Online Social Networks and Media

Graph ML

Graph Machine Learning

Outline

Part I: Introduction, Traditional ML

Part II: Graph Embeddings

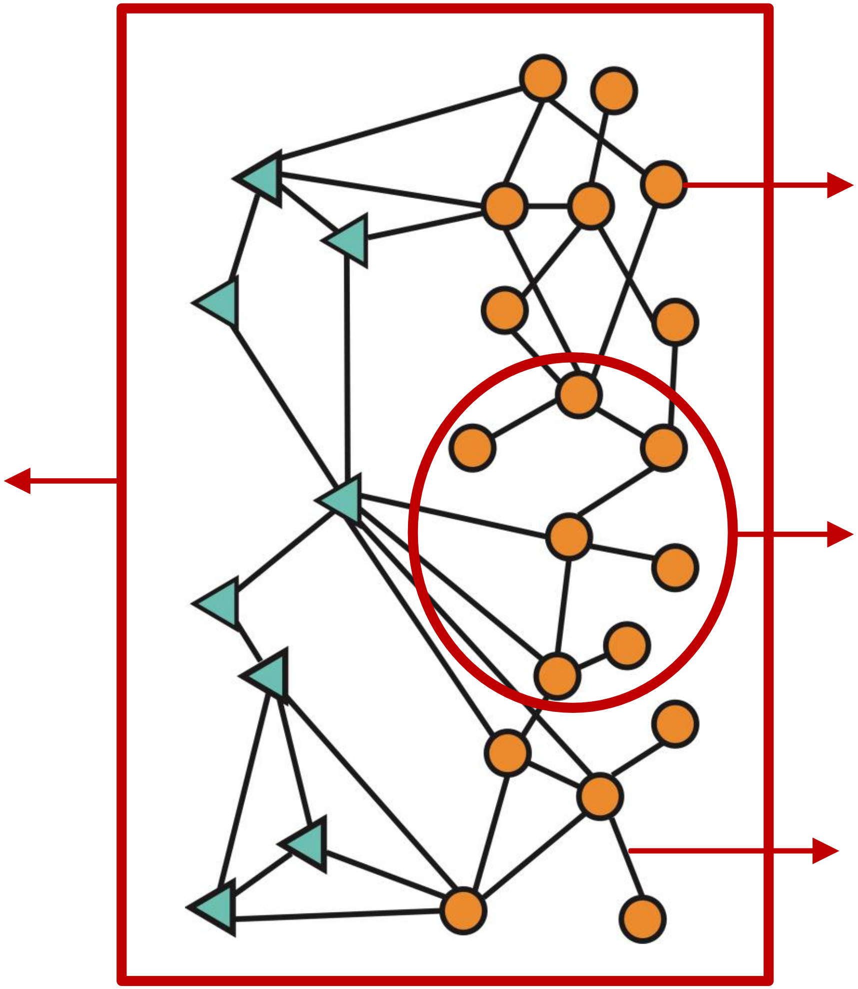
Part III: GNNs

Part IV (if time permits): Knowledge Graphs

Slides used based on:

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

Types of ML tasks in graphs



Node level

Graph-level prediction, Graph generation

Community (subgraph) level

Edge (link) level

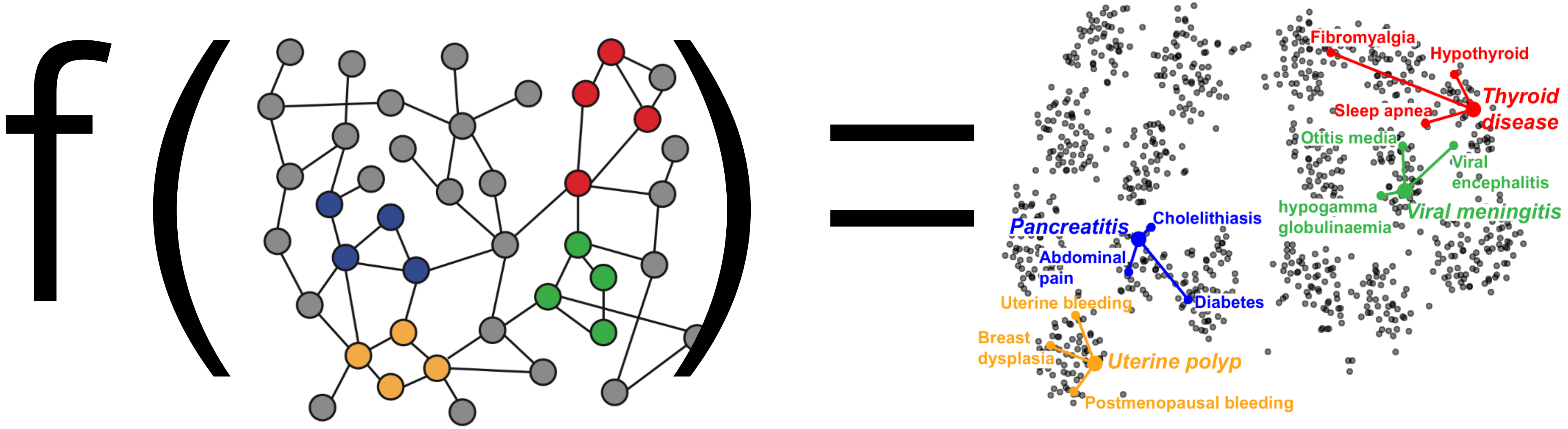
Example Tasks

Tasks we will be able to solve:

- Node classification
  - Predict the 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

Recap: Node Embeddings

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



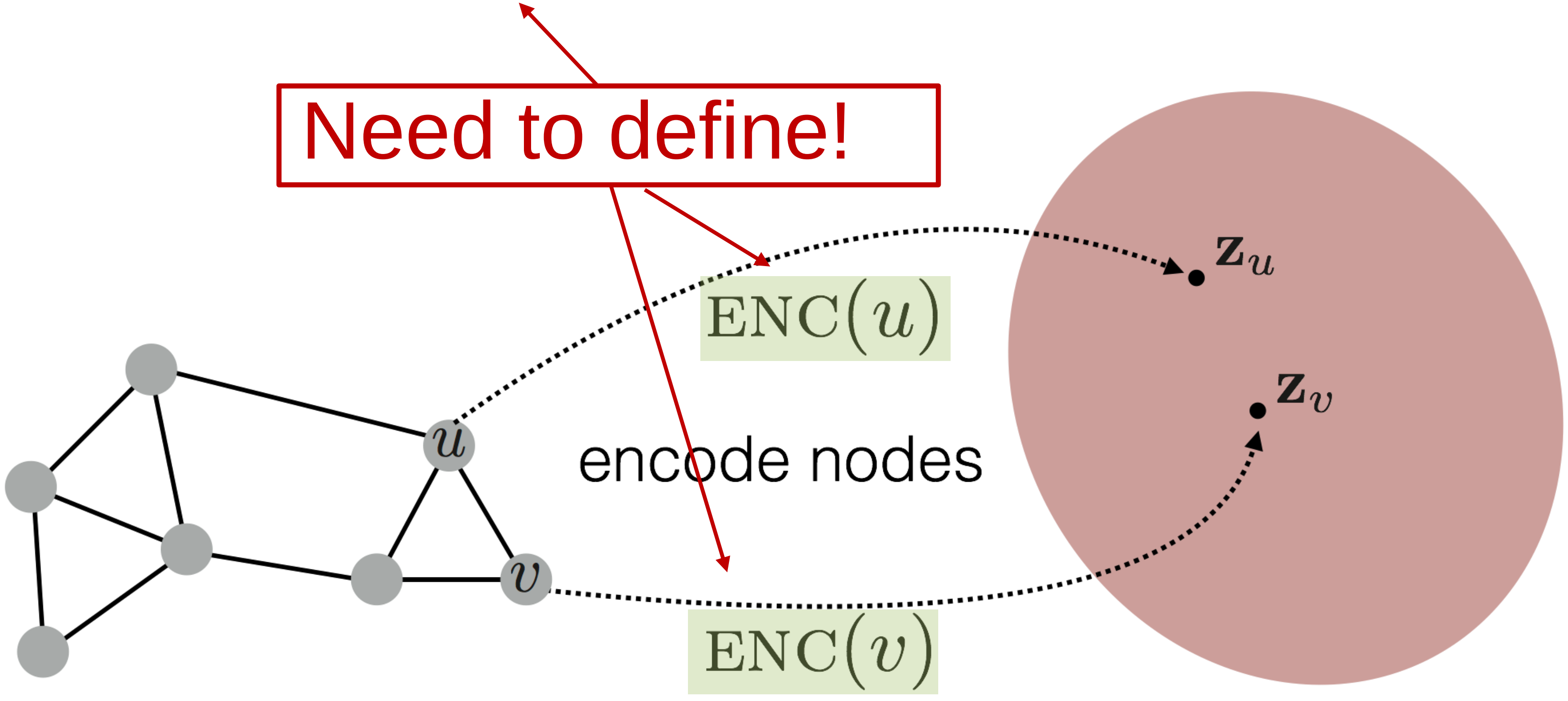
Input graph

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!

$\mathrm{ENC}(u)$

encode nodes

$\mathrm{ENC}(v)$

Input network

d-dimensional embedding space

Recap: Two Key Components

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

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

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

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

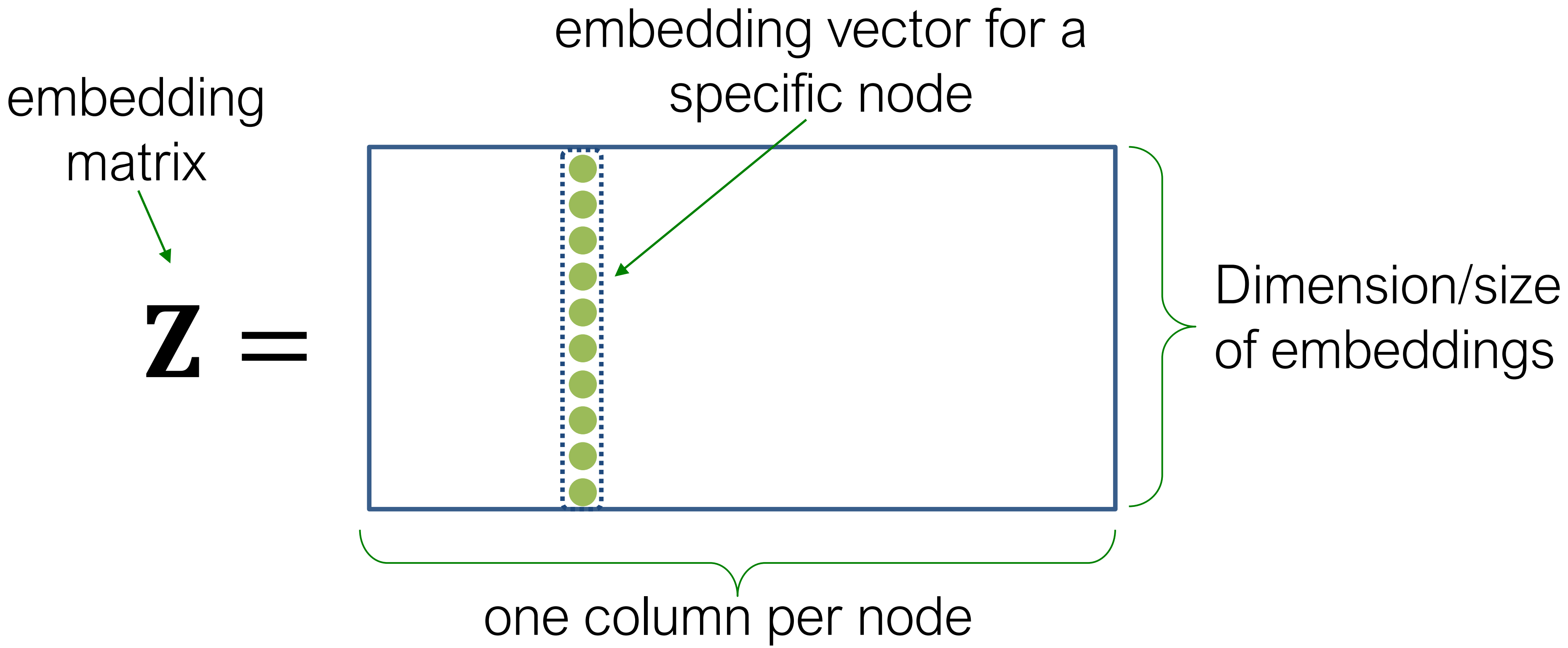
Decoder

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**


embedding vector for a specific node
embedding matrix
$\mathbf{Z} =$
Dimension/size of embeddings
one column per node

Recap: Shallow Encoders

Limitations of shallow embedding methods:

- **$O(|V|d)$ 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**:
  - Nodes in many graphs have features that we can and should leverage

Deep Graph Encoders

- Deep learning methods based on **graph neural networks (GNNs):**

$$\mathrm{ENC}(v) =$$

multiple layers of non-linear transformations based on graph structure

Note: All these deep encoders can be combined with node similarity functions defined in previous lectures

Part III:

General Framework

A single GNN layer: Aggregation and Message

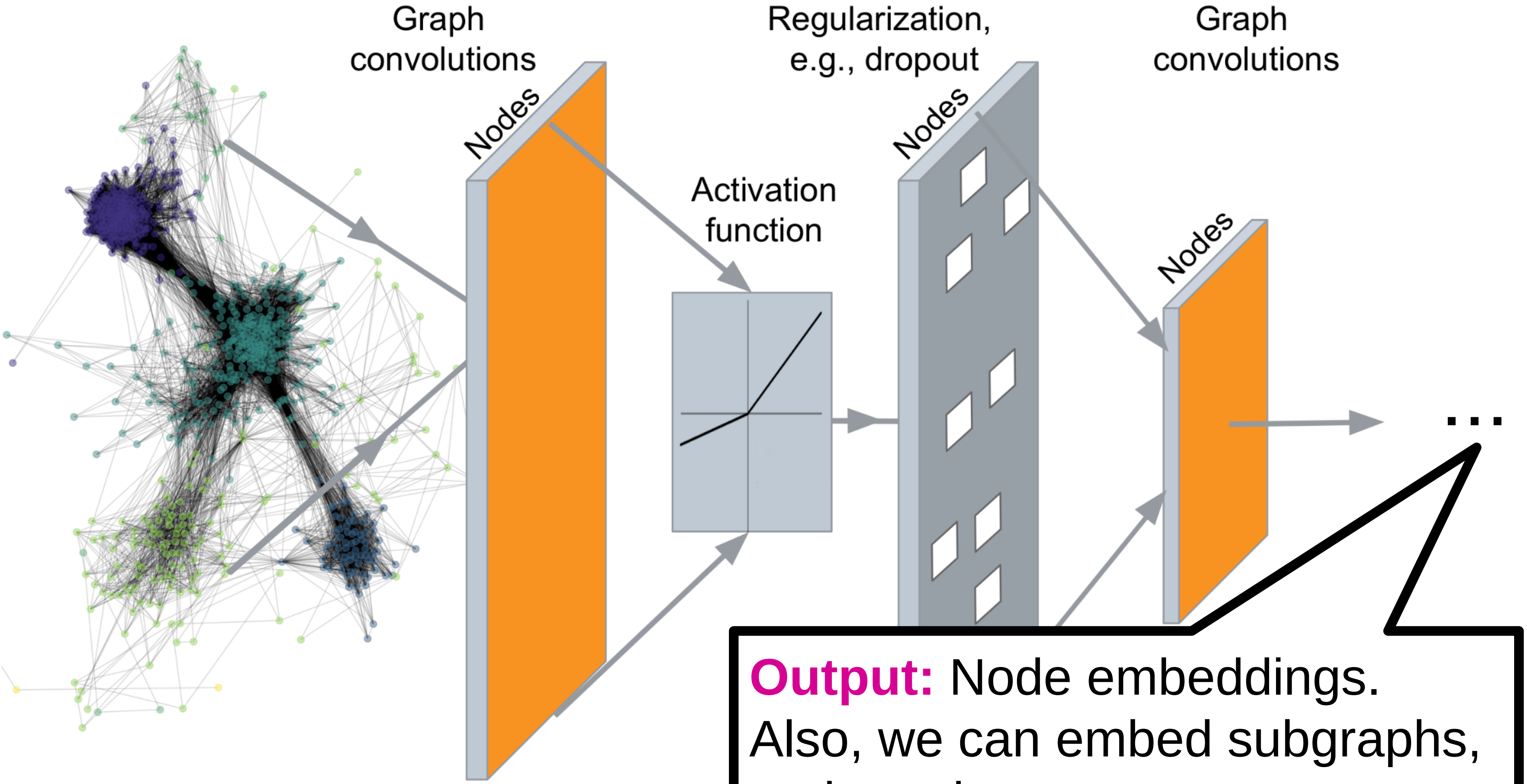
Layer connectivity: Stacking

Graph manipulations

Learning objectives

OVERVIEW AND GENERAL FRAMEWORK

Deep Graph Encoders



Graph convolutions

Nodes

Activation function

Regularization, e.g., dropout

Nodes

Graph convolutions

Nodes

...

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

Basics of Deep Learning

- **Loss function:**

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

- f can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN)

- 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 $\mathcal{L}$ for Θ over many iterations.

Setup

Assume we have a graph $\boldsymbol{G}$:

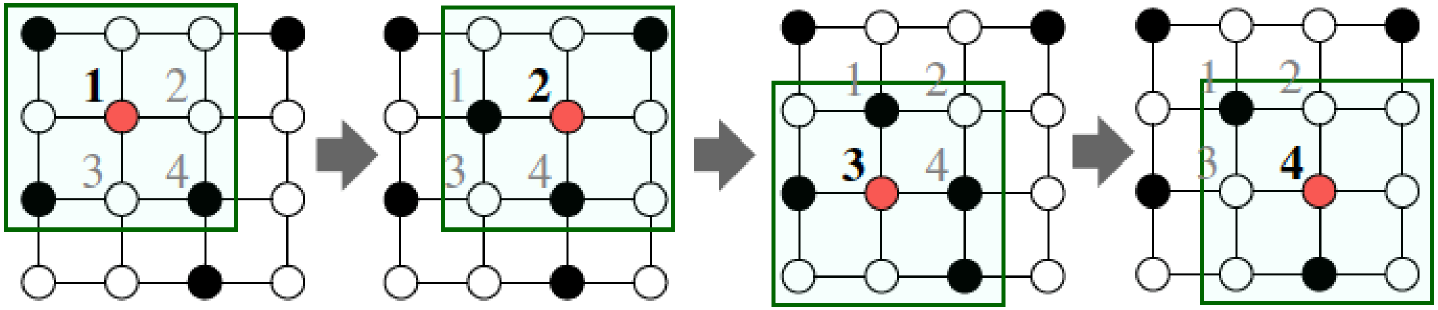
- V is the set of nodes
- $\boldsymbol{A}$ is the adjacency matrix (assume binary)
- v: a node in V; $N(v)$: the set of neighbors of v.

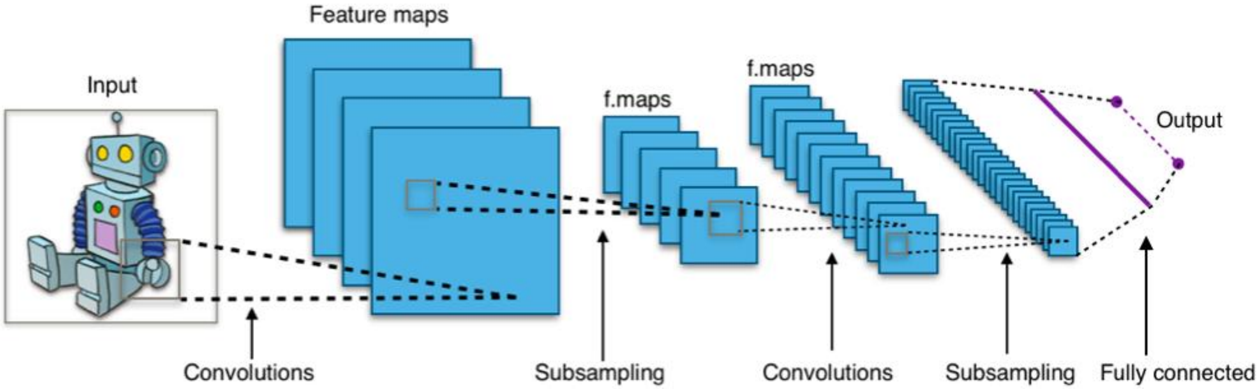
$\boldsymbol{X} \in \mathbb{R}^{|V| \times \boldsymbol{m}}$ is a matrix of **node features**

- **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]

Idea: Convolutional Networks

CNN on an image:





Nice description of CNNs: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

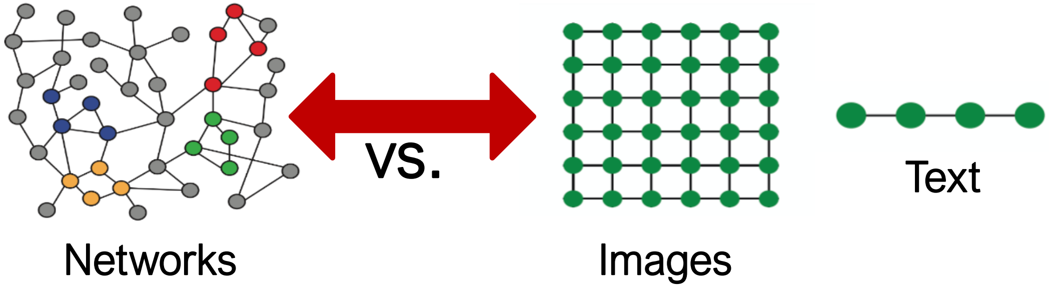
Can we generalize convolutions beyond simple lattices?

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

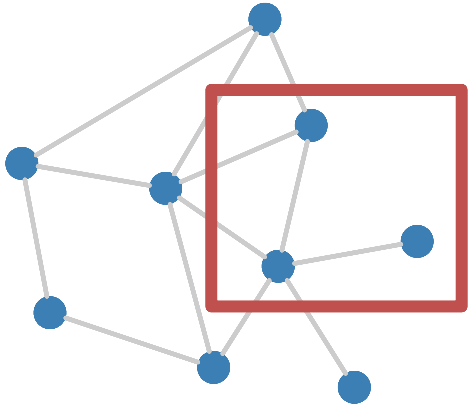
Why is it hard?

Graphs are far more complex!

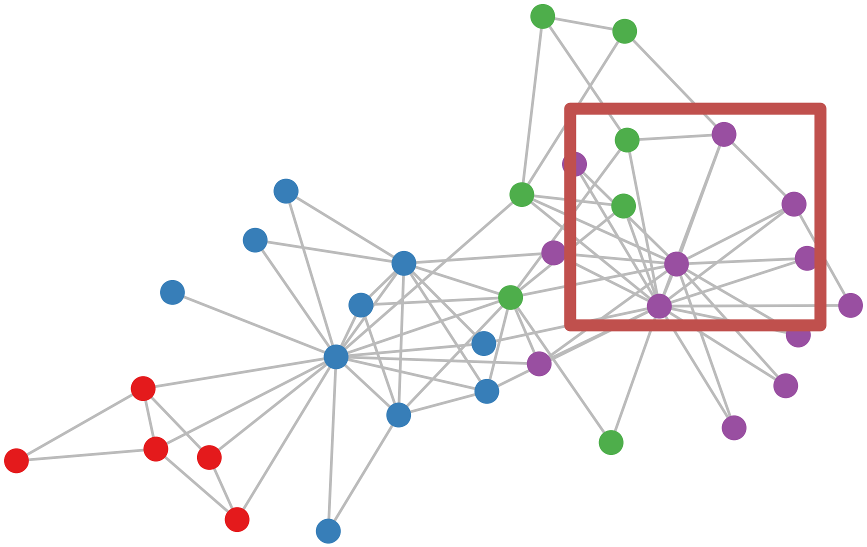
arbitrary size and complex topological structure



Graphs look like this:



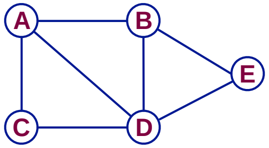
or this:



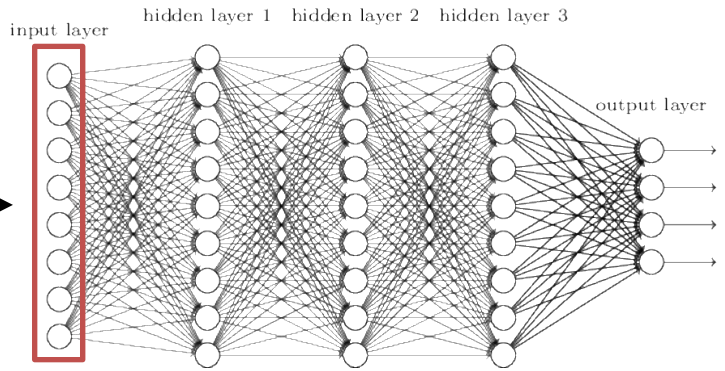
- No fixed notion of (spatial) locality or sliding window on the graph
- No fixed node ordering or reference point
- Often dynamic and have multimodal features

A Naïve Approach

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



$$\begin{array}{c|ccccccc}
 & A & B & C & D & E & \multicolumn{2}{c}{Feat} \\
\hline
A & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
B & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
C & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
D & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
E & 0 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}$$



?

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

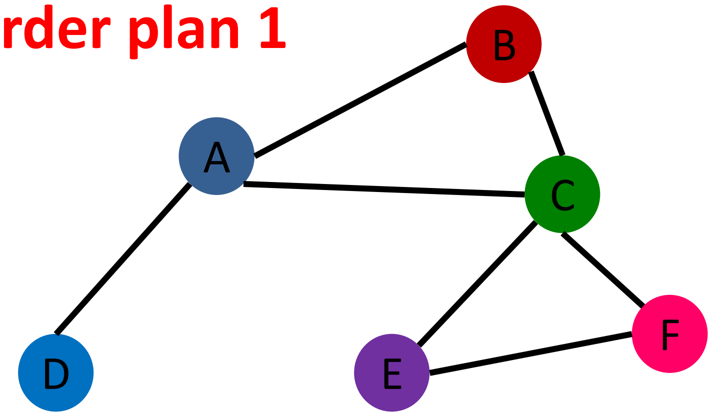
Permutation Invariance

- **Graph does not have a canonical order of the nodes!**
- We can have many different order plans.

