - All the neighbors are used for message passing
- Problem: Dense/large graphs, high-degree nodes



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

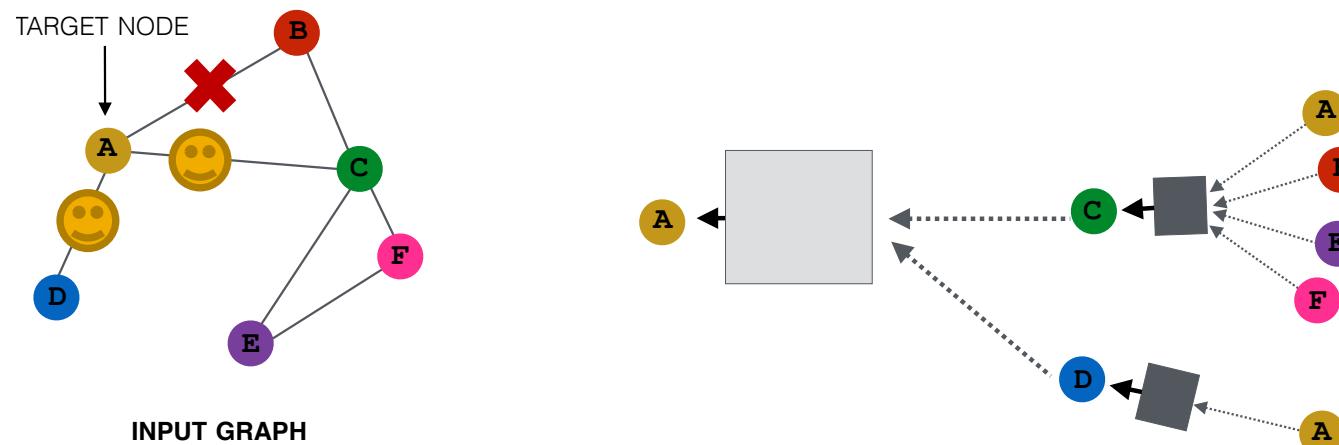
Neighborhood Sampling Example

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



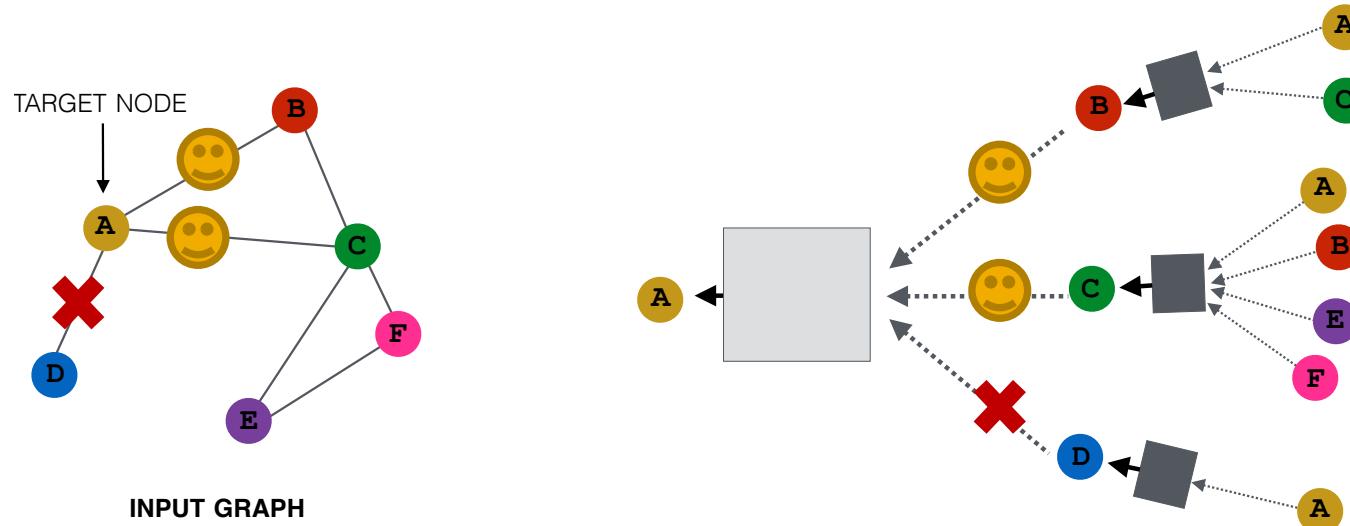
Neighborhood Sampling Example

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



Neighborhood Sampling Example

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



Note to other teachers and users of these slides: We would be delighted if you found our material useful for giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: <http://cs224w.Stanford.edu>

Stanford CS224W: GNN Training

CS224W: Machine Learning with Graphs

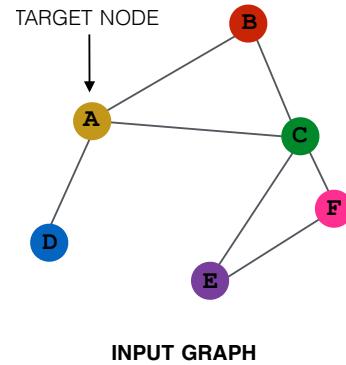
Jure Leskovec, Stanford University

Charilaos Kanatsoulis, Stanford University

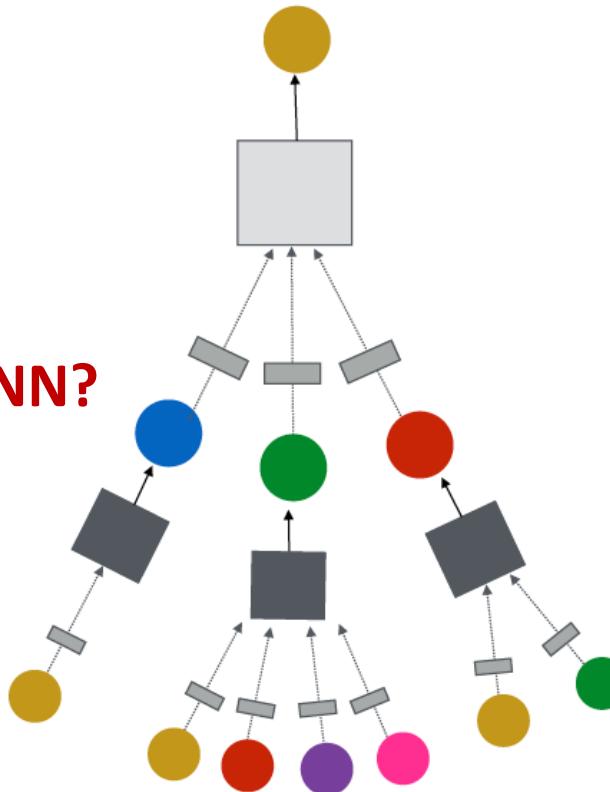
<http://cs224w.stanford.edu>



A General GNN Framework (4)



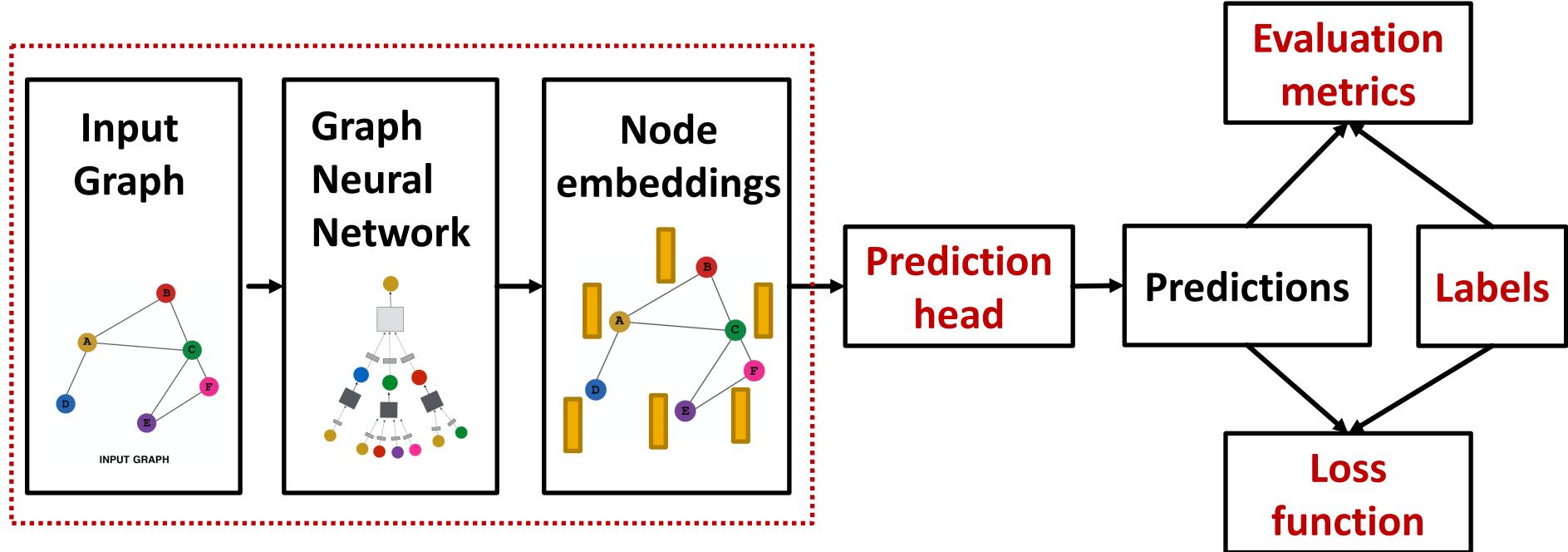
(5) Learning objective



Next: How do we train a GNN?

