

Stanford CS224W: GNN Augmentation and Training

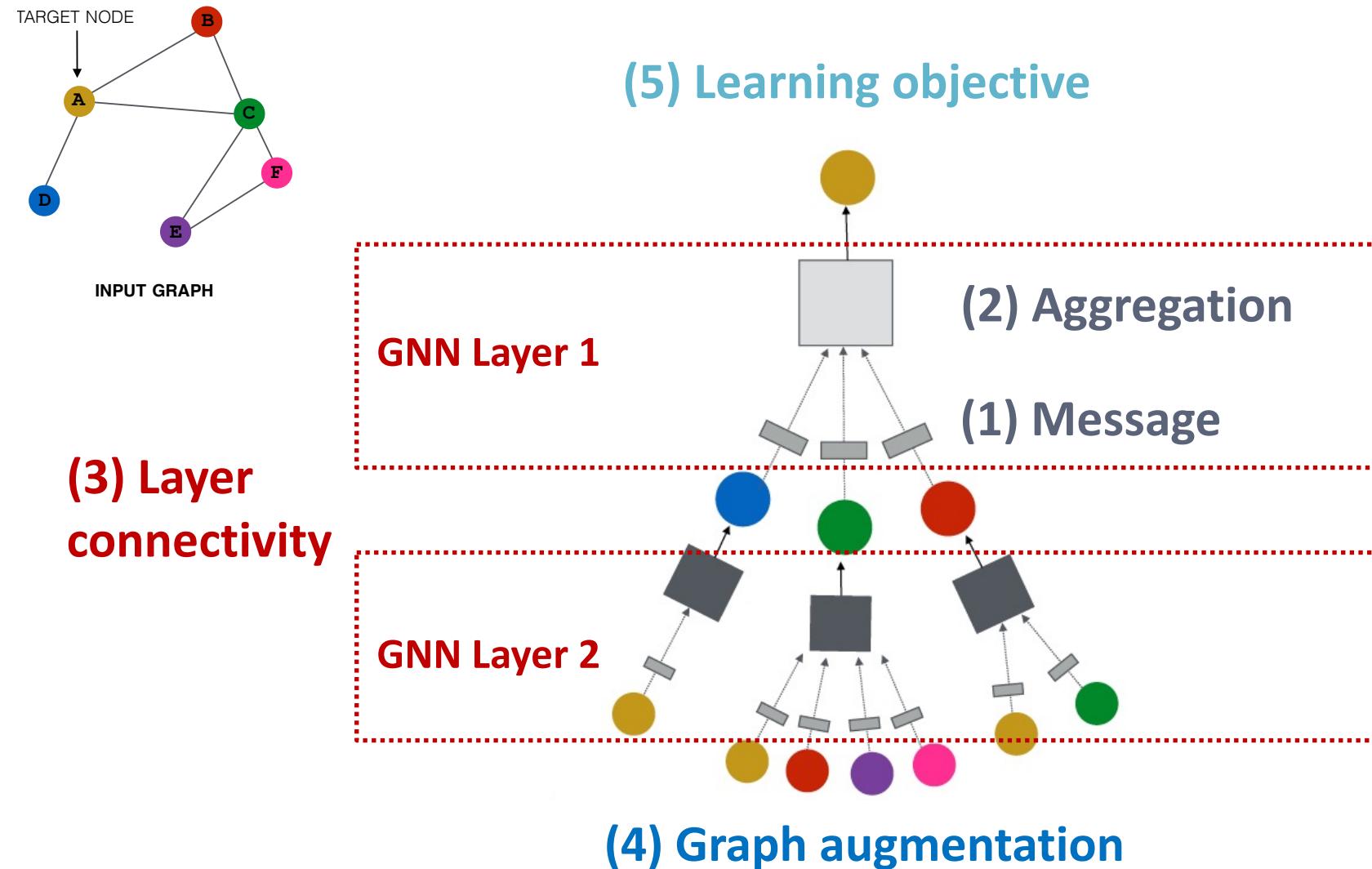
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