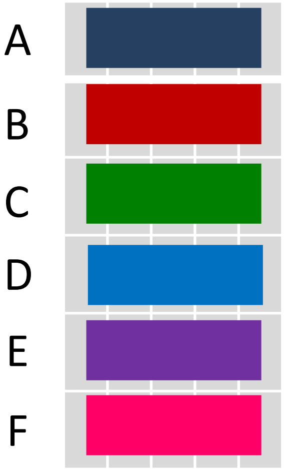
Permutation Invariance

- **Graph does not have a canonical order of the nodes!**

Order plan 1



Node features X_1



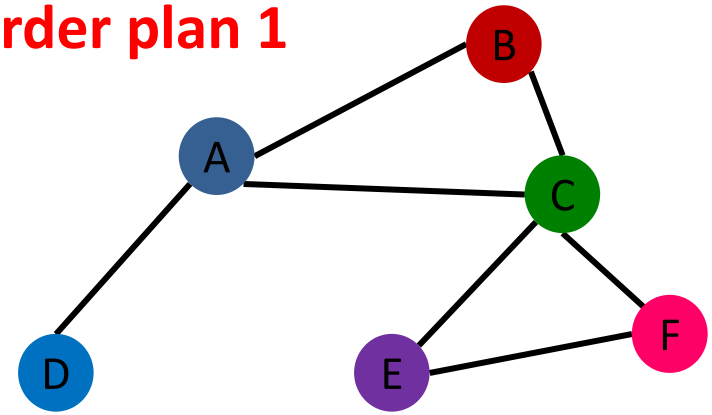
Adjacency matrix A_1

	A	B	C	D	E	F
A	⬛	🟦	🟦	🟦	⬛	⬛
B	🟦	⬛	🟦	⬛	⬛	⬛
C	🟦	🟦	⬛	⬛	🟦	🟦
D	🟦	⬛	⬛	⬛	⬛	⬛
E	⬛	⬛	🟦	⬛	⬛	🟦
F	⬛	⬛	🟦	⬛	🟦	⬛

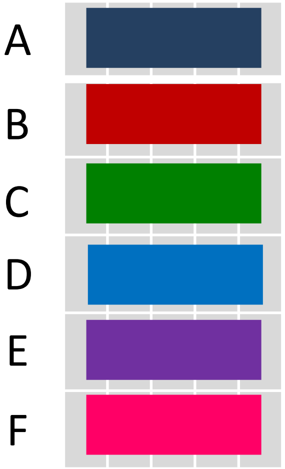
Permutation Invariance

- **Graph does not have a canonical order of the nodes!**

Order plan 1



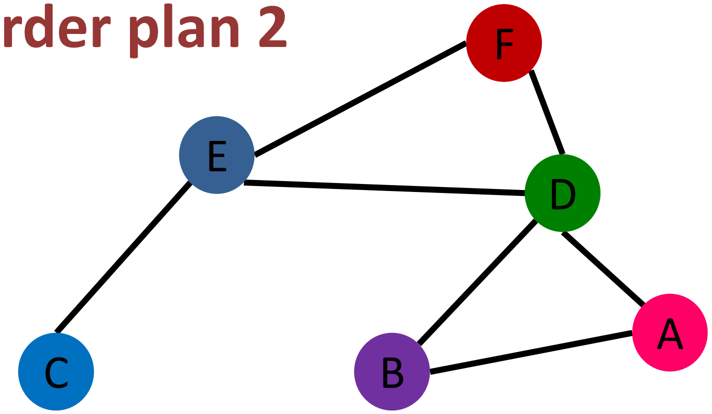
Node features X_1



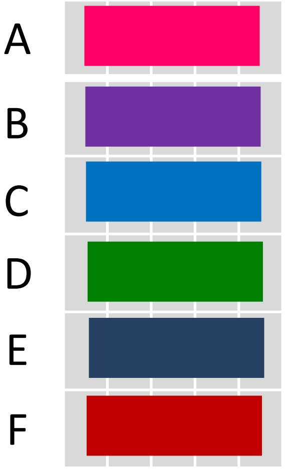
Adjacency matrix A_1

	A	B	C	D	E	F
A	⬛	🟦	🟦	🟦	⬛	⬛
B	🟦	⬛	🟦	⬛	⬛	⬛
C	🟦	🟦	⬛	⬛	🟦	🟦
D	🟦	⬛	⬛	⬛	⬛	⬛
E	⬛	⬛	🟦	⬛	⬛	🟦
F	⬛	⬛	🟦	⬛	🟦	⬛

Order plan 2



Node features X_2



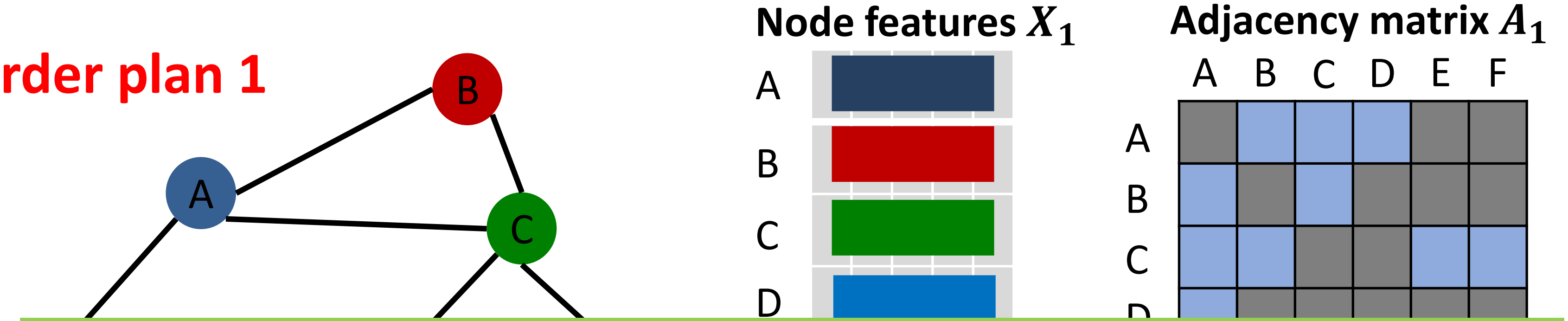
Adjacency matrix A_2

	A	B	C	D	E	F
A	⬛	🟦	⬛	🟦	⬛	⬛
B	🟦	⬛	⬛	🟦	⬛	⬛
C	⬛	⬛	⬛	⬛	🟦	⬛
D	🟦	🟦	⬛	⬛	🟦	🟦
E	⬛	⬛	🟦	🟦	⬛	🟦
F	⬛	⬛	⬛	🟦	🟦	⬛

Permutation Invariance

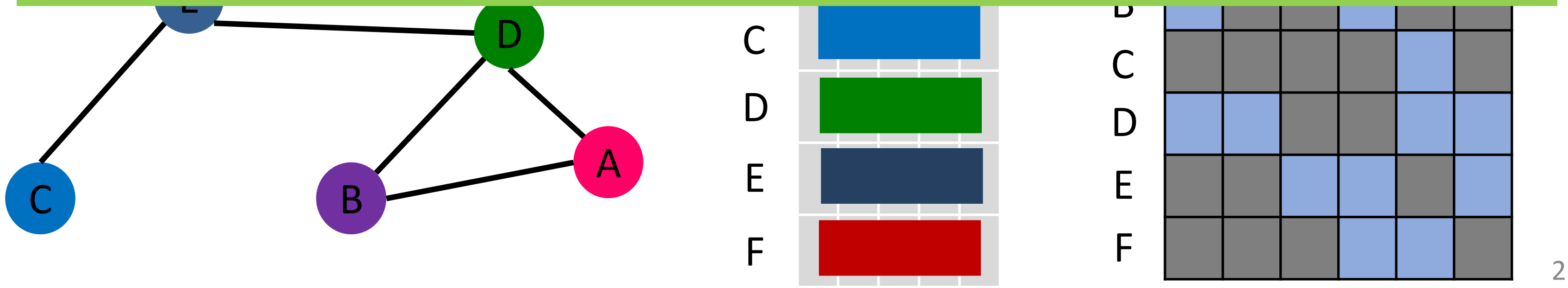
- **Graph does not have a canonical order of the nodes!**

Order plan 1



Or

Graph and node representations should be the same for Order plan 1 and Order plan 2



Invariance and Equivariance

- **Permutation-invariant**

$$f(A,X) = f\left(PAP^T, PX\right)$$

Permute the input, the output stays the same.

- **Permutation-equivariant**

$$Pf(A,X) = f\left(PAP^T, PX\right)$$

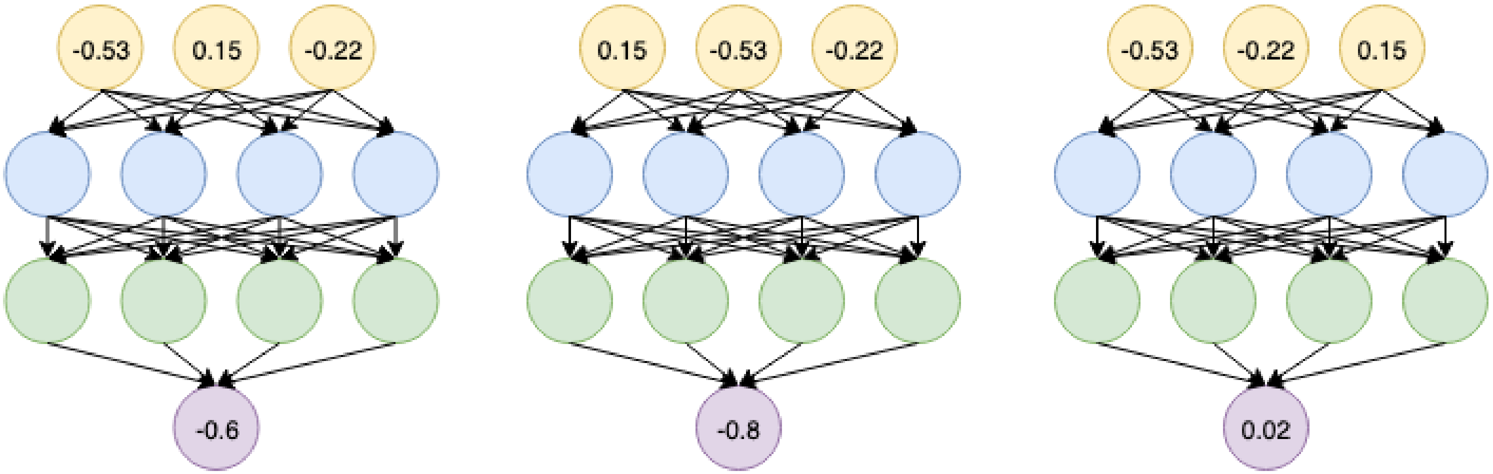
Permute the input, output also permutes accordingly.

Graph Neural Network Overview

Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

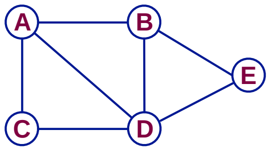
- **No**

Switching the order of the input leads to different outputs!

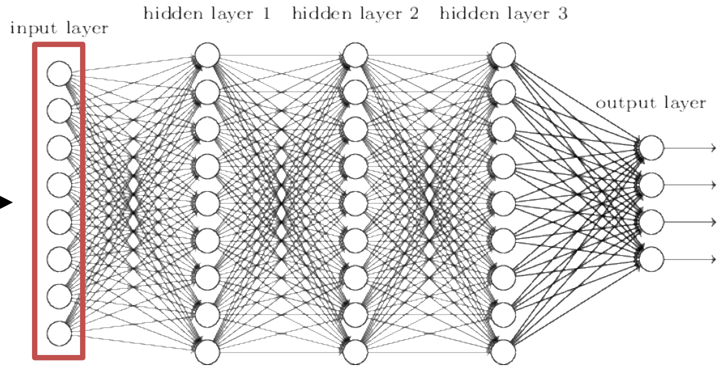


Graph Neural Network Overview

Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?



	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

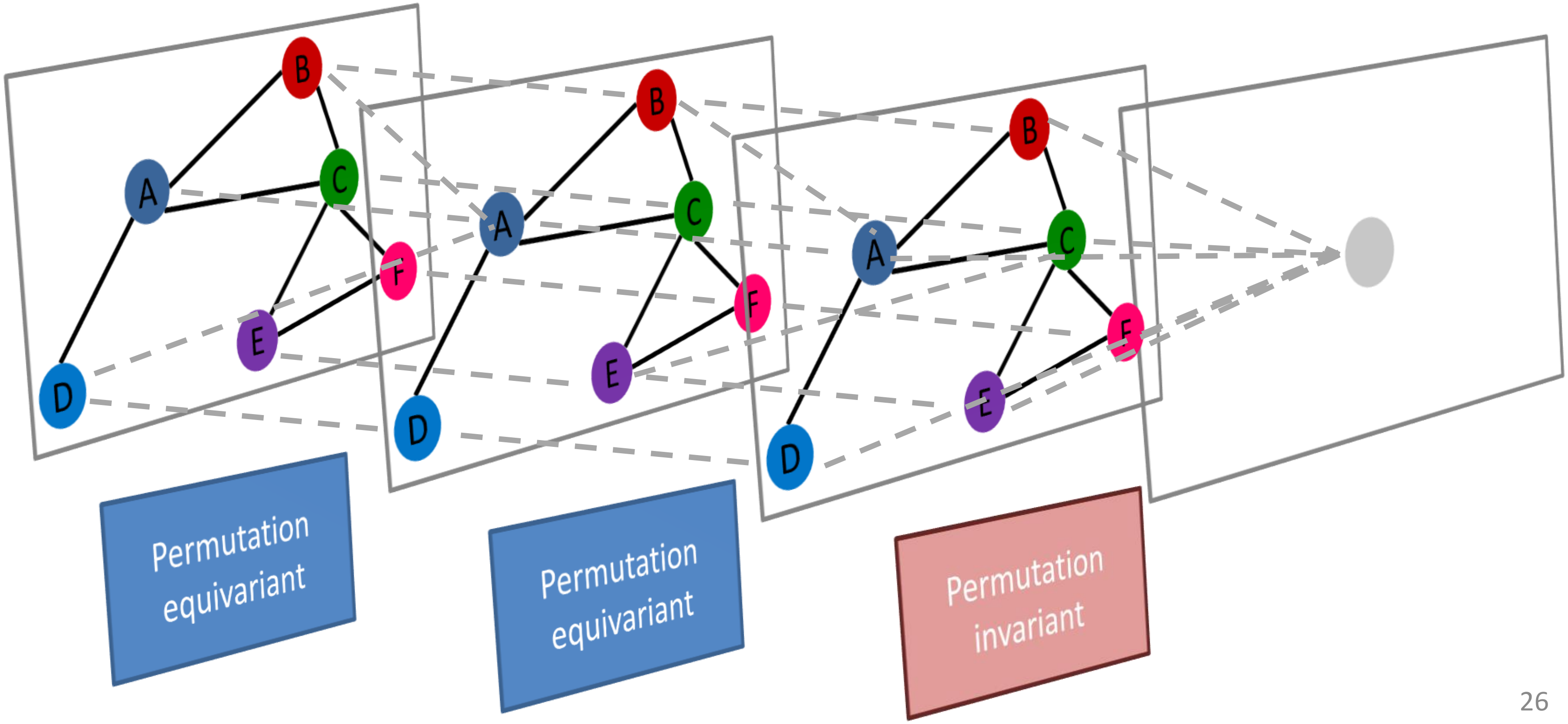


?

This explains why **the naïve MLP approach fails for graphs**!

Graph Neural Network Overview

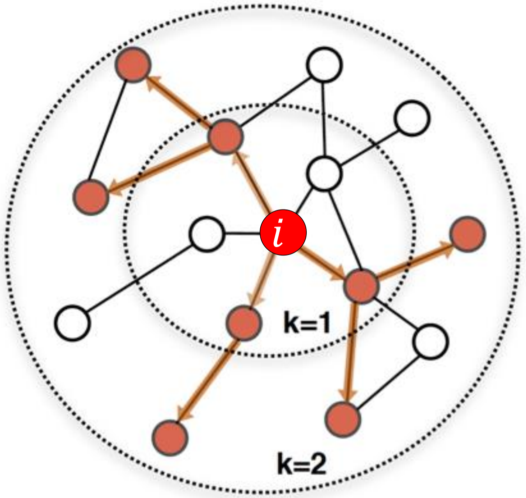
- **Graph neural networks consist of multiple permutation equivariant/invariant functions.**



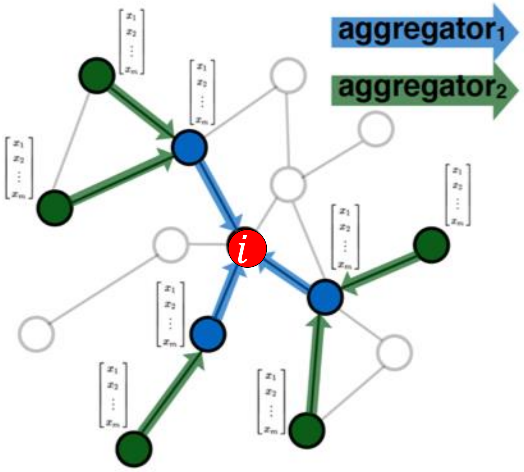
[Kipf and Welling, ICLR 2017]

Graph Convolutional Networks

Idea: The neighborhood of a node defines a computation graph



Determine node computation graph

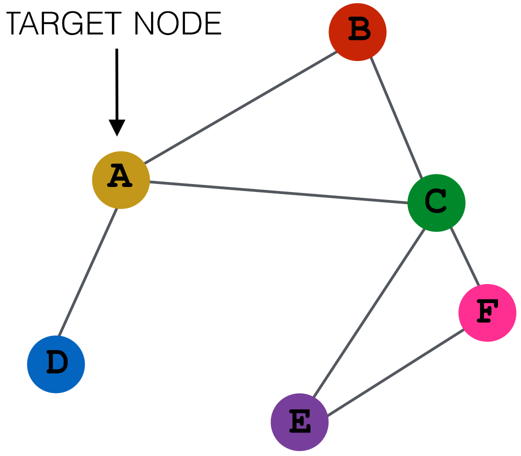


Propagate and transform information

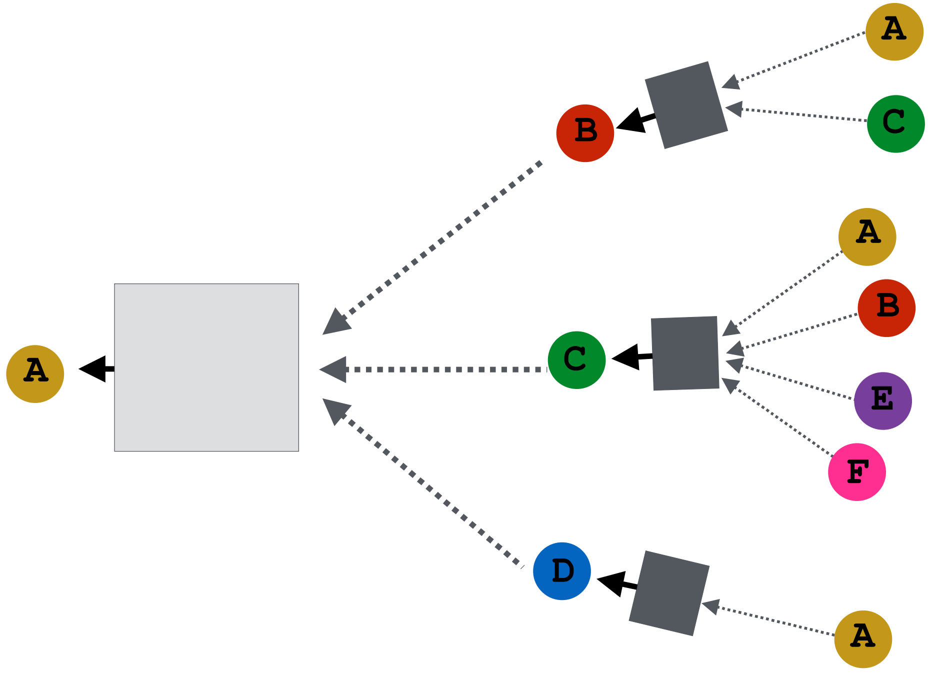
Learn how to propagate information across the graph to compute node features

Idea: Aggregate Neighbors

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

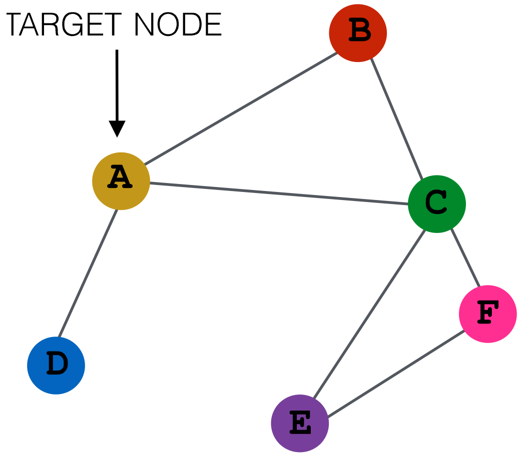


INPUT GRAPH

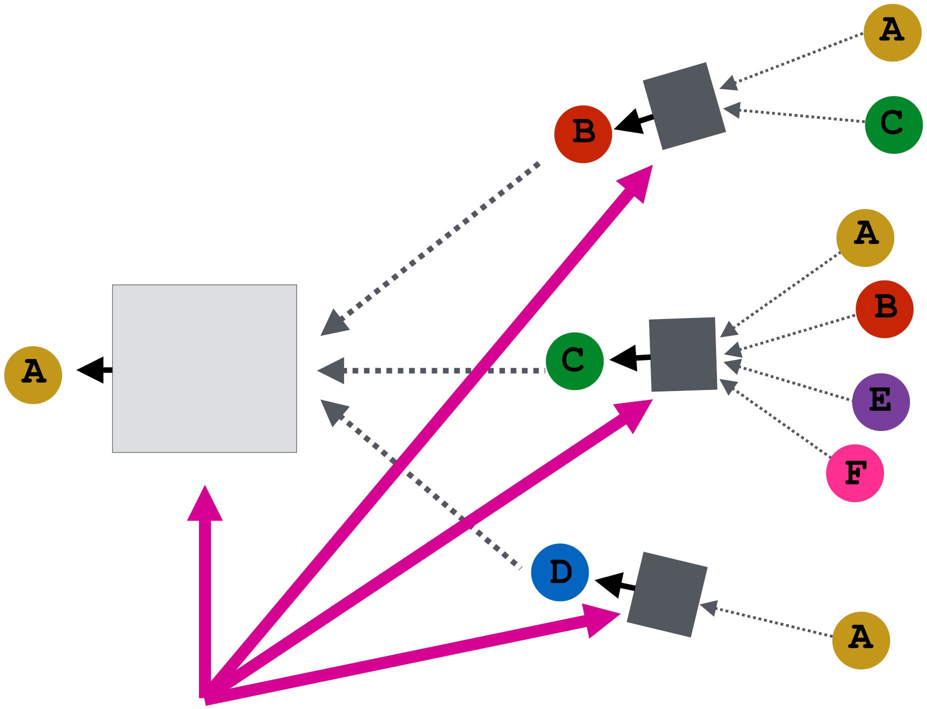


Idea: Aggregate Neighbors

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



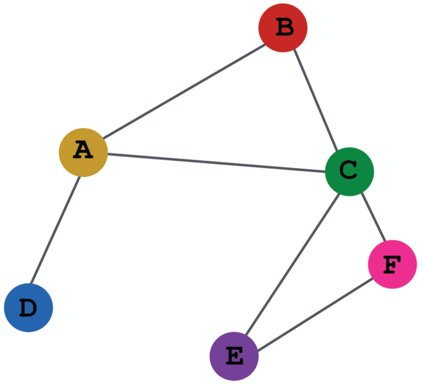
INPUT GRAPH



Neural networks

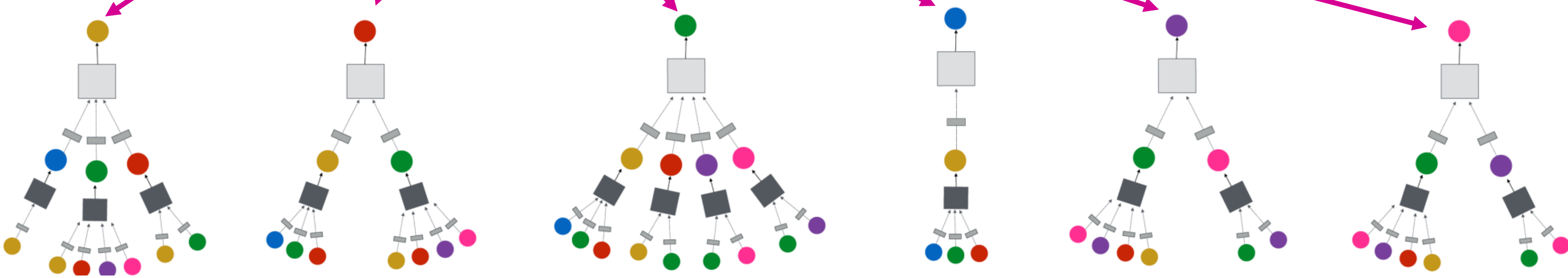
Idea: Aggregate Neighbors

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



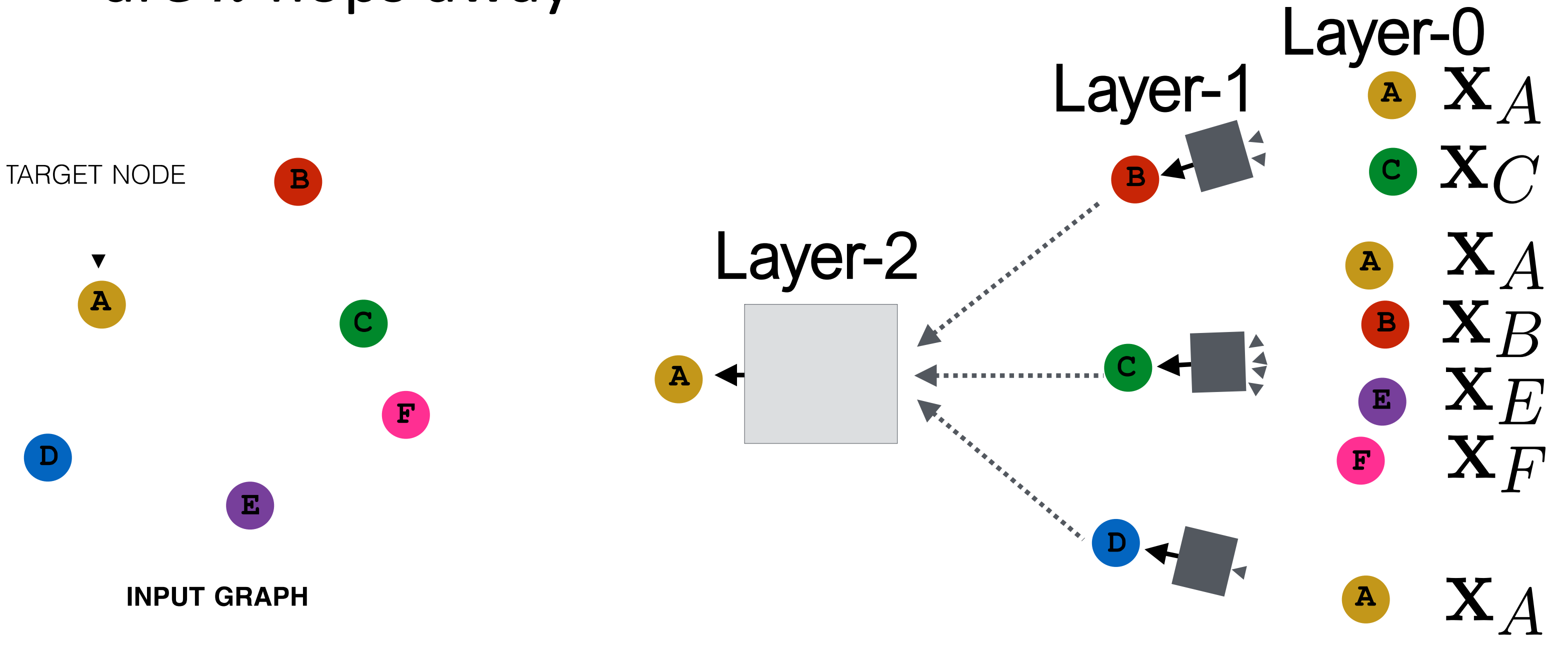
INPUT GRAPH

Every node defines a computation graph based on its neighborhood!