GNN Training Pipeline

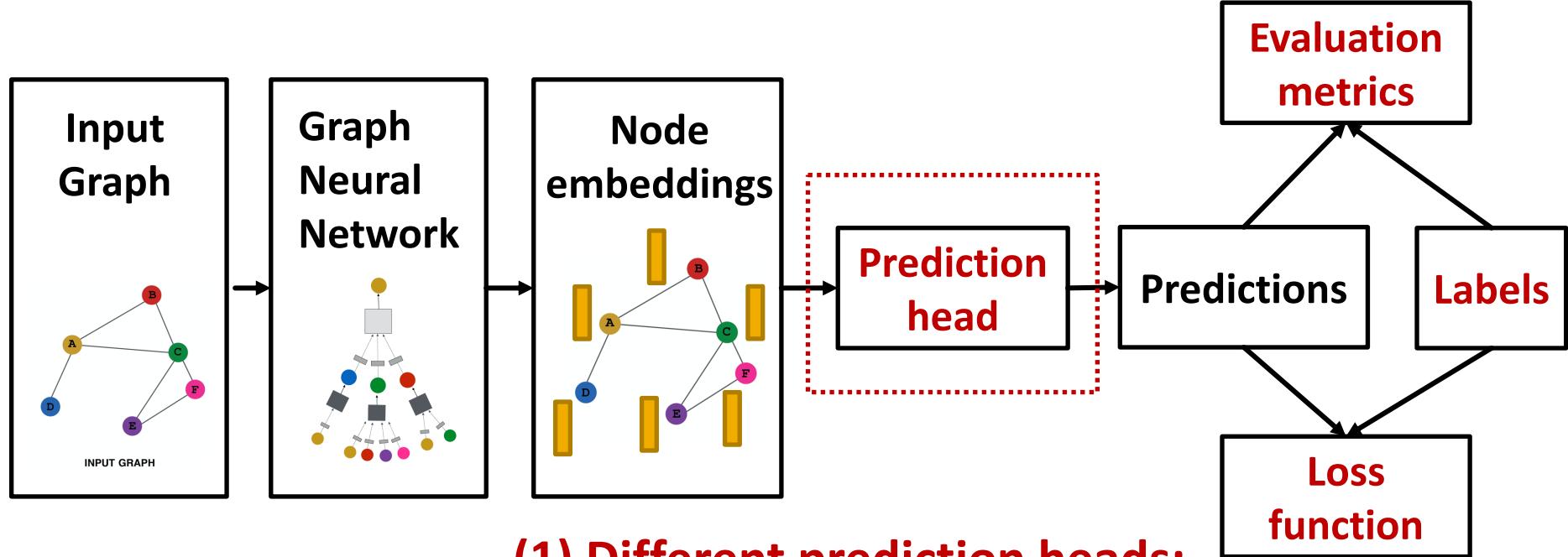
So far what we have covered



Output of a GNN: set of node embeddings

$$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$$

GNN Training Pipeline (1)

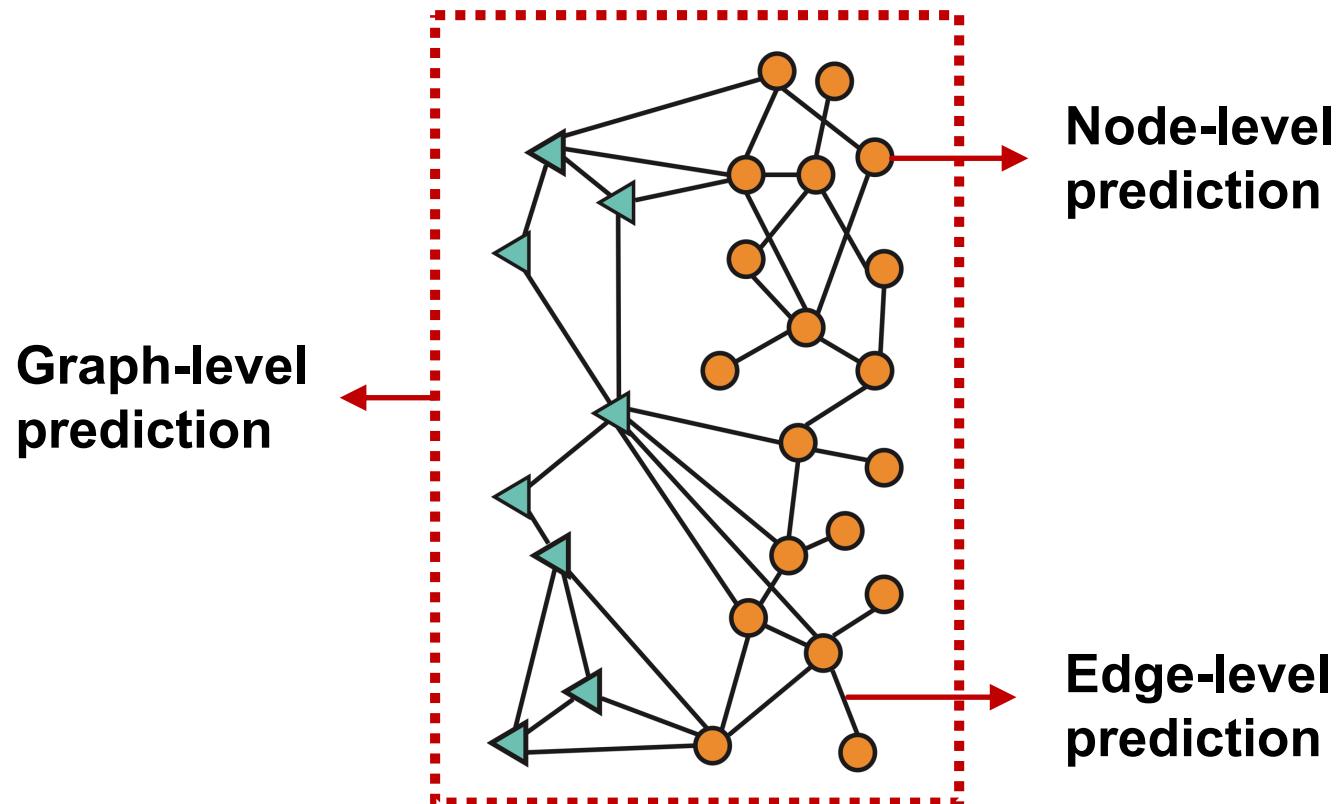


(1) Different prediction heads:

- **Node-level tasks**
- **Edge-level tasks**
- **Graph-level tasks**

GNN Prediction Heads

- Idea: Different task levels require different prediction heads

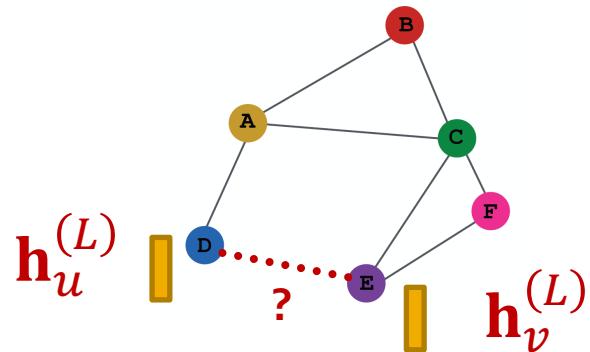


Prediction Heads: Node-level

- **Node-level prediction:** We can directly make prediction using node embeddings!
- After GNN computation, we have d -dim node embeddings: $\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\}$
- Suppose we want to make k -way prediction
 - Classification: classify among k categories
 - Regression: regress on k targets
- $\hat{y}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)} \mathbf{h}_v^{(L)}$
 - $\mathbf{W}^{(H)} \in \mathbb{R}^{k \times d}$: We map node embeddings from $\mathbf{h}_v^{(L)} \in \mathbb{R}^d$ to $\hat{y}_v \in \mathbb{R}^k$ so that we can compute the loss

Prediction Heads: Edge-level

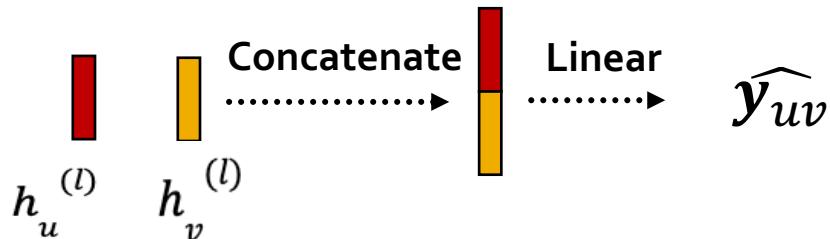
- **Edge-level prediction:** Make prediction using pairs of node embeddings
- Suppose we want to make k -way prediction
- $\hat{y}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



- What are the options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$?

