Advanced Deep Learning Architectures
COMP 5214 & ELEC 5680

Instructor: Dr. Qifeng Chen

https://cqf.io

Graph Neural Networks

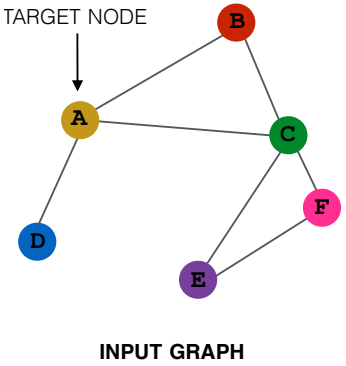
Stanford CS224W: Prediction with GNNs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

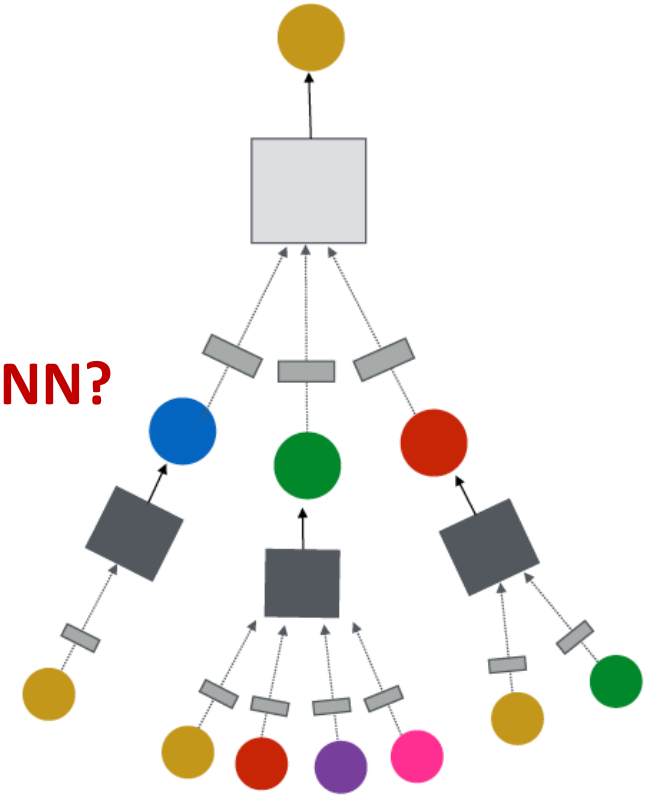
http://cs224w.stanford.edu

A General GNN Framework (4)



INPUT GRAPH

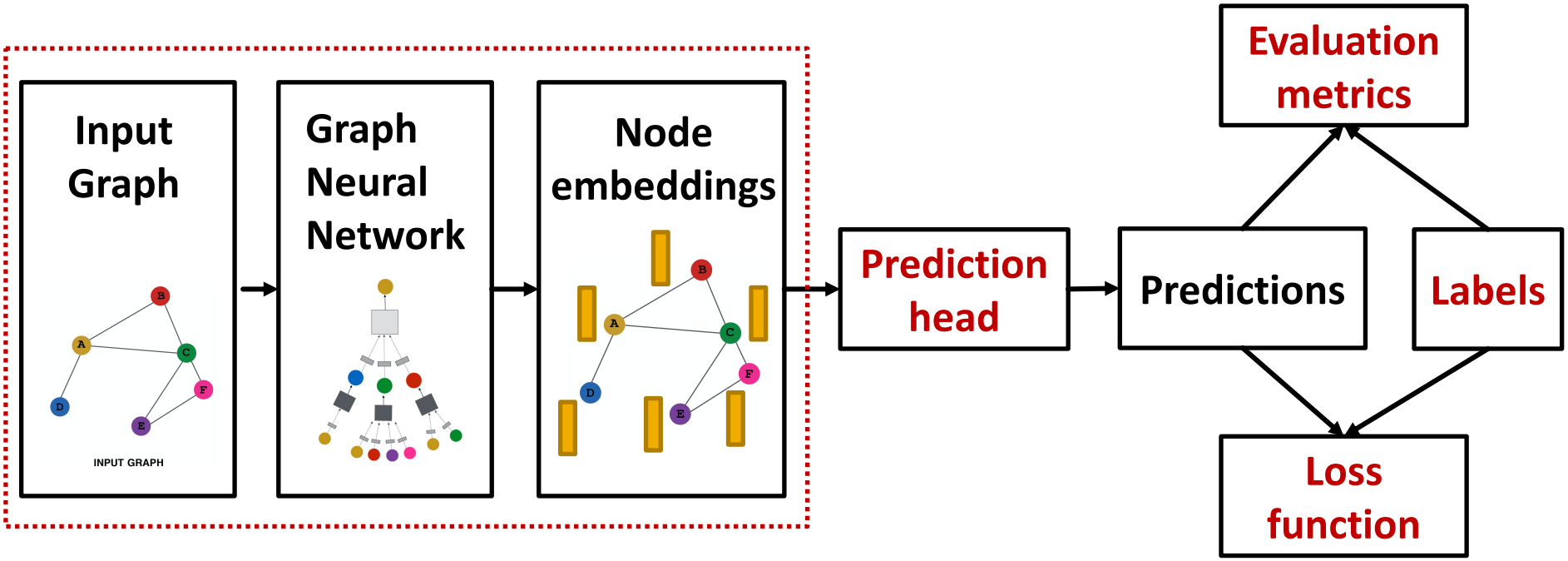
(5) Learning objective



Next: How do we train a GNN?

GNN Training Pipeline

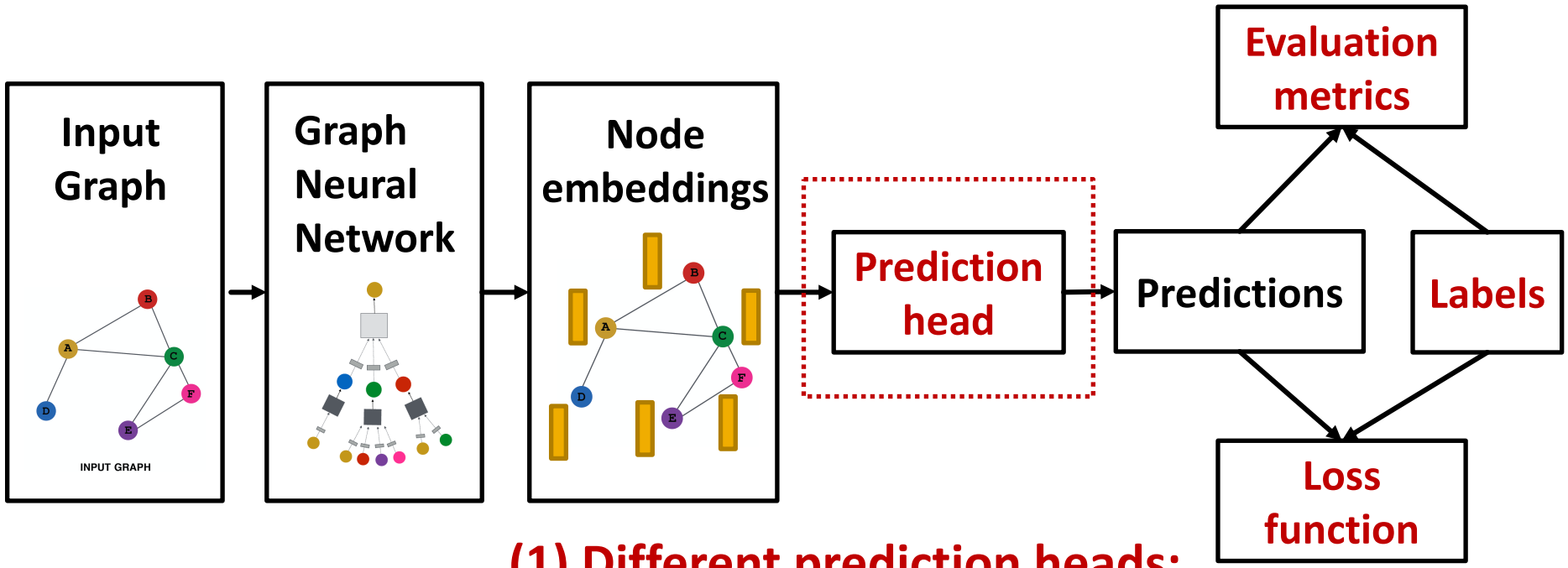
So far what we have covered



Output of a GNN: set of node embeddings

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

GNN Training Pipeline (1)

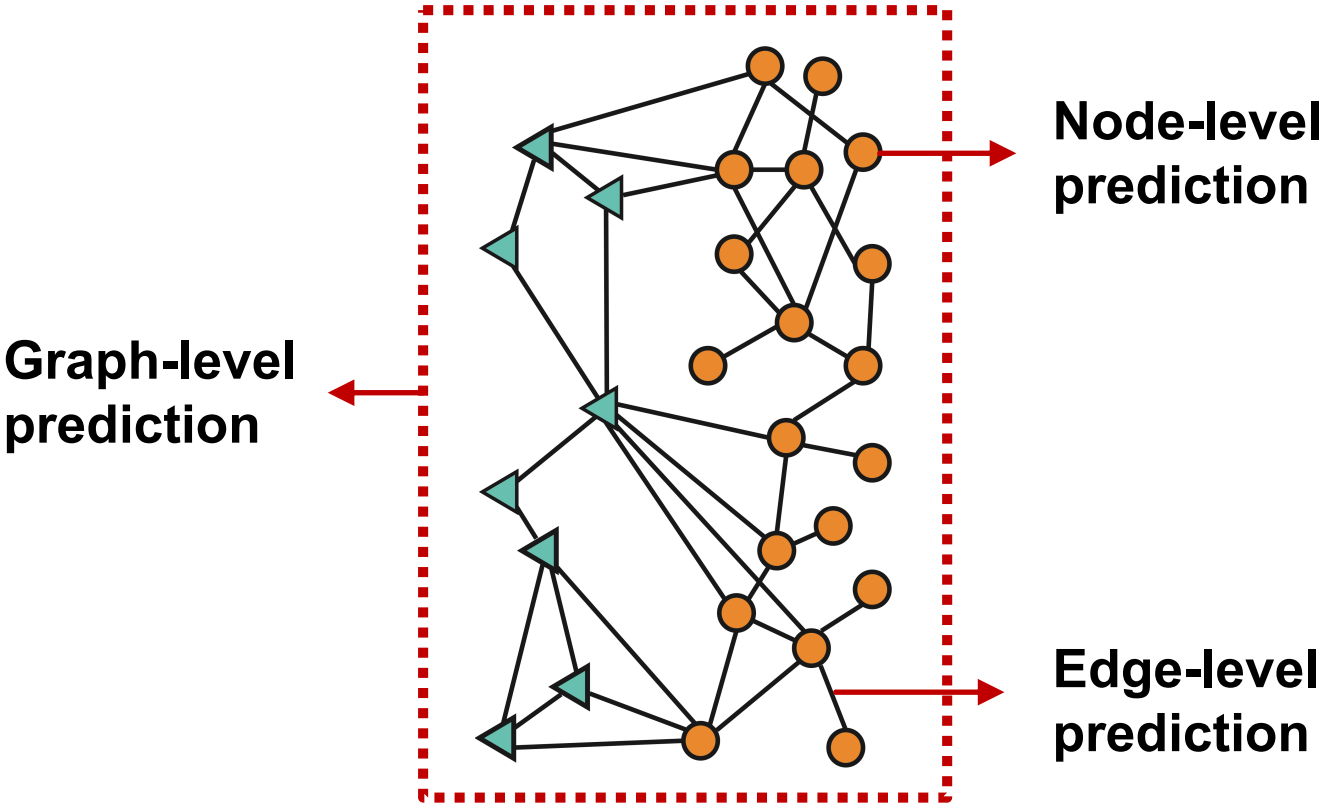


(1) Different prediction heads:

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

GNN Prediction Heads

- **Idea:** Different task levels require different prediction heads

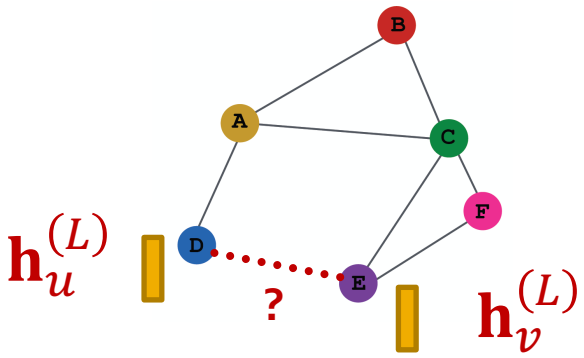


Prediction Heads: Node-level

- **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
- $\hat{\boldsymbol{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)}\mathbf{h}_v^{(L)}$
  - $\mathbf{W}^{(H)} \in \mathbb{R}^{k*d}$: We map node embeddings from $\mathbf{h}_v^{(L)} \in \mathbb{R}^d$ to $\hat{\boldsymbol{y}}_v \in \mathbb{R}^k$ so that we can compute the loss

Prediction Heads: Edge-level

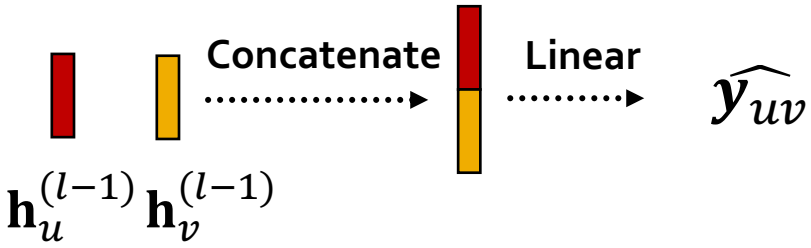
- **Edge-level prediction**: Make prediction using pairs of node embeddings
- Suppose we want to make k-way prediction
- $\hat{\boldsymbol{y}}_{\boldsymbol{uv}} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



- 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



  - $\hat{\boldsymbol{y}}_{\boldsymbol{uv}} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
  - 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**
  - $\hat{\boldsymbol{y}}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
  - **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**: $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(k)}$ trainable

$$\hat{\boldsymbol{y}}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$

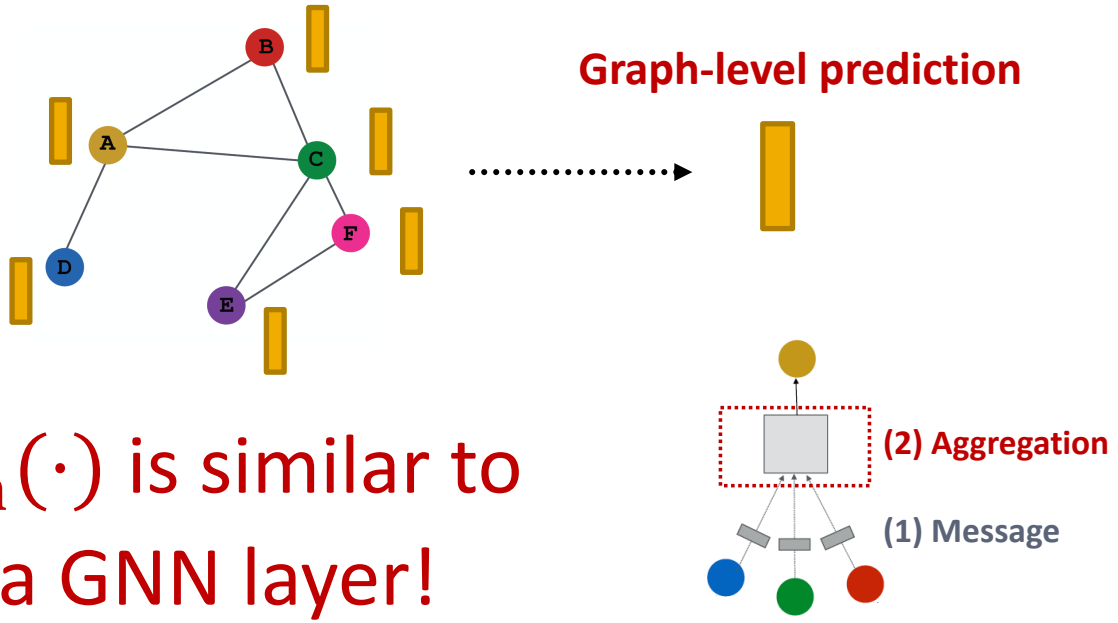
$$\ldots$$

$$\hat{\boldsymbol{y}}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$

$$\hat{\boldsymbol{y}}_{uv} = \text{Concat}(\hat{\boldsymbol{y}}_{uv}^{(1)}, \ldots, \hat{\boldsymbol{y}}_{uv}^{(k)}) \in \mathbb{R}^k$$

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



- $\text{Head}_{\text{graph}}(\cdot)$ 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**

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

- **(2) Global max pooling**

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

- **(3) Global sum pooling**

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

- 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

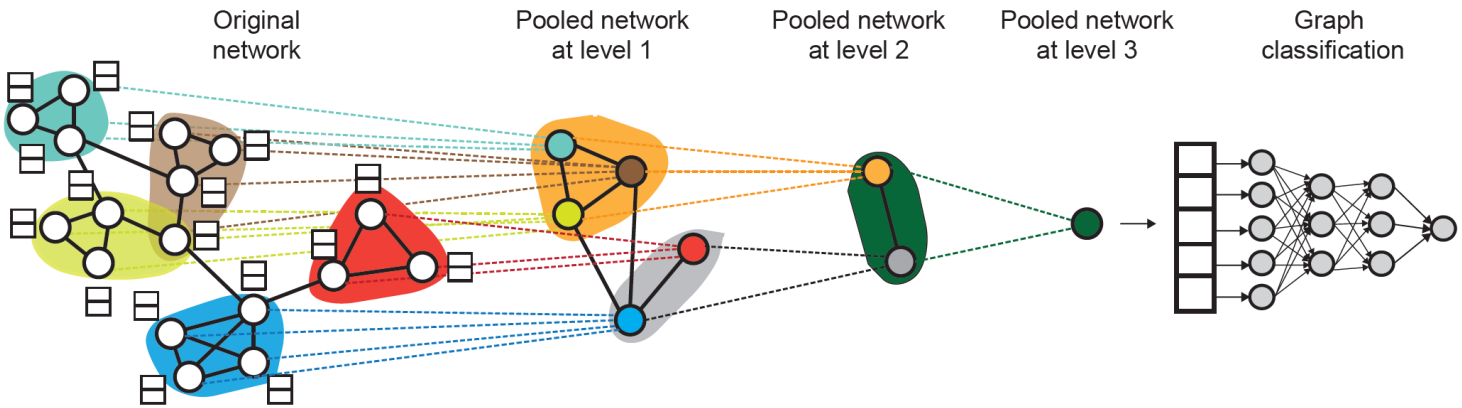
- **A solution:** Let's aggregate all the node embeddings **hierarchically**
  - **Toy example:** We will aggregate via $\mathrm{ReLU}(\mathrm{Sum}(\cdot))$
    - We first separately aggregate the first 2 nodes and last 3 nodes
    - Then we aggregate again to make the final prediction
  - G_1 node embeddings: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

Now we can differentiate G_1 and G_2 !

Ying et al. Hierarchical Graph Representation Learning with Differentiable Pooling, NeurIPS 2018

Hierarchical Pooling In Practice

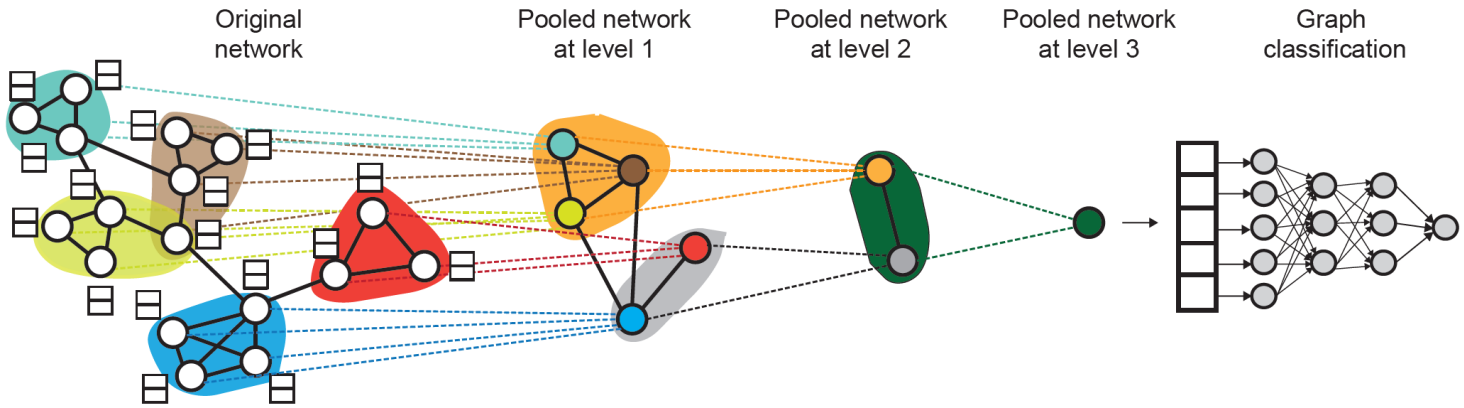
- **DiffPool idea:**
  - **Hierarchically pool node embeddings**



  - **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 generated a new **pooled** network
  - **Jointly train GNN A and GNN B**

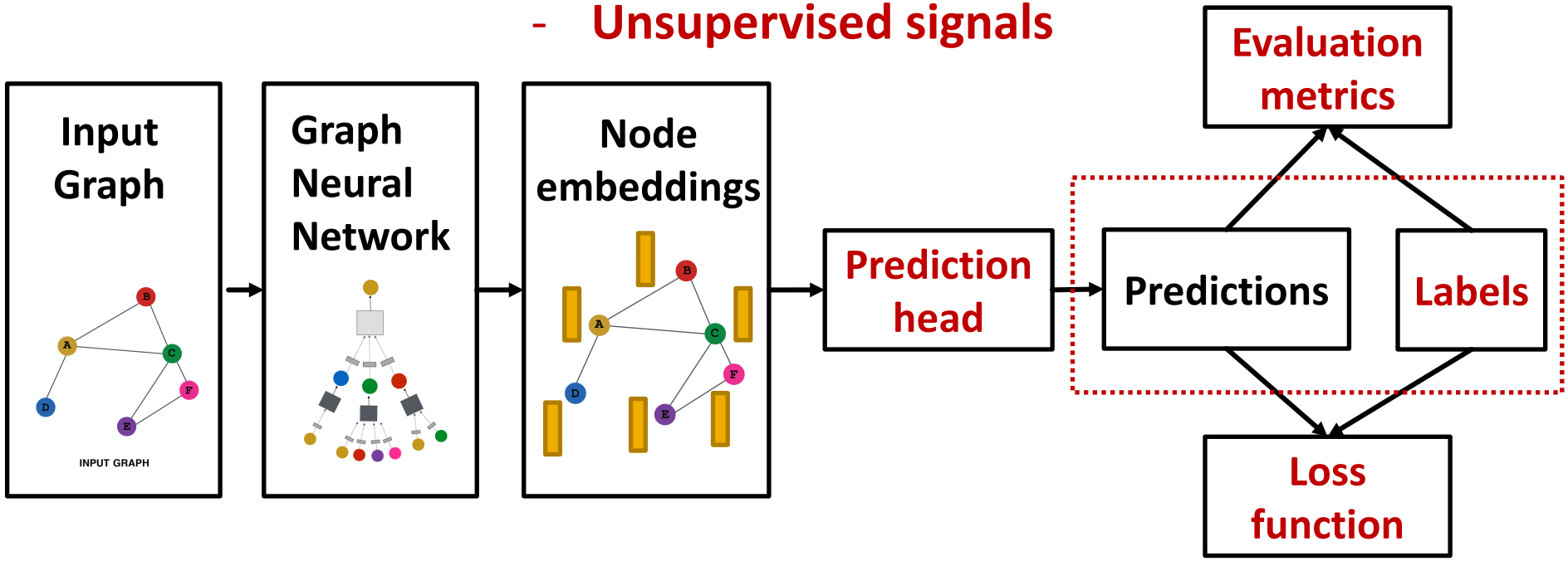
Stanford CS224W: Training Graph Neural Networks

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**"