Recap: A General GNN Framework



Stanford CS224W: Stacking Layers of a GNN

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

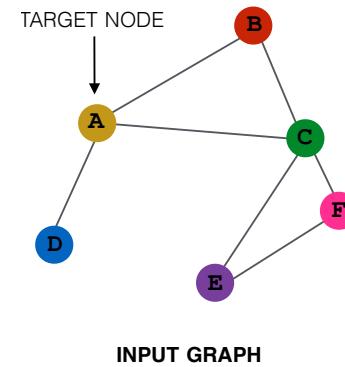
<http://cs224w.stanford.edu>



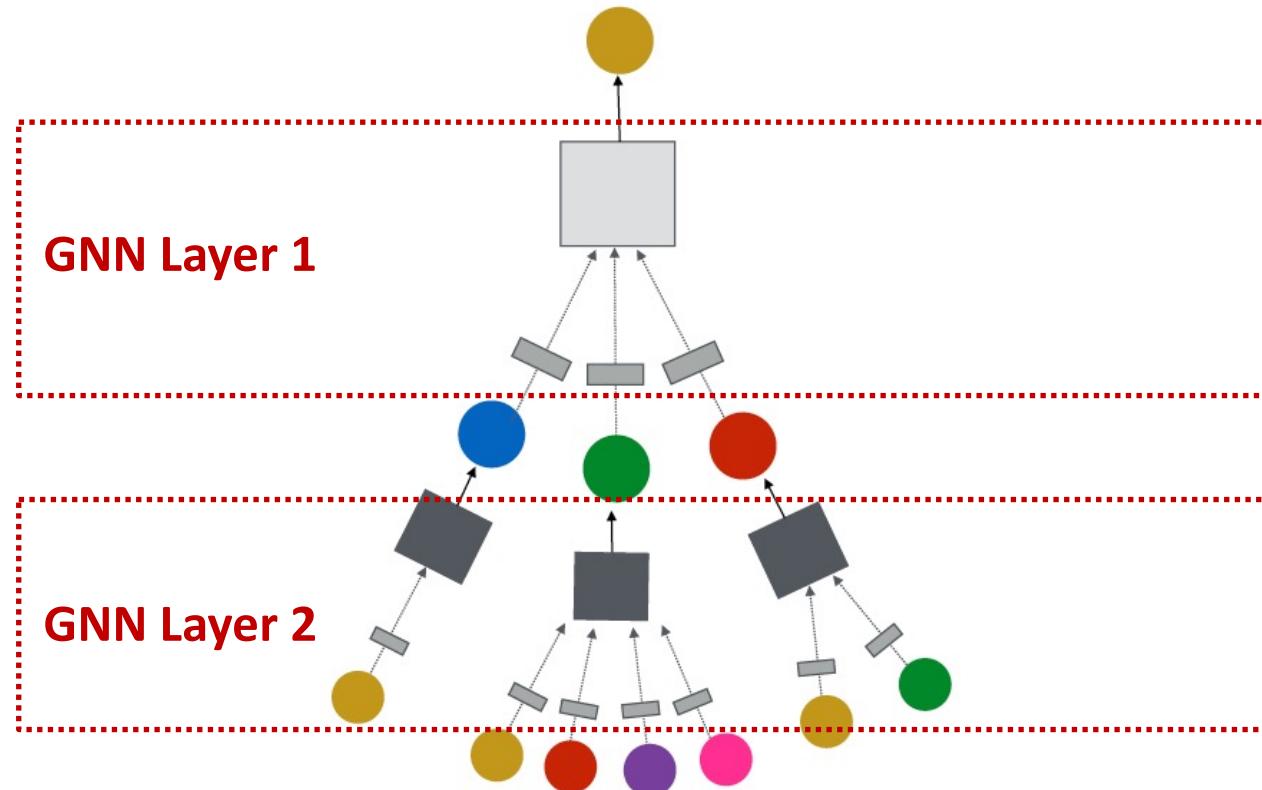
Stacking GNN Layers

How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections

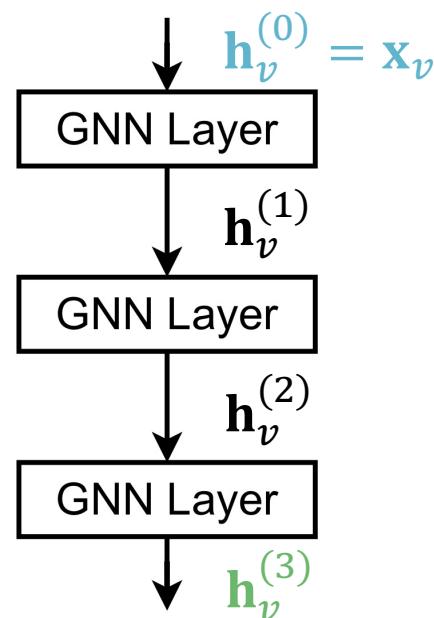


(3) Layer connectivity



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



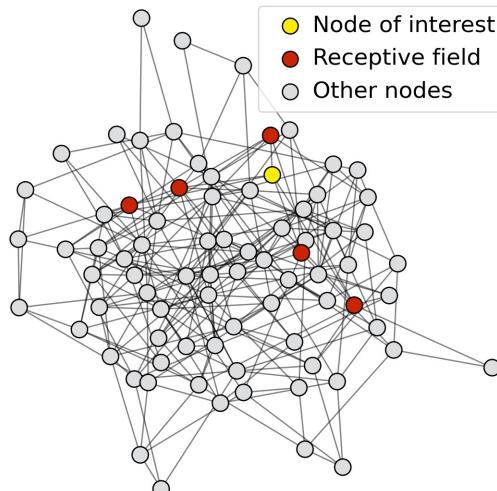
The Over-smoothing Problem

- **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?**

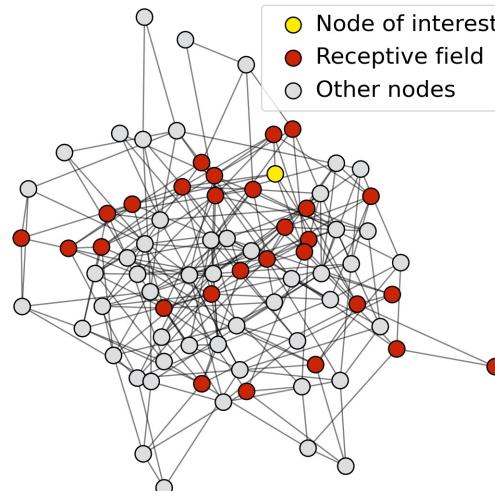
Receptive Field of a GNN

- **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

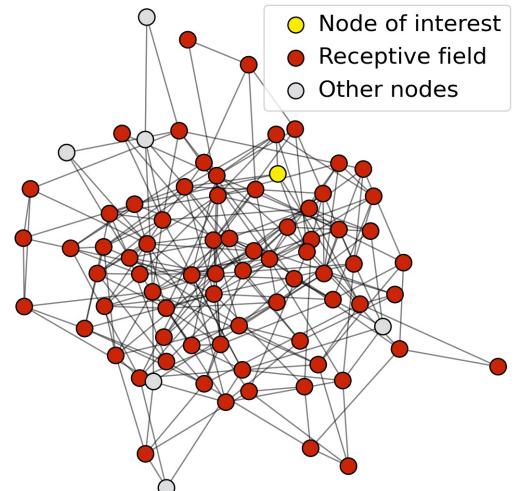
Receptive field for
1-layer GNN



Receptive field for
2-layer GNN



Receptive field for
3-layer GNN

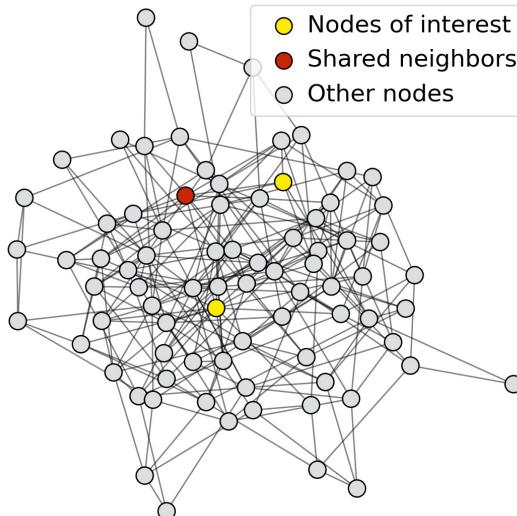


Receptive Field of a GNN

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

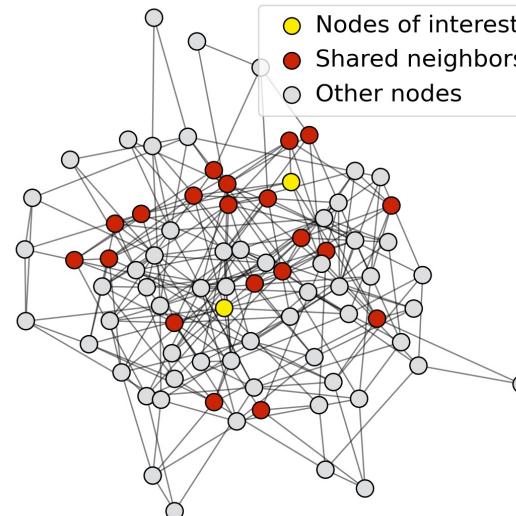
1-hop neighbor overlap

Only 1 node



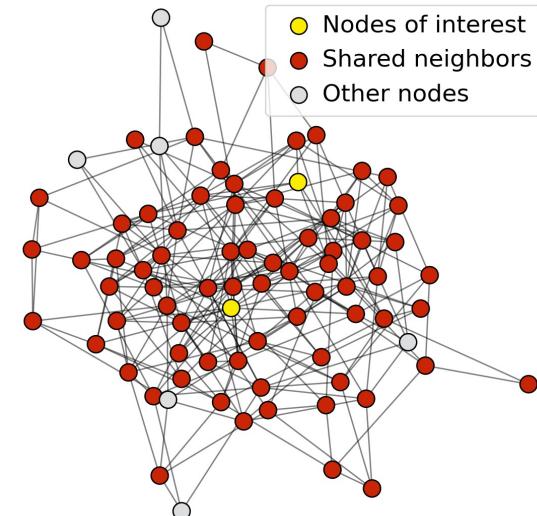
2-hop neighbor overlap

About 20 nodes



3-hop neighbor overlap

Almost all the nodes!



Receptive Field & Over-smoothing

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

Design GNN Layer Connectivity

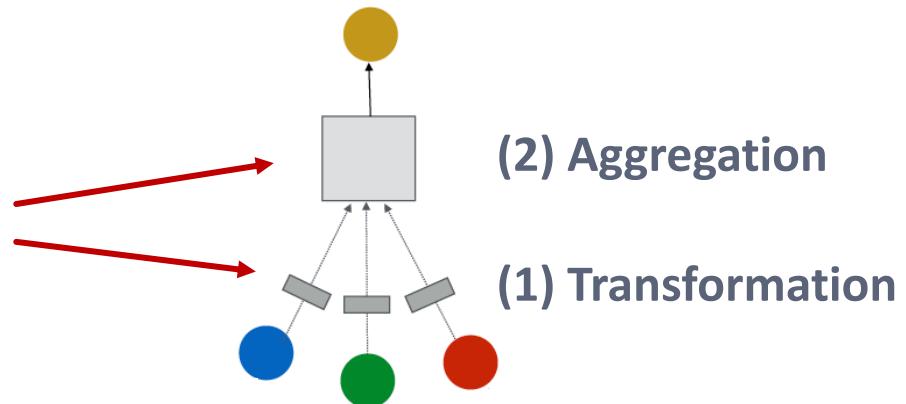
- What do we learn from the over-smoothing problem?
- Lesson 1: Be cautious when adding GNN layers
 - Unlike neural networks in other domains (CNN for image classification), adding more GNN layers do not always help
 - Step 1: Analyze the necessary receptive field to solve your problem. E.g., by computing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

Graph diameter: maximum distance between the pair of vertices

Expressive Power for Shallow GNNs

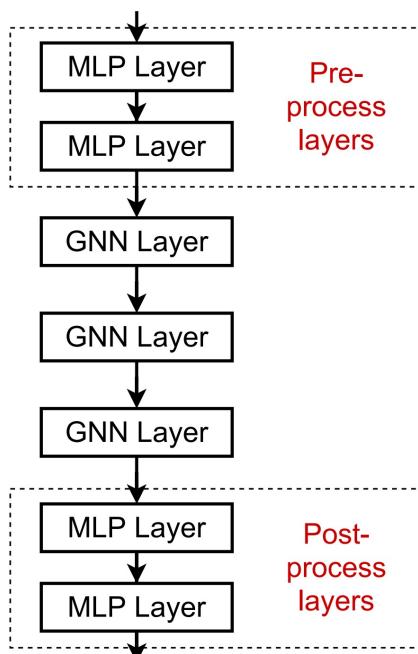
- How to make a shallow GNN more expressive?
- Solution 1: Increase the expressive power within each GNN layer
 - In our previous examples, each transformation or aggregation function only include one linear layer
 - We can make aggregation / transformation become a deep neural network!

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**



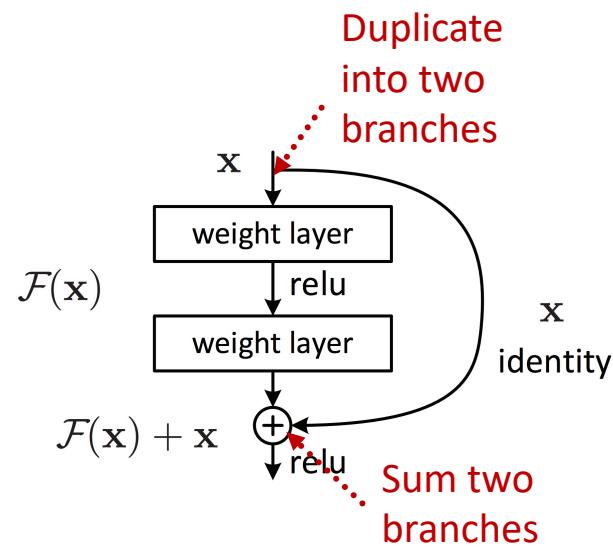
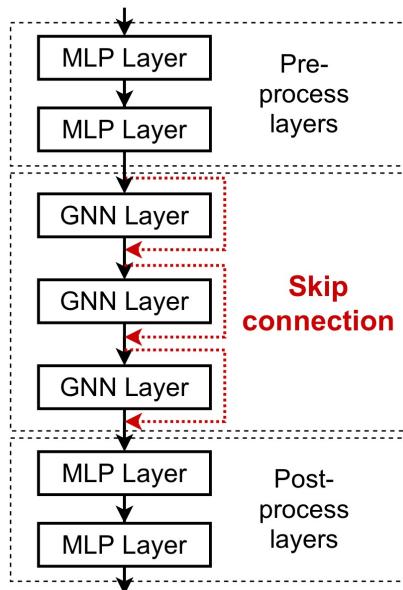
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

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:

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

Example: GCN with Skip Connections

- A standard GCN layer

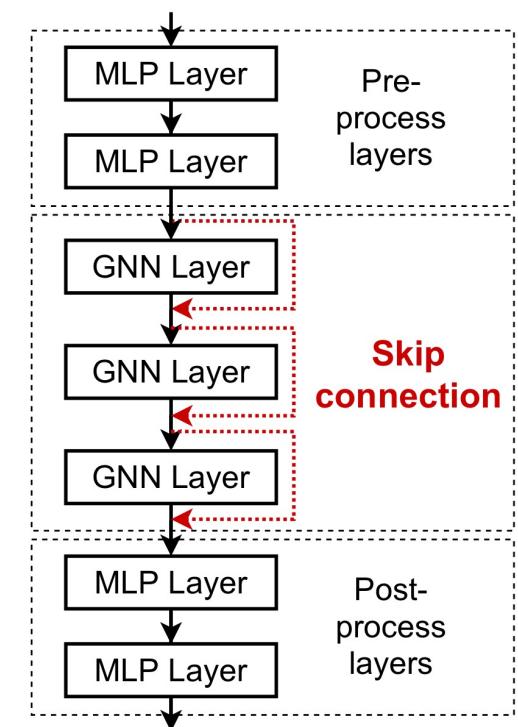
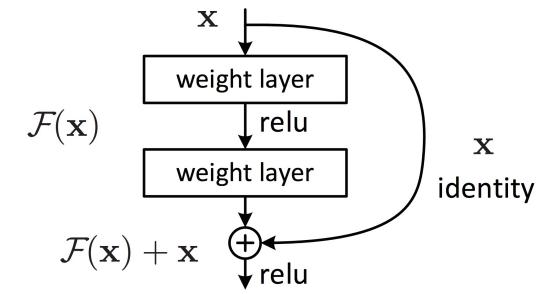
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

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

- A GCN layer with skip connection

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

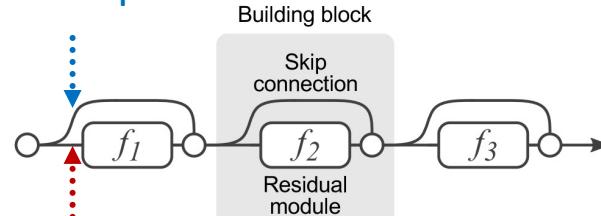
$F(\mathbf{x})$ + \mathbf{x}



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

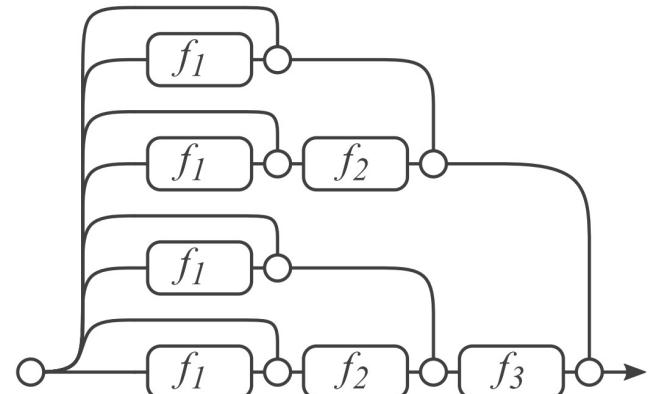


Path 1: include this module

(a) Conventional 3-block residual network

All the possible paths:

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

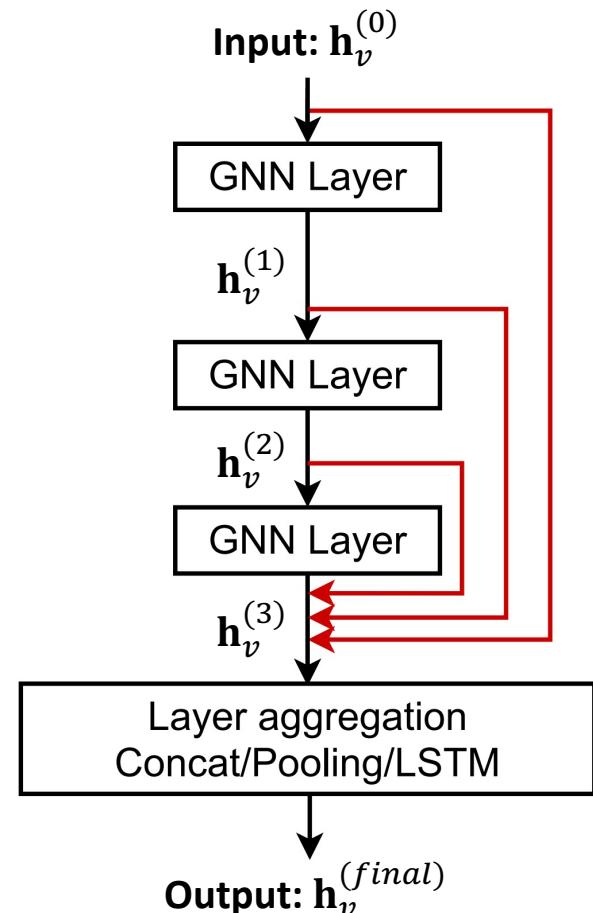


(b) Unraveled view of (a)

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

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 Augmentation for GNNs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

