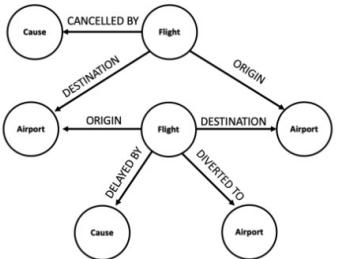


# Graph neural networks

Content draws on material from Jure Leskovec's [CS244W](#)

# Many types of data are graphs

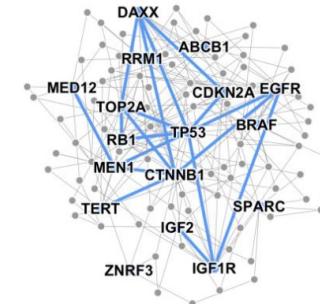


**Event Graphs**



Image credit: [SalientNetworks](#)

**Computer Networks**



**Disease Pathways**

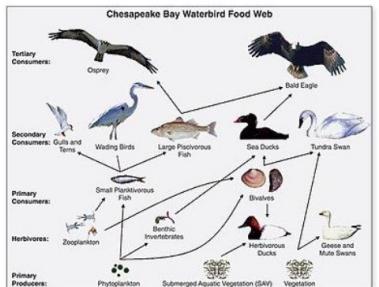


Image credit: [Wikipedia](#)

**Food Webs**



Image credit: [Pinterest](#)

**Particle Networks**



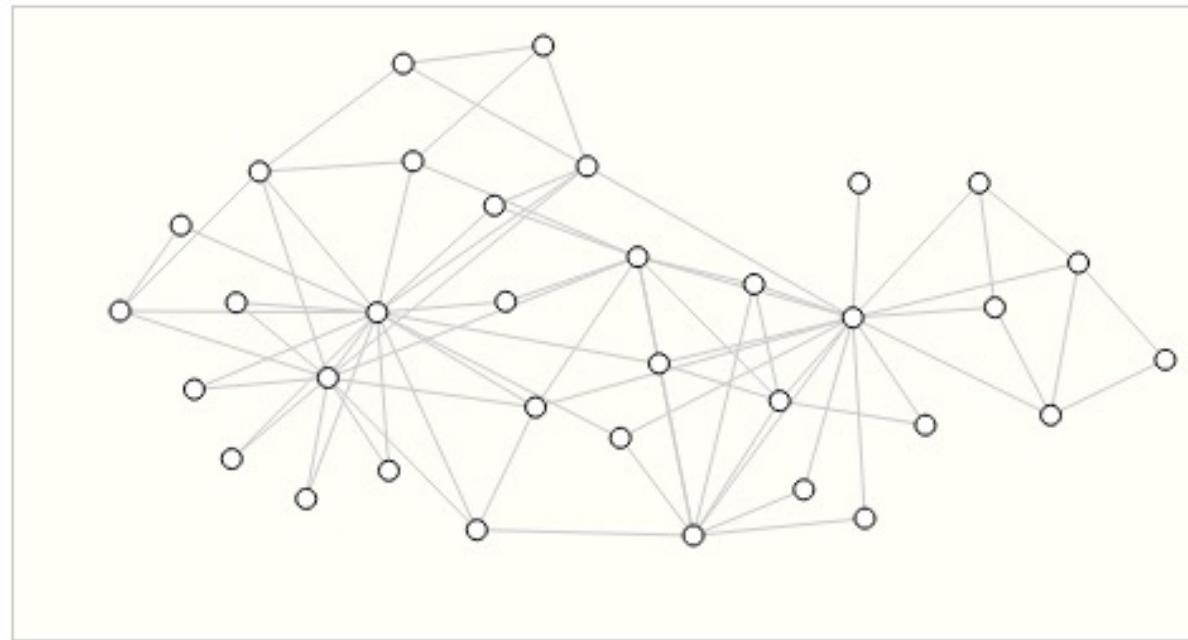
Image credit: [visitlondon.com](#)

**Underground Networks**

# GNN motivation

**Main question:**

How to utilize relational structure for better prediction?



# Today: Modern ML toolbox

Modern DL toolbox is designed for simple sequences & grids

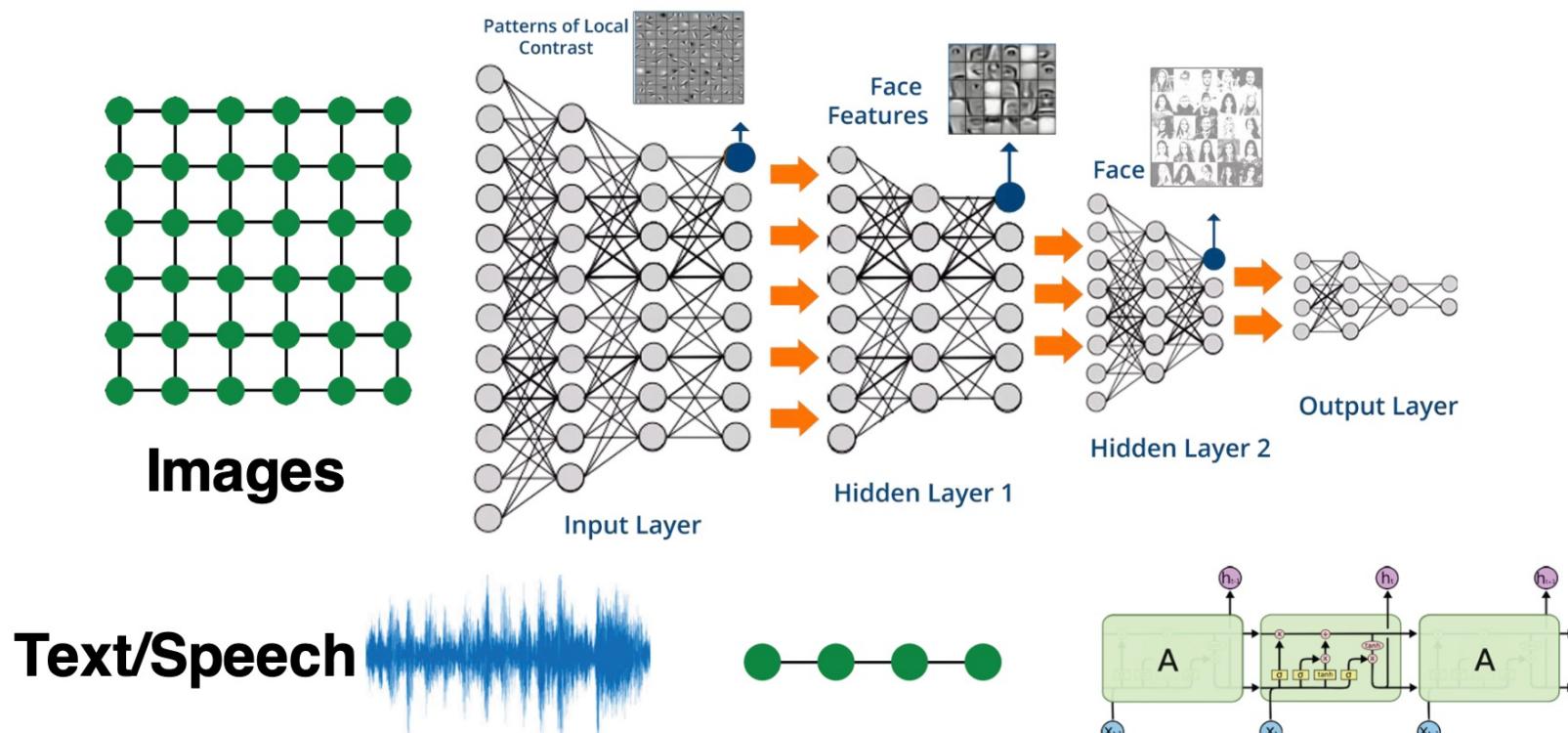
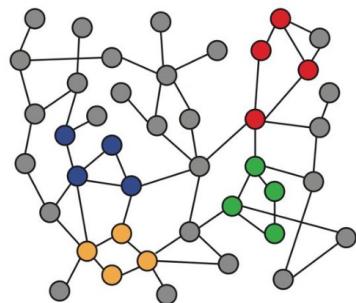


Figure by Leskovec

# Why is graph deep learning hard?

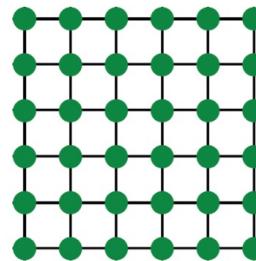
Networks are complex

- Arbitrary size and complex topological structure



**Networks**

versus



**Images**



**Text**

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

# Different types of tasks

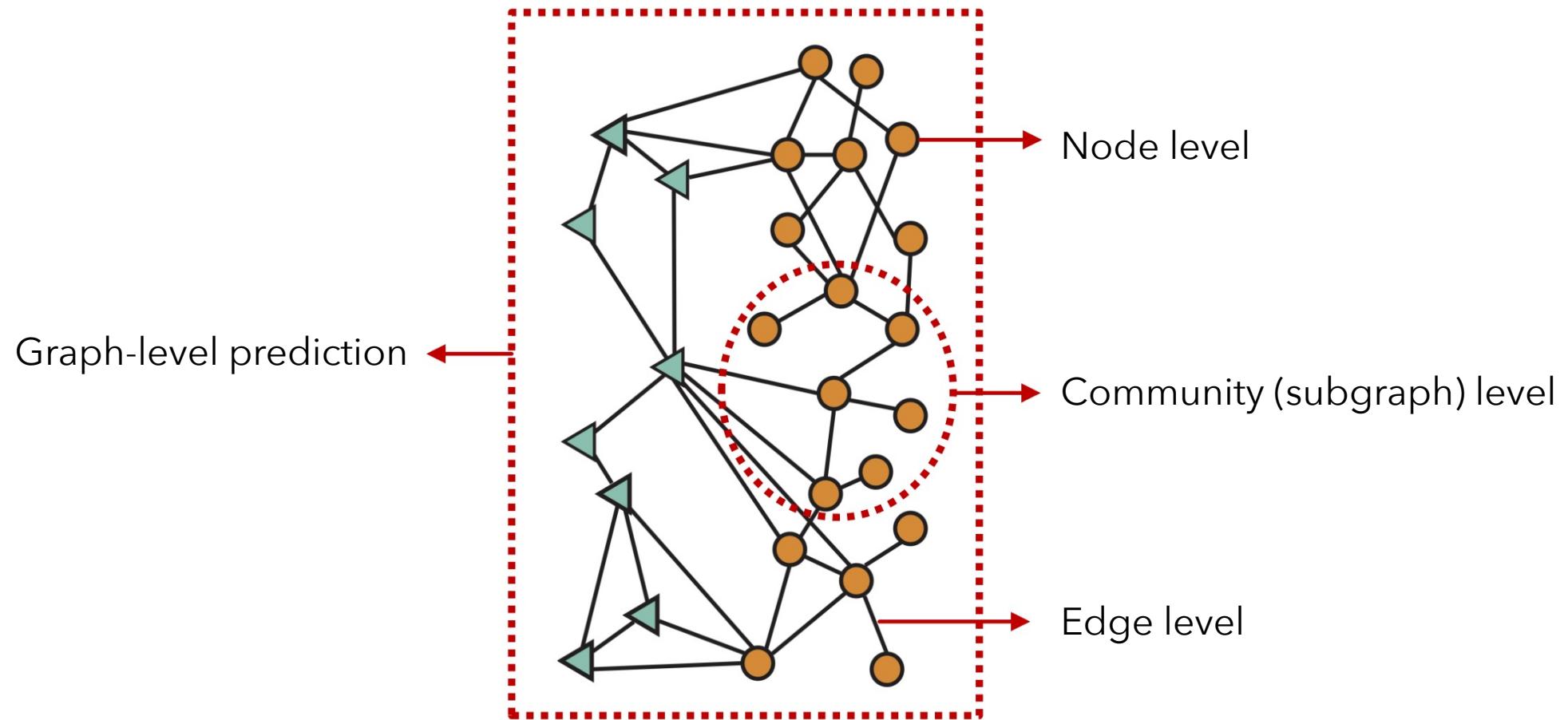
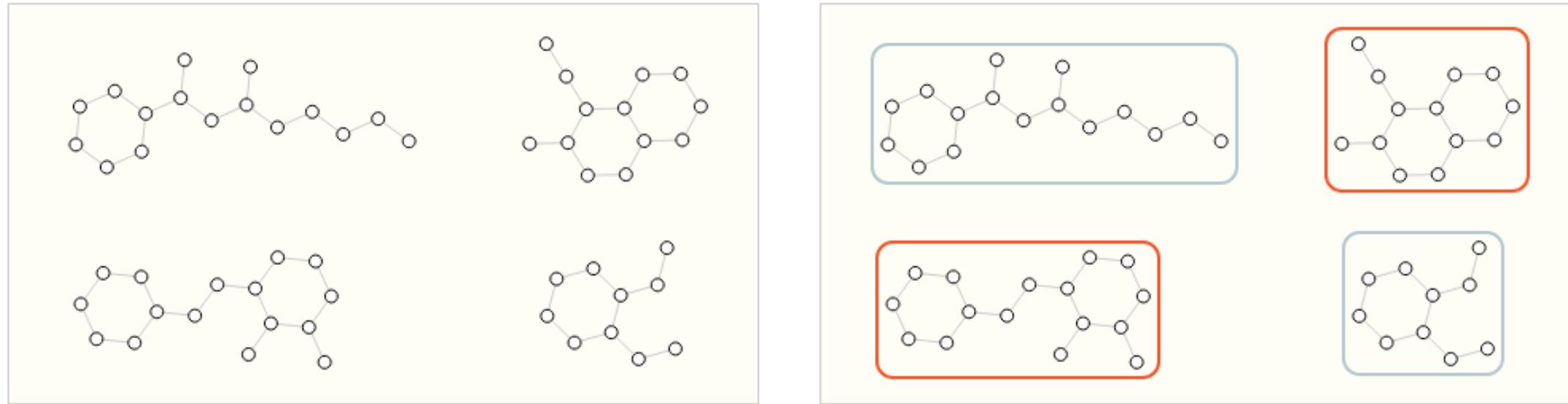


Figure by Leskovec

# Prediction with graphs: Examples

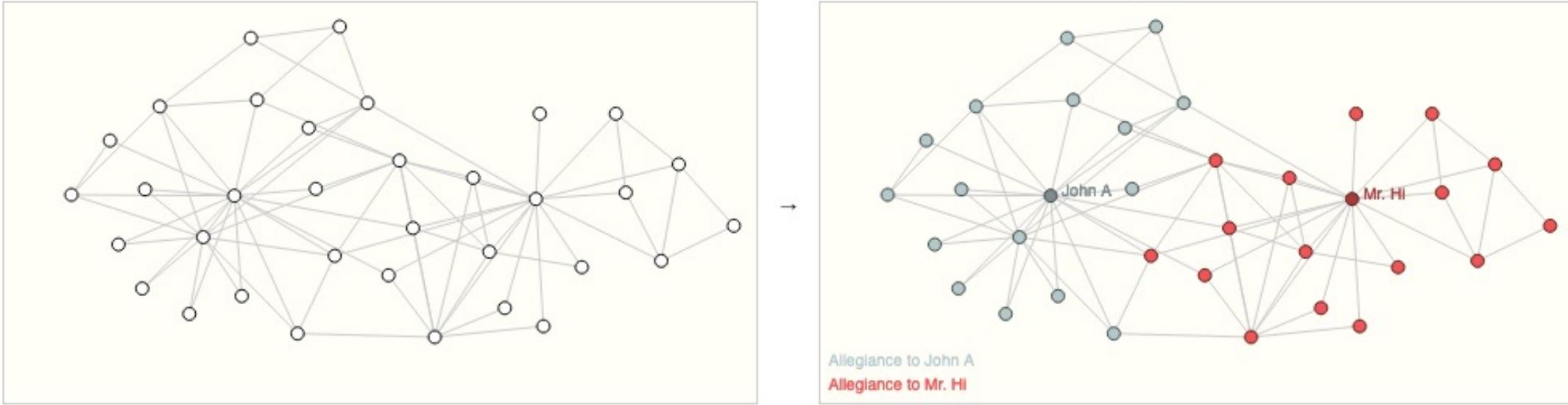


## Graph-level tasks:

E.g., for a molecule represented as a graph, could predict:

- What the molecule smells like
- Whether it will bind to a receptor implicated in a disease

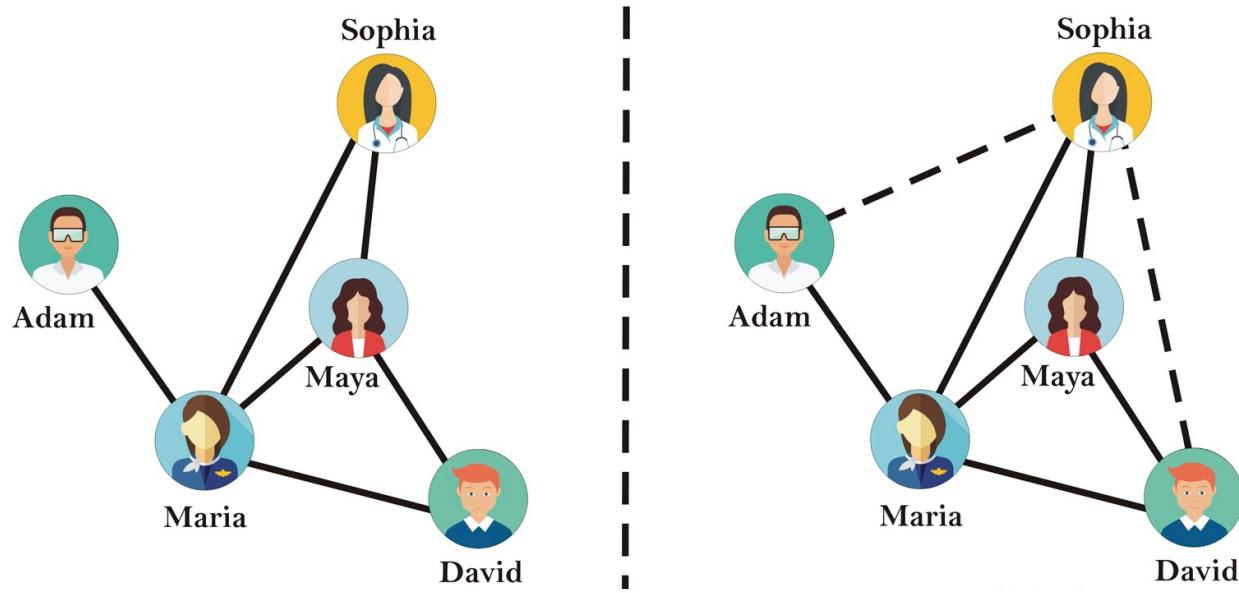
# Prediction with graphs: Examples



## **Node-level tasks:**

E.g., political affiliations of users in a social network

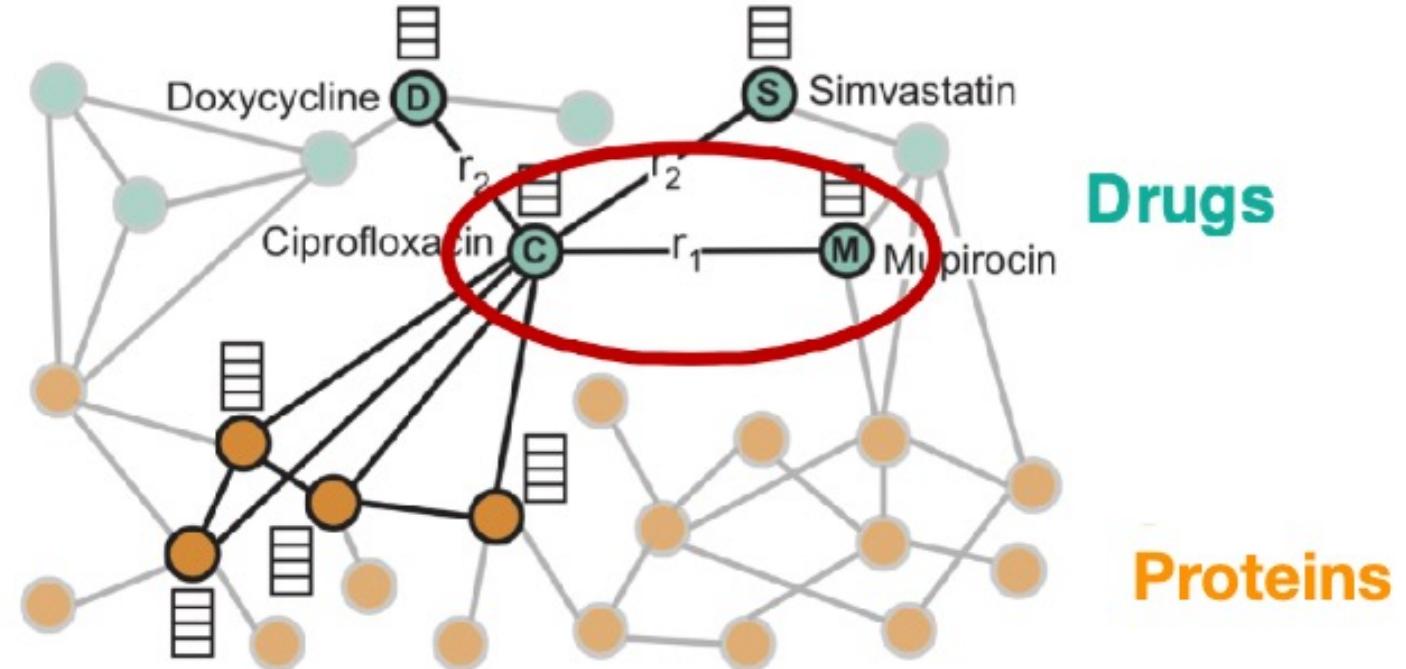
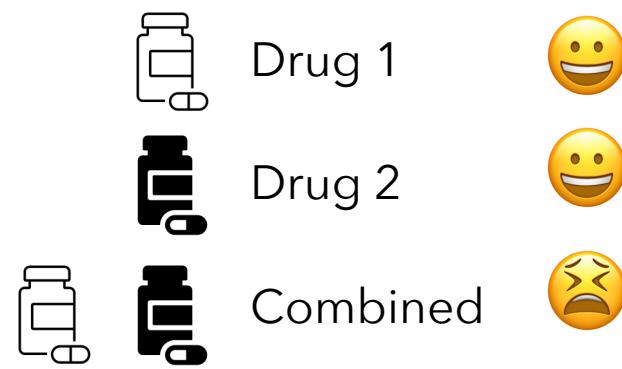
# Prediction with graphs: Examples



**Edge-level tasks:** E.g.:

- Suggesting new friends
- Recommendations on Amazon, Netflix, ...

