Stanford CS224W: Graph Neural Networks

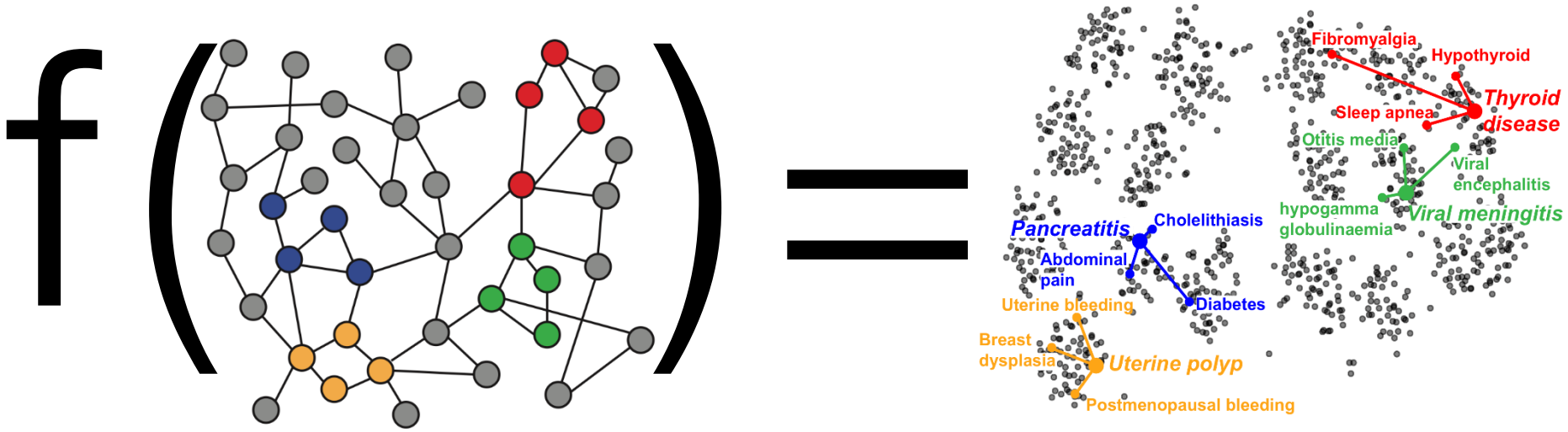
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Recap: Node Embeddings

- **Intuition:** Map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together



Input graph

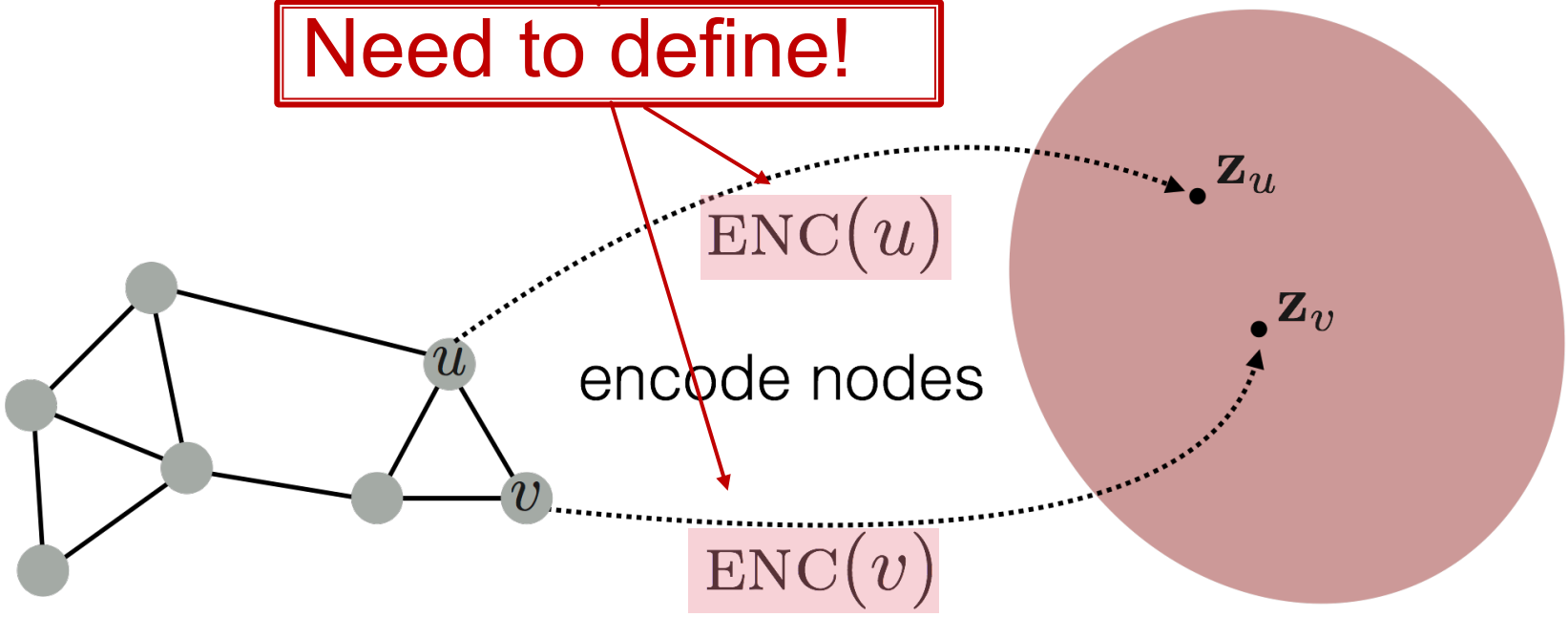
2D node embeddings

How to learn mapping function f?

Recap: Node Embeddings

Goal: $\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$

Need to define!



Input network

d-dimensional embedding space

Recap: Two Key Components

- **Encoder:** maps each node to a low-dimensional vector

$$\text{ENC}(v) = \mathbf{z}_v$$

d-dimensional embedding

node in the input graph

- **Similarity function:** specifies how the relationships in vector space map to the relationships in the original network

$$\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$$

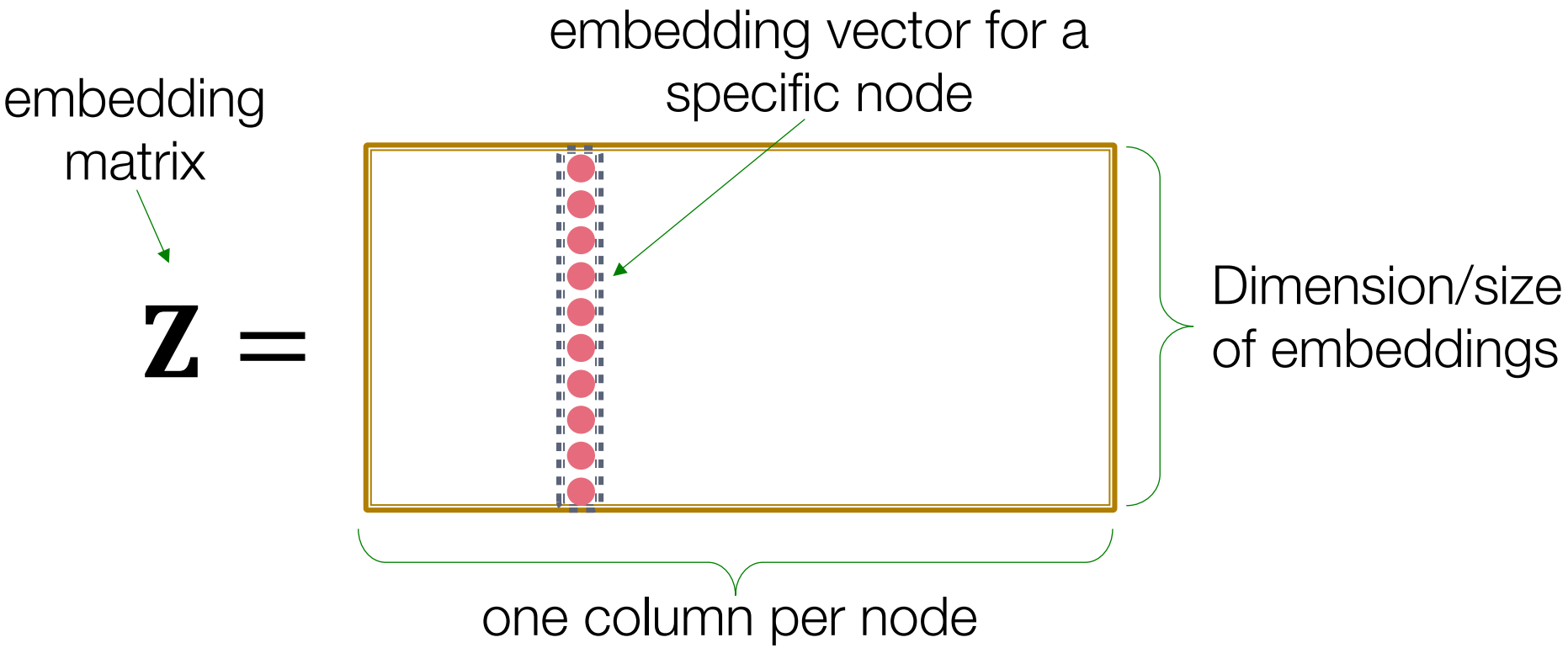
Decoder

Similarity of u and v in the original network

dot product between node embeddings

Recap: "Shallow" Encoding

Simplest encoding approach: **encoder is just an embedding-lookup**



Recap: Shallow Encoders

- **Limitations** of shallow embedding methods:
  - $O(|V|)$ **parameters are needed**:
    - No sharing of parameters between nodes
    - Every node has its own unique embedding
  - **Inherently "transductive"**:
    - Cannot generate embeddings for nodes that are not seen during training
  - **Do not incorporate node features**:
    - Many graphs have features that we can and should leverage

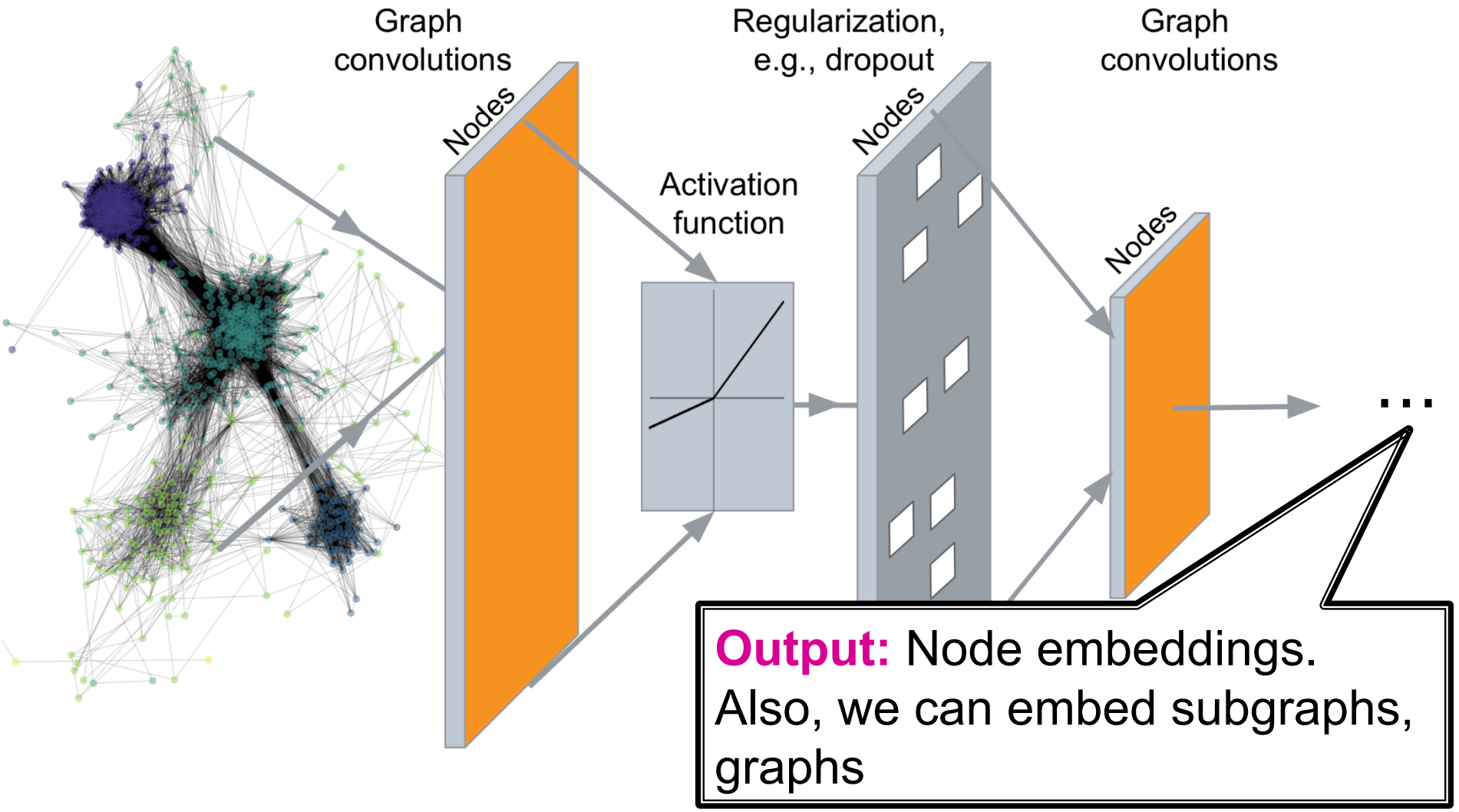
Today: Deep Graph Encoders

- **Today:** We will now discuss deep methods based on **graph neural networks (GNNs):**

$$\text{ENC}(v) = \text{multiple layers of non-linear transformations based on graph structure}$$

- **Note:** All these deep encoders can be **combined with node similarity functions** defined in the lecture 3

Deep Graph Encoders



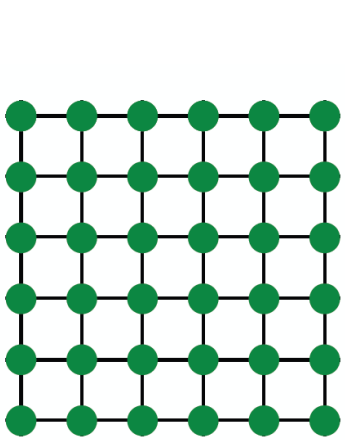
Output: Node embeddings.
Also, we can embed subgraphs, graphs

Tasks on Networks

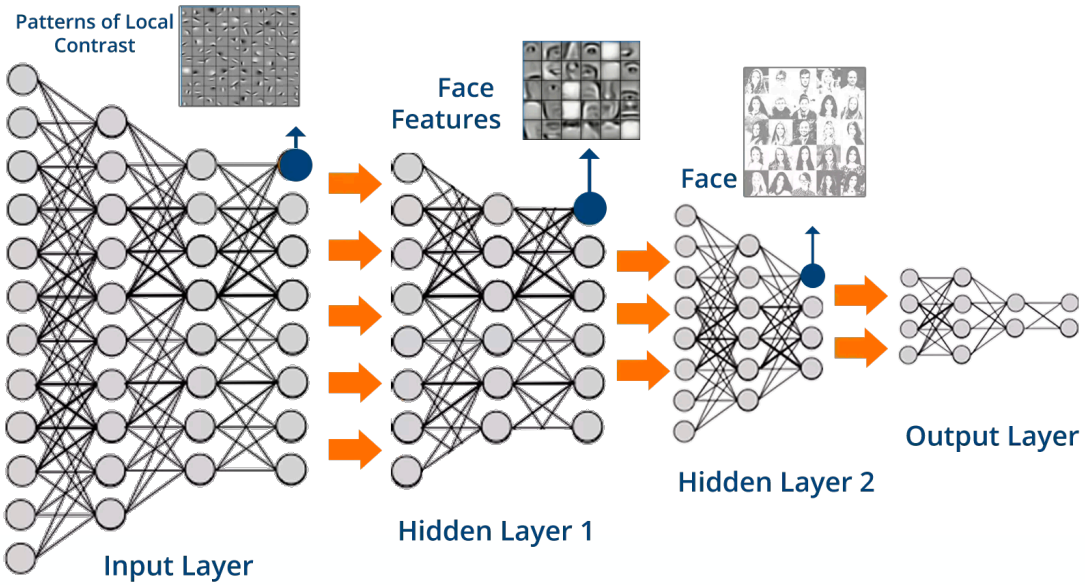
Tasks we will be able to solve:

- Node classification
  - Predict a type of a given node
- Link prediction
  - Predict whether two nodes are linked
- Community detection
  - Identify densely linked clusters of nodes
- Network similarity
  - How similar are two (sub)networks

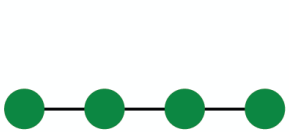
Modern ML Toolbox

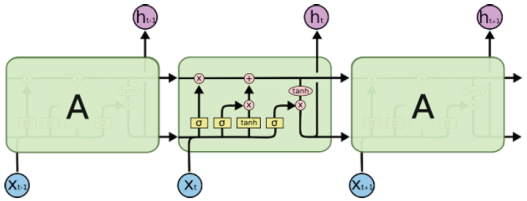


Images



Text/Speech



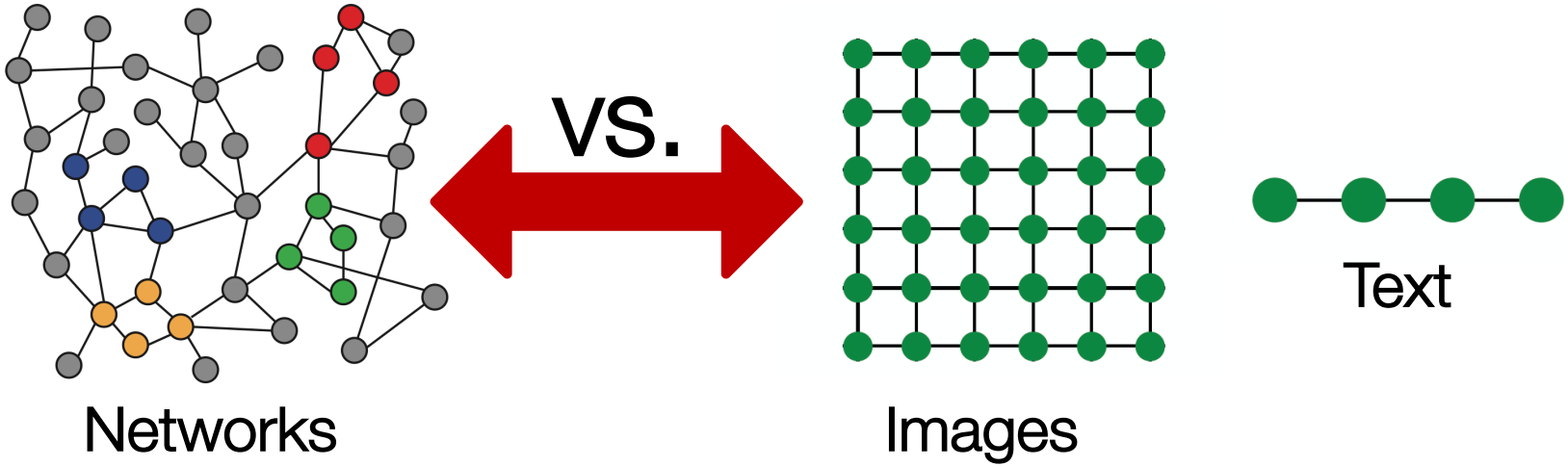


Modern deep learning toolbox is designed for simple sequences & grids

Why is it Hard?

But networks are far more complex!

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)



- No fixed node ordering or reference point
- Often dynamic and have multimodal features

Outline of Today's Lecture

1. **Basics of deep learning**

2. **Deep learning for graphs**

3. **Graph Convolutional Networks and GraphSAGE**

Stanford CS224W: Basics of Deep Learning

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Machine Learning as Optimization

- **Supervised learning:** we are given input $\boldsymbol{x}$, and the goal is to predict label $\boldsymbol{y}$
- **Input x can be:**
  - Vectors of real numbers
  - Sequences (natural language)
  - Matrices (images)
  - Graphs (potentially with node and edge features)
- **We formulate the task as an optimization problem**

Machine Learning as Optimization

- **Formulate the task as an optimization problem:**

$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$$

Objective function

- Θ: a set of **parameters** we optimize
  - Could contain one or more scalars, vectors, matrices …
  - E.g. $\Theta = \{Z\}$ in the shallow encoder (the embedding lookup)

- $\mathcal{L}$: **loss function**. Example: L2 loss

$$\mathcal{L}\big(\boldsymbol{y}, f(\boldsymbol{x})\big) = \|y - f(x)\|_2$$

  - Other common loss functions:
    - L1 loss, huber loss, max margin (hinge loss), cross entropy …
    - See https://pytorch.org/docs/stable/nn.html#loss-functions

Loss Function Example

- One common loss for classification: **cross entropy (CE)**
- Label $\boldsymbol{y}$ is a categorical vector (one-hot encoding)
  - e.g. $\boldsymbol{y} =$ | 0 | 0 | 1 | 0 | 0 |

y is of class "3"

- $f(\boldsymbol{x}) = \text{Softmax}(g(\boldsymbol{x}))$
  - Recall from lecture 3: $f(\boldsymbol{x})_i = \frac{e^{g(x)_i}}{\sum_{j=1}^{C} e^{g(x)_j}}$,

$g(x)_i$ denotes i-th coordinate of the vector output of func. $g(x)$

where C is the number of classes.

  - e.g. $f(\boldsymbol{x}) =$ | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |

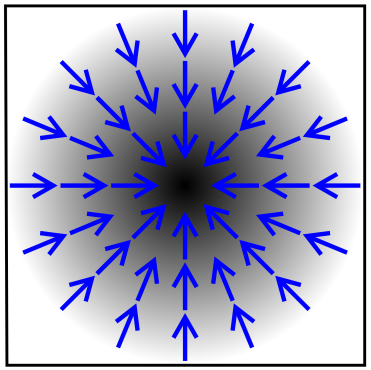
- $\text{CE}(\boldsymbol{y}, f(\boldsymbol{x})) = -\sum_{i=1}^{C}(y_i \log f(x)_i)$
  - $y_i, f(x)_i$ are the **actual** and **predicted** value of the i-th class.
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Total loss over all training examples:**
  - $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \text{CE}(\boldsymbol{y}, f(\boldsymbol{x}))$
  - $\mathcal{T}$: training set containing all pairs of data and labels $(\boldsymbol{x}, \boldsymbol{y})$

Machine Learning as Optimization

- **How to optimize the objective function?**
- **Gradient vector:** Direction and rate of fastest increase

Partial derivative

$$\nabla_{\Theta}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \ldots\right)$$



https://en.wikipedia.org/wiki/Gradient

  - Θ_1, Θ_2 … : components of Θ
- Recall **directional derivative** of a multi-variable function (e.g. $\mathcal{L}$) along a given vector represents the instantaneous rate of change of the function along the vector.
- Gradient is the directional derivative in the **direction of largest increase**

Gradient Descent

- **Iterative algorithm:** repeatedly update weights in the (opposite) direction of gradients until convergence

$$\Theta \leftarrow \Theta - \eta \nabla_{\Theta} \mathcal{L}$$

- **Training:** Optimize Θ iteratively
  - **Iteration**: 1 step of gradient descent

- **Learning rate (LR) η:**
  - Hyperparameter that controls the size of gradient step
  - Can vary over the course of training (LR scheduling)
- Ideal termination condition: **0** gradient
  - In practice, we stop training if it no longer improves performance on **validation set** (part of dataset we hold out from training)

Stochastic Gradient Descent (SGD)

- **Problem with gradient descent:**
  - Exact gradient requires computing $\nabla_\Theta \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$, where $\boldsymbol{x}$ is the **entire** dataset!
    - This means summing gradient contributions over all the points in the dataset
    - Modern datasets often contain billions of data points
    - Extremely expensive for every gradient descent step
- **Solution: Stochastic gradient descent (SGD)**
  - At every step, pick a different **minibatch** $\boldsymbol{\mathcal{B}}$ containing a subset of the dataset, use it as input $\boldsymbol{x}$

Minibatch SGD

- **Concepts:**
  - **Batch size**: the number of data points in a minibatch
    - E.g. number of nodes for node classification task
  - **Iteration**: 1 step of SGD on a minibatch
  - **Epoch**: one full pass over the dataset (# iterations is equal to ratio of dataset size and batch size)

- SGD is unbiased estimator of full gradient:
  - But there is no guarantee on the rate of convergence
  - In practice often requires tuning of learning rate
- Common optimizer that improves over SGD:
  - Adam, Adagrad, Adadelta, RMSprop …

Neural Network Function

- **Objective:** $\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$
- In deep learning, the function f can be very complex
- To start simple, consider linear function

$$f(\boldsymbol{x}) = W \cdot \boldsymbol{x}, \qquad \Theta = \{W\}$$

- If f returns a scalar, then W is a learnable **vector**

$$\nabla_W f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \ldots\right)$$

- If f returns a vector, then W is the **weight matrix**

$$\nabla_W f = W^T$$

Jacobian matrix of f

Back-propagation

- **How about a more complex function:**

$$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x}), \qquad \Theta = \{W_1, W_2\}$$

- Recall **chain rule**:

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

In other words:

$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$

$h(x) = W_1\boldsymbol{x}$

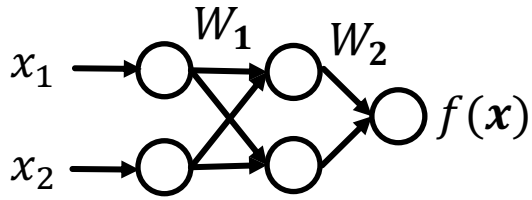
$\mathrm{g}(z) = W_2 z$

- E.g. $\nabla_{\boldsymbol{x}} f = \frac{\partial f}{\partial (W_1\boldsymbol{x})} \cdot \frac{\partial (W_1\boldsymbol{x})}{\partial \boldsymbol{x}}$

- **Back-propagation**: Use of **chain rule** to propagate gradients of intermediate steps, and finally obtain gradient of $\mathcal{L}$ w.r.t. Θ

Back-propagation Example (1)

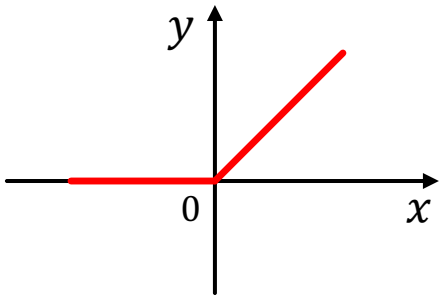
- **Example:** Simple 2-layer linear network
- $f(\boldsymbol{x}) = g\big(h(x)\big) = W_2(W_1\boldsymbol{x})$

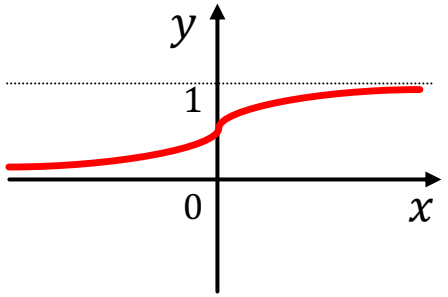


- $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{B}} \big|\big|\big(\boldsymbol{y}, -f(\boldsymbol{x})\big)\big|\big|_2$ sums the L2 loss in a minibatch $\mathcal{B}$
- **Hidden layer:** intermediate representation for input $\boldsymbol{x}$
  - Here we use $h(x) = W_1\boldsymbol{x}$ to denote the hidden layer
  - $f(\boldsymbol{x}) = W_2 h(x)$

Non-linearity

- Note that in $f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$, W_2W_1 is another matrix (vector, if we do binary classification)
- Hence $f(\boldsymbol{x})$ is still linear w.r.t. $\boldsymbol{x}$ no matter how many weight matrices we compose
- **Introduce non-linearity:**
  - **Rectified linear unit (ReLU)**
    $ReLU(x) = \max(x, 0)$
  - **Sigmoid**
    $$\sigma(x) = \frac{1}{1 + e^{-x}}$$



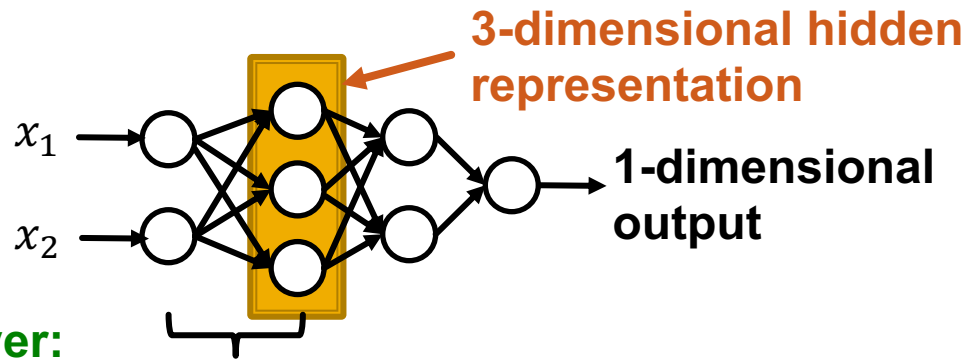


Multi-layer Perceptron (MLP)

- **Each layer of MLP combines linear transformation and non-linearity:**

$$\boldsymbol{x}^{(l+1)} = \sigma(W_l \boldsymbol{x}^{(l)} + b^l)$$

  - where W_l is weight matrix that transforms hidden representation at layer l to layer $l+1$
  - b^l is bias at layer l, and is added to the linear transformation of $\boldsymbol{x}$
  - σ is non-linearity function (e.g., sigmod)
- Suppose $\boldsymbol{x}$ is 2-dimensional, with entries x_1 and x_2



Summary

- **Objective function:**

$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$$

- f can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN later)
- Sample a minibatch of input $\boldsymbol{x}$
- **Forward propagation:** compute $\mathcal{L}$ given $\boldsymbol{x}$
- **Back-propagation:** obtain gradient $\nabla_{\Theta}\mathcal{L}$ using a chain rule
- Use **stochastic gradient descent (SGD)** to optimize for Θ over many iterations

Outline of Today's Lecture

1. **Basics of deep learning** ✓
2. **Deep learning for graphs** 
3. **Graph Convolutional Networks and GraphSAGE**

Stanford CS224W: Deep Learning for Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Content

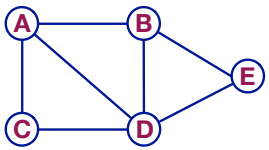
- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

Setup

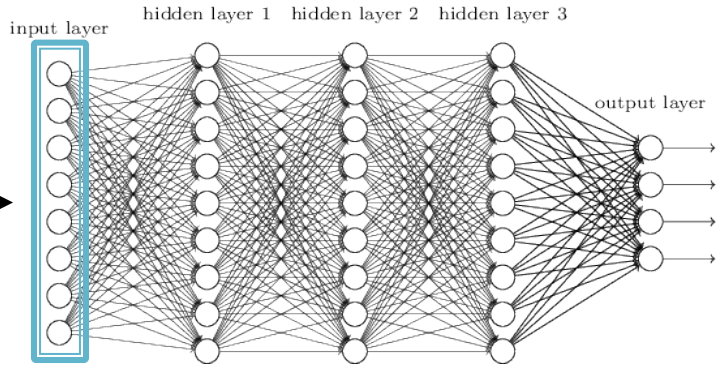
- **Assume we have a graph G:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the set of neighbors of v.
  - **Node features:**
    - Social networks: User profile, User image
    - Biological networks: Gene expression profiles, gene functional information
    - When there is no node feature in the graph dataset:
      - Indicator vectors (one-hot encoding of a node)
      - Vector of constant 1: [1, 1, …, 1]

A Naïve Approach

- Join adjacency matrix and features
- Feed them into a deep neural net:



	A	B	C	D	E	Feat	
A	0	1	1	1	0	1	0
B	1	0	0	1	1	0	0
C	1	0	0	1	0	0	1
D	1	1	1	0	1	1	1
E	0	1	0	1	0	1	0

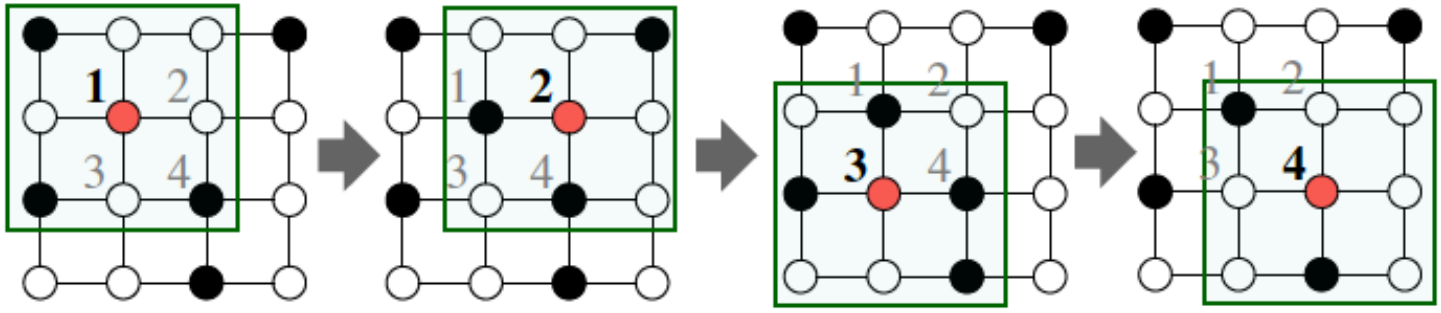


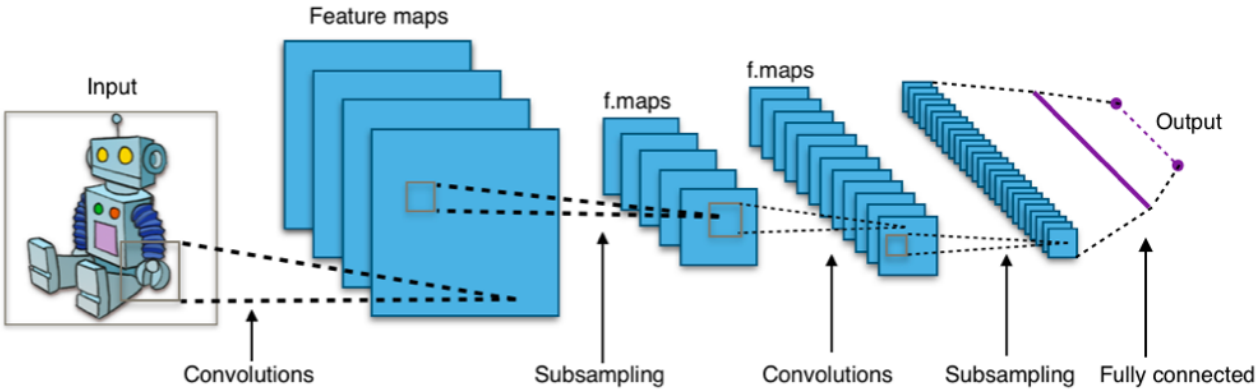
?