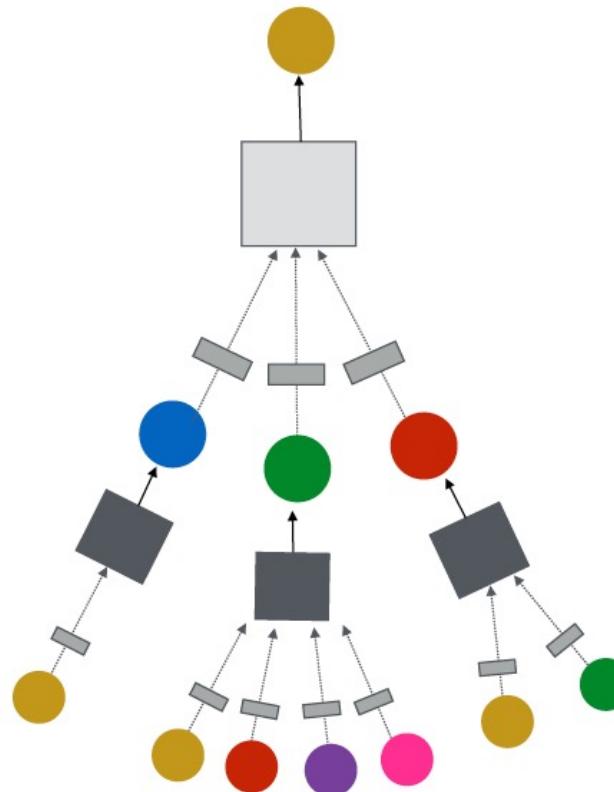
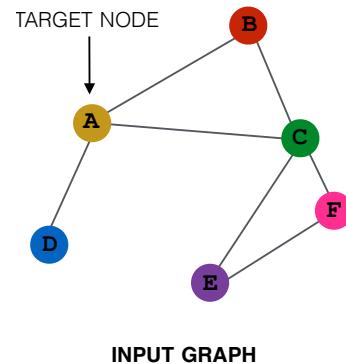
<http://cs224w.stanford.edu>



General GNN Framework

Idea: Raw input graph \neq computational graph

- Graph feature augmentation
- Graph structure augmentation



(4) Graph augmentation

Why Augment Graphs

Our assumption so far has been

- Raw input graph = computational graph

Reasons for breaking this assumption

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

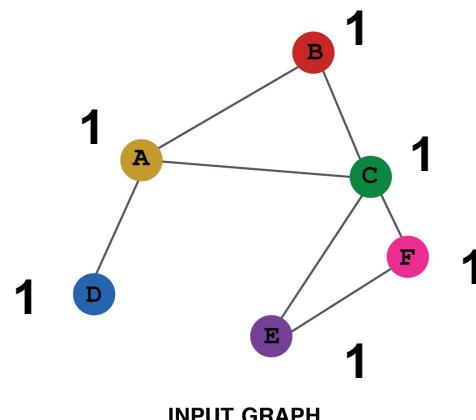
Graph Augmentation Approaches

- **Graph Feature augmentation**
 - The input graph lacks features → **feature augmentation**
- **Graph Structure augmentation**
 - The graph is **too sparse** → **Add virtual nodes / edges**
 - The graph is **too dense** → **Sample neighbors when doing message passing**
 - The graph is **too large** → **Sample subgraphs to compute embeddings**
 - Will cover later in lecture: Scaling up GNNs

Feature Augmentation on Graphs

Why do we need feature augmentation?

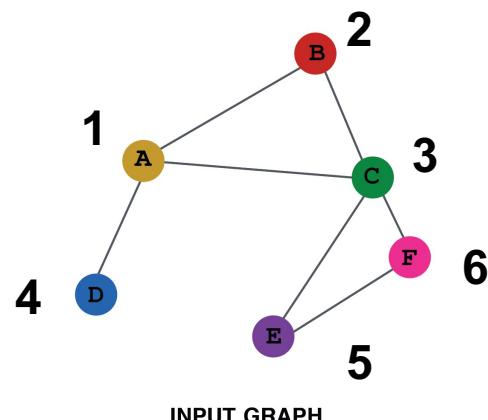
- **(1) Input graph does not have node features**
 - This is common when we only have the adj. matrix
- **Standard approaches:**
- **a) Assign constant values to nodes**



Feature Augmentation on Graphs

Why do we need feature augmentation?

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



One-hot vector for node with ID=5

ID = 5
↓
[0, 0, 0, 0, 1, 0]

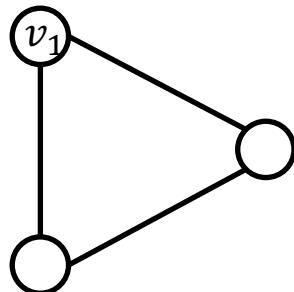
Total number of IDs = 6

Feature Augmentation on Graphs

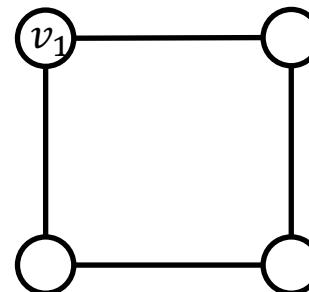
Why do we need feature augmentation?

- **(2) Certain structures are hard to learn by GNN**
- **Example:** Cycle count feature:
 - Can GNN learn the length of a cycle that v_1 resides in?
 - **Unfortunately, no**

v_1 resides in a cycle with length 3



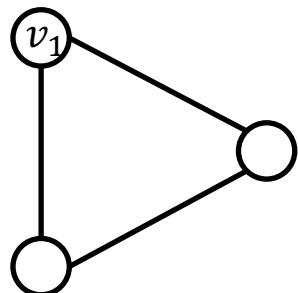
v_1 resides in a cycle with length 4



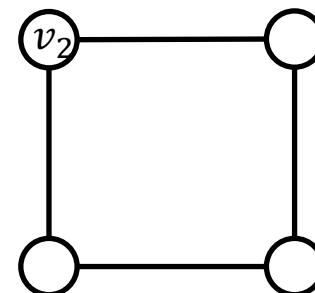
Feature Augmentation on Graphs

- v_1 cannot differentiate which graph it resides in
 - Because all the nodes in the graph have degree of 2
 - The computational graphs will be the same binary tree

v_1 resides in a cycle with length 3



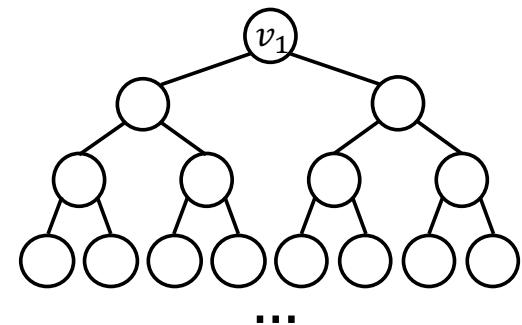
v_1 resides in a cycle with length 4



v_1 resides in a cycle with infinite length



The computational graphs for node v_1 are always the same



More about this topic later!

Feature Augmentation on Graphs

Why do we need feature augmentation?

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

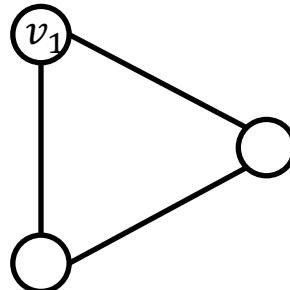
We start
from cycle
with length 0

Augmented node feature for v_1

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



v_1 resides in a cycle with length 3

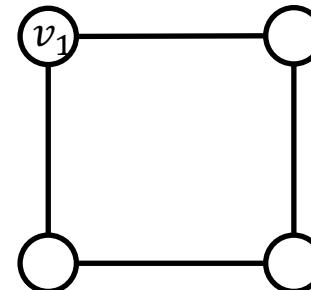


Augmented node feature for v_1

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



v_1 resides in a cycle with length 4



Feature Augmentation on Graphs

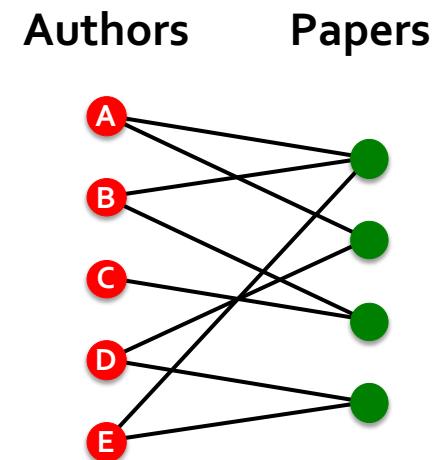
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- Other commonly used augmented features:
 - Node degree
 - Clustering coefficient
 - PageRank
 - Centrality
 - ...
- Any feature we have introduced in Lecture 2 can be used!

Add Virtual Nodes / Edges

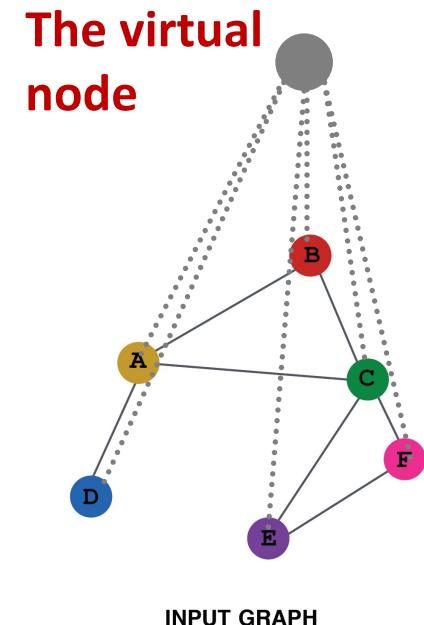
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
 - **Common approach:** Connect 2-hop neighbors via virtual edges
 - **Intuition:** Instead of using adj. matrix A for GNN computation, use $A + A^2$

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



Add Virtual Nodes / Edges

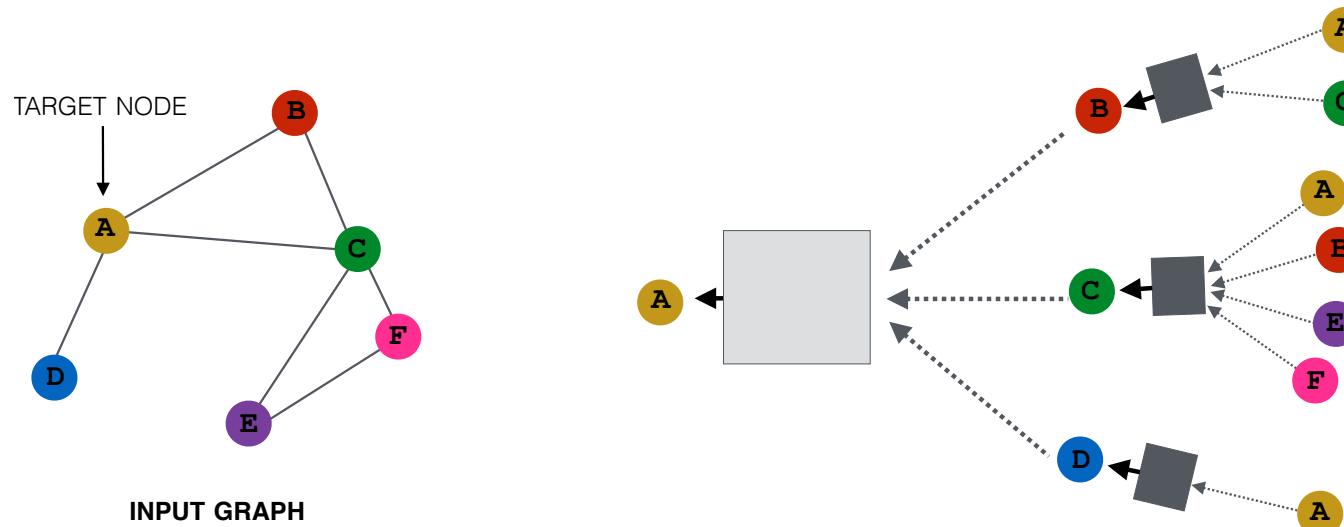
- **Motivation:** Augment sparse graphs
- **(2) Add virtual nodes**
 - The virtual node will connect to all the nodes in the graph
 - Suppose in a sparse graph, two nodes have shortest path distance of 10
 - After adding the virtual node, **all the nodes will have a distance of two**
 - Node A – Virtual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



Node Neighborhood Sampling

- Previously:

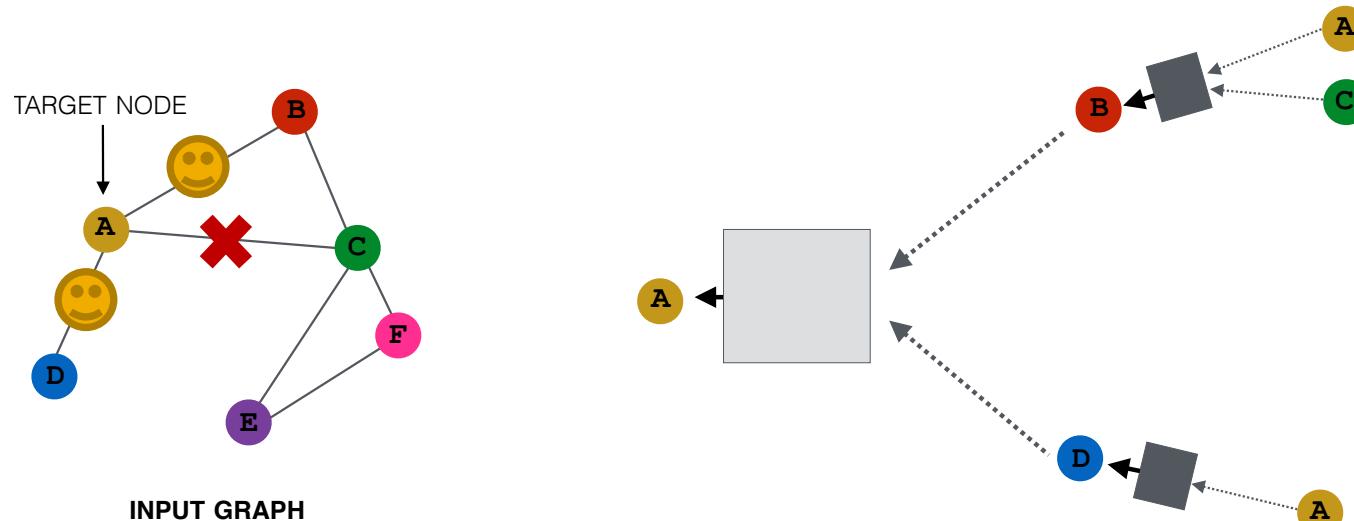
- All the nodes are used for message passing



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

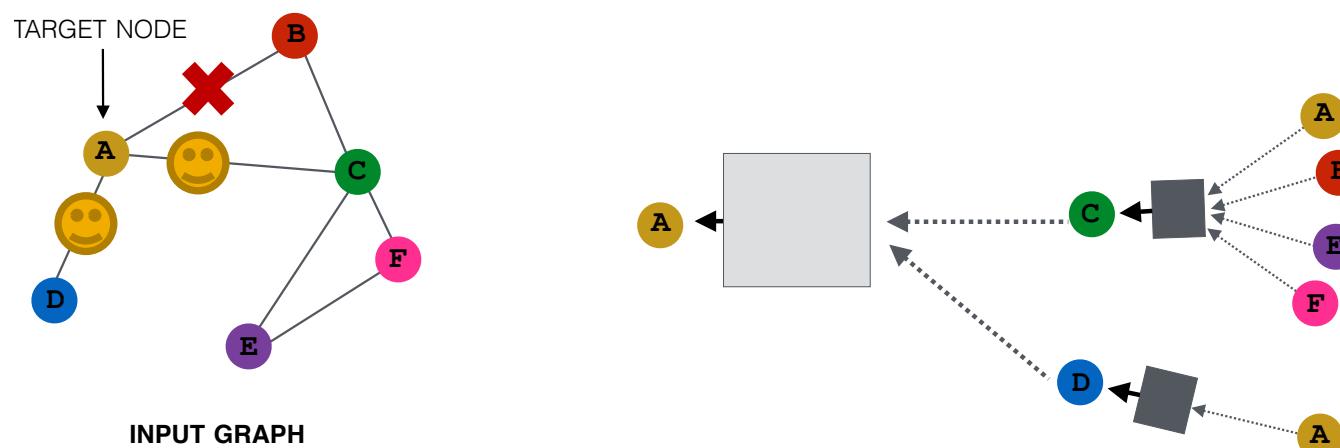
Neighborhood Sampling Example

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



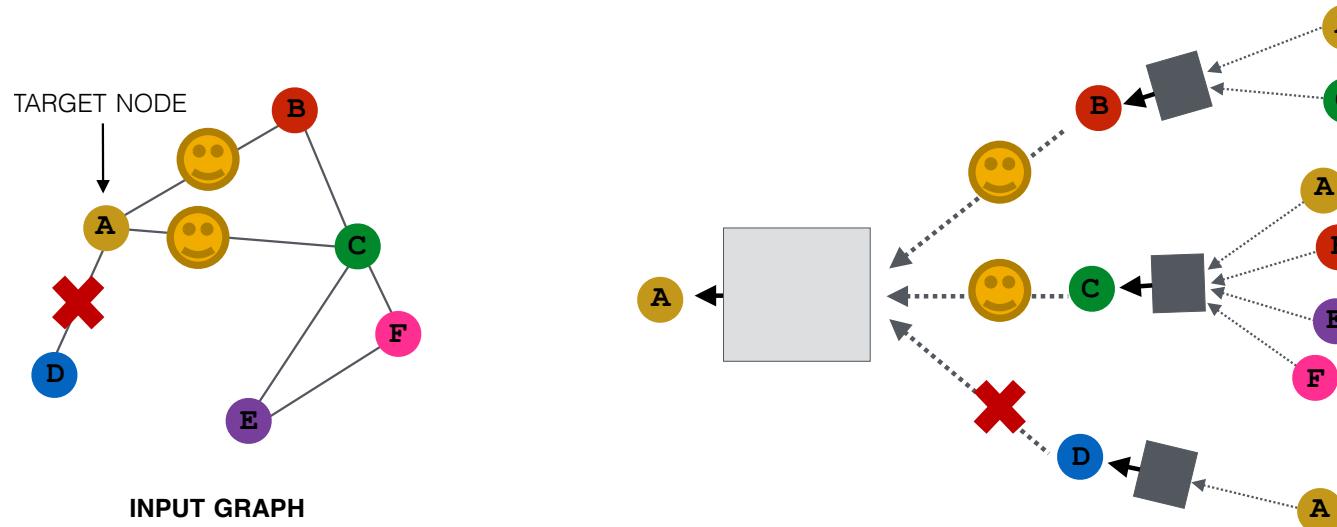
Neighborhood Sampling Example

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



Neighborhood Sampling Example

- In expectation, we get embeddings similar to the case where all the neighbors are used
 - Benefits: Greatly reduces computational cost
 - Allows for scaling to large graphs (more about this later)
 - And in practice it works great!



Stanford CS224W: Prediction with GNNs

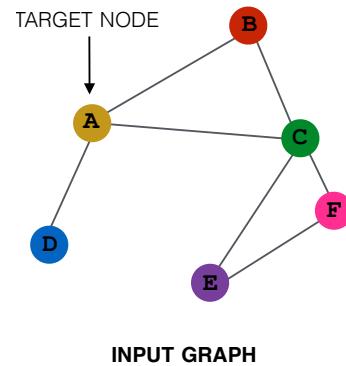
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

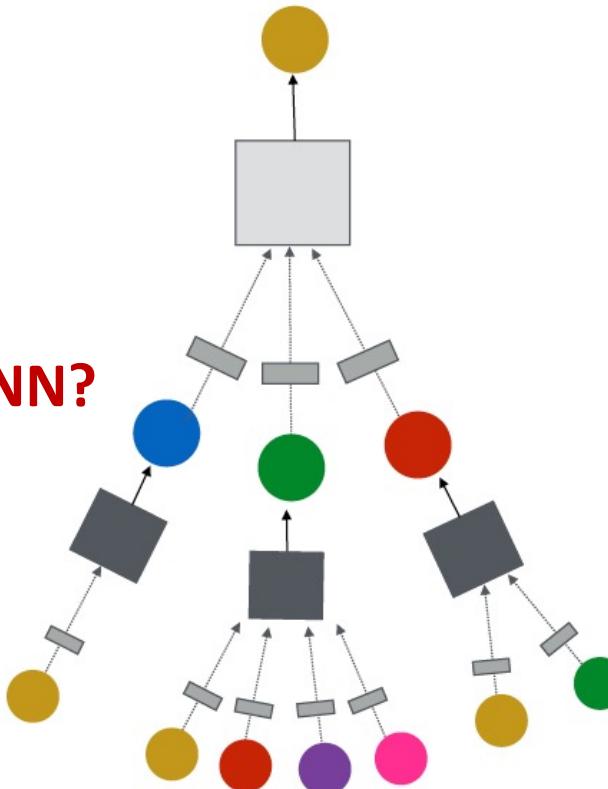
<http://cs224w.stanford.edu>



A General GNN Framework (4)



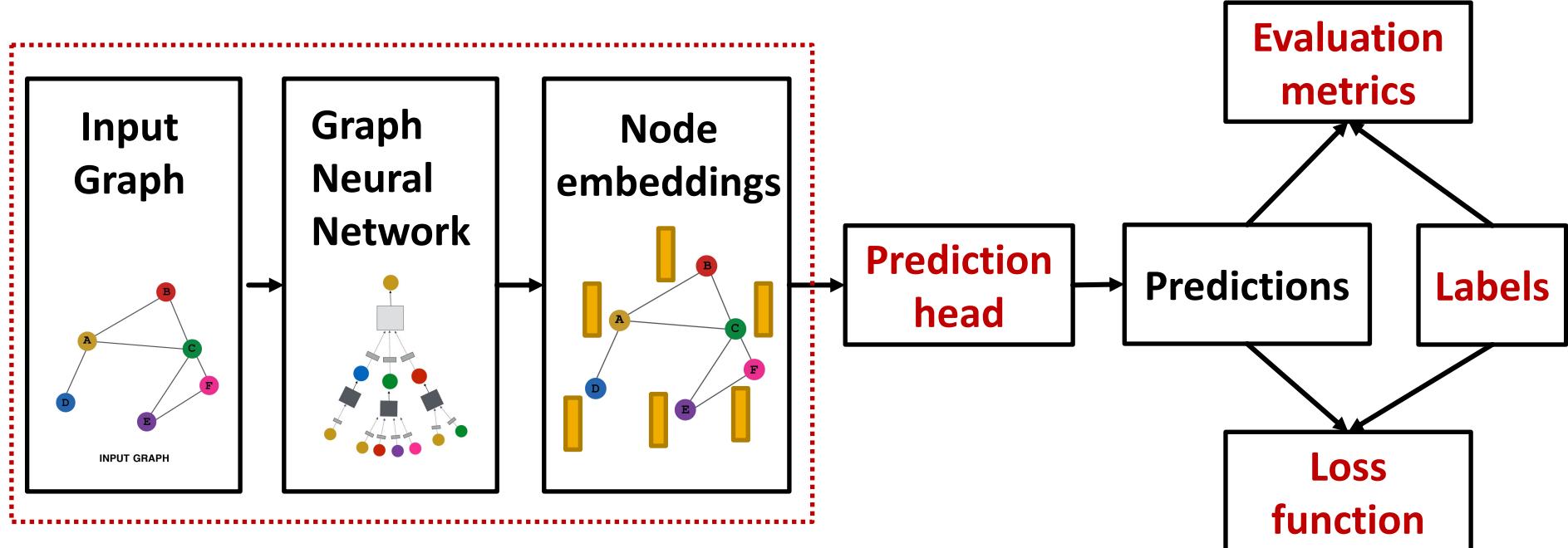
(5) Learning objective



Next: How do we train a GNN?