Prediction Heads: Edge-level

- Options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- (1) Concatenation + Linear
 - We have seen this in graph attention



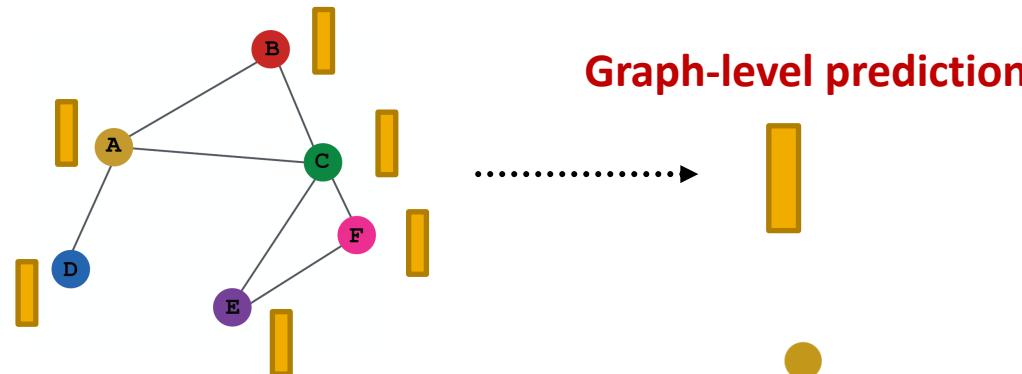
- $\widehat{y}_{uv} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
- Here $\text{Linear}(\cdot)$ will map **2d-dimensional** embeddings (since we concatenated embeddings) to **k-dim** embeddings (k -way prediction)

Prediction Heads: Edge-level

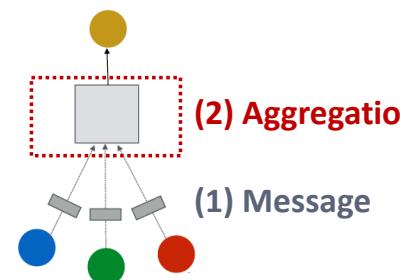
- Options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- **(2) Dot product**
 - $\hat{y}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
 - This approach only applies to **1-way prediction** (e.g., link prediction: predict the existence of an edge)
 - Applying to **k -way prediction**:
 - Similar to **multi-head attention**: $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(k)}$ trainable
$$\hat{y}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$
$$\dots$$
$$\hat{y}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$
$$\hat{y}_{uv} = \text{Concat}(\hat{y}_{uv}^{(1)}, \dots, \hat{y}_{uv}^{(k)}) \in \mathbb{R}^k$$

Prediction Heads: Graph-level

- **Graph-level prediction:** Make prediction using all the node embeddings in our graph
- Suppose we want to make k -way prediction
- $\hat{\mathbf{y}}_G = \text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$



- $\text{Head}_{\text{graph}}(\cdot)$ is similar to $\text{AGG}(\cdot)$ in a GNN layer!



Prediction Heads: Graph-level

- Options for $\text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$

- **(1) Global mean pooling**

$$\hat{\mathbf{y}}_G = \text{Mean}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(2) Global max pooling**

$$\hat{\mathbf{y}}_G = \text{Max}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(3) Global sum pooling**

$$\hat{\mathbf{y}}_G = \text{Sum}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- These options work great for small graphs
- **Can we do better for large graphs?**

Issue of Global Pooling

- **Issue:** Global pooling over a (large) graph will lose information
- **Toy example:** we use 1-dim node embeddings
 - Node embeddings for G_1 : $\{-1, -2, 0, 1, 2\}$
 - Node embeddings for G_2 : $\{-10, -20, 0, 10, 20\}$
 - Clearly G_1 and G_2 have very different node embeddings
→ Their structures should be different
- **If we do global sum pooling:**
 - **Prediction for G_1 :** $\hat{y}_G = \text{Sum}(\{-1, -2, 0, 1, 2\}) = 0$
 - **Prediction for G_2 :** $\hat{y}_G = \text{Sum}(\{-10, -20, 0, 10, 20\}) = 0$
 - We cannot differentiate G_1 and G_2 !

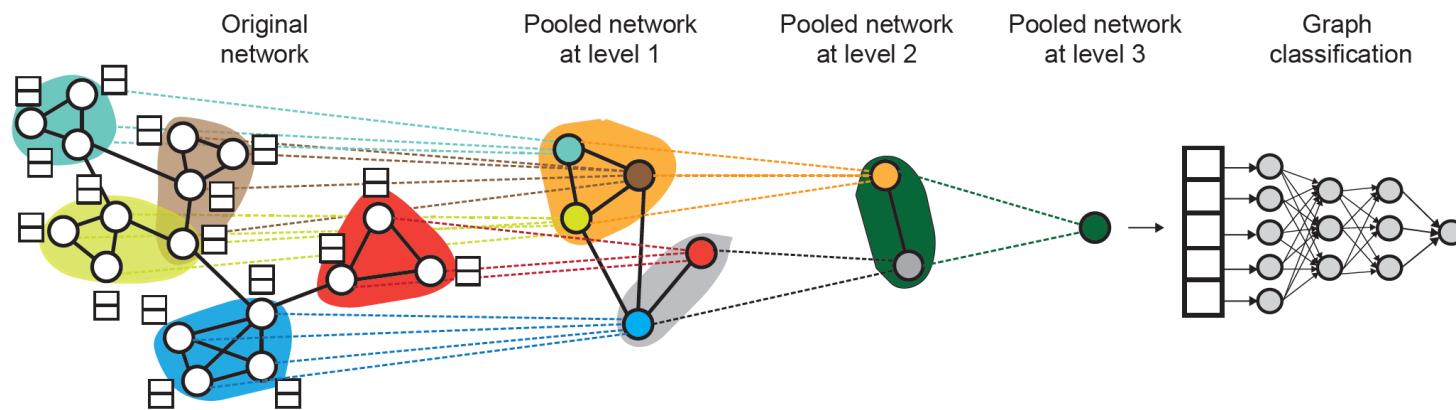
Hierarchical Global Pooling

- **A solution:** Let's aggregate all the node embeddings **hierarchically**
 - **Toy example:** We will aggregate via $\text{ReLU}(\text{Sum}(\cdot))$
 - We first **separately** aggregate the first 2 nodes and last 3 nodes
 - Then we aggregate again to make the final prediction
 - G_1 node embeddings: $\{-1, -2, 0, 1, 2\}$
 - **Round 1:** $\hat{y}_a = \text{ReLU}(\text{Sum}(\{-1, -2\})) = 0$, $\hat{y}_b = \text{ReLU}(\text{Sum}(\{0, 1, 2\})) = 3$
 - **Round 2:** $\hat{y}_G = \text{ReLU}(\text{Sum}(\{\hat{y}_a, \hat{y}_b\})) = 3$
 - G_2 node embeddings: $\{-10, -20, 0, 10, 20\}$
 - **Round 1:** $\hat{y}_a = \text{ReLU}(\text{Sum}(\{-10, -20\})) = 0$, $\hat{y}_b = \text{ReLU}(\text{Sum}(\{0, 10, 20\})) = 30$
 - **Round 2:** $\hat{y}_G = \text{ReLU}(\text{Sum}(\{\hat{y}_a, \hat{y}_b\})) = 30$

Now we can
differentiate
 G_1 and G_2 !

Hierarchical Pooling In Practice

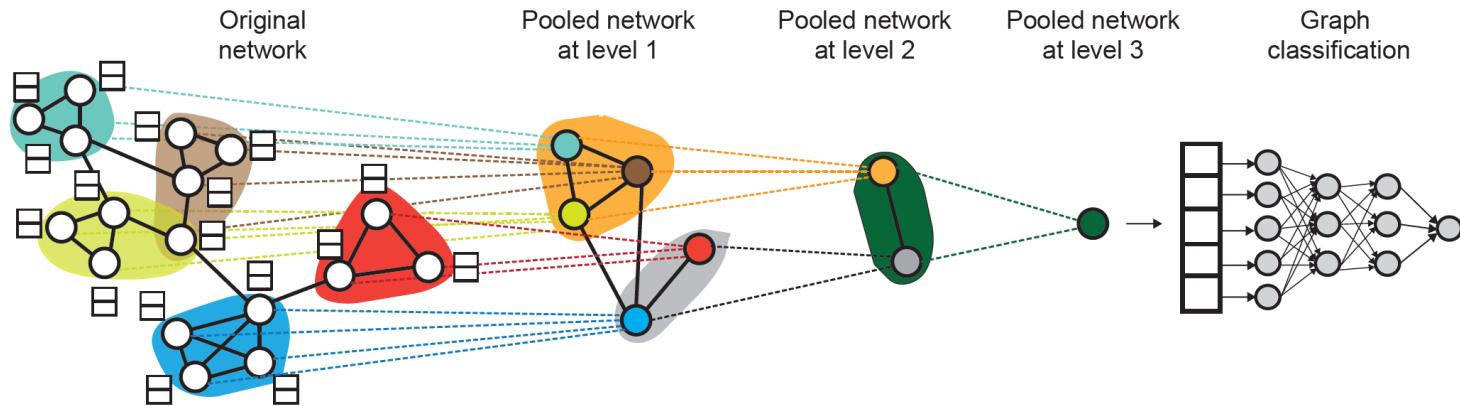
- DiffPool idea:
 - Hierarchically pool node embeddings



- Leverage 2 independent GNNs at each level
 - **GNN A:** Compute node embeddings
 - **GNN B:** Compute the cluster that a node belongs to
- **GNNs A and B at each level can be executed in parallel**

Hierarchical Pooling In Practice

■ DiffPool idea:

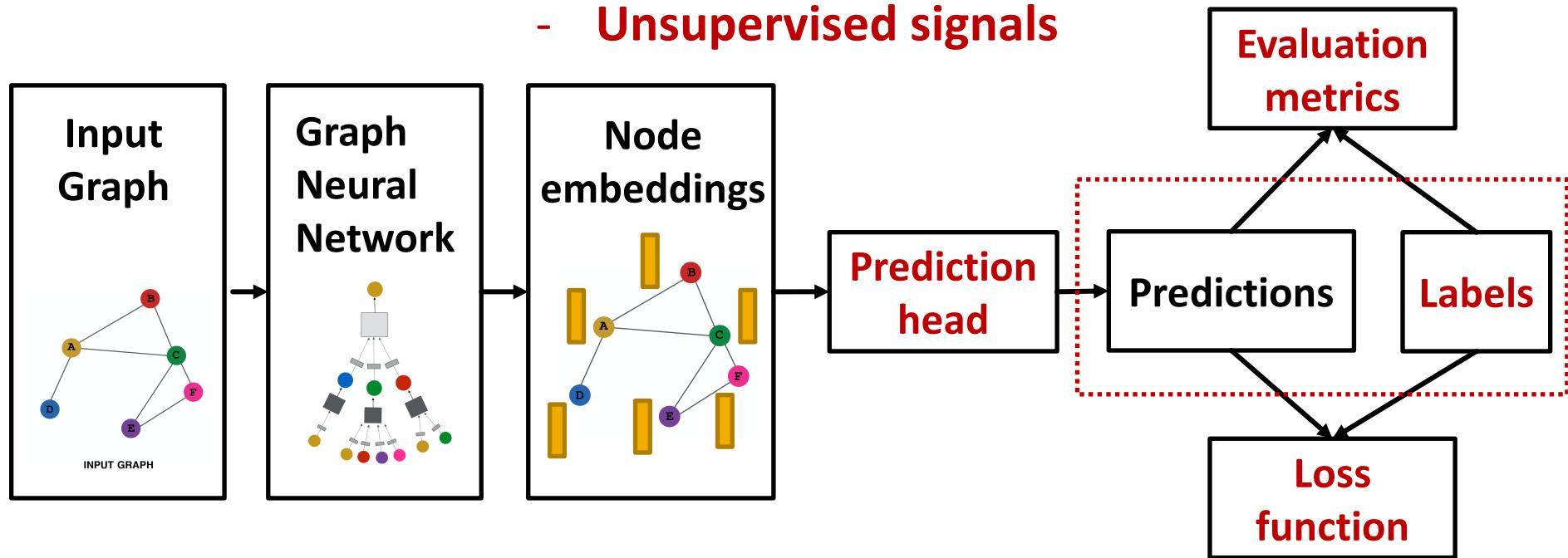


- **For each Pooling layer**
 - Use clustering assignments from **GNN B** to aggregate node embeddings generated by **GNN A**
 - Create a **single new node** for each cluster, maintaining edges between clusters to generate a new **pooled** network
- **Jointly train GNN A and GNN B**

GNN Training Pipeline (2)

(2) Where does ground-truth come from?

- Supervised labels
- Unsupervised signals



Supervised vs Unsupervised

- **Supervised learning on graphs**
 - **Labels come from external sources**
 - E.g., predict drug likeness of a molecular graph
- **Unsupervised learning on graphs**
 - **Signals come from graphs themselves**
 - E.g., link prediction: predict if two nodes are connected
- **Sometimes the differences are blurry**
 - We still have “supervision” in unsupervised learning
 - E.g., train a GNN to predict node clustering coefficient
 - An alternative name for “**unsupervised**” is “**self-supervised**”

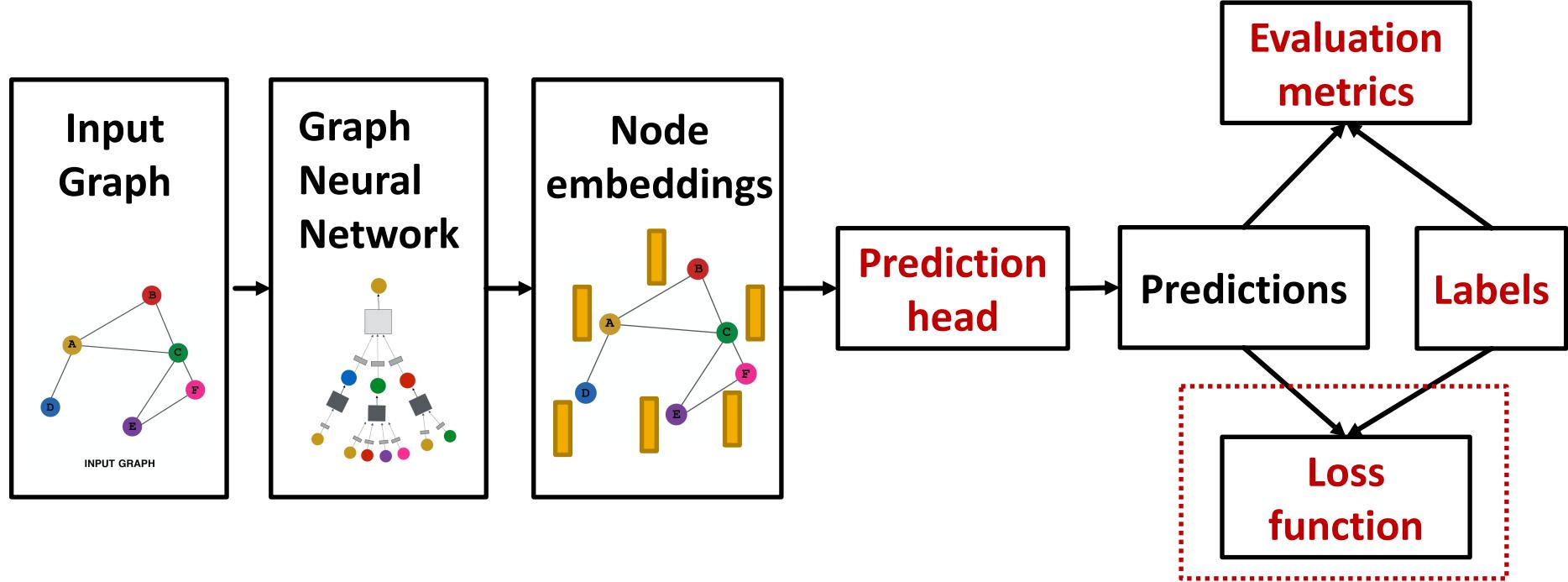
Supervised Labels on Graphs

- **Supervised labels come from the specific use cases.** For example:
 - **Node labels y_v :** in a citation network, which subject area does a node belong to
 - **Edge labels y_{uv} :** in a transaction network, whether an edge is fraudulent
 - **Graph labels y_G :** among molecular graphs, the drug likeness of graphs
- **Advice:** Reduce your task to node / edge / graph labels, since they are easy to work with
 - E.g., we knew some nodes form a cluster. We can treat the cluster that a node belongs to as a **node label**

Unsupervised Signals on Graphs

- **The problem:** sometimes **we only have a graph, without any external labels**
- **The solution:** “self-supervised learning”, we can find supervision signals within the graph.
 - For example, we can let **GNN** predict the following:
 - **Node-level** y_v . Node statistics: such as clustering coefficient, PageRank, ...
 - **Edge-level** y_{uv} . Link prediction: hide the edge between two nodes, predict if there should be a link
 - **Graph-level** y_G . Graph statistics: for example, predict if two graphs are isomorphic
 - **These tasks do not require any external labels!**

GNN Training Pipeline (3)



(3) How do we compute the final loss?

- Classification loss
- Regression loss

Settings for GNN Training

- **The setting:** We have N data points
 - Each data point can be a node/edge/graph
 - **Node-level:** prediction $\hat{y}_v^{(i)}$, label $y_v^{(i)}$
 - **Edge-level:** prediction $\hat{y}_{uv}^{(i)}$, label $y_{uv}^{(i)}$
 - **Graph-level:** prediction $\hat{y}_G^{(i)}$, label $y_G^{(i)}$
 - We will use prediction $\hat{y}^{(i)}$, label $y^{(i)}$ to refer **predictions at all levels**

Classification or Regression

- **Classification:** labels $y^{(i)}$ with discrete value
 - E.g., Node classification: which category does a node belong to
- **Regression:** labels $y^{(i)}$ with continuous value
 - E.g., predict the drug likeness of a molecular graph
- GNNs can be applied to both settings
- **Differences: loss function & evaluation metrics**

Classification Loss

- **Cross entropy (CE)** is a very common loss function in classification
- *K-way prediction* for i -th data point:

$$\text{CE}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = - \sum_{j=1}^K y_j^{(i)} \log(\hat{y}_j^{(i)})$$

Label Prediction

i-th data point
j-th class

where:

E.g.

0	0	1	0	0
---	---	---	---	---

$\mathbf{y}^{(i)} \in \mathbb{R}^K$ = one-hot label encoding

$\hat{\mathbf{y}}^{(i)} \in \mathbb{R}^K$ = prediction after $\text{Softmax}(\cdot)$

E.g.

0.1	0.3	0.4	0.1	0.1
-----	-----	-----	-----	-----

- Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^N \text{CE}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$$

Regression Loss

- For regression tasks we often use **Mean Squared Error (MSE)** a.k.a. **L2 loss**
- K*-way regression for data point (i):

$$\text{MSE}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = \sum_{j=1}^K (\mathbf{y}_j^{(i)} - \hat{\mathbf{y}}_j^{(i)})^2$$

i-th data point
j-th target

where:

E.g.

1.4	2.3	1.0	0.5	0.6
-----	-----	-----	-----	-----

$\mathbf{y}^{(i)} \in \mathbb{R}^k$ = Real valued vector of targets

$\hat{\mathbf{y}}^{(i)} \in \mathbb{R}^k$ = Real valued vector of predictions

E.g.