- Issues with this idea:
  - $O(|V|)$ parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering

Idea: Convolutional Networks

CNN on an image:



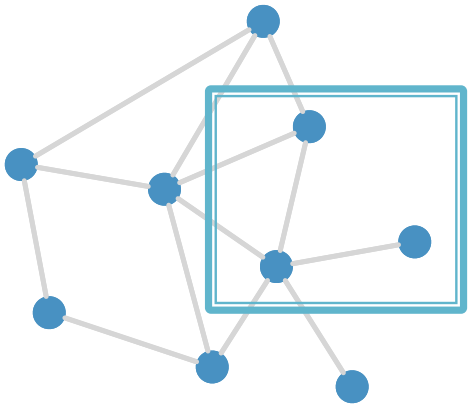


Goal is to generalize convolutions beyond simple lattices

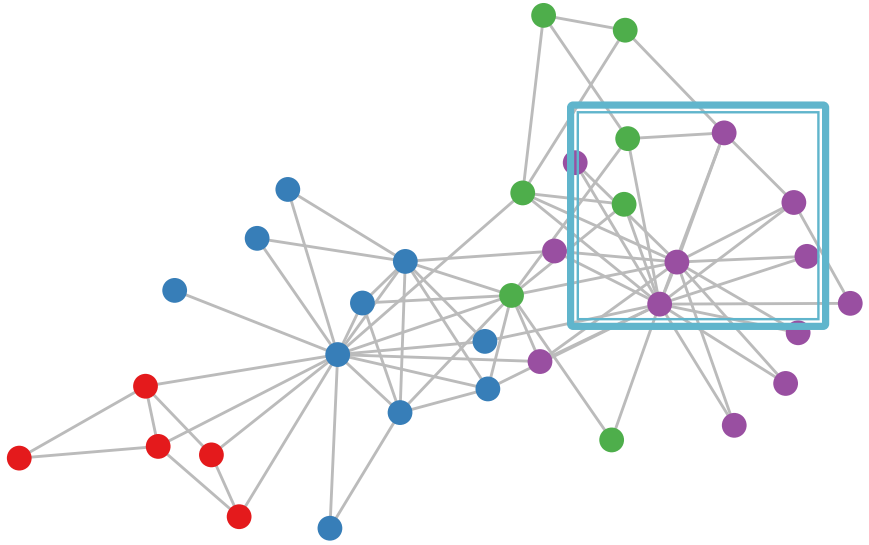
Leverage node features/attributes (e.g., text, images)

Real-World Graphs

But our graphs look like this:



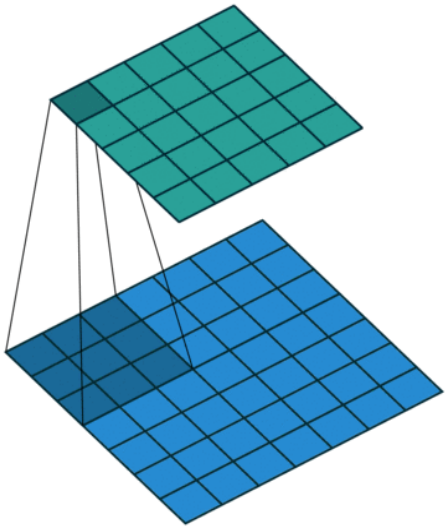
or this:



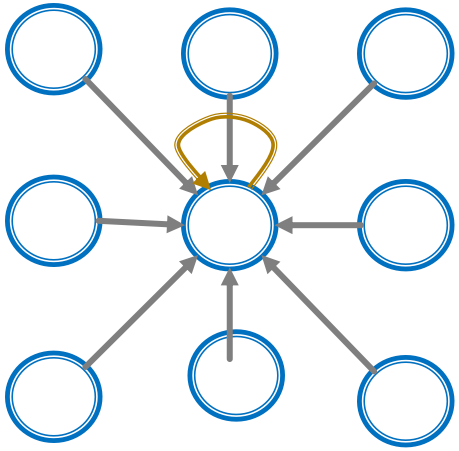
- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

From Images to Graphs

Single Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

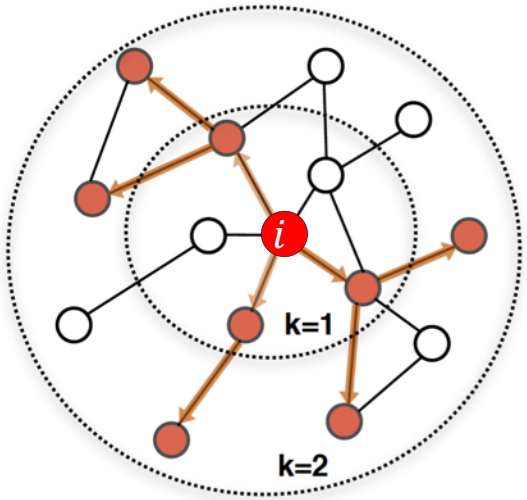
Idea: transform information at the neighbors and combine it:

- Transform "messages" h_i from neighbors: $W_i\ h_i$
- Add them up: $\sum_i W_i\ h_i$

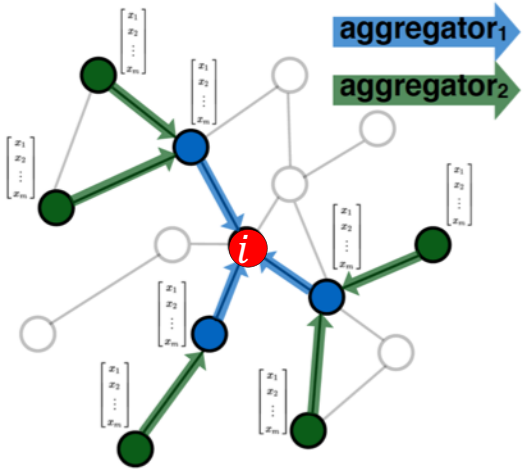
[Kipf and Welling, ICLR 2017]

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

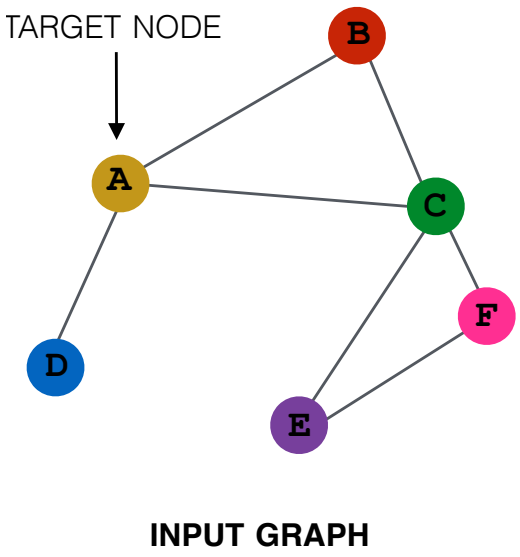


Propagate and transform information

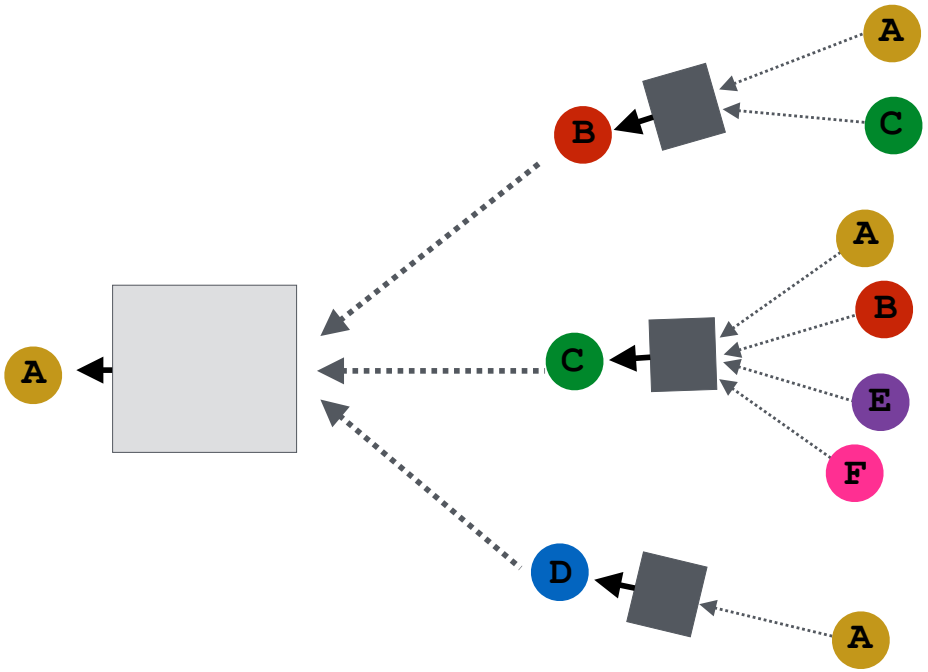
Learn how to propagate information across the graph to compute node features

Idea: Aggregate Neighbors

- **Key idea:** Generate node embeddings based on **local network neighborhoods**

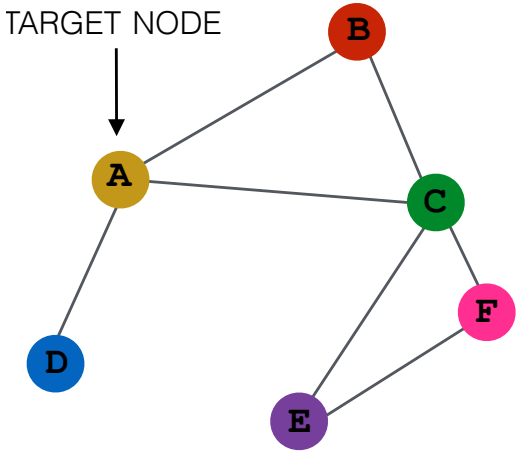


INPUT GRAPH

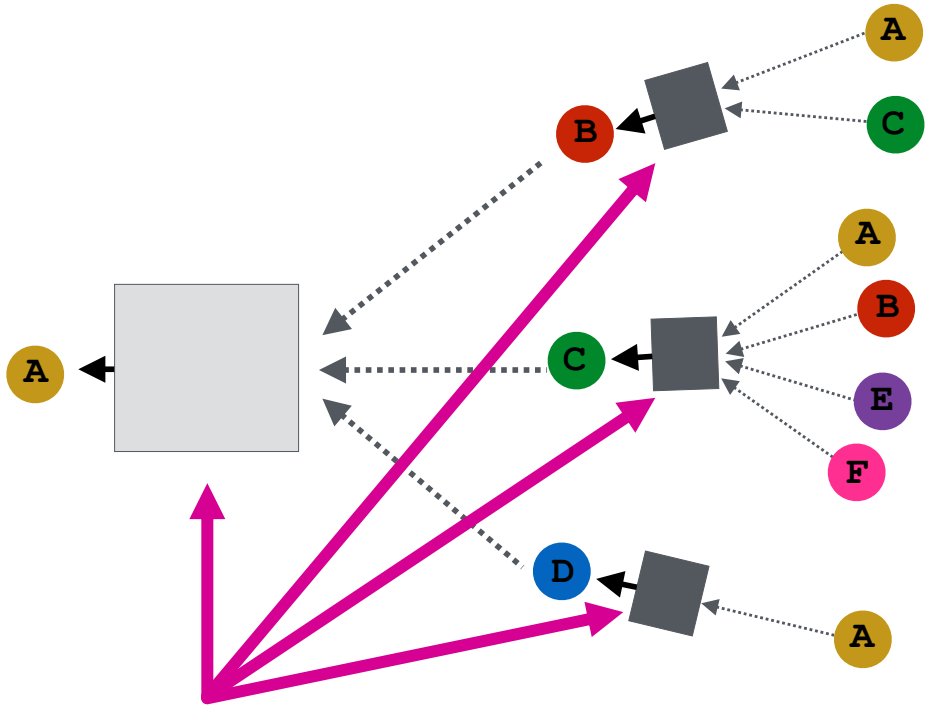


Idea: Aggregate Neighbors

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



INPUT GRAPH

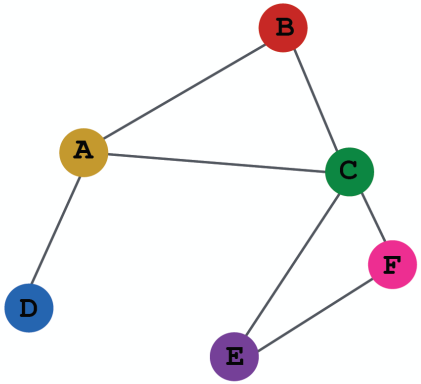


Neural networks

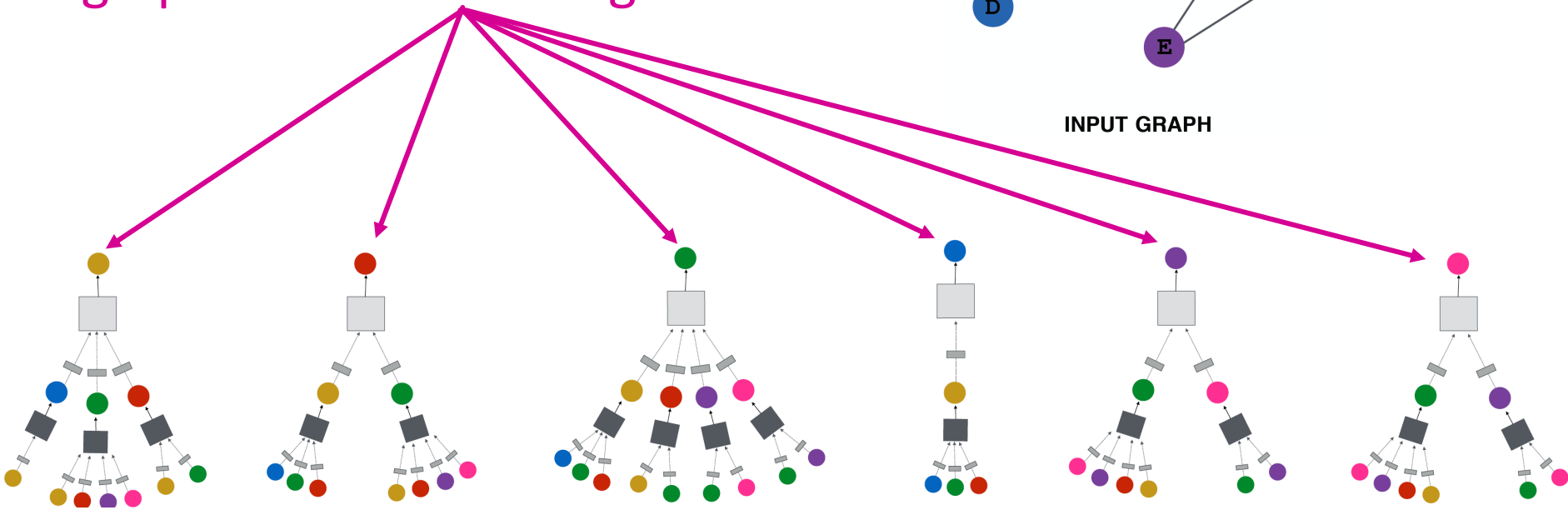
Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

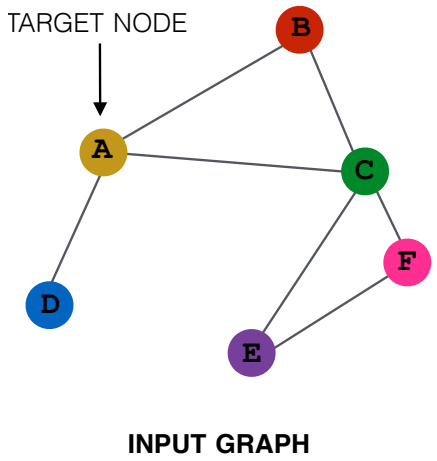


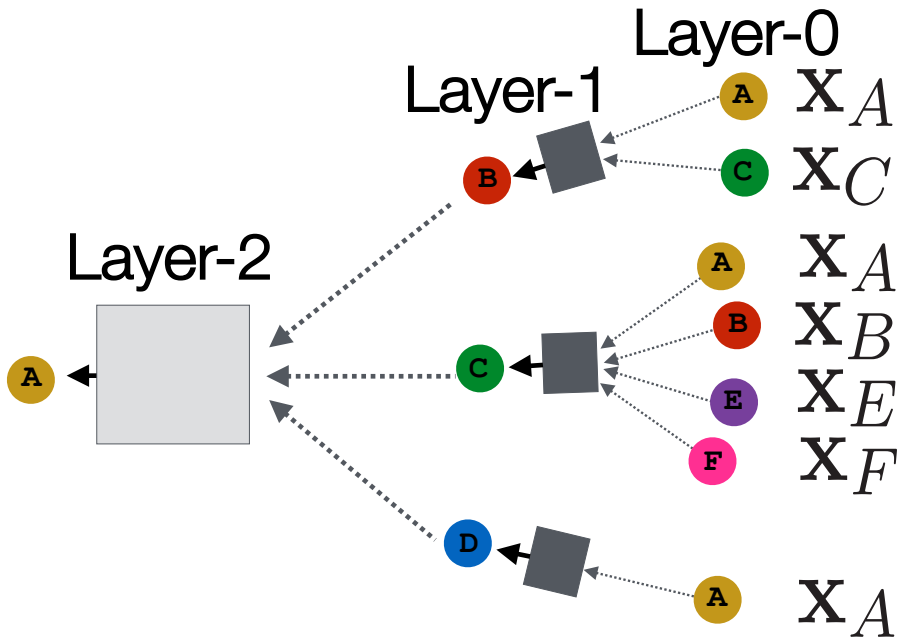
INPUT GRAPH



Deep Model: Many Layers

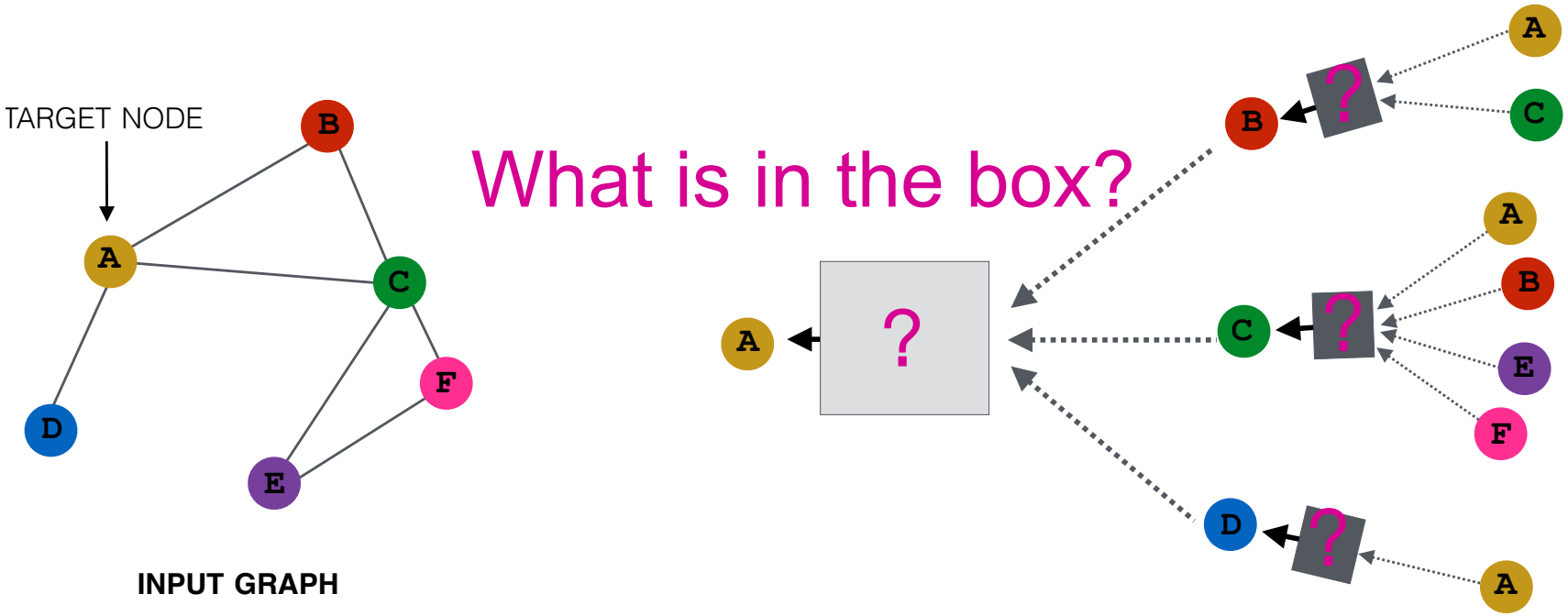
- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node u is its input feature, x_u
  - Layer-k embedding gets information from nodes that are K hops away





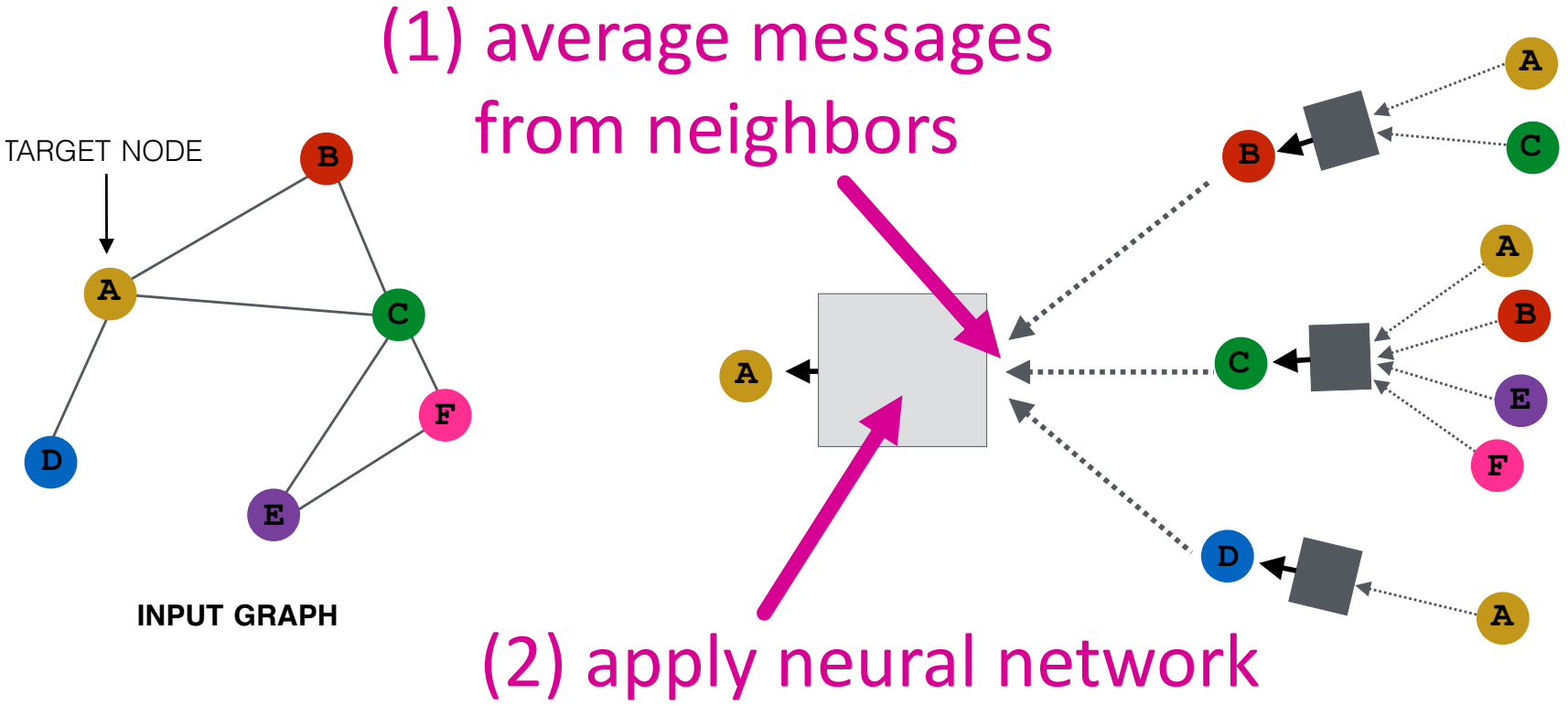
Neighborhood Aggregation

- **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers



Neighborhood Aggregation

- **Basic approach:** Average information from neighbors and apply a neural network



The Math: Deep Encoder

- **Basic approach:** Average neighbor messages and apply a neural network

Initial 0-th layer embeddings are equal to node features

$$\mathbf{h}_v^0 = \mathbf{x}_v$$

embedding of v at layer l

$$\mathbf{h}_v^{(l+1)} = \sigma\left(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathbf{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathbf{B}_l \mathbf{h}_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

$$\mathbf{z}_v = \mathbf{h}_v^{(L)}$$

Embedding after L layers of neighborhood aggregation

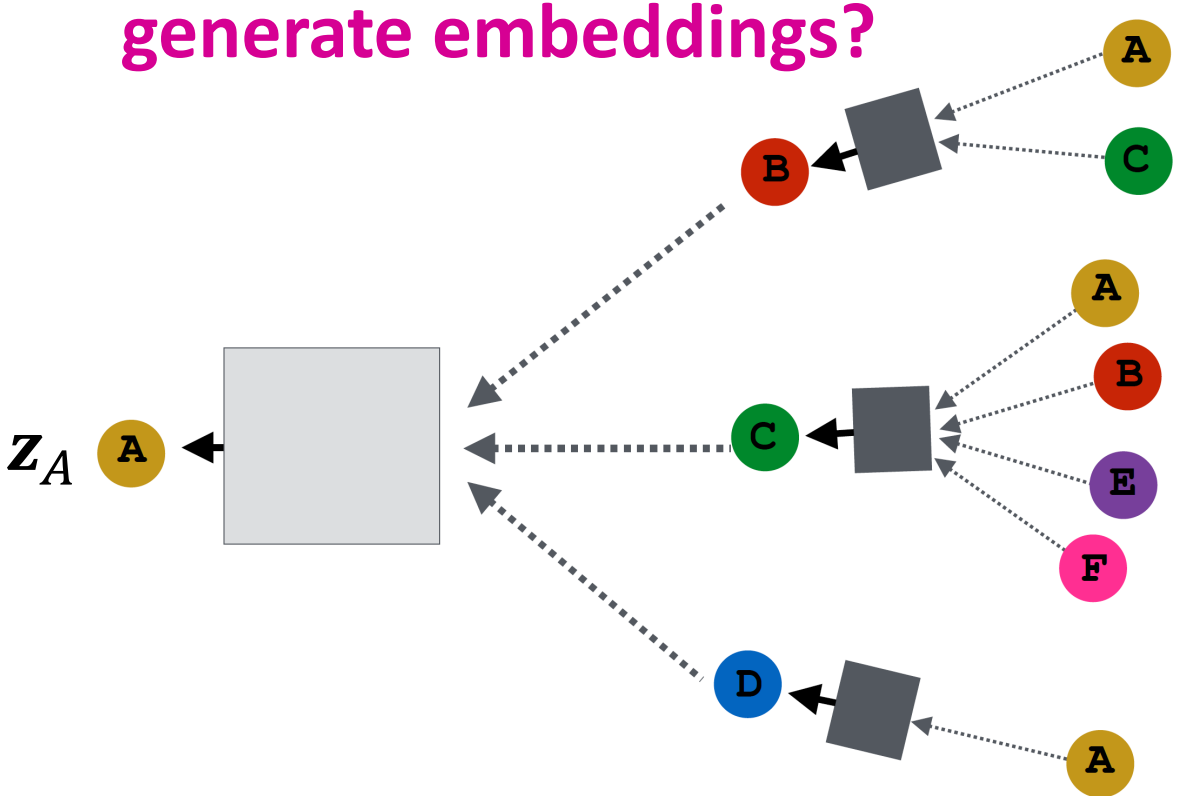
Non-linearity (e.g., ReLU)

Average of neighbor's previous layer embeddings

Total number of layers

Training the Model

How do we train the model to generate embeddings?



Need to define a loss function on the embeddings

Model Parameters

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

$$\mathrm{h}_v^{(0)} = \mathrm{x}_v$$

$$\mathrm{h}_v^{(l+1)} = \sigma\left(\mathrm{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

$$\mathrm{z}_v = \mathrm{h}_v^{(L)}$$

Final node embedding

We can feed these **embeddings into any loss function** and run SGD to **train the weight parameters**

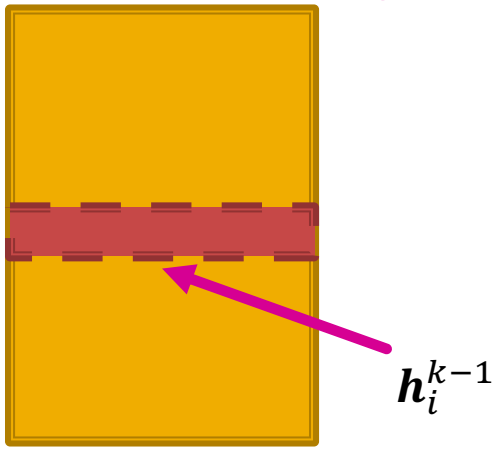
h_v^l: the hidden representation of node v at layer l

- W_k: weight matrix for neighborhood aggregation
- B_k: weight matrix for transforming hidden vector of self

Matrix Formulation (1)

- **Many aggregations can be performed efficiently by (sparse) matrix operations**
- Let $H^{(l)} = [h_1^{(l)} \dots h_{|V|}^{(l)}]^{\mathrm{T}}$
- Then: $\sum_{u \in N_v} h_u^{(l)} = \mathrm{A}_{v,:}\mathrm{H}^{(l)}$
- Let 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,**

Matrix of hidden embeddings H^{k-1}



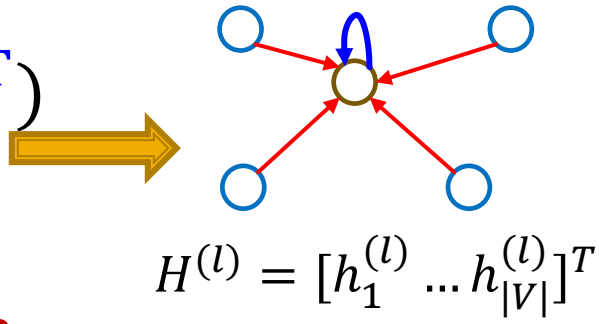
$$\sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} \Longrightarrow H^{(l+1)} = D^{-1}AH^{(l)}$$

Matrix Formulation (2)

- Re-writing update function in matrix form:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W_l^{\mathrm{T}} + H^{(l)}B_l^{\mathrm{T}})$$

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



  - Red: neighborhood aggregation
  - Blue: self transformation

- In practice, this implies that efficient sparse matrix multiplication can be used ($\tilde{A}$ is sparse)

- **Note**: not all GNNs can be expressed in matrix form, when aggregation function is complex

How to train a GNN

- Node embedding $\boldsymbol{z}_v$ is a function of input graph
- **Supervised setting**: we want to minimize the loss $\mathcal{L}$ (see also slide 15):

$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{z}_v))$$

  - $\boldsymbol{y}$: node label
  - $\mathcal{L}$ could be L2 if $\boldsymbol{y}$ is real number, or cross entropy if $\boldsymbol{y}$ is categorical
- **Unsupervised setting:**
  - No node label available
  - **Use the graph structure as the supervision!**

Unsupervised Training

- **"Similar" nodes have similar embeddings**

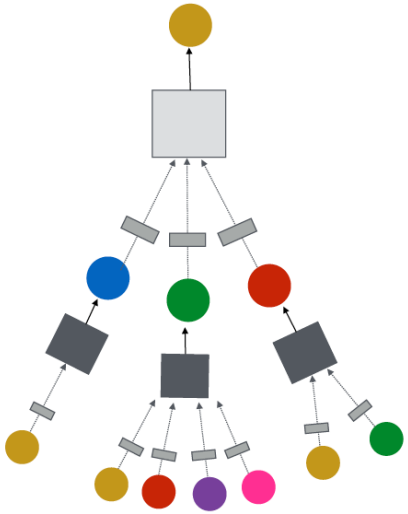
$$\mathcal{L} = \sum_{z_u, z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v))$$

  - Where $y_{u,v} = 1$ when node u and v are **similar**
  - CE is the cross entropy (slide 16)
  - DEC is the decoder such as inner product (lecture 4)
- **Node similarity** can be anything from lecture 3, e.g., a loss based on:
  - **Random walks** (node2vec, DeepWalk, struc2vec)
  - **Matrix factorization**
  - **Node proximity in the graph**

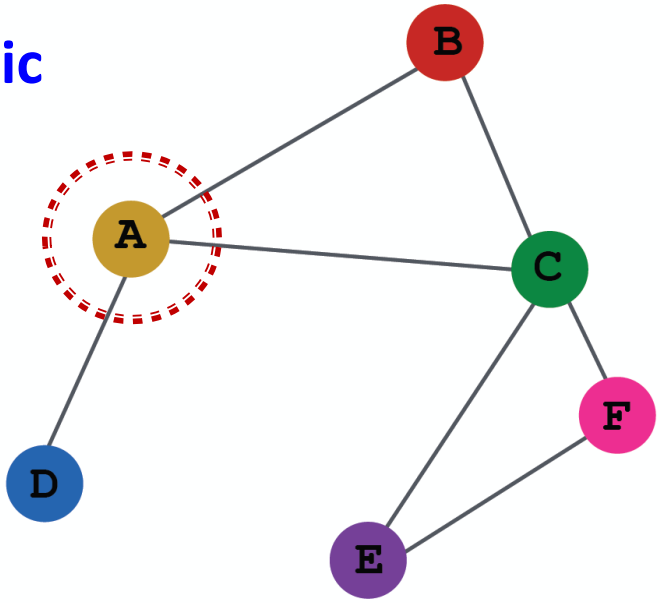
Supervised Training

Directly train the model for a supervised task (e.g., node classification)

Safe or toxic drug?



Safe or toxic drug?



E.g., a drug-drug interaction network

Supervised Training

Directly train the model for a supervised task (e.g., **node classification**)

- Use cross entropy loss (slide 16)

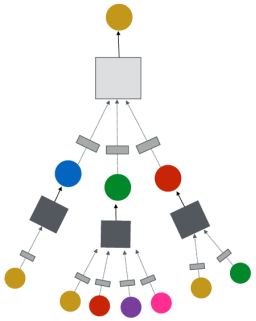
$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

Encoder output: node embedding

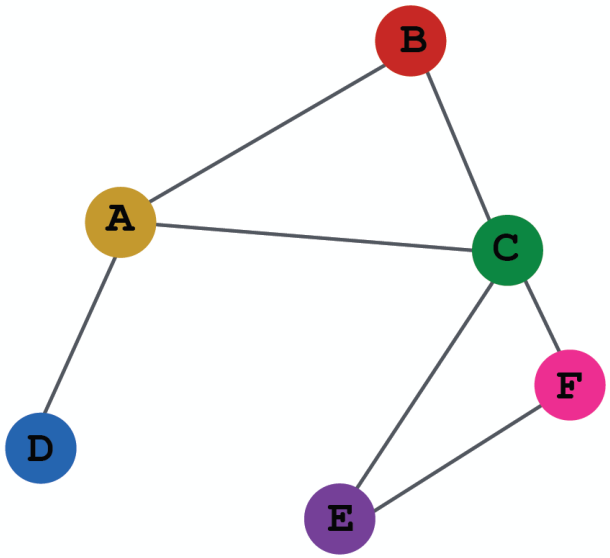
Classification weights

Node class label

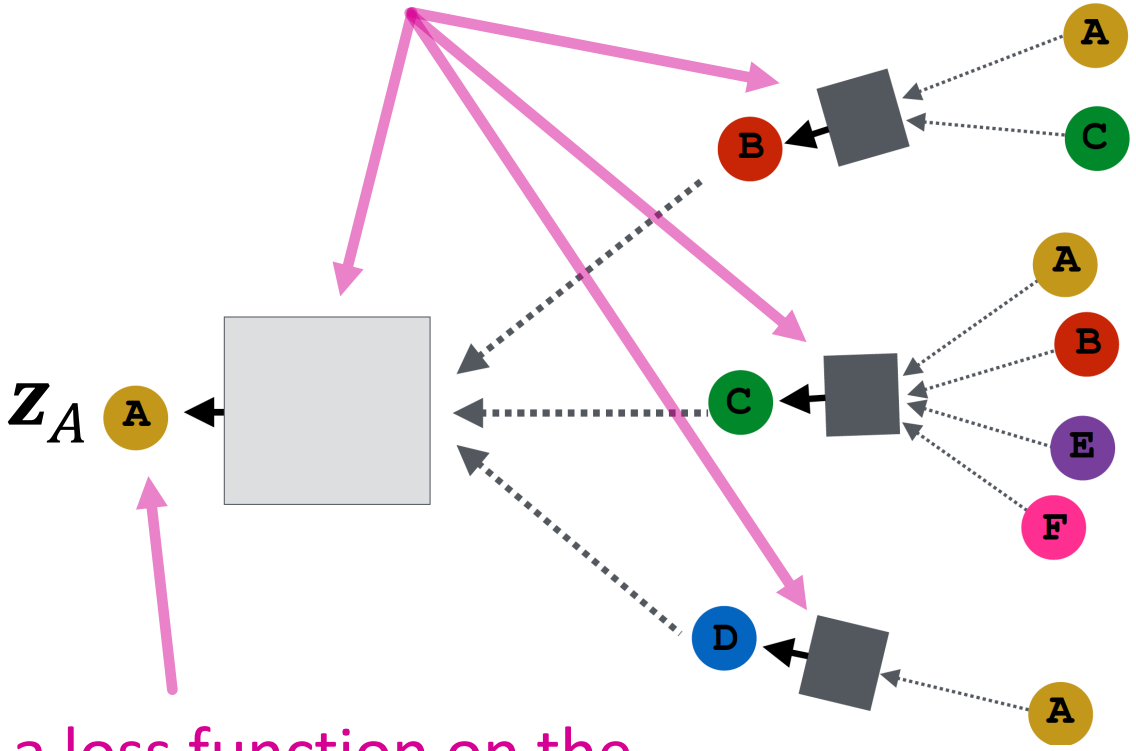
Safe or toxic drug?



