Department of Computer Engineering
University of Kurdistan

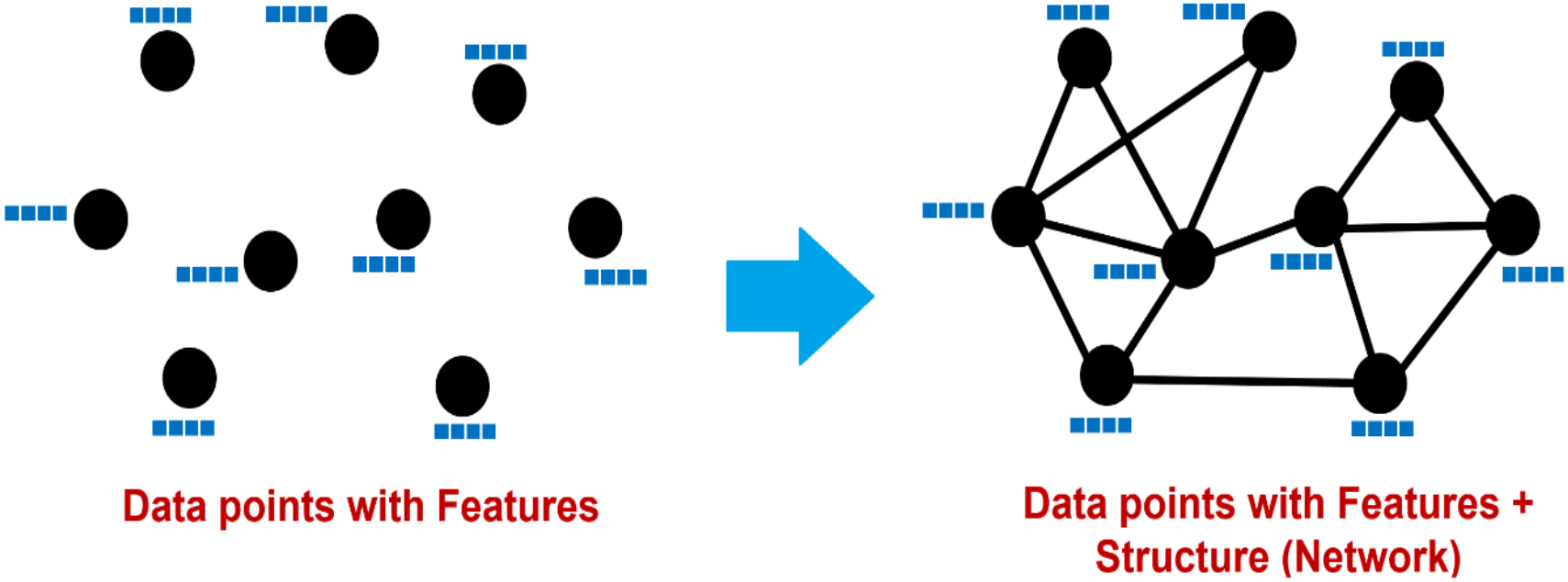
Complex Networks

Graph Machine Learning

By: Dr. Alireza Abdollahpouri

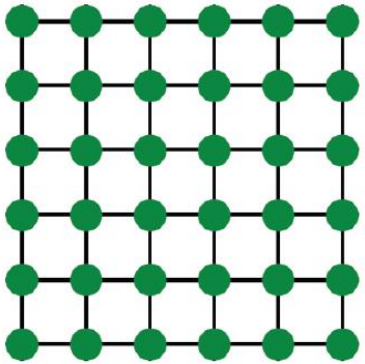
(Almost all the slides are taken from Stanford's CS224W)

Graph Machine Learning

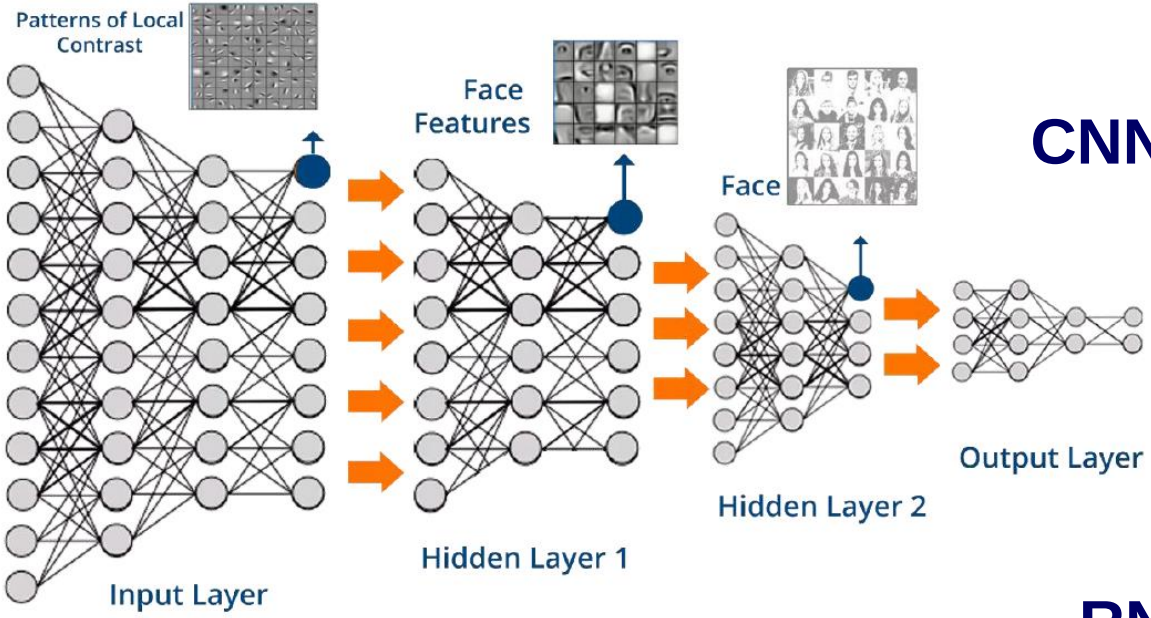


Traditional machine learning relies on pre-defined features from isolated data points, while graph machine learning leverages both features and relations between entities to capture complex dependencies in networked data

Why Graph Machine Learning?

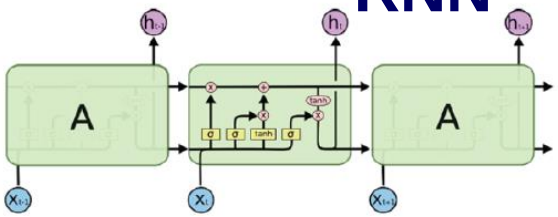


Images



Text/Speech

RNN

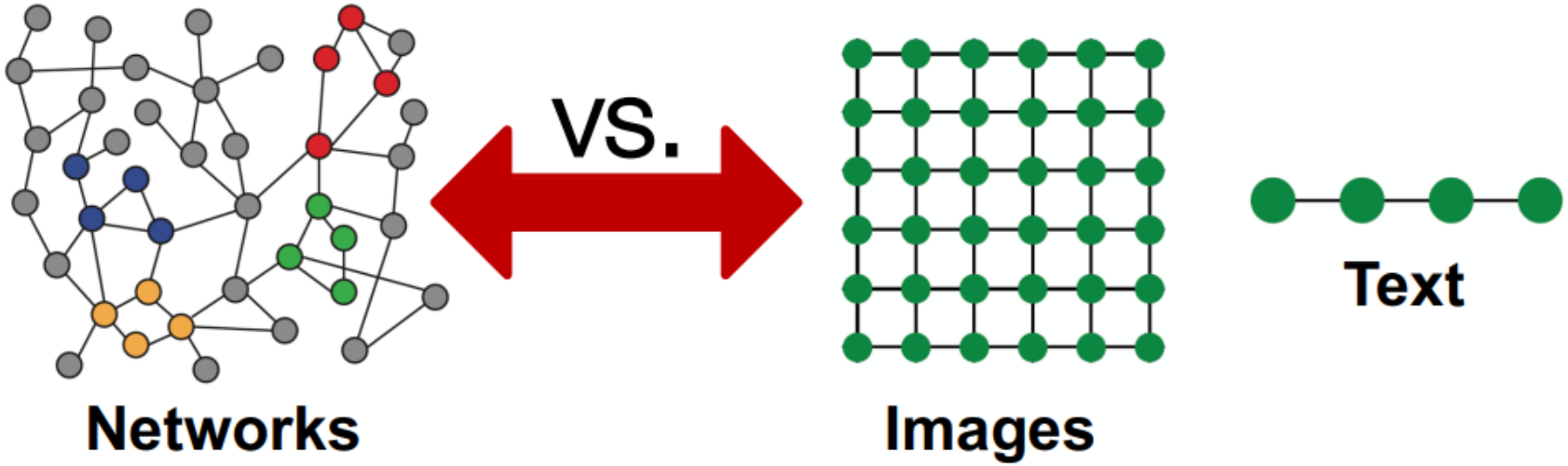


Modern deep learning toolbox is designed for simple sequences & grids

Not everything can be represented as a sequence or a grid

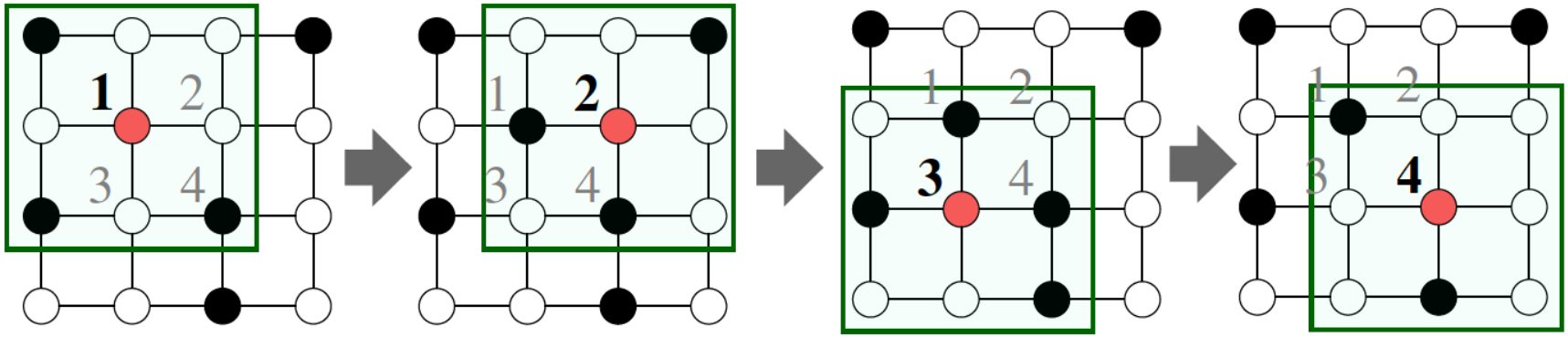
Networks are complex

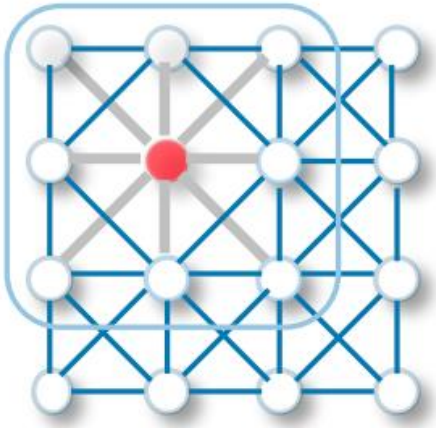
- Arbitrary size and complex topological structure (*i.e.*, no spatial locality like grids)

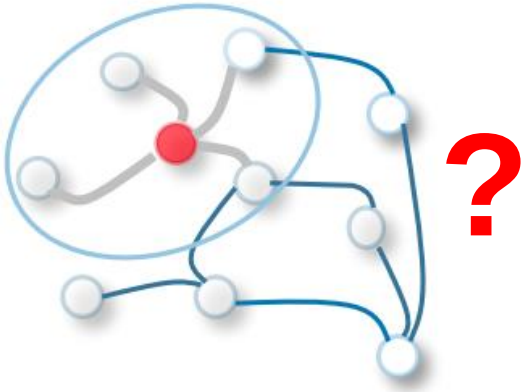


- No fixed node ordering or reference point
- Often dynamic and have multimodal features

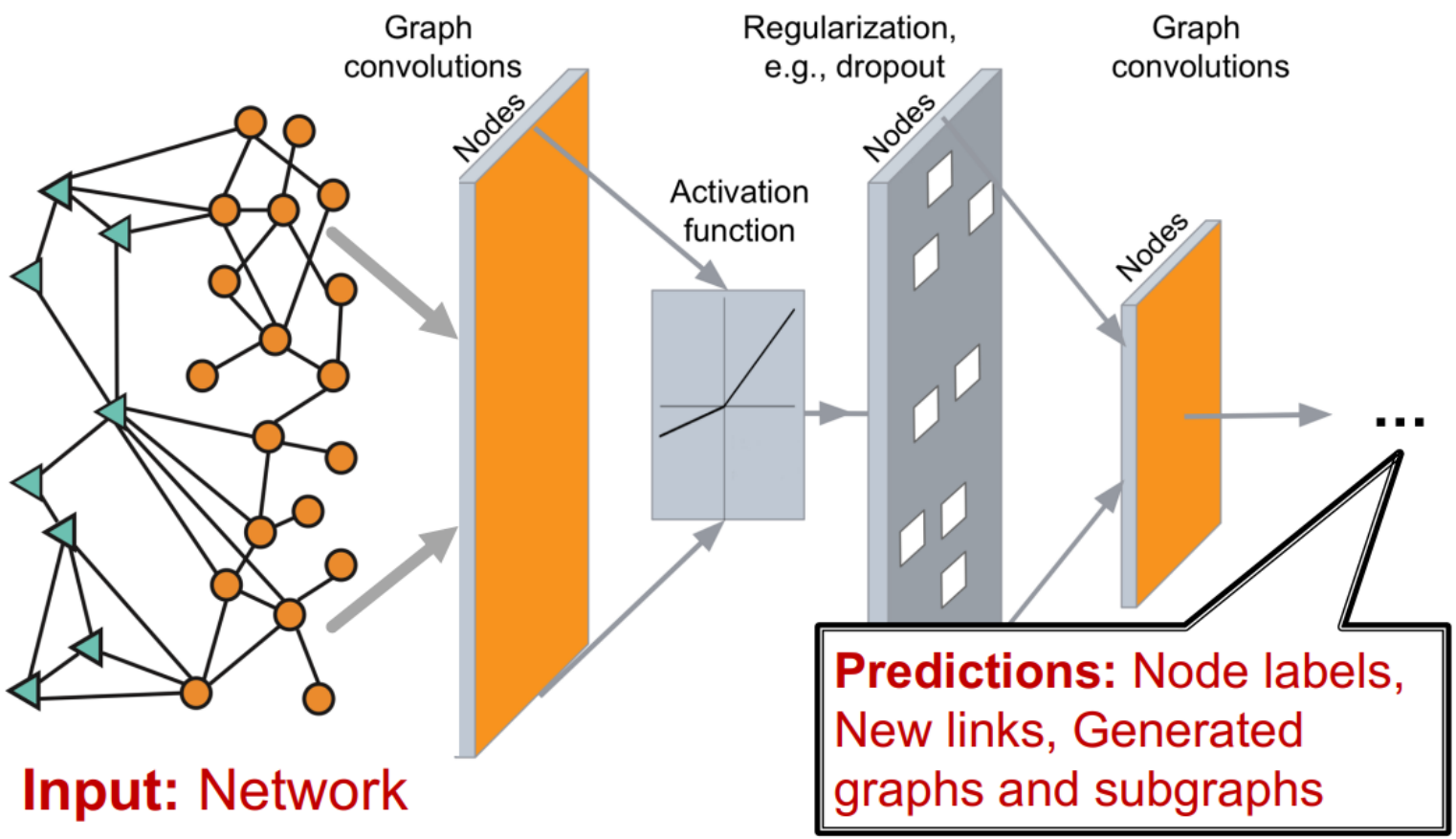
Why Is It Hard?





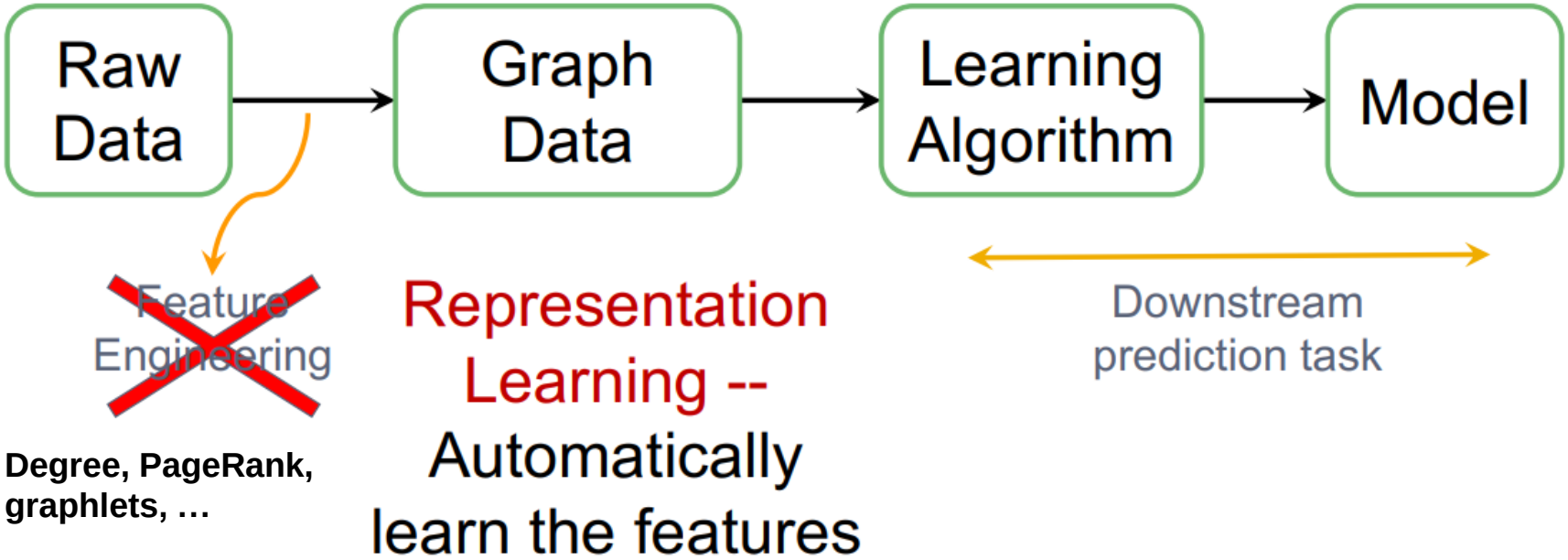


Deep Learning in Graphs



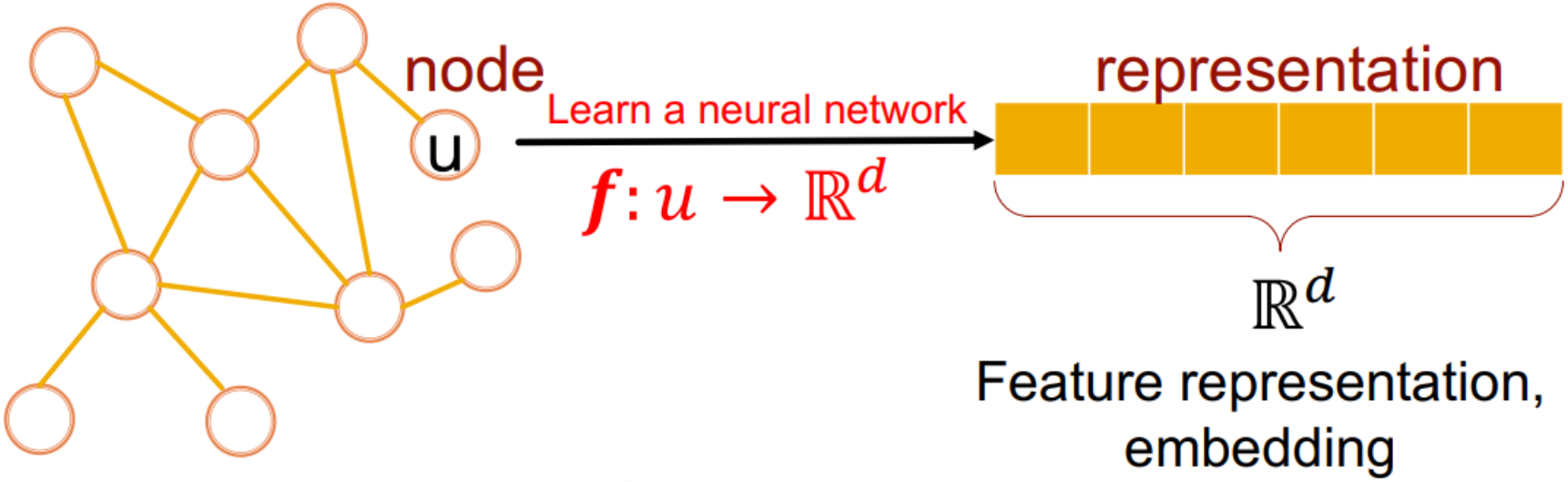
Key point: “Representation Learning”

(Supervised) Machine Learning Lifecycle: This feature, that feature.
Every single time!



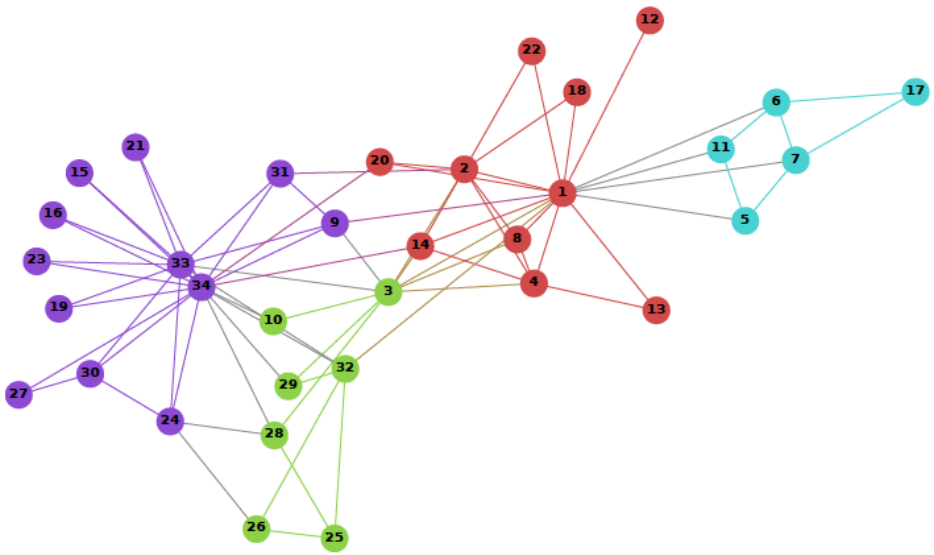
Deep graph representation learning

To learn a low-dimensional dense vector that encodes node structures and attributes, enables efficient feature learning for graph-structured data

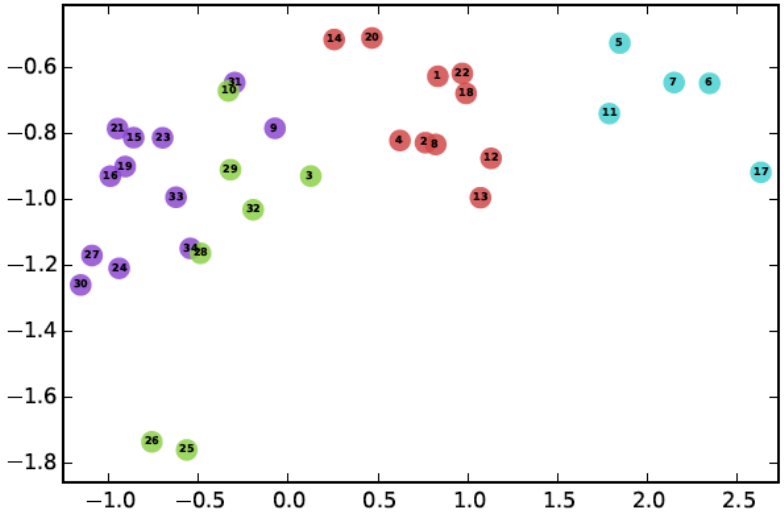


Example

- Zachary's Karate Club Network:

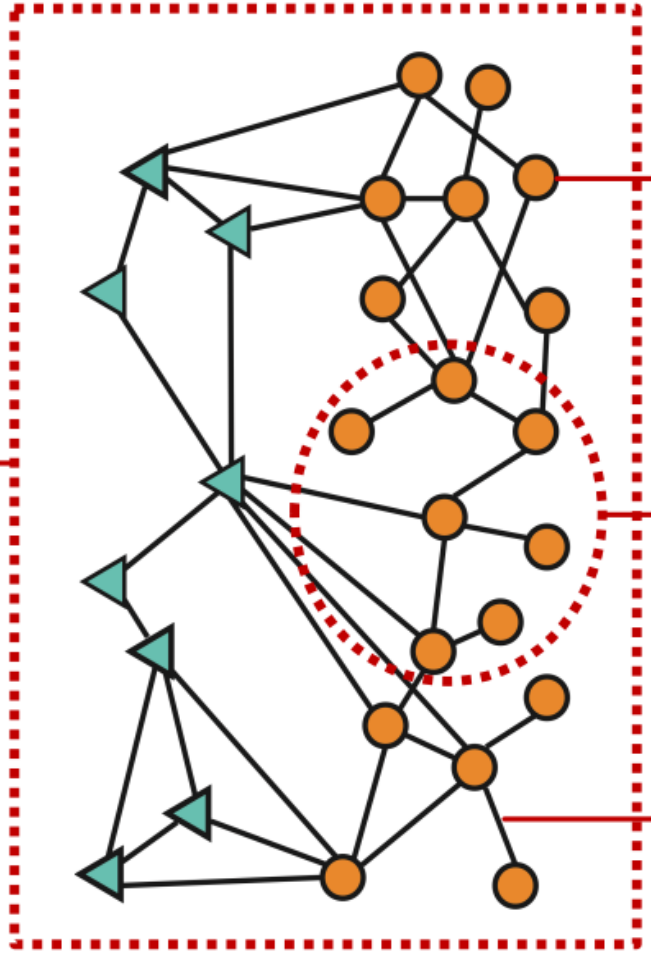


Input



Output

Diverse level of tasks



Node level

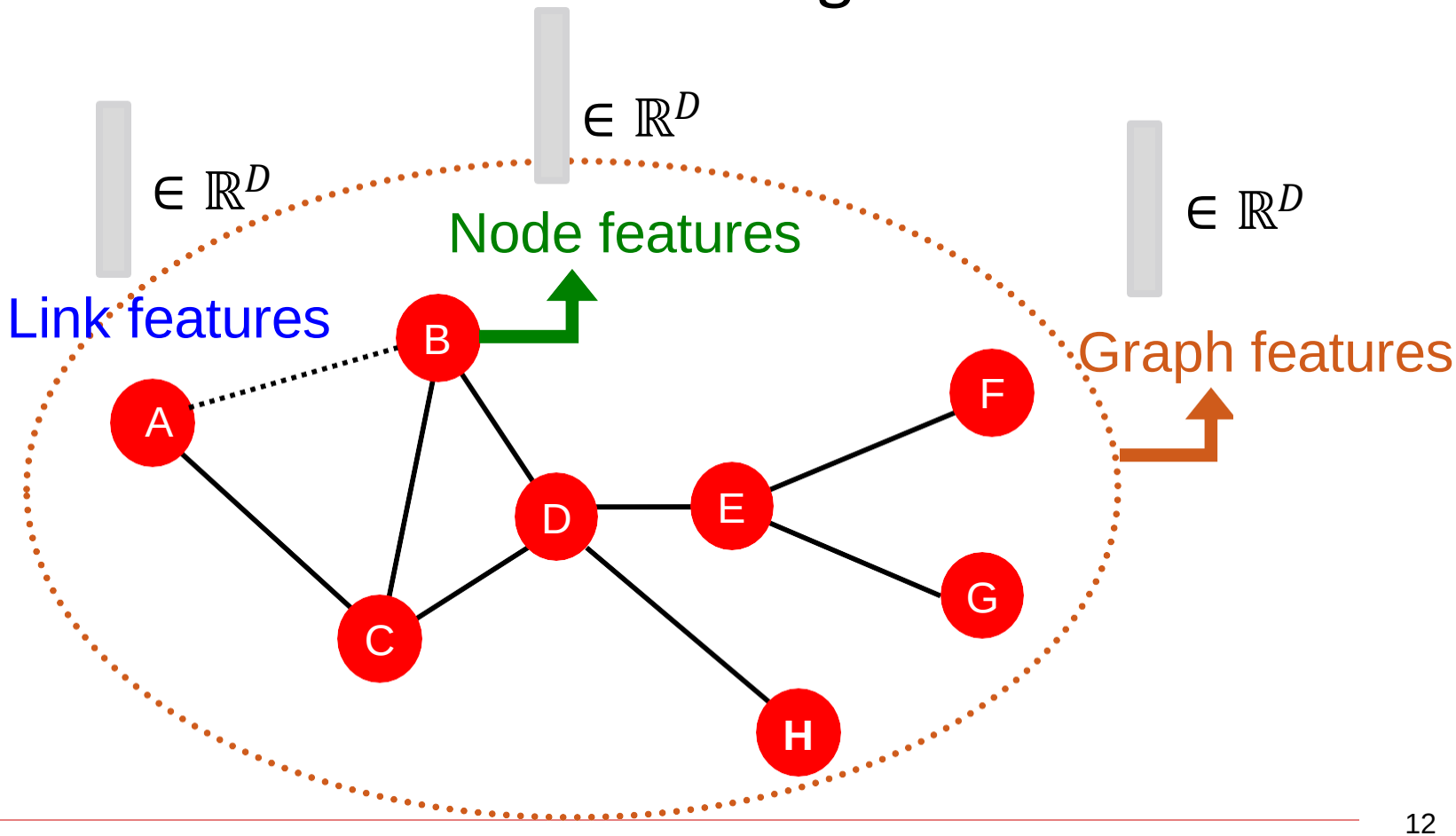
Graph-level prediction, Graph generation

Community (subgraph) level

Edge-level

Traditional ML Pipeline

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



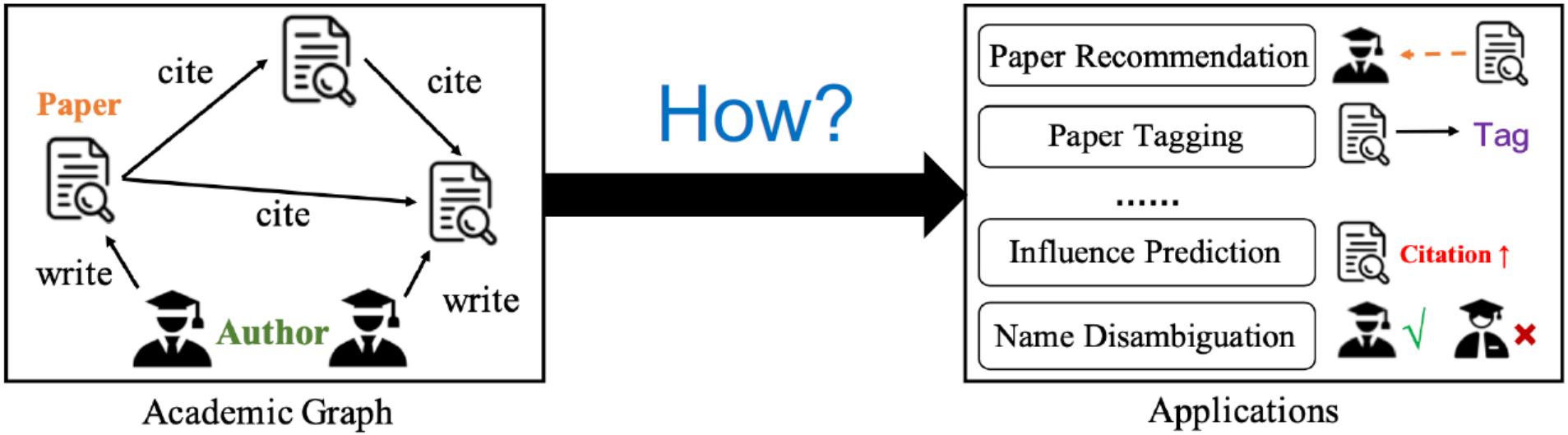
Machine Learning with Networks

- **Node classification**
  - Predict a type of a given node (categorizing users/items)
- **Link prediction**
  - Predict whether two nodes are linked (knowledge graph completion, Friend recommendation)
- **Community detection**
  - Identify densely linked clusters of nodes
- **Network similarity**
  - How similar are two (sub)networks
- **Graph Classification**
  - Categorize different graphs(Molecule property prediction)

Example: Academic Graph Mining

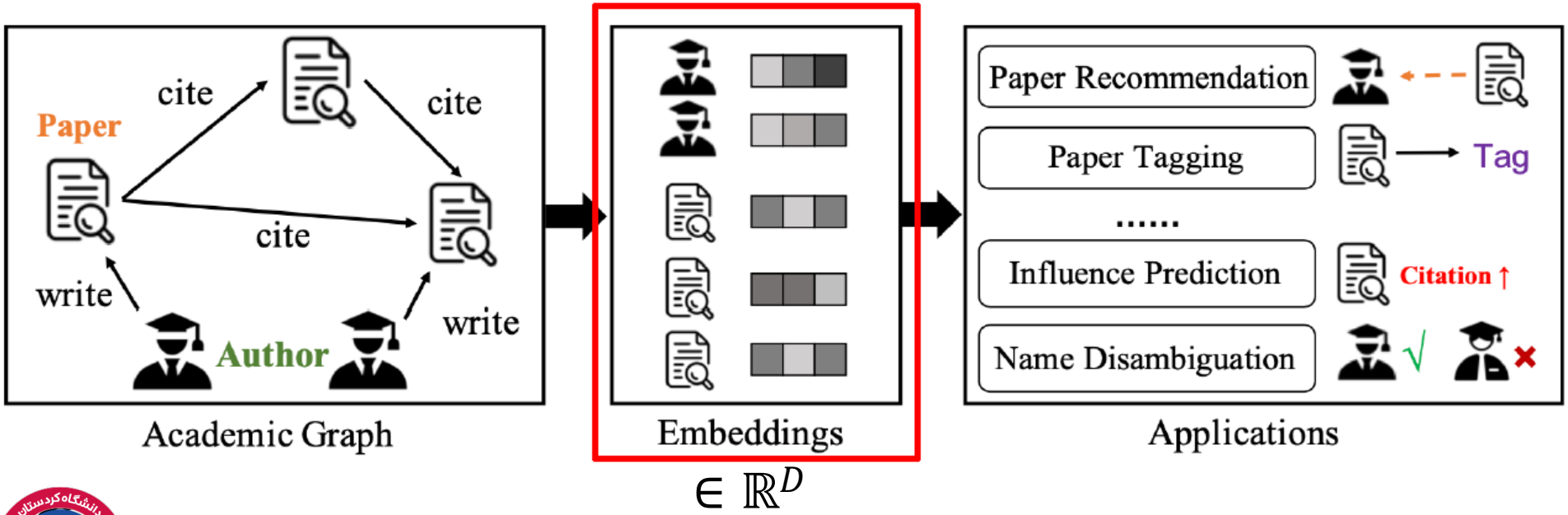
- **Input:**
  – an academic graph (papers, citation links, …)

- **Applications:**
  – recommendation, tagging, disambiguation, …



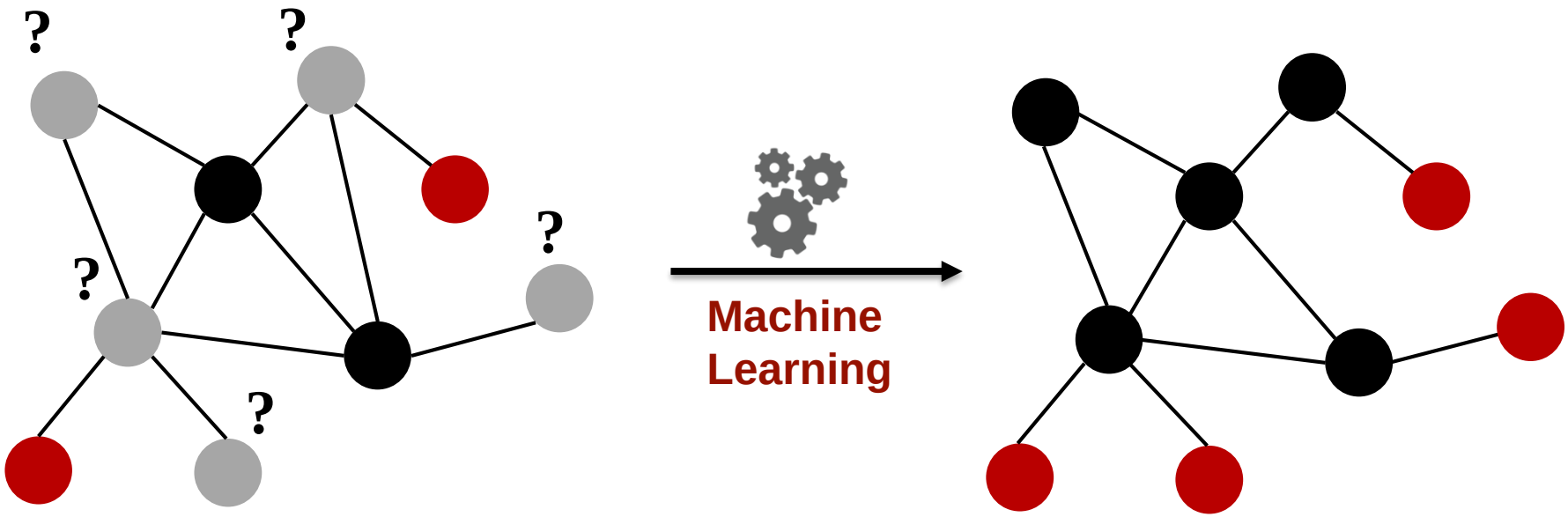
Question

- How to represent a node in a graph to help downstream tasks?
- Node Embedding!