0.9	2.8	2.0	0.3	0.8
-----	-----	-----	-----	-----

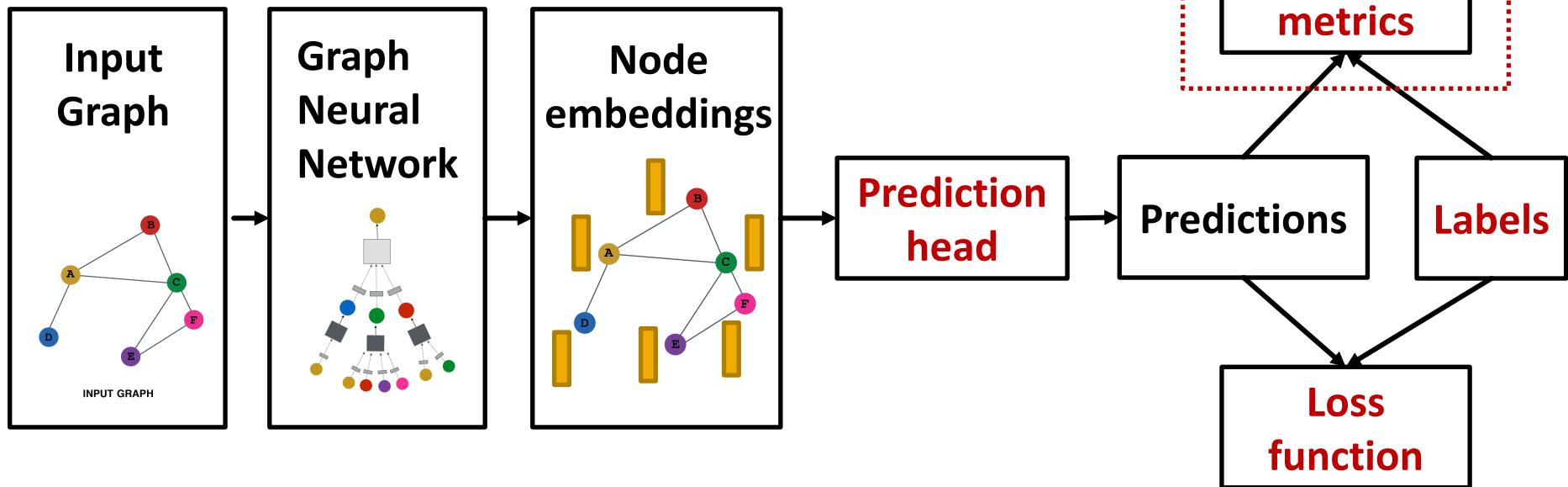
- Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^N \text{MSE}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$$

GNN Training Pipeline (4)

(4) How do we measure the success of a GNN?

- Accuracy
- ROC AUC



Evaluation Metrics: Regression

- We use standard evaluation metrics for GNN
 - (Content below can be found in any ML course)
 - In practice we will use [sklearn](#) for implementation
 - Suppose we make predictions for N data points
- Evaluate regression tasks on graphs:

- Root mean square error (RMSE)

$$\sqrt{\sum_{i=1}^N \frac{(\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2}{N}}$$

- Mean absolute error (MAE)

$$\frac{\sum_{i=1}^N |\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}|}{N}$$

Evaluation Metrics: Classification

- Evaluate classification tasks on graphs:
- (1) Multi-class classification

- We simply report the accuracy

$$\frac{1[\operatorname{argmax}(\hat{\mathbf{y}}^{(i)}) = \mathbf{y}^{(i)}]}{N}$$

- (2) Binary classification

- Metrics sensitive to classification threshold
 - Accuracy
 - Precision / Recall
 - If the range of prediction is [0,1], we will use 0.5 as threshold
 - Metric Agnostic to classification threshold
 - ROC AUC

Metrics for Binary Classification

- **Accuracy:**

$$\frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{|\text{Dataset}|}$$

- **Precision (P):**

$$\frac{TP}{TP + FP}$$

Confusion matrix

- **Recall (R):**

$$\frac{TP}{TP + FN}$$

- **F1-Score:**

$$\frac{2P * R}{P + R}$$

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

Sklearn Classification Report

(4) Evaluation Metrics

- **ROC Curve:** Captures the tradeoff in TPR and FPR as the classification threshold is varied for a binary classifier.

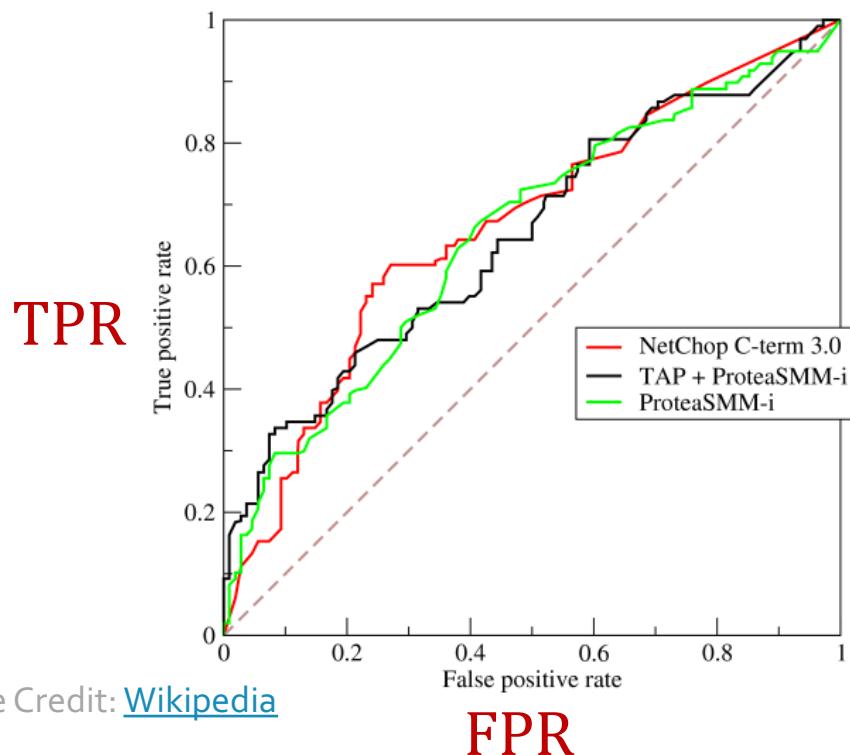


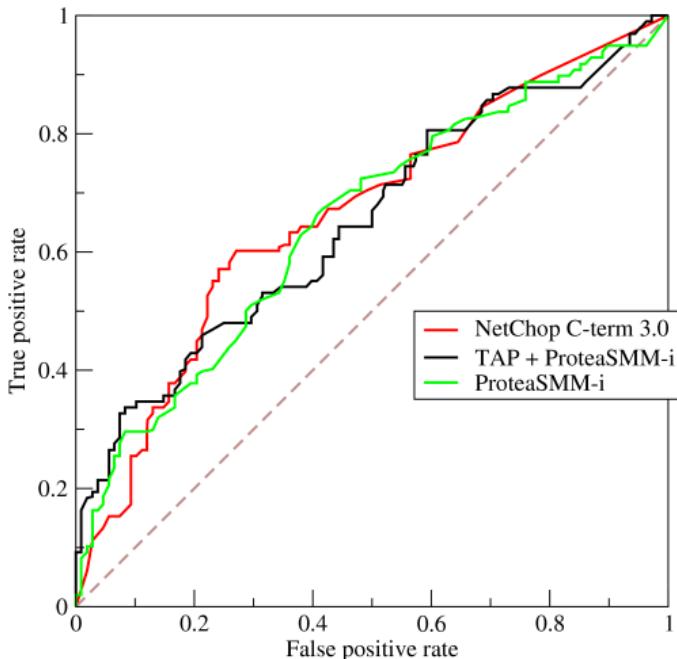
Image Credit: [Wikipedia](#)

$$\text{TPR} = \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

Note: the dashed line represents performance of a random classifier

(4) Evaluation Metrics



Content Credit: [Wikipedia](#)

- **ROC AUC: Area under the ROC Curve.**
- **Intuition:** The probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one