Model Design: Overview

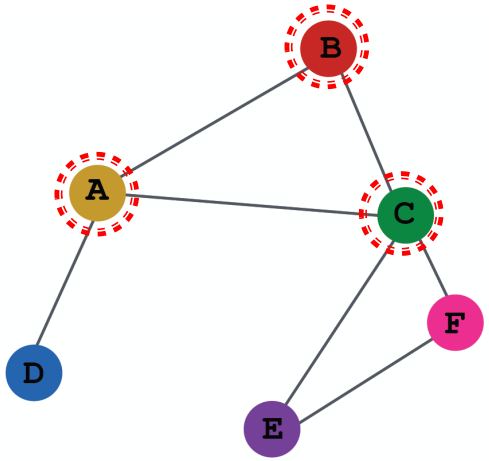


(1) Define a neighborhood aggregation function



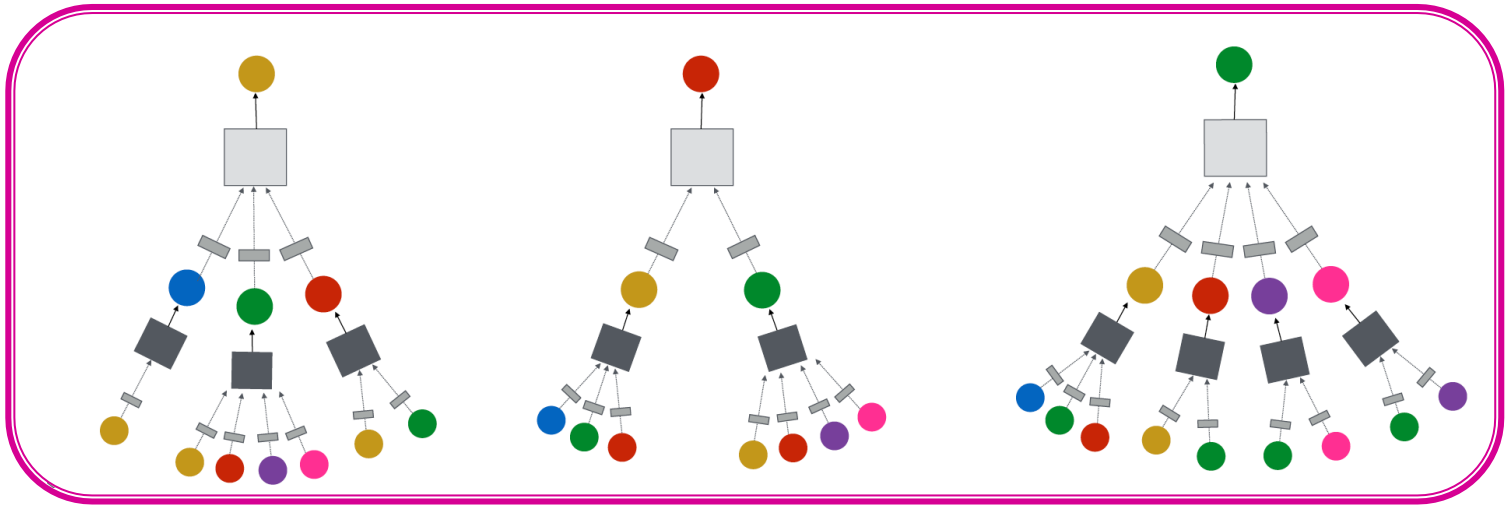
(2) Define a loss function on the embeddings

Model Design: Overview

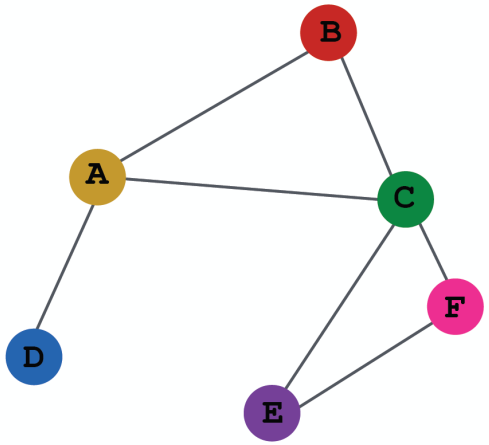


INPUT GRAPH

(3) Train on a set of nodes, i.e., a batch of compute graphs



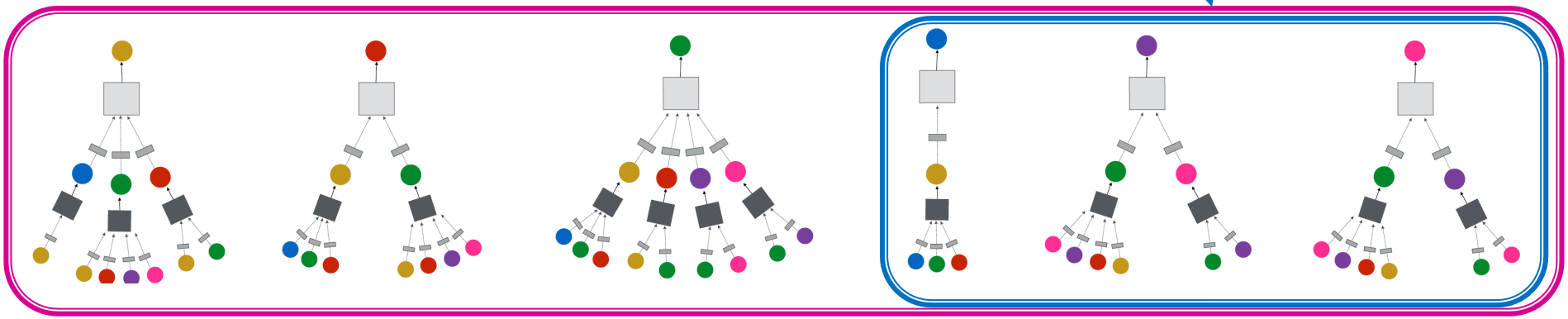
Model Design: Overview



INPUT GRAPH

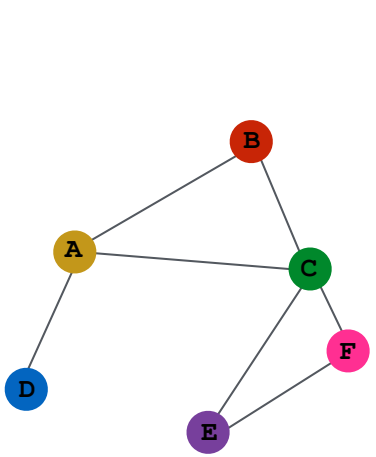
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

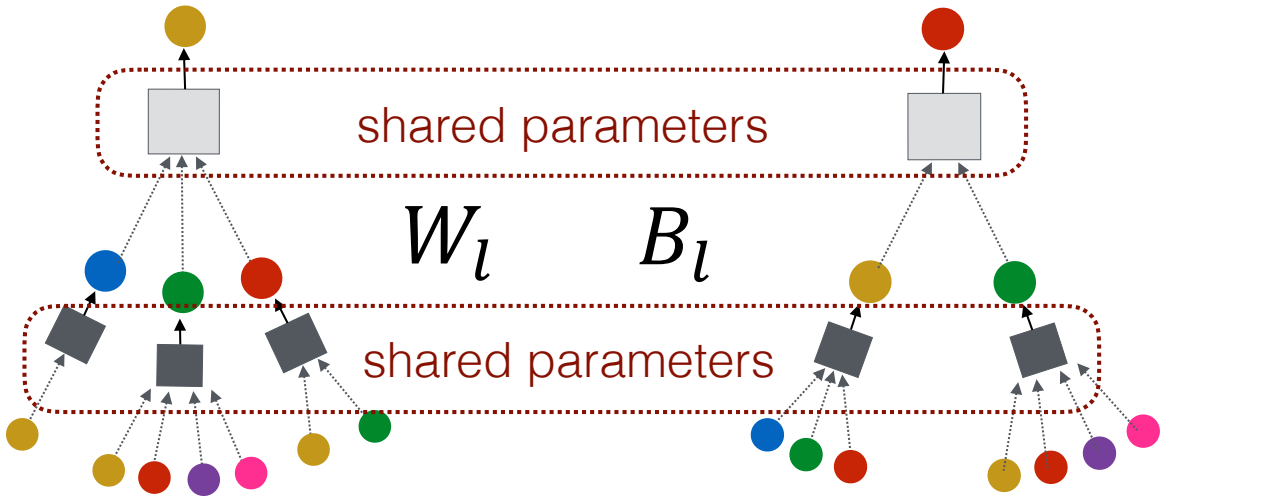


Inductive Capability

- **The same aggregation parameters are shared for all nodes:**
  - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes**!



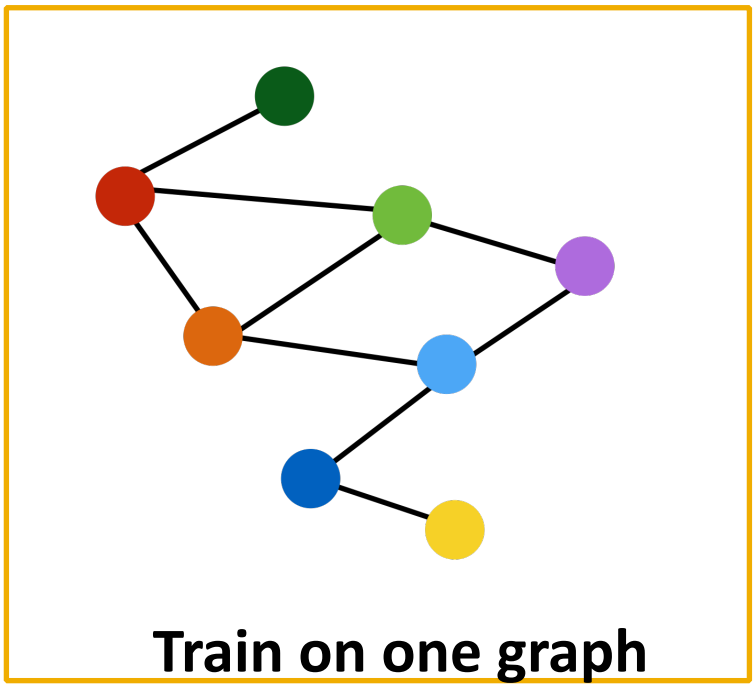
INPUT GRAPH



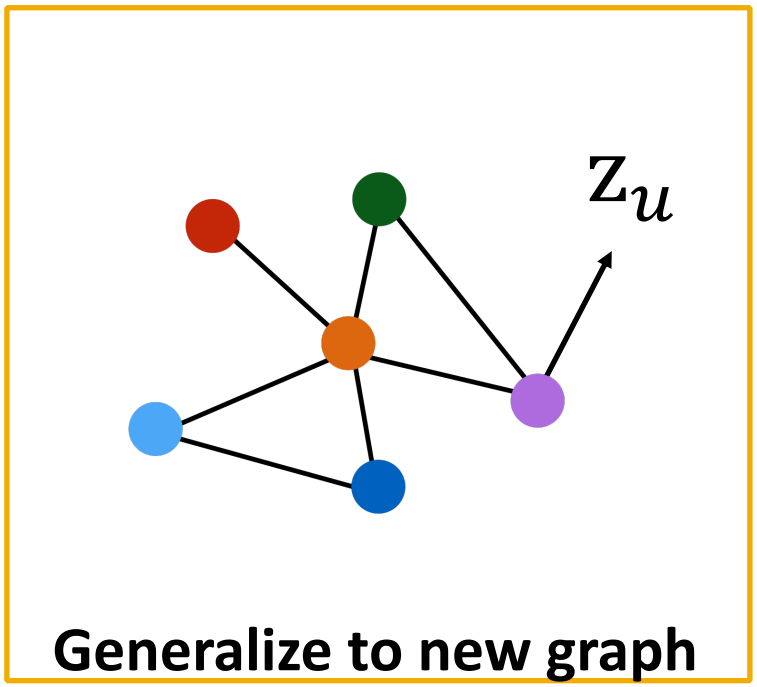
Compute graph for node A

Compute graph for node B

Inductive Capability: New Graphs



Train on one graph

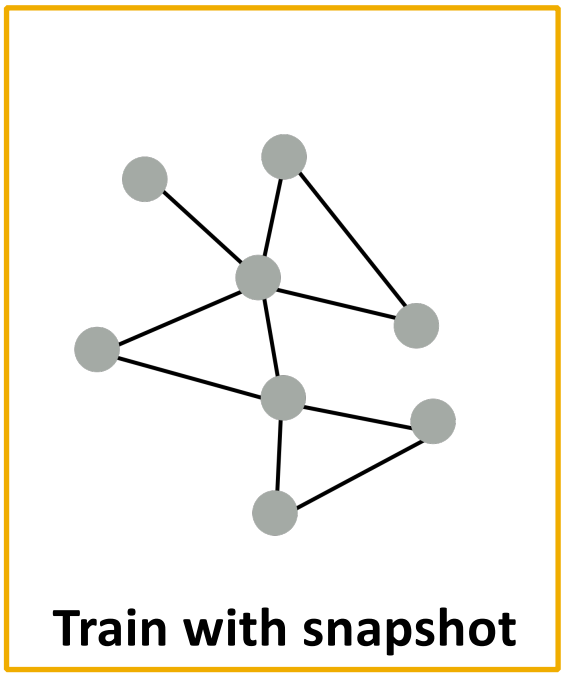


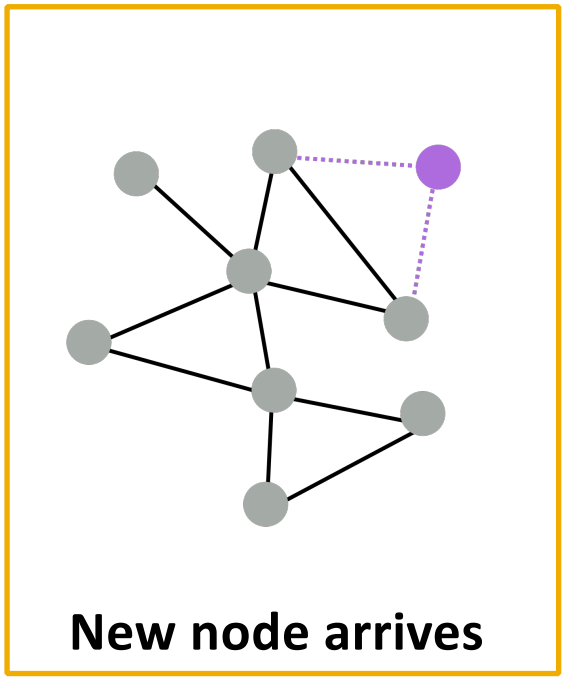
Generalize to new graph

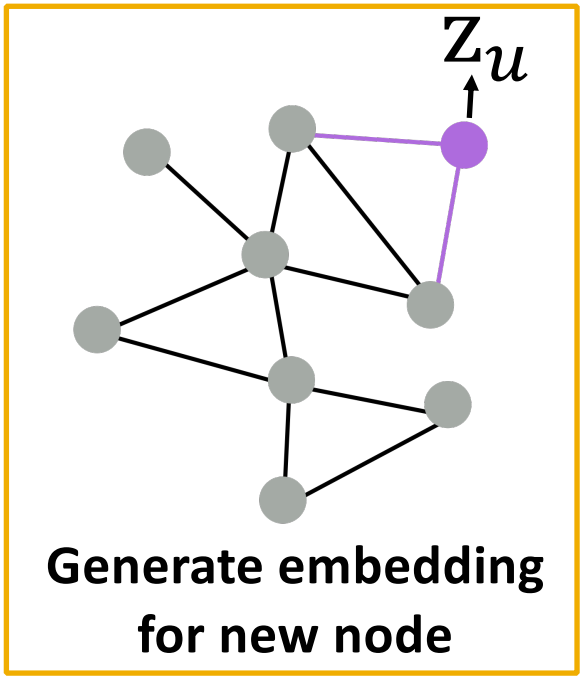
Inductive node embedding → Generalize to entirely unseen graphs

E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

Inductive Capability: New Nodes







- Many application settings constantly encounter previously unseen nodes:
  - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings "on the fly"

Summary

- **Recap:** Generate node embeddings by aggregating neighborhood information
  - We saw a **basic variant of this idea**
  - Key distinctions are in how different approaches aggregate information across the layers

- **Next:** Describe GraphSAGE graph neural network architecture

Outline of Today's Lecture

1. **Basics of deep learning** ✓
2. **Deep learning for graphs** ✓
3. **Graph Convolutional Networks and GraphSAGE**

Stanford CS224W: Graph Convolutional Networks and GraphSAGE

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

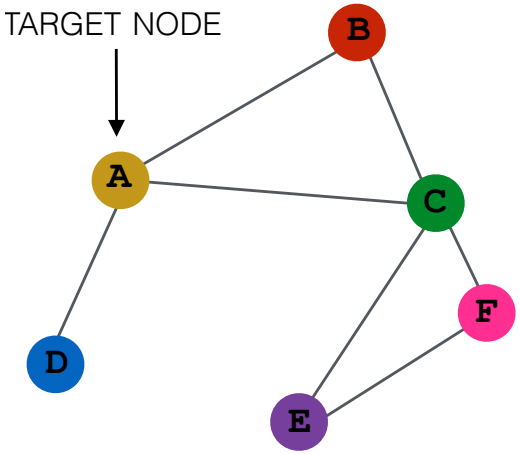
http://cs224w.stanford.edu

GraphSAGE Idea

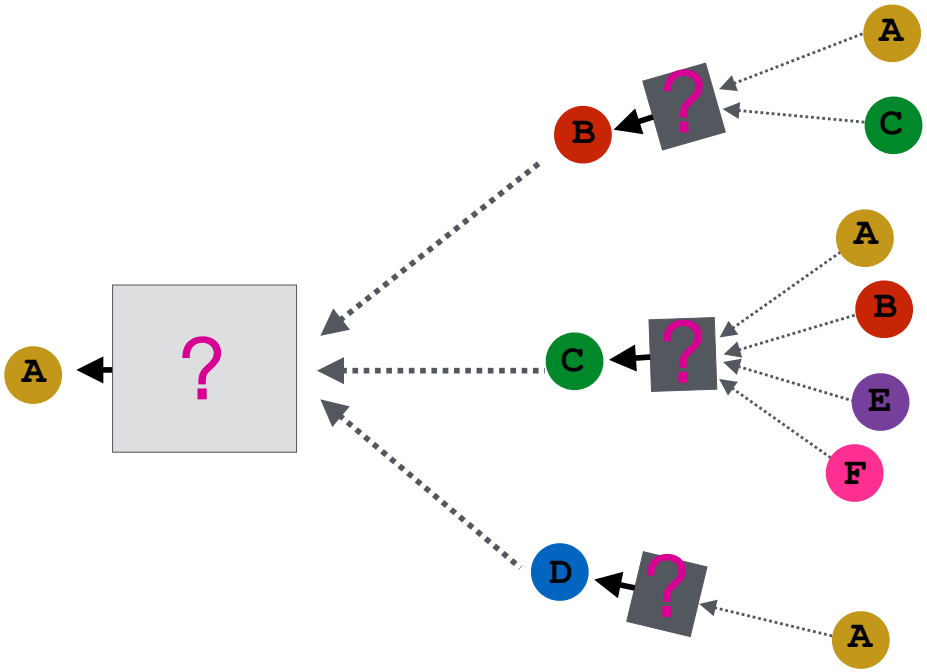
[Hamilton et al., NIPS 2017]

So far we have aggregated the neighbor messages by taking their (weighted) average

Can we do better?

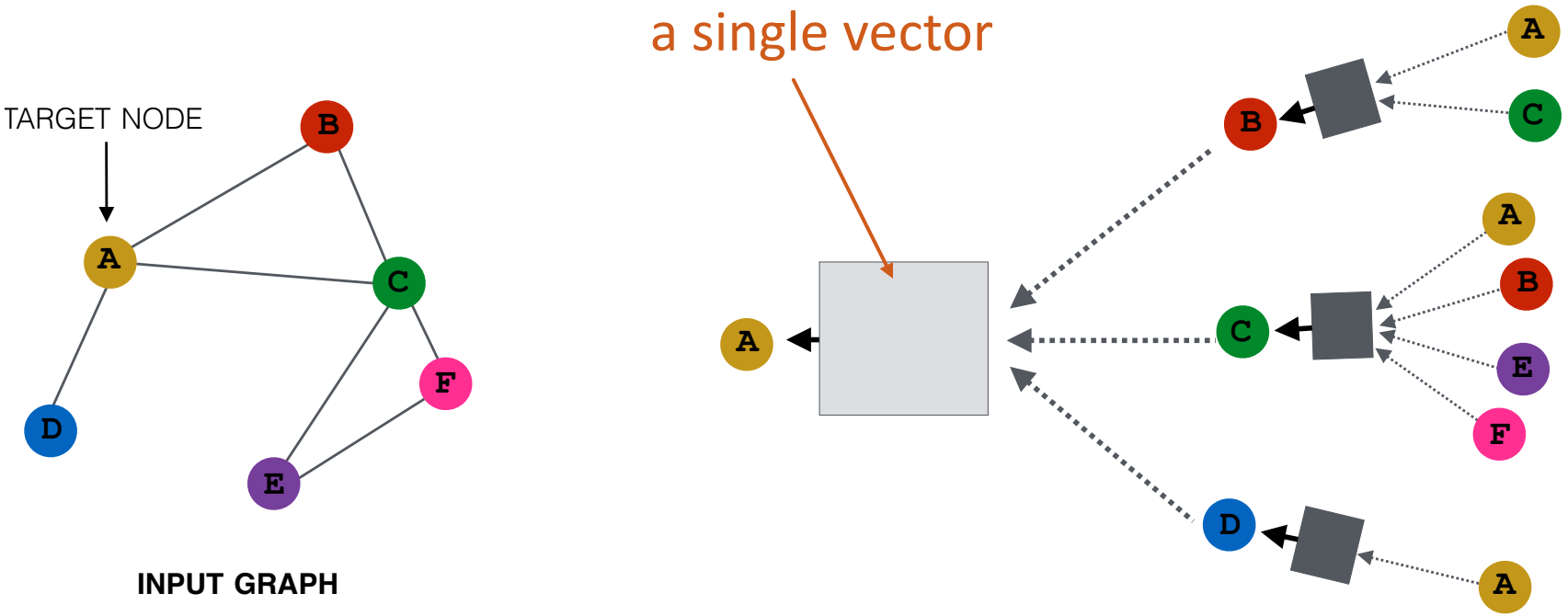


INPUT GRAPH



GraphSAGE Idea (1)

Any differentiable function that maps set of vectors in $N(u)$ to a single vector



$$\mathrm{h}_v^{(l+1)} = \sigma([\mathrm{W}_l \cdot \mathrm{AGG}\left(\left\{\mathrm{h}_u^{(l)}, \forall u \in N(v)\right\}\right), \mathrm{B}_l \mathrm{h}_v^{(l)}])$$

How does this message passing architecture differ?

GraphSAGE Idea (2)

$$\mathrm{h}_v^{(l+1)} = \sigma([\mathrm{W}_l \cdot \mathrm{AGG}\left(\left\{\mathrm{h}_u^{(l)}, \forall u \in N(v)\right\}\right), \mathrm{B}_l \mathrm{h}_v^{(l)}])$$

Optional: Apply L2 normalization to $\mathrm{h}_v^{(l+1)}$ embedding at every layer

- ℓ_2 **Normalization:**
  - $h_v^k \leftarrow \frac{h_v^k}{\|h_v^k\|_2}$ $\forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2-norm)
  - Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement
  - After ℓ_2 normalization, all vectors will have the same ℓ_2-norm

Neighborhood Aggregation

- **Simple neighborhood aggregation:**

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

- **GraphSAGE:**

Concatenate neighbor embedding and self embedding

$$\mathrm{h}_v^{(l+1)} = \sigma\left([\mathrm{W}_l \cdot \mathrm{AGG}\left(\left\{\mathrm{h}_u^{(l)}, \forall u \in N(v)\right\}\right), \mathrm{B}_l \mathrm{h}_v^{(l)}]\right)$$

Flexible aggregation function instead of mean

Neighbor Aggregation: Variants

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

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

- **Pool:** Transform neighbor vectors and apply symmetric vector function

Element-wise mean/max

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

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

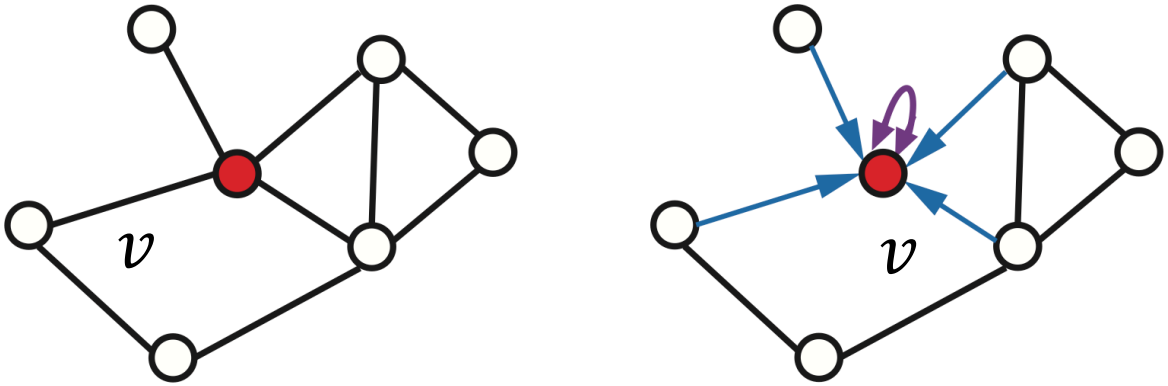
$$\text{AGG} = \text{LSTM}([\text{h}_u^{(l)}, \forall u \in \pi(N(v))])$$

Recap: GCN, GraphSAGE

Key idea: Generate node embeddings based on local neighborhoods

- Nodes aggregate "messages" from their neighbors using neural networks

- **Graph convolutional networks:**
  - **Basic variant:** Average neighborhood information and stack neural networks
- **GraphSAGE:**
  - Generalized neighborhood aggregation



Summary

- **In this lecture, we introduced**
  - Basics of neural networks
    - Loss, Optimization, Gradient, SGD, non-linearity, MLP
  - Idea for Deep Learning for Graphs
    - Multiple layers of embedding transformation
    - At every layer, use the embedding at previous layer as the input
    - Aggregation of neighbors and self embeddings
  - Graph Convolutional Network
    - Mean aggregation; can be expressed in matrix form
  - GraphSAGE: more flexible aggregation

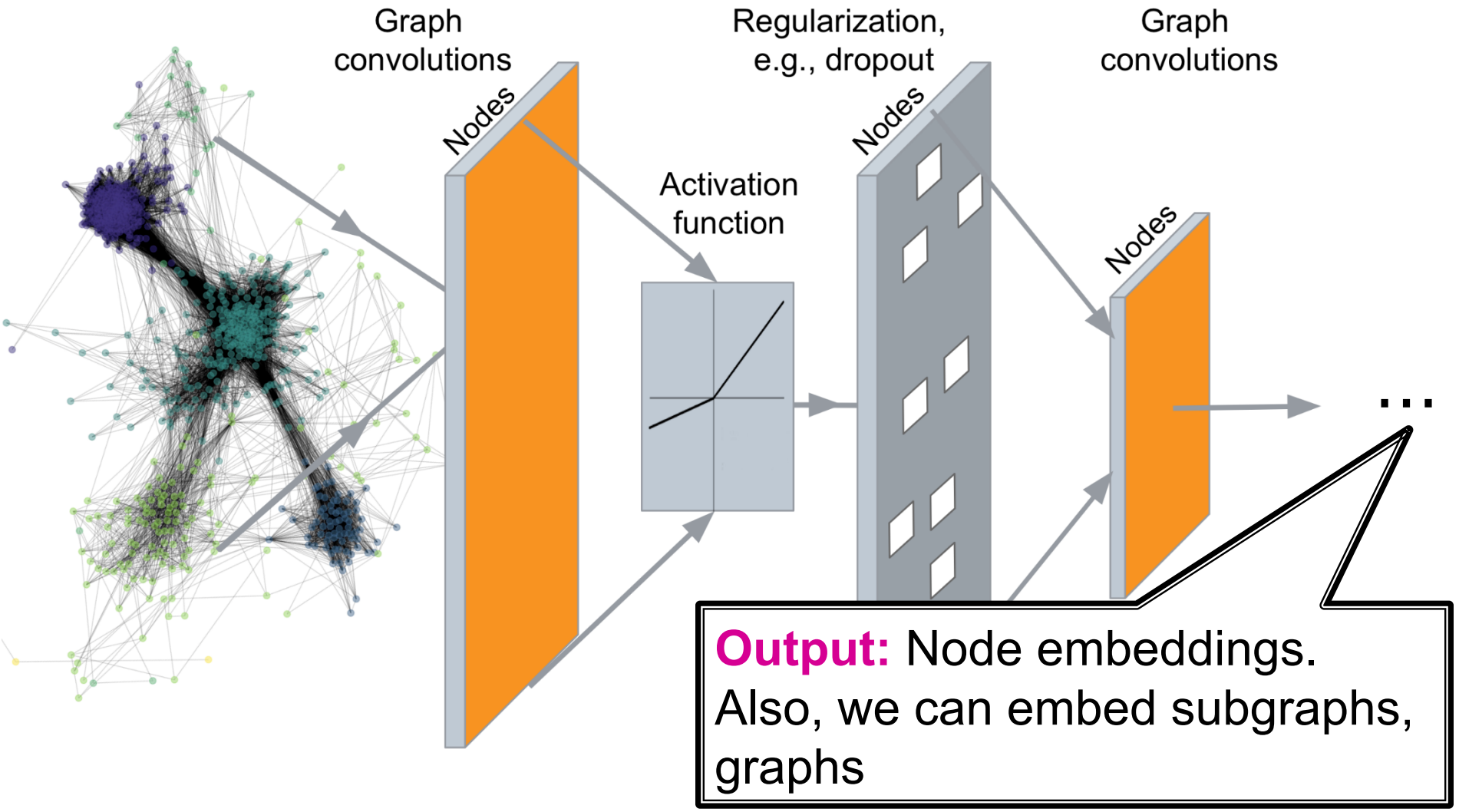
Stanford CS224W: A General Perspective on Graph Neural Networks

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

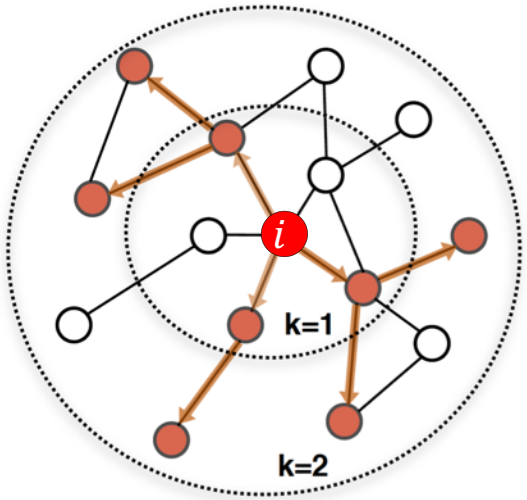
Recap: Deep Graph Encoders



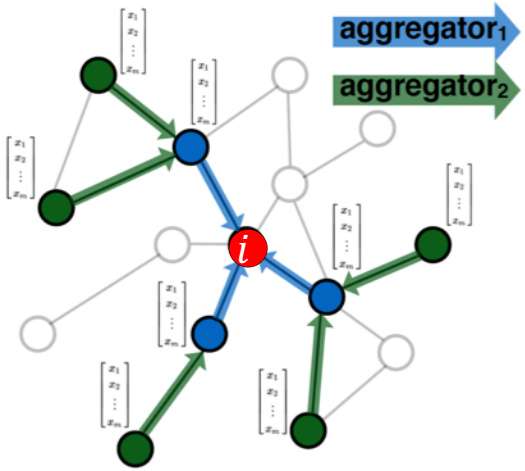
Output: Node embeddings.
Also, we can embed subgraphs, graphs

Recap: Graph Neural Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

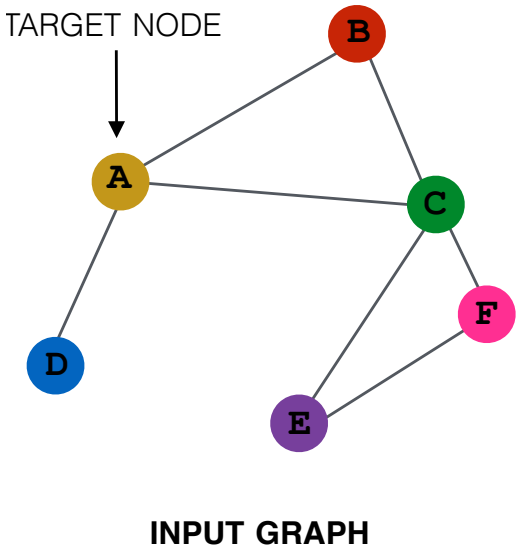


Propagate and transform information

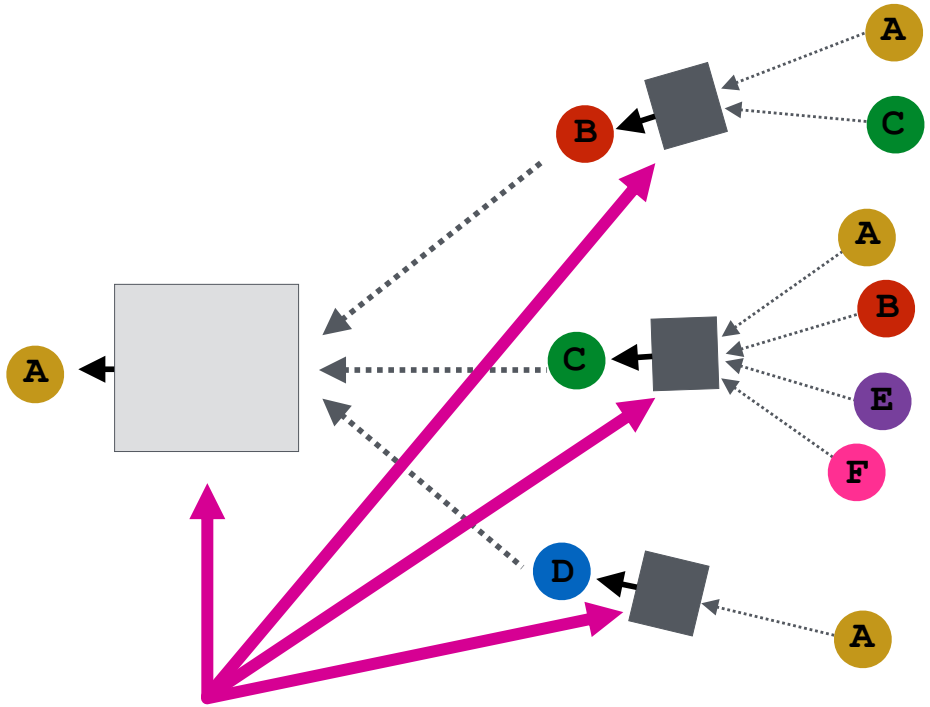
Learn how to propagate information across the graph to compute node features

Recap: Aggregate from Neighbors

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



INPUT GRAPH

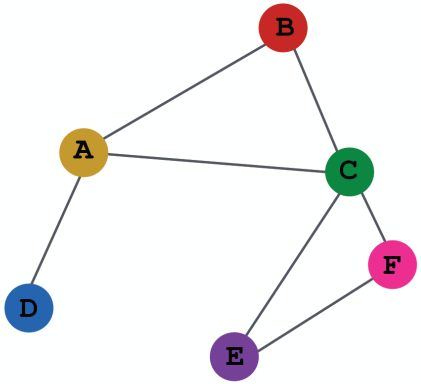


Neural networks

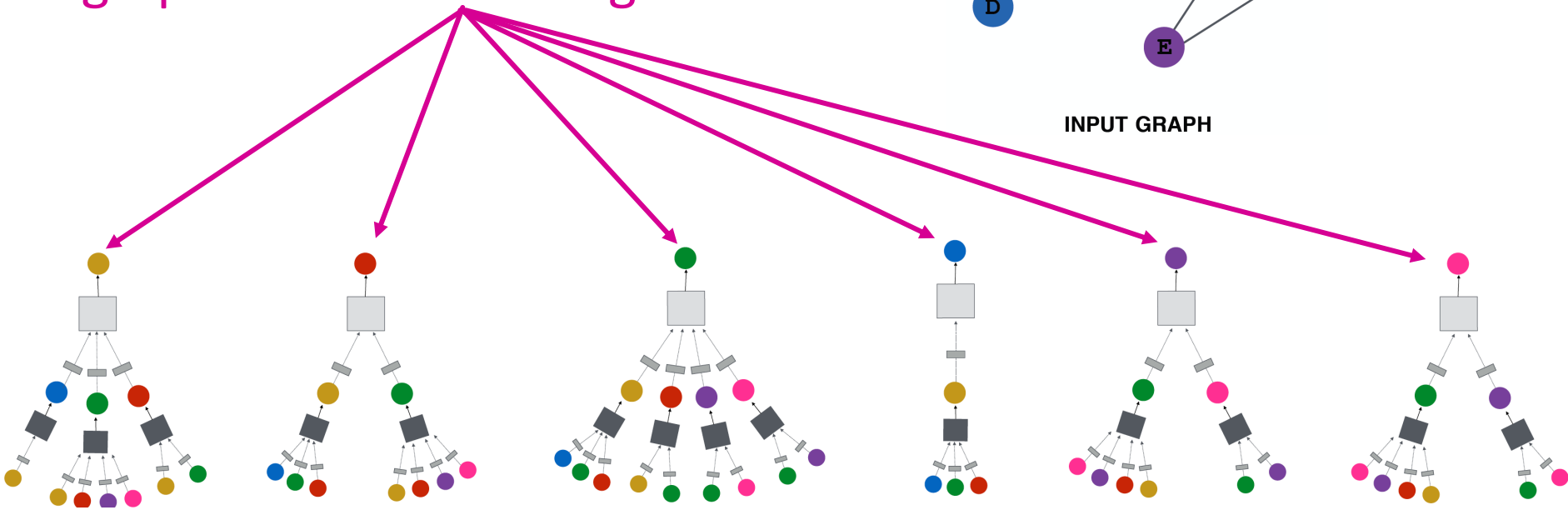
Recap: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!