# Example: Polypharmacy side effects



**Nodes:** Drugs  
**Edges:** Interaction type

# Example: Traffic routing



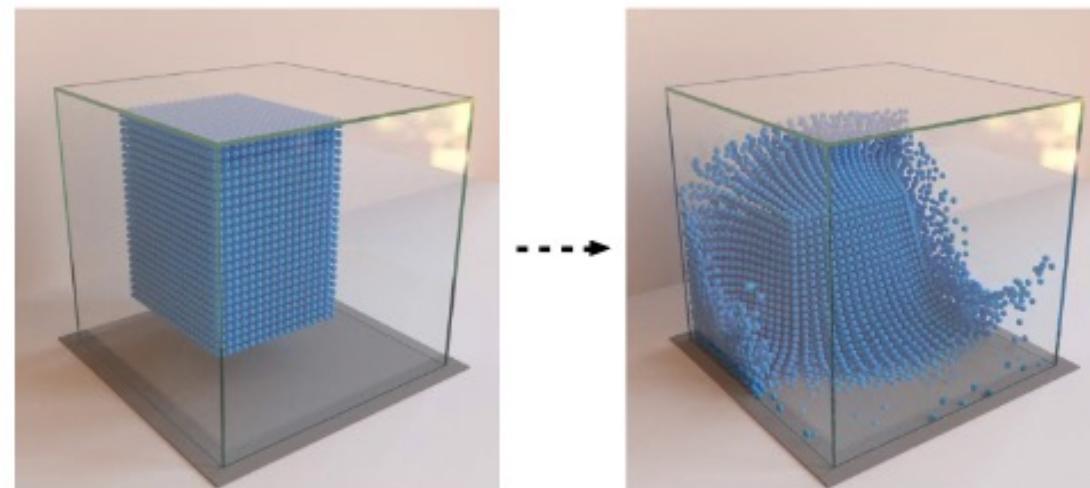
E.g., Google maps

[deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks](https://deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks)

# Example: Learning to simulate physics

**Nodes:** Particles

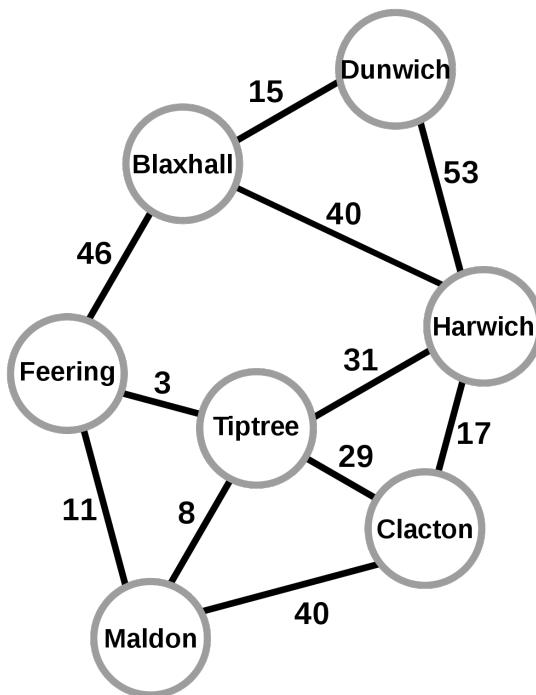
**Edges:** Interaction between particles



**Goal:** Predict how a graph will evolve over time

# Example: Combinatorial optimization

Replace full algorithm or learn steps (e.g., branching decision)



$$\begin{aligned} &\text{maximize} \quad c \cdot z \\ &\text{subject to} \quad Az \leq b \\ &\quad z \in \mathbb{Z}^n \end{aligned}$$

# Outline

1. Introduction
- 2. Feature engineering for graphs**
3. GNN architecture
4. Training a GNN

# Traditional ML pipeline

- Design features for nodes/links/graphs
- Obtain features for all training data

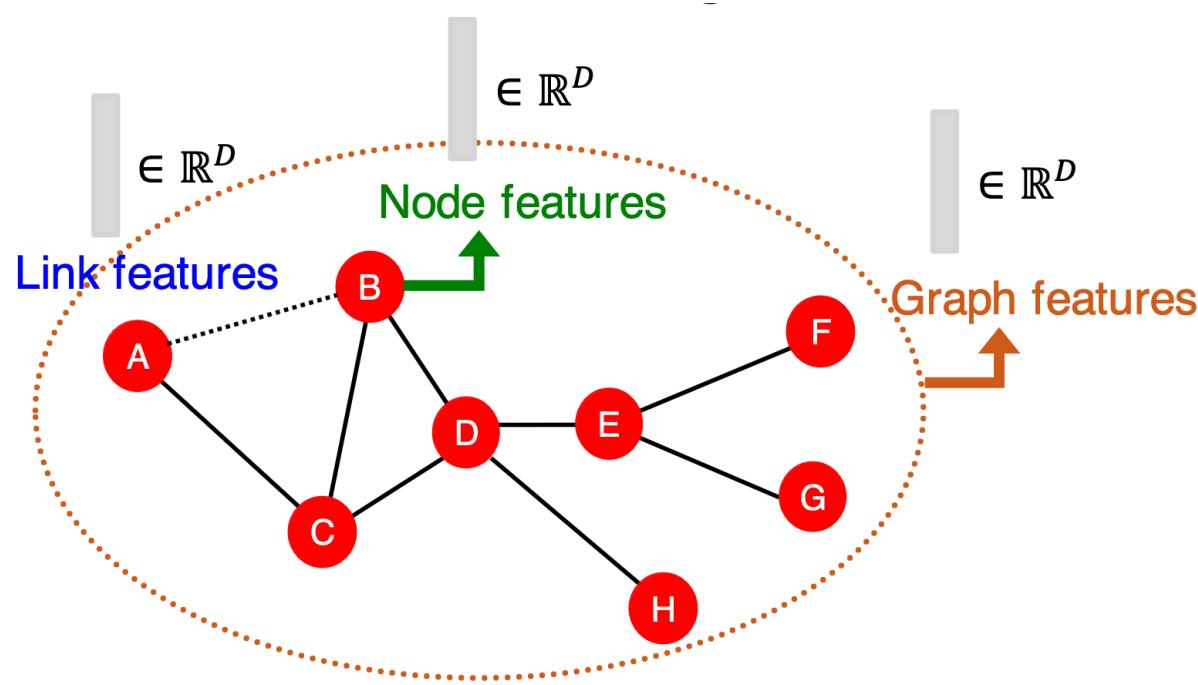
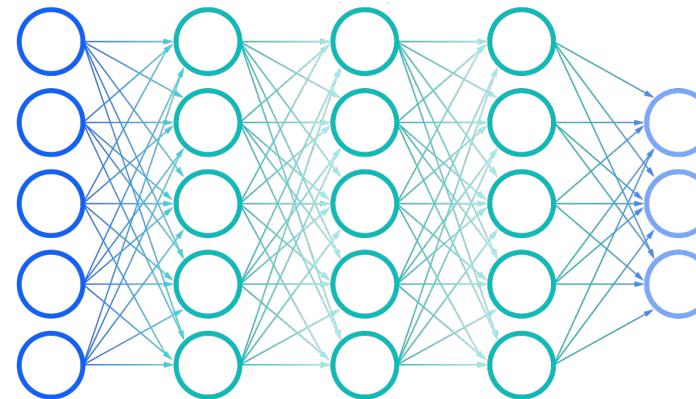


Figure by Leskovec

# Traditional ML pipeline

Train an ML model:

- Logistic Regression, random forest, NN, etc.



Apply the model:

- Given new node/link/graph, obtain features and make prediction

Using effective features is key to achieving good performance

# Different types of features

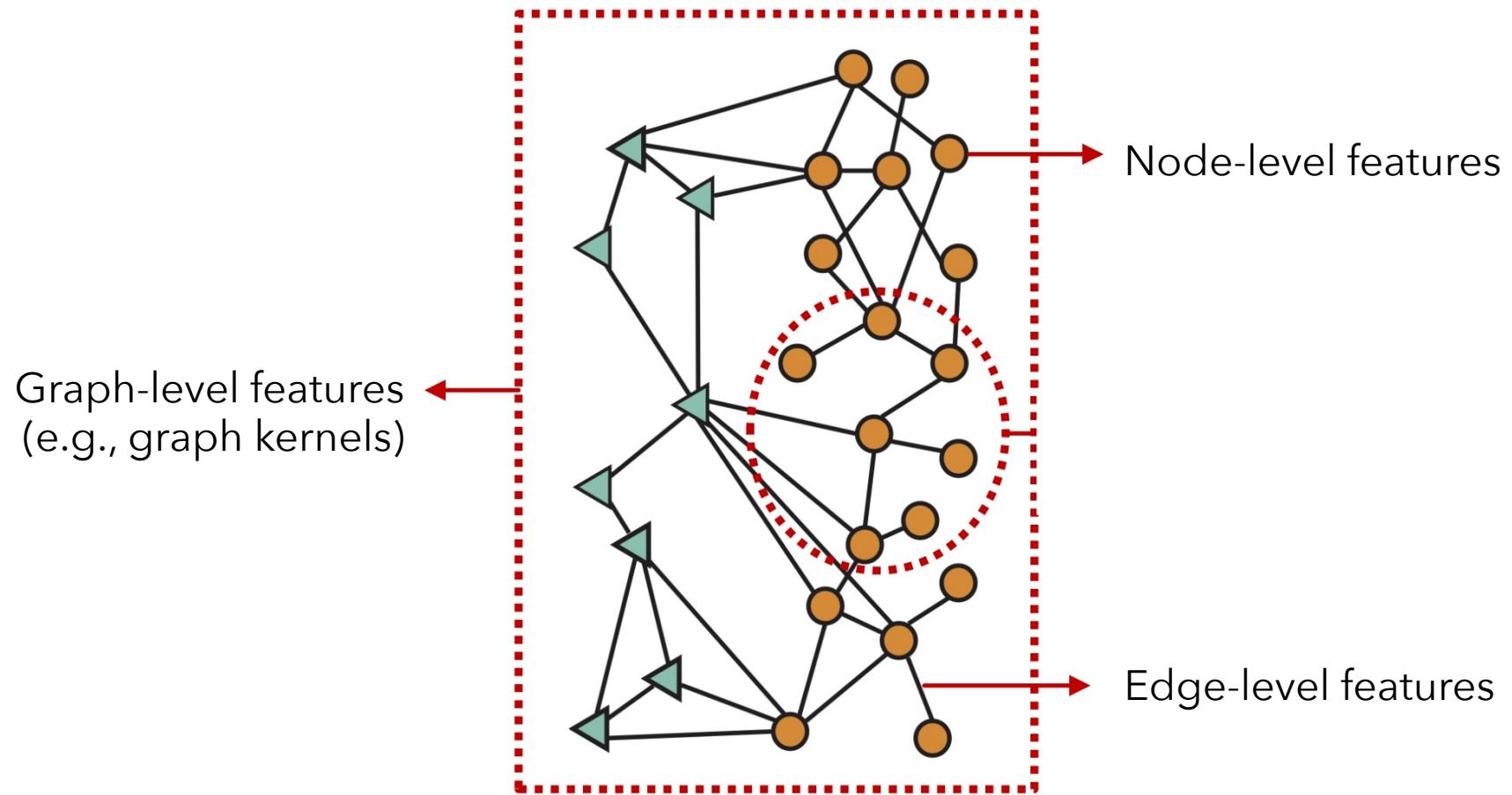


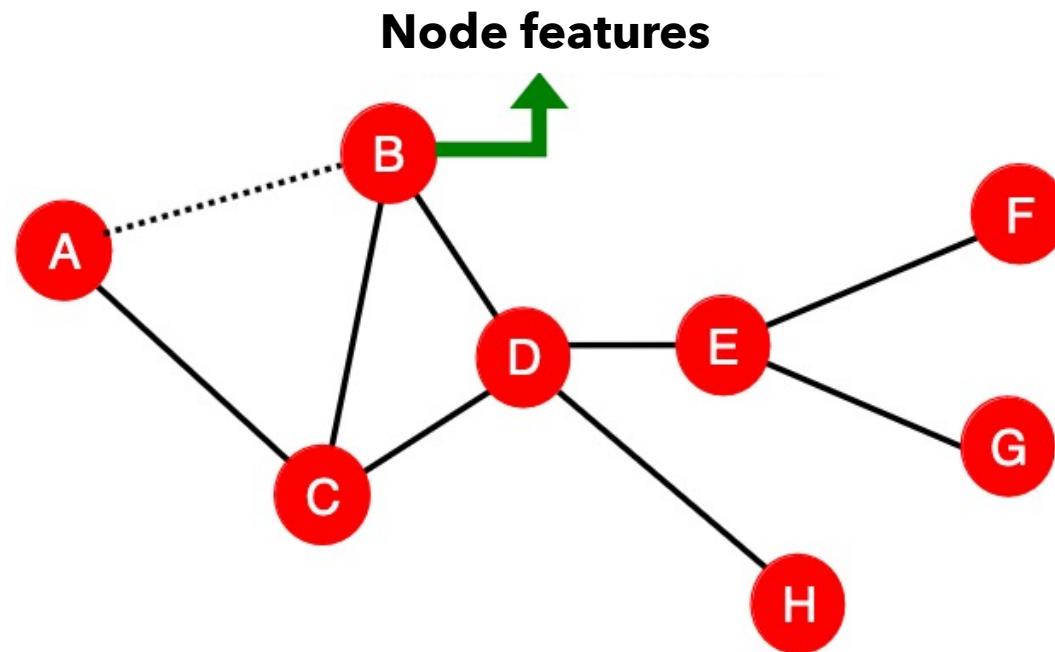
Figure by Leskovec

# Outline

1. Introduction
2. Feature engineering for graphs
  - a. **Node-level prediction**
  - b. Edge-level prediction
3. GNN architecture
4. Training a GNN

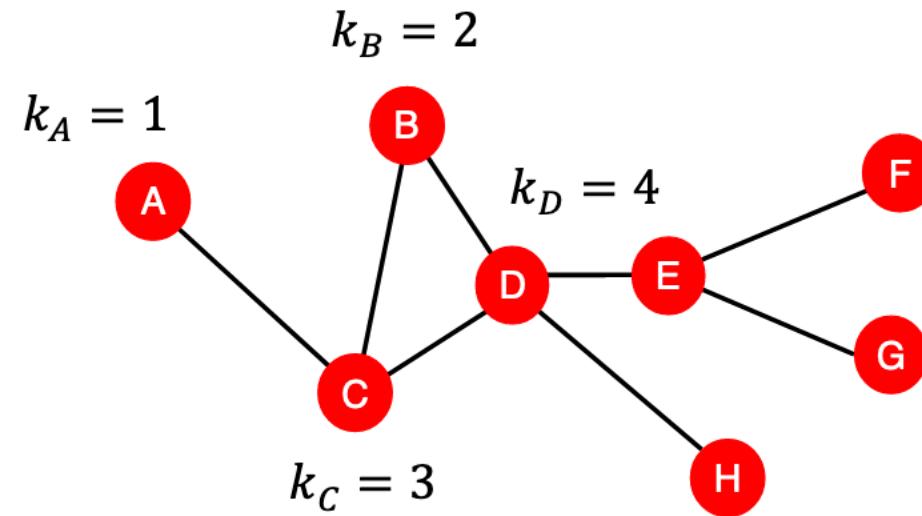
# Node-level features

**Goal:** Characterize structure and position of a node in network  
*Node degree, node centrality, clustering coefficient, graphlets*



# Node-level features: Degree

Degree  $k_v$  of node  $v$  = # neighboring nodes that the node has



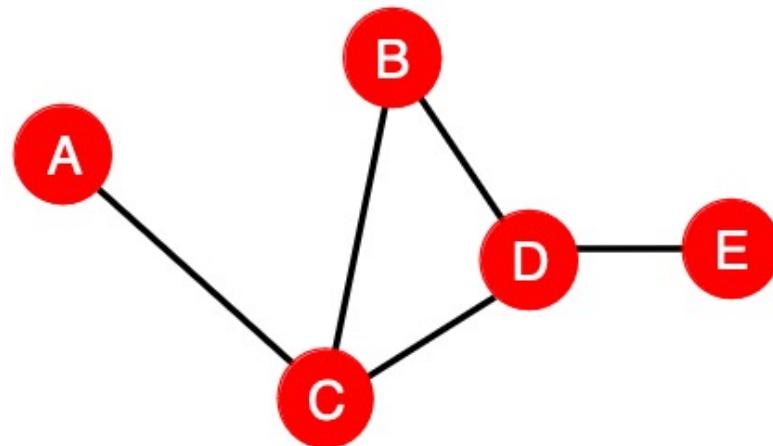
Treats all neighboring nodes equally

Node centrality takes the node importance in a graph into account

# Node-level features: Centrality

E.g., betweenness centrality:

Node is important if it's on many shortest paths between other nodes



Centrality  $c_A = c_B = c_E = 0$

$c_C = 3$

- A-**C**-B
- A-**C**-D
- A-**C**-D-E

$c_D = 3$

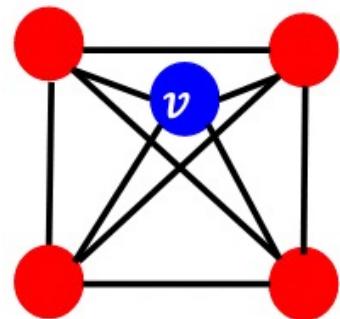
- A-C-**D**-E
- B-**D**-E
- C-**D**-E

# Node-level features: Clustering coeff.

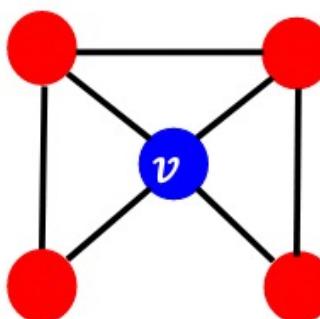
Captures topological properties of local neighborhood

Measures how connected  $v$ 's neighboring nodes are

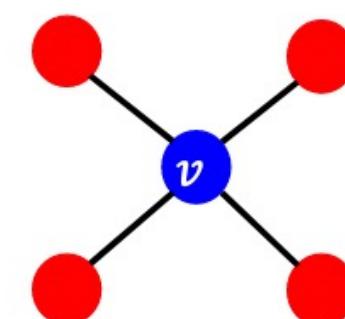
$$e_v = \frac{\text{\#(edges among neighboring nodes)}}{\binom{k_v}{2}}$$



$$e_v = 1$$



$$e_v = 0.5$$



$$e_v = 0$$

$$\binom{k_v}{2} = \binom{4}{2} = 6$$

# Outline

1. Introduction
2. Feature engineering for graphs
  - a. Node-level prediction
  - b. Edge-level prediction**
3. GNN architecture
4. Training a GNN

# Edge-level features

E.g., local neighborhood overlap:

Captures # neighboring nodes shared between nodes  $u, v$

Common neighbors:  $|N(v_1) \cap N(v_2)|$

E.g.,  $|N(A) \cap N(B)| = |\{C\}| = 1$

Jaccard's coefficient:  $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$

E.g.,  $\frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{|\{C\}|}{|\{C,D\}|} = \frac{1}{2}$

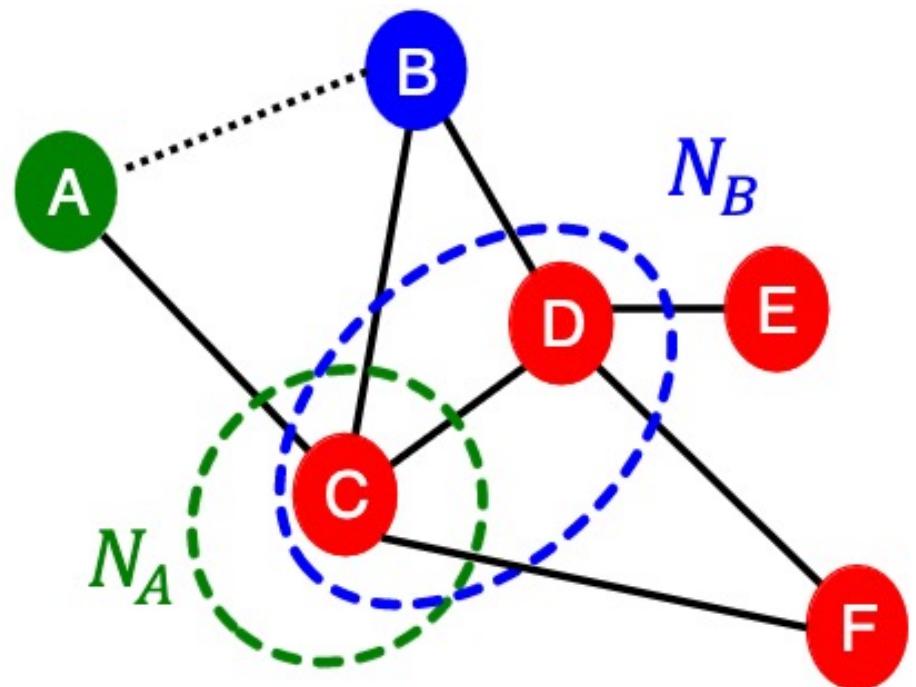


Figure by Leskovec

# Outline

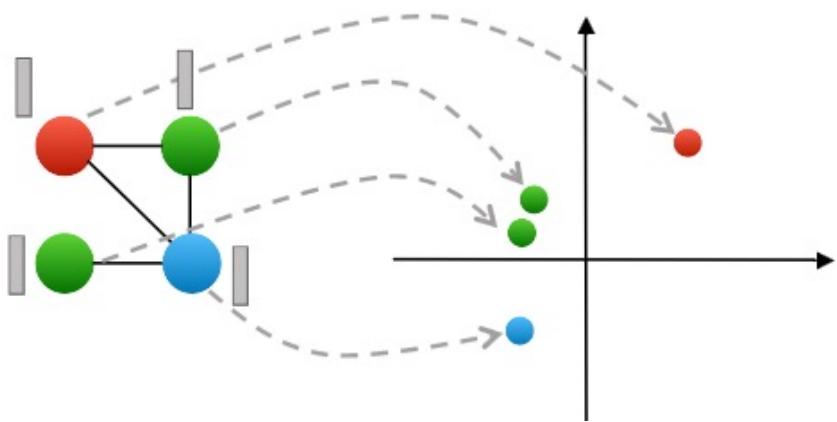
1. Introduction
2. Feature engineering for graphs
- 3. GNN architecture**
4. Training a GNN