Stanford CS224W: Setting-up GNN Prediction Tasks

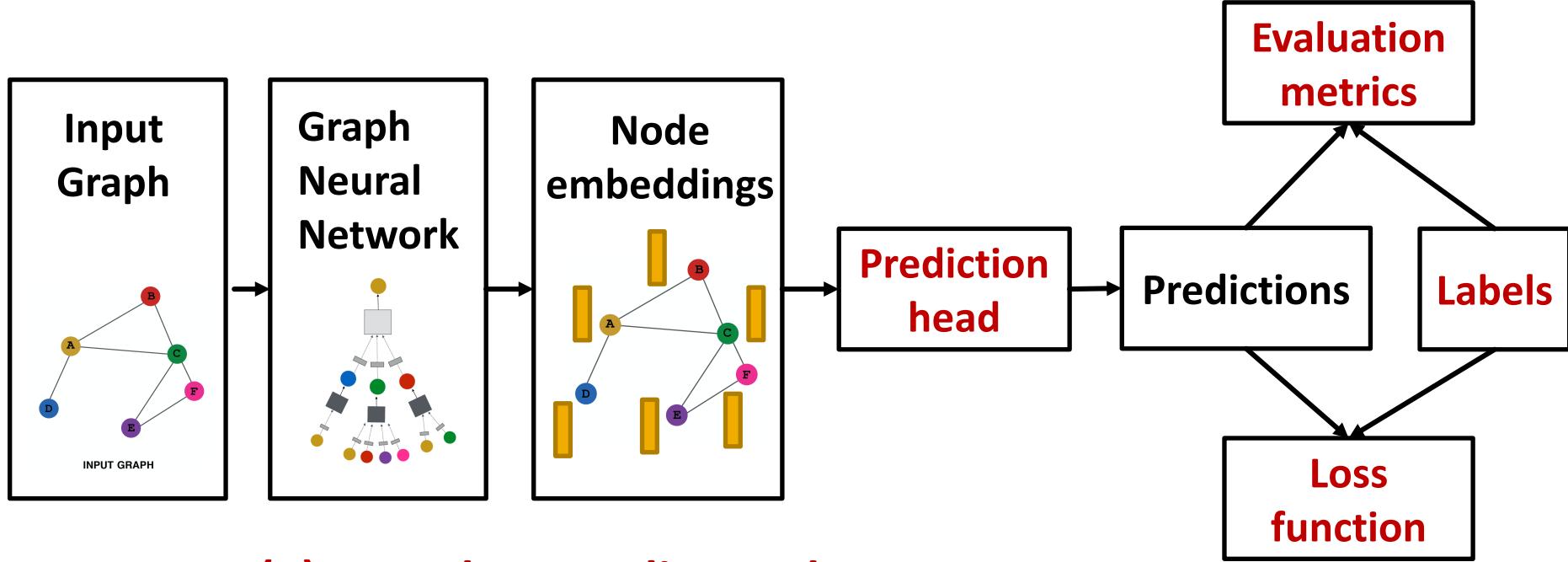
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

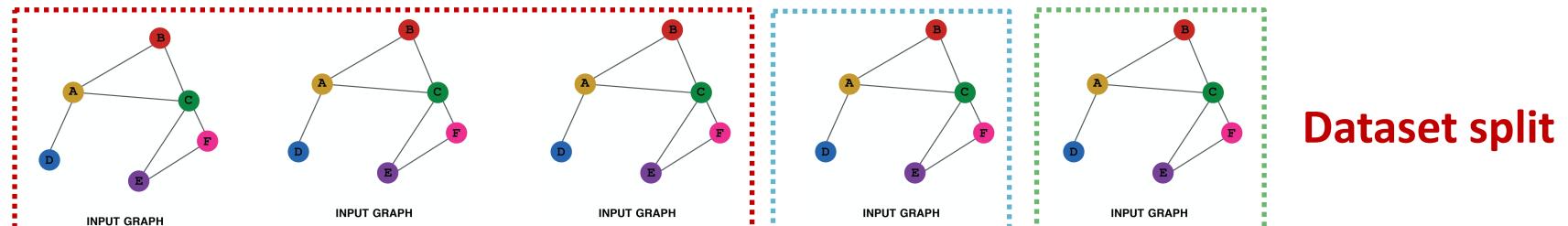
<http://cs224w.stanford.edu>



GNN Training Pipeline (5)



(5) How do we split our dataset into train / validation / test set?

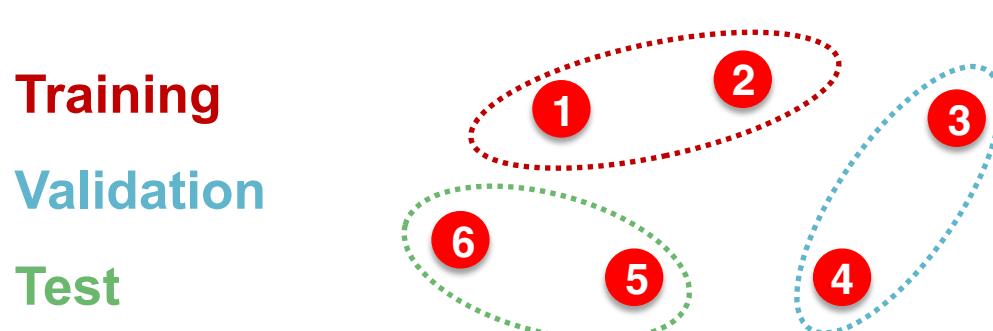


Dataset Split: Fixed / Random Split

- **Fixed split:** We will split our dataset **once**
 - **Training set:** used for optimizing GNN parameters
 - **Validation set:** develop model/hyperparameters
 - **Test set:** held out until we report final performance
- **A concern:** sometimes we cannot guarantee that the test set will really be held out
- **Random split:** we will **randomly split** our dataset into training / validation / test
 - We report **average performance over different random seeds**

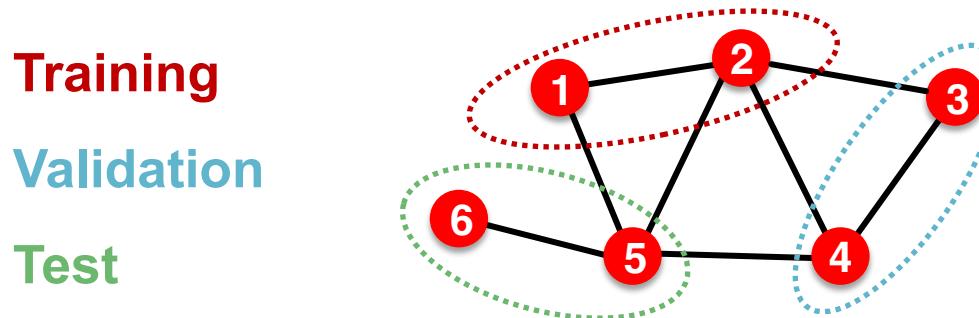
Why Splitting Graphs is Special

- Suppose we want to split an image dataset
 - **Image classification:** Each data point is an image
 - Here **data points are independent**
 - Image 5 will not affect our prediction on image 1



Why Splitting Graphs is Special

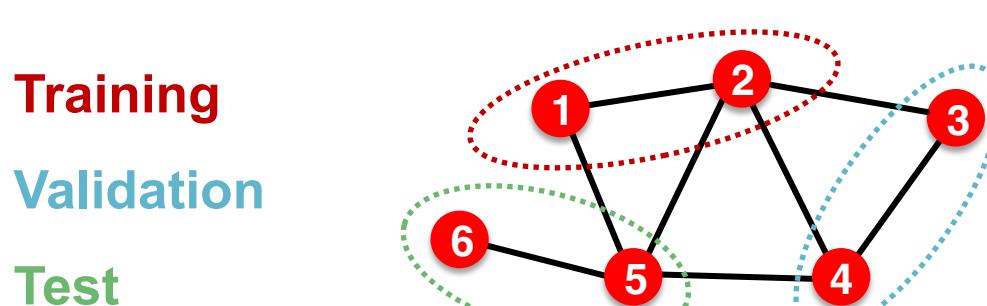
- **Splitting a graph dataset is different!**
 - **Node classification:** Each data point is a node
 - Here **data points are NOT independent**
 - Node 5 will affect our prediction on node 1, because it will participate in message passing → affect node 1's embedding



- **What are our options?**

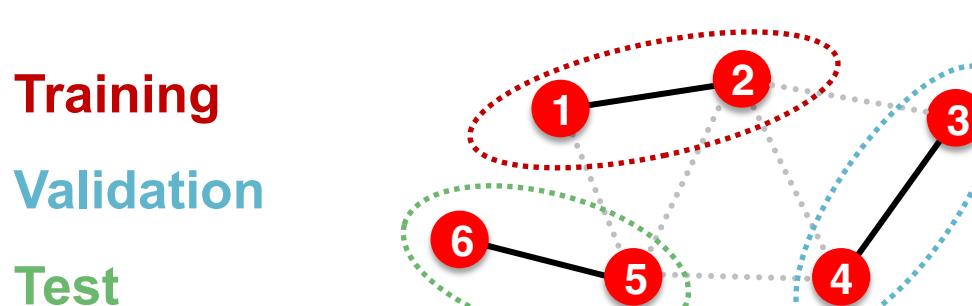
Why Splitting Graphs is Special

- **Solution 1 (Transductive setting): The input graph can be observed in all the dataset splits (training, validation and test set).**
- **We will only split the (node) labels**
 - At training time, we compute embeddings using the entire graph, and train using node 1&2's labels
 - At validation time, we compute embeddings using the entire graph, and evaluate on node 3&4's labels



Why Splitting Graphs is Special

- **Solution 2 (Inductive setting): We break the edges between splits to get multiple graphs**
 - Now we have 3 graphs that are independent. Node 5 will not affect our prediction on node 1 any more
 - At training time, we compute embeddings using the graph over node 1&2, and train using node 1&2's labels
 - At validation time, we compute embeddings using the graph over node 3&4, and evaluate on node 3&4's labels