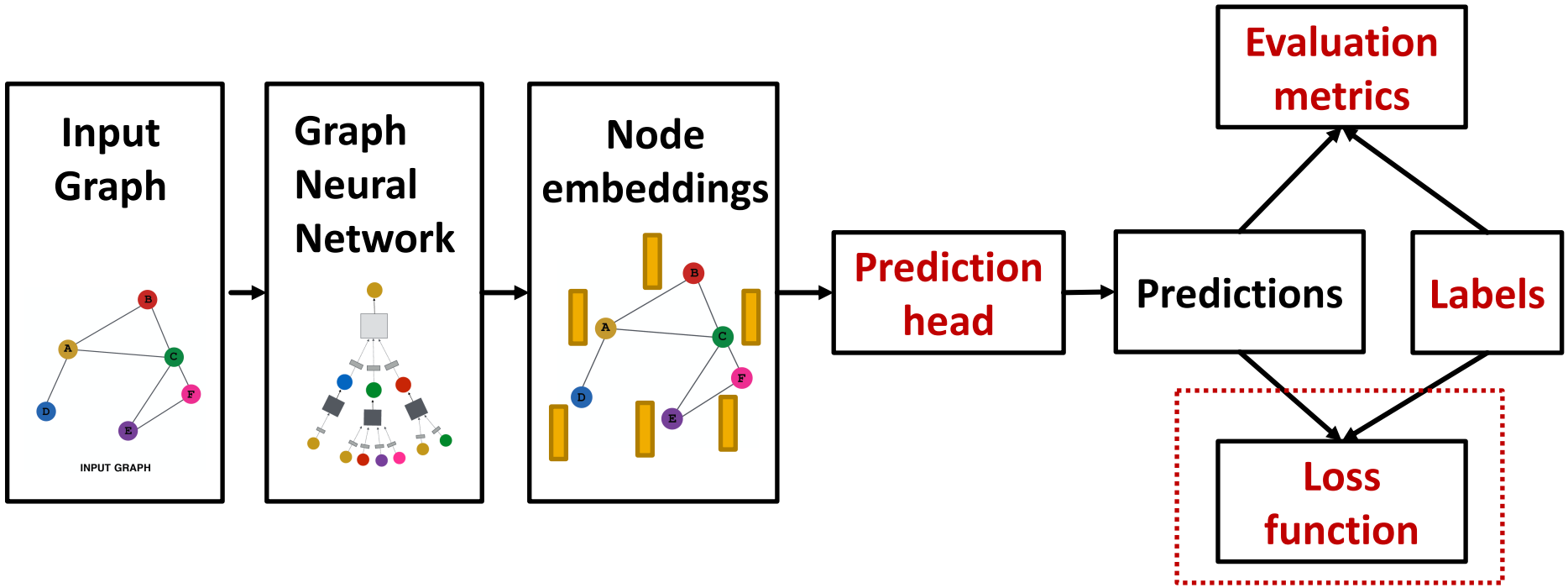
Supervised Labels on Graphs

- **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 $\hat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
  - **Edge-level**: prediction $\hat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
  - **Graph-level**: prediction $\hat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$
  - We will use prediction $\hat{\boldsymbol{y}}^{(i)}$, label $\boldsymbol{y}^{(i)}$ to refer **predictions at all levels**

Classification or Regression

- **Classification**: labels $\boldsymbol{y}^{(i)}$ with discrete value
  - E.g., Node classification: which category does a node belong to
- **Regression**: labels $\boldsymbol{y}^{(i)}$ 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

- As discussed in lecture 6, **cross entropy (CE)** is a very common loss function in classification
- K-way prediction for i-th data point:

$$\mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right) = -\sum_{j=1}^{K} \boldsymbol{y}_j^{(i)} \log\left(\hat{\boldsymbol{y}}_j^{(i)}\right)$$

$\boldsymbol{y}^{(i)}$: Label; $\hat{\boldsymbol{y}}^{(i)}$: Prediction; (i): i-th data point; j: j-th class

where:

E.g. | 0 | 0 | 1 | 0 | 0 |

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

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

E.g. | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |

- Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^{N} \mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right)$$

Regression Loss

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

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

***i*-th data point**

***j*-th target**

where:

E.g.

1.4	2.3	1.0	0.5	0.6

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

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

E.g.

0.9	2.8	2.0	0.3	0.8

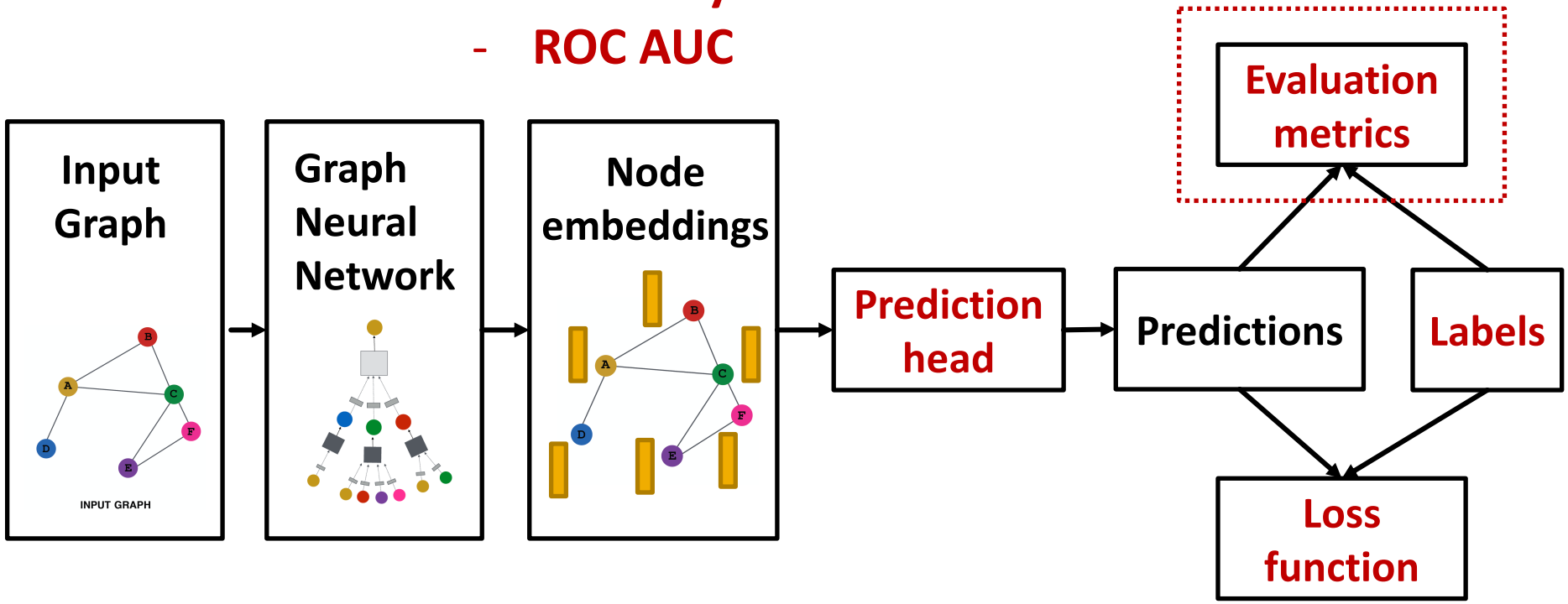
- Total loss over all N training examples

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

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)**

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

  - **Mean absolute error (MAE)**

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

Evaluation Metrics: Classification

- **Evaluate classification tasks on graphs:**
- **(1) Multi-class classification**
  - **We simply report the accuracy**

$$\frac{1\left[\text{argmax}\left(\hat{\boldsymbol{y}}^{(i)}\right) = \boldsymbol{y}^{(i)}\right]}{N}$$

- **(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**:

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

- **Precision (P)**:

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

- **Recall (R)**:

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

- **F1-Score**:

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

Confusion matrix

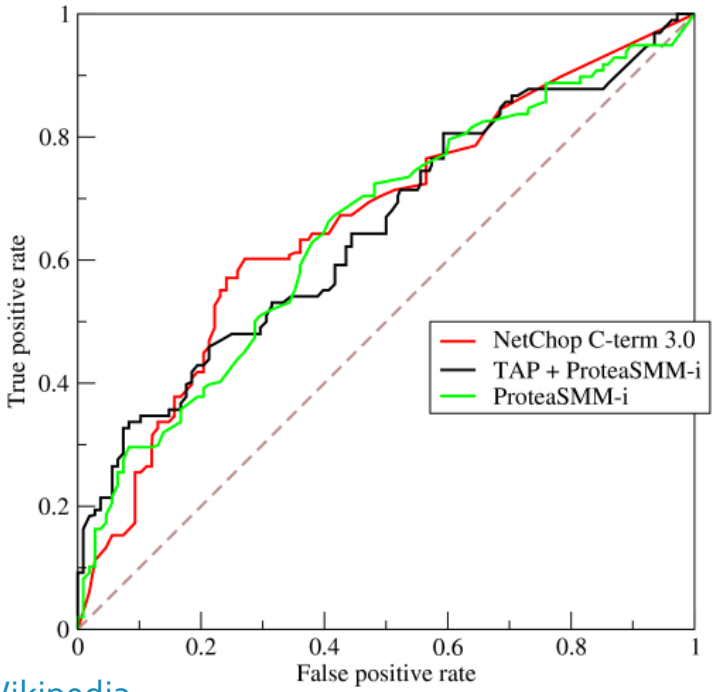
	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