INPUT GRAPH



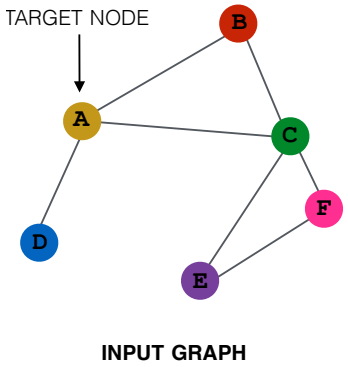
Stanford CS224W: A General Perspective on Graph Neural Networks

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

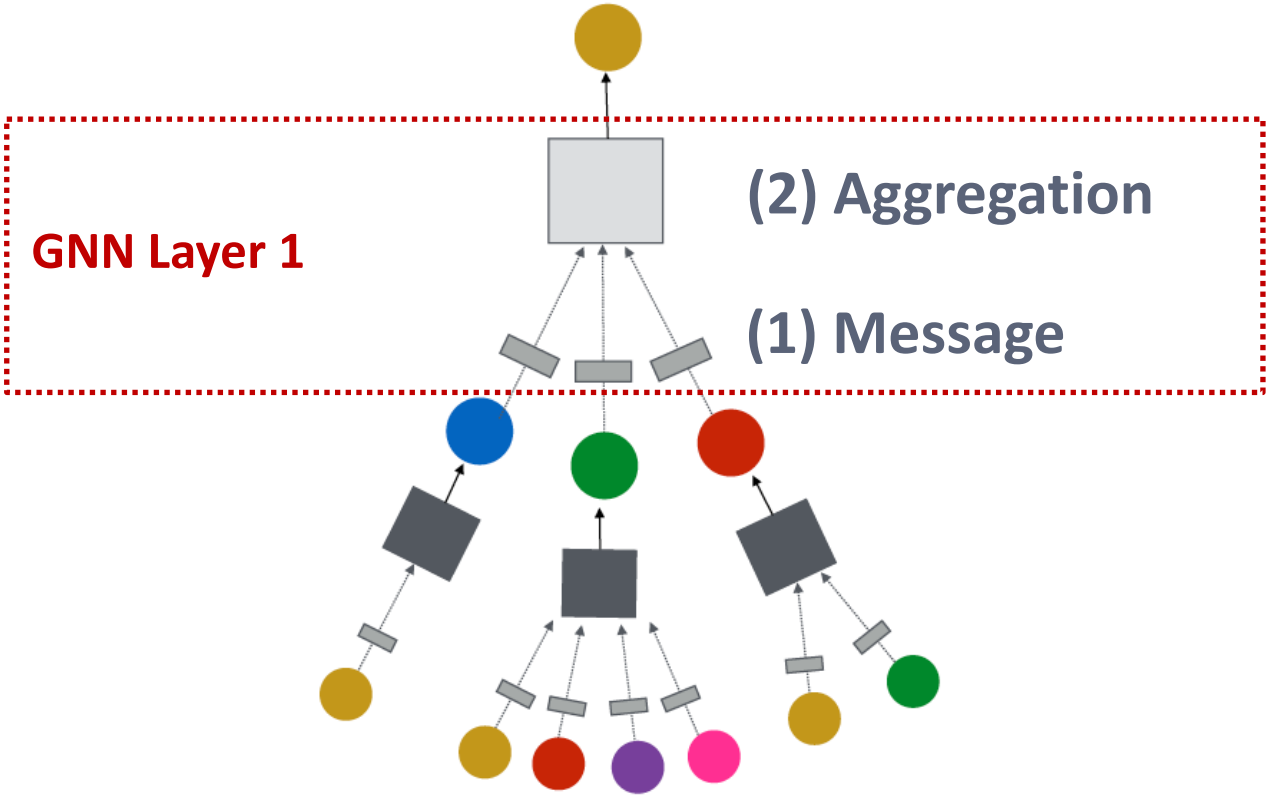
A General GNN Framework (1)



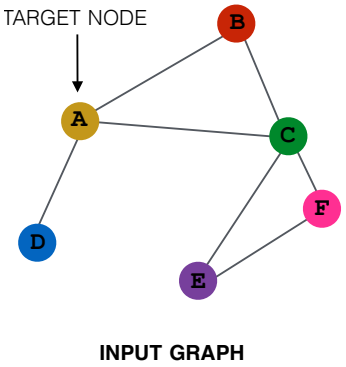
INPUT GRAPH

GNN Layer = Message + Aggregation

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



A General GNN Framework (2)

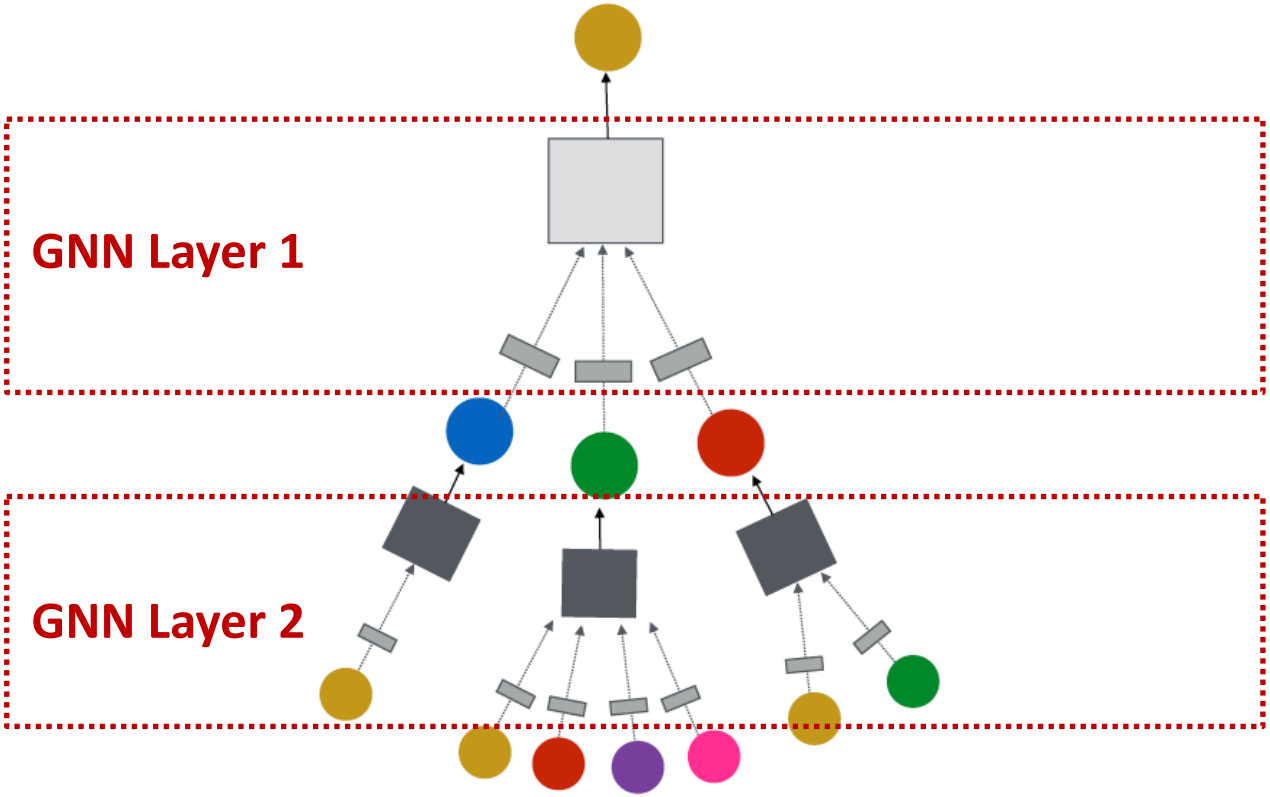


INPUT GRAPH

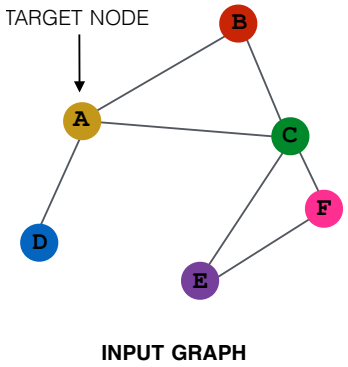
Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity



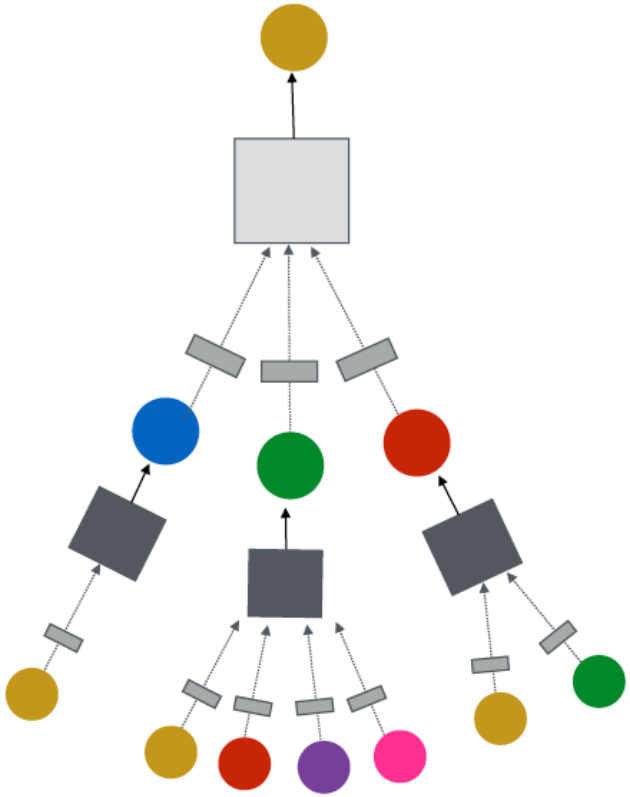
A General GNN Framework (3)



INPUT GRAPH

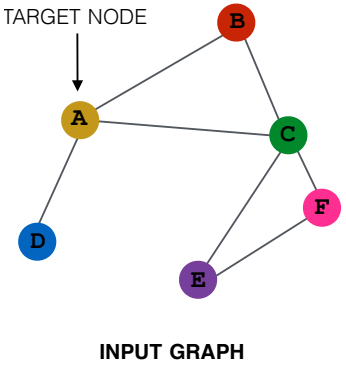
Idea: Raw input graph ≠ computational graph

- **Graph feature augmentation**
- **Graph structure augmentation**



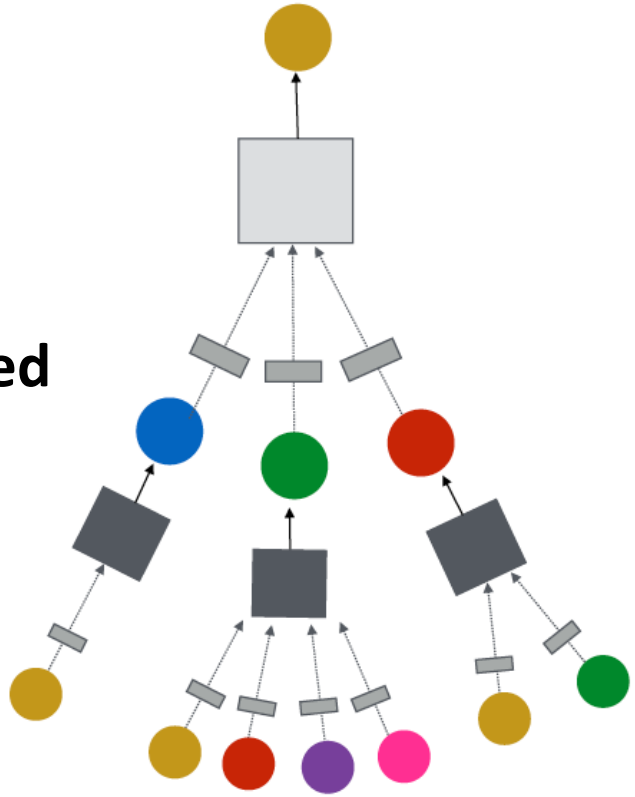
(4) Graph augmentation

A General GNN Framework (4)



INPUT GRAPH

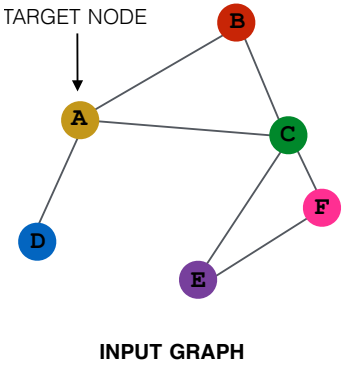
(5) Learning objective



How do we train a GNN

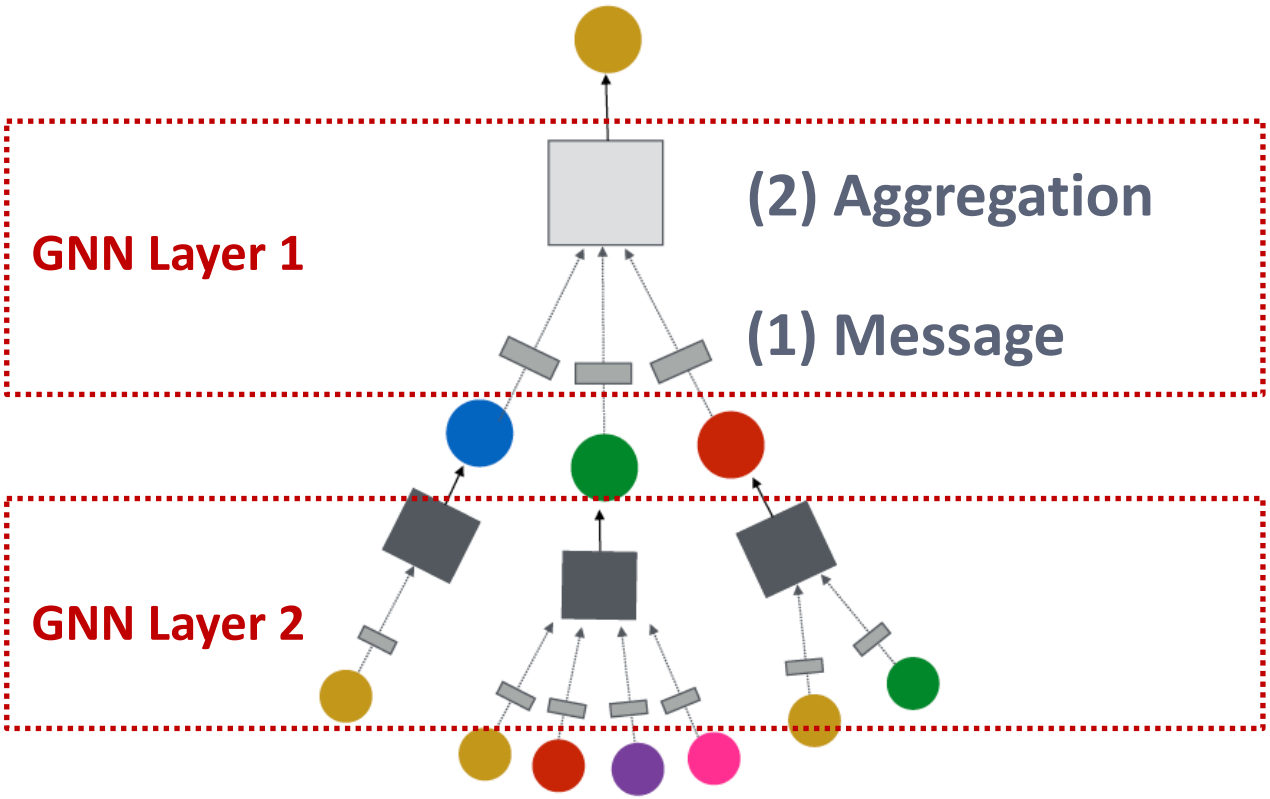
- **Supervised/Unsupervised objectives**
- **Node/Edge/Graph level objectives**

A General GNN Framework (5)



INPUT GRAPH

(5) Learning objective



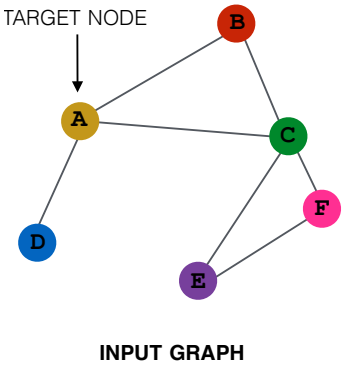
(3) Layer connectivity

(4) Graph augmentation

Stanford CS224W: A Single Layer of a GNN

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

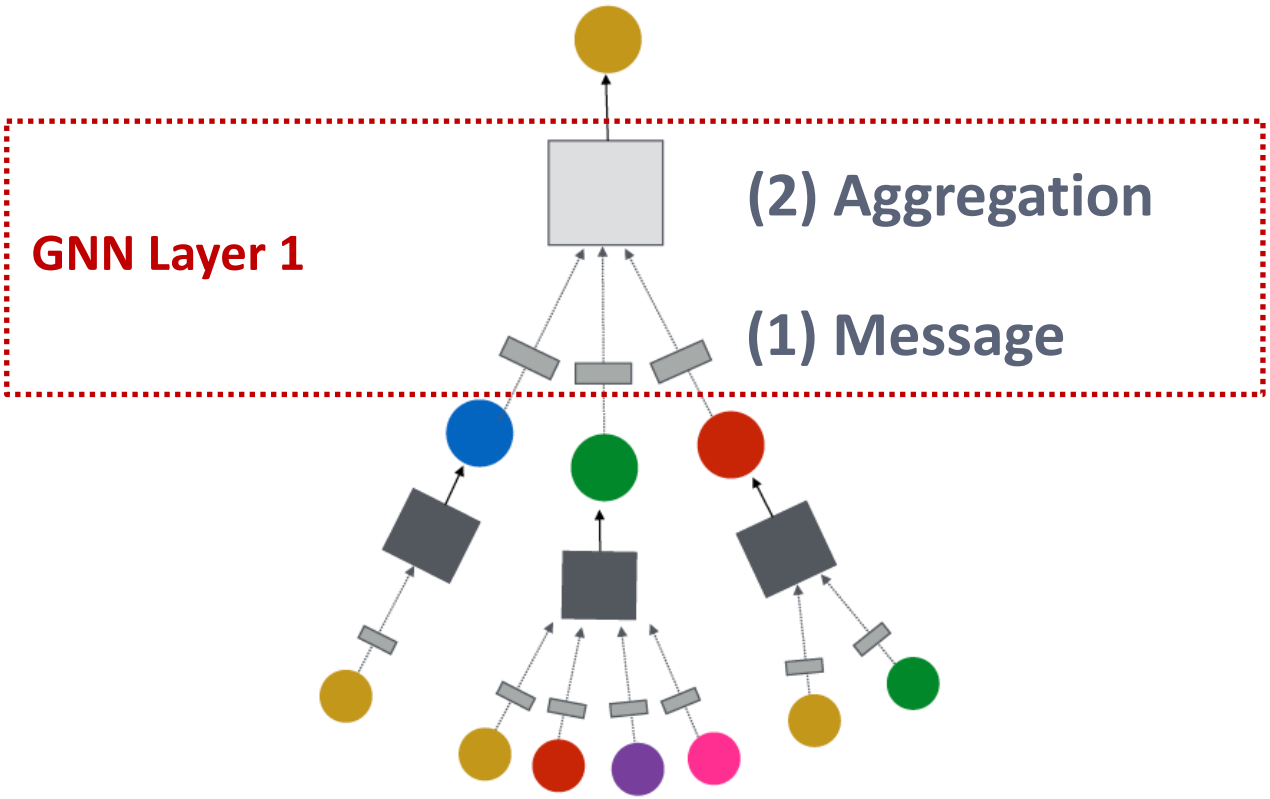
A GNN Layer



INPUT GRAPH

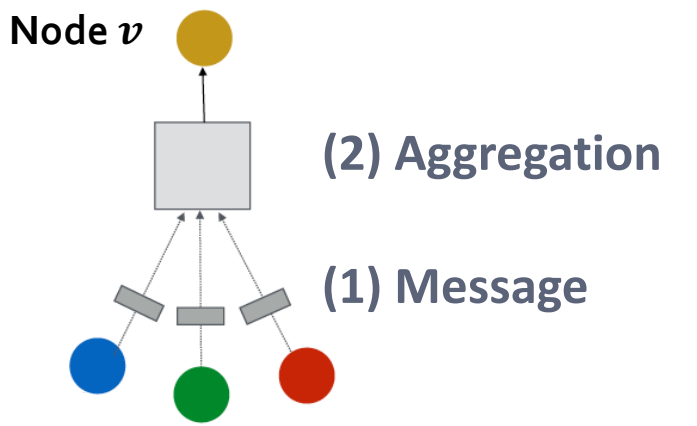
GNN Layer = Message + Aggregation

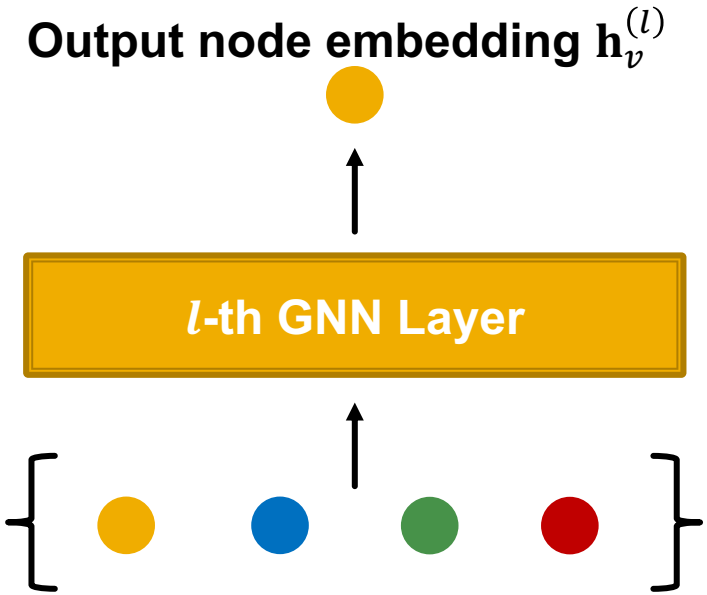
- **Different instantiations under this perspective**
- **GCN, GraphSAGE, GAT, …**



A Single GNN Layer

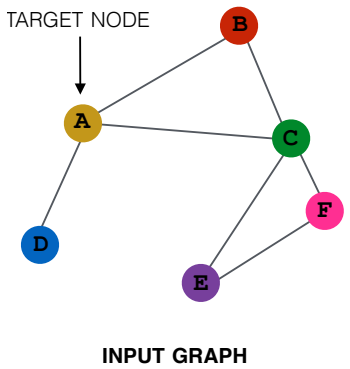
- **Idea of a GNN Layer:**
  - Compress a set of vectors into a single vector
  - **Two step process:**
    - **(1) Message**
    - **(2) Aggregation**

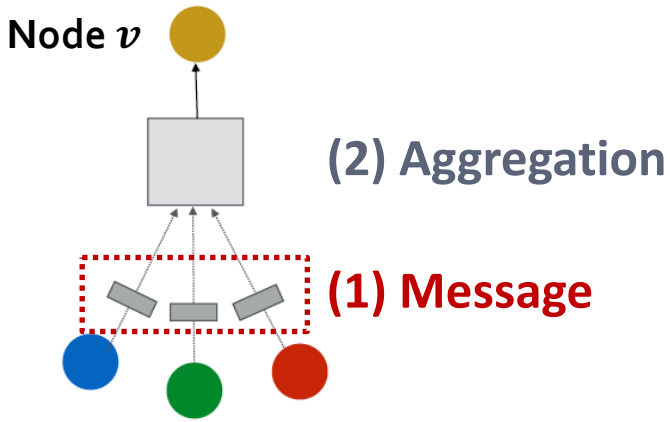




Message Computation

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



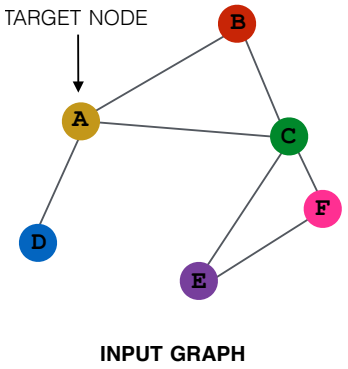


Message Aggregation

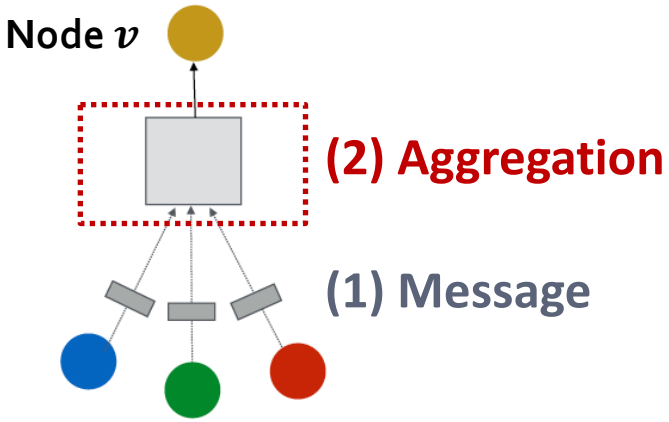
- **(2) Aggregation**
  - **Intuition:** Each node will aggregate the messages from node v's neighbors

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

  - **Example:** $\mathrm{Sum}(\cdot)$, $\mathrm{Mean}(\cdot)$ or $\mathrm{Max}(\cdot)$ aggregator
    - $\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$



INPUT GRAPH



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**

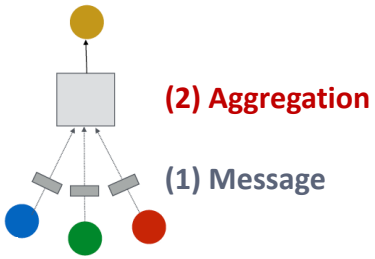
Then aggregate from node itself

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

First aggregate from neighbors

A Single GNN Layer

- **Putting things together:**
  - **(1) Message**: each node computes a message
$$\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$
  - **(2) Aggregation**: aggregate messages from neighbors
$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}, \mathbf{m}_v^{(l)}\right)$$
  - **Nonlinearity (activation):** Adds expressiveness
    - Often written as $\sigma(\cdot)$: $\mathrm{ReLU}(\cdot)$, $\mathrm{Sigmoid}(\cdot)$, ...
    - Can be added to **message or aggregation**



Classical GNN Layers: GCN (1)

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

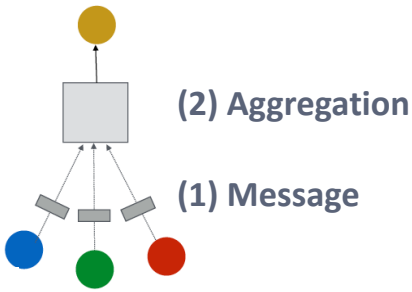
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

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

Message

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

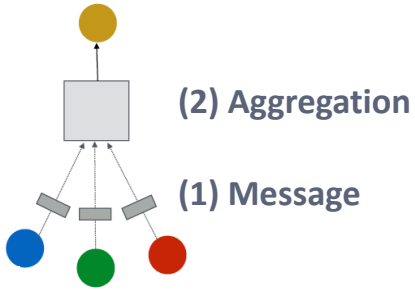
Aggregation



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

Classical GNN Layers: GraphSAGE

- **(2) GraphSAGE**

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

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

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

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

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

GraphSAGE Neighbor Aggregation

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

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

Aggregation — Message computation

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

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

Aggregation — Message 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
  - $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\left\|\mathbf{h}_v^{(l)}\right\|_2} \; \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2-norm)
  - Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement
  - 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
  - $\Longrightarrow \alpha_{vu}$ is defined **explicitly** based on the structural properties of the graph (node degree)
  - $\Longrightarrow$ All neighbors $u \in N(v)$ are equally important to node v

Classical GNN Layers: GAT (2)

- **(3) Graph Attention Networks**

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

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.

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

Graph Attention Networks

Can we do better than simple neighborhood aggregation?

Can we let weighting factors α_{vu} to be learned?