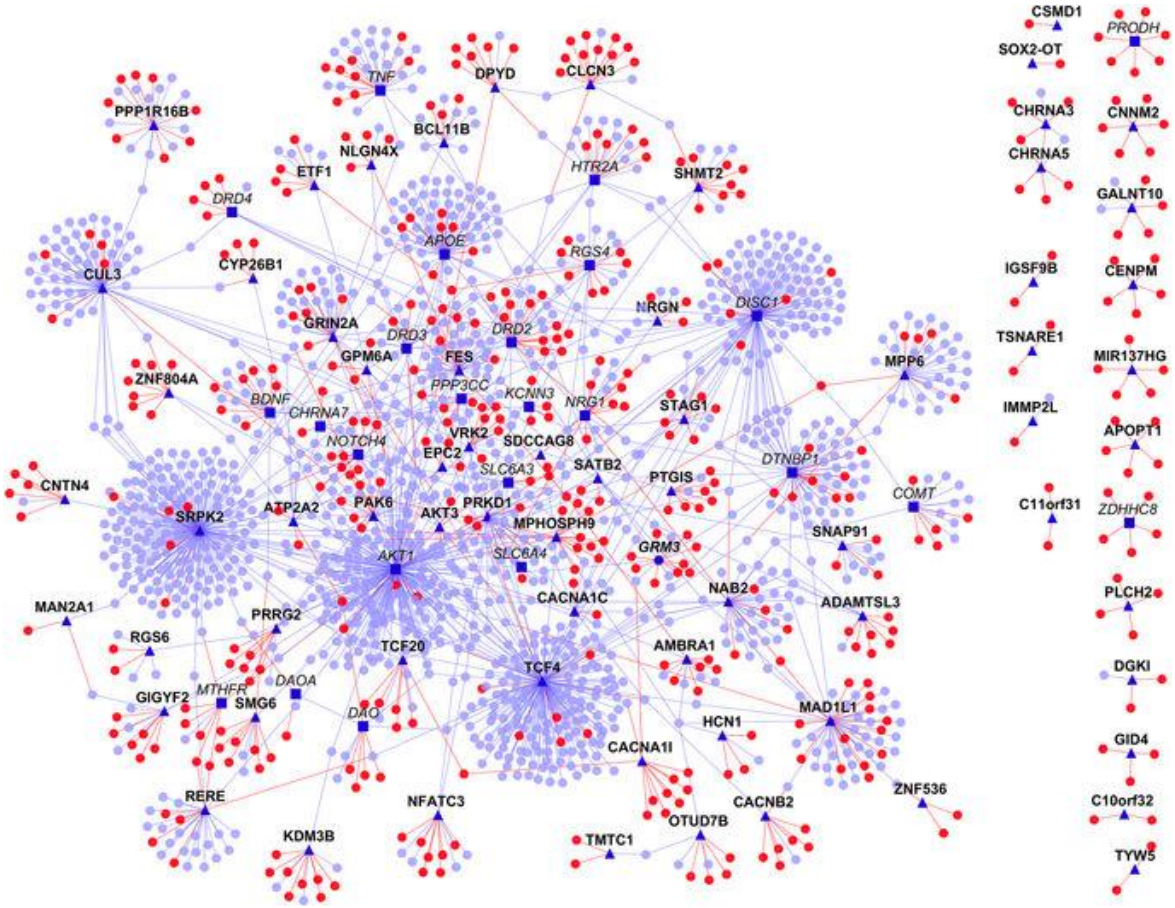
Examples of Node-Level Tasks

Example: Node Classification

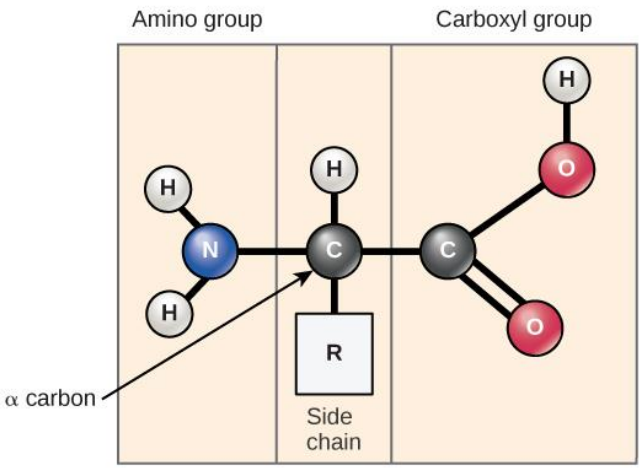


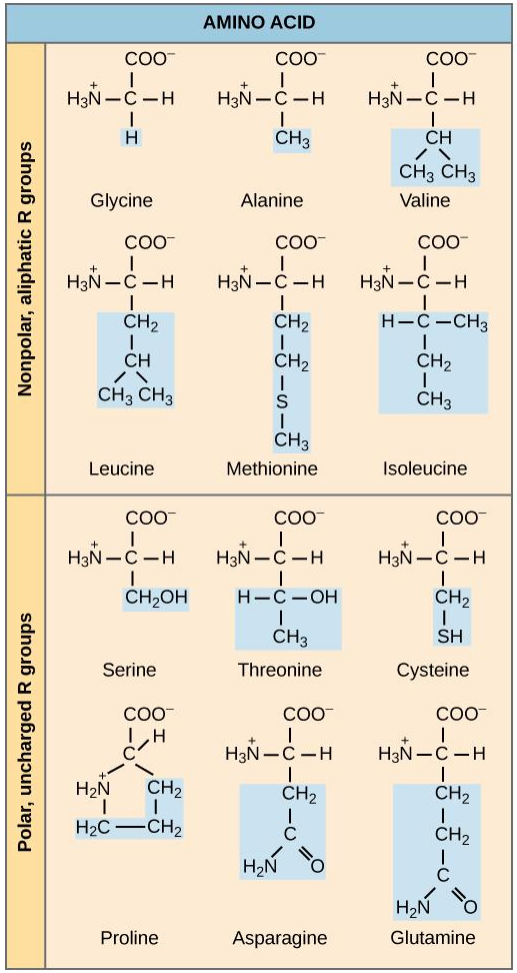
Example: Node Classification

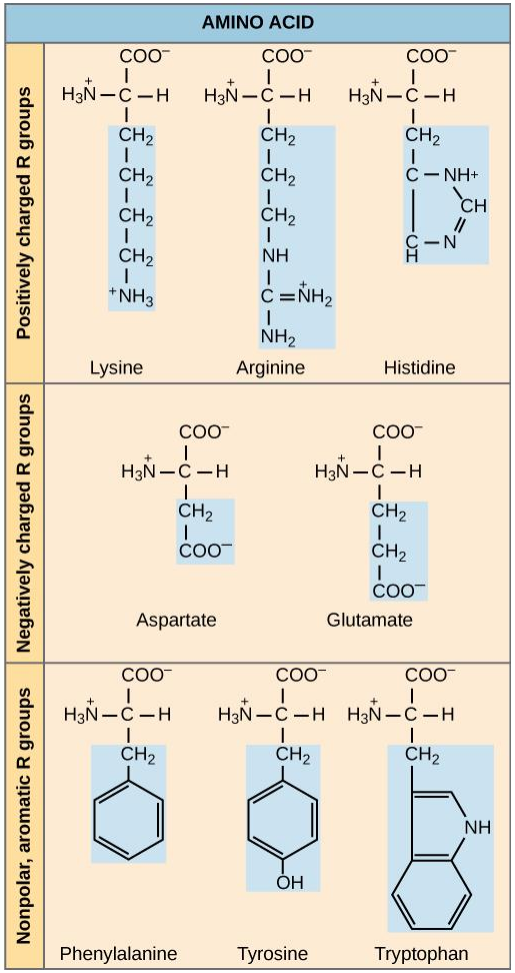
Classifying the function of proteins in the interactome!



Example of “Node-level” ML







Example of “Node-level” ML

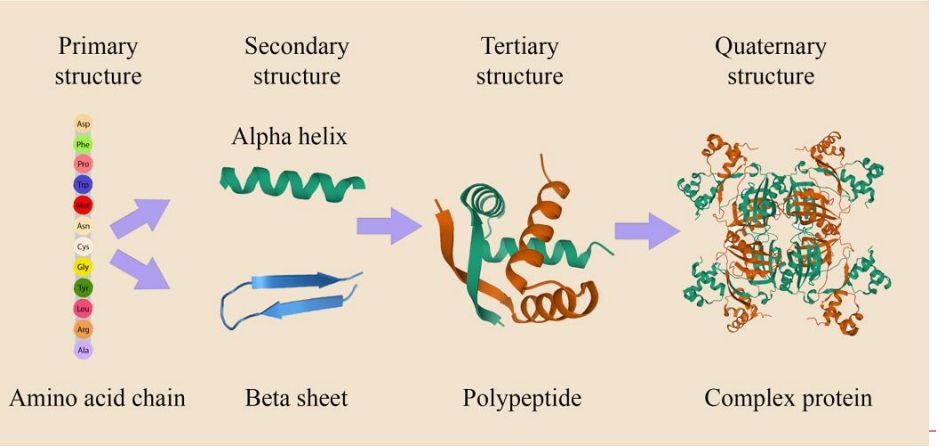
ex) Protein Folding

- protein= sequence of amino acid
- 3d structure
- interact with each other

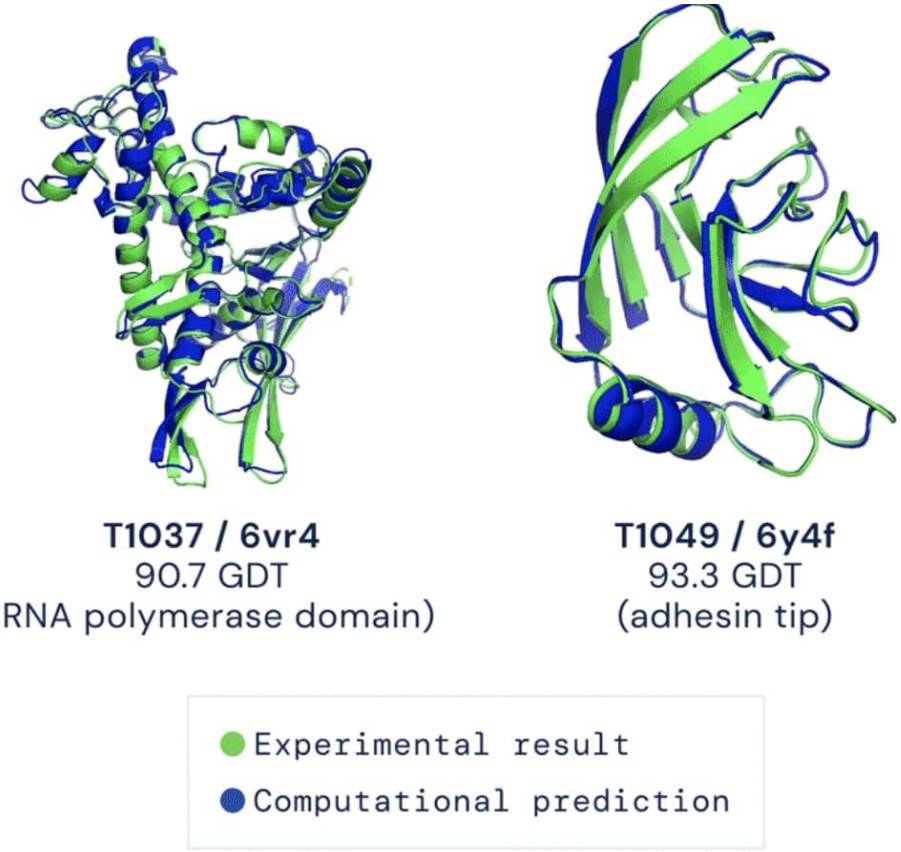
•Goal: **predict 3D structure** based on **amino acid sequence**

•key idea of **AlphaFold**: “spatial graph”

- (1) node: amino acids
- (2) edges: proximity between nodes



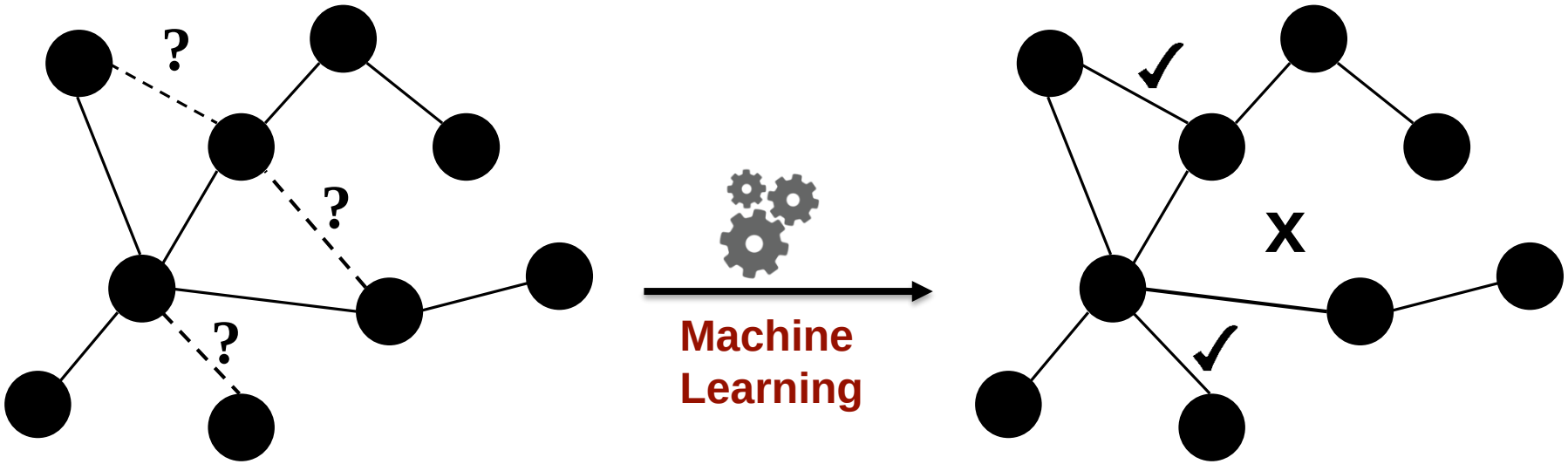
Example of “Node-level” ML



TWO EXAMPLES OF PROTEIN TARGETS IN THE FREE MODELLING CATEGORY. ALPHAFOLD PREDICTS HIGHLY ACCURATE STRUCTURES MEASURED AGAINST EXPERIMENTAL RESULT.

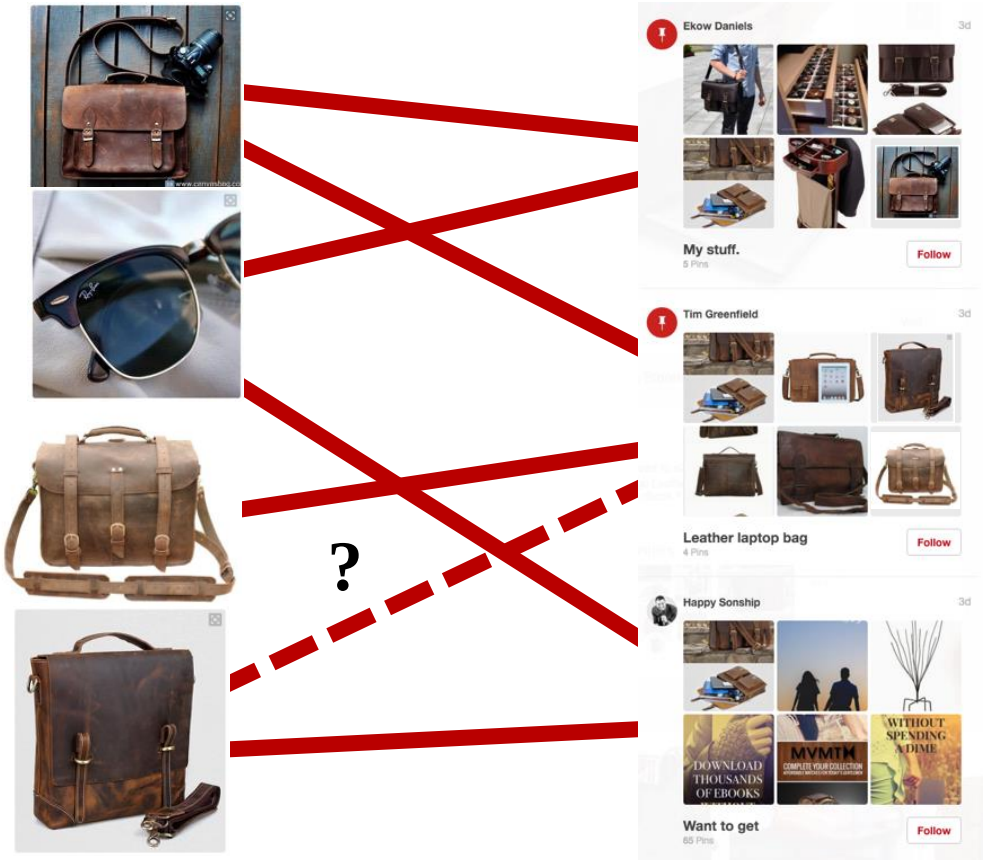
Examples of Edge-Level Tasks

Example: Link Prediction



Example: Link Prediction

Content recommendation is link prediction!



Example of “Edge-level” ML

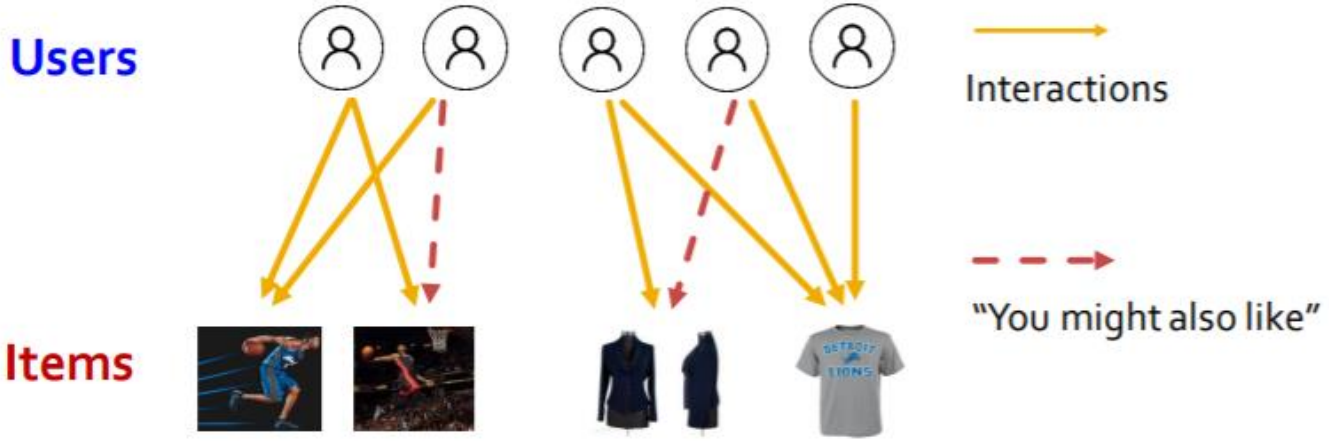
ex) Recommender Systems

•Formulation

- (1) node: user & items
- (2) edge: user & item interaction

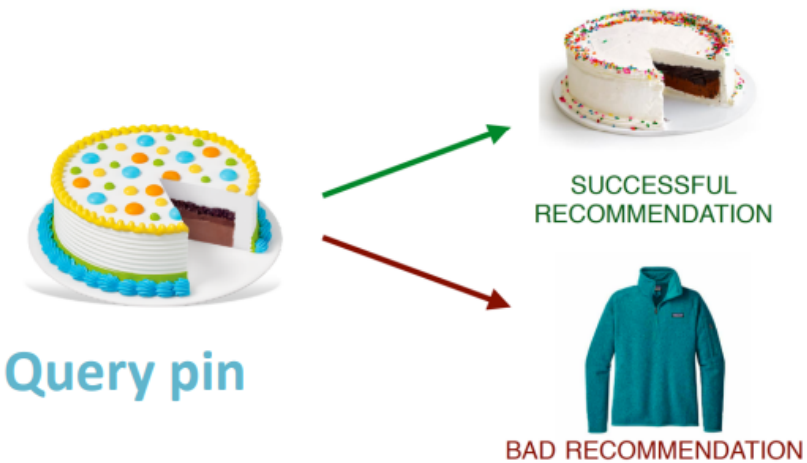
• **Goal: “Recommend item to users”**

• (predict whether 2 nodes are related)



Example of "Edge-level" ML

Task: Recommend related pins to users



Task: Learn node embeddings z_i such that

$$d(z_{cake1}, z_{cake2}) < d(z_{cake1}, z_{sweater})$$

Predict whether two nodes in a graph are related

Example of “Edge-level” ML

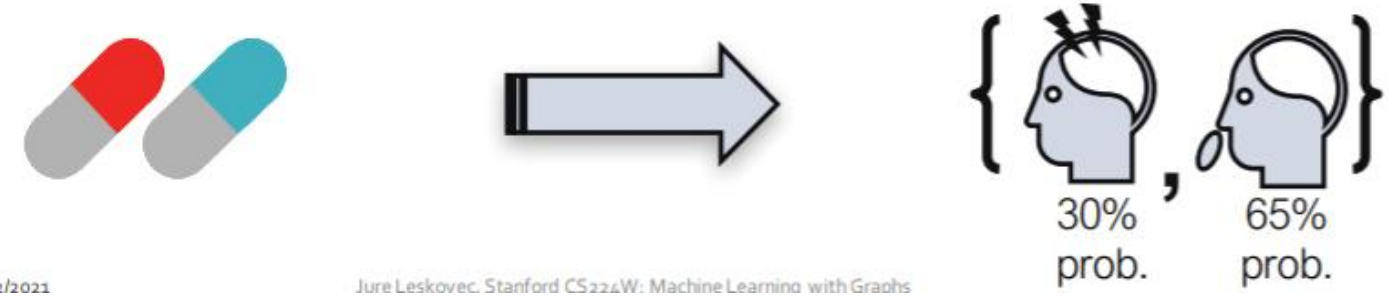
ex) Drug Side Effects

Background: many patients & many drugs

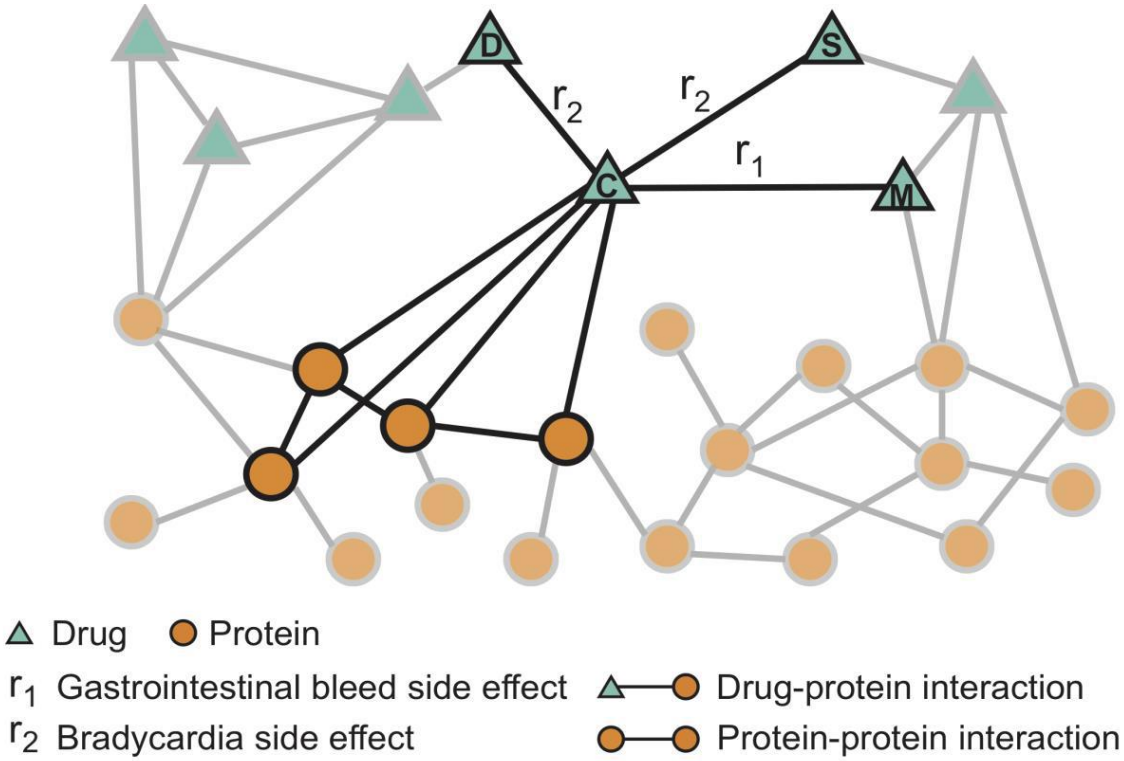
•Goal: predict adverse side effects of “pair of drugs”

• Formulation

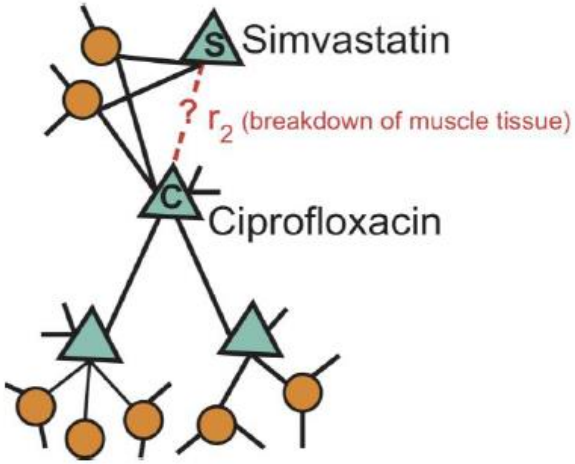
- (1) node: drugs & proteins
- (2) edges: interactions
  - drug-protein interaction
  - protein-protein interaction
  - drug-drug interaction



Example of "Edge-level" ML



Query: How likely will Simvastatin and Ciprofloxacin, when taken together, break down muscle tissue?



Example of A SubGraph Task

Example of “Subgraph-level” ML

- **Nodes:** Road segments
- **Edges:** Connectivity between road segments

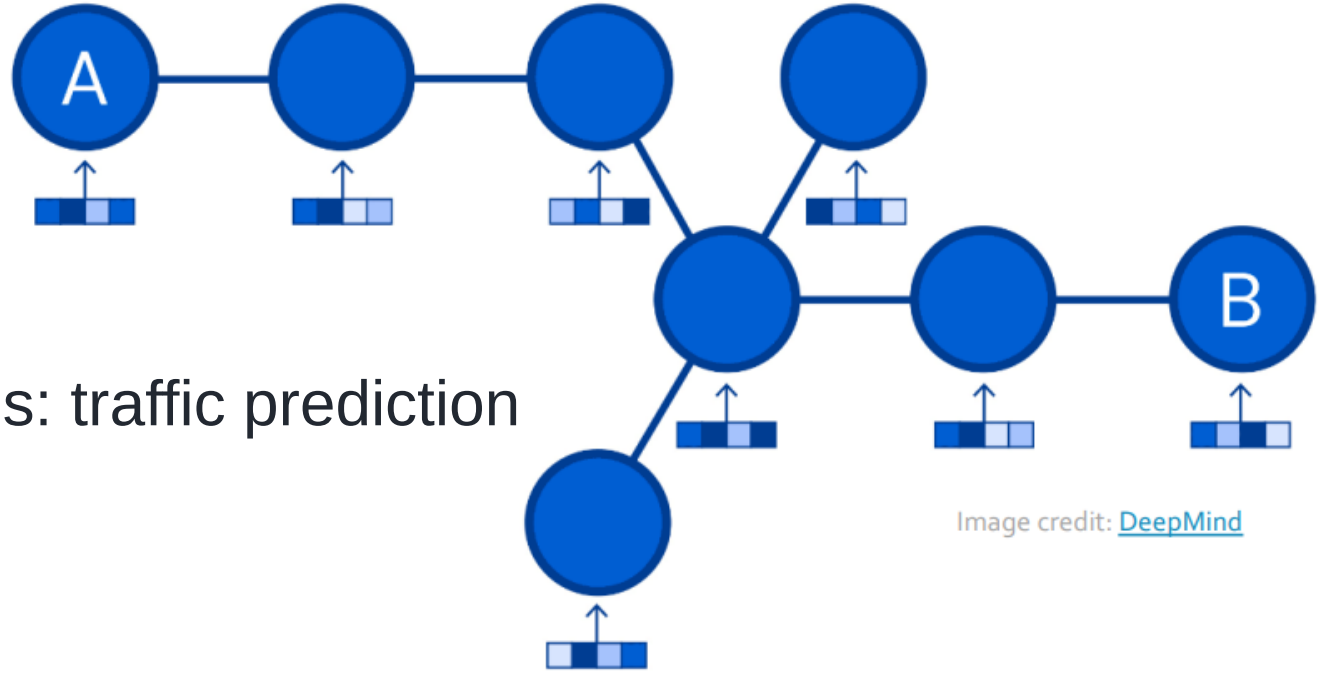
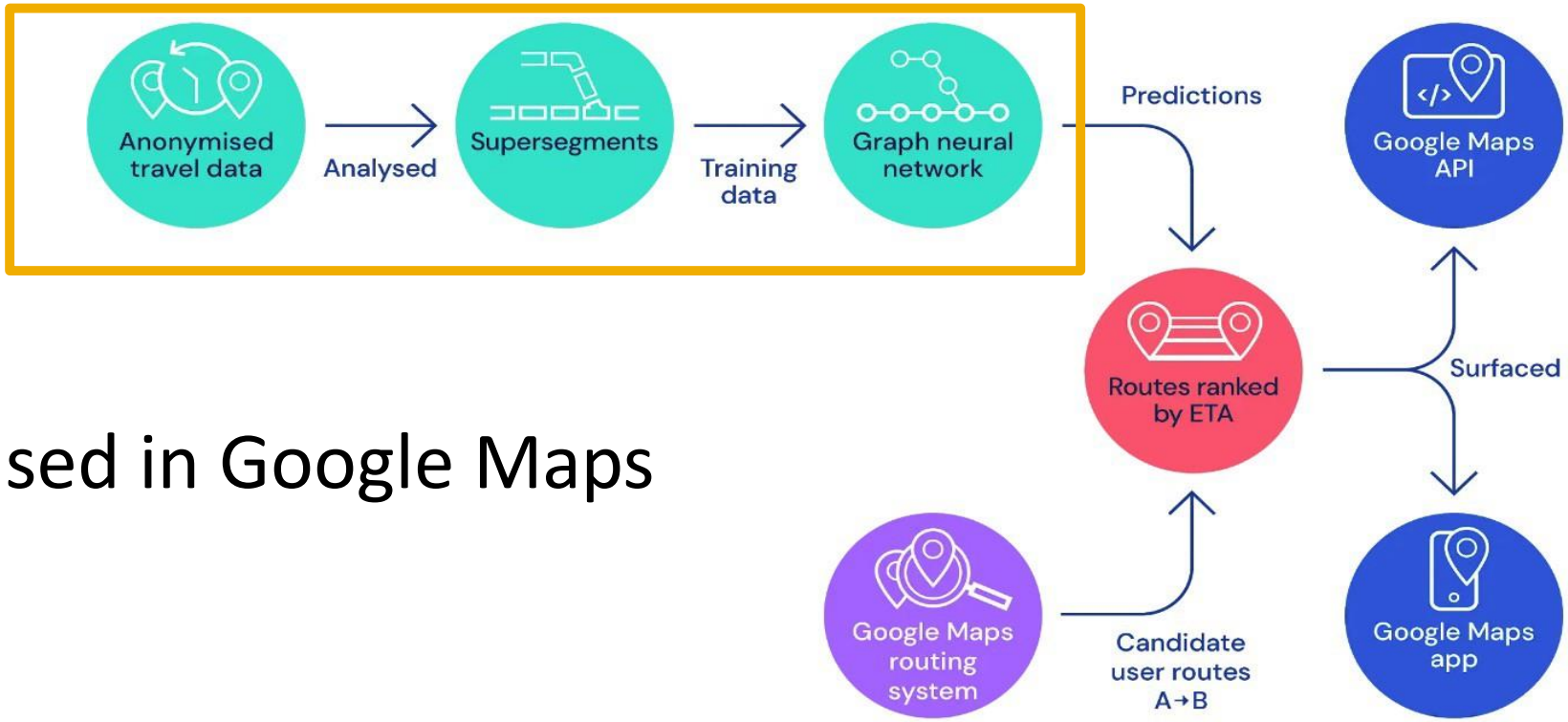


Image credit: DeepMind

Google Maps: traffic prediction with GNN

Traffic Prediction with GNNs

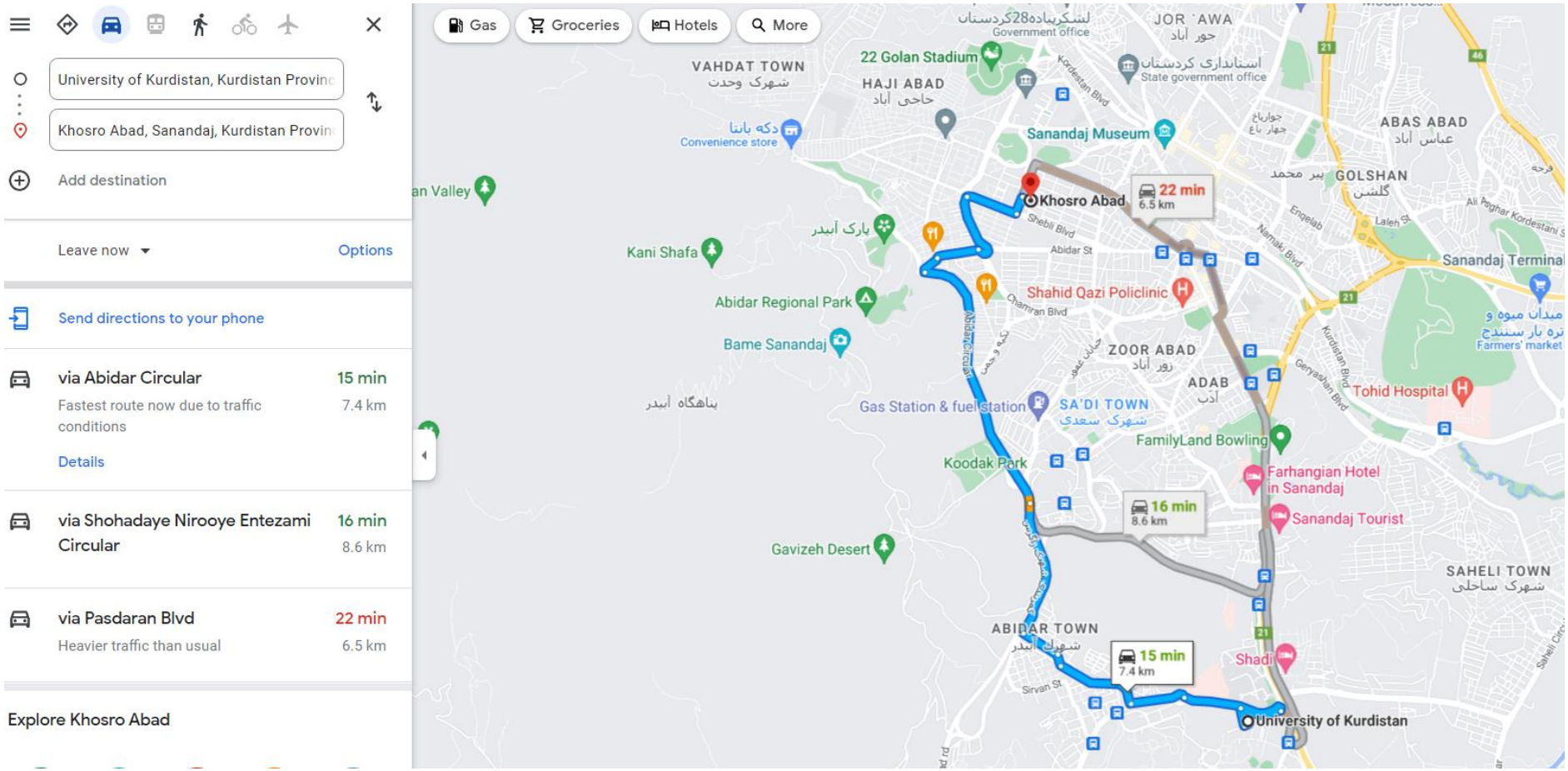
Predicting Time of Arrival with GNNS



Used in Google Maps

THE MODEL ARCHITECTURE FOR DETERMINING OPTIMAL ROUTES AND THEIR TRAVEL TIME.

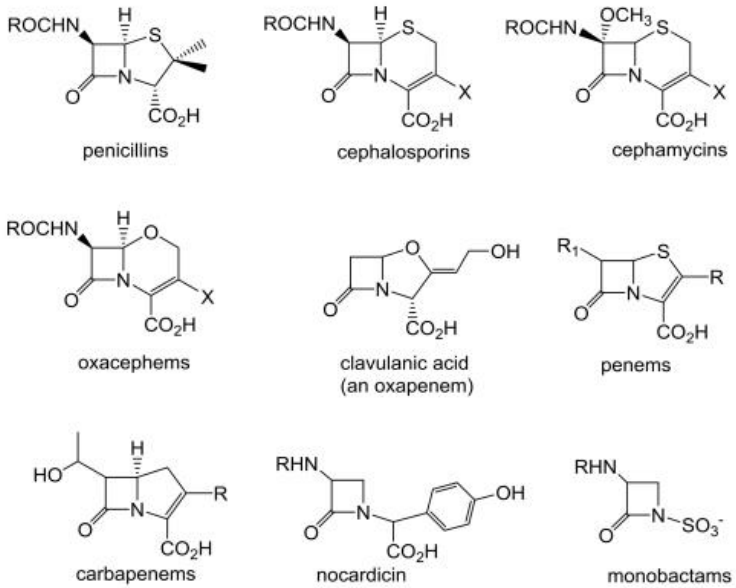
Google Maps : traffic prediction with GNN



Examples of Graph-Level Tasks

Example of “Graph-level” ML

- **Antibiotics are small molecular graphs**
  - **Nodes:** Atoms
  - **Edges:** Chemical bonds



Konaklieva, Monika I. "Molecular targets of β-lactam-based antimicrobials: beyond the usual suspects." Antibiotics 3.2 (2014): 128-142.

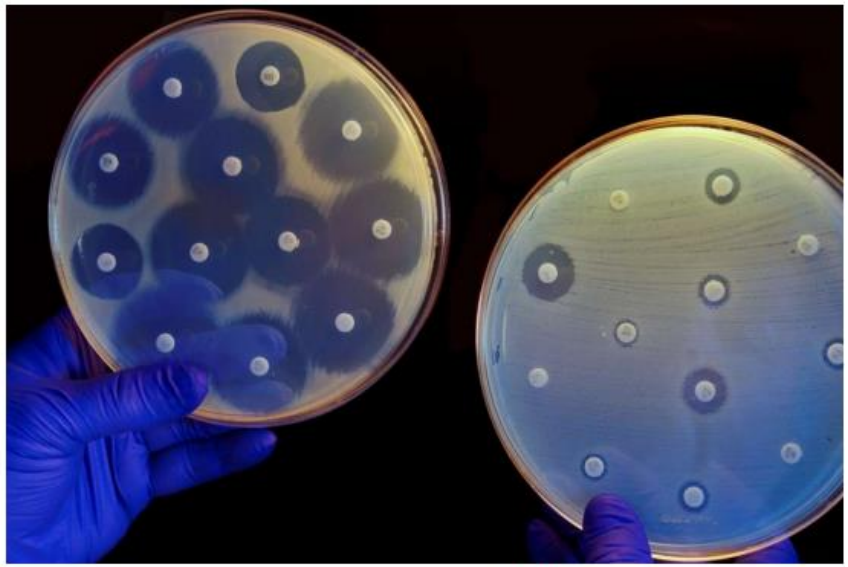
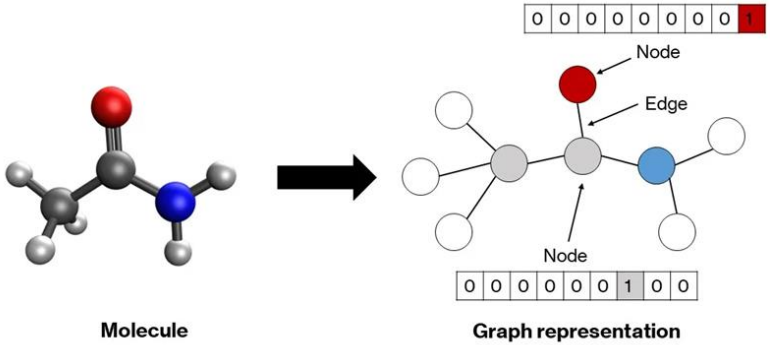


Image credit: CNN

Example of “Graph-level” ML

ex) Drug Discovery Permalink

- Antibiotics = small molecular graphs
- Formulation
  - (1) node: atoms
  - (2) edges: chemical bonds
- (Q) Which molecules should be prioritized?
- ex) graph classification model
  - predict promising molecules among candidates

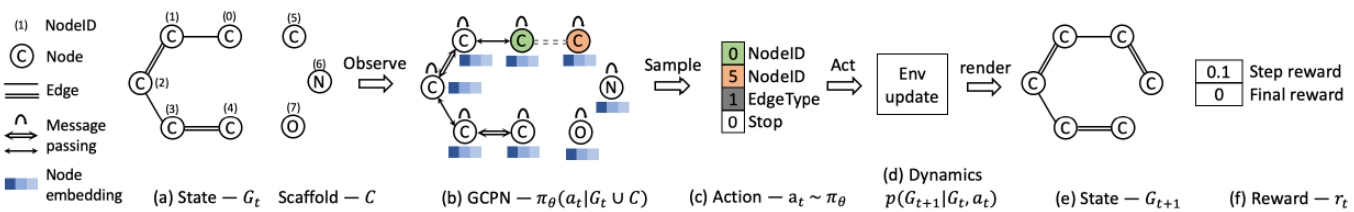


Example of “Graph-level” ML

Generate novel molecules (new structure)

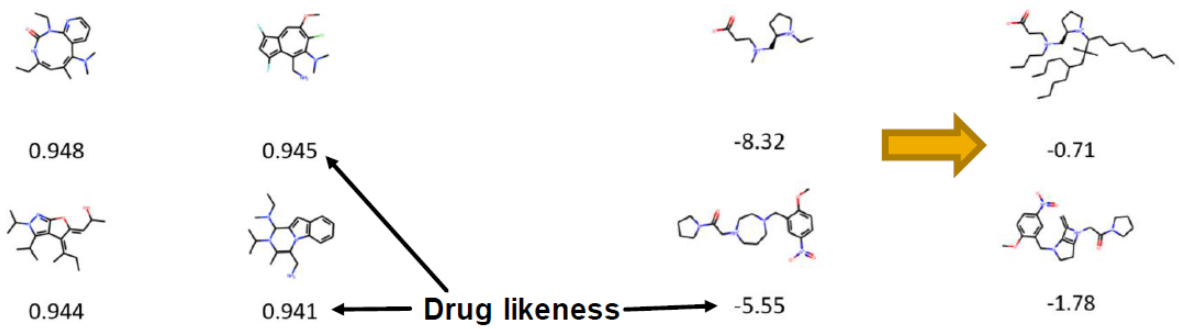
- with “high drug likeness”
- with “desirable properties”

Graph generation: Generating novel molecules



Use case 1: Generate novel molecules with high Drug likeness value

Use case 2: Optimize existing molecules to have desirable properties



Graph Embedding Methods

Shallow embedding

- Matrix factorization-based approaches
- **Random Walk-Based (Deepwalk- Node2vec)**

Deep embedding

- **Graph Neural Networks (GCN- GAT- GraphSAGE)**
- Autoencoder-Based Methods
- Temporal/Dynamic Graph Embeddings (TGAT)
- Heterogeneous Graph Embeddings (HAN, Metapath2vec)
- Graph Transformers (Graphormer, GTN)