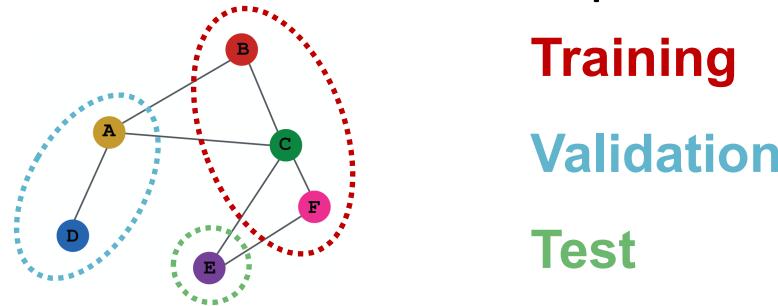


Transductive / Inductive Settings

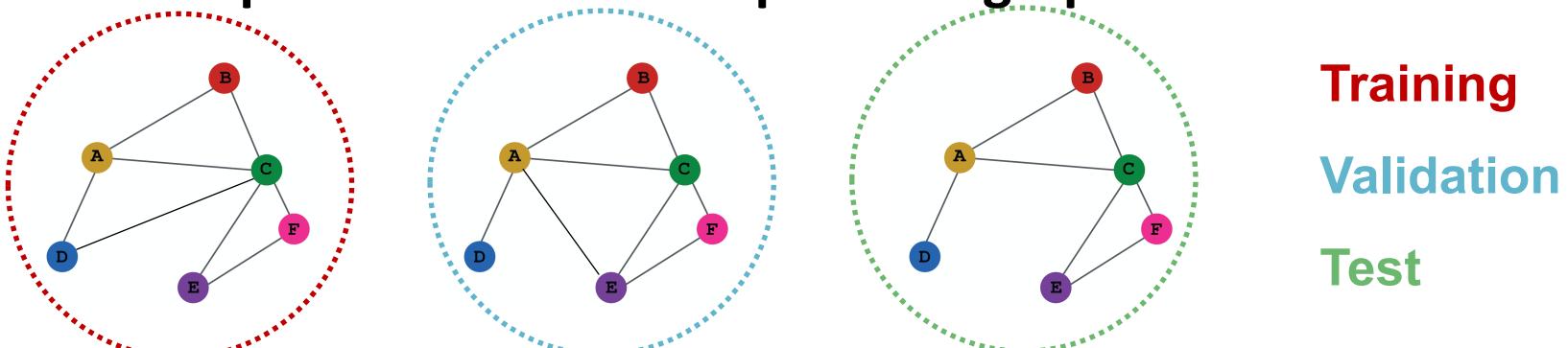
- **Transductive setting:** training / validation / test sets are **on the same graph**
 - The **dataset consists of one graph**
 - **The entire graph can be observed in all dataset splits, we only split the labels**
 - Only applicable to **node / edge** prediction tasks
- **Inductive setting:** training / validation / test sets are **on different graphs**
 - The **dataset consists of multiple graphs**
 - Each split can **only observe the graph(s) within the split.** A successful model should **generalize to unseen graphs**
 - Applicable to **node / edge / graph** tasks

Example: Node Classification

- **Transductive node classification**
 - All the splits can observe the entire graph structure, but can only observe the labels of their respective nodes

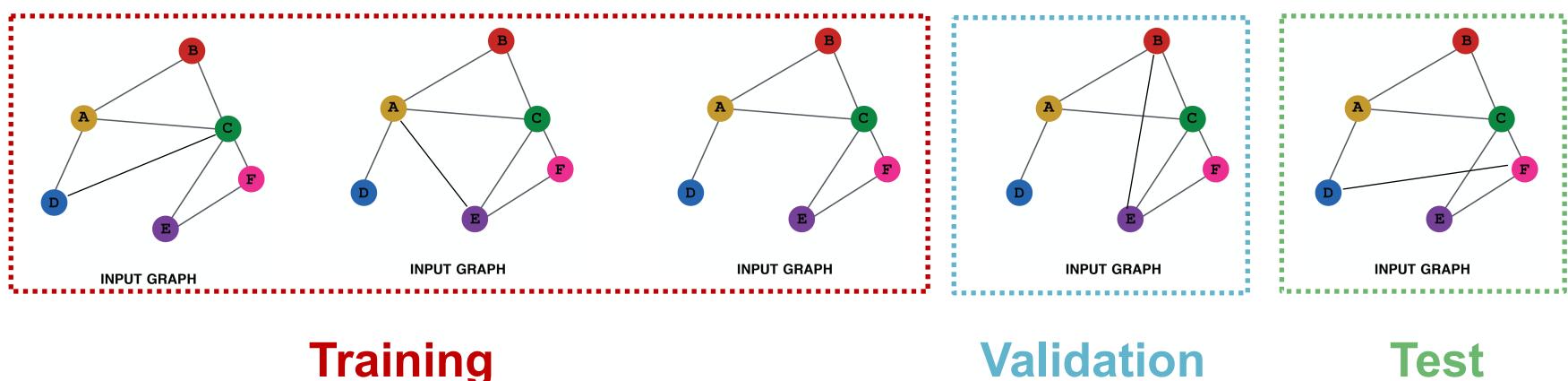


- **Inductive node classification**
 - Suppose we have a dataset of 3 graphs
 - **Each split contains an independent graph**



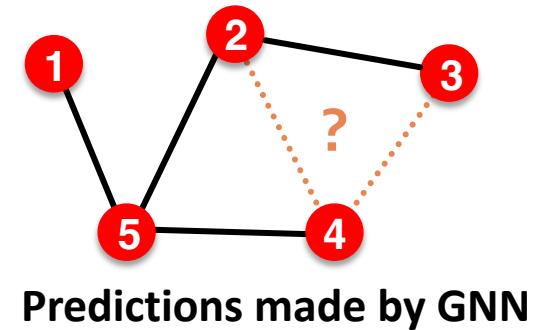
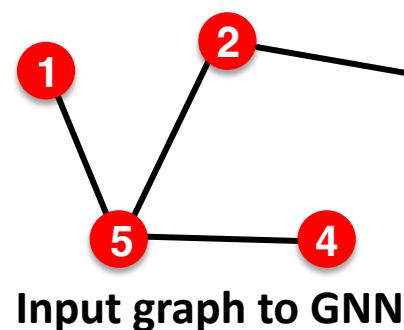
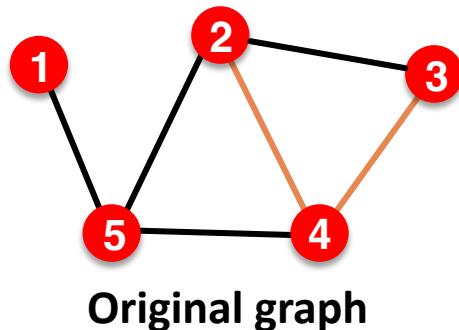
Example: Graph Classification

- Only the **inductive setting** is well defined for **graph classification**
 - Because **we have to test on unseen graphs**
 - Suppose we have a dataset of 5 graphs. Each split will contain independent graph(s).

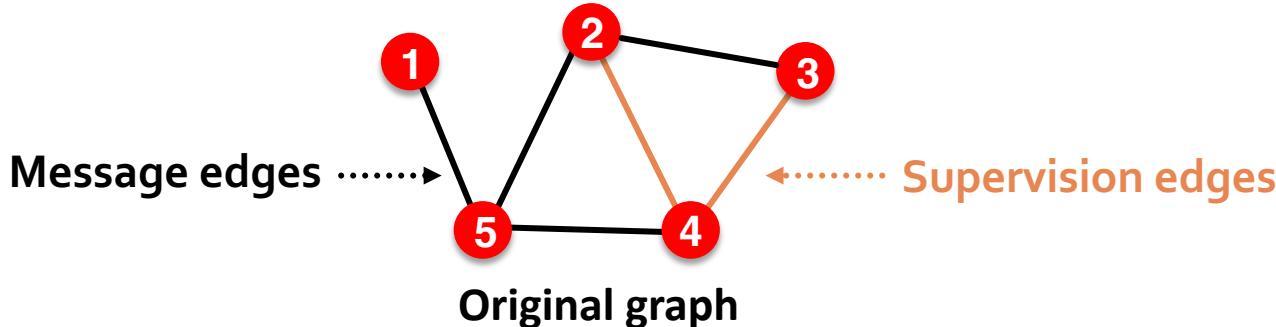


Example: Link Prediction

- **Goal of link prediction:** predict missing edges
- **Setting up link prediction is tricky:**
 - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
 - Concretely, we need to **hide some edges** from the **GNN** and let the **GNN** predict if the edges exist



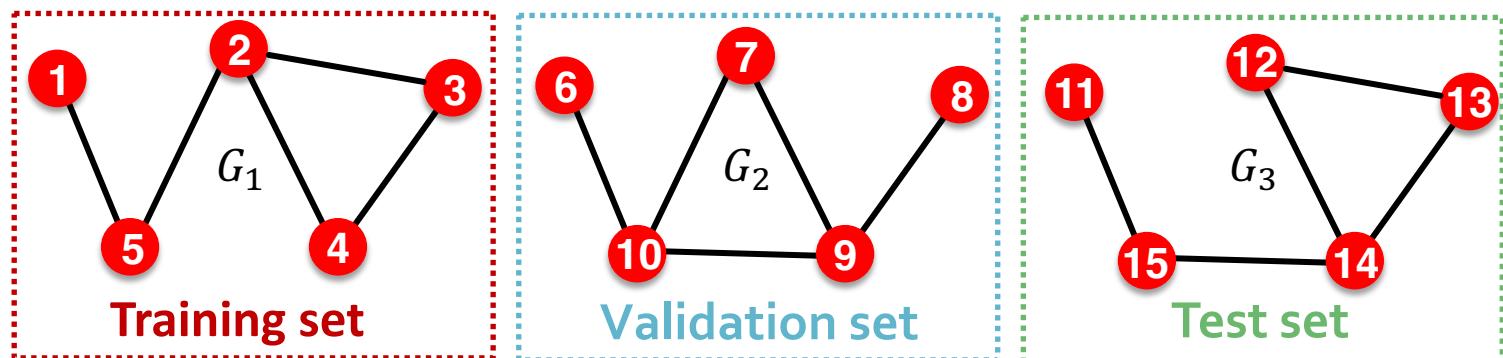
Setting up Link Prediction



- For link prediction, we will split edges twice
- Step 1: Assign 2 types of edges in the original graph
 - Message edges: Used for GNN message passing
 - Supervision edges: Use for computing objectives
- After step 1:
 - Only message edges will remain in the graph
 - Supervision edges are used as supervision for edge predictions made by the model, will not be fed into GNN!