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.**

TPR



FPR

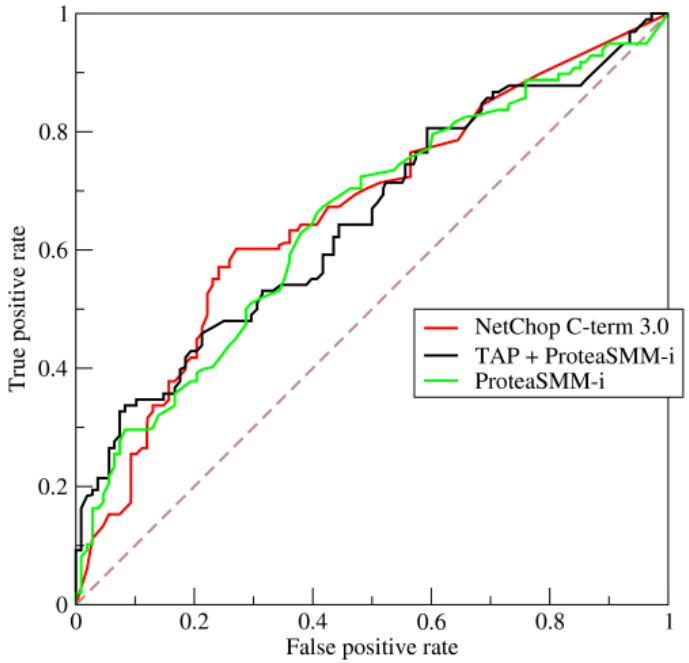
Image Credit: Wikipedia

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

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

Note: the dashed line represents **performance of a random classifier**

(4) Evaluation Metrics



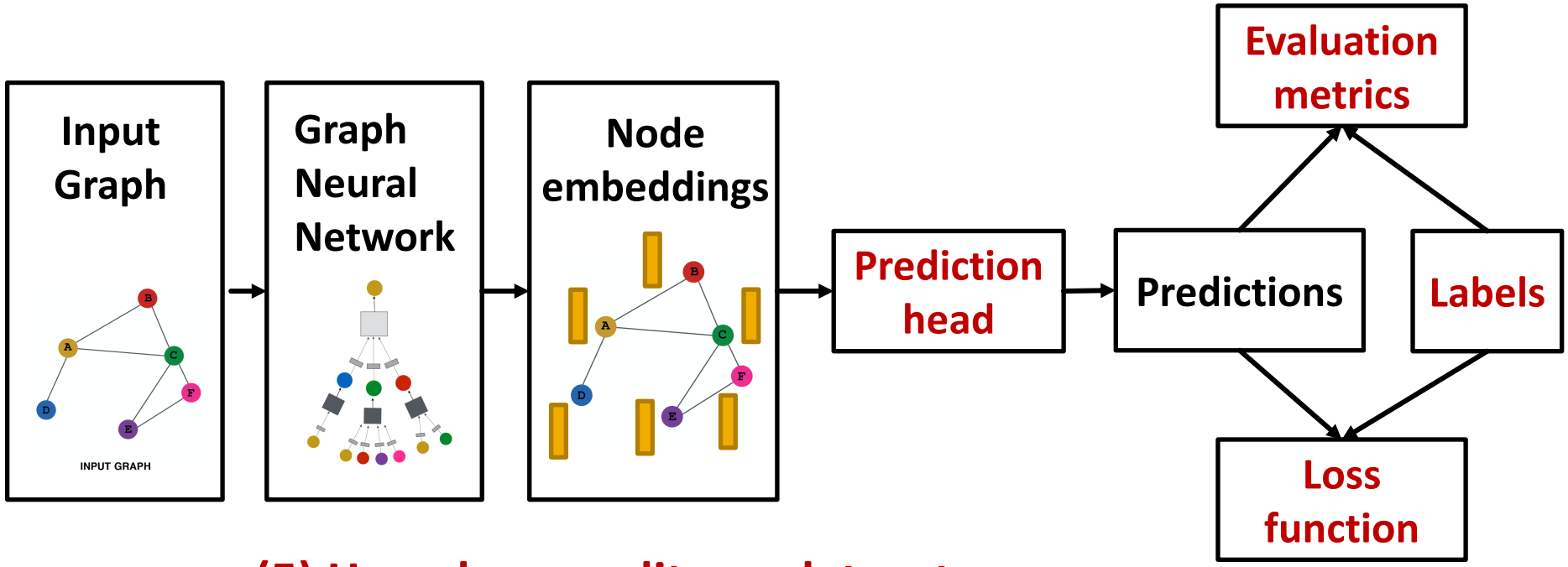
Content Credit: Wikipedia

- **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

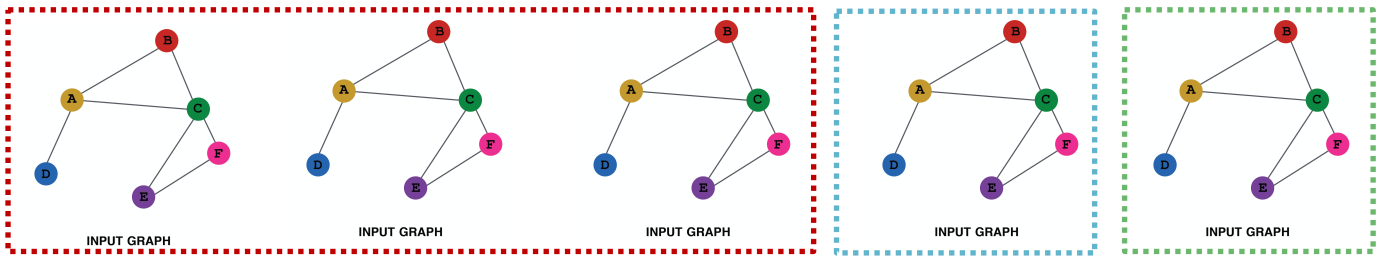
Stanford CS224W: Setting-up GNN Prediction Tasks

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

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**

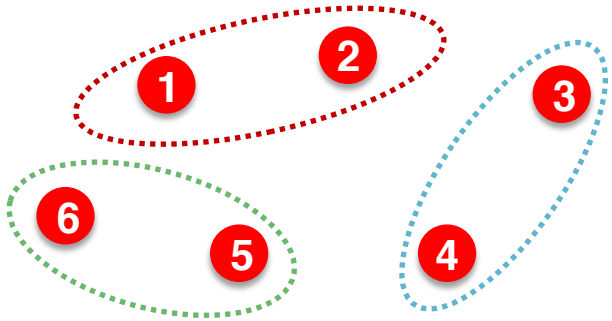
Why Splitting Graphs is Special

- **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**

Training

Validation

Test



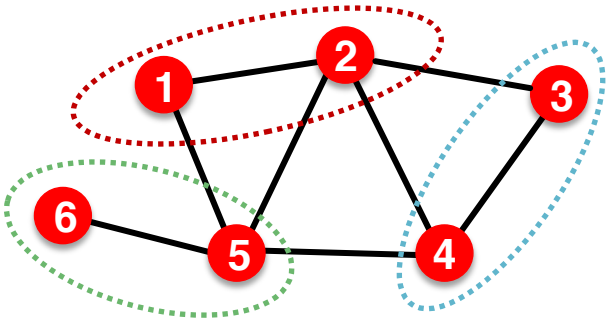
Why Splitting Graphs is Special

- **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

Training

Validation

Test



- **What are our options?**

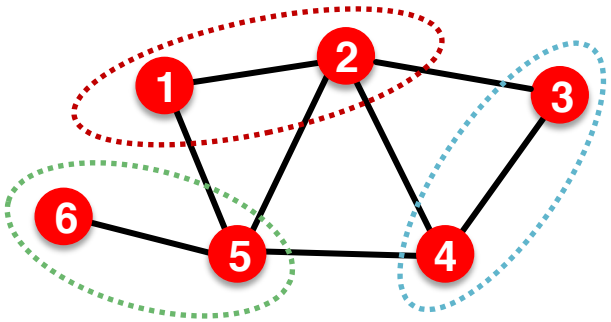
Why Splitting Graphs is Special

- **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**

Training

Validation

Test



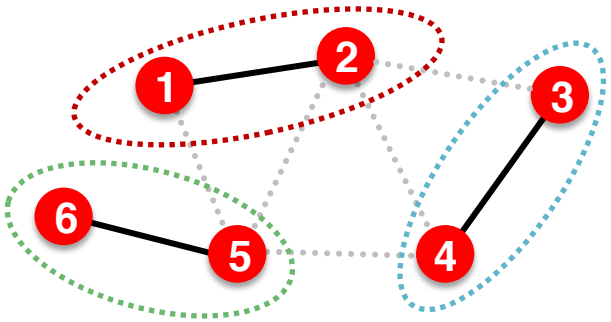
Why Splitting Graphs is Special

- **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**

Training

Validation

Test

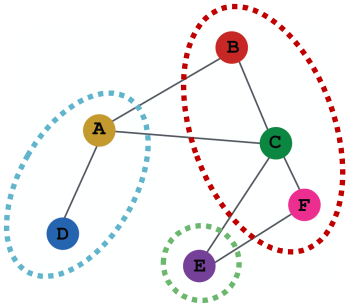


Transductive / Inductive Settings

- **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

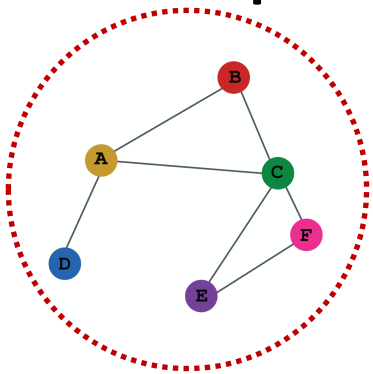


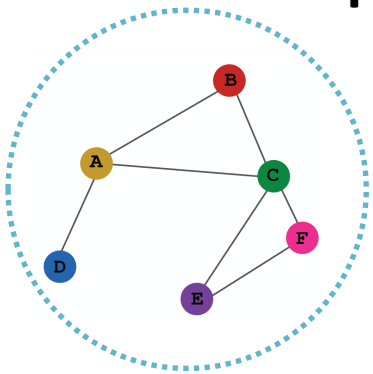
Training

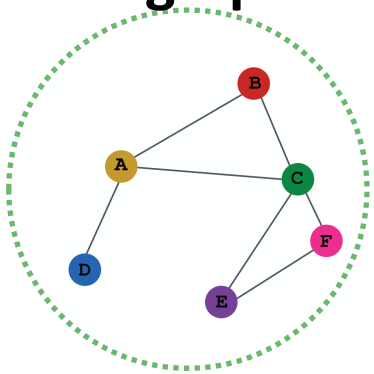
Validation

Test

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







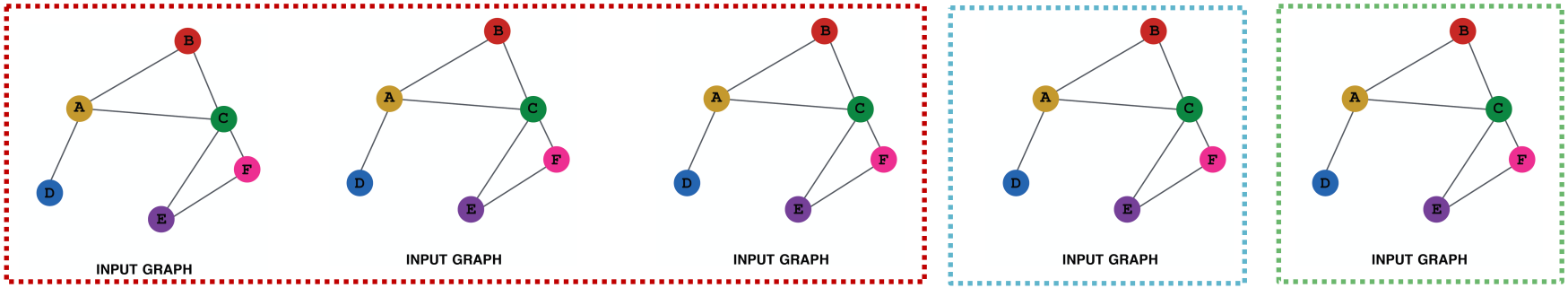
Training

Validation

Test

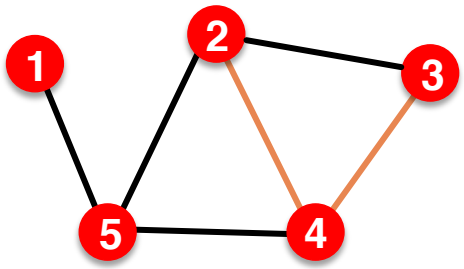
Example: Graph Classification

- 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).

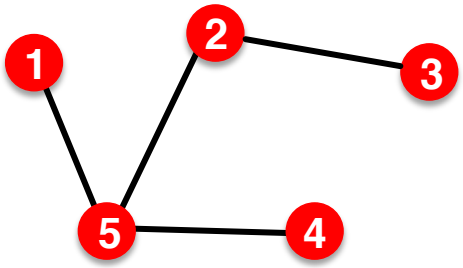


Example: Link Prediction

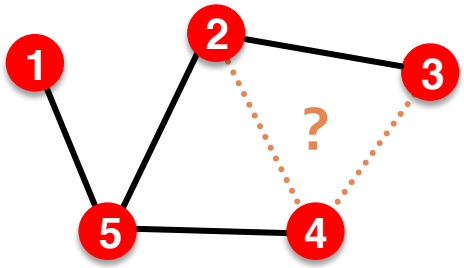
- **Goal of link prediction: predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and the **let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



Original graph



Input graph to GNN



Predictions made by GNN

Scaling Up Graph Neural Networks to Large Graphs

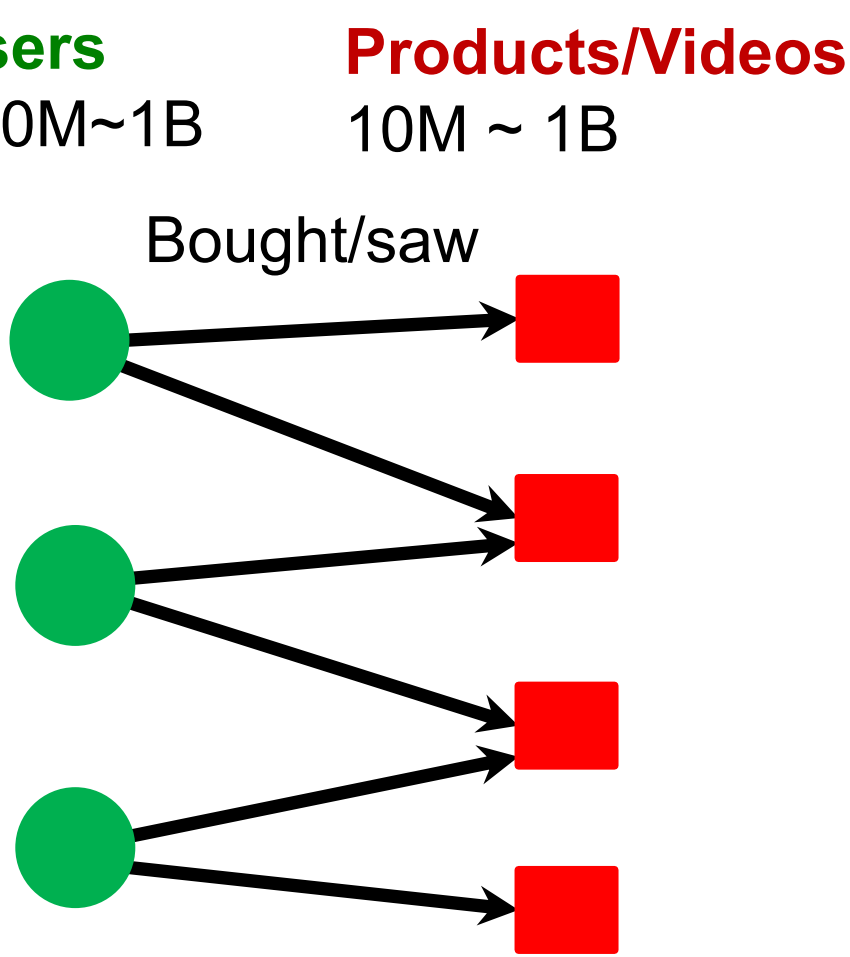
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