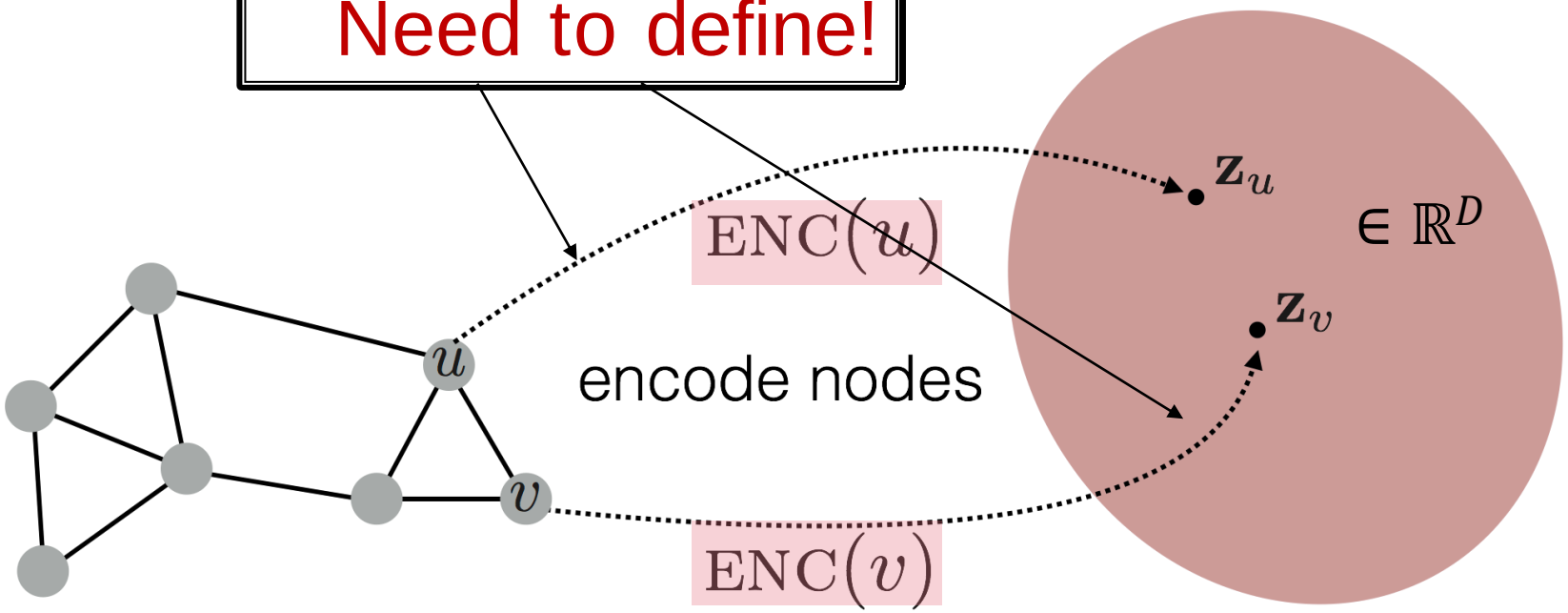
Shallow Embedding

Embedding nodes

Goal: similarity(u, v) $\approx$ $z_u{}^T \cdot z_v$ Similarity of the embedding

in the original network

Need to define!



original network **embedding space**

Learning node embeddings

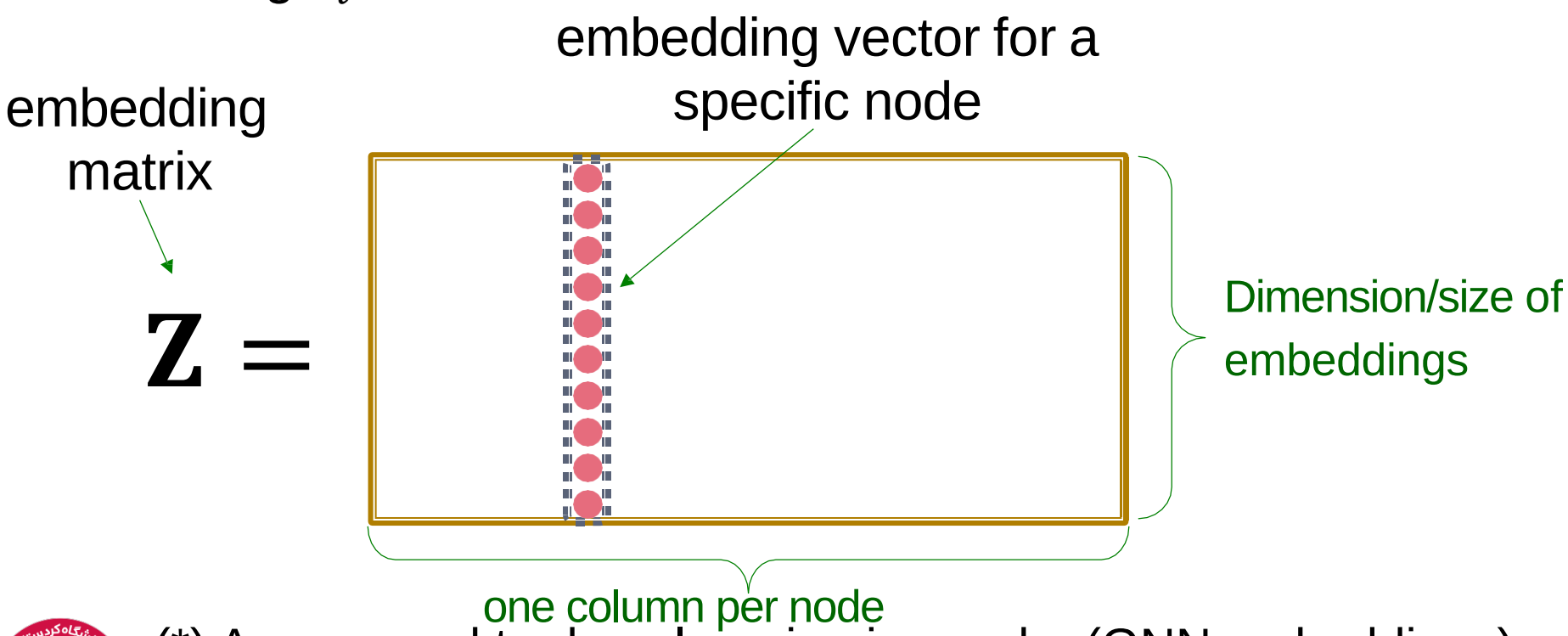
1. Define an encoder ENC that maps nodes to low dimensional spaces
2. Define *a node similarity function* (i.e., a measure of similarity in the original network).
3. Decoder $\mathbf{DEC}$ maps from embeddings to the similarity score
4. Optimize the parameters of the encoder so that we minimize *a loss function* L that looks (roughly) like:

$$L = \sum_{u,v \in V} (similarity(u,v) - z_u^T \cdot z_v)^2$$

Shallow embeddings(*)

Each node is assigned a single d-dimensional vector

Learn $|V| \times d$ embedding matrix Z: each column i is the embedding z_i of node i

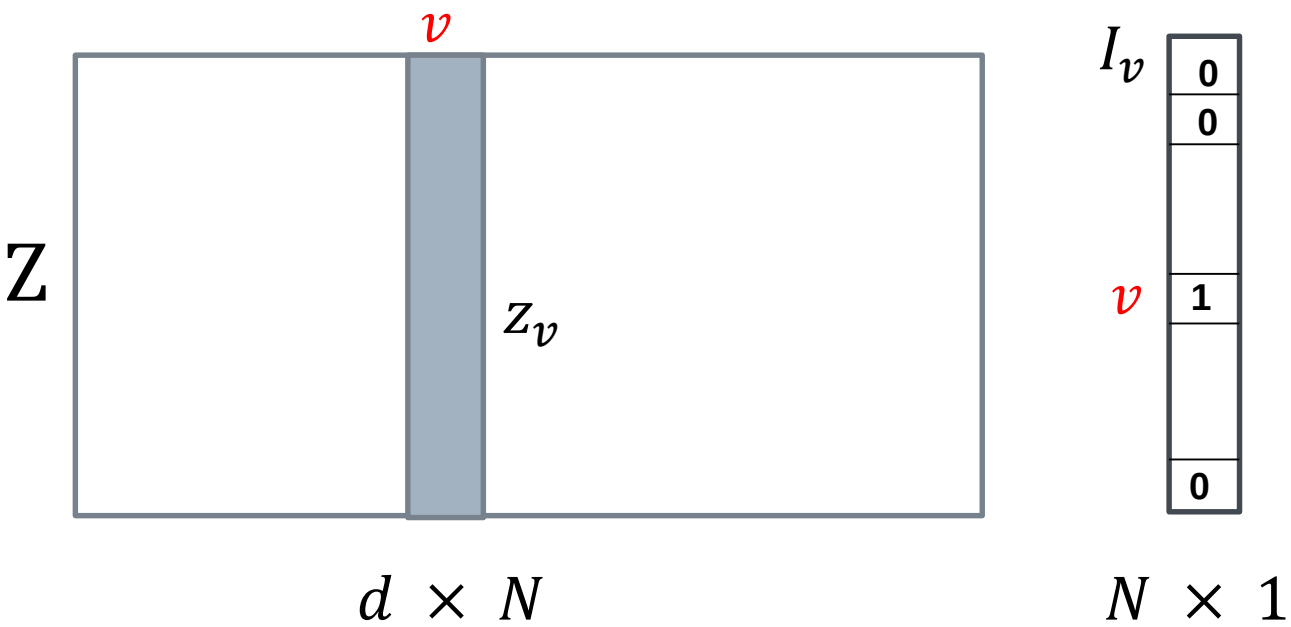


(*) As opposed to deep learning in graphs (GNN embeddings)

Shallow embeddings

Encoder is just an embedding **lookup**

$$ENC(v) = Z_v = Z\, I_v$$



One-hot or indicator vector, all 0s but position v

Framework Summary

Encoder + Decoder Framework

- Shallow encoder: Embedding lookup
- Parameters to optimize: $\mathbf{Z}$ which contains node embeddings for all nodes $u \in V$
- We will cover deep encoders in the GNNs

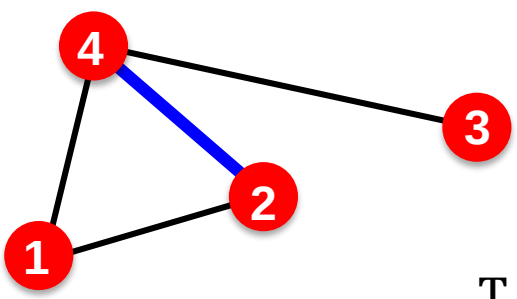
- Decoder: based on **node similarity**.
- Objective: maximize $z_u^T \cdot z_v$ for node pairs (u, v) that are **similar**

How to define node similarity

- Key choice of methods is how they define node similarity.
- Should two nodes have a similar embedding if they…
  - are linked?
  - share neighbors?
  - have similar “structural roles”?

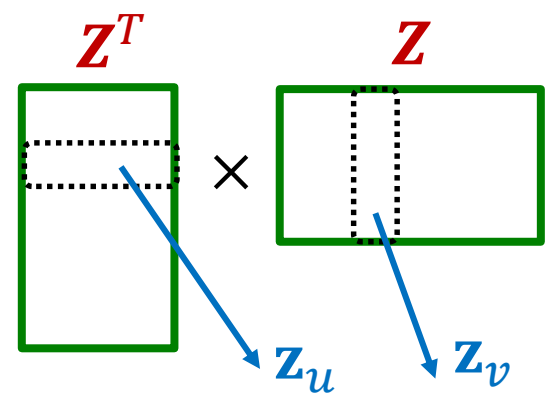
Adjacency Matrix

- Simplest node similarity: Nodes u, v are similar if they are connected by an edge
- This means: $z_v^T z_u = A_{u,v}$
  which is the (u, v) entry of the graph adjacency matrix A
- Therefore, $Z^T Z = A$



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

$$z_2^T z_4 \approx \mathbf{1}$$



Adjacency-based approach

- The embedding dimension d (number of rows in $\boldsymbol{Z}$) is much smaller than number of nodes n. ($d << n$)
- Inner product decoder with node similarity defined by edge connectivity is equivalent to matrix factorization of A.
- Exact factorization $A = \boldsymbol{Z}^T \boldsymbol{Z}$ is generally not possible
- Matrix decomposition (for example, SVD decomposition)
  1. Scalability issues
  2. Produced matrices that are very dense

Adjacency-based approach

- However, we can learn $\boldsymbol{Z}$ approximately
- **Objective**: $\min_{\mathbf{Z}} \| \mathrm{A} - \boldsymbol{Z}^T \boldsymbol{Z} \|^2$
  - We optimize $\boldsymbol{Z}$ such that it minimizes the L2 norm (Frobenius norm) of $\mathbf{A} - \boldsymbol{Z}^T \boldsymbol{Z}$
  - We used softmax instead of L2. But the goal to approximate $\mathbf{A}$ with $\boldsymbol{Z}^T \boldsymbol{Z}$ is the same.

How: stochastic gradient descent

Adjacency-based approach

The loss that what we want to minimize

embedding similarity

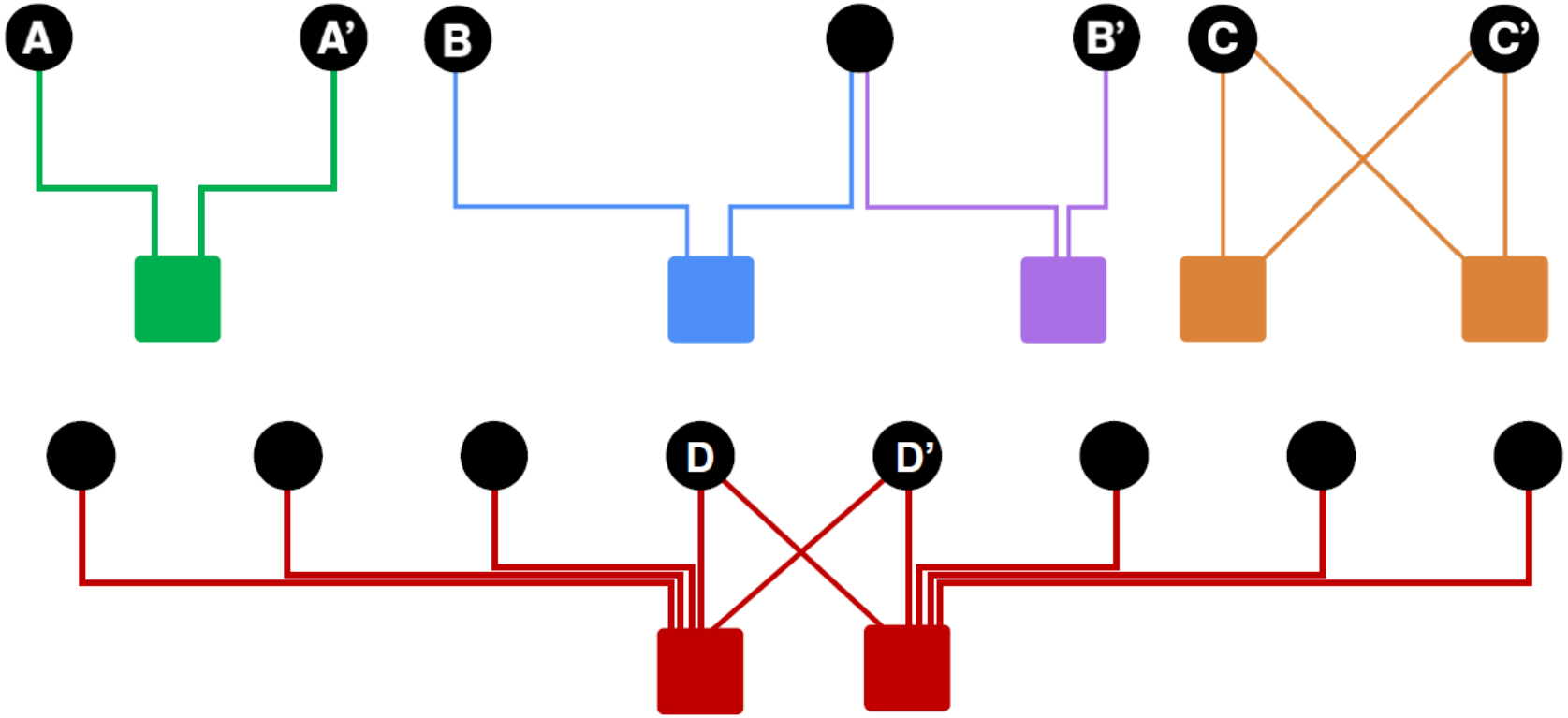
$$L = \sum_{u,v \in V \times V} ||A_{u,v} - Z_u^T \cdot Z_v||^2$$

(possibly weighted) adjacency matrix for the graph

RANDOM -WALK BASED EMBEDDINGS

Node Similarity Measure

- Which is more related A,A', B,B' or C,C'?

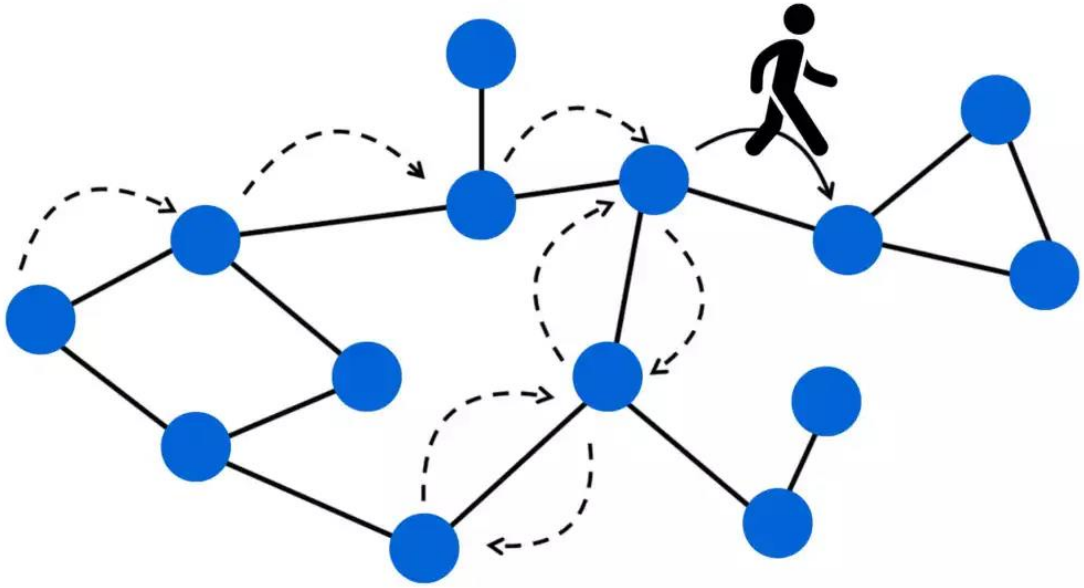


Random Walk Strategy

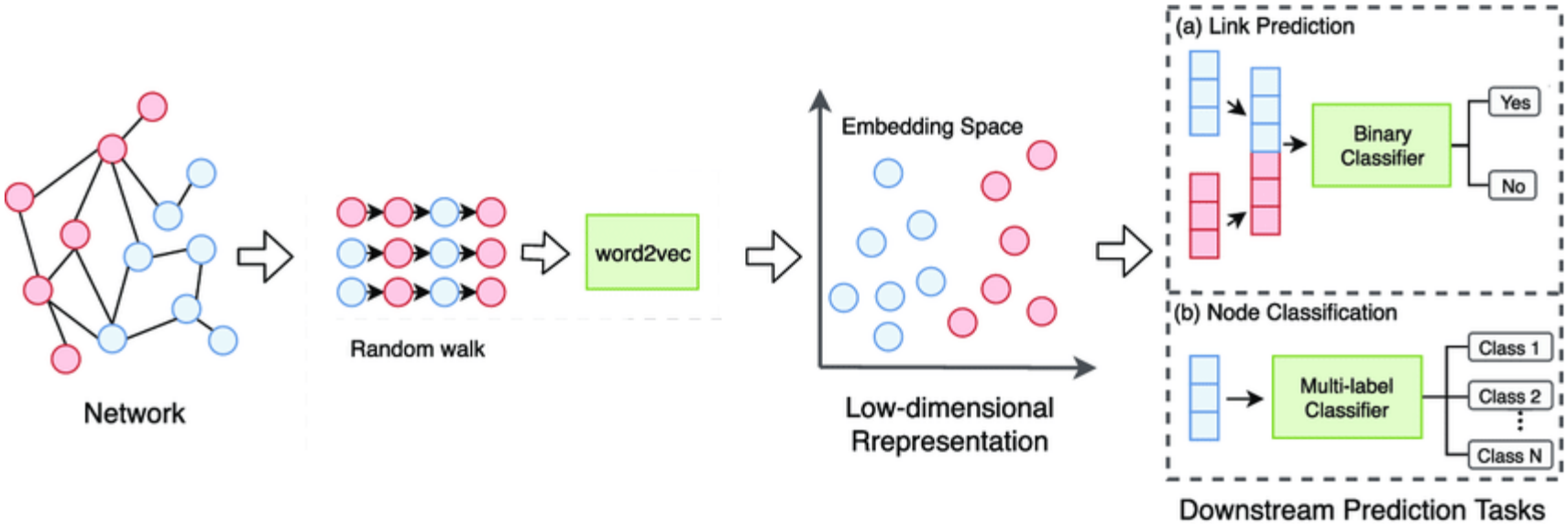
How?

Words = Nodes

Sentences = Paths, Random walks

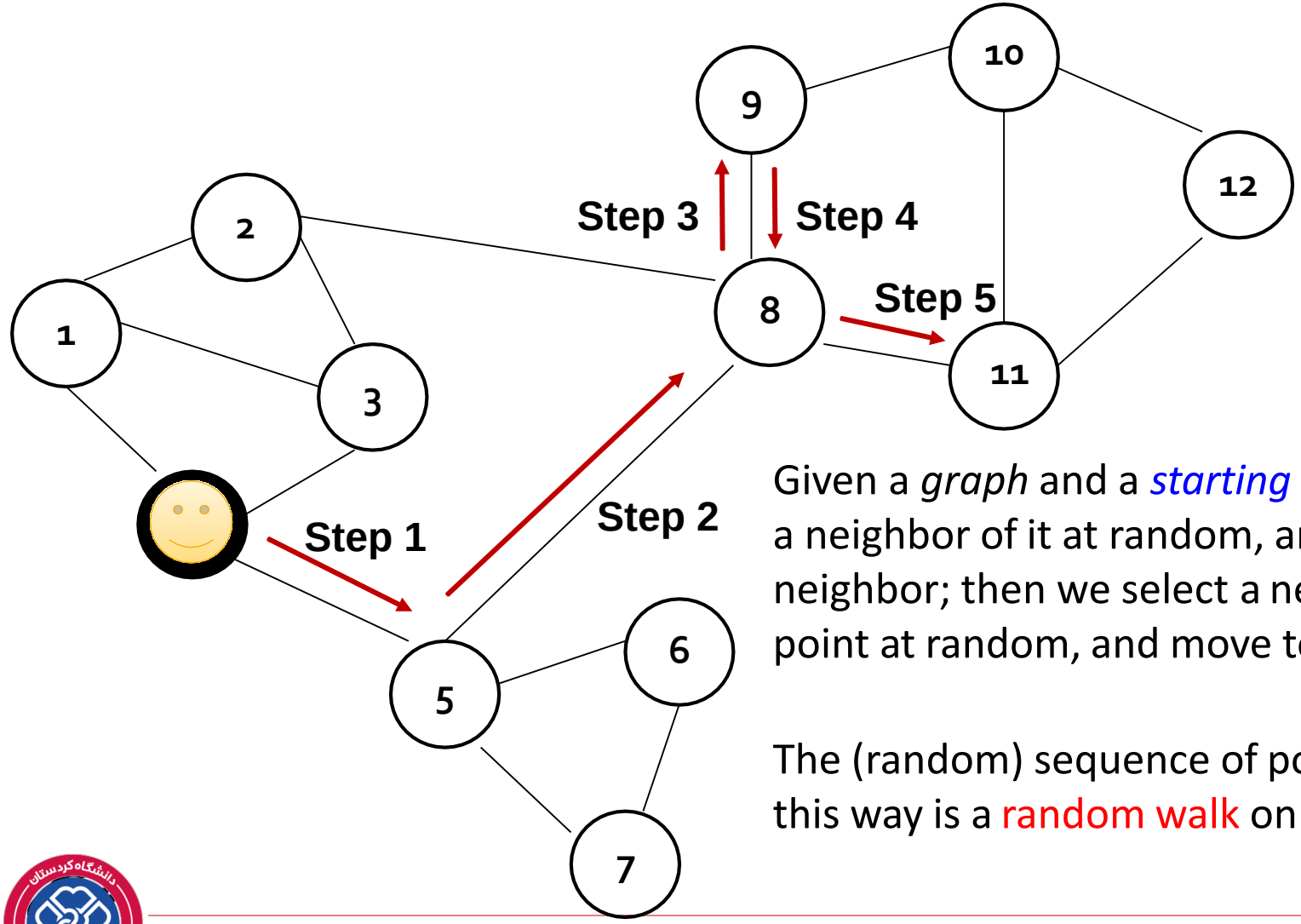


Random Walk Strategy



To generate node representations by simulating random walks on a graph, capturing structural and relational patterns in a low-dimensional space.

Random Walk



Given a *graph* and a *starting point*, we select a neighbor of it at random, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc.

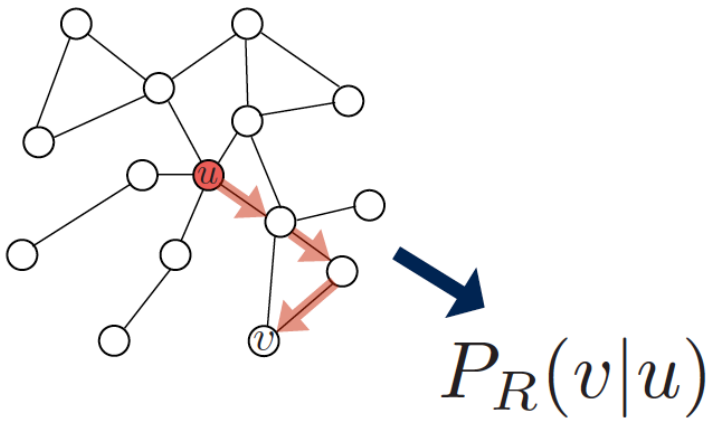
The (random) sequence of points visited this way is a random walk on the graph.

Random-walk embeddings

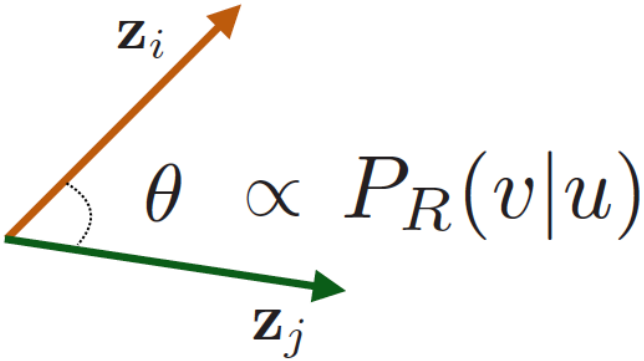
$$z_i \cdot z_j \approx$$ probability that *i* and *j* co-occur on a random walk over the network

Random-walk Embeddings

1. Estimate probability of visiting node v on a random walk starting from node u using some random walk strategy R.



2. Optimize embeddings to encode these random walk statistics.



Why Random Walks?

1. **Expressivity**: Flexible stochastic definition of node similarity that incorporates both local and higher-order neighborhood information. *Idea:* if random walk starting from node u visits v with high probability, u and v are similar (high-order multi-hop information)

2. **Efficiency**: Do not need to consider all node pairs when training; only need to consider pairs that co-occur on random walks.

Unsupervised Feature Learning

- Intuition: Find embedding of nodes in d-dimensional space that preserves similarity
- Idea: Learn node embedding such that nearby nodes are close together in the network
- Given a node u, how do we define nearby nodes?
  - $N_R(u)$: neighbourhood of u obtained by some random walk strategy R

Random Walk Optimization

1. Run **short fixed-length random walks** starting from each node u in the graph using some random walk strategy R.
2. For each node u collect $N_R(u)$, the multiset* of nodes visited on random walks starting from u.
3. Optimize embeddings according to: Given node u, predict its neighbors $N_{\mathrm{R}}(u)$.

$$\arg\max_{z} \sum_{u \in V} \log \mathrm{P}(N_{\mathrm{R}}(u) \mid \mathbf{z}_u)$$

$\Rightarrow$ Maximum likelihood objective

***$N_R(u)$ can have repeat elements since nodes can be visited multiple times on random walks**

Random Walk Optimization

Equivalently,

$$\arg\min_{z} \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

Intuition: Optimize embeddings z_u to minimize the negative log-likelihood of random walk neighborhoods $N(u)$.

Parameterize $P(v|z_u)$ using softmax:

$$P(v|\mathbf{z}_u) = \frac{\exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_n)}$$

Why softmax?
We want node v to be most similar to node u (out of all nodes n).
Intuition: $\sum_i \exp(x_i) \approx \max_i \exp(x_i)$

Random Walk Optimization

Putting it all together:

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\left(\frac{\exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_n)}\right)$$

sum over all nodes u

sum over nodes v seen on random walks starting from u

predicted probability of u and v co-occuring on random walk

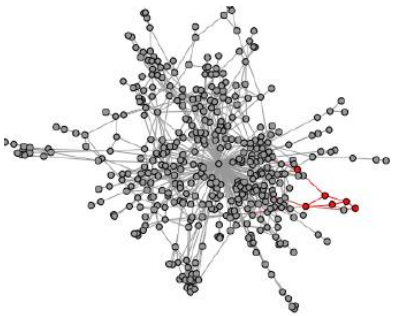
Optimizing random walk embeddings = Finding embeddings z_u that minimize Loss

How should we randomly walk?

- **DeepWalk** just runs fixed-length, unbiased random walks starting from each node
- **Node2vec**: biased random walks that can trade-off between local and global views of the network

DeepWalk

Short random walks = sentences

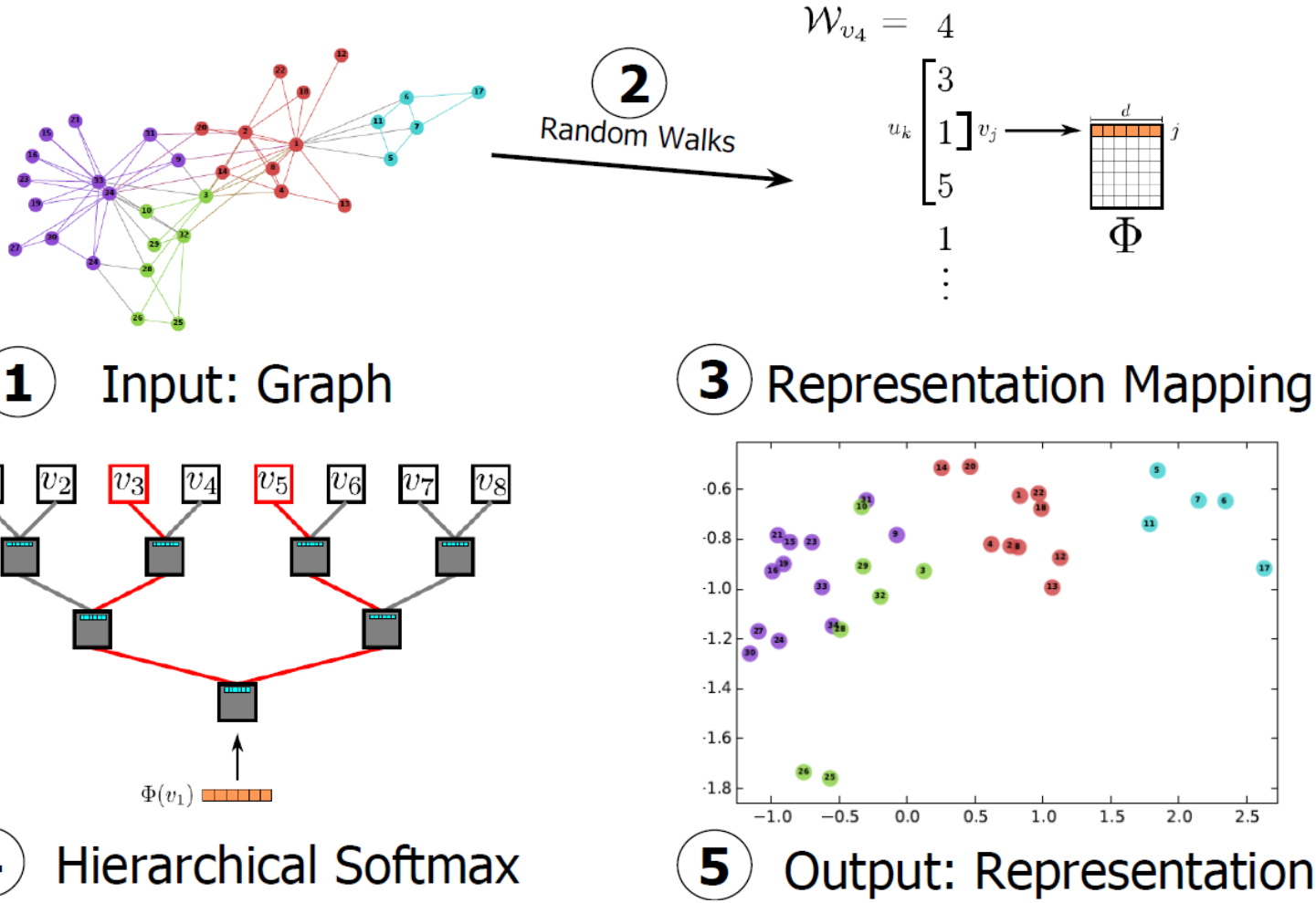


Scale Free Graph

$$
\begin{array}{cccccc}
v_{71} \rightarrow & v_{24} \rightarrow & v_{5} \rightarrow & v_{1} \rightarrow & v_{17} \rightarrow & v_{80} \rightarrow \\
v_{92} \rightarrow & v_{2} \rightarrow & v_{3} \rightarrow & v_{1} \rightarrow & v_{12} \rightarrow & v_{73} \rightarrow \\
v_{37} \rightarrow & v_{34} \rightarrow & v_{9} \rightarrow & v_{1} \rightarrow & v_{10} \rightarrow & v_{94} \rightarrow \\
v_{73} \rightarrow & v_{64} \rightarrow & v_{5} \rightarrow & v_{1} \rightarrow & v_{12} \rightarrow & v_{1} \rightarrow \\
v_{75} \rightarrow & v_{14} \rightarrow & v_{6} \rightarrow & v_{1} \rightarrow & v_{13} \rightarrow & v_{61} \rightarrow
\end{array}
$$

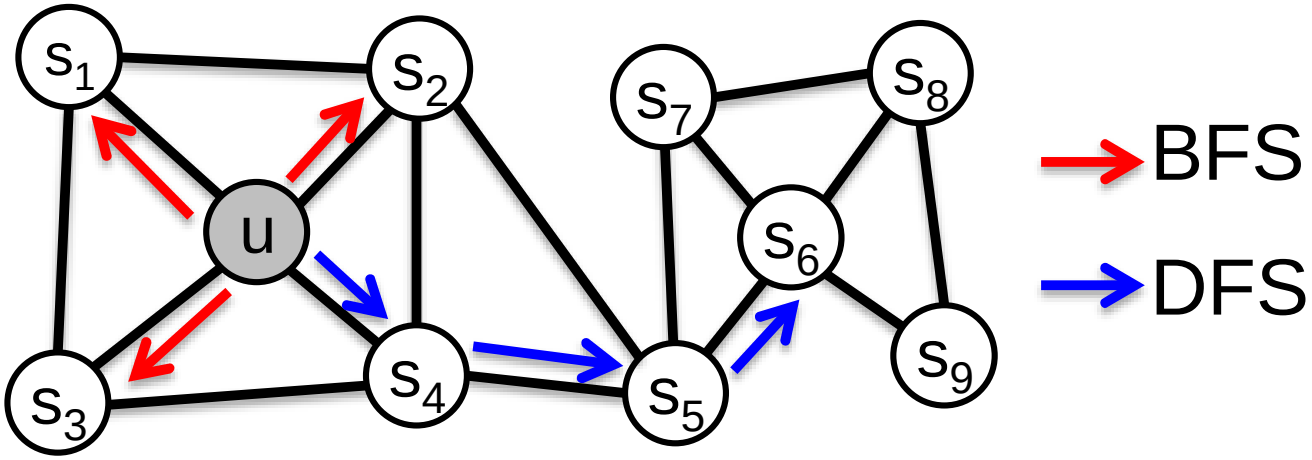
Short truncated random walks are sentences in an artificial language

DeepWalk


2
Random Walks
$\mathcal{W}_{v_4} = 4$
3
1
5
1
u_k
v_j
d
j
Φ
1 Input: Graph
3 Representation Mapping
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
$\Phi(v_1)$
4 Hierarchical Softmax
5 Output: Representation

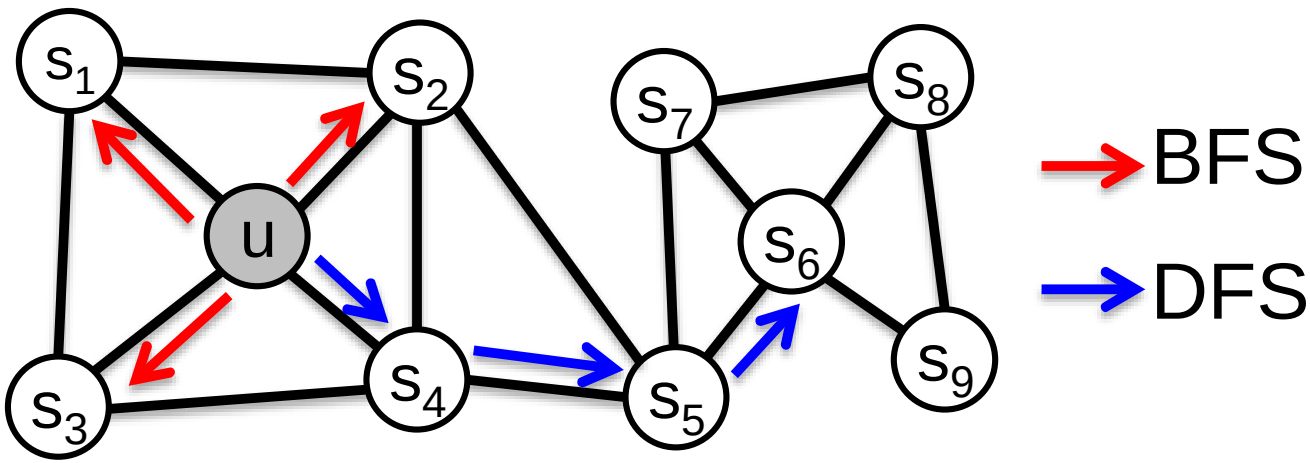
Node2vec: Biased Walks

Idea: use flexible, biased random walks that can trade off between **local** and **global** views of the network (Grover and Leskovec, 2016).



Node2vec: Biased Walks

Two classic strategies to define a neighborhood $N_R(u)$ of a given node u:

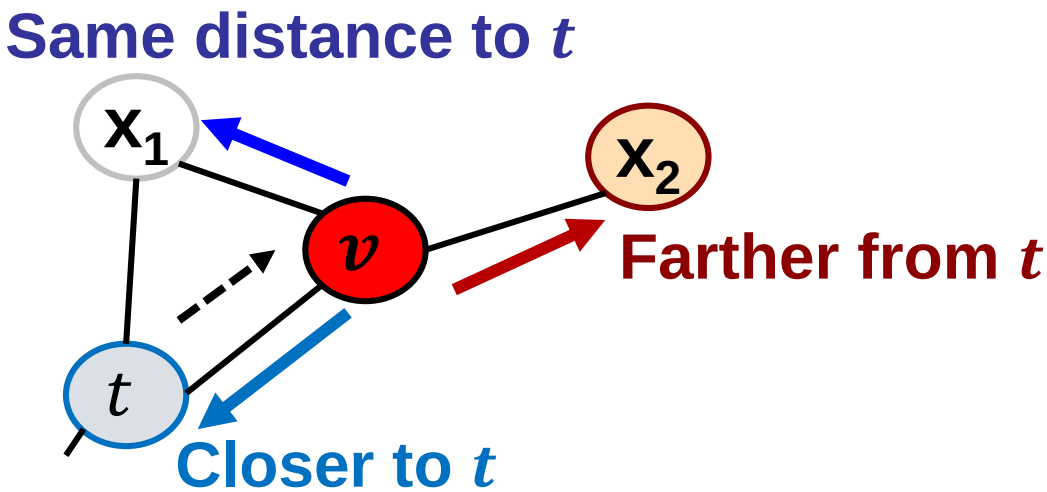


$N_{BFS}(u) = \{ s_1, s_2, s_3 \}$ Local microscopic view

$N_{DFS}(u) = \{ s_4, s_5, s_6 \}$ Global macroscopic view

Biased 2nd Order Random Walks

Walker from t, traversed (t, v) and is now in v, where to go next?



How much far away from t? Only three possible choices:

- Farther distance (distance =2)
- Same distance (distance = 1)
- Back to t (distance = 0)

Interpolating BFS and DFS

Biased random walk R that given a node u generates neighborhood $N_R(u)$

- Two parameters:
  - Return parameter p:
    - Return to the previous node
  - In-out parameter q:
    - Moving outwards (DFS) vs. inwards (BFS)
    - Intuitively, q is the "ratio" of BFS vs. DFS
- Specify how a single step of biased random walk is performed
  - Random walk is then just a sequence of these steps.