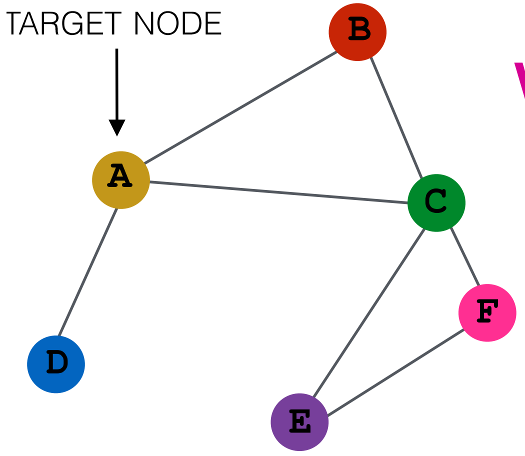
Deep Model: Many Layers

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



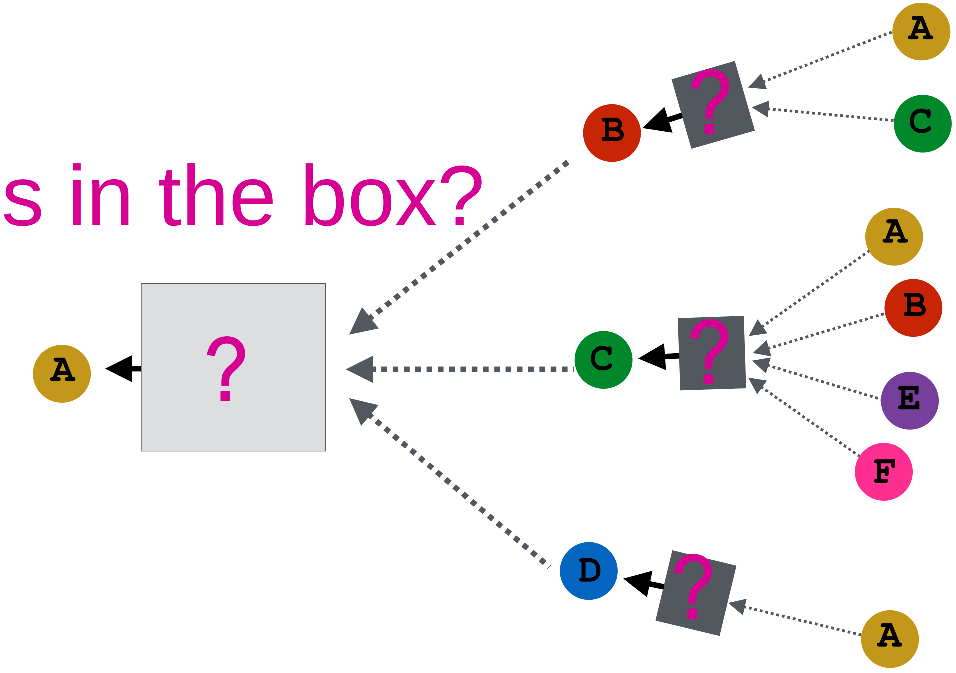
Neighborhood Aggregation

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



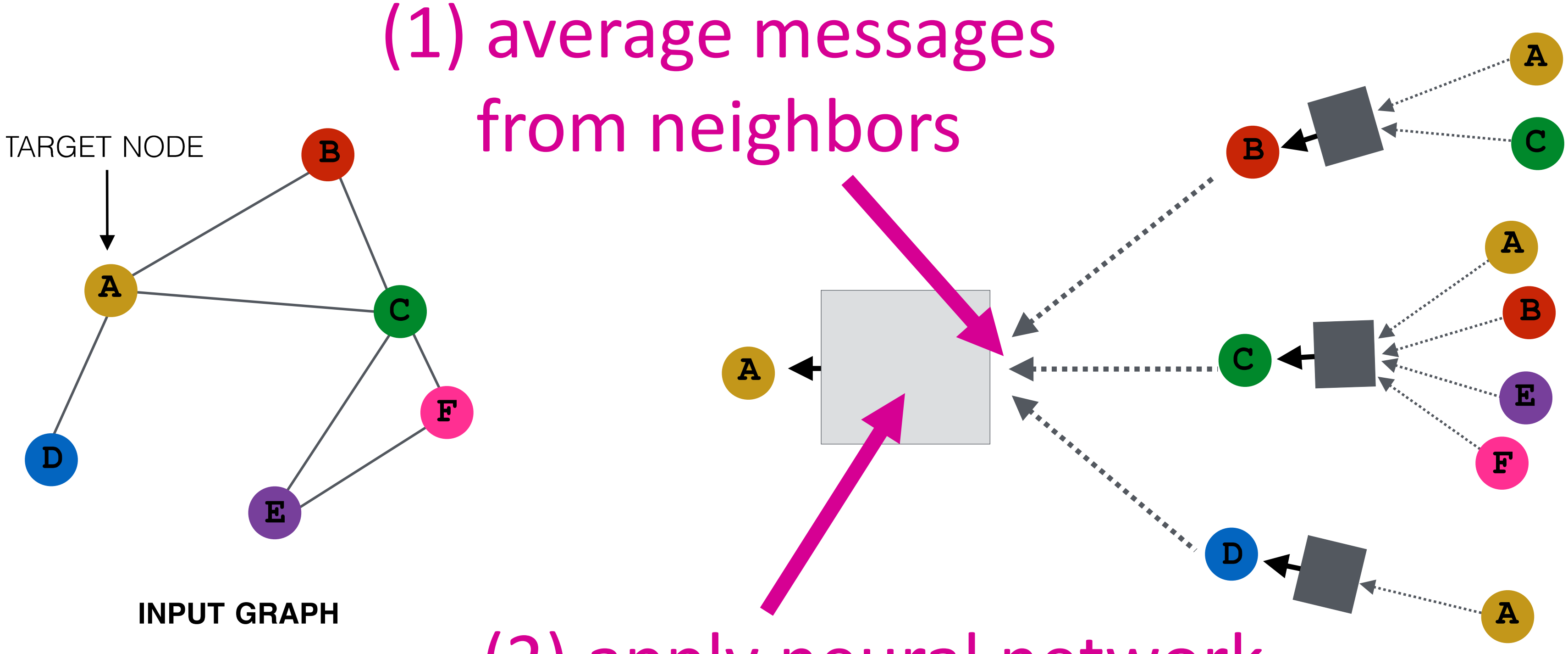
INPUT GRAPH

What is in the box?



Neighborhood Aggregation

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



(1) average messages from neighbors

(2) apply 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

$$\boxed{\mathrm{h}_v^0 = \mathrm{x}_v}$$

embedding of v at layer k

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

Total number of layers

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

Average of neighbor's previous layer embeddings

Embedding after *K* layers of neighborhood aggregation

Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.

Model Parameters

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

weight matrices are shared

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

$$\mathrm{h}_v^{(k+1)} = \sigma\left(\boxed{\mathrm{W}_k} \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|} + \boxed{\mathrm{B}_k}\mathrm{h}_v^{(k)}\right), \forall k \in \{0..K-1$$

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

Final node embedding

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

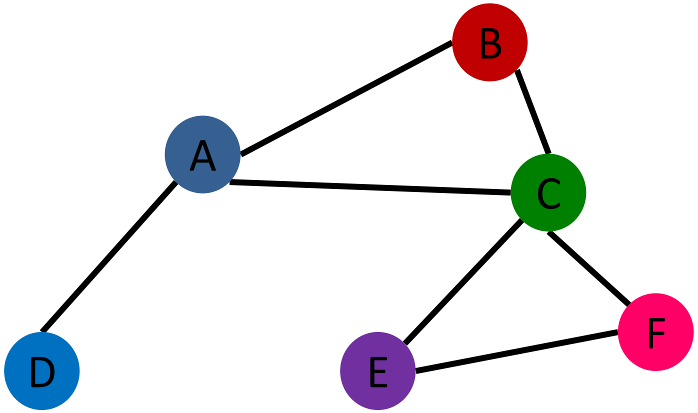
h_v^k: the hidden representation of node v at layer k

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

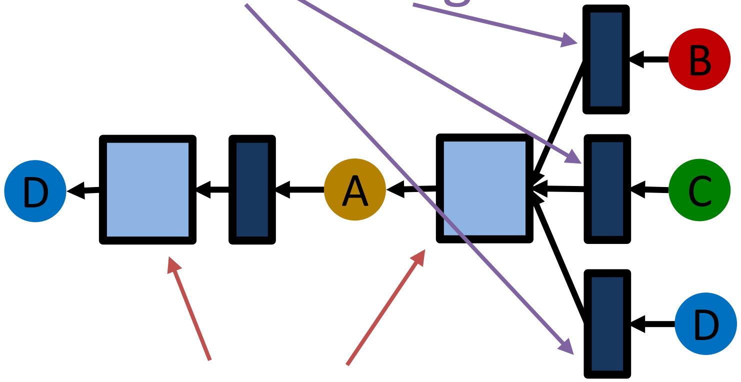
GCN: Invariance and Equivariance

What are the invariance and equivariance properties for a GCN?

- **Given a node**, the GCN that computes its embedding is **permutation invariant**



Target Node

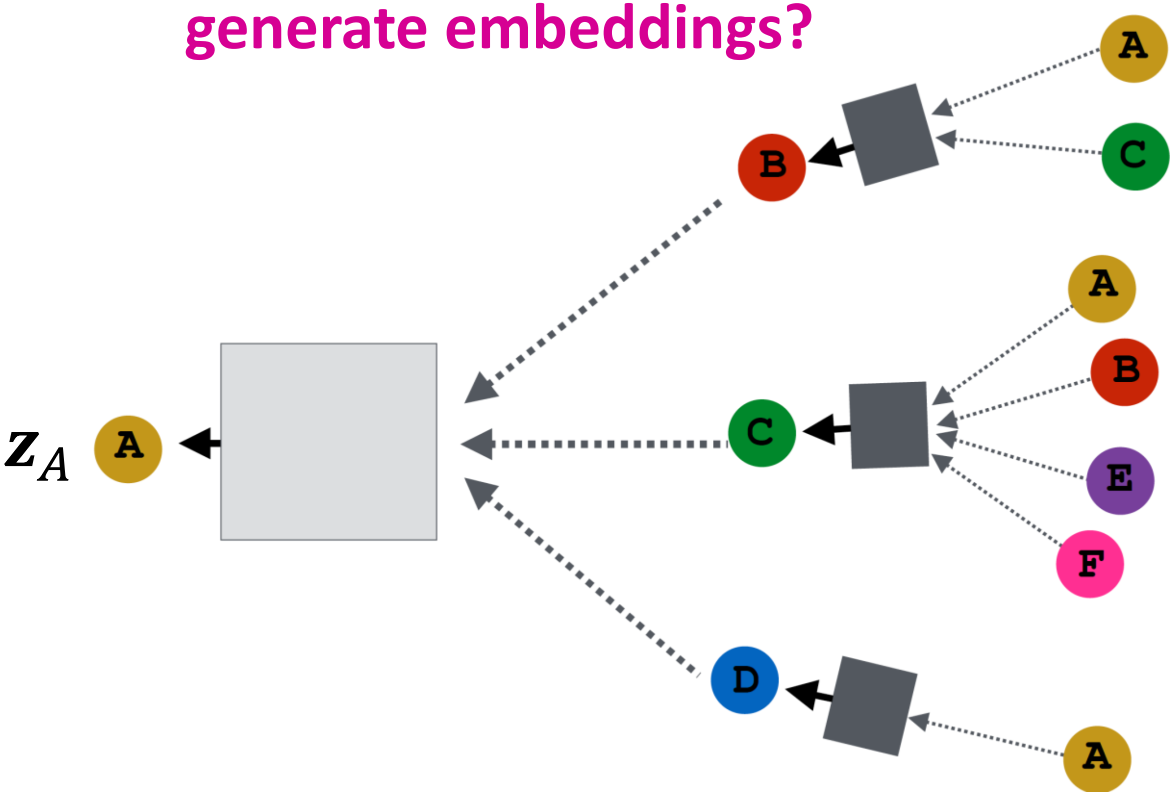


Shared NN weights

Average of neighbor's previous layer embeddings - **Permutation invariant**

Training the Model

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

How to Train A GNN

- Node embedding $\boldsymbol{z}_v$ is a function of input graph
- **Supervised setting**: We want to minimize loss $\mathcal{L}$:

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

  - $\boldsymbol{y}$: node label
  - $\mathcal{L}$ could be L2 if $\boldsymbol{y}$ is real number, or cross entropy if $\boldsymbol{y}$ is categorical (loss in Maximum Likelihood Estimation)
    - Cross entropy loss (CE):
      - $\mathrm{CE}(\boldsymbol{y}, f(\boldsymbol{x})) = -\sum_{i=1}^{C}(y_i \log f_\Theta(x)_i)$
      - y_i and $f_\Theta(x)_i$ are the **actual** and **predicted** values of the i-th class
      - **Intuition:** the lower the loss, the closer the prediction is to one-hot

- **Unsupervised setting:**
  - No node label available
  - **Use the graph structure as the supervision!**

Unsupervised Training

One possible idea: "Similar" nodes have similar embeddings:

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

- where $y_{u,v} = 1$ when node u and v are **similar**
- $z_u = f_{\Theta}(u)$ and $\mathrm{DEC}(\cdot,\cdot)$ is the dot product

Node similarity can be anything from embeddings, e.g., a loss based on:

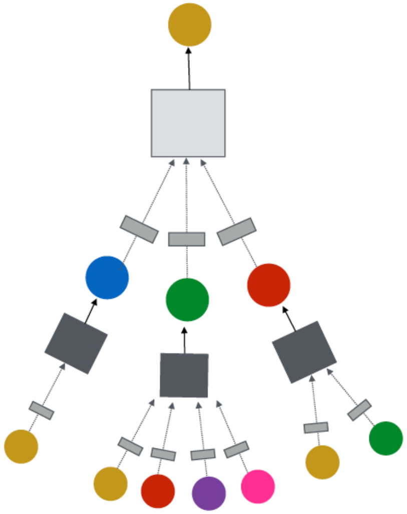
– **Random walks** (node2vec, DeepWalk, struc2vec)

– **Matrix factorization**

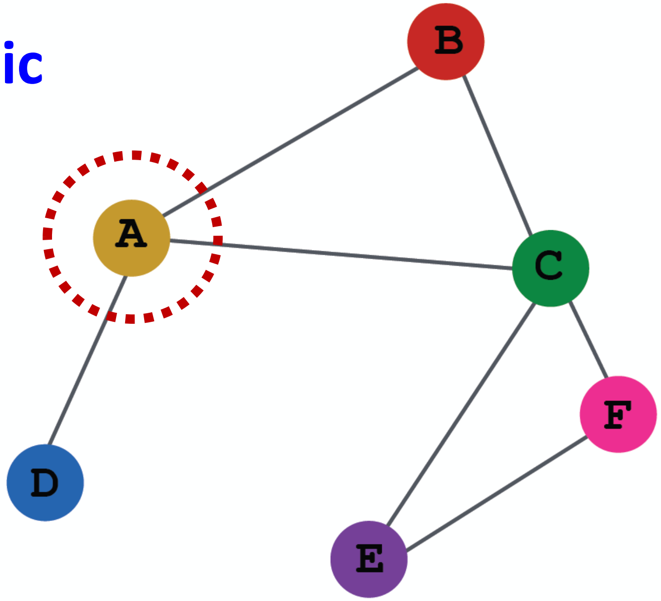
Supervised Training

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**)

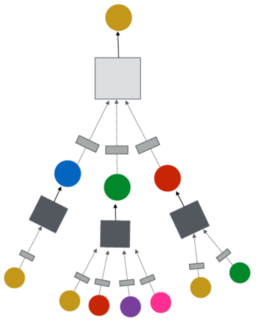
Use cross entropy loss

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

Encoder output: node embedding

Classification weights

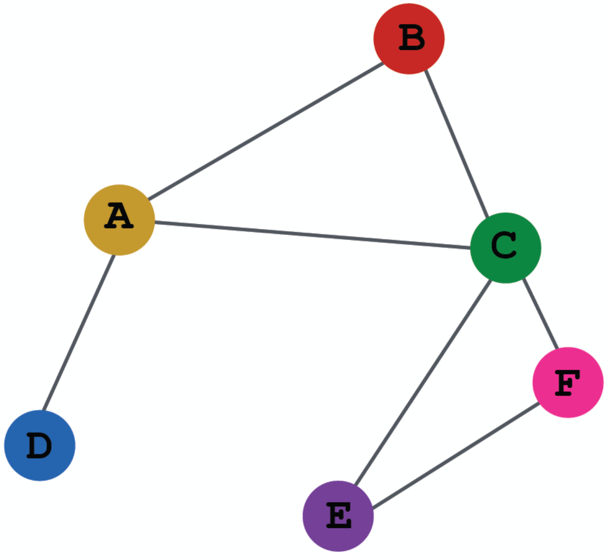
Node class label

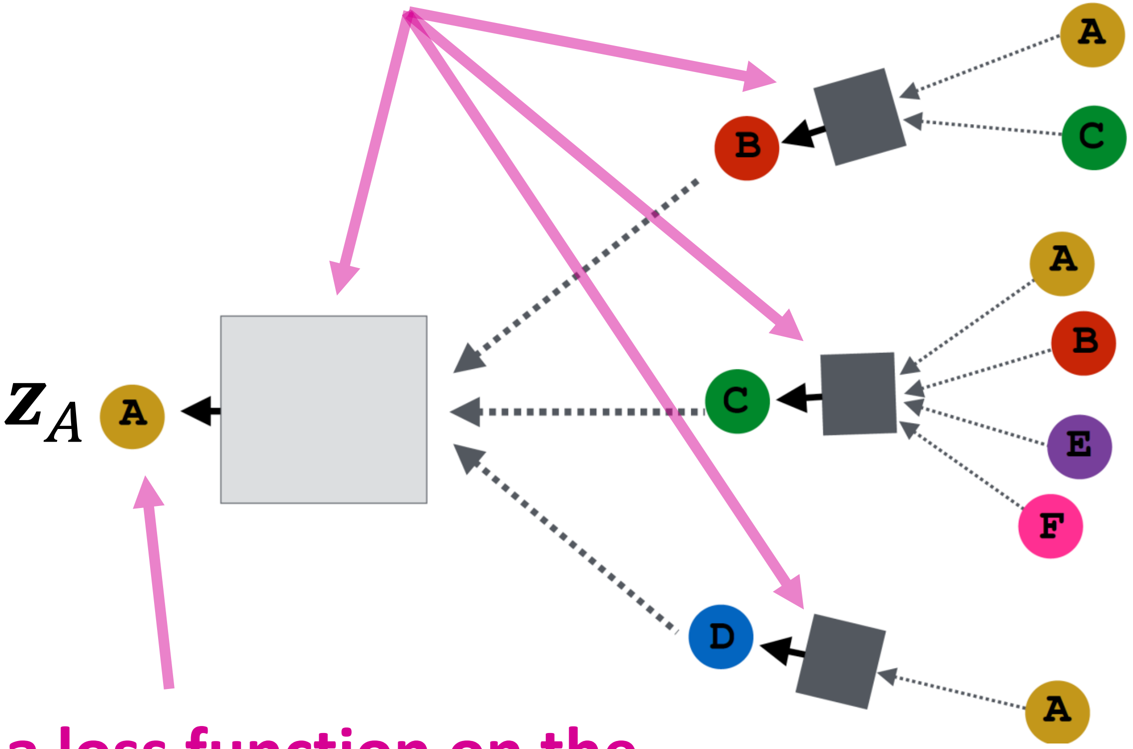


Safe or toxic drug?

Model Design: Overview

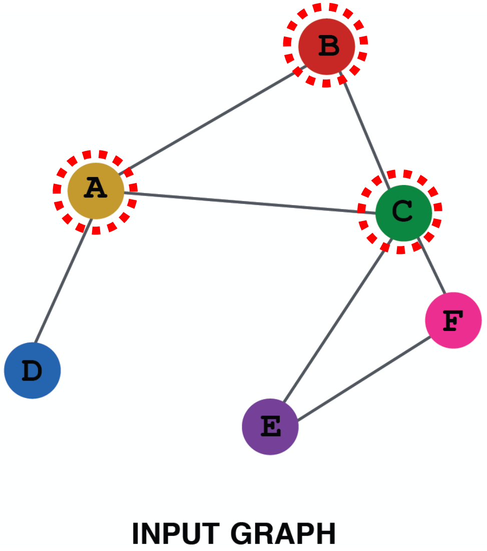
(1) Define a neighborhood aggregation function



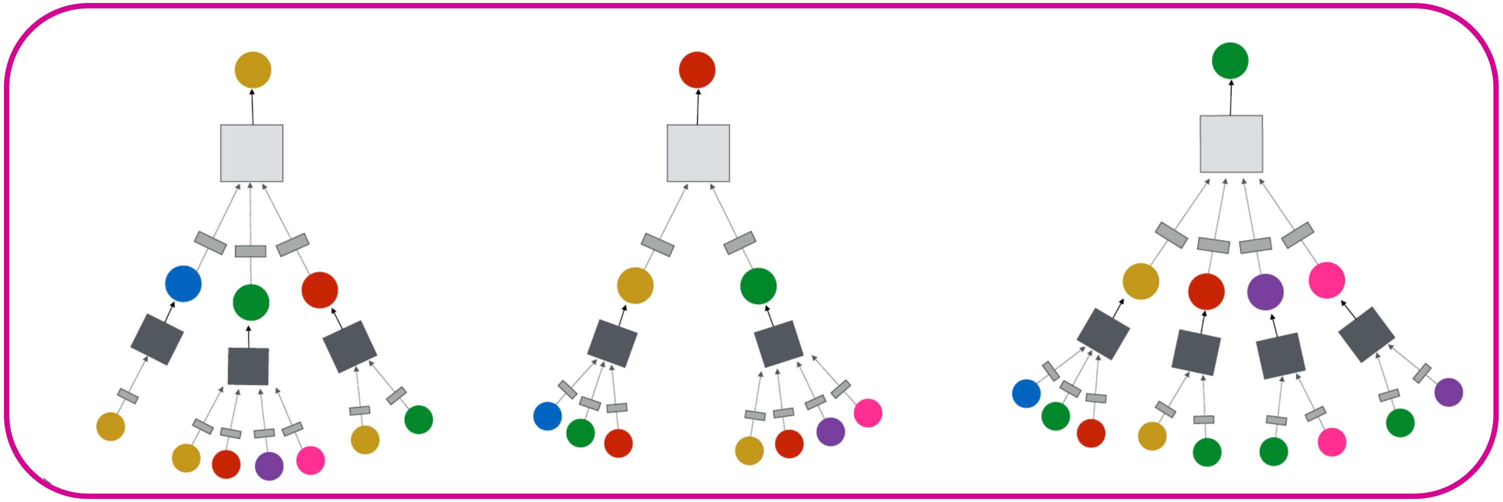


(2) Define a loss function on the embeddings

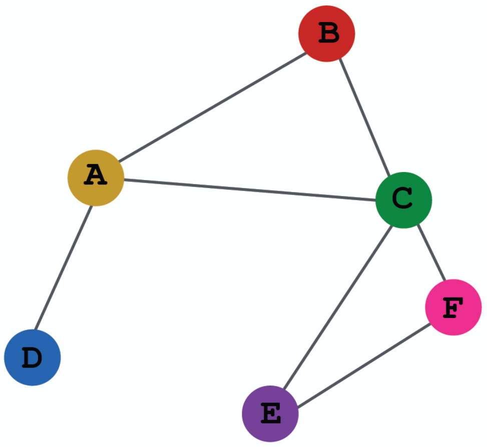
Model Design: Overview



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



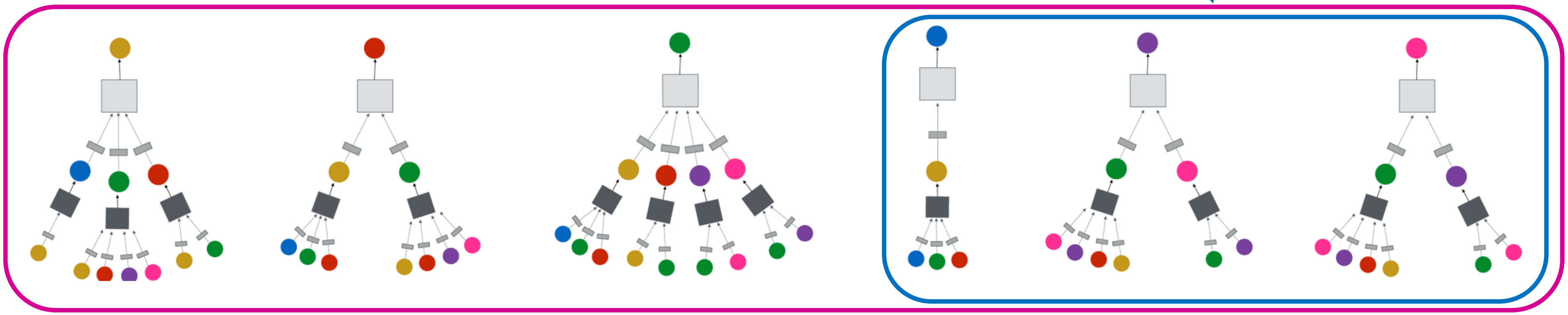
Model Design: Overview



INPUT GRAPH

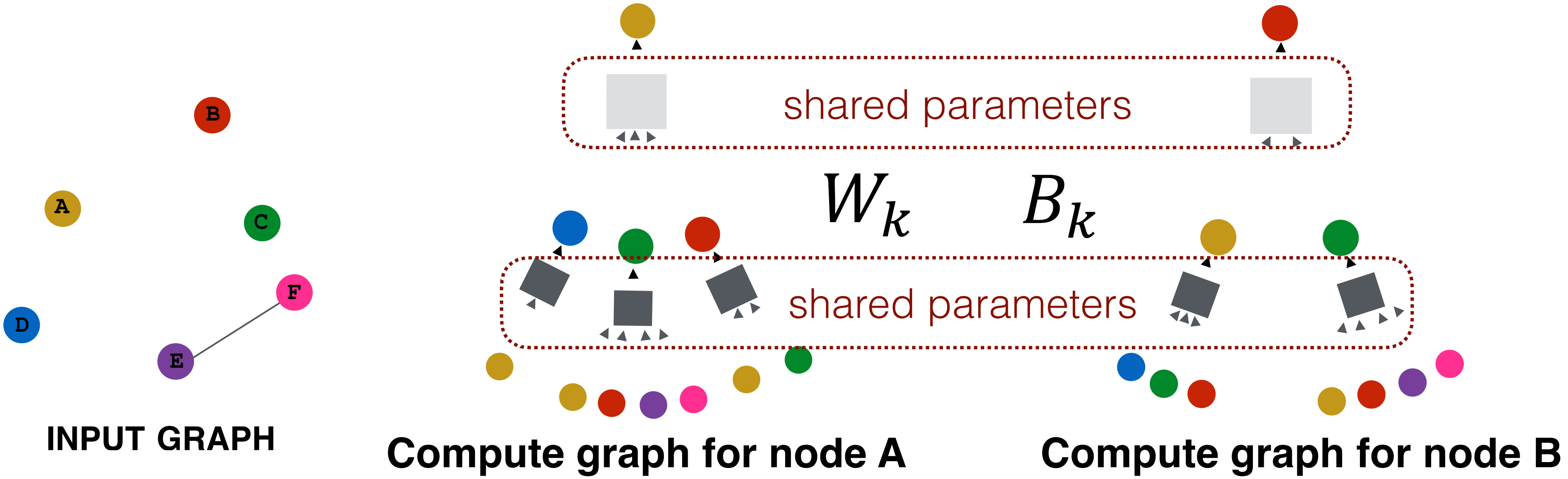
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

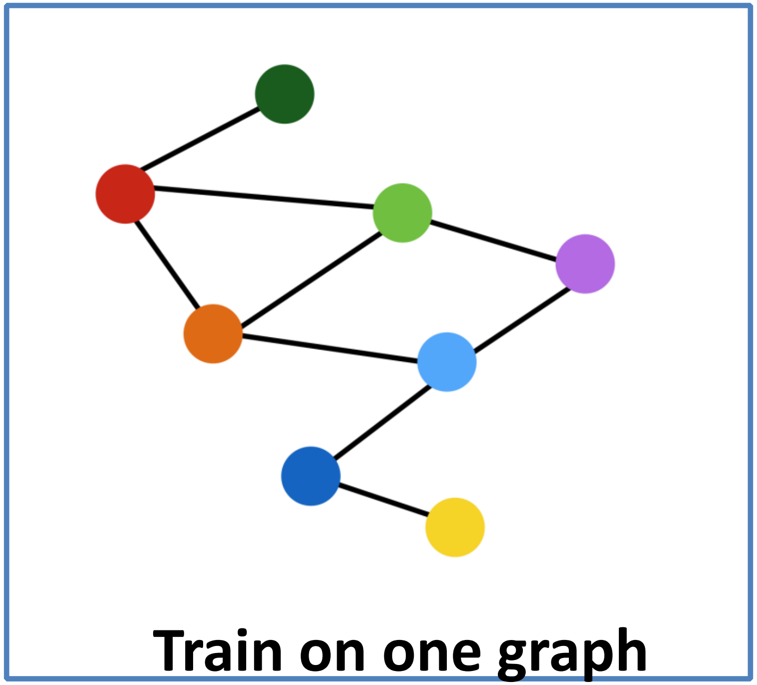


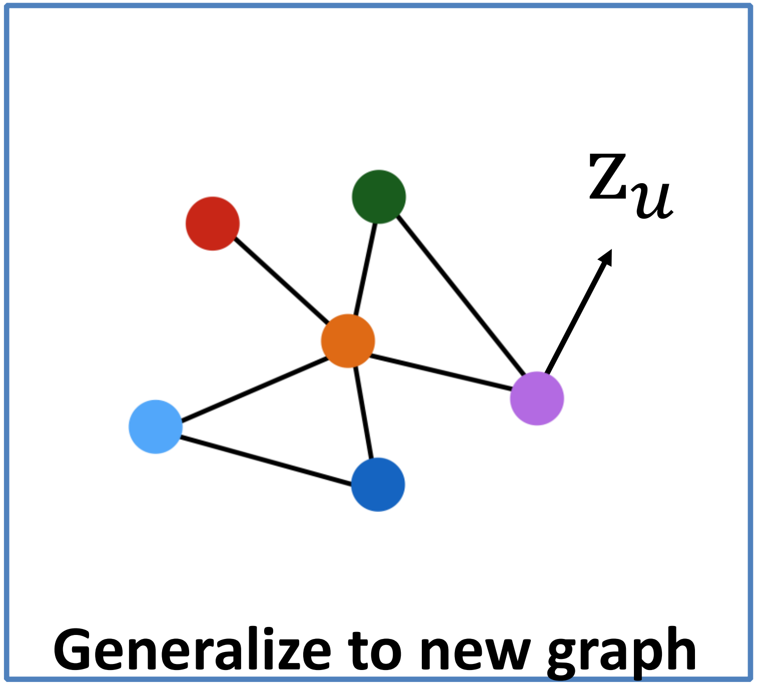
Inductive Capability

- **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**!