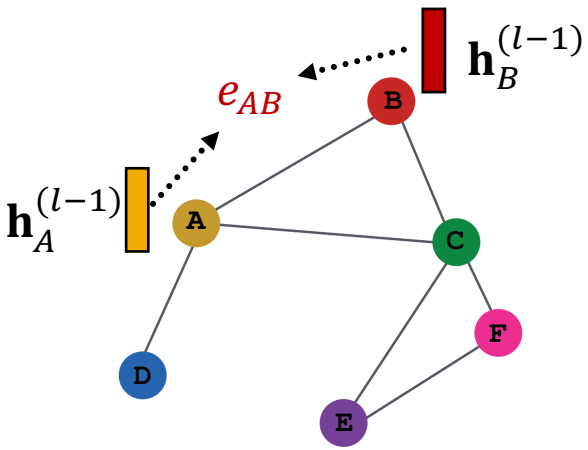
- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding $\boldsymbol{h}_v^{(l)}$ of each node in the graph following an **attention strategy:**
  - Nodes attend over their neighborhoods' message
  - Implicitly specifying different weights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

$$e_{vu} = a(\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)}\boldsymbol{h}_v^{(l-1)})$$

    - **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 (2)

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

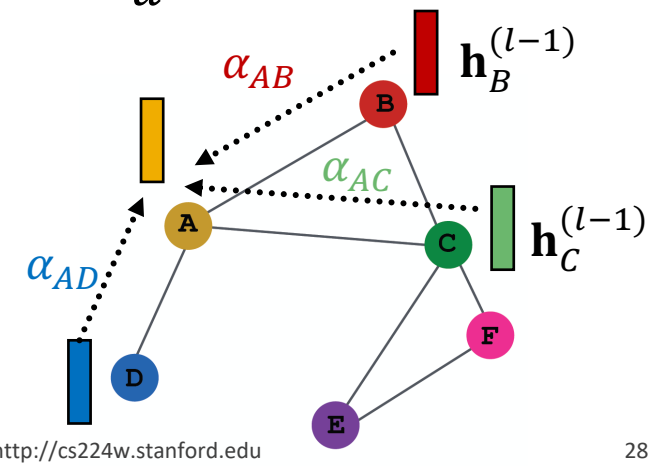
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

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

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

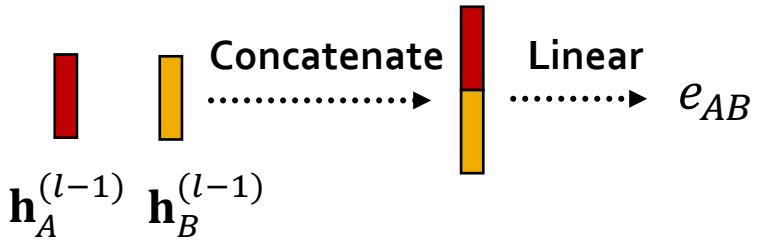
Weighted sum using α_{AB}, α_{AC}, α_{AD}:

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



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)



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

  - 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):

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

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

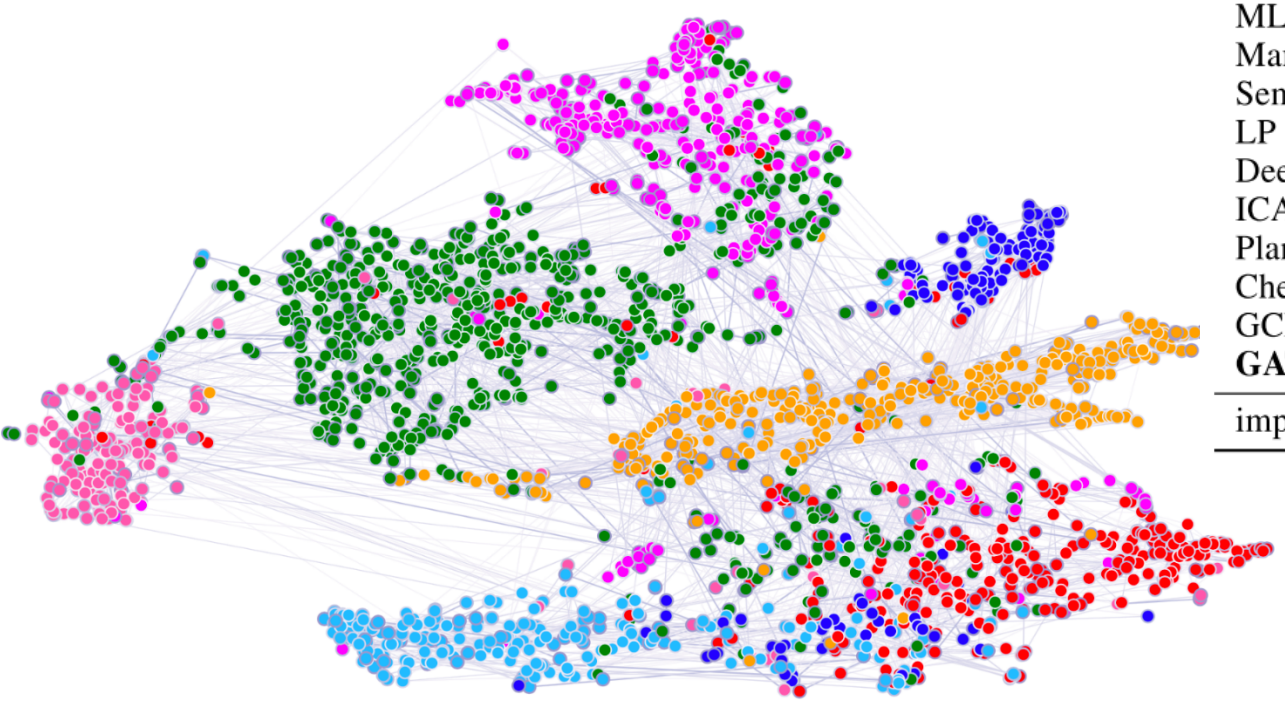
$$\mathbf{h}_v^{(l)}[3] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^{3} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

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

Benefits of Attention Mechanism

- **Key benefit:** Allows for (implicitly) specifying **different importance values (α_{vu}) to different neighbors**
- **Computationally efficient**:
  - Computation of attentional coefficients can be parallelized across all edges of the graph
  - Aggregation may be parallelized across all nodes
- **Storage efficient**:
  - Sparse matrix operations do not require more than $O(V + E)$ entries to be stored
  - **Fixed** number of parameters, irrespective of graph size
- **Localized**:
  - Only **attends over local network neighborhoods**
- **Inductive capability**:
  - It is a shared *edge-wise* mechanism
  - It does not depend on the global graph structure

GAT Example: Cora Citation Net



Method	Cora
MLP	55.1%
ManiReg (Belkin et al., 2006)	59.5%
SemiEmb (Weston et al., 2012)	59.0%
LP (Zhu et al., 2003)	68.0%
DeepWalk (Perozzi et al., 2014)	67.2%
ICA (Lu & Getoor, 2003)	75.1%
Planetoid (Yang et al., 2016)	75.7%
Chebyshev (Defferrard et al., 2016)	81.2%
GCN (Kipf & Welling, 2017)	81.5%
GAT	**83.3%**
improvement w.r.t GCN	1.8%

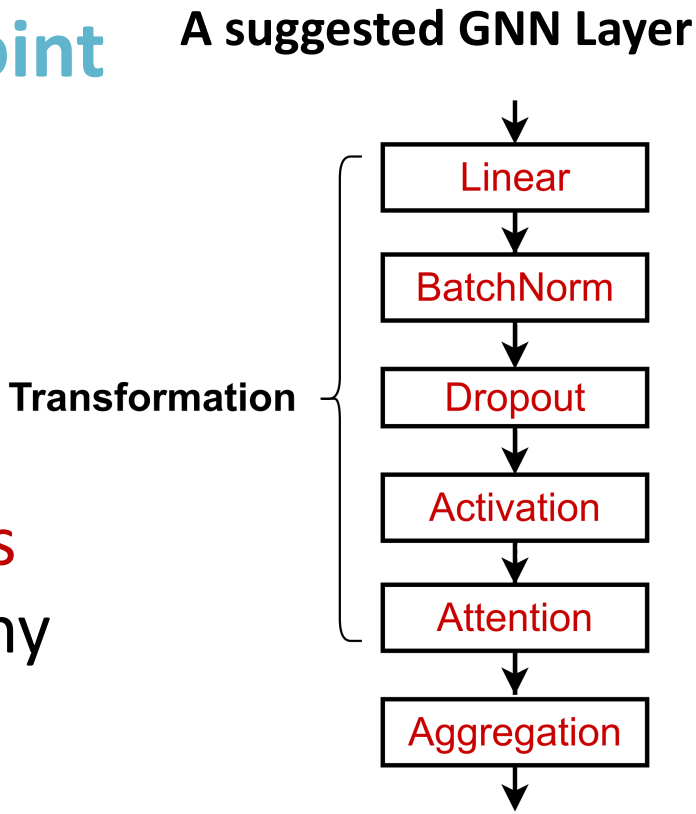
Attention mechanism can be used with many different graph neural network models

In many cases, attention leads to performance gains

- **t-SNE plot of GAT-based node embeddings:**
  - Node color: 7 publication classes
  - Edge thickness: Normalized attention coefficients between nodes i and j, across eight attention heads, $\sum_k (\alpha_{ij}^k + \alpha_{ji}^k)$

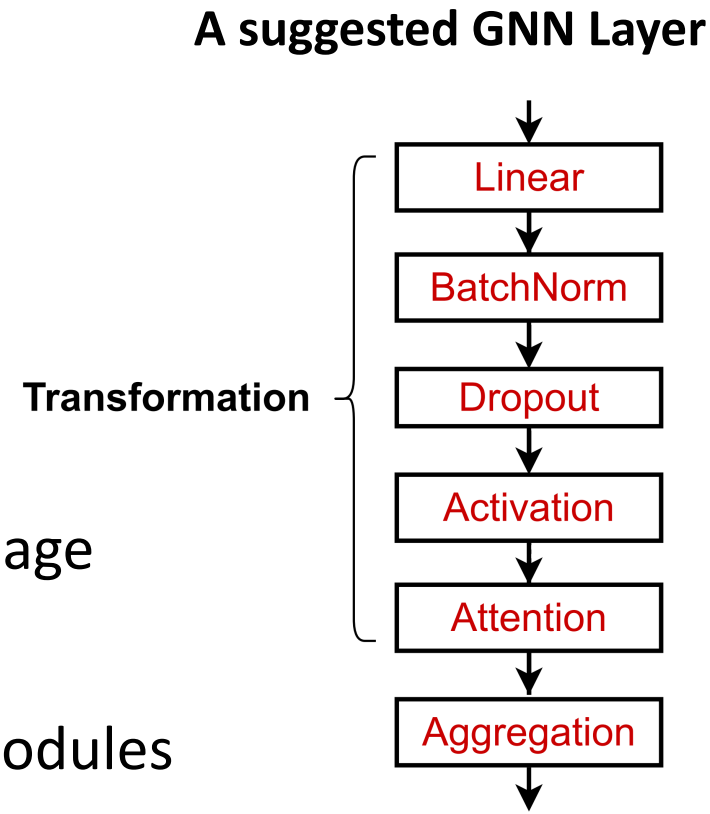
GNN Layer in Practice

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

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

Step 1:
Compute the mean and variance over N embeddings

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

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

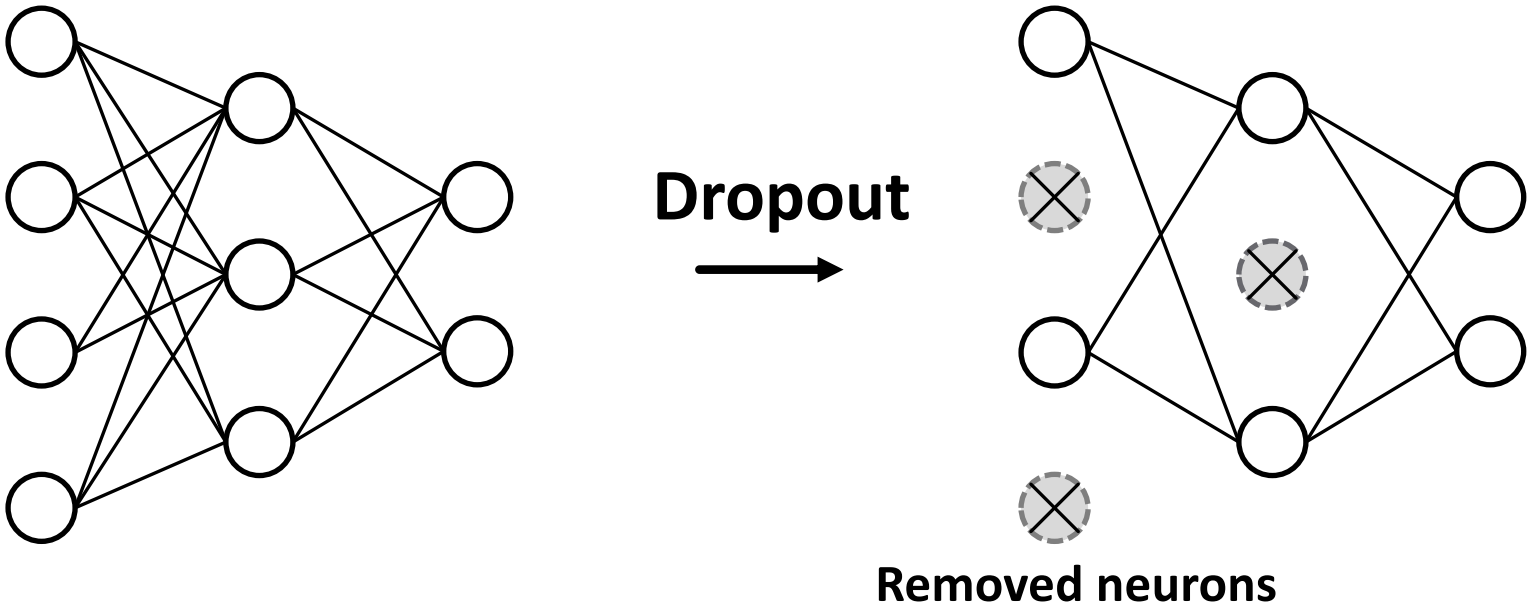
Step 2:
Normalize the feature using computed mean and variance

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

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

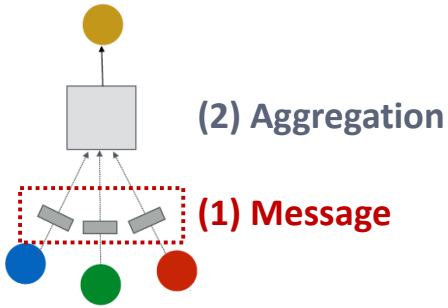
Dropout

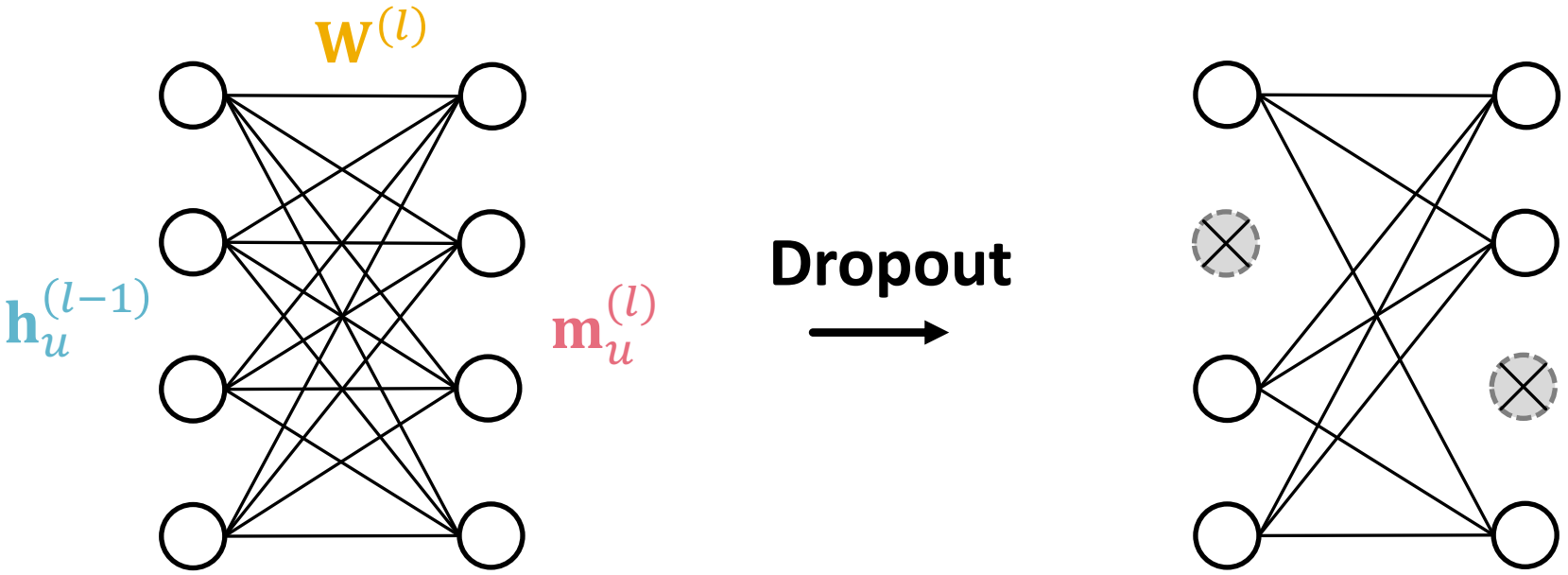
- **Goal**: Regularize a neural net to prevent overfitting.
- **Idea**:
  - **During training**: with some probability p, randomly set neurons to zero (turn off)
  - **During testing:** Use all the neurons for computation



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

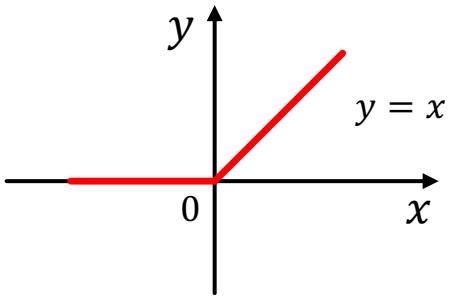
Activation (Non-linearity)

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

- **Rectified linear unit (ReLU)**

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

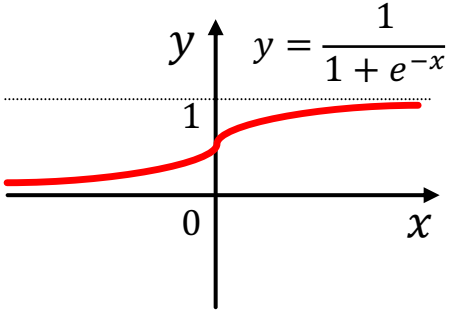
  - Most commonly used



- **Sigmoid**

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

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

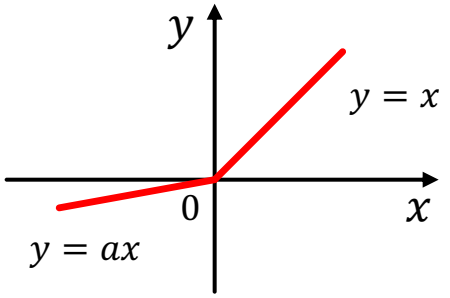


- **Parametric ReLU**

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

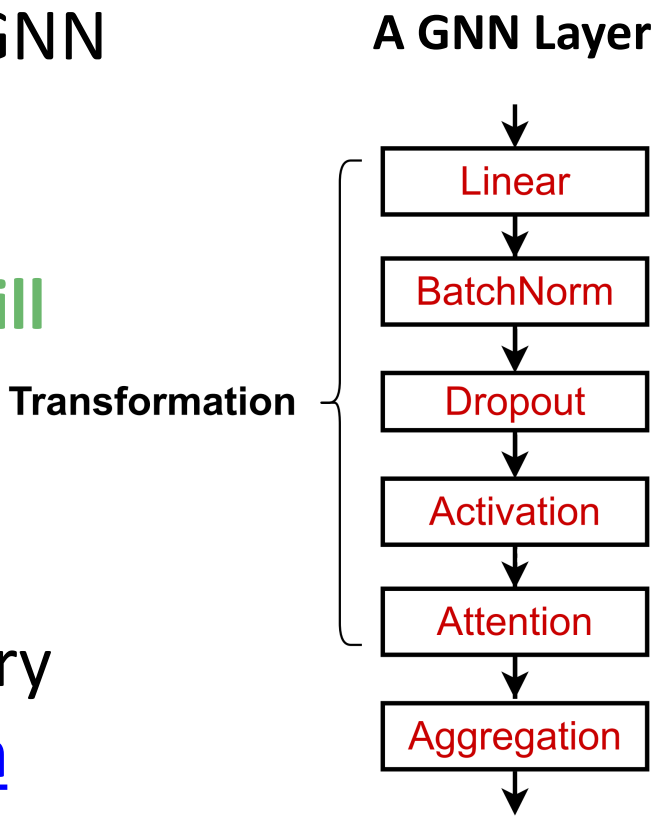
  a_i is a trainable parameter

  - Empirically performs better than ReLU



GNN Layer in Practice

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



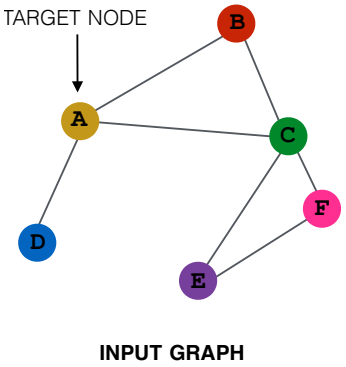
Stanford CS224W: Stacking Layers of a GNN

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Stacking GNN Layers

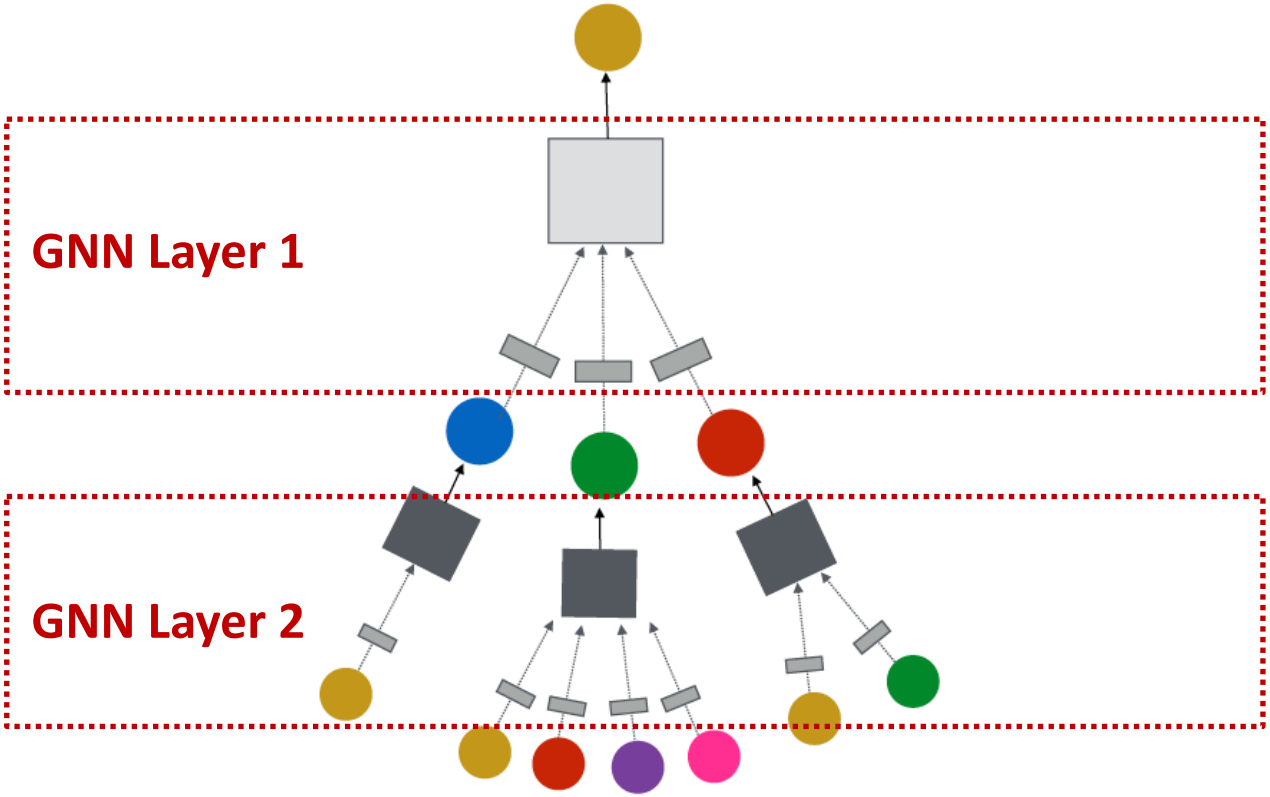


INPUT GRAPH

How to connect GNN layers into a GNN?

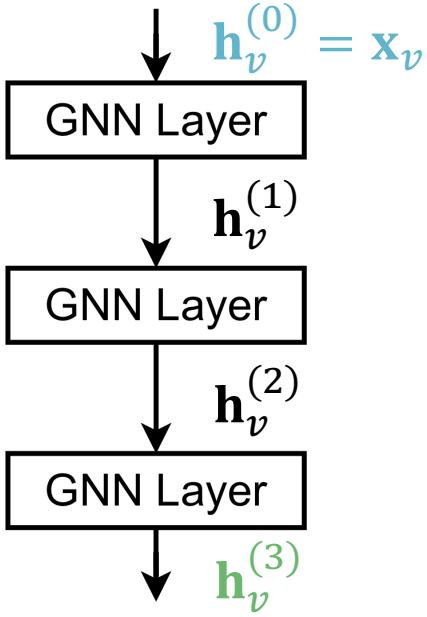
- Stack layers sequentially
- Ways of adding skip connections

(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



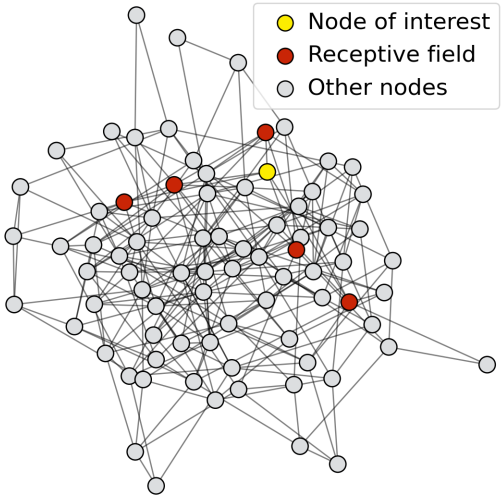
The Over-smoothing Problem

- **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?**

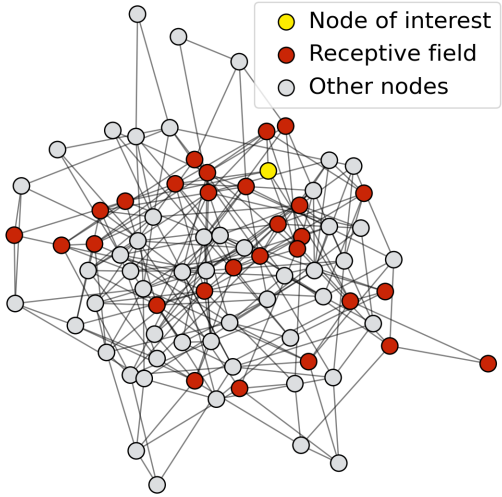
Receptive Field of a GNN

- **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**

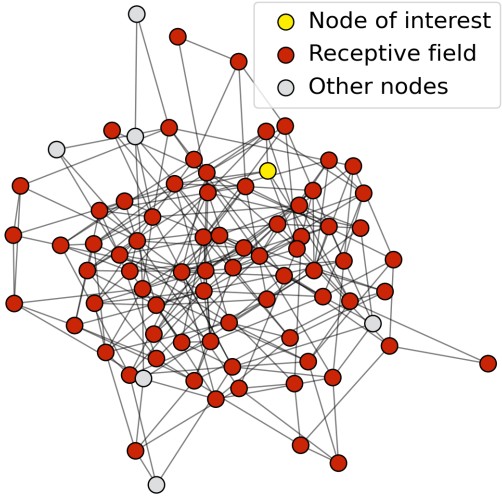
Receptive field for 1-layer GNN



Receptive field for 2-layer GNN



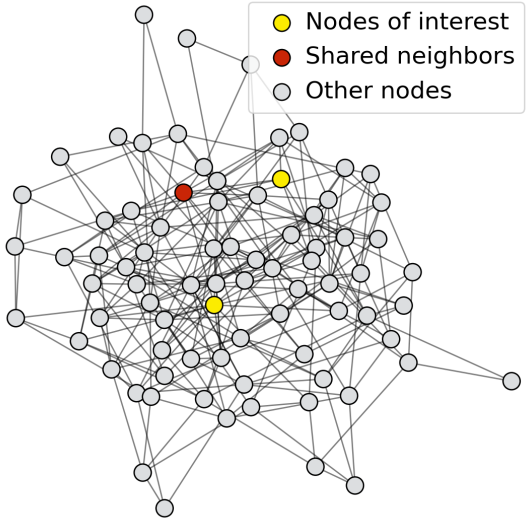
Receptive field for 3-layer GNN



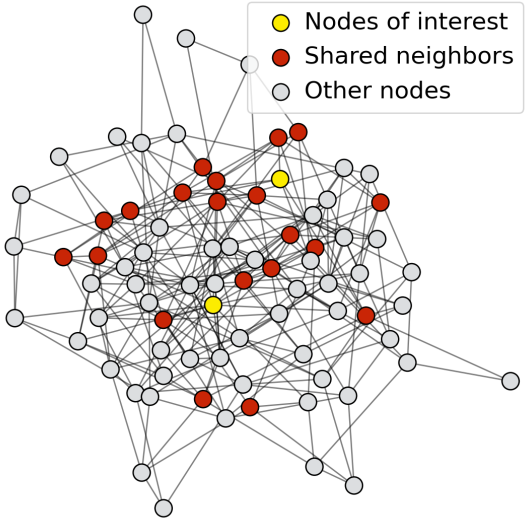
Receptive Field of a GNN

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

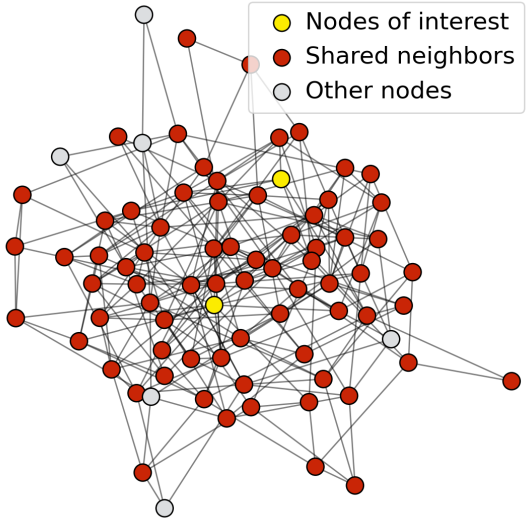
1-hop neighbor overlap
Only 1 node



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?

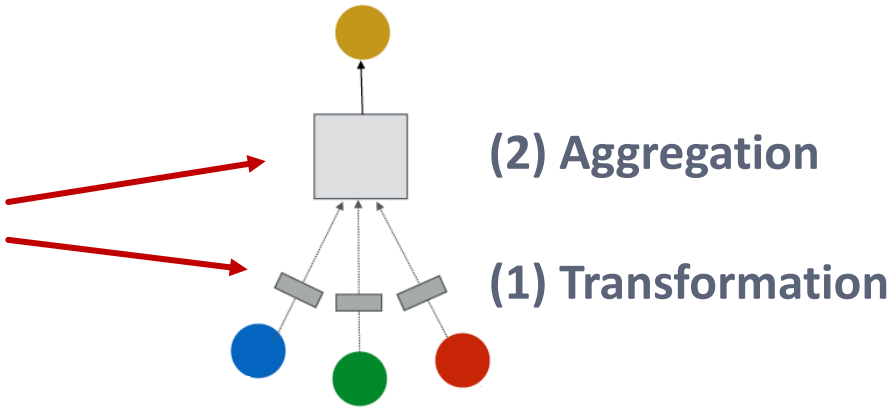
Design GNN Layer Connectivity

- **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?

Expressive Power for Shallow GNNs

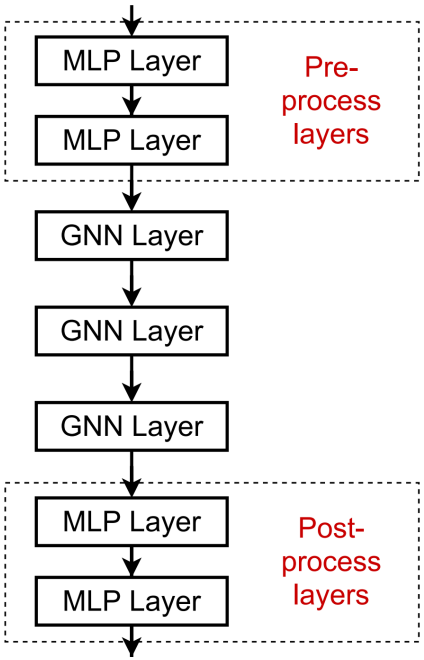
- **How to make a shallow GNN more expressive?**
- **Solution 1:** Increase the expressive power **within each GNN layer**
  - In our previous examples, each transformation or aggregation function only include one linear layer
  - We can **make aggregation / transformation become a deep neural network**!