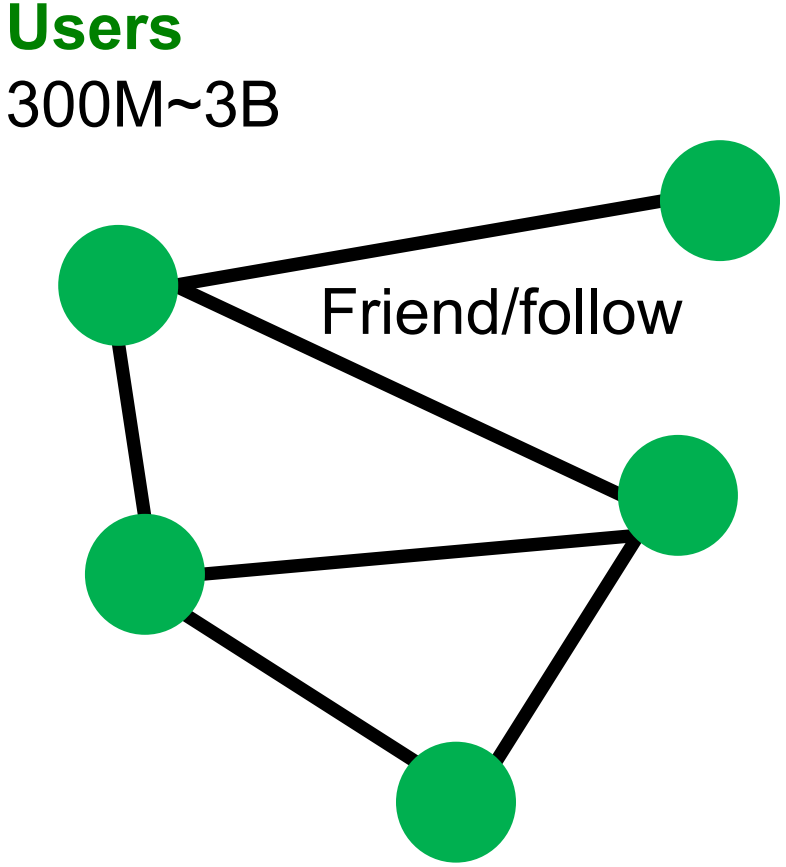
Graphs in Modern Applications

- **Recommender systems:**
  - Amazon
  - YouTube
  - Pinterest
  - Etc.
- **ML tasks:**
  - Recommend items (link prediction)
  - Classify users/items (node classification)



Graphs in Modern Applications

- **Social networks:**
  - Facebook
  - Twitter
  - Instagram
  - Etc.
- **ML tasks:**
  - Friend recommendation (link-level)
  - User property prediction (node-level)



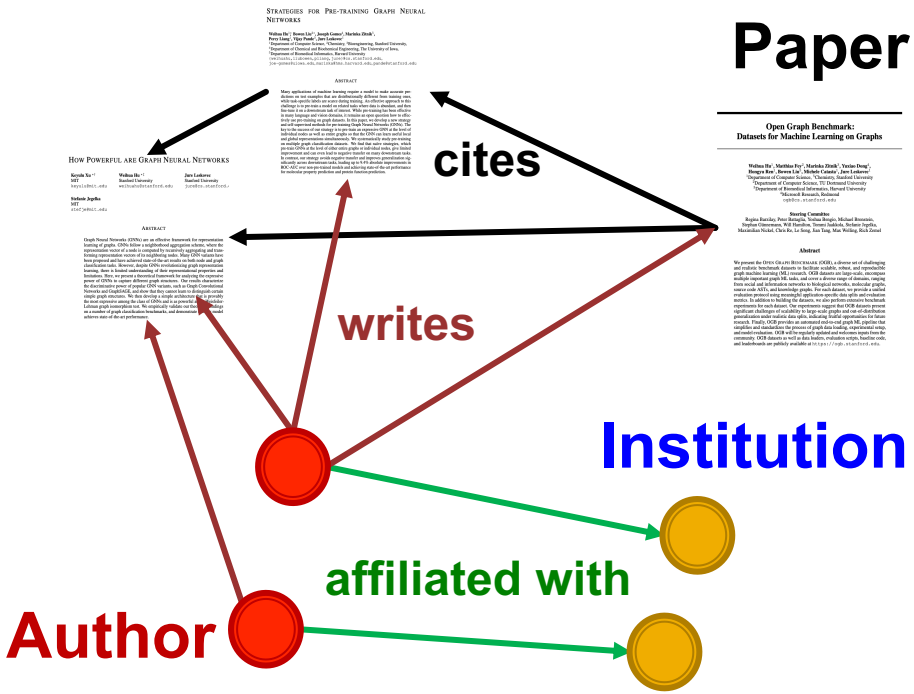
Graphs in Modern Applications

- **Academic graph:**
  - Microsoft Academic Graph

Papers 120M

Authors 120M

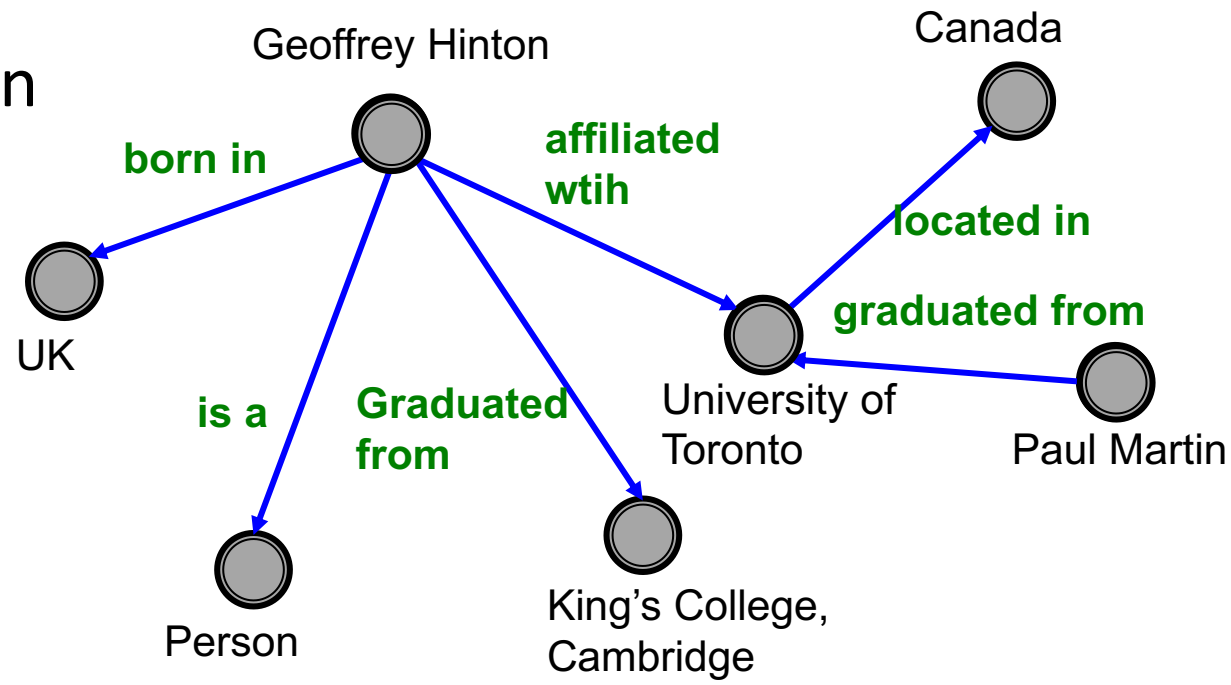
- **ML tasks:**
  - Paper categorization (node classification)
  - Author collaboration recommendation
  - Paper citation recommendation (link prediction)



Graphs in Modern Applications

- **Knowledge Graphs (KGs):**
  - Wikidata
  - Freebase
- **ML tasks:**
  - KG completion
  - Reasoning

Entities
80M—90M



What is in Common?

- **Large-scale:**
  - #nodes ranges from 10M to 10B.
  - #edges ranges from 100M to 100B.
- **Tasks**
  - **Node-level**: User/item/paper classification.
  - **Link-level**: Recommendation, completion.
- **Todays' lecture**
  - **Scale up GNNs to large graphs!**

Why is it Hard?

- **Recall: How we usually train an ML model on large data (N=#data is large)**
- **Objective**: Minimize the averaged loss

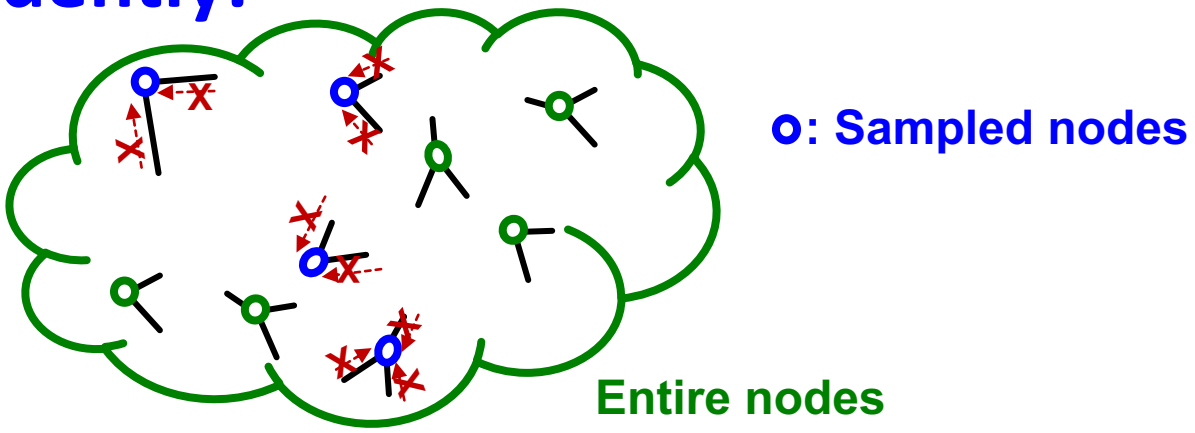
$$\ell(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=0}^{N-1} \ell_i(\boldsymbol{\theta})$$

  - $\boldsymbol{\theta}$: model parameters, $\ell_i(\boldsymbol{\theta})$: loss for i-th data point.
- We perform **Stochastic Gradient Descent (SGD)**.
  - Randomly sample M $(<< N)$ data (**mini-batches**).
  - Compute the $\ell_{sub}(\boldsymbol{\theta})$ over the M data points.
  - Perform SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla\ell_{sub}(\boldsymbol{\theta})$

Why is it Hard?

What if we use the standard SGD for GNN?