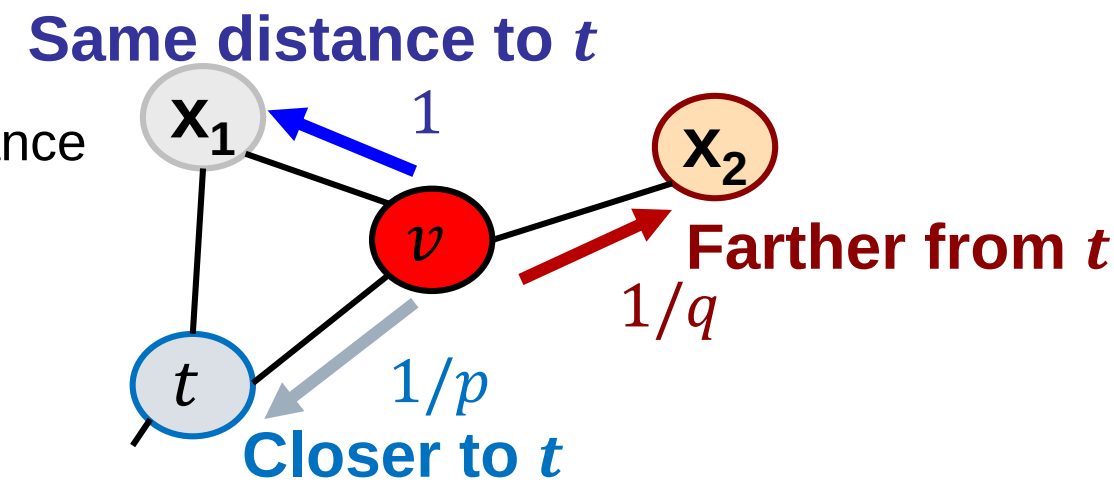
One step of the biased random walk

At v from t, where to go next?

Define the random walk by specifying the walk transition probabilities on edges adjacent to the current node v:

- 1 to node with same distance
- 1/q node further apart
- 1/p back to t

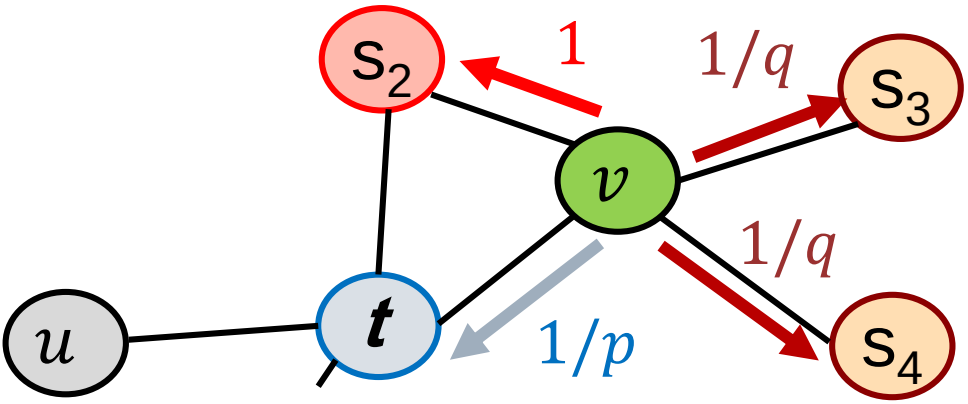
(unnormalized probabilities)



BFS-like walk: Low value of p
DFS-like walk: Low value of q

One step of the biased random walk

At v from S_1



$v \rightarrow$

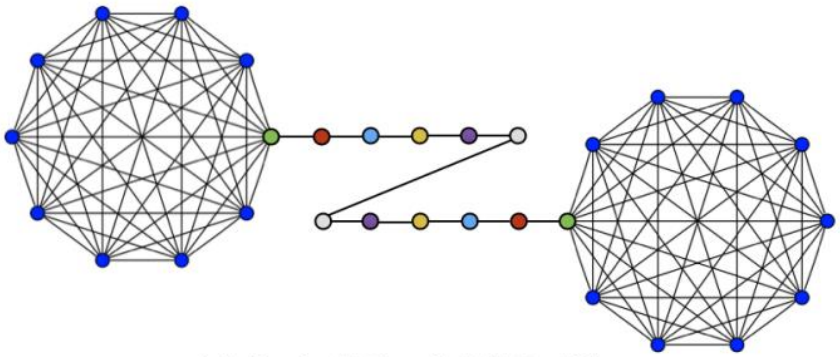
Target	Prob.	Dist. (S_i, t)
t	$1/p$	0
s_2	1	1
s_3	$1/q$	2
s_4	$1/q$	2

Unnormalized transition prob. segmented based on distance from t

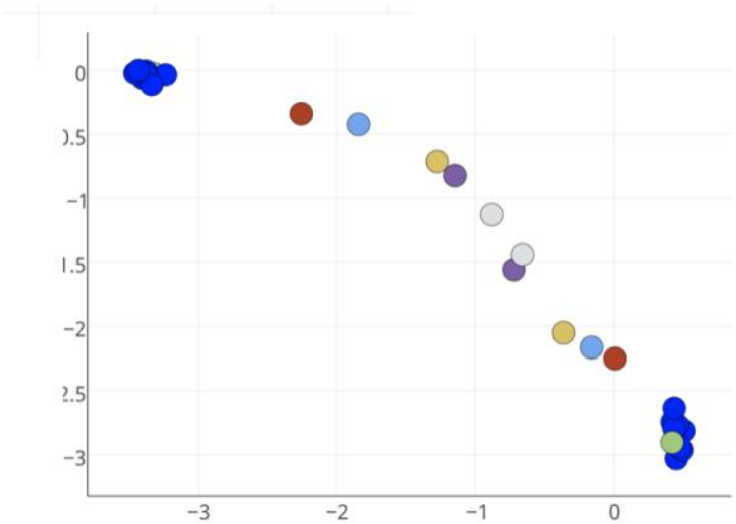
$N_R(v)$ are the nodes visited by the biased walk

Node2vec limitation

node2vec tend to fail in structural equivalence tasks.



(a) Barbell Graph B(10, 10)

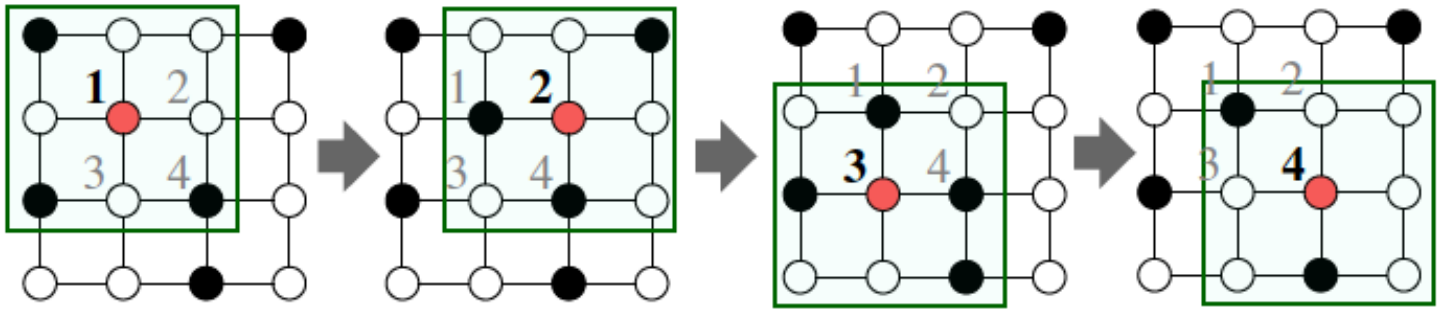


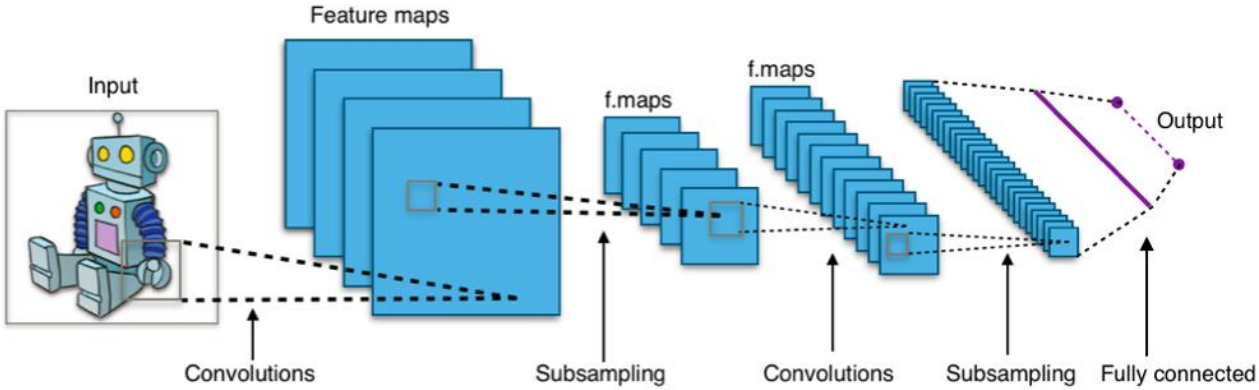
(d) *node2vec*

GRAPH NEURAL NEWTORKS

Idea: Convolutional Networks

CNN on an image:





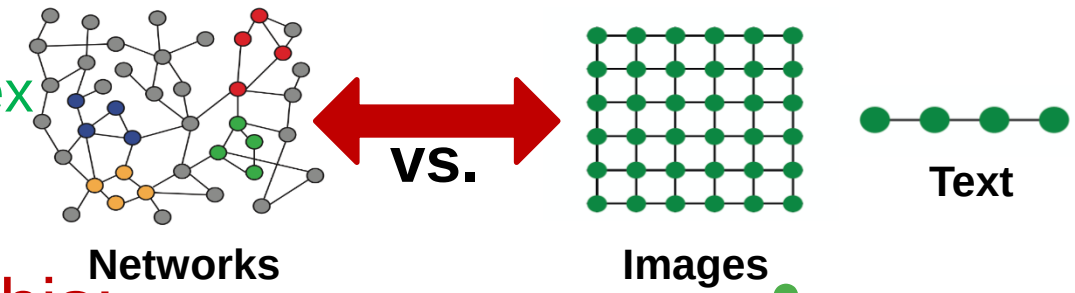
Can we generalize convolutions beyond simple lattices?

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

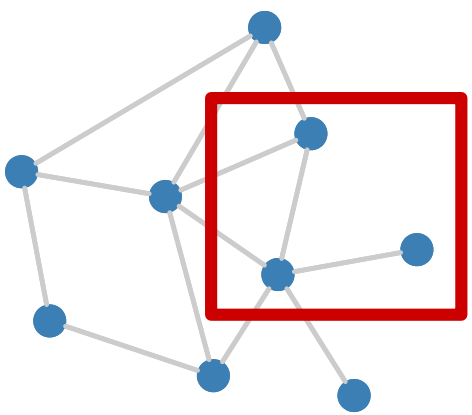
Why is it hard?

Graphs are far more complex!

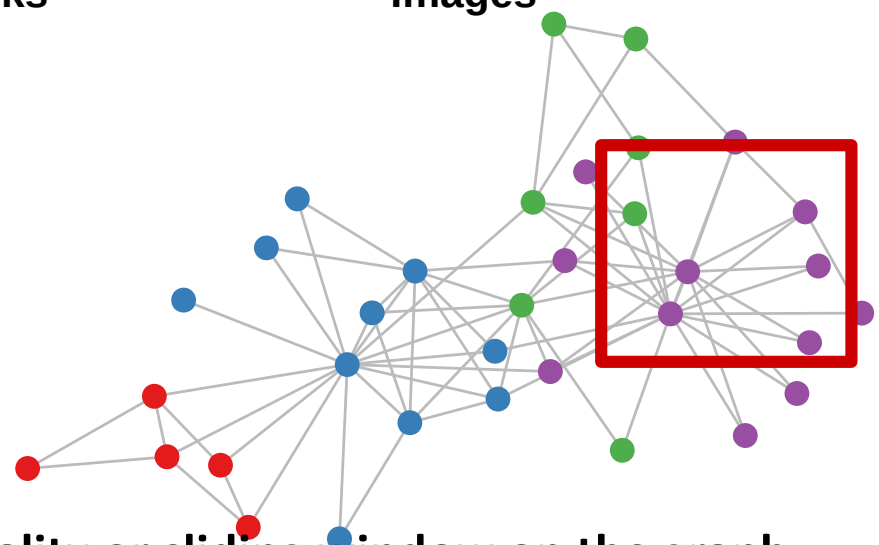
arbitrary size and complex topological structure



Graphs look like this:



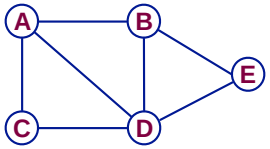
or this:



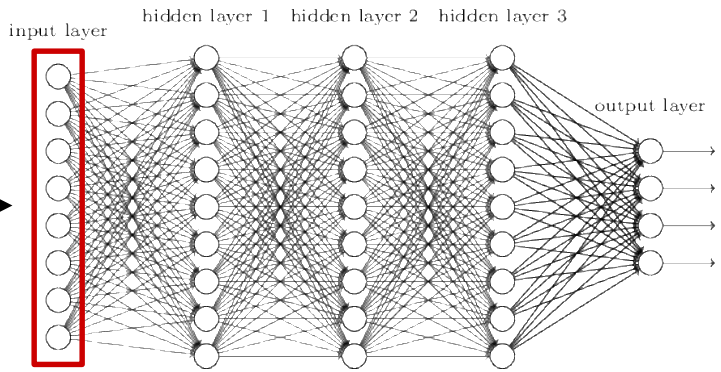
- **No fixed notion of (spatial) locality or sliding window on the graph**
- **No fixed node ordering or reference point**
- **Often dynamic and have multimodal features**

A Naïve Approach

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



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



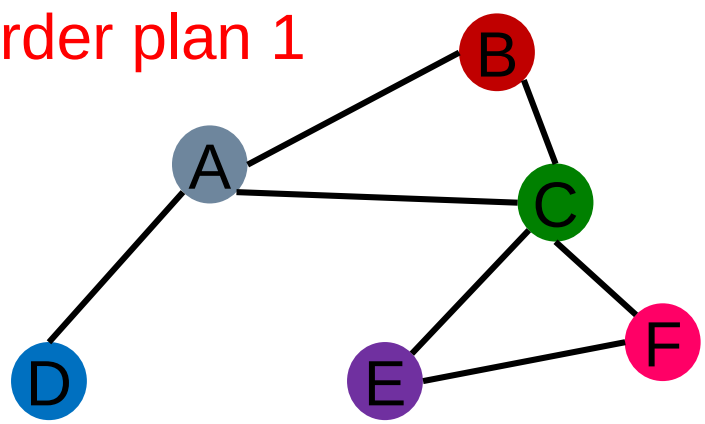
?

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

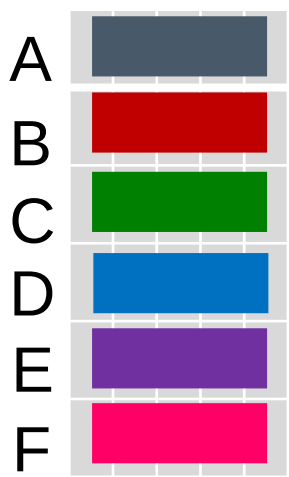
Permutation Invariance

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

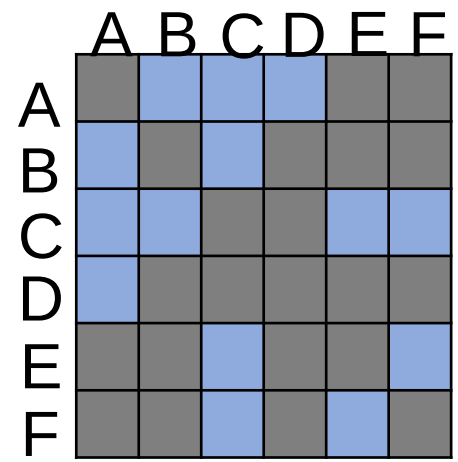
Order plan 1



Node features X_1



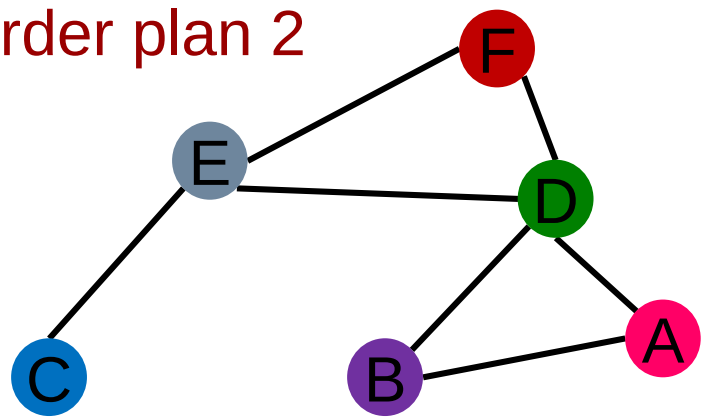
Adjacency matrix A_1



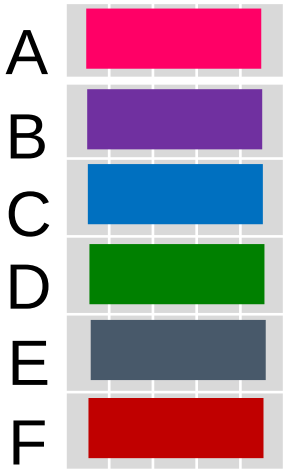
Permutation Invariance

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

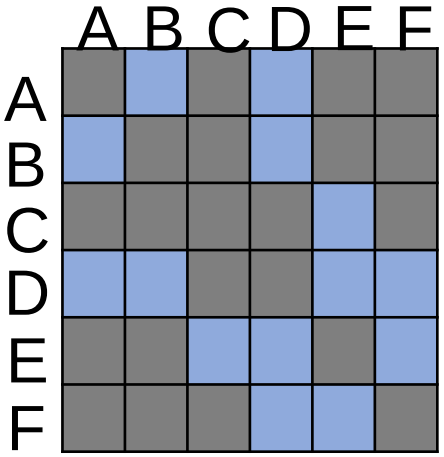
Order plan 2



Node features X_2



Adjacency matrix A_2



Permutation Invariance

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

Invariance and Equivariance

- Permutation-invariant

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

Permute the input, the output stays the same.

- Permutation-equivariant

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

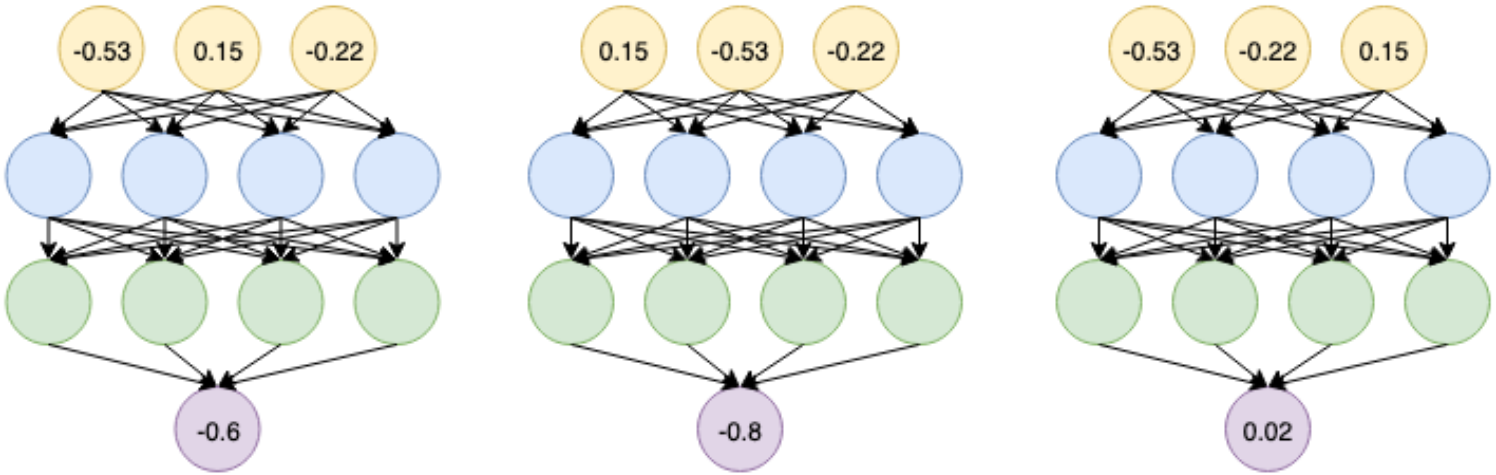
Permute the input, output also permutes accordingly.

Graph Neural Network Overview

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

- No

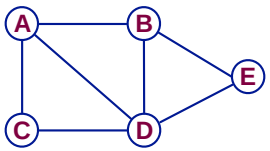
Switching the order of the input leads to different outputs!



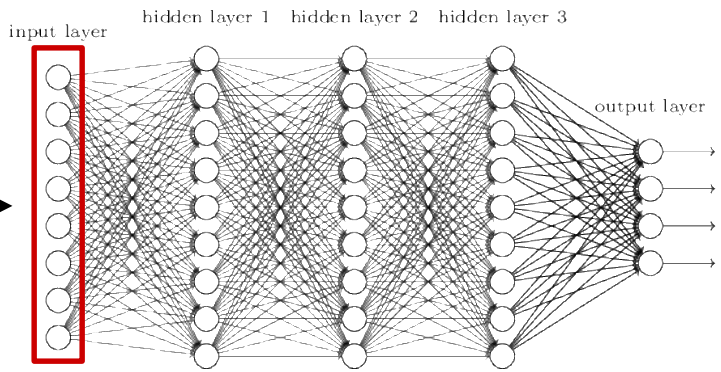
Graph Neural Network Overview

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

- No.



	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

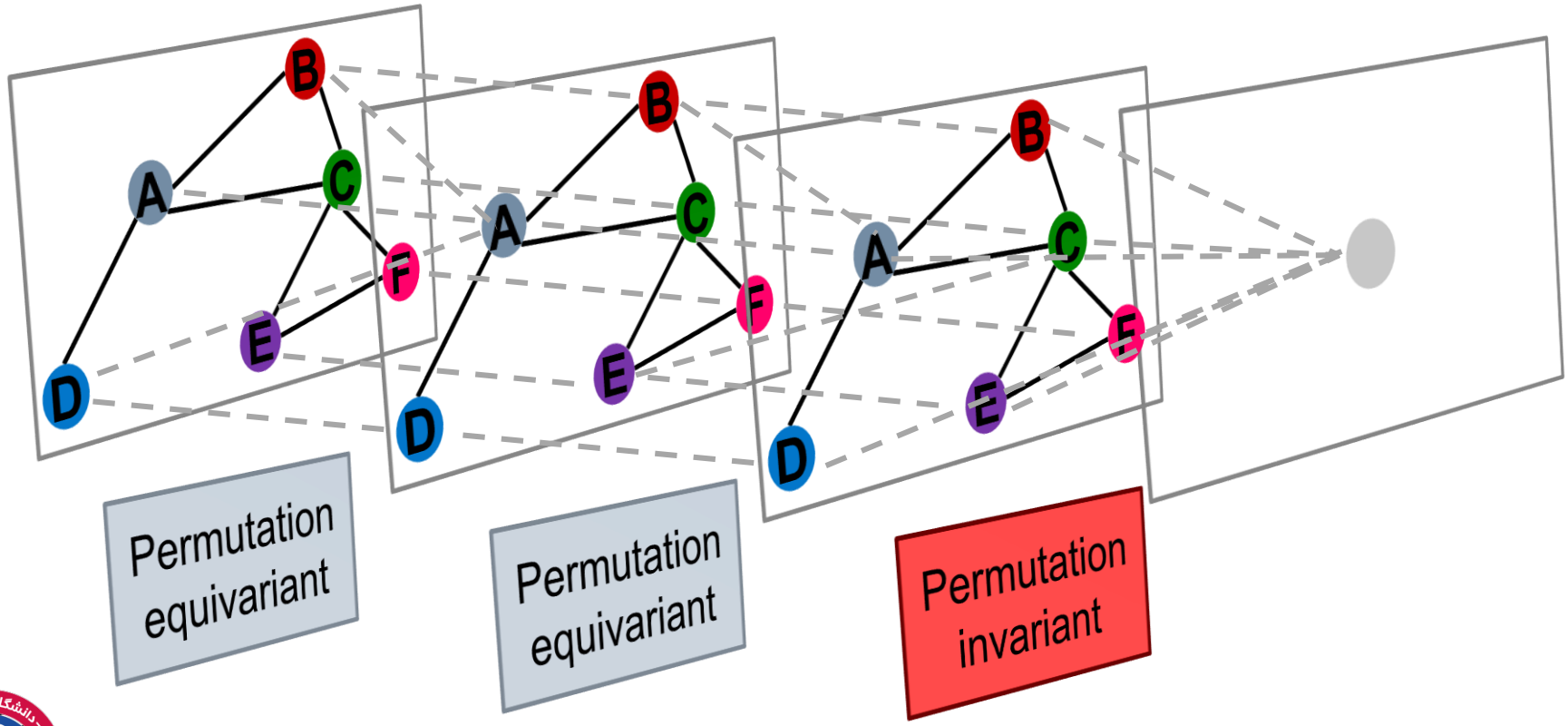


?

This explains why the naïve MLP approach fails for graphs!

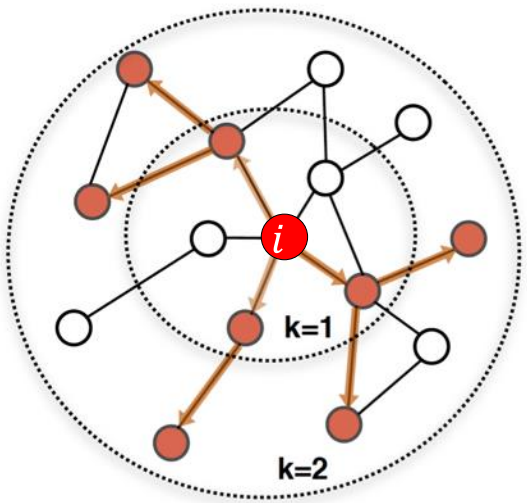
Graph Neural Network Overview

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

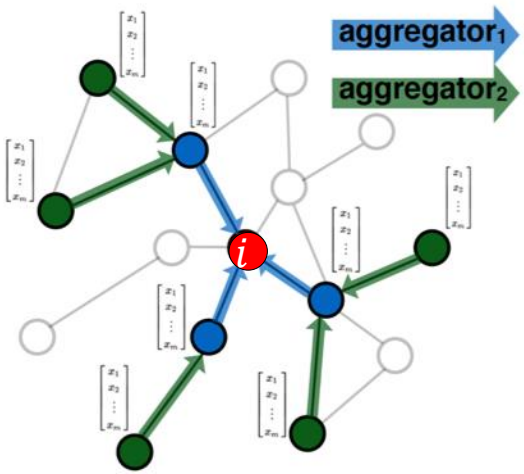


Graph Convolutional Networks

Idea: **The neighborhood of a node defines a computation graph**



Determine node computation graph

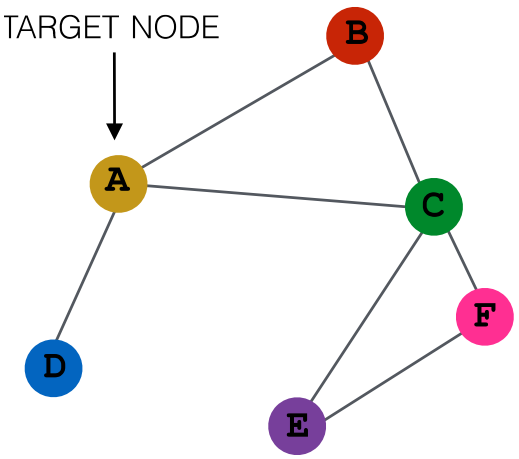


Propagate and transform information

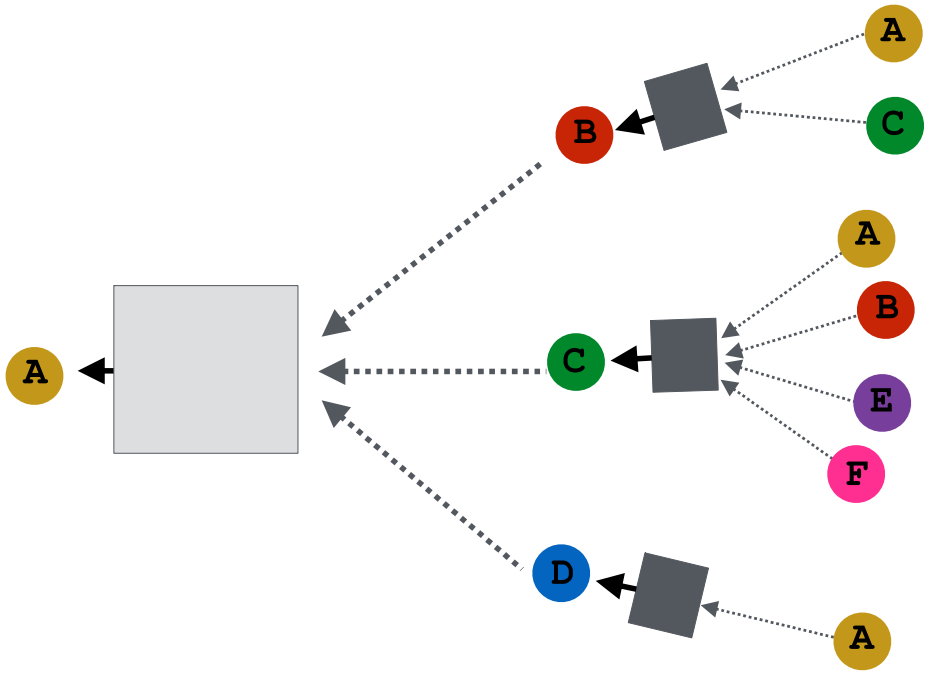
Learn how to propagate information across the graph to compute node features

Idea: Aggregate Neighbors

Key idea: Generate node embeddings based on local network neighborhoods

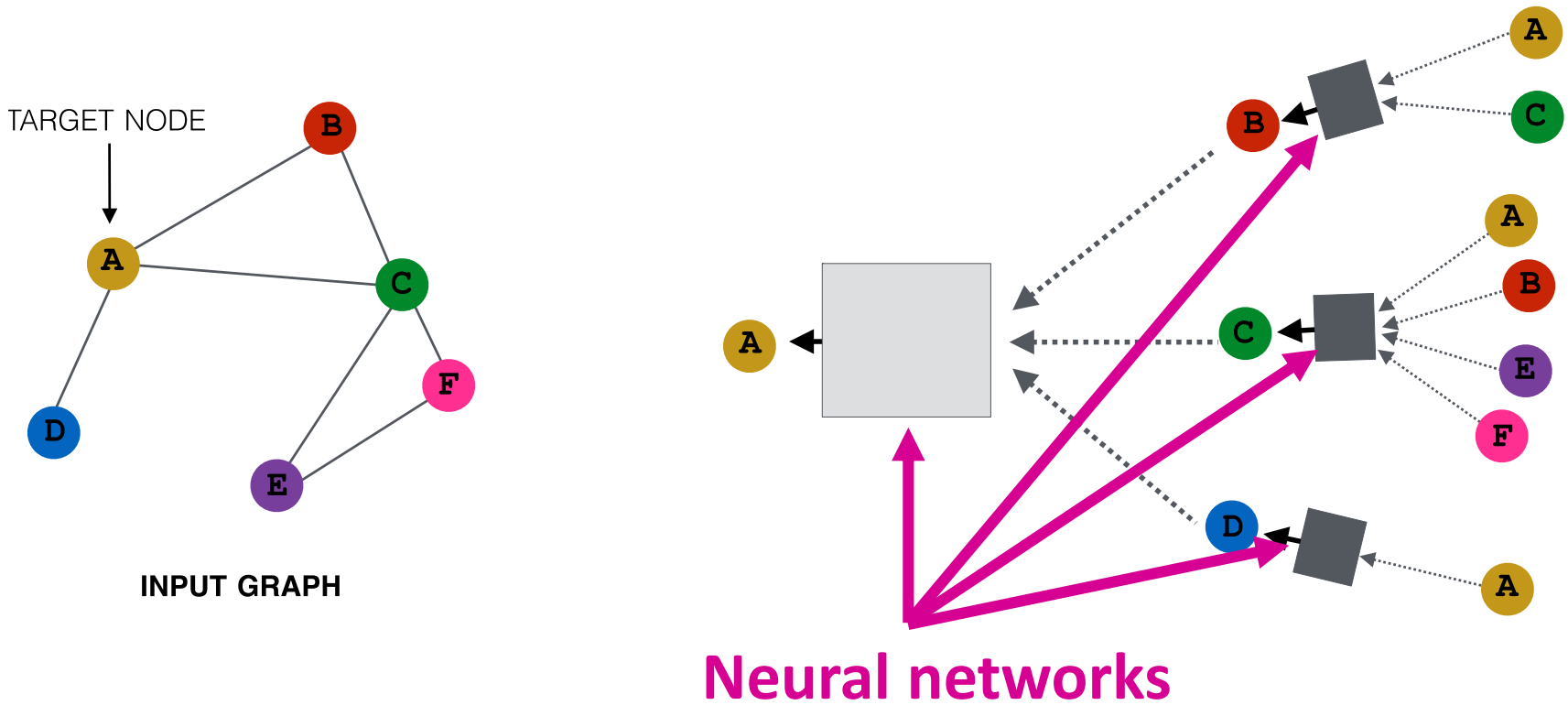


INPUT GRAPH



Idea: Aggregate Neighbors

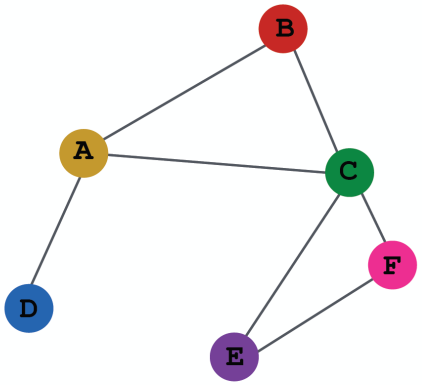
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



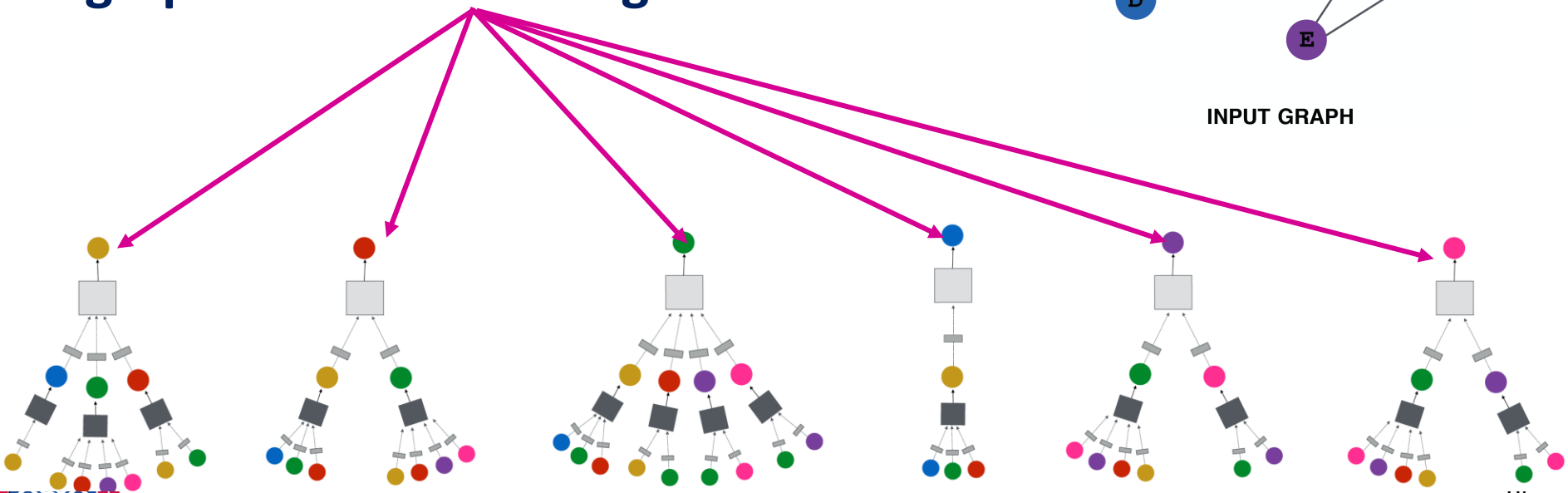
Idea: Aggregate Neighbors

- Intuition: Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!

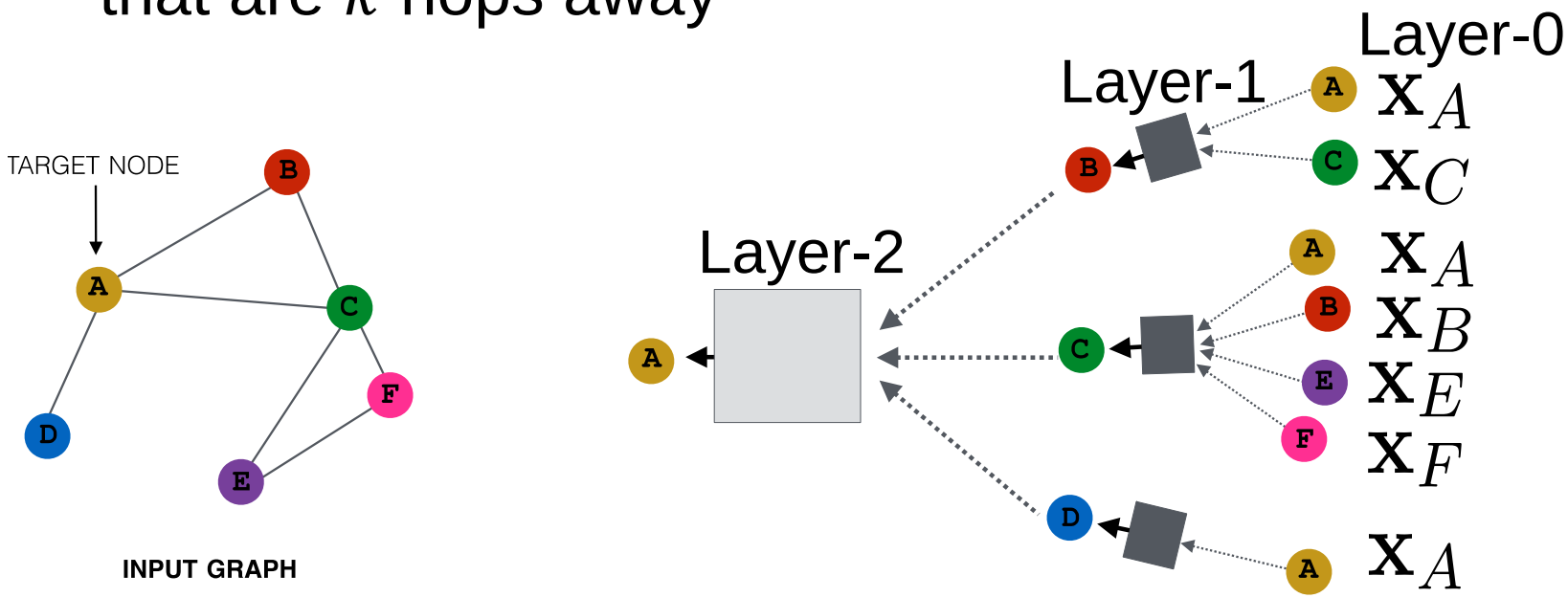


INPUT GRAPH



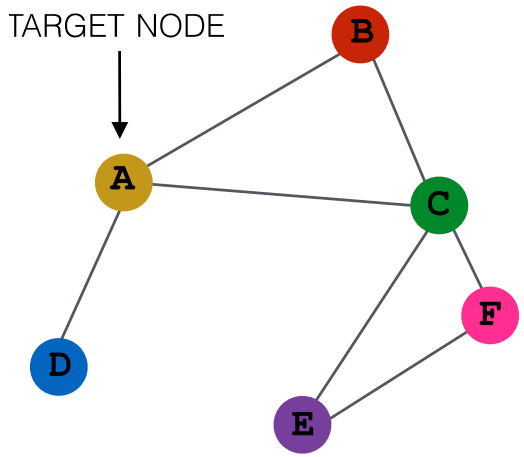
Deep Model: Many Layers

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



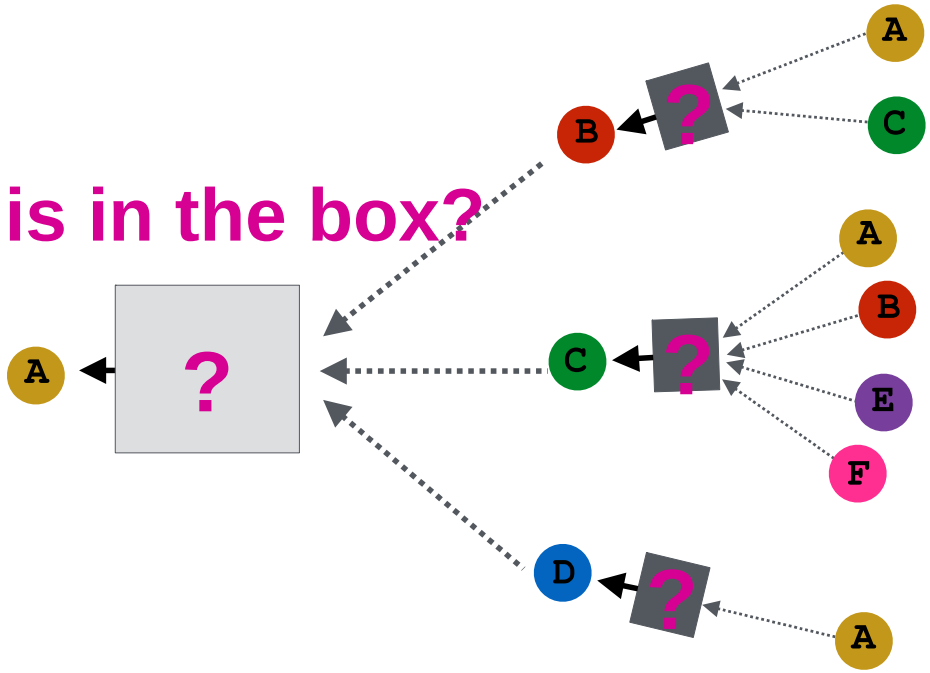
Neighborhood Aggregation

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



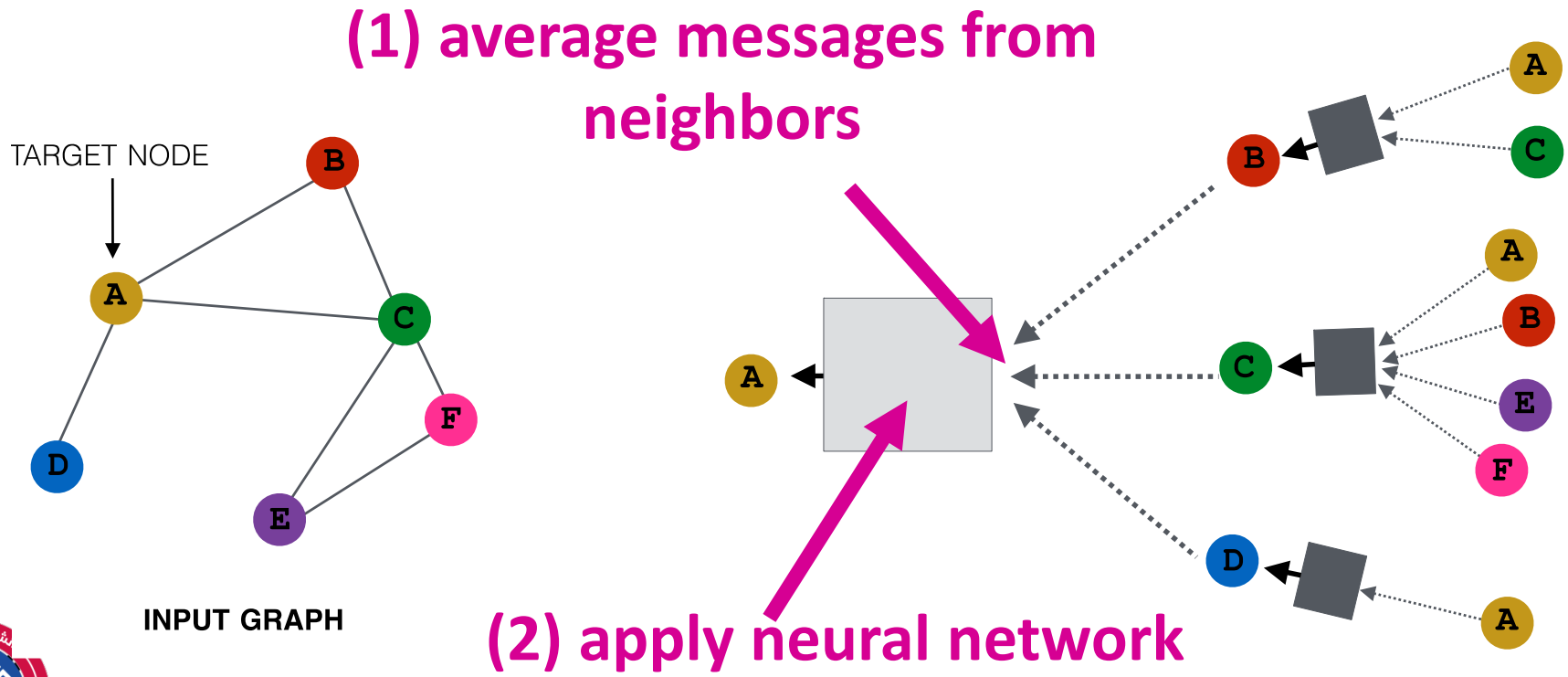
INPUT GRAPH

What is in the box?