Inductive Capability: New Graphs

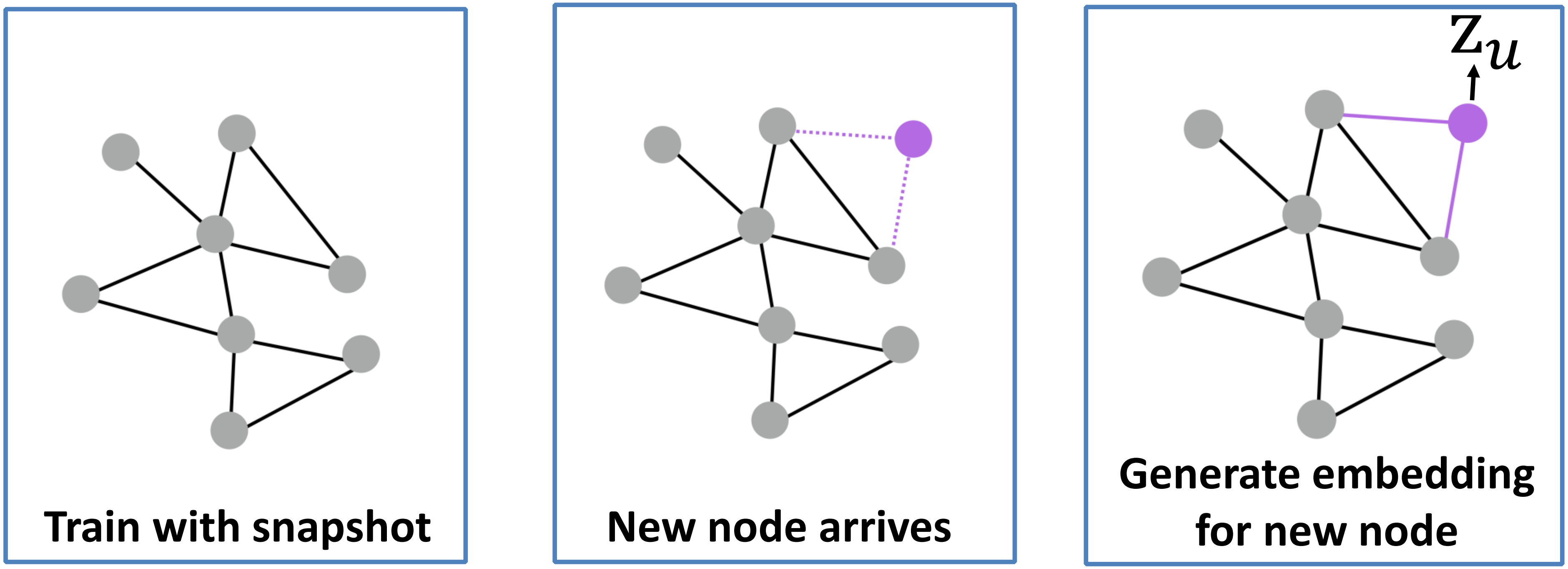




Inductive node embedding → Generalize to entirely unseen graphs

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



Train with snapshot

New node arrives

Generate embedding for new node

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

Summary so far

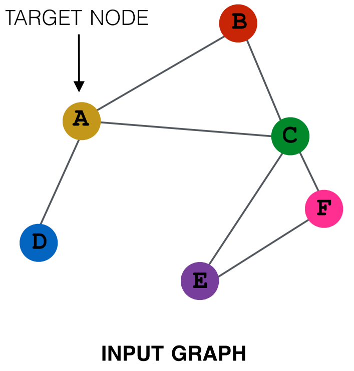
- How to build CNNs for graphs

  use local neighborhood of a node

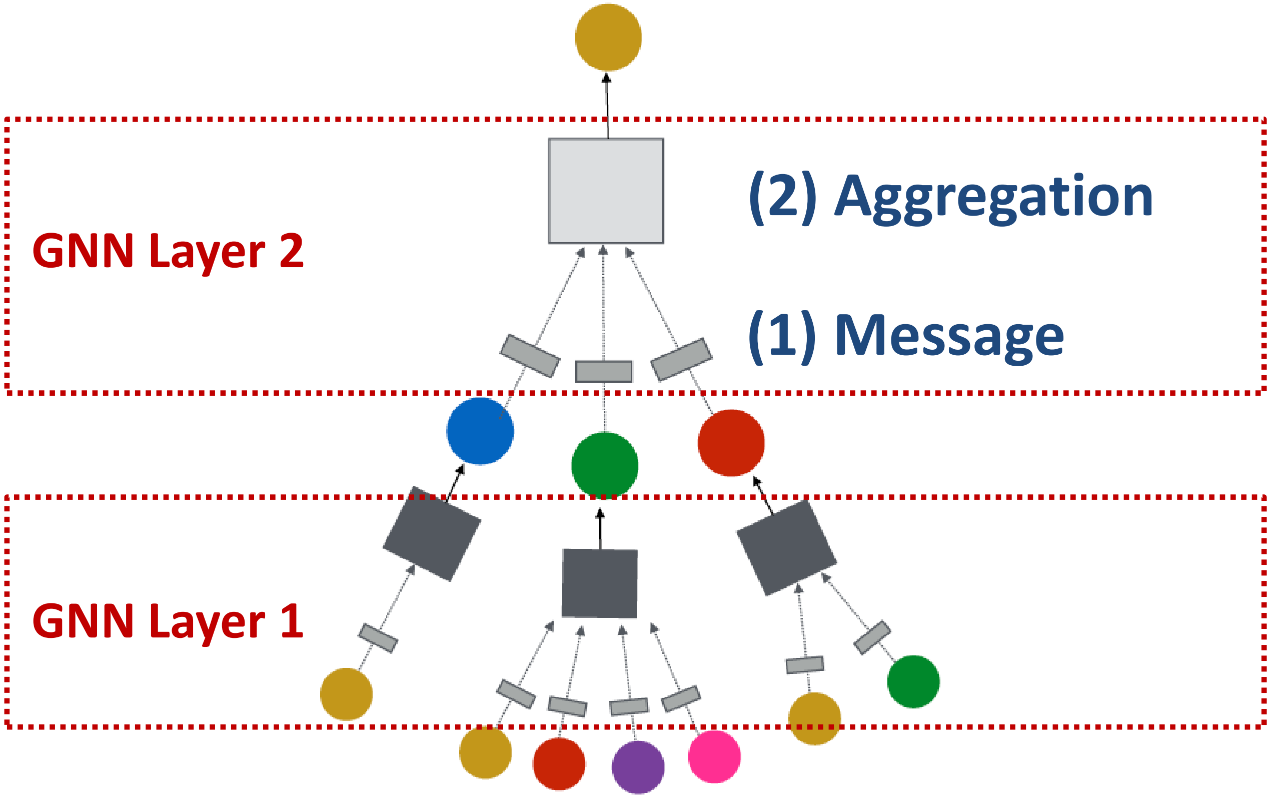
- Next: more details using a general GNN framework

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

A General GNN Framework



(5) Learning objective



GNN Layer 2

(2) Aggregation

(1) Message

(3) Layer connectivity

GNN Layer 1

(4) Graph augmentation

Outline

- General Framework
- A single GNN layer: Aggregation and Message
- Layer Connectivity: Stacking
- Graph manipulations
- Learning objectives

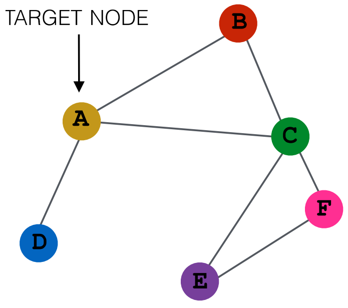
A SINGLE GNN LAYER

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

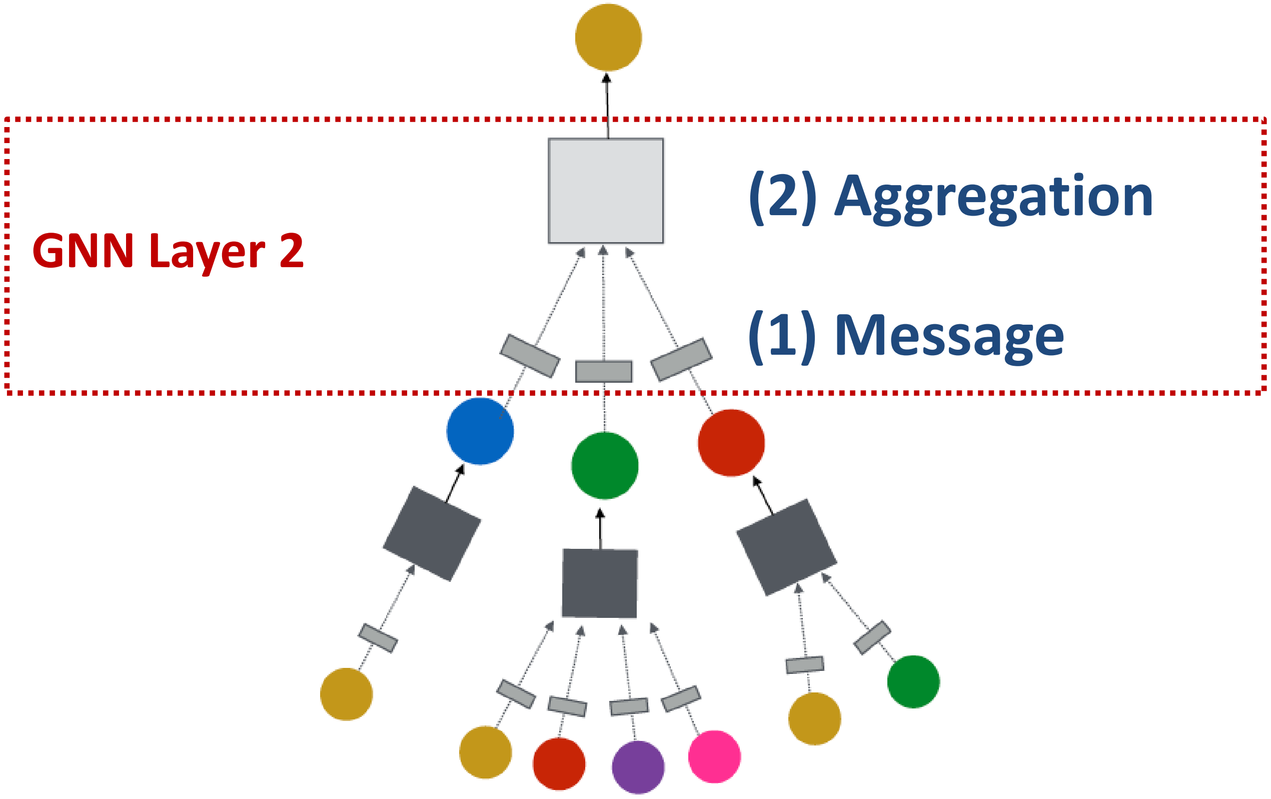
A GNN Layer

GNN Layer = Message + Aggregation

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



INPUT GRAPH



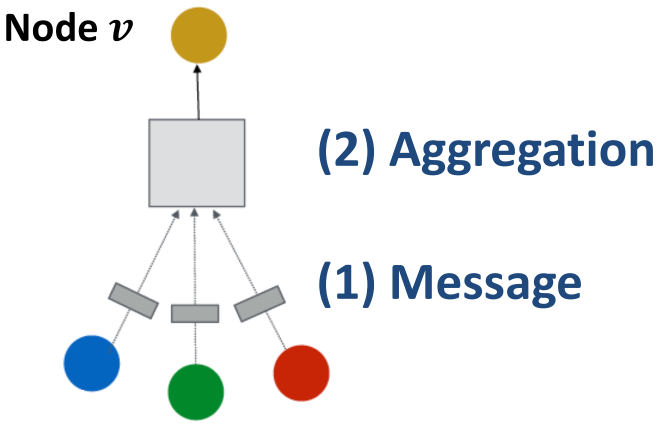
GNN Layer 2

(2) Aggregation

(1) Message

A Single GNN Layer

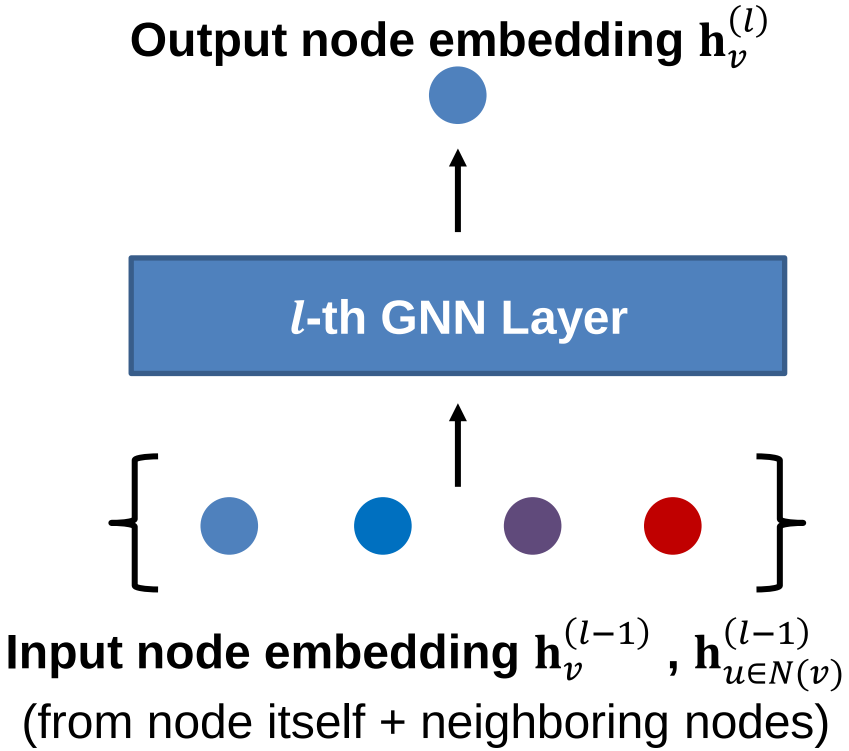
- **Idea of a GNN Layer:**
  - Compress a set of vectors into a single vector
  - **Two-step process:**
    - **(1) Message**
    - **(2) Aggregation**



Node v

(2) Aggregation

(1) Message



Output node embedding $\mathbf{h}_v^{(l)}$

***l*-th GNN Layer**

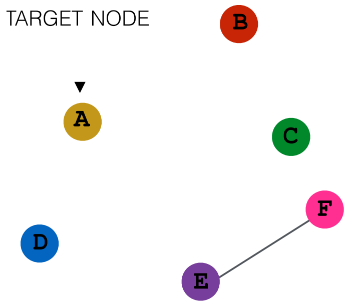
Input node embedding $\mathbf{h}_v^{(l-1)}$, $\mathbf{h}_{u\in N(v)}^{(l-1)}$

(from node itself + neighboring nodes)

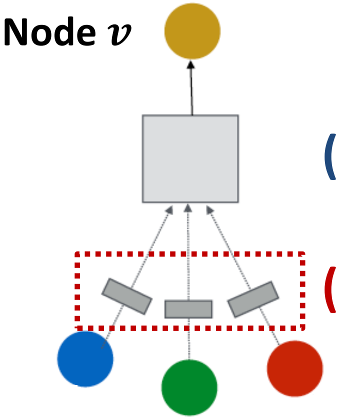
Message Computation

(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
  - **Example:** A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$
    - Multiply node features with weight matrix $\mathbf{W}^{(l)}$



INPUT GRAPH



Node v

(2) Aggregation

(1) Message

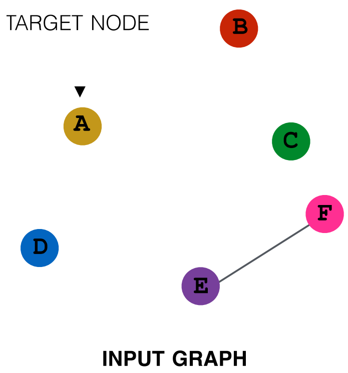
Message Aggregation

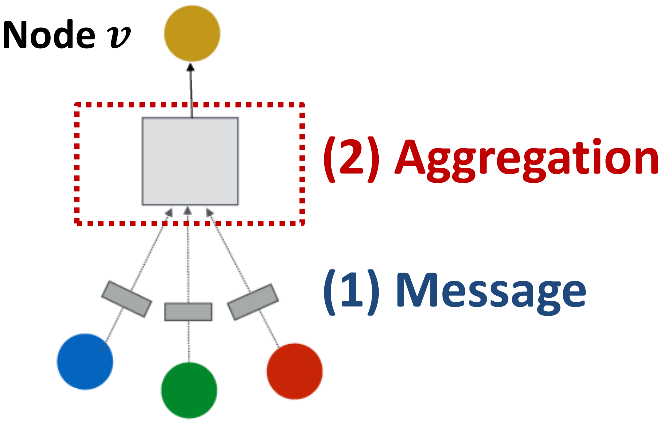
(2) Aggregation

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

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

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





Message Aggregation: Issue

Issue: Information from node v itself **could get lost**

- Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$

Solution: Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$

- **(1) Message: compute message from node v itself**
  - Usually, a **different message computation** will be performed

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

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

Then aggregate from node itself

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

First aggregate from neighbors

A Single GNN Layer

Putting things together:

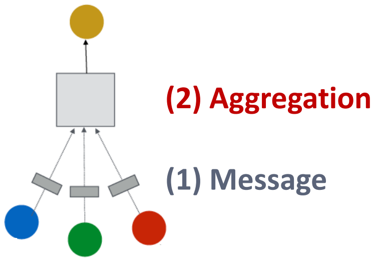
- **(1) Message**: each node computes a message

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

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



T. Kipf, M. Welling. Semi-Supervised Classification with Graph Convolutional Networks, ICLR 2017

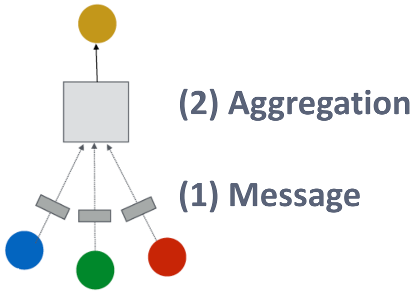
Classical GNN Layers: GCN (1)

(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?**

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



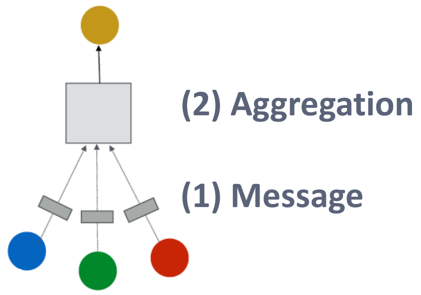
(2) Aggregation

(1) Message

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

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

Hamilton et al. Inductive Representation Learning on Large Graphs, NeurIPS 2017

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

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

Aggregation

Message computation

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

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

Aggregation

Message computation

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

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

Aggregation

applied to a random permutation

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

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

Classical GNN Layers: GAT (1)

(3) Graph Attention Networks

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

Attention weights

- weighting factor (importance) of the message of node u to node v

- **In GCN and GraphSAGE:**
  - $\alpha_{vu} = \frac{1}{|N(v)|}$ defined **explicitly** based on the structural properties of the graph (node degree)
  - All neighbors $u \in N(v)$ are equally important to node v

Classical GNN Layers: GAT (2)

(3) Graph Attention Networks

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

Attention weights

Not all node's neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention** $\boldsymbol{\alpha_{vu}}$ 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 weighting factors α_{vu} 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

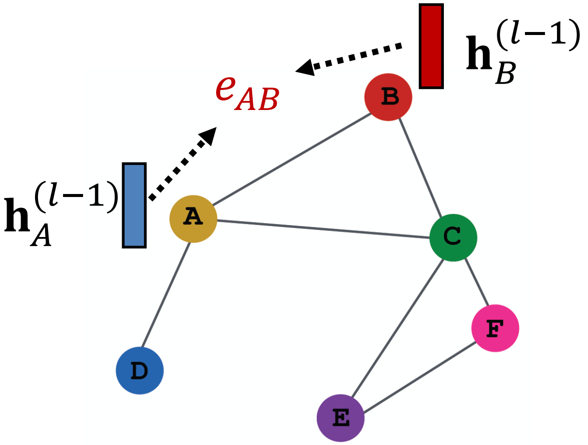
Attention Mechanism (1)

Let α_{vu} be computed as a byproduct of an **attention mechanism $\boldsymbol{a}$:**

– (1) Let a compute **attention coefficients $\boldsymbol{e_{vu}}$** across pairs of nodes u, v based on their messages:

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

- **$\boldsymbol{e_{vu}}$ indicates the importance of $\boldsymbol{u'}$s message to node $\boldsymbol{v}$**



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

Attention Mechanism (2)

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

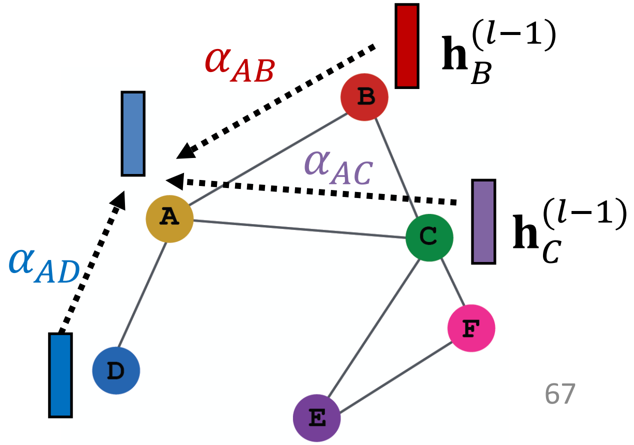
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** $\boldsymbol{\alpha_{vu}}$:

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

Weighted sum using $\alpha_{AB}, \alpha_{AC}, \alpha_{AD}$**:**

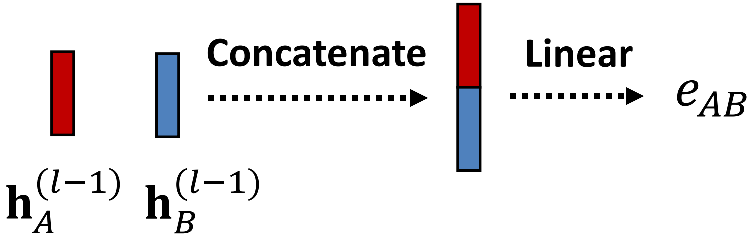
$$\mathbf{h}_A^{(l)} = \sigma(\alpha_{AB}\mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)} + \alpha_{AC}\mathbf{W}^{(l)}\mathbf{h}_C^{(l-1)} + \alpha_{AD}\mathbf{W}^{(l)}\mathbf{h}_D^{(l-1)})$$



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)



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

- 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):

$$\mathbf{h}_v^{(l)}[1] = \sigma(\textstyle\sum_{u \in N(v)} \alpha_{vu}^{1} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$
$$\mathbf{h}_v^{(l)}[2] = \sigma(\textstyle\sum_{u \in N(v)} \alpha_{vu}^{2} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$
$$\mathbf{h}_v^{(l)}[3] = \sigma(\textstyle\sum_{u \in N(v)} \alpha_{vu}^{3} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