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**



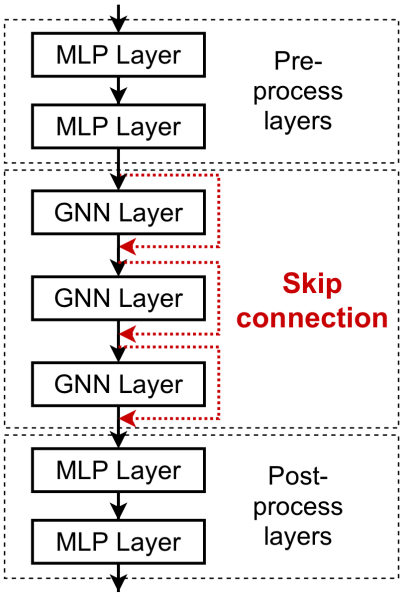
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

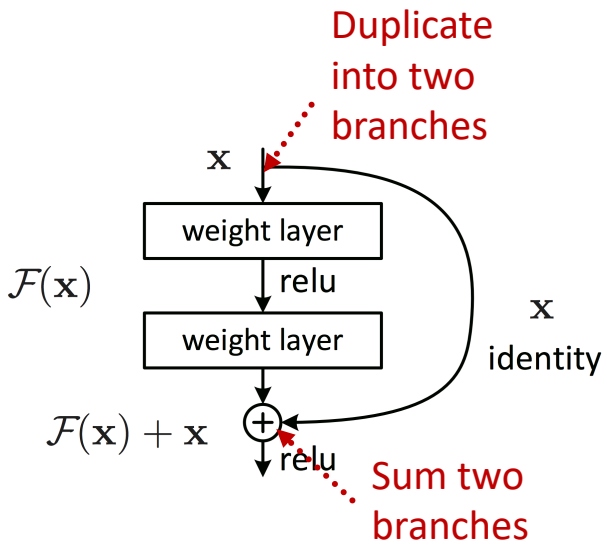
Post-processing layers: Important when reasoning / transformation over node embeddings are needed
E.g., graph classification, knowledge graphs

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:

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

After adding shortcuts:

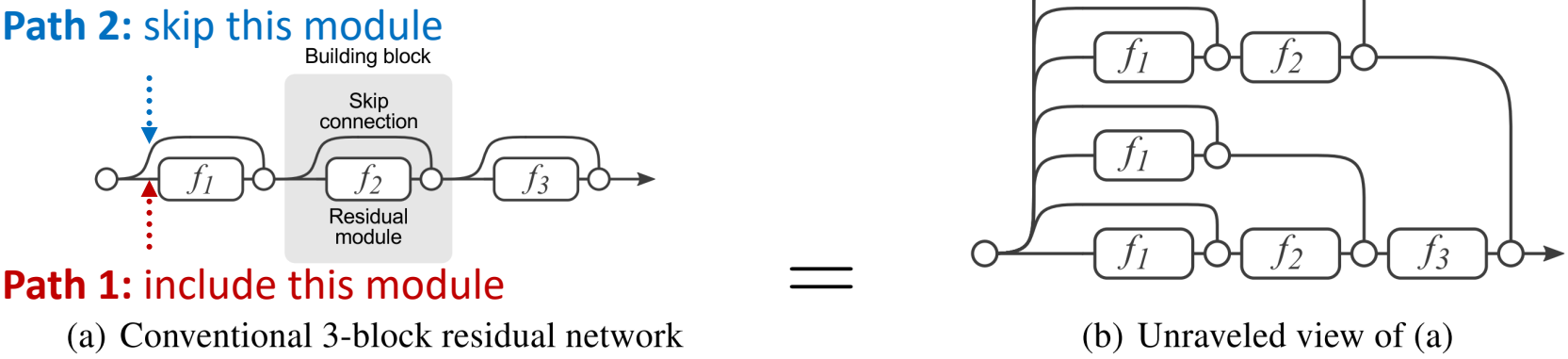
$$\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$$

Idea of Skip Connections

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

All the possible paths:

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



(a) Conventional 3-block residual network

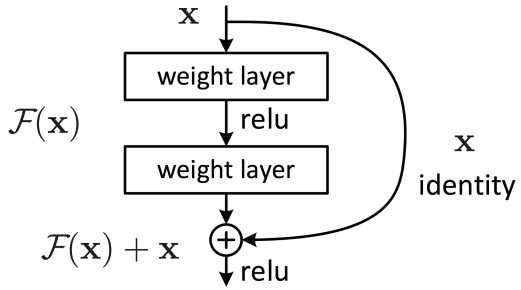
(b) Unraveled view of (a)

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

Example: GCN with Skip Connections

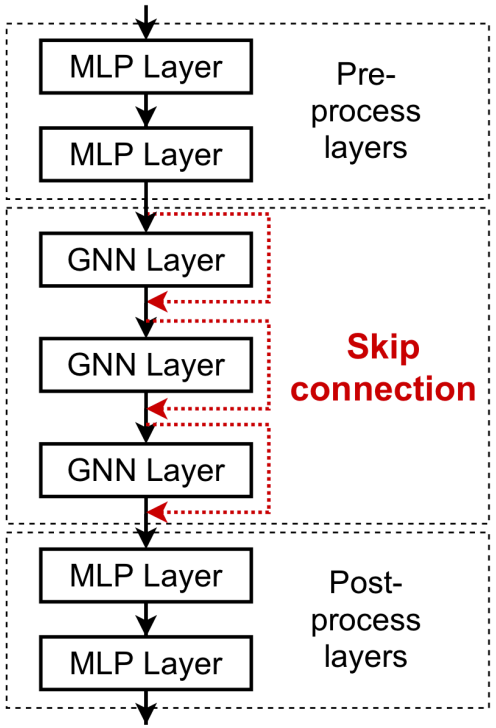
- **A standard GCN layer**
- $\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$

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



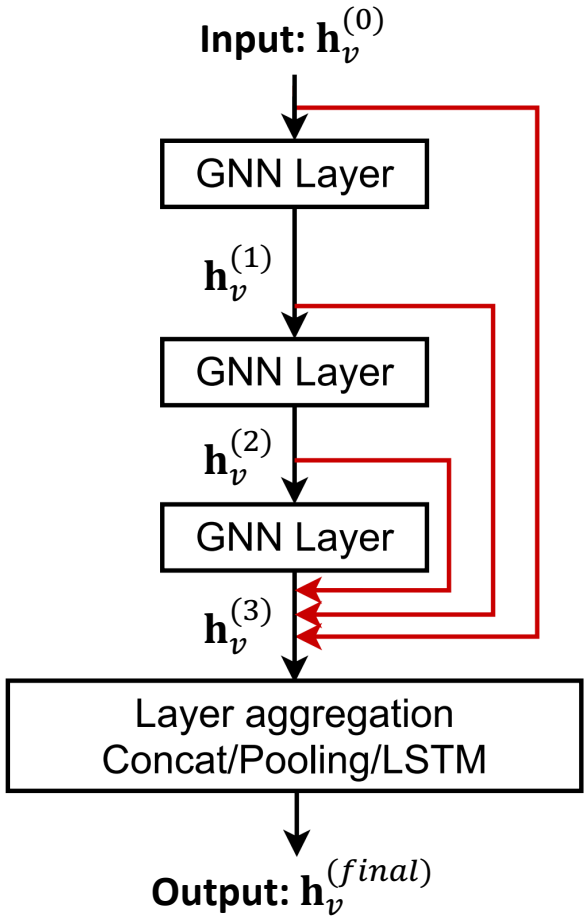
- **A GCN layer with skip connection**
- $\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)}\right)$

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



Other Options of Skip Connections

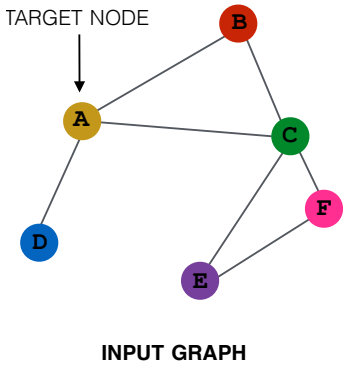
- **Other options:** Directly skip to the last layer
  - The final layer directly **aggregates from the all the node embeddings** in the previous layers



Stanford CS224W: Graph Manipulation in GNNs

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

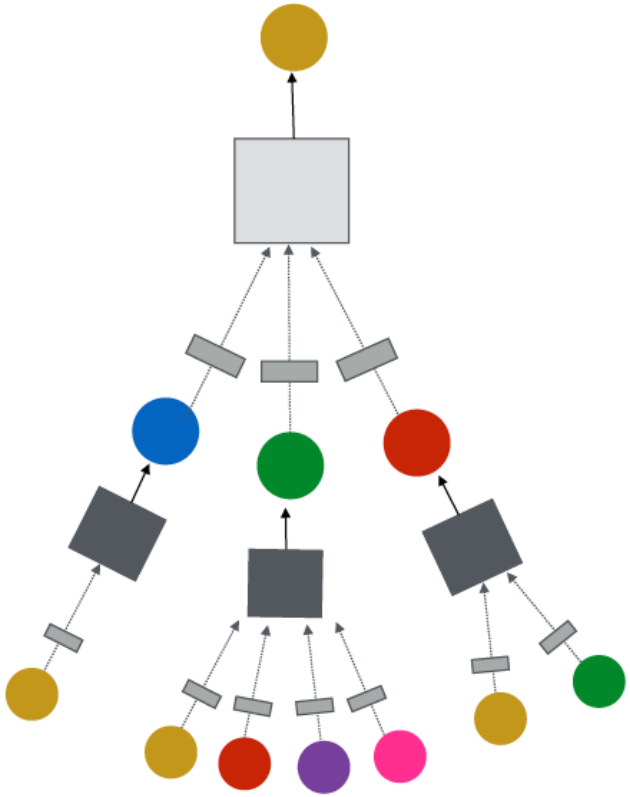
General GNN Framework



INPUT GRAPH

Idea: Raw input graph ≠ 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

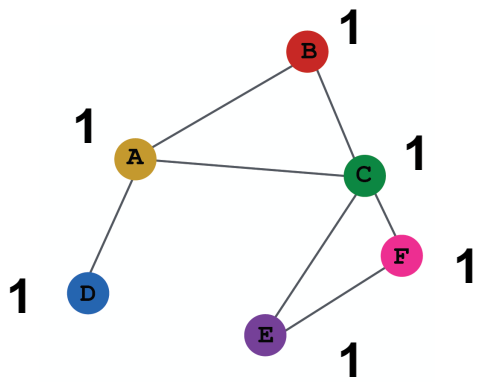
Graph Manipulation Approaches

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

Feature Augmentation on Graphs

Why do we need feature augmentation?

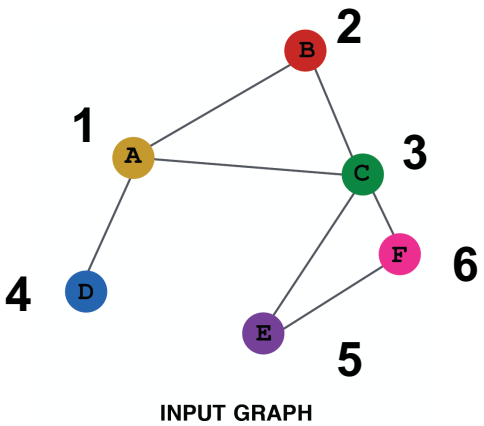
- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix
- **Standard approaches:**
- **a) Assign constant values to nodes**



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**



INPUT GRAPH

One-hot vector for node with ID=5

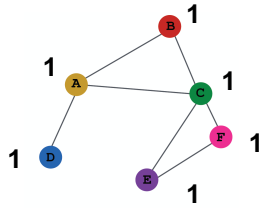
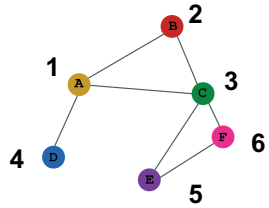
ID = 5

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

Total number of IDs = 6

Feature Augmentation on Graphs

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

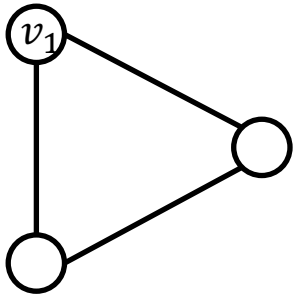
	Constant node feature	**One-hot node feature**
		
Expressive power	**Medium**. All the nodes are identical, but GNN can still learn from the graph structure	**High**. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High**. Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN	**Low**. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs
Computational cost	**Low**. Only 1 dimensional feature	**High**. $O(\|V\|)$ dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

Feature Augmentation on Graphs

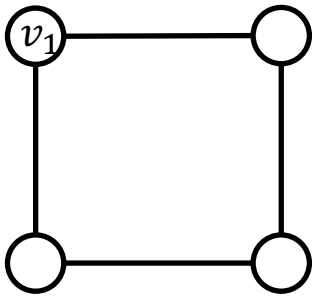
Why do we need feature augmentation?

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

v_1 resides in a cycle with length 3



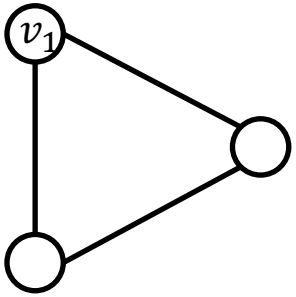
v_1 resides in a cycle with length 4



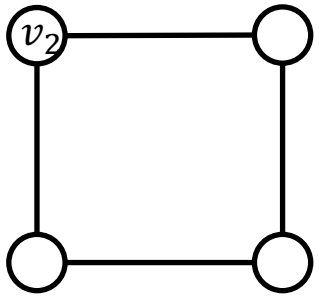
Feature Augmentation on Graphs

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

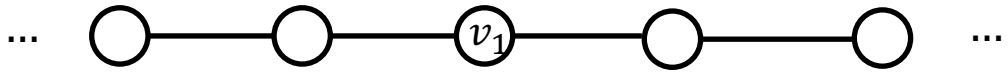
v_1 resides in a cycle with length 3



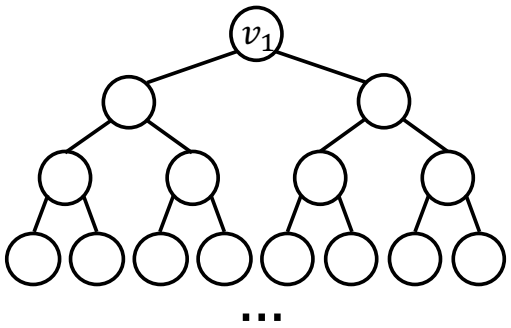
v_1 resides in a cycle with length 4



v_1 resides in a cycle with infinite length



The computational graphs for node v_1 are always the same



Feature Augmentation on Graphs

Why do we need feature augmentation?

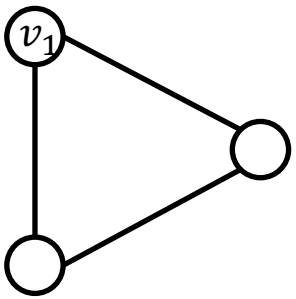
- **(2) Certain structures are hard to learn by GNN**
- **Solution:**
  - We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for v_1

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

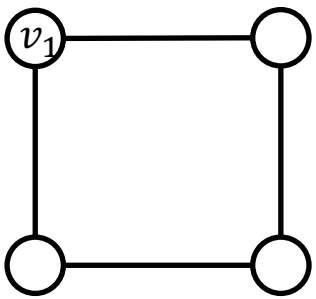
v_1 resides in a cycle with length 3



Augmented node feature for v_1

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

v_1 resides in a cycle with length 4



Feature Augmentation on Graphs

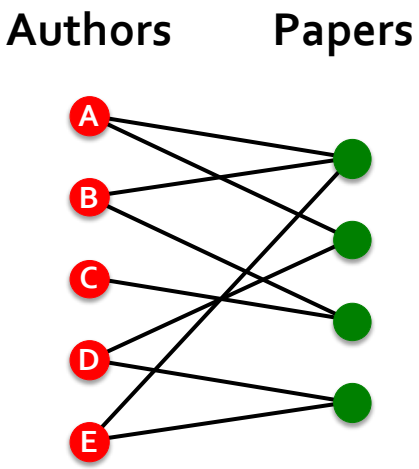
Why do we need feature augmentation?

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

Add Virtual Nodes / Edges

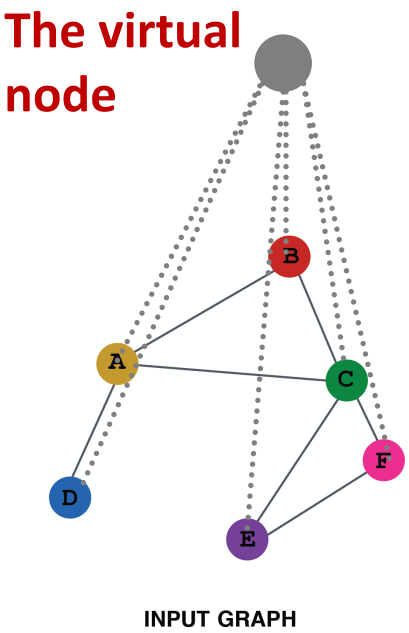
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
  - **Common approach:** Connect 2-hop neighbors via virtual edges
  - **Intuition:** Instead of using adj. matrix A for GNN computation, use $A + A^2$

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



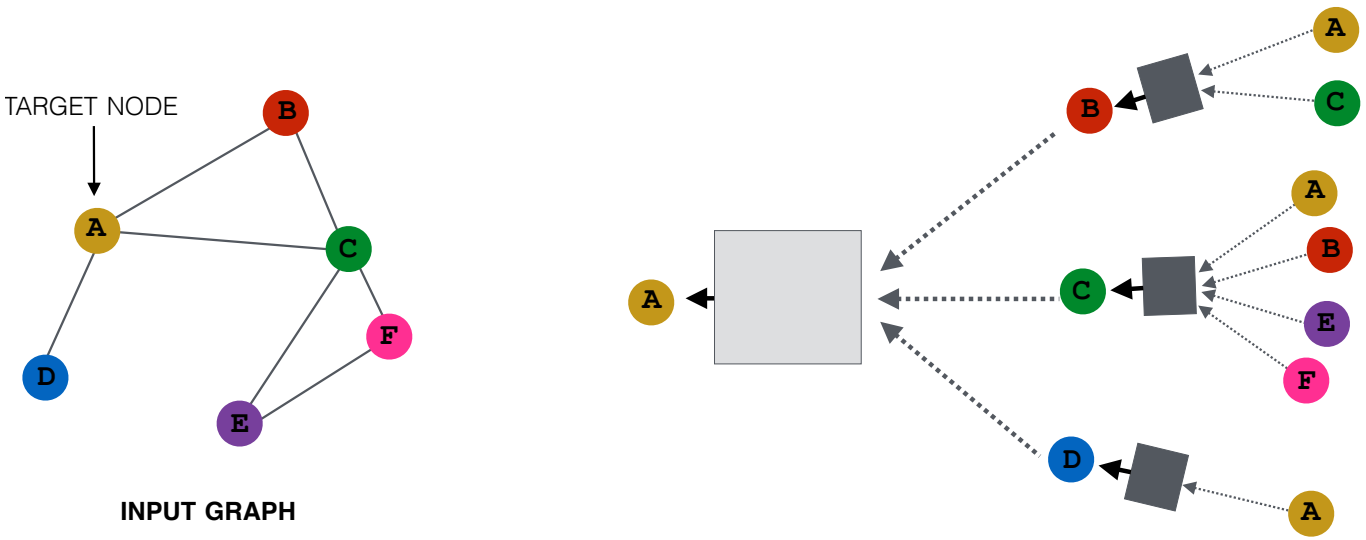
Add Virtual Nodes / Edges

- **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 2**
      - **Node A – Virtual node – Node B**
  - **Benefits:** Greatly **improves message passing in sparse graphs**



Node Neighborhood Sampling

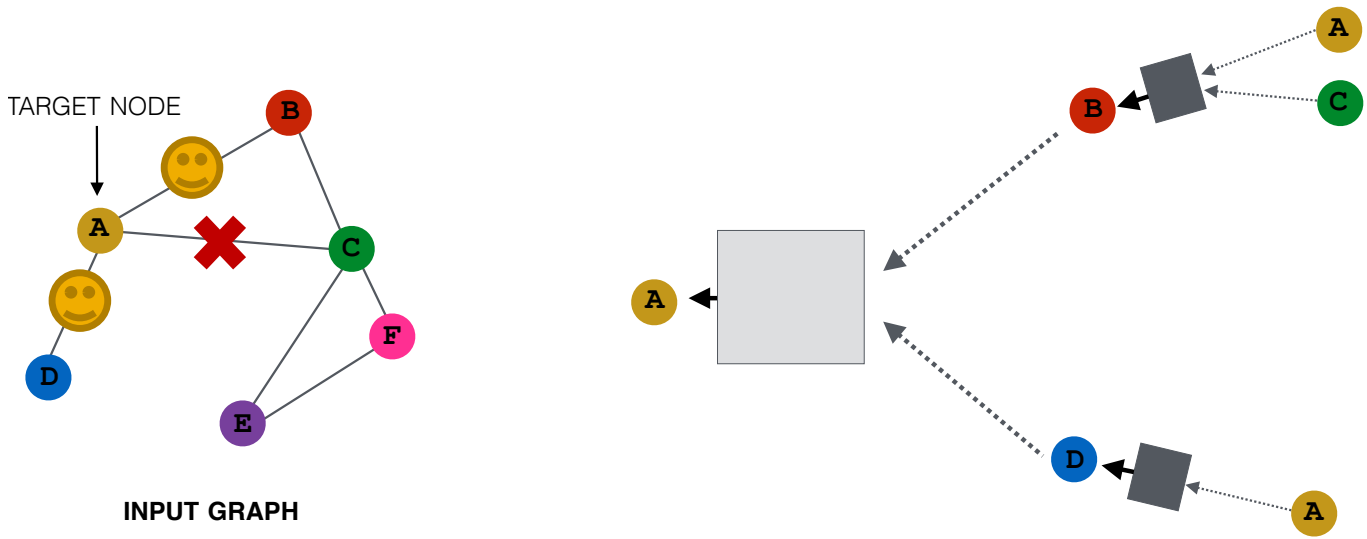
- **Previously:**
  - All the nodes are used for message passing



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

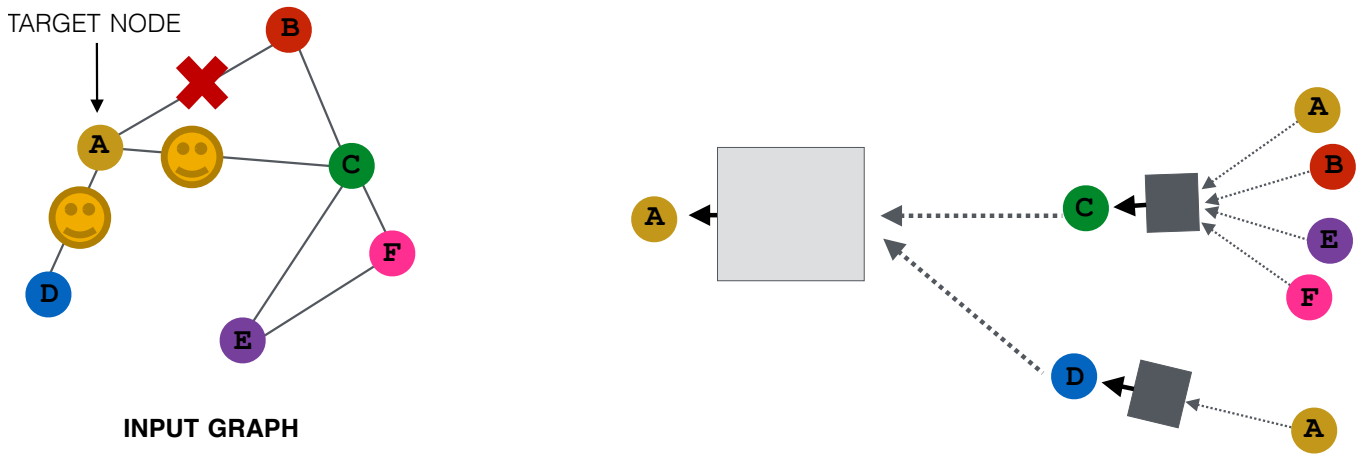
Neighborhood Sampling Example

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



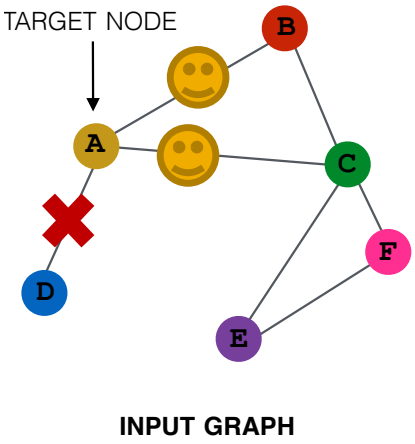
Neighborhood Sampling Example

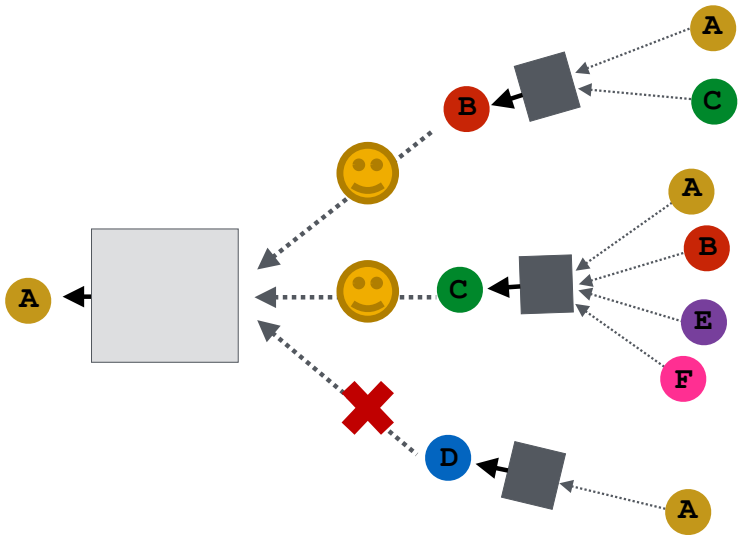
- **Next time when we compute the embeddings, we can sample different neighbors**
  - Only nodes C and D will pass message to A



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!





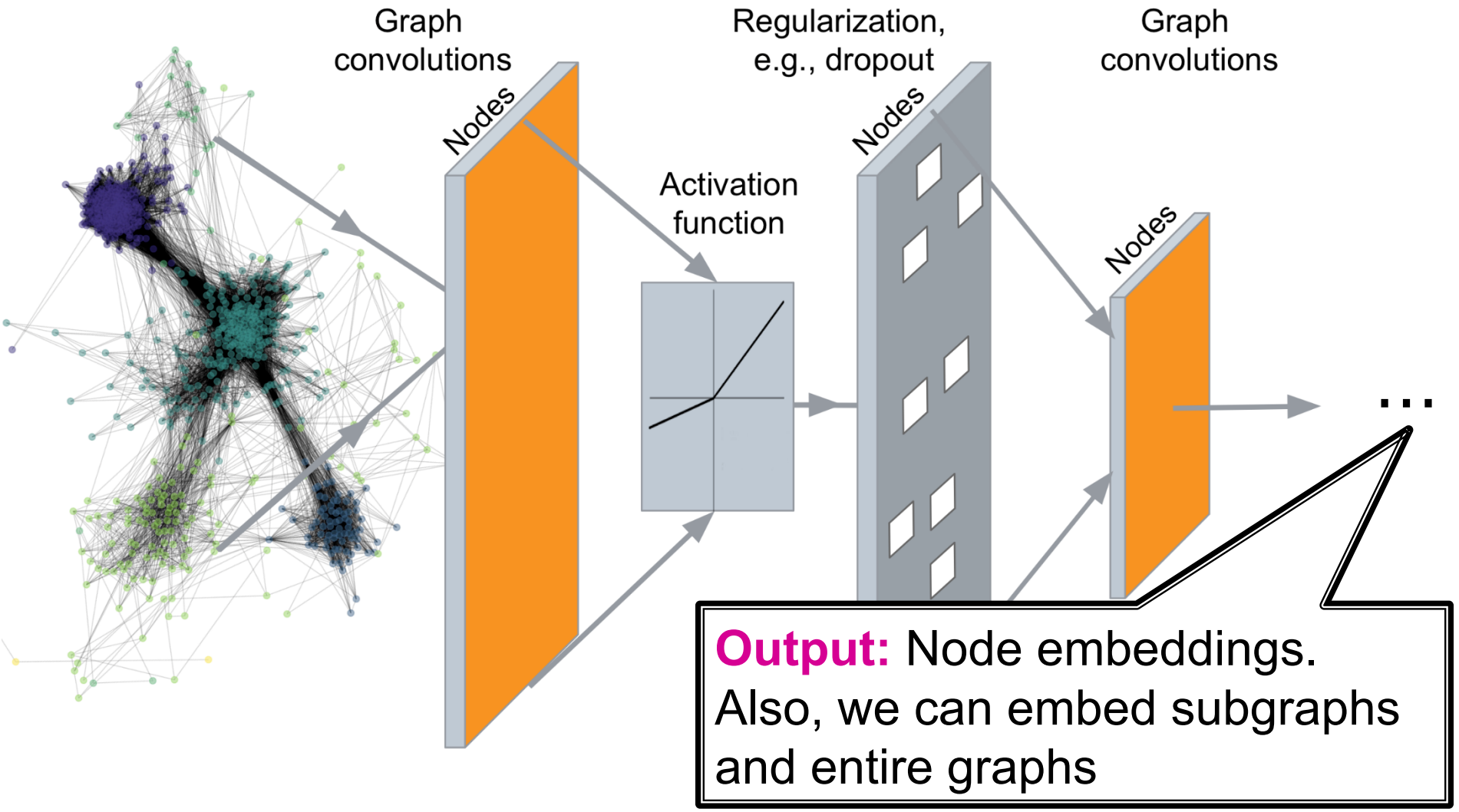
Summary of the lecture

- **Recap: A general perspective for GNNs**
  - **GNN Layer**:
    - Transformation + Aggregation
    - Classic GNN layers: GCN, GraphSAGE, GAT
  - **Layer connectivity**:
    - Deciding number of layers
    - Skip connections
  - **Graph Manipulation:**
    - Feature augmentation
    - Structure manipulation
- **Next:** GNN objectives, GNN in practice

Stanford CS224W: GNN Augmentation and Training

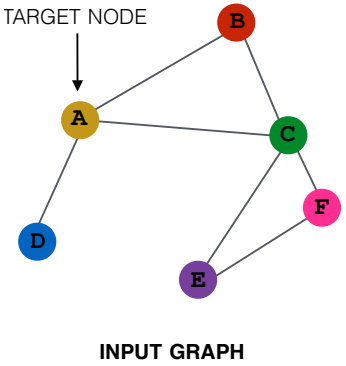
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

Recap: Deep Graph Encoders



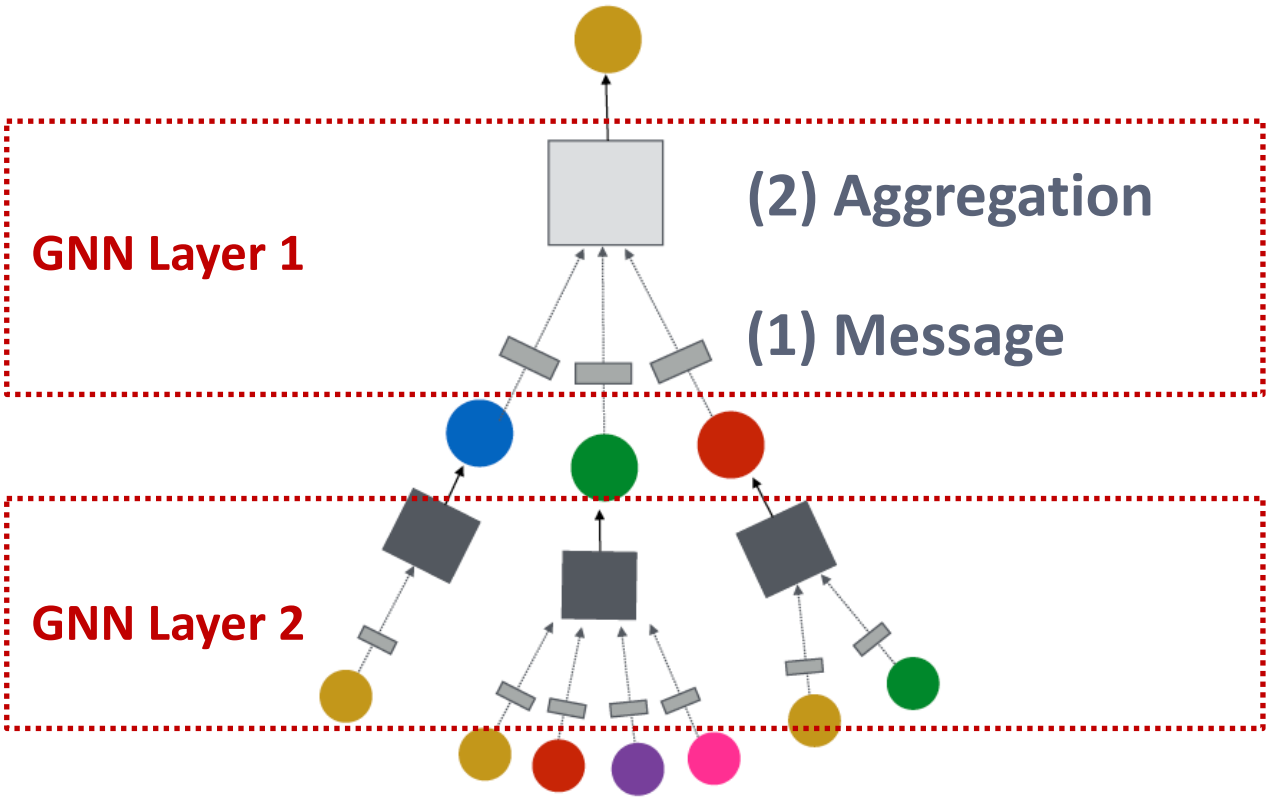
Output: Node embeddings. Also, we can embed subgraphs and entire graphs

Recap: A General GNN Framework



INPUT GRAPH

(5) Learning objective



(3) Layer connectivity

(4) Graph augmentation

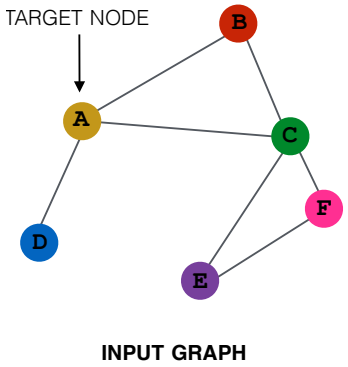
Stanford CS224W: Graph Augmentation for GNNs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

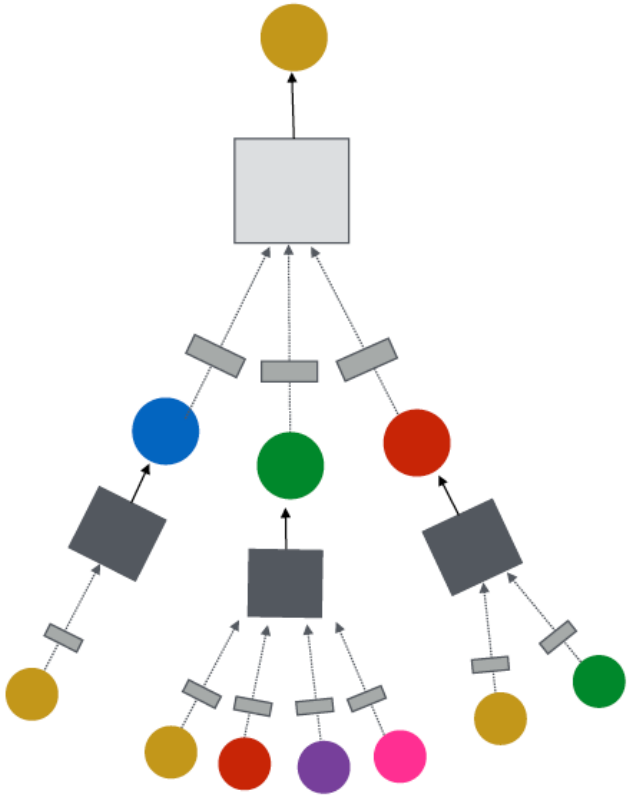
General GNN Framework



INPUT GRAPH

Idea: Raw input graph ≠ 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

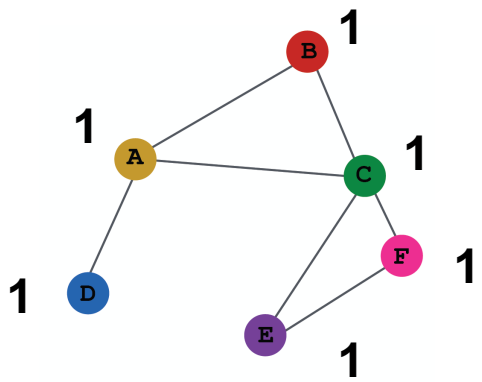
Graph Augmentation Approaches

- **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**

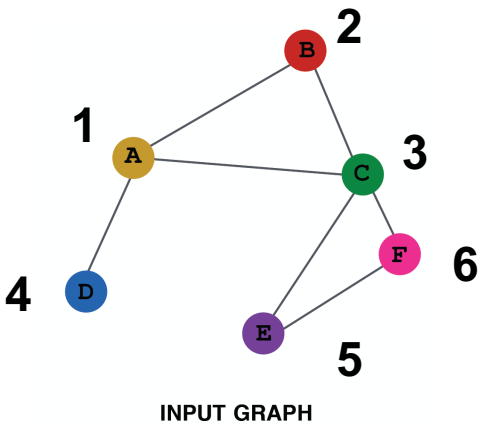


INPUT GRAPH

Feature Augmentation on Graphs

Why do we need feature augmentation?

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



One-hot vector for node with ID=5

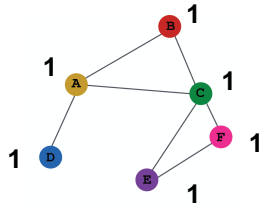
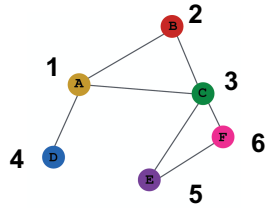
ID = 5

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

Total number of IDs = 6

Feature Augmentation on Graphs

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

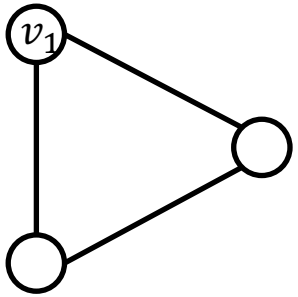
	Constant node feature	**One-hot node feature**
	 INPUT GRAPH	 INPUT GRAPH
Expressive power	**Medium**. All the nodes are identical, but GNN can still learn from the graph structure	**High**. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High**. Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN	**Low**. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs
Computational cost	**Low**. Only 1 dimensional feature	**High**. $O(\|V\|)$ dimensional feature, cannot apply to large graphs
Use cases	**Any graph, inductive settings** (generalize to new nodes)	**Small graph, transductive settings** (no new nodes)

Feature Augmentation on Graphs

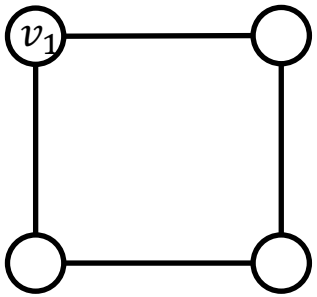
Why do we need feature augmentation?