# Setup

- $V$  is the vertex set
- $A$  is the adjacency matrix (assume binary)
- $X \in \mathbb{R}^{|V| \times d}$  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 info

# Two goals

## 1. Node embeddings



## 2. Graph embedding

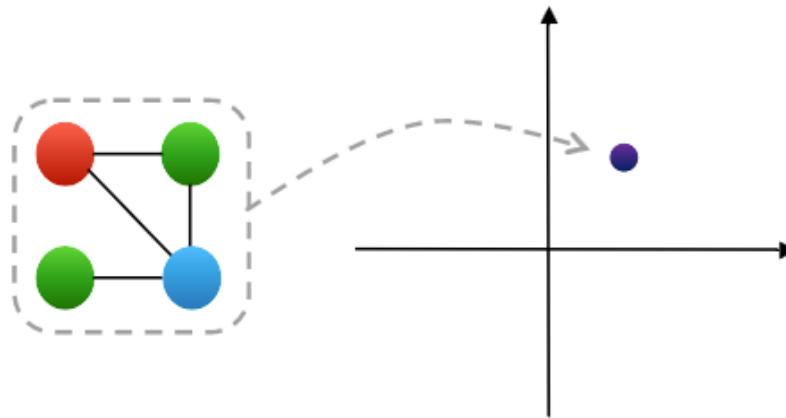
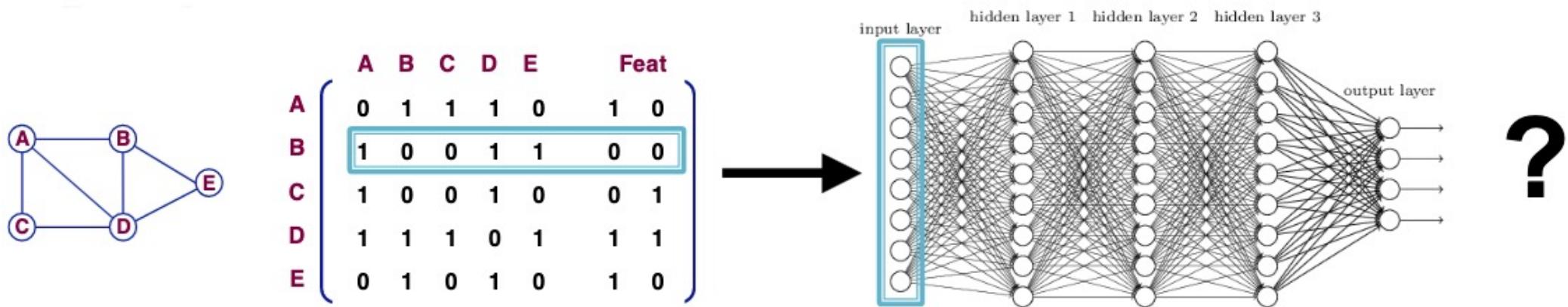


Figure by Jegelka

# Idea 1: fully connected NN?

**Idea:** Join adjacency matrix & features, give as input to NN

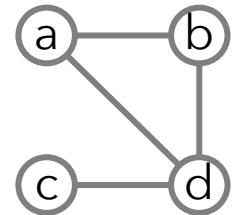


## Issues:

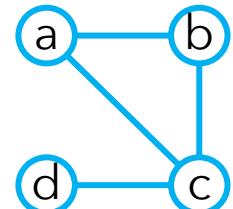
- Huge input
- Doesn't generalize across graph size
- Sensitive to node ordering

# Permutation invariance & equivariance

How to avoid sensitivity to node orderings?



Permutation matrix



$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$PAP^T = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

We want either:

- **Permutation invariance**
  - Graph embedding
  - Output: single vector
  - $f(PAP^T, PX) = f(A, X)$
- **Permutation equivariance**
  - Node embedding
  - Output: vector per node
  - $f(PAP^T, PX) = Pf(A, X)$

# Permutation invariance & equivariance

GNNs consist of permutation equivariant/invariant functions

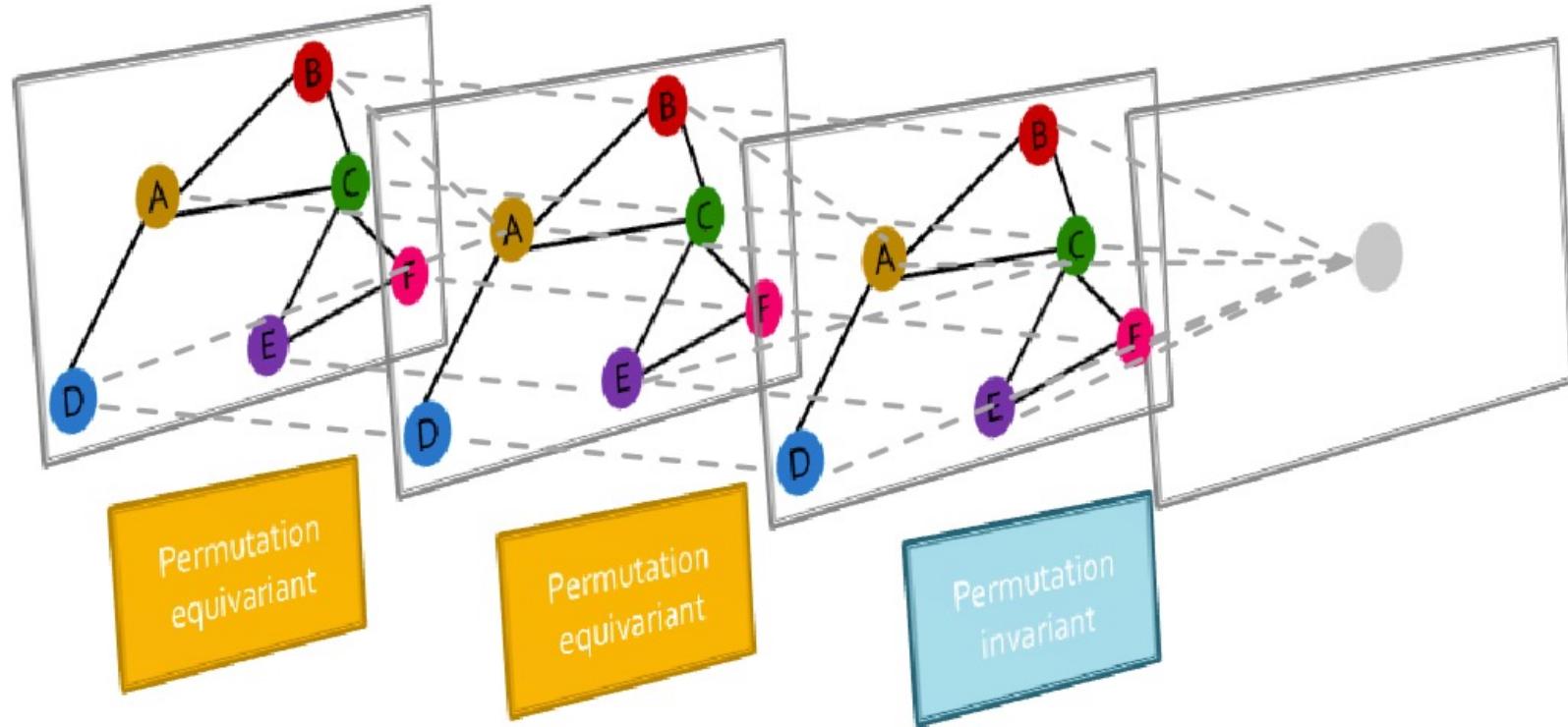


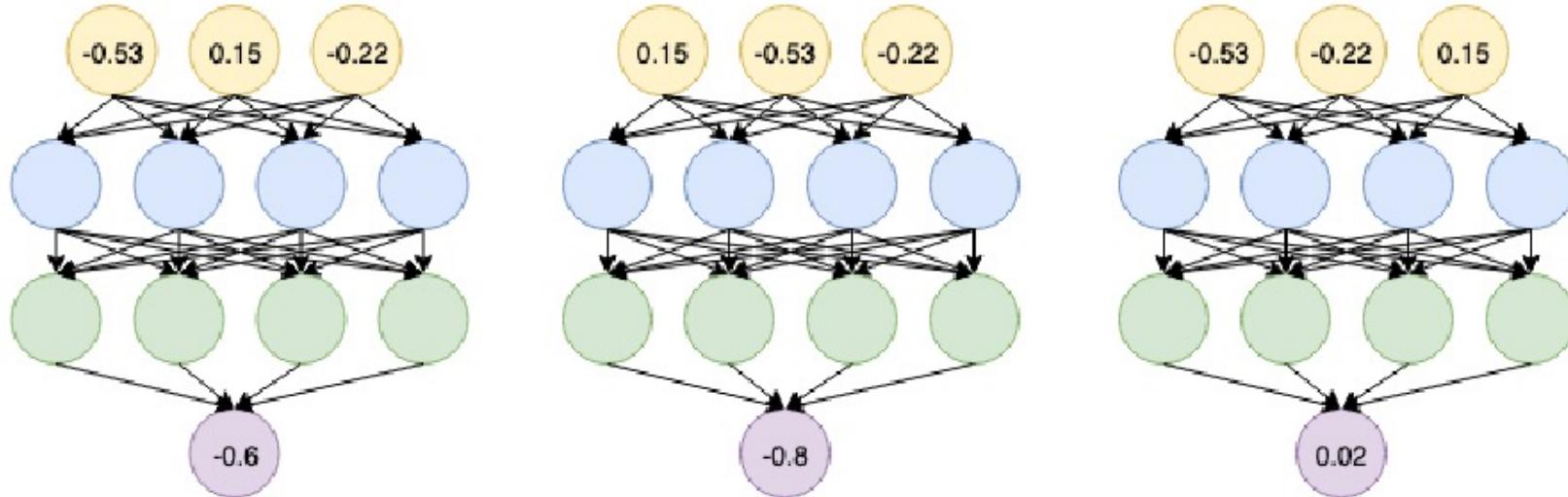
Figure by Bronstein, ICLR 2021 keynote

# Permutation invariance & equivariance

Are other NN architectures permutation invariant / equivariant?

E.g., *MLP*

**No.**



Explains why naïve MLP approach fails for graphs

# Graph neural networks

## Idea:

1. Encode each node (node's neighborhood) with embedding
2. Aggregate set of node embeddings into graph embedding

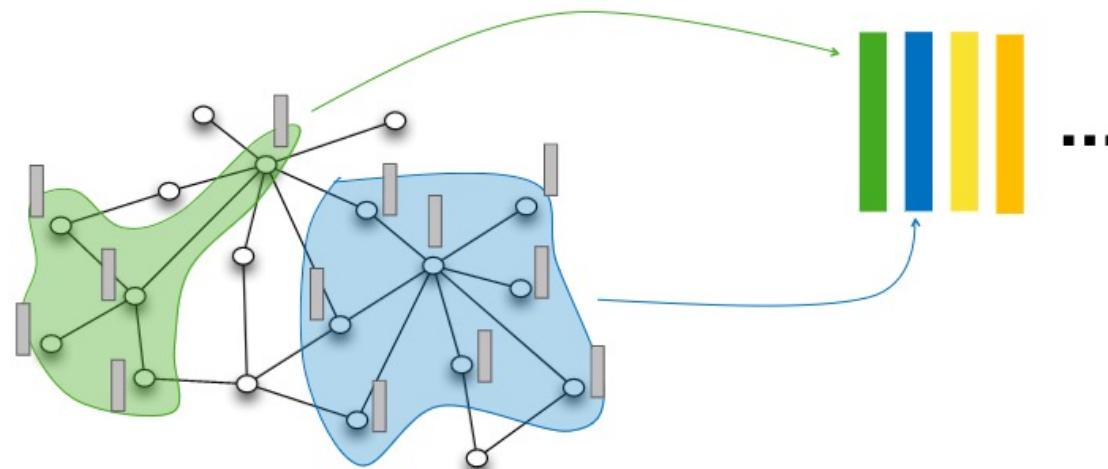
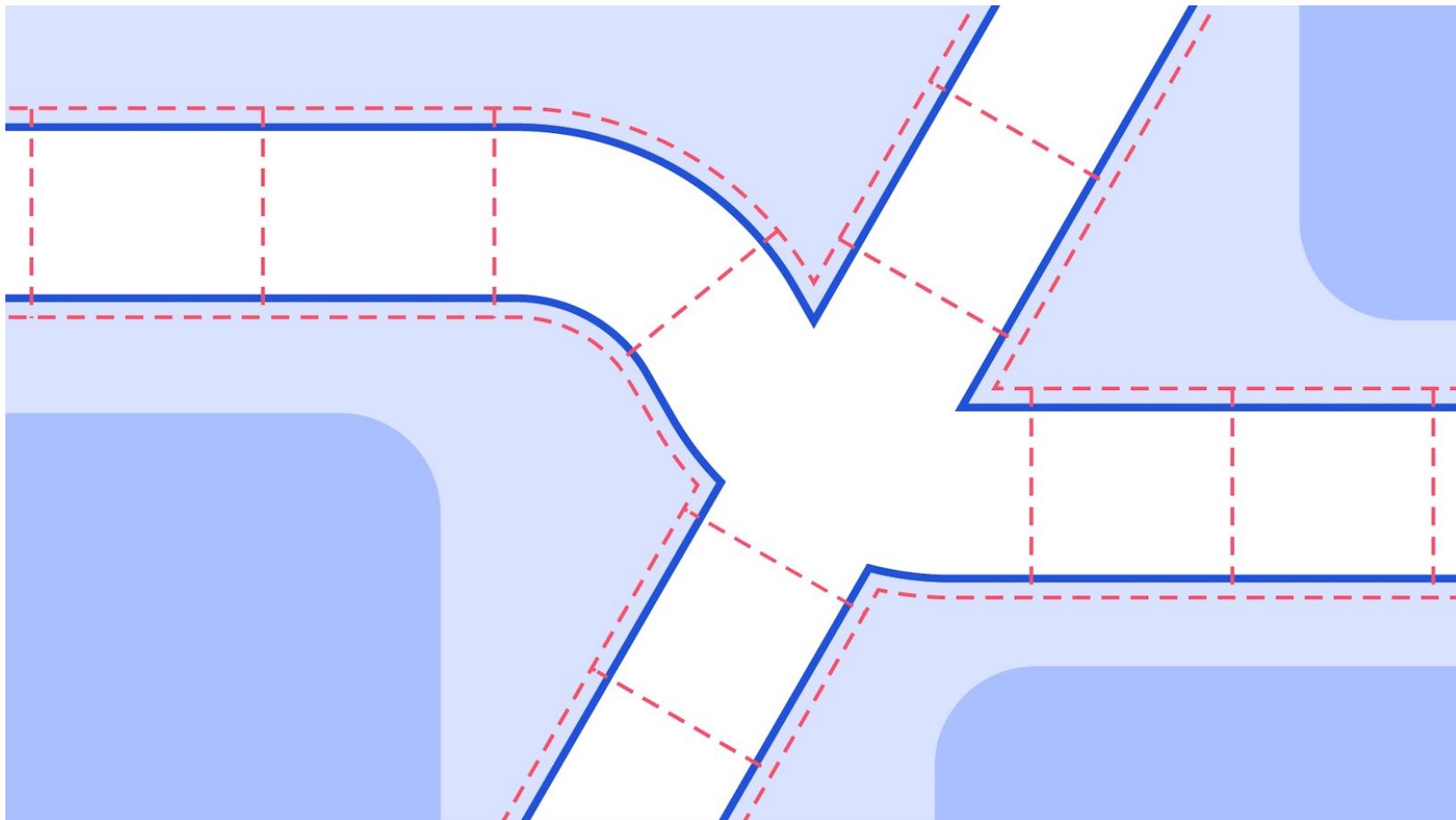
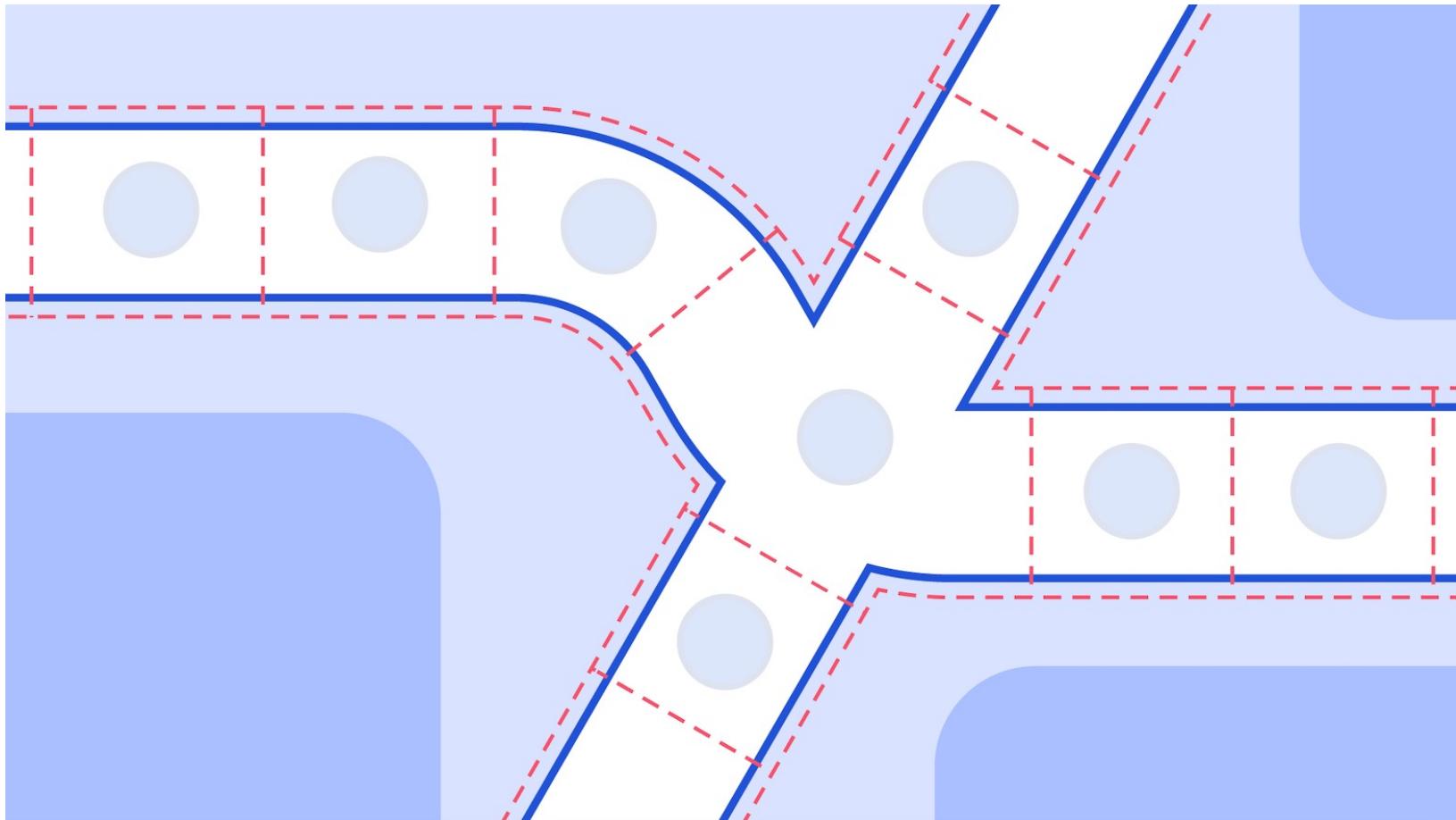