GNN Training Pipeline

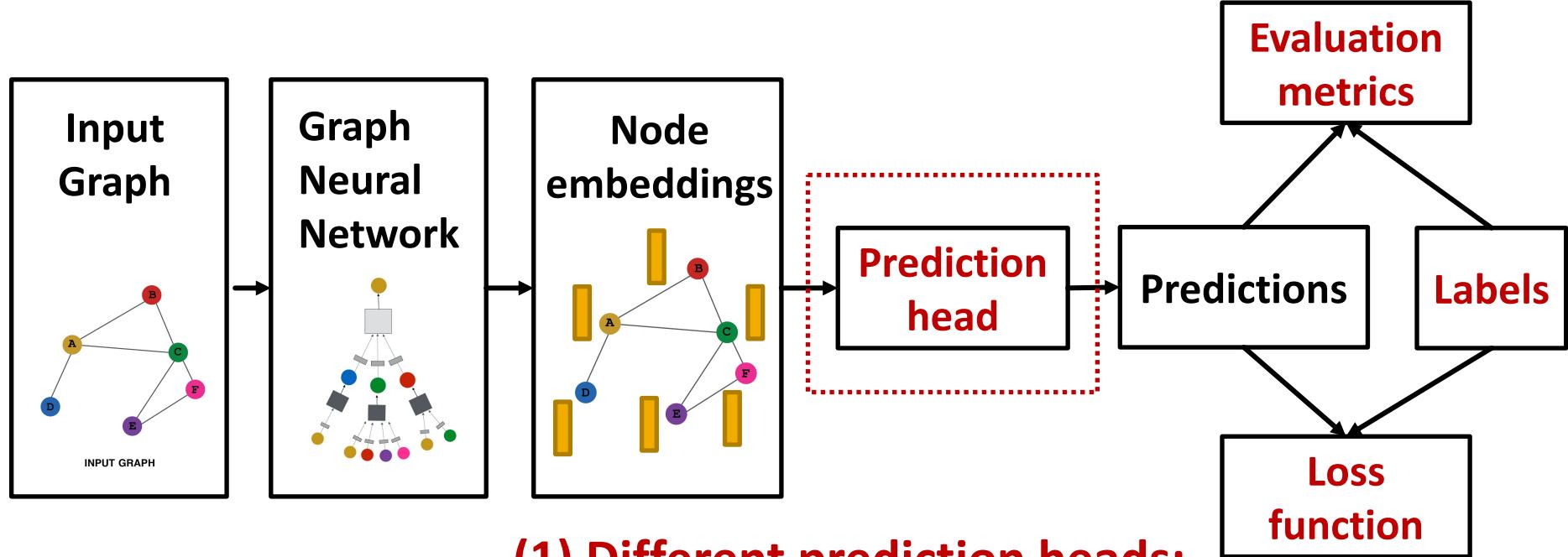
So far what we have covered



Output of a GNN: set of node embeddings

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

GNN Training Pipeline (1)

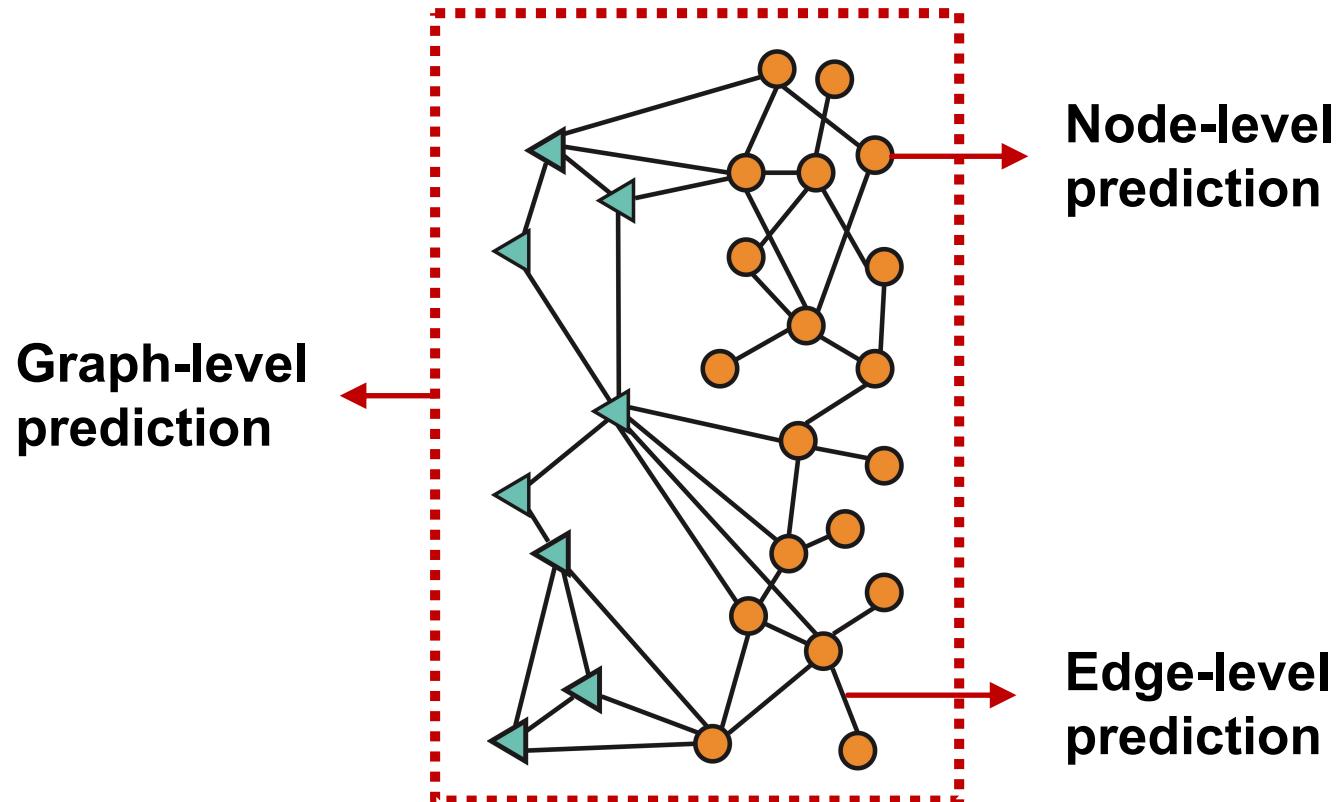


(1) Different prediction heads:

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

GNN Prediction Heads

- Idea: Different task levels require different prediction heads

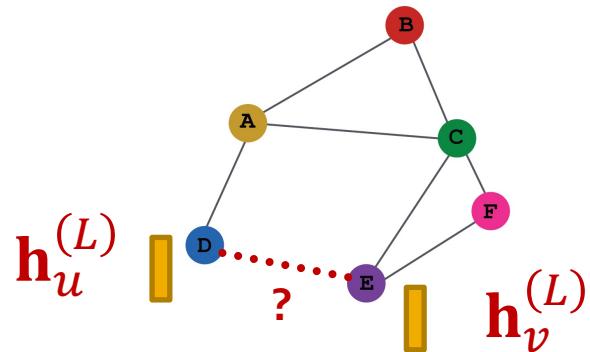


Prediction Heads: Node-level

- **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
- $\hat{y}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)} \mathbf{h}_v^{(L)}$
 - $\mathbf{W}^{(H)} \in \mathbb{R}^{k*d}$: 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

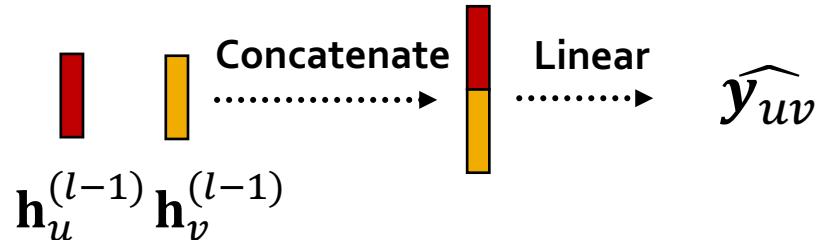
- **Edge-level prediction:** Make prediction using pairs of node embeddings
- Suppose we want to make k -way prediction
- $\hat{y}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



- 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



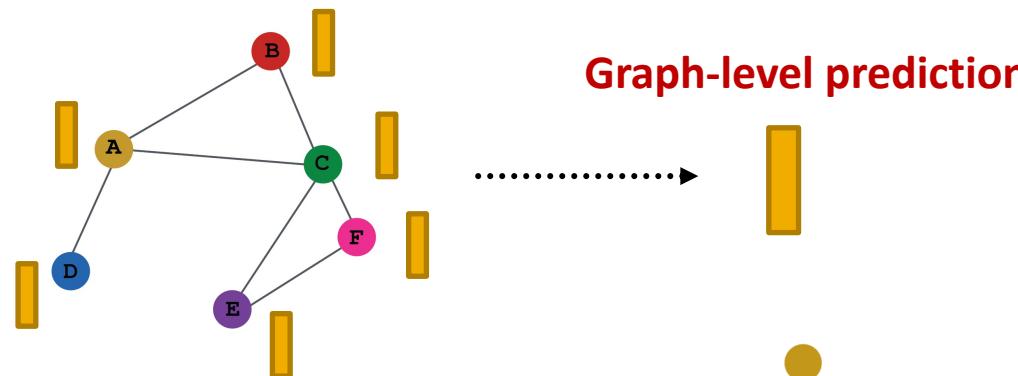
- $\hat{y}_{uv} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
- Here $\text{Linear}(\cdot)$ will map **2d-dimensional** embeddings (since we concatenated embeddings) to **k-dim** embeddings (k -way prediction)

Prediction Heads: Edge-level

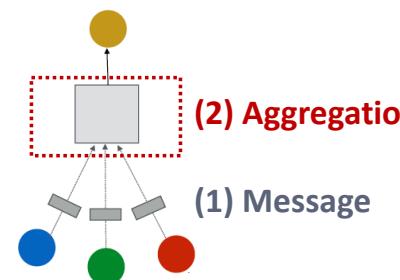
- Options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- **(2) Dot product**
 - $\hat{y}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
 - This approach only applies to **1-way prediction** (e.g., link prediction: predict the existence of an edge)
 - Applying to **k -way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

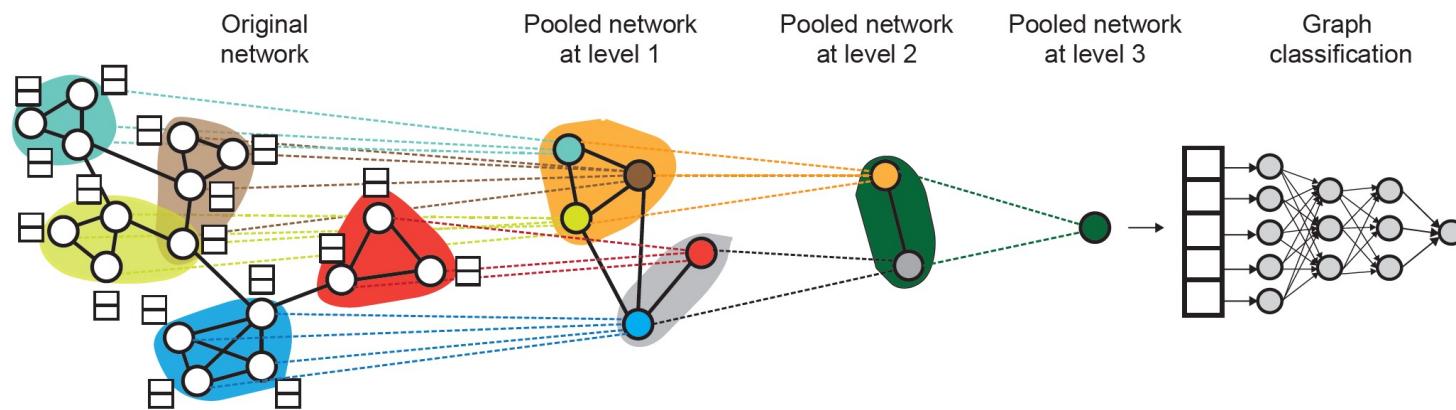
Hierarchical Global Pooling

- **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

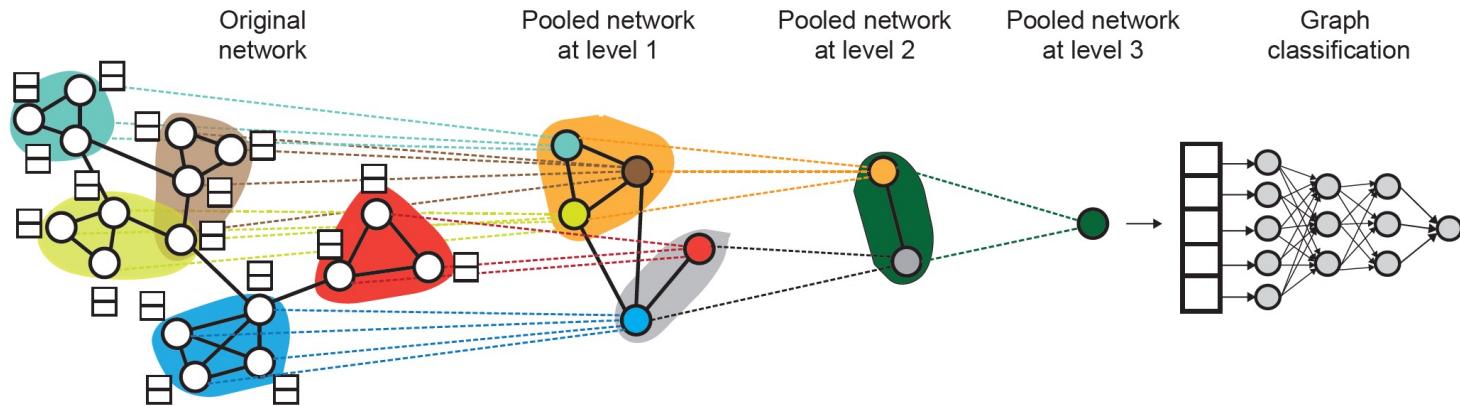
- DiffPool idea:
 - Hierarchically pool node embeddings