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

v_1 resides in a cycle with length 3



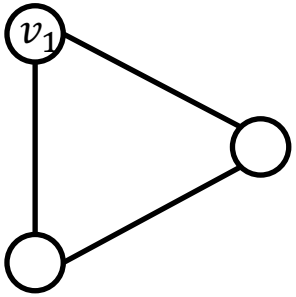
v_1 resides in a cycle with length 4



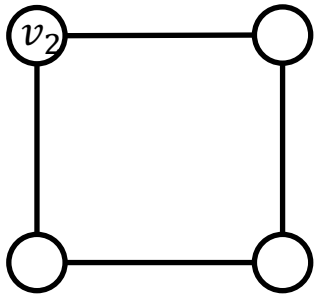
Feature Augmentation on Graphs

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

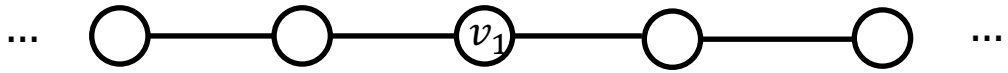
v_1 resides in a cycle with length 3



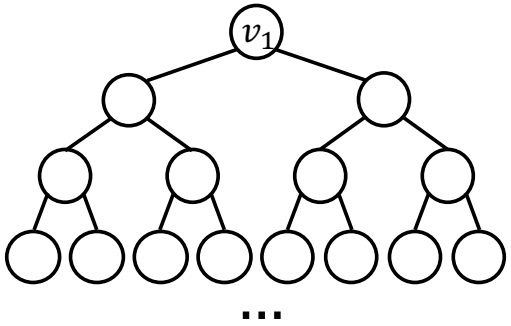
v_1 resides in a cycle with length 4



v_1 resides in a cycle with infinite length



The computational graphs for node v_1 are always the same



More about this topic later!

Feature Augmentation on Graphs

Why do we need feature augmentation?

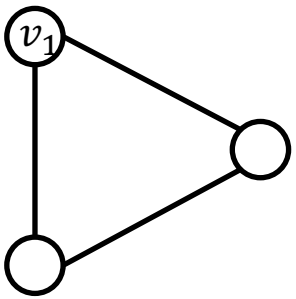
- **(2) Certain structures are hard to learn by GNN**
- **Solution:**
  - We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for v_1

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

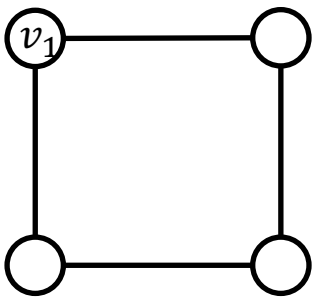
v_1 resides in a cycle with length 3



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

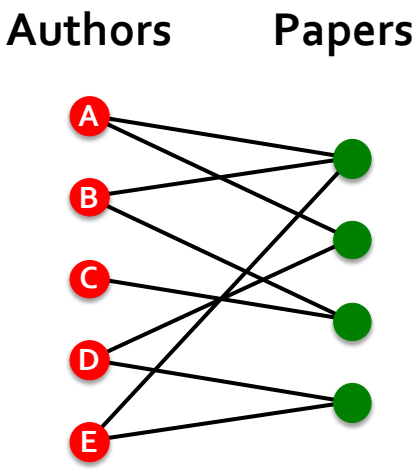
Why do we need feature augmentation?

- **(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

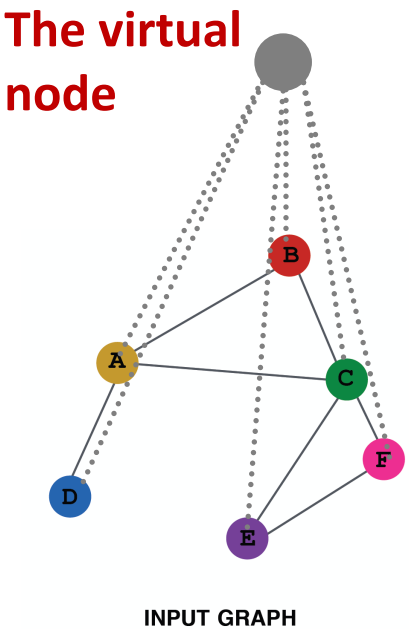
- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
  - **Common approach:** Connect 2-hop neighbors via virtual edges
  - **Intuition:** Instead of using adj. matrix A for GNN computation, use $A + A^2$

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



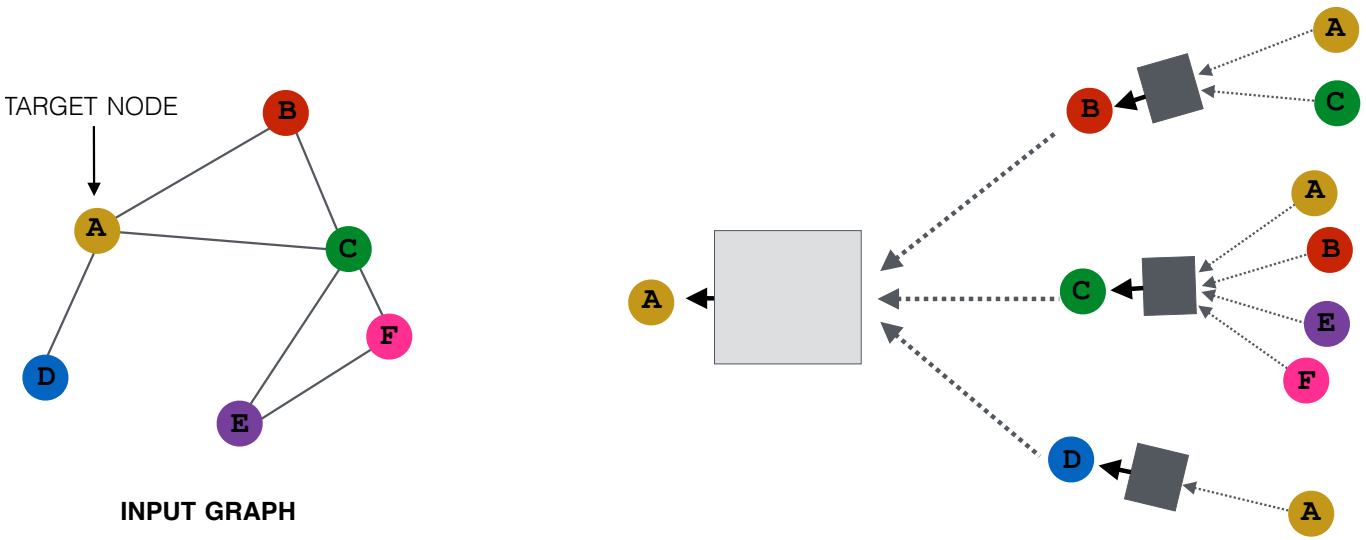
Add Virtual Nodes / Edges

- **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

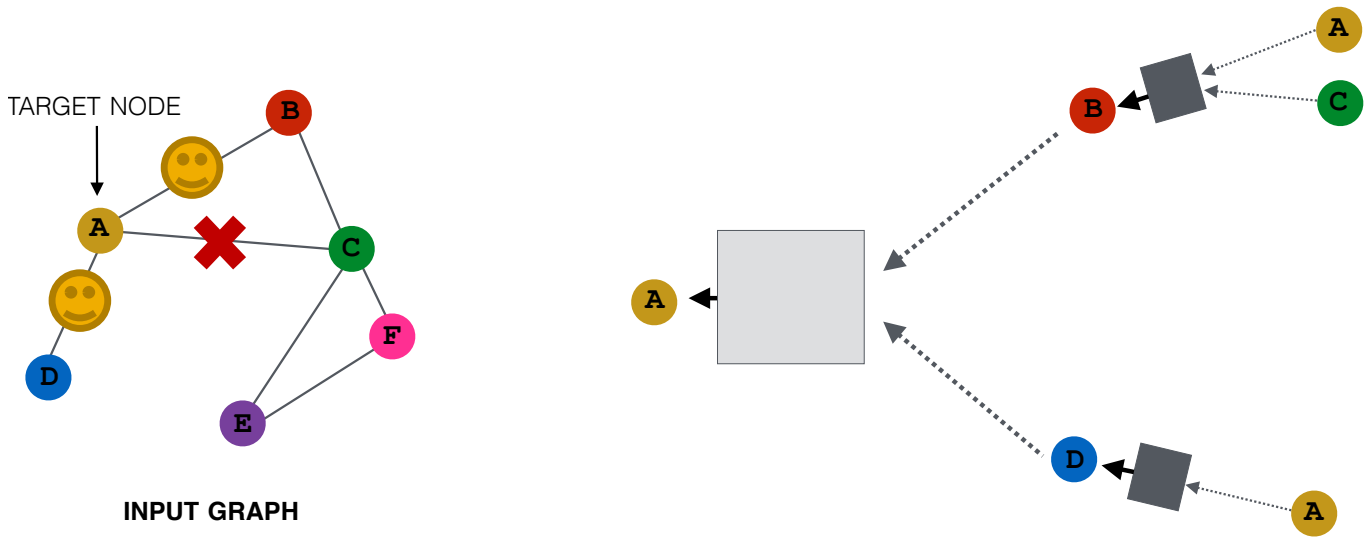
- **Previously:**
  - All the nodes are used for message passing



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

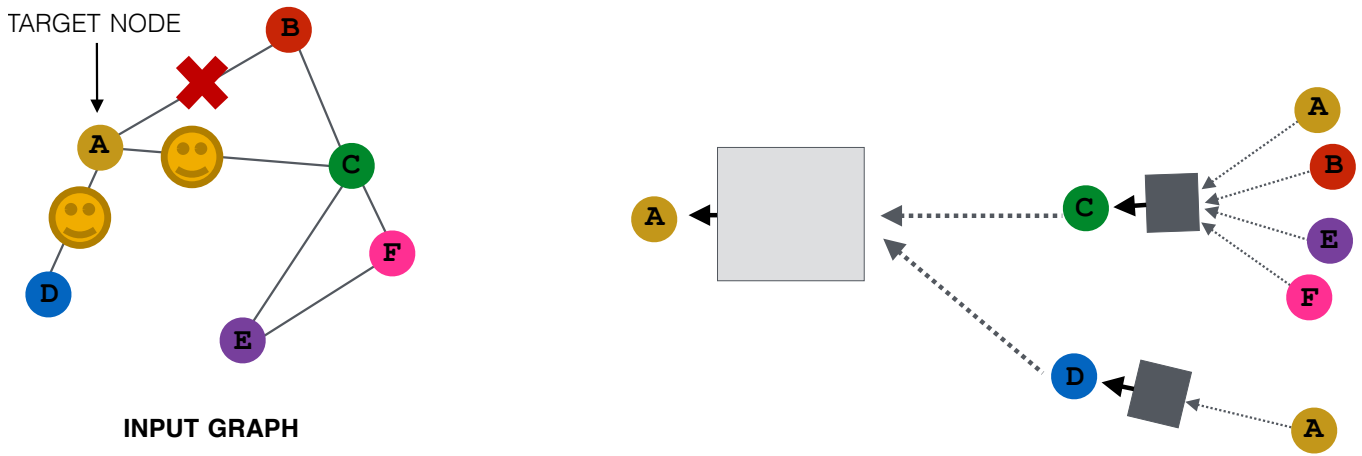
Neighborhood Sampling Example

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



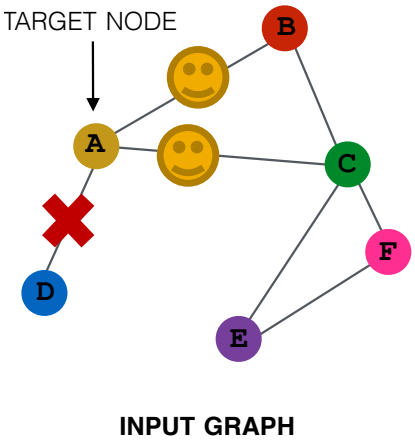
Neighborhood Sampling Example

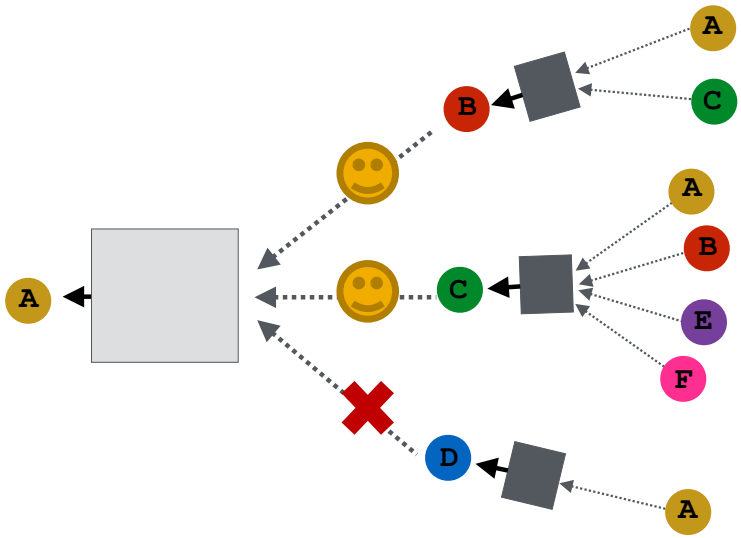
- **In the next layer when we compute the embeddings, we can sample different neighbors**
  - Only nodes C and D will pass messages to A



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!

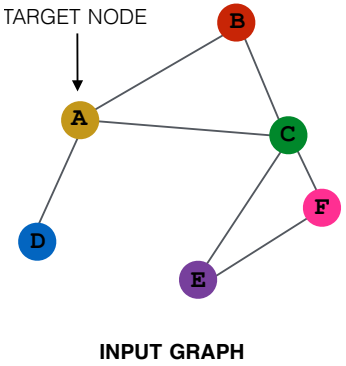




Stanford CS224W: Training Graph Neural Networks

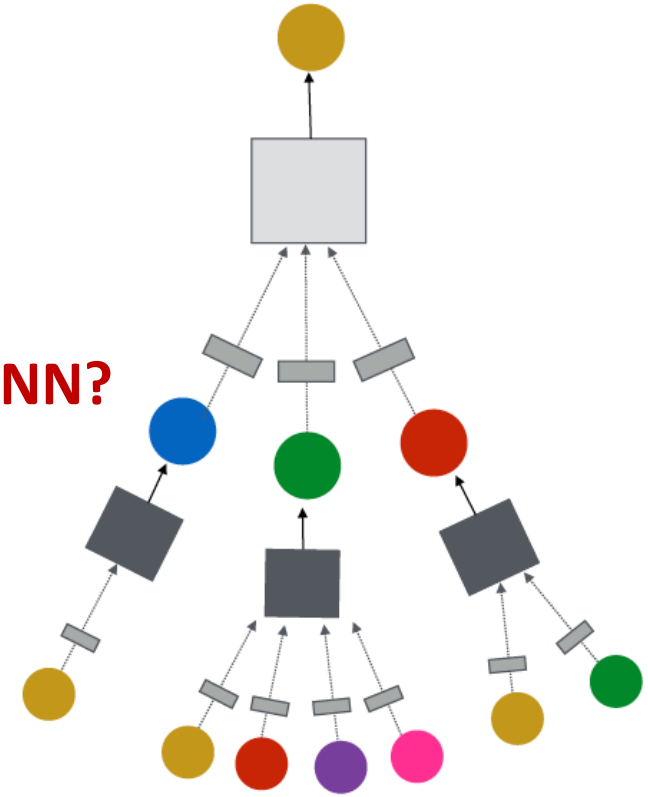
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

A General GNN Framework (4)



INPUT GRAPH

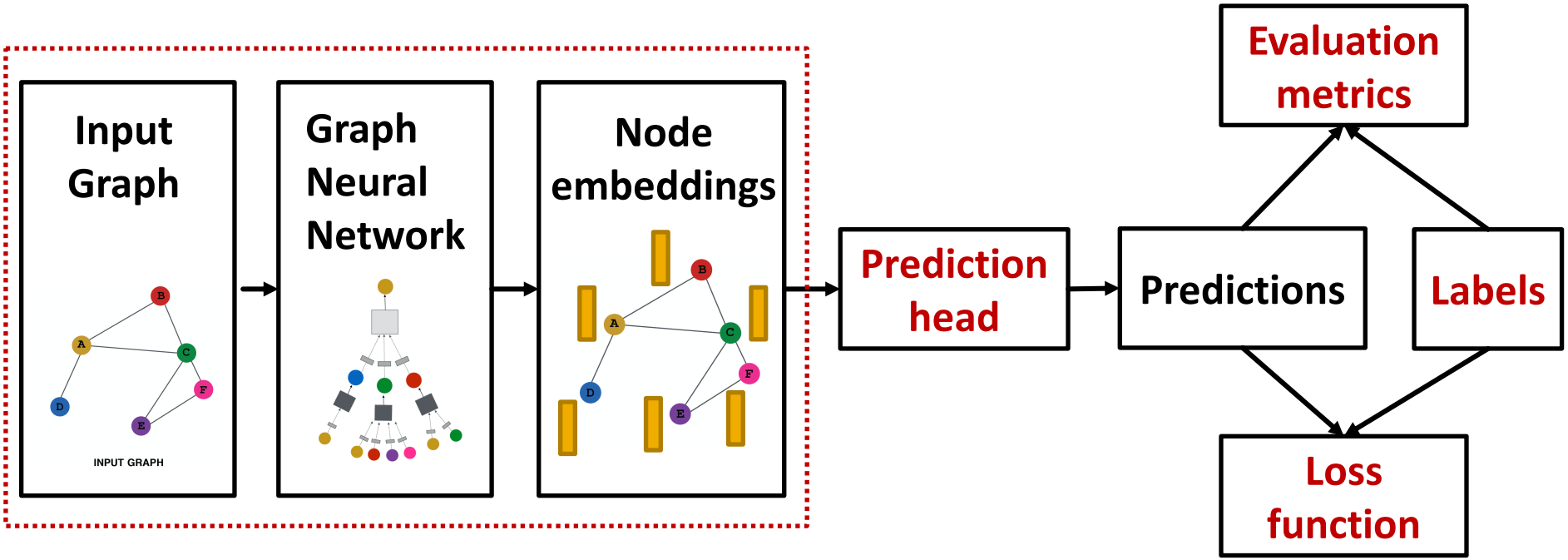
(5) Learning objective



Next: How do we train a GNN?

GNN Training Pipeline

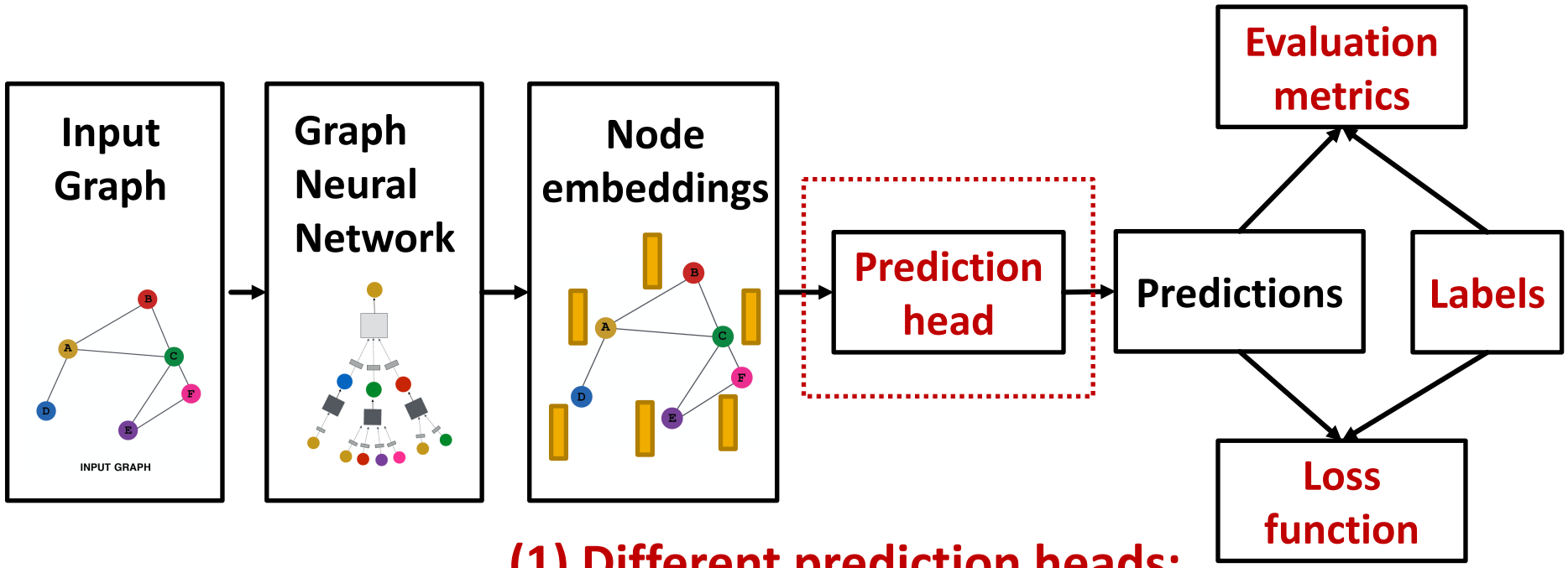
So far what we have covered



Output of a GNN: set of node embeddings

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

GNN Training Pipeline (1)

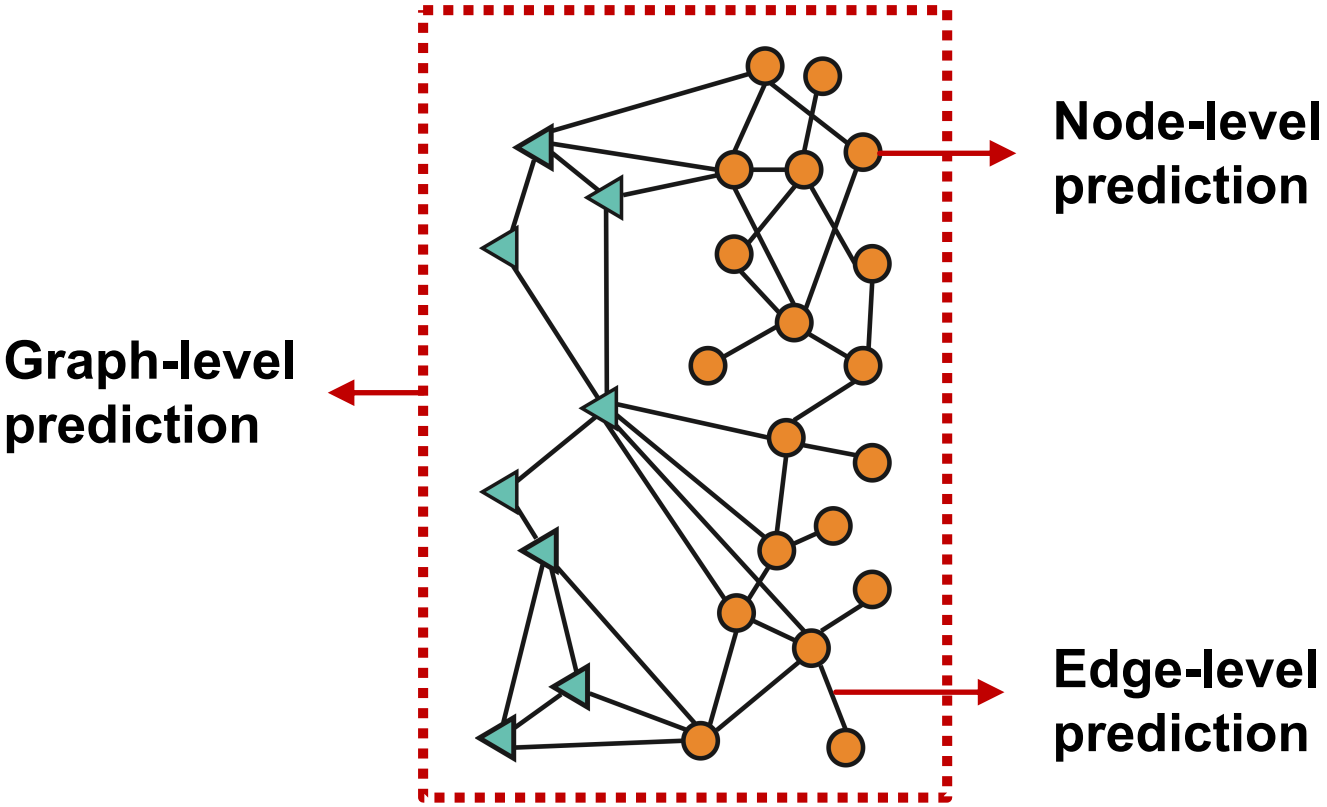


(1) Different prediction heads:

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

GNN Prediction Heads

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

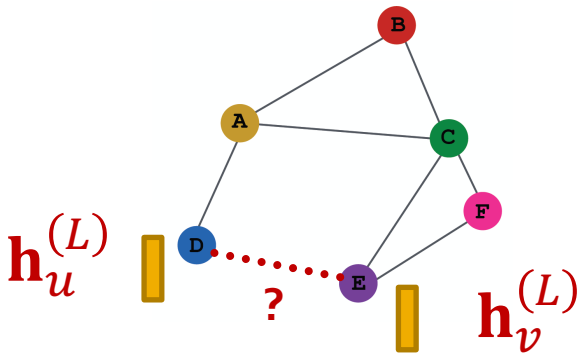


Prediction Heads: Node-level

- **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

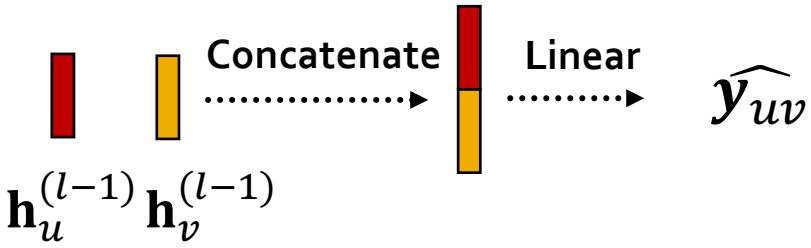
- **Edge-level prediction**: Make prediction using pairs of node embeddings
- Suppose we want to make k-way prediction
- $\hat{\boldsymbol{y}}_{\boldsymbol{uv}} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



- 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

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

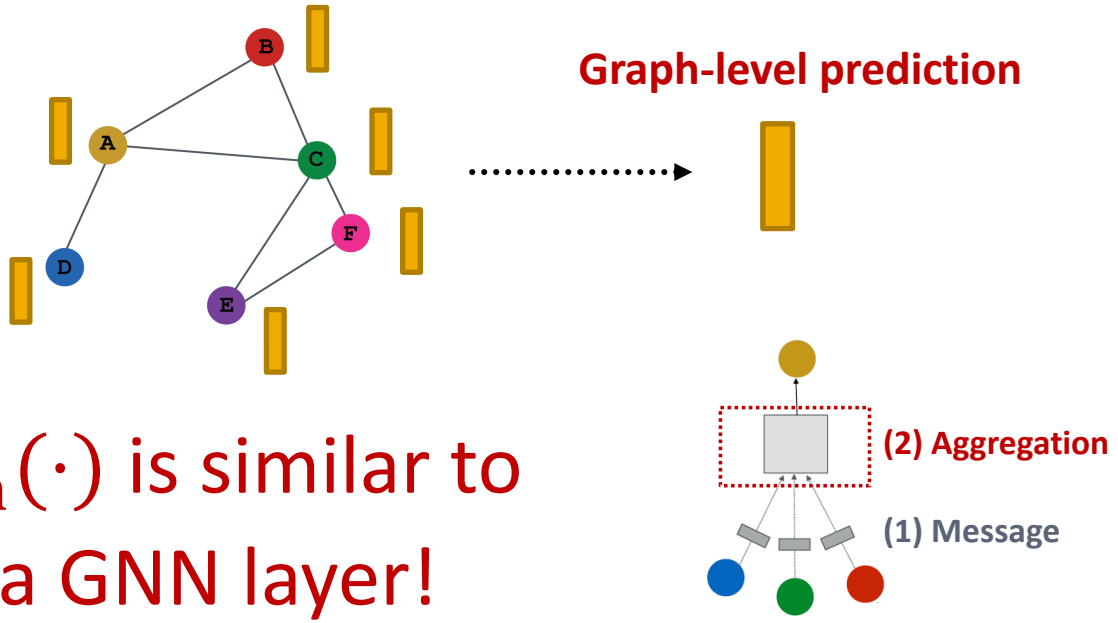
$$\ldots$$

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

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

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

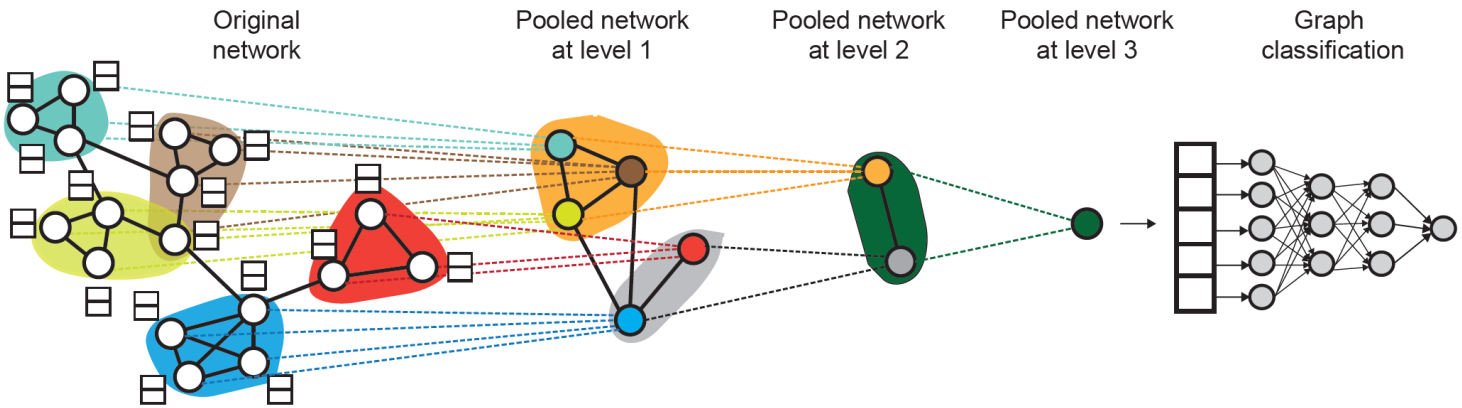
- **A solution:** Let's aggregate all the node embeddings **hierarchically**
  - **Toy example:** We will aggregate via $\mathrm{ReLU}(\mathrm{Sum}(\cdot))$
    - We first separately aggregate the first 2 nodes and last 3 nodes
    - Then we aggregate again to make the final prediction
  - G_1 node embeddings: $\{-1, -2, 0, 1, 2\}$
    - **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

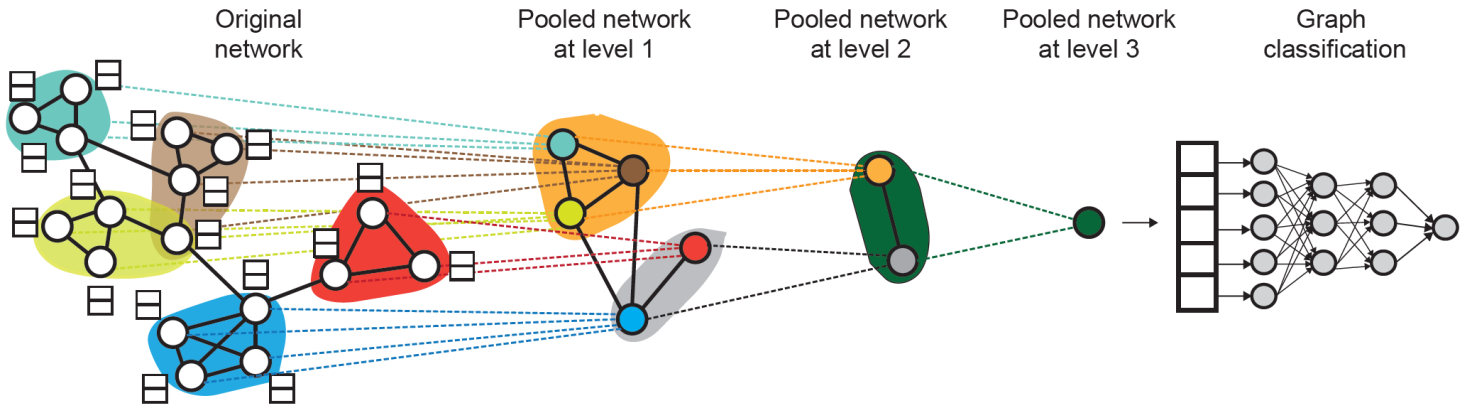
- **DiffPool idea:**
  - **Hierarchically pool node embeddings**



  - **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**

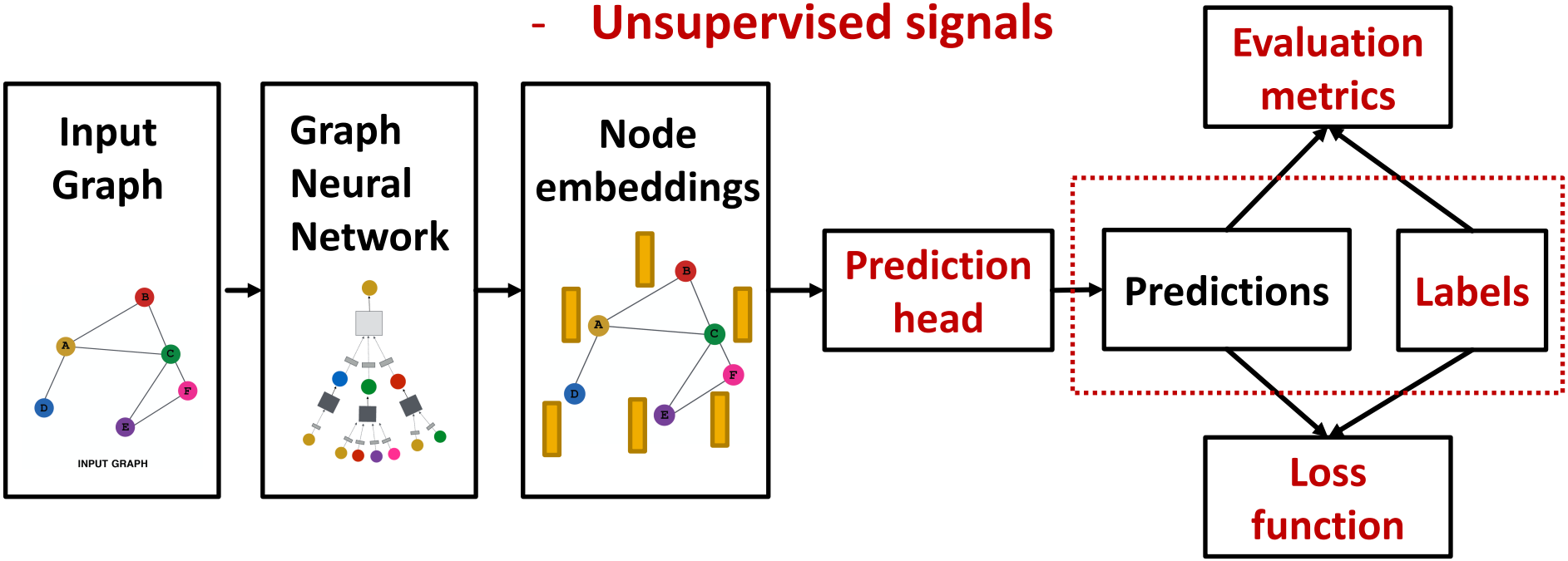
Stanford CS224W: Training Graph Neural Networks

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**"