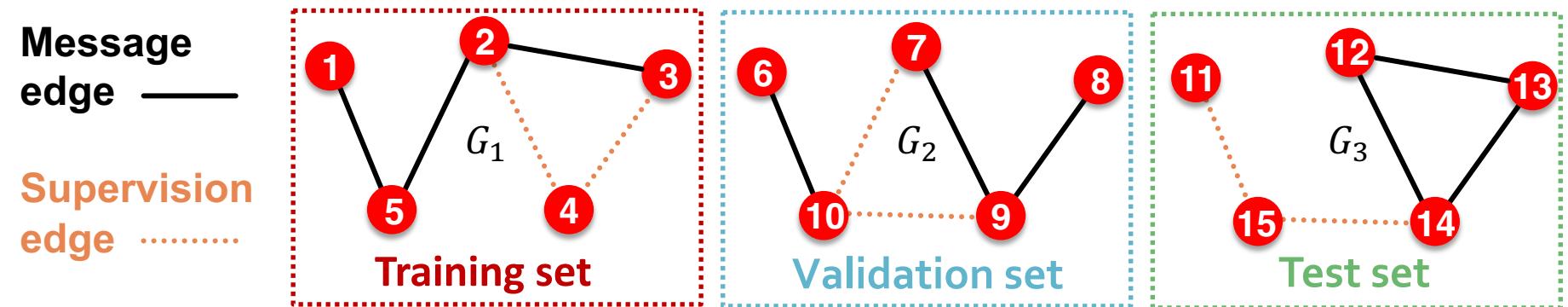
Setting up Link Prediction

- Step 2: Split edges into train / validation / test
- Option 1: Inductive link prediction split
 - Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph



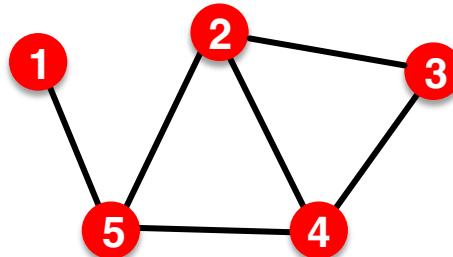
Setting up Link Prediction

- Step 2: Split edges into train / validation / test
- Option 1: Inductive link prediction split
 - Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph
 - In **train** or **val** or **test** set, each graph will have **2 types of edges: message edges + supervision edges**
 - **Supervision edges** are not the input to GNN



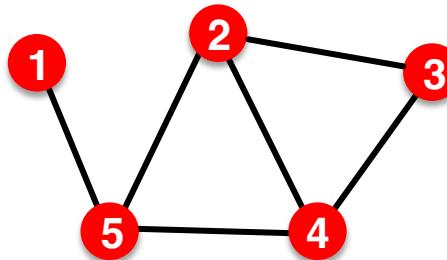
Setting up Link Prediction

- **Option 2: Transductive link prediction split:**
 - This is the default setting when people talk about link prediction
 - Suppose we have a dataset of 1 graph



Setting up Link Prediction

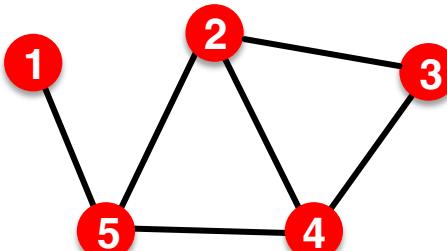
- **Option 2: Transductive link prediction split:**
 - By definition of “transductive”, the entire graph can be observed in all dataset splits
 - But since edges are both part of graph structure and the supervision, we need to hold out validation / test edges
 - To train the training set, we further need to hold out supervision edges for the training set



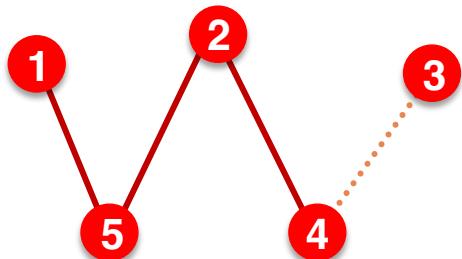
- **Next:** we will show the exact settings

Setting up Link Prediction

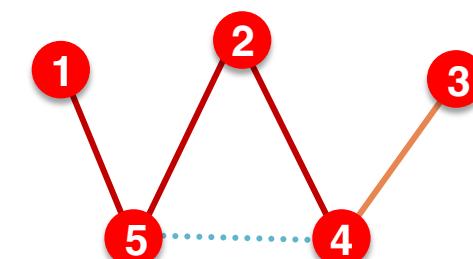
■ Option 2: Transductive link prediction split:



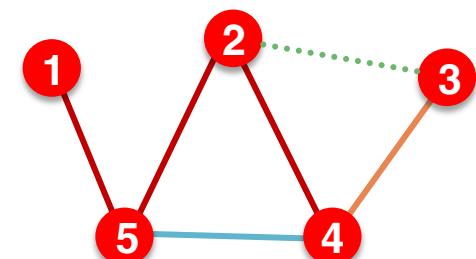
The original graph



(1) At training time:
Use **training message edges** to predict **training supervision edges**



(2) At validation time:
Use **training message edges & training supervision edges** to predict **validation edges**



(3) At test time:
Use **training message edges & training supervision edges & validation edges** to predict **test edges**

Setting up Link Prediction

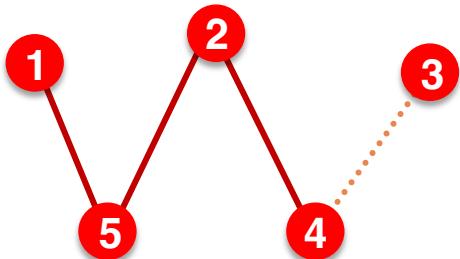
■ Option 2: Transductive link prediction split:

Why do we use growing number of edges?

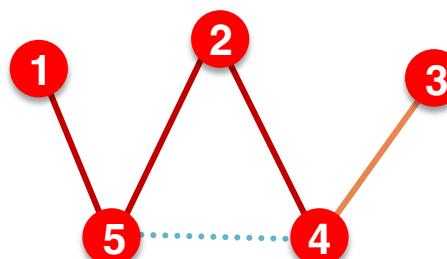
After training, supervision edges are known to GNN.

Therefore, an ideal model should use supervision edges in message passing at validation time.

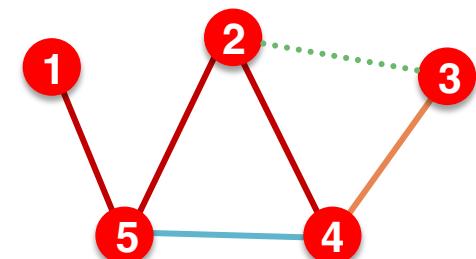
The same applies to the test time.



(1) At training time:
Use **training message edges** to predict **training supervision edges**



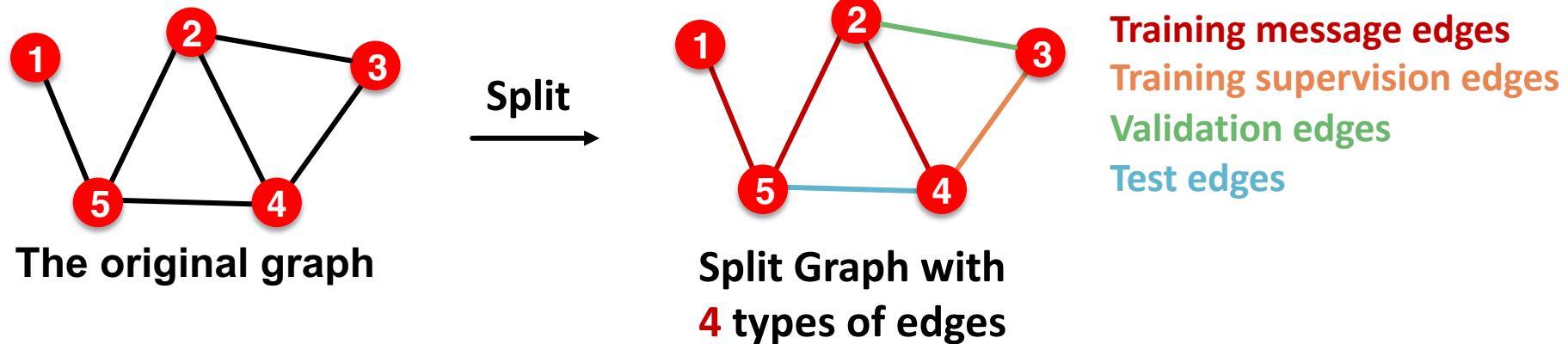
(2) At validation time:
Use **training message edges & training supervision edges** to predict **validation edges**



(3) At test time:
Use **training message edges & training supervision edges & validation edges** to predict **test edges**

Setting up Link Prediction

■ Summary: Transductive link prediction split:



- **Note:** Link prediction settings are tricky and complex. You may find papers do link prediction differently.
- Luckily, we have full support in PyG and [GraphGym](#)