Neighborhood Aggregation

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



The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

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

embedding of v at layer k

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

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

Average of neighbor's previous layer embeddings

Total number of layers

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**

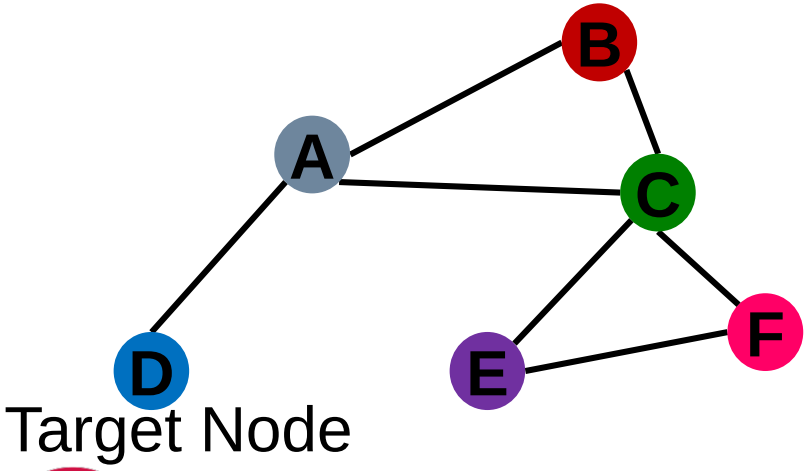
h_v^k: the hidden representation of node v at layer k

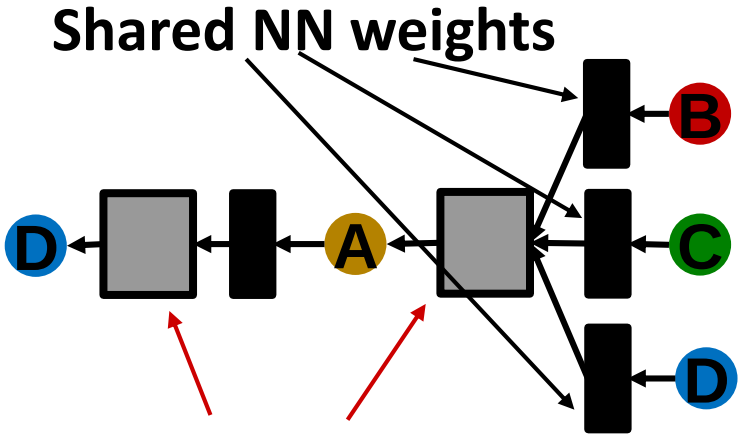
- W_k: weight matrix for neighborhood aggregation
- B_k: weight matrix for transforming hidden vector of self

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

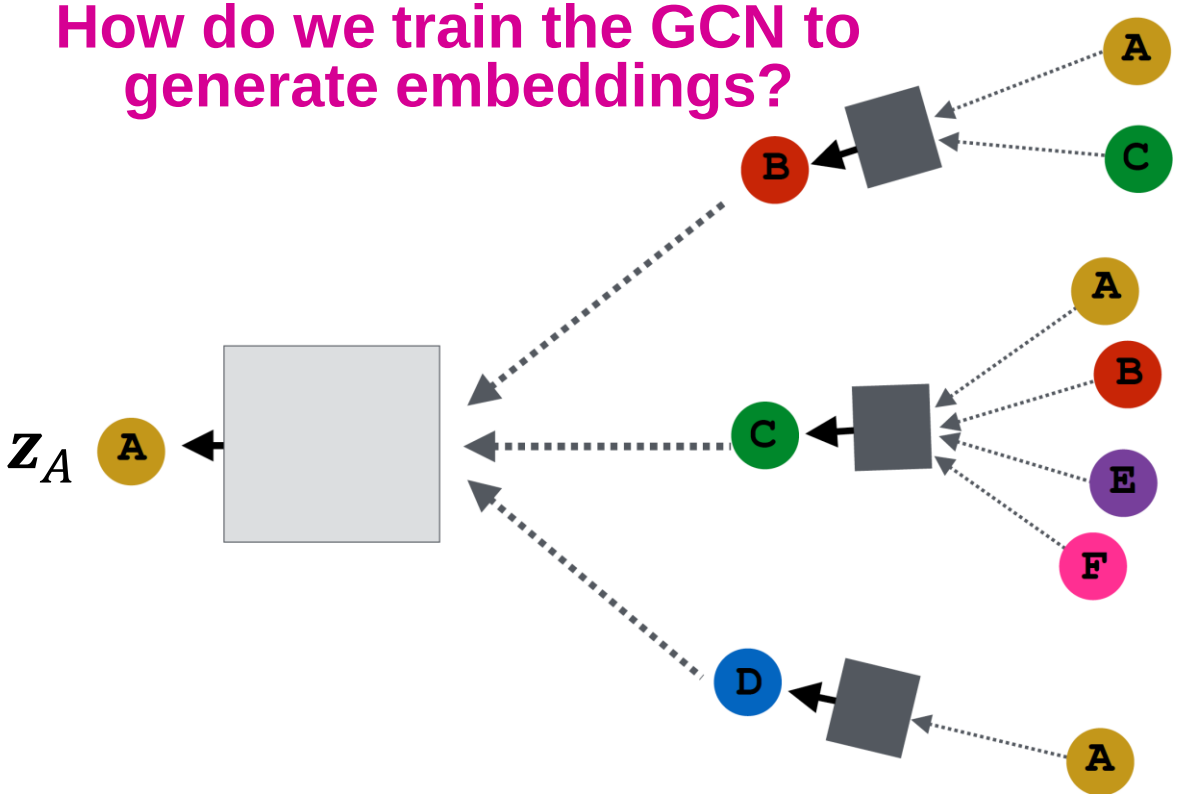




Average of neighbor's previous layer embeddings - Permutation invariant

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:

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

- 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

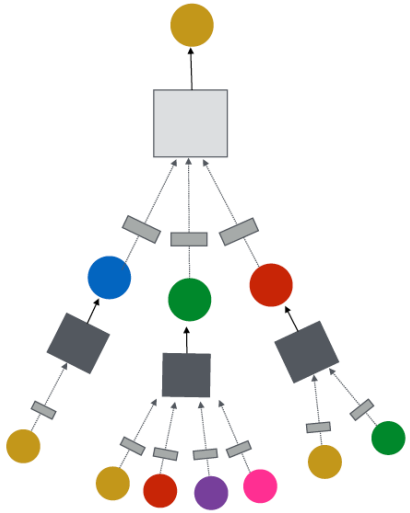
Node similarity can be anything from embeddings, e.g., a loss based on:

- **Random walks** (node2vec, DeepWalk, struc2vec)
- **Matrix factorization**

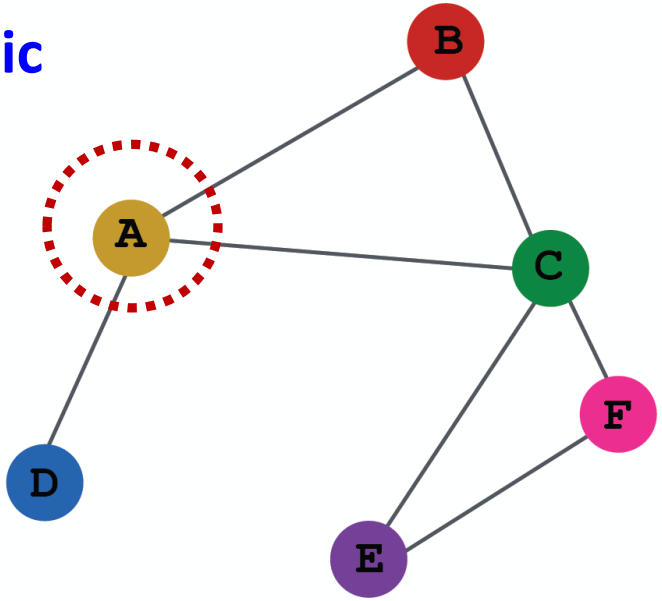
Supervised Training

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**

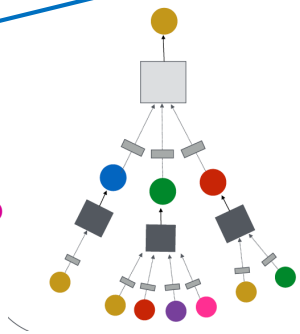
$$\mathcal{L} = -\sum_{v \in V} \boxed{y_v} \log(\sigma(\boxed{z_v^T}\boxed{\theta})) + (1 - \boxed{y_v}) \log(1 - \sigma(\boxed{z_v^T}\boxed{\theta}))$$

Encoder output: node embedding

Classification weights

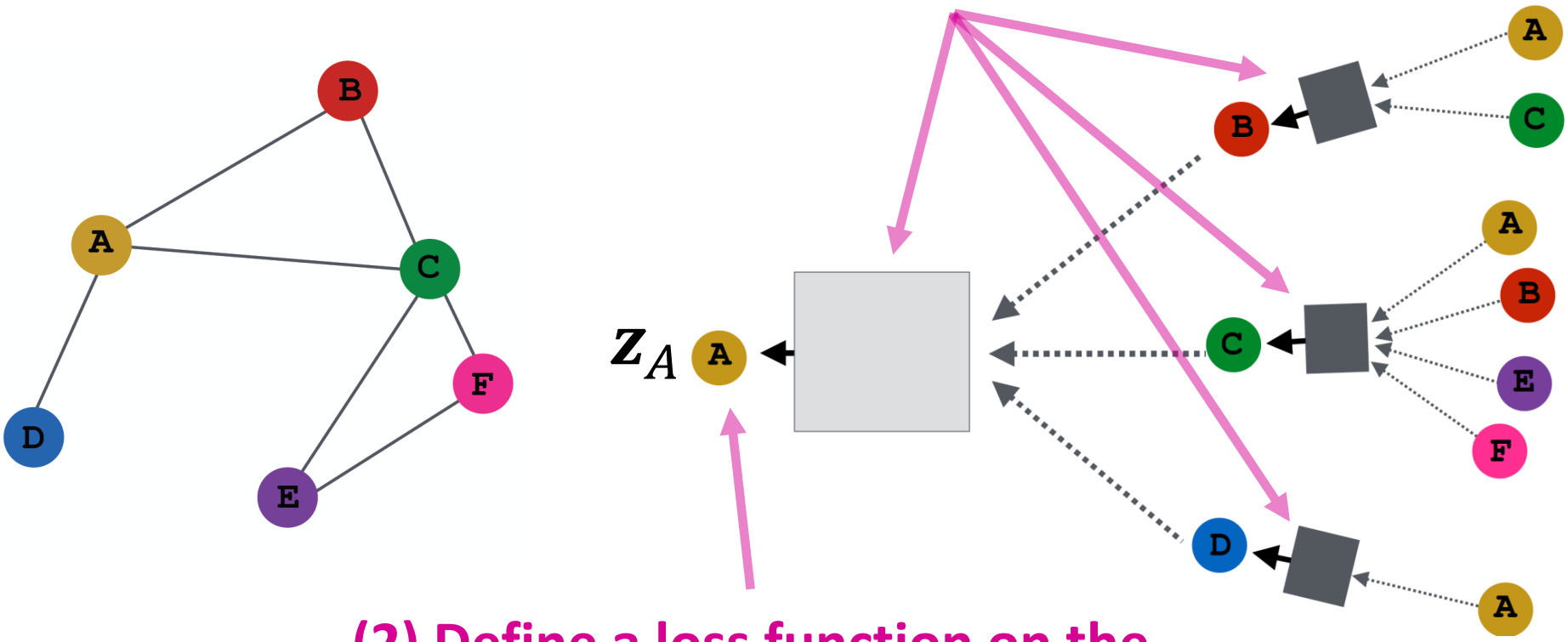
Node class label

Safe or toxic drug?



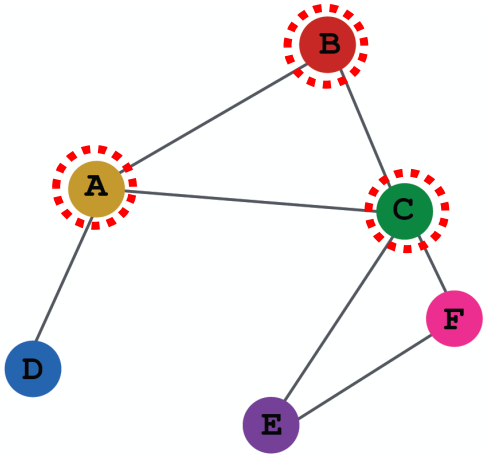
Model Design: Overview

(1) Define a neighborhood aggregation function



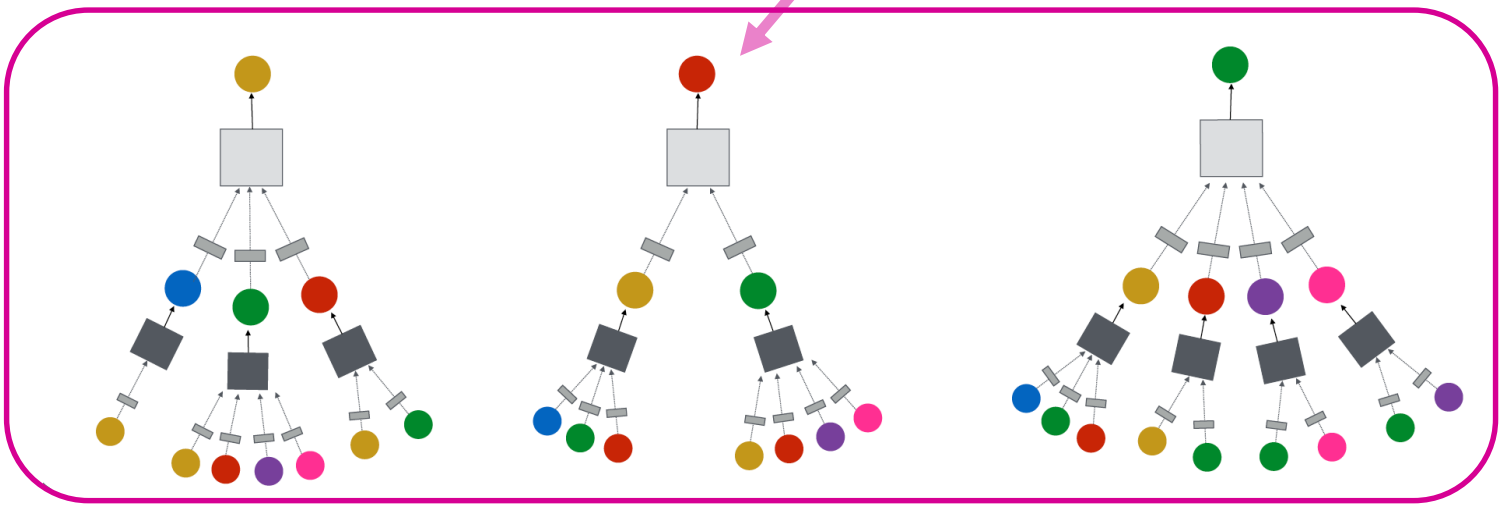
(2) Define a loss function on the embeddings

Model Design: Overview

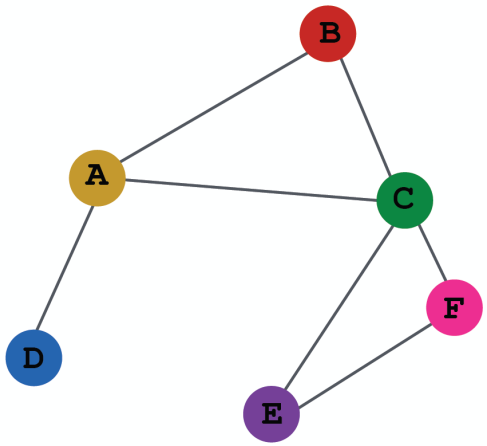


INPUT GRAPH

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



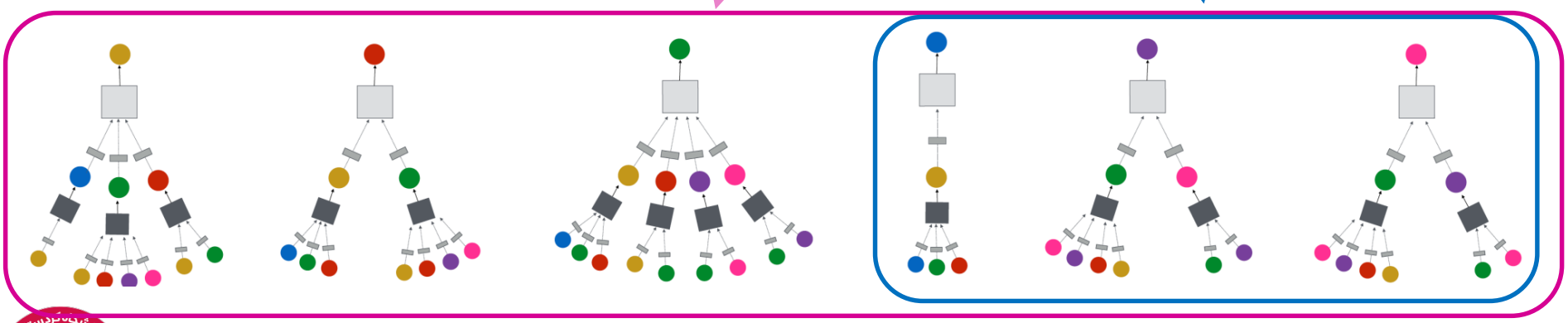
Model Design: Overview



INPUT GRAPH

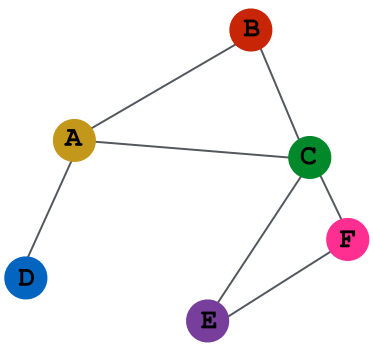
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

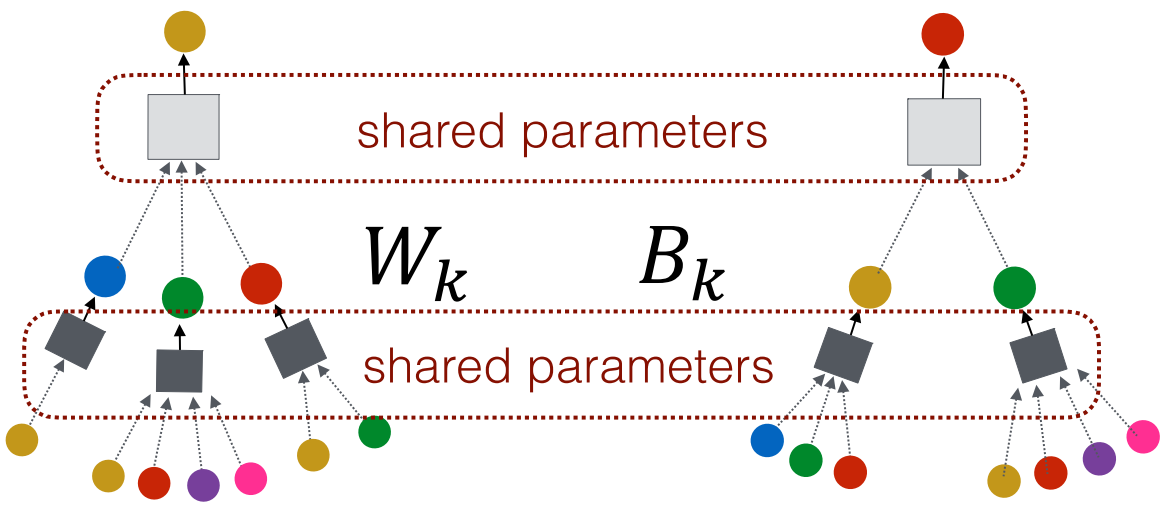


Inductive Capability

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

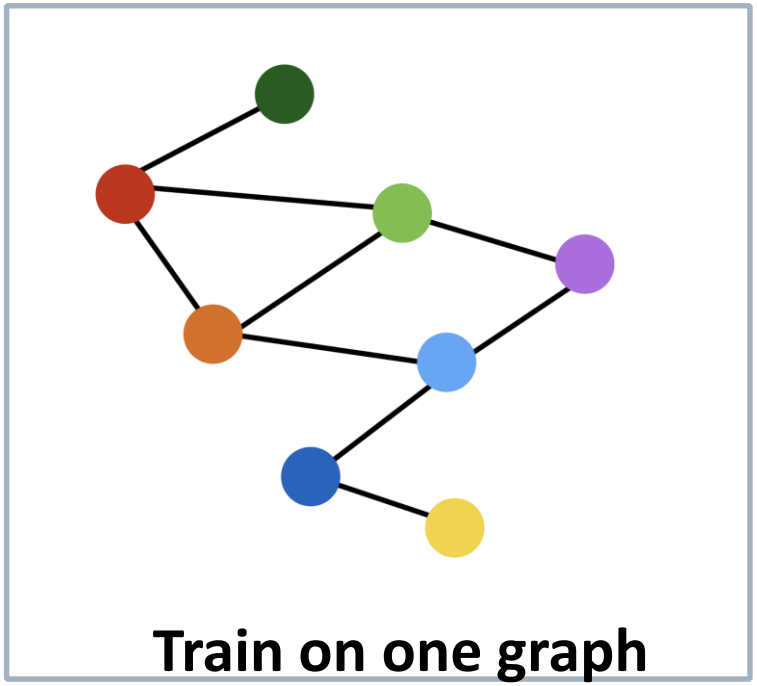


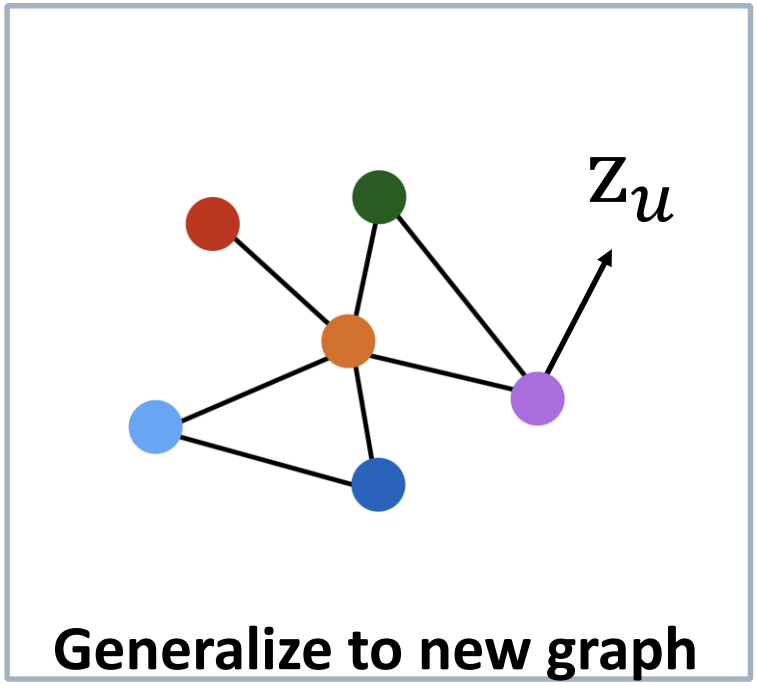
INPUT GRAPH



Compute graph for node A **Compute graph for node B**

Inductive Capability: New Graphs

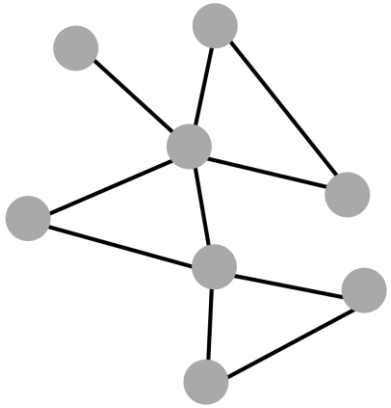




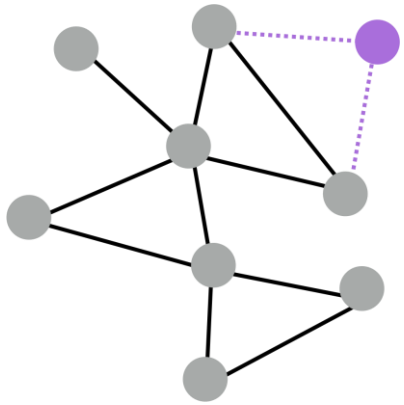
Inductive node embedding → **Generalize to entirely unseen graphs**

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

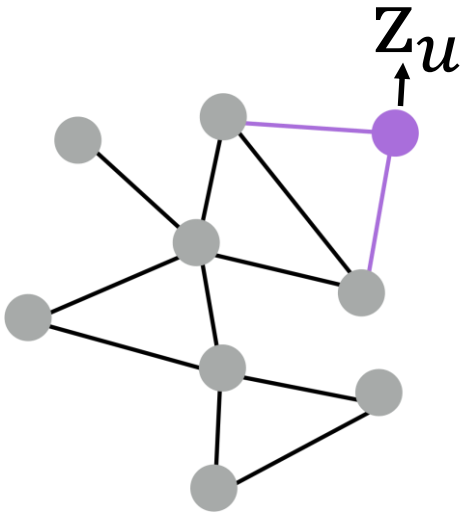
Inductive Capability: New Nodes



Train with snapshot



New node arrives



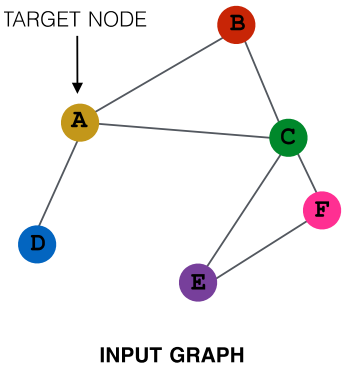
Generate embedding for new node

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

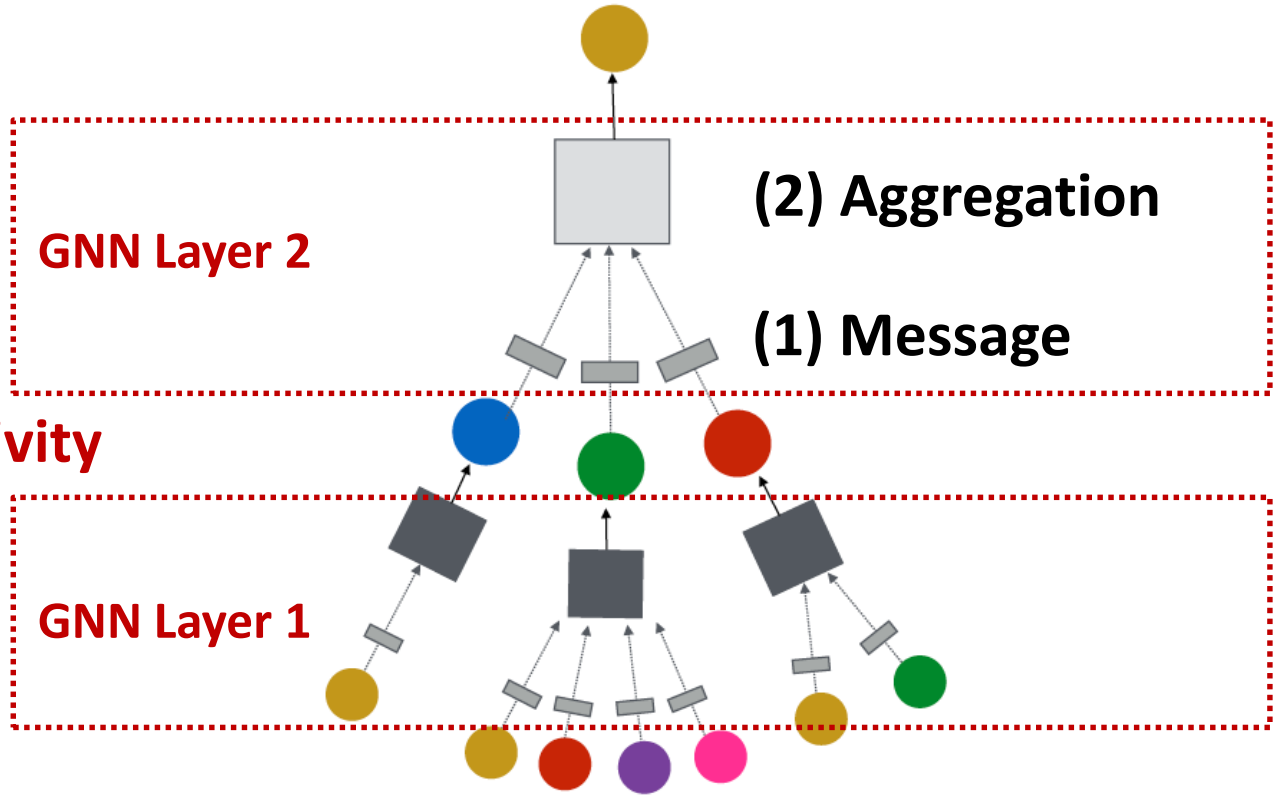
Summary so far

- How to build CNNs for graphs use local neighborhood of a node
- Next: more details using a general GNN framework

A General GNN Framework (5 main issues)



(5) Learning objective



(3) Layer connectivity

(4) Graph augmentation

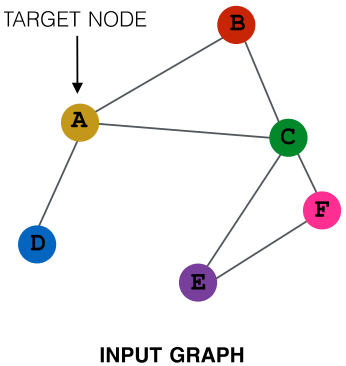
General Framework

5 main issues

- A single GNN layer: Aggregation (1) and Message (2)
- Layer connectivity: Stacking (3)
- Graph manipulations(augmentation) (4)
- Learning objectives/metrics (5)

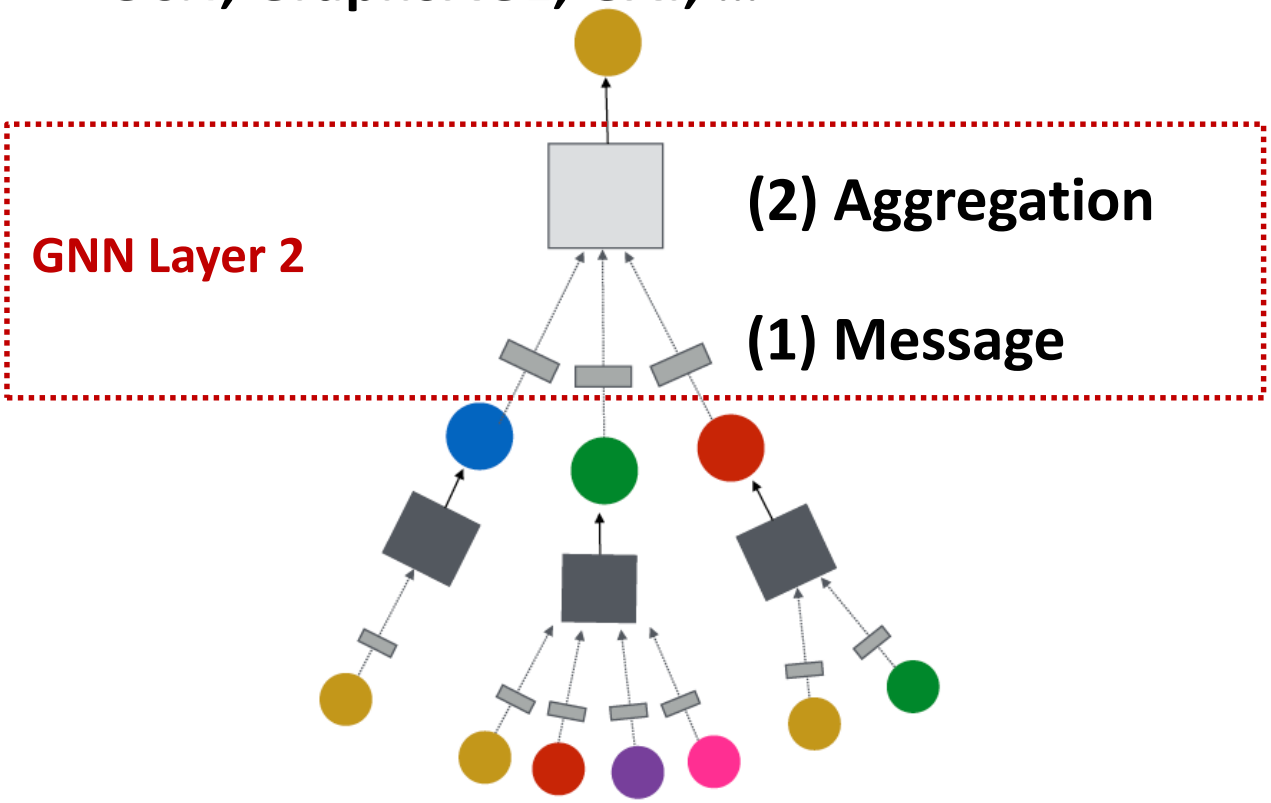
A SINGLE GNN LAYER

A GNN Layer



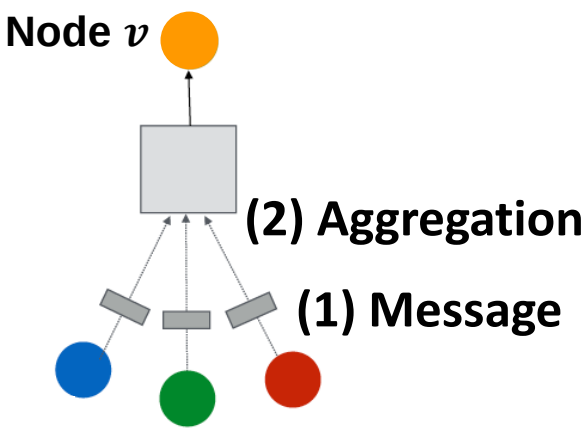
GNN Layer = Message + Aggregation

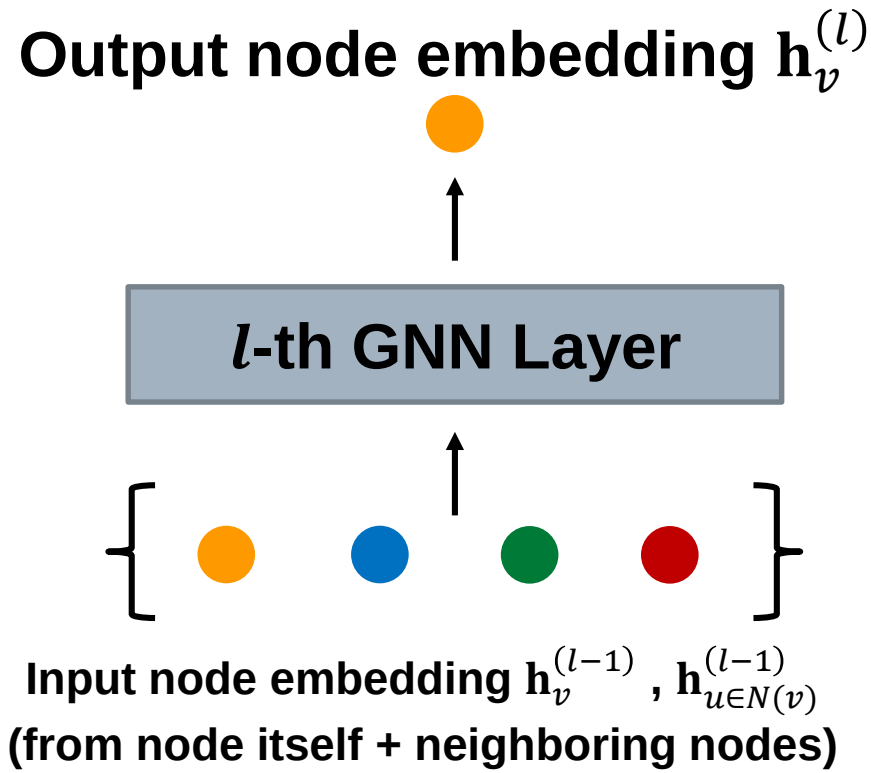
- **Different instantiations under this perspective**
- **GCN, GraphSAGE, GAT, …**



A Single GNN Layer

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

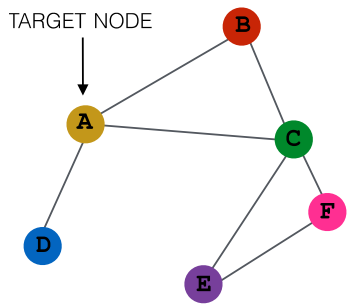




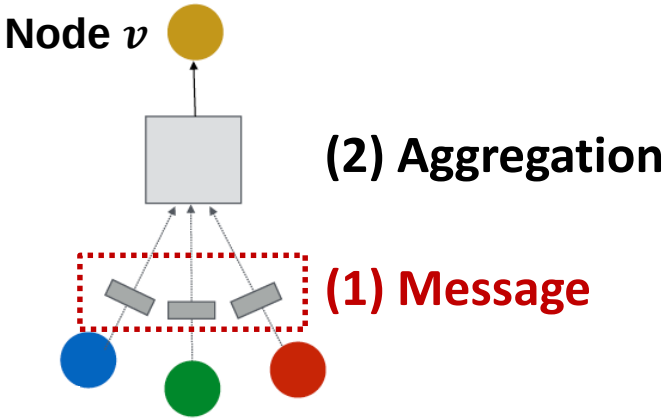
Message Computation

(1) Message computation

- **Message function**: $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right)$
  - **Intuition:** Each node will create a message, which will be sent to other nodes
  - **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



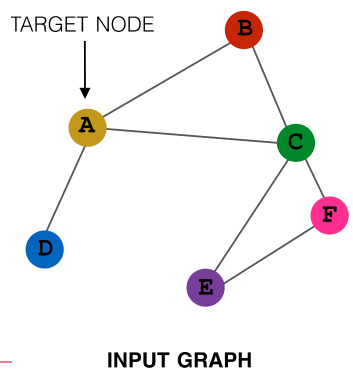
Message Aggregation

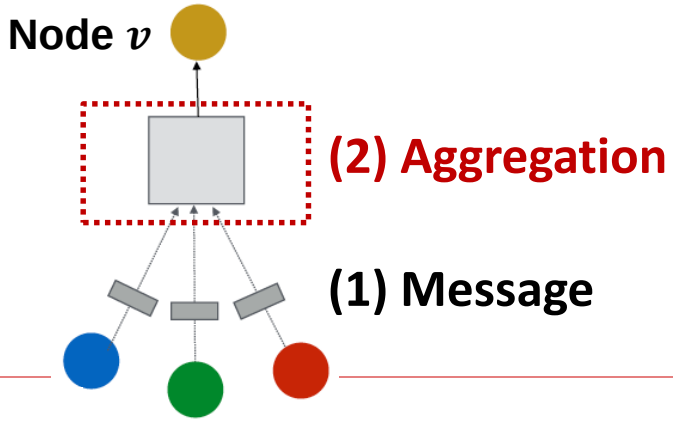
(2) Aggregation

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





Classical GNN Layers: GCN (1)

(1) Graph Convolutional Networks (GCN)

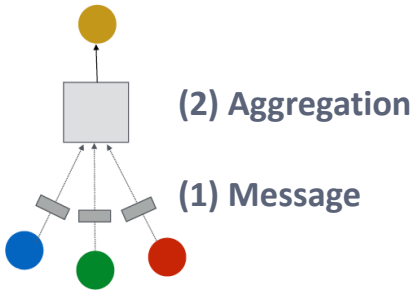
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

- How to write this as Message + Aggregation?