- 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**

Stanford CS224W: Training Graph Neural Networks

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

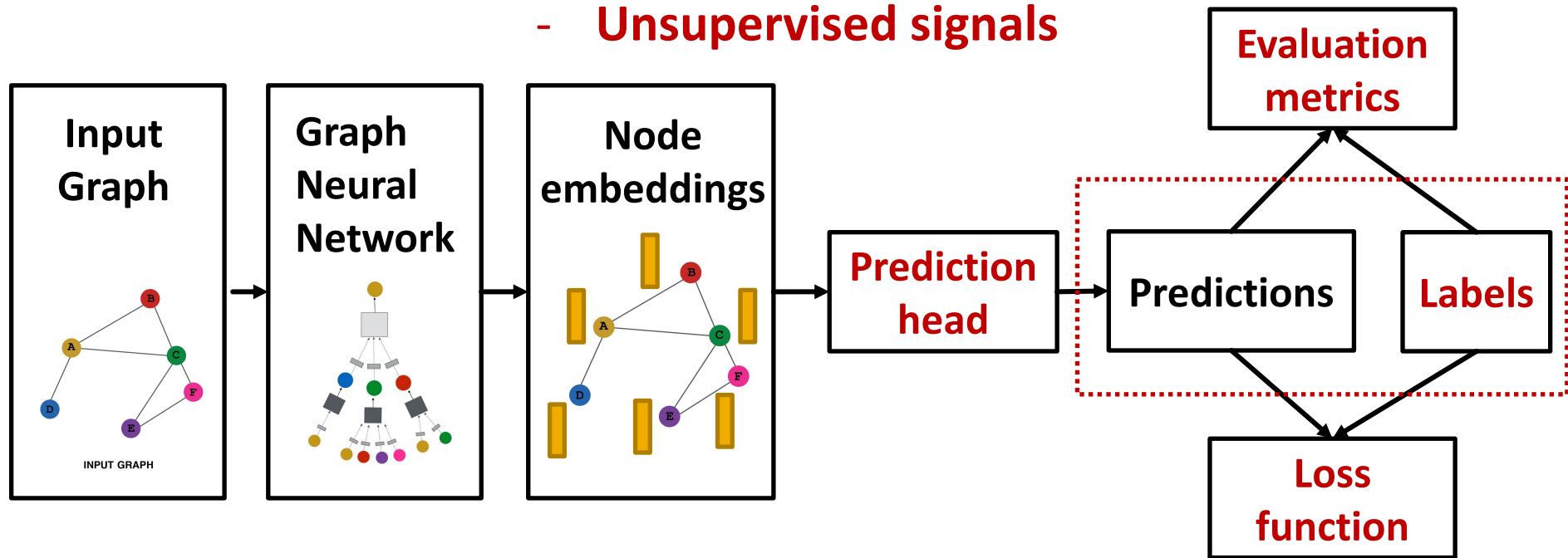
<http://cs224w.stanford.edu>



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**”

clustering coefficient: the degree to which nodes in a graph tend to cluster together

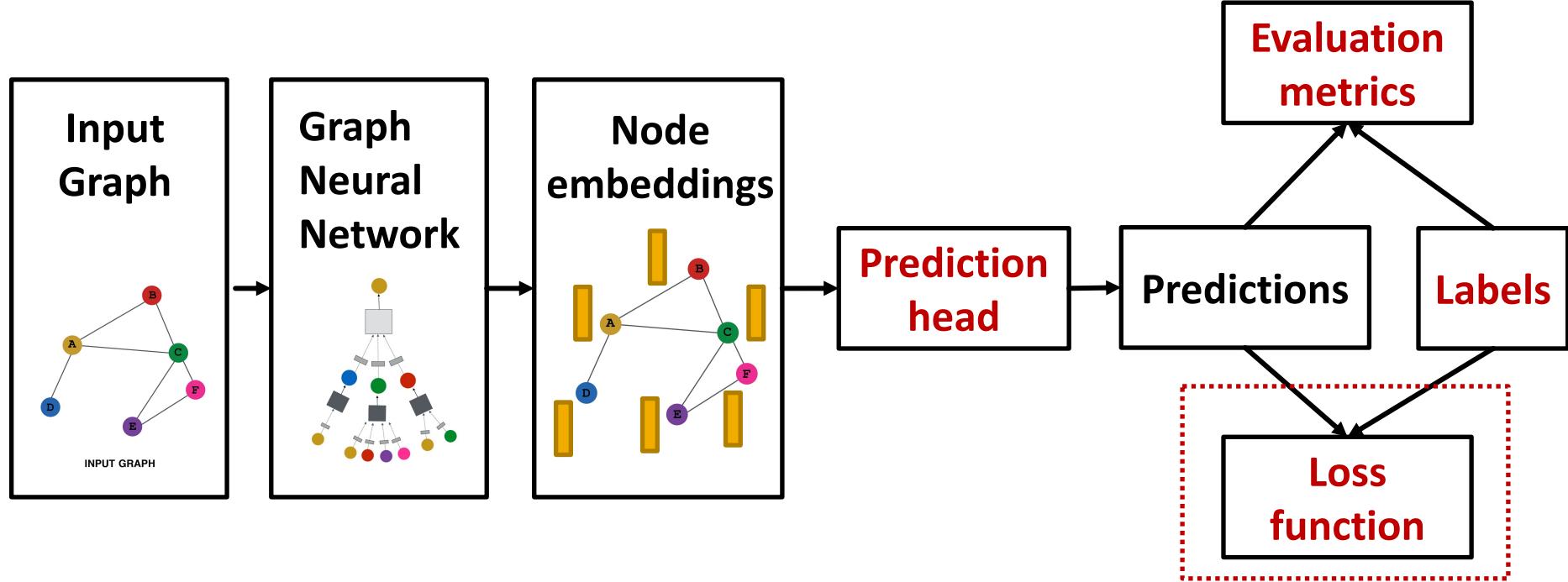
Supervised Labels on Graphs

- **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 equivalence relation on graphs
 - **These tasks do not require any external labels!**

GNN Training Pipeline (3)



(3) How do we compute the final loss?

- **Classification loss** cross entropy
- **Regression loss** mean squared error

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

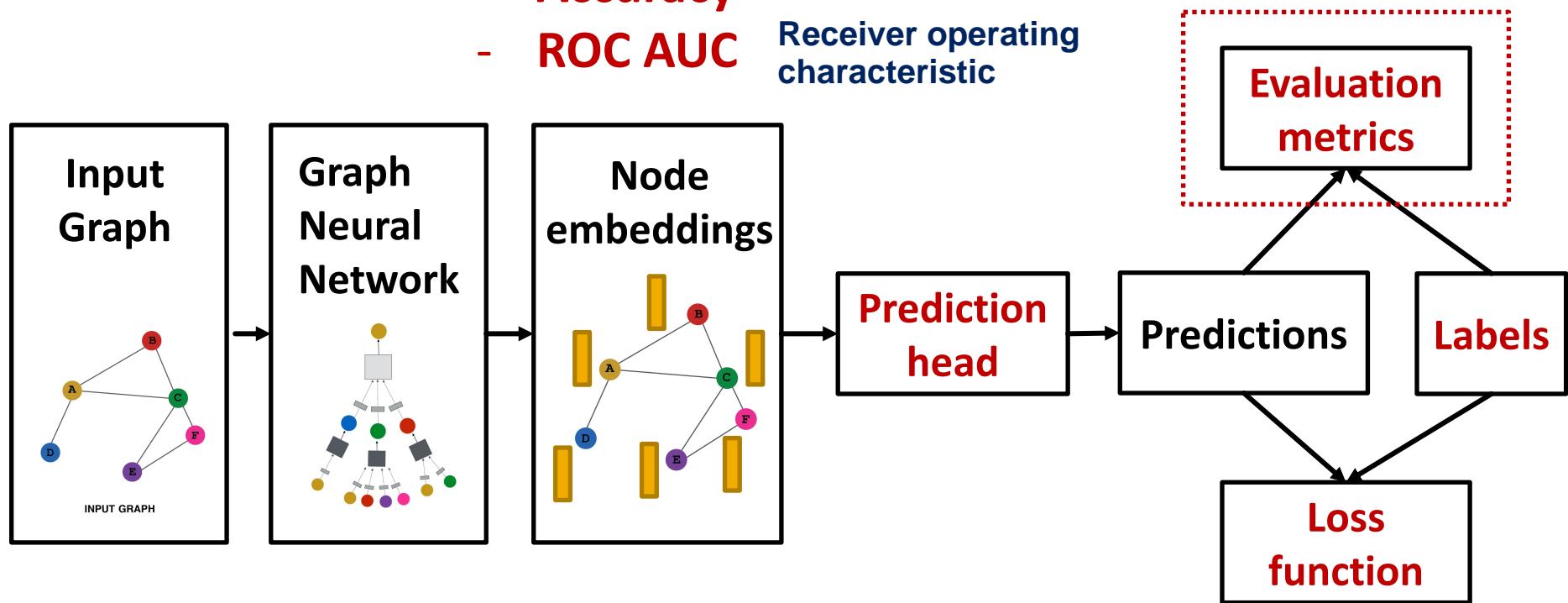
- **Classification:** labels $y^{(i)}$ with discrete value
 - E.g., Node classification: which category does a node belong to
- **Regression:** labels $y^{(i)}$ with continuous value
 - E.g., predict the drug likeness of a molecular graph
- GNNs can be applied to both settings
- **Differences: loss function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

GNN Training Pipeline (4)