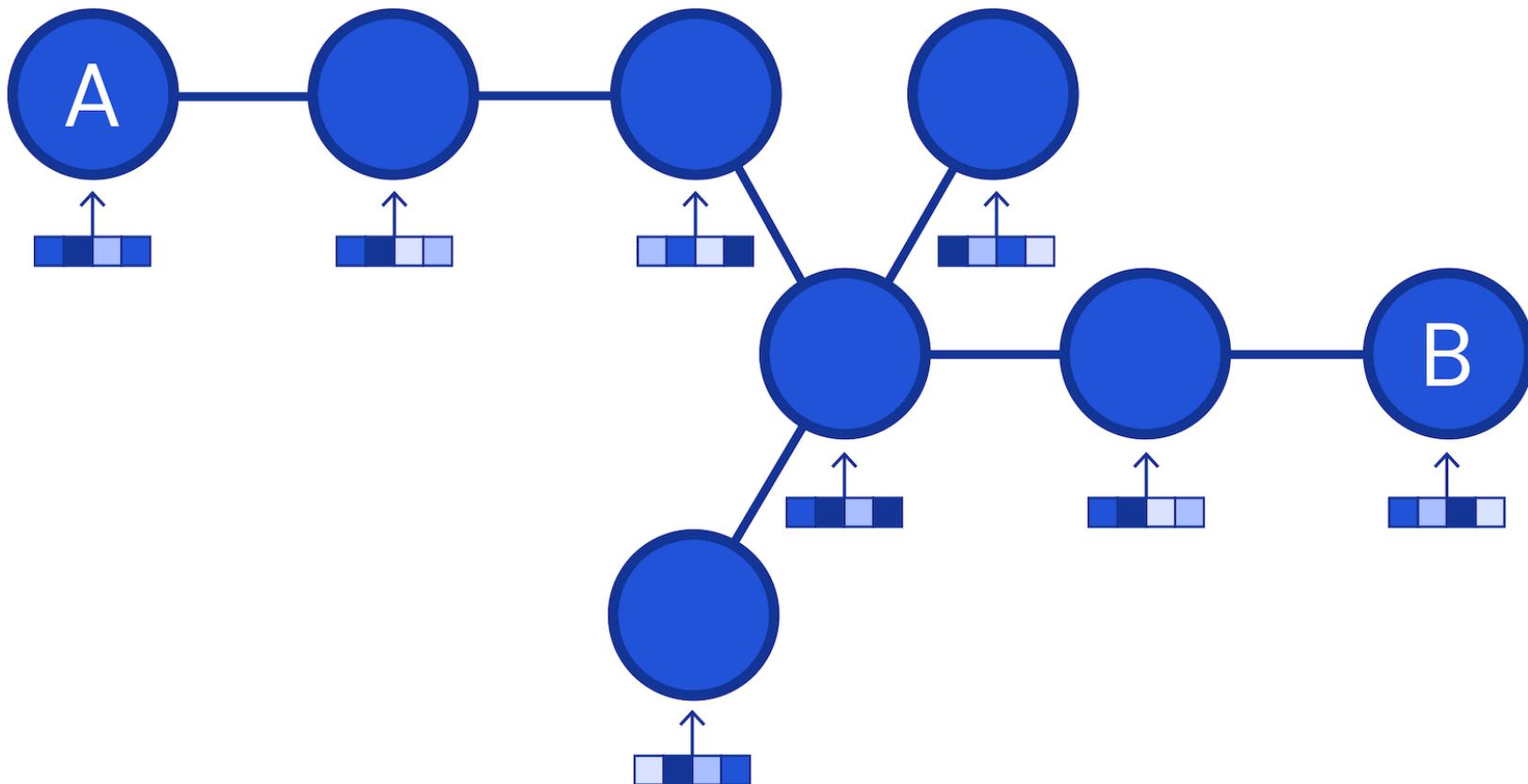
Figure by Jegelka

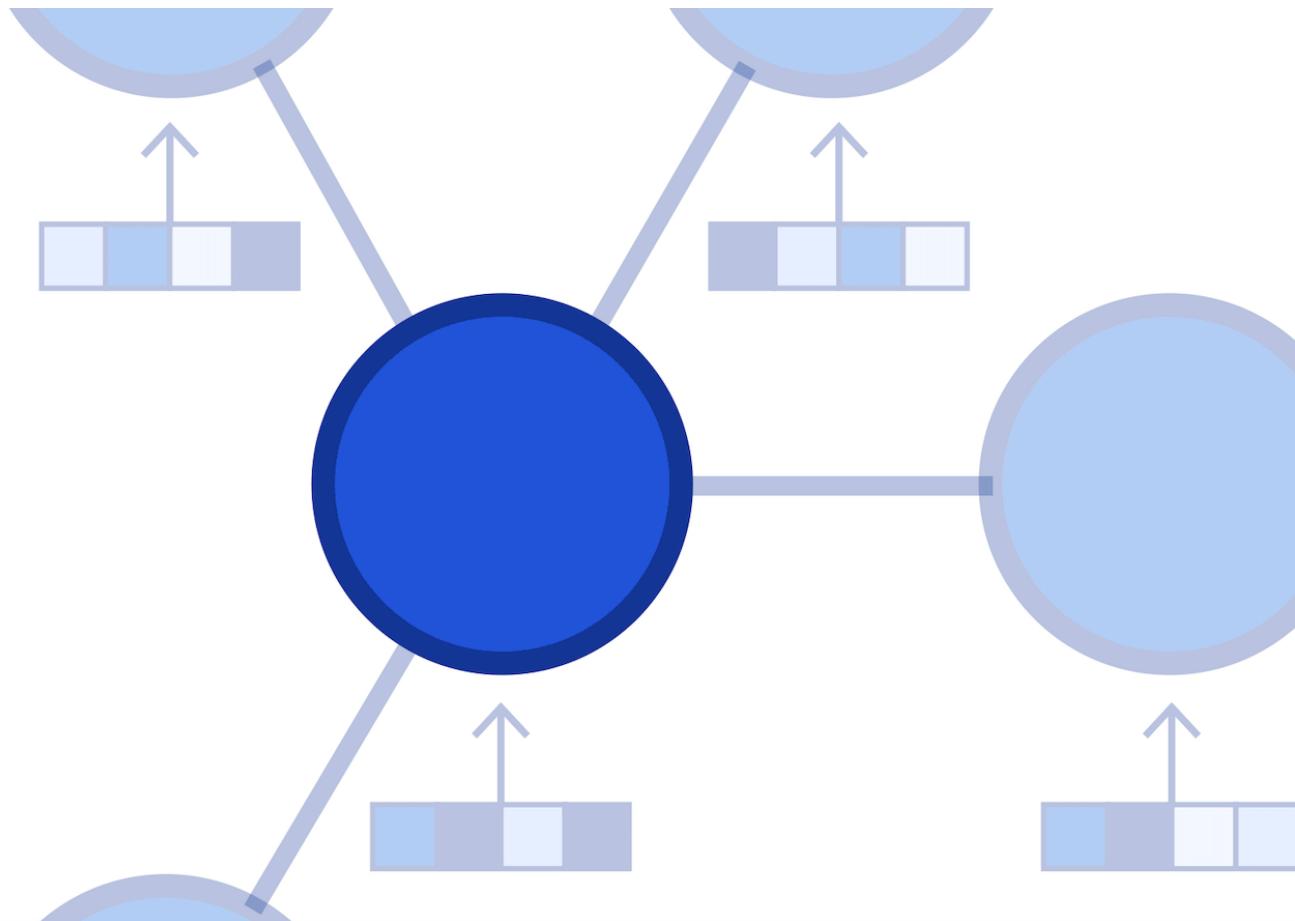


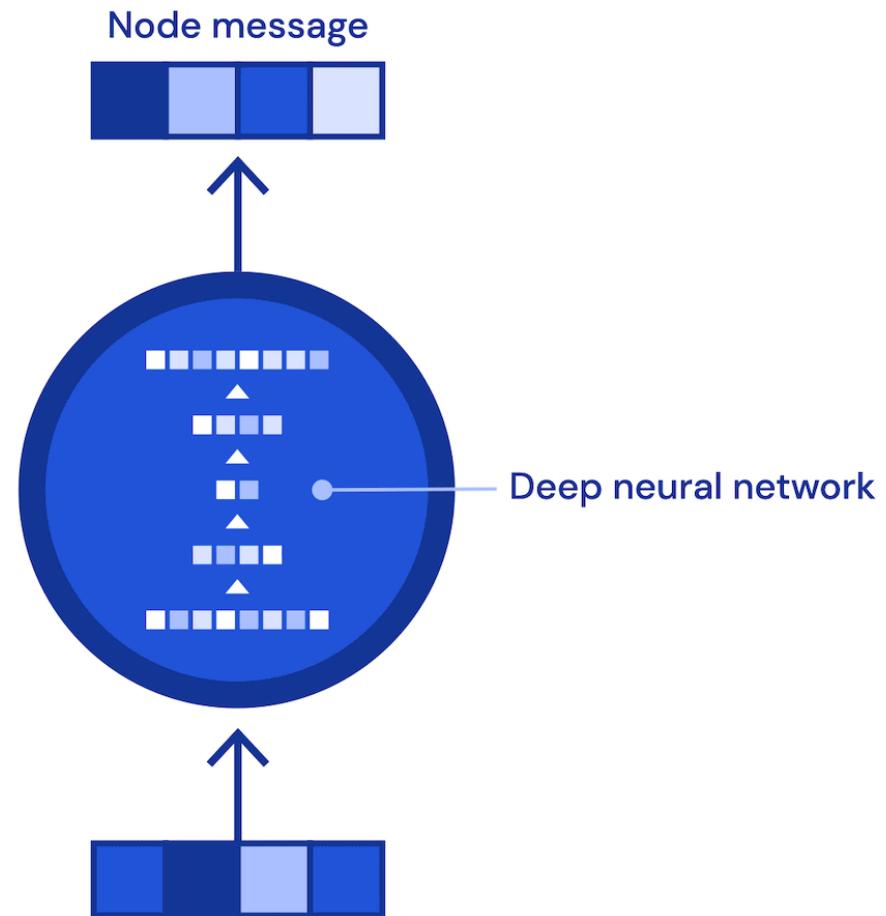
<https://www.deepmind.com/blog/traffic-prediction-with-advanced-graph-neural-networks>

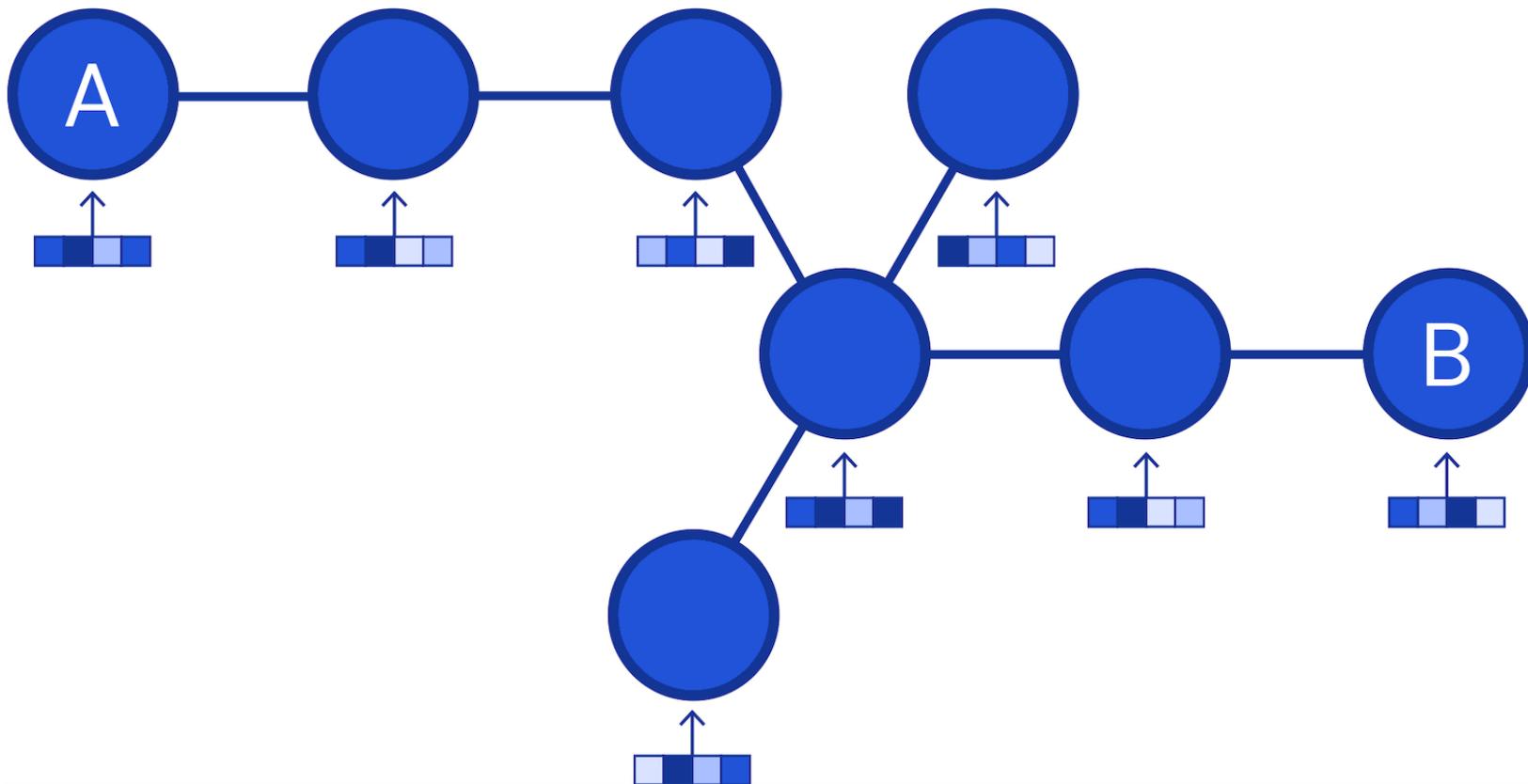


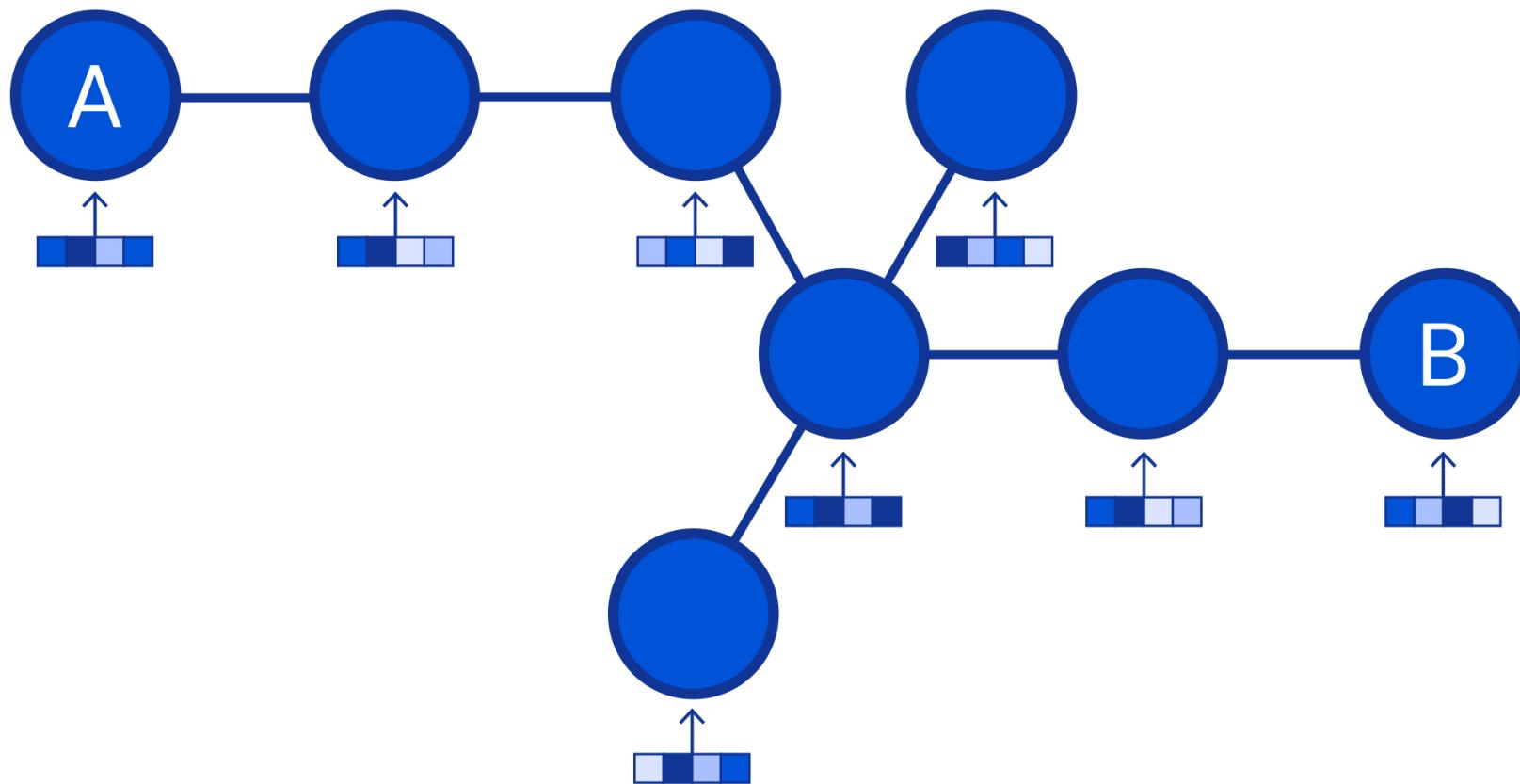












# Encoding neighborhoods: General form

$\mathbf{h}_u^{(0)} = \mathbf{x}_u$  (feature representation for node  $u$ )

In each round  $k \in [K]$ , for each node  $v$ : :

1. **Aggregate** over neighbors

$$\mathbf{m}_{\overline{N(v)}}^{(k)} = \text{AGGREGATE}^{(k)} \left( \left\{ \mathbf{h}_u^{(k-1)} : u \in N(v) \right\} \right)$$

Neighborhood of  $v$

# Encoding neighborhoods: General form

$$\mathbf{h}_u^{(0)} = \mathbf{x}_u \text{ (feature representation for node } u\text{)}$$

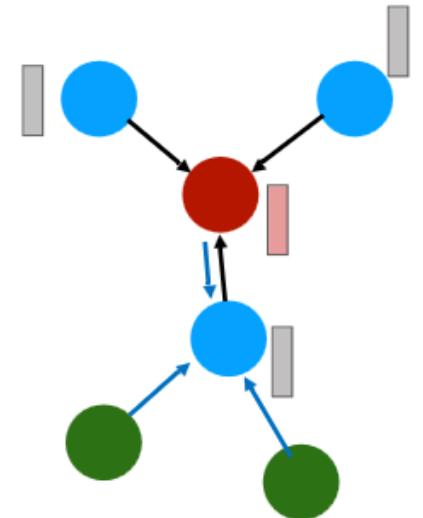
In each round  $k \in [K]$ , for each node  $v$ :

1. **Aggregate** over neighbors

$$\mathbf{m}_{N(v)}^{(k)} = \text{AGGREGATE}^{(k)} \left( \left\{ \mathbf{h}_u^{(k-1)} : u \in N(v) \right\} \right)$$

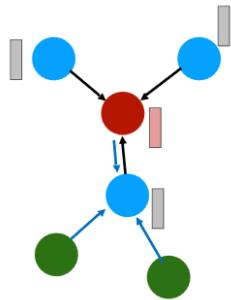
2. **Update** current node representation

$$\mathbf{h}_v^{(k)} = \text{COMBINE}^{(k)} \left( \mathbf{h}_v^{(k-1)}, \mathbf{m}_{N(v)}^{(k)} \right)$$



# The basic GNN

[Merkwirth and Lengauer '05; Scarselli et al. '09]



$$\mathbf{m}_{N(v)} = \text{AGGREGATE}(\{\mathbf{h}_u : u \in N(v)\}) = \sum_{u \in N(v)} \mathbf{h}_u$$

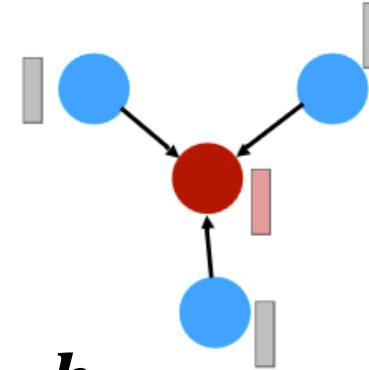
$$\text{COMBINE}(\mathbf{h}_v, \mathbf{m}_{N(v)}) = \sigma(W_{\text{self}} \mathbf{h}_v + W_{\text{neigh}} \mathbf{m}_{N(v)} + \mathbf{b})$$

Trainable parameters

Non-linearity (e.g.,  
tanh or ReLU)

# Aggregation functions

$$\mathbf{m}_{N(v)} = \text{AGGREGATE}(\{\mathbf{h}_u : u \in N(v)\}) = \sum_{u \in N(v)} \mathbf{h}_u$$



Unstable, highly sensitive to node degrees

Instead, take averages, e.g.:

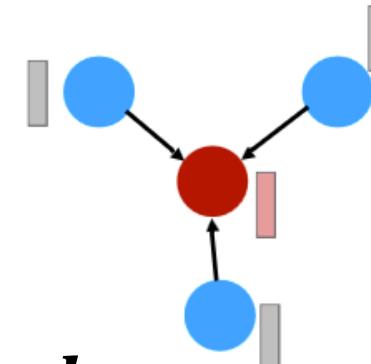
- $\mathbf{m}_{N(v)} = \frac{1}{|N(v)|} \sum_{u \in N(v)} \mathbf{h}_u$  [Merkwirth & Lengauer '05, Scarselli et al. '09]
- $\mathbf{m}_{N(v)} = \sum_{u \in N(v)} \frac{1}{\sqrt{|N(u)||N(v)|}} \mathbf{h}_u$  [Kipf & Welling '16, Hamilton et al. '17]

Can we do more to improve?