Message

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

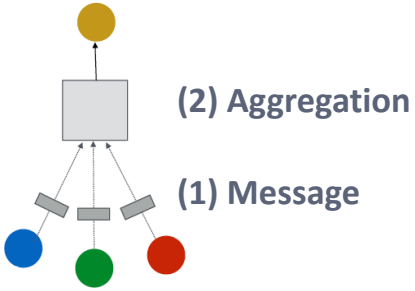
Aggregation



Classical GNN Layers: GCN (2)

(1) Graph Convolutional Networks (GCN)

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



- **Message:**
  - Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{|N(v)|}\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$

Normalized by node degree

(In the GCN paper they use a slightly different normalization)

- **Aggregation:**
  - **Sum** over messages from neighbors, then apply activation
  - $\mathbf{h}_v^{(l)} = \sigma\left(\text{Sum}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)\right)$

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

Classical GNN Layers: GCN

Basic Neighborhood Aggregation

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

VS.

GCN Neighborhood Aggregation

Kipf & Welling

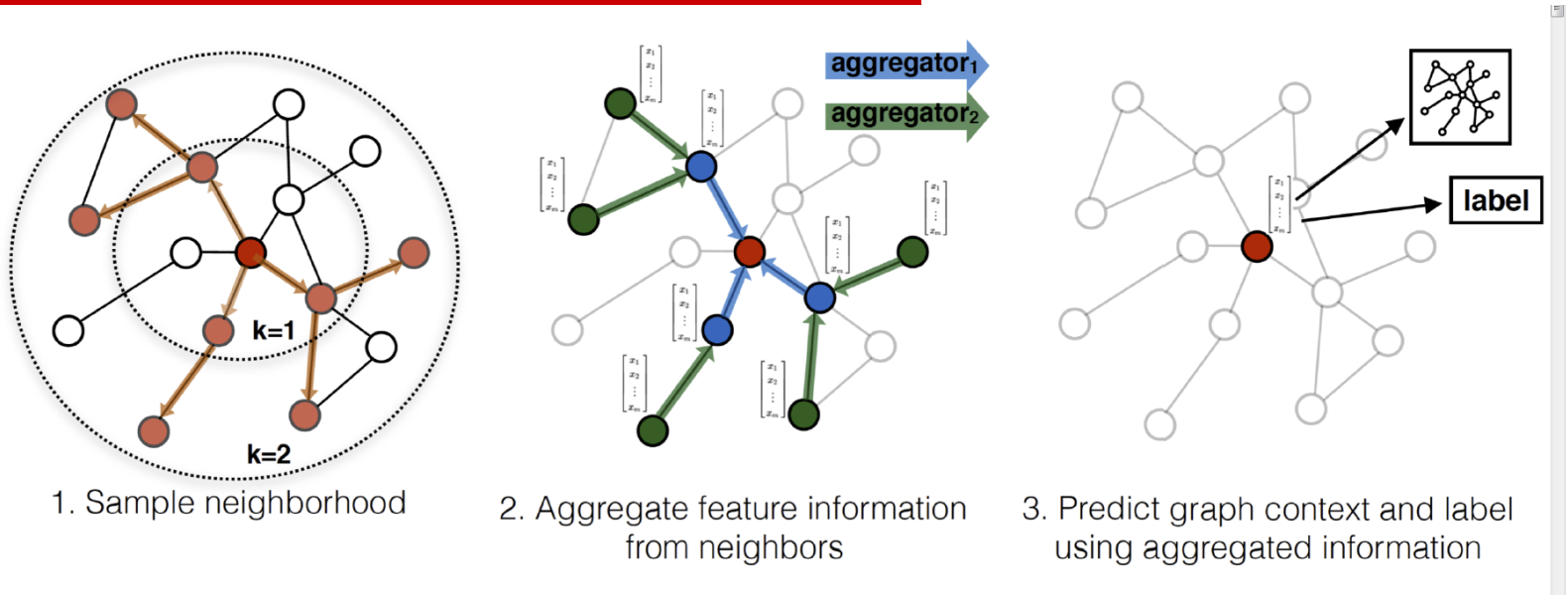
$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v) \cup v} \frac{\mathbf{h}_u^{k-1}}{\sqrt{|N(u)||N(v)|}}\right)$$

same matrix for self and neighbor embeddings

per-neighbor normalization

$$H^{(k)} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(k-1)} W^{(k-1)}\right), \quad \tilde{A} = A + I_N$$

Classical GNN Layers: GraphSAGE



(**SA**mple and aggre**G**at**E**),

- A general inductive framework that efficiently generate node embeddings for previously unseen data.
- Uniformly sample a fixed-size set of neighbors, instead of
- using full neighborhood sets

Classical GNN Layers: GraphSAGE

(2) GraphSAGE

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

-
  - **Message** is computed within the $\mathrm{AGG}(\cdot)$
  - **Two-stage aggregation**
    - **Stage 1:** Aggregate from node neighbors

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

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

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

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

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

applied to a random permutation

Aggregation

GraphSAGE: L2 Normalization

ℓ_2 Normalization:

- **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
- $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\left\|\mathbf{h}_v^{(l)}\right\|_2} \; \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2-norm)
  - Without ℓ_2 normalization, the *embedding vectors have different scales* (ℓ_2-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement

Classical GNN Layers: GAT (1)

(3) Graph Attention Networks

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

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

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

Attention weights

Not all node's neighbors are equally important

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

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)

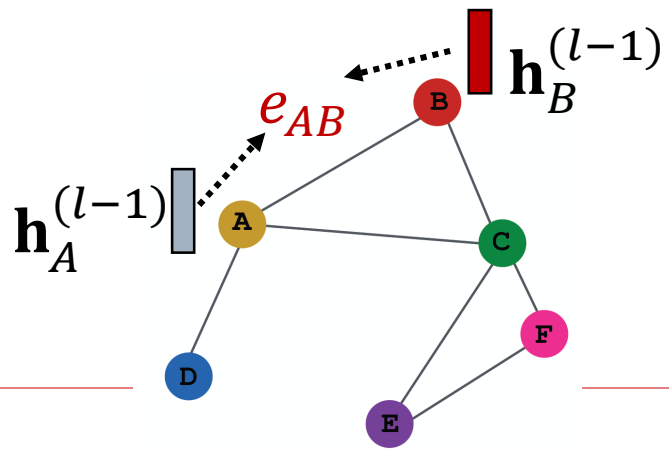
Let α_{vu} be computed as a byproduct of an **attention mechanism a:**

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

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

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

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



Attention Mechanism (2)

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

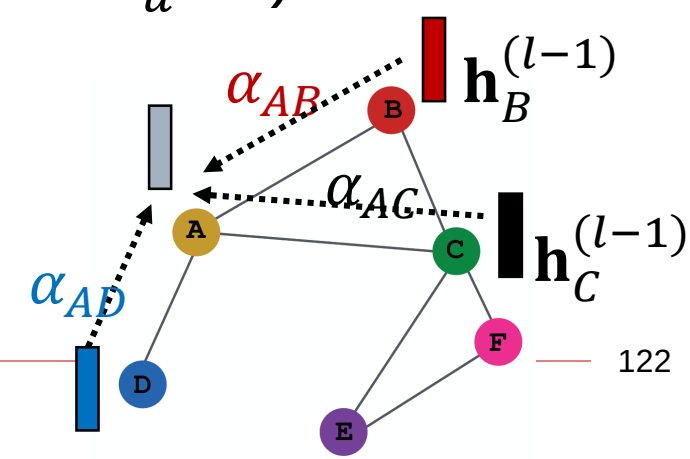
$$= \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** :

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

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

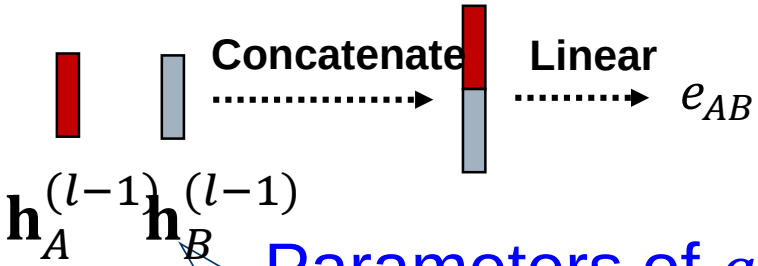
$$\mathbf{h}_A^{(l)} = \sigma\left(\alpha_{AB} \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)} + \alpha_{AC} \mathbf{W}^{(l)} \mathbf{h}_C^{(l-1)} + \alpha_{AD} \mathbf{W}^{(l)} \mathbf{h}_D^{(l-1)}\right)$$



Attention Mechanism (3)

What is the form of attention mechanism a?

- The approach is agnostic to the choice of a
  - E.g., use a simple single-layer neural network
    - a have trainable parameters (weights in the Linear layer)



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

- 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)

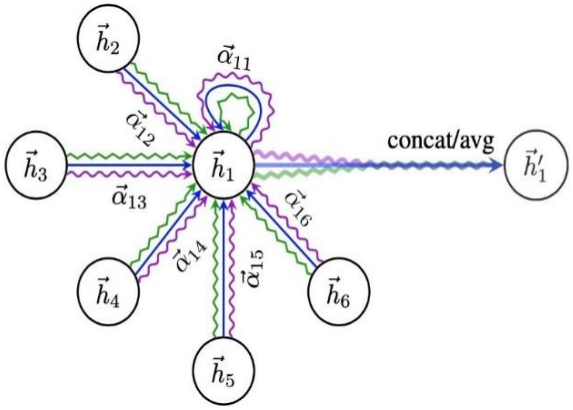
- **Multi-head attention:** Stabilizes the learning process of attention mechanism
  - **Create multiple attention scores** (each replica with a different set of parameters):

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

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

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

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



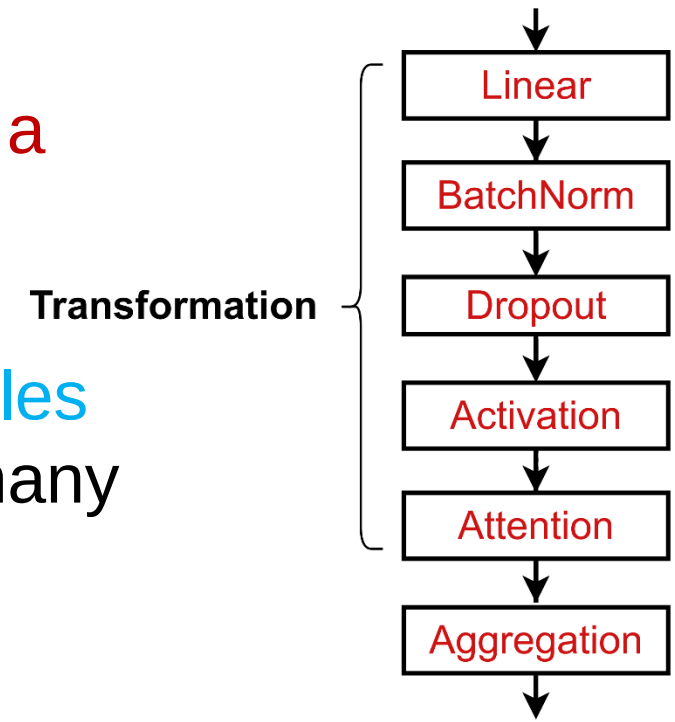
Benefits of Attention Mechanism

- **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

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

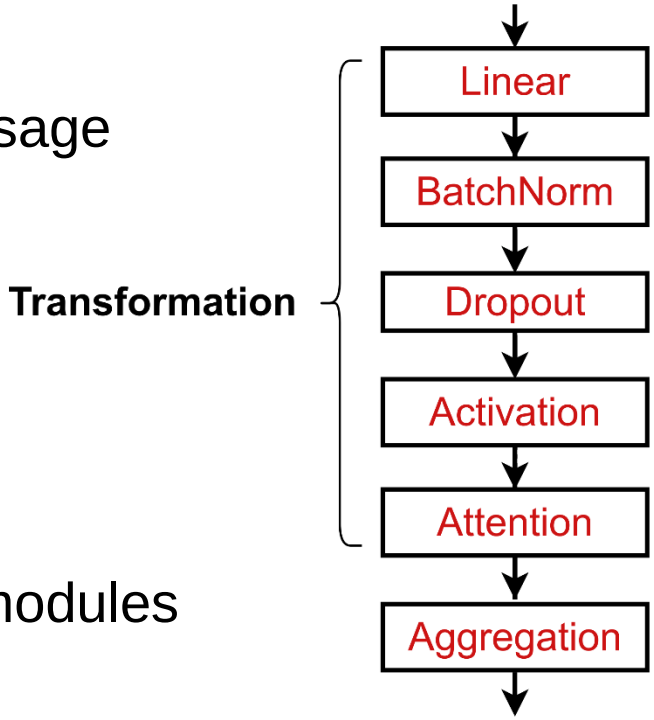
A suggested GNN Layer



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



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:

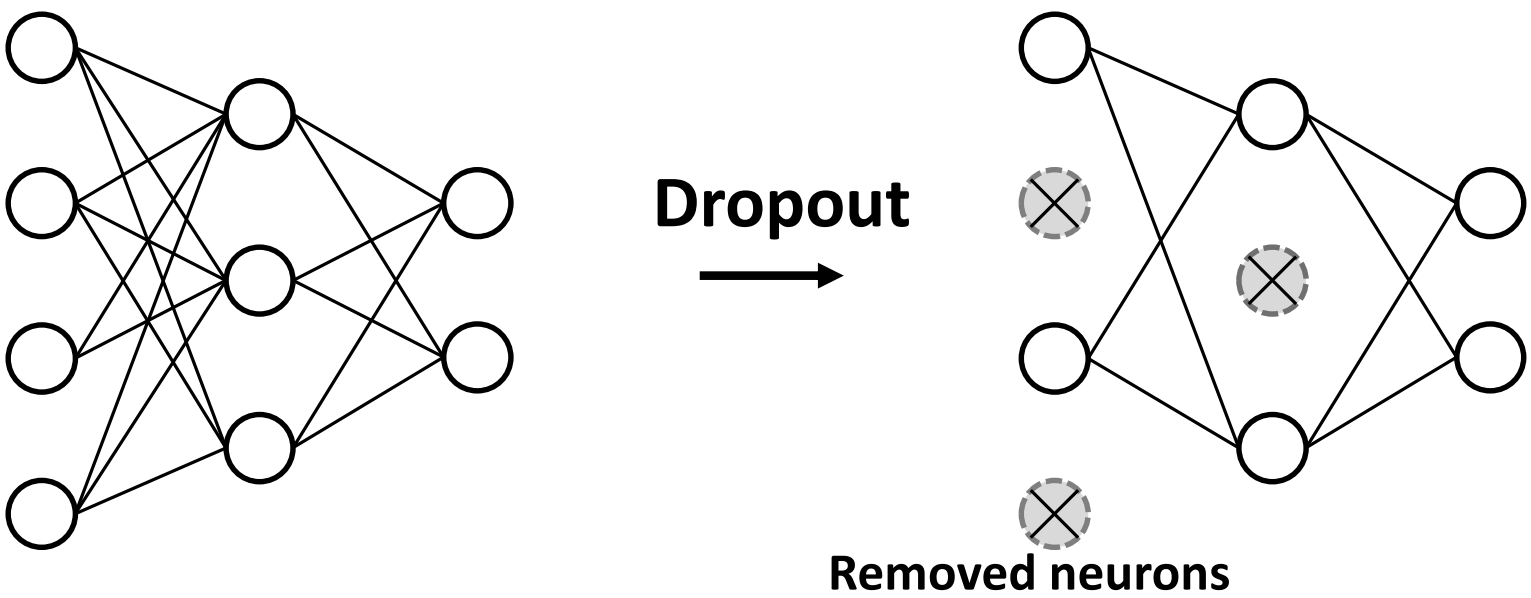
Normalize the feature using computed mean and variance

$$\widehat{\mathrm{X}}_{i,j} = \frac{\mathrm{X}_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$\mathrm{Y}_{i,j} = \gamma_j \widehat{\mathrm{X}}_{i,j} + \beta_j$$

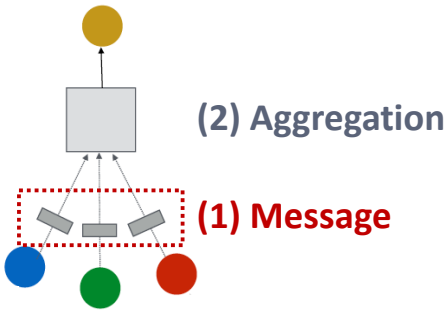
Dropout

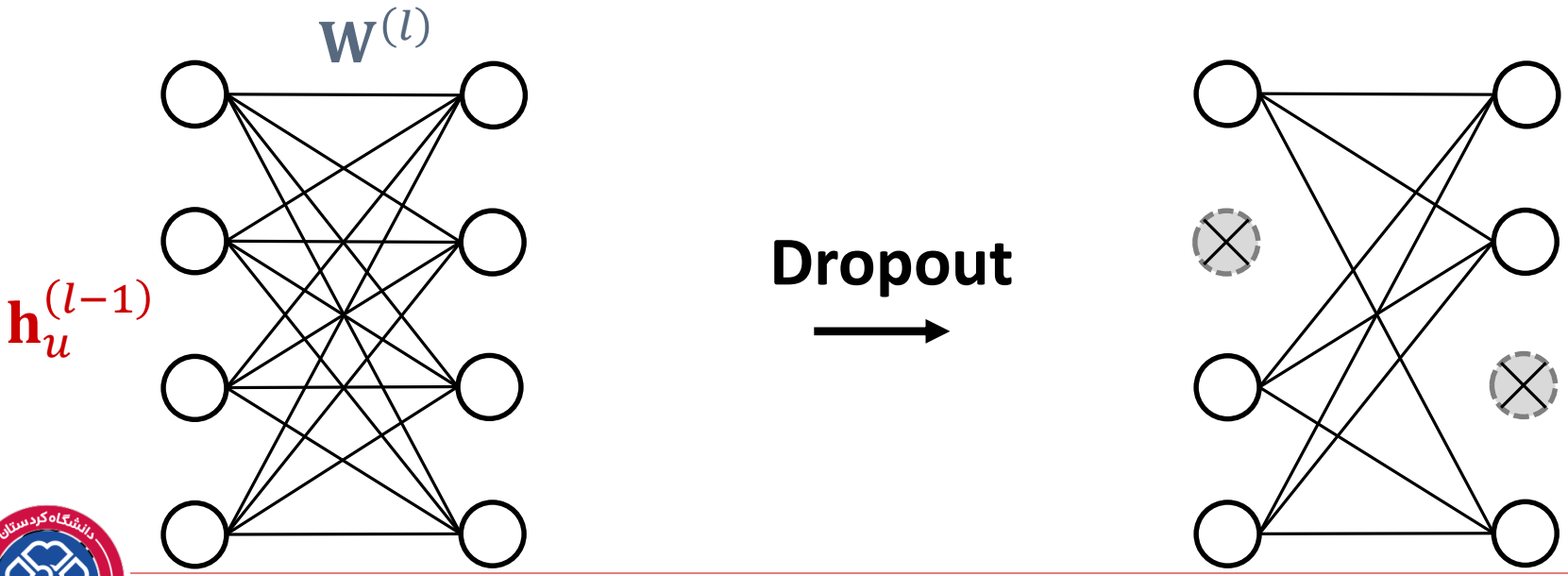
- **Goal**: Regularize a neural net to prevent overfitting.
- **Idea**:
  - **During training**: with some probability p, randomly set neurons to zero (turn off)
  - **During testing:** Use all the neurons for computation



Dropout for GNNs

- In GNN, Dropout is applied to the linear layer in the message function
  - **A simple message function with linear layer:** $= \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$



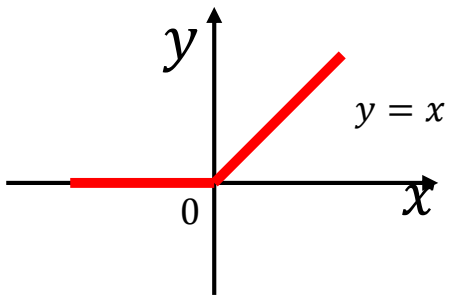


Activation (Non-linearity)

- **Rectified linear unit (ReLU)**

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

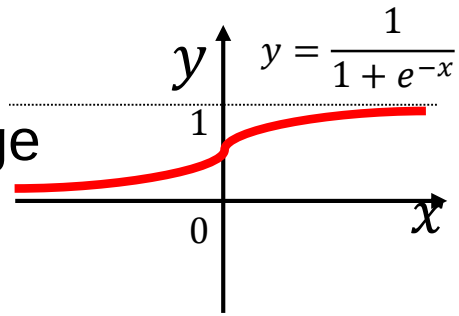
  - Most commonly used



- **Sigmoid**

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

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

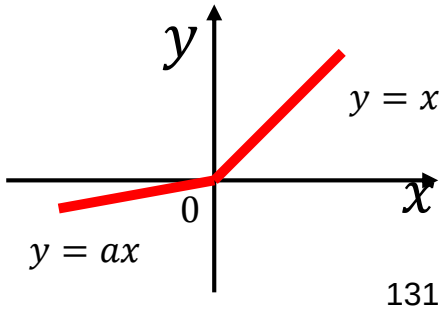


- **Parametric ReLU**

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

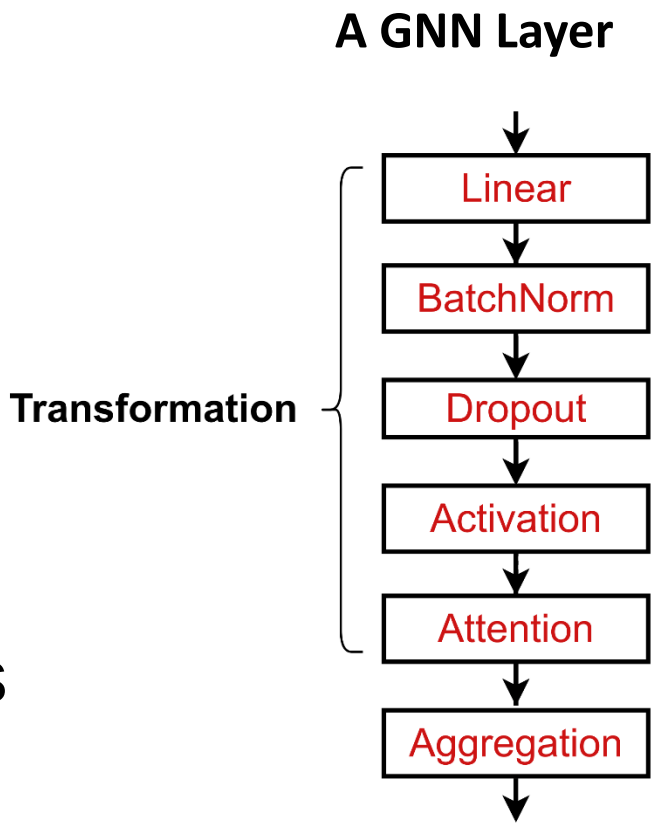
  a_i is a trainable parameter

  - 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

General Framework

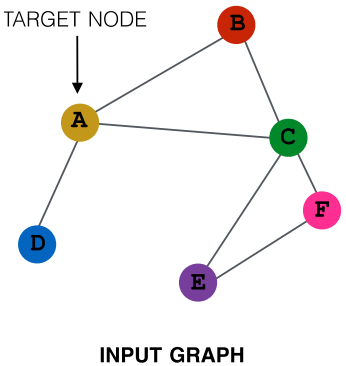
5 main issues

- A single GNN layer: Aggregation (1) and Message (2)
- Layer connectivity: Stacking (3)
- Graph manipulations(augmentation) (4)
- Learning objectives/metrics (5)

STACKING LAYERS

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

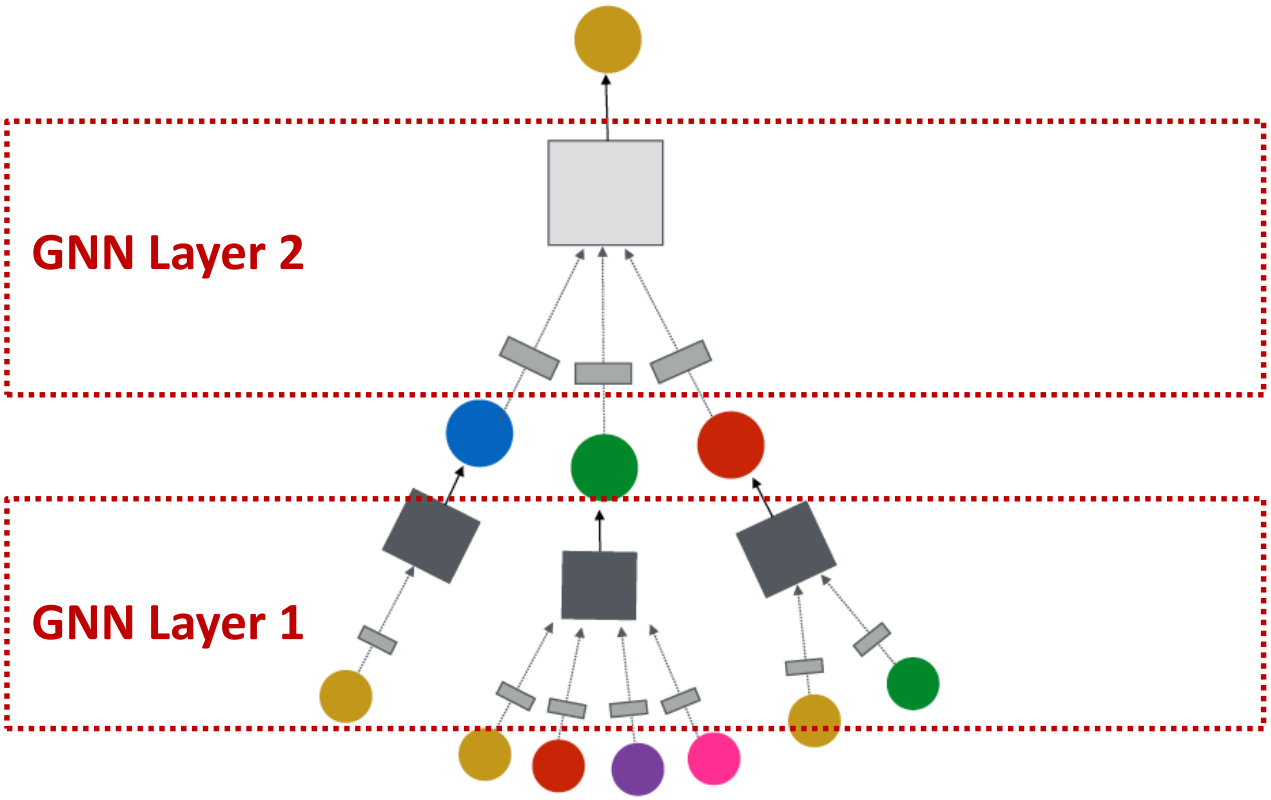
Stacking GNN Layers



How to connect GNN layers into a GNN?

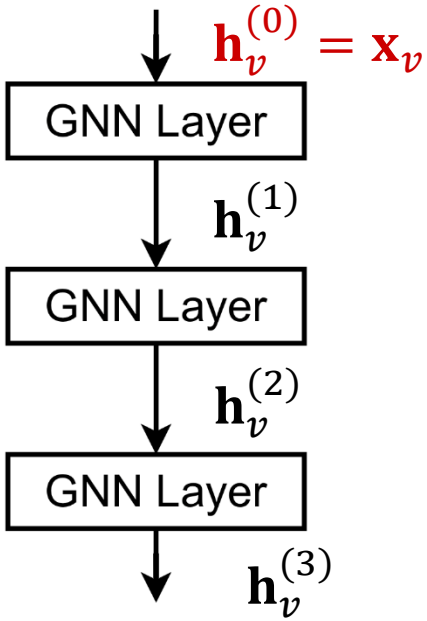
- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity

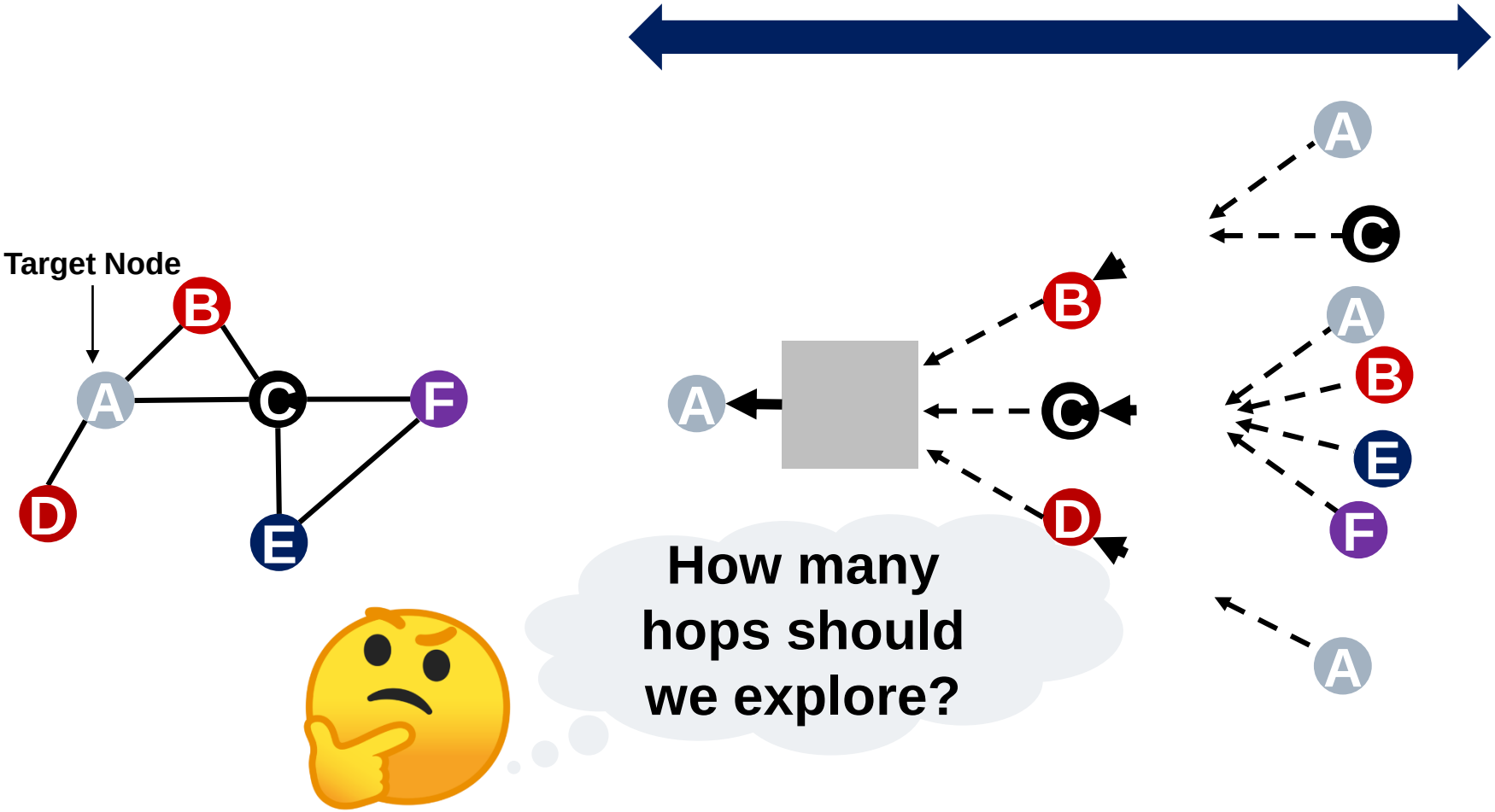


Stacking GNN Layers

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



Graph Neural Networks - Depth



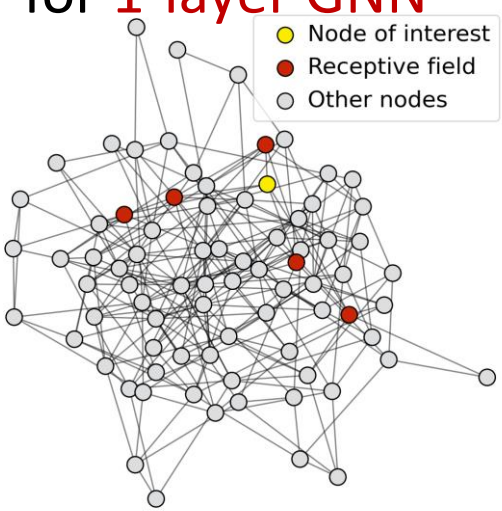
The Over-Smoothing Problem

- 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?

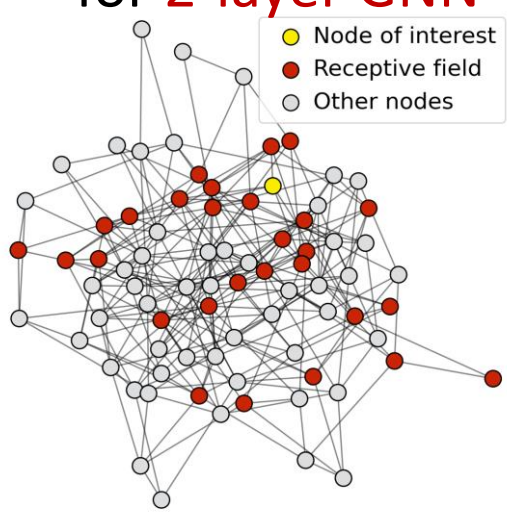
Receptive Field of a GNN

- **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**

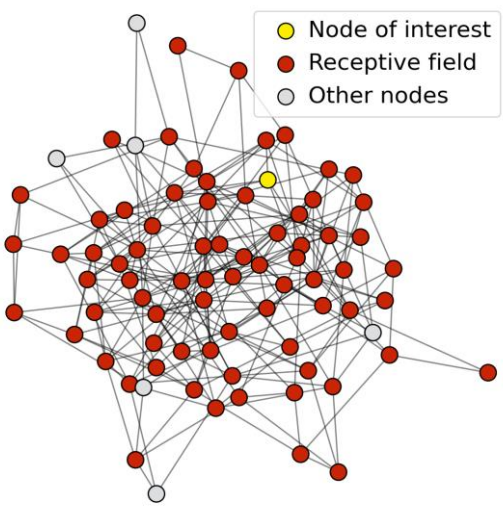
Receptive field for 1-layer GNN



Receptive field for 2-layer GNN



Receptive field for 3-layer GNN

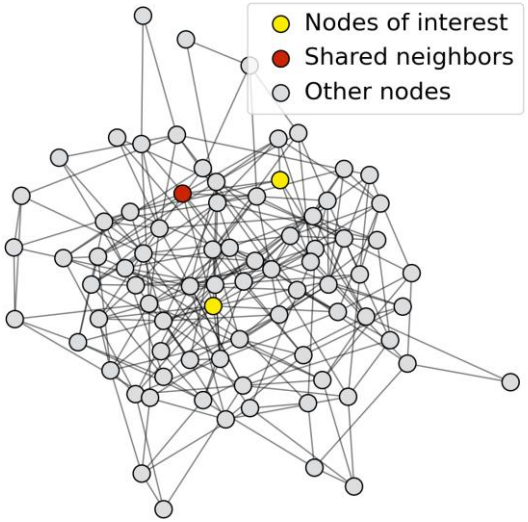


Receptive Field of a GNN

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

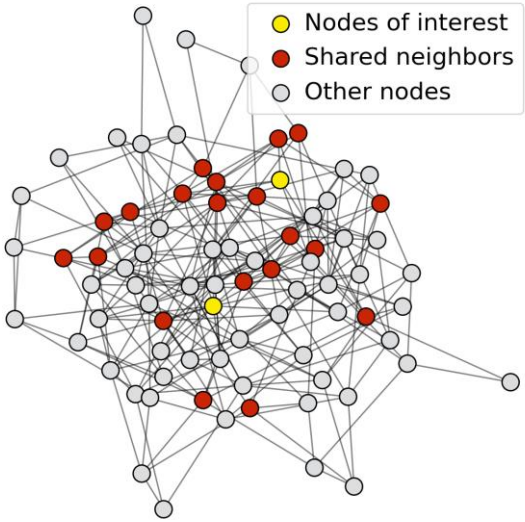
1-hop neighbor overlap

Only 1 node



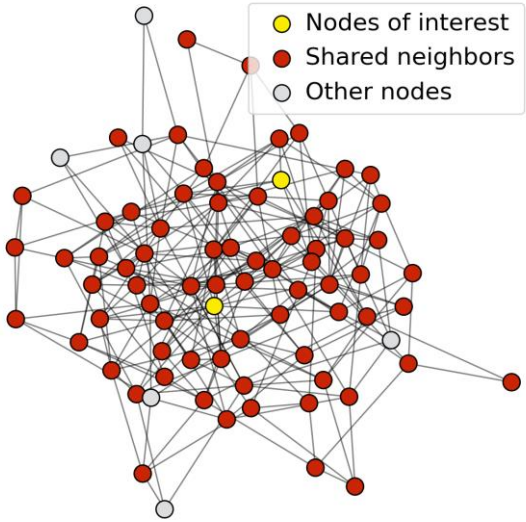
2-hop neighbor overlap

About 20 nodes



3-hop neighbor overlap

Almost all the nodes!

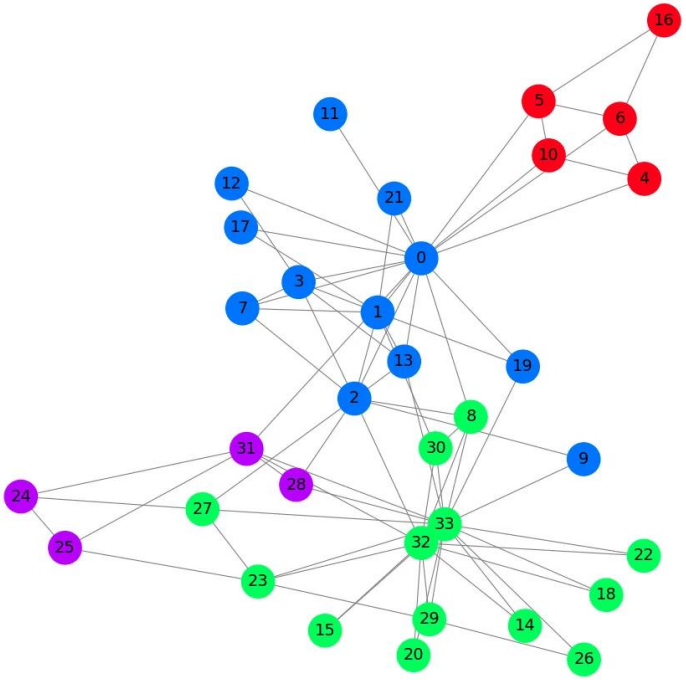


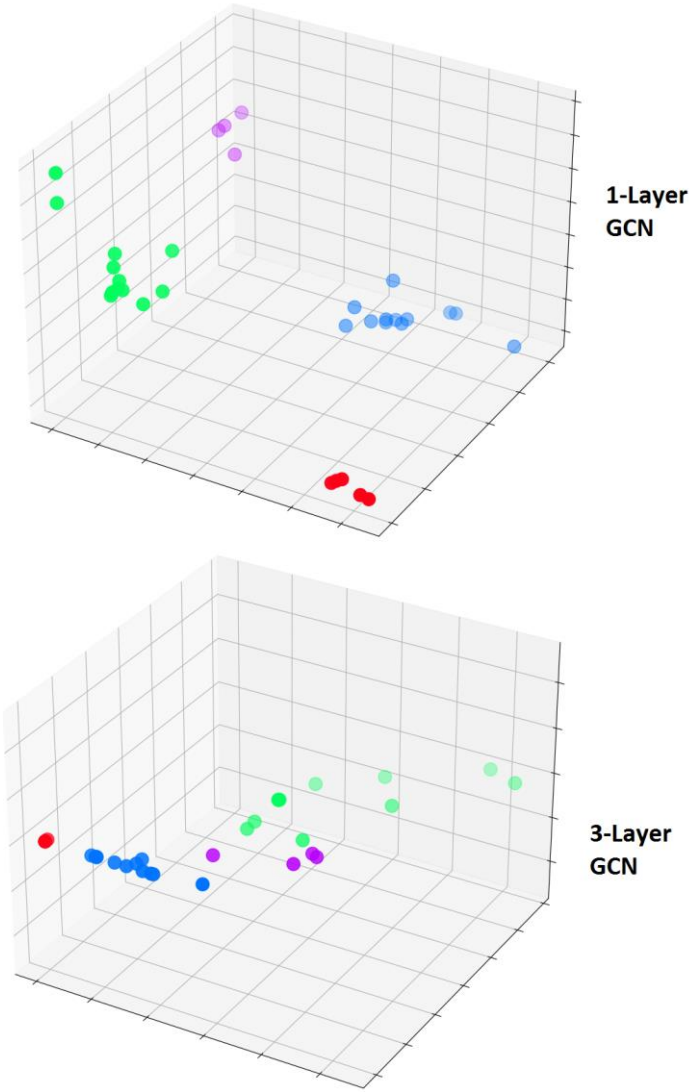
Receptive Field & Over-smoothing

- **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**
  - → **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?