- **In mini-batch, we sample M ($<<N$) nodes independently:**



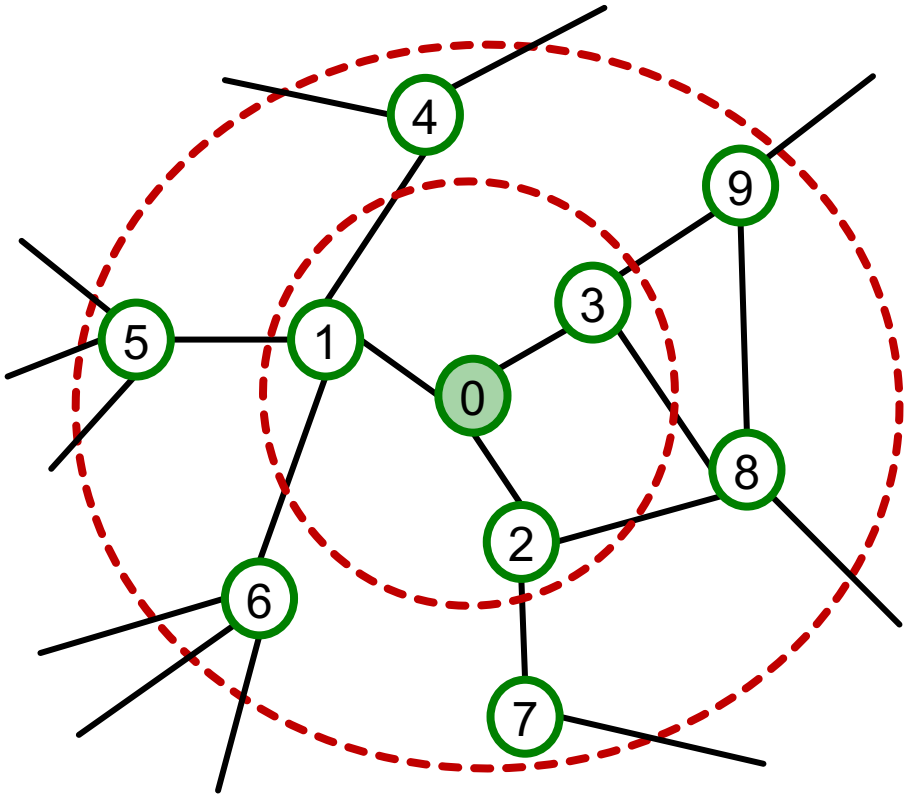
  - **Sampled nodes tend to be isolated from each other.**
  - **Recall:** GNN generate node embeddings by aggregating neighboring node features.
    - **GNN does not access to neighboring nodes within the mini-batch!**
- **Standard SGD cannot effectively train GNNs.**

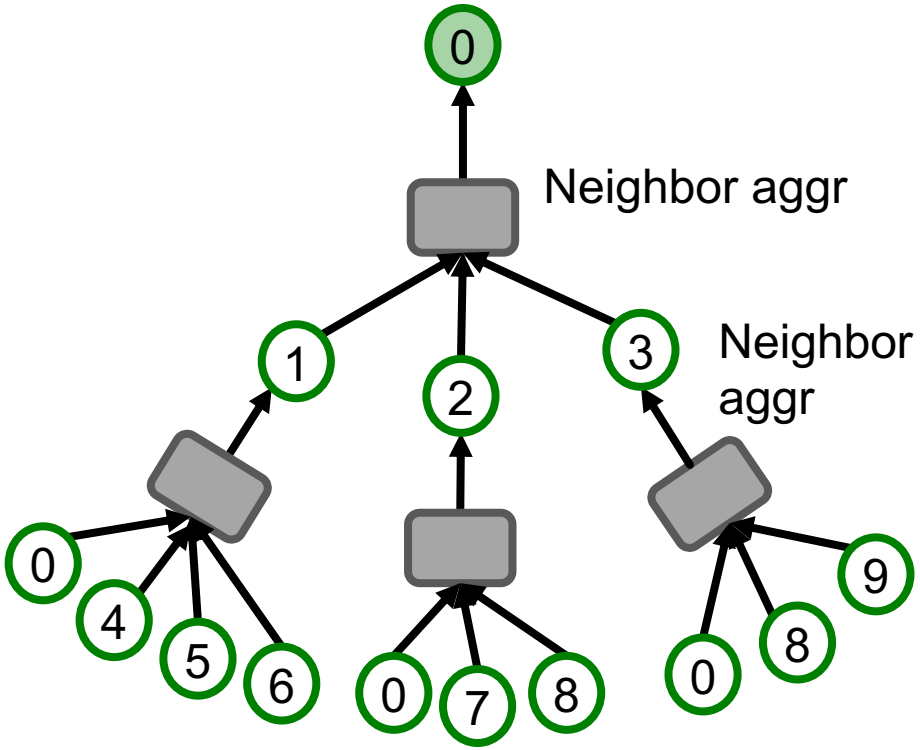
GraphSAGE Neighbor Sampling: Scaling up GNNs

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

Recall: Computational Graph

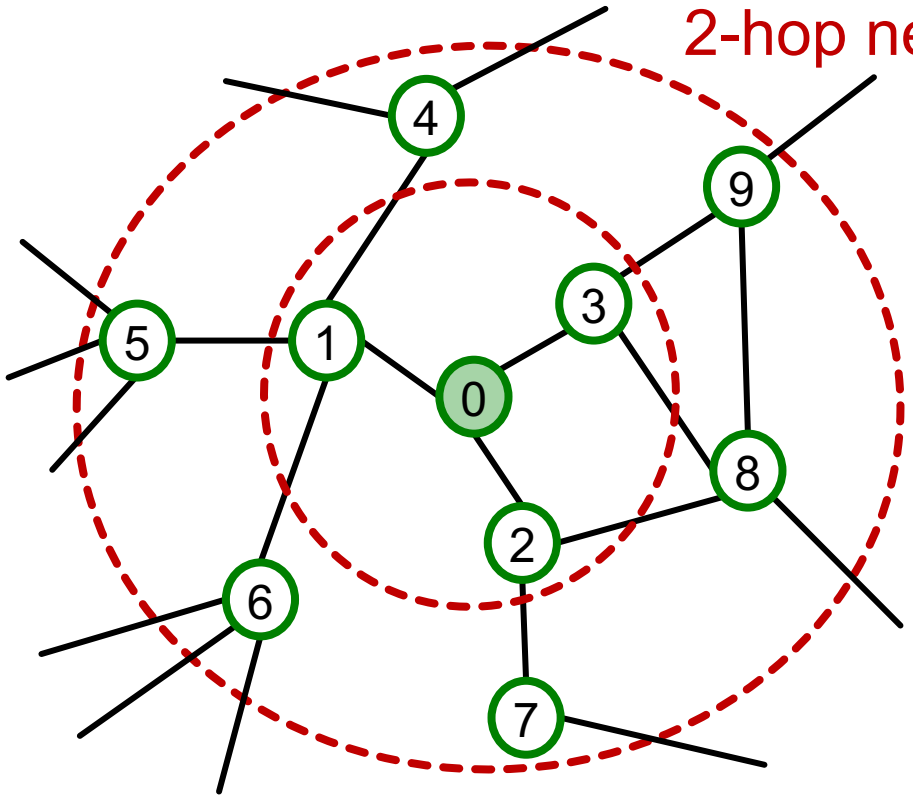
- **Recall**: GNNs generate node embeddings via neighbor aggregation.
  - Represented as a computational graph (right).

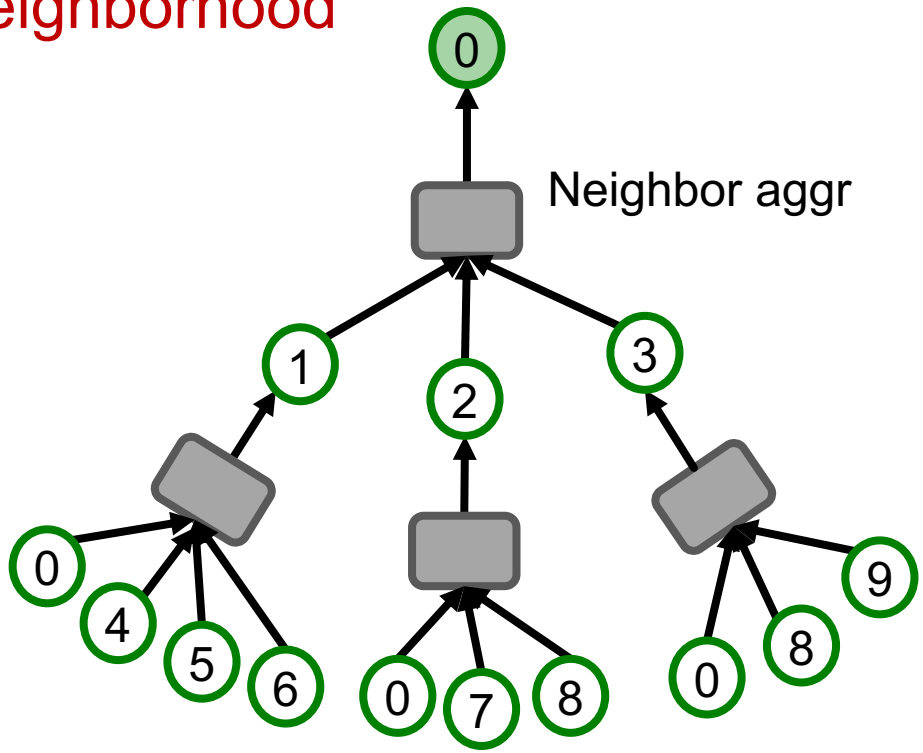




Recall: Computational Graph

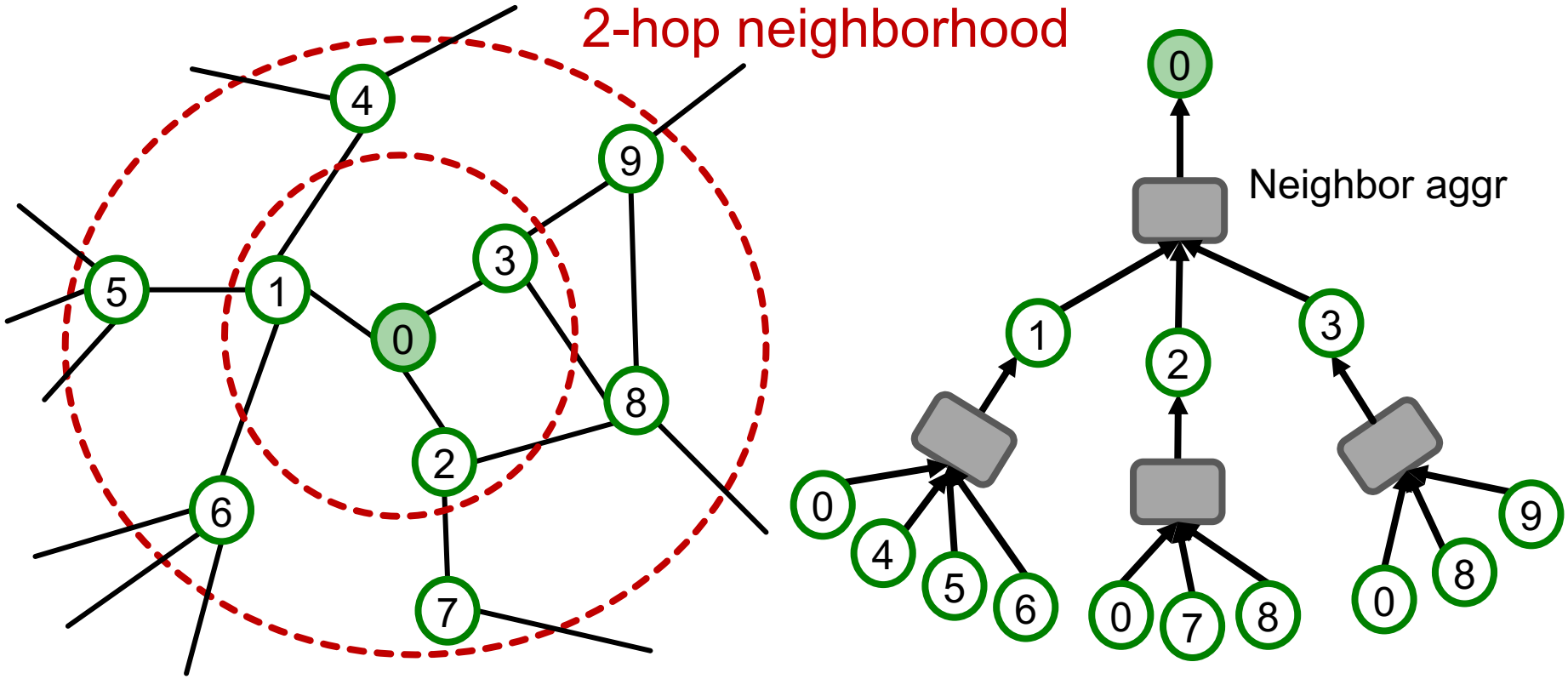
- **Observation:** A 2-layer GNN generates embedding of node "0" using 2-hop neighborhood structure and features.