# Aggregation functions

$$\mathbf{m}_{N(v)} = \text{AGGREGATE}(\{\mathbf{h}_u : u \in N(v)\}) = \sum_{u \in N(v)} \mathbf{h}_u$$

Unstable, highly sensitive to node degrees



Should be a permutation invariant, multi-set function

# Aggregation functions

$$\begin{aligned}\mathbf{m}_{N(v)} &= \text{AGGREGATE}(\{\mathbf{h}_u : u \in N(v)\}) \\ &= \text{MLP}_2 \left( \sum_{u \in N(v)} \text{MLP}_1(\mathbf{h}_u, \mathbf{h}_v) \right)\end{aligned}$$

Universal approximation of multi-set functions

[Zaheer et al. '17, Qi et al. '17, Xu et al. '19]

$$\text{COMBINE}(\mathbf{h}_v, \mathbf{m}_{N(v)}) = \sigma(W_{\text{self}}\mathbf{h}_v + W_{\text{neigh}}\mathbf{m}_{N(v)} + b)$$

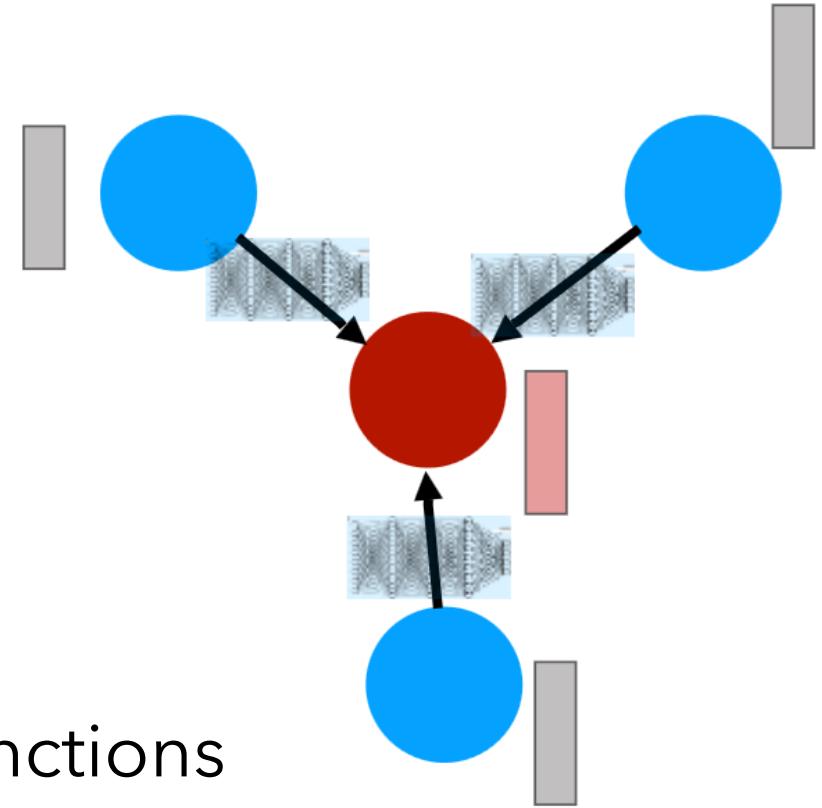


Figure by Jegelka

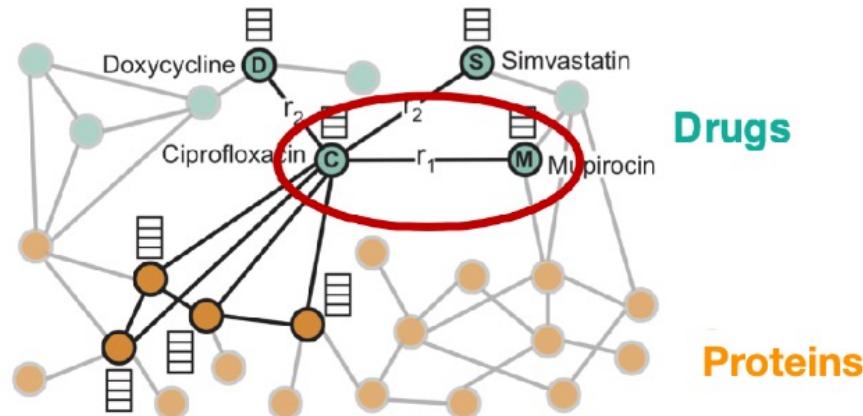
# Generalizations

- Use edge attributes/features in aggregation

$$m_{N(v)} = \text{AGGREGATE}(\{\mathbf{h}_u : u \in N(v)\}) = \sum_{u \in N(v)} \text{MLP}(\mathbf{h}_u, \mathbf{h}_v, w_{uv})$$

- Different aggregations for different types of edges

E.g., Zitnik et al. ['18]

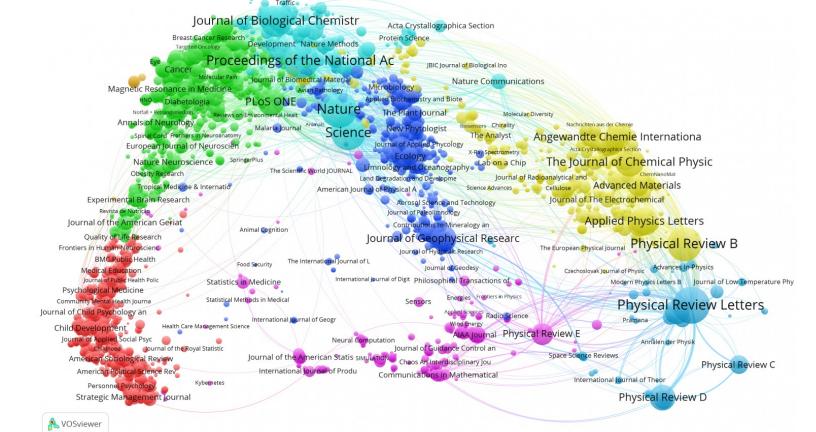


# Generalizations

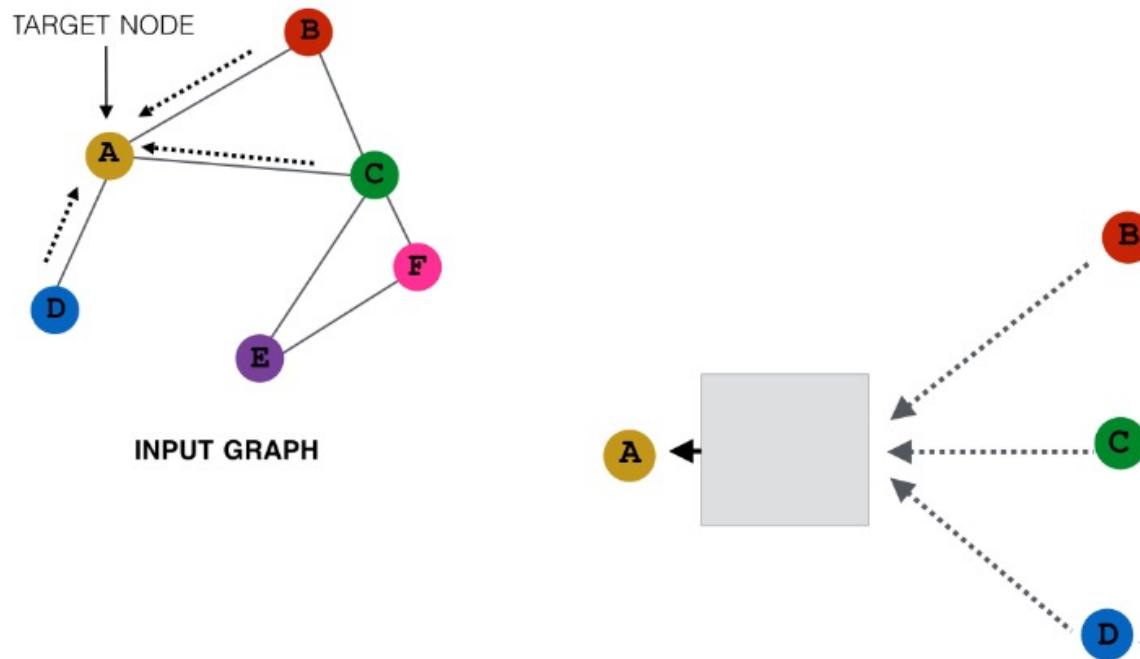
**Attention** [Velickovic et al. '18]:

$$m_{N(v)} = \text{AGGREGATE}(\{\mathbf{h}_u : u \in N(v)\}) = \alpha_{v,u} \mathbf{h}_u$$

- Useful when some neighbors might be more/less informative
- E.g., classifying papers by topic based on citation networks
  - Some papers that span topical boundaries, highly-cited across fields
  - GNN should learn to ignore uninformative neighbors

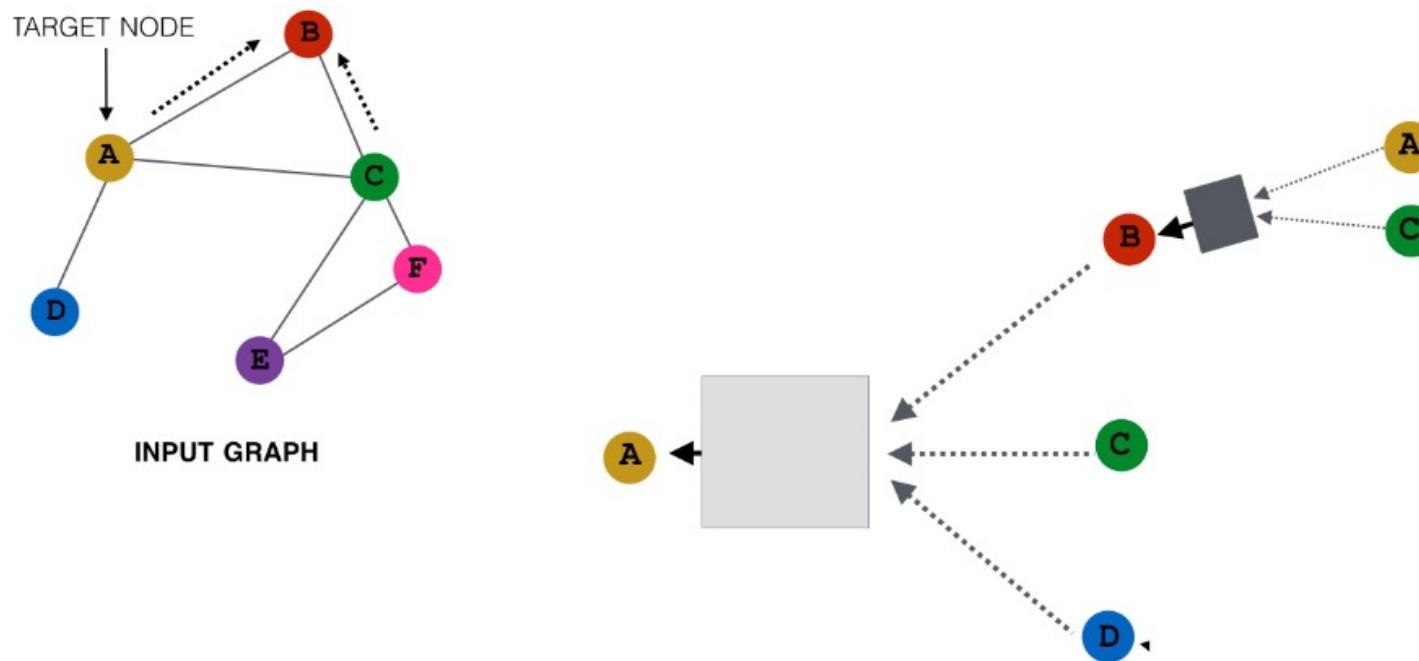


# Node embeddings unrolled



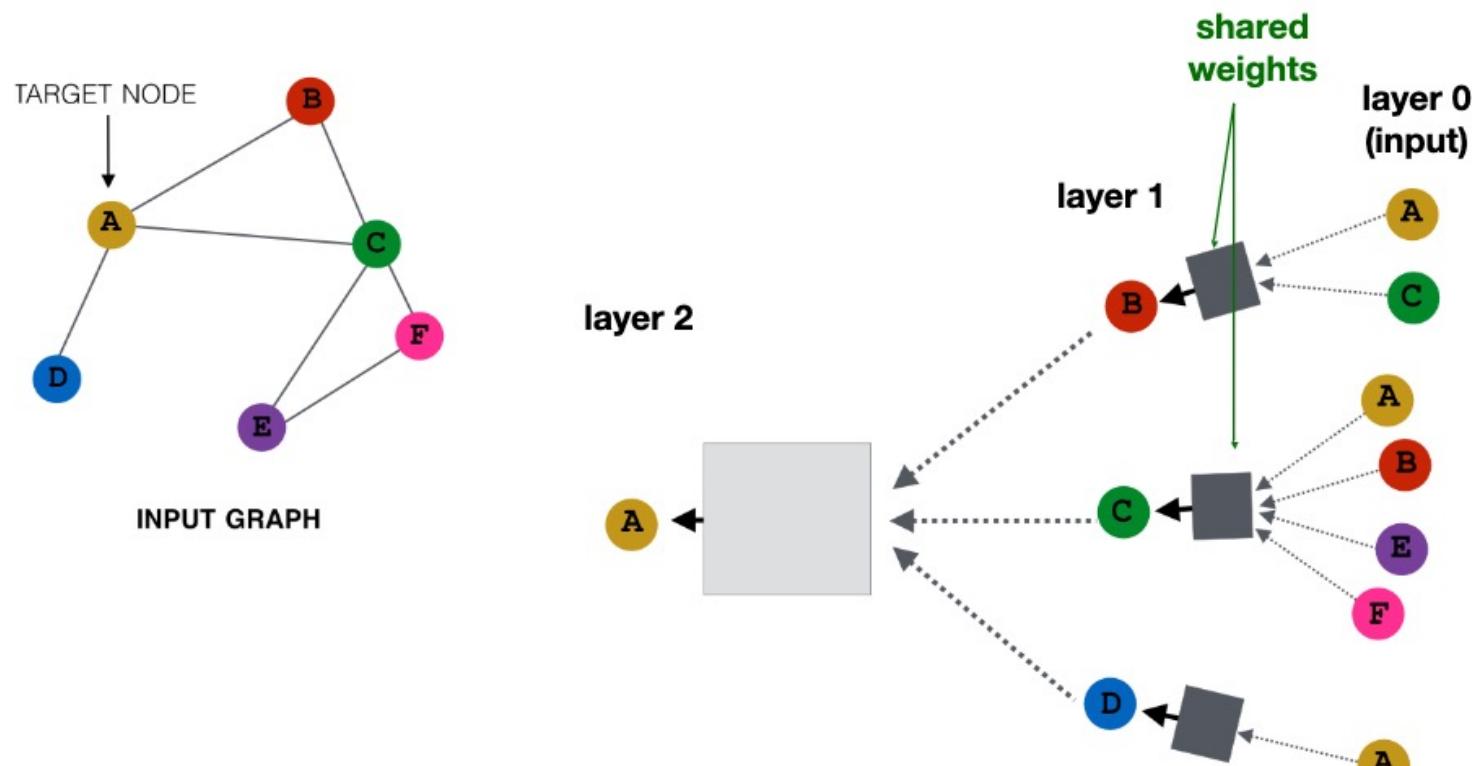
Grey boxes: aggregation functions that we learn

# Node embeddings unrolled



Grey boxes: aggregation functions that we learn

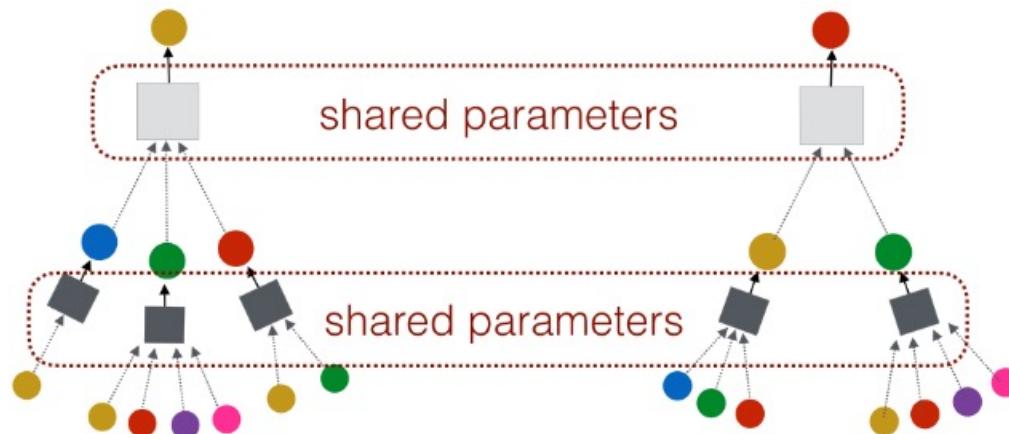
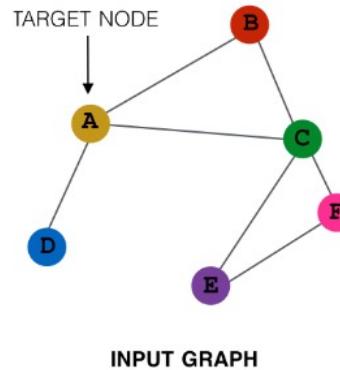
# Node embeddings unrolled



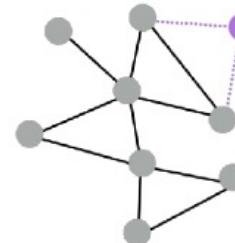
Grey boxes: aggregation functions that we learn

# Weight sharing

Use the same aggregation functions for all nodes



Can generate encodings for  
previously unseen nodes & graphs!

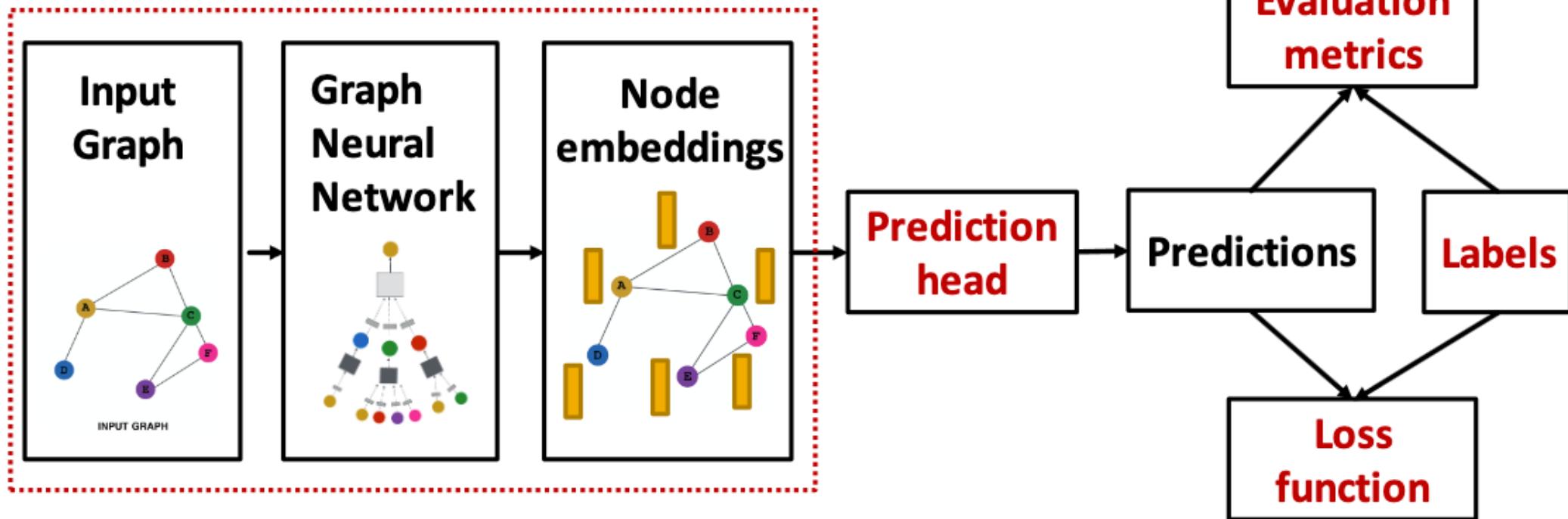


# Outline

1. Introduction
2. Feature engineering for graphs
  - a. Node-level prediction
  - b. Edge-level prediction
3. GNN architecture
- 4. Training a GNN**
  - a. GNN pipeline**
  - b. Train/validation/test splits
  - c. Skip connections
  - d. Graph manipulations