Over-smoothing example





Design GNN Layer Connectivity

What do we learn from the over-smoothing problem?

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

Question: How to enhance the expressive power of a GNN, if the number of GNN layers is small?

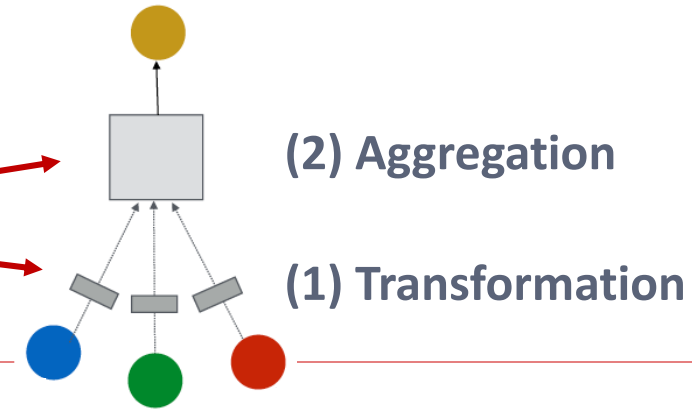
Expressive Power for Shallow GNNs

- **How to make a shallow GNN more expressive?**

Solution 1: Increase the expressive power **within each GNN layer**

- In our previous examples, each transformation or aggregation function only include one linear layer
- We can **make aggregation/transformation become a deep neural network**!

If needed, each box could include a 3-layer MLP

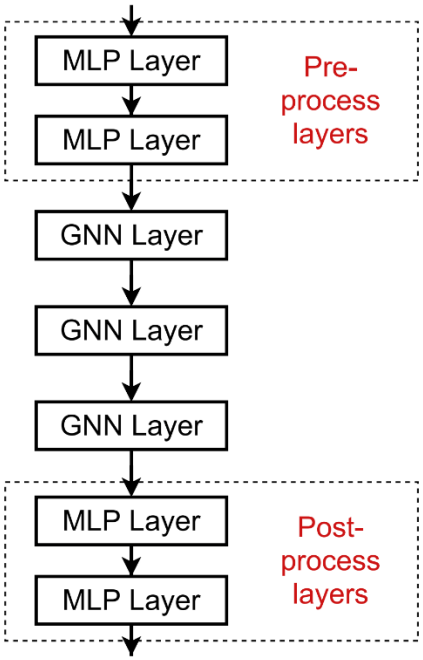


Expressive Power for Shallow GNNs

- **How to make a shallow GNN more expressive?**

Solution 2: Add layers that do not pass messages

- A GNN does not necessarily only contain GNN layers
  - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process** 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!

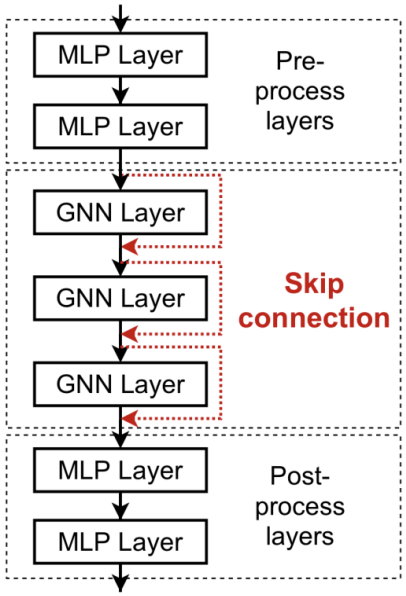
He et al. Deep Residual Learning for Image Recognition, CVPR 2015

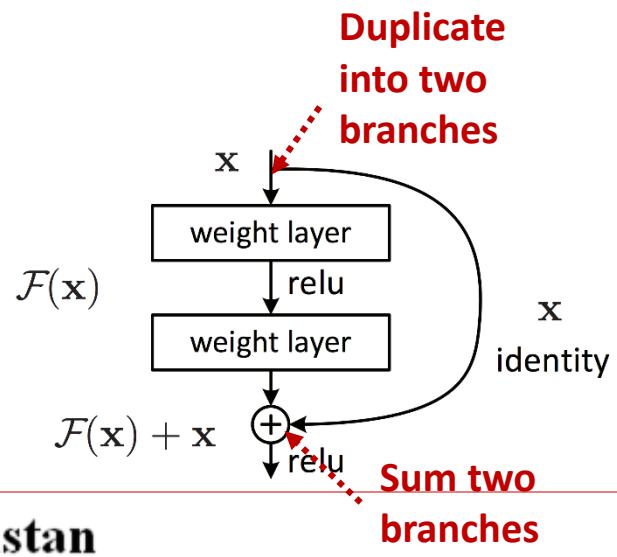
Design GNN Layer Connectivity

- **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**





Idea of skip connections:

Before adding shortcuts:

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

After adding shortcuts:

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

Idea of Skip Connections

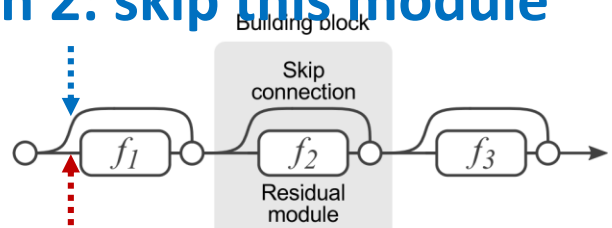
- **Why do skip connections work?**
  - **Intuition:** Skip connections create **a mixture of models**
  - N skip connections → 2^N possible paths
  - Each path could have up to N modules

- **We automatically get a mixture of shallow GNNs and deep GNNs**

All the possible paths:

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

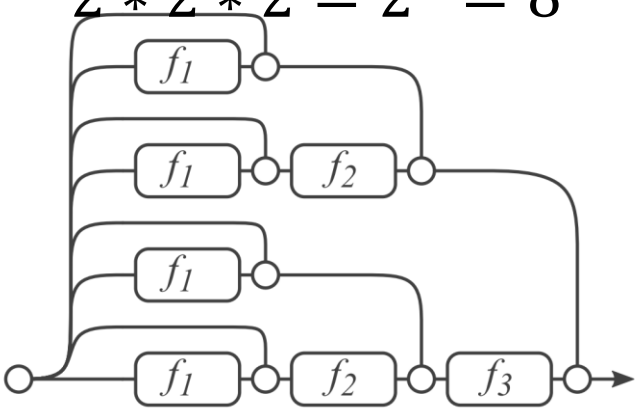
Path 2: skip this module



Path 1: include this module

(a) Conventional 3-block residual network

=



(b) Unraveled view of (a)

Example: GCN with Skip Connections

- A standard GCN layer

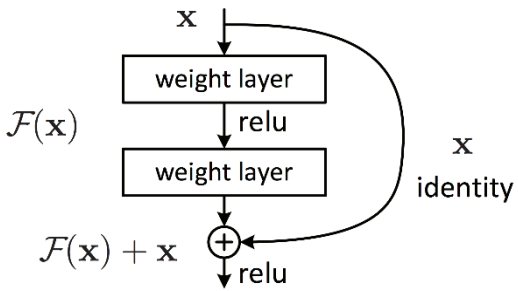
- $h_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathrm{W}^{(l)} \frac{h_u^{(l-1)}}{|N(v)|}\right)$

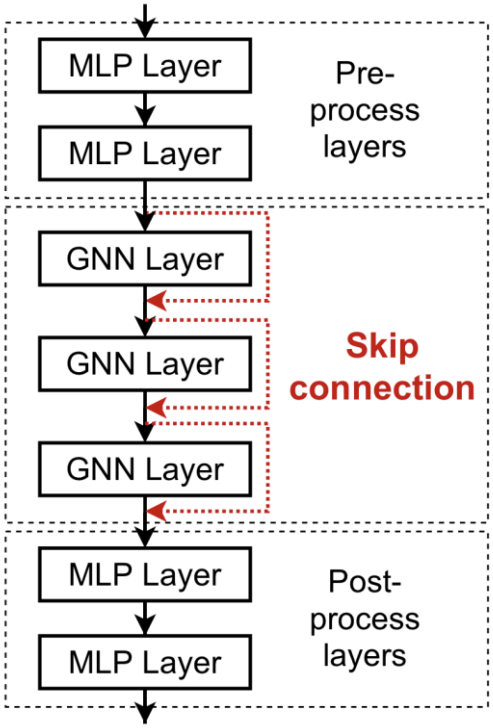
This is our $F(\mathrm{x})$

- A GCN layer with skip connection

- $h_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathrm{W}^{(l)} \frac{h_u^{(l-1)}}{|N(v)|} + h_v^{(l-1)}\right)$

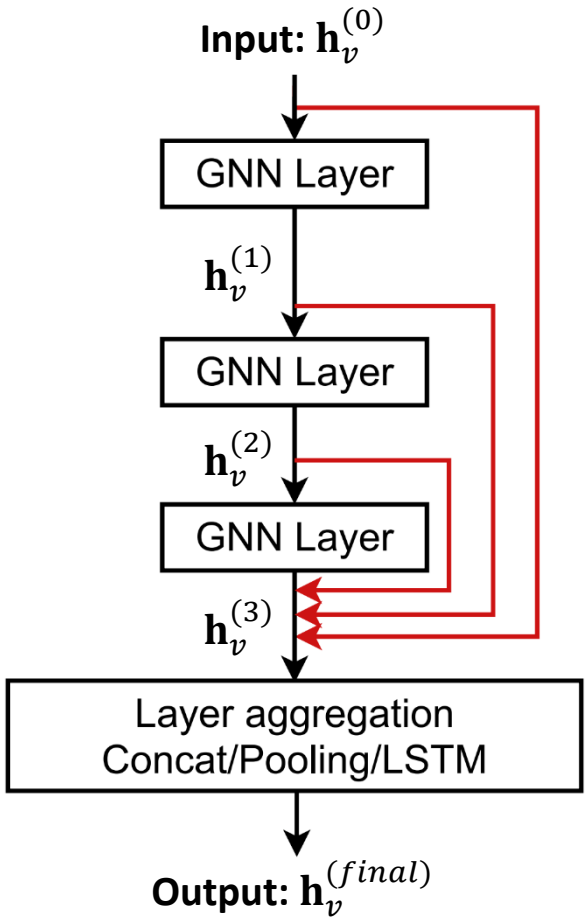
$F(\mathrm{x})$ + x





Other Options of Skip Connections

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



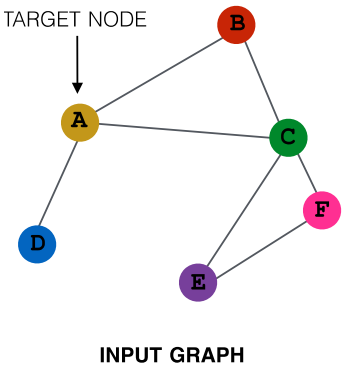
General Framework

5 main issues

- A single GNN layer: Aggregation (1) and Message (2)
- Layer connectivity: Stacking (3)
- Graph manipulations(augmentation) (4)
- Learning objectives/metrics (5)

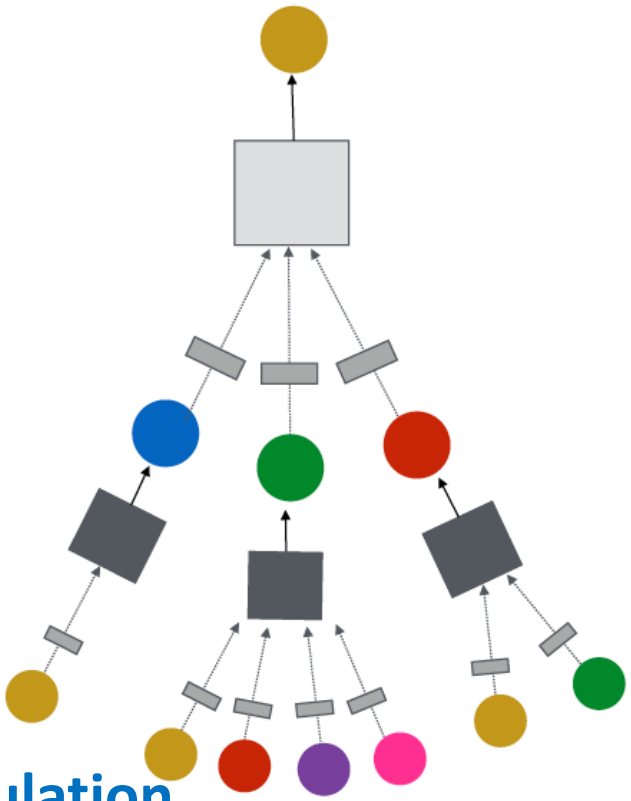
GRAPH MANIPULATIONS

General GNN Framework



Idea: Raw input graph ≠ computational graph

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



(4) Graph manipulation

Why Manipulate Graphs

Our assumption so far has been

- Raw input graph = computational graph

Reasons for breaking this assumption

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

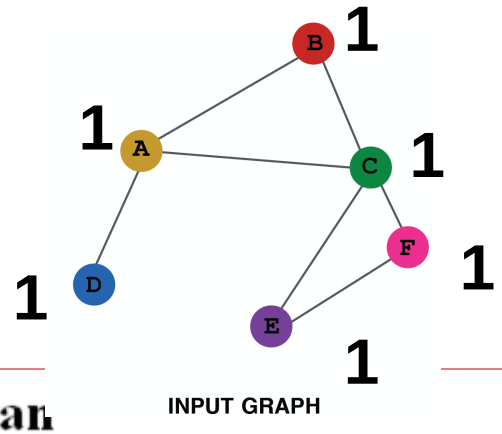
Feature Augmentation on Graphs

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:

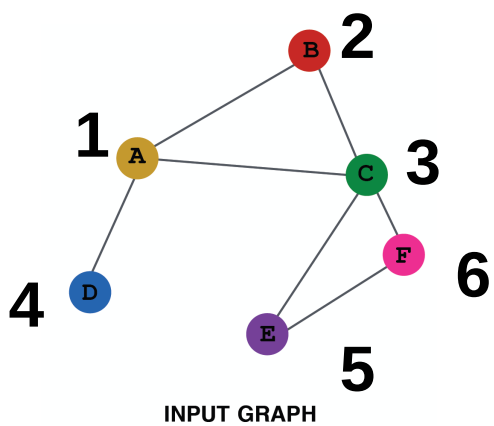
(a) Assign constant values to nodes



Feature Augmentation on Graphs

(b) Assign unique IDs to nodes

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



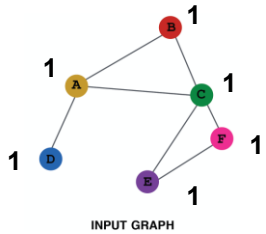
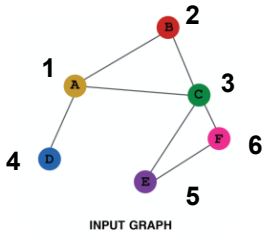
One-hot vector for node with ID=5

ID = 5

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

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)

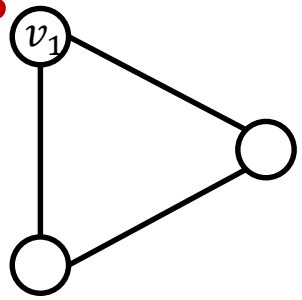
Feature Augmentation on Graphs

Why do we need feature augmentation?

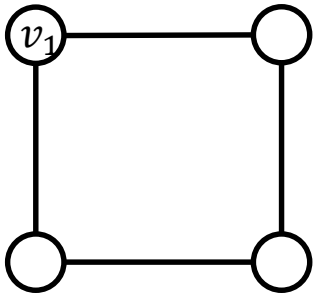
(2) Certain structures are hard to learn by GNN

- **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?

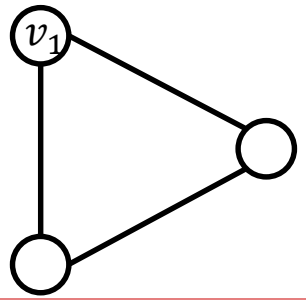
- (2) Certain structures are hard to learn by GNN
- **Solution:**
  - We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for v_1

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

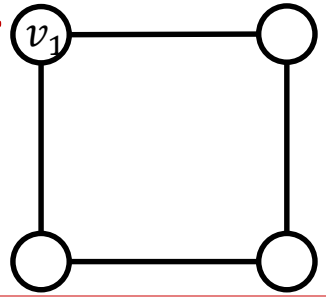
v_1 **resides in a cycle with length 3**



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

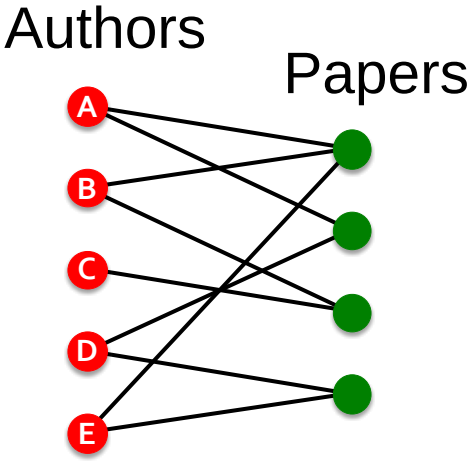
Add Virtual Nodes / Edges

- **(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)

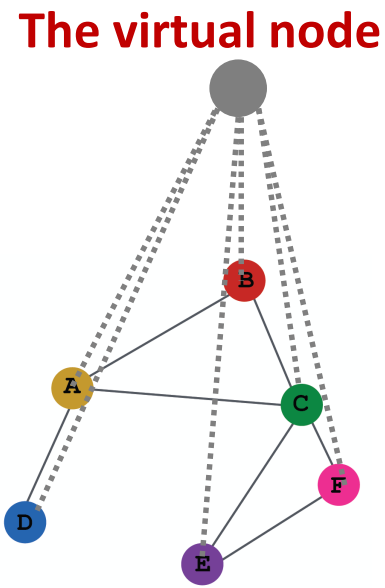
  2-hop virtual edges make an author-author collaboration graph



Add Virtual Nodes / Edges

(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**



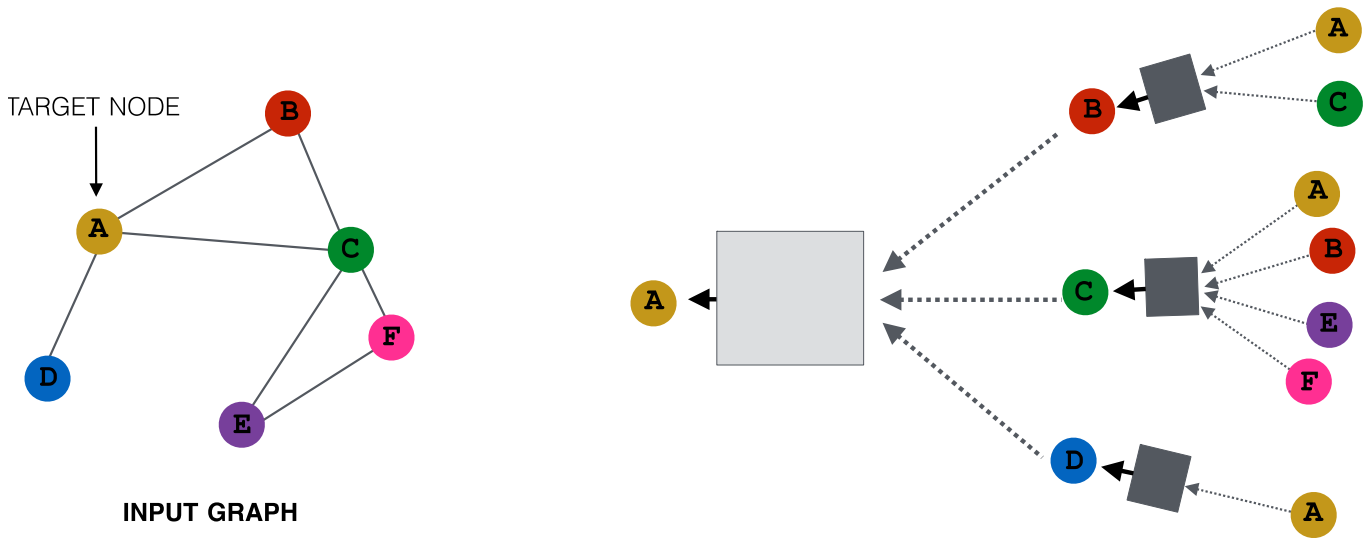
INPUT GRAPH

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**

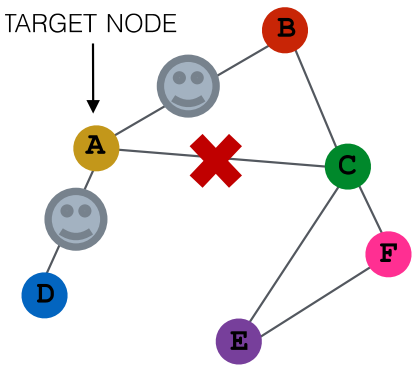


New idea: (Randomly) determine a node's neighborhood for message passing

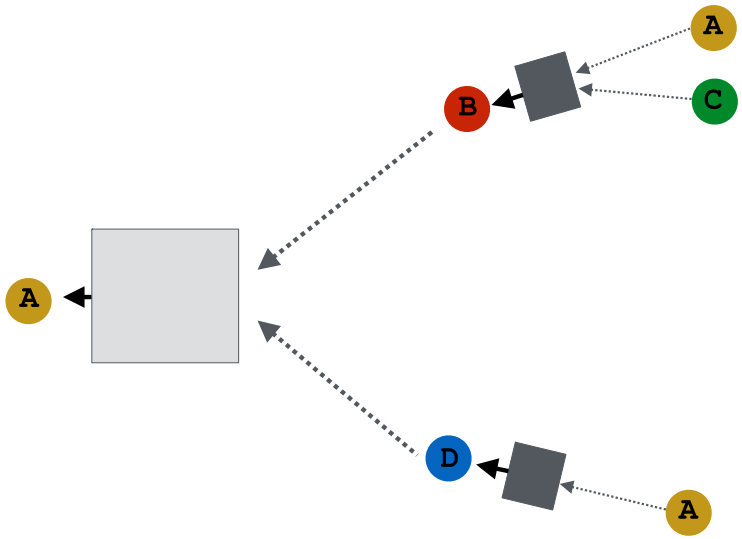
Neighborhood Sampling Example

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

- Only nodes B and D will pass message to A



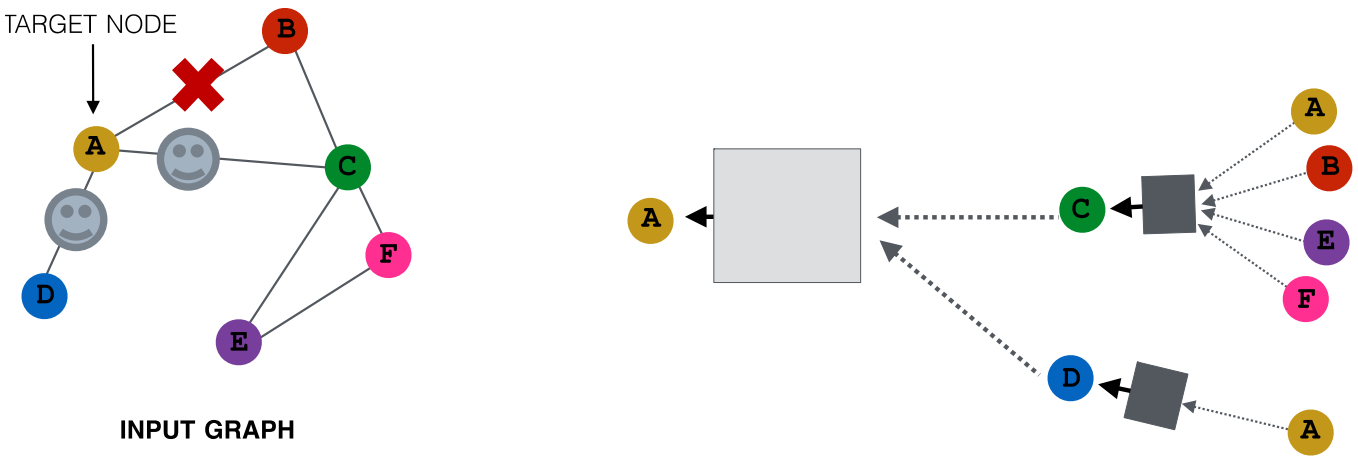
INPUT GRAPH



Neighborhood Sampling Example

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

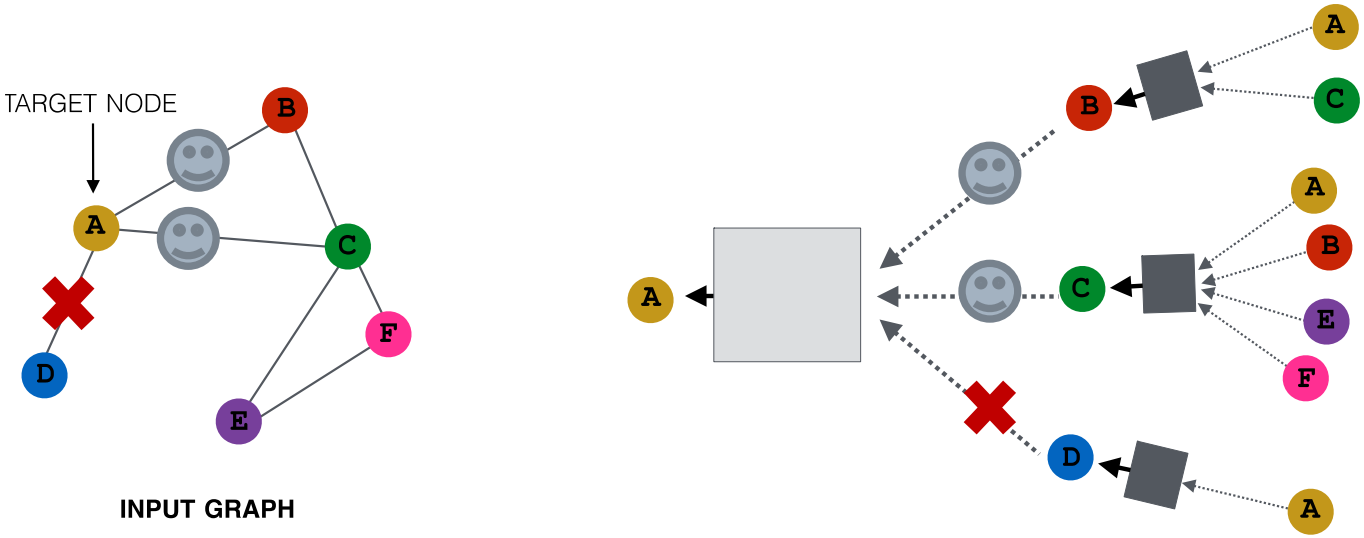
- Only nodes C and D will pass message to A



Neighborhood Sampling Example

In expectation, we can get embeddings similar to the case where all the neighbors are used

- **Benefits:** Greatly reduce computational cost
- And in practice it works great!



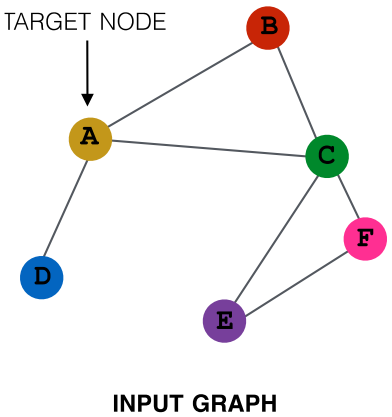
General Framework

5 main issues

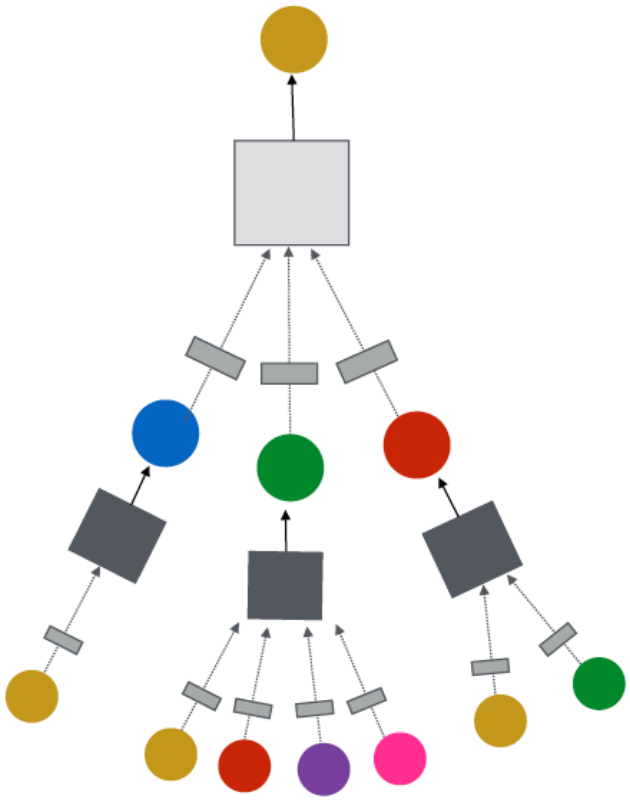
- A single GNN layer: Aggregation (1) and Message (2)
- Layer connectivity: Stacking (3)
- Graph manipulations(augmentation) (4)
- Learning objectives/metrics (5)

LEARNING WITH GNNS

A General GNN Framework



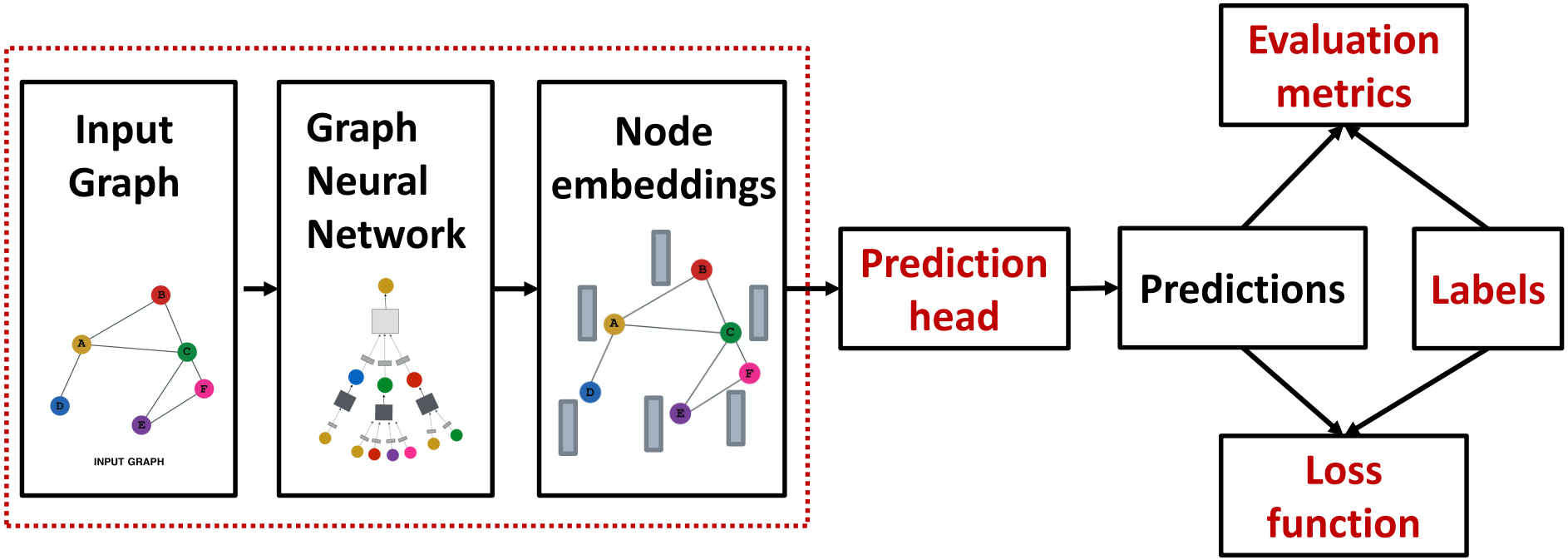
(5) Learning objective



How do we train a GNN?

GNN Training Pipeline

So far what we have covered

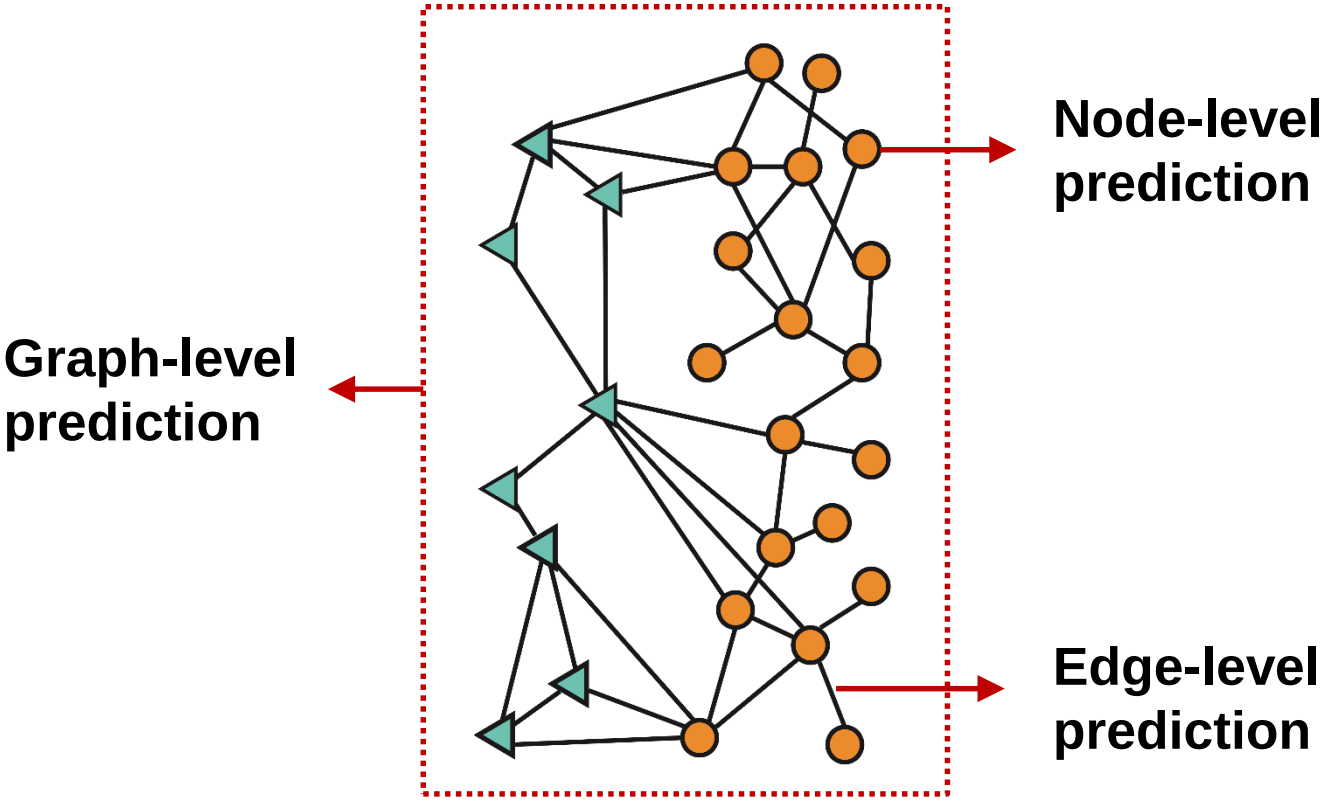


Output of a GNN: set of node embeddings

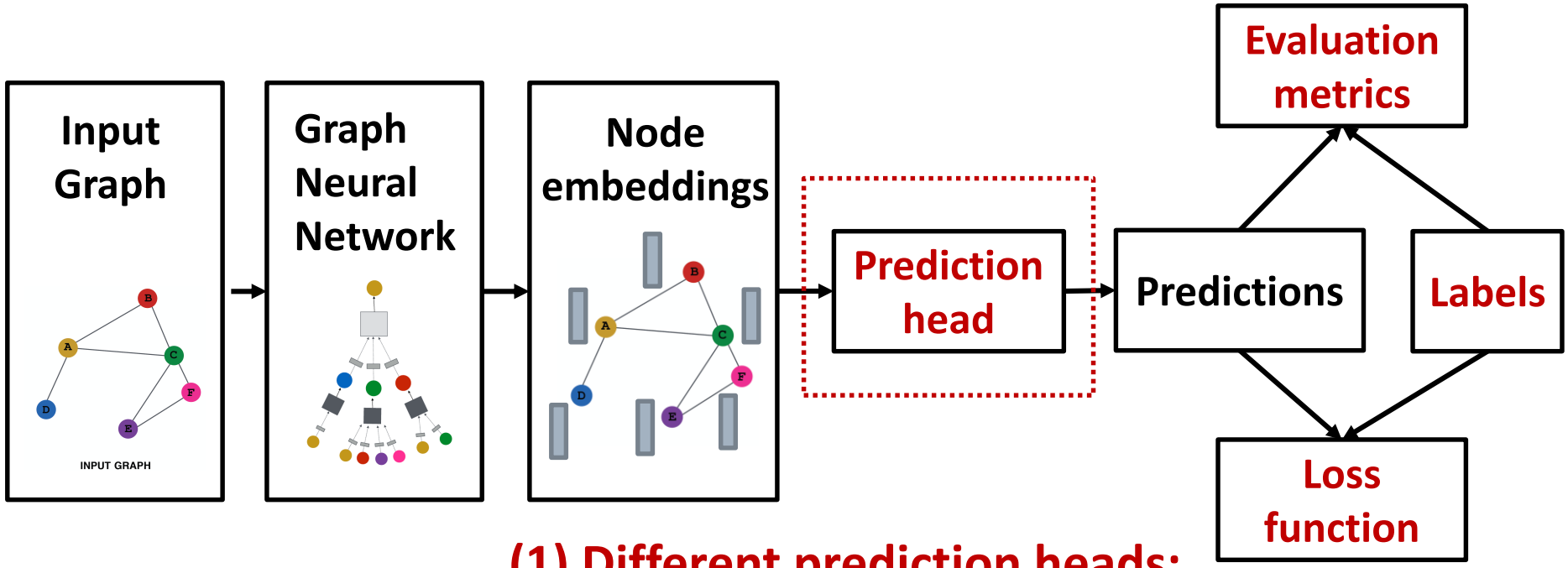
$$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$$

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

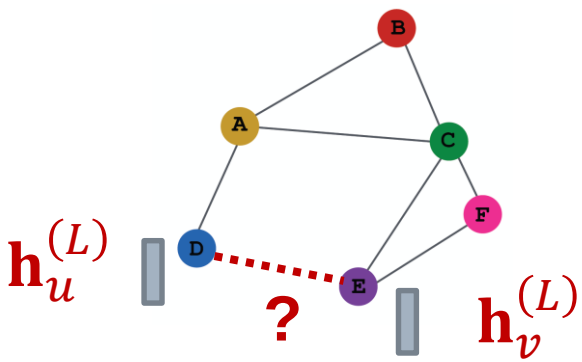
Output of the classifier

Prediction Heads: Edge-level

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

- Suppose we want to make k-way prediction

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



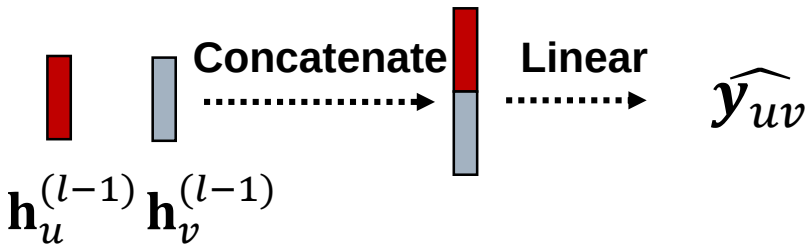
- What are the options for $\mathrm{Head}_{\mathrm{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$?