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

- Accuracy
- ROC AUC

Receiver operating
characteristic



Stanford CS224W: Setting-up GNN Prediction Tasks

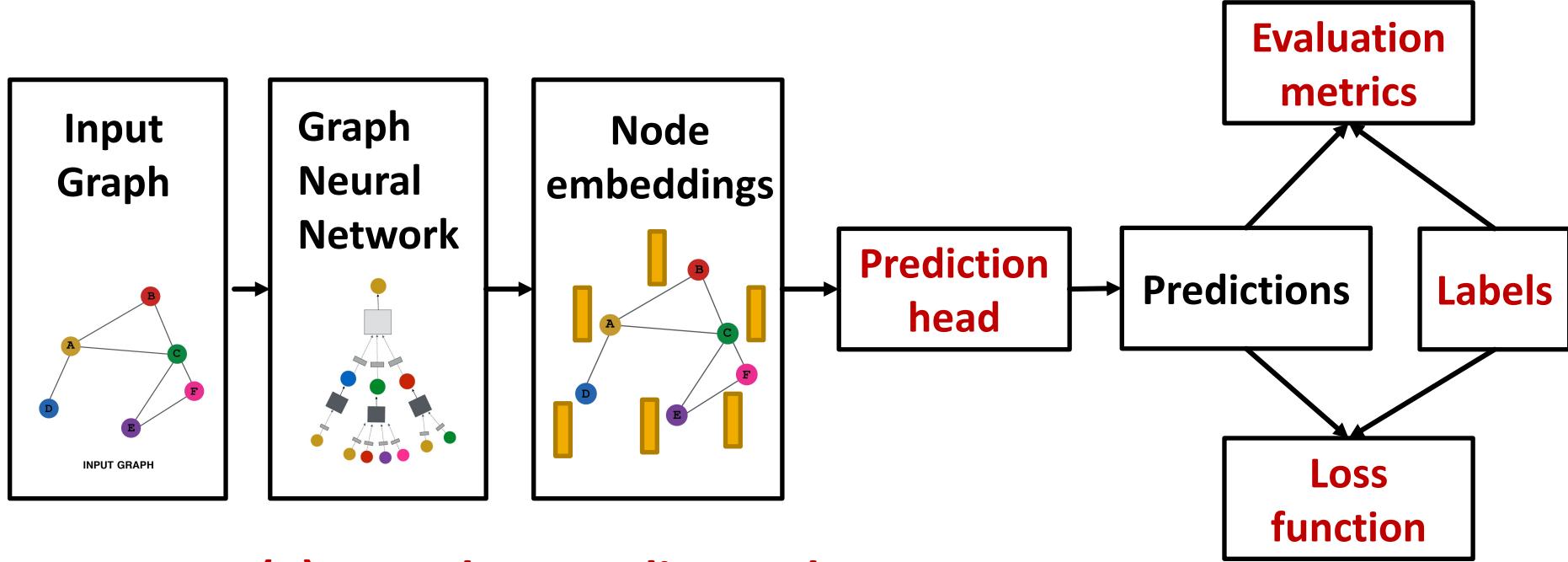
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

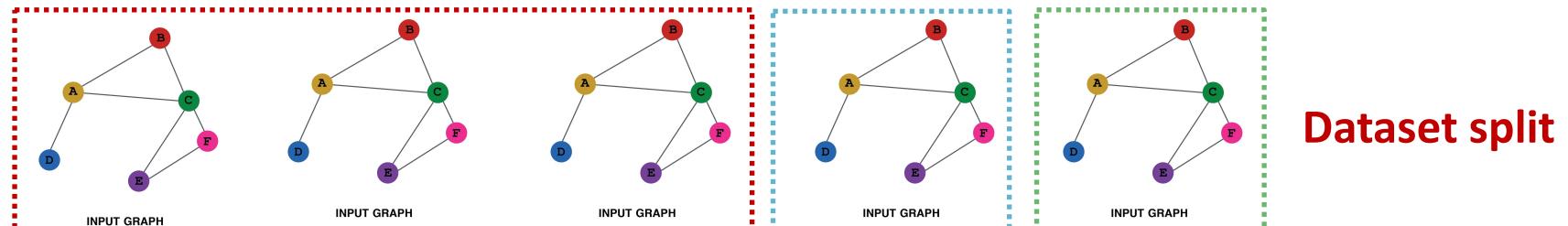
<http://cs224w.stanford.edu>



GNN Training Pipeline (5)



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

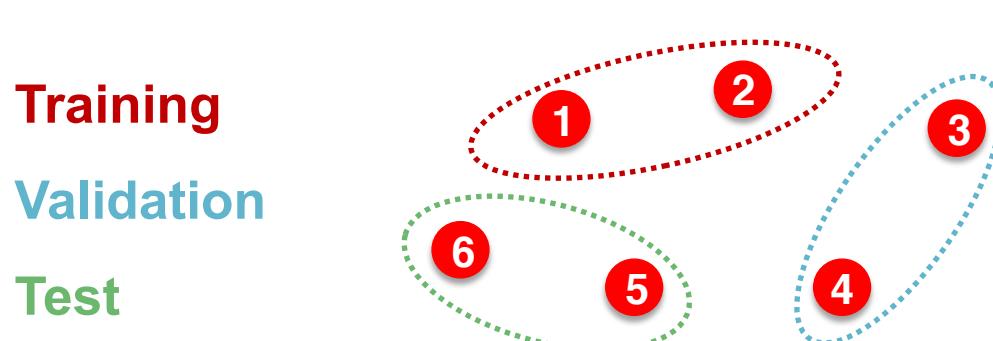


Dataset Split: Fixed / Random Split

- **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
- **Random split:** we will **randomly split** our dataset into training / validation / test
 - We report **average performance over different random seeds**

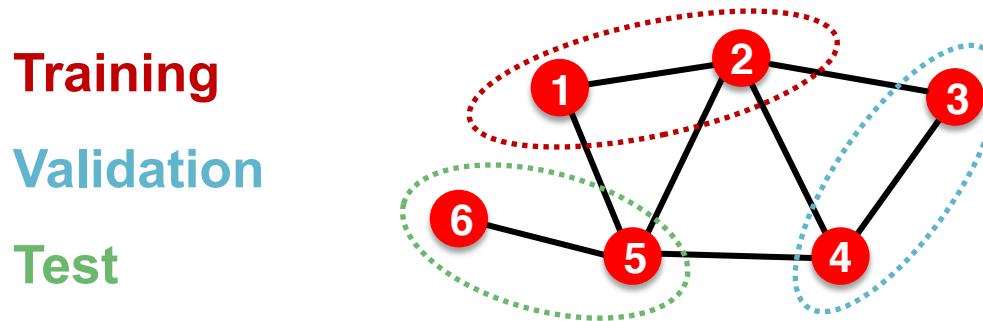
Why Splitting Graphs is Special

- 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

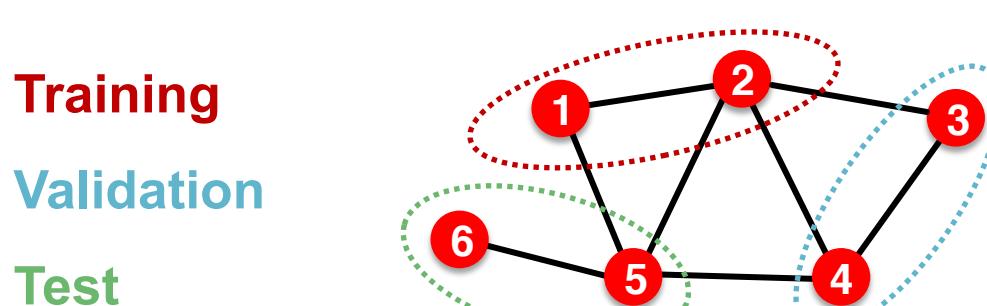
- **Splitting a graph dataset is different!**
 - **Node classification:** Each data point is a node
 - Here **data points are NOT independent**
 - Node 5 will affect our prediction on node 1, because it will participate in message passing → affect node 1's embedding



- **What are our options?**

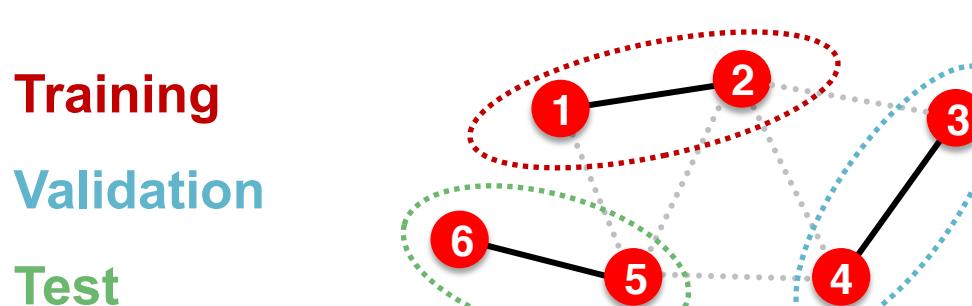
Why Splitting Graphs is Special

- **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

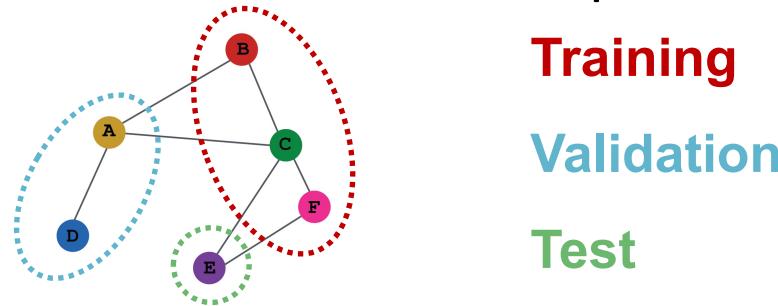


Transductive / Inductive Settings

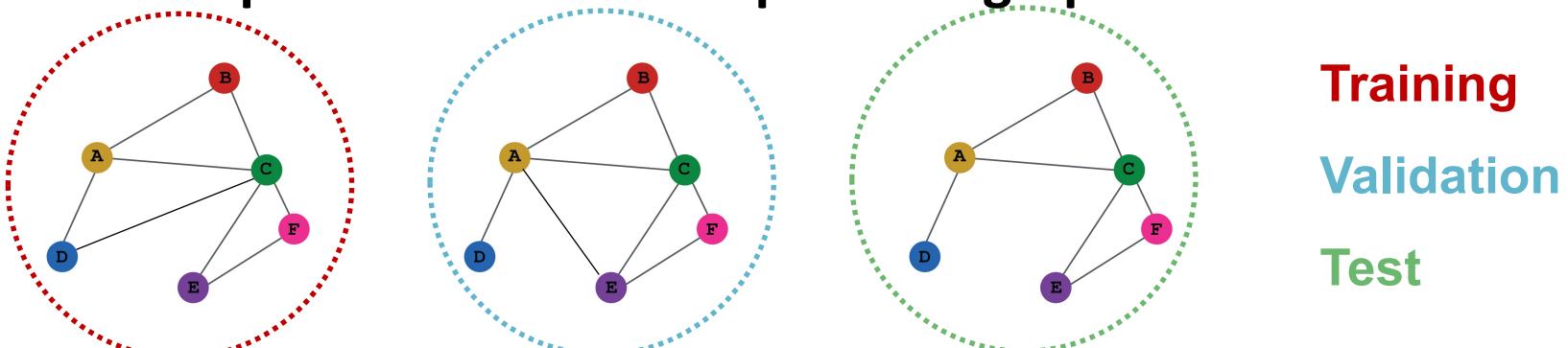
- **Transductive setting:** training / validation / test sets are **on the same graph**
 - The **dataset consists of one graph**
 - **The entire graph can be observed in all dataset splits, we only split the labels**
 - Only applicable to **node / edge** prediction tasks
- **Inductive setting:** training / 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

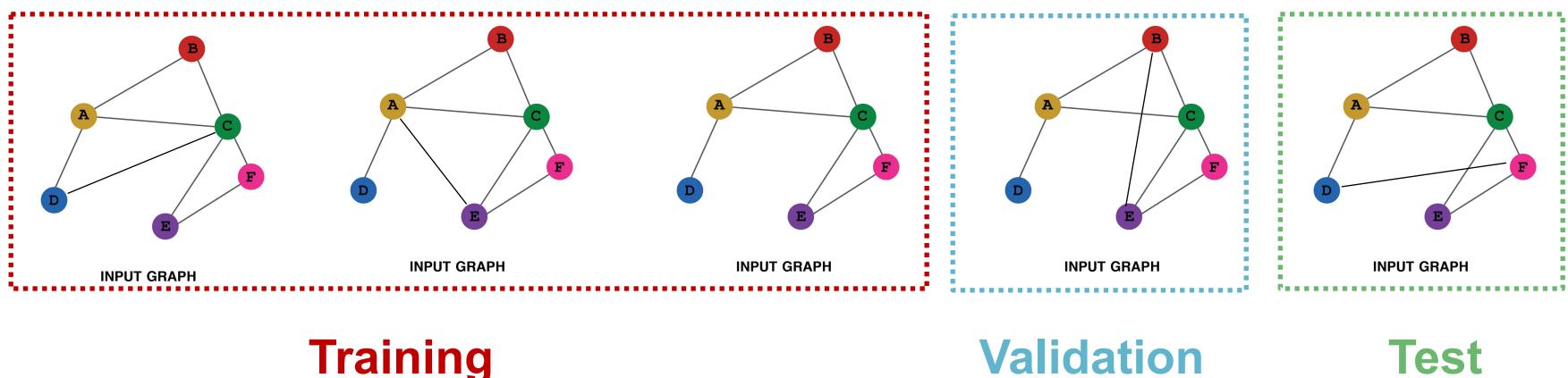


- **Inductive** node classification
 - Suppose we have a dataset of 3 graphs
 - **Each split contains an independent graph**



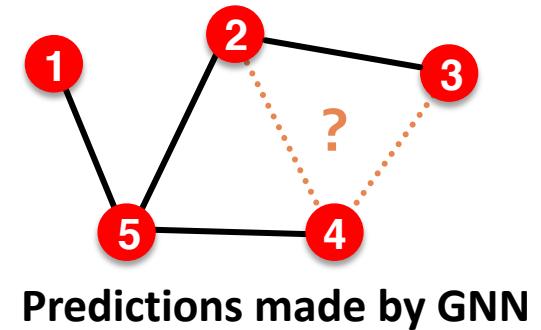
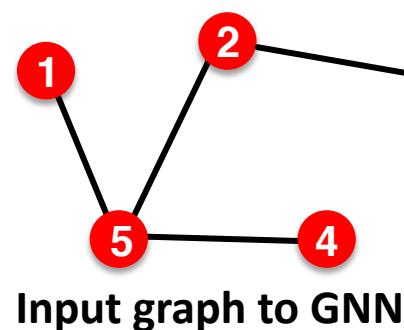
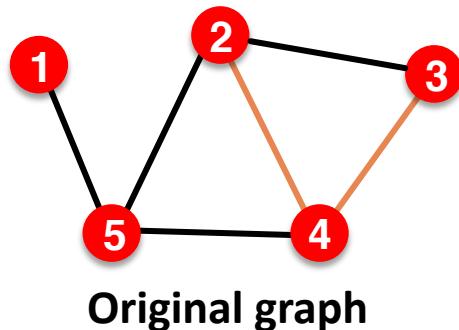
Example: Graph Classification

- 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).