# GNN Pipeline

What we've covered so far



# Prediction heads

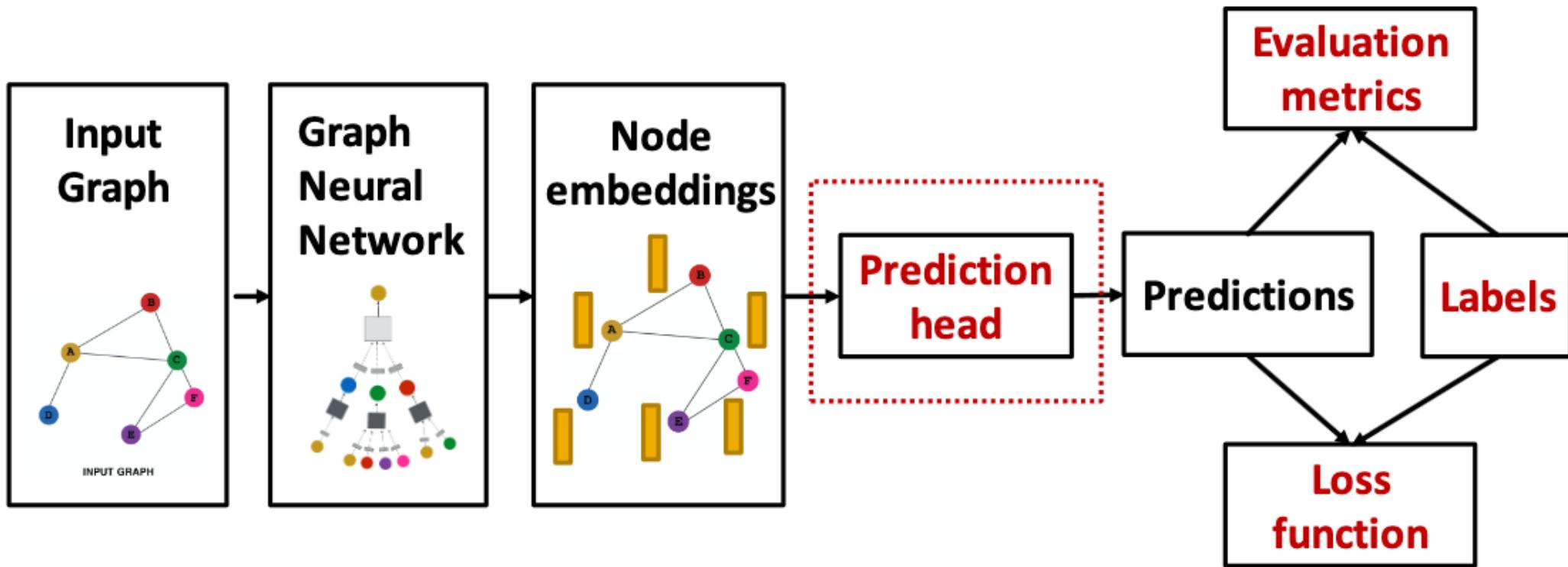
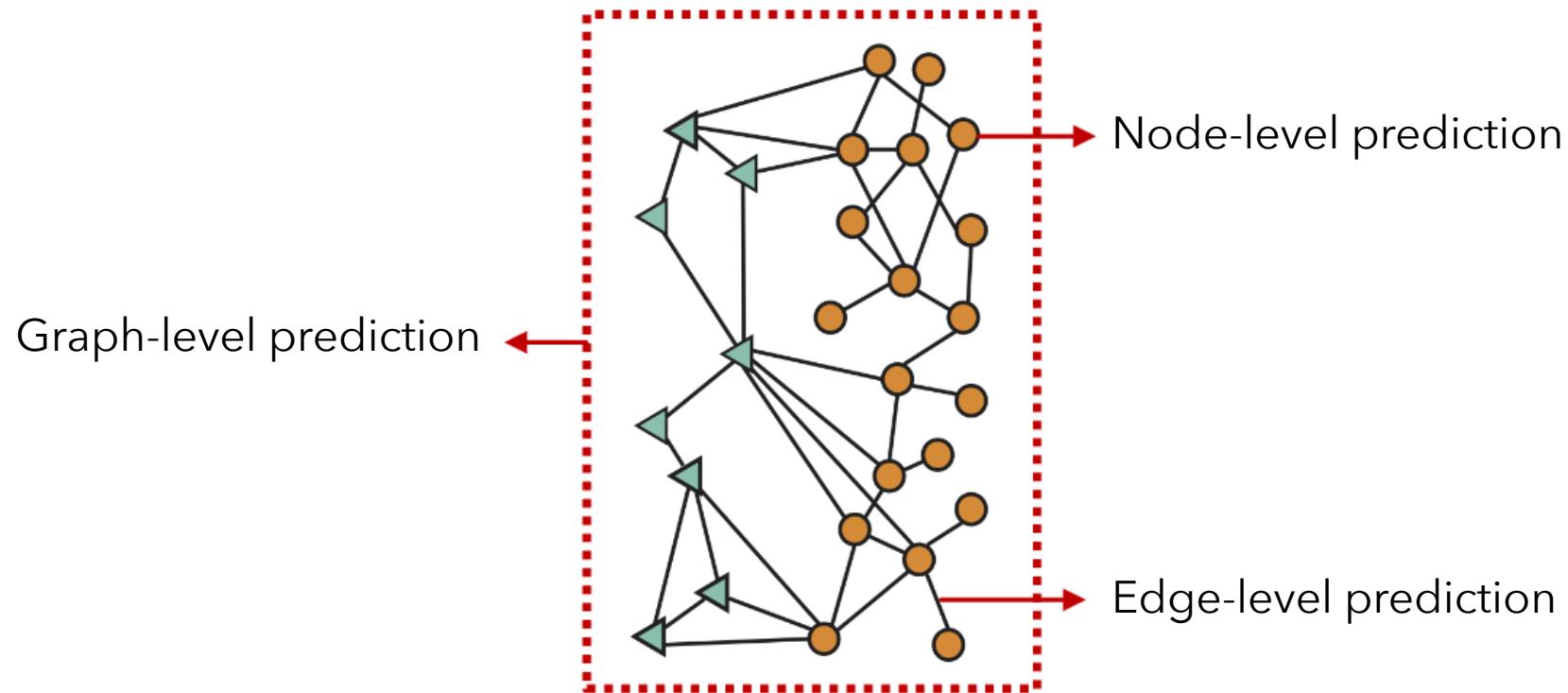


Figure by Leskovec

# Prediction heads

Different task levels require different prediction heads



# Prediction heads: Node-level

After GNN computation, we have node embeddings

$$\{\mathbf{h}_v^{(K)} \in \mathbb{R}^d, \forall v \in V\}$$

Suppose we want to make  $k$ -way predictions

- Classification: classify among  $k$  categories
- Regression: regress on  $k$  targets

$$\hat{\mathbf{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(K)}) = W^{(H)} \mathbf{h}_v^{(K)}$$

- $W^{(H)} \in \mathbb{R}^{k \times d}$  so  $\hat{\mathbf{y}}_v \in \mathbb{R}^k$

# Prediction heads: Edge-level

Suppose we want to make  $k$ -way predictions

$$\hat{y}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(K)}, \mathbf{h}_v^{(K)})$$

$$\hat{y}_{uv} = \text{Linear} \left( \text{Concatenate} \left( \mathbf{h}_u^{(K)}, \mathbf{h}_v^{(K)} \right) \right)$$

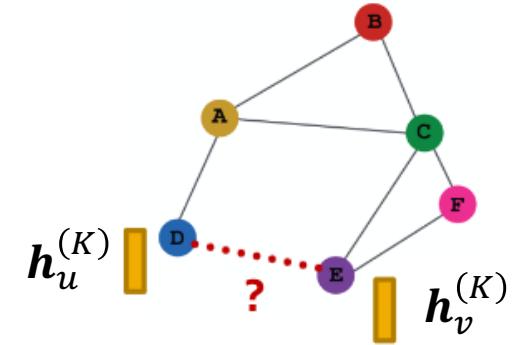
Linear maps  $2d$ -dimensional embedding to  $k$ -way embedding

Similar to multi-head attention:

$$\hat{y}_{uv}[1] = \mathbf{h}_u^{(K)} W^{(1)} \mathbf{h}_v^{(K)}$$

⋮

$$\hat{y}_{uv}[k] = \mathbf{h}_u^{(K)} W^{(k)} \mathbf{h}_v^{(K)}$$

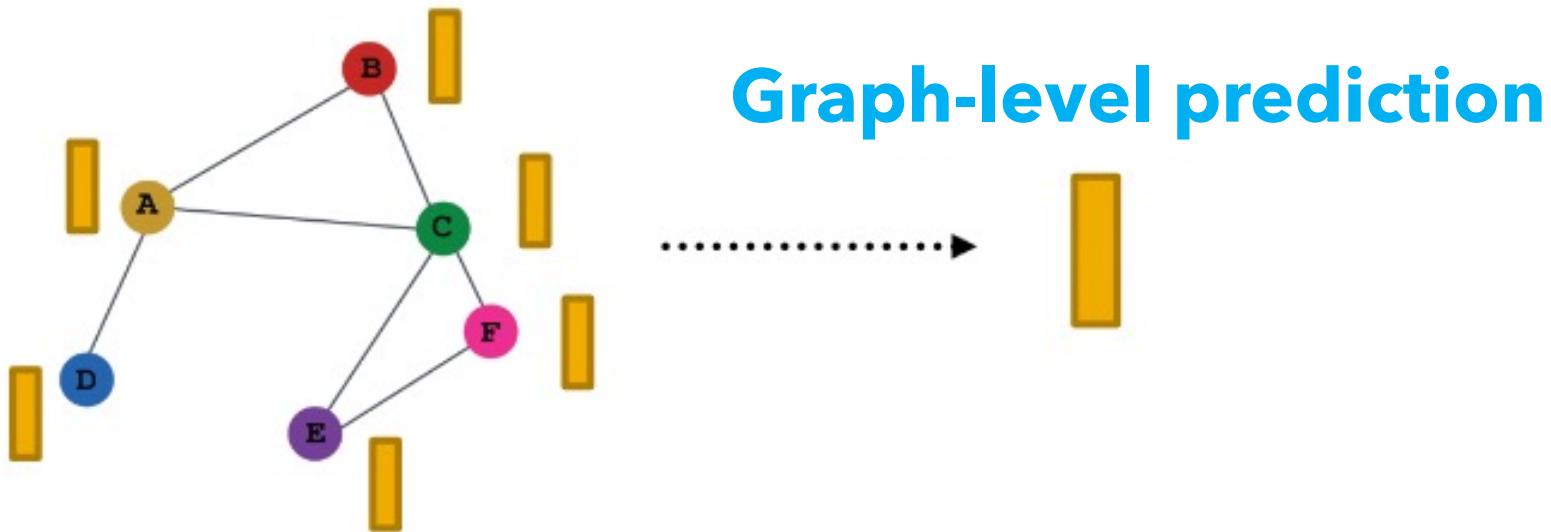


# Prediction heads: Graph-level

## Graph-level prediction:

Make prediction using all node embeddings

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



# Prediction heads: Graph-level

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

- Global **mean** pooling  $\hat{\mathbf{y}}_G = \text{Mean} \left( \left\{ \mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G \right\} \right)$
- Global **max** pooling  $\hat{\mathbf{y}}_G = \text{Max} \left( \left\{ \mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G \right\} \right)$
- Global **sum** pooling  $\hat{\mathbf{y}}_G = \text{Sum} \left( \left\{ \mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G \right\} \right)$

Work well for **small** graphs

*What about large graphs?*

# Issue of global pooling

**Issue:** Global pooling over a (large) graph will lose information

**Toy example** with 1-dim node embeddings:

- Node embeddings for  $G_1$ :  $\{-1, -2, 0, 1, 2\}$
- Node embeddings for  $G_2$ :  $\{-10, -20, 0, 10, 20\}$
- If we do global sum pooling:
  - Prediction for  $G_1$ :  $\hat{y}_{G_1} = \text{Sum}(\{-1, -2, 0, 1, 2\}) = 0$
  - Prediction for  $G_2$ :  $\hat{y}_{G_2} = \text{Sum}(\{-10, -20, 0, 10, 20\}) = 0$
  - **Cannot differentiate between  $G_1$  and  $G_2$ !**

# Hierarchical global pooling

- **A solution:** Aggregate all node embeddings **hierarchically**
- **Toy example:** Aggregate via  $\text{ReLU}(\text{Sum}(\cdot))$ 
  - First separately aggregate the first 2 nodes and the last 3 nodes
  - Then aggregate again to make final prediction
- $G_1$  node embeddings:  $\{-1, -2, 0, 1, 2\}$ 
  - **Round 1:**  $\hat{y}_a = \text{ReLU}(\text{Sum}(\{-1, -2\})) = 0,$   
 $\hat{y}_b = \text{ReLU}(\text{Sum}(\{0, 1, 2\})) = 3$
  - **Round 2:**  $\hat{y}_{G_1} = \text{ReLU}(\text{Sum}(\{\hat{y}_a, \hat{y}_b\})) = 3$

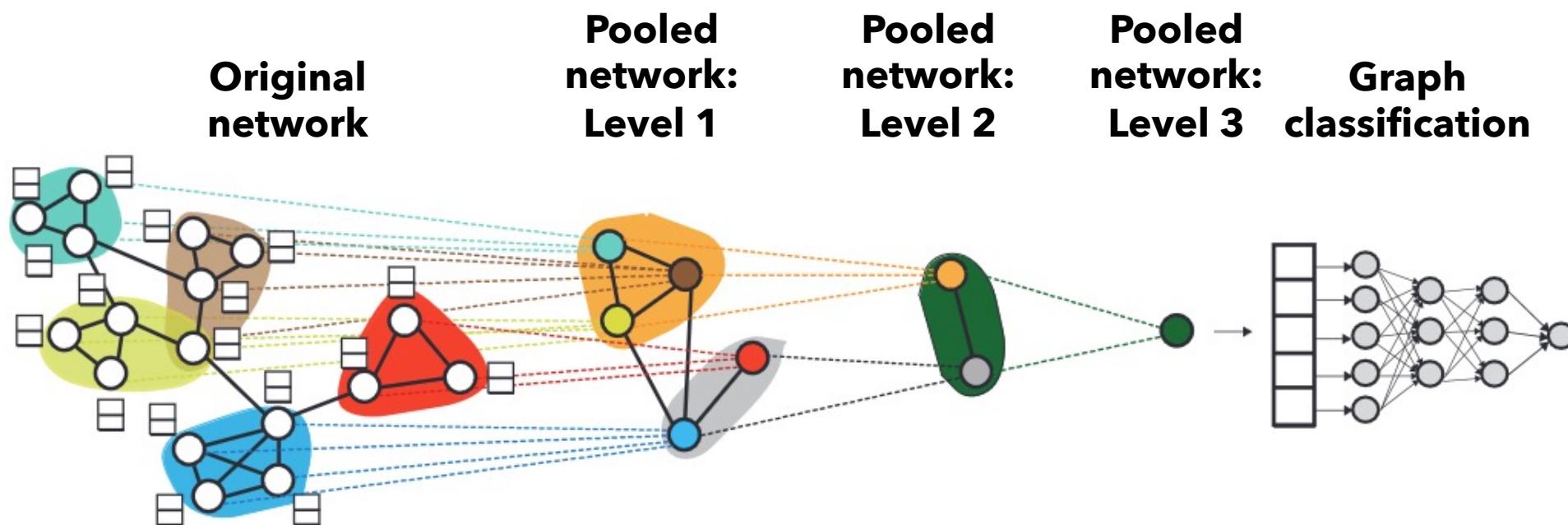
# Hierarchical global pooling

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

Can differentiate  
between  $G_1$  and  $G_2$

# Hierarchical pooling in practice

**DiffPool idea:** Hierarchically pool node embeddings



# Hierarchical pooling in practice

Leverage 2 independent GNNs at each level

- **GNN A:** Compute node embeddings
- **GNN B:** Compute the cluster that a node belongs to

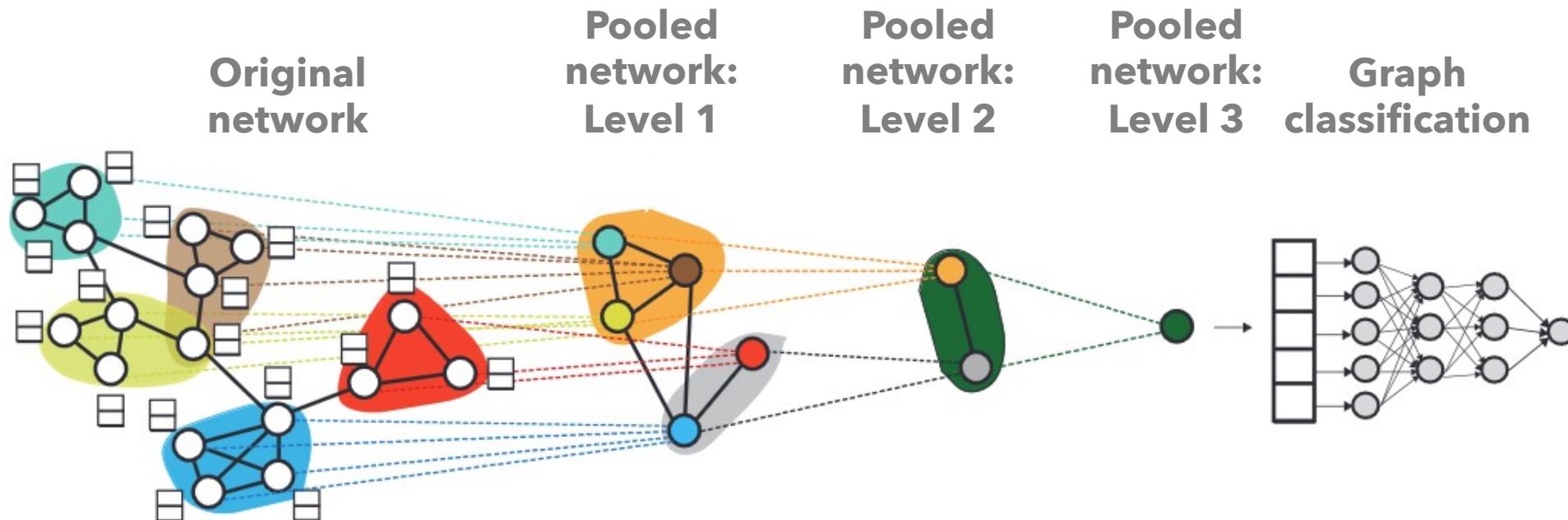
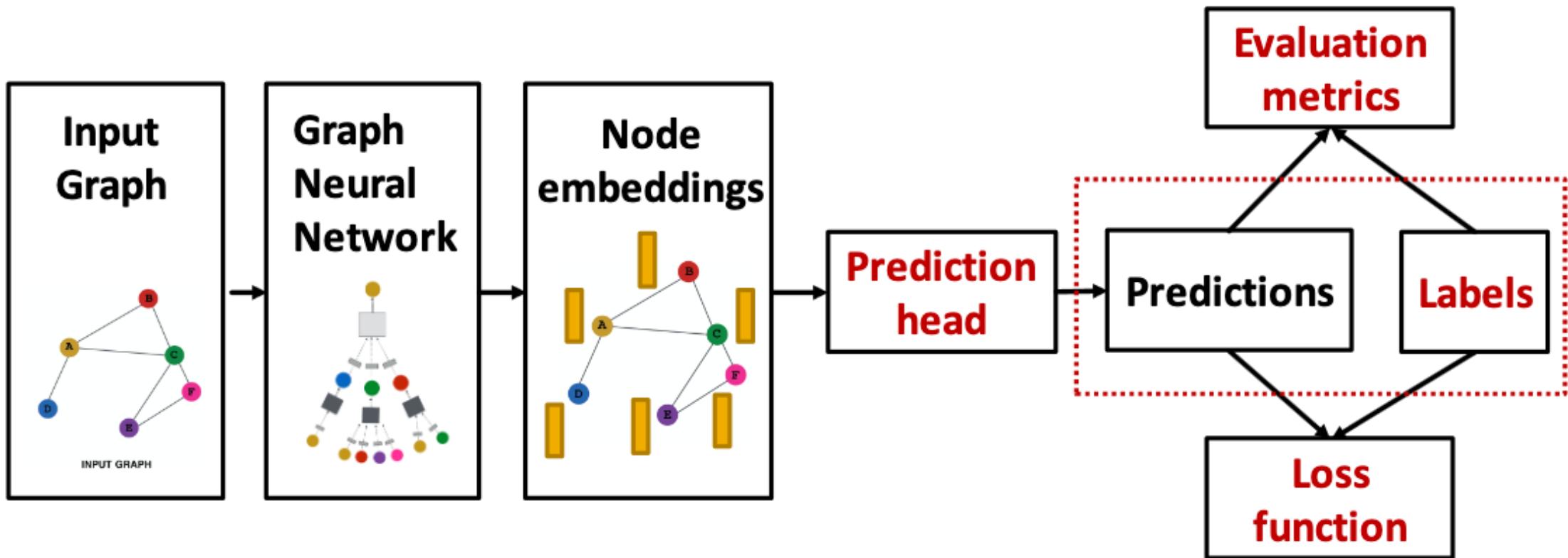


Figure by Ying et al. NeurIPS'18

# GNN Pipeline



# Supervised vs unsupervised

## **Supervised** learning on graphs

- Labels come from external sources
- E.g., predict drug likeness of a molecular graph

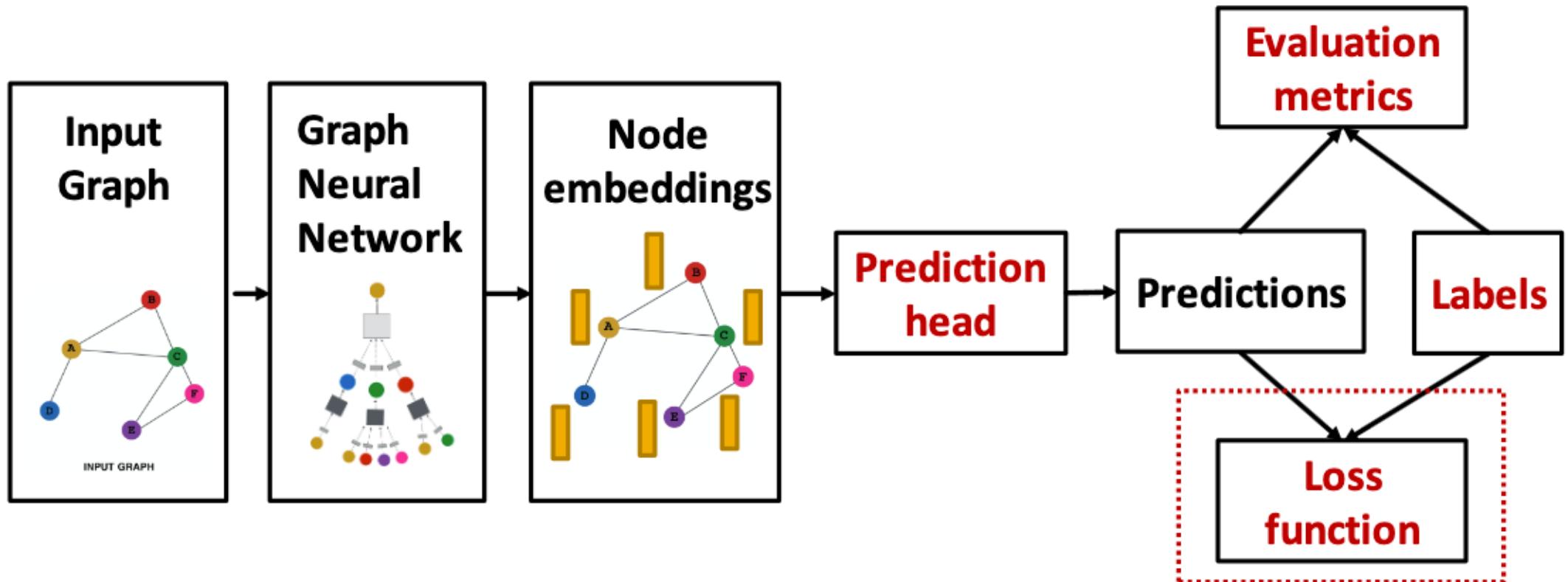
## **Unsupervised** learning on graphs

- Signals come from graphs themselves
- E.g., link prediction: delete edges, predict if 2 nodes are connected