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



- $\hat{\boldsymbol{y}}_{\boldsymbol{uv}} = \mathrm{Linear}(\mathrm{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
- 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 $\mathrm{Head}_{\mathrm{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:

(2) Dot product

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

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

$$\ldots$$

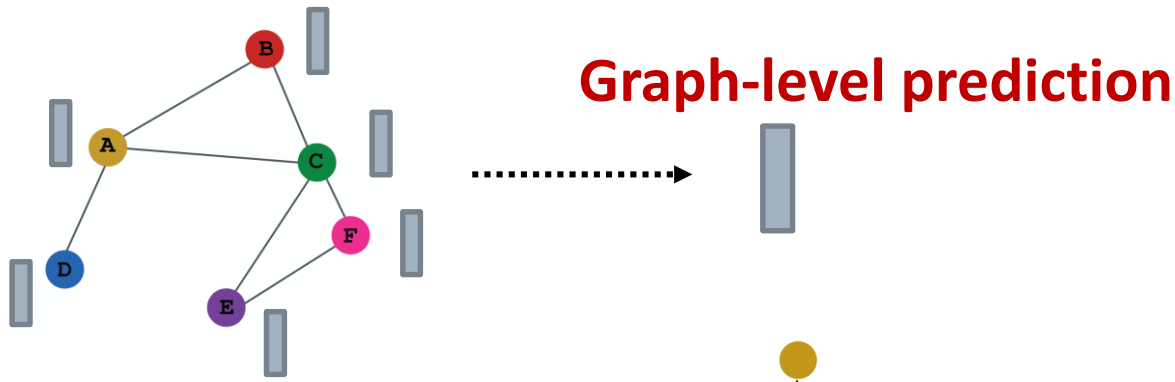
$$\hat{\boldsymbol{y}}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$

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

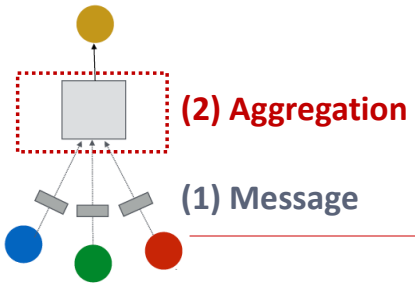
Prediction Heads: Graph-level

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

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



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



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**

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

- **(2) Global max pooling**

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

- **(3) Global sum pooling**

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

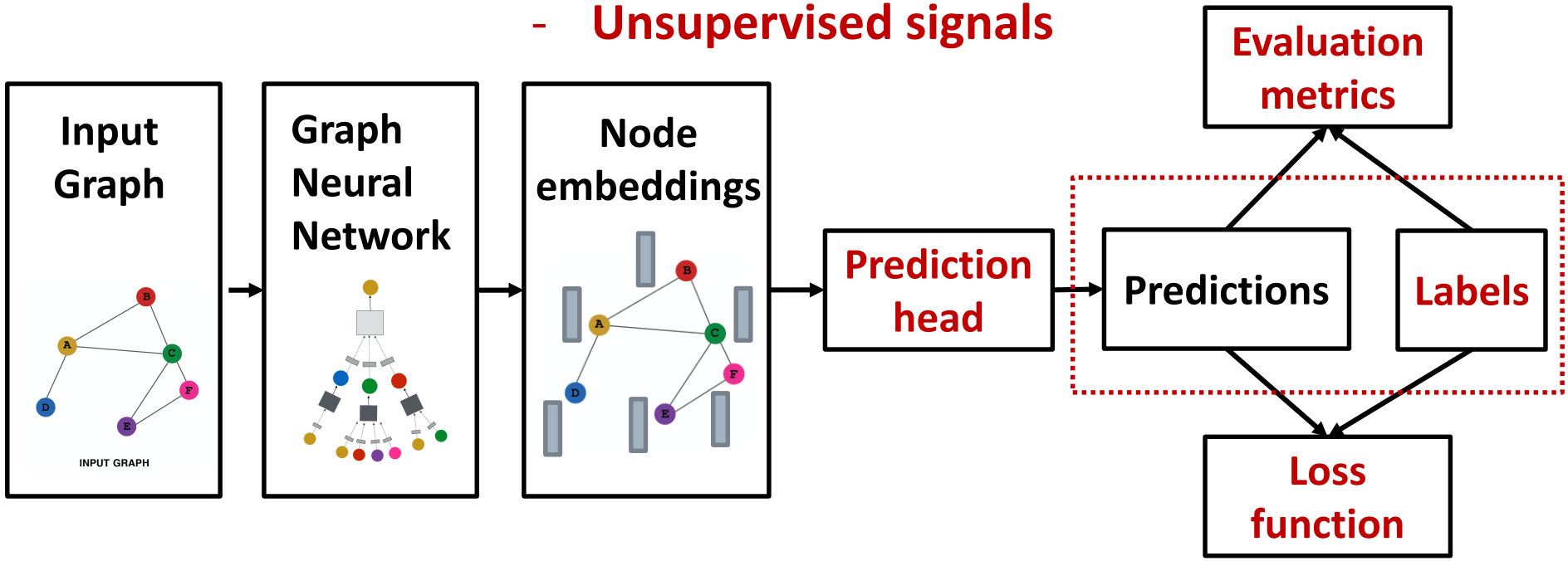
- These options work great for small graphs

For large graphs, hierarchical aggregation

GNN Training Pipeline (2)

(2) Where does ground-truth come from?

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



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

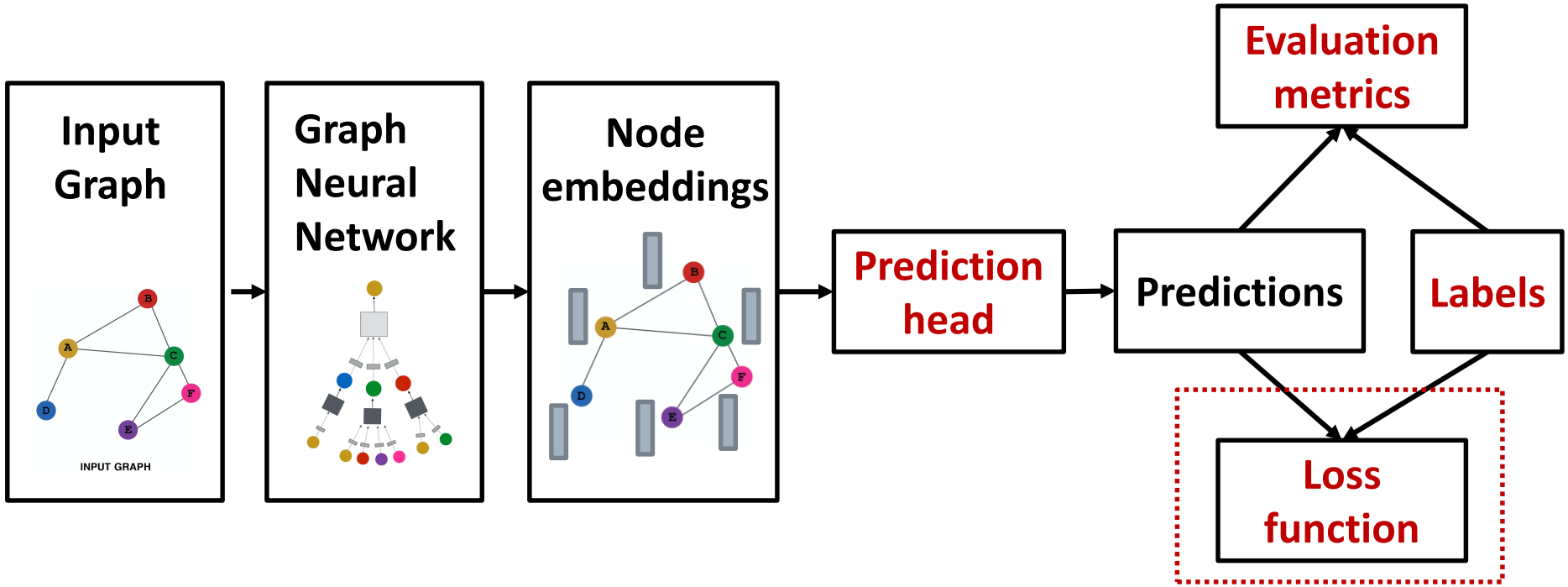
- **Supervised labels come from the specific use cases**. For example:
  - **Node labels y_v:** in a citation network, which subject area does a node belong to
  - **Edge labels y_{uv}:** in a transaction network, whether an edge is dishonest
  - **Graph labels y_G:** among molecular graphs**,** the 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 y_v.** Node statistics: such as clustering coefficient, PageRank, …
  - **Edge-level y_{uv}.** Link prediction: hide the edge between two nodes, predict if there should be a link
  - **Graph-level y_G.** Graph statistics: for example, predict if two graphs are isomorphic
  - **These tasks do not require any external labels!**

GNN Training Pipeline (3)



(3) How do we compute the final loss?

- **Classification loss**
- **Regression loss**

Settings for GNN Training

- **The setting:** We have N data points
  - Each data point can be a node/edge/graph
  - **Node-level**: prediction $\hat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
  - **Edge-level**: prediction $\hat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
  - **Graph-level**: prediction $\hat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$
  - We will use prediction $\hat{\boldsymbol{y}}^{(i)}$, label $\boldsymbol{y}^{(i)}$ to refer **predictions at all levels**

Classification or Regression

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

Classification Loss

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

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

i-th data point

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

j-th class

where: **Label** **Prediction**

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

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

Regression Loss

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

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

***i*-th data point**

***j*-th target**

where:

E.g.

1.4	2.3	1.0	0.5	0.6

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

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

E.g.

0.9	2.8	2.0	0.3	0.8

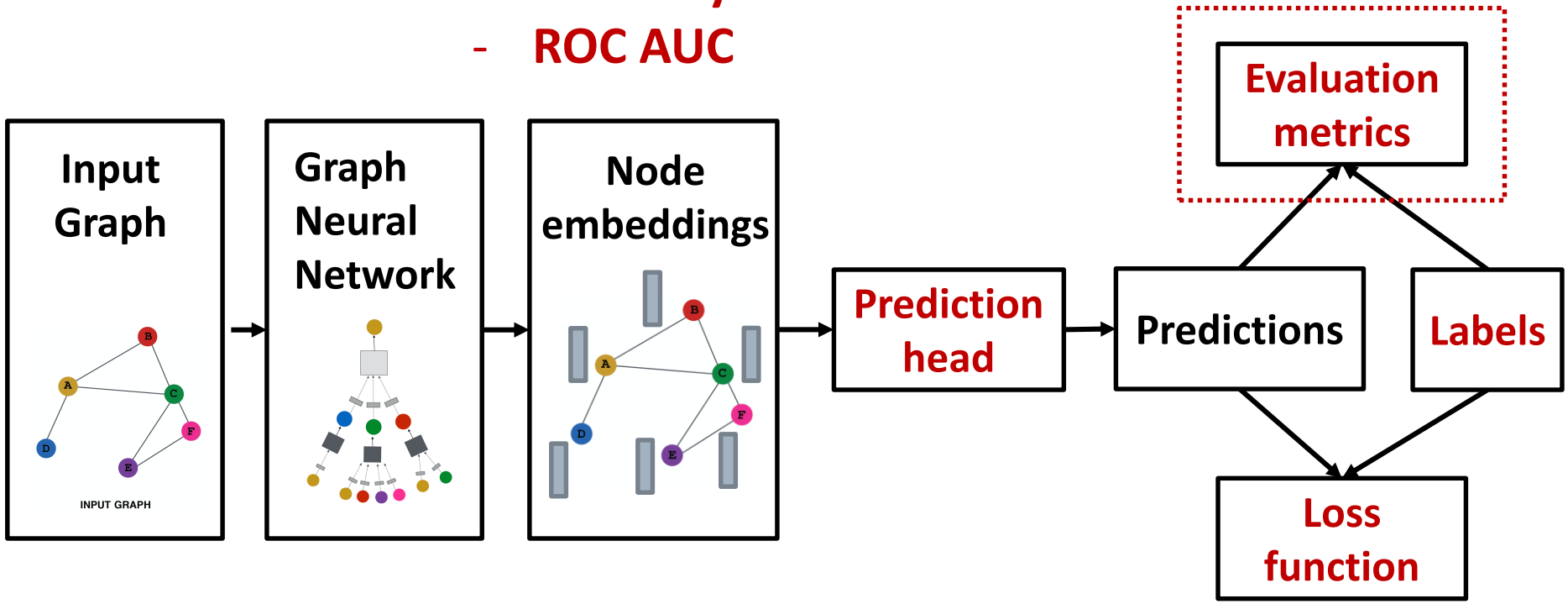
- Total loss over all N training examples

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

GNN Training Pipeline (4)

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

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



Evaluation Metrics: Regression

- **We use standard evaluation metrics for GNN**
  - In practice we will use sklearn for implementation
  - Suppose we make predictions for N data points
- **Evaluate regression tasks on graphs:**
  - **Root mean square error (RMSE)**

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

  - **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**

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

- **(2) Binary classification**
  - Metrics sensitive to classification threshold
    - **Accuracy**
    - **Precision / Recall**
    - If the range of prediction is $[0,1]$, we will use 0.5 as threshold
  - Metric Agnostic to classification threshold
    - **ROC AUC**

Metrics for Binary Classification

- **Accuracy:** $\frac{TP+TN}{TP+TN+FP+FN} = \frac{TP+TN}{|Dataset|}$

- **Precision (P):** $\frac{TP}{TP+FP}$

- **Recall (R):** $\frac{TP}{TP+FN}$

- **F1-Score:** $\frac{2P*R}{P+R}$

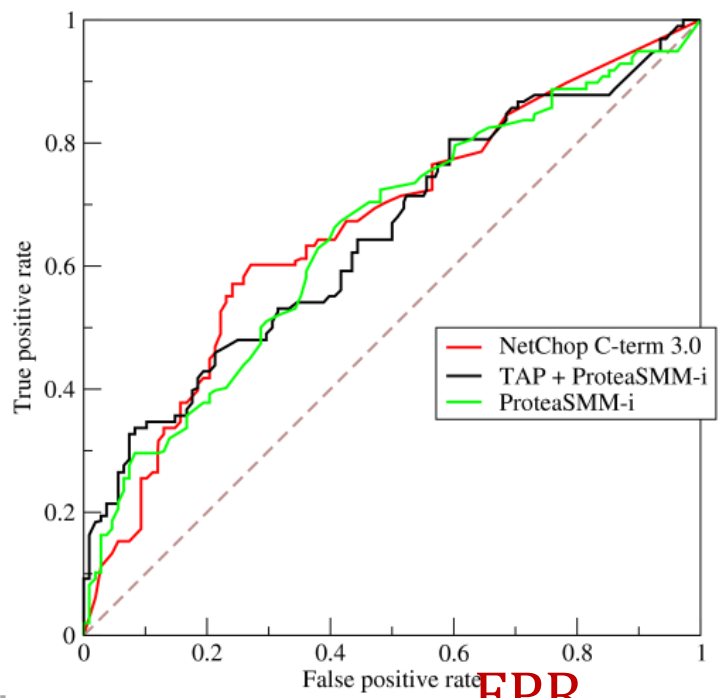
Confusion matrix

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

(4) Evaluation Metrics

- **ROC Curve:** Captures the tradeoff in TPR and FPR **as the classification threshold is varied for a binary classifier.**

TPR



FPR

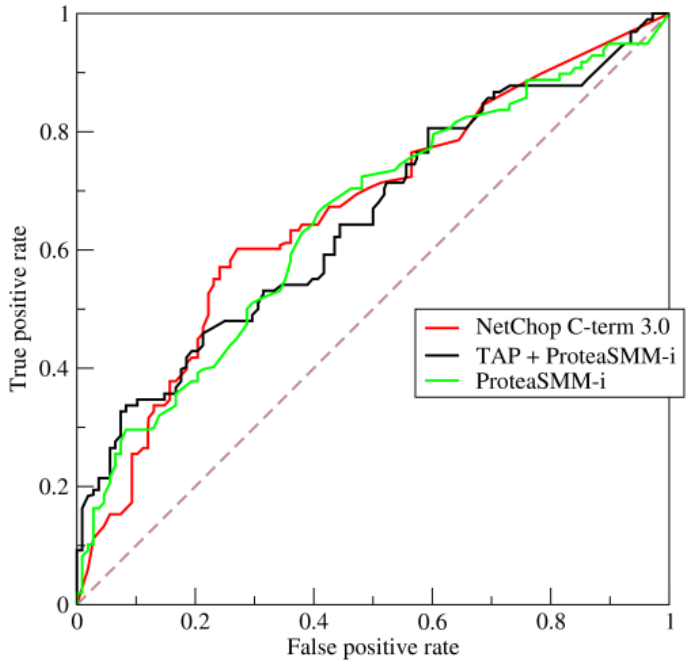
$$\text{TPR} = \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

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

Note: the dashed line represents performance of a random classifier

Image Credit: Wikipedia

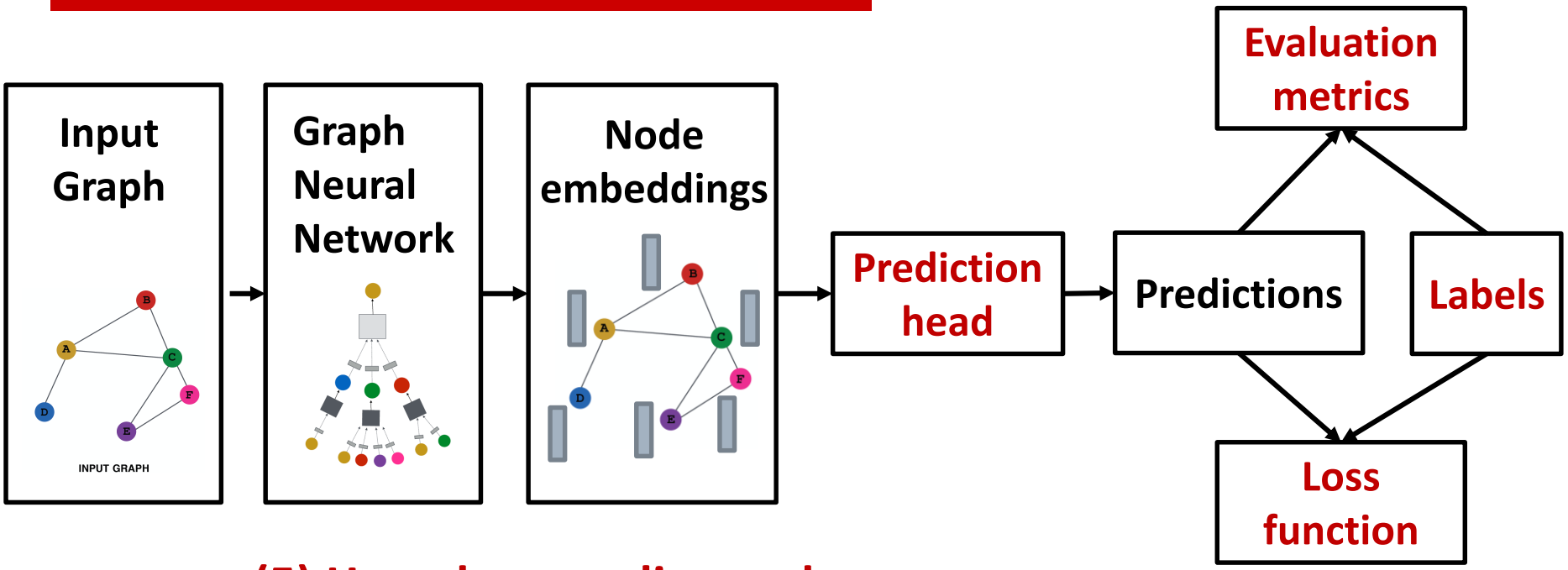
(4) Evaluation Metrics



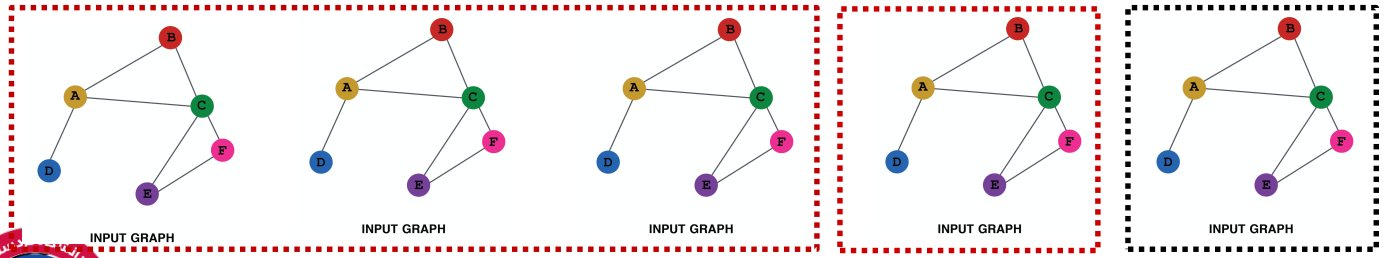
Content Credit: **Wikipedia**

- **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)



(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**

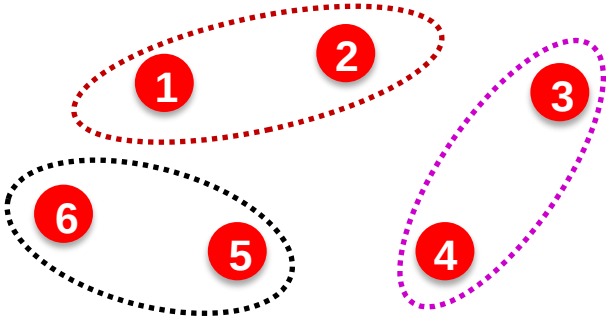
Why Splitting Graphs is Special

- **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



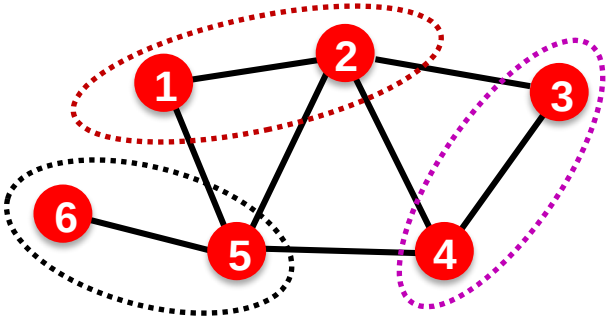
Why Splitting Graphs is Special

- **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

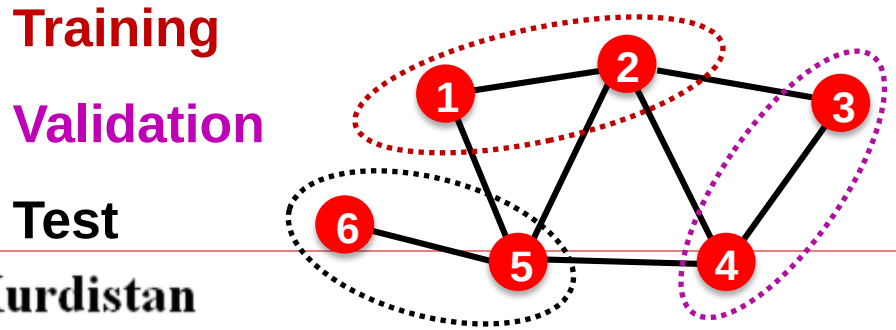


What are our options?

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).

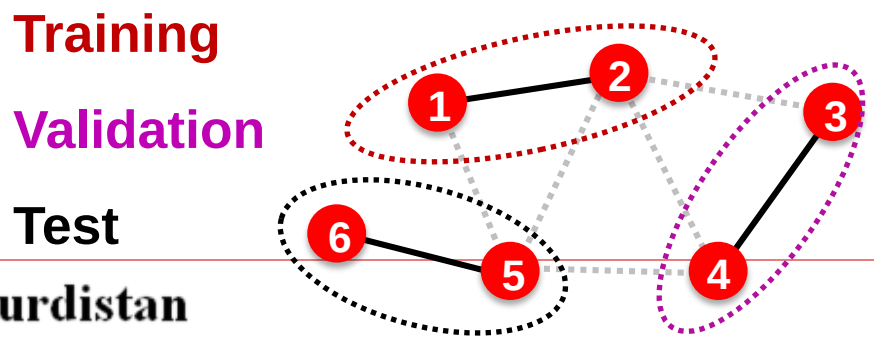
- **We will only split the (node) labels**
  - **At training time,** we compute embeddings **using the entire graph**, and train **using node 1&2's labels**
  - **At validation time,** we compute embeddings **using the entire graph**, and **evaluate on node 3&4's labels**



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**

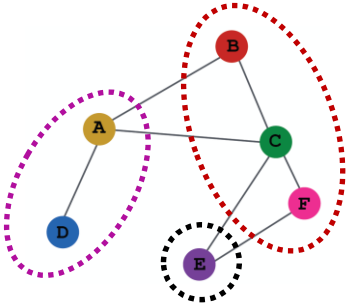


Transductive/Inductive Settings

- **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

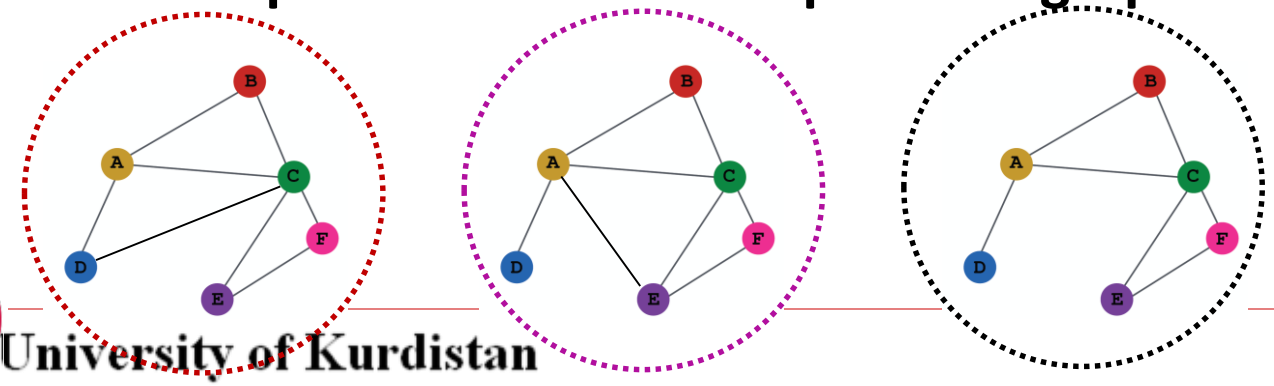


Training

Validation

Test

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



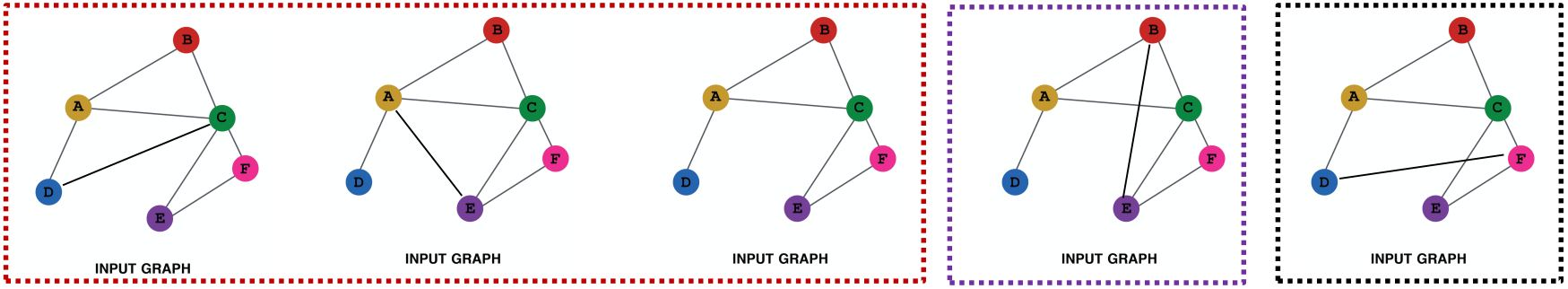
Training

Validation

Test

Example: Graph Classification

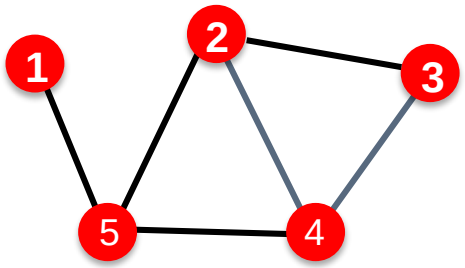
- Only the **inductive setting** is well defined for **graph classification**
  - Because **we have to test on unseen graphs**
  - Suppose we have a dataset of 5 graphs. Each split will contain independent graph(s).



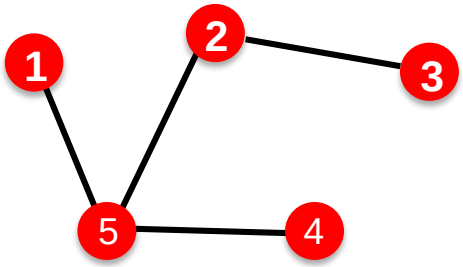
Training **Validation** **Test**

Example: Link Prediction

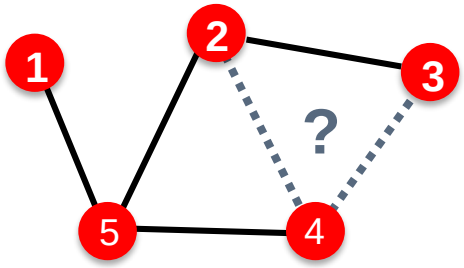
- **Goal of link prediction: predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised/self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and the **let the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



Original graph

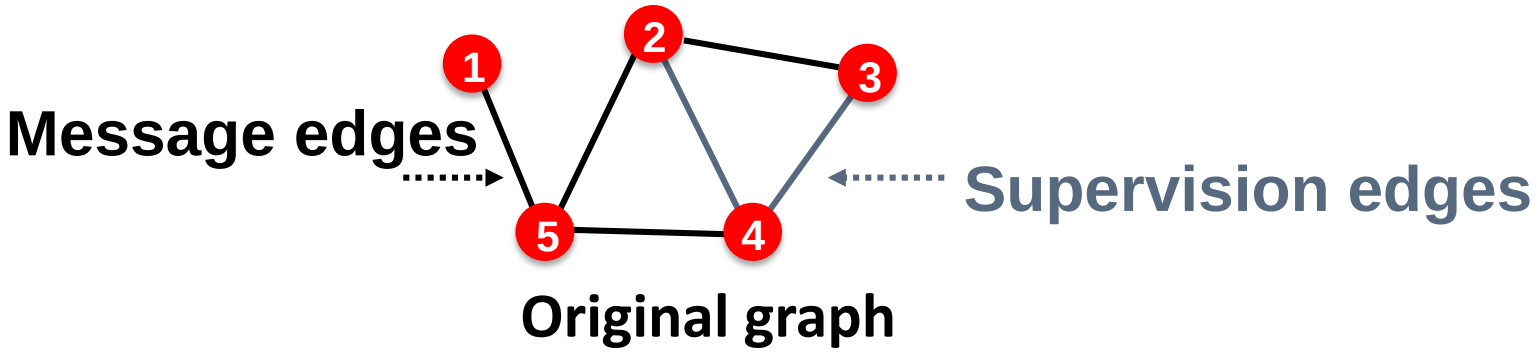


Input graph to GNN



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

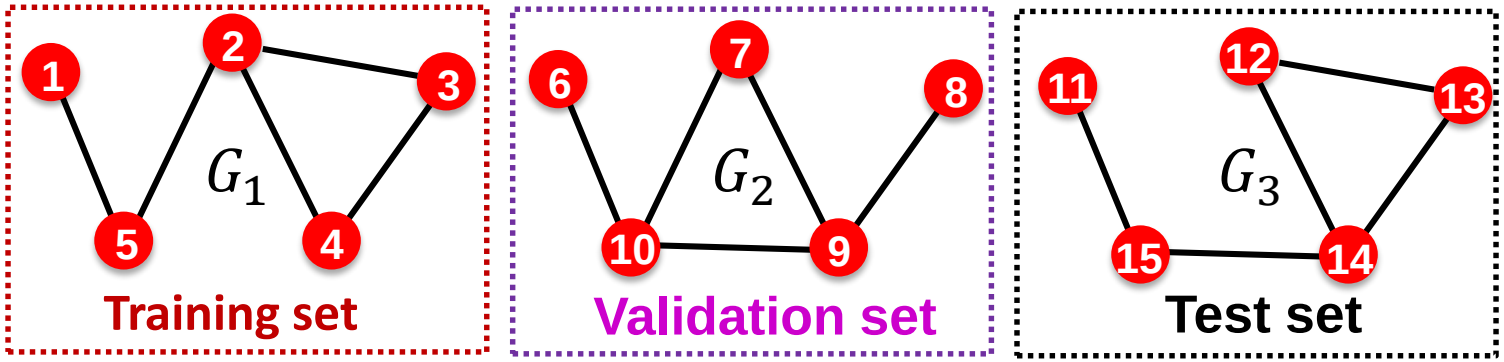
- **Message edges: Used for GNN message passing**
- **Supervision edges: Use for computing objectives**

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**

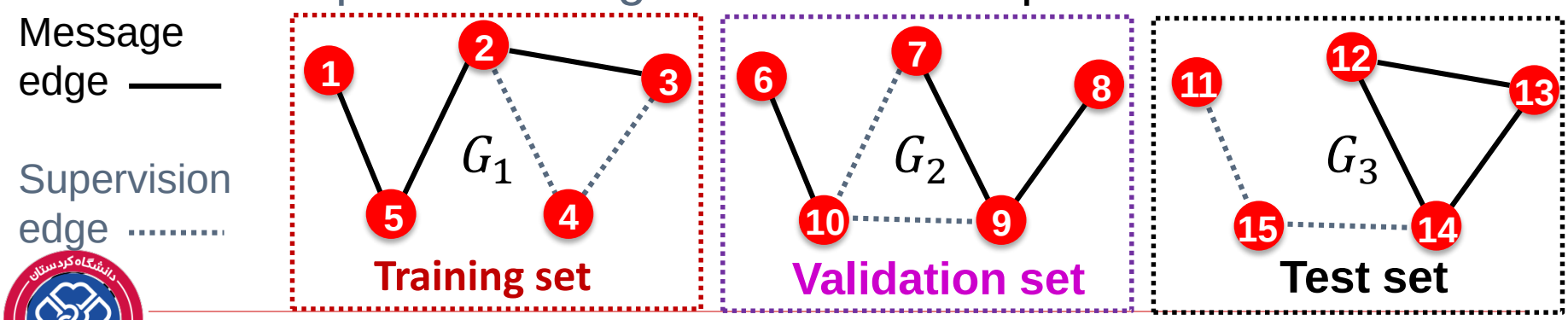


Setting up Link Prediction

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

Option 1: Inductive link prediction split

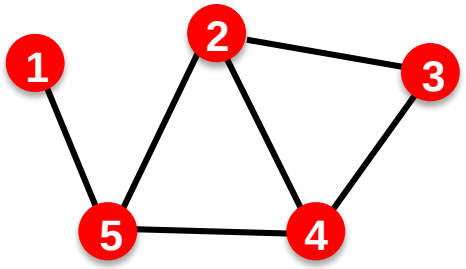
- Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph
- In train or val or test set, each graph will have 2 types of edges: message edges + supervision edges
  - Supervision edges are not the input to GNN



Setting up Link Prediction

Option 2: Transductive link prediction split:

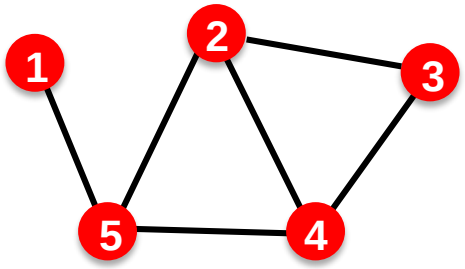
- This is the <u>default</u> setting when people talk about link prediction
- Suppose we have a dataset of 1 graph



Setting up Link Prediction

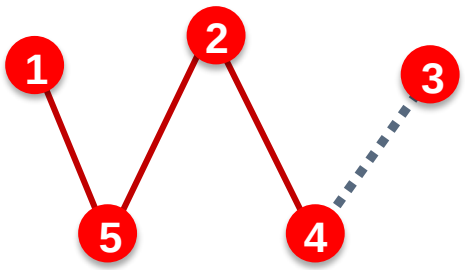
Option 2: Transductive link prediction split:

- 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

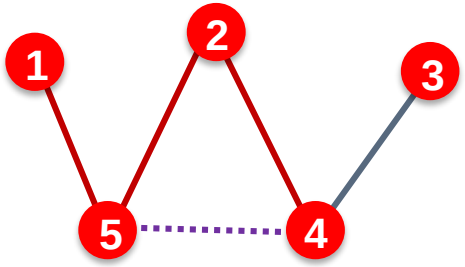


Setting up Link Prediction

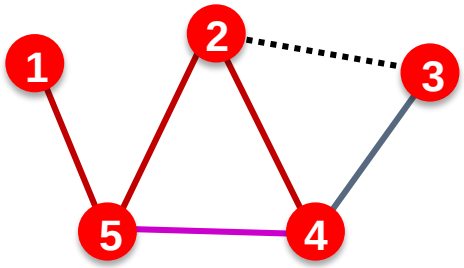
Option 2: Transductive link prediction split:



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



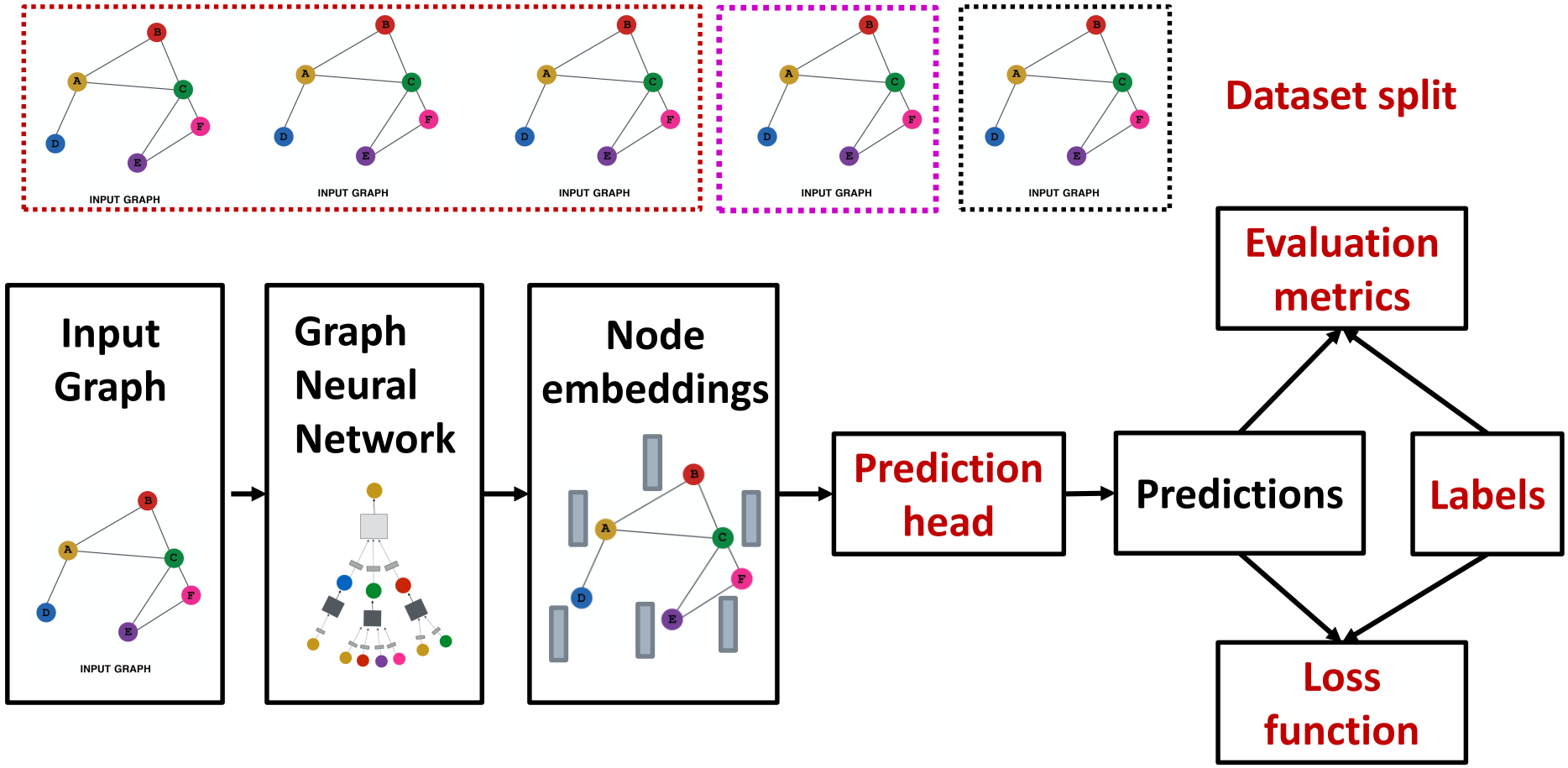
(2) At validation time:
Use training message edges & training supervision edges to predict validation edges



(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


Questions