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

Benefits of Attention Mechanism

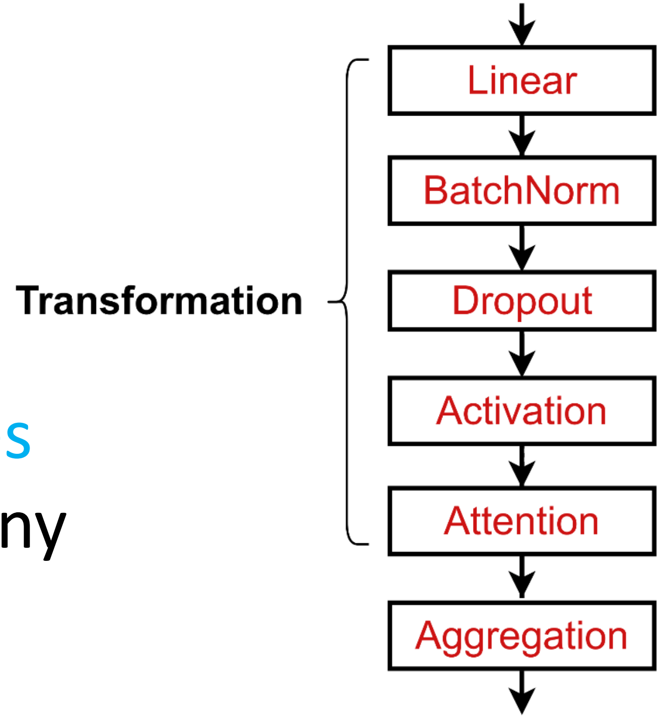
- **Key benefit:** Allows for (implicitly) specifying **different importance values (α_{vu}) to different neighbors**
- **Computationally efficient**:
  - Computation of attentional coefficients can be parallelized across all edges of the graph
  - Aggregation may be parallelized across all nodes
- **Storage 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

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

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

A suggested GNN Layer

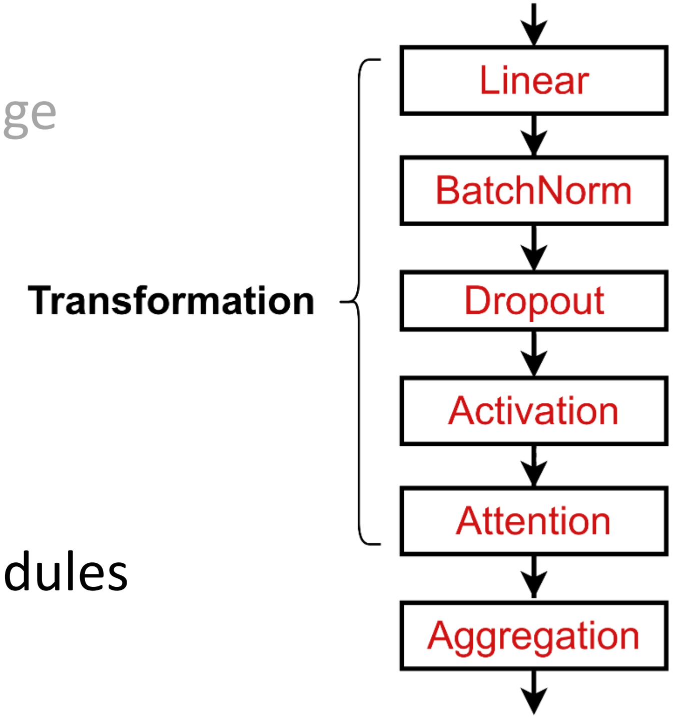


J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

GNN Layer in Practice

- **Many modern deep learning modules can be incorporated into a GNN layer**
  - **Attention/Gating:**
    - Control the importance of a message
  - **Batch Normalization:**
    - Stabilize neural network training
  - **Dropout:**
    - Prevent overfitting
  - **More:**
    - Any other useful deep learning modules

A suggested GNN Layer



S. Loffe, C.Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch Normalization

- **Goal**: Stabilize neural networks training
- **Idea**: Given a batch of inputs (node embeddings)
  - Re-center the node embeddings into zero mean
  - Re-scale the variance into unit variance

Input: $\mathbf{X} \in \mathbb{R}^{N \times d}$

N node embeddings

Trainable Parameters:

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

Output: $\mathbf{Y} \in \mathbb{R}^{N \times d}$

Normalized node embeddings

Step 1:

Compute the mean and variance over N embeddings

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

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

Step 2:

Normalize the feature using computed mean and variance

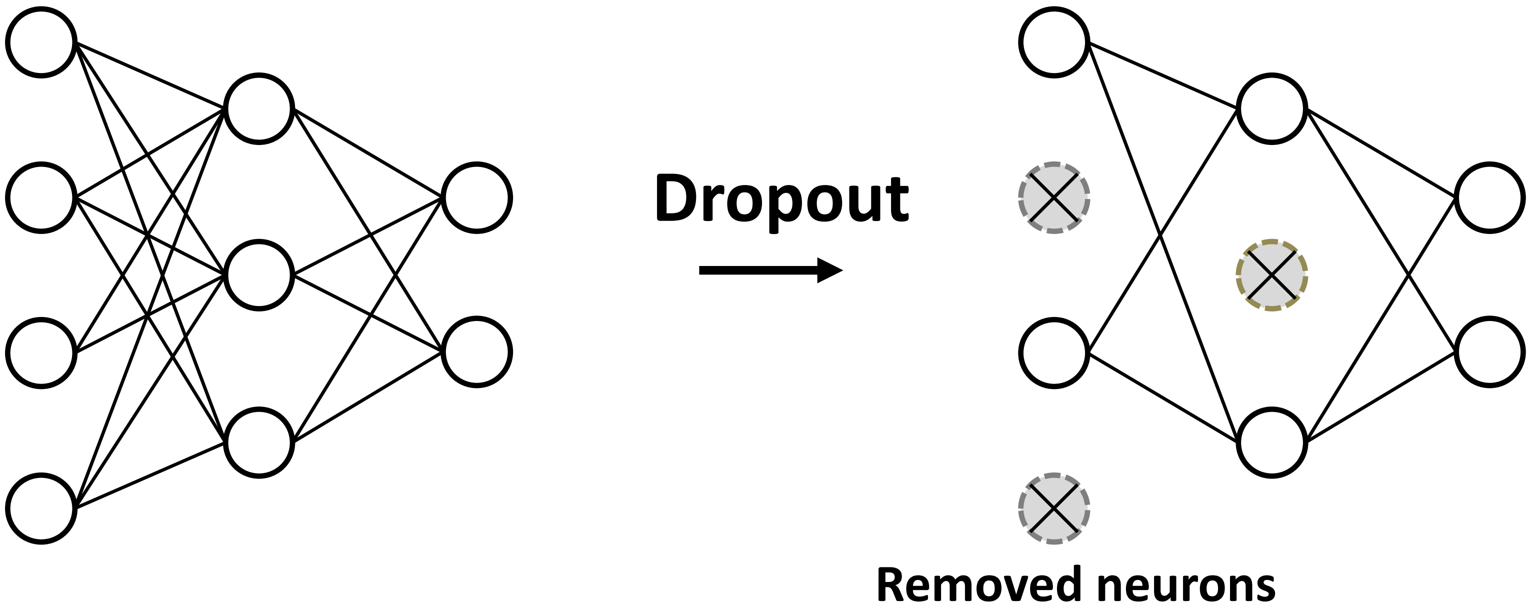
$$\widehat{\mathbf{X}}_{i,j} = \frac{\mathbf{X}_{i,j} - \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{\sigma}_j^2 + \epsilon}}$$

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

Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

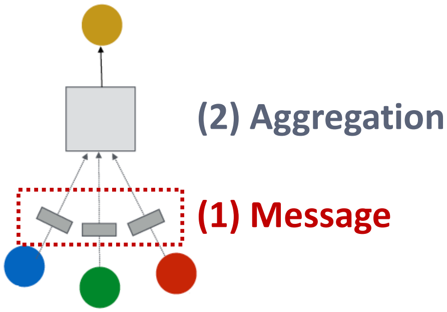
Dropout

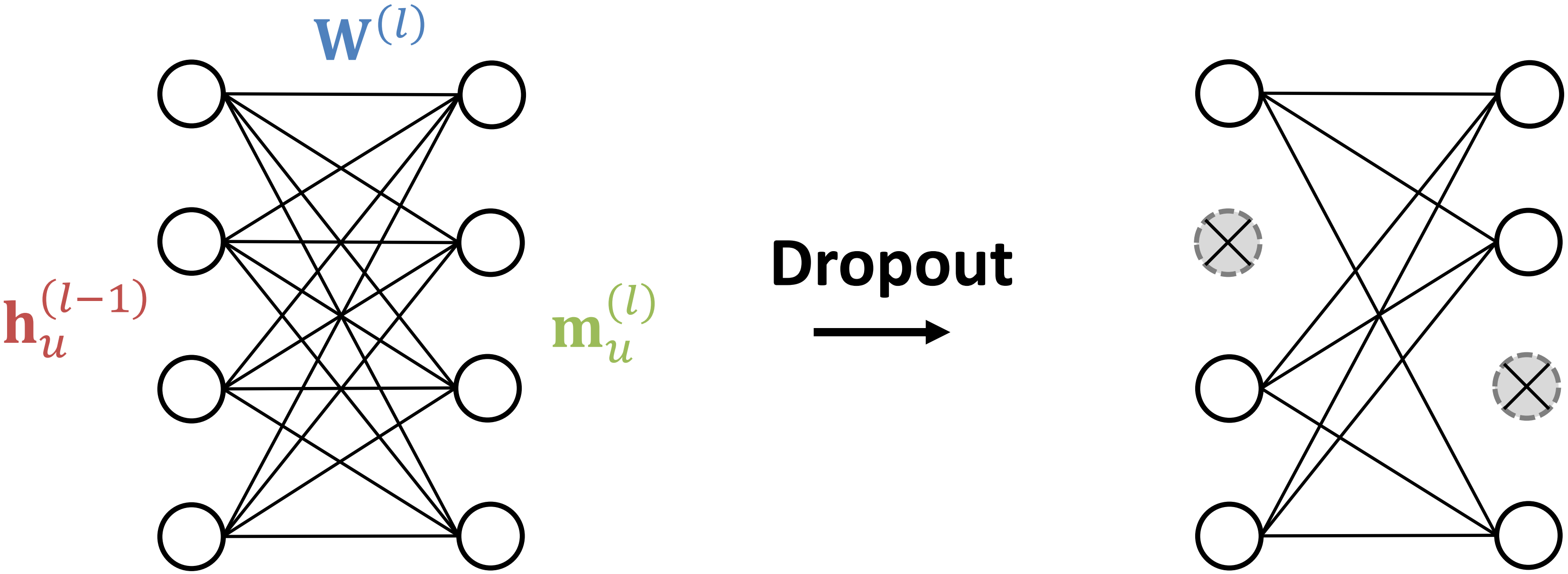
- **Goal**: Regularize a neural net to prevent overfitting.
- **Idea**:
  - **During training**: with some probability p, randomly set neurons to zero (turn off)
  - **During testing:** Use all the neurons for computation



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

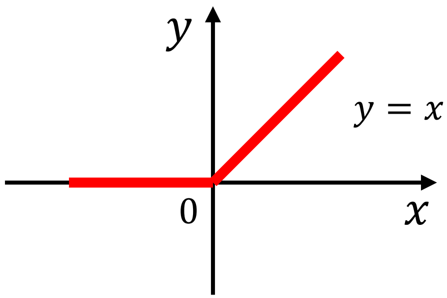
Activation (Non-linearity)

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

- **Rectified linear unit (ReLU)**

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

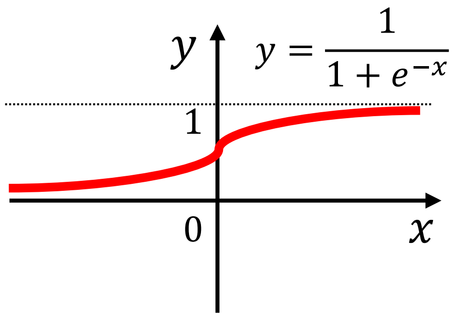
  – Most commonly used



- **Sigmoid**

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

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

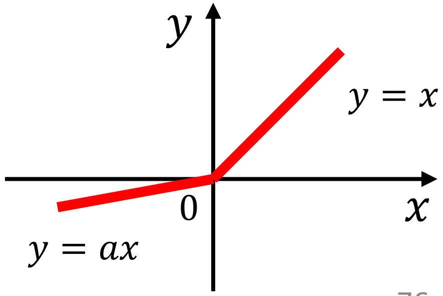


- **Parametric ReLU**

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

  a_i is a trainable parameter

  – Empirically performs better than ReLU

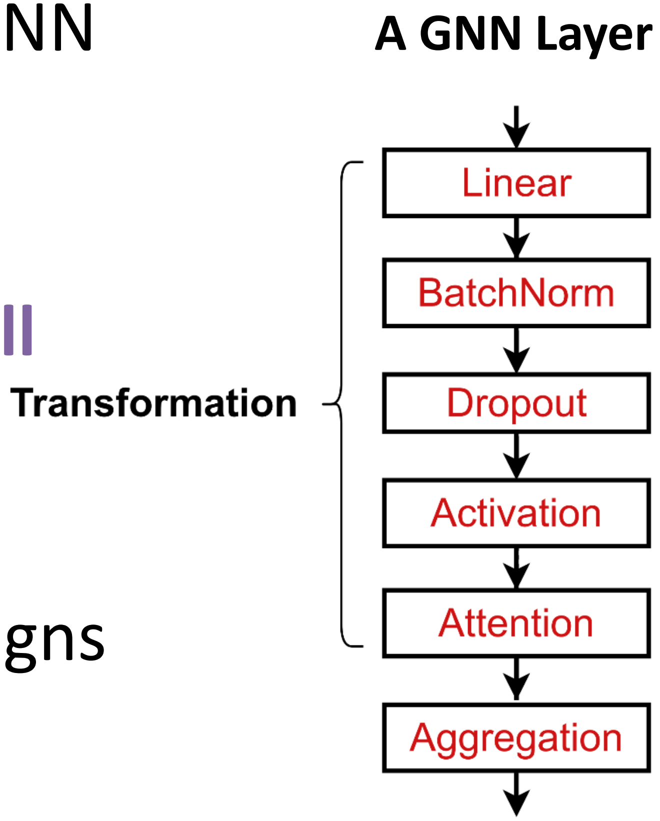


GNN Layer in Practice

- **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**

- You can explore diverse GNN designs or try out your own ideas in [GraphGym]



Summary

- Single GNN layer:
  - Message
  - Aggregation

Apply ML modules

- Attention
- Drop out
- Normalization
- Non-linearity

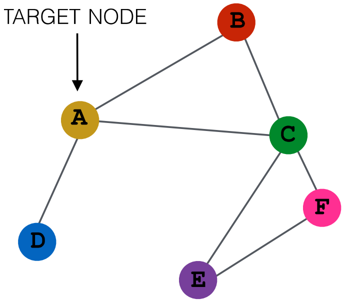
Outline

- General Framework
- A single GNN layer: Aggregation and Message
- Layer Connectivity: Stacking
- Graph manipulations
- Learning objectives

STACKING LAYERS

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

Stacking GNN Layers

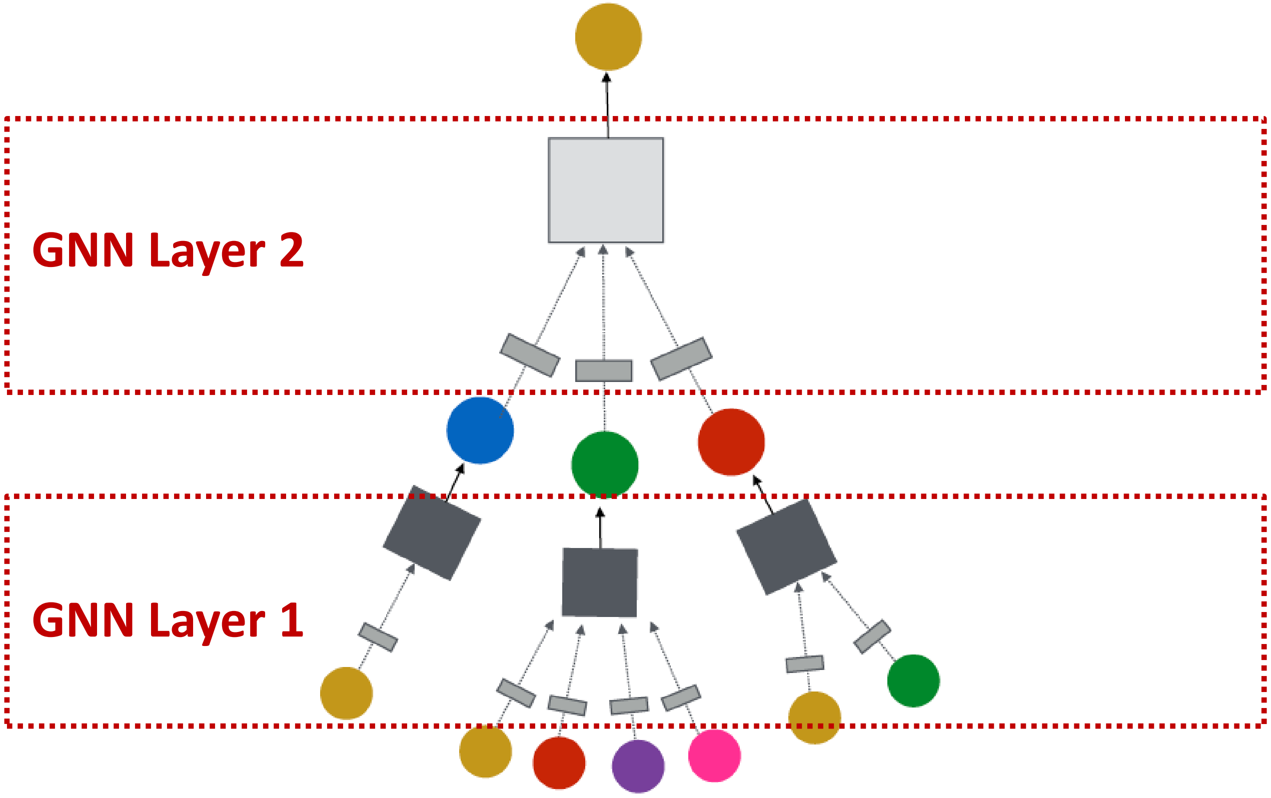


INPUT GRAPH

How to connect GNN layers into a GNN?

- **Stack layers sequentially**
- **Ways of adding skip connections**

(3) Layer connectivity

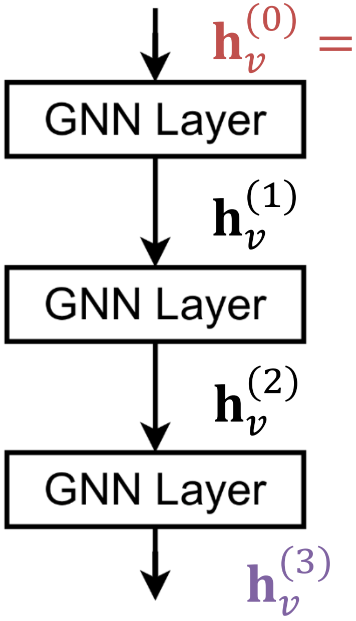


GNN Layer 2

GNN Layer 1

Stacking GNN Layers

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



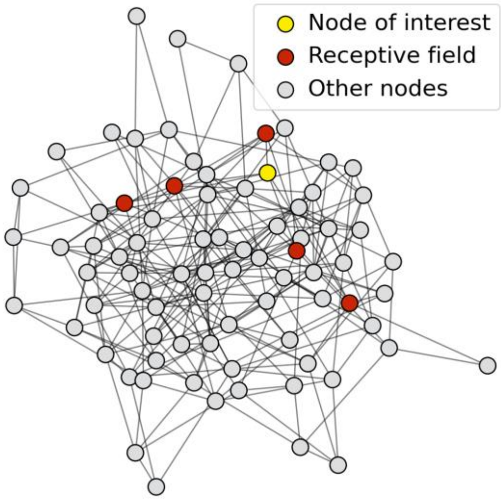
The Over-Smoothing Problem

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

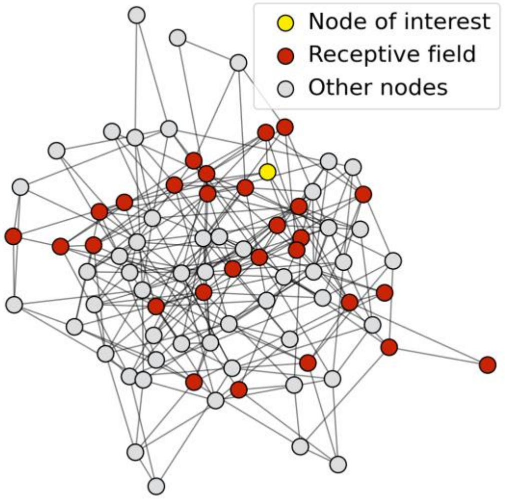
Receptive Field of a GNN

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

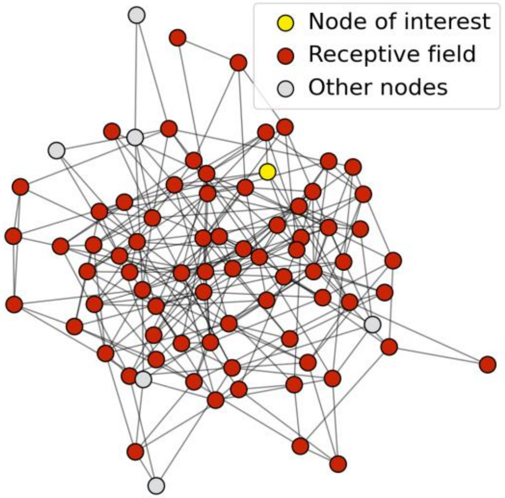
Receptive field for
1-layer GNN



Receptive field for
2-layer GNN



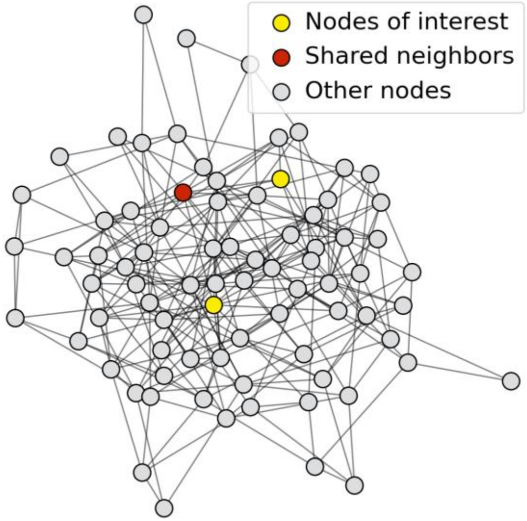
Receptive field for
3-layer GNN



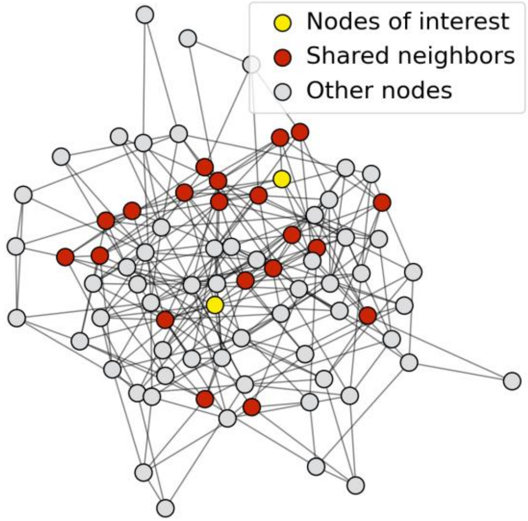
Receptive Field of a GNN

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

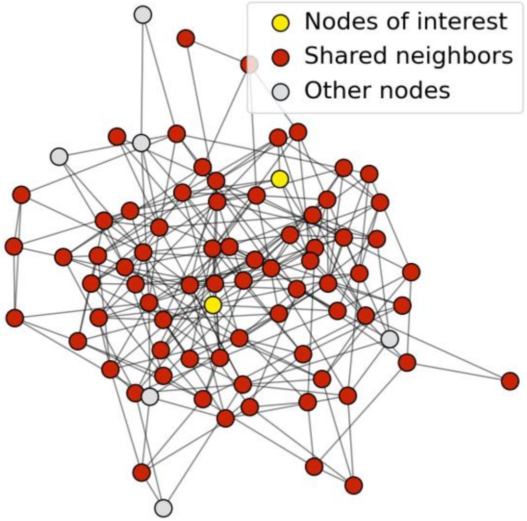
1-hop neighbor overlap
Only 1 node



2-hop neighbor overlap
About 20 nodes



3-hop neighbor overlap
Almost all the nodes!



Receptive Field & Over-smoothing

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

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?

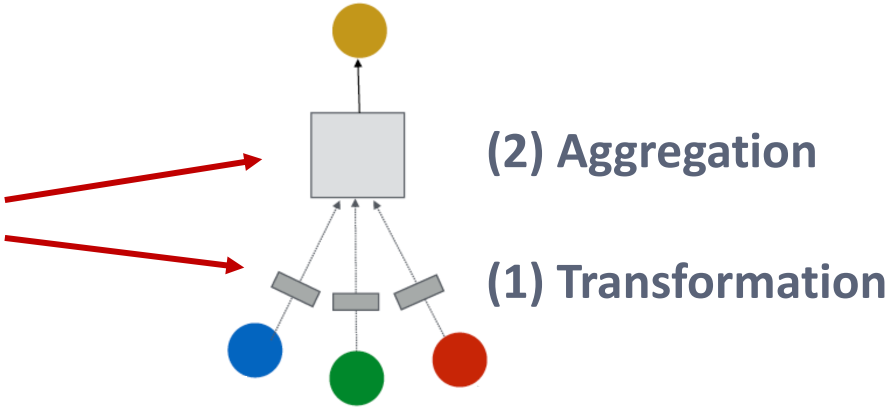
Expressive Power for Shallow GNNs

- **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



(2) Aggregation

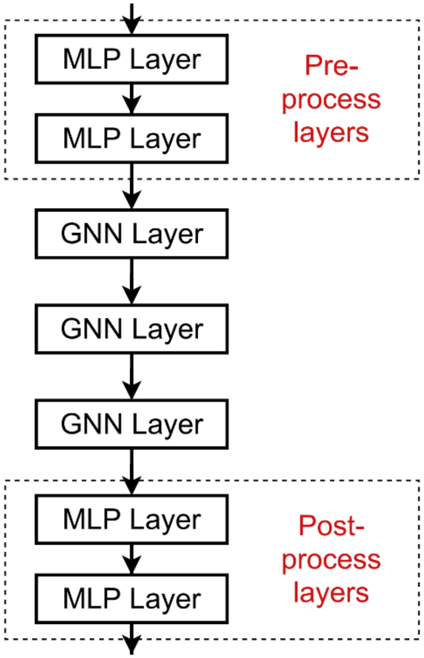
(1) Transformation

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!