Sometimes the differences are blurry

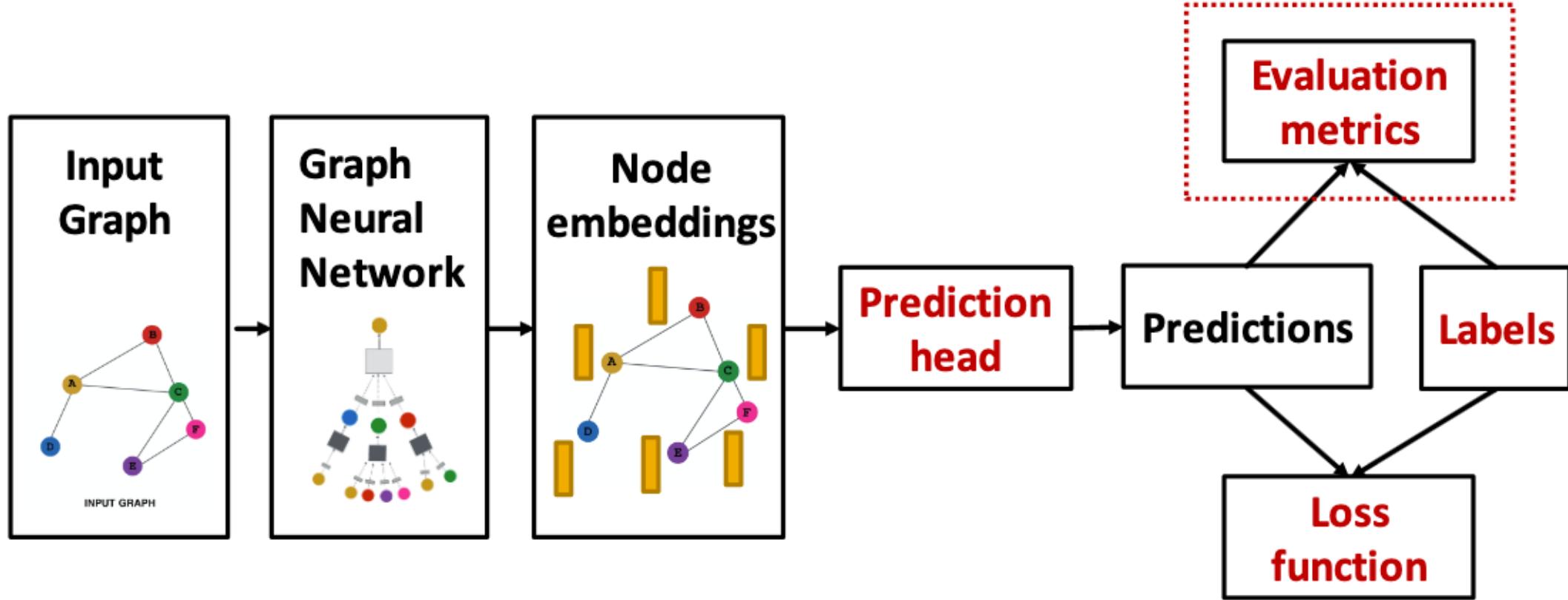
- We still have “supervision” in unsupervised learning
- Alternative name for “unsupervised” is “**self-supervised**”

# GNN pipeline



E.g., cross entropy for **classification**, MSE for **regression**

# Loss functions



E.g., accuracy:  $\frac{1}{N} \sum_{i=1}^N \mathbf{1}_{\{y_i \neq \hat{y}_i\}}$

# Outline

1. Introduction
2. Feature engineering for graphs
  - a. Node-level prediction
  - b. Edge-level prediction
3. GNN architecture
4. Training a GNN
  - a. GNN pipeline
  - b. Train/validation/test splits**
  - c. Skip connections
  - d. Graph manipulations

# Training, validation, and test sets

**Training set:** used for optimizing GNN parameters

**Validation set:** develop model/hyperparameters

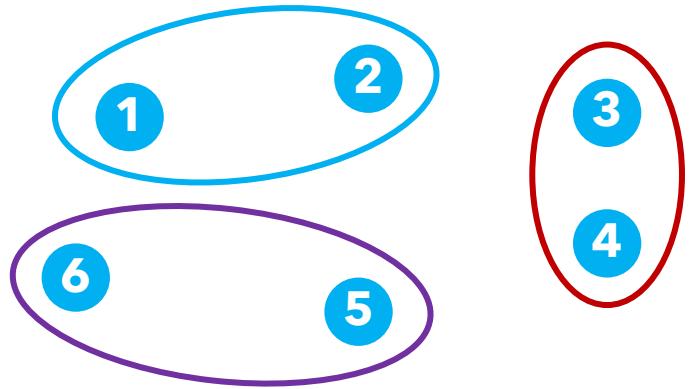
**Test set:** held out until we report final performance

# Why graphs are special

Suppose we want to split an image dataset

- Each data point is an image
- Data points are independent
- Image 5 will not affect our prediction on image 1

**Training**  
**Validation**  
**Test**

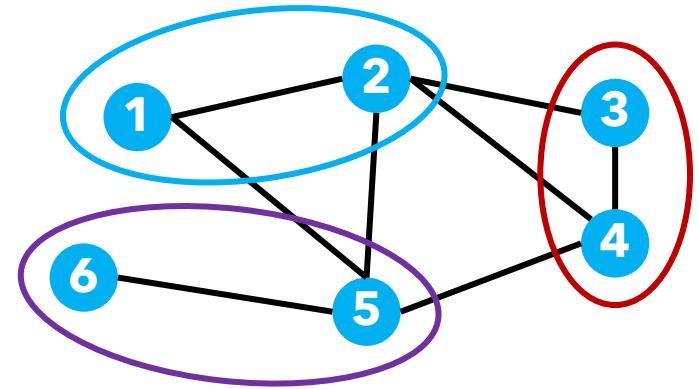


# Why graphs are special

Splitting a graph dataset is different

- Node classification: Each data point is a node
- Data points are NOT independent
  - Node 5 will affect our prediction on node 1 due to message passing

Training  
Validation  
Test

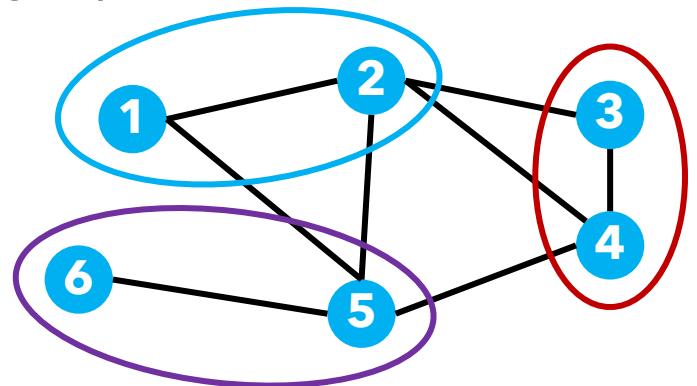


# Transductive learning

## Solution 1 (**Transductive setting**):

- Input graph can be observed in all the dataset splits
  - Training, validation and test set
- Only split the (node) labels
- Training: compute embeddings using entire graph
  - Train using node 1&2's labels
- Validation: compute embeddings using entire graph
  - Evaluate on node 3&4's labels

**Training**  
**Validation**  
**Test**

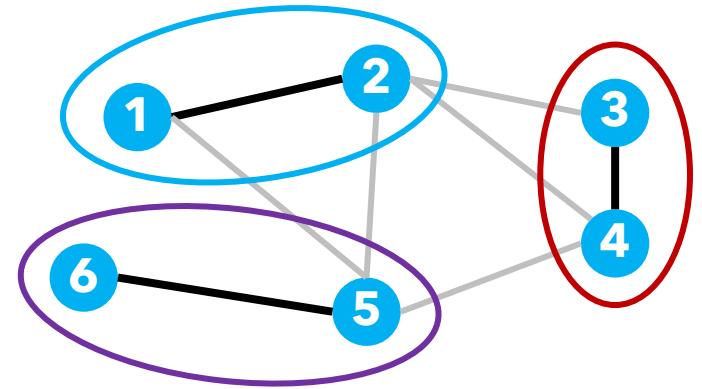


# Inductive learning

## Solution 2 (Inductive setting):

- Break the edges between splits to get multiple graphs
- 3 graphs are independent: node 5 won't affect prediction on node 1
- Training: compute embeddings using graph over node 1&2
  - Train using node 1&2's labels
- Validation: compute embeddings using the graph over node 3&4
  - Evaluate on node 3&4's labels

Training  
Validation  
Test



# Transductive vs inductive

## **Transductive setting:**

- Training / validation / test sets are on the same graph
- Dataset consists of one graph
- Entire graph can be observed in all dataset splits: only split labels
- Only applicable to node / edge prediction tasks

## **Inductive setting:**

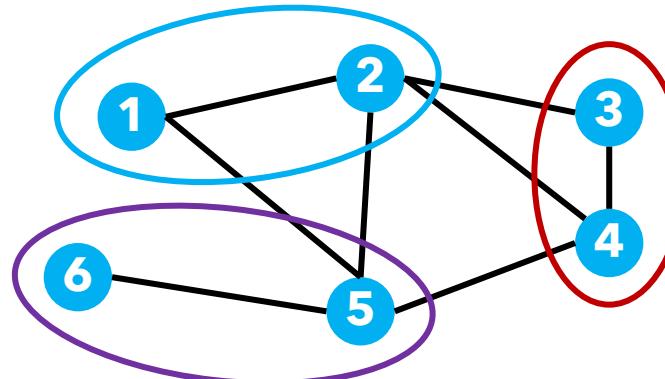
- Training / validation / test sets are on different graphs
- Dataset consists of multiple graphs
- Each split can only observe the graph(s) within the split
- Successful model should generalize to unseen graphs
- Applicable to node / edge / graph tasks

# Example: Node classification

## Transductive setting:

- All splits can observe the entire graph structure
- Can only observe the labels of their respective nodes

Training  
Validation  
Test

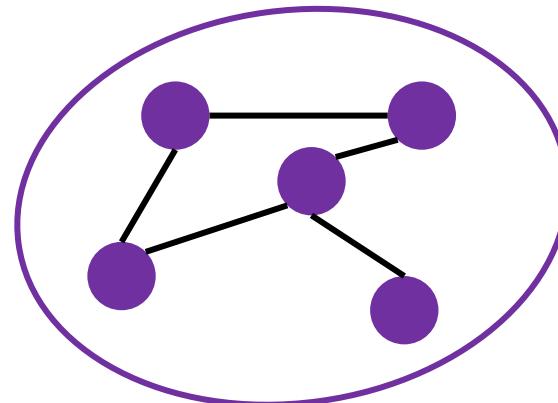
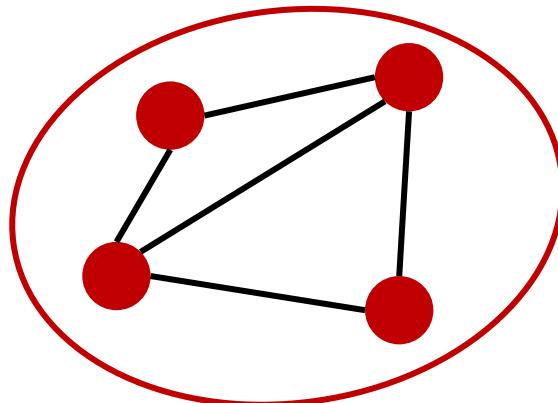
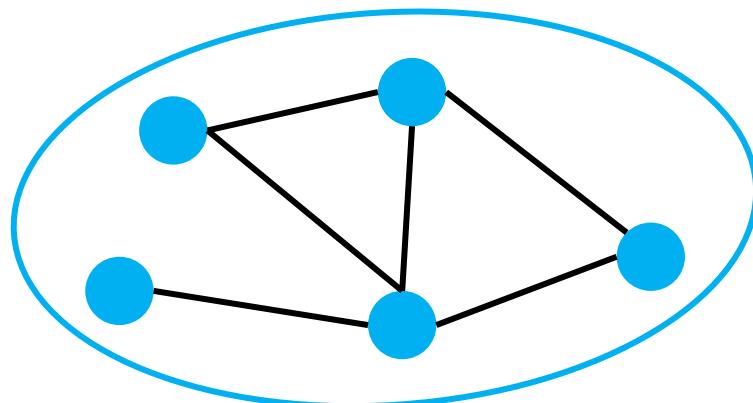


# Example: Node classification

## Inductive setting:

- Suppose have a dataset of 3 graphs
- Each split contains a different graph

Training  
Validation  
Test

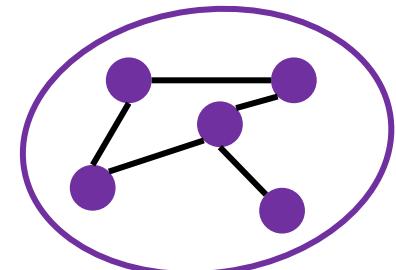
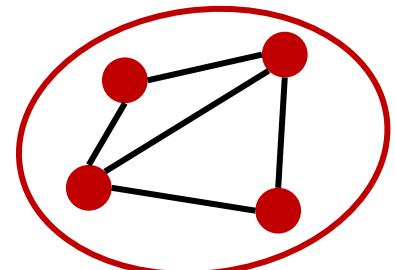
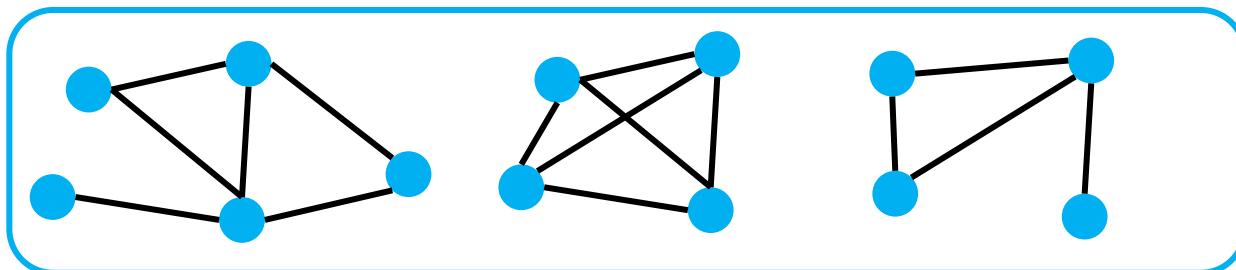


# Example: Graph classification

Only the **inductive** setting is well defined for graph classification

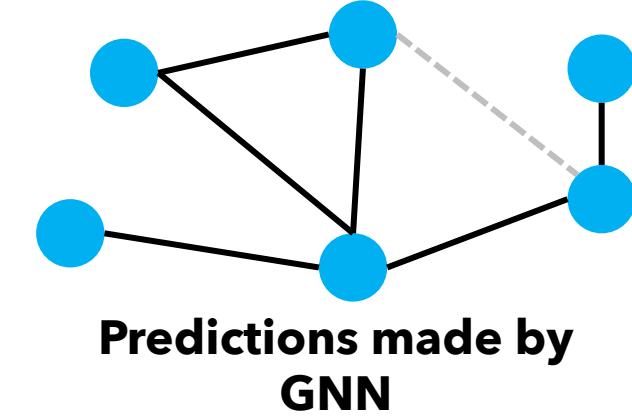
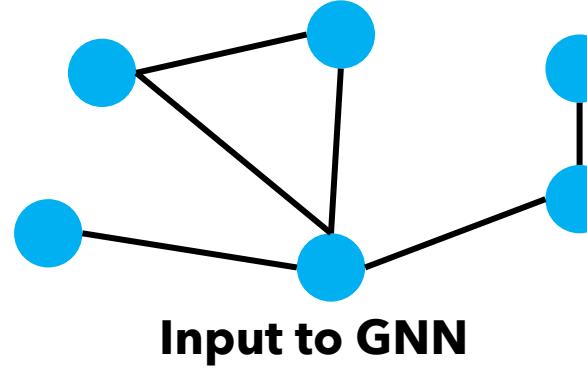
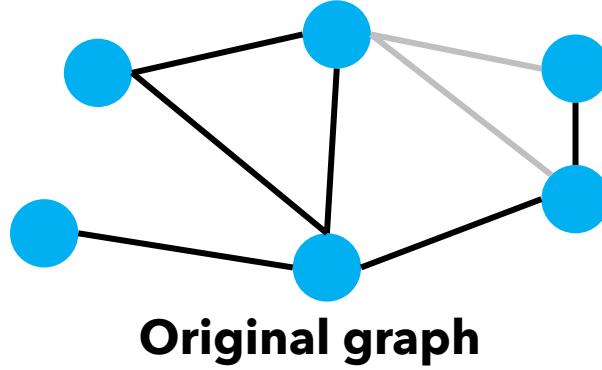
- Have to test on unseen graphs
- Suppose we have a dataset of 5 graphs
  - Each split will contain independent graph(s)

**Training**  
**Validation**  
**Test**



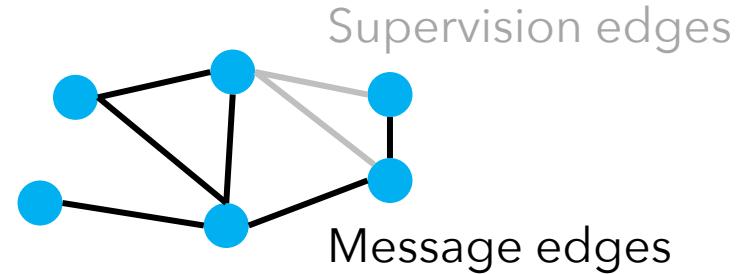
# Example: Link prediction

- **Goal:** predict missing edges
- Link prediction is an unsupervised / self-supervised task
- Need to hide some edges from the GNN
  - Let the GNN predict if the edges exist



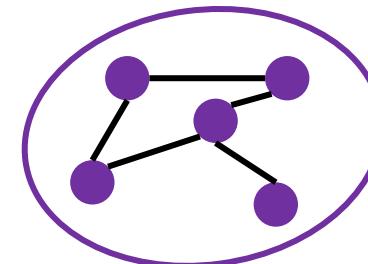
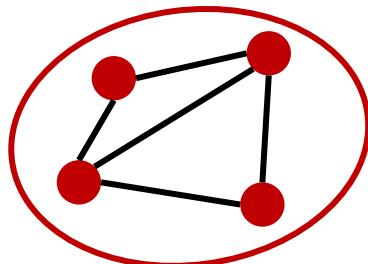
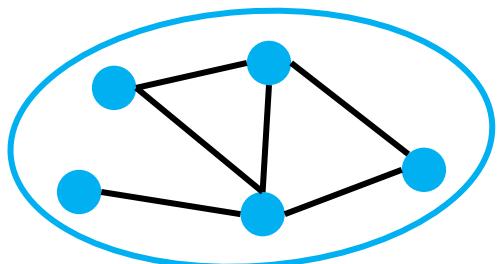
# Setting up link prediction

- For link prediction, we'll **split edges twice**
- Step 1: Assign **two 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 used as supervision for model's predictions  
*Will not be fed into GNN!*



# Setting up link prediction

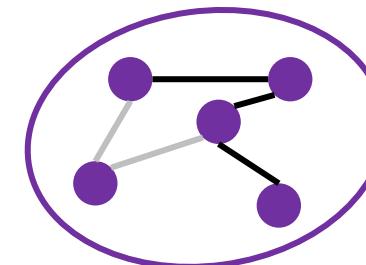
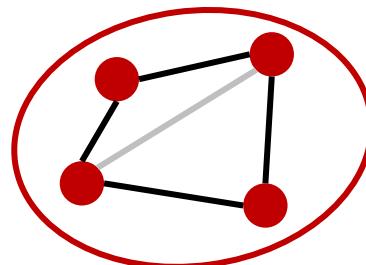
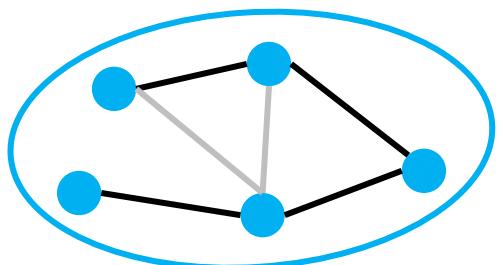
- Step 2: Split edges into train / validation / test
- Option 1: **Inductive** link prediction split
  - Suppose we have a dataset of 3 graphs
  - Each inductive split will contain an independent graph



**Training**  
**Validation**  
**Test**

# 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/val/test set, each graph will have 2 types of edges:
    - Message edges
    - Supervision edges (not the input to GNN)

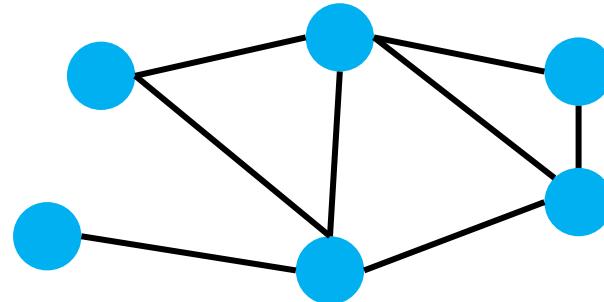


**Training**  
**Validation**  
**Test**

# Setting up link prediction

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

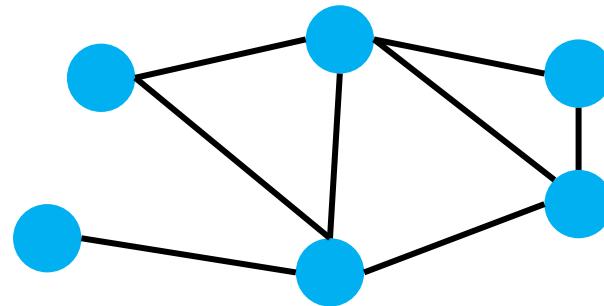
- Default setting when people talk about link prediction
- Suppose we have a dataset of 1 graph