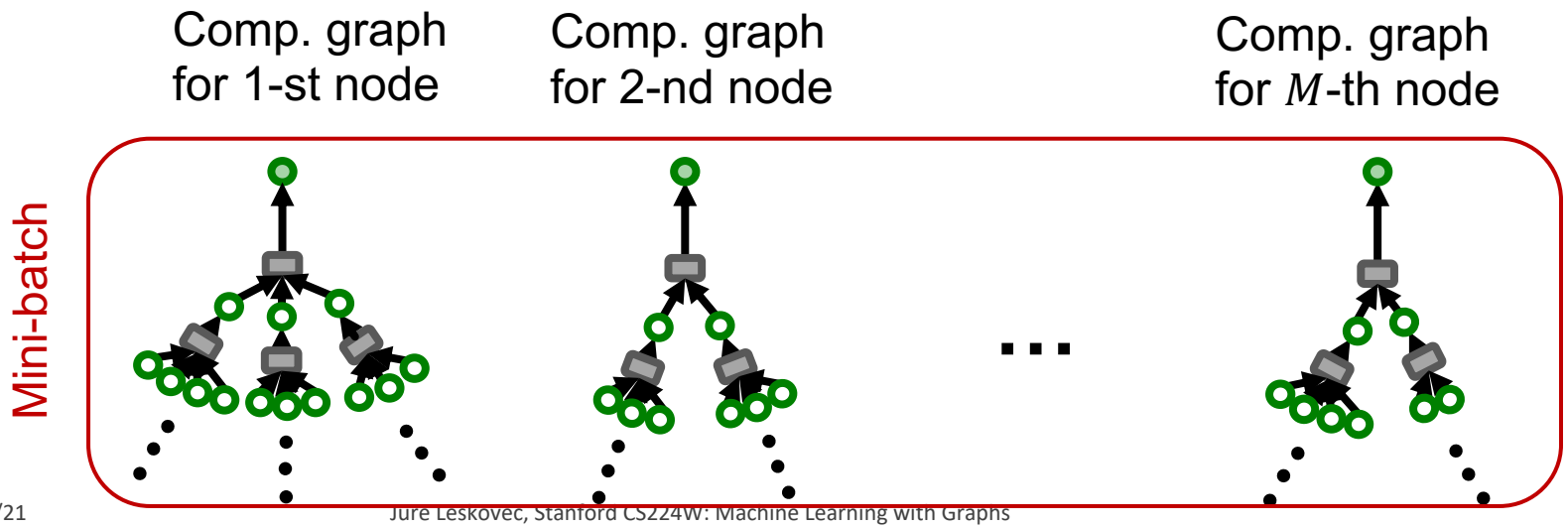
Recall: Computational Graph

- **Observation:** More generally, K-layer GNNs generate embedding of a node using K-hop neighborhood structure and features.



Computing Node Embeddings

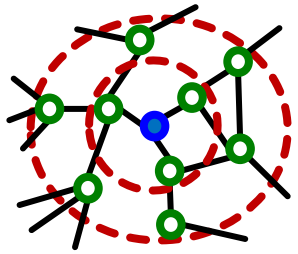
- **Key insight**: To compute embedding of a single node, all we need is **the K-hop neighborhood** (which defines the computation graph).
- Given a set of M different nodes in a **mini-batch**, we can generate their embeddings using M computational graphs. **Can be computed on GPU!**



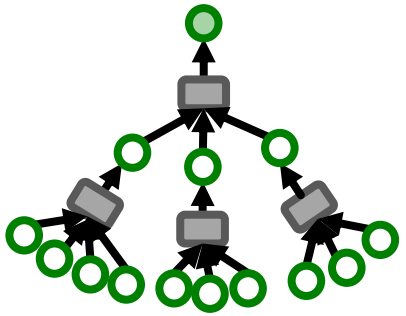
Stochastic Training of GNNs

- **We can now consider the following SGD strategy for training K-layer GNNs:**
  - Randomly sample M ($<< N$) nodes.
  - For each sampled node v:
    - Get **K-hop neighborhood**, and construct the **computation graph**.
    - Use the above to generate v's embedding.
  - Compute the loss $\ell_{sub}(\boldsymbol{\theta})$ averaged over the M nodes.
  - Perform SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \ell_{sub}(\boldsymbol{\theta})$

K-hop neighborhood

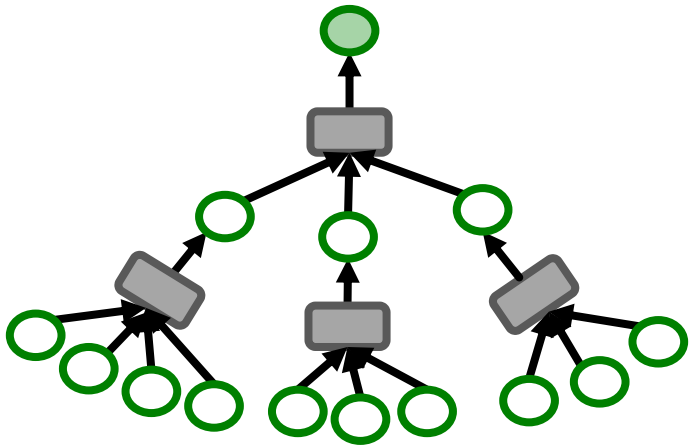


Computational graph



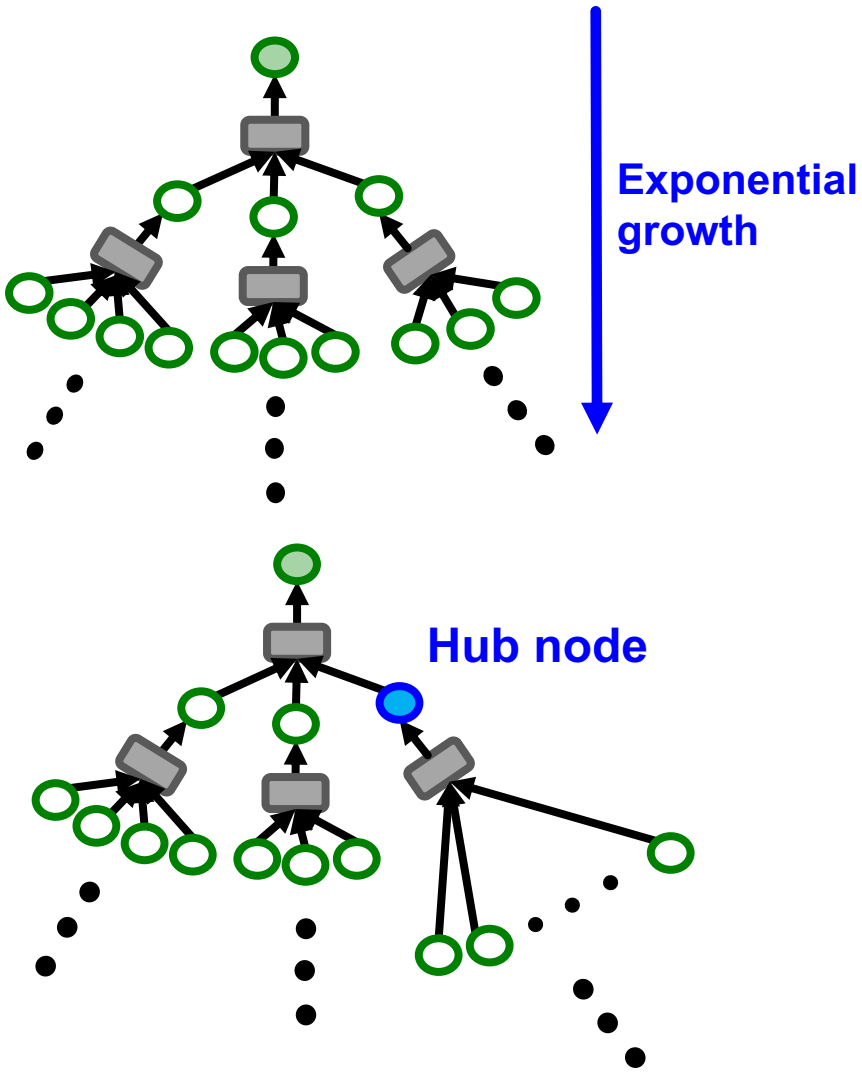
Issue with Stochastic Training

- **For each node**, we need to get **the entire K-hop neighborhood** and pass it through the computation graph.
- We need to aggregate lot of information just to compute one node embedding.
- **Computationally expensive.**



Issue with Stochastic Training

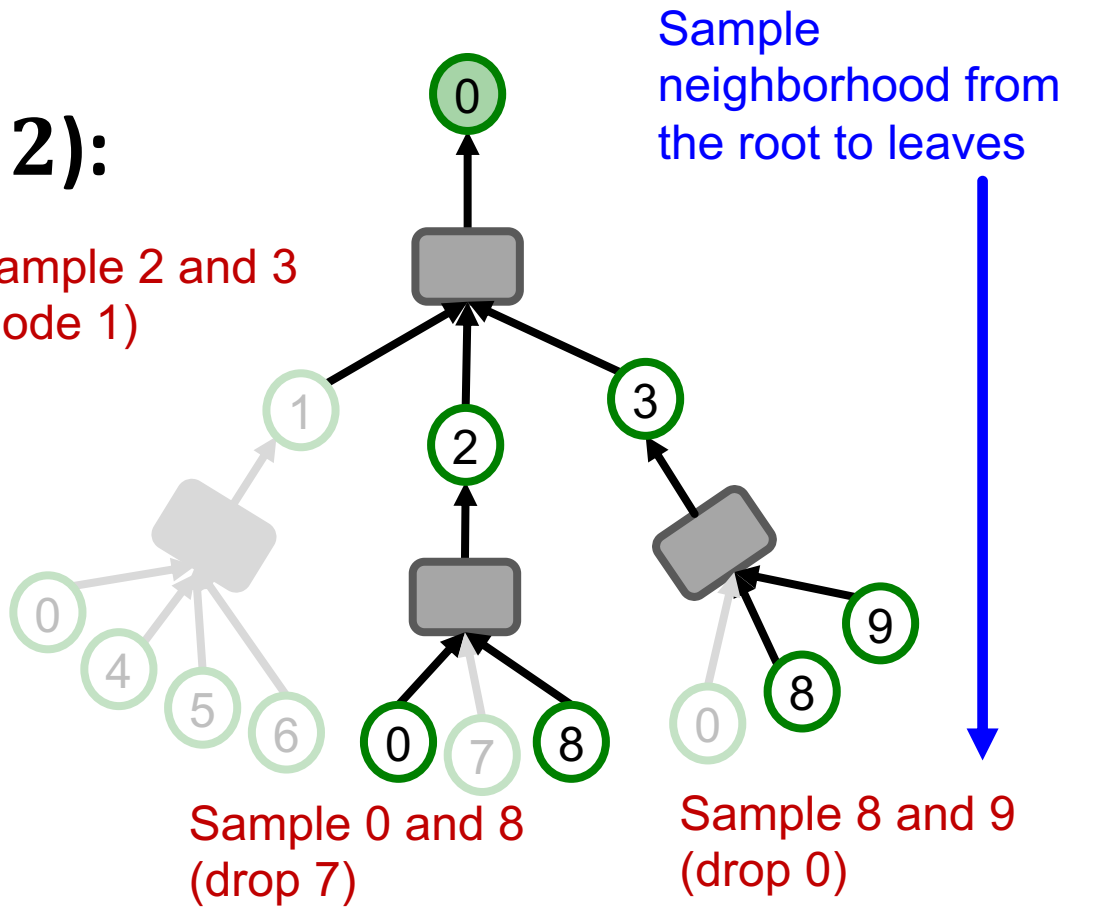
- **More details:**
  - Computation graph becomes **exponentially large** with respect to the layer size K.
  - Computation graph explodes when it hits a **hub node** (high-degree node).
- **Next:** Make the comp. graph more compact!



Neighbor Sampling

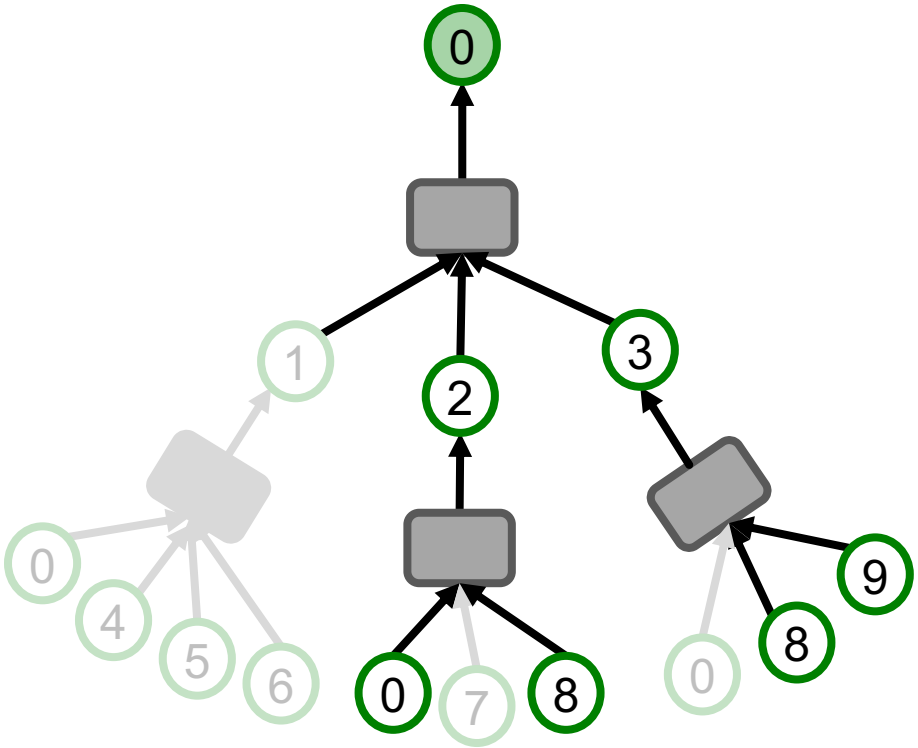
Key idea: **Construct the computational graph** by (randomly) sampling at most H neighbors at each hop.

- **Example ($H = 2$):**



Neighbor Sampling

We can use the pruned computational graph to more efficiently compute node embeddings.



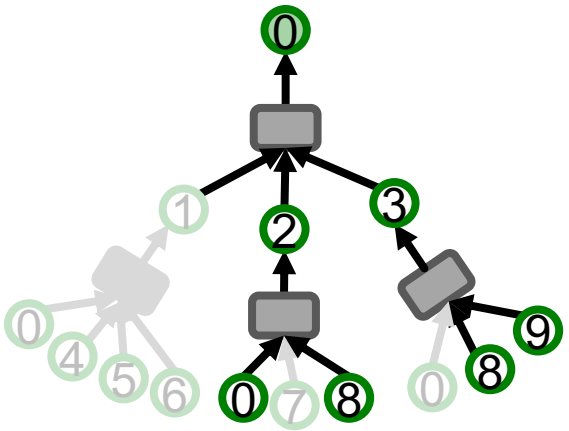
Neighbor Sampling Algorithm

Neighbor sampling for K-layer GNN

- For $k = 1, 2, \ldots, K$:
  - For each node in k-hop neighborhood:
  - (Randomly) sample at most H_k neighbors:

1st-hop neighborhood Sample $H_1 = 2$ neighbors

2nd-hop neighborhood Sample $H_2 = 2$ neighbors



- K-layer GNN will at most involve $\prod_{k=1}^{K} H_k$ leaf nodes in comp. graph.

Scaling up by Simplifying GNNs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

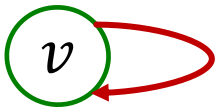
http://cs224w.stanford.edu

Roadmap of Simplifying GCN

- **We start from Graph Convolutional Network** (GCN) [Kipf & Welling ICLR 2017].
- We simplify GCN by **removing the non-linear activation** from the GCN [Wu et al. ICML 2019].
  - *Wu et al.* demonstrated that the performance on benchmark is not much lower by the simplification.
- Simplified GCN turns out to be extremely scalable by the model design.

Recall: GCN (mean-pool)

- **Given**: Graph $G = (V, E)$ with input node features X_v for $v \in V$, where $\boldsymbol{E}$ **includes the self-loop**:
  - $(\boldsymbol{v}, \boldsymbol{v}) \in \boldsymbol{E}$ **for all** $\boldsymbol{v} \in \boldsymbol{V}$.



- Set input node embeddings: $h_v^{(0)} = X_v$ for $v \in V$.
- For $k \in \{0, \ldots, K-1\}$:
  - For all $v \in V$, aggregate neighboring information as

$$h_v^{(k+1)} = \mathrm{ReLU}\left(\boldsymbol{W_k} \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)}\right)$$

Trainable weight matrices (i.e., what we learn)

Mean-pooling

- **Final node embedding**: $z_v = h_v^{(K)}$

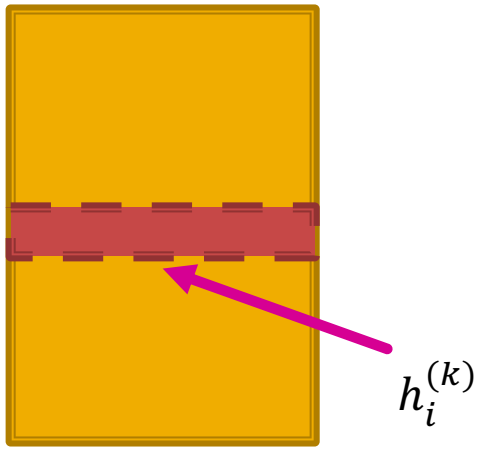
Recall: Matrix Formulation of GCN

GCN aggregations can be formulated as matrix vector product:

- Let $\boldsymbol{H}^{(k)} = [h_1^{(1)} \ldots h_{|V|}^{(k)}]^{\mathrm{T}}$
- Let $\boldsymbol{A}$ be the adjacency matrix (w/ self-loop)
- Then: $\sum_{u \in N(v)} h_u^{(k)} = A_{v,:}\boldsymbol{H}^{(k)}$
- Let $\boldsymbol{D}$ be diagonal matrix where $D_{v,v} = \mathrm{Deg}(v) = |N(v)|$
  - The inverse of D: D^{-1} is also diagonal: $D_{v,v}^{-1} = 1/|N(v)|$
- **Therefore,**

$$\frac{1}{|N(v)|}\sum_{u \in N(v)} h_u^{(k)} \Rightarrow \boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{H}^{(k)}$$

Matrix of hidden embeddings $\boldsymbol{H}^{(k)}$



Recall: Matrix Formulation of GCN

- **GCN's neighbor aggregation:**

$$h_v^{(k+1)} = \mathrm{ReLU}\left(\boldsymbol{W_k} \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)}\right)$$

- **In matrix form**:

$$\boldsymbol{H}^{(k+1)} = \mathrm{ReLU}\left(\tilde{\boldsymbol{A}} \boldsymbol{H}^{(k)} \boldsymbol{W}_k^{\mathrm{T}}\right)$$

where $\tilde{\boldsymbol{A}} = \boldsymbol{D}^{-1}\boldsymbol{A}$.

- **Note:** The original GCN uses re-normalized version: $\tilde{\boldsymbol{A}} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$.
  - Empirically, this version of $\tilde{\boldsymbol{A}}$ often gives better performance than $\boldsymbol{D}^{-1}\boldsymbol{A}$.

Simplifying GCN

- **Simplify GCN by removing ReLU non-linearity:**

$$\boldsymbol{H}^{(k+1)} = \widetilde{\boldsymbol{A}}\boldsymbol{H}^{(k)}\boldsymbol{W}_{\boldsymbol{k}}^{\mathrm{T}}$$

- The final node embedding matrix is given as

$$\boldsymbol{H}^{(K)} = \widetilde{\boldsymbol{A}}\,\boldsymbol{H}^{(K-1)}\,\boldsymbol{W}_{\boldsymbol{K-1}}^{\mathrm{T}}$$

$$= \widetilde{\boldsymbol{A}}\left(\widetilde{\boldsymbol{A}}\boldsymbol{H}^{(K-2)}\boldsymbol{W}_{\boldsymbol{K-2}}^{T}\right)\boldsymbol{W}_{\boldsymbol{K-1}}^{\mathrm{T}}$$

$$\cdots = \widetilde{\boldsymbol{A}}\left(\widetilde{\boldsymbol{A}}\left(\cdots\left(\widetilde{\boldsymbol{A}}\,\boldsymbol{H}^{(0)}\,\boldsymbol{W}_{\boldsymbol{0}}^{\mathrm{T}}\right)\cdots\right)\boldsymbol{W}_{\boldsymbol{K-2}}^{\mathrm{T}}\right)\boldsymbol{W}_{\boldsymbol{K-1}}^{\mathrm{T}}$$

$$= \widetilde{\boldsymbol{A}}^{K}\,\boldsymbol{X}\left(\boldsymbol{W}_{\boldsymbol{0}}^{\mathrm{T}}\cdots\boldsymbol{W}_{\boldsymbol{K-1}}^{\mathrm{T}}\right)$$

Composition of linear transformation is still linear!

$$= \widetilde{\boldsymbol{A}}^{K}\,\boldsymbol{X}\,\boldsymbol{W}^{\mathrm{T}} \quad \text{where } \boldsymbol{W} \equiv \boldsymbol{W}_{\boldsymbol{K-1}}\cdots\boldsymbol{W}_{\boldsymbol{0}}$$

Simplifying GCN (1)

- Removing ReLU significantly simplifies GCN!

$$\boldsymbol{H}^{(K)} = \widetilde{\boldsymbol{A}}^{K}\,\boldsymbol{X}\,\boldsymbol{W}^{\mathrm{T}}$$

- Notice $\widetilde{\boldsymbol{A}}^{K}\,\boldsymbol{X}$ does not contain any learnable parameters; hence, it **can be pre-computed.**
  - Efficiently computable as a sequence of sparse-matrix vector products:
  - Do $\boldsymbol{X} \leftarrow \widetilde{\boldsymbol{A}}\,\boldsymbol{X}$ for K times.

Simplifying GCN (2)

- Let $\tilde{\boldsymbol{X}} = \tilde{\boldsymbol{A}}^K \boldsymbol{X}$ be pre-computed matrix.
- Simplified GCN's final embedding is

$$\boldsymbol{H}^{(K)} = \tilde{\boldsymbol{X}} \boldsymbol{W}^{\mathrm{T}}$$

- It's just a **linear transformation of pre-computed matrix!**
- Back to the node embedding form:

$$h_v^{(K)} = \boldsymbol{W} \boxed{\tilde{\boldsymbol{X}}_v}$$

Pre-computed feature vector for node v

- Embedding of node v only depends on its own (pre-processed) feature!

Simplifying GCN (3)

- Once $\tilde{\boldsymbol{X}}$ is pre-computed, embeddings of M nodes can be generated in time linear in M:
  - Given M nodes $\{v_1, v_2, \ldots, v_M\}$, their embeddings are
    - $h_{v_1}^{(K)} = \boldsymbol{W}\tilde{\boldsymbol{X}}_{v_1}$,
    - $h_{v_2}^{(K)} = \boldsymbol{W}\tilde{\boldsymbol{X}}_{v_2}$,
    - ...
    - $h_{v_M}^{(K)} = \boldsymbol{W}\tilde{\boldsymbol{X}}_{v_M}$.

Simplified GCN: Summary

In summary, simplified GCN consists of **two steps**:

- **Pre-processing step**:
  - Pre-compute $\widetilde{\boldsymbol{X}} = \widetilde{\boldsymbol{A}}^K \boldsymbol{X}$. Can be done on CPU.
- **Mini-batch training step**:
  - For each mini-batch, randomly-sample M nodes $\{v_1, v_2, \ldots, v_M\}$.
  - Compute their embeddings by
    - $h_{v_1}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_1}, h_{v_2}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_2}, \ldots, h_{v_M}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_M}$
  - Use the embeddings to make prediction and compute the loss averaged over the M data points.
  - Perform SGD parameter update.