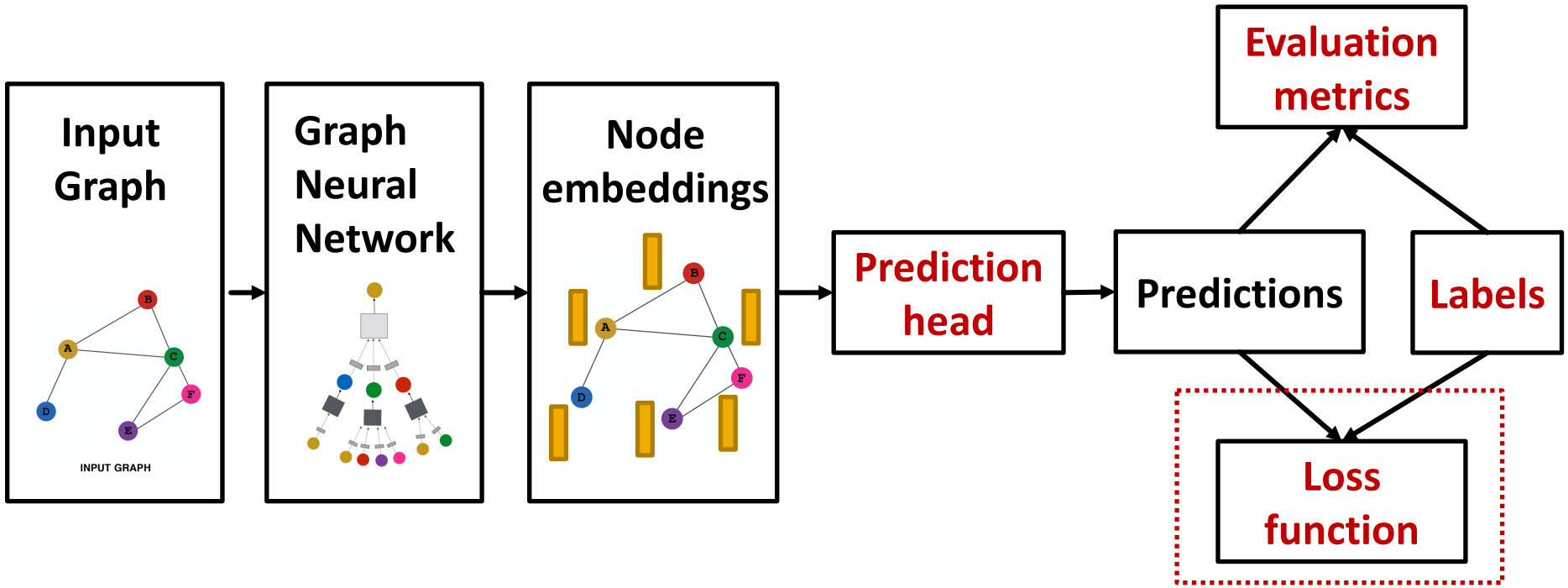
Supervised Labels on Graphs

- **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:

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

Label: $\boldsymbol{y}^{(i)}$ — Prediction: $\hat{\boldsymbol{y}}^{(i)}$

i-th data point: (i) — j-th class: j

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

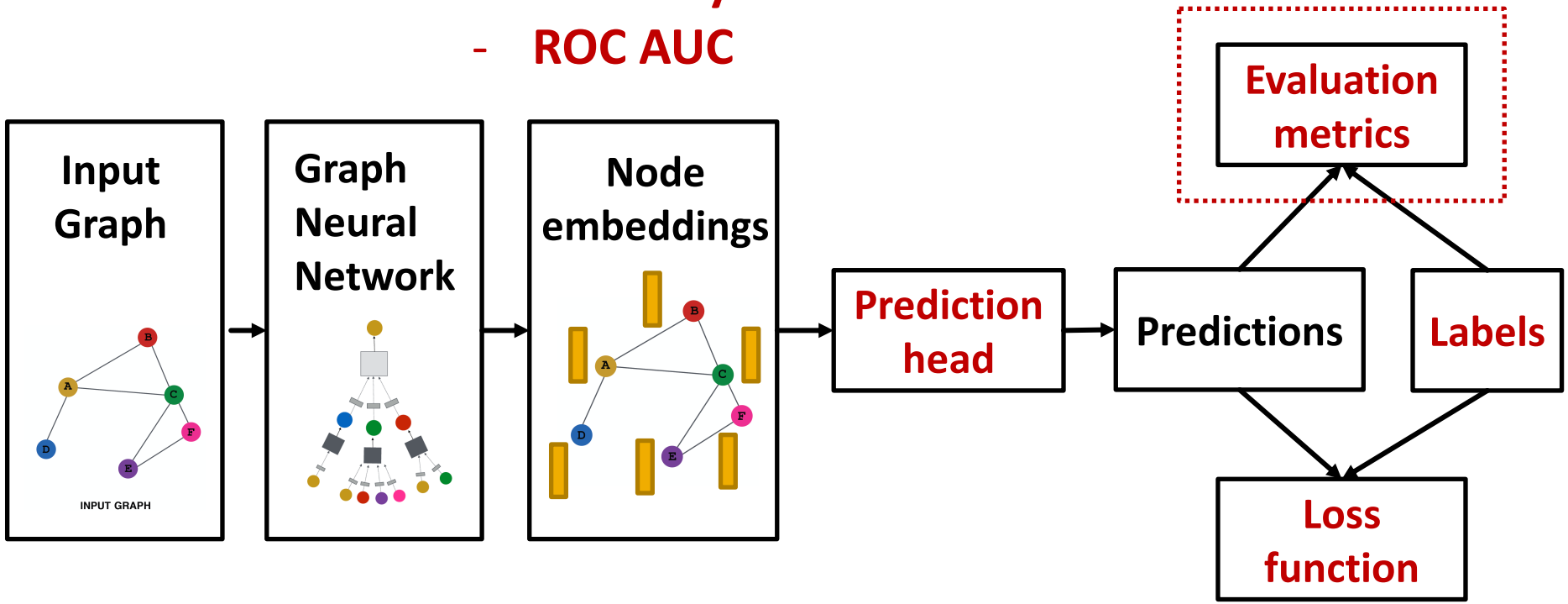
- Total loss over all N training examples

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

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

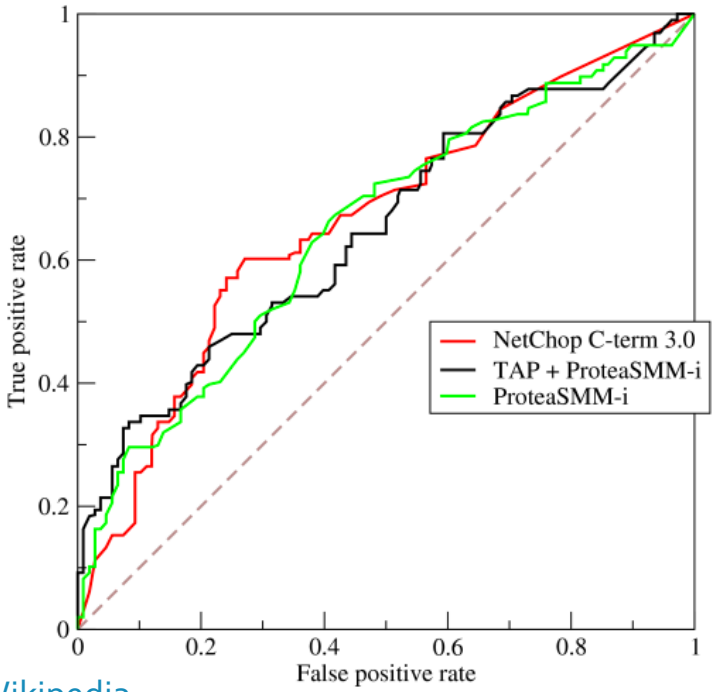
	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

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

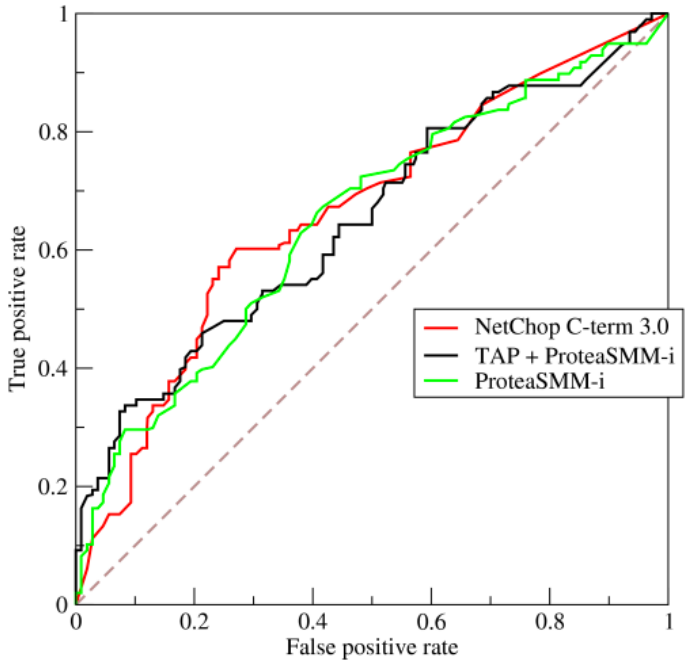
Image Credit: Wikipedia

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

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

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

(4) Evaluation Metrics



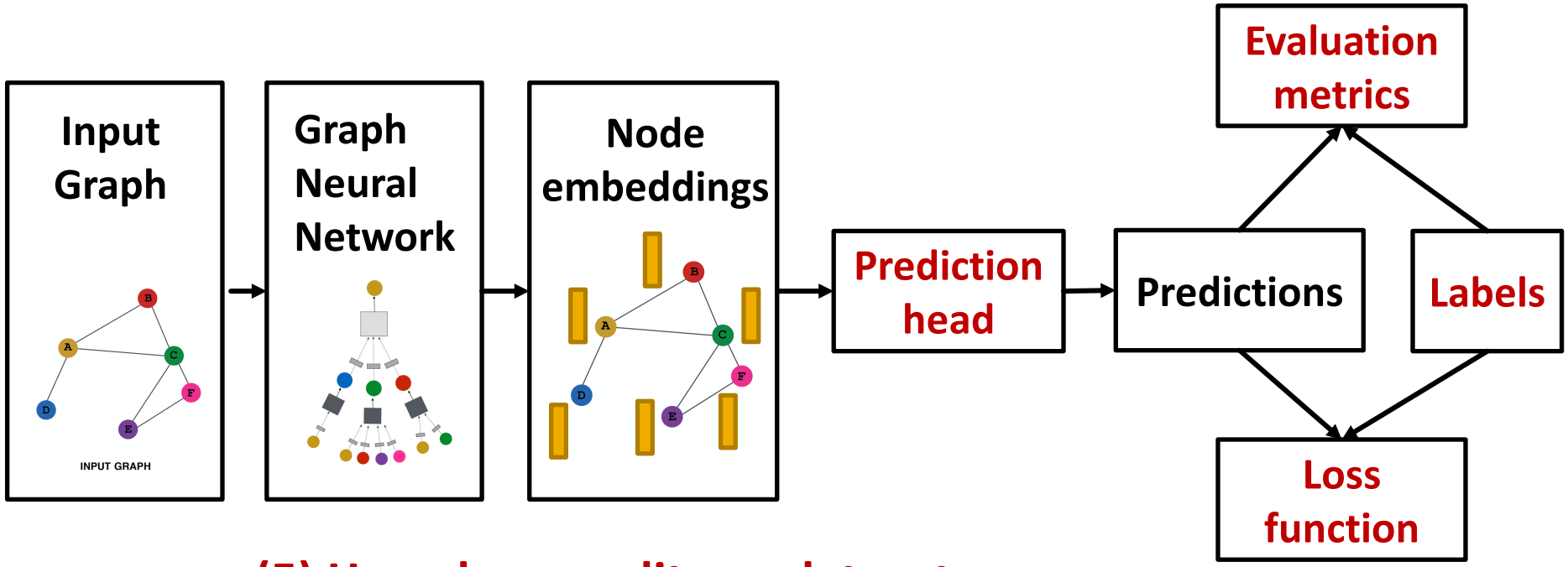
Content Credit: Wikipedia

- **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

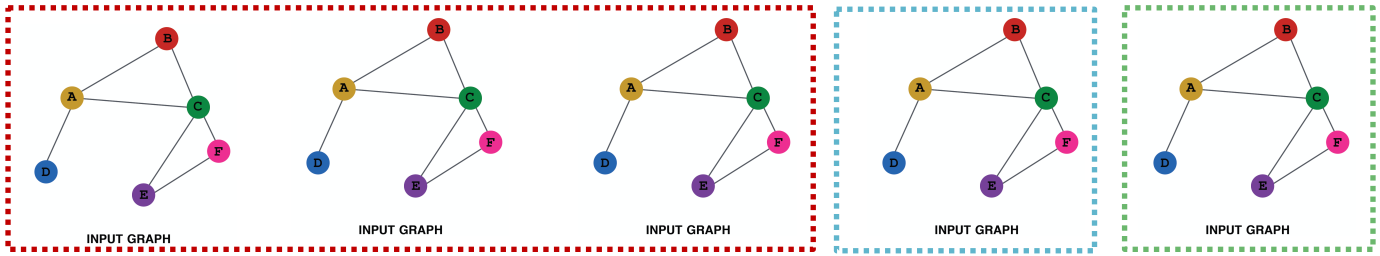
Stanford CS224W: Setting-up GNN Prediction Tasks

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**

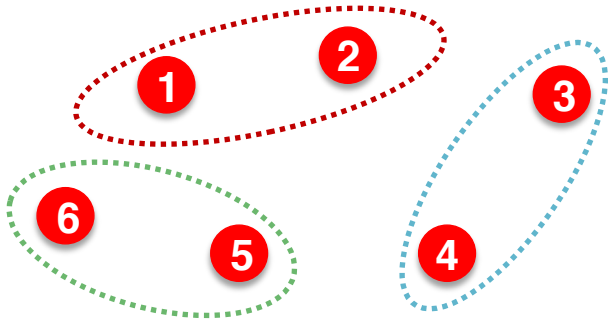
Why Splitting Graphs is Special

- **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



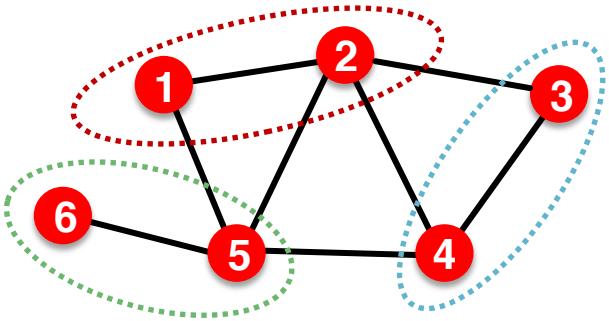
Why Splitting Graphs is Special

- **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?**

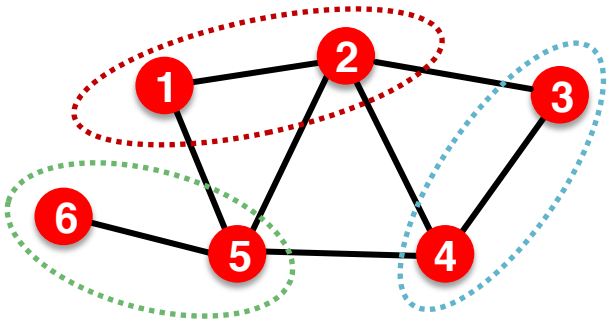
Why Splitting Graphs is Special

- **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



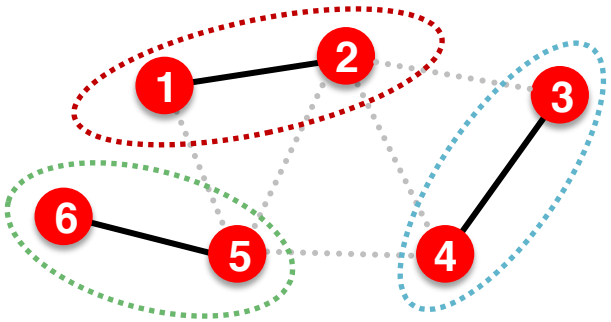
Why Splitting Graphs is Special

- **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

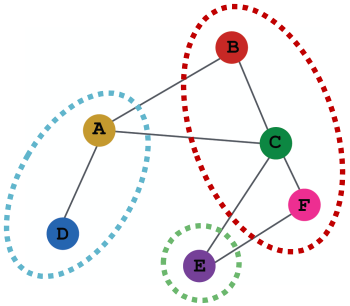


Transductive / Inductive Settings

- **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

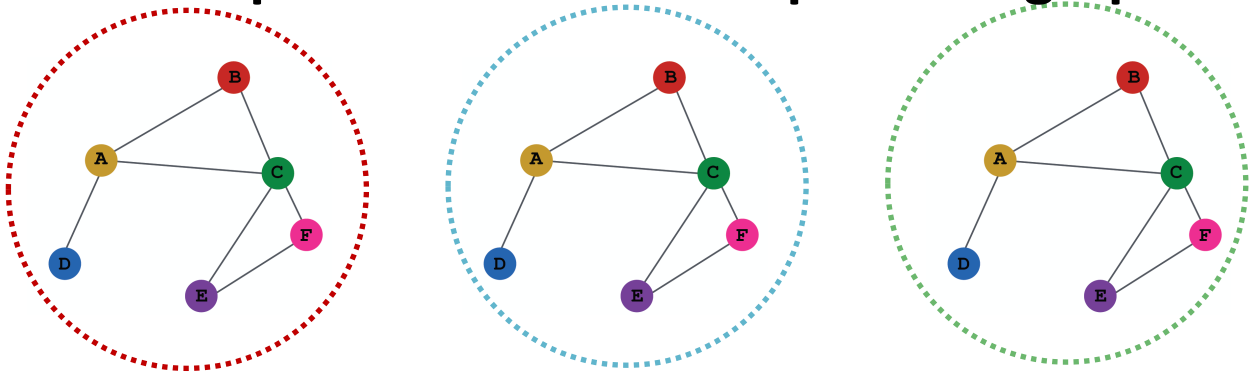


Training

Validation

Test

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



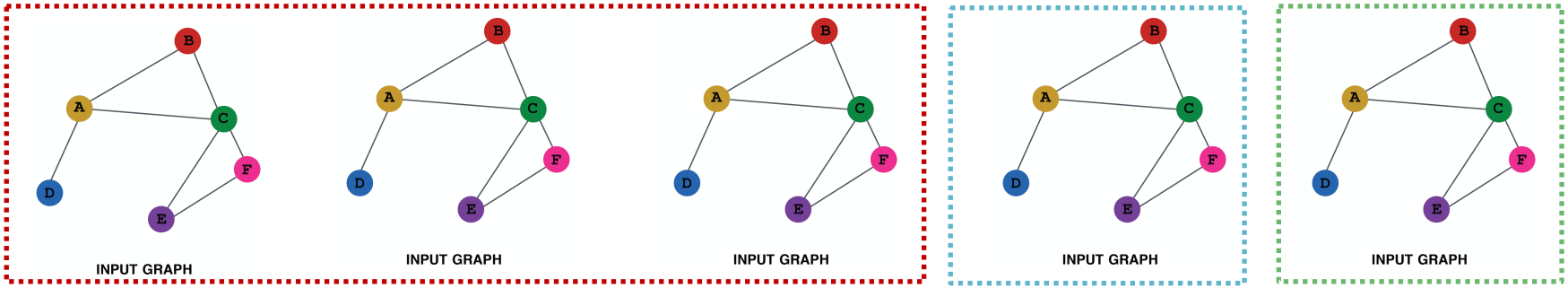
Training

Validation

Test

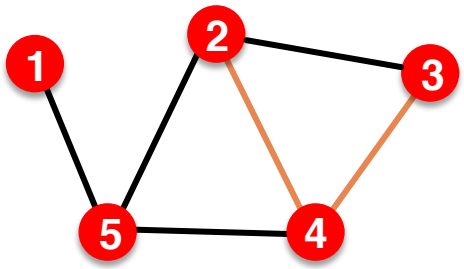
Example: Graph Classification

- 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).

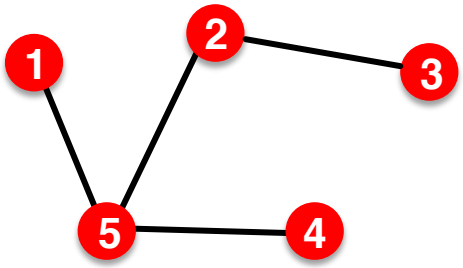


Example: Link Prediction

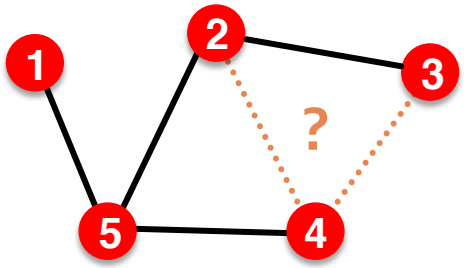
- **Goal of link prediction: predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and the **let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



Original graph

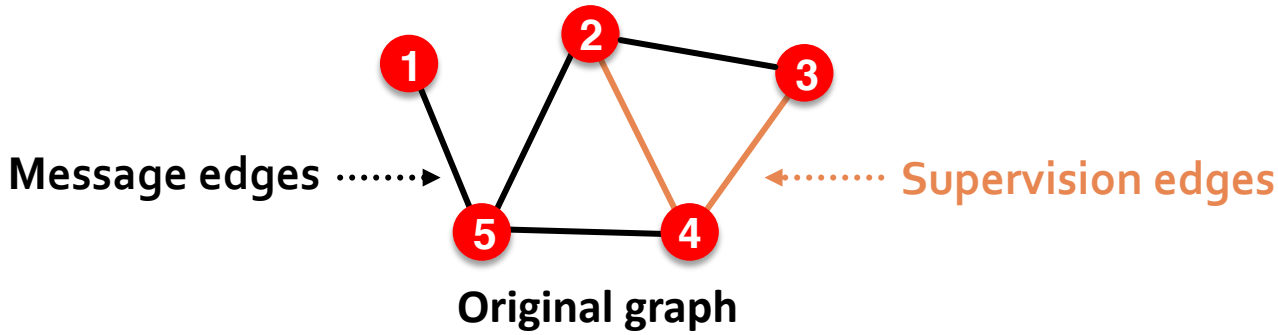


Input graph to GNN



Predictions made by GNN

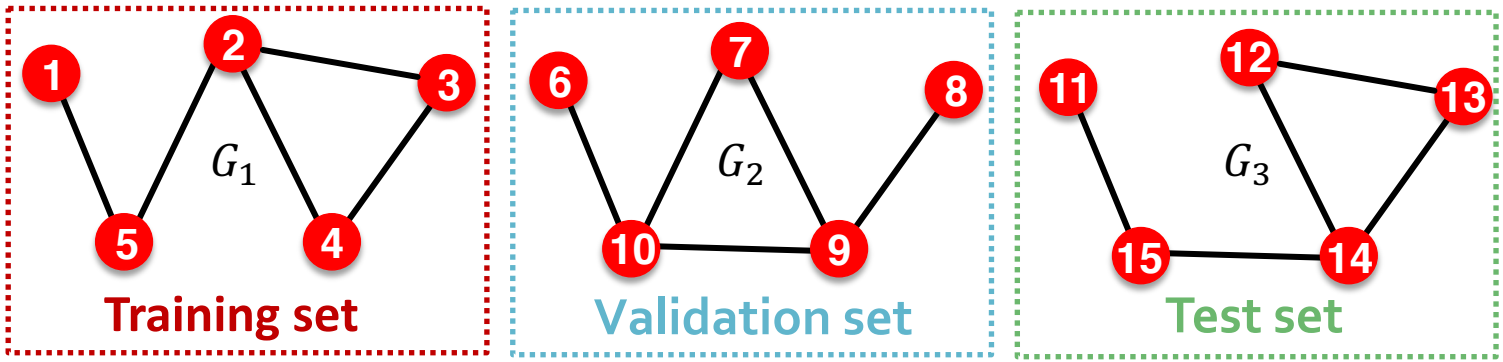
Setting up Link Prediction



- 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!

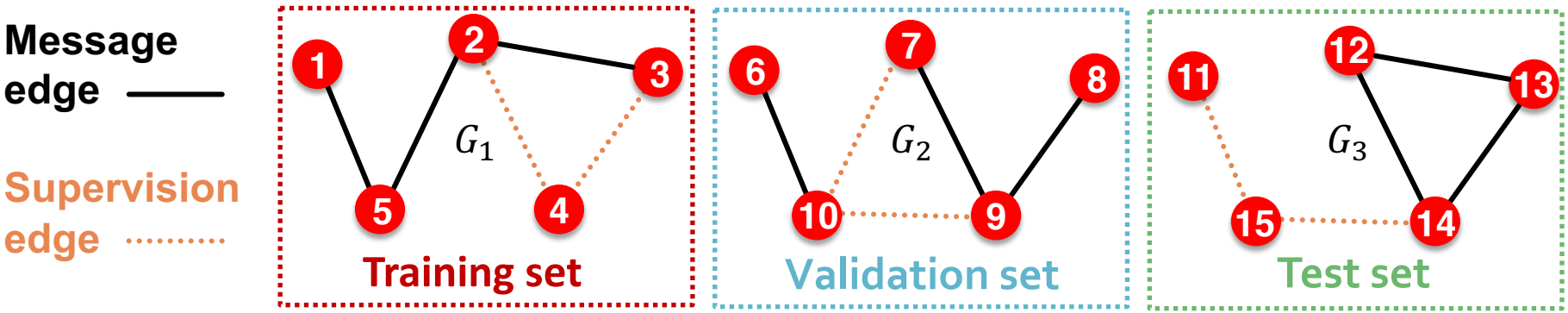
Setting up Link Prediction

- 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



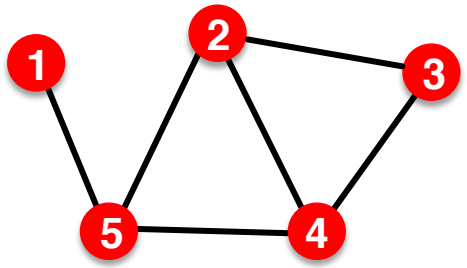
Setting up Link Prediction

- **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



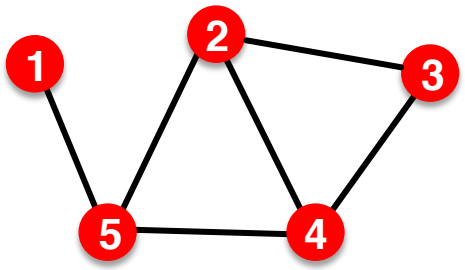
Setting up Link Prediction

- **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

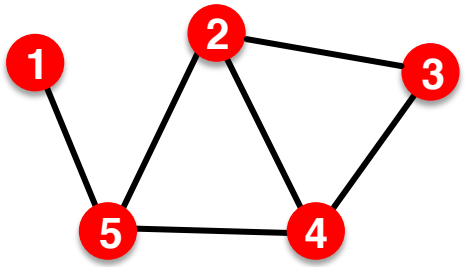
- **Option 2: Transductive link prediction split:**
  - **By definition of "transductive", the entire graph can be observed in all dataset splits**
    - **But since edges are both part of graph structure and the supervision, we need to hold out validation / test edges**
    - **To train the training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



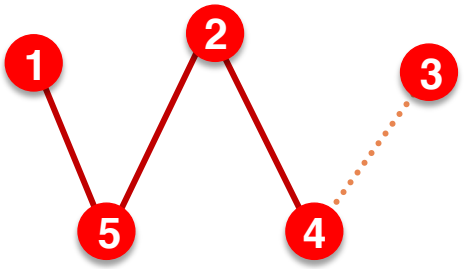
  - **Next:** we will show the exact settings

Setting up Link Prediction

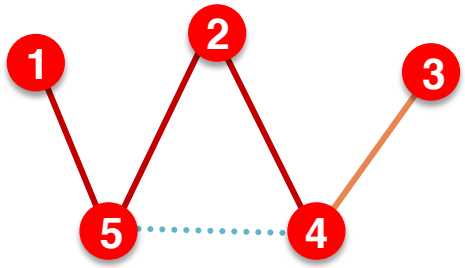
- **Option 2: Transductive link prediction split:**



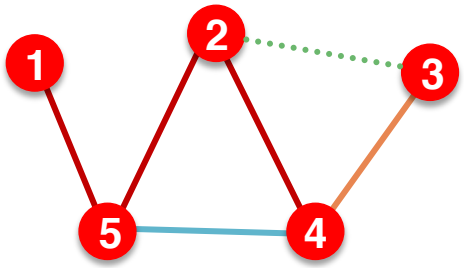
The original graph



(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**

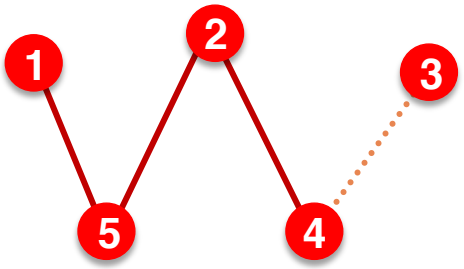


(3) At test time:
Use **training message edges & training supervision edges & validation edges** to predict **test edges**

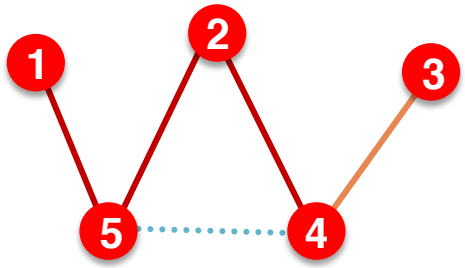
Setting up Link Prediction

- **Option 2: Transductive link prediction split:**

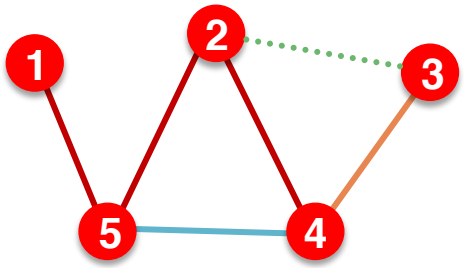
  Why do we use growing number of edges?
  After training, supervision edges are known to GNN.
  Therefore, an ideal model should use supervision edges in message passing at validation time.
  The same applies to the test time.



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



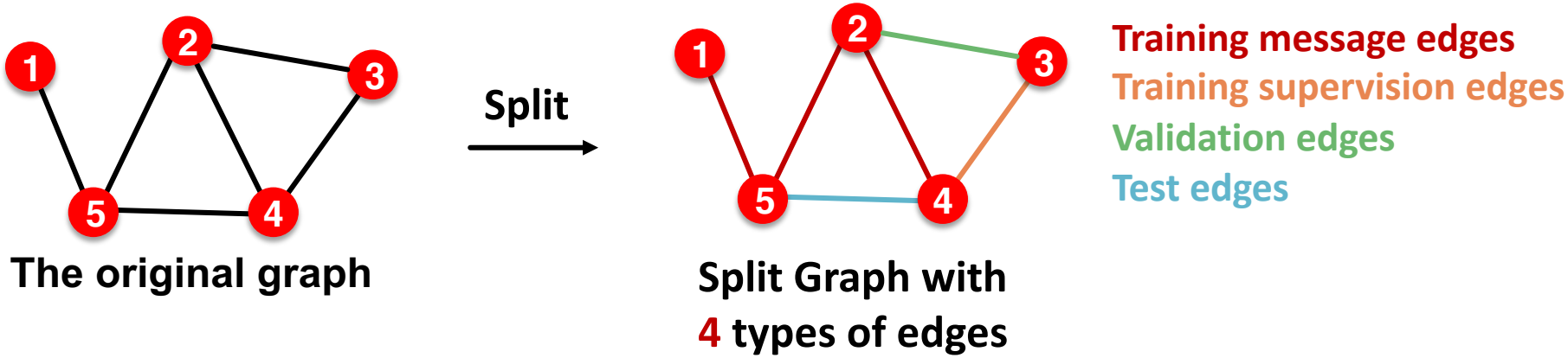
(2) At validation time:
Use **training message edges & training supervision edges** to predict **validation edges**



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

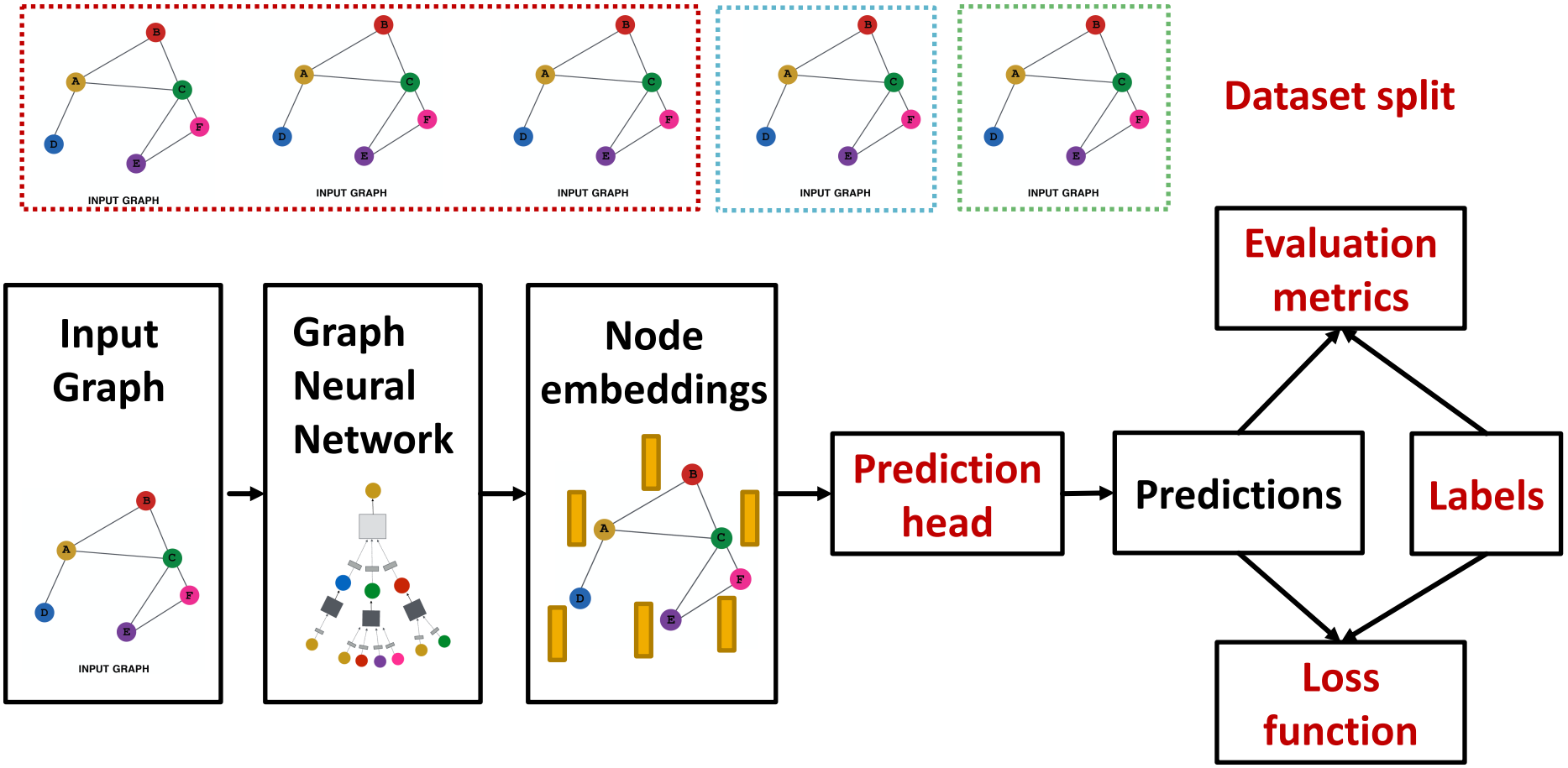
Setting up Link Prediction

- **Summary: Transductive link prediction split:**



  - **Note:** Link prediction settings are tricky and complex. You may find papers do link prediction differently. But if you follow our reasoning steps, **this should be the right way to implement link prediction**
  - Luckily, we have full support in **DeepSNAP 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 perspective for GNNs**
  - **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