# Setting up link prediction

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

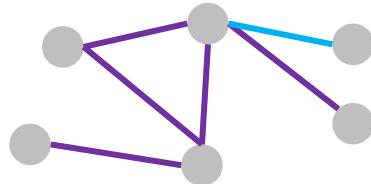
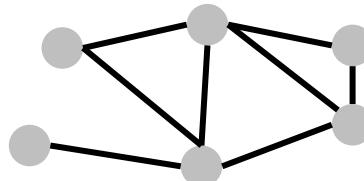
- Entire graph can be observed in all dataset splits
- Need to hold out validation / test edges
- To train, must hold out supervision edges for the training set



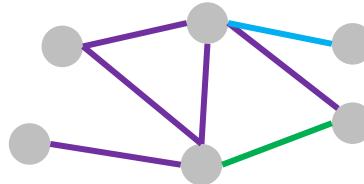
# Setting up link prediction

Option 2: **Transductive** link prediction split

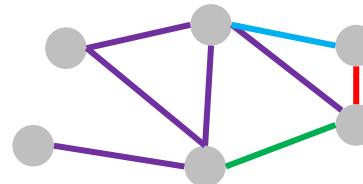
**Original graph**



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



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

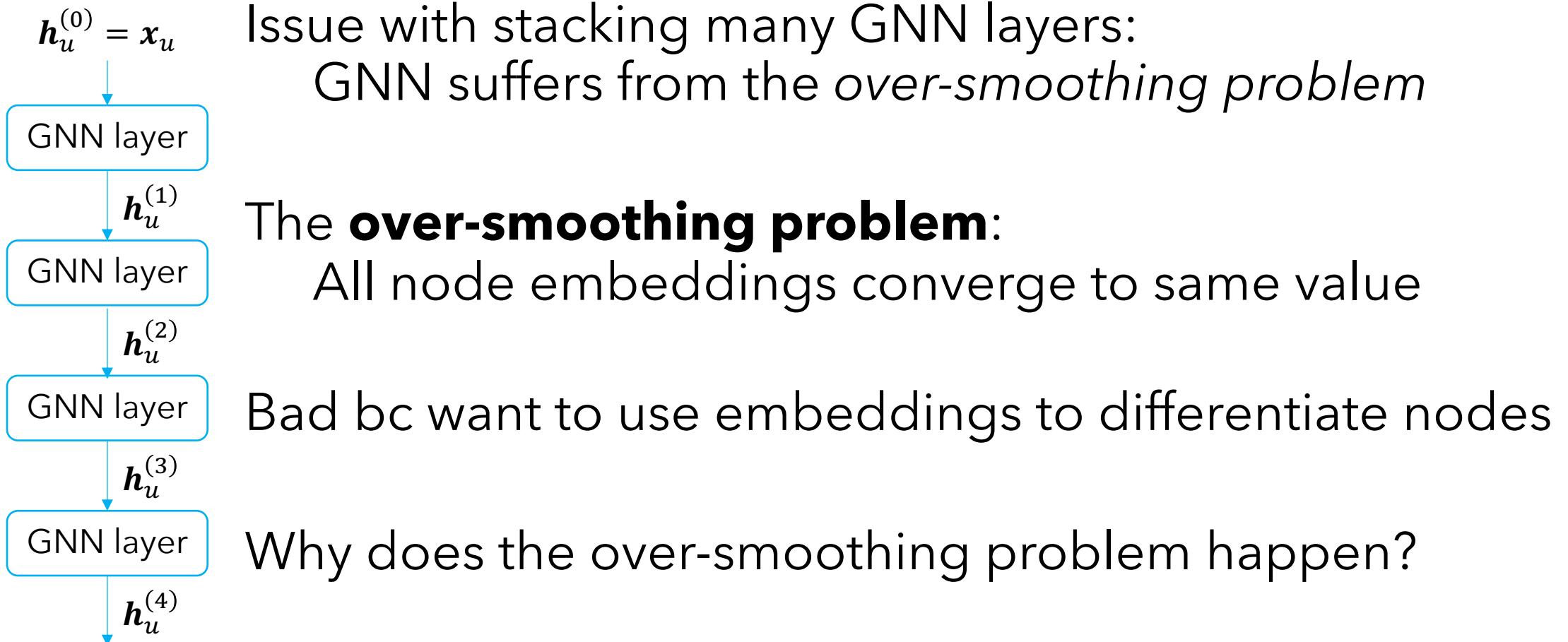


**(3) At test time:**  
Use **training message** + **supervision** edges and **validation** edges to predict **test** edges

# Outline

1. Introduction
2. Feature engineering for graphs
3. GNN architecture
4. Training a GNN
  - a. GNN pipeline
  - b. Train/validation/test splits
  - c. Skip connections**
  - d. Graph manipulations

# Over-smoothing problem



# Receptive field of a GNN

- Receptive field:  
Set of nodes that determine embedding of node of interest
- $K$ -layer GNN: node's receptive field is its  $K$ -hop neighborhood

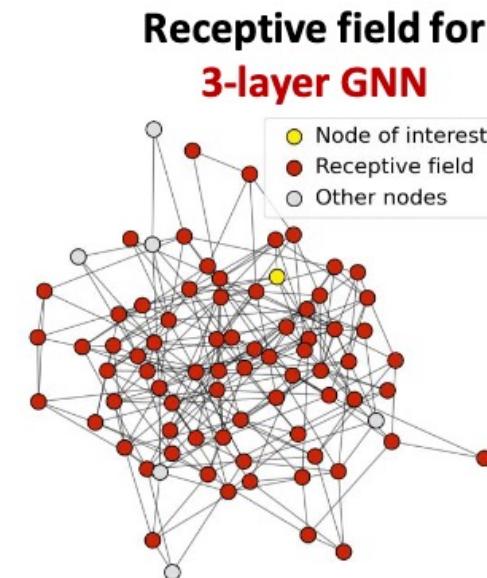
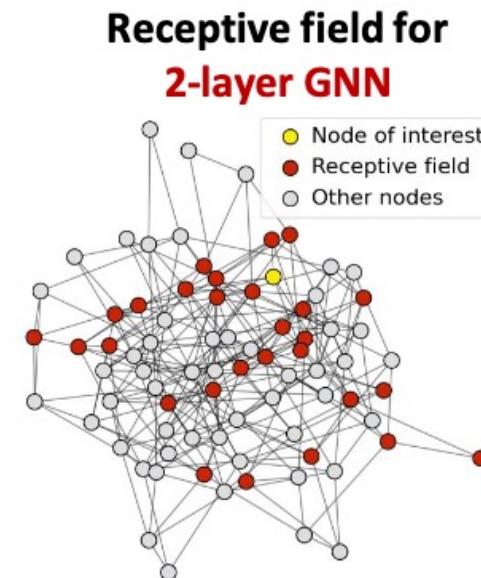
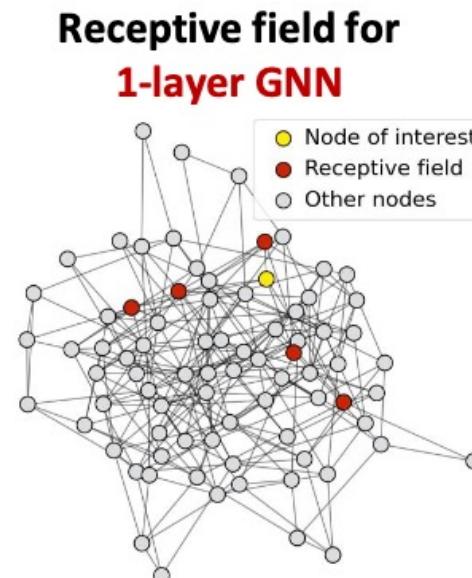


Figure by Leskovec

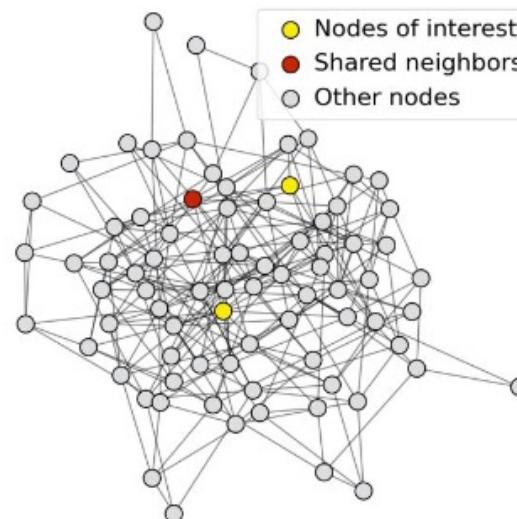
# Receptive field of a GNN

Receptive field **overlap** for two nodes

Shared neighbors quickly grows when we increase # GNN layers

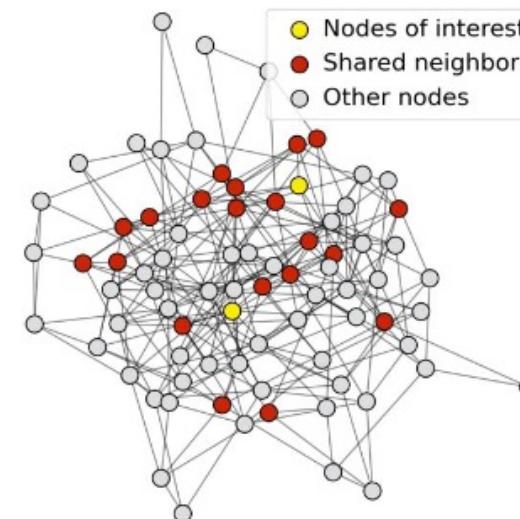
## 1-hop neighbor overlap

Only 1 node



## 2-hop neighbor overlap

About 20 nodes



## 3-hop neighbor overlap

Almost all the nodes!

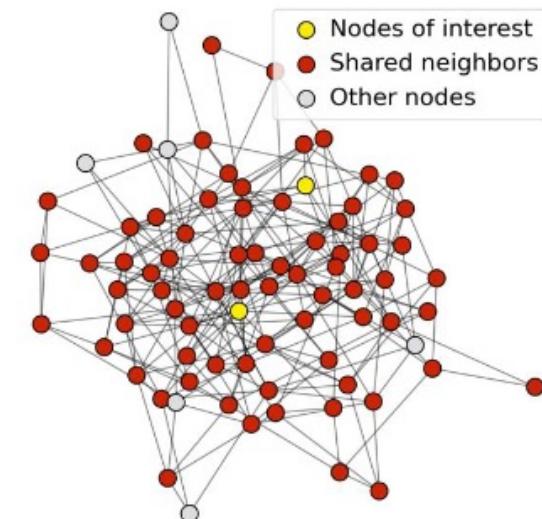


Figure by Leskovec

# Receptive field and oversmoothing

Can explain over-smoothing via **receptive fields**

Embedding of a node is determined by its receptive field

- Nodes have very overlapped receptive fields  $\Rightarrow$  similar embeddings
- Stack many GNN layers
  - $\rightarrow$  nodes will have highly-overlapped receptive fields
  - $\rightarrow$  node embeddings will be highly similar
  - $\rightarrow$  suffer from the over-smoothing problem

How to overcome over-smoothing problem?

# Design GNN layer connectivity

Lesson: Be cautious when adding GNN layers

Adding GNN layers doesn't always help, unlike NNs in other domains

**Step 1:** Analyze necessary receptive field to solve problem

E.g., by computing graph's diameter

**Step 2:** Set # GNN layers to be a bit more than receptive field

# Skip connections

What if my problem still requires many GNN layers?

## **Observation:**

Embeddings in early layers can better differentiate nodes

**Solution:** add **shortcuts** in GNN (skip connections)

# Skip connections

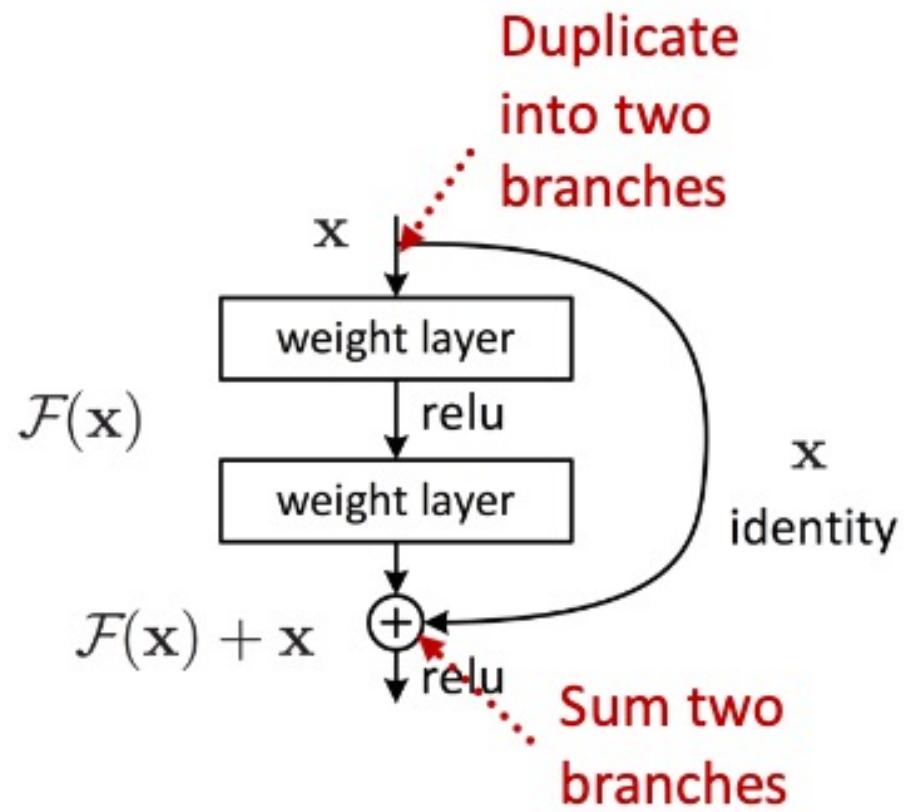
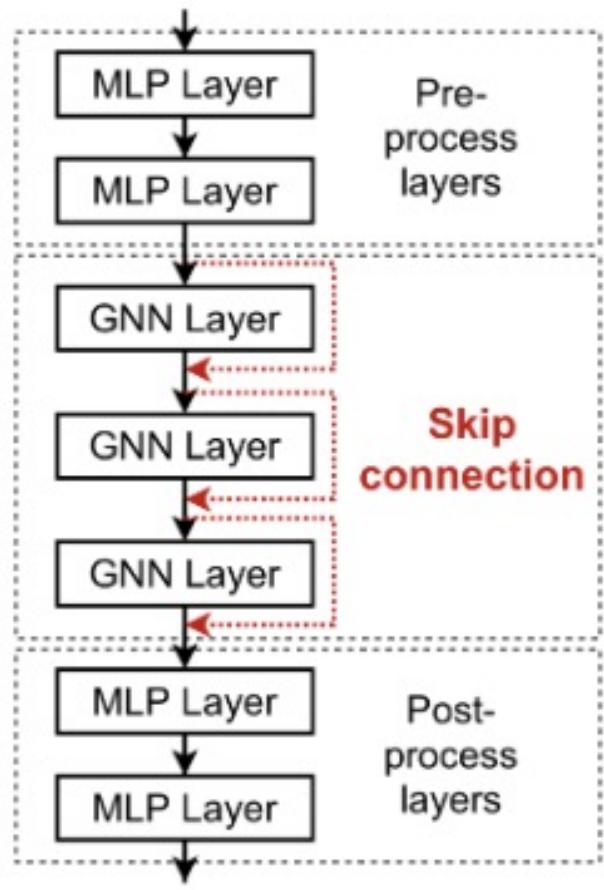


Figure by Leskovec

# Outline

1. Introduction
2. Feature engineering for graphs
3. GNN architecture
4. Training a GNN
  - a. GNN pipeline
  - b. Train/validation/test splits
  - c. Skip connections
  - d. Graph manipulations**

# Node feature augmentation

Useful if, e.g., input graph does not have node features  
Common when we only have the adj. matrix

Standard approach: assign unique IDs to nodes

- 👍 High expressive power
- 👎 Can't generalize to new nodes
- 👎 High computational cost (many features)

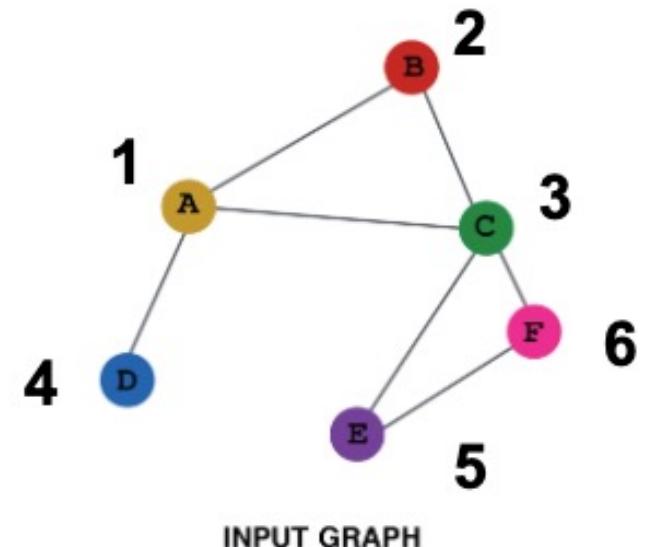
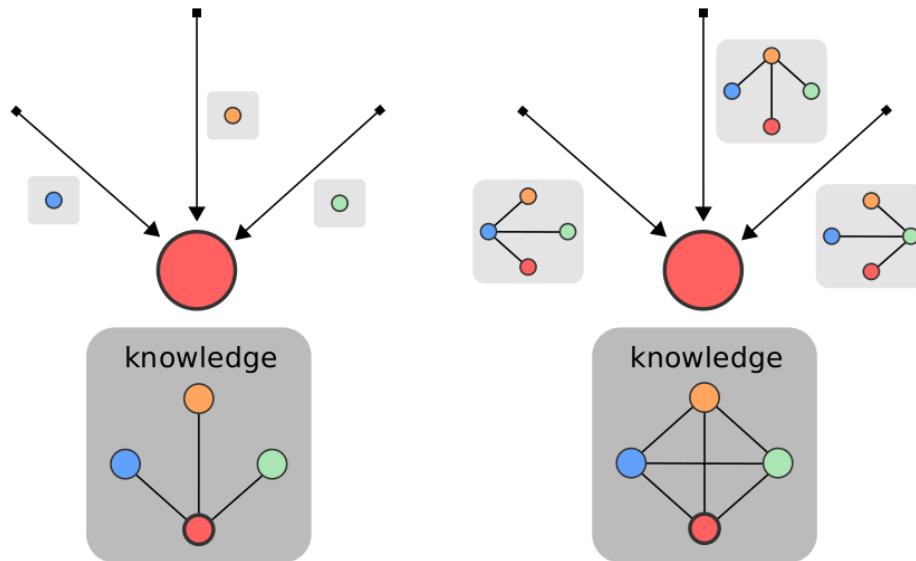


Figure by Leskovec

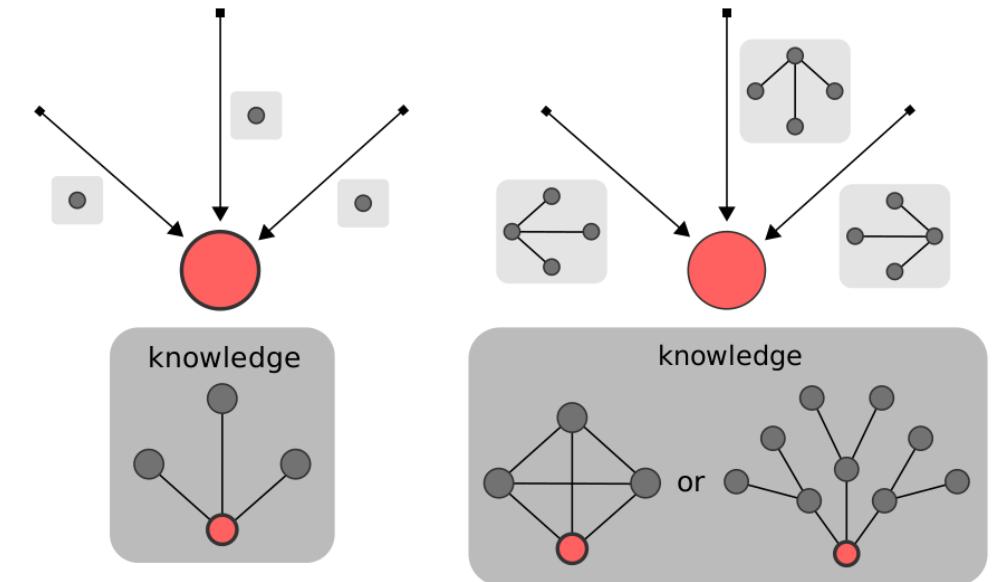
# Why do we need feature augmentation?

Certain structures are hard to learn by GNN

E.g., cycle count



Nodes **can** distinguish each other



Nodes **can't** distinguish each other

# Outline

1. Introduction
2. Feature engineering for graphs
  - a. Node-level prediction
  - b. Edge-level prediction
3. GNN architecture
4. Training a GNN
  - a. GNN pipeline
  - b. Train/validation/test splits
  - c. Skip connections
  - d. Graph manipulations

# Papers we'll read

Veličković, Petar, et al. "Neural execution of graph algorithms."  
*ICLR*. 2020.

- GNNs don't work off-the-shelf for combinatorial tasks
- How to **align** GNN architectures to these tasks

Cappart, Quentin, et al. "Combinatorial optimization and reasoning with GNNs." *arXiv*.

- **Broad overview** of the field; current & future directions