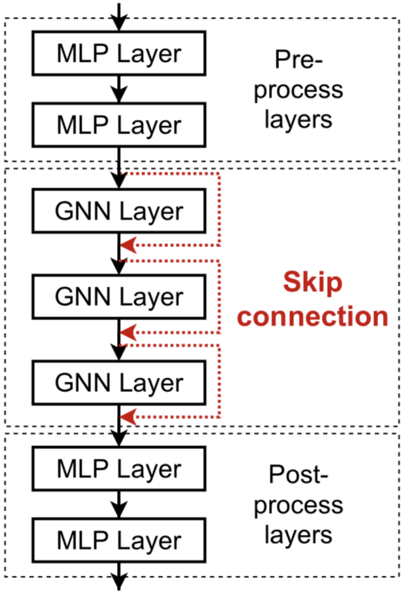
He et al. Deep Residual Learning for Image Recognition, CVPR 2015

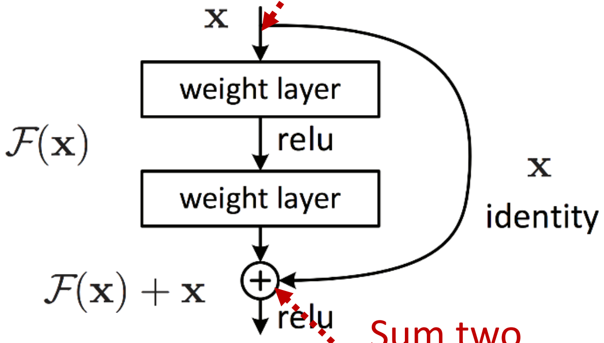
Design GNN Layer Connectivity

- **What if my problem still requires many GNN layers?**

Lesson 2: Add skip connections in GNNs

- **Observation from over-smoothing:** Node embeddings in earlier GNN layers can sometimes better differentiate nodes
- **Solution:** We can increase the impact of earlier layers on the final node embeddings, **by adding shortcuts in GNN**





Duplicate into two branches

Sum two branches

Idea of skip connections:

Before adding shortcuts:

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

After adding shortcuts:

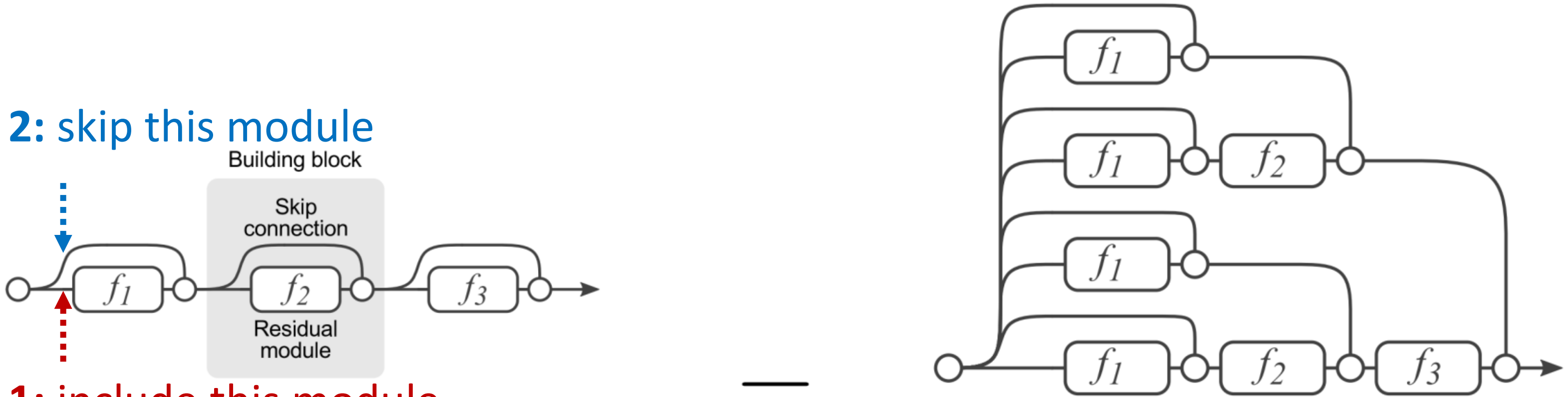
$$\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$$

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

Path 2: skip this module



Path 1: include this module

(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**

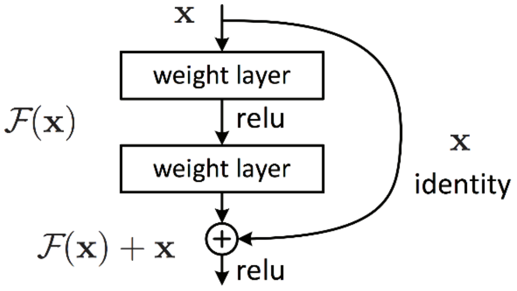
- $$\mathbf{h}_v^{(l)} = \sigma\left(\boxed{\sum_{u\in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}}\right)$$

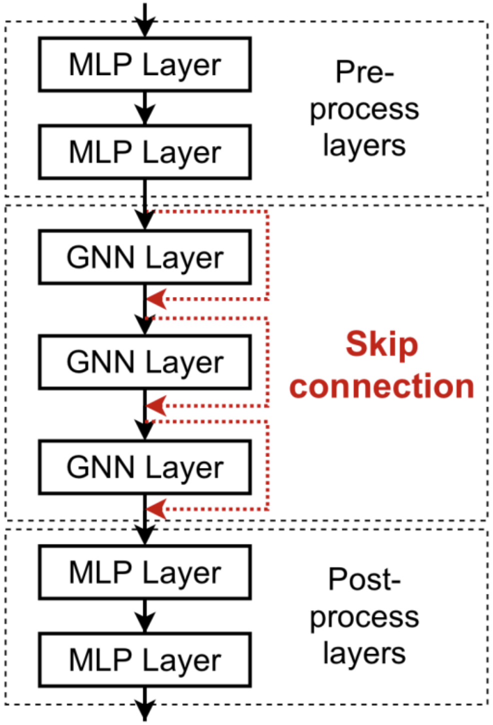
This is our $F(\mathbf{x})$

- **A GCN layer with skip connection**

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

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

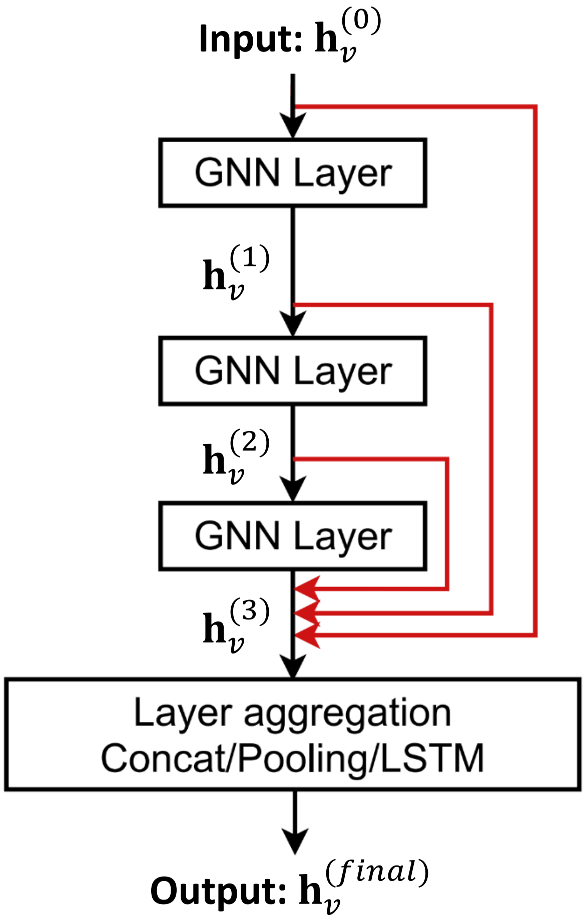




Xu et al. Representation learning on graphs with jumping knowledge networks, ICML 2018

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



Summary so far

A general perspective for GNNs

- **GNN Layer**:
  - Transformation + Aggregation
  - Classic GNN layers: GCN, GraphSAGE, GAT
- **Layer connectivity**:
  - Deciding number of layers
  - Skip connections

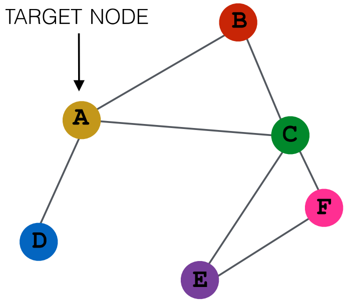
Outline

- General Framework
- A single GNN layer: Aggregation and Message
- Layer Connectivity: Stacking
- Graph manipulations
- Learning objectives

GRAPH MANIPULATIONS

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

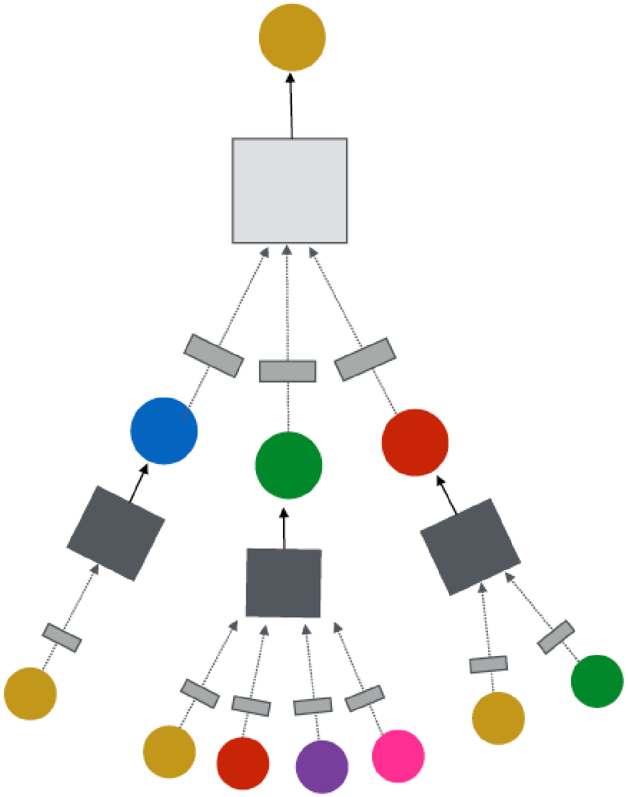
General GNN Framework



INPUT GRAPH

Idea: Raw input graph $\neq$ computational graph

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



(4) Graph manipulation

Why Manipulate Graphs

Our assumption so far has been

- **Raw input graph = computational graph**

Reasons for breaking this assumption

- **Feature level:**
  - The input graph **lacks features** → feature augmentation
- **Structure level:**
  - The graph is **too sparse** → inefficient message passing
  - The graph is **too dense** → message passing is too costly
  - The graph is **too large** → cannot fit the computational graph into a GPU
- It is 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**

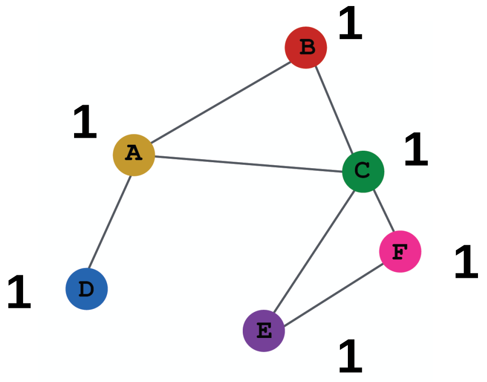
Feature Augmentation on Graphs

Why do we need feature augmentation?

- **(1) Input graph does not have node features**
  - This is common when we only have the adjacency matrix

Standard approaches:

(a) Assign constant values to nodes

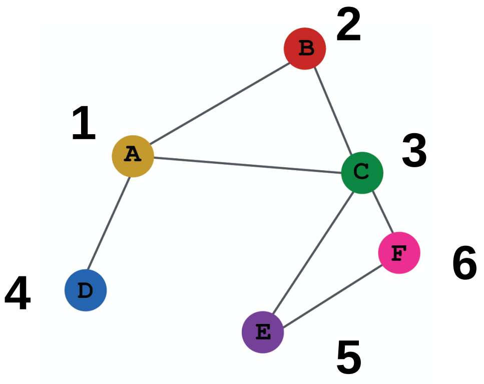


INPUT GRAPH

Feature Augmentation on Graphs

(b) Assign unique IDs to nodes

- These IDs are converted into **one-hot vectors**



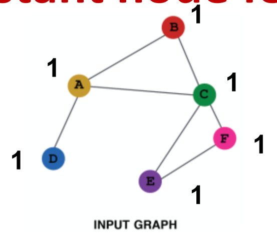
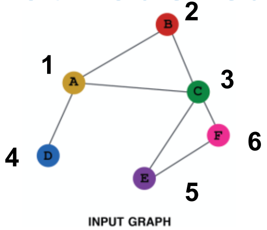
INPUT GRAPH

One-hot vector for node with ID=5

$$\underbrace{[0, 0, 0, 0, \overset{\text{ID = 5}}{\overset{\downarrow}{1}}, 0]}_{\text{Total number of IDs = 6}}$$

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**. High dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

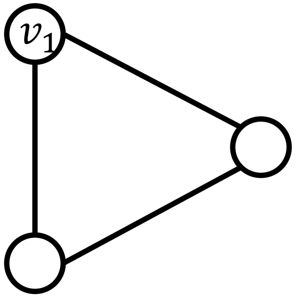
Feature Augmentation on Graphs

Why do we need feature augmentation?

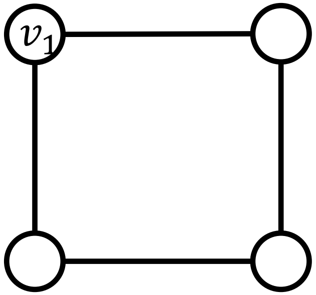
(2) Certain structures are hard to learn by GNN

- **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



J. You, J. Gomes-Selman, R. Ying, J. Leskovec. Identity-aware Graph Neural Networks, AAAI 2021

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

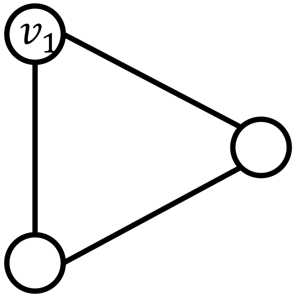
We start from cycle with length 0

Augmented node feature for v_1

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

↑

v_1 resides in a cycle with length 3

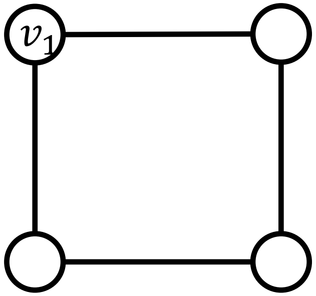


Augmented node feature for v_1

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

↑

v_1 resides in a cycle with length 4



Feature Augmentation on Graphs

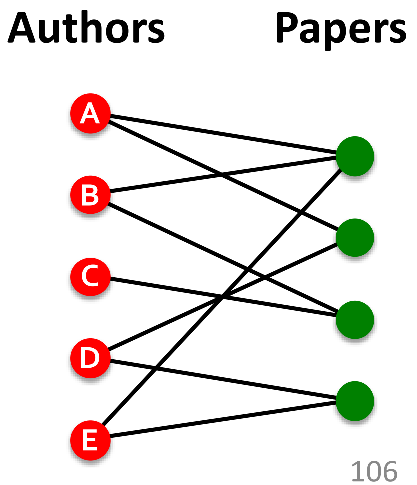
Why do we need feature augmentation?

- **(2) Certain structures are hard to learn by GNN**
- Other commonly used augmented features:
  - **Clustering coefficient**
  - **PageRank**
  - **Centrality**
  - **…**
- Any **feature** we have introduced when we talked about traditional ML approaches

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 adjacency matrix A for GNN computation, use $A + A^2$



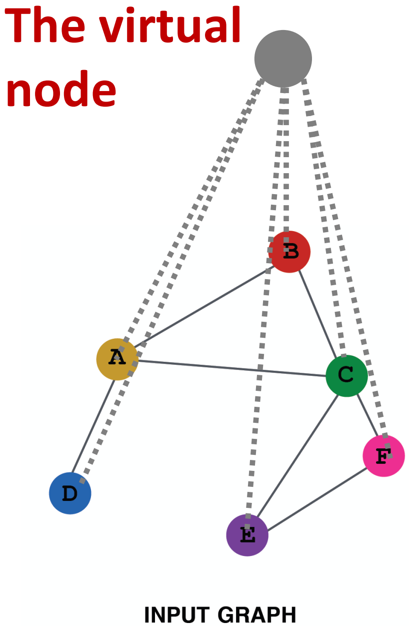
- **Use cases:** Bipartite graphs
  - Author-to-papers (they authored)
  - 2-hop virtual edges make an author-author collaboration graph

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**



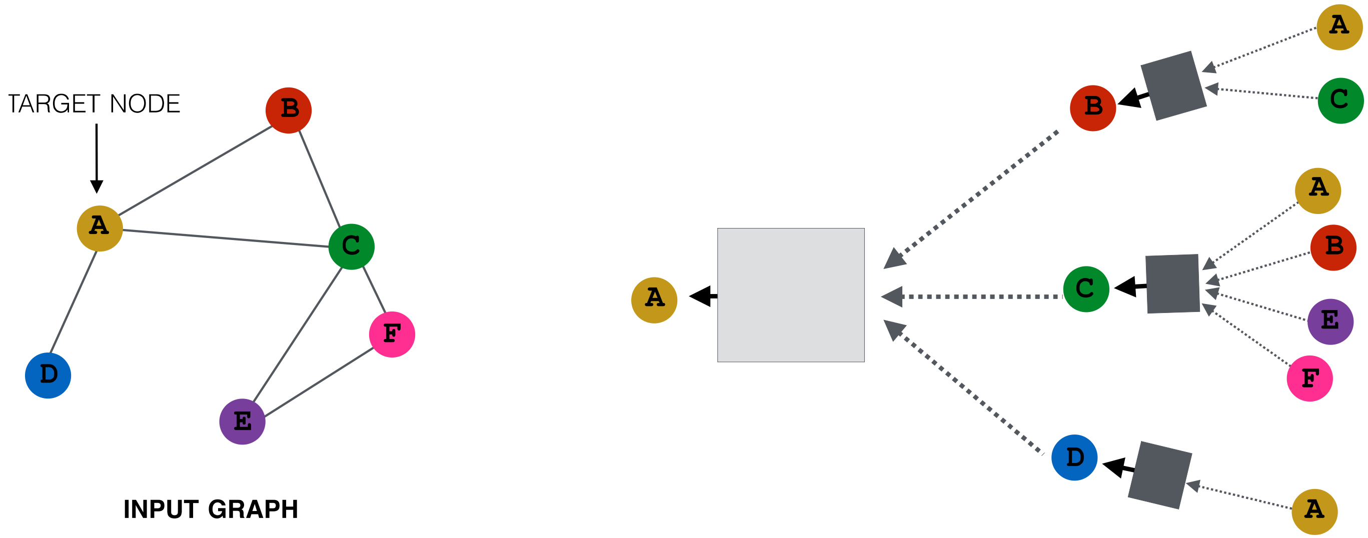
Hamilton et al. Inductive Representation Learning on Large Graphs, NeurIPS 2017

Node Neighborhood Sampling

Our approach so far:

– All the neighbors are used for message passing

- **Problem: Dense/large graphs, high-degree nodes**

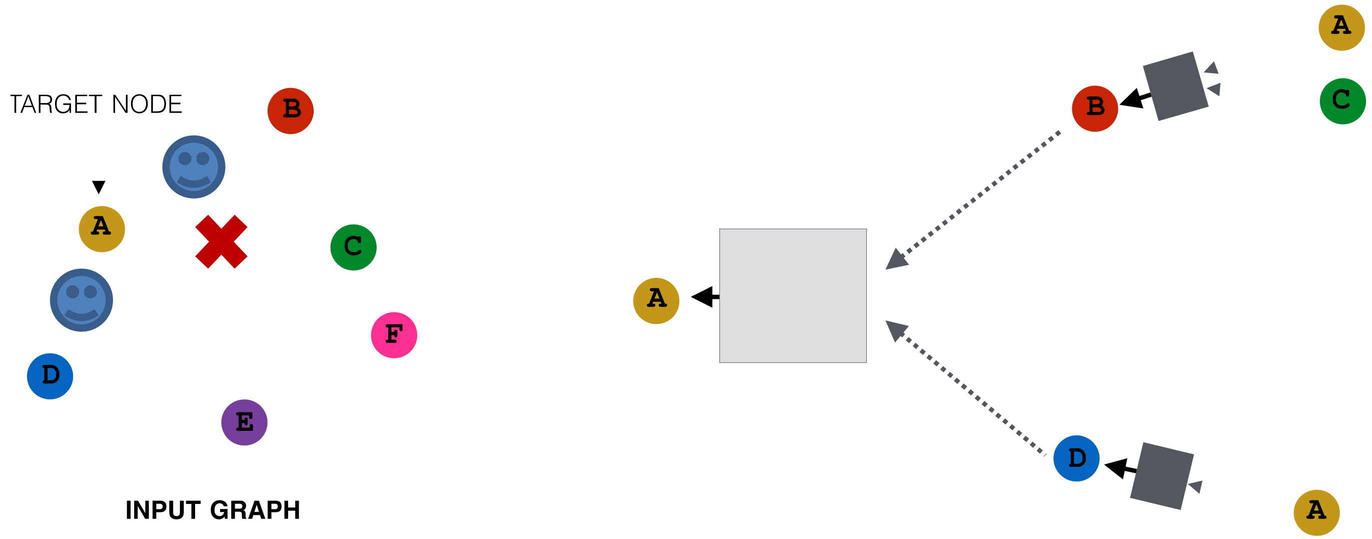


New idea: (Randomly) determine a node's neighborhood for message passing

Neighborhood Sampling Example

For example, we can randomly choose 2 neighbors to pass messages

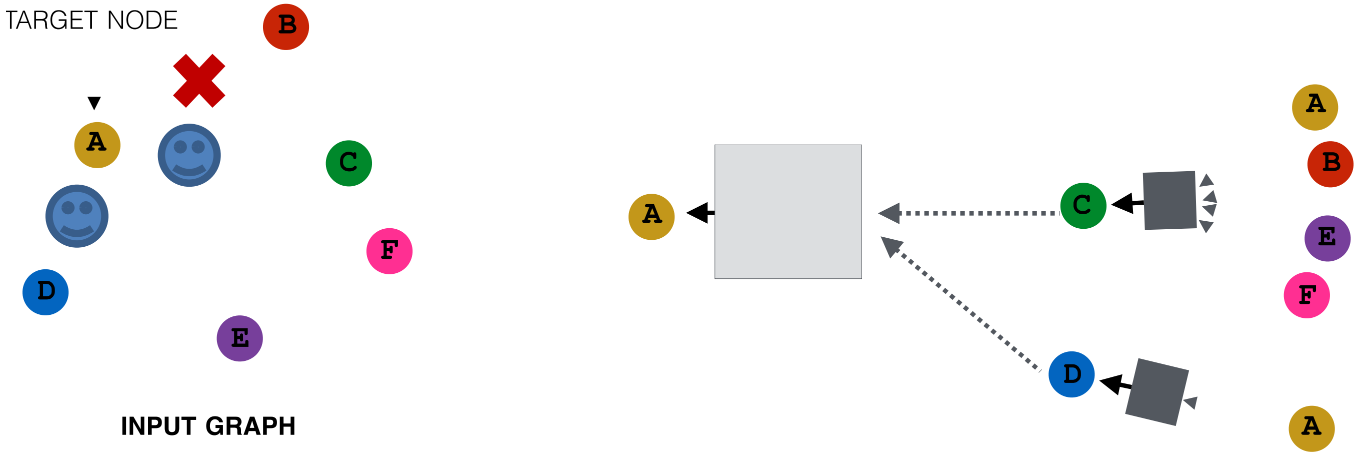
- Only nodes B and D will pass message to A



Neighborhood Sampling Example

Next time when we compute the embeddings, we can sample different neighbors

- Only nodes C and D will pass message to A

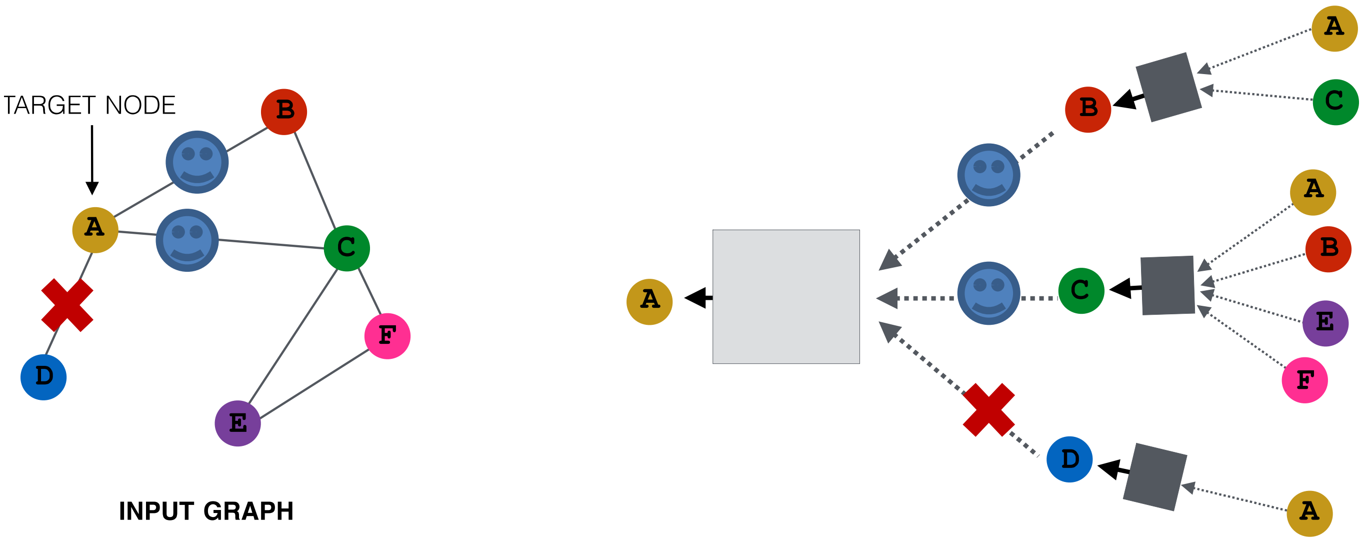


Ying et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems, KDD 2018

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!



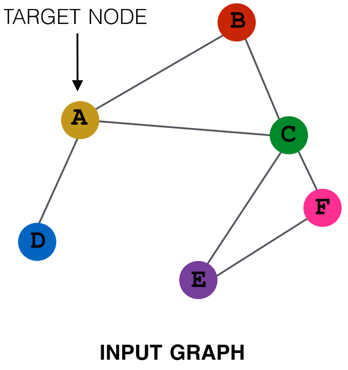
Outline

- General Framework
- A single GNN layer: Aggregation and Message
- Layer Connectivity: Stacking
- Graph augmentation
- Learning objectives

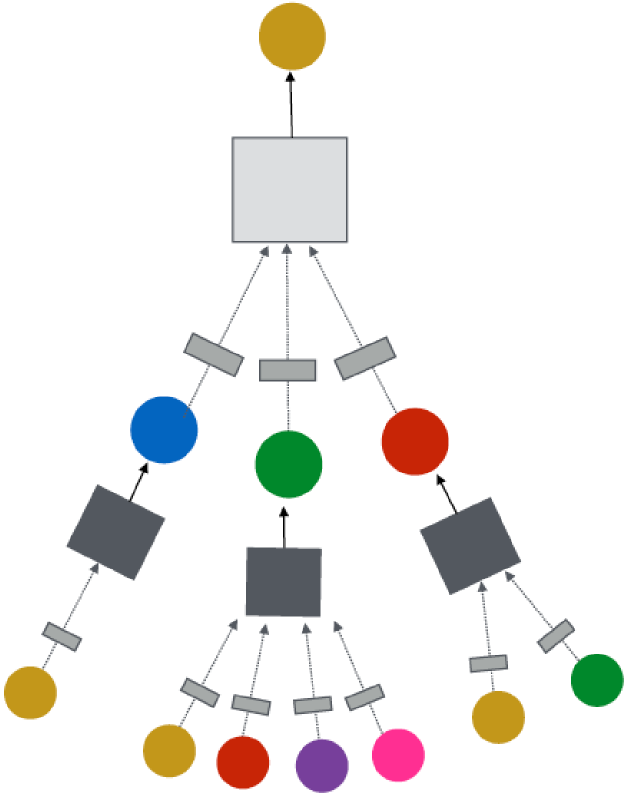
LEARNING WITH GNNS

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

A General GNN Framework



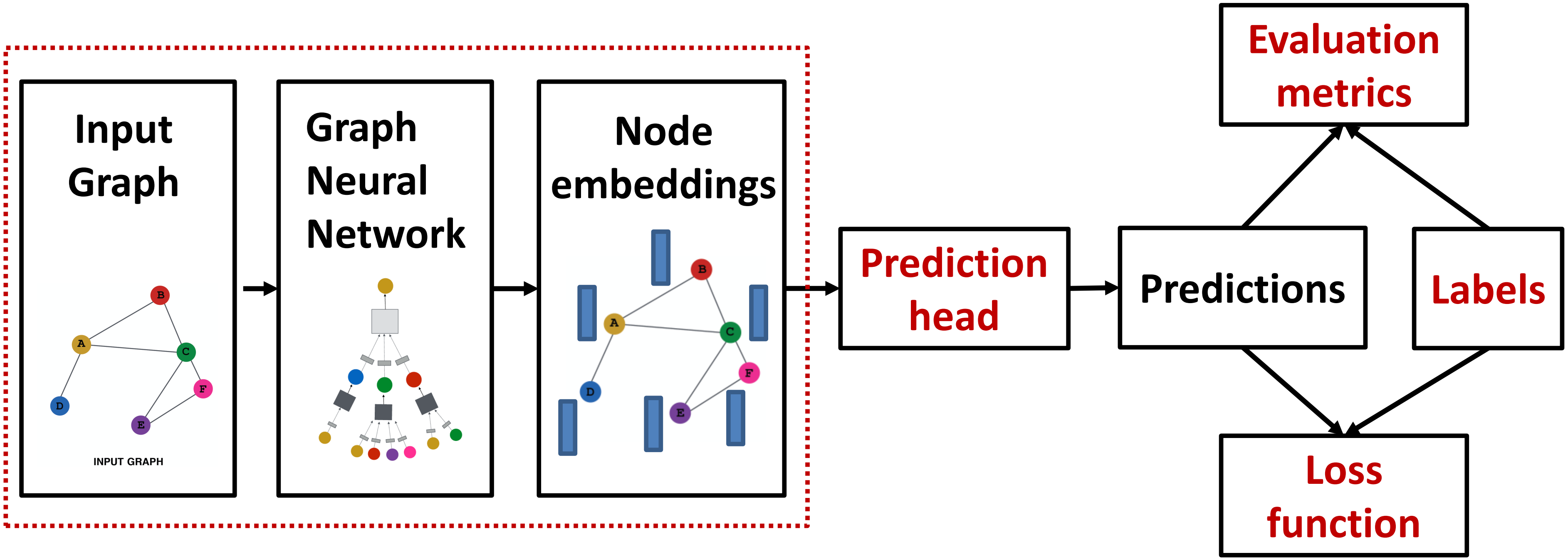
(5) Learning objective



How do we train a GNN?

GNN Training Pipeline

So far what we have covered



Input Graph

Graph Neural Network

Node embeddings

Prediction head

Predictions

Labels

Evaluation metrics

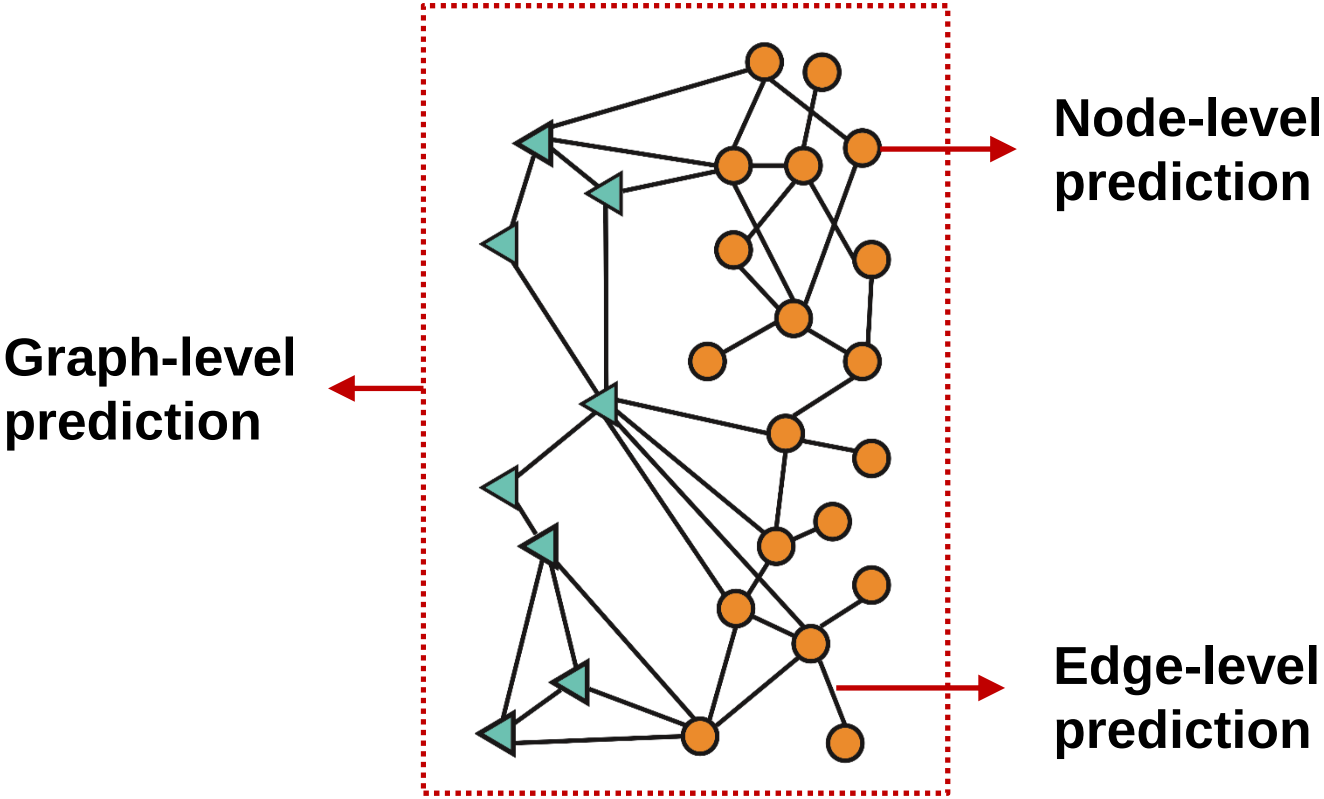
Loss function

Output of a GNN: set of node embeddings

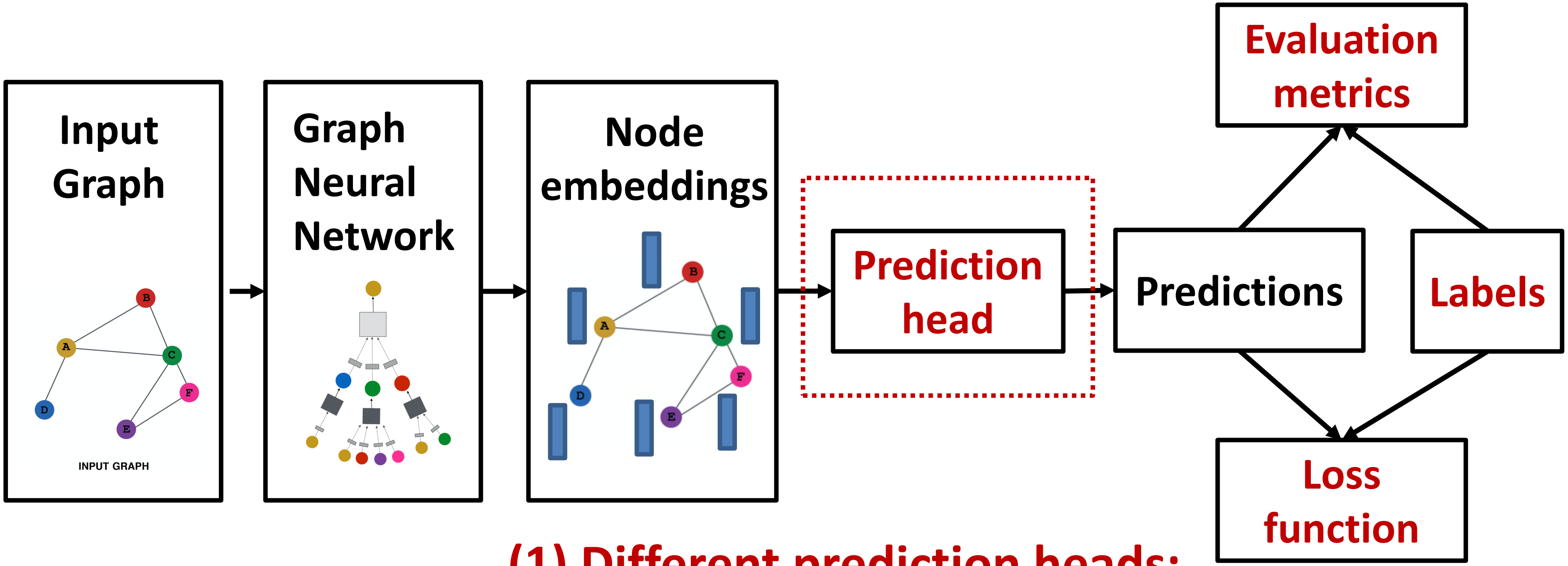
$$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$$

GNN Prediction Heads

Idea: Different task levels require different prediction heads



GNN Training Pipeline (1)



(1) Different prediction heads:

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

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

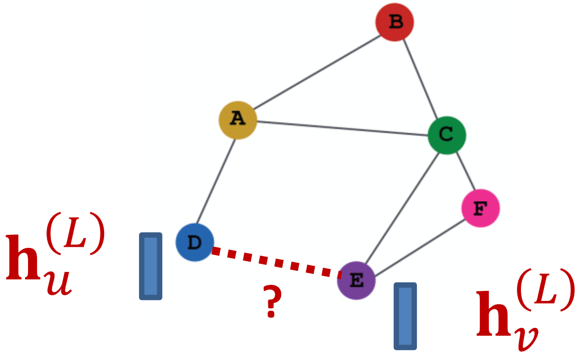
Output of the classifier

Prediction Heads: Edge-level

Edge-level prediction: Make prediction using pairs of node embeddings

- Suppose we want to make k-way prediction

$$\widehat{\boldsymbol{y}}_{\boldsymbol{uv}} = \mathrm{Head}_{\mathrm{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$$



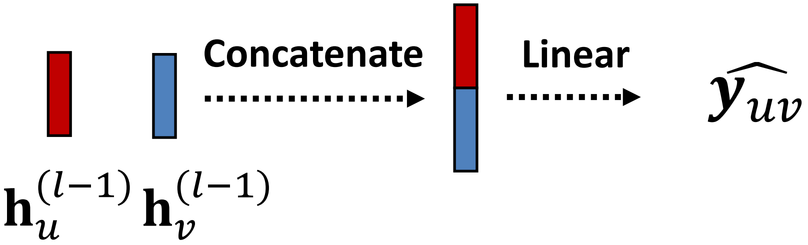
- What are the options for $\mathrm{Head}_{\mathrm{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$?

Prediction Heads: Edge-level

- **Options for** $\mathrm{Head}_{\mathrm{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:

(1) Concatenation + Linear

– We have seen this in graph attention



– $\widehat{\boldsymbol{y}}_{\boldsymbol{uv}} = \mathrm{Linear}(\mathrm{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$

– Here $\mathrm{Linear}(\cdot)$ 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

- $\widehat{\boldsymbol{y}}_{\boldsymbol{uv}} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
- **This approach only applies to 1-way prediction** (e.g., link prediction: predict the existence of an edge)
- **Applying to $\boldsymbol{k}$-way prediction:**
  - Similar to **multi-head attention**: $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(k)}$ trainable

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

$$\dots$$

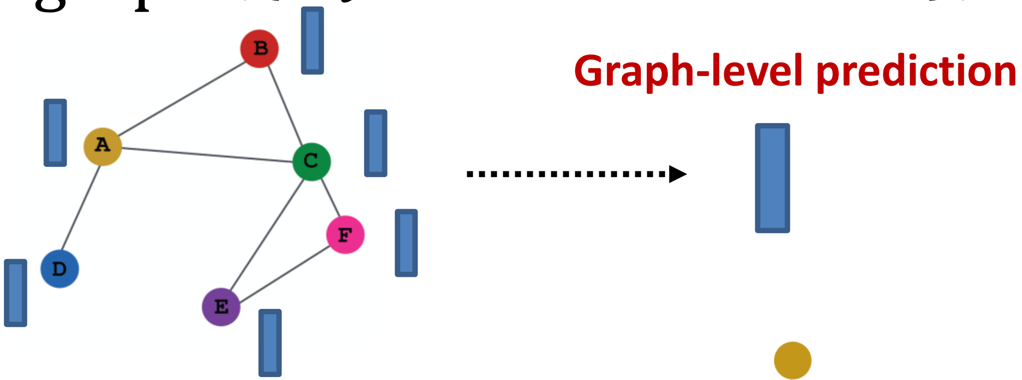
$$\widehat{\boldsymbol{y}}_{\boldsymbol{uv}}^{(\boldsymbol{k})} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$

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

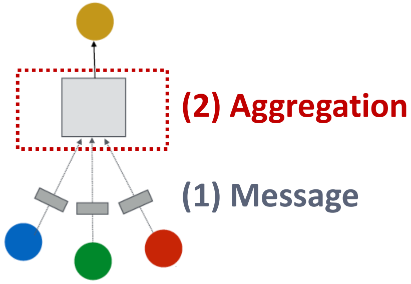
Prediction Heads: Graph-level

Graph-level prediction: Make prediction using all the node embeddings in our graph

- Suppose we want to make k-way prediction
- $\hat{\boldsymbol{y}}_G = \mathrm{Head}_{\mathrm{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$



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



K. Xu, W. Hu, J. Leskovec, S. Jegelka. How Powerful Are Graph Neural Networks, ICLR 2019

Prediction Heads: Graph-level

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

- **(1) Global mean pooling**

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

- **(2) Global max pooling**

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

- **(3) Global sum pooling**

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

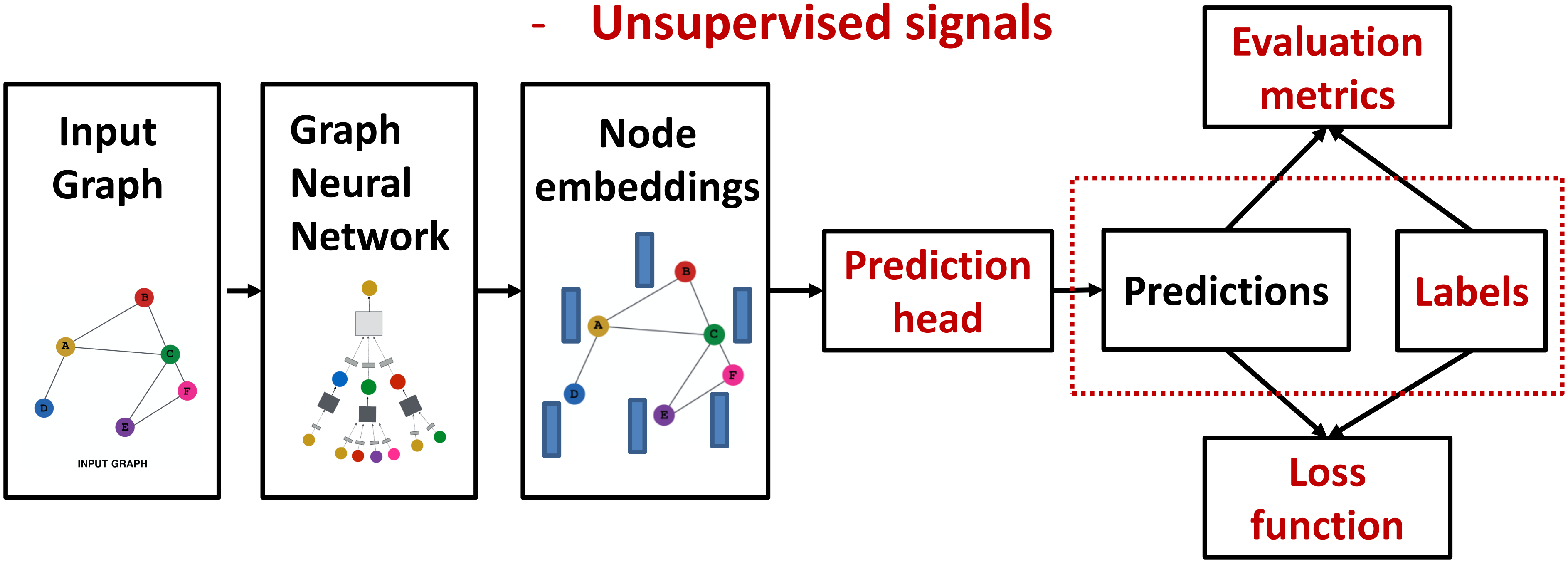
- These options work great for small graphs

For large graphs, hierarchical aggregation

GNN Training Pipeline (2)

(2) Where does ground-truth come from?

- **Supervised labels**
- **Unsupervised signals**



Input Graph

Graph Neural Network

Node embeddings

Prediction head

Predictions

Labels

Evaluation metrics

Loss function

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 dishonest
  - **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:**

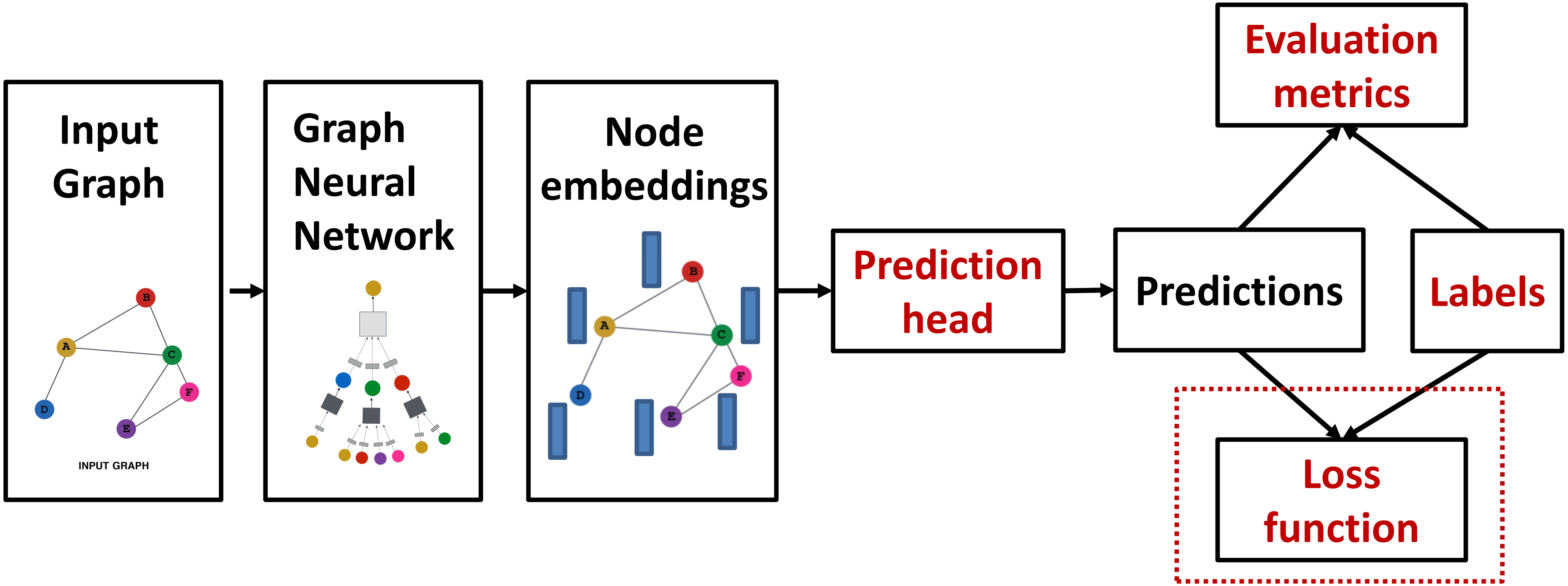
  – **Node-level $\boldsymbol{y_v}$.** Node statistics: such as clustering coefficient, PageRank, …

  – **Edge-level $\boldsymbol{y_{uv}}$.** Link prediction: hide the edge between two nodes, predict if there should be a link

  – **Graph-level $\boldsymbol{y_G}$.** 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 $\widehat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
  - **Edge-level**: prediction $\widehat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
  - **Graph-level**: prediction $\widehat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$
  - We will use prediction $\widehat{\boldsymbol{y}}^{(i)}$, 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

Cross entropy (CE) is a very common loss function in classification

- K-way prediction for i-th data point:

$$\mathrm{CE}\big(\underbrace{\boxed{\boldsymbol{y}^{(i)}}}_{\text{Label}}, \underbrace{\boxed{\hat{\boldsymbol{y}}^{(i)}}}_{\text{Prediction}}\big) = -\sum_{j=1}^{K} \boldsymbol{y}_j^{(i)} \log(\hat{\boldsymbol{y}}_{\boxed{j}}^{\boxed{(i)}})$$

***i*-th data point**

***j*-th class**

where:

E.g.

0	0	1	0	0

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

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

E.g.

0.1	0.3	0.4	0.1	0.1

- Total loss over all N training examples

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

Regression Loss

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

$$\mathrm{MSE}\left(\boldsymbol{y}^{(i)}, \widehat{\boldsymbol{y}}^{(i)}\right) = \sum_{j=1}^{K} \left(\boldsymbol{y}_j^{(i)} - \widehat{\boldsymbol{y}}_{\boxed{j}}^{\boxed{(i)}}\right)^2$$

***i*-th data point**

***j*-th target**

where:

E.g.

1.4	2.3	1.0	0.5	0.6

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

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

E.g.

0.9	2.8	2.0	0.3	0.8

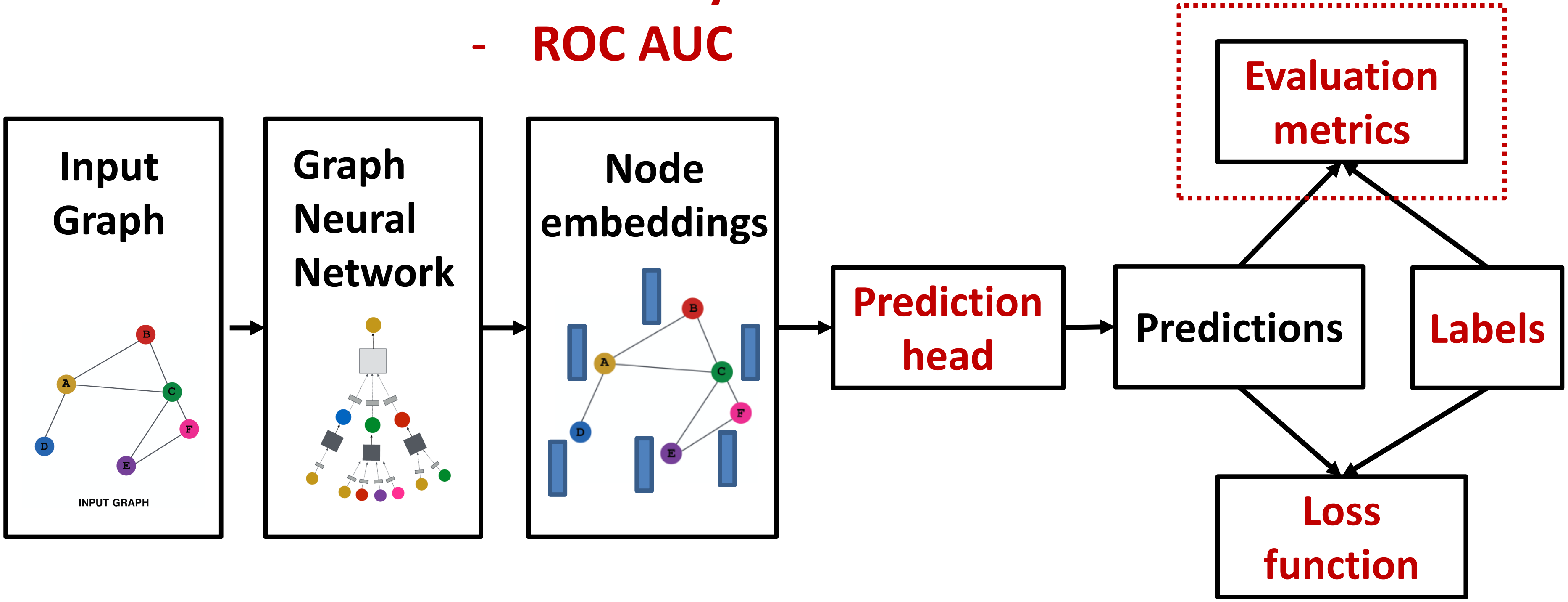
- Total loss over all N training examples

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

GNN Training Pipeline (4)

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

- **Accuracy**
- **ROC AUC**



Input Graph

Graph Neural Network

Node embeddings

Prediction head

Predictions

Labels

Evaluation metrics

Loss function

Evaluation Metrics: Regression

- **We use standard evaluation metrics for GNN**
  - 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)**

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

  - **Mean absolute error (MAE)**

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

Evaluation Metrics: Classification

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

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

- **(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
    - **OC AUC**

Metrics for Binary Classification

- **Accuracy:**

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

- **Precision (P):**

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

- **Recall (R):**

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

- **F1-Score:**

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

Confusion matrix

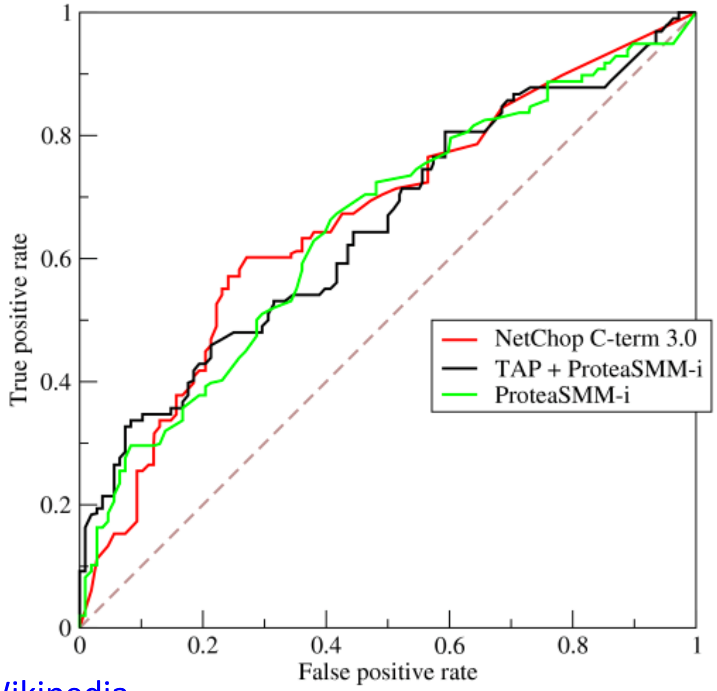
	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

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

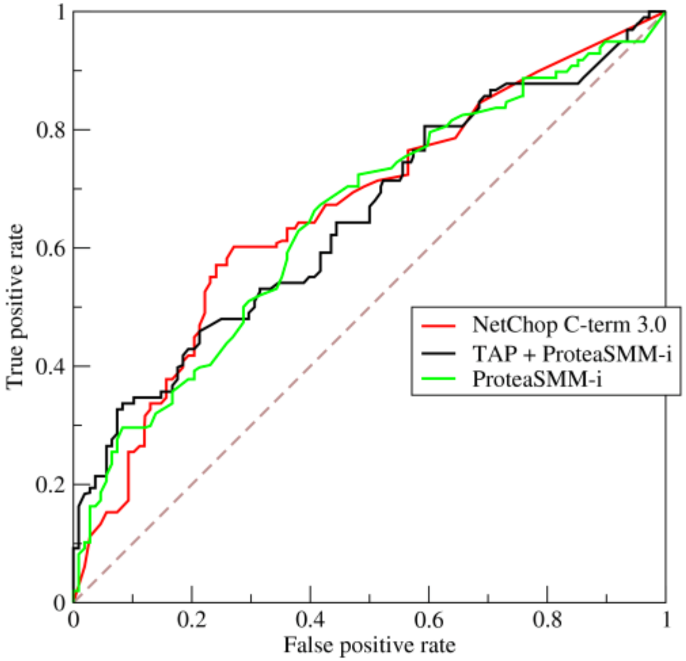
Image Credit: Wikipedia

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

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

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

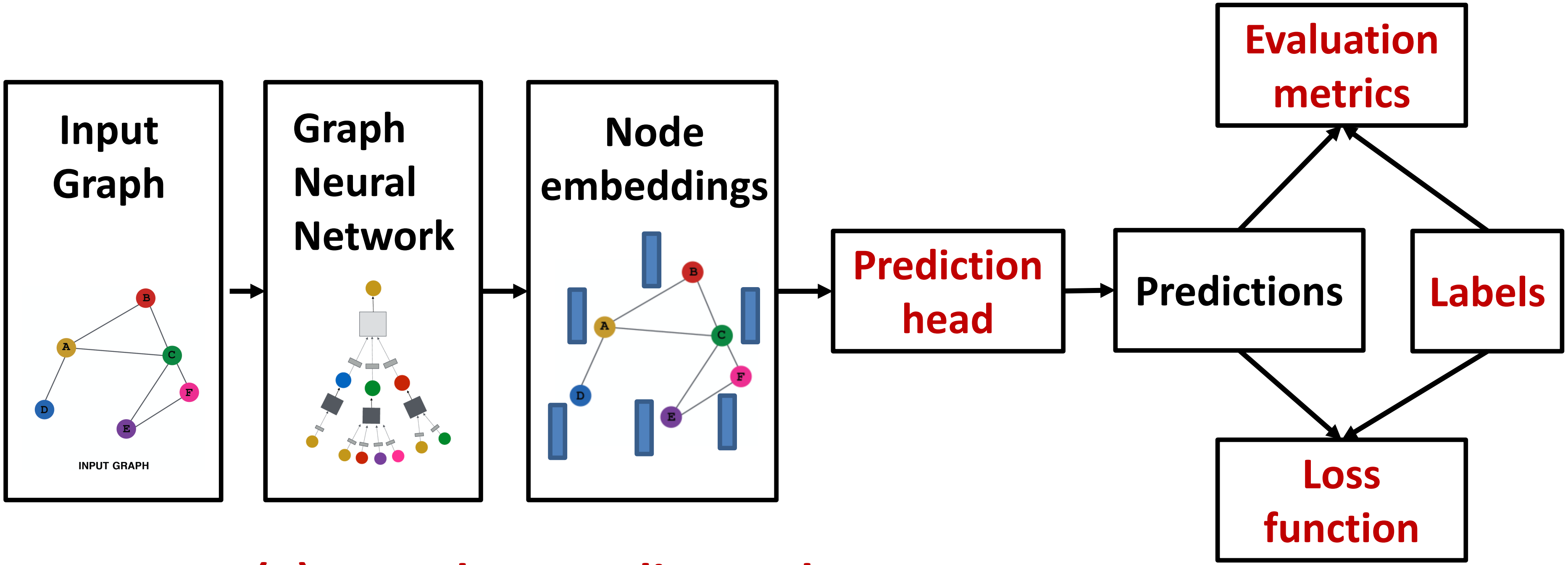
(4) Evaluation Metrics



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

GNN Training Pipeline (5)



Input Graph

Graph Neural Network

Node embeddings

Prediction head

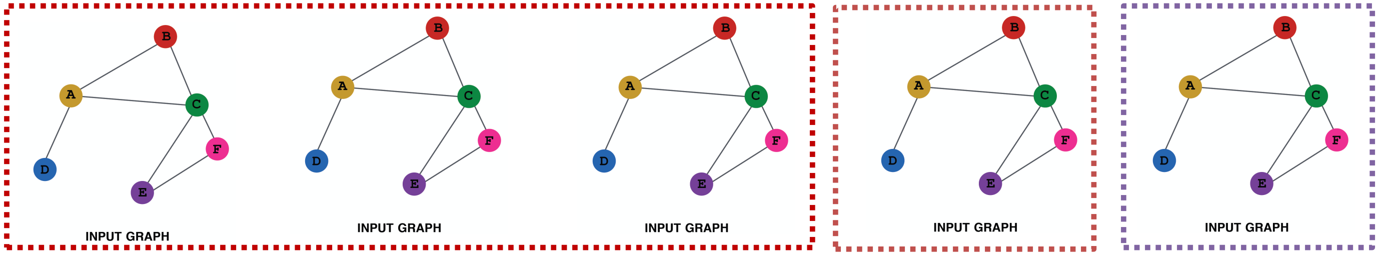
Predictions

Labels

Evaluation metrics

Loss function

(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

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

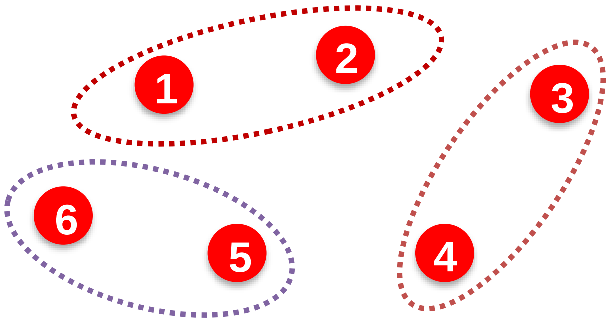
Why Splitting Graphs is Special

- **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

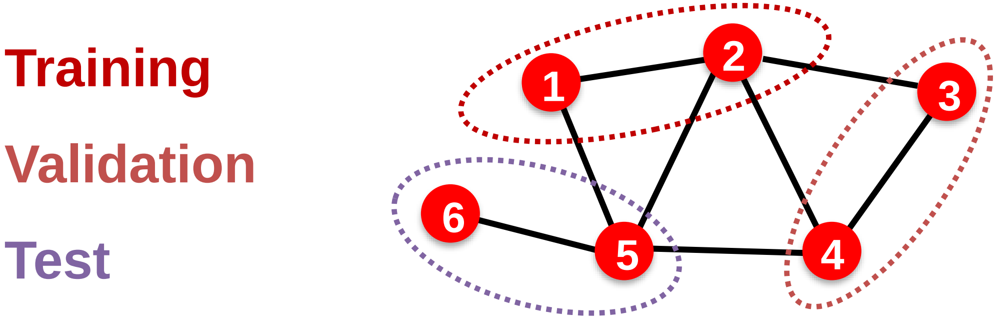
Validation

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

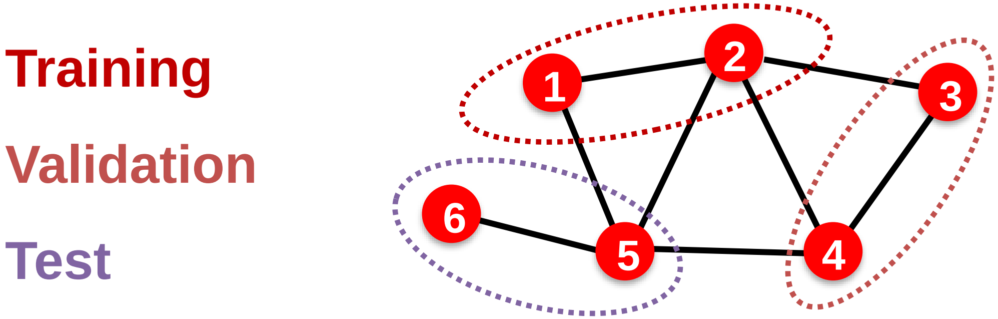
Test

- **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).

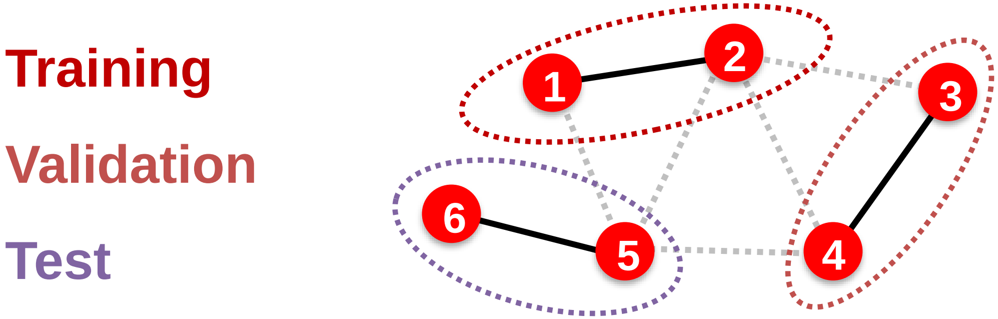
- **We will only split the (node) labels**
  - **At training time**, we compute embeddings **using the entire graph**, and train **using node 1&2's labels**
  - **At validation time**, we compute embeddings **using the entire graph**, and **evaluate on node 3&4's labels**



Why Splitting Graphs is Special

Solution 2 (Inductive setting): We break the edges between splits to get multiple graphs

- **Now we have 3 graphs that are independent. Node 5 will not affect our prediction on node 1 any more**
- **At training time**, we compute embeddings **using the graph over node 1&2**, and train **using node 1&2's labels**
- **At validation time**, we compute embeddings **using the graph over node 3&4**, and **evaluate on node 3&4's labels**

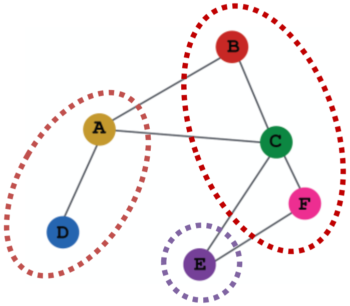


Transductive/Inductive Settings

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

Example: Node Classification

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

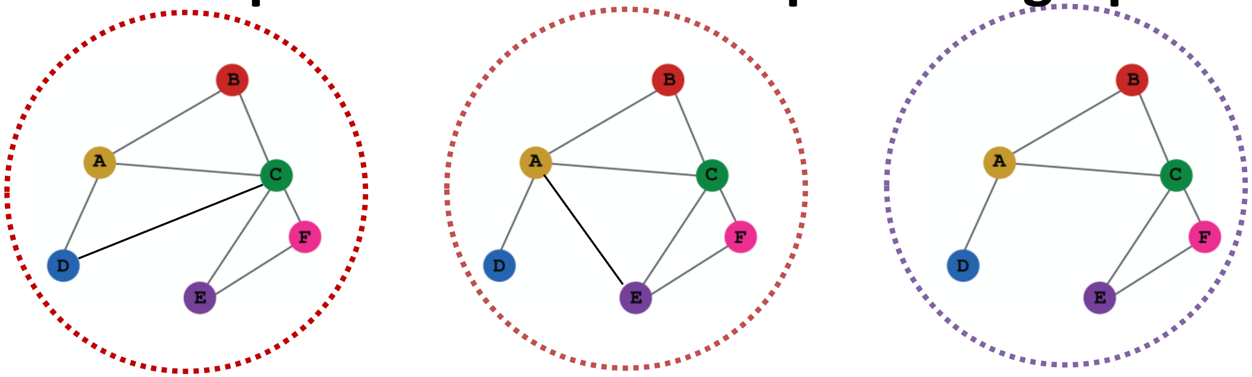


Training

Validation

Test

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



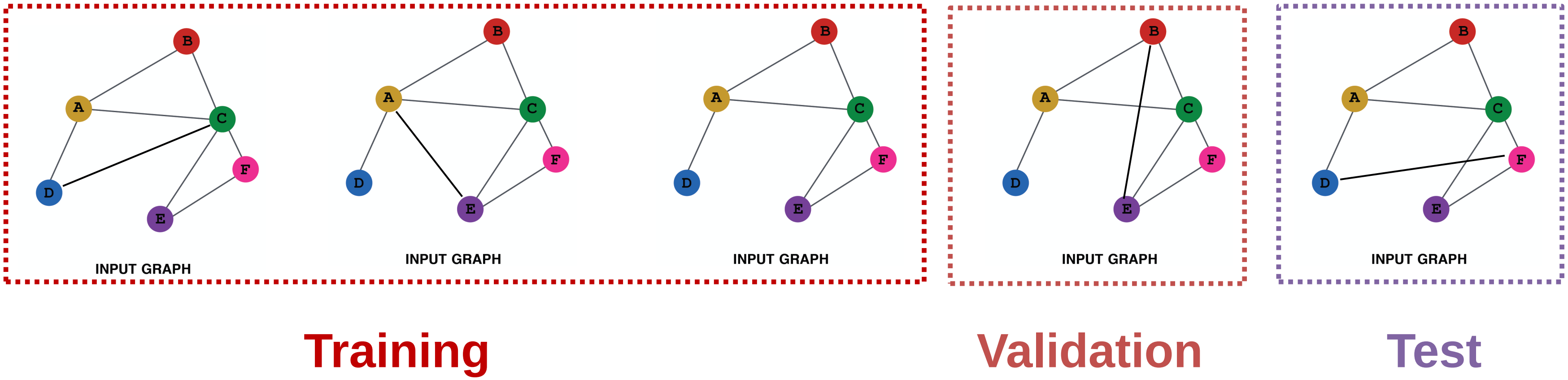
Training

Validation

Test

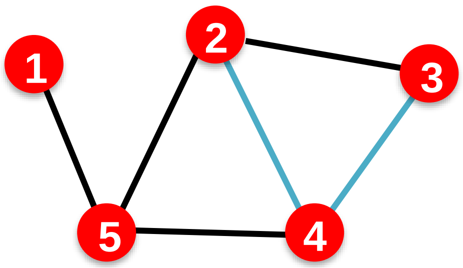
Example: Graph Classification

- 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).

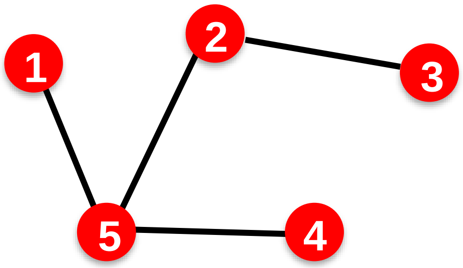


Example: Link Prediction

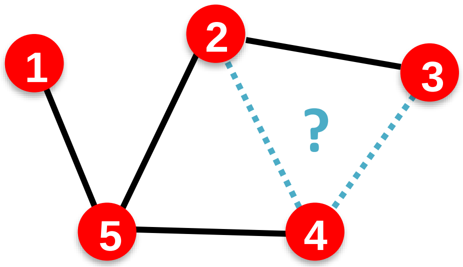
- **Goal of link prediction: predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised/self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and the **let the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



Original graph

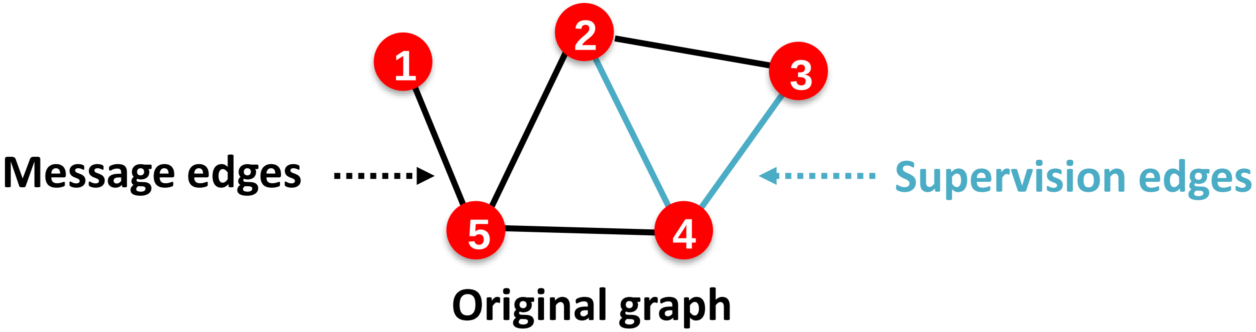


Input graph to GNN



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

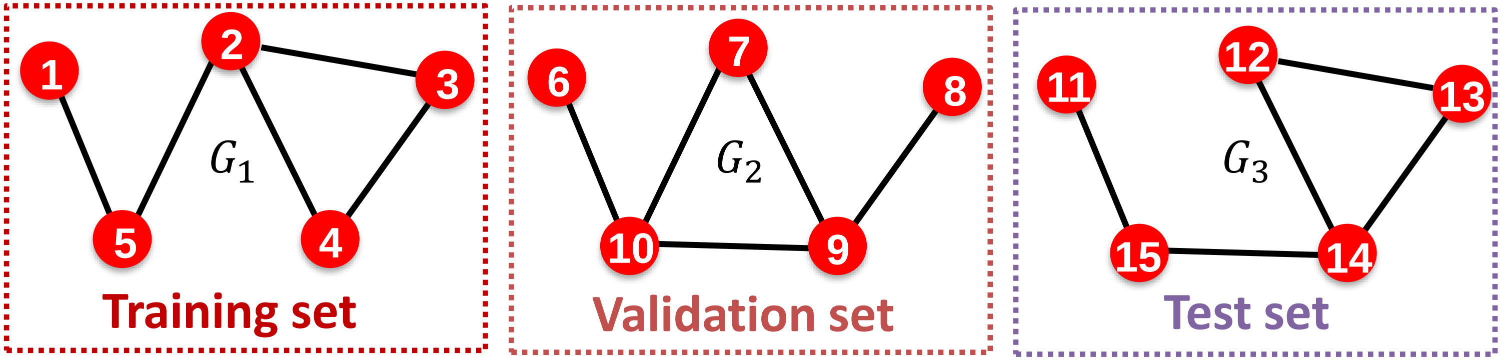
- Message edges: Used for GNN message passing
- Supervision edges: Use for computing objectives

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**

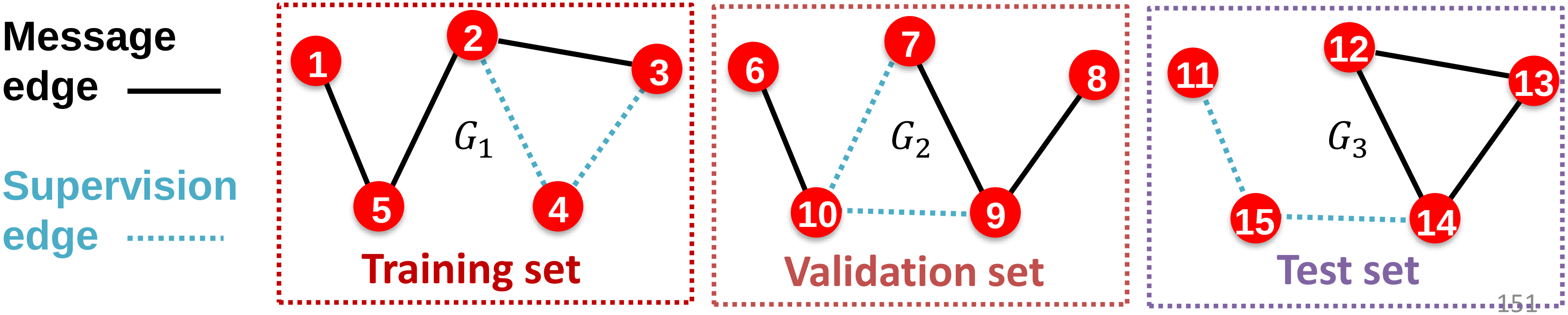


Setting up Link Prediction

- **Step 2: Split edges into train/validation/test**

Option 1: Inductive link prediction split

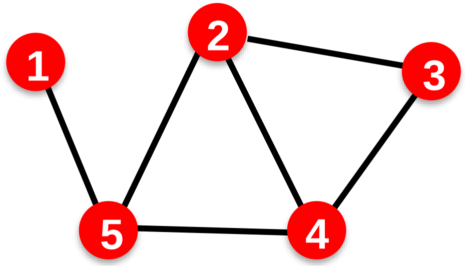
- **Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph**
- **In train** or **val** or **test** set, **each graph will have 2 types of edges**: **message edges** + **supervision edges**
  - **Supervision edges** are not the input to GNN



Setting up Link Prediction

Option 2: Transductive link prediction split:

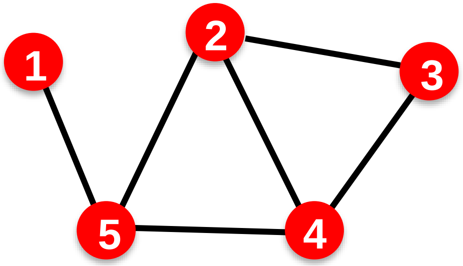
- **This is the <u>default</u> setting when people talk about link prediction**
- **Suppose we have a dataset of 1 graph**



Setting up Link Prediction

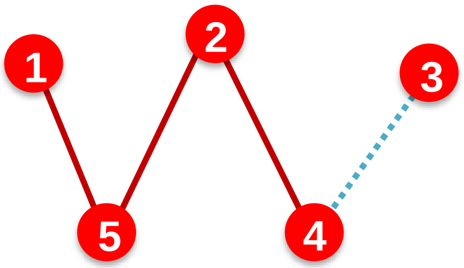
Option 2: Transductive link prediction split:

- 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

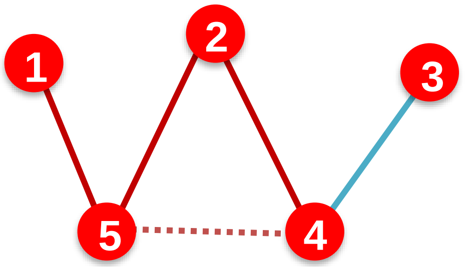


Setting up Link Prediction

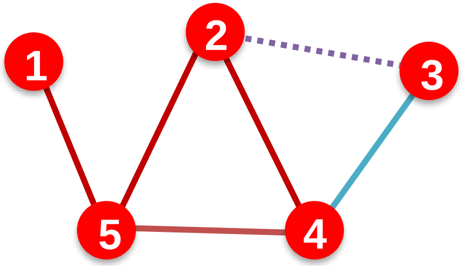
Option 2: Transductive link prediction split:



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



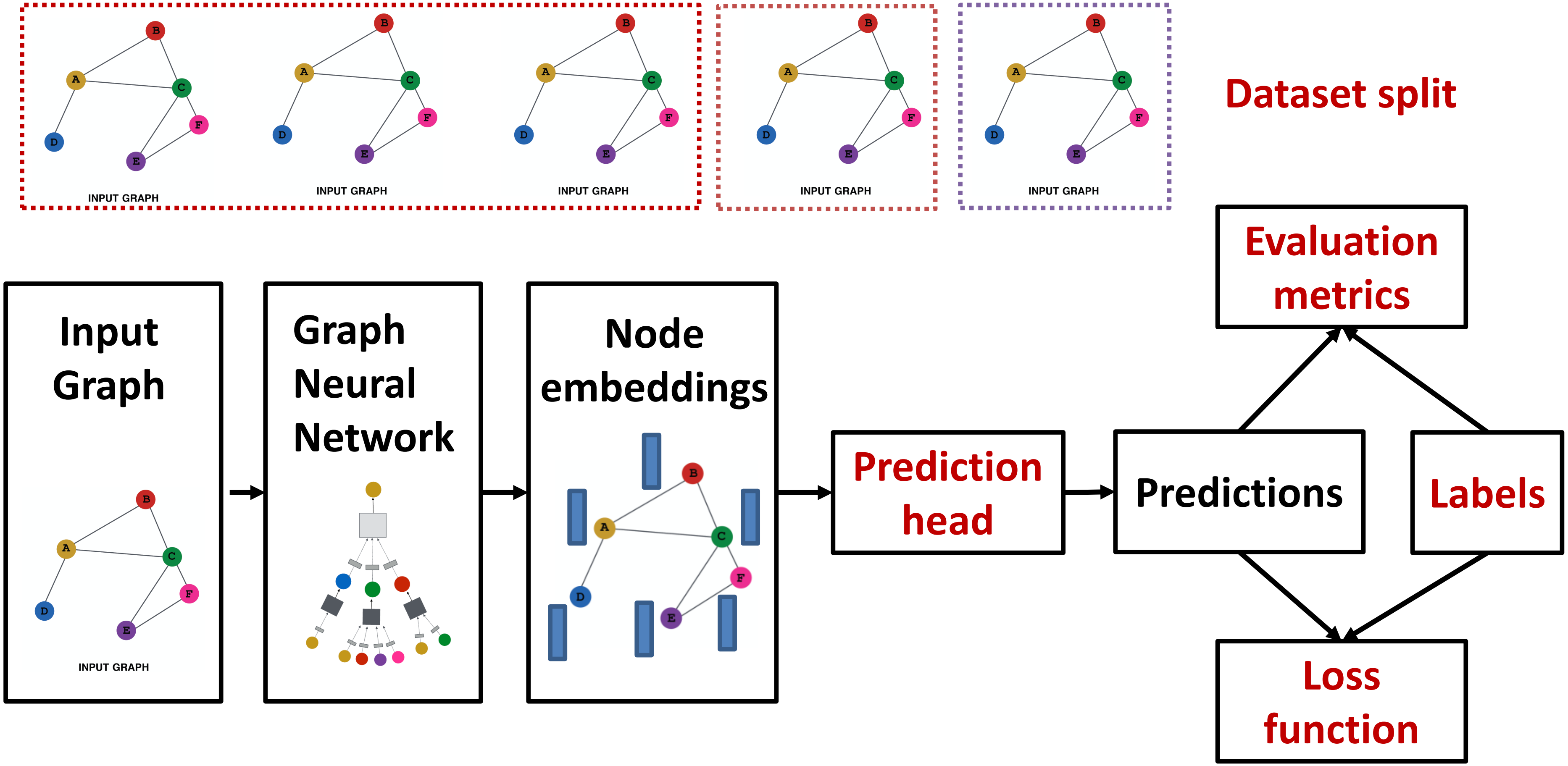
(2) At validation time:
Use **training message edges & training supervision edges** to predict **validation edges**



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

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.

GNN Training Pipeline



Implementation resources:

GraphGym further implements the full pipeline to facilitate GNN design

Summary

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

Acknowledgement

Most slides from

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