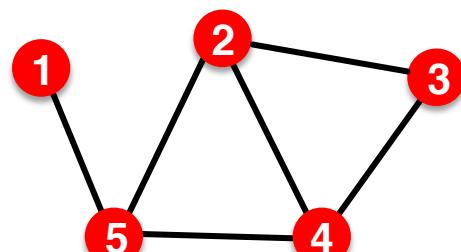
Example: Link Prediction

- **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**



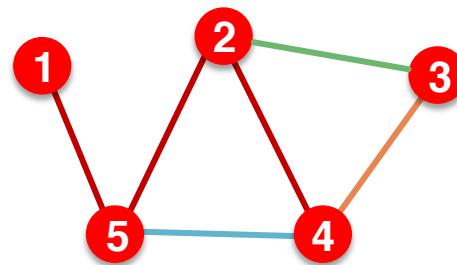
Setting up Link Prediction

■ Summary: Transductive link prediction split:



The original graph

Split

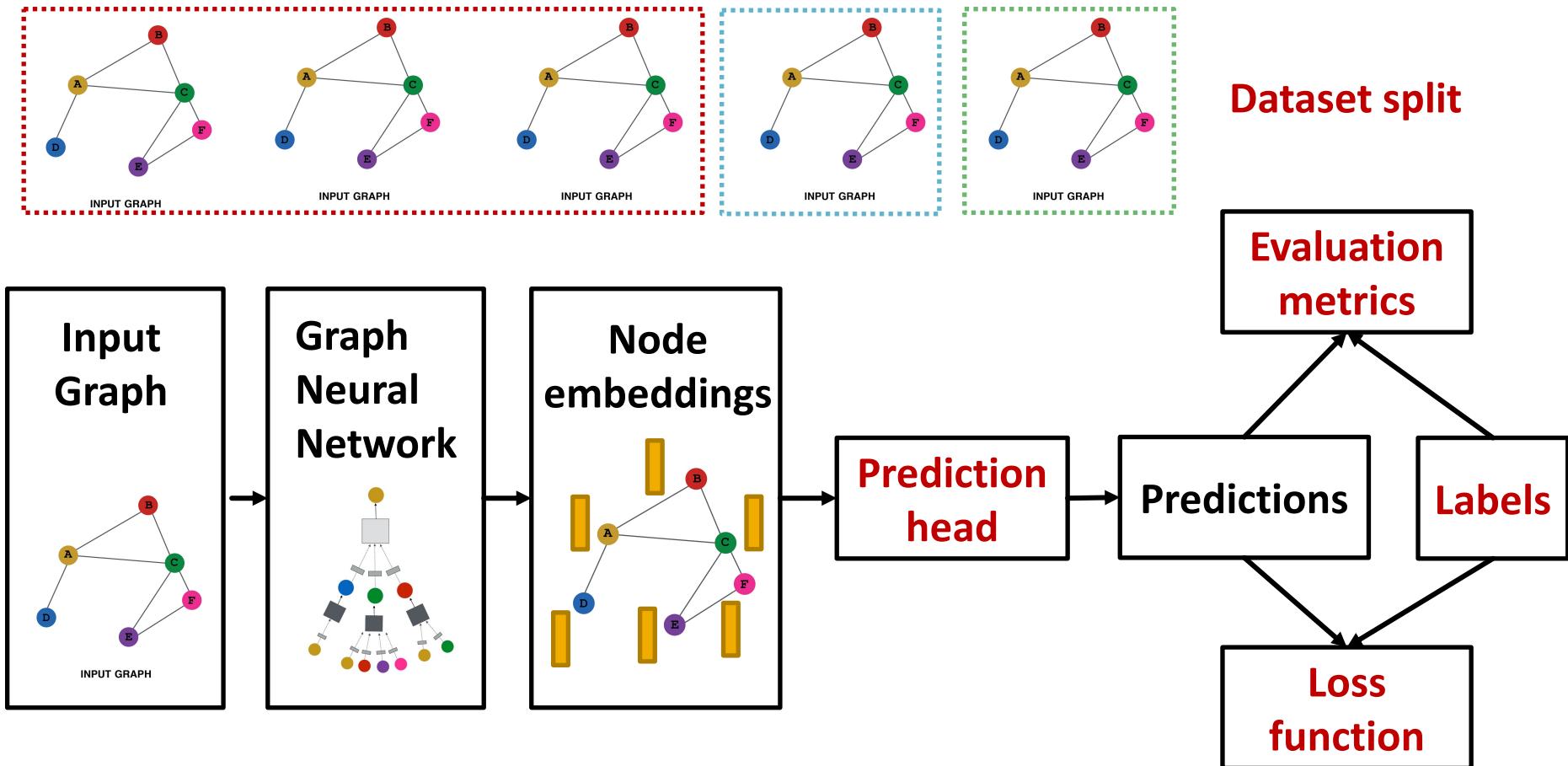


Split Graph with
4 types of edges

Training message edges
Training supervision edges
Validation edges
Test edges

- **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](#)

GNN Training Pipeline



Implementation resources:

[DeepSNAP](#) provides core modules for this pipeline

[GraphGym](#) further implements the full pipeline to facilitate GNN design

Summary of the Lecture

- We introduce a general GNN framework:
 - GNN Layer:
 - Transformation + Aggregation
 - Classic GNN layers: GCN, GraphSAGE, GAT
 - Layer connectivity:
 - The over-smoothing problem
 - Solution: skip connections
 - Graph Augmentation:
 - Feature augmentation
 - Structure augmentation
 - Learning Objectives
 - The full training pipeline of a GNN

Stanford CS224W: When Things Don't Go As Planned

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



General Tips

- **Data preprocessing is important:**
 - Node attributes can vary a lot!
 - E.g. probability ranges $(0,1)$, but some inputs could have much larger range, say $(-1000, 1000)$
 - Use normalization
- **Optimizer:**
 - ADAM is relatively robust to learning rate
- **Activation function**
 - ReLU activation function often works well
 - Other alternatives: [LeakyReLU](#), [SWISH](#), [rational activation](#)
 - No activation function at your output layer:
- **Include bias term in every layer**
- **Embedding dimensions:**
 - 32, 64 and 128 are often good starting points

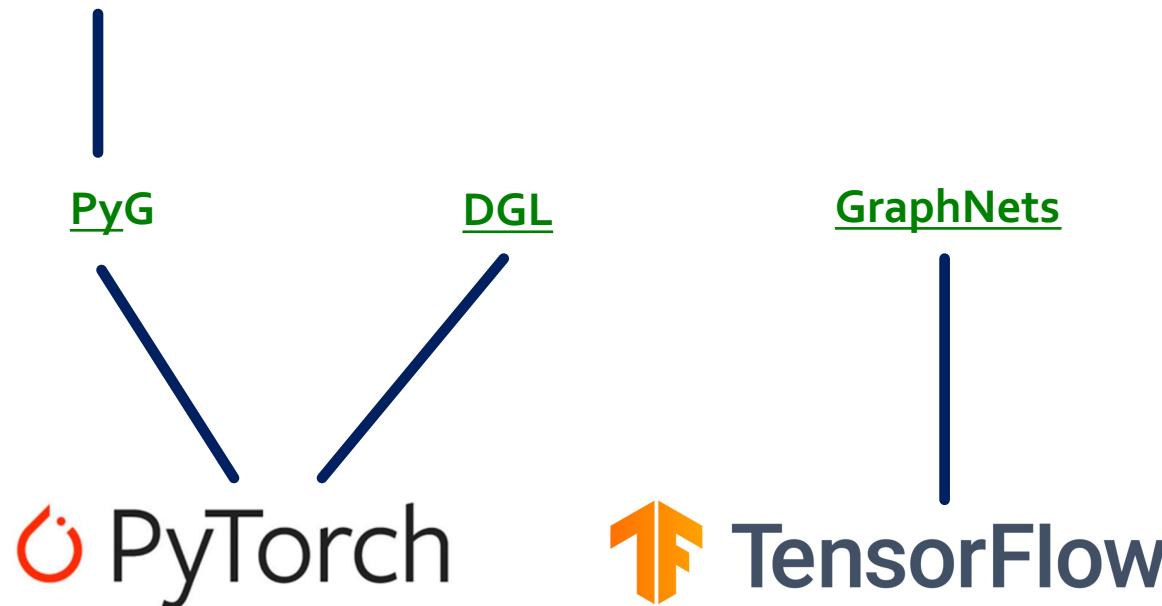
Debugging Deep Networks

- **Debug issues:** Loss/accuracy not converging during training
 - Check pipeline (e.g. in PyTorch we need zero_grad)
 - Adjust hyperparameters such as learning rate
 - Pay attention to weight parameter initialization
- **Important for model development:**
 - **Overfit on (part of) training data:**
 - With a small training dataset, loss should be essentially close to 0, with an expressive neural network
 - If neural network cannot overfit a single data point, something is wrong
 - **Scrutinize loss function!**
 - **Scrutinize visualizations!**

Resources on Graph Neural Networks

GraphGym:

Easy and flexible implementation
support based on PyTorch Geometric



GNN framework:
Implements a variety
of GNN architectures

Auto-differentiation
framework

Resources on Graph Neural Networks

Tutorials and overviews:

- Relational inductive biases and graph networks (Battaglia et al., 2018)
- Representation learning on graphs: Methods and applications (Hamilton et al., 2017)

Attention-based neighborhood aggregation:

- Graph attention networks (Hoshen, 2017; Velickovic et al., 2018; Liu et al., 2018)

Embedding entire graphs:

- Graph neural nets with edge embeddings (Battaglia et al., 2016; Gilmer et. al., 2017)
- Embedding entire graphs (Duvenaud et al., 2015; Dai et al., 2016; Li et al., 2018) and graph pooling (Ying et al., 2018, Zhang et al., 2018)
- Graph generation and relational inference (You et al., 2018; Kipf et al., 2018)
- How powerful are graph neural networks(Xu et al., 2017)

Embedding nodes:

- Varying neighborhood: Jumping knowledge networks (Xu et al., 2018), GeniePath (Liu et al., 2018)
- Position-aware GNN (You et al. 2019)

Spectral approaches to graph neural networks:

- Spectral graph CNN & ChebNet (Bruna et al., 2015; Defferrard et al., 2016)
- Geometric deep learning (Bronstein et al., 2017; Monti et al., 2017)

Other GNN techniques:

- Pre-training Graph Neural Networks (Hu et al., 2019)
- GNNExplainer: Generating Explanations for Graph Neural Networks (Ying et